An Engineer's Guide to MATLAB®

with Applications from Mechanical, Aerospace, Electrical, Civil and Biological Systems Engineering

THIRD EDITION

Edward B. Magrab | Shapour Azarm | Balakumar Balachandran
James H. Duncan | Keith E. Herold | Gregory C. Walsh

An Engineer's Guide to MATLAB®

This page intentionally left blank

An Engineer's Guide to MATLAB®

With Applications from Mechanical, Aerospace, Electrical, Civil, and Biological Systems Engineering

Third Edition

Edward B. Magrab
Department of Mechanical Engineering, University of Maryland, College Park, MD

Shapour Azarm
Department of Mechanical Engineering, University of Maryland, College Park, MD

Balakumar Balachandran
Department of Mechanical Engineering, University of Maryland, College Park, MD

James H. Duncan
Department of Mechanical Engineering, University of Maryland, College Park, MD

Keith E. Herold
Fischell Department of Bioengineering, University of Maryland, College Park, MD

Gregory C. Walsh
Leica Geosystems, Inc., San Ramon, CA

Prentice Hall
Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

VP/Editorial Director, Engineering/Computer Science: *Marcia J. Horton*
Assistant/Supervisor: *Dolores Mars*
Senior Editor: *Tacy Quinn*
Associate Editor: *Dee Bernhard*
Director of Marketing: *Margaret Waples*
Senior Marketing Manager: *Tim Galligan*
Marketing Assistant: *Mack Patterson*
Senior Managing Editor: *Scott Disanno*
Project Manager: *Greg Dulles*
Senior Operations Supervisor: *Alan Fischer*
Production Manager: *Wanda Rockwell*
Creative Director: *Jayne Conte*
Cover Designer: *Bruce Kenselaar*
Cover Art: *Getty Images, Inc.*
Media Editor: *Daniel Sandin*
Composition: *Integra*
Printer/Binder: *Courier Companies, Inc.*

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on appropriate pages within text.

Library of Congress Cataloging in Publication Data

Prentice Hall
is an imprint of

PEARSON

www.pearsonhighered.com

ISBN 13: 978-0-13-199110-1
ISBN 10: 0-13-199110-8

For June Coleman Magrab

This page intentionally left blank

Contents

This page intentionally left blank

List of Examples

Chapter 1

Chapter 2

Chapter 4

Chapter 5

Chapter 6

Chapter 7

Chapter 8

Chapter 9

Chapter 10

Chapter 11

Chapter 12

Chapter 13

Chapter 14

Preface to Third Edition

In going from the previous edition to this third edition, we have made many significant changes. A new chapter, "Biological Systems: Transport of Heat, Mass, and Electric Charge," has been added. To make room for this new material, Chapter 8, "Machine Design," of the previous edition has been removed. In Chapter 1, "Introduction," more details on the setup of user preferences and the use of the MATLAB editor are provided, and the number of exercises has been significantly increased. Also, the Symbolic toolbox has been moved to this chapter. In Chapter 5, "Function Creation and Selected MATLAB Functions," the section dealing with the differential equation solvers now includes the delay differential equations solver (`dde23`) and the one-dimensional parabolic–elliptic partial differential equations solver (`pdepe`). In addition, the range of examples for the ordinary differential equations solver `bvp4c` has been expanded to better illustrate its wide applicability. Chapter 6, "2D Graphics," contains twice the number of special-purpose graph functions, more material on the enhancement of graphs, and several new examples replacing those used in the second edition. Chapter 9, "Vibrations," has been extensively revised and expanded to include a wider range of applications. Chapter 13 "Optimization," has also been expanded to demonstrate the use of the new Genetic Algorithm and Direct Search toolbox.

Overall, the book has been "refreshed" to reflect the authors' collective experiences with MATLAB, to introduce the new enhancements that are available in the MATLAB editor, and to include some of the new functions that have been introduced since the last edition. Overall, the examples, exercises, and MATLAB functions presented in the book have been increased by more than 25%. The book now contains 190 numbered examples, almost 300 exercises, and more than 375 MATLAB functions that are illustrated. The programs in this edition have been run on Version 2009a.

NEW TO THE EDITION

- Text was revised and tested throughout for the latest version of the software: release 2009a
- A new chapter has been added: *Biological Systems: Transport of Heat, Mass, and Electric Charge*
- 25% increase in number of examples, exercises, and Matlab functions
- Range of applications increased to include biology and electrical engineering
- Chapter 5 *Function Creation and Selected Matlab Functions* now includes the delay differential equations solver (dde23) and the one-dimensional parabolic-elliptic partial differential equations solver (pdepe).

- Expanded coverage in Chapter 9 *Vibrations* gives a wider range of applications.
- Chapter 13 *Optimization* has been expanded to demonstrate the use of the new Genetic Algorithm and Direct Search toolbox.

We have also created additional resources for the instructor and for the user. In addition to a solution manual that is available to instructors, we also provide a set of PowerPoint slides covering the material presented in Chapters 1–7. For the user of the book, we have created M files of all the numbered examples in each chapter. These ancillary materials can be accessed from the publisher's Web site.

E. B. MAGRAB
S. AZARM
B. BALACHANDRAN
J. H. DUNCAN
K. E. HEROLD
G. C. WALSH
College Park, MD

1

Introduction

Edward B. Magrab

The characteristics of the MATLAB environment and MATLAB's basic syntax are introduced.

1.1 INTRODUCTION

MATLAB, which derives its name from *Mat*rix *Lab*oratory, is a computing language devoted to processing data in the form of arrays of numbers. MATLAB integrates computation and visualization into a flexible computer environment, and provides a diverse family of built-in functions that can be used in a straightforward manner to obtain numerical solutions to a wide range of engineering problems.

1.1.1 Organization of the Book and Its Goals

The primary goal of this book is to enable the reader to generate readable, compact, and verifiably correct MATLAB programs that obtain numerical solutions to a wide range of physical and empirical models and display the results with fully annotated graphics.

The book can be used in several ways:

- To learn MATLAB
- As a companion to engineering texts
- As a reference for obtaining numerical solutions to a wide range of engineering problems
- As a source of applications of a wide variety of MATLAB solution techniques

The level of the book assumes that one has some fluency in calculus, linear algebra, and engineering mathematics, can employ the engineering approach to problem solving, and has some experience in using mathematical models to predict the response of elements, devices, and systems. These qualities play an important role in creating programs that function correctly.

The book has two interrelated parts. The first part consists of Chapters 1–7, which introduces the fundamentals of MATLAB syntax and commands and structured programming techniques. The second part, consisting of Chapters 8–14, makes extensive use of the first seven chapters to obtain numerical solutions to engineering problems for a wide range of topics. In several of these topical areas, MATLAB toolboxes are used extensively to minimize programming complexity so that one can obtain numerical solutions to engineering problems of varying degrees of difficulty. In particular, we illustrate the use of the Controls toolbox in Chapters 9 and 10, Simulink in Chapter 10, the Optimization toolbox in Chapter 13, the Statistics toolbox in Chapter 8, and the Symbolic toolbox in Chapters 1–5 and 9.

1.1.2 Some Suggestions on How to Use MATLAB

Listed below are some suggestions on how to use the MATLAB environment to effectively create MATLAB programs.

- *Write scripts and functions in a text editor and save them as M-files*. This will save time, save the code, and greatly facilitate the debugging process, especially if the MATLAB Editor is used.
- *Use the Help files extensively*. This will minimize errors caused by incorrect syntax and by incorrect or inappropriate application of a MATLAB function.
- *Attempt to minimize the number of expressions comprising a program*. This usually leads to a trade-off between readability and compactness, but it can

encourage the search for MATLAB functions and procedures that can perform some of the programming steps faster and more directly.

- *When practical, use graphical output as a program is being developed*. This usually shortens the code development process by identifying potential coding errors and can facilitate the understanding of the physical process being modeled or analyzed.
- *Most importantly, verify by* independent *means that the output from the program is correct*.

1.1.3 Book Notation Conventions

In order to facilitate the recognition of the significance of variable names and the origin of numerical values; that is, whether they are input values or output results, the following font conventions are employed.

Variable/Function Name	Font	Example
User-created variable	Times Roman	ExitPressure, a2, sig
MATLAB function	Courier	`cosh`(x), `pi`
MATLAB reserved word	Courier	`for, switch, while`
User-created function	Times Roman Bold	**BeamRoots**(a, x, k)

Numerical Value	Font	Example
Provided in program	Times Roman	5.672
Output to command window or to a graphical display	Helvetica	5.672

1.2 THE MATLAB ENVIRONMENT

1.2.1 Introduction

When the MATLAB program is launched, four windows appear as shown in Figure 1.1. The upper right-hand window is the *Workspace* window, which displays a list of the variables that the user has currently defined and their properties. The center window is the MATLAB *Command* window. The lower right-hand window is the *Command History* window, which displays all entries made in the command window during each session. A session is the interval between the start of MATLAB and its termination. The time and date appear before each list in this window to indicate when these entries began being recorded. It is a convenient way to review past sessions and to recapture previously used commands. The command histories are maintained until it is cleared using the *Clear Command History* selection from the *Edit* menu. Similar choices exist for the *Workspace* and for the *Command* windows. These latter two clearing operations will be discussed subsequently. The left-hand window displays the files in the current directory.

To bring up the MATLAB Editor/Debugger, which provides the preferred means to create and run programs, one clicks on the white rectangular icon that

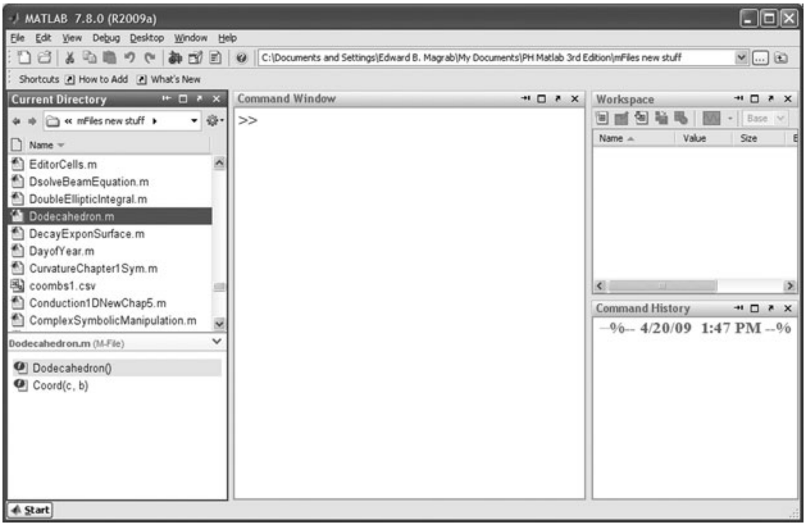


Figure 1.1 MATLAB default windows.

appears under *File* in the left uppermost corner of the window. This results in the configuration shown in Figure 1.2. Other windows can be employed and can be accessed from the *View* menu. To eliminate any of the windows, simply close it by clicking on the × in its respective upper right-hand corner. One way to configure these windows is to use only the command window and the editor window and to call up the other windows when needed. One such configuration of these two windows is shown in Figure 1.3. Upon restarting MATLAB, the system will remember this configuration and this arrangement of the windows will appear.

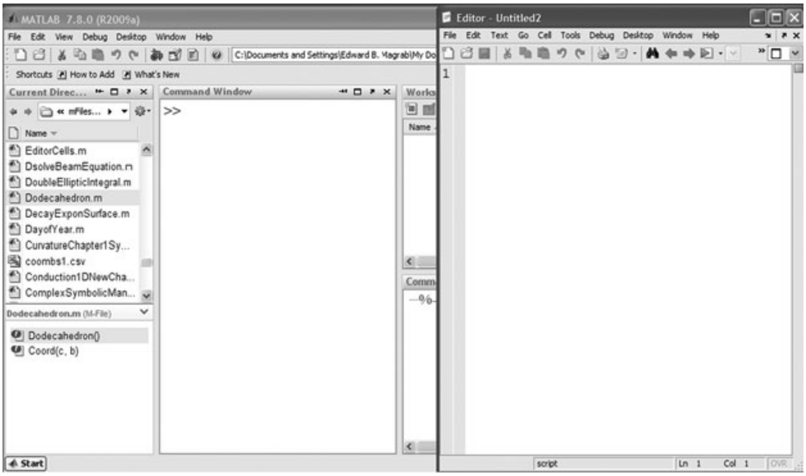


Figure 1.2 MATLAB default windows and the Editor.

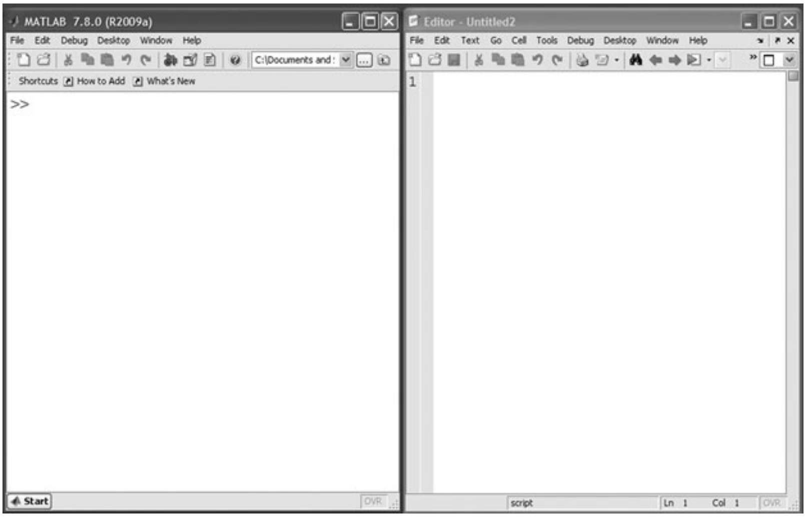


Figure 1.3 MATLAB command window (left) and the Editor (right) after closing the command history, current directory, and workspace windows and opening the Editor.

1.2.2 Preliminaries—Command Window Management

During any MATLAB session—that is, during any time until the program is exited—MATLAB retains in its memory the most recently obtained values of all variables defined by each expression that has been either typed in the command window or evaluated from a script file, unless the `clear` function is invoked. The clearing of the variables in the workspace can also be obtained by selecting *Clear Workspace* from the *Edit* pull-down menu, as shown in Figure 1.4. The `clear` function deletes all the variables from memory. The numerical values most recently assigned to these variables are accessible anytime during the session (provided that `clear` hasn't been used) by simply typing the variable's name or by using it in an expression.

Typing performed in the MATLAB command window remains in the window and can be accessed by scrolling back until the scrolling memory has been exceeded, at which point the earliest entered information has been lost. However, the expressions evaluated from the execution of a script file are not available in the command window, although the variable names and their numerical values are available as indicated in the preceding paragraph. This record of previously typed expressions in the command window can be removed by going to the *Edit* pull-down menu at the top of the MATLAB command window and selecting *Clear Command Window*, which clears the MATLAB command window, but does not delete the variables, which have to be removed by using `clear`. Refer to Figure 1.4. One could also clear the command window by typing `clc` in the command window. In addition, the copy and paste icons can be used either to reproduce previously typed expressions in the current (active) line in the MATLAB command window or to paste MATLAB expressions from the MATLAB command window into the Editor or vice versa.

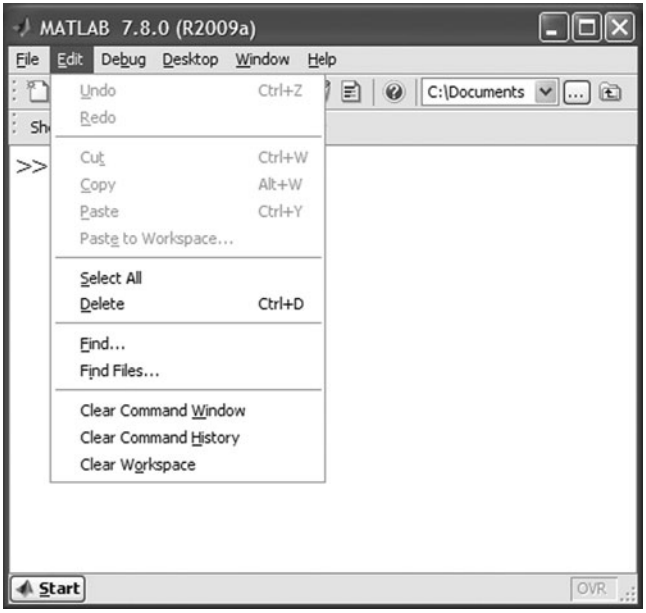


Figure 1.4 *Edit* pull-down menu selections.

For a listing of what variables have been created since the last application of `clear`, one either types `whos` in the MATLAB command window or goes to the pull-down *View* menu and selects *Workspace*, which opens a window with this information. Either method reveals the names of the variables, their size, the number of bytes of storage that each variable uses, and their class: double (8 byte numerical value), which is discussed in Chapter 2; string (literal), which is discussed in Section 3.1; symbolic, which is discussed in Section 1.4; cell, which is discussed in Section 3.4; or function, which is discussed in Section 5.2. The *Workspace* window can be unlocked from its default location by clicking on the icon next to the × in its upper right-hand corner. When one is done with the window, it can be minimized so that this information is readily available for the next time. To make the numbers that appear in the command window more readable, MATLAB offers several options with the `format` function. Two functions that are particularly useful are

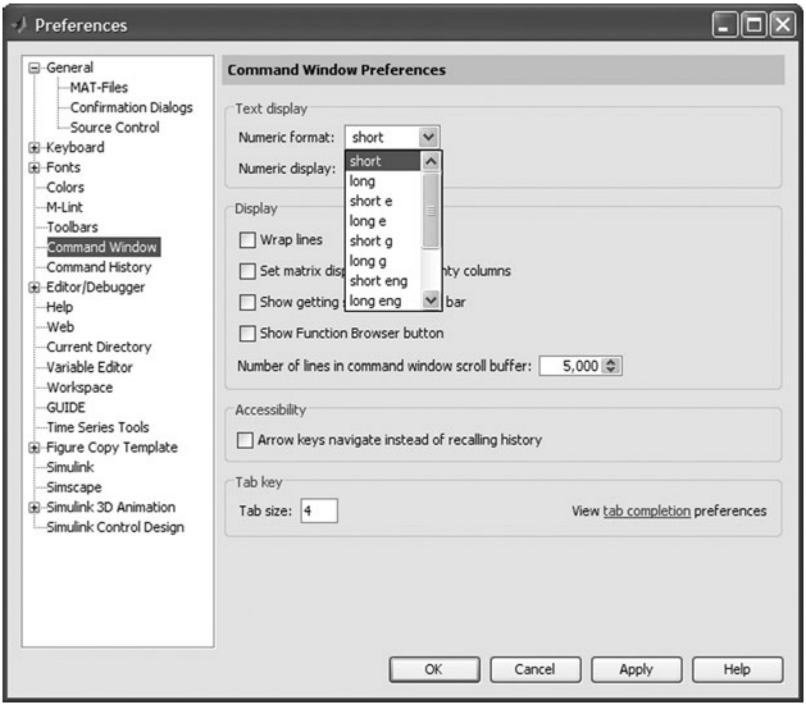
```
format compact
```

and

```
format long e
```

The former removes empty (blank) lines and the latter provides a toggle from the default format of 5 digits to a format with 16 digits plus a 3-digit exponent. The `format long e` option is useful when debugging scripts that produce numbers that either change by very small amounts or vary over a wide range. To toggle back to the default settings, one types the command

```
format short
```



Numeric format: short

Numeric display:

□ Wrap lines

□ Set matrix disp... ...nty columns

□ Show getting ... bar

□ Show Function Browser button

Number of lines in command window scroll buffer: 5,000

□ Arrow keys navigate instead of recalling history

Tab size: 4

Figure 1.5 *Preferences* menu selection for command window format.

These attributes can also be changed by selecting *Preferences* from the *File* pull-down menu and selecting *Command Window* as shown in Figure 1.5. The changes are then made by selecting the desired format from the list of available formats. The different formats that are available are listed in Table 1.1.

Two keyboard entries that are very useful are ^c (Ctrl and c simultaneously) and ^p (Ctrl and p simultaneously). Application of ^p places in the MATLAB

TABLE 1.1 Examples of the Command Window `format` Options

Option	Display number > 1	Display $0 <$ number < 1
`short`	444.4444	0.0044
`long`	4.444444444444445e+002	0.004444444444444
`short e`	4.4444e+002	4.4444e-003
`long e`	4.444444444444445e+002	4.444444444444444e-003
`short g`	444.44	0.0044444
`long g`	444.444444444444	0.00444444444444444
`short eng`	444.4444e+000	4.4444e-003
`long eng`	444.444444444444e+000	4.44444444444444e-003
`rational`	4000/9	1/225
`hex`	407bc71c71c71c72	3f723456789abcdf
`bank`	444.44	0.00

command window the last entry typed from the keyboard, which can then be implemented by pressing *Enter*. In addition, prior to pressing *Enter*, one can modify the expression. If *Enter* is not pressed and instead ^p is entered again, then the next most recently typed entry replaces the most recent entry, and so on. This same result can be obtained using the up-arrow (↑) and down-arrow (↓) keys. The application of ^c either aborts a running program or exits a paused program.

1.2.3 Executing Expressions from the MATLAB Command Window—Basic MATLAB Syntax

MATLAB permits the user to create variable names with a length of up to sixty-three alphanumeric characters, with the characters after the sixty-third being ignored. Each variable name must start with either an uppercase or lowercase letter, which can then be followed by any combination of uppercase and lowercase letters, numbers, and the underscore character (_). No blank spaces may appear between these characters. Variable names are case sensitive, so a variable named *junk* is different from *junK*. There are two commonly used conventions: one that uses the underscore and the other that uses capital letters. For example, if the exit pressure is a quantity that is being evaluated, then two possible definitions that could be defined in a MATLAB command line, script, or function are *exit_pressure* and *ExitPressure*. There are, however, several variable names called *keywords* that are explicitly reserved for MATLAB as part of its programming language. These keywords, which are listed in Table 1.2, may never be used as variable names. The usage of most of these keywords will be given in the subsequent chapters.

Creating suitable variable names is a trade-off between easily recognizable and descriptive identifiers and readability of the resulting expressions. If the expression has many variable names, then short variable names are preferable. This becomes increasingly important as the grouping of the symbols becomes more complex. Shorter names tend to decrease errors caused by the improper grouping of terms and the placement of arithmetic operators. In addition, one can neither use Greek letters literally as variable names nor can one use subscripts and superscripts. However, one can spell the Greek letter or can simply precede the subscript by the underscore character. For example, one could represent σ_r as sigma_r and c_3 as c3 or c_3.

TABLE 1.2 Keywords Reserved Explicitly for the MATLAB Programming Language

`break`	`global`
`case`	`if`
`catch`	`otherwise`
`continue`	`persistent`
`else`	`return`
`elseif`	`switch`
`end`	`try`
`for`	`while`
`function`	

We shall illustrate the two ways in which one can evaluate expressions in MATLAB: one from the command window and the other from the Editor. When using the command window, one must define one or more variables at the prompt (>>). MATLAB requires that all variables, except those defined as symbolic quantities and used by the Symbolic toolbox, be assigned numerical values prior to being used in an expression. The assignment operator is the equal sign (=). Typing the variable name, an equal sign, the numerical value(s), and then *Enter* performs the assignment. Thus, if we wish to assign three variables p, x, and k the values 7.1, 4.92, and −1.7, respectively, then the following interaction in the MATLAB command window is obtained.

```
» p = 7.1     ←——— User types and hits Enter
p =          ┐
  7.1000     ┘←——— System response
» x = 4.92    ←——— User types and hits Enter
x =          ┐
  4.9200     ┘←——— System response
» k = -1.7    ←——— User types and hits Enter
k =          ┐
  -1.7000    ┘←——— System response
```

This command window interaction was obtained using `format compact`.

In order to suppress the system's response, one places a semicolon (;) as the last character of the expression. Thus, typing each of the following three expressions on their respective lines followed by *Enter*, gives

```
»  p = 7.1;
»  x = 4.92;
»  k = -1.7;
»
```

MATLAB also lets one place several expressions on one line, a line being terminated by *Enter*. In this case, each expression is separated by either a comma (,) or a semicolon (;). When a comma is used, the system echoes the input. Thus, if the following is typed,

```
p = 7.1, x = 4.92, k = -1.7
```

then the system responds with

```
p =
  7.1000
x =
  4.9200
k =
  -1.7000
»
```

The use of semicolons instead of the commas would have suppressed this output.

Arithmetic Operators

The five arithmetic operators to perform addition, subtraction, multiplication, division, and exponentiation are +, −, *, /, and ^, respectively. For example, the mathematical expression

$$t = \left(\frac{1}{1 + px} \right)^k$$

can be written in MATLAB as

```
t = (1/(1+p*x))^k
```

when p, x, and k are scalar quantities. The quantities p, x, and k must be assigned numerical values prior to the execution of this statement. If this has not been done, then an error message to that effect will appear. Assuming that the quantities p, x, and k were those entered previously in the command window and not cleared, the system returns

```
t=
  440.8779
```

Mathematical Operations Hierarchy

The parentheses in the MATLAB expression for t have to be used so that the mathematical operations are performed on the proper collections of quantities in their proper order within each set of parentheses. There is a hierarchy and a specific order that MATLAB uses to compute arithmetic statements. One can take advantage of this to minimize the number of parentheses. However, parentheses that are unnecessary from MATLAB's point of view can still be used to remove visual ambiguity and to make the expression easier to understand. The parentheses are the highest level in the hierarchy, followed by exponentiation,[1] then by multiplication and division, and finally by addition and subtraction. Within each set of parentheses and within each level of hierarchy, MATLAB performs its operations from left to right. Consider the examples shown in Table 1.3 involving the scalar quantities c, d, g, and x. The MATLAB function

```
sqrt(x)
```

takes the square root of its argument x. Notice that in the first row of Table 1.3 the parentheses around the quantity $x + 2$ are required. If they weren't used; that is, the relation was written as

```
1-d*c^x+2
```

then, we would have coded the expression $1 - dc^x + 2$. The same reasoning is true for the exponent in the third row of the table. In the third row, notice that the form

```
2*c^(x+2)/d
```

[1] The matrix transpose, which is discussed in Section 2.2, is also on the same level as exponentiation. The matrix transpose symbol in MATLAB is the apostrophe (').

TABLE 1.3 Examples of MATLAB Syntax: All Quantities Are Scalars

Mathematical expression	MATLAB expression
$1 - dc^{x+2}$	1−d*c^(x+2)
$dc^x + 2$	d*c^x+2 *or* 2+d*c^x
$(2/d)c^{x+2}$	(2/d)*c^(x+2) *or* 2/d*c^(x+2) *or* 2*c^(x+2)/d
$(dc^x + 2)/g^{2.7}$	(d*c^x+2)/g^2.7
$\sqrt{dc^x + 2}$	`sqrt`(d*c^x+2) *or* (d*c^x+2)^0.5

is correct because of the hierarchy rules. The innermost set of parenthesis is $(x + 2)$ and is computed first. Then, exponentiation is performed, because this is the next highest level of the computational order. Next, the multiplications and divisions are performed from left to right, because the three quantities, 2, the result of c^{x+2}, and d are all on the same hierarchical level: multiplication and division.

1.2.4 Clarification and Exceptions to MATLAB's Syntax

Scalars versus Arrays

MATLAB considers all variables as arrays of numbers; therefore, when using the five arithmetic operators (+, −, *, /, and ^), these operations have to obey the rules of linear algebra. These rules are discussed in Section 2.6. When the variables are scalar quantities, that is, when they are arrays of one element (one row and one column), the usual rules of algebra apply. However, one can operate on arrays of numbers and suspend the rules of linear algebra by using dot operations, which are discussed in Section 2.5. Dot operations provide a means of performing a sequence of arithmetic operations on arrays of the same size on an array element by array element basis. When using the dot operators, the multiplication, division, and exponentiation operators become .*, ./, and .^, respectively.

Blanks

In an arithmetic expression, the use of blanks can be employed with no consequence. Variable names on the right-hand side of the equal sign must be separated by one of the five arithmetic operators, a comma (,), or a semicolon (;).

There are **two exceptions** to this usage of blanks. The first is when one represents a complex number $z = a + jb$ or $z = a + ib$, where $i = j = \sqrt{-1}$. Consider the following script

```
a = 2; b = 3;
z = a +1j*b % or a+b*1j
```

which upon execution gives

```
z =
  2.0000 + 3.0000i
```

The number 1 that precedes the `j` is not required, but it is strongly recommended by MATLAB that it be used for increased speed and robustness. Notice that the program used j, but the system responded with an i, showing the system's equivalent treatment of these two quantities. Also, note that j was not defined previously; therefore, MATLAB assumes that it is equal to $\sqrt{-1}$. However, when a and b are replaced with numerical values directly in the expression, no arithmetic operator is required. Thus, the script

```
a = 2; b = 3;
z = (a+1j*b)*(4-7j)
```

upon execution gives

```
z = 29.0000 – 2.0000i
```

In this usage, the j (or i) must follow the number without a space.

The second exception is when we express a number in exponential form such as $x = 4.56 \times 10^{-2}$. This number can be expressed as either

```
x = 0.0456
```

or

```
x = 4.56*10^-2
```

or as

```
x = 4.56e-2
```

The last expression is the exponential form. Notice that no arithmetic operator is placed between the last digit of the magnitude and the 'e'. The maximum number of digits that can follow the 'e' is 3. Thus, if we desired the quantity x^2 and we used the exponential form, the script would be

```
x2 = 4.56e-2^2
```

which upon execution displays to the command window

```
x2 =
   0.0021
```

If the value of x were 4.56×10^2, the implied '+' sign may be omitted; that is, the square of x can be written as either

```
x = 4.56e2^2
```

or

```
x = 4.56e+2^2
```

System Assignment of Variable Names

When the command window is used as a calculator and no assignment of the expression has been made, MATLAB will always assign the answer to the variable named *ans*. For example, let us assume that one wants to determine the value of the cosine of $\pi/3$. We simply type in the command window[2]

```
cos(pi/3)
```

and the system will respond with

```
ans =
   0.5000
```

The variable *ans* can now be used as one would use any other variable name. If we now want to add 2 to the previous result, then we would type in the command window

```
      ans+2
```

and the system would respond with

```
ans =
   2.5000
```

Thus, *ans* has been assigned the new value of 2.5. The previous value of *ans* (= 0.5) is no longer available.

Complex Numbers

MATLAB permits one to mix real and complex numbers without any special operation on the part of the user. Thus, if one types in the command window

```
z = 4 + sqrt(-4)
```

then the system would display

```
z =
   4.0000 + 2.0000i
```

As another example, consider the evaluation of the expression i^i, which is obtained by typing in the command window

```
z = i^i
```

The execution of this expression gives

```
z =
   0.2079
```

since $i^i = (e^{\pi i/2})^i = e^{-\pi/2} = 0.2079$.

[2] In the command window, the alphanumeric characters will appear in the same font. We are using different fonts to enhance the readability of the expressions as mentioned in Section 1.1.3.

TABLE 1.4 Some Elementary MATLAB Functions

Mathematical function	MATLAB expression
e^x	`exp(x)`
$e^x - 1 \qquad x << 1$	`expm1(x)`
$\sqrt{x}$	`sqrt(x)`[a]
$\ln(x)$ or $\log_e(x)$	`log(x)`[b]
$\log_{10}(x)$	`log10(x)`
$\lvert x \rvert$	`abs(x)`
signum(x)	`sign(x)`
$\log_e(1+x) \qquad x << 1$	`log1p(x)`
$n!$	`factorial(n)`[c]
All prime numbers $\leq n$	`primes(n)`

[a] If x is an array with each element in the array > 0, use `realsqrt(x)` to increase computational speed.

[b] If x is an array with each element in the array > 0, use `reallog(x)` to increase computational speed.

[c] 15 digits accuracy for $n \leq 21$; for larger n, only the magnitude and the 15 most significant digits will be correct.

1.2.5 MATLAB Functions

MATLAB provides a large set of elementary functions and specialized mathematical functions. Some of the elementary functions and some built-in constants are listed in Tables 1.4, 1.5, and 1.6. In Tables 1.4 and 1.5, x can be a real or complex scalar, a vector, or a matrix; the quantity n is a real positive integer. The definitions of vectors and matrices and their creation in MATLAB are given in Sections 2.3 and 2.4. In Table 1.7, we have listed the relational operators that are used in MATLAB.

Several MATLAB functions are available to round decimal numbers to the nearest integer value using different rounding criteria. These functions are `fix`, `round`, `ceil`, and `floor`. The results of the different operations performed by these four functions are summarized in Table 1.8.

TABLE 1.5 MATLAB Trigonometric and Hyperbolic Functions

	Trigonometric			Hyperbolic	
Function	(x in radians)	(x in degrees)	Inverse		Inverse
sine	`sin(x)`	`sind(x)`	`asin(x)`	`sinh(x)`	`asinh(x)`
cosine	`cos(x)`	`cosd(x)`	`acos(x)`	`cosh(x)`	`acosh(x)`
tangent	`tan(x)`	`tand(x)`	`atan(x)`[†]	`tanh(x)`	`atanh(x)`
secant	`sec(x)`	`secd(x)`	`asec(x)`	`sech(x)`	`asech(x)`
cosecant	`csc(x)`	`cscd(x)`	`acsc(x)`	`csch(x)`	`acsch(x)`
cotangent	`cot(x)`	`cotd(x)`	`acot(x)`	`coth(x)`	`acoth(x)`

[†] `atan2(y, x)` is the four-quadrant version, which must be used when the signs of y and x can each be positive or negative.

TABLE 1.6 Some MATLAB Constants and Special Quantities

Mathematical quantity or operation	MATLAB expression	Comments
π	pi	3.141592653589793
$\sqrt{-1}$	i or j	Used to indicate a complex quantity as a + 1j*b, where *a* and *b* are real.
Floating point relative accuracy	eps	The distance from 1.0 to the next largest floating-point number ($\approx 2.22 \times 10^{-16}$)
∞	inf	–
$0/0$, $0\times\infty$, ∞/∞	NaN	Indicates an undefined mathematical operation.
Largest floating-point number before overflow	realmax	$\approx 1.7977e+308$
Smallest floating-point number before underflow	realmin	$\approx 2.2251e-308$

TABLE 1.7 MATLAB Relational Operators

Conditional	Mathematical symbol	MATLAB symbol
equal	=	==
not equal	≠	~=
less than	<	<
greater than	>	>
less than or equal	≤	<=
greater than or equal	≥	>=

TABLE 1.8 MATLAB Decimal-to-Integer Conversion Functions

MATLAB function	*x*	*y*	Description
y = fix(x)	2.7	2.0000	Round toward zero
	−1.9	−1.0000	
	2.49 − 2.51j	2.0000 − 2.0000i	
y = round(x)	2.7	3.0000	Round to nearest integer
	−1.9	−2.0000	
	2.49 − 2.51j	2.0000 − 3.0000i	
y = ceil(x)	2.7	3.0000	Round toward infinity
	−1.9	−1.0000	
	2.49 − 2.51j	3.0000 − 2.0000i	
y = floor(x)	2.7	2.0000	Round toward minus infinity
	−1.9	−2.0000	
	2.49 − 2.51j	2.0000 − 3.0000i	

TABLE 1.9 MATLAB Complex Number Manipulation Functions

MATLAB function	z	y	Description
z = `complex`(a, b)	a+b*j	–	Form complex number; a and b real
y = `abs`(z)	3+4j	5	Absolute value: $\sqrt{a^2 + b^2}$
y = `conj`(z)	3+4j	3−4j	Complex conjugate
y = `real`(z)	3+4j	3	Real part
y = `imag`(z)	3+4j	4	Imaginary part
y = `angle`(z)	a+b*j	`atan2`(b, a)	Phase angle in radians: $-\pi \leq y \leq \pi$

There are also several MATLAB functions that are used to create and manipulate complex numbers. These are `complex`, `abs`, `conj`, `real`, `imag`, and `angle`. The operations of these six functions are summarized in Table 1.9. Lastly, in Table 1.10, we have listed several specialized mathematical functions and elementary descriptive statistical functions. Additional MATLAB functions for various classes of mathematical operations are given in subsequent chapters. Numerous MATLAB array creation and manipulation functions are summarized in Table 2.1. MATLAB functions that can

TABLE 1.10 Several Specialized Mathematical Functions and Descriptive Statistical Functions*

Mathematical function	MATLAB Expression	Description
Specialized mathematics		
Ai(x), Bi(x)	`airy`(0,x), `airy`(2,x)	Airy functions
$I_\nu(x)$	`besseli`(nu, x)	Modified Bessel function of first kind
$J_\nu(x)$	`besselj`(nu, x)	Bessel function of first kind
$K_\nu(x)$	`besselk`(nu, x)	Modified Bessel function of second kind
$Y_\nu(x)$	`bessely`(nu, x)	Bessel function of second kind
$B(x,w)$	`beta`(x, w)	Beta function
$K(m)$, $E(m)$	`ellipke`(m)	Complete elliptic integrals of first and second kind
erf(x), erfc(x)	`erf`(x), `erfc`(x)	Error function and complementary error function
$E_1(z)$	`expint`(x)	Exponential integral
$\Gamma(a)$	`gamma`(a)	Gamma function
$P_n^m(x)$	`legendre`(n, x)	Associated Legendre function
Descriptive Statistics		
maximum value of x	`max`(x)	Largest element(s) in an array of values
μ	`mean`(x)	Average or mean value of array of values
median	`median`(x)	Median value of an array of values
minimum value of x	`min`(x)	Smallest element(s) in an array of values
mode	`mode`(x)	Most frequent values in an array of values
σ or s	`std`(x)	Standard deviation of an array of values
σ^2 or s^2	`var`(x)	Variance of an array of values

*See Tables 8.2 and 8.21 for additional statistical functions.

be used to create and manipulate string expressions (literals) are summarized in Table 3.1 and those that can be used to analyze data arrays and mathematical expressions are summarized in Table 5.4. Lastly, MATLAB functions that are used to create 2D and 3D graphic displays are summarized in Tables 6.15 and 7.10, respectively. Specialized functions that are used to model and analyze control systems are summarized in Table 10.1. Functions that deal with optimization of systems are summarized in Table 13.8, and those that deal with statistics are summarized in Table 8.21. Functions that are used by the Symbolic toolbox are summarized in Table 1.12.

In addition to the five arithmetic operators (+, −, *, /, and ^) that were discussed previously, there are several other symbols that are reserved by MATLAB to have special meaning. These are listed in Table 1.11 and their usage is discussed in Chapters 1–5.

TABLE 1.11 Special Characters and a Summary of Their Usage†

Character	Name	Usage
.	Period	(a) Decimal point. (b) Part of arithmetic operators to indicate a special type of vector or matrix operation, called the dot operation, such as c = a.*b. (c) Delimiter in a structure, such as *name.first*.
,	Comma	(a) Separator within parentheses of matrix elements such as b(2,7) and functions such as besselj(1, x) or brackets creating vectors such as v = [1, x] or the output of function arguments such as [x, s] = max(a). (b) Placed at the end of an expression when several expressions appear on one line.
;	Semicolon	(a) Suppresses display of the results when placed at end of an expression. (b) Indicates the end of a row in matrix creation statement such as *m* = [x y z; a b c].
:	Colon	(a) Separator in the vector creation expression x = a:b:c. (b) For a matrix *z*, it indicates “all rows” when written as z(:,k) or “all columns” when written as z(k,:).
( )	Parentheses	(a) Denotes subscript of an element of matrix z, where z(j, k) ↔ z_{jk} is the element in row *j* and column *k*. (b) Delimiters in mathematical expressions such as a^(b+c). (c) Delimiters for the arguments of functions, such as sin(x).
[ ]	Brackets	Creates an array of numbers, either a vector or a matrix, or an array of strings (literals).
{ }	Braces	Creates a cell array.
%	Percentage	Comment delimiter; used to indicate the beginning of a comment wherein MATLAB ignores everything to its right. The exception is when it is used inside a pair of single quotes to define a string such as a = 'p = 14 % of the total'.

(*Continued*)

TABLE 1.11 *(Continued)*

Character	Name	Usage
%%	Percentage	Used to delimit the start and end of a cell in the MATLAB Editor, which is a portion of program code.

%{
%}
	Percentage and brace	Used to enclose a block of contiguous comment lines. Comments placed between these delimiters do not have to be preceded by a %. However, no text can be placed on the line containing these delimiters.

'	Quote or Apostrophe	(a) '*Expression*' indicates that *Expression* is a string (literal). (b) Indicates the transpose of a vector or matrix. If the vector or matrix is complex, then, in addition, the complex conjugate of each element is taken.
...	Ellipsis	Continuation of a MATLAB expression to the next line. Used to create code that is more readable.
	Blank	Context dependent: either ignored, indicates a delimiter in a data creation statement such as c = [a b], is a character in a string statement, or is a delimiter in an optional form of certain MATLAB functions such as `syms` a b and `format long`.
@	At sign	Constructs a function handle by placing @ before a function name, such as @**FunctionName**.
\	Backslash	(a) A mathematical operator to perform certain matrix operations. (b) A character that is used to display Greek letters and mathematical symbols in graph annotation.

†See Table 4.1 for a list of logical operators.

Overloading

Although the choice of variable names is virtually unlimited, one should avoid choosing names that are the same as those used for MATLAB's built-in functions or for user-created functions. MATLAB permits one to overload a built-in function name. For example, the following expression is a valid MATLAB expression

```
cos = a+b*x^2;
```

However, since 'cos' is also the name used for the cosine function, cos(x), this is a poor choice for a variable name and it is strongly recommended that such redefinitions be avoided. An exception to this recommendation is when all quantities in one's program are real variables. In this case, overloading i and j will not cause any unexpected results.

Example 1.1 Usage of MATLAB functions

To illustrate the use of the MATLAB's built-in functions, consider the following expression to be evaluated at $x = 0.1$ and $a = 0.5$:

$$y = \sqrt{|e^{-\pi x} - \sin x/\cosh a - \ln_e(x + a)|}$$

This expression is evaluated with the following script:

```
x = 0.1;    a = 0.5;
y = sqrt(abs(exp(-pi*x)-sin(x)/cosh(a)-log(x+a)))
```

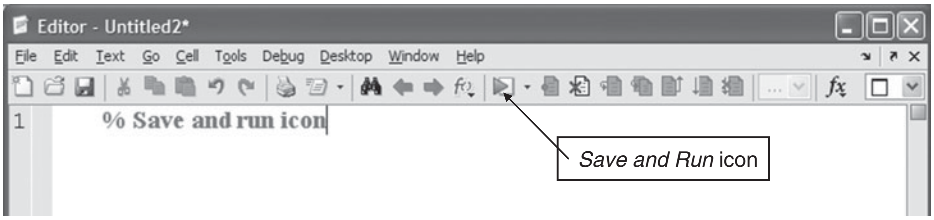
where the MATLAB function pi = π. Upon execution, the following result is displayed in the command window:

```
y =
   1.0736
```

1.2.6 Creating Scripts and Executing Them from the MATLAB Editor[3]

A script file is a file that contains a list of commands, each of which will be operated on as if it were typed at the command line in the command window. A script file is created in a word processor, a text editor, or the MATLAB Editor/Debugger, and saved as a text file with the suffix ".m". Such files are called M-files. If a word processor or text editor is used, then the file is executed by typing the file name without the suffix ".m" in the MATLAB command window. If the MATLAB Editor is used, one can use the previous method or can click on the *Save and Run* icon on the top of the Editor's window as shown in Figure 1.6. However, before one can use this icon, the file must be saved the first time by using the *Save As* option from the *File* pull-down menu. The file-naming convention is the same as that for variable names: It must start with an upper or lower case letter followed by up to sixty-two contiguous alphanumeric characters and the underscore character. No blank spaces are allowed in file names. (This is different from what is allowed by the Windows operating system.) When the MATLAB Editor/Debugger is used, an ".m" suffix will be affixed to the file name.

Another form of a file created in the Editor is the function file. These functions are created because one of MATLAB's built-in functions requires them or one wants to use them to better manage the programming task. Functions differ from scripts in that they allow a structured approach to managing the programming task. They differ from expressions entered at the command line in that MATLAB allots them their own private workspace and they have formally defined input–output relationships within the MATLAB environment. Functions are discussed in Chapter 5.



```
% Save and run icon
```

Figure 1.6 *Save and Run* (execute) icon in the Editor.

[3] In terms of execution time, functions generally run faster than scripts. Functions are introduced in Chapter 5, where from that point forward, scripts are used less frequently.

Script files are usually employed in those cases where:

1. The program will contain more than a few lines of code.
2. The program will be used again.
3. A permanent record is desired.
4. It is expected that occasional upgrading will be required.
5. Substantial debugging is required.
6. One wants to transfer the listing to another person or organization.

A script or a function typically has the following attributes:

1. *Documentation*, which at a minimum indicates the:

   Purpose and operations performed
   Programmer's name
   Date originated
   Date(s) revised
   Description of the input variable names: number, meaning, and type (class)
   Description of the output variable names: number, meaning, and type (class)

2. *Input*, which for those quantities that are entered externally, includes numerous checks to ensure that all input values have the qualities required for the script/function to work properly.
3. *Initialization*, where the appropriate variables are assigned their initial values.
4. *Computation*, where the numerical evaluations are performed.
5. *Output*, where the results are presented as annotated graphical and/or numerical quantities.

The MATLAB Editor

The MATLAB Editor has several features that make it especially suitable for creating scripts and functions files.

Converting Executable Code to Comment Statements During program development, one can convert one or more lines of code to comment statements and convert them back to executable statements. This switching from commented quantities to executable code is done with the cursor; one highlights the lines to be converted, goes to the *Text* pull-down menu on top of the window, and then selects the appropriate action—comment or uncomment— as shown in Figure 1.7.

Visual Aids When creating program flow control structures as described in Chapter 4, one can indent lines of code to improve readability. This can be done in two ways. One is to go to the *Text* pull-down menu and select *Smart Indent*, as shown in Figure 1.7. The second way is to use the cursor to highlight the lines of code that are to be indented and press ^i (Ctrl and i simultaneously). An example of these actions is shown in Figure 1.8.

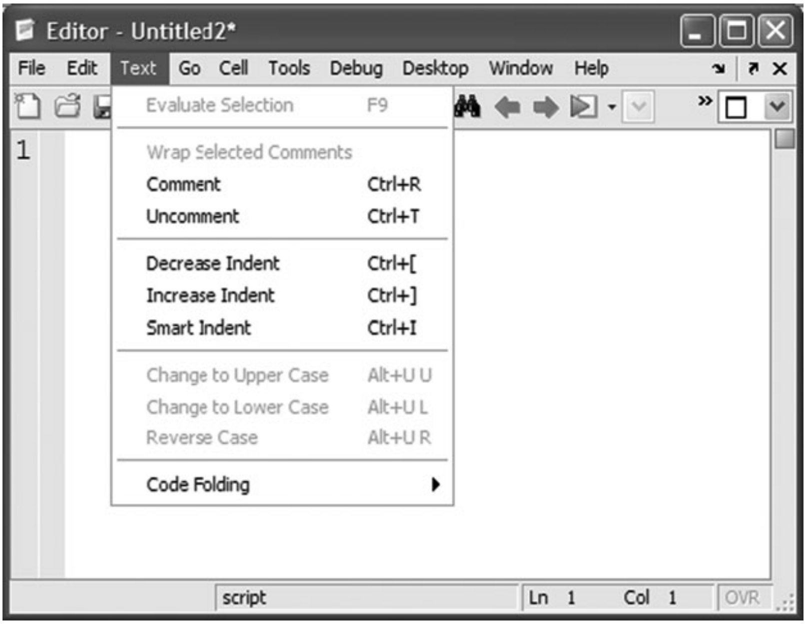
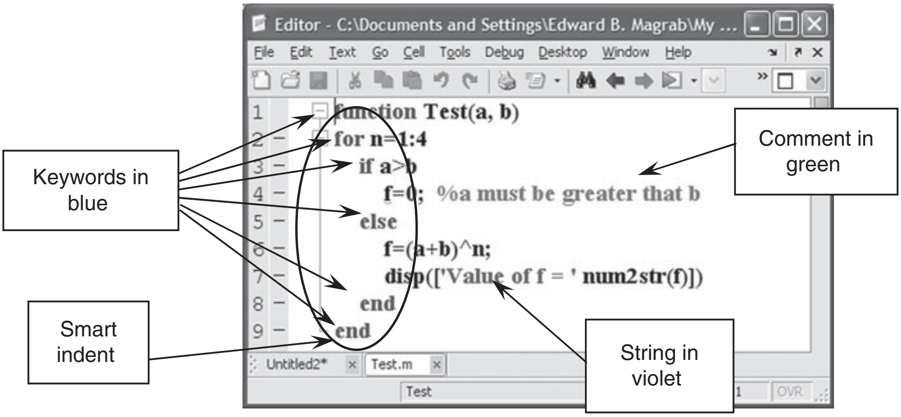


Figure 1.7 *Text* pull-down menu in the Editor.

The Editor also employs a color scheme. Keywords appear in blue, letters and numbers appearing between a pair of apostrophes are in violet, and comments appear in green. These features also are indicated in Figure 1.8.

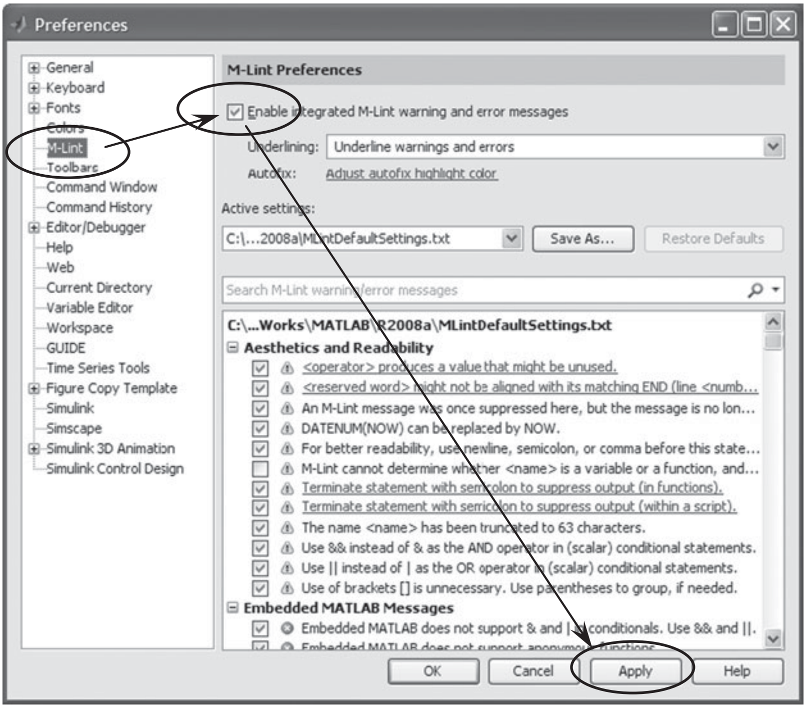
Parentheses Grouping The Editor also keeps track of open and closed parentheses when typing a line of code. Every time a closed parenthesis is typed, the appropriate open parenthesis to the left is momentarily highlighted or underscored. This can be of great aid in verifying the grouping of terms.


Editor - C:\Documents and Settings\Edward B. Magrab\My ...
File Edit Text Go Cell Tools Debug Desktop Window Help
Keywords in blue
Smart indent
Comment in green
String in violet
Untitled2*
Test.m
Test
1
OVR

```
function Test(a, b)
for n=1:4
    if a>b
        f=0;  %a must be greater that b
    else
        f=(a+b)^n;
        disp(['Value of f = ' num2str(f)])
    end
end
```

Figure 1.8 Color visual aids and “smart” indenting in the Editor.

M-Lint In addition to the features mentioned above, the Editor also has a real-time syntax evaluator called *M-Lint*. When this feature is enabled, all syntax errors are detected as the code is being entered into the Editor. In other words, all syntax errors will be detected prior to executing the program. M-Lint is activated by going to *Preferences*, selecting *M-Lint*, and as shown in Figure 1.9, checking the box that enables the notification of warnings and error messages. Then, one clicks on *Apply* at the bottom of the window. In Figure 1.10, we have shown a very small program in which we have deliberately typed an expression that will generate a warning and another expression that contains an error. These errors are brought to our attention with color-coded horizontal bars in the right-hand column of the window. The top bar is orange, which indicates a warning. For this type of error, the program will execute. We see that in this case, line five does not end with a semicolon. For line six, the bar is red, which indicates that until the error is corrected, the program will not execute. To determine the error, one places the cursor over the bar, and the error message is displayed. In this case, it is indicating that a parenthesis is missing. The overall error status of the program is given by a colored square at the top of the right-hand column: green for no errors or warnings; orange for warnings, but no errors; and red for errors that prohibit program execution. In long programs, these bars do not necessarily align themselves with the line that contains the error. One has to place the cursor over each bar to determine the line number to which it is referring.



M-Lint Preferences

☑ Enable integrated M-Lint warning and error messages

Underlining: Underline warnings and errors

Autofix: Adjust autofix highlight color

Active settings: C:\...2008a\MLintDefaultSettings.txt

Search M-Lint warning/error messages

C:\...Works\MATLAB\R2008a\MLintDefaultSettings.txt

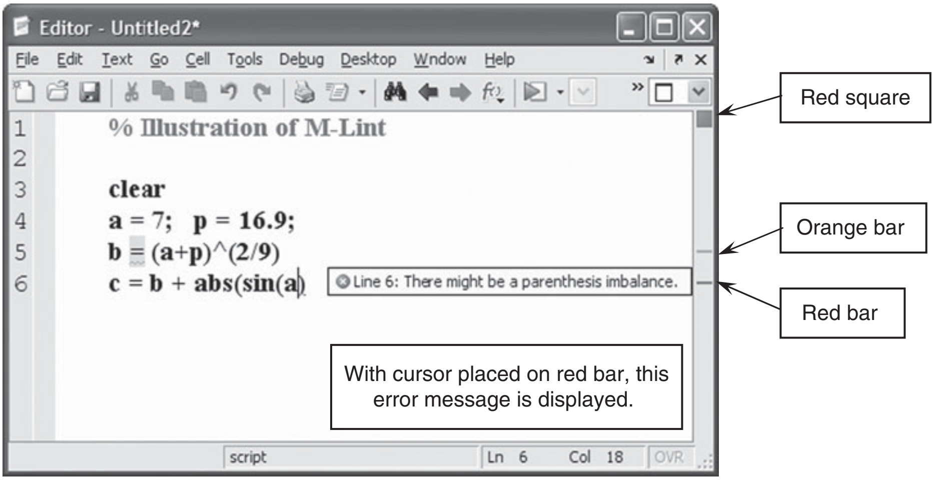
Aesthetics and Readability

- ☑ <operator> produces a value that might be unused.
- ☑ <reserved word> might not be aligned with its matching END (line <numb...
- ☑ An M-Lint message was once suppressed here, but the message is no lon...
- ☑ DATENUM(NOW) can be replaced by NOW.
- ☑ For better readability, use newline, semicolon, or comma before this state...
- □ M-Lint cannot determine whether <name> is a variable or a function, and...
- ☑ Terminate statement with semicolon to suppress output (in functions).
- ☑ Terminate statement with semicolon to suppress output (within a script).
- ☑ The name <name> has been truncated to 63 characters.
- ☑ Use && instead of & as the AND operator in (scalar) conditional statements.
- ☑ Use || instead of | as the OR operator in (scalar) conditional statements.
- ☑ Use of brackets [] is unnecessary. Use parentheses to group, if needed.

Embedded MATLAB Messages

- ☑ Embedded MATLAB does not support & and | in conditionals. Use && and ||.
- ☑ Embedded MATLAB does not support anonymous functions.

Figure 1.9 Enabling *M-Lint* from the *Preferences* menu.



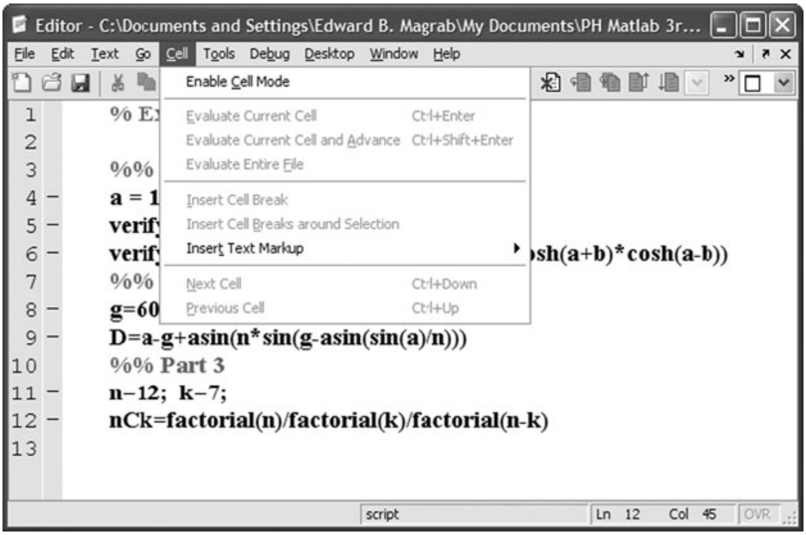
```
% Illustration of M-Lint

clear
a = 7;   p = 16.9;
b = (a+p)^(2/9)
c = b + abs(sin(a)
```

Line 6: There might be a parenthesis imbalance.

With cursor placed on red bar, this error message is displayed.

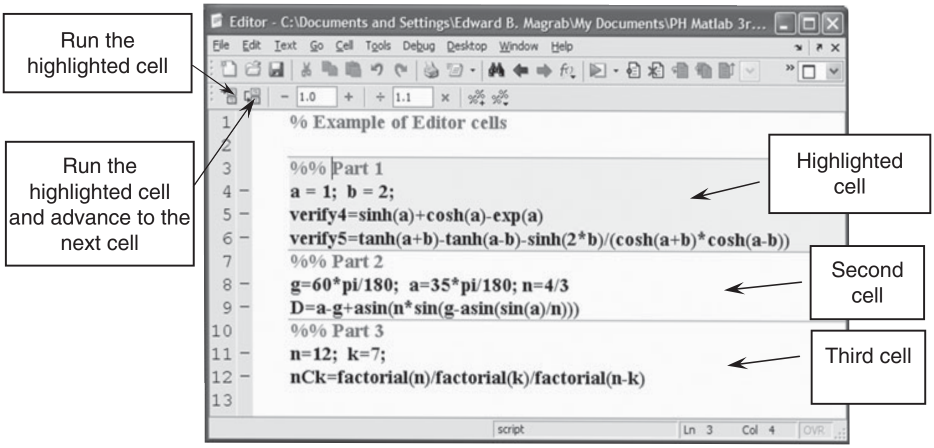
Figure 1.10 An example of *M-Lint* detecting two errors. In the error message, the designation "imbalance" refers to the omission of a right-most parenthesis, whose location has been underscored with a red line (darker shade). The message associated with the orange bar (lighter shade) indicates the omission of a semicolon at the end of the expression.

Cells The last feature of the Editor that we shall discuss is the ability to define blocks of code in a program, which are called cells, such that each cell can then be run independently of any other cell in a program or of any other line of code in the program. The cells are delineated by a pair of percentage signs (%%), one pair placed prior to the first line of code and one pair placed after the last line of the code. The cell feature is activated by clicking on *Cell* and selecting *Enable Cell Mode*, as shown in Figure 1.11. When the cell mode has been enabled, a set of additional icons appear


Editor - C:\Documents and Settings\Edward B. Magrab\My Documents\PH Matlab 3r...
File Edit Text Go Cell Tools Debug Desktop Window Help
Enable Cell Mode
Evaluate Current Cell Ctrl+Enter
Evaluate Current Cell and Advance Ctrl+Shift+Enter
Evaluate Entire File
Insert Cell Break
Insert Cell Breaks around Selection
Insert Text Markup
Next Cell Ctrl+Down
Previous Cell Ctrl+Up
script
Ln 12 Col 45 OVR

```
% E

%%
a = 1
verif
verif                                    sh(a+b)*cosh(a-b))
%%
g=60
D=a-g+asin(n*sin(g-asin(sin(a)/n)))
%% Part 3
n–12;  k–7;
nCk=factorial(n)/factorial(k)/factorial(n-k)
```

Figure 1.11 Enabling the cell feature of the Editor.



Editor - C:\Documents and Settings\Edward B. Magrab\My Documents\PH Matlab 3r...
File Edit Text Go Cell Tools Debug Desktop Window Help
1.0
1.1
script
Ln 3 Col 4
OVR

```
% Example of Editor cells

%% Part 1
a = 1;  b = 2;
verify4=sinh(a)+cosh(a)-exp(a)
verify5=tanh(a+b)-tanh(a-b)-sinh(2*b)/(cosh(a+b)*cosh(a-b))
%% Part 2
g=60*pi/180;  a=35*pi/180; n=4/3
D=a-g+asin(n*sin(g-asin(sin(a)/n)))
%% Part 3
n=12;  k=7;
nCk=factorial(n)/factorial(k)/factorial(n-k)

```

Run the highlighted cell

Run the highlighted cell and advance to the next cell

Highlighted cell

Second cell

Third cell

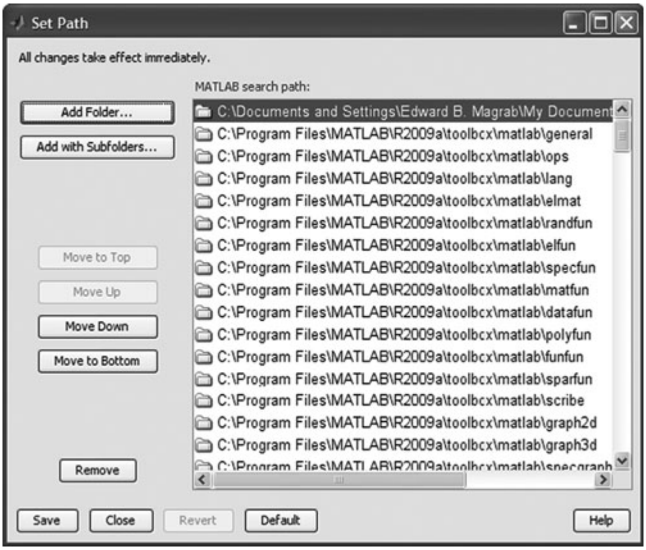
Figure 1.12 Program with three cells. The cell denoted Part 1 has been selected.

as shown in Figure 1.12. In Figure 1.12, we have written a small program that has three parts. Each part is delineated with a pair of parentheses to indicate the extent of the three cells. The first cell, designated as Part 1, has been selected by clicking the cursor anywhere in the region between the parentheses pairs. This results in the area being highlighted with a light tan background. There are two ways to evaluate this cell as indicated in Figure 1.12.

Recommendation Because of the features mentioned above and because of the seamless integration of the Editor and its features with the rest of MATLAB, it is strongly recommended that the reader create all programs, no matter how small, in the Editor and then run them directly from the Editor. It is also recommended that one keep M-Lint enabled.

Executing Programs

In order to execute script files and function files, MATLAB must be provided with the path to the directory in which the files reside. The path information is entered by going to the *File* pull-down menu and selecting *Set Path*. This opens the *Set Path* window shown in Figure 1.13. One then clicks on the *Add Folder* text box and chooses the folder in which the file will reside as shown in Figure 1.14. Before leaving the *Path Browser*, it is suggested that *Save* be selected; this saves the path for the next time MATLAB is used. If one attempts to execute a script from the Editor that is not in the current path, MATLAB will ask, via the pop-up window shown in Figure 1.15, if the current path should be changed to the one in which the file resides. If the answer is yes, one clicks on the *Change Directory* icon. One can also set the current path name by clicking the icon in the command window shown in Figure 1.16. This brings up a directory browser that permits one to select a directory as the current directory. After a selection is made, the system also changes the directories that will appear in the *Editor* when either *Open*, *Save*, or *Save As* is selected in the *File* pull-down menu.



All changes take effect immediately.

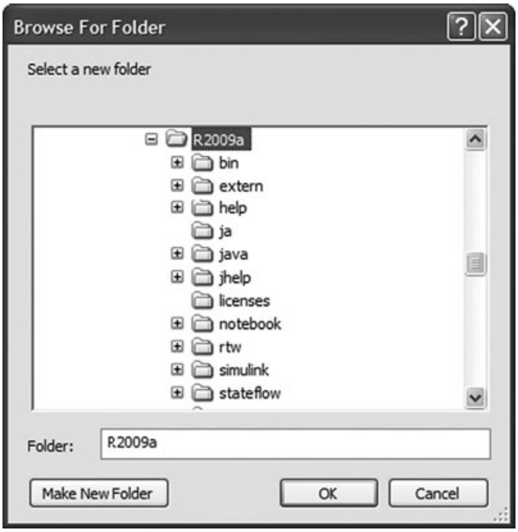
MATLAB search path:

```
C:\Documents and Settings\Edward B. Magrab\My Document
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\general
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\ops
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\lang
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\elmat
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\randfun
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\elfun
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\specfun
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\matfun
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\datafun
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\polyfun
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\funfun
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\sparfun
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\scribe
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\graph2d
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\graph3d
C:\Program Files\MATLAB\R2009a\toolbcx\matlab\specgraph
```

Figure 1.13 *Set Path* window.

If either a script or a function requires the user to enter a numerical value (or a series of numerical values if the quantity is either a vector or a matrix, as discussed in Sections 2.3 and 2.4) from the MATLAB command window, then the script (or function) file contains the statement

VariableName = `input`('Any message')

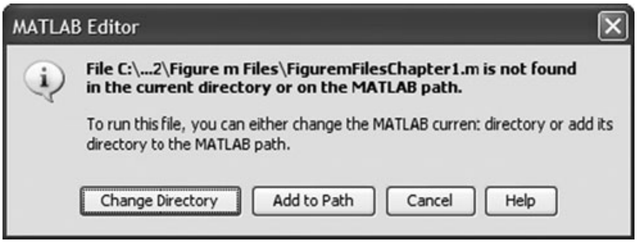


Select a new folder

```
⊟ R2009a
    ⊞ bin
    ⊞ extern
    ⊞ help
      ja
    ⊞ java
    ⊞ jhelp
      licenses
    ⊞ notebook
    ⊞ rtw
    ⊞ simulink
    ⊞ stateflow
```

Folder: R2009a

Figure 1.14 Pop-up window used to locate a directory.



MATLAB Editor

File C:\...2\Figure m Files\FiguremFilesChapter1.m is not found in the current directory or on the MATLAB path.

To run this file, you can either change the MATLAB curren: directory or add its directory to the MATLAB path.

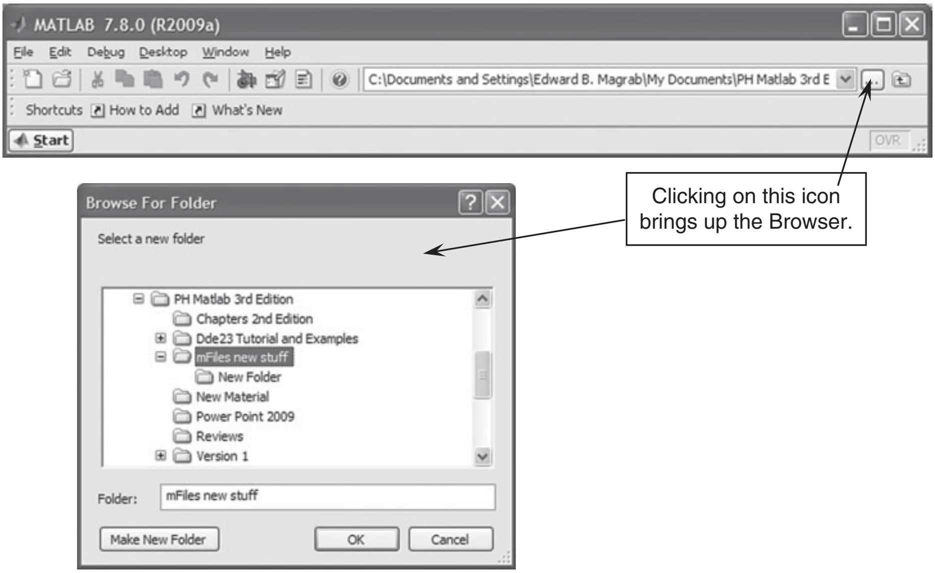
Figure 1.15 Pop-up window used to change current path to path of file to be executed.

where `input` is a MATLAB function and *Any message* is displayed in the MATLAB command window. After this expression is executed, the response typed, and *Enter* pressed, the value (or series of values) entered is assigned to *VariableName*. Other methods of data entry are given in Section 3.3, and further clarifications of the usage of input are given in Section 3.2.

There are several ways of getting program results to the command window. The first is simply to omit the semicolon (;) at the end of an expression. In this case, MATLAB displays in the command window the variable's name followed by an equal sign, and then skips to the next line and displays the value(s) of the variable. This method is useful during debugging. When output values are to be annotated for clarity, one uses either

```
disp
```

or



MATLAB 7.8.0 (R2009a)

C:\Documents and Settings\Edward B. Magrab\My Documents\PH Matlab 3rd E

Browse For Folder

Select a new folder

Folder: mFiles new stuff

Clicking on this icon brings up the Browser.

Figure 1.16 Accessing the *Browser* window to change current path (directory).

```
fprintf
```

which are discussed in Section 3.1.2.

It is good practice when creating scripts to start each script with the functions `clc` and `clear`. This clears all variables created previously and also clears the command window. In addition, when one has created graphics, which appear in separate windows as discussed in Section 6.1, one should close them using `close all`. Lastly, when global variables have been created, as discussed in Section 5.2.2, one should use `clear global`. The more inclusive function `clear all` also clears global variables. Thus, in general, each program should start with the following functions:

```
clear % or clear all
clc
clear global % not required if clear all is used
close all
```

We shall now summarize this material with an example, and we shall show what the command window and the workspace window look like after the program has been executed.

Example 1.2 Flow in a circular channel

The flow rate Q in m^3/s in an open channel of circular cross-section shown in Figure 1.17 is given by[4]

$$Q = \frac{2^{3/2} D_c^{5/2} \sqrt{g}\,(\theta - 0.5\sin(2\theta))^{3/2}}{8\sqrt{\sin\theta}\,(1 - \cos\theta)^{5/2}}$$

where $g = 9.8$ m/s^2 is the gravitational constant and D_c is given by

$$D_c = \frac{d}{2}(1 - \cos\theta)$$

If we assume that $d = 2$ m and $\theta = 60° = \pi/3$, then the MATLAB script is that shown in Figure 1.18a and repeated below for clarity.

```
g = 9.8; d = 2; th = pi/3; % Input
Dc = d/2*(1-cos(th));
Qnum = 2^(3/2)*Dc^(5/2)*sqrt(g)*(th-0.5*sin(2*th))^(3/2);
Qden = 8*sqrt(sin(th))*(1-cos(th))^(5/2);
Q = Qnum/Qden % m^3/s
```

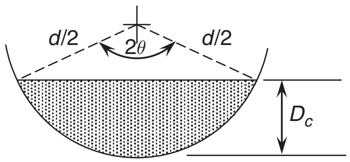
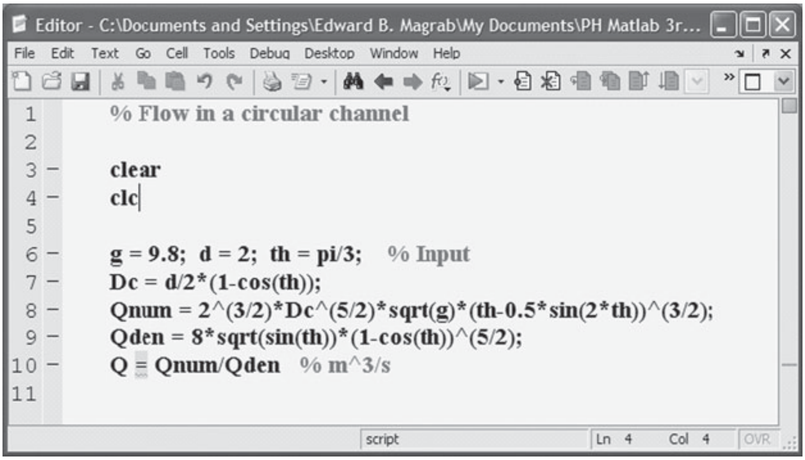


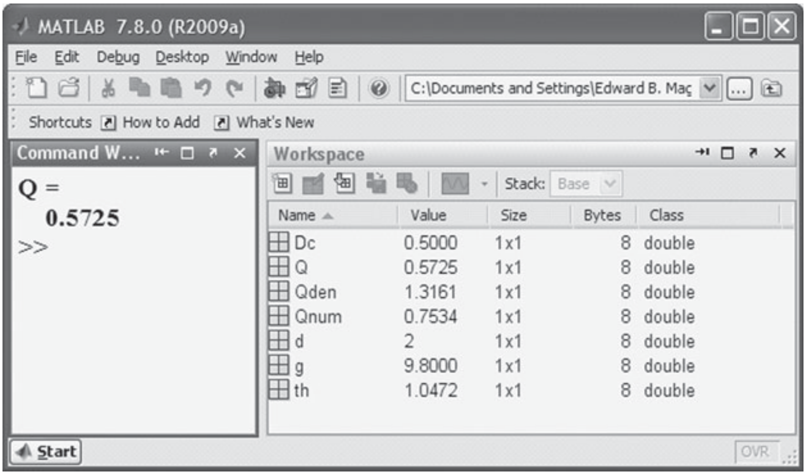
Figure 1.17 Circular channel.

[4] T. G. Hicks, *Mechanical Engineering Formulas: Pocket Guide*, McGraw Hill, NY, 2003, p. 254.


Editor - C:\Documents and Settings\Edward B. Magrab\My Documents\PH Matlab 3r...
File Edit Text Go Cell Tools Debug Desktop Window Help
script
Ln 4
Col 4
OVR

```
% Flow in a circular channel

clear
clc

g = 9.8;  d = 2;  th = pi/3;    % Input
Dc = d/2*(1-cos(th));
Qnum = 2^(3/2)*Dc^(5/2)*sqrt(g)*(th-0.5*sin(2*th))^(3/2);
Qden = 8*sqrt(sin(th))*(1-cos(th))^(5/2);
Q = Qnum/Qden   % m^3/s
```

(a)



```
Q =
   0.5725
>>
```

Name	Value	Size	Bytes	Class
Dc	0.5000	1x1	8	double
Q	0.5725	1x1	8	double
Qden	1.3161	1x1	8	double
Qnum	0.7534	1x1	8	double
d	2	1x1	8	double
g	9.8000	1x1	8	double
th	1.0472	1x1	8	double

(b)

Figure 1.18 (a) Editor window for the script for Example 1.1, and (b) workspace and command windows.

After clicking the *Save and Run* icon in the Editor, the answer, $Q = 0.5725$, is displayed in the command window. Upon execution of the script file, the *Workspace* window is populated as shown in Figure 1.18b. It displays a record of the seven variables that have been created: *Dc*, *Q*, *Qnum*, *Qden*, *g*, *d*, and *th*. Since all the commands have been issued from the Editor, the *Command History* window is empty and has not been displayed.

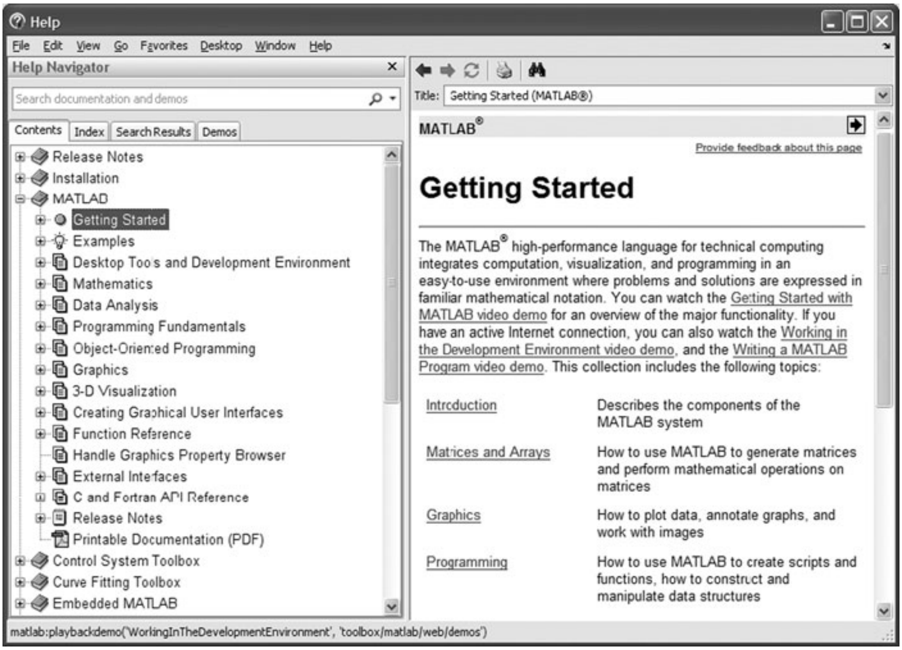
As a final comment, the form of the definitions of the various quantities appearing in the script was chosen to make the independent calculations used to verify the script both easier to perform and easier to compare with what the script gives. During the debugging stage, each quantity should be calculated independently and be compared to those computed by the script by temporarily omitting the semicolon at the end of each expression.

1.3 ONLINE HELP

MATLAB has a complete online help capability, which can be accessed in several ways. One way is to click on the question mark (?) icon on the toolbar of the command window. This opens the *Help* window shown in Figure 1.19. It should be minimized, rather than closed, after each use so that it is readily available. Going to the *Help* pull-down menu shown in Figure 1.20 also brings up the window shown in Figure 1.19, except that it opens at specific locations depending on what has been selected. If *MATLAB Help* is selected, then the same starting point as selecting the question mark (?) appears. If *Using the Desktop* is selected, then it opens the right-hand window to the section titled *Desktop*, as shown in Figure 1.21. If *Using the Command Window* is selected, then it opens the right-hand window to the section titled *Running Functions — Command Window and History* as shown in Figure 1.22.

If one wants specific information about a particular MATLAB function and the name of that function is known, then the *Index* tab in the left-hand window is clicked and the command is typed in the blank area as shown in Figure 1.23 and *Enter* pressed. If enter is not pressed, the function name is selected. In either case, the information in the right-hand window is displayed. Another way to obtain access to information about a specific function is to type in the MATLAB command window

`help` FunctionName


Help
File Edit View Go Favorites Desktop Window Help
Help Navigator
Search documentation and demos
Contents
Index
Search Results
Demos
Title:
Getting Started (MATLAB®)
MATLAB®
Provide feedback about this page

Release Notes
Installation
MATLAD
Getting Started
Examples
Desktop Tools and Development Environment
Mathematics
Data Analysis
Programming Fundamentals
Object-Oriented Programming
Graphics
3-D Visualization
Creating Graphical User Interfaces
Function Reference
Handle Graphics Property Browser
External Interfaces
C and Fortran API Reference
Release Notes
Printable Documentation (PDF)
Control System Toolbox
Curve Fitting Toolbox
Embedded MATLAB

Getting Started

The MATLAB® high-performance language for technical computing integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. You can watch the Getting Started with MATLAB video demo for an overview of the major functionality. If you have an active Internet connection, you can also watch the Working in the Development Environment video demo, and the Writing a MATLAB Program video demo. This collection includes the following topics:

Introduction	Describes the components of the MATLAB system
Matrices and Arrays	How to use MATLAB to generate matrices and perform mathematical operations on matrices
Graphics	How to plot data, annotate graphs, and work with images
Programming	How to use MATLAB to create scripts and functions, how to construct and manipulate data structures

matlab:playbackdemo('WorkingInTheDevelopmentEnvironment', 'toolbox/matlab/web/demos')

Figure 1.19 *Help* window.

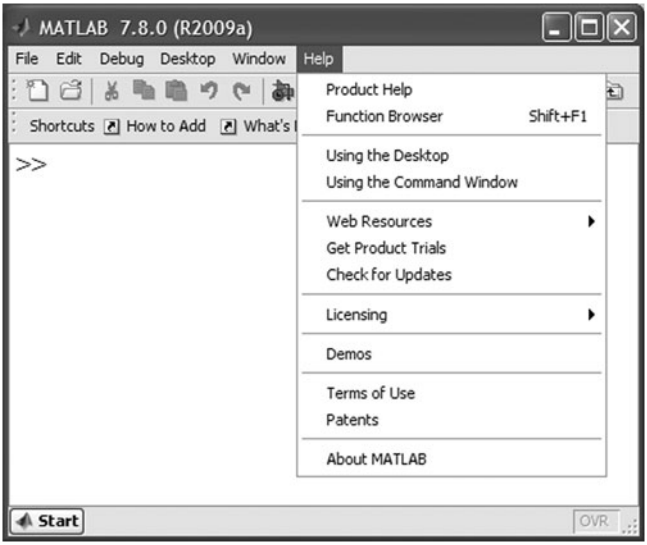
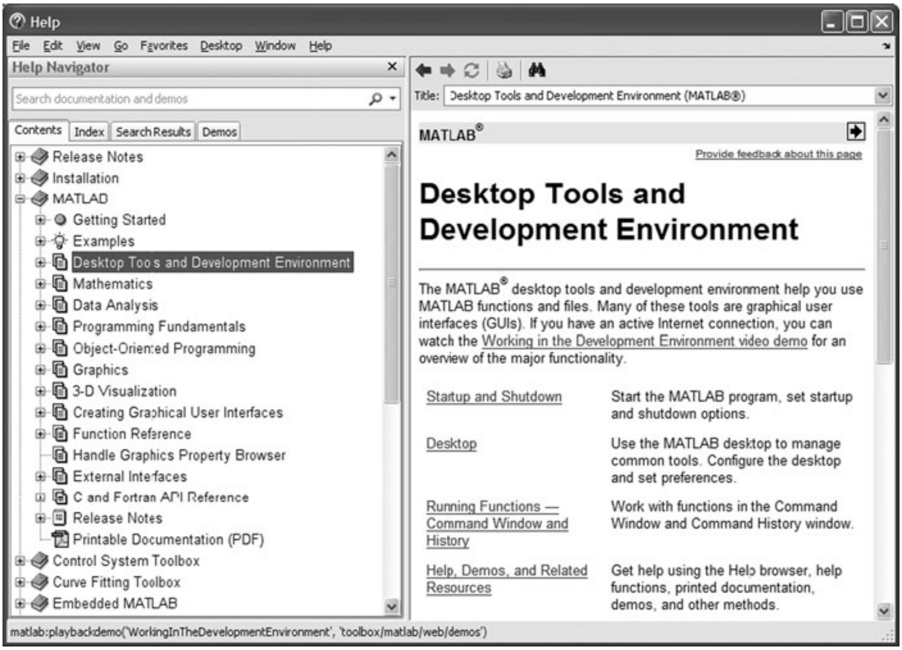


Figure 1.20 *Help* pull-down menu.

where *FunctionName* is the name of the function about which information is sought. Almost the same information that appears from using the *Index* search in the *Help* window is obtained, only in a less elegant format. Usually, the *Command* window


Help
File Edit View Go Favorites Desktop Window Help
Help Navigator
Search documentation and demos
Contents Index Search Results Demos
Release Notes
Installation
MATLAB
Getting Started
Examples
Desktop Tools and Development Environment
Mathematics
Data Analysis
Programming Fundamentals
Object-Oriented Programming
Graphics
3-D Visualization
Creating Graphical User Interfaces
Function Reference
Handle Graphics Property Browser
External Interfaces
C and Fortran API Reference
Release Notes
Printable Documentation (PDF)
Control System Toolbox
Curve Fitting Toolbox
Embedded MATLAB
Title: Desktop Tools and Development Environment (MATLAB®)
MATLAB®
Provide feedback about this page

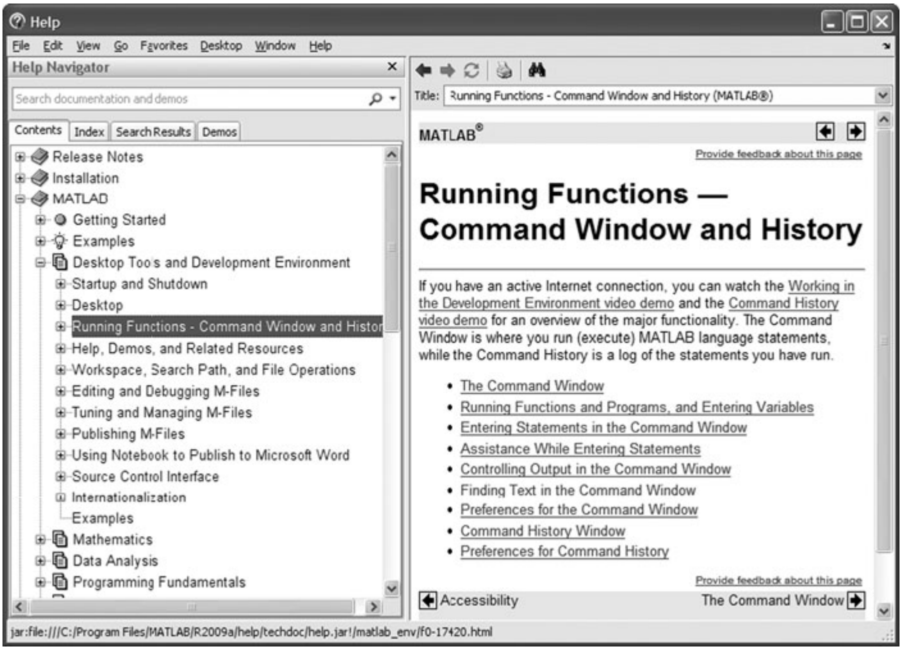
Desktop Tools and Development Environment

The MATLAB® desktop tools and development environment help you use MATLAB functions and files. Many of these tools are graphical user interfaces (GUIs). If you have an active Internet connection, you can watch the Working in the Development Environment video demo for an overview of the major functionality.

Startup and Shutdown	Start the MATLAB program, set startup and shutdown options.
Desktop	Use the MATLAB desktop to manage common tools. Configure the desktop and set preferences.
Running Functions — Command Window and History	Work with functions in the Command Window and Command History window.
Help, Demos, and Related Resources	Get help using the Help browser, help functions, printed documentation, demos, and other methods.

```
matlab:playbackdemo('WorkingInTheDevelopmentEnvironment', 'toolbox/matlab/web/demos')
```

Figure 1.21 Help window when *Using the Desktop* has been selected from the *Help* pull-down menu.


Help
File Edit View Go Favorites Desktop Window Help
Help Navigator
Search documentation and demos
Contents Index Search Results Demos
Release Notes
Installation
MATLAB
Getting Started
Examples
Desktop Tools and Development Environment
Startup and Shutdown
Desktop
Running Functions - Command Window and Histor
Help, Demos, and Related Resources
Workspace, Search Path, and File Operations
Editing and Debugging M-Files
Tuning and Managing M-Files
Publishing M-Files
Using Notebook to Publish to Microsoft Word
Source Control Interface
Internationalization
Examples
Mathematics
Data Analysis
Programming Fundamentals
Title: Running Functions - Command Window and History (MATLAB®)
MATLAB®
Provide feedback about this page

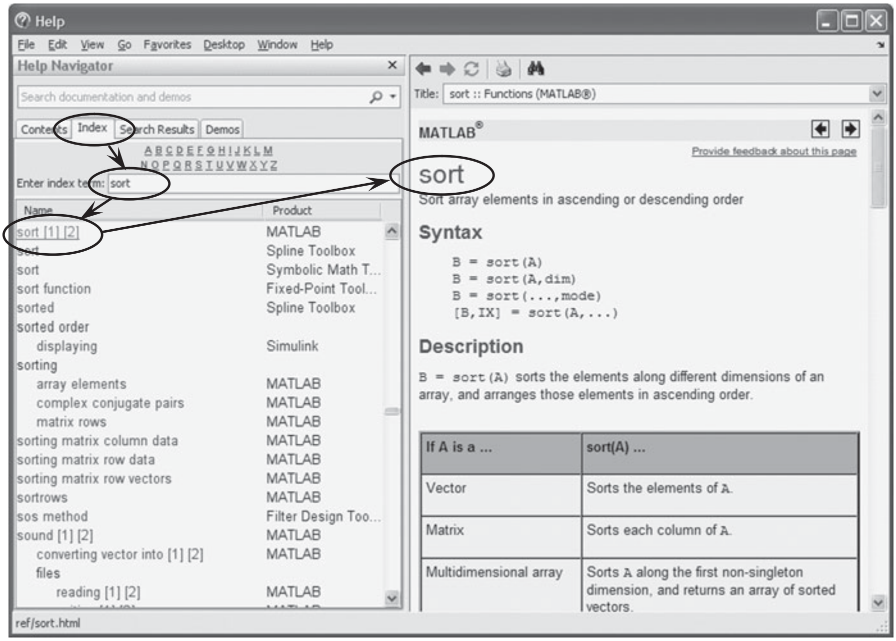
Running Functions — Command Window and History

If you have an active Internet connection, you can watch the Working in the Development Environment video demo and the Command History video demo for an overview of the major functionality. The Command Window is where you run (execute) MATLAB language statements, while the Command History is a log of the statements you have run.

- The Command Window
- Running Functions and Programs, and Entering Variables
- Entering Statements in the Command Window
- Assistance While Entering Statements
- Controlling Output in the Command Window
- Finding Text in the Command Window
- Preferences for the Command Window
- Command History Window
- Preferences for Command History

Provide feedback about this page
Accessibility
The Command Window
jar:file:///C:/Program Files/MATLAB/R2009a/help/techdoc/help.jar!/matlab_env/f0-17420.html

Figure 1.22 Help window when *Using the Command Window* has been selected from the *Help* pull-down menu.


Help
File Edit View Go Favorites Desktop Window Help
Help Navigator
Search documentation and demos
Contents Index Search Results Demos
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
Title: sort :: Functions (MATLAB®)
MATLAB®
Provide feedback about this page
ref/sort.html

Enter index term: sort

Name	Product
sort [1] [2]	MATLAB
sort	Spline Toolbox
sort	Symbolic Math T...
sort function	Fixed-Point Tool...
sorted	Spline Toolbox
sorted order	
displaying	Simulink
sorting	
array elements	MATLAB
complex conjugate pairs	MATLAB
matrix rows	MATLAB
sorting matrix column data	MATLAB
sorting matrix row data	MATLAB
sorting matrix row vectors	MATLAB
sortrows	MATLAB
sos method	Filter Design Too...
sound [1] [2]	MATLAB
converting vector into [1] [2]	MATLAB
files	
reading [1] [2]	MATLAB

sort

Sort array elements in ascending or descending order

Syntax

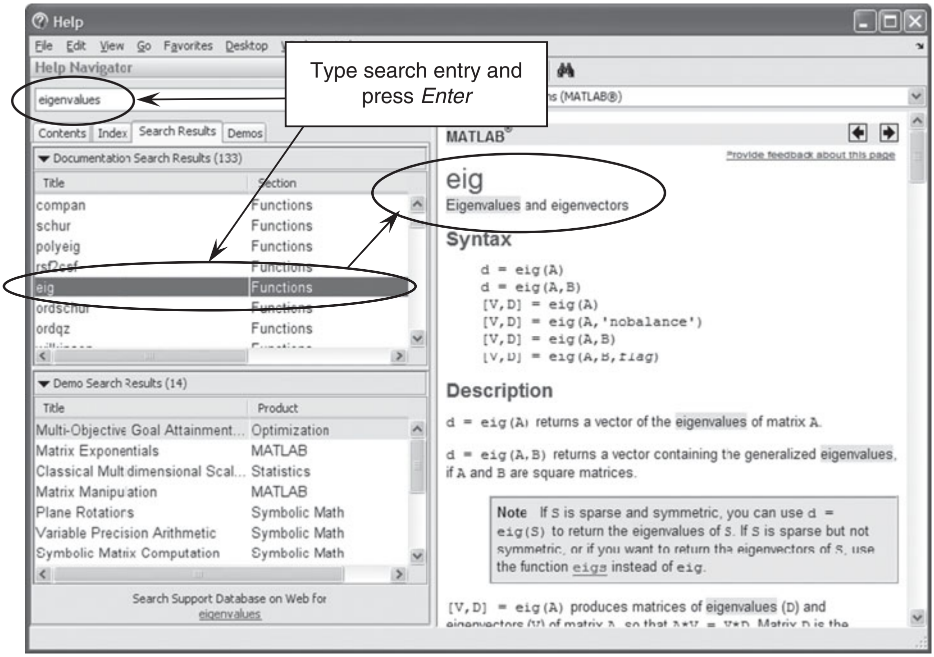
```
B = sort(A)
B = sort(A,dim)
B = sort(...,mode)
[B,IX] = sort(A,...)
```

Description

`B = sort(A)` sorts the elements along different dimensions of an array, and arranges those elements in ascending order.

If A is a ...	sort(A) ...
Vector	Sorts the elements of A.
Matrix	Sorts each column of A.
Multidimensional array	Sorts A along the first non-singleton dimension, and returns an array of sorted vectors.

Figure 1.23 Using *Index* in the Help window when the function name is known.


Help
File Edit View Go Favorites Desktop
Help Navigator
eigenvalues
Contents Index Search Results Demos
MATLAB®
Provide feedback about this page
Search Support Database on Web for eigenvalues

Type search entry and press *Enter*

Documentation Search Results (133)

Title	Section
compan	Functions
schur	Functions
polyeig	Functions
rsf2csf	Functions
eig	Functions
ordschur	Functions
ordqz	Functions

Demo Search Results (14)

Title	Product
Multi-Objective Goal Attainment...	Optimization
Matrix Exponentials	MATLAB
Classical Multidimensional Scal...	Statistics
Matrix Manipulation	MATLAB
Plane Rotations	Symbolic Math
Variable Precision Arithmetic	Symbolic Math
Symbolic Matrix Computation	Symbolic Math

eig

Eigenvalues and eigenvectors

Syntax

```
d = eig(A)
d = eig(A,B)
[V,D] = eig(A)
[V,D] = eig(A,'nobalance')
[V,D] = eig(A,B)
[V,D] = eig(A,B,flag)
```

Description

`d = eig(A)` returns a vector of the eigenvalues of matrix `A`.

`d = eig(A,B)` returns a vector containing the generalized eigenvalues, if `A` and `B` are square matrices.

Note If `S` is sparse and symmetric, you can use `d = eig(S)` to return the eigenvalues of `S`. If `S` is sparse but not symmetric, or if you want to return the eigenvectors of `S`, use the function `eigs` instead of `eig`.

`[V,D] = eig(A)` produces matrices of eigenvalues (`D`) and eigenvectors (`V`) of matrix `A`, so that `A*V = V*D`. Matrix `D` is the

Figure 1.24 Using *Search* when the function name is not known.

version of *Help* does not include equations and all the information that is available in the *Help* window.

When the name of a function is not known, one uses the *Search* portion of the *Help* window. As shown in Figure 1.24, one types in one or more descriptive words, clicks on the *Go* button, and clicks on the line that appears to contain the best match to one's search criteria. Not infrequently, one has to click on additional lines until the specific information that is being sought is found, as shown in Figure 1.24.

1.4 THE SYMBOLIC TOOLBOX

The Symbolic Math toolbox provides the capability of manipulating symbols to perform algebraic, matrix, and calculus operations symbolically. When one couples the results obtained from symbolic operations with the creation of functions, one has a means of numerically evaluating symbolically obtained expressions. This method is discussed in Section 5.6. In this section, we will introduce several of the basic operations that one can do with the Symbolic Math toolbox and then illustrate where this toolbox can be useful in engineering applications.

We will illustrate by example the Symbolic toolbox syntax, variable precision arithmetic, means of obtaining a Taylor series expansion, performing differentiation and integration, taking limits, substituting one expression for another, and obtaining the inverse Laplace transform. We will then summarize the use of many of these results with two examples. Additional uses of the Symbolic toolbox are given in Examples 2.19 and 2.24, and Examples 5.25–5.28.

Syntax

The shorthand way to create the symbolic variables a, b, and c is with

```
syms a b c
```

where a, b, and c are now symbolic variables. The spaces between the variable names are the delimiters. If the variables are restricted to being real variables, then we modify this statement as

```
syms a b c real
```

These symbols can be intermixed with nonsymbolic variable names, numbers, and MATLAB functions, with the result being a symbolic expression.

Consider the relation

$$f = 11.92e^{-a^2} + b/d$$

Assuming that $d = 4.2$, the script to represent this expression symbolically is

```
syms a b
d = 4.2;
f = 11.92*exp(-a^2)+b/d
```

which upon execution displays

```
f =
   (5*b)/21 + 298/(25*exp(a^2))
```

where f is a symbolic object. Notice that $21/5 = 4.2$ and $298/25 = 11.92$. Numbers in a symbolic expression are always converted to the ratio of two integers. If the decimal representation of numbers is desired, then one uses

```
vpa(f, n)
```

where f is the symbolic expression and n is the number of digits. Thus, to revert to the decimal notation with five decimal digits, the script becomes

```
syms a b
d = 4.2;
f = vpa(11.92*exp(-a^2)+b/d, 5)
```

The execution of this script gives the symbolic expression

```
f =
   0.2381*b+11.92/exp(1.0*a^2)
```

Variable Precision Arithmetic

The symbolic toolbox can also be used to calculate quantities with more than 15 digits of accuracy by using vpa in the following manner.

```
vpa('Expression', n)
```

where *Expression* is a valid MATLAB symbolic relation and *n* is the desired number of digits of precision.

To illustrate the use of this function, consider the evaluation of the following expression

$$y = 32! - e^{100}$$

The script to evaluate this relation with 50 digits of precision is

```
y = vpa('factorial(32)-exp(100)', 50)
```

The execution of this script gives

```
y =
   -26881171155030517550432725348582123713611118.773742
```

If variable precision arithmetic had not been used, the result would be y = –2.688117115503052 × 10^{43}.

Differentiation and Integration

Differentiation is performed with the function[5]

```
diff(f, x, n)
```

where $f = f(x)$ is a symbolic expression, x is the variable with which differentiation is performed, and n is the number of differentiations to be performed; for example, when $n = 2$ the second derivative is obtained.

We illustrate this function by taking the derivative of $b\cos(bt)$, first with respect to t and then with respect to b. The script is

```
syms b t
dt = diff(b*cos(b*t), t, 1)
db = diff(b*cos(b*t), b, 1)
```

Upon execution of this script, we obtain

```
dt =
   -b^2*sin(b*t)
db =
   cos(b*t)-b*t*sin(b*t)
```

[5] It is noted that diff is also a MATLAB function that is used in nonsymbolic computations as discussed in Example 5.5.

Integration is performed with the function

```
int(f, x, c, d)
```

where $f = f(x)$ is a symbolic expression, x is the variable of integration, c is the lower limit of integration, and d the upper limit. When c and d are omitted, the application of int results in the indefinite integral of $f(x)$.

Let us illustrate the use of int by integrating the results of the differentiation performed in the previous script. Thus,

```
syms b t
f = b*cos(b*t);
dt = diff(f, t, 1);
db = diff(f, b, 1);
it = int(dt, t)
ib = int(db, b)
```

The execution of the script results in

```
it =
   b*cos(b*t)
ib =
   b*cos(b*t)
```

Limits

One can take the limit of a symbolic expression as the independent variable approaches a specified value. The function that does this computation is

```
limit(f, x, z)
```

where $f = f(x)$ is the symbolic function whose limit is to be determined and x is the symbolic variable that is to assume the limiting value z.

To illustrate the use of this function, consider the expression

$$\operatorname*{Lim}_{a\to\infty}\left(\frac{2a+b}{3a-4}\right)$$

The script is

```
syms a b
Lim = limit((2*a+b)/(3*a-4), a, inf)
```

where inf stands for infinity (recall Table 1.6). The execution of this script gives

```
Lim =
   2/3
```

For another example, consider the limit

$$\operatorname*{Lim}_{x\to\infty}\left(1+\frac{y}{x}\right)^{x}$$

The script to determine this limit is

```
syms y x
Lim = limit((1+y/x)^x, x, inf)
```

Upon execution, we obtain

```
Lim =
  exp(y)
```

In other words, the limit is e^y.

Substitution

If one wants to substitute one expression b for another expression a, then the following function is used

```
subs(f, a, b)
```

where $f = f(a)$. The substitution function is frequently used to convert solutions to a more readable form. We shall illustrate its use in the examples that follow.

Taylor Series Expansion

An n-term Taylor series expansion of a function $f(x)$ about the point a is obtained with

```
taylor(f, n, a, x)
```

where the quantities in this function correspond to those in the expression

$$\sum_{k=0}^{n-1}(x - a)^k \frac{f^{(k)}(a)}{k!}$$

To illustrate this function, consider a four-term expansion of cosθ about θ_o. The script is

```
syms  x tho
Tay = taylor(cos(x), 4, tho, x)
```

Upon execution, we obtain

```
Tay =
  cos(tho) - (cos(tho)*(tho - x)^2)/2 - (sin(tho)*(tho - x)^3)/6 + sin(tho)*(tho - x)
```

Inverse Laplace Transform

If the Laplace transform of a function $f(t)$ is $F(s)$, where s is the Laplace transform parameter, then the inverse Laplace transform is obtained from

```
ilaplace(F, s, t)
```

To illustrate this function, consider the expression

$$F(s) = \frac{1}{s^2 + 2\zeta s + 1}$$

where $0 < \zeta < 1$. The inverse Laplace transform is obtained from the following script.

```
syms s t z
f = ilaplace(1/(s^2+2*z*s+1), s, t)
```

The execution of this script gives

```
f =
  sinh(t*(z^2 - 1)^(1/2))/(exp(t*z)*(z^2 - 1)^(1/2))
```

To simplify this expression, we make use of the following change of variables by noting that

$$\text{(z^2 - 1)^(1/2)} \rightarrow \sqrt{\zeta^2 - 1} = j\sqrt{1 - \zeta^2} = jr$$

Then, a simplified expression can be obtained by modifying the original script as follows:

```
syms s t z r
f = ilaplace(1/(s^2+2*z*s+1), s, t)
[Nu De] = numden(f);
fNu = subs(Nu,(z^2-1)^(1/2), i*r);
fDe = subs(De,(z^2-1)^(1/2), i*r);
pretty(simple(fNu/fDe))
```

Upon execution, we obtain

```
exp(-t z) sin(r t)
------------------
        r
```

where we used `numden` to isolate the numerator and denominator of the resulting inverse Laplace transform in order to perform the substitutions and we used `pretty` to place the output in a form that more closely represents standard algebraic notation.

We shall now illustrate the usage of these symbolic operations with two different examples. In each of these examples, we shall see that an effective way to use the symbolic operations is to use them interactively. This interaction with the toolbox's functions is equivalent to performing a series of manual algebraic and calculus operations with the objective of putting the final result into a compact irreducible form.

Example 1.3 Determination of curvature

The curvature κ of a plane curve with Cartesian parametric equations $x = x(t)$ and $y = y(t)$ is given by

$$\kappa = \frac{x'y'' - y'x''}{\left(x'^2 + y'^2\right)^{3/2}}$$

where the prime denotes the derivative with respect to t. Let us determine an expression for the curvature for a deltoid whose parametric equations are given by[6]

$$x = 2b\cos t + b\cos 2t$$
$$y = 2b\sin t - b\sin 2t$$

The script to obtain the curvature is

```
syms  t a b
x = 2*b*cos(t)+b*cos(2*t);
y = 2*b*sin(t)-b*sin(2*t);
xp = diff(x, t, 1);
xpp = diff(x, t, 2);
yp = diff(y, t, 1);
ypp = diff(y, t, 2);
n = xp*ypp-yp*xpp;
d = xp^2+yp^2;
n = factor(simple(n))
d = factor(simple(d))
```

The execution of this script gives

```
n =
  4*b^2*(cos(3*t) - 1)
d =
  (-8)*b^2*(cos(3*t) - 1)
```

The functions `simple` and `factor` are used to reduce n and d to their simplest forms. In addition, the numerator and denominator of k are treated separately until all simplifications are completed; then they are combined to form the final expression.

This result can be simplified further when the following trigonometric identity is used

$$1 - \cos a = 2\sin^2(a/2)$$

Then,

$$\texttt{cos}(3*t) - 1 \rightarrow \cos 3t - 1 = -2\sin^2(3t/2) = -Z$$

To make this change, we employ `subs` as follows:

```
syms x t y a b c
x = 2*b*cos(t) +b*cos(2*t);
y = 2*b*sin(t)-b*sin(2*t);
xp = diff(x, t, 1);
xpp = diff(x, t, 2);
yp = diff(y, t, 1);
ypp = diff(y, t, 2);
n = xp*ypp-yp*xpp;
d = xp^2+yp^2;
n = factor(simple(n));
```

[6] E. W. Weisstein, *CRC Concise Encyclopedia of Mathematics*, Chapman & Hall, Boca Raton, FL, 2003, pp. 697–698.

```
d = factor(simple(d));
n = collect(subs(n, 'cos(3*t) - 1', '-Z'))
d = collect(subs(d, 'cos(3*t) - 1', '-Z'))
```

The execution of this script gives

```
n =
  (-4)*Z*b^2
d =
  8*Z*b^2
```

From this point on, it is easier to complete the algebra manually. Thus,

$$\kappa = \frac{n}{d^{3/2}} = \frac{-4Zb^2}{\left(8Zb^2\right)^{3/2}} = \frac{-1}{4b\sqrt{2Z}} = \frac{-1}{4b\sqrt{4\sin^2(3t/2)}} = \frac{-1}{8b\sin(3t/2)}$$

Example 1.4 Maximum response amplitude of a single-degree-of-freedom system

The nondimensional response $y(t)$ of a single-degree-of-freedom system whose mass is subjected to a suddenly applied and maintained force of unit magnitude at $t = 0$ is given by[7]

$$y(t) = \frac{e^{-\xi t}}{r} \int_0^t e^{\xi\eta} \sin\left[r(t - \eta)\right] d\eta$$

where $r = \sqrt{1 - \zeta^2}$ and $0 < \xi < 1$. We shall first obtain a symbolic solution to this integral and then from that result determine the earliest time greater than zero when $y(t)$ is a maximum.

The script to evaluate this integral is

```
syms t z n r
arg = exp(z*n)*sin(r*(t-n));
yt = exp(-z*t)*int(arg, n, 0, t)/r
```

Upon execution, we have

```
yt =
  -(z*sin(r*t) + r*(cos(r*t) - exp(t*z)))/(r*exp(t*z)*(r^2 + z^2))
```

This result can be simplified further by noting that

$$\text{z\textasciicircum 2} + \text{r\textasciicircum 2} \rightarrow z^2 + r^2 = \zeta^2 + 1 - \zeta^2 = 1$$

and by using the following identity

$$a \sin x \pm b \cos x = \sqrt{a^2 + b^2} \sin(x \pm \varphi)$$

where

$$\varphi = \tan^{-1} b/a$$

Thus,

$$\text{r*cos(r*t)} + \text{z*sin(r*t)} \rightarrow \zeta \sin rt + \sqrt{1 - \zeta^2} \cos rt$$
$$\rightarrow \sqrt{\zeta^2 + 1 - \zeta^2} \sin(rt + \varphi) = \sin(rt + \varphi)$$

[7] B. Balachandran and E. B. Magrab, *Vibrations*, 2nd ed., Cengage Learning, Toronto, ON, 2009, p. 301.

where

$$\varphi = \tan^{-1}\frac{\sqrt{1-\zeta^2}}{\zeta}$$

Thus, the previous script is modified to reflect these relations by using subs as shown below.

```
syms t z n r p
arg = exp(z*n)*sin (r*(t-n));
yt = exp(-z*t)*int(arg, n, 0, t)/r;
yt = subs(yt, z^2+r^2, 1);
yt = simple(subs(yt, (z*sin(r*t) + r*(cos(r*t) - exp(t*z))), (sin(r*t+p)-
     r*exp(z*t))))
```

The execution of this script results in

```
yt =
  (r - sin(p + r*t)/exp(t*z))/r
```

Expressing this result in its traditional format, we have

$$y(t) = 1 - \frac{e^{-\zeta t}}{\sqrt{1-\zeta^2}}\sin\left(t\sqrt{1-\zeta^2}+\varphi\right)$$

We now determine the time at which $y(t)$ is the maximum by determining the earliest time greater than zero at which $dy/dt = 0$. This determination is accomplished by using diff as follows:

```
syms t z n r p
arg = exp(z*n)*sin(r*(t-n));
yt = exp(-z*t)*int(arg, n, 0, t)/r;
yt = subs(yt, z^2+r^2, 1);
yt = simple(subs(yt, (z*sin(r*t) + r*(cos(r*t) - exp(t*z))), (sin(r*t+p)-
     r*exp(z*t))))
dydt = simple(diff(yt, t, 1))
```

The execution of this script gives

```
dydt =
  -(r*cos(p + r*t) - z*sin(p + r*t))/(r*exp(t*z))
```

We can again simplify this expression by using the above identity. From the identity, we see that

$$\text{-z*sin(r*t + p) + r*cos(r*t + p)} \rightarrow -\left(\zeta\sin(rt+\varphi) - \sqrt{1-\zeta^2}\cos(rt+\varphi)\right)$$

$$\rightarrow -\sqrt{\zeta^2+1-\zeta^2}\sin(rt+\varphi-\varphi) = -\sin(rt)$$

Thus, the script is further modified as follows:

```
syms t z n r p
arg = exp(z*n)*sin(r*(t-n));
yt = exp(-z*t)*int(arg, n, 0, t)/r;
yt = subs(yt, z^2+r^2, 1);
yt = simple(subs(yt, (z*sin(r*t) + r*(cos(r*t) - exp(t*z))), (sin(r*t+p)-r*exp(z*t))))
```

```
dydt = simple(diff(yt, t, 1))
dydt = simple(subs(dydt, (r*cos(p + r*t) - z*sin(p + r*t)), -sin(r*t)))
```

The execution of this script gives

```
dydt =
  sin(r*t)/(r*exp(t*z))
```

Thus, $dy/dt = 0$ when $\sin(rt) = 0$ and the earliest time greater than zero that $\sin(rt) = 0$ is when $rt = \pi$, or $t = \pi/r = \pi/\sqrt{1 - \zeta^2}$.

1.5 SUMMARY OF FUNCTIONS INTRODUCED IN CHAPTER 1

Some elementary mathematical functions are given in Table 1.4, trigonometric and hyperbolic functions are given in Table 1.5, some special constants are given in Table 1.6, and the language's special characters are given in Tables 1.7 and 1.11. In Tables 1.8 and 1.9, decimal-to-integer conversion and manipulation of complex numbers, respectively, are summarized. In Table 1.10 several specialized mathematical functions and descriptive statistical functions have been listed. A summary of the Symbolic toolbox commands is given in Table 1.12 and the additional functions introduced in the chapter are presented in Table 1.13.

TABLE 1.12 Symbolic Math Toolbox Functions Introduced in Chapter 1

MATLAB function	Description
`collect`	Collects a variable or an expression within a symbolic function
`diff`	Differentiates a symbolic function
`factor`	Factors a symbolic function
`ilaplace`	Determines the inverse Laplace transform of a symbolic function
`int`	Determines the definite or indefinite integral of a symbolic function
`limit`	Takes the limit of a symbolic expression
`simple`	Simplifies a symbolic expression using `factor`, `collect`, and `simplify`
`simplify`	Attempts to simplify an expression using known identities and algebraic rules
`subs`	Substitutes one symbolic expression for another symbolic expression
`syms`	Shortcut means of constructing (defining) symbolic objects
`taylor`	Obtains symbolic expression for a Taylor series expansion of a function
`vpa`	Uses variable precision arithmetic to compute a value to a specified number of digits

TABLE 1.13 Additional MATLAB Functions Introduced in Chapter 1

MATLAB function	Description
`clc`	Clears the command window
`clear`	Removes variables from the workspace (computer memory)
`close all`	Closes (deletes) all graphic windows
`format`	Formats the display of numerical output to the command window

EXERCISES

1.1 Verify numerically the following relations.

$$\begin{aligned}
153 &= 1^3 + 5^3 + 3^3 \\
1634 &= 1^4 + 6^4 + 3^4 + 4^4 \\
548834 &= 5^6 + 4^6 + 8^6 + 8^6 + 3^6 + 4^6 \\
71^2 &= 1! + 7! \\
215^2 &= 1! + 4! + 5! + 6! + 7! + 8!
\end{aligned}$$

1.2 Show that the following relation gives the first 14 digits of π.

$$\pi \approx \left(100 - \frac{2125^3 + 214^3 + 30^3 + 37^2}{82^5}\right)^{1/4}$$

1.3 Show numerically that the following expressions are almost equal to an integer.

$$I_1 = \frac{53453}{\ln 53453}$$

$$I_2 = \frac{613}{37} e - \frac{35}{991}$$

1.4 Evaluate the following expression for R and display the value of x.

$$R = \frac{1}{2}\sqrt{\frac{8 \times 2^{2/3} - 16x + 2^{1/3}x^2}{8 \times 2^{2/3} - 10x + 2^{1/3}x^2}}$$

where

$$x = \sqrt[3]{49 - 27\sqrt{5} + 3\sqrt{6}\sqrt{93 - 49\sqrt{5}}}$$

1.5 Demonstrate numerically the validity of the following expressions.

$$\cot(\pi/5) = \frac{1}{5}\sqrt{25 + 10\sqrt{5}}$$

$$\sin(\pi/15) = \frac{1}{4}\sqrt{7 - \sqrt{5} - \sqrt{30 - 6\sqrt{5}}}$$

$$\pi = 16 \tan^{-1}\frac{1}{5} - 4 \tan^{-1}\frac{1}{239}$$

1.6 Demonstrate numerically the validity of the following expression

$$\sin(\pi/17) = \frac{\sqrt{2}}{8}\sqrt{\gamma^2 - \sqrt{2}(\alpha + \gamma)}$$

where

$$\begin{aligned}
\gamma &= \sqrt{17 - \sqrt{17}} \\
\beta &= \sqrt{17 + \sqrt{17}} \\
\alpha &= \sqrt{34 + 6\sqrt{17} + \sqrt{2}\left(\sqrt{17} - 1\right)\gamma - 8\beta\sqrt{2}}
\end{aligned}$$

1.7 For $a = 1$ and $b = 2$, verify numerically the following identities:

$$\sinh a + \cosh a = e^a$$

$$\tanh(a + b) - \tanh(a - b) = \frac{\sinh 2b}{\cosh(a + b)\cosh(a - b)}$$

1.8 Show that for the expression for V given below, $V = 3.2209$ when $L = 1.5$, $h = 1$, and $r = 1.6$.

$$V = L\left[r^2 \cos^{-1}\left(\frac{r - h}{r}\right) - (r - h)\sqrt{2rh - h^2}\right]$$

1.9 For the following expression

$$D = \alpha - \gamma + \sin^{-1}\left[n \sin\left(\gamma - \sin^{-1}\left\{\frac{\sin\alpha}{n}\right\}\right)\right]$$

show that $D = 0.4203$ when $\gamma = 60°$, $\alpha = 35°$, and $n = 4/3$.

1.10 The binomial coefficient is given by

$$_nC_k = \frac{n!}{k!(n - k)!}$$

Determine the value of $_{12}C_7$.

1.11 The moment of inertia of a sector of a circle is[8]

$$I = \left(\frac{\pi}{8} - \frac{8}{9\pi}\right)r^4$$

where r is the radius of the circle. Determine I when $r = 2.5$ cm.

1.12 The correction for curvature of a helical compression spring is[9]

$$K = \frac{4c - 1}{4c - 4} + \frac{0.615}{c}$$

where $c = D/d$, D is the diameter of the spring coil and d is the diameter of the wire forming the coil. Determine K when $c = 5$.

1.13 The shape factor for the deflection of a flat trapezoidal leaf spring is[10]

$$K = \frac{3}{(1 - B)^3}\left[0.5 - 2B + B(1.5 - \ln B)\right]$$

where $B < 1$ is the ratio of the ends of the trapezoid. Determine K when $B = 0.6$.

1.14 The length L of a belt that traverses two pulley wheels, one of radius R and one of radius r and whose centers are a distance S apart, is given by[11]

$$L = 2S\cos\theta + \pi(R + r) + 2\theta(R - r)$$

[8] Hicks, *Mechanical Engineering*, p. 8.

[9] *Ibid*, p. 78.

[10] *Ibid*, p. 95.

[11] *Ibid*, pp. 105–106.

where

$$\theta = \sin^{-1}\left(\frac{R - r}{S}\right) \text{ rad}$$

Determine L when $R = 30$ cm, $r = 12$ cm, and $S = 50$ cm.

1.15 The torque T on a block brake is given by[12]

$$T = \frac{4fF_n r \sin(\theta/2)}{\theta + \sin\theta}$$

where θ is the contact angle in radians, f is the coefficient of friction, r is the radius of the drum, and F_n is the normal force acting on the drum. Determine T when $F = 250\,N$, $f = 0.35$, $r = 0.4$ m, and $\theta = 60°$.

1.16 Air flow in a rectangular duct with sides of length A and B has an equivalent flow resistance to that of a circular duct of diameter D, which is given by the following equation[13]

$$D = 1.265\left[\frac{(AB)^3}{A + B}\right]^{1/5}$$

Determine D when $A = 1.7$ m and $B = 1.2$ m.

1.17 The maximum angular acceleration of a Geneva wheel containing n slots is[14]

$$a_G = \omega^2 \frac{M\left(1 - M^2\right)\sin\alpha}{\left(1 + M^2 - 2M\cos\alpha\right)^2}$$

where

$$\cos\alpha = \sqrt{\left(\frac{1 + M^2}{4M}\right)^2 + 2} - \left(\frac{1 + M^2}{4M}\right)$$

$$M = \frac{1}{\sin(\pi/n)}$$

Determine a_G/ω^2 when $n = 6$.

1.18 The pressure drop of air at standard condition flowing through a steel pipe is[15]

$$\Delta p = \frac{0.03L}{d^{1.24}}\left(\frac{V}{1000}\right)^{1.84}$$

where L is the length of the pipe in m, V is the velocity of air in m/min, and d is the diameter of the pipe in mm. Determine Δp when $L = 3000$ m, $d = 45$ mm, and $V = 1600$ m/min.

[12] *Ibid*, p. 109.

[13] *Ibid*, p. 165.

[14] *Ibid*, p. 125.

[15] *Ibid*, p. 223.

1.19 The following expressions[16] describe the principal contact stresses in the x-, y-, and z-directions, respectively, when two spheres are pressed together with a force F.

$$\sigma_x = \sigma_y = -p_{\max}\left[\left(1 - \frac{z}{a}\tan^{-1}\left(\frac{a}{z}\right)\right)(1 - \upsilon_1) - 0.5\left(1 + \frac{z^2}{a^2}\right)^{-1}\right]$$

$$\sigma_z = \frac{-p_{\max}}{1 + z^2/a^2}$$

where

$$a = \sqrt[3]{\frac{3F}{8}\frac{(1 - \nu_1^2)/E_1 + (1 - \nu_2^2)/E_2}{1/d_1 + 1/d_2}}$$

$$p_{\max} = \frac{3F}{2\pi a^2}$$

and ν_j, E_j, and d_j, $j = 1{,}2$, are the Poisson's ratio, Young's modulus, and diameter, respectively, of the two spheres. Determine the principal stresses when $\nu_1 = \nu_2 = 0.3$, $E_1 = E_2 = 206 \times 10^9$ N/m^2, $d_1 = 38$ mm, $d_2 = 70$ mm, $F = 450$ N, and $z = 0.25$ mm. [Answer: $\sigma_x = 2.078 \times 10^8$ N/m^2, and $\sigma_z = -1.242 \times 10^9$ N/m^2.]

1.20 The following expressions[17] describe the principal contact stresses in the x-, y-, and z-directions, respectively, when two cylinders, whose axes are parallel, are pressed together with a force F.

$$\sigma_x = -2\nu_2 p_{\max}\left(\sqrt{1 + \frac{z^2}{b^2}} - \frac{z}{b}\right)$$

$$\sigma_y = -p_{\max}\left(\left(2 - \left(1 + \frac{z^2}{b^2}\right)^{-1}\right)\sqrt{1 + \frac{z^2}{b^2}} - 2\frac{z}{b}\right)$$

$$\sigma_2 = \frac{-p_{\max}}{\sqrt{1 + z^2/b^2}}$$

$$\tau_{yz} = 0.5(\sigma_y - \sigma_z)$$

where

$$p_{\max} = \frac{2F}{\pi b L}$$

$$b = \sqrt{\frac{2F}{\pi L}\frac{(1 - \nu_1^2)/E_1 + (1 - \nu_2^2)/E_2}{1/d_1 + 1/d_2}}$$

and ν_j, E_j, and d_j for $j = 1, 2$, are the Poisson's ratio, Young's modulus, and diameter, respectively, of the two cylinders. Determine the principal stresses when $\nu_1 = \nu_2 = 0.3$, $E_1 = E_2 = 206 \times 10^9$ N/m^2, $d_1 = 38$ mm, $d_2 = 70$ mm, $F = 450$ N, $L = 50$ mm, and

[16] J. E. Shigley and C. R. Mischke, *Mechanical Engineering Design*, 5th ed., McGraw-Hill, New York, 1989.

[17] *Ibid*.

$z = 0.025$ mm. [Answer: $\sigma_x = -5.036 \times 10^7$ N/m^2, $\sigma_y = -3.543 \times 10^7$ N/m^2, and $\sigma_z = -1.324 \times 10^8$ N/m^2.]

1.21 The load number of a hydrodynamic bearing is given by[18]

$$N_L = \frac{\pi\varepsilon\sqrt{\pi^2\left(1 - \varepsilon^2\right) + 16\varepsilon^2}}{\left(1 - \varepsilon^2\right)^2}$$

where ε is the eccentricity ratio. Determine the value of N_L when $\varepsilon = 0.8$. [Answer: $N_L = 72.022$.]

1.22 Consider a threaded bolt of height h and whose material has a Young's modulus E. The stiffness k of the bolt when it is passed through a hole of diameter d_0 can be estimated from[19]

$$k = \frac{\pi E d_o \tan 30^\circ}{\ln\left(\dfrac{(d_2 - d_0)(d_1 + d_0)}{(d_2 + d_0)(d_1 - d_0)}\right)}$$

where d_1 is the diameter of the washer under the bolt, and

$$d_2 = d_1 + h \tan 30^\circ$$

Determine the value of k when $h = 30$ mm, $d_0 = 6$ mm, $d_1 = 16$ mm, and $E = 206 \times 10^9$ N/m^2. [Answer: $k = 5.283 \times 10^9$ N/m^2.]

1.23 The radial and tangential stresses in long tubes due to a temperature T_a at its inner surface of radius a and a temperature T_b at its outer surface of radius b are, respectively,[20]

$$\sigma_r = \frac{\alpha E(T_a - T_b)}{2(1 - \upsilon)\ln(b/a)}\left[\frac{a^2}{b^2 - a^2}\left(\frac{b^2}{r^2} - 1\right)\ln\left(\frac{b}{a}\right) - \ln\left(\frac{b}{r}\right)\right]$$

$$\sigma_t = \frac{\alpha E(T_a - T_b)}{2(1 - \upsilon)\ln(b/a)}\left[1 - \frac{a^2}{b^2 - a^2}\left(\frac{b^2}{r^2} + 1\right)\ln\left(\frac{b}{a}\right) - \ln\left(\frac{b}{r}\right)\right]$$

where r is the radial coordinate of the tube, E is the Young's modulus of the tube material, and α is the coefficient of thermal expansion. The temperature distribution through the wall of the tube in the radial direction is

$$T = T_b + \frac{(T_a - T_b)\ln(b/r)}{\ln(b/a)}$$

Determine the stresses and the temperature T when $\alpha = 2 \times 10^{-5}$ mm/mm/°C, $E = 206 \times 10^9$ N/m^2, $v = 0.3$, $T_a = 260$ °C, $T_b = 150$ °C, $a = 6$ mm, $b = 12$ mm, $r = 10$ mm. [Answer: $\sigma_r = -3.767 \times 10^7$ N/m^2, $\sigma_t = 1.185 \times 10^8$ N/m^2, and $T = 178.93$ °C.]

[18] R. L. Norton, *Machine Design, An Integrated Approach*, Prentice-Hall, Upper Saddle River, NJ, 1996.

[19] A. H. Burr and J. B. Cheatham, *Mechanical Analysis and Design*, 2nd ed., Prentice Hall, Upper Saddle River, NJ, 1995, p. 423.

[20] *Ibid.*, p. 496.

1.24 The mass flow rate of a gas escaping from a tank at pressure p_0 and under reversible adiabatic conditions is proportional to[21]

$$\psi = \sqrt{\frac{k}{k-1}}\sqrt{\left(\frac{p_e}{p_0}\right)^{2/k} - \left(\frac{p_e}{p_0}\right)^{(k+1)/k}}$$

where p_e is the pressure exterior to the tank's exit and k is the adiabatic reversible gas constant. Determine ψ when $k = 1.4$ and $p_e/p_0 = 0.3$. [Answer: $\psi = 0.4271$.]

1.25 The discharge factor for flow through an open channel of parabolic cross-section is[22]

$$K = \frac{1.2}{x}\left[\sqrt{16x^2 + 1} + \frac{1}{4x}\ln\left(\sqrt{16x^2 + 1} + 4x\right)\right]^{-2/3}$$

where x is the ratio of the maximum water depth to the breadth of the channel at the top of the water. Determine K when $x = 0.45$. [Answer: $K = 1.3394$.]

1.26 Show that with the following formula[23] one can approximate π to within less than 10^{-7} with one term ($n = 0$) and to within less than 10^{-15} with two terms ($n = 0$ and 1). In fact, for each term used, the approximation of π improves by almost a factor of 10^{-8}. Thus, after summing the first four terms ($n = 0, 1, 2, 3$) one would correctly obtain the first 31 digits of π; which can be verified using the Symbolic toolbox.

$$\frac{1}{\pi} = \frac{\sqrt{8}}{9801}\sum_{n=0}^{\infty}\frac{(4n)!(1103 + 26390n)}{(n!)^4 396^{4n}}$$

1.27 The thermal efficiency of a Diesel cycle on a cold air standard basis is expresses as[24]

$$\eta = 1 - \frac{1}{r^{k-1}}\left[\frac{r_c^k - 1}{k(r_c - 1)}\right]$$

where r is the compression ratio, r_c is the cutoff ratio, and $k = 1.4$ for air. Determine η for air when $r = 10$ and $r_c = 3$.

1.28 In a converging–diverging nozzle, the expression for the ratio of the area A of any section to the area A^* that would be required for sonic flow; that is, when the Mach number $M = 1$, is given by[25]

$$\frac{A}{A^*} = \frac{1}{M}\left[\left(\frac{2}{k+1}\right)\left(1 + \frac{k-1}{2}M^2\right)\right]^{(k+1)/[2(k-1)]}$$

where, for air, $k = 1.4$. Determine A/A^* for air when $M = 2$.

[21] W. Beitz and K. H. Kuttner, Eds., *Handbook of Mechanical Engineering*, Springer-Verlag, New York, 1994, p. C15.

[22] H. W. King, *Handbook of Hydraulics*, 4th ed., McGraw-Hill, NY, 1954, pp. 7–24.

[23] S. Ramanujan, "Modular equations and approximations to π," *Quarterly Journal of Mathematics*, **45**, 1914, pp. 350–372.

[24] M. J. Moran and H. N. Shapiro, *Fundamentals of Engineering Thermodynamics*, John Wiley and Sons, NY, 1992, p. 367.

[25] *Ibid*, p. 418.

1.29 The magnitude of the transfer function of a Chebyshev filter is given by[26]

$$|T_n(j\omega)| = \frac{1}{\sqrt{1 + \varepsilon^2 C_n^2(\omega)}}$$

where ω is a frequency ratio, $\varepsilon < 1$. is a positive integer, and

$$C_n(\omega) = \cos(n\cos^{-1}\omega) \quad |\omega|\leq 1$$
$$C_n(\omega) = \cosh(n\cosh^{-1}\omega) \quad |\omega|\geq 1$$

When $\varepsilon = 0.1$ and $n = 5$, determine the value of $|T_n(j\omega)|$ for $\omega = 0.5, 1.0$, and 1.5.

1.30 The natural frequency coefficient of a cylindrical shell clamped at one end and restrained by an attached stiff circular plate at the other end is given by[27]

$$\Omega_{nm} = \frac{\left(1 - \nu^2\right)\lambda_m^4}{\lambda_m^2 + n^2 + 1.78n^2\lambda_m^2} + \frac{1}{12}\left(\frac{h}{R}\right)^2\left(\lambda_m^4 + n^4 + 1.78m^2\lambda_m^2\right)$$

where n and m are positive integers and

$$\lambda_m = \left(\frac{\pi}{4}\right)\left(\frac{R}{l}\right)(4m + 1)$$

Show that when $n = 2$, $m = 1$, $h/R = 0.05$, $R/l = 0.1$, and $\nu = 0.3$, $\Omega_{21} = 7.51587 \times 10^{-3}$.

1.31 The coefficient of surface resistance (friction factor) in a pipe is given by[28]

$$f = \left\{\left(\frac{64}{\text{Re}}\right)^8 + 9.5\left[\ln\left(\frac{\varepsilon}{3.7D} + \frac{5.74}{\text{Re}^{0.9}}\right) - \left(\frac{2500}{\text{Re}}\right)^6\right]^{-16}\right\}^{1/8}$$

where ε is the average height of roughness of the pipe, D is the diameter of the pipe, and $\text{Re} = VD/\nu$ is the Reynolds number where V is the velocity of the fluid in the pipe and ν is the kinematic velocity of the fluid. Determine f when $D = 0.3$ m, $\nu = 1.012 \times 10^{-6}$ m^2/s, $\varepsilon = 0.00025$ m, and $V = 2.9$ m/s.

1.32 The acceleration of a reciprocating engine piston is given by[29]

$$a = r\omega^2\left[\cos\theta + \frac{n^2\cos 2\theta + \sin^4\theta}{\left(n^2 - \sin^2\theta\right)^{3/2}}\right]$$

where $n = l/r$, r is the crank arm length, l is the length of the connecting rod, ω is the angular velocity of the crank arm, and θ is the rotation angle of the crank arm. Determine the ratio $a/(r\omega^2)$ when $n = 3$ and $\theta = \pi/7$.

Symbolic Toolbox

1.33 Use the Symbolic toolbox to obtain the curvatures given for their corresponding parametric curves. Use the appropriate simplification and factoring commands and trigonometric identities to obtain the results indicated.

[26] R. Schaumann and M. E. Van Valkenburg, *Design of Analog Filters*, Oxford University Press, NY, 2001, p. 278.

[27] A. Leissa, "Vibration of Shells," NASA Report SP-288, 1973, p. 115.

[28] P. K. Swamee and A. K. Sharma, *Design of Water Supply Pipe Networks*, John Wiley & Sons, Hoboken, NJ, 2008, p. 14.

[29] V. Ramamurti, *Mechanics of Machines*, 2nd ed., Alpha Science International Ltd. Harrow, U.K., 2005, p. 33.

a. [This exercise has to be solved manually after the different expressions comprising the numerator and denominator of the curvature have been obtained and simplified. Note that $2\cos^2 x = 1 + \cos 2x$.]

$$x = a\sin t$$
$$y = \frac{a(2 + \cos t)\cos^2 t}{3 + \sin^2 t}$$
$$\kappa = \frac{6\sqrt{2}(\cos t - 2)^3(3\cos t - 2)\sec t}{a[73 - 80\cos t + 9\cos(2t)]^{3/2}}$$

b.

$$x = \frac{3at}{1 + t^3}$$
$$y = \frac{3at^2}{1 + t^3}$$
$$\kappa = \frac{2(1 + t^3)^4}{3(1 + 4t^2 - 4t^3 - 4t^5 + 4t^6 + t^8)^{3/2}}$$

c. [Note: $\cosh^2 x - \sinh^2 x = 1$]

$$x = a(t - \tanh t)$$
$$y = a\operatorname{sech} t$$
$$\kappa = \operatorname{csch} t$$

1.34 a. Use variable precision arithmetic to 40 digits to show that

$$\cos(\pi\cos(\pi\cos(\ln(\pi + 20)))) \approx -1 + 0.393216 \times 10^{-34}$$

b. Use variable precision arithmetic to 35 digits to show that $e^{\pi\sqrt{163}}$ is almost an integer.

1.35 Use the Symbolic toolbox to find the first five terms of the Taylor series expansions around zero; that is, $a = 0$, for the following functions. For some functions, the coefficients of the Taylor series can be expressed as a polynomial $g(n)$, which are given beside the respective functions.

$$f(x) = \frac{x(7x + 1)}{(1 - x)^3} \qquad [g(n) = 4n^2 - 3n]$$
$$f(x) = \frac{1}{4}\left(1 + x - \sqrt{1 - 6x + x^2}\right)$$
$$f(x) = \frac{2x}{(1 - x)^3} \qquad [g(n) = n(n + 1)]$$
$$f(x) = \frac{x\left(x^2 + 4x + 1\right)}{(1 - x)^3} \qquad [g(n) = 3n^2 - 3n + 1]$$

1.36 Use the Symbolic toolbox to find the following limits.

$$\lim_{\varepsilon\to 0}\frac{x^\varepsilon - 1}{\varepsilon} \qquad \lim_{x\to 0}(1 - \sin(2x))^{1/x}$$
$$\lim_{t\to 0}\left(\frac{e^t - 1}{t}\right)^{-a} \qquad \lim_{x\to 1}\frac{\ln x^n}{1 - x^2}$$

1.37 The response in the Laplace transform domain of one of the masses of a two-degree-of-freedom system subjected to an initial displacement is[30]

$$X_1(s) = \frac{0.1s^3 + 0.0282s^2 - 0.0427s + 0.0076}{s^4 + 0.282s^3 + 4.573s^2 + 0.4792s + 2.889}$$

Use the Symbolic toolbox to obtain its response in the time domain. Hint: To get the result in a simple form, use `vpa` with 5 digits.

1.38 The Fourier series coefficients for a periodic tone-burst of frequency f, duration NT, and period MT, where $T = 1/f$, are given by[31]

$$a_n = \frac{1}{\pi}\int_0^{2\pi\text{N/M}} \sin(M\tau)\cos(n\tau)d\tau \qquad n = 0,1,2,\ldots$$

$$b_n = \frac{1}{\pi}\int_0^{2\pi\text{N/M}} \sin(M\tau)\sin(n\tau)d\tau \qquad n = 1,2,3,\ldots$$

where $M > N$ and M and N are integers. Use the Symbolic toolbox to evaluate these integrals and simplify them by noting that $\cos(2\pi k \pm \alpha) = \cos\alpha$ and $\sin\alpha(2\pi k \pm \alpha) = \pm\sin\alpha$, where k is an integer. Using these solutions, show that

$$\begin{aligned} c_n &= \sqrt{a_n^2 + b_n^2} \\ &= \frac{2M}{\pi}\frac{\sin(\pi nN/M)}{M^2 - n^2} \qquad n \neq M \\ &= \frac{N}{M} \qquad n = M \end{aligned}$$

Note that each integral must be evaluated twice: once for $n \neq M$ and once for $n = M$. In addition, the following identity will be needed:

$$1 - \cos a = 2\sin^2(a/2)$$

[30] Balachandran and Magrab, *Vibrations*, p. 480.

[31] *Ibid*, p. 281.

2

Vectors and Matrices

Edward B. Magrab

MATLAB syntax is introduced in the context of vectors and matrices and their manipulation.

2.1 INTRODUCTION

MATLAB is a language whose operating instructions and syntax are based on a set of fundamental matrix operations and their extensions. Therefore, in order to fully utilize the advantages and compactness of the MATLAB language, we summarize some basic matrix definitions and symbolism and present many examples of their usage. The material presented in this section is used extensively in the programs developed throughout this and subsequent chapters.

2.2 DEFINITIONS OF MATRICES AND VECTORS

An array A of m rows and n columns is called a matrix of order $(m \times n)$, which consists of a total of mn elements arranged in the following rectangular array:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & & \\ \vdots & & \ddots & \\ a_{m1} & & & a_{mn} \end{bmatrix} \rightarrow (m \times n)$$

The elements of the matrix are denoted a_{ij}, where i indicates the row number and j the column number. In MATLAB, the order of the matrix is referred to as its size. Several special cases of this general matrix are as follows:

Square Matrix

When $m = n$, we have a square matrix.

Diagonal Matrix

When $m = n$ and $a_{ij} = 0, i \neq j$, we have the diagonal matrix

$$A = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & & \\ \vdots & & \ddots & \\ 0 & & & a_{nn} \end{bmatrix} \rightarrow (n \times n)$$

Identity Matrix

In a diagonal matrix, when $a_{ii} = 1$, we have the identity matrix I, that is,

$$I = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & & \\ \vdots & & \ddots & \\ 0 & & & 1 \end{bmatrix}$$

Vectors: Column and Row Matrices

When $a_{ij} = a_{i1}$, that is, there is only one column, then $a = A$ is called a column matrix or, more commonly, a vector, that is,

$$a = \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \rightarrow (m \times 1)$$

The quantity m is the length of the vector.

When $a_{ij} = a_{1j}$, that is, we have only one row, a is called a row matrix or a vector—that is,

$$a = [a_{11} \quad a_{12} \quad \ldots \quad a_{1n}] = [a_1 \quad a_2 \quad \ldots \quad a_n] \rightarrow (1 \times n)$$

This is the default definition of a vector in MATLAB. In this case, n is the length of the vector. Thus, a vector can be represented by a row or a column matrix.

Transpose of a Matrix and a Vector

The transpose of a matrix is denoted by an apostrophe ($'$), and is defined as follows. When A is the $(m \times n)$ matrix,

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & & \\ \vdots & & \ddots & \\ a_{m1} & & & a_{mn} \end{bmatrix} \rightarrow (m \times n)$$

then its transpose $W = A'$ is the following $(n \times m)$ matrix:

$$W = A' = \begin{bmatrix} w_{11} = a_{11} & w_{12} = a_{21} & \ldots & w_{1m} = a_{m1} \\ w_{21} = a_{12} & w_{22} = a_{22} & & \\ \vdots & & \ddots & \\ w_{n1} = a_{1n} & & & w_{nm} = a_{mn} \end{bmatrix} \rightarrow (n \times m)$$

that is, the rows and columns are interchanged.

For column and row vectors, we have the following: if

$$a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \rightarrow (m \times 1) \quad \text{then} \quad a' = [a_1 \quad a_2 \quad \ldots \quad a_m] \rightarrow (1 \times m)$$

and if

$$a = [a_1 \quad a_2 \quad \ldots \quad a_m] \rightarrow (1 \times m) \quad \text{then} \quad a' = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \rightarrow (m \times 1)$$

2.3 CREATION OF VECTORS

Vectors in MATLAB are expressed as either

$$f = [a\,x\,b \ldots]$$

or

$$f = [a, x, b, \ \ldots]$$

where $a, x, b, \ldots$ are either variable names, numbers, expressions, or strings (see Section 3.1). If they are variable names or expressions, then all variable names and the variable names composing the expressions must be defined such that a numerical value has been obtained for each of these variable names prior to the execution of this statement. Variable names, expressions, and numbers can appear in any combination and in any order. In the form

$$f = [a\ x\ b \ldots]$$

the space (blank) between symbols is required, whereas in the form

$$f = [a, x, b, \ \ldots]$$

the blank is optional.[1]

It is noted that if a is an expression that is explicitly written in the location where a is, then the expression for a can be written several ways. For example, if $a = h + d^s$, then some ways in which f can be written are

$$f = [h{+}d\hat{\ }s\ \ x\ \ b \ldots] \text{ or } f = [h{+}d\hat{\ }s, x, b, \ \ldots]$$

or

$$f = [(h{+}d\hat{\ }s), x, b, \ \ldots] \text{ or } f = [h{+}d\hat{\ }s, x, b, \ \ldots]$$

Colon Notation

MATLAB gives several additional ways to assign numerical values to the elements of a vector. The techniques for the creation of matrices are given in Section 2.4. The first means, described below, uses the *colon notation* to specify the range of the values and the increment between adjacent values. The second method specifies the range of the values and the number of values desired. In the former method, the increment is either important or has been specified, whereas in the latter method, the number of values is important.

The colon notation to create a vector is

```
x = s:d:f
```

or

```
x = (s:d:f)
```

or

```
x = [s:d:f]
```

where

s = start or initial value
d = increment or decrement
f = end or final value

[1] MATLAB occasionally has two or more equivalent ways which represent quantities. We will present, when appropriate, these equivalent representations. There is often no preferred form; however, readability can be used as a deciding factor.

Thus, the following row vector x is created:

$$x = [s, s + d, s + 2d, . . ., s + nd]$$

where $s + nd \le f$. Note that the number of values n created for x is not specified directly. The quantities s, d, and f can be any combination of numerical values, variable names, and expressions. The number of terms (i.e., the length of the vector) that this expression has created is determined from the MATLAB function

```
length(x)
```

When d is omitted, MATLAB assumes that $d = 1$. Then,

```
x = s:f
```

creates the vector

$$x = [s, s + 1, s + 2, . . ., s + n]$$

where $s + n \le f$. Again, s and f can be any combination of numerical values, variable names, and expressions.

We now illustrate these functions with the following examples. We first create a row vector that goes from 0.2 to 1.0 in increments of 0.1. The script is

```
x = 0.2:0.1:1
n = length(x)
```

which when executed gives

```
x =
  0.2000  0.3000  0.4000  0.5000  0.6000  0.7000  0.8000  0.9000  1.0000
n =
  9
```

On the other hand, if we want to create a vector that starts at 1 and decreases to 0.2 in increments of 0.1, the script is

```
x = 1:-0.1:0.2
```

which when executed gives

```
x =
  1.0000  0.9000  0.8000  0.7000  0.6000  0.5000  0.4000  0.3000  0.2000
```

We can create a column vector by taking the transpose of a row vector. Thus, the above script is modified to

```
x = (0.2:0.1:1)'
n = length(x)
```

where we have to use the form that employs the parentheses. If we did not use the parentheses (or, equivalently, the brackets) MATLAB would have only taken the transpose of the number 1, which is equal to 1. Upon execution of this script, we obtain

```
x =
   0.2000
   0.3000
   0.4000
   0.5000
   0.6000
   0.7000
   0.8000
   0.9000
   1.0000
n =
   9
```

If we now wish to create a row vector that goes from 0.2 to 1.0 in increments of 0.12, the script is

```
x = 0.2:0.12:1
n = length(x)
```

which when executed gives

```
x =
   0.2000   0.3200   0.4400   0.5600   0.6800   0.8000   0.9200
n =
   7
```

We notice that in this case, the highest value is 0.92, since $0.92 + 0.12 = 1.04 > 1$.

Now let us generate a row vector that goes from 1 to 7 in increments of 1. The script is

```
x = 1:7
n = length(x)
```

Upon execution, we obtain

```
x =
   1  2  3  4  5  6  7
n =
   7
```

since upon omitting the increment, MATLAB assumed an increment of +1.

However, when we create a row vector from 0.5 to 7 in increments of 1, the script becomes

```
x = 0.5:7
n = length(x)
```

which upon execution gives

```
x =
   0.5000   1.5000   2.5000   3.5000   4.5000   5.5000   6.5000
n =
   7
```

Generation of n Equally Spaced Values

In the second method, one specifies n equally spaced values starting at s and ending at f as follows:

```
x = linspace(s, f, n)
```

where the increment (decrement) is computed by MATLAB from

$$d = \frac{f - s}{n - 1}$$

The values of s and f can be either positive or negative and either $s > f$ or $s < f$. When n is not specified, it is assigned a value of 100. Thus, `linspace` creates the vector

$$x = [s, s + d, s + 2d, \ldots, f = s + (n - 1)d]$$

Notice that when `linspace` is used, the end points are always included in the vector of values that it creates; when using the colon notation, this is not necessarily true.

Thus, if we wish to create eight equally spaced values from –2 to 6.5, the script is

```
x = linspace(-2, 6.5, 8)
```

which upon execution gives

```
x =
  -2.0000  -0.7857  0.4286  1.6429  2.8571  4.0714  5.2857  6.5000
```

where MATLAB computes and uses the value $d = (6.5 - (-2))/(8 - 1) = 1.2143$.

To get an idea of which method one should use to create a vector of values, we shall create the same array using both methods. We first create the previous data using colon notation. For this case, the script is

```
s = -2;   f = 6.5;   n = 8;
d = (f-s)/(n-1);
x = s:d:f
```

which upon execution gives

```
x =
  -2.0000  -0.7857  0.4286  1.6429  2.8571  4.0714  5.2857  6.5000
```

On the other hand, if we want to create a vector that starts at 0.2, ends at 0.92, and increases by 0.12 using `linspace`, the script is

```
s = 0.2;   f = 0.92;   d = 0.12;
n = (f-s)/d+1;
x = linspace(s, f, n)
```

which upon execution gives

```
x =
  0.2000  0.3200  0.4400  0.5600  0.6800  0.8000  0.9200
```

It should be apparent from these two examples that the colon notation and `linspace` have different intended uses.

If equal spacing on a logarithmic scale is desired, then one uses

```
x = logspace(s, f, n)
```

where the initial value is 10^s, the final value is 10^f, and d is defined above. Thus, this expression creates the row vector

$$x = [10^s\ 10^{s+\mathrm{d}}\ 10^{s+2d} \ldots 10^f]$$

When n is not specified, MATLAB assigns it a value of 50. Thus, if we wish to create five equally spaced values on a logarithmic scale from 1 to 100, the script is

```
x = logspace(0, 2, 5)
```

which upon execution gives

```
x =
   1.0000   3.1623   10.0000   31.6228   100.0000
```

that is, $x = [10^0\ \ 10^{0.5}\ \ 10^1\ \ 10^{1.5}\ \ 10^2]$.

Transpose of a Vector with Complex Values

If a vector has elements that are complex numbers, then taking its transpose also converts the complex quantities to their complex conjugate values. To illustrate this, consider the following script

```
z = [1, 7+4j, -15.6, 3.5-0.12j];
w = z'
```

which upon execution gives

```
w =
   1.0000
   7.0000 - 4.0000i
   -15.6000
   3.5000 + 0.1200i
```

If the transpose is desired, but not the conjugate, then the script is written as

```
z = [1, 7+4j, -15.6, 3.5-0.12j];
w = conj(z')
```

which upon execution gives

```
w =
   1.0000
   7.0000 + 4.0000i
   -15.6000
   3.5000 - 0.1200i
```

Accessing Elements of Vectors

We now show how to access individual elements of vectors and perform arithmetic operations on them. Let

$$b = [b_1 \; b_2 \; \ldots \; b_n]$$

This means that we have created a vector b that has one row and n columns. In order to access individual elements of this vector, we use MATLAB's subscript notation. This notation has the form $b(j)$, that is, $b(j)$ corresponds to b_j, where $j = 1, 2, \ldots, n$ is the jth location in the vector b. Thus, if we write $b(3)$, then MATLAB will return the numerical value assigned to b_3, the third element of the vector. However, MATLAB's interpreter is smart enough to know that the matrix b is a $(1 \times n)$ matrix, and in some sense, it ignores the double subscript requirement. That is, writing $b(3)$, where b is a vector defined above, is the same as writing it as $b(1,3)$. However, if one were to either directly or implicitly require $b(3,1)$, then an error message would appear because this row (the third row) hasn't been defined (created).

Conversely, if we let

$$b = [b_1 \; b_2 \; \ldots \; b_n]'$$

we have created a column vector, that is, a $(n \times 1)$ matrix. If we want to locate the third element of this vector, then we again write $b(3)$ and MATLAB returns the numerical value corresponding to b_3. This is the same as having written $b(3,1)$. If one were to either directly or implicitly require $b(1,3)$, then an error message would appear because this column (the third column) hasn't been defined (created).

Accessing Elements of Vectors Using Subscript Colon Notation

The elements of a vector can also be accessed using subscript colon notation. In this case, the subscript colon notation is a shorthand method of accessing a group of elements. For example, for a vector b with n elements, one can access a group of them using the notation

```
b(k:d:m)
```

where $1 \le k < m \le n$, k and m are positive integers and, in this case, d is a positive integer. Thus, if one has ten elements in a vector b, the third through seventh elements are selected by using

```
b(3:7)
```

Subscript colon notation is used throughout the book.

Manipulation of Vector Elements

Suppose that we want to create a vector x that is to have the seven values $[-2, 1, 3, 5, 7, 9, 10]$. This can be created with either the expression

```
x= [-2, 1:2:9, 10]
```

or

```
x = [-2, 1, 3, 5, 7, 9, 10]
```

which means that the elements of this vector are $x_1 = -2, x_2 = 1, x_3 = 3, x_4 = 5, x_5 = 7, x_6 = 9$, and $x_7 = 10$ and its length is 7. We access the elements of x with the MATLAB expression $x(j)$, $j = 1, 2, \ldots, 7$. For example, the expression $x(5)$ returns the value 7. To access the last element in a vector, one can use the reserved word end as follows:

```
x = [-2, 1:2:9, 10];
xlast = x(end)
```

Upon execution, the following is displayed in the command window:

```
xlast =
   10
```

When we add or subtract a scalar from a vector, the scalar is added or subtracted from each element of the vector. Thus, the execution of

```
x = [-2, 1, 3, 5, 7, 9, 10];
z = x-1
```

results in

```
z =
   -3   0   2   4   6   8   9
```

However, the rules for multiplication, division, and exponentiation have restrictions, as discussed in Sections 2.5 and 2.6.2.

On the other hand, we may want to modify only some of the elements of a vector. For example, let $z = [-2\ 1\ 3\ 5\ 7\ 9\ 10]$. Then, to divide only the second element by 2, we have

```
z = [-2, 1, 3, 5, 7, 9, 10];
z(2) = z(2)/2;
z
```

which upon execution gives

```
z =
   -2.0000   0.5000   3.0000   5.0000   7.0000   9.0000   10.0000
```

If, further, we multiply the third and fourth elements by 3 and subtract 1 from each of them, the script is

```
z = [-2, 1, 3, 5, 7, 9, 10];
z(2) = z(2)/2;
z(3:4) = z(3:4)*3-1;
z
```

The execution of this script gives

```
z =
   -2.0000   0.5000   8.0000   14.0000   7.0000   9.0000   10.0000
```

Notice that in both examples the rest of the elements remain unaltered. The assignment statement

```
z(3:4) = z(3:4)*3-1
```

is interpreted by MATLAB as follows. The existing values of the elements *z*(3) and *z*(4) are each multiplied by 3, and then 1 is subtracted from their respective results. These new results are then used to replace the original values of *z*(3)and *z*(4). This syntax is very effective in writing compact code and is a frequently used programming construction, as we shall illustrate in Chapter 4.

One can access the elements of a vector in several ways in order to create new vectors. Consider the eight-element vector:

```
y = [-1, 6, 15, -7, 31, 2, -4, -5];
```

If one wanted to create a new vector *x* composed of the third through fifth elements of *y*, then the execution of the script

```
y = [-1, 6, 15, -7, 31, 2, -4, -5];
x = y(3:5)
```

creates the three-element vector

```
x =
  15   -7   31
```

Suppose, instead, we wanted to create a vector *x* composed of the first two and the last two elements of *y*. This can be done either by

```
y = [-1, 6, 15, -7, 31, 2, -4, -5];
x = [y(1), y(2), y(7), y(8)]
```

or by first defining the locations in the vector array as a variable called *index* and employing it as follows:

```
y = [-1, 6, 15, -7, 31, 2, -4, -5];
index = [1, 2, 7, 8];
x = y(index)
```

or, more compactly, as

```
y = [-1, 6, 15, -7, 31, 2, -4, -5];
x = y([1, 2, 7, 8])
```

The last two representations have many useful applications. Let us assume that corresponding to the vector *y* is a vector *z*, which is also a vector with eight elements. The vectors are assumed to have the values

```
y = [-1, 6, 15, -7, 31, 2, -4, -5];
z = [10, 20, 30, 40, 50, 60, 70, 80];
```

One can think of this set of vectors as correlated or linked such that *y*(*j*) corresponds to *z*(*j*). Suppose that we want to sort the vector *y* in ascending order (most negative to most positive) using the `sort` function and then rearrange the order of the elements of *z* to correspond to the new order of the elements of *y*. From the *Help* file, we find that one form of the `sort` function is[2]

```
[ynew, indx] = sort(y, mode)
```

where mode = 'ascend'; (apostrophes required) to sort in ascending order (most negative to most positive: default value) and mode = 'descend' to sort in descending order (most positive to most negative), *ynew* is the vector with the rearranged (sorted) elements of *y* and *indx* is a vector containing the *original* locations of the elements in *y*. Thus, the script

```
y = [-1, 6, 15, -7, 31, 2, -4, -5];
z = [10, 20, 30, 40, 50, 60, 70, 80];
[ynew, indx] = sort(y)
znew = z(indx)
```

when executed gives

```
ynew =
  -7   -5   -4   -1   2   6   15   31
indx =
  4   8   7   1   6   2   3   5
znew =
  40   80   70   10   60   20   30   50
```

Therefore, we see that *indx*(1) = 4 means that *ynew*(1) used to be *y*(4). Thus, to obtain the corresponding *z* we defined *znew* as the vector *z* whose indices (order) are now given by *indx*.

We can extend this capability further by introducing

```
find(Relation)
```

which determines the locations (not the values) of all the elements in a vector (or matrix) that satisfy a user-specified condition as represented by *Relation*. We illustrate its usage by creating a new vector *s* that contains only those elements of *y* that are either negative or zero. The MATLAB relational operator "<=" stands for "less than or equal to" (recall Table 1.7). Then,

```
y = [-1, 6, 15, -7, 31, 2, -4, -5];
indxx = find(y<=0)
s = y(indxx)
```

which when executed results in

```
indxx =
  1   4   7   8
s =
  -1   -7   -4   -5
```

[2] `sort` can also sort a cell array of strings in dictionary order; see the end of Section 3.4.

The script could be written compactly as

```
y = [-1, 6, 15, -7, 31, 2, -4, -5];
s = y(find(y<=0))
```

One of the great advantages of MATLAB's implicit vector and matrix notation is that it provides the user with a compact way of performing a series of operations on a vector of values. For example, suppose that we would like to determine sin(x) at ten equally spaced values from $-\pi \leq x \leq \pi$. Then, the MATLAB statements

```
x = linspace(-pi, pi, 10);
y = sin(x)
```

yield the following vector:

```
y =
   -0.0000  -0.6428  -0.9848  -0.8660  -0.3420  0.3420  0.8660  0.9848  0.6428  0.0000
```

Minimum and Maximum Values in a Vector

MATLAB provides a means to find the extreme values in a vector. See Section 2.4 on how to determine these values in a matrix. To find the magnitude of the smallest element *xmin* and its location *locmin* in a vector, we use

```
[xmin, locmin] = min(x)
```

and to find the magnitude of the largest element *xmax* and its location *locmax* in a vector, we use

```
[xmax, locmax] = max(x)
```

Thus, to determine the minimum and maximum values created by the previous script, we have

```
x = linspace(-pi, pi, 10);
y = sin(x);
[ymax, kmax] = max(y)
[ymin, kmin] = min(y)
```

Upon execution, we find that

```
ymax =
   0.9848
kmax =
   8
ymin =
   -0.9848
kmin =
   3
```

Thus, the maximum value is 0.9848 and it is the eighth element in the vector y. The minimum value is –0.9848 and it is the third element in the vector y.

Example 2.1 Analysis of the elements of a vector

Given the following function

$$f(t) = \sin(t) \qquad 0 \le t \le 2\pi$$

For fifty equally spaced values of this function in this region, we shall (a) determine the time at which the minimum positive value of $f(t)$ occurs and (b) the average value of its negative values. The script is

```
t = linspace(0, 2*pi, 50);
f = sin(t);
fAvgNeg = mean(f(find(f<0)))
MinValuef = min(f(find(f>0)));
tMinValuef = t(find(f==MinValuef))
```

The MATLAB relational operator "==" stands for "equal to" and the relational operator ">" stands for "greater than" (recall Table 1.7). Upon execution, we find that

```
fAvgNeg =
  -0.6237
tMinValuef =
  3.0775
```

2.4 CREATION OF MATRICES

Consider the following (4 × 3) matrix:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix} \rightarrow (4 \times 3)$$

This matrix can be created several ways. The basic syntax to create a matrix is

$$A = [a_{11}\ a_{12}\ \ a_{13};\ \ a_{21}\ \ a_{22}\ \ a_{23}; a_{31}\ \ a_{32}\ \ a_{33;}\ \ a_{41}\ \ a_{42}\ \ a_{43}]$$

where the semicolons are used to indicate the end of a row. Each row must have the *same* number of columns. If a more readable presentation is desired, then one can use the form

$$\begin{aligned} A = [&a_{11}\ a_{12}\ a_{13}; \ldots \\ &a_{21}\ \ a_{22}\ \ a_{23}; \ldots \\ &a_{31}\ \ a_{32}\ \ a_{33}; \ldots \\ &a_{41}\ \ a_{42}\ \ a_{43}] \end{aligned}$$

where the ellipses (. . .) are required to indicate that the expression continues on the next line. One can omit the ellipsis (. . .) and instead use the *Enter* key to indicate the end of a row. In this case, the expression will look like

$$\begin{aligned} A = [&a_{11}\ a_{12}\ a_{13} \\ &a_{21}\ \ a_{22}\ \ a_{23} \\ &a_{31}\ \ a_{32}\ \ a_{33} \\ &a_{41}\ \ a_{42}\ \ a_{43}] \end{aligned}$$

A fourth way is to create four separate row vectors, each with the same number of columns, and then combine these vectors to form the matrix. In this case, we have

$$\begin{aligned} v_1 &= [a_{11}\ a_{12}\ a_{13}]; \\ v_2 &= [a_{21}\ a_{22}\ a_{23}]; \\ v_3 &= [a_{31}\ a_{32}\ a_{33}]; \\ v_4 &= [a_{41}\ a_{42}\ a_{43}]; \\ A &= [v_1;\ v_2;\ v_3;\ v_4] \end{aligned}$$

where the semicolons in the first four lines are used to suppress display to the command window. The fourth form is infrequently used.

In all the forms above, the a_{ij} are numbers, variable names, expressions, or strings. If they are either variable names or expressions, then the variable names or the variable names comprising the expressions must have been assigned numerical values, either by the user or from previously executed expressions, prior to the execution of this statement. Expressions and numbers can appear in any combination. If they are strings, then the number of characters in each row must be the same. See Section 3.1.

Let us represent the following matrix in MATLAB:

$$A = \begin{bmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \\ 41 & 42 & 43 & 44 \end{bmatrix}$$

The script is either

```
A = [11, 12, 13, 14; 21, 22, 23, 24; 31, 32, 33, 34; 41, 42, 43, 44]
```

or

```
A = [11:14; 21:24; 31:34; 41:44]
```

or

```
A = [11, 12, 13, 14; . . .
     21, 22, 23, 24; . . .
     31, 32, 33, 34; . . .
     41, 42, 43, 44]
```

or

```
A = [11, 12, 13, 14 %<Enter>
      21, 22, 23, 24 %<Enter>
      31, 32, 33, 34 %<Enter>
      41, 42, 43, 44]
```

or

```
v1 = [11, 12, 13, 14];
v2 = [21, 22, 23, 24];
v3 = [31, 32, 33, 34];
v4 = [41, 42, 43, 44];
A = [v1; v2; v3; v4]
```

Execution of any of these expressions displays in the command window

```
A =
   11   12   13   14
   21   22   23   24
   31   32   33   34
   41   42   43   44
```

The order of the matrix is determined by

```
[m, n] = size(A)
```

where *m* is the number of rows and *n* is the number of columns. Thus, to confirm that the order of *A* is a (4 × 4) matrix, the script is

```
A = [11, 12, 13, 14; 21, 22, 23, 24; 31, 32, 33, 34; 41, 42, 43, 44];
[m, n] = size(A)
```

Which upon execution gives,

```
m =
   4
n =
   4
```

If one were to use the length function for a matrix quantity, length would return the number of *columns* in the array. For example, if *A* is a (2 × 4) array, then the script

```
A = [1 2 3 4; 5 6 7 8];
L = length(A)
```

when executed gives

```
L =
   4
```

The transpose of a matrix is obtained by using the apostrophe ('). Thus, the transpose of *A* is

```
A = [11, 12, 13, 14; 21, 22, 23, 24; 31, 32, 33, 34; 41, 42, 43, 44]'
```

which upon execution gives

```
A =
   11  21  31  41
   12  22  32  42
   13  23  33  43
   14  24  34  44
```

Transpose of a Matrix with Complex Elements

If a matrix has elements that are complex numbers, then taking its transpose also converts the complex quantities to their complex conjugate values. To illustrate this, consider the following script:

```
Z = [1+2j, 3+4j; 5+6j, 7+9j]
W = Z'
```

the execution of which gives

```
Z =
   1.0000 + 2.0000i   3.0000 + 4.0000i
   5.0000 + 6.0000i   7.0000 + 9.0000i
W =
   1.0000 - 2.0000i   5.0000 - 6.0000i
   3.0000 - 4.0000i   7.0000 - 9.0000i
```

Generation of Special Matrices

We shall introduce four functions that can be used to create matrices with specific values for its elements.

Matrix with All Elements Equal to 1

An ($r \times c$) matrix in which each element has the value 1 is created with

```
ones(r, c)
```

The function `ones` is a convenient replacement for the equivalent expression

```
ones(r, c) → one(1:r,1:c) = 1
```

To create a (2 × 5) matrix with all its elements equal to one, the script is

```
on = ones(2, 5)
```

which upon execution gives

```
on =
   1  1  1  1  1
   1  1  1  1  1
```

Null Matrix

An ($r \times c$) matrix in which each element has the value 0, which is called a null matrix, is created with

```
zeros(r, c)
```

The function `zeros` is a convenient replacement for the equivalent expression

```
zeros(r, c) → zero(1:r,1:c) = 0
```

To create a (3 × 2) matrix with all of its elements equal to zero, the script is

```
zer = zeros(3, 2)
```

which upon execution gives

```
zer =
  0   0
  0   0
  0   0
```

Diagonal Matrix

To create an ($n \times n$) diagonal matrix whose diagonal elements are given by a vector a of length n, we use

```
diag(a)
```

This function can also be used to extract the diagonal elements of an ($n \times n$) matrix A. Thus,

```
b = diag(A)
```

where b is a column vector of length n containing the diagonal elements of the matrix A.

To create a (3 × 3) diagonal matrix A with elements $a_{11} = 4$, $a_{22} = 9$, and $a_{33} = 1$, the script is

```
a = [4, 9, 1];
A = diag(a)
```

or, more compactly,

```
A = diag([4, 9, 1])
```

Either script when executed gives

```
A =
  4   0   0
  0   9   0
  0   0   1
```

On the other hand, if we are given an $(n \times n)$ matrix, then `diag` can be used to extract its diagonal elements. If the matrix is the (4×4) matrix A defined previously, then the script to extract its diagonal elements is

```
A = diag([11, 12, 13, 14; 21, 22, 23, 24; 31, 32, 33, 34; 41, 42, 43, 44])
```

Upon execution, we obtain

```
A =
  11
  22
  33
  44
```

Furthermore, we can create a diagonal matrix composed of the diagonal elements of A as follows:

```
A = [11, 12, 13, 14; 21, 22, 23, 24; 31, 32, 33, 34; 41, 42, 43, 44];
Adiag = diag(diag(A))
```

Upon execution, we obtain

```
Adiag =
  11    0    0    0
  0    22    0    0
  0     0   33    0
  0     0    0   44
```

Identity Matrix

To create an $(n \times n)$ identity matrix I, we use

```
eye(n)
```

To create a (3×3) identity matrix, the script is

```
d = eye(3)
```

Upon execution, we obtain

```
d =
  1  0  0
  0  1  0
  0  0  1
```

The identity matrix can be used to emulate the kronecker delta δ_{mn}, which equals 1 when $m = n$ and equals 0 when $m \neq n$, in other words, $\delta_{mn} \rightarrow d(m,n)$.

Manipulation of Matrix Elements

Consider the construction of the (3×5) matrix:

$$A = \begin{bmatrix} 3 & 5 & 7 & 9 & 11 \\ 20.0 & 20.25 & 20.5 & 20.75 & 21.0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \rightarrow (3 \times 5)$$

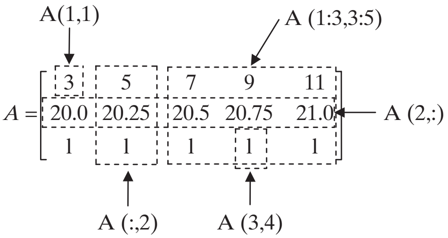


Figure 2.1 Accessing elements of a matrix.

This matrix is created with the statement

A = [3:2:11; `linspace`(20,21,5); `ones`(1,5)]

which yields

```
A =
    3.0000    5.0000    7.0000    9.0000   11.0000
   20.0000   20.2500   20.5000   20.7500   21.0000
    1.0000    1.0000    1.0000    1.0000    1.0000
```

Referring to Figure 2.1, one accesses the elements $a_{ij}, i = 1, 2, 3$ and $j = 1, 2, \ldots, 5$ of this matrix as follows. The element in the first row and first column a_{11} is

A(1,1) → 3

and the element in the third row and fourth column a_{34} is

A(3,4) → 1

All the elements in the second column, a_{12}, a_{22}, and a_{32}, are accessed by

A(:,2) → $[5, 20.25, 1]'$

where we have used the transpose symbol to indicate that it is a column vector. The notation

A(:,2)

means "all the elements of column 2." All the elements of row 2: a_{21}, a_{22}, a_{23}, a_{24}, and a_{25}, can be accessed by

A(2,:) → $[20, 20.25, 20.5, 20.75, 21]$

where the notation

A(2,:)

means "all the elements of row 2."

To access the submatrix composed of the elements in columns 3 to 5 and rows 1 to 3, we use the colon notation as follows:

A(1:3,3:5) → $[7, 9, 11; 20.5, 20.75, 21; 1, 1, 1]$

which is a (3 × 3) matrix. We see that in writing the indices of A, we used the default form that sets the increment to +1. Thus, if we construct the script

```
A = [3:2:11; linspace(20, 1, 5); ones(1, 5)];
B = A(1:3,3:5)
```

then, its execution gives the (3 × 3) matrix

```
B =
    7.0000    9.0000   11.0000
   20.5000   20.7500   21.0000
    1.0000    1.0000    1.0000
```

Let us now create a matrix that is of the same size as A, but with all of its elements equal to 4. This is done with the script

```
A = [3:2:11; linspace(20, 21, 5); ones(1, 5)];
[r, c] = size(A);
Z = 4*ones(r, c)
```

which creates

```
Z =
   4   4   4   4   4
   4   4   4   4   4
   4   4   4   4   4
```

This can be written more compactly as

```
A = [3:2:11; linspace(20, 21, 5); ones(1, 5)];
Z = 4*ones(size(A))
```

One can alter the elements of a matrix in a manner similar to that used for vectors. Let us use the `magic` function[3] to create a matrix. The `magic` function creates a matrix in which the sum of the elements in each column, the sum of the elements of each row, and the sum of the elements in each diagonal are equal. For a (4 × 4) matrix, this sum is 34. Then, executing

```
Z = magic(4)
```

we obtain

```
Z =
   16    2    3   13
    5   11   10    8
    9    7    6   12
    4   14   15    1
```

[3] There are more than fifty special matrices available: see `gallery` in *Help*.

Let us divide all the elements in row 2 of this matrix by 2 and, additionally, add all the elements in column 2 to those in column 4 and place the result in column 4. The script is

```
Z = magic(4);
Z(2,:) = Z(2,:)/2;
Z(:,4) = Z(:,4)+Z(:,2);
Z
```

which results in

```
Z =
  16.0000    2.0000    3.0000   15.0000
   2.5000    5.5000    5.0000    9.5000
   9.0000    7.0000    6.0000   19.0000
   4.0000   14.0000   15.0000   15.0000
```

To set all the diagonal elements of the original matrix *Z* to zero, we use the previously discussed technique regarding `diag` to obtain

```
Z = magic(4);
Z = Z-diag(diag(Z))
```

which results in

```
z =
  0   2   3  13
  5   0  10   8
  9   7   0  12
  4  14  15   0
```

To replace all the diagonal elements with the value 5, we use the script

```
Z = magic(4);
Z = Z-diag(diag(Z))+5*eye(4)
```

which results in

```
Z =
  5   2   3  13
  5   5  10   8
  9   7   5  12
  4  14  15   5
```

To place the values 11, 23, 54, and 61 in the diagonal elements of *Z*, we use the script

```
Z = magic(4);
Z = Z-diag(diag(Z))+diag([11, 23, 54, 61])
```

which results in

```
Z =
  11   2   3  13
   5  23  10   8
   9   7  54  12
   4  14  15  61
```

Minimum and Maximum Values in a Matrix

The minimum and maximum values of a matrix are also determined using `min` and `max`, except that for a matrix, these functions determine the minimum/maximum on a column-by-column basis. Thus, if the order of the matrix is $(m \times n)$, the output of `min` and `max` are vectors of length n, where each element of the vector is the minimum/maximum value of each column. For example, let us determine the minimum and maximum values of the `magic`(4). The script is

```
M = magic(4)
minM = min(M)
maxM = max(M)
```

The execution of the script gives

```
M =
  16   2   3  13
   5  11  10   8
   9   7   6  12
   4  14  15   1
minM =
  4   2   3   1
maxM =
  16   14   15   13
```

In order to find the maximum value of all the elements, we have to use `max` twice. Thus,

```
M = magic(4);
maxM = max(max(M))
```

which upon execution gives

```
maxM =
  16
```

Use of `find` with Matrices

The function `find` is used on a matrix of values in the same manner as with vectors except that the output is interpreted differently. For a matrix, we use

```
[row, col] = find(Relation)
```

where *row* and *col* are column vectors of the locations of the elements in the matrix that satisfied the condition(s) represented by *Relation*.

To illustrate the use of `find` on a matrix, let us find all the elements of `magic`(3) that are greater than 5. The script is

```
m = magic(3)
[r, c] = find(m > 5);
subscr = [r c] % See subsequent discussion on column augmentation
```

Upon execution we obtain

```
m =
   8   1   6
   3   5   7
   4   9   2
subscr =
   1   1
   3   2
   1   3
   2   3
```

Thus, the locations in the matrix *m* whose elements are greater than 5 are (1,1), (3,2), (1,3), and (2,3).

Example 2.2 Creation of a special matrix

We shall create the following (9 × 9) array, where the dashed lines have been added to enhance visual clarity:

$$A = \left[\begin{array}{ccc|ccc|ccc} 0 & 1 & 1 & 0 & 2 & 2 & 0 & 3 & 3 \\ 1 & 0 & 1 & 2 & 0 & 2 & 3 & 0 & 3 \\ 1 & 1 & 0 & 2 & 2 & 0 & 3 & 3 & 0 \\ \hline 0 & 4 & 4 & 0 & 5 & 5 & 0 & 6 & 6 \\ 4 & 0 & 4 & 5 & 0 & 5 & 6 & 0 & 6 \\ 4 & 4 & 0 & 5 & 5 & 0 & 6 & 6 & 0 \\ \hline 0 & 7 & 7 & 0 & 8 & 8 & 0 & 9 & 9 \\ 7 & 0 & 7 & 8 & 0 & 8 & 9 & 0 & 9 \\ 7 & 7 & 0 & 8 & 8 & 0 & 9 & 9 & 0 \end{array}\right]$$

The script is

```
a = ones(3, 3)-eye(3);
A = [a, 2*a, 3*a; 4*a, 5*a, 6*a; 7*a, 8*a, 9*a;]
```

Upon execution, we obtain

```
A =
   0   1   1   0   2   2   0   3   3
   1   0   1   2   0   2   3   0   3
   1   1   0   2   2   0   3   3   0
   0   4   4   0   5   5   0   6   6
```

```
4  0  4  5  0  5  6  0  6
4  4  0  5  5  0  6  6  0
0  7  7  0  8  8  0  9  9
7  0  7  8  0  8  9  0  9
7  7  0  8  8  0  9  9  0
```

Example 2.3 Rearrangement of submatrices of a matrix

Consider the following [9 × 9] array:

$$A = \begin{array}{c}
\begin{array}{ccccccccc} & \textcircled{1} & & & & & & \textcircled{2} & \end{array}\\
\left[\begin{array}{ccc|ccc|ccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9\\
10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18\\
19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27\\
\hline
28 & 29 & 30 & 31 & 32 & 33 & 34 & 35 & 36\\
37 & 38 & 39 & 40 & 41 & 42 & 43 & 44 & 45\\
46 & 47 & 48 & 49 & 50 & 51 & 52 & 53 & 54\\
\hline
55 & 56 & 57 & 58 & 59 & 60 & 61 & 62 & 63\\
64 & 65 & 66 & 67 & 68 & 69 & 70 & 71 & 72\\
73 & 74 & 75 & 76 & 77 & 78 & 79 & 80 & 81
\end{array}\right]\\
\begin{array}{ccccccccc} & \textcircled{4} & & & & & & \textcircled{3} & \end{array}
\end{array}$$

For the four (3 × 3) submatrices identified by the circled numbers 1 to 4, we shall perform a series of swaps of these submatrices to produce the following array:

$$A = \begin{array}{c}
\begin{array}{ccccccccc} & \textcircled{3} & & & & & & \textcircled{4} & \end{array}\\
\left[\begin{array}{ccc|ccc|ccc}
61 & 62 & 63 & 4 & 5 & 6 & 55 & 56 & 57\\
70 & 71 & 72 & 13 & 14 & 15 & 64 & 65 & 66\\
79 & 80 & 81 & 22 & 23 & 24 & 73 & 74 & 75\\
\hline
28 & 29 & 30 & 31 & 32 & 33 & 34 & 35 & 36\\
37 & 38 & 39 & 40 & 41 & 42 & 43 & 44 & 45\\
46 & 47 & 48 & 49 & 50 & 51 & 52 & 53 & 54\\
\hline
7 & 8 & 9 & 58 & 59 & 60 & 1 & 2 & 3\\
16 & 17 & 18 & 67 & 68 & 69 & 10 & 11 & 12\\
25 & 26 & 27 & 76 & 77 & 78 & 19 & 20 & 21
\end{array}\right]\\
\begin{array}{ccccccccc} & \textcircled{2} & & & & & & \textcircled{1} & \end{array}
\end{array}$$

The script is

```
Matr = [1:9; 10:18; 19:27; 28:36; 37:45; 46:54; 55:63; 64:72; 73:81];
Temp1 = Matr(1:3, 1:3);
Matr(1:3, 1:3) = Matr(7:9, 7:9);
Matr(7:9, 7:9) = Temp1;
Temp1 = Matr(1:3, 7:9);
```

```
Matr(1:3, 7:9) = Matr(7:9, 1:3);
Matr(7:9, 1:3) = Temp1;
Matr
```

Upon execution, we obtain

```
Matr =
61  62  63   4   5   6  55  56  57
70  71  72  13  14  15  64  65  66
79  80  81  22  23  24  73  74  75
28  29  30  31  32  33  34  35  36
37  38  39  40  41  42  43  44  45
46  47  48  49  50  51  52  53  54
 7   8   9  58  59  60   1   2   3
16  17  18  67  68  69  10  11  12
25  26  27  76  77  78  19  20  21
```

This type of swapping procedure is used frequently when elements of arrays are being rearranged. It requires the introduction of another variable (in this case *Temp1*) to temporarily store a value or set of values prior to completing the swap.

Additional Array Creation Functions

MATLAB provides two functions that can be used to create matrices by replicating a specified number of times a scalar, a column vector, a row vector, a matrix, or strings. These two functions are

```
repmat
```

and

```
meshgrid
```

which uses `repmat`. The general form of `repmat` is

```
repmat(x, r, c)
```

where *x* is either a scalar, vector, or matrix, *r* is the number copies of *x* that will be replicated as rows, and *c* is the number of copies of *x* that will be replicated as columns. The `repmat` function is very useful in generating annotated output as illustrated in Section 3.1.2. The `meshgrid` function has numerous applications in evaluating series as shown in Section 2.6.2 and subsequent chapters and in displaying three-dimensional surfaces as shown in Chapter 7 and subsequent chapters.

We shall now illustrate how `repmat` can be used to create different matrices and vectors from an original vector or matrix. We shall first create a column or a row vector of specified length in which each element of the vector has the same numerical value. Thus, if we wish to create a row vector *w* composed of six values of the number 45.72, the script is

```
w = repmat(45.72, 1, 6)
```

This expression is equivalent to a vector created from the fundamental form

```
w = [45.72, 45.72, 45.72, 45.72, 45.72, 45.72]
```

or created using subscript colon notation

```
w(1,1:6) = 45.72
```

If, instead, we want to create a (3 × 3) matrix of these values, then we have the script

```
W = repmat(45.72, 3, 3)
```

which could either have been created using the fundamental form

```
W = [45.72, 45.72, 45.72; 45.72, 45.72, 45.72; 45.72, 45.72, 45.72]
```

or by using subscript colon notation

```
W(1:3,1:3) = 45.72
```

Each of these expressions will produce in the command window

```
W =
   45.7200   45.7200   45.7200
   45.7200   45.7200   45.7200
   45.7200   45.7200   45.7200
```

Now consider the vector

$$s = [a_1 \quad a_2 \quad a_3 \quad a_4]$$

The expression

```
V = repmat(s, 3, 1)
```

creates the numerical equivalent[4] of the matrix

$$V = \left\{\begin{matrix} s \\ s \\ s \end{matrix}\right\} = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ a_1 & a_2 & a_3 & a_4 \\ a_1 & a_2 & a_3 & a_4 \end{bmatrix}$$

That is, it creates three rows of the vector s, with each row in this case having four columns. The expression

```
repmat(s, 3, 2)
```

creates the numerical equivalent of the matrix

$$V = \begin{bmatrix} s & s \\ s & s \\ s & s \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 & a_1 & a_2 & a_3 & a_4 \\ a_1 & a_2 & a_3 & a_4 & a_1 & a_2 & a_3 & a_4 \\ a_1 & a_2 & a_3 & a_4 & a_1 & a_2 & a_3 & a_4 \end{bmatrix}$$

[4] By numerical equivalent, we mean that in MATLAB, the v_{ij} have had numerical values assigned to them. The notation here is used to better illustrate what `repmat` does by symbolically showing the arrangement of the elements of the resulting array.

On the other hand, the command

```
V = repmat(s', 1, 3)
```

yields a matrix of three columns of the numerical equivalent of the column vector *s*', with each column in this case having four rows:

$$V = \{s' \quad s' \quad s'\} = \begin{bmatrix} a_1 & a_1 & a_1 \\ a_2 & a_2 & a_2 \\ a_3 & a_3 & a_3 \\ a_4 & a_4 & a_4 \end{bmatrix}$$

The expression

```
V = repmat(s', 2, 3)
```

gives the numerical equivalent of the matrix

$$V = \begin{bmatrix} s' & s' & s' \\ s' & s' & s' \end{bmatrix} = \begin{bmatrix} a_1 & a_1 & a_1 \\ a_2 & a_2 & a_2 \\ a_3 & a_3 & a_3 \\ a_4 & a_4 & a_4 \\ a_1 & a_1 & a_1 \\ a_2 & a_2 & a_2 \\ a_3 & a_3 & a_3 \\ a_4 & a_4 & a_4 \end{bmatrix}$$

If we have two row vectors *s* and *t*, then the MATLAB expression

```
[U, V] = meshgrid(s, t)
```

gives the same result as that produced by the following two expressions:

```
U = repmat(s, length(t), 1)
V = repmat(t', 1, length(s))
```

In either case, *U* and *V* are each matrices of order (length(t) × length(s)). Thus, if

$$s = [s_1 \quad s_2 \quad s_3 \quad s_4]$$
$$t = [t_1 \quad t_2 \quad t_3]$$

then,

```
[U, V] = meshgrid (s, t)
```

produces the numerical equivalent of the two (3 × 4) matrices

$$U = \begin{bmatrix} s_1 & s_2 & s_3 & s_4 \\ s_1 & s_2 & s_3 & s_4 \\ s_1 & s_2 & s_3 & s_4 \end{bmatrix} \tag{2.1}$$

$$V = \begin{bmatrix} t_1 & t_1 & t_1 & t_1 \\ t_2 & t_2 & t_2 & t_2 \\ t_3 & t_3 & t_3 & t_3 \end{bmatrix} \tag{2.2}$$

The `meshgrid` function can also be used to return only one matrix as follows:

```
W = meshgrid(s, t)
```

which creates $W = U$, where U is given by Eq. (2.1). The use of this form is illustrated in Example 2.8.

To illustrate `meshgrid`, consider the following script:

```
u = [1, 2, 3, 4];
v = [5, 6, 7];
[U, V] = meshgrid(u, v)
```

Upon execution, we obtain

```
U =
  1  2  3  4
  1  2  3  4
  1  2  3  4
V =
  5  5  5  5
  6  6  6  6
  7  7  7  7
```

However, when the u and v are interchanged, that is,

```
u = [1, 2, 3, 4];
v = [5, 6, 7];
[V, U] = meshgrid(v, u)
```

we obtain

```
V =
  5  6  7
  5  6  7
  5  6  7
  5  6  7
U =
  1  1  1
  2  2  2
  3  3  3
  4  4  4
```

There are two matrix manipulation functions that are useful in certain applications:

```
fliplr(A)
```

which flips the columns from left to right and

```
flipud(A)
```

which flips the rows from bottom to top.

Consider the (2 × 5) matrix:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \end{bmatrix} \rightarrow (2 \times 5)$$

which is created with the statement

$$A = [a_{11}\, a_{12}\, a_{13}\, a_{14}\, a_{15};\, a_{21}\, a_{22}\, a_{23}\, a_{24}\, a_{25}]$$

then,

$$\texttt{fliplr}(A) = \begin{bmatrix} a_{15} & a_{14} & a_{13} & a_{12} & a_{11} \\ a_{25} & a_{24} & a_{23} & a_{22} & a_{21} \end{bmatrix} \rightarrow (2 \times 5)$$

$$\texttt{flipud}(A) = \begin{bmatrix} a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \end{bmatrix} \rightarrow (2 \times 5)$$

and

$$\texttt{flipud}(\texttt{fliplr}(A)) = \begin{bmatrix} a_{25} & a_{24} & a_{23} & a_{22} & a_{21} \\ a_{15} & a_{14} & a_{13} & a_{12} & a_{11} \end{bmatrix} \rightarrow (2 \times 5)$$

The results of $\texttt{fliplr}(A)$ and $\texttt{flipud}(A)$ can also be obtained with the colon notation. For example,

```
C = fliplr(A)
```

produces the same results as

```
C = A(:,length(A):-1:1)
```

Now consider the array

$$C = [A, \texttt{fliplr}(A)]' = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \\ a_{13} & a_{21} \\ a_{14} & a_{24} \\ a_{15} & a_{25} \\ a_{15} & a_{25} \\ a_{14} & a_{24} \\ a_{13} & a_{23} \\ a_{12} & a_{22} \\ a_{11} & a_{21} \end{bmatrix} \rightarrow (10 \times 2)$$

Which has created two identical rows: 5 and 6. Suppose that we wish to remove one of these repeating rows. This is done by setting one of the rows to a null value using the expression [], where there is no space (blank) between the brackets. Then, either the expression

C(`length`(A),:)=[]

or

C(`length`(A)+1,:)=[]

reduces C to

$$C = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \\ a_{13} & a_{23} \\ a_{14} & a_{24} \\ a_{15} & a_{25} \\ a_{14} & a_{24} \\ a_{13} & a_{23} \\ a_{12} & a_{22} \\ a_{11} & a_{21} \end{bmatrix} \rightarrow (9 \times 2)$$

where the order of C is now (9×2). The expression $C($`length`$(A),:) = []$ means that all the columns of row number `length`(A) in C are to be assigned the value [] (removed, in this case). Although we know that the length of A is 5, it is a good practice to let MATLAB do the counting, hence, the use of the function `length`(A).

We further clarify the above notation by presenting the results of three different MATLAB operations. First we create the following two (2×5) matrices A and B

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \end{bmatrix}$$

Now consider their use in the following three MATLAB operations:

Addition/Subtraction: $\boldsymbol{C = A \pm B}$

$$C = \begin{bmatrix} a_{11} \pm b_{11} & a_{12} \pm b_{12} & a_{13} \pm b_{13} & a_{14} \pm b_{14} & a_{15} \pm b_{15} \\ a_{21} \pm b_{21} & a_{22} \pm b_{22} & a_{23} \pm b_{23} & a_{24} \pm b_{24} & a_{25} \pm b_{25} \end{bmatrix} \rightarrow (2 \times 5)$$

Thus, C is a (2×5) matrix.

Column Augmentation: $\boldsymbol{C = [A, B]}$

$$C = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \end{bmatrix} \rightarrow (2 \times 10)$$

Thus, C is a (2×10) matrix.

***Row Augmentation: C* = [*A*; *B*]**

$$C = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \end{bmatrix} \rightarrow (4 \times 5)$$

Thus, C is a (4 × 5) matrix.

Furthermore, if

$$x = [x_1 \;\; x_2 \;\; x_3]$$
$$y = [y_1 \;\; y_2 \;\; y_3]$$

then either

```
Z = [x', y']
```

or

```
Z = [x; y]'
```

produces

$$Z = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \end{bmatrix} \rightarrow (2 \times 3)$$

whereas

```
Z = [x'; y']
```

yields

$$Z = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} \rightarrow (6 \times 1)$$

These relationships are very useful when one must place data in a specific order.

Let us illustrate these results with the following two vectors: $a = [1, 2, 3]$ and $b = [4, 5, 6]$. The script is

```
x = [1, 2, 3];
y = [4, 5, 6];
Z1 = [x', y']
Z2 = [x; y]'
Z3 = [x'; y']
```

The execution of this script gives

```
Z1 =
   1   4
   2   5
   3   6
Z2 =
   1   4
   2   5
   3   6
Z3 =
   1
   2
   3
   4
   5
   6
```

2.5 DOT OPERATIONS

We now introduce MATLAB's dot (.) notation, which is MATLAB's syntax for performing on matrices of the *same* order, arithmetic operations on an element-by-element basis. Consider the following (3 × 4) matrices:

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{bmatrix}$$

and

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix}$$

We now write out explicitly the numerical equivalent form of the following MATLAB dot (.) operations:

$$Z_m = X.{*}M = \begin{bmatrix} x_{11}{*}m_{11} & x_{12}{*}m_{12} & x_{13}{*}m_{13} & x_{14}{*}m_{14} \\ x_{21}{*}m_{21} & x_{22}{*}m_{22} & x_{23}{*}m_{23} & x_{24}{*}m_{24} \\ x_{31}{*}m_{31} & x_{32}{*}m_{32} & x_{33}{*}m_{33} & x_{34}{*}m_{34} \end{bmatrix}$$

$$Z_d = X./M = \begin{bmatrix} x_{11}/m_{11} & x_{12}/m_{12} & x_{13}/m_{13} & x_{14}/m_{14} \\ x_{21}/m_{21} & x_{22}/m_{22} & x_{23}/m_{23} & x_{24}/m_{24} \\ x_{31}/m_{31} & x_{32}/m_{32} & x_{33}/m_{33} & x_{34}/m_{34} \end{bmatrix}$$

$$Z_e = X.{}^{\wedge}M = \begin{bmatrix} x_{11}{}^{\wedge}m_{11} & x_{12}{}^{\wedge}m_{12} & x_{13}{}^{\wedge}m_{13} & x_{14}{}^{\wedge}m_{14} \\ x_{21}{}^{\wedge}m_{21} & x_{22}{}^{\wedge}m_{22} & x_{23}{}^{\wedge}m_{23} & x_{24}{}^{\wedge}m_{24} \\ x_{31}{}^{\wedge}m_{31} & x_{32}{}^{\wedge}m_{32} & x_{33}{}^{\wedge}m_{33} & x_{34}{}^{\wedge}m_{34} \end{bmatrix}$$

Note that the dot (.) must be placed before the symbol for multiplication, division, and exponentiation, that is, * → .*; / → ./; and ^ → .^. The dot operation for either addition or subtraction is not required, since the matrix notation causes the same operation, that is, an element-by-element addition or subtraction. Recall the results at the end of the previous section and see Eq. (2.6).

We now examine several special cases of these three operations. For dot multiplication, if $X = x_0$, a scalar constant, then the dot operation is not needed and the multiplication can be written as

$$Z_m = x_0 * M$$

Similarly, when $M = m_0$, a scalar constant, we can write the multiplication as

$$Z_m = X * m_0$$

In both cases, the dot operations are not required.

For dot division, when $M = m_0$, a scalar constant, we have

$$Z_d = X/m_0$$

and the dot operation is not required. However, when $X = x_0$, a scalar constant, we must use dot division, that is,

$$Z_d = x_0./M$$

For exponentiation, we must always use dot operations whether $M = m_0$, a scalar constant, or $X = x_0$, a scalar constant, that is,

$$Z_e = x_0.{}^{\wedge}M$$

and

$$Z_e = X.{}^{\wedge}m_0$$

Example 2.4 Vector exponentiation

We shall illustrate the dot operation for exponentiation. Consider the computation of 2^k for $k = 1, 2, \ldots, 8$. The script is

```
x = 1:8;
y = 2.^x
```

which yields

```
y =
  2  4  8  16  32  64  128  256
```

Thus, the placement of a decimal point before the exponentiation operator (^) signifies to MATLAB that it is to take the scalar 2, compute its power at each of the values of x, and then place the results in the corresponding elements of a vector y of the same length. The previous script can be written more compactly as

```
y = 2.^(1:8)
```

where the parenthesis are required, or

```
y = 2.^[1:8]
```

where the brackets are required.

If the problem were reversed and, instead, we determine k^2, then the script is

```
y = (1:8).^2       % or y = [1:8].^2
```

which yields

```
y =
  1  4  9  16  25  36  49  64
```

If we let $f(Y)$ stand for any function, such as $f(Y) = \sin(Y)$, or $f(Y) = \cosh(Y)$, etc., on the matrix Y, then if Y, for example, is a (3×4) matrix

$$Z = f(Y) = \begin{bmatrix} f(y_{11}) & f(y_{12}) & f(y_{13}) & f(y_{14}) \\ f(y_{21}) & f(y_{22}) & f(y_{23}) & f(y_{24}) \\ f(y_{31}) & f(y_{32}) & f(y_{33}) & f(y_{34}) \end{bmatrix}$$

we can perform dot operations only if the order of each quantity is the same. For example, if a, b, c, d, and g are each a (3×2) matrix, then the expression

$$Z = \left[\tan a - g\left(\frac{b}{c}\right)^d \right]^2$$

is written as (and assuming that a, b, c, d, and g have all been assigned numerical values prior to this expression)

```
Z = (tan(a)-g.*(b./c).^d).^2;
```

which results in the elements of Z having the numerical values computed from the following expressions:

$$Z = \begin{bmatrix} (\tan(a_{11})-g_{11}{*}(b_{11}/c_{11})\hat{}\,d_{11})\hat{}\,2 & (\tan(a_{12})-g_{12}{*}(b_{12}/c_{12})\hat{}\,d_{12})\hat{}\,2 \\ (\tan(a_{21})-g_{21}{*}(b_{21}/c_{21})\hat{}\,d_{21})\hat{}\,2 & (\tan(a_{22})-g_{22}{*}(b_{22}/c_{22})\hat{}\,d_{22})\hat{}\,2 \\ (\tan(a_{31})-g_{31}{*}(b_{31}/c_{31})\hat{}\,d_{31})\hat{}\,2 & (\tan(a_{32})-g_{32}{*}(b_{32}/c_{32})\hat{}\,d_{32})\hat{}\,2 \end{bmatrix}$$

To illustrate the dot operations for expressions, let us evaluate

$$v = e^{-a_1 t}\frac{\sin(b_1 t + c_1)}{t + c_1}$$

for six equally spaced values of t in the interval $0 \le t \le 1$. We assume that $a_1 = 0.2$, $b_1 = 0.9$, and $c_1 = \pi/6$. The script is

```
a1 = 0.2;   b1 = 0.9;   c1 = pi/6;
t = linspace(0, 1, 6);
v = exp(-a1*t).*sin(b1*t+c1)./(t+c1)
```

Upon executing this script, we obtain

```
v =
0.9549   0.8590   0.7726   0.6900   0.6097   0.5316
```

Notice that, since a_1, b_1, and c_1 are scalar quantities and they only involve multiplication and addition, we do not have to use the dot operators.

Example 2.5 Creation of matrix elements

We shall create the elements of an $(N \times N)$ matrix H in which each element is given by

$$h_{mn} = \frac{1}{m + n - 1} \qquad m, n = 1, 2, \ldots, N$$

The script to generate this array for $N = 4$ is

```
N = 4;
mm = 1:N;    nn = mm;
[n, m] = meshgrid(nn, mm)
h = 1./(m+n-1)
```

Upon execution, we get

```
n =
   1   2   3   4
   1   2   3   4
   1   2   3   4
   1   2   3   4
m =
   1   1   1   1
   2   2   2   2
   3   3   3   3
   4   4   4   4
h =
   1.0000   0.5000   0.3333   0.2500
   0.5000   0.3333   0.2500   0.2000
   0.3333   0.2500   0.2000   0.1667
   0.2500   0.2000   0.1667   0.1429
```

where we have displayed the values of m and n to show that their respective values correspond to the subscripts of h_{mn}.

To show another application of dot operations, let us return to the meshgrid example where the two vectors $s = [s_1, s_2, s_3, s_4]$ and $t = [t_1, t_2, t_3]$ were used in the statement

```
[U, V] = meshgrid(s, t)
```

to produce the two (3×4) matrices

$$U = \begin{bmatrix} s_1 & s_2 & s_3 & s_4 \\ s_1 & s_2 & s_3 & s_4 \\ s_1 & s_2 & s_3 & s_4 \end{bmatrix} \quad \text{and} \quad V = \begin{bmatrix} t_1 & t_1 & t_1 & t_1 \\ t_2 & t_2 & t_2 & t_2 \\ t_3 & t_3 & t_3 & t_3 \end{bmatrix} \tag{2.3}$$

Suppose that we wish to multiply the corresponding elements of U and V. Then, the dot multiplication

```
Z = U.*V
```

results in numerical equivalent

$$Z = \begin{bmatrix} s_1{}^*t_1 & s_2{}^*t_1 & s_3{}^*t_1 & s_4{}^*t_1 \\ s_1{}^*t_2 & s_2{}^*t_2 & s_3{}^*t_2 & s_4{}^*t_2 \\ s_1{}^*t_3 & s_2{}^*t_3 & s_3{}^*t_3 & s_4{}^*t_3 \end{bmatrix} \tag{2.4}$$

The elements of Z can be interpreted as corresponding to the product of all combinations of the elements of vectors s and t. A similar interpretation is obtained when addition, subtraction, division, and exponentiation are performed, since the multiplication symbol (*) can be replaced by the respective operator.

Example 2.6 Polar to Cartesian coordinates

We shall map an array of polar coordinates to Cartesian coordinates through the relationships

$$x = r\cos\theta$$
$$y = r\sin\theta$$

as shown in Figure 2.2. We shall select three values of r in the range $0.5 \leq r \leq 1$ and four values of θ in the range $0 \leq \theta \leq \pi/2$. The script is

```
rr = linspace(0.5, 1, 3);
thet = linspace(0, pi/2, 4);
[r, theta] = meshgrid(rr, thet);
x = r.*cos(theta)
y = r.*sin(theta)
```

Upon execution, we find that

```
x =
   0.5000   0.7500   1.0000
   0.4330   0.6495   0.8660
   0.2500   0.3750   0.5000
   0.0000   0.0000   0.0000
```

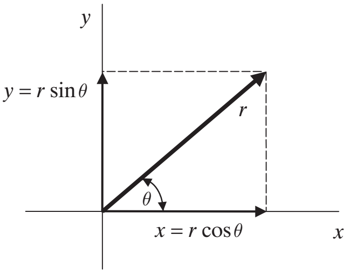


Figure 2.2 Transformation from polar coordinates to Cartesian coordinates.

```
y =
      0         0         0
 0.2500    0.3750    0.5000
 0.4330    0.6495    0.8660
 0.5000    0.7500    1.0000
```

Another way to get these results is shown in Example 2.11.

Summation

We now introduce the summation function

`sum(x)`

and the cumulative summation function

`cumsum(x)`

which are often used in conjunction with dot operations.

We examine `sum` first. When $v = [v_1, v_2, \ldots, v_n]$, then

$$S = \texttt{sum}(v) \rightarrow \sum_{k=1}^{\texttt{length}(v)} v_k$$

where S a scalar. When the argument is a matrix, the function sums the columns of the matrix, and returns a row vector whose length is equal to the number of columns of the original matrix. Thus, if Z is a (3×4) matrix with elements z_{ij}, then

$$S = \texttt{sum}(Z) \rightarrow \left[\sum_{n=1}^{3} z_{n1} \quad \sum_{n=1}^{3} z_{n2} \quad \sum_{n=1}^{3} z_{n3} \quad \sum_{n=1}^{3} z_{n4} \right] \rightarrow (1 \times 4) \tag{2.5}$$

is a four-element vector. To sum all the elements in an array, we use

$$S = \texttt{sum}(\texttt{sum}(Z)) \rightarrow \sum_{n=1}^{3} z_{n1} + \sum_{n=1}^{3} z_{n2} + \sum_{n=1}^{3} z_{n3} + \sum_{n=1}^{3} z_{n4} \rightarrow \sum_{i=1}^{4} \sum_{n=1}^{3} z_{ni} \rightarrow (1 \times 1)$$

which is a scalar.

Example 2.7 Summing a series

To illustrate the use of `sum`, consider the evaluation of the following equation:

$$z = \sum_{m=1}^{4} m^m$$

The script to evaluate this expression is

```
m = 1:4;
z = sum(m.^m)
```

which when executed gives

```
z =
  288
```

This script can be written more compactly as

```
z = sum((1:4).^(1:4))
```

Example 2.8 Approximation to the normal cumulative distribution function

An approximation to the normal cumulative probability distribution function, which estimates the probability P that $0 \leq X \leq x$ is given as[5]

$$P(x) = P(X \leq x) \cong 1 - \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \sum_{m=1}^{5} b_m (1 + 0.2316419x)^{-m} \qquad 0 \leq x \leq \infty$$

where $0.5 \leq P(x) \leq 1$ and

$$\begin{aligned} b_1 &= 0.319381530 \\ b_2 &= -0.356563782 \\ b_3 &= 1.781477937 \\ b_4 &= -1.821255978 \\ b_5 &= 1.330274429 \end{aligned}$$

The region for $-\infty \leq x \leq 0$ is obtained from $1-P(|x|)$, where $0 \leq 1-P(|x|) \leq 0.5$.

The objective is to compute and plot the cumulative distribution for $-3 \leq x \leq 3$ every $\Delta x = 0.2$. The script is

```
b = [0.319381530, –0.356563782, 1.781477937, . . .
   –1.821255978, 1.330274429];                               % (1×5)
m = 1:length(b);                                             % (1×5)
x = 0:0.2:3;                                                 % (1×16)
[mm, Xm] = meshgrid(m, (1./(1+0.231641*x)));                 % (16×5)
bmx = meshgrid(b, x);                                        % (16×5)
Px = 1 – exp(-0.5*x.^2).*sum((bmx.*(Xm.^mm))')/sqrt(pi*2);   % (1×16)
plot([–fliplr(x), x], [fliplr(1–Px), Px])
```

The first step is to convert the vector variables into matrices of the order (16 × 5) so that we are able to perform dot operations and the summation. The conversion of the various vectors is done with meshgrid. However, in the sum function, we must take the transpose of the results from meshgrid because the sum function sums a matrix on a column-by-column basis. In our case, we want to sum over five terms. Once we have the function for the region $0 \leq x \leq 3$, we can compute it for the region $-3 \leq x \leq 0$. The arguments of the plot function are the x- and y-coordinates. See Section 6.2 for a discussion on this function. The vector [-fliplr(x), x] ranges from $x = -3$ to $x = 3$ since the expression -fliplr(x) is the same as creating a new vector $x = -3$:0.2:0. The expression fliplr(1–Px) reverses the order of the elements of the vector $1-P(x)$ and creates a

[5] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards, Applied Mathematics Series 55, U.S. Government Printing Office, Washington D.C., 1964, p. 932.

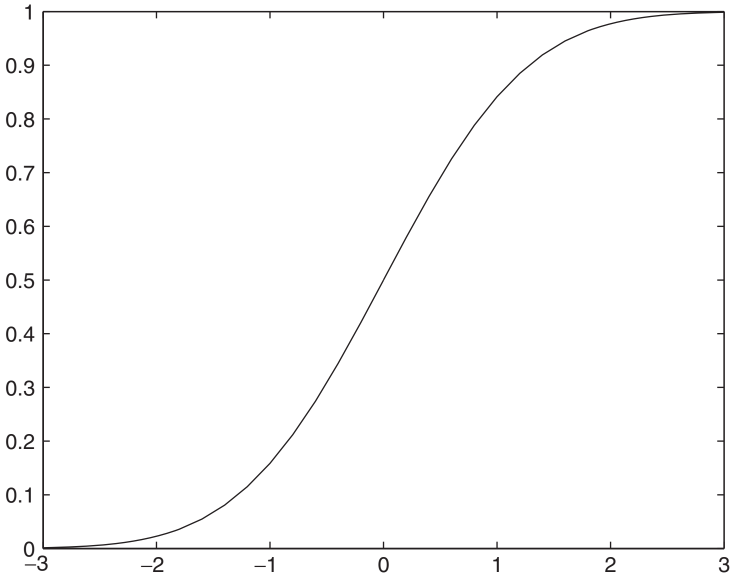


Figure 2.3 Normal cumulative probability distribution.

vector whose elements correspond to the negative x-values given by `-fliplr`(x). The execution of this script produces Figure 2.3.

The `cumsum` function for a vector v composed of n elements v_j is another vector of length n whose elements are

$$y = \texttt{cumsum}(v) \rightarrow \left[\sum_{k=1}^{1} v_k \quad \sum_{k=1}^{2} v_k \quad \dots \quad \sum_{k=1}^{n} v_k \right] \rightarrow (1 \times n)$$

On the other hand, if W is an $(m \times n)$ matrix composed of elements w_{jk}, then `cumsum`(W) is the following matrix:

$$Y = \texttt{cumsum}(W) \rightarrow \begin{bmatrix} \sum_{k=1}^{1} w_{k1} & \sum_{k=1}^{1} w_{k2} & \dots & \sum_{k=1}^{1} w_{kn} \\ \sum_{k=1}^{2} w_{k1} & \sum_{k=1}^{2} w_{k2} & & \\ \vdots & & \ddots & \\ \sum_{k=1}^{m} w_{k1} & & & \sum_{k=1}^{m} w_{kn} \end{bmatrix} \rightarrow (m \times n)$$

Example 2.9 Convergence of a series

We consider the series

$$S = \sum_{n=1}^{10} \frac{1}{n^2}$$

and explore its convergence for the first 10 terms. The script is

```
S = cumsum(1./(1:10).^2)
```

which upon execution gives

```
S =
  1.0000   1.2500   1.3611   1.4236   1.4636   1.4914   1.5118   1.5274   1.5398   1.5498
```

Each element of S is a partial sum of n terms of the series; that is, $S(1)$ is the first term of the series, $S(2)$ is the sum of the first two terms of the series, and so on.

Example 2.10 Evaluation of the hyperbolic secant

Let us evaluate the following series expression[6] for the hyperbolic secant for $N = 305$ and for five equally spaced values of x from $0 \le x \le 2$.

$$\operatorname{sech} x = 4\pi \sum_{n=1,3,5}^{N\to\infty} \frac{n(-1)^{(n-1)/2}}{(n\pi)^2 + 4x^2}$$

We shall also compare the summed values to the exact values. In order to perform dot operations, we first have to convert two vectors, one for the summation index n and the other for the argument x, into matrices of the same order using meshgrid.

The script is

```
nn = 1:2:305;                                                    % (1×153)
xx = linspace(0, 2, 5);                                          % (1×5)
se = sech(xx);                                                   % (1×5)
[x, n] = meshgrid(xx, nn);                                       % (153×5)
s = 4*pi*sum(n.*(−1).^((n−1)/2)./((pi*n).^2 + 4*x.^2));          % (1×5)
compare = [s' se' (100*(s–se)./se)']                             % (5×3)
```

which upon execution displays in the command window

```
compare =

  1.0021   1.0000   0.2080
  0.8889   0.8868   0.2346
  0.6501   0.6481   0.3210
  0.4272   0.4251   0.4894
  0.2679   0.2658   0.7827
```

where the middle column is the exact value and the right-hand column is the percentage error. We have selected the order of the arguments of meshgrid to produce matrices of order (153 × 5) because sum performs the summation on a column-by-column basis.

[6] L. B. W. Jolley, *Summation of Series*, 2nd ed., Dover Publications, New York, 1961.

2.6 MATHEMATICAL OPERATIONS WITH MATRICES

We now define several fundamental matrix operations: addition, subtraction, multiplication, inversion, determinants, solutions of systems of equations, and roots (eigenvalues). These results are then used to obtain numerical solutions to classes of engineering problems.

2.6.1 Addition and Subtraction

If we have two matrices A and B, each of the order $(m \times n)$, then

$$A \pm B = \begin{bmatrix} a_{11} \pm b_{11} & a_{12} \pm b_{12} & \dots & a_{1n} \pm b_{1n} \\ a_{21} \pm b_{21} & a_{22} \pm b_{22} & & \\ \vdots & & \ddots & \\ a_{m1} \pm b_{m1} & & & a_{mn} \pm b_{mn} \end{bmatrix} \rightarrow (m \times n) \qquad (2.6)$$

2.6.2 Multiplication

If we have an $(m \times k)$ matrix A and a $(k \times n)$ matrix B, then

$$C = AB = \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & & \vdots \\ \vdots & & \ddots & \\ c_{m1} & \dots & & c_{mn} \end{bmatrix} \rightarrow (m \times n) \qquad (2.7a)$$

where

$$c_{lp} = \sum_{j=1}^{k} a_{lj} b_{jp} \qquad (2.7b)$$

and C is of order $(m \times n)$. Notice that the product of two matrices is defined only when the adjacent integers of their respective orders are equal; k in this case. In other words, $(m \times k)(k \times n) \rightarrow (m \times n)$, where the notation indicates that we have summed k terms as indicated in Eq. (2.7b). The MATLAB expression for matrix multiplication is

```
C = A*B
```

Upon comparing Eq. (2.4) with Eq. (2.7), we see clearly the difference between dot multiplication and matrix multiplication. Dot multiplication performs the product on an element-by-element basis on matrices of the same order and results in a matrix of values of that order. Matrix multiplication performs an operation that sums the values of the appropriate column and row vectors as indicated in Eq. (2.7b) and places the result in a specific element of the resulting matrix, whose order, in general, is not the same as the order of either of the matrices being multiplied.

When $m = n$, it is noted that, in general, $AB \neq BA$. It can be shown that if $C = AB$, then its transpose is

$$C' = (AB)' = B'A'$$

If A is an identity matrix ($A = I$) and $m = n$, then

$$C = IB = BI = B$$

For example, let us multiply the following two matrices and then show numerically that the transpose can be obtained by either of the two ways given above.

$$A = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \end{bmatrix}$$

$$B = \begin{bmatrix} 11 & 12 \\ 21 & 22 \\ 31 & 32 \end{bmatrix}$$

The script is

```
A = [11, 12, 13; 21, 22, 23];
B = [11, 12; 21, 22; 31, 32];
C = A*B
Ctran1 = C'
Ctran2 = B'*A'
```

Upon execution, we obtain

```
C =
  776    812
  1406  1472
Ctran1 =
  776   1406
  812   1472
Ctran2 =
  776   1406
  812   1472
```

Let us examine the results of the matrix multiplication further and give one interpretation to them. Consider the following series:[7]

$$w(x, y) = \sum_{j=1}^{k} f_j(x) g_j(y)$$

Suppose that we are interested in the value of $w(x, y)$ over a range of values for x and y: $x = x_1, x_2, \ldots, x_m$ and $y = y_1, y_2, \ldots, y_n$. Then, one can consider

$$w(x_i, y_j) = \sum_{l=1}^{k} f_l(x_i) g_l(y_j) \quad i = 1, 2, \ldots, m \quad j = 1, 2, \ldots, n$$

[7] This form of a series results from a family of solutions to differential equations with certain boundary conditions.

as one element of a matrix W of order $(m \times n)$ as follows. Let F be a matrix of order $(m \times k)$

$$F = \begin{bmatrix} f_1(x_1) & f_2(x_1) & \dots & f_k(x_1) \\ f_1(x_2) & f_2(x_2) & & \\ \vdots & & \ddots & \\ f_1(x_m) & \dots & & f_k(x_m) \end{bmatrix} \rightarrow (m \times k)$$

and G be a matrix of order $(k \times n)$

$$G = \begin{bmatrix} g_1(y_1) & g_1(y_2) & \dots & g_1(y_n) \\ g_2(y_1) & g_2(y_2) & & \\ \vdots & & \ddots & \\ g_k(y_1) & \dots & & g_k(y_n) \end{bmatrix} \rightarrow (k \times n)$$

Then, from Eqs. (2.7),

$$W = FG = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & & \dots \\ \dots & & \dots & \\ w_{m1} & \dots & & w_{mn} \end{bmatrix} \rightarrow (m \times n) \tag{2.8}$$

where

$$w_{ij} = w(x_i, y_j) = \sum_{l=1}^{k} f_l(x_i) g_l(y_j) \tag{2.9}$$

In other words, matrix multiplication performs the summation of the series at each combination of values for x and y. As we shall see subsequently, this observation provides a very compact means of summing a series at each point in a grid that is defined by all combinations of the elements of the vectors x and y.

We now consider three special cases of this general matrix multiplication given by Eqs. (2.7):

1. The product of a row vector and a column vector.
2. The product of a column vector and a row vector.
3. The product of a row vector and a matrix.

These three cases provide one means by which we can take advantage of MATLAB's compact notation and matrix solution methods for a class of engineering problems.

Case 1—Product of a row vector and a column vector

Let a be the row vector

$$a = [a_1 \quad a_2 \quad \dots \quad a_k] \rightarrow (1 \times k)$$

which is of order $(1 \times k)$ and b be the column vector

$$b = [b_1 \quad b_2 \quad \ldots \quad b_k]' \rightarrow (k \times 1)$$

which is of order $(k \times 1)$. Then, from Eqs. (2.7), the product $d = ab$ is the scalar

$$d = ab = [a_1 \quad a_2 \quad \ldots \quad a_k]\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{bmatrix} = \left[\sum_{j=1}^{k} a_j b_j\right] = \sum_{j=1}^{k} a_j b_j \rightarrow (1 \times 1) \quad (2.10)$$

since the product of the orders gives $(1 \times k)(k \times 1) \rightarrow (1 \times 1)$. This is called the dot product of two vectors. The MATLAB expression for matrix multiplication of the two vectors as defined above is either

```
d = a*b
```

or

```
d = dot(a, b)
```

Case 2—Product of a column vector and a row vector

Let b be an $(m \times 1)$ column vector and a a $(1 \times n)$ row vector. Then, the product $H = ba$ is

$$H = ba = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}[a_1 \;\; a_2 \;\; \ldots \;\; a_n] = \begin{bmatrix} b_1a_1 & b_1a_2 & \ldots & b_1a_n \\ b_2a_1 & b_2a_2 & & \\ \vdots & & \ddots & \\ b_ma_1 & & \ldots & b_ma_n \end{bmatrix} \rightarrow (m \times n) \quad (2.11)$$

which is a matrix of order $(m \times n)$, since the product of their orders gives $(m \times 1)(1 \times n) \rightarrow (m \times n)$. Thus, the elements of H, which are $h_{ij} = b_i a_j$, are the individual products of all the combinations of the elements of b and a. Notice that this operation produces the same result as that given by Eq. (2.4).

Example 2.11 Polar to Cartesian coordinates revisited

Let us again examine the transformation from polar coordinates to Cartesian coordinates as shown in Figure 2.2, that is,

$$x = r\cos\theta$$
$$y = r\sin\theta$$

If we have a vector of radial values $r = [r_1\, r_2 \ldots r_m]$ and a vector of angular values $\theta = [\theta_1\, \theta_2 \ldots \theta_n]$, then the corresponding Cartesian coordinates are[8]

[8] This conversion can also be performed with `pol2cart`, however, this function is restricted to the case when $m = n$.

$$X = r'{*}\cos(\theta) = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \end{bmatrix} [\cos\theta_1 \quad \cos\theta_2 \quad \ldots \quad \cos\theta_n]$$

$$= \begin{bmatrix} r_1\cos\theta_1 & r_1\cos\theta_2 & \ldots & r_1\cos\theta_n \\ r_2\cos\theta_1 & r_2\cos\theta_2 & & \\ \vdots & & \ddots & \\ r_m\cos\theta_1 & & \ldots & r_m\cos\theta_n \end{bmatrix} \tag{2.12a}$$

and

$$Y = r'{*}\sin(\theta) = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_m \end{bmatrix} [\sin\theta_1 \quad \sin\theta_2 \quad \ldots \quad \sin\theta_n]$$

$$= \begin{bmatrix} r_1\sin\theta_1 & r_1\sin\theta_2 & \ldots & r_1\sin\theta_n \\ r_2\sin\theta_1 & r_2\sin\theta_2 & & \\ \vdots & & \ddots & \\ r_m\sin\theta_1 & & \ldots & r_m\sin\theta_n \end{bmatrix} \tag{2.12b}$$

Thus, we have mapped the polar coordinates into their Cartesian counterparts. This procedure is very useful in plotting results, as illustrated in Example 2.12.

Example 2.12 Mode shape of a circular membrane

Consider the following mode shape for a solid circular membrane clamped along its outer boundary $r = 1$

$$z(r, \phi) = J_1(3.8316r)\cos(\phi)$$

where $J_1(x)$ is the Bessel function[9] of the first kind of order 1 and (r, ϕ) are the polar coordinates of any point on the membrane. The Bessel function is determined by the function

```
besselj(n, x)
```

where n is the order and x its argument. The origin of the coordinate system is at the center of the membrane, which is at $r = 0$. The value 3.8316 is one of the natural frequency coefficients for the membrane. This mode shape can be plotted by using the surface plotting function

```
mesh(x, y, z)
```

where (x, y) are the Cartesian coordinates of a point on the surface $z(x, y)$. The mesh function is discussed in Section 7.2. We shall plot the surface over the range $0 \leq r \leq 1$ in increments of 0.05 and over the range $0 \leq \theta \leq 2\pi$ in increments of $\pi/20$. Using Eqs. (2.12), the script is

[9] See, for example, F. B. Hildebrand, *Advanced Calculus for Applications*, Prentice-Hall, Saddle River, NJ, 1976.

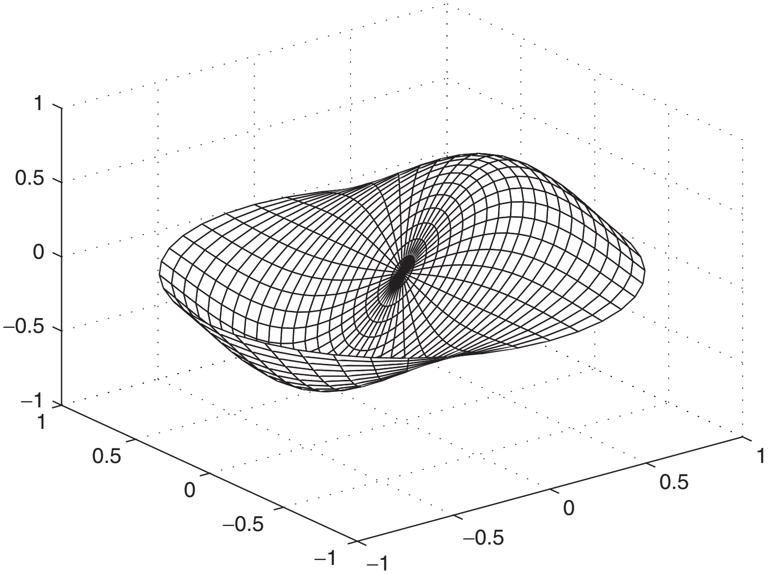


Figure 2.4 Mode shape of a clamped solid circular membrane.

```
r = [0:0.05:1]';                         % (21×1)
phi = 0:pi/20:2*pi;                      % (1×41)
x = r*cos(phi);                          % (21×41)
y = r*sin(phi);                          % (21×41)
z = besselj(1, 3.8316*r)*cos(phi);       % (21×41)
mesh(x, y, z)
```

The coordinate transformations are required because the surface as defined by the `mesh` function has to be plotted in the Cartesian coordinate system. It should also be realized that this technique works because the functions `cos`, `sin`, and `besselj` accept matrices for their arguments and return matrices of the same order. The execution of this script results in Figure 2.4.

Example 2.13 A solution to the Laplace equation

The solution of the Laplace equation in terms of the variable $u(\xi, \eta)$ and subject to the boundary conditions $u(0, \eta) = u(1, \eta) = u(\xi, 1) = 0$ and $u(\xi, 0) = \xi(1-\xi)$ is

$$u(\xi, \eta) = 4 \sum_{n=1}^{N \to \infty} \frac{1-\cos(n\pi)}{(n\pi)^3} e^{-n\pi\xi} \sin(n\pi\eta)$$

where $0 \le \eta \le 1$ and $\xi \ge 0$. We shall plot the surface $u(\xi, \eta)$ using `mesh`$(\eta, \xi, u(\xi, \eta))$ for $N = 25$ and for increments $\Delta\eta = 0.025$ and $\Delta\xi = 0.05$ up to $\xi_{\max} = 0.7$.

The approach to programming this series expression is to manipulate the multiplicative terms comprising the summation so that the summation is a natural outcome of matrix multiplication. That is, we would like to manipulate the expressions appearing in the summation so that Eqs. (2.8), (2.9), and (2.11) apply. The manipulation is performed in several stages as follows: First, we assume that the sizes of the vectors are

$n \rightarrow (1 \times N)$, $\eta \rightarrow (1 \times N_e)$, and $\xi \rightarrow (1 \times N_x)$. Next, we place these orders beneath the appropriate terms in the summation as shown below.

$$\underbrace{\left(\frac{1 - \cos n\pi}{(n\pi)^3}\right)}_{(1 \times N)} \times \underbrace{(e^{-n\pi\xi})}_{(1 \times N)(1 \times N_x)} \times \underbrace{(\sin n\pi\eta)}_{(1 \times N)(1 \times N_e)}$$

We see that the sizes of the various vectors in the three expressions are not yet correctly sized so that we can perform matrix and dot multiplications. In the second and third terms, the inner products are incorrect, therefore, we first have to take the transpose of the $(1 \times N)$ arrays. When this is done, the second term becomes $(1 \times N)'(1 \times N_x) \rightarrow (N \times 1)(1 \times N_x) \rightarrow (N \times N_x)$ and the third term becomes $(1 \times N)'(1 \times N_e) \rightarrow (N \times 1)(1 \times N_e) \rightarrow (N \times N_e)$. Thus,

$$\underbrace{\left(\frac{1 - \cos n\pi}{(n\pi)^3}\right)}_{(1 \times N)} \times \underbrace{(e^{-n'\pi\xi})}_{\substack{(N \times 1)(1 \times N_x) \\ \rightarrow (N \times N_x)}} \times \underbrace{(\sin n'\pi\eta)}_{\substack{(N \times 1)(1 \times N_e) \\ \rightarrow (N \times N_e)}}$$

The first expression does not require any multiplication of vectors; it only requires dot division. However, if we make this term of the same size (order) as the second term, we can perform a dot multiplication with the second term because MATLAB performs its operations on each level in the hierarchy from left to right. Thus, we must convert the first term, which is a $(1 \times N)$ vector, into an array that is the size of the adjacent expression, which is $(N \times N_x)$. This is accomplished by using `meshgrid`. Thus,

$$\underbrace{\left(\frac{1 - \cos n\pi}{(n\pi)^3}\right)}_{\texttt{meshgrid} \rightarrow (N \times N_x)} \times \underbrace{(e^{-n'\pi\xi})}_{(N \times N_x)} \times \underbrace{(\sin n'\pi\eta)}_{(N \times N_e)}$$

After this conversion, we can use dot multiplication to multiply the modified first and second expressions. The result, which we denote R_{nx}, is a matrix of order $(N \times N_x)$. Thus,

$$\underbrace{\left(\frac{1 - \cos n\pi}{(n\pi)^3}\right) \times \left(e^{-n'\pi\xi}\right)}_{R_{nx} = (N \times N_x)[\text{dot multiplication}]} \times \underbrace{(\sin n'\pi\eta)}_{(N \times N_e)}$$

Next, R_{nx} has to be multiplied with the third term, whose modified order is $(N \times N_e)$. Before this multiplication can be performed, however, we must take the transpose of R_{nx} so that the dimensions of the inner product are equal. This results in the matrix product $(R_{nx})'(N \times N_e) \rightarrow (N \times N_x)'(N \times N_e) \rightarrow (N_x \times N)(N \times N_e) \rightarrow (N_x \times N_e)$. Finally,

$$\underbrace{\left[\left(\frac{1 - \cos n\pi}{(n\pi)^3}\right) \times \left(e^{-n'\pi\xi}\right)\right]'}_{(N_x \times N)} \times \underbrace{(\sin n'\pi\eta)}_{(N \times N_e)}$$

Thus, we have summed over n for each combination of ξ and η, which is what we set out to do.

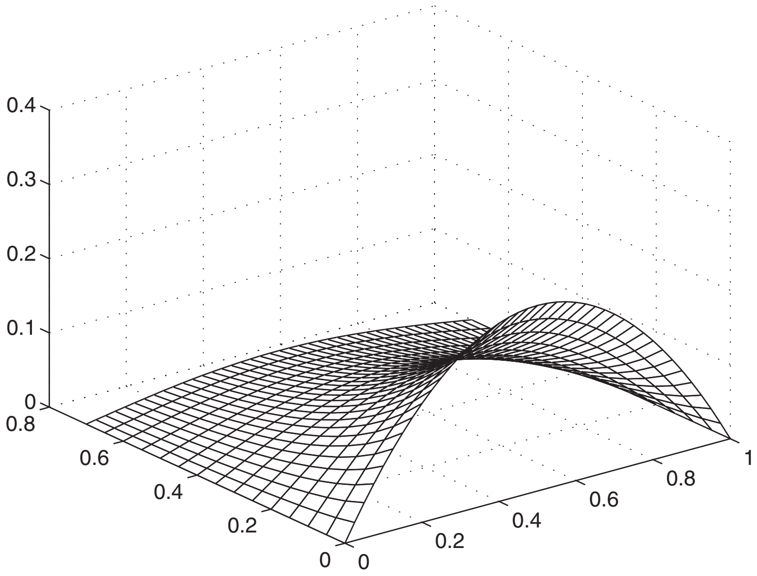


Figure 2.5 Display of a series solution to the Laplace equation.

The script is

```
n = (1:25)*pi;                                          % (1×25)
eta = 0:0.025:1;                                        % (1×41)
xi = 0:0.05:0.7;                                        % (1×15)
[X1, temp1] = meshgrid(xi, (1-cos(n))./n.^3);           % (25×15)
temp2 = exp(-n′*xi);                                    % (25×15)
Rnx = temp1.*temp2;                                     % (25×15)
temp3 = sin(n′*eta);                                    % (25×41)
u = 4*Rnx′*temp3;                                       % (15×41)
mesh(eta, xi, u)
```

The result is shown in Figure 2.5. In the mesh command, MATLAB allows one to have the first two arguments, *eta* and *xi*, be vectors whose lengths agree with the respective values of the order of *u*. See the *Help* file for mesh.

Case 3—Product of a row vector and a matrix

Let B be an $(m \times n)$ matrix and a a $(1 \times m)$ row vector. Then, the product $g = aB$ is

$$g = aB = [a_1\ a_2\ \cdots\ a_m]\begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & & \\ \vdots & & \ddots & \\ b_{m1} & & \cdots & b_{mn} \end{bmatrix} \tag{2.13}$$

$$= \left[\sum_{k=1}^{m} a_k b_{k1}\ \sum_{k=1}^{m} a_k b_{k2}\ \ldots\ \sum_{k=1}^{m} a_k b_{kn} \right] \rightarrow (1 \times n)$$

which is a row vector of order $(1 \times n)$, since the product of their orders gives $(1 \times m)(m \times n) \rightarrow (1 \times n)$.

This result can be interpreted as follows. Consider the series[10]

$$r(x) = \sum_{k=1}^{m} p_k h_k(x) \tag{2.14}$$

Suppose that we are interested in the value of $r(x)$ over a range of values $x_1, x_2, \ldots, x_n$. Then, one can consider

$$r(x_i) = \sum_{k=1}^{m} p_k h_k(x_i) \qquad i = 1, 2, \ldots, n$$

one element of a vector r of order $(1 \times n)$ as follows. Let p be a vector of order $(1 \times m)$ with elements p_k and V be the $(m \times n)$ matrix

$$V = \begin{bmatrix} h_1(x_1) & h_1(x_2) & \ldots & h_1(x_n) \\ h_2(x_1) & h_2(x_2) & & \\ \vdots & & \ddots & \\ h_m(x_1) & & & h_m(x_n) \end{bmatrix} \rightarrow (m \times n)$$

Then, $r = pV$ gives

$$r = pV = \left[\sum_{k=1}^{m} p_k h_k(x_1) \quad \sum_{k=1}^{m} p_k h_k(x_2) \quad \ldots \quad \sum_{k=1}^{m} p_k h_k(x_n) \right] \rightarrow (1 \times n)$$

Example 2.14 Summation of a Fourier series

The Fourier series representation of a rectangular pulse of duration d and period T is given by[11]

$$f(\tau) = \frac{d}{T}\left[1 + 2\sum_{k=1}^{K\rightarrow\infty} \frac{\sin(k\pi d/T)}{(k\pi d/T)} \cos(2\pi k\tau)\right]$$

where $\tau = t/T$. We see that this equation is of the form given by Eq. (2.14). Let us sum 150 terms of $f(\tau)$ and plot it from $-1/2 < \tau < 1/2$ when $d/T = 0.25$. To do this, we use `plot` (x, y), where $x = \tau$ and $y = f(\tau)$. See Section 6.2 for a discussion of `plot`.

Comparing the series expression for $f(\tau)$ with that given in Eq. (2.14), we find that

$$\begin{aligned} r(x_i) &\rightarrow f(\tau_i) \\ p_k &\rightarrow \frac{\sin(k\pi d/T)}{(k\pi d/T)} \\ h_k(x_i) &\rightarrow h_k(\tau_i) \rightarrow \cos(2\pi k\tau_i) \end{aligned}$$

If the order of k is $(1 \times K)$ and that of τ is $(1 \times N_\tau)$, then the order p_k is $(1 \times K)$. We see that, as written, the vector product of $k\tau \rightarrow (1 \times K)(1 \times N_\tau)$ cannot be performed because the dimensions of the inner product do not agree. In order to obtain a valid

[10] Series of this form result from the solution of differential equations with certain boundary conditions and from Fourier series expansions of periodic functions.

[11] See, for example, H. P. Hsu, *Applied Fourier Analysis*, Harcourt Brace Jovanovich, San Diego, CA, 1984.

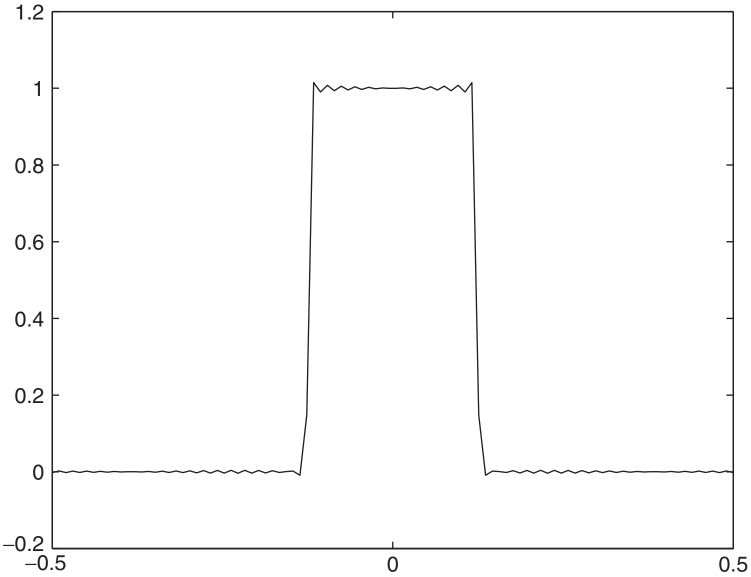


Figure 2.6 Summation of 150 terms of a Fourier series representation of a periodic pulse.

multiplication, we first take the transpose of k, which results in $k'\tau \rightarrow (1 \times K)'(1 \times N_\tau) \rightarrow (K \times N_\tau)$. Then, the matrix product $ph \rightarrow (1 \times K)(K \times N_\tau) \rightarrow (1 \times N_\tau)$, with the summation being taken over all k, $k = 1, \ldots, K$.

The script is

```
k = 1:150;                              % (1×150)
tau = linspace(-0.5, 0.5, 100);         % (1×100)
sk = sin(pi*k/4)./(pi*k/4);             % (1×150)
cntau = cos(2*pi*k'*tau);               % (150×100)
f = 0.25*(1+2*sk*cntau);                % (1×150)(150×100) → (1×100)
plot(tau, f)
```

The execution of this script produces Figure 2.6.

2.6.3 Determinants

A determinant of a matrix A of order $(n \times n)$ is represented symbolically as

$$|A| = \begin{vmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & & \vdots \\ \vdots & & \ldots & \\ a_{n1} & \ldots & & a_{nn} \end{vmatrix}$$

For $n = 2$

$$|A| = a_{11}a_{22} - a_{12}a_{21}$$

For $n = 3$

$$|A| = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33}$$

The MATLAB expression for the determinant is

```
det(a)
```

For example, if A is defined as

$$A = \begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix}$$

then, the determinant of A is obtained from

```
A = [1, 3; 4, 2];
d = det(A)
```

which upon execution gives

```
d =
  -10
```

A class of problems, called the eigenvalue problems, results in a determinant of the form

$$|A - \lambda B| = 0$$

where A and B are $(n \times n)$ matrices and $\lambda_j, j = 1, 2, \ldots, n$ are the roots (called eigenvalues) of this equation. See Section 9.3 for applications of this equation in the area of vibrations. The solution of this polynomial equation is obtained from eig, which has several forms, one of which is

```
lambda = eig(A, B)
```

Example 2.15 Eigenvalues of an oscillating spring-mass system

We shall determine the eigenvalues of a three-degree-of-freedom spring-mass system whose characteristic equation is

$$|K - \omega^2 M| = 0$$

where the stiffness matrix K is given by

$$K = \begin{bmatrix} 50 & -30 & 0 \\ -30 & 70 & -40 \\ 0 & -40 & 50 \end{bmatrix} \text{N/m}$$

the mass matrix is given by

$$M = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 1.4 & 0 \\ 0 & 0 & 5 \end{bmatrix} \text{kg}$$

and the eigenvalue $\lambda = \omega^2$ is related to the natural frequency of the system by $\omega_j = \sqrt{\lambda_j}$ rad/s, $j = 1, 2, 3$.

The natural frequencies are obtained from the following script:

```
K = [50, -30, 0; -30, 70, -40; 0, -40, 50];
M = diag([3, 1.4, 5]);
w = sqrt(eig(K, M))
```

which upon execution gives

```
w =
   1.6734
   3.7772
   7.7201
```

Example 2.16 Transformation of a polynomial

A polynomial of the form

$$ax^2 + by^2 + cz^2 + 2dxy + 2exz + 2gyz$$

where a, b, c, d, e, and g are real numbers, can be transformed into the real diagonal form

$$r_1x'^2 + r_2y'^2 + r_3z'^2$$

where x', y', and z' is another coordinate system whose origin is also at (0, 0, 0), and $r_1 \geq r_2 \geq r_3$ are the roots of

$$|A - rI| = 0$$

The matrix I is the identity matrix and A is the real symmetric matrix

$$A = \begin{bmatrix} a & d & e \\ d & b & g \\ e & g & c \end{bmatrix}$$

Consider the polynomial

$$4x^2 + 3y^2 - z^2 - 12xy + 4exz - 8gyz$$

Thus,

$$A = \begin{bmatrix} 4 & -6 & 2 \\ -6 & 3 & -4 \\ 2 & -4 & -1 \end{bmatrix}$$

To determine the roots r_j, we use the following script:

```
r = sort(eig([4, -6, 2; -6, 3, -4; 2, -4, -1], eye(3)), 'descend')
```

which upon execution gives

```
r =
  11.0000
   -1.0000
   -4.0000
```

We have used the sort option 'descend' so that we can have the roots in the order that is required: $r_1 \geq r_2 \geq r_3$. The real diagonal form, therefore, is

$$11x'^2 - y'^2 - 4z'^2$$

Example 2.17 Equation of a straight line determined from two distinct points

The equation of a straight line in a plane can be determined for the points (x_1, y_1) and (x_2, y_2) from the symbolic solution to

$$\det \begin{vmatrix} x & y & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix} = 0$$

If $(x_1, y_1) = (1, 2)$ and $(x_2, y_2) = (3, 5)$, then the equation of a straight line is determined from the following script:

```
syms x y
x1 = 1;   y1 = 2;
x2 = 3;   y2 = 5;
z = det([x y 1; x1 y1 1; x2 y2 1])
```

where det is also used to obtain the determinant symbolically. Upon execution, we obtain

```
z =
  2*y - 3*x - 1
```

Thus, the equation of the line connecting the two points is

$$y = \frac{3}{2}x + \frac{1}{2}$$

2.6.4 Matrix Inverse

The inverse of a square matrix A is an operation such that

$$A^{-1}A = AA^{-1} = I$$

provided that A is not singular, that is, its determinant is not equal to zero ($|A| \neq 0$). The quantity I is the identity matrix. The superscript "-1" denotes the inverse. The expression for obtaining the inverse of matrix A is either

```
inv(A)
```

or

```
A^-1
```

It is important to note that $1/A \neq A$^(-1); $1/A$ will cause the system to respond with an error message. However, $1./A$ is a valid expression, but it is not, in general, equal to A^{-1}. The inverse can also be obtained using the backslash operator, which is discussed in Section 2.6.5.

Example 2.18 Inverse of a matrix

Consider the (3×3) matrix M created by the `magic` function. Its inverse is obtained from the script

```
invM = inv(magic(3))
```

Upon executing this script, we obtain

```
invM =
   0.1472   -0.1444    0.0639
  -0.0611    0.0222    0.1056
  -0.0194    0.1889   -0.1028
```

We can verify that the product of a square matrix and its inverse is the identity matrix by modifying the above script as shown below:

```
invM = inv(magic(3));
IdentMat = invM*magic(3)
```

Its execution gives

```
IdentMat =
  1.0000        0     -0.0000
       0   1.0000           0
       0   0.0000      1.0000
```

Now let us determine the inverse of the matrix

$$C = \begin{bmatrix} 1 & 2 & 3 \\ 6 & 9 & 13 \\ 6 & 12 & 18 \end{bmatrix}$$

The script is

```
C = [1, 2, 3; 6, 9, 13; 6, 12, 18];
Iv = inv(C)
```

The execution of this script gives

```
Warning: Matrix is close to singular or badly scaled.
Results may be inaccurate. RCOND = 2.176908e-018.
Iv =
  1.0e+015 *
   0.0000   -0.2502    1.5012
  -0.0000    1.2510   -7.5060
        0   -0.7506    4.5036
```

which contains an error message and an inverse that has a column of zeros. The number and operator in the second line, 1.0e+015*, indicates that each number that follows is to be multiplied by 10^{15}. The quantity RCOND is the condition number of the matrix; a well-conditioned matrix has a condition number close to 1 and an ill-conditioned matrix has a value close to zero. If det had been used, we would have found that $|C| = 0$ and, therefore, the matrix does not have an inverse.

Another way to determine whether the matrix has an inverse is to use

```
rank(A)
```

which provides an estimate of the number of linearly independent rows or columns of a full matrix. Thus, for an $(n \times n)$ matrix, the number of linearly independent rows or columns is n minus its rank. In this case, rank(C) brings back a 2, indicating that $3-2 = 1$ row or column is linearly proportional to another one. In this case, we see that row three is six times that of row one.

Example 2.19 Symbolic inverse of a matrix

Consider the matrix

$$w = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

We shall show symbolically that this matrix has the following properties:

$$\begin{aligned} \det|w| &= 1 \\ w' &= w^{-1} \end{aligned}$$

The script is

```
syms th real
w = [cos(th), sin(th); -sin(th), cos(th)];
D = simple(det(w))
Invw = simple(inv(w))
Transw = simple(w')
```

where the qualification that *th* is a real quantity is needed since the symbolic transpose assumes that *th* can be a complex quantity. Note that inv is also used to obtain the inverse symbolically. Upon execution, we obtain

```
D =
  1
Invw =
  [cos(th), -sin(th)]
  [sin(th), cos(th)]
Transw =
  [cos(th), -sin(th)]
  [sin(th), cos(th)]
```

2.6.5 Solution of a System of Equations

Consider the following system of n equations and n unknowns x_k, $k = 1, 2, \ldots, n$

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

We can rewrite this system of equations in matrix notation as follows:

$$Ax = b$$

where A is the $(n \times n)$ matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & & \vdots \\ \vdots & & \ddots & \\ a_{n1} & \ldots & & a_{nn} \end{bmatrix} \rightarrow (n \times n)$$

and x and b are, respectively, the $(n \times 1)$ column vectors

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \rightarrow (n \times 1) \qquad \text{and} \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \rightarrow (n \times 1)$$

The solution is obtained by premultiplying both sides of the matrix equation by A^{-1}. Thus,

$$\begin{aligned} A^{-1}Ax &= A^{-1}b \\ x &= A^{-1}b \end{aligned}$$

since $A^{-1}A = I$, the identity matrix, and $Ix = x$. The preferred expression for solving this system of equations is[12]

x = A\b

or

x = `linsolve`(A, b)

where the backslash operator indicates matrix division and is referred to by MATLAB as left matrix divide. Left division uses a procedure that is more

[12] The notation $A\backslash b$ can be applied even when A is not a square matrix, whereas `inv`(A) is only applicable when A is square. That is, if A is an $(m \times n)$ matrix, x an $(n \times 1)$ vector, and b an $(m \times 1)$ vector, then if $Ax = b$, left division $A\backslash b$ finds $x = cb$, where $c = (A'A)^{-1}A'$ is the pseudo-inverse of A.

numerically stable when compared to the methods used for either of the following alternative notations:

```
x = A^-1*b
```

or

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

For large matrices, these two alternative expressions execute considerably slower than when the backslash operator is used.

Systems of equations can also be solved by using the symbolic toolbox. For example, consider the following two equations:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{Bmatrix} x \\ y \end{Bmatrix} = \begin{Bmatrix} w \\ z \end{Bmatrix}$$

To solve for x and y, we use the following script:

```
syms a b c d w z real
A = [a b; c d];
b = [w, z];
s = simple(inv(A)*b')
```

the execution of which gives

```
s =
  -(d*w-b*z)/(-a*d+b*c)
  (c*w-a*z)/(-a*d+b*c)
```

where $s(1) = x$ and $s(2) = y$.

Example 2.20 Solution of a system of equations

Consider the following system of equations

$$\begin{aligned} 8x_1 + x_2 + 6x_3 &= 7.5 \\ 3x_1 + 5x_2 + 7x_3 &= 4 \\ 4x_1 + 9x_2 + 2x_3 &= 12 \end{aligned}$$

which, in matrix notation, is

$$\begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7.5 \\ 4 \\ 12 \end{bmatrix}$$

The solution is obtained with the following script:

```
A = [8, 1, 6; 3, 5, 7; 4, 9, 2];
b = [7.5, 4, 12]';
x = A\b
```

which upon execution gives

```
x =
   1.2931
   0.8972
  -0.6236
```

This script could also have been written compactly as

```
x = [8, 1, 6; 3, 5, 7; 4, 9, 2]\[7.5, 4, 12]'
```

Since $b = Ax$, we can verify that the above solution is correct by modifying the above script as follows:

```
A = [8, 1, 6; 3, 5, 7; 4, 9, 2];
b = [7.5, 4, 12]';
x = A\b;
z = A*x
```

Upon execution, we find that

```
z =
    7.5000
    4.0000
   12.0000
```

Example 2.21 Temperatures in a slab

A thin square metal plate has a uniform temperature of 80°C on two opposite edges, a temperature of 120°C on the third edge, and a temperature of 60°C on the remaining edge. A mathematical procedure to approximate the temperature at six uniformly spaced interior points results in the following equations:[13]

$$\begin{aligned}
4T_1 - T_2 - T_6 &= 200\\
-T_1 + 4T_2 - T_3 - T_5 &= 80\\
-T_2 + 4T_3 - T_4 &= 140\\
-T_3 + 4T_4 - T_5 &= 140\\
-T_2 - T_4 + 4T_5 - T_6 &= 80\\
-T_1 - T_5 + 4T_6 &= 200
\end{aligned}$$

The temperatures are determined from the following script:

```
c = [4  -1   0   0   0  -1
     -1  4  -1   0  -1   0
     0  -1   4  -1   0   0
     0   0  -1   4  -1   0
     0  -1   0  -1   4  -1
     -1  0   0   0  -1   4];
d = [200, 80, 140, 140, 80, 200];
T = linsolve(c, d')
```

[13] F. Szabo, *Linear Algebra: An Introduction Using Maple*, Harcourt/Academic Press, San Diego, CA, 2002, p. 120.

The execution of this script gives

```
T =
  94.2857
  82.8571
  74.2857
  74.2857
  82.8571
  94.2857
```

where the first number is the value of T_1 and the last number the value of T_6.

Example 2.22 Current flowing in an electrical resistor circuit

An analysis of an electrical resistor circuit with two dc voltage sources E_1 and E_2 produces the following equations from which the loop currents i_2, i_2, and i_3 are determined:

$$\begin{aligned}(R_1 + R_2 + R_3)i_1 - R_2i_2 - R_3i_3 &= -E_1\\ -R_2i_1 + (R_2 + R_4 + R_5)i_2 - R_5i_3 &= 0\\ -R_3i_1 - R_5i_2 + (R_3 + R_5 + R_6)i_3 &= E_2\end{aligned}$$

For $R_1 = R_3 = R_5 = 1$ ohm, $R_2 = R_4 = R_6 = 2$ ohm, $E_1 = 2$ V, and $E_2 = 3$ V, the loop currents i_1, i_2, and i_3 are determined from the following script:

```
R1 = 1;   R3 = R1;   R5 = R1;
R2 = 2;   R4 = R2;   R6 = R2;
E1 = 2;   E2 = 3;
R = [R1+R2+R3, -R2, -R3; ...
      -R2, R2+R4+R6, -R5; ...
      -R3, -R5, R3+R5+R6];
E = [-E1 0 E2]';
Curr = R\E
```

The execution of this script gives

```
Curr =
  -0.3333
  0.0000
  0.6667
```

Therefore, $i_1 = 1/3$ A, $i_2 = 0$ A, and $i_3 = 2/3$ A.

Example 2.23 Static deflection of a clamped square plate

In the determination of the solution to the static deflection of a square plate clamped on all four edges and subjected to a uniform loading over its surface, one must first obtain the constants E_m from the truncation of the following infinite set of equations:[14]

$$a_iE_i + \sum_{m=1,3,\ldots} b_{im}E_m = c_i \qquad i = 1, 3, 5, \ldots$$

[14] S. Timoshenko and S. Woinowsky-Krieger, *Theory of Plates and Shells*, McGraw-Hill, New York, 1959, pp. 197–202.

where

$$a_i = \frac{1}{i}\left(\tanh\alpha_i + \frac{\alpha_i}{\cosh^2\alpha_i}\right)$$

$$b_{im} = 8i\left(\pi m^3\left(1 + \frac{i^2}{m^2}\right)^2\right)^{-1}$$

$$c_i = \frac{4}{\pi^3 i^4}\left(\frac{\alpha_i}{\cosh^2\alpha_i} - \tanh\alpha_i\right)$$

and $\alpha_i = i\pi/2$. If we take only the first four terms of this system of equations, then we have the following set of equations in matrix notation:

$$\begin{bmatrix} a_1 + b_{11} & b_{13} & b_{15} & b_{17} \\ b_{31} & a_3 + b_{33} & b_{35} & b_{37} \\ b_{51} & b_{53} & a_5 + b_{55} & b_{57} \\ b_{71} & b_{73} & b_{75} & a_7 + b_{77} \end{bmatrix} \begin{bmatrix} E_1 \\ E_3 \\ E_5 \\ E_7 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_3 \\ c_5 \\ c_7 \end{bmatrix}$$

We see that the coefficients b_{im} are a function of the indices i and m, which are each vectors of length 4. Thus, we convert these vectors into two (4×4) matrices using `meshgrid` so that we can use dot operations.

The solution of these four equations is obtained with the following script:

```
m = 1:2:7;    i = m;                                                % (1×4)
alp = m*pi/2;                                                       % (1×4)
ai = (tanh(alp)+alp./cosh(alp).^2)./i;                              % (1×4)
ci = 4.*(alp./(cosh(alp).^2)-tanh(alp))./((pi^3)*i.^4);             % (1×4)
[ii, mm] = meshgrid(i, m);                                          % (4×4)
bim = (8/pi)*ii./(((1+(ii.^2)./(mm.^2)).^2).*mm.^3);                % (4×4)
ee = (diag(ai)+bim)\ci'
```

The execution of the script yields

```
ee =
  -0.0480
   0.0049
   0.0023
   0.0011
```

Thus, $E_1 = \text{ee}(1,1) = -0.0480$, $E_3 = \text{ee}(2,1) = 0.0049$, $E_5 = \text{ee}(3,1) = 0.0023$, and $E_7 = \text{ee}(4,1) = 0.0011$.

Example 2.24 Symbolically obtained Euler transformation matrix

Consider two coordinate systems: one an xyz Cartesian frame with origin O and another an $x'y'z'$ Cartesian frame with origin O. If the yz-plane rotates an angle β about the $O-x$-axis, then the y'- and z'-axes are related to the y- and z-axes through the rotation matrix

$$[R_x(\beta)] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\beta & \sin\beta \\ 0 & -\sin\beta & \cos\beta \end{bmatrix}$$

For a rotation β about $O-z$-axis, the rotation matrix is

$$[R_z(\beta)] = \begin{bmatrix} \cos\beta & \sin\beta & 0 \\ -\sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Finally, for a rotation β about $O-y$-axis, the rotation matrix is

$$[R_y(\beta)] = \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix}$$

The Euler angle sequence consists of three consecutive rotations ψ, θ, and φ defined as follows. We rotate about the $O-z$-axis an angle ψ and denote the new location of the x-axis as u. We then rotate about the $O-u$-axis an angle θ and denote the new location of the z-axis as w. Finally, we rotate an angle φ about the $O-w$-axis. Thus, the resulting Euler angle sequence is

$$[E(\psi,\theta,\varphi)] = [R_z(\psi)][R_{x=u}(\theta)][R_{z=w}(\varphi)]$$

The resulting Euler angle transformation matrix in symbolic form is determined from the following script:

```
syms ppsi th phi
Rz = [cos(th), sin(th), 0; -sin(th), cos(th), 0; 0, 0, 1];
Ru = [1, 0, 0; 0, cos(ppsi), sin(ppsi); 0, -sin(ppsi), cos(ppsi)];
Rw = [cos(phi), sin(phi), 0; -sin(phi), cos(phi), 0; 0, 0, 1];
E = Rz*Ru*Rw
```

Upon execution, we obtain the symbolic result

```
E =
  [cos(phi)*cos(th) - cos(ppsi)*sin(phi)*sin(th), cos(th)*sin(phi) + cos(phi)*cos(ppsi)*sin(th),
  sin(ppsi)*sin(th)]
  [-cos(phi)*sin(th) - cos(ppsi)*cos(th)*sin(phi), cos(phi)*cos(ppsi)*cos(th) - sin(phi)*sin(th),
  cos(th)*sin(ppsi)]
  [sin(phi)*sin(ppsi), -cos(phi)*sin(ppsi), cos(ppsi)]
```

where $th = \theta$, $ppsi = \psi$, and $phi = \varphi$.

2.7 SUMMARY OF FUNCTIONS INTRODUCED IN CHAPTER 2

A summary of the functions introduced in this chapter and their descriptions are presented in Table 2.1. See also Table 1.11 for a summary of the special symbols used in this chapter: colon, backslash, apostrophe, semicolon, period (dot), parentheses, brackets, and comma.

TABLE 2.1 MATLAB Functions Introduced in Chapter 2

MATLAB function	Description
`cumsum`	Cumulative sum of an array
`det`	Determinant of a square matrix (numerically and symbolically)
`diag`	Diagonal of a square matrix; creates a diagonal matrix
`dot`	Dot product of two vectors
`eig`	Eigenvalues and eigenvectors of special matrix equations
`end`	Last index in an array (See also Table 4.2)
`eye`	Creates the identity matrix
`find`	Finds indices of a vector satisfying a logical expression
`fliplr`	Flips elements of an array from left to right
`flipud`	Flips elements of an array from bottom to top
`inv`	Inverse of a square matrix (numerically and symbolically)
`length`	Length of a vector
`linsolve`	Solves a linear system of equations
`linspace`	Creates equally spaced elements of a vector
`logspace`	Creates equally spaced elements of a vector on a $\log_{10}$ scale
`magic`	Creates a square matrix whose sum of each row, each column, and diagonals is equal
`max`	Determines the maximum value in an array
`mesh`	Creates surface plot
`meshgrid`	Transforms two different vectors into arrays of the same size
`min`	Determines the minimum value in an array
`ones`	Creates an array whose elements equal 1
`plot`	Plots curves in a plane using linear axes
`rank`	Estimates number of linearly independent rows or columns of a matrix
`repmat`	Replicates arrays
`size`	Order (size) of an array
`sort`	Sorts elements of an array in ascending order
`sum`	Sums elements of an array
`zeros`	Creates an array whose elements equal 0

EXERCISES

Section 2.3

2.1 Create two vectors, one whose elements are $a_n = 2n - 1$ and the other whose elements are $b_n = 2n + 1, n = 0, 1, \ldots, 7$. Determine the following: (a) $a + b$, (b) $a - b$, (c) $a'b$, (d) determinant of $a'b$, and (e) ab'.

2.2 Given the vector $x = [17\ -3\ -47\ 5\ 29\ -37\ 51\ -7\ 19]$. Create a script that rearranges them into the following vector: $y = [-3\ -7\ -37\ -47\ 51\ 29\ 19\ 17\ 5]$. The script should be written to work on a vector of arbitrary length. Place the value 0 (for the general vector) with the negative quantities, that is, when 0 is the value of an element of the vector, it will be the first element of y.

2.3 Given the vector $y = [0, -0.2, 0.4, -0.6, 0.8, -1.0, -1.2, -1.4, 1.6]$. If $z = \sin(y)$, then:

a. Determine the minimum and maximum of only the negative values of z.

b. Determine the square root of only the positive values of z.

2.4 **a.** Create a vector of eight values that are equally spaced on a logarithm scale. The first value of the vector is 6 and the last value is 106.

b. Display the value of the fifth element of the vector created in (a).

c. Create a new vector whose elements are the first, third, fifth, and seventh elements of the vector created in (a).

Section 2.4

2.5 Let z = `magic`(5).

a. Perform the following operations to z in the order given:

i. Divide column 2 by $\sqrt{3}$.

ii. Add the elements of the third row to the elements in the fifth row (the third row remains unchanged).

iii. Multiply the elements of the first column by the corresponding elements of the fourth column and place the result in the first column.

iv. Set the diagonal elements to 2.

b. If the result obtained in (a) (iv) is denoted q, then obtain the diagonal of qq' (Answer: [486 104189 7300 44522 111024]').

c. Determine the maximum and minimum values of q.

2.6 Let w = `magic`(2) and w' be the transpose of w.

a. Using `repmat` create the following (4 × 4) matrix

$$\begin{bmatrix} w & w \\ w & w \end{bmatrix}$$

b. Using `repmat` create the following (6 × 2) matrix

$$\begin{bmatrix} w \\ w \\ w \end{bmatrix}$$

c. Use `repmat` and the column augmentation procedure to create the following (6 × 4) matrix:

$$\begin{bmatrix} w & w' \\ w & w' \\ w & w' \end{bmatrix}$$

d. Repeat (a), (b), and (c) without using `repmat`, that is, using only column and row augmentation procedures.

2.7 Let $x = \texttt{magic}(3)$. Using element swapping technique of Example 2.3

a. Create a new matrix in which each row of x has been moved up by one row and the first row becomes the last row.

b. Create a new matrix in which each column of x has been moved to the right and the last column becomes the first column.

2.8 Manipulate the output of `magic`(5) to produce the following altered matrix:

$$\text{magic}(5) = \begin{bmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{bmatrix} \rightarrow \begin{bmatrix} 17 & 24 & 1 & 8 & 0 \\ 23 & 5 & 7 & 0 & 16 \\ 4 & 6 & 0 & 20 & 22 \\ 10 & 0 & 19 & 21 & 3 \\ 0 & 18 & 25 & 2 & 9 \end{bmatrix}$$

Section 2.5

2.9 The displacement of the slider of the slider–crank mechanism shown in Figure 2.7 is given by

$$s = a\cos(\varphi) + \sqrt{b^2 - (a\sin(\varphi) - e)^2}$$

Plot the displacement s as a function of the angle φ (in degrees) when $a = 1$, $b = 1.5$, $e = 0.3$, and $0 \le \varphi \le 360°$, that is, use $\texttt{plot}(\varphi, s)$.

2.10 The percentage of the total power P in a periodic series of rectangular-shaped pulses as a function of the number of terms N_H in its series expansion is

$$P = 100P_o/P_T\%$$

where P_T is the total nondimensional power in the signal

$$P_o = 1 + 2\sum_{n=1}^{N_H} \frac{\sin^2(n\pi\tau_o/T)}{(n\pi\tau_o/T)^2}$$

and τ_o/T is the ratio of the pulse duration to its period. If we let $\tau_o/T = 1/\sqrt{19}$, then $P_T \cong 4.3589$. For this case, plot the percentage total power as a function of N_H for $2 \le N_H \le 25$, that is, use $\texttt{plot}(N_H, P_o)$.

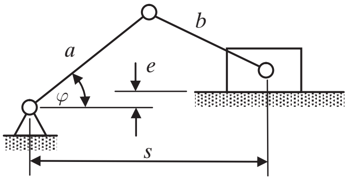


Figure 2.7 Slider–crank mechanism.

2.11 Consider the following product[15]

$$S_N = \prod_{n=1}^{N}\left(1-\frac{x^2}{n^2-a^2}\right)$$

where when $N \to \infty$,

$$S_\infty = \frac{a}{\sqrt{a^2+x^2}}\,\frac{\sin\left(\pi\sqrt{a^2+x^2}\right)}{\sin(\pi a)}$$

The percentage error between S_N and S_∞ is defined as

$$e_N = 100\,\frac{S_N - S_\infty}{S_\infty}\,\%$$

If x varies from 1 to 5 in increments of 0.5 and $a = \sqrt{2.8}$, then what is the percentage error at these nine values of x when $N = 100$. Use the function `prod` to obtain S_N (Answer: e_{100} = [1.0001 2.2643 4.0610 6.4176 9.3707 12.9670 17.2642 22.3330 28.2588]).

2.12 One method of finding an estimate of a parameter δ appearing in the Weibull probability density function (see Section 8.2.2) is obtained from

$$\delta = \left[\frac{1}{n}\sum_{i=1}^{n} x_i^{\beta}\right]^{1/\beta}$$

where x_i are obtained from a random sample of size n and β is a known parameter. If $x = [72, 82, 97, 103, 113, 117, 126, 127, 127, 139, 154, 159, 199, 207]$ and $\beta = 3.644$, then determine the value of δ.

2.13 The transformation from spherical to Cartesian coordinates is given by

$$x = b\sin\phi\cos\theta$$
$$y = b\sin\phi\sin\theta$$
$$z = b\cos\phi$$

Take ten equally spaced values of ϕ in the range $0 \le \phi \le 90°$ and twenty-four equally spaced values of θ in the range $0 \le \theta \le 360°$ and plot the hemisphere using `mesh` (x,y,z) when $b = 2$.

2.14 Evaluate the following series for $N = 25$ and for five equally spaced values of x from $0.1 \le x \le 1$. Compare these values with the exact values ($N \to \infty$).

$$S_N = \sum_{n=1}^{N}\frac{1}{n^4+x^4}$$

The exact value is

$$S_\infty = \frac{2\pi^4}{y^3}\,\frac{\sinh y + \sin y}{\cosh y - \cos y} \qquad \text{where } y = \pi x\sqrt{2}$$

2.15 The series representation for the Bessel function of the first kind of order n is given by

$$J_n(x) = \sum_{k=0}^{K\to\infty}\frac{(-1)^k(x/2)^{2k+n}}{k!\,\Gamma(k+1+n)}$$

[15] Jolley, *Summation of Series*.

For $K = 25$, determine a vector of values of $J_n(x)$ for six equally spaced values of x in the range $1 \le x \le 6$ when $n = 2$. The gamma function Γ is obtained with `gamma`. Compare your answers with those obtained from MATLAB's built-in function `besselj`.

2.16 Show numerically that the following series[16] sums to the value indicated when $n = 7$:

$$\sum_{k=1}^{2n-1} \cos(k\pi/n) = -1$$

2.17 Verify the following identities[17] for $n = 6$:

$$\sum_{k=1}^{n} k^2 = \frac{n}{6}(2n^2 + 3n + 1)$$

$$\sum_{k=1}^{n} k^4 = \frac{n}{30}(6n^4 + 15n^3 + 10n^2 - 1)$$

$$\sum_{k=1}^{n} k^5 = \frac{n^2}{12}(2n^4 + 6n^3 + 5n^2 - 1)$$

2.18 Verify the following formula[18] for the evaluation of π for $N = 10$:

$$\pi = \sum_{n=0}^{N\to\infty} \left(\frac{4}{8n + 1} - \frac{2}{8n + 4} - \frac{1}{8n + 5} - \frac{1}{8n + 6}\right)\left(\frac{1}{16}\right)^n$$

2.19 The answer to the question "How far can a stack of n identical books protrude over the end a table without the stack falling over?" is[19]

$$d_n = \frac{1}{2}\sum_{k=1}^{n} \frac{1}{k}$$

where d_n is the distance from the edge of the table to the outside edge of the top book. Use `cumsum` and `find` to show that one needs four stacked books for the overhang to slightly equal more than one book length (i.e., $d_4 = 1.0417$) and thirty-one books to equal slightly more than two book lengths. How many books are needed to attain three book lengths overhang? Let $n = 300$.

2.20 Verify the following relations[20] for $N = 10$.

$$e = \sum_{k=0}^{N\to\infty} \frac{1}{k!}$$

$$J_0(2) = \sum_{k=0}^{N\to\infty} \frac{(-1)^k}{(k!)^2}$$

$$\cos(1) = \sum_{k=0}^{N\to\infty} \frac{(-1)^k}{(2k)!}$$

[16] *Ibid*, pp. 86–87.
[17] Weisstein, *CRC Concise Encyclopedia,* p. 2344.
[18] *Ibid*, p. 155.
[19] *Ibid*, p. 262.
[20] *Ibid*, p. 1009.

2.21 Given[21]

$$S_K = \sum_{k=1,3,5}^{K} \frac{1}{k} e^{-kx} \sin ky$$

When $K = \infty$,

$$S_\infty = \frac{1}{2} \tan^{-1}\left(\frac{\sin y}{\sinh x}\right) \qquad x > 0$$

Verify that for $K = 41, x = 0.75$ and $y = \pi/3, S_{\infty}. \cong S_{41}$.

2.22 Given the identity

$$a^x = \sum_{k=0}^{K\to\infty} \frac{(x \ln a)^k}{k!} \qquad a > 0, \qquad -\infty < x < \infty$$

For $K = 14$, show that this relation produces seven-digit agreement with the exact value for 3^{-2}.

2.23 If

$$S_N = \sum_{n=1}^{N} \frac{1}{n^2 + z^2}$$

then,

$$S_\infty = \frac{\pi}{2z} \coth \pi z - \frac{1}{2z^2}$$

Determine for $z = 10$ the value of N for which $|S_{\infty}. - S_N| < 3 \times 10^{-3}$.

Section 2.6.2

2.24 A matrix is an orthogonal matrix if

$$X'X = I$$

and, therefore $(X'X)^{-1} = I$. Show that each of the following matrices is orthogonal.

$$w = \frac{1}{2}\begin{bmatrix} -1 & -1 \\ 1 & -1 \\ -1 & 1 \\ 1 & 1 \end{bmatrix} \qquad q = \frac{1}{2}\begin{bmatrix} 1 & -1 & -1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

2.25 Given the following matrix:

$$A = \begin{bmatrix} 2 & -2 & -4 \\ -1 & 3 & 4 \\ 1 & -2 & -3 \end{bmatrix}$$

Show that $A^2 - A = 0$.

[21] *Ibid*, p. 1483.

2.26 If

$$A = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \end{bmatrix}$$

Show that $A^2 - 4A - 5I = 0$, where I is the identity matrix.

2.27 If

$$A = \begin{bmatrix} 1 & -1 \\ 2 & -1 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1 & 1 \\ 4 & -1 \end{bmatrix}$$

then show that $(A + B)^2 = A^2 + B^2$.

2.28 If

$$A = \begin{bmatrix} 7 & -2 & 1 \\ -2 & 10 & -2 \\ 1 & -2 & 7 \end{bmatrix} \quad \text{and} \quad P = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{3} & 1/\sqrt{6} \\ 0 & 1/\sqrt{3} & -2/\sqrt{6} \\ -1/\sqrt{2} & 1/\sqrt{3} & 1/\sqrt{6} \end{bmatrix}$$

then show that $P^{-1}AP = \texttt{eig}(A)$.

2.29 Consider the planar three degree-of-freedom linkages shown in Figure 2.8. The location and orientation of point O_3 with respect to the fixed coordinate system O_0 is

$$T_3 = A_1 A_2 A_3$$

where

$$A_j = \begin{bmatrix} \cos\theta_j & -\sin\theta_j & 0 & a_j \cos\theta_j \\ \sin\theta_j & \cos\theta_j & 0 & a_j \sin\theta_j \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad j = 1, 2, 3$$

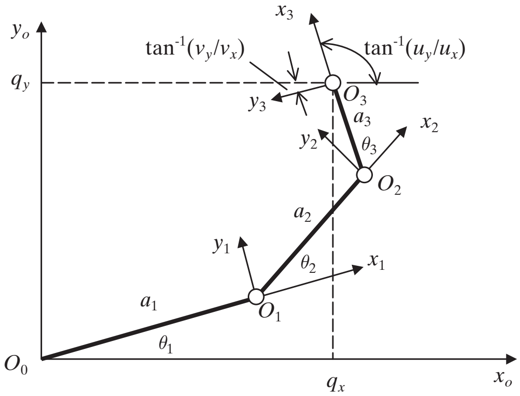


Figure 2.8 Planar three degree-of-freedom linkages.

and

$$T_3 = \begin{bmatrix} u_x & v_x & 0 & q_x \\ u_y & v_y & 0 & q_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The components q_x and q_y are the (x, y)-coordinates of the point O_3 with respect to the coordinate system centered at O_0. If $\theta_j = 30°$, $j = 1, 2, 3$, and $a_1 = 1$, $a_2 = 2$, and $a_3 = 3$, then what is the location of point O_3 with respect to the coordinate system centered at O_0 and the orientation of the (x_3, y_3) axes system (Answer: $q_x = 1.8660, q_y = 5.2321, x_3$ is parallel to y_0 and y_3 is parallel to x_0, but in the opposite direction).

2.30 The coordinate transformation matrices that relate the rotation and axis offset of each component of a robot arm are given by[22]

$$T_{12} = \begin{bmatrix} \cos\theta_1 & 0 & \sin\theta_1 & 0 \\ \sin\theta_1 & 0 & -\cos\theta_1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad T_{23} = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & a_2\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & a_2\sin\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{34} = \begin{bmatrix} \cos\theta_3 & 0 & \sin\theta_3 & a_3\cos\theta_3 \\ \sin\theta_3 & 0 & -\cos\theta_2 & a_3\sin\theta_3 \\ 0 & 1 & 0 & r_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad T_{45} = \begin{bmatrix} \cos\theta_4 & 0 & \sin\theta_4 & 0 \\ \sin\theta_4 & 0 & -\cos\theta_4 & 0 \\ 0 & 1 & 0 & r_4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{56} = \begin{bmatrix} \cos\theta_5 & 0 & \sin\theta_5 & 0 \\ \sin\theta_5 & 0 & -\cos\theta_5 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad T_{67} = \begin{bmatrix} \cos\theta_6 & 0 & \sin\theta_6 & 0 \\ \sin\theta_6 & 0 & -\cos\theta_6 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The position of a point in the seventh coordinate system with respect to the fixed coordinate system is given by

$$T_{17} = T_{12}\,T_{23}\,T_{34}\,T_{45}\,T_{56}\,T_{67}$$

Determine T_{17} when $a_2 = r_4 = 0.431\,\text{m}, a_3 = 0.019\,\text{m}, r_3 = 0.125\,\text{m}, \theta_1 = 15°, \theta_2 = 105°, \theta_3 = 145°, \theta_4 = 0°, \theta_5 = 215°$, and $\theta_6 = 55°$.

2.31 In multiple linear regression analysis, the following matrix quantity has some utility (see Exercise 8.12):

$$H = X\,(X'X)^{-1}\,X'$$

[22] S. M. Megahed, *Principles of Robot Modeling and Simulation,* John Wiley & Sons, New York, NY, 1993, pp. 69–70.

If

$$X = \begin{bmatrix} 17 & 31 & 5 \\ 6 & 5 & 4 \\ 19 & 28 & 9 \\ 12 & 11 & 10 \end{bmatrix}$$

determine the diagonal of H (Answer: diagonal $H = [0.7294\ \ 0.9041\ \ 0.4477\ \ 0.9188]'$).

2.32 Plot the Fourier series[23] given below for 250 values of τ over its indicated range using `plot`$(\tau, f(\tau))$. Obtain the solutions by using the vector multiplication procedures given for Case 3 in Section 2.6.2.

a. *Square wave*

$$f(\tau) = \frac{4}{\pi}\sum_{n=1,3,5,\ldots}^{401}\frac{1}{n}\sin(2n\pi\tau) \qquad -\frac{1}{2} \le \tau \le \frac{1}{2}$$

b. *Sawtooth*

$$f(\tau) = \frac{1}{2} + \frac{1}{\pi}\sum_{n=1}^{200}\frac{1}{n}\sin(2n\pi\tau) \qquad -1 \le \tau \le 1$$

c. *Sawtooth*

$$f(\tau) = \frac{1}{2} - \frac{1}{\pi}\sum_{n=1}^{200}\frac{1}{n}\sin(2n\pi\tau) \qquad -1 \le \tau \le 1$$

d. *Triangular wave*

$$f(\tau) = \frac{\pi}{2} - \frac{4}{\pi}\sum_{n=1}^{200}\frac{1}{(2n-1)^2}\cos((2n-1)\pi\tau) \qquad -1 \le \tau \le 1$$

e. *Rectified sine wave*

$$f(\tau) = \frac{2}{\pi} + \frac{4}{\pi}\sum_{n=1}^{200}\frac{1}{1-4n^2}\cos(2n\pi\tau) \qquad -1 \le \tau \le 1$$

f. *Half sine wave*

$$f(\tau) = \frac{1}{\pi} + \frac{1}{2}\sin\pi\tau - \frac{2}{\pi}\sum_{n=2,4,6,\ldots}^{106}\frac{\cos n\pi\tau}{n^2-1} \qquad -2 \le \tau \le 2$$

[23] Hsu, *Applied Fourier Analysis.*

g. *Exponential*

$$f(\tau) = \frac{e^{2\pi} - 1}{\pi}\left[\frac{1}{2} + \sum_{n=1}^{250} \frac{1}{1 + n^2}(\cos n\tau - n\sin n\tau)\right] \qquad 0 \le \tau \le 4\pi$$

Use 350 values of τ to display the results.

h. *Trapezoidal*

$$f(\tau) = \frac{4}{\alpha^2} \sum_{n=1,3,5,\ldots}^{105} \frac{\sin n\pi\alpha}{(\pi n)^2} \sin n\pi\tau \qquad -2 \le \tau \le 2$$

Let $\alpha = 0.25$.

2.33 Consider the following two series:[24]

$$S_{1N} = \sum_{n=1}^{N} \frac{\cos(n\theta)}{n^2 + a^2} \qquad 0 < \theta < \pi$$

$$S_{2N} = \sum_{n=1}^{N} \frac{n\sin(n\theta)}{n^2 + a^2} \qquad 0 < \theta < 2\pi$$

where when $N \rightarrow \infty$,

$$S_{1\infty} = \frac{\pi \cosh(a(\pi-\theta))}{2a \sinh \pi a} - \frac{1}{2a^2} \qquad 0 < \theta < \pi$$

$$S_{2\infty} = \frac{\pi \sinh(a(\pi-\theta))}{2 \sinh \pi a} \qquad 0 < \theta < 2\pi$$

The percentage error between S_{jN} and $S_{j\infty}$ is defined as

$$e_j = 100\frac{S_{jN}-S_{j\infty}}{S_{j\infty}}\% \qquad j = 1, 2$$

If θ varies from 10° to 80° in increments of 10° and $a = \sqrt{3}$, then determine the percentage error for the two series at the eight values of θ when $N = 25$. Obtain the solutions by using the vector multiplication procedures given for Case 3 in Section 2.6.2 (Answer: $e_1 = [-1.2435\ 0.8565\ 0.8728\ -1.9417\ -0.9579\ -8.1206\ 0.7239\ 1.1661]$ and $e_2 = [8.0538\ 10.4192\ -8.9135\ -5.4994\ 12.9734\ -0.5090\ -17.2259\ 11.2961]$).

2.34 The nondimensional steady-state temperature distribution in a rectangular plate that is subjected to a constant temperature along the edge $\eta = 1$ is given by[25]

$$T(\eta, \xi) = \frac{4}{\pi} \sum_{n=1,3,5}^{\infty} \frac{\sinh(n\pi\alpha\eta)}{n \sinh(n\pi\alpha)} \sin(n\pi\xi)$$

where $\eta = x/d$, $\xi = y/b$, d and b are the lengths of the plate in the x- and y-directions, respectively, $\alpha = d/b$, $0 \le \eta \le 1$, and $0 \le \xi \le 1$. Display the temperature distribution throughout the plate when $\alpha = 2$ using `mesh`(ξ, η, $T(\eta,\xi)$). Let $\Delta\eta = \Delta\xi = 1/14$. Obtain the solutions using the vector multiplication procedures given for Case 2 in Section 2.6.2.

[24] Jolley, *Summation of Series*.

[25] Hsu, *Applied Fourier Analysis*.

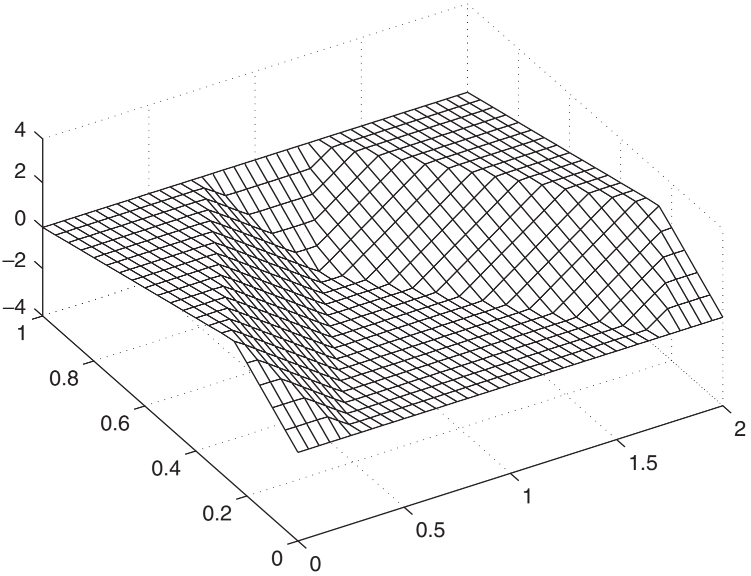


Figure 2.9 Propagation of an initially displaced string.

2.35 The displacement of a wave propagating in a string subject to an initial velocity of zero and an initial displacement of

$$u(\eta,0) = \frac{\eta}{a} \qquad 0 \le \eta \le a$$

$$u(\eta,0) = \frac{1-\eta}{1-a} \qquad a \le \eta \le 1$$

is given by

$$u(\eta,\tau) = \frac{2}{a\pi(1-a)} \sum_{n=1}^{N\to\infty} \frac{\sin n\pi a}{n^2} \sin(n\pi\eta)\cos(n\pi\tau)$$

Using mesh(τ, η, $u(\eta,\tau)$), display $u(\eta,\tau)$ over the range $0 \le \tau \le 2$ and $0 \le \eta \le 1$ when $N = 50$ and $a = 0.25$. In addition, let $\Delta\eta = 0.05$ and $\Delta\tau = 0.05$. Obtain the solutions using the vector multiplication procedures given for Case 2 in Section 2.6.2. The result is shown in Figure 2.9, which was obtained after using the rotate icon in the figure window.

Section 2.6.3

2.36 Given the two matrices

$$A = \begin{bmatrix} 1 & 3 & 4 \\ 31 & 67 & 9 \\ 7 & 5 & 9 \end{bmatrix} \qquad \text{and} \qquad B = \begin{bmatrix} 11 & 34 & 6 \\ 7 & 13 & 8 \\ 43 & 10 & 53 \end{bmatrix}$$

Show numerically that $|AB| = |A||B|$.

2.37 The equation of a parabola can be determined for the three noncollinear points (x_1, y_1), (x_2, y_2), and (x_3, y_3) from[26]

$$\det\begin{vmatrix} x^2 & x & y & 1 \\ x_1^2 & x_1 & y_1 & 1 \\ x_2^2 & x_2 & y_2 & 1 \\ x_3^2 & x_3 & y_3 & 1 \end{vmatrix} = 0$$

Use the Symbolic toolbox to determine the equation of a parabola that passes through the points $(x_1, y_1) = (-1, 1)$, $(x_2, y_2) = (1, 1)$, and $(x_3, y_3) = (2, 2)$.

2.38 The equation of a circle can be determined for the three noncollinear points (x_1, y_1), (x_2, y_2) and (x_3, y_3) from[27]

$$\det\begin{vmatrix} x^2 + y^2 & x & y & 1 \\ x_1^2 + y_1^2 & x_1 & y_1 & 1 \\ x_2^2 + y_2^2 & x_2 & y_2 & 1 \\ x_3^2 + y_2^2 & x_2 & y_3 & 1 \end{vmatrix} = 0$$

Use the Symbolic toolbox to determine the equation of a circle that passes through the points $(x_1, y_1) = (-1, 1)$, $(x_2, y_2) = (0, 0)$, and $(x_3, y_3) = (1, 1)$.

Section 2.6.5

2.39 Given the following system of equations

$$\begin{aligned} 16s + 32u + 33p + 13w &= 91 \\ 5s + 11u + 10p + 8w &= 16 \\ 9s + 7u + 6p + 12w &= 5 \\ 34s + 14u + 15p + w &= 43 \end{aligned}$$

determine the values of s, u, p, and w, the value of the determinant, and the inverse of the coefficients of s, u, p, and w (Answer: $s = -0.1258$, $u = -8.7133$, $p = 11.2875$, and $w = -0.0500$. Determinant $= 7{,}680$).

2.40 Consider two long cylinders of two different materials where one cylinder just fits inside the other. The inner radius of the inner cylinder is a, and its outer radius is b. The inner radius of the outer cylinder is also b, and its outer radius is c. The Young's modulus and Poisson ratio of the inner cylinder are E_1 and ν_1, respectively, and those of the outer cylinder are E_2 and ν_2, respectively. The radial stress σ_{rr}, hoop stress $\sigma_{\theta\theta}$, and radial displacement u_r are given, respectively, by,

$$\begin{aligned} \sigma_{rri}(r) &= \frac{A_i}{r^2} + B_i \\ \sigma_{\theta\theta i}(r) &= \frac{-A_i}{r^2} + B_i \qquad i = 1, 2 \\ u_{ri}(r) &= \frac{-(1 + \upsilon_i)}{rE_i} A_i + \frac{(1-\upsilon_i)}{E_i} rB_i \end{aligned} \tag{a}$$

where $i = 1$ refers to the inner cylinder and $i = 2$ to the outer cylinder.

[26] Szabo, *Linear Algebra*, p. 265.

[27] *Ibid.*, p. 266.

If the outer surface of the outer cylinder is subjected to a compressive radial displacement U_o and the inner surface of the inner cylinder has no radial stress, then the following four boundary conditions can be used to determine A_i and $B_i, i = 1, 2$:

$$\begin{aligned} \sigma_{rr1}(a) &= 0 \\ \sigma_{rr1}(b) &= \sigma_{rr2}(b) \\ u_{r1}(b) &= u_{r2}(b) \\ u_{r2}(c) &= -U_o \end{aligned} \tag{b}$$

Substituting Eqs. (a) into (b), the following system of equations in matrix form is obtained:

$$\begin{bmatrix} 1 & a^2 & 0 & 0 \\ 1 & b^2 & -1 & -b^2 \\ -(1+\nu_1) & (1-\nu_1)b^2 & (1+\nu_2)E_1/E_2 & -(1-\nu_2)b^2E_1/E_2 \\ 0 & 0 & -(1+\nu_2) & (1-\nu_2)c^2 \end{bmatrix} \begin{Bmatrix} A_1 \\ B_1 \\ A_2 \\ B_2 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ -U_oE_2c \end{Bmatrix}$$

Determine the hoop stress in the inner and outer cylinders at $r = b$ when $\nu_1 = \nu_2 = 0.4, E_1 = 2.1 \times 10^9$ N/m^2, $E_2 = 0.21 \times 10^9$ N/m^2, $U_o = 0.25$ mm, $a = 5$ mm, $b = 6.4$ mm, and $c = 8$ mm (Answer: $\sigma_{\theta\theta1}(b) = -6.301 \times 10^7$ N/m^2 and $\sigma_{\theta\theta2}(b) = -1.179 \times 10^7$ N/m^2)

2.41 For the electric circuit shown in Figure 2.10, the governing equations for the three loop currents are given by

$$\begin{aligned} V_1 - 6Ri_1 + 4R(i_2 - i_1) &= 0 \\ V_2 + 2R(i_3 - i_2) - 3Ri_2 - 4R(i_2 - i_1) &= 0 \\ -V_3 - Ri_3 - 2R(i_3 - i_2) &= 0 \end{aligned}$$

Use `solve` from Symbolic toolbox to show that the three currents are given by

$$\begin{aligned} i_1 &= \frac{1}{182R}(12V_2 - 8V_3 + 23V_1) \\ i_2 &= \frac{1}{91R}(6V_1 + 15V_2 - 10V_3) \\ i_3 &= \frac{1}{91R}(4V_1 + 10V_2 - 37V_3) \end{aligned}$$

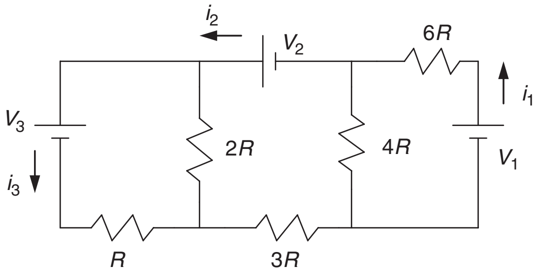


Figure 2.10 Electric circuit.

2.42 An ac electric circuit with resistors R and capacitances C is described by the following set of equations in the Laplace transformed domain:

$$\begin{bmatrix} 2s + \tau_o & -s & 0 \\ -s & 2s + \tau_o & -s \\ 0 & -s & s + \tau_o \end{bmatrix} \begin{Bmatrix} \overline{V}_1(s) \\ \overline{V}_2(s) \\ \overline{V}_3(s) \end{Bmatrix} = \begin{Bmatrix} s\overline{U}(s) \\ 0 \\ 0 \end{Bmatrix}$$

where s is the Laplace transform parameter, $\tau_o = 1/RC, \overline{V}_j(s),\ j = 1, 2, 3,$ are the transformed nodal voltages, and $\overline{U}_j(s)$ is the transformed applied voltage. Use the Symbolic toolbox to solve for $\overline{V}_j(s)$.

3

Data Input/Output

Edward B. Magrab

The means of displaying annotated numerical results in the MATLAB command window and the means of storing and retrieving data from files are presented.

3.1 STRINGS AND ANNOTATED OUTPUT

3.1.1 Creating Strings

Strings are collections of any combination of letters, numbers, and special characters. They are typically used for displaying information to the command window, for annotating data displayed to the command window, and for annotating graphs. They are created, stored, and manipulated in arrays and are defined in a manner similar to that for vectors and matrices. A string differs from an array of numerical values in that each character in the string occupies one element in the array and the string is defined by enclosing all its characters between a pair of single quotes (' . . . ').

Consider the following examples. Let *s* be the string 'testing123'. The MATLAB expression to define this string is the vector

```
s = 'testing123'
```

or

```
s = ['testing123']
```

where each character within the pair of single quotes is a location in the vector *s*. Thus, the length of the string is 10. To retrieve specific characters in the string *s*, we can use expressions like

```
s(7) → g
s(3:6) → stin
```

Strings can also be manipulated in a manner similar to numerical values. For example, consider the script

```
s = 'testing123'
fs = fliplr(s)
```

Its execution gives

```
fs =
321gnitset
```

Strings can also be concatenated (added to form a longer string) in a manner similar to that used for numerical values. Thus,

```
sc = ['testing123', 'testing123']
```

produces the (1 × 20) string

```
sc =
testing123testing123
```

whereas the script

```
scs = ['testing123'; 'testing123']
```

creates the (2 × 10) matrix

```
scs =
testing123
testing123
```

Thus,

```
scs(1,:) → testing123
scs(2,:) → testing123
```

Notice that both rows of *scs* have the same number of characters (columns).

In order to find the starting locations of strings within strings, one uses

```
findstr(string1, string2)
```

which searches the longer of the two strings for the occurrences of the shorter of the two strings. Let us find the occurrences of ‘123’ in the concatenated string shown in the script below.

```
sc = ['testing123', 'testing123']
Loc = findstr(sc, '123')
```

Upon execution, we obtain

```
sc =
testing123testing123
Loc =
  8   18
```

Thus, the first occurrence of the ‘123’ starts at location 8 and the second occurrence starts at location 18. If the string does not exist, then *Loc* would be a null vector; that is, [].

If we place a string in each row of a matrix, we have a convenient way in which to access string expressions. The requirement is that each row must contain the same number of characters. This requirement can be met by employing blanks to pad the rest of the string when the individual string expressions are of unequal length. Thus, if we have the expression

```
lab = ['first ';'last  ';'middle']
```

then

lab(1,:) → first*b*
lab(2,:) → last*bb*
lab(3,:) → middle

and *b* indicates a blank space. MATLAB provides a way to do this padding with the function

```
char
```

Thus, the above expression can be replaced by the easier-to-use expression

```
lab = char('first','last','middle')
ord = size(lab)
```

which when executed displays

```
lab =
first
last
middle
ord =
  3   6
```

where each string expression is a row in the matrix *lab* and *lab* is a (3 × 6) array. The trailing blanks are not visible in the display to the command window. The trailing blanks can be removed with

```
deblank
```

The leading and trailing blanks can be removed with

```
strtrim
```

Two strings can be compared by using

```
L = strcmp(A, B)
```

where A and B are strings and $L = 1$ (true) if $A = B$ and $L = 0$ (false) if $A \neq B$. This function is intended to compare character data.

We now illustrate strtrim and strcmp with the following script. In the two strings defined in the script, A has two additional leading blanks and two additional trailing blanks than string B.

```
A = '   Yes and No   ';
B = ' Yes and No ';
C1 = strcmp(A, B)
C2 = strcmp(strtrim(A), strtrim(B))
```

which upon execution gives

```
C1 =
  0
C2 =
  1
```

We see that comparison $C1$ is false (not identical) because blanks are legitimate characters. However, after the leading and trailing blanks are stripped from A and B, the strings are identical.

3.1.2 Converting Numerical Values to Strings and Displaying Them

To convert a numerical value to a string, we use

```
z = num2str(num, N)
```

where z is a string and *num* is either a number, an array of numbers, or an expression resulting in a number or an array of numbers. The quantity N is the number of digits to be displayed. If the number of digits specified is less than the number of digits to the left of the decimal place, then MATLAB converts the number to its exponential representation with the number of significant digits equal to N. Consider the following examples in which $num = 1000\pi = 3141.592653589$. Then, the various values of N will display the digits as shown below.

```
num2str(num, 1) → 3e+003
num2str(num, 3) → 3.14e+003
```

```
num2str(num, 4) → 3142
num2str(num, 5) → 3141.6
num2str(num, 8) → 3141.5927
```

Notice that the decimal point (.) does not count as a digit. On the other hand, when $num = \pi/1000 = 0.003141592653589$, we have

```
num2str(num, 1) → 0.003
num2str(num, 3) → 0.00314
num2str(num, 4) → 0.003142
num2str(num, 5) → 0.0031416
num2str(num, 8) → 0.0031415927
```

To convert an integer to a string, we use

```
z = int2str(num)
```

where *num* is an integer. If *num* is not an integer, then it is rounded so that it becomes an integer.

These two functions are most often used to place annotated numerical output in the MATLAB command window or on a figure. A typical way in which it is used is to concatenate the converted numerical value with some identifying text. Thus, if *num* is, say, the length in meters, and it is to be identified as such, then to display it to the MATLAB command window we use

```
disp(x)
```

where *x* is a number, vector, matrix, or string. To illustrate the use of `disp`, consider the following script:

```
num = 12.567;
disp(['Object length = ' num2str(num) ' m'])
```

At least one blank space on each side of `num2str(. . .)` is required. When executed, this script displays

```
Object length = 12.567 m
```

in the command window. Internally, this string is a vector of length 24. Notice that blank spaces are acceptable string characters and are preserved as such.

Let *num* be a vector of the lengths of the object. Then, the script that displays a vector of values is

```
num = [12.567, 3.458, 9.111];
disp([' Object length = ' num2str(num) ' m'])
```

Upon execution, we obtain

```
Object length = 12.567      3.458      9.111 m
```

However, to create annotation that accompanies each value of *num*, we use `repmat` as follows:

```
num = [12.567, 3.458, 9.111];
n = length(num);
disp([repmat('Object length = ', n, 1) num2str(num') repmat(' m', n, 1)])
```

which upon execution displays

```
Object length = 12.567 m
Object length = 3.458 m
Object length = 9.111 m
```

If one were to display *num* without annotation, then the script

```
num = [12.567, 3.458, 9.111];
disp(num)
```

displays in the MATLAB command window

```
12.5670      3.4580      9.1110
```

whereas

```
num = [12.567, 3.458, 9.111];
disp(num')
```

displays

```
12.5670
3.4580
9.1110
```

An alternative function that can be used to display formatted data to the MATLAB command window is `fprintf`, which has the advantage over `disp` in that it can better control the format of the numerical values. The syntax of the `fprintf` function to print to the command window is

```
fprintf(1, '%….', variables)
```

where the first argument, the ‘1’, indicates that the output is to be to the command window, and everything inside the quotes is the format specification pertaining to *variables*. When *variables* is a vector or a matrix, the format specification is cycled through the format specification on a column-by-column basis. The order of the format specifications corresponds to the order of the variables. The % symbol precedes each specific format specification, which is of the form

x.yq

where the quantity *q* specifies the format and is given in the *Help* file for `fprintf`. The quantity *x* is an integer that specifies the minimum number of digits to be displayed and the quantity y is the number of these digits that will appear to the right of the decimal point. We illustrate the use of fprintf by selecting $q = f$, which is a fixed point format, and using it to display the vector

```
num = [12, -14, 3098.458, 0.11167];
```

in several different ways.

To display this vector on one line using fprintf, we have the script

```
num = [12, -14, 3098.458, 0.11167];
fprintf(1, '%5.2f  ', num)
```

which results in

```
12.00  -14.00  3098.46  0.11
```

Notice that *num*(1) and *num*(2) had two zeros added to their representation, whereas *num*(4) was rounded to two digits after the decimal point. Each of the numbers has two blank spaces between them, which was obtained by leaving two blank spaces between the *f* and the apostrophe in the fprintf argument. To display these values as a column of four numbers, we use the line feed delimiter \n as follows:

```
num = [12 -14 3098.458 0.11167];
fprintf(1, '%5.2f\n', num)
```

which when executed displays

```
12.00
-14.00
3098.46
0.11
```

To reproduce the four numbers with the same digital precision as given, we include the format specification for each quantity. Thus, the script is

```
num = [12, -14, 3098.458, 0.11167];
fprintf(1, '%2.0f  %2.0f  %5.3f  %5.5f', num)
```

where we have placed two blank spaces between each letter *f* and the symbol % so that the numbers are separated by two spaces when displayed. The execution of this script gives

```
12  -14  3098.458  0.11167
```

To annotate each number, we use the following procedure[1]

```
num = [12, -14, 3098.458, 0.11167];
fprintf(1, 'weight = %2.0f kg pressure = %2.0f kPa time = %5.3f s length =
            %5.5f m', num)
```

[1] The fprintf statement cannot be broken up as shown. It has been presented in two lines because of page width restrictions.

Upon execution, we obtain

```
weight = 12 kg pressure = -14 kPa time = 3098.458 s length = 0.11167 m
```

To display the values in a column, the previous script is modified as[2]

```
num = [12, -14, 3098.458, 0.11167];
fprintf(1, 'weight = %2.0f kg\npressure = %2.0f kPa\ntime = %5.3f s\nlength =
          %5.5f m', num)
```

The execution of this script results in

```
weight = 12 kg
pressure = -14 kPa
time = 3098.458 s
length = 0.11167 m
```

If we are willing to have each number appear with the same format, then we can simplify the format specification and still generate annotated output, albeit in a somewhat less informative manner, as follows:

```
num = [12, -14, 3.458, 0.11167];
nn = 1:length(num);
fprintf(1, 'x(%1.0f) = %7.5f\n', [nn; num])
```

Upon execution, we obtain

```
x(1) = 12.00000
x(2) = -14.00000
x(3) = 3.45800
x(4) = 0.11167
```

The num2str function can also employ the format specifications of fprintf by replacing the second argument *N* in num2str with % followed by a format specification. For example, suppose that we want to display a very small number as 0 instead of in exponent form. If this number were $x = 0.00045$, then the script is

```
x = 0.00045;
disp(['x = ' num2str(x,'%2.1f')])
```

displays

```
x = 0.0
```

whereas

```
x = 0.00045;
disp(['x = ' num2str(x, 1)])
```

[2] The fprintf statement cannot be broken up as shown. It has been presented in two lines because of page-width restrictions.

displays

```
x = 0.0004
```

One can also format data and then convert the formatted data to a string. This operation is done with

```
sprintf
```

To illustrate the use of sprintf, consider the following script:

```
z = 100*magic(3)/17;
x = sprintf('%6.2f    %6.2f    %6.2f\n', z');
disp(z)
disp(' ')
disp(x)
```

Upon execution, we obtain

```
47.0588    5.8824   35.2941
17.6471   29.4118   41.1765
23.5294   52.9412   11.7647

47.06       5.88     35.29
17.65      29.41     41.18
23.53      52.94     11.76
```

It is pointed out that z is a matrix of numbers and x is a string of size (1×93).

3.2 ENTERING DATA WITH input

Arrays of data can be solicited by a script or function and then entered by the user by employing input. In addition, input can display to the MATLAB command window a message instructing the user what is to be entered. However, the actual form of the data depends on whether the data represent a scalar, vector, or matrix and whether these quantities are numbers or strings. We now illustrate these various cases. Other methods of data entry are discussed in Sections 3.3, 3.5, and 5.2.2.

3.2.1 Entering a Scalar with input

To input a single numerical quantity, we illustrate the use of input as follows:

```
InputData = input('Enter the temperature in degrees C: ');
```

Upon execution, the following is displayed in the command window:

```
Enter the temperature in degrees C: 121.7
```

where the number 121.7 was entered by the user. The semicolon at the end of the expression in the script suppresses the echoing of the value entered. The variable *InputData* has the value of 121.7.

One can also perform modification to user-entered values in the same expression. For example, one can convert degrees to radians as follows:

```
InputData = input('Enter the starting angle in degrees: ')*pi/180;
```

When executed, the following is displayed to the command window:

```
Enter the starting angle in degrees: 45
```

where the value 45 was entered by the user. However, the value of *InputData* is 0.7854 (= $45\pi/180$).

Now consider the conversion of temperature from °C to °F. The script is

```
InputData = 1.8*input('Enter the temperature in degrees C: ')+32;
```

which upon execution displays in the command window

```
Enter the temperature in degrees C: 100
```

where the value 100 was entered by the user. However, the value of *InputData* is 212.

A message may be printed on several lines by including within the message's quotation delimiters a \n at the appropriate places. Thus,

```
InputData = input('Enter the starting angle\nin degrees: ')*pi/180;
```

when executed displays

```
Enter the starting angle
in degrees:
```

Notice that in the input string, there is no space between the '\n' and the 'in'. This was done so that the two lines, when displayed, would be left justified.

3.2.2 Entering a String with `input`

To input a single string quantity, we append an 's' at the end of the `input` function. Thus,

```
InputData = input('Enter file name, including its extension: ', 's');
```

which displays in the MATLAB command window

```
Enter file name, including its extension: DataSet3.txt
```

where the string *DataSet3.txt* was entered by the user. Notice that no single quotation marks are required by the person entering the file name. The value of *InputData* is the string *DataSet3.txt*, which is a vector of length 12.

3.2.3 Entering a Vector with `input`

To input a vector of numerical values, we use

```
InputData = input('Enter four temperatures in degrees C: ');
```

which upon execution displays in the command window

```
Enter four temperatures in degrees C: [120, 141, 169, 201]
```

where the vector of numbers [120, 141, 169, 201] was entered by the user. The square brackets are required. If a column vector was required, then the user's response would be either [120, 141, 169, 201]' or [120; 141; 169; 201].

3.2.4 Entering a Matrix with `input`

To input a matrix of numerical values, we use the script

```
InputData = input('Enter three temperatures in degrees C\nfor levels 1 and 2: ');
```

which displays in the MATLAB command window

```
Enter the three temperatures in degrees C
for levels 1 and 2: [67, 35, 91;44, 51, 103]
```

where the array [67, 35, 91;44, 51, 103] was entered by the user. The variable *InputData* is a (2×3) array.

3.3 INPUT/OUTPUT DATA FILES

As shown in the previous sections, one method of entering data for execution by a script is to use `input`. The second means is to define data within a script using the methods discussed in Sections 2.3 and 2.4. These data creation statements can also appear in a function, which is discussed in Chapter 5. In fact, one can define a function such that it only contains data. See Section 5.2.2 for an example of this.

Another way to enter data is to place data in an ASCII[3] text file and use[4]

```
load(s)
```

where *s* is a string containing the file name. The `load` function reads data on a row-by-row basis, with each row separated by using *Enter* and with each data

[3] ASCII stands for American Standard Code for Information Interchange. It has come to signify plain text; that is, text with no formatting, and is used when one wants to easily exchange data and text between different computer programs. Usually, specifying a text file means an ASCII text file.

[4] `load` and `save` are faster than lower-level calls such as `fread` and are the file read/write combination that MATLAB recommends for most applications.

value separated by either one or more blanks or by a comma. The number of columns of data in each row must be the same. These requirements are analogous to those that must be followed when creating matrices. Here, the *Enter* key is used instead of the semicolon. When creating a row vector, one enters the data without using the *Enter* key. When creating a column vector, each data value is followed by using the *Enter* key.

Let us illustrate two ways of using `load`. For specificity, we shall assume that the data reside in the ASCII text file *DataSection33.txt* in the form

```
11  12  13
21  22  23
31  32  33
41  42  43
```

A useful feature of `load` is that the file name without the extension (the suffix '.txt') becomes the name of the variable whose vector or matrix elements are the data as they appear in the file. Thus, in the script, the variable named *DataSection33* is a (4 × 3) matrix of numbers, and it is used in the script as if a variable named *DataSection33* had been placed on the left side of an equal sign.

The `load` function is given by

```
load('DataSection33.txt')
```

It is assumed that the file resides in the current directory. If not, one uses the procedures described in Section 1.2.6. The function `load` is used when the filename is known at the time of the creation of a script and it will not change.

For an example, let us square each element of the matrix previously given. The script is

```
load ('DataSection33.txt')
y = DataSection33.^2
```

which upon execution results in

```
y =
    121    144    169
    441    484    529
    961   1024   1089
   1681   1764   1849
```

On the other hand, if one wants to operate on data in different files, each having a different file name, then one has to employ a different technique. Here, the user will enter the file name when requested to do so by the script or function and, as before, the script will square the data residing in the file whose name is specified when the script is executed. The script is

```
FileName1 = input('Enter file name containing data (including suffix): ', 's');
load(FileName1);
m = findstr(FileName1, '.');
```

```
data1 = eval(FileName1(1:m-1));
y = data1.^2
```

As discussed previously, the `findstr` function locates the position of the first occurrence in the string of characters within the apostrophes (' '), in this case the period (.), and brings back its value. We have used it here to limit the string of characters comprising *FileName1* to those up to, but not including, the period; hence, the string length of interest is *m*–1. Thus, we have stripped the suffix and the period from the file's name. Since the stripped version of *FileName1* is still unknown to the remaining expressions in the script, it must be converted to a numerical quantity. This is done by `eval`, which evaluates the string quantity appearing in its argument.

Upon execution of this script, we are first asked to provide the file name. We will use the file *DataSection33.txt*. Thus,

```
Enter file name containing data (including suffix): DataSection33.txt
```

is displayed in the command window and *DataSection33.txt* is the user's response. After hitting the *Enter* key, the program displays

```
y =
    121    144    169
    441    484    529
    961   1024   1089
   1681   1764   1849
```

which is what we obtained previously.

If one wants to save numerical values resulting from the execution of a script or function to a file, then we use

```
save('File name', 'Variable 1', 'Variable 2', ..., '-ascii')
```

where '*File name*' is a string containing the name of the file to be saved and its directory, if other than the current directory, '*Variable n*' are strings containing the names of the *n* variables that are to be saved in the order that they appear, and '*-ascii*' is a string indicating that the data will be saved in ASCII format.

Suppose that we want to save the square of each value in *DataSection33.txt* as ASCII text. Then the script is

```
load('DataSection33.txt')
y = DataSection33.^2;
save('SavedDataSection33.txt', 'y', '-ascii')
```

Upon execution, the script creates a text file whose contents are

```
1.2100000e+002 1.4400000e+002 1.6900000e+002
4.4100000e+002 4.8400000e+002 5.2900000e+002
9.6100000e+002 1.0240000e+003 1.0890000e+003
1.6810000e+003 1.7640000e+003 1.8490000e+003
```

When just the file name is given, MATLAB places the file in the current directory. In order to place the file in a specific directory, the entire path name must be given. For example, consider the script

```
load ('DataSection33.txt')
y = DataSection33.^2;
save('c:\Matlab mfiles\Matlab results\SavedDataSection33.txt', 'y', '-ascii')
```

If we want to save additional quantities in this file, we will have to append their respective variable names to the `save` statement as follows. Suppose that the above script is also to compute the square root of the values in *DataSection33.txt*. Then,

```
load('DataSection33.txt')
y = DataSection33.^2;
z = sqrt(DataSection33);
save('c:\Matlab mfiles\Matlab results\SavedDataSection331.txt', 'y', 'z', '-ascii')
```

Execution of this script creates the file *SavedDataSection331.txt* with the contents

```
1.2100000e+002 1.4400000e+002 1.6900000e+002
4.4100000e+002 4.8400000e+002 5.2900000e+002
9.6100000e+002 1.0240000e+003 1.0890000e+003
1.6810000e+003 1.7640000e+003 1.8490000e+003
3.3166248e+000 3.4641016e+000 3.6055513e+000
4.5825757e+000 4.6904158e+000 4.7958315e+000
5.5677644e+000 5.6568542e+000 5.7445626e+000
6.4031242e+000 6.4807407e+000 6.5574385e+000
```

The data in the first four rows correspond to *y* and those in the last four rows correspond to *z*.

To save strings, the following method can be used. Let us create again a formatted string array using `sprintf` and save it using `save`. The script is

```
z = 100*magic(3)/17;
x = sprintf('%6.2f    %6.2f    %6.2f\n', z');
save('SaveStringArray.txt', 'x', '-ascii')
% Display what has been stored in the file
% as numerical values
load('SaveStringArray.txt');
disp(char(SaveStringArray))
```

Upon execution, we obtain

```
47.06    5.88   35.29
17.65   29.41   41.18
23.53   52.94   11.76
```

The `char` function was used to convert the ascii string characters to numerical values.

3.4 CELL ARRAYS

Cells provide a hierarchical way of storing dissimilar kinds of data. Cells are a special class of arrays whose elements usually contain data that differ in each cell both in the type and size of data that each cell contains. Any cell in a cell array can be accessed through matrix indexing as is done with standard vectors and matrices. Cell notation differs from standard matrix notation in that open brace '{' and the closed brace '}' are used instead of open bracket '[' and closed bracket ']'. We shall now illustrate how to create and access cell arrays.

Let us create four different arrays of data as shown in the following script:

```
A = ones(3,2)
B = magic(3)
C = char('Pressure', 'Temperature', 'Displacement')
D = [6+7j, 15]
```

Upon executing this script, we obtain

```
A =
  1   1
  1   1
  1   1
B =
  8   1   6
  3   5   7
  4   9   2
C =
Pressure
Temperature
Displacement
D =
  6.0000 + 7.0000i 15.0000
```

We now add to this script the cell assignment statement to create a (2×2) cell array, which is analogous to that used for standard arrays, except that we use the braces as delimiters. Thus,

```
A = ones(3,2);
B = magic(3);
C = char('Pressure', 'Temperature', 'Displacement');
D = [6+7j, 15];
Cel = {A, B; C, D}
```

After executing this script, we obtain

```
Cel =
  [3x2 double]    [3x3 double]
  [3x12 char ]    [1x2 double]
```

Notice that we do not get what is specifically in each cell, only what the size of the data arrays in each cell are and their type. To display the contents of *Cel* to the command window, we use

```
celldisp(c)
```

where *c* is a cell. Then, the script is modified as follows:

```
A = ones(3,2);
B = magic(3);
C = char('Pressure', 'Temperature', 'Displacement');
D = [6+7j, 15];
Cel = {A, B; C, D};
celldisp(Cel)
```

which upon execution gives

```
Cel{1,1} =
  1   1
  1   1
  1   1
Cel{2,1} =
Pressure
Temperature
Displacement
Cel{1,2} =
  8   1   6
  3   5   7
  4   9   2
Cel{2,2} =
  6.0000 + 7.0000i 15.0000
```

To access each cell individually, we use the index notation employed for standard array variables, except that we use the braces instead of the parentheses. Thus, to access the element at the intersection of the first row and the second column of *Cel*, we modify our script as follows:

```
A = ones(3,2);
B = magic(3);
C = char('Pressure', 'Temperature', 'Displacement');
D = [6+7j,15];
Cel = {A, B; C, D};
Cell_1_2 = Cel{1,2}
```

Upon execution, the following results are displayed to the command window

```
Cell_1_2 =
  8   1   6
  3   5   7
  4   9   2
```

With cell arrays, the use of `sort` can be extended to sort words in dictionary order. Consider the following script, where we will create a cell array of five words and then sort them.

```
Words = {'application', 'apple', 'friend', 'apply', 'fiend'};
WordSort = sort(Words)'
```

The execution of this script gives

```
'apple'
'application'
'apply'
'fiend'
'friend'
```

Notice that to obtain a column of sorted words, we took the transpose of the result. Also, since *WordSort* is a cell vector, the apostrophes are preserved.

These results could also be obtained in the following manner:

```
Words = cellstr(char('application', 'apple', 'friend', 'apply', 'fiend'));
WordSort = sort(Words)
```

where `char` produces a column vector of five strings of equal length as described previously and `cellstr` converts the five strings to a cell of five string elements. Each element has only the letters because in the conversion process `cellstr` removes trailing blanks.

These same operations can also be performed by using `sortrows` as follows:

```
A = char('application', 'apple', 'friend', 'apply', 'fiend');
B = sortrows(A)
```

where both A and B are matrices of order (5×11).

3.5 INPUT MICROSOFT EXCEL FILES

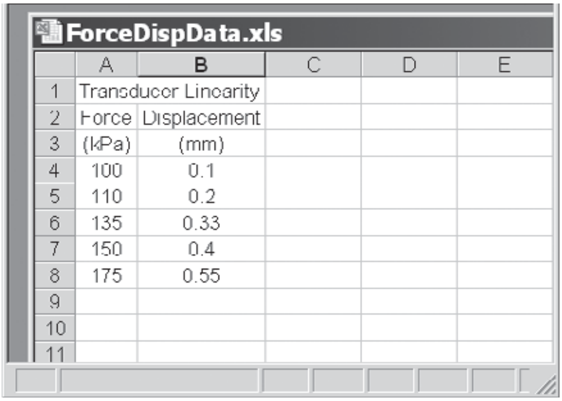
Data files created in Microsoft Excel can be read into MATLAB with the function

```
[X, Y] = xlsread('Filename')
```

where X will be an array containing the columns and rows of data and Y will be a cell array containing any text headers that accompany the data. The file name must contain the suffix '.xls'.

To illustrate the use of this function, consider the data generated in Excel as shown in Figure 3.1. These data are saved in a file named *ForceDispData.xls*. The script to read this file is

```
[X, Y] = xlsread('ForceDispData.xls')
```



ForceDispData.xls

	A	B	C	D	E
1	Transducer Linearity				
2	Force	Displacement			
3	(kPa)	(mm)			
4	100	0.1			
5	110	0.2			
6	135	0.33			
7	150	0.4			
8	175	0.55			
9					
10					
11					

Figure 3.1 Data recorded in Microsoft Excel.

The path has been set prior to this statement, so that the system knows where the file resides. Upon execution, the following data are displayed to the command window.

```
X =
   100.0000 0.1000
   110.0000 0.2000
   135.0000 0.3300
   150.0000 0.4000
   175.0000 0.5500
Y =
   [1x20 char]      []
   'Force'   'Displacement'
   '(kPa)'   '(mm)'
```

The size of the cell array Y is $\{3 \times 2\}$. Since there was only a text statement in the first row of column one (Excel cell A1) and none in the second column (Excel cell B1), MATLAB does not consider this text to be part of the column headers. It collects it, but does not display it. However, it is accessible by typing $Y\{1, 1\}$ in the command window. When this is done, we find that the system responds with

```
ans =
Transducer Linearity
```

3.6 SUMMARY OF FUNCTIONS INTRODUCED IN CHAPTER 3

A summary of the functions introduced in the chapter and their descriptions are presented in Table 3.1.

TABLE 3.1 MATLAB Functions Introduced in Chapter 3

MATLAB function	Description
`char`	Places each string into a row of a matrix and pad each row with blanks
`celldisp`	Displays to the command window the contents of a cell array
`cellstr`	Creates a cell array of strings from a character array
`deblank`	Removes trailing blanks in a string expression
`eval`	Executes a string containing a MATLAB expression
`disp`	Displays text or an array to the command window
`findstr`	Finds a string in another string
`fprintf`	Writes formatted data to a file or to the command window
`input`	Requests user input from the command window
`int2str`	Converts an integer or an array of integers to a string
`load`	Reads variables from a file
`num2str`	Converts a number or an array of numbers to a string
`save`	Saves arrays or strings to a file
`sortrows`	Sorts rows in ascending order
`sprintf`	Formats data and converts the formatted data to a string
`strcmp`	Compares two strings; case sensitive
`strtrim`	Removes leading and trailing blanks in a string expression
`xlsread`	Reads a Microsoft Excel file

EXERCISES

3.1 Generate a script that converts from the English length unit of feet to the metric unit of meters. The display of the result to the command window should look like

```
Enter the value of length in feet: 11.4
11.4 ft = 3.4747 m
```

where the value 11.4 was entered by the user.

3.2 There are 43,560 sq. ft per acre and 0.0929 sq. m per sq. ft. Generate a script that converts the number of acres to the number of square meters. The display of the result to the command window should look like

```
Enter the number of acres: 2.4
2.4 acres = 9712.4554 sq. m
```

where 2.4 was entered by the user.

3.3 Generate a script that converts a positive integer less than 2^{52} (4.5036×10^{15}) to a binary number. The MATLAB function that performs the conversion is `dec2bin`, whose argument is the decimal number and whose output is a string of the binary equivalent. The display to the command window should look like

```
Enter a positive integer < 4.5x10^15: 37
The binary representation of 37 is 100101
```

where 37 was entered by the user.

3.4 Generate a script that displays the magnitude and phase angle in degrees of a complex number. The display to the command window should look like

```
Enter the real part of a complex number: -7
Enter the imaginary part of a complex number: 13
The magnitude and phase of -7+13i is
Magnitude = 14.7648 Phase angle = 118.3008 degrees
```

where –7 and 13 were entered by the user.

3.5 The Fibonacci numbers can be generated from the relation

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] \qquad n = 0, 1, 2, \ldots$$

Generate the first 16 numbers using both `fprintf` and `disp` and present them to the MATLAB command window as follows:

```
F 0 =   0
F 1 =   1
F 2 =   1
F 3 =   2
⋮
F15 =   610
```

3.6 Given the array of angles $\theta_n = \pi/n, n = 2, 3, \ldots, 9$. Generate a script that computes $\cos\theta_n$ and displays the results as

```
cos(pi/9) = 0.93969;   pi/9 = 20.000 degrees
cos(pi/8) = 0.92388;   pi/8 = 22.500 degrees
cos(pi/7) = 0.90097;   pi/7 = 25.714 degrees
cos(pi/6) = 0.86603;   pi/6 = 30.000 degrees
cos(pi/5) = 0.80902;   pi/5 = 36.000 degrees
cos(pi/4) = 0.70711;   pi/4 = 45.000 degrees
cos(pi/3) = 0.50000;   pi/3 = 60.000 degrees
cos(pi/2) = 0.00000;   pi/2 = 90.000 degrees
```

Obtain this display by using `disp` and by using `fprintf`.

3.7 Generate a script that alphabetizes the months of the year when given in calendar order and then displays them as

```
April       June
August      March
December    May
February    November
January     October
July        September
```

3.8 For $N < 12$, generate a script that displays $n!, n = 1, 2, \ldots, N$ and the sum of these factorials as shown for $N = 4$.

```
Enter an integer < 12: 4
For n = 1,    1! = 1
For n = 2,    2! = 2
For n = 3,    3! = 6
For n = 4,    4! = 24
The sum of these 4 factorials = 33
```

where the value of 4 was entered by the user.

4

Program Flow Control

Edward B. Magrab

The various means of controlling the order in which a program's expressions get evaluated are presented.

4.1 INTRODUCTION—THE LOGICAL OPERATOR

The control of the order in which a program's expressions get evaluated are achieved by four program flow control structures: `while`, `if`, `for`, and `switch`. Each time one of these statements appears, it must be followed at a later place within the program by an `end` statement. All expressions that appear between the control structure statement and the `end` statement are executed until all requirements of the structure are satisfied. Each of these control structure statements can appear as often as necessary within themselves or within other control structures. When this occurs, they are called nested structures.

Control structures frequently rely on relational and logical operators to determine whether a condition has been met. When a condition has been met, the structure directs the program to a specific part of the program to execute one or more expressions. Several of MATLAB's relational and logical operators are given in Table 4.1. The && and || operators are special operators that perform AND and OR operations on logical expressions containing *scalar* values. They are called in MATLAB short-circuit operators and have been introduced to increase execution speed. They work in the following manner. For the expression (A<a)&&(B>b), the second operand (B>b) is only evaluated when the first operand (A<a) is true. Since it is an AND operation, it is unnecessary to evaluate the second operand (B>b) when the first operand has already failed the test. Similar reasoning holds when we replace the && operator with the || operator.

When using control structures, it is recommended that the statements following each control structure definition up to, but not including, the `end` statement be indented. This greatly improves the readability of the script or function. When the structures are nested, the entire nested structure is indented, with the nested structure's indentation preserved. When using MATLAB's editor/debugger, this can be done automatically.

One can use the relational and logical operators appearing in Table 4.1 to create a logical function whose output is 1 if the relational and logical operations are true and 0 if they are false. Suppose that we want to create a function $g(x)$ such that

$$\begin{aligned} g(x) &= f(x) \qquad a \le x < b \\ &= 0 \qquad\quad x < a \text{ and } b \ge x \end{aligned}$$

The logical operator is formed by

$$\text{y} = ((\text{a}<=\text{x})\&(\text{x}<\text{b}));$$

TABLE 4.1 Several Relational and Logical Operators

Conditional	Mathematical symbol	MATLAB symbol
Relational operators		
Equal	=	==
Not equal	≠	~=
Less than	<	<
Greater than	>	>
Less than or equal	≤	<=
Greater than or equal	≥	>=
Logical operators		
And	AND	& or &&
Or	OR	| or ||
Not	NOT	~

where a and b have been assigned numerical values prior to this statement and

```
((a<=x) & (x<b))
```

is the logical operator that has a value of 1 (true) when $x \geq a$ and $x < b$. Its value is 0 (false) for all other values of x. Thus, if we let $a = -1$, $b = 2$, $f(x) = e^{x/2}$, and $x = [-4\ -1\ 1\ 4]$, then a script using this logical operator is

```
a = -1;   b = 2;
x = [-4, -1, 1, 4];
r = (a<=x)
p = (x<b)
logi = (r & p)
gofx = exp(x/2).*logi
```

which upon execution yields

```
r =
  0   1   1   1
p =
  1   1   1   0
logi =
  0   1   1   0
gofx =
  0   0.6065   1.6487   0
```

Notice that dot multiplication was employed because x and *logi* are each (1×4) vectors. The intermediate expressions r, p, and *logi* were introduced to explicitly show that they are each a vector of logical results: ones (true) and zeros (false). If a and b were not scalars, then each would have to be of the same size as x. This case is discussed below.

In practice, the expressions r, p, *logi*, and *gofx* would be combined into one expression as shown below:

```
a = -1; b = 2;
x = [-4, -1, 1, 4];
gofx = exp(x/2).*((a<=x) & (x<b))
```

The logical operator can be used to create the unit step function $u(t)$, which is defined as

$$\begin{aligned} u(t) &= 1 \qquad t \geq 0 \\ &= 0 \qquad t < 0 \end{aligned}$$

For example, if t varies by increments of 0.25 in the range $-1 \leq t \leq 1$, then the following script creates the unit step function:

```
t = -1:0.25:1;
UnitStep = (t>=0);
disp('   t   UnitStep')
disp([t'  UnitStep'])
```

Upon execution, the following results are displayed to the command window:

```
        t        UnitStep
   -1.0000          0
   -0.7500          0
   -0.5000          0
   -0.2500          0
         0       1.0000
    0.2500       1.0000
    0.5000       1.0000
    0.7500       1.0000
    1.0000       1.0000
```

In the previous illustrations, we compared a vector to a scalar. In this final illustration, we compare two vectors of equal length. Consider the following script:

```
a = [4, 5, 6, 7, 8];
b = [4, 3, 2, 1, 8];
d = (a == b)
e = (a > b)
```

Its execution gives

```
d =
  1   0   0   0   1
e =
  0   1   1   1   0
```

4.2 CONTROL OF PROGRAM FLOW

Program flow control is performed by branching or looping. The branching is done with the `if` and `switch` statements and the looping with either the `for` or the `while` statements. We shall now discuss these four control structures.

4.2.1 Branching—If Statement

The `if` statement is a conditional statement that branches to different parts of its structure depending on the satisfaction of conditional expressions. The general form of the `if` statement is

```
if condition #1
   expressions #1
elseif condition #2
   expressions #2
else
   expressions #3
end
```

When condition #1 is satisfied, expressions #1 are executed, followed by the next statement after the end statement. When condition #1 is not satisfied, then condition #2 is examined. If it is satisfied, then expressions #2 are executed, and they are followed by the next statement after the end statement. If neither condition #1 nor condition #2 is satisfied, then expressions #3 are executed, followed by the next statement after the end statement. The statements elseif and else are optional. Also, there can be more than one elseif statement.

The following script illustrates the use of the if statement. The quantities *j*, *x*, and *nnum* have numerical values that were either assigned or determined from a computational procedure earlier in the program.

```
if j == 1
   z = sin(x);       ←— Executed only when j = 1.
   if nnum <= 4      ←— This if statement encountered only when j = 1.
      nr = 1;     ]  ←— These statements executed only when j = 1 and nnum ≤ 4.
      nc = 1;     ]
   else
      nr = 1;     ]  ←— These statements executed only when j = 1 and nnum > 4.
      nc = 2;     ]
   end
else
   nr = 2;        ]  ←— These statements executed only when j ≠ 1.
   nc = 1;        ]
end
```

It is seen that in the above script we have a nested if statement and, therefore, we require a second end statement. Note how the indenting of the expressions inside the various nested structures makes the code more readable.

When one uses a condition statement that compares a vector to a scalar, the condition is satisfied only when each element in the vector satisfies the condition. To illustrate this, consider the following script:

```
a = [4, 5, 6, 7, 8];
if a > 2
   disp(a)
end
```

The execution of this script gives

```
4   5   6   7   8
```

whereas the execution of

```
a = [4, 5, 6, 7, 8];
if a>5
   disp(a)
end
```

does not display anything.

To terminate a script (or a function; see Chapter 5) because a specified condition had not been satisfied, one would use `error`. The `error` function is usually used to ensure that the program is using numerical values that lead to meaningful results. When the program encounters the `error` function, it will display the message contained within it to the command window. After displaying the message, the execution of the script or function is terminated and control is returned to the command line in the command window. An example of the use of `error` is given in Example 4.1.

Example 4.1 Fatigue strength factors

Consider the relationships that govern the correction factors used to estimate the fatigue strength of metals.

Factor	Range	Correction
Load	Bending	$C_{\text{load}} = 1$
	Axial	$C_{\text{load}} = 0.70$
Size	$d \leq 8$ mm	$C_{\text{size}} = 1$
	$8 \leq d \leq 250$ mm	$C_{\text{size}} = 1.189d^{-0.097}$
Temperature	$T < 450°\text{C}$	$C_{\text{temp}} = 1$
	$450°\text{C} \leq \text{T}$	$C_{\text{temp}} = 1 - 0.0032(T - 840)$

A portion of a script that can be used to determine these factors is given below. The values of *lode*, *d*, and *temp* have had numerical values either assigned or computed previously in the program. The quantity *lode* is a string, and it too has been assigned a value.

```
if lode == 'bending'
   cload = 1;
elseif lode == 'axial'
   cload = 0.7;
else
   error('No such loading')
end
if d < 0
   error('Negative diameter not allowed')
elseif d <= 8
   csize = 1;
else
   csize = 1.189*d^(-0.097);
end
if temp <= 450
   ctemp = 1;
else
   ctemp = 1-0.0032*(T-840);
end
```

Notice that we have included several tests to ensure that the data values are permissible. When they aren't, an error message is sent to the command window and the program is terminated.

4.2.2 Branching—switch Statement

The switch structure is essentially an alternative to using a series of if-elseif-else-end structures. The general form of the switch statement is

```
switch switch_expression
   case case_expression #1
      statements #1
   case case_expression #2
      statements #2
      . . .
   case case_expression #n
      statements #n
   otherwise
      statements #n+1
end
```

The first *case_expression #j*, where $j = 1, 2, \ldots, n$, that is encountered in which *case_expression #j = switch_expression* will cause *statements #j* to be executed. Only one case is executed when the switch structure is entered. The *switch_expression* can be a numerical value, a logical value (0 or 1), or a string. Following the execution of *statements #j*, the next statement to be executed is that following the end statement. If none of the *case_expression #j* is satisfied, then statements $\#n + 1$ are executed. The otherwise statement is optional. If the otherwise statement has been omitted and *case_expression #j* does not equal *switch_expression* for any *j*, then the next statement after the end statement is executed.

The following switch structure acts as indicated. The quantity *k* has been assigned a value or had its value computed prior to encountering this structure.

```
a = 3;
switch k
   case 1
      disp('Case 1')          ←— This statement executed only when k = 1.
   case{2, 3}                     %Notice the use of a cell
      disp('Case 2 or 3')     ←— This statement executed only when k = 2, 3.
   case a^2
      disp('Case 9')          ←— This statement executed only when k = 9.
   otherwise
      disp('Otherwise')       ←— This statement executed only when k ≠ 1, 2, 3, or 9.
end
```

Example 4.2 Selecting one of four views of a surface

Consider the situation where one wants to view a surface $z(x, y)$ in one of four orientations: regular (*reg*), top (*top*), right side (*rside*), or left side (*lside*). We shall use the switch function to display the view selected. The surf function is used to plot a 3D perspective of the array of values for z as a function of x and y. The view function sets the viewing angles. The script requests that the user enter the view to be displayed. The quantities x, y, and z either have been previously assigned values or had their values previously computed. The portion of the script that uses switch is as follows:

```
surf(x, y, z)
str = input('Enter view reg, top, rside, or lside: ', 's')
switch str
  case 'reg'
    view(-37.5, 30)
  case 'top'
    view(-90, 90)
  case 'rside'
    view(0, 0)
  case 'lside'
    view(-90, 0)
  otherwise
    error('No such view')
end
```

4.2.3 For Loop

A for loop repeats a series of statements a specific number of times. Its general form is

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

where *statements* can be a function of *variable*.

Array Pre-Allocation

Before proceeding with several examples illustrating the use of the for loop, we shall discuss a typical for structure that is frequently employed and the means by which this structure should be used in order to obtain the shortest execution time. The first structure that we consider is a single for loop of the form

```
A = zeros(Nrow, 1);        % Array pre-allocation
for r = 1:Nrow
  Statements
  A(r) = . . .
end
```

where *Nrow* is a positive integer (that has been previously assigned a numerical value). The addition of the array assignment statement

```
A = zeros(Nrow, 1);
```

is necessary in order for the loop to execute at maximum speed.[1] The arguments in the assignment statement for A must have the order shown.

The second for structure that we consider is the nested for loops of the form

```
B = zeros(Nrow, Ncol)        % Pre-allocation
for c = 1:Ncol               % Column index must be outer loop
  Statements
  for r = 1:Nrow             % Row index must be inner loop
    Statements
    B(r, c) = . . .
  end
end
```

where *Nrow* and *Ncol* are positive integers (that have been previously assigned numerical values). The subscripts in the assignment statement

```
B = zeros(Nrow, Ncol);
```

must have the order shown and the subscript order (nesting) for r and c must be in the order shown; that is, the outer loop indexes the columns and the inner loop the rows.[2]

One should not infer from these two typical usages of a for loop that one needs to use subscripted variables (A and B in these illustrations) when a for loop is used. In fact, one should avoid the use of any unnecessary subscripted variables.

We now give several examples of the application of the for loop.

Example 4.3 Creation of a sequentially numbered matrix

We shall create a script that generates an $(N \times N)$ matrix in which the elements of each row of the matrix consist of sequential numbers as follows. The first row has sequential values for its elements such that $a_{11} = 1$ and $a_{1n} = N$; the second row has elements such that $a_{21} = N + 1$ and $a_{2n} = 2N$; and the last row has elements such that $a_{n1} = (N - 1)N + 1$ and $a_{nn} = N^2$. The script is

```
N = input('Enter a positive integer < 15: ');
Matr = zeros(N, N);
for r = 1:N
  Matr(r, 1:N) = ((r-1)*N+1):r*N;
end
disp(Matr)
```

[1] MATLAB states that execution speeds can be increased by a factor of greater than 500 if this statement is included. [S. McGarrity, "Maximizing Code Performance by Optimizing Memory Access," *TheMathWorks News & Notes*, June 2007, pp. 12–14.]

[2] MATLAB states that execution speeds can be increased by an additional factor of between two and three if this statement is included. [McGarrity, "Maximizing Code Performance."]

Upon execution, we obtain

```
Enter a positive integer < 15:9
   1   2   3   4   5   6   7   8   9
  10  11  12  13  14  15  16  17  18
  19  20  21  22  23  24  25  26  27
  28  29  30  31  32  33  34  35  36
  37  38  39  40  41  42  43  44  45
  46  47  48  49  50  51  52  53  54
  55  56  57  58  59  60  61  62  63
  64  65  66  67  68  69  70  71  72
  73  74  75  76  77  78  79  80  81
```

where the value of 9 was entered by the user.

Example 4.4 Dot multiplication of matrices

We shall create a script that performs the dot multiplication of two matrices A and B of the same order. The script is equivalent to $A.*B$. In our case, we shall illustrate the procedure using $A = $ magic(3) and $B = A'$. However, in general, before the multiplication can be performed, one must ensure that the order of the matrices is equal. We will include the procedure for doing this, although it is not necessary since for our choice of A and B we know that they are of the same order. The script is

```
A = magic(3);
B = A';
[rA, cA] = size(A);
[rB, cB] = size(B);
if(rA~=rB)||(cA~=cB)
  error('Matrices must be the same size')
end
M = zeros(rA, cA);
for c = 1:cA
  for r = 1:rA
    M(r, c) = A(r, c)*B(r, c);
  end
end
disp(M)
```

Upon execution, we obtain

```
64   3  24
 3  25  63
24  63   4
```

Example 4.5 Analysis of the amplitude response of a two-degree-of-freedom system

Consider a structurally damped two-degree-of-freedom system that is being subjected to a harmonic forcing at each of its masses at a nondimensional forcing frequency Ω. If the ratio of two masses is m_r and the ratio of the uncoupled natural frequencies of the system is ω_r, then the nondimensional displacements Y_1 and Y_2 of each mass can be determined from

$$\begin{Bmatrix} Y_1 \\ Y_2 \end{Bmatrix} = \begin{bmatrix} (1 + j\eta_1) + \omega_r^2 m_r(1 + j\eta_2) - \Omega^2 & -\omega_r^2 m_r(1 + j\eta_2) \\ -\omega_r^2(1 + j\eta_2) & \omega_r^2(1 + j\eta_2) - \Omega^2 \end{bmatrix}^{-1} \begin{Bmatrix} 1 \\ 1 \end{Bmatrix}$$

where $0 < \eta_1 < 1$ and $0 < \eta_2 < 1$ represent the dissipative loss in the springs.

We shall determine the maximum value of $|Y_1|$ and $|Y_2|$ when $m_r = 0.1$, $\omega_r = 0.6$, $\eta_1 = \eta_2 = 0.1$, and we take 500 values of Ω in the range $0 \le \Omega \le 2$. The script is

```
N = 500;   eta1 = 0.1;   eta2 = eta1;
wr = 0.6;   mr = 0.1;   B = [1; 1];
Om2 = linspace(0, 2, N).^2;
A = [1+1j*eta1+wr^2*mr*(1+1j*eta2), -wr^2*mr*(1+1j*eta2); . . .
     -wr^2*(1+1j*eta2), wr^2*(1+1j*eta2)];
sav = zeros(N, 2);
for k = 1:N
   sav(k,:) = abs(inv(A-diag([Om2(k), Om2(k)]))*B);
end
for h = 1:2
   [mx, ix] = max(sav(:,h));
   disp(['Max of|Y('int2str(h) ')| = 'num2str(mx, 6) . . .
          ' at Omega = ' num2str(sqrt(Om2(ix)), 4)])
end
```

Upon execution, we obtain

```
Max of |Y(1)| =   8.75776 at Omega = 1.026
Max of |Y(2)| = 43.2219 at Omega = 0.5852
```

Notice that the matrix A has been evaluated before the for loop is entered, since the for loop is being used to increment Ω and A is independent of Ω.

Example 4.6 Example 2.2 revisited

We shall revisit Example 2.2 and show how that result can be generalized so that we can create N^2 subarrays of order $(N \times N)$ so that each subarray is of the form

$$n\begin{bmatrix} 0 & 1 & \dots & 1 \\ 1 & 0 & & \\ \dots & & & \\ 1 & & & 0 \end{bmatrix} \rightarrow (N \times N)$$

where $n = 1, 2, \dots, N^2$. Each of these subarrays is then arranged to form the $(N^2 \times N^2)$ array shown in Example 2.2 for $N = 3$. The script is

```
N = input('Enter a positive integer<6: ');
cnt = 0;
a = ones(N, N)-eye(N);
A = zeros(N^2, N^2);
```

```
for c = 1:N
   row = ((c-1)*N+1):c*N;
   for r = 1:N
      cnt = cnt+1;
      col = ((r-1)*N+1):r*N;
      A(row, col) = cnt*a;
   end
end
disp(A)
```

Upon execution, we obtain

```
Enter a positive integer < 6: 3
   0   1   1   0   2   2   0   3   3
   1   0   1   2   0   2   3   0   3
   1   1   0   2   2   0   3   3   0
   0   4   4   0   5   5   0   6   6
   4   0   4   5   0   5   6   0   6
   4   4   0   5   5   0   6   6   0
   0   7   7   0   8   8   0   9   9
   7   0   7   8   0   8   9   0   9
   7   7   0   8   8   0   9   9   0
```

where the value of 3 was entered by the user.

Example 4.7 Total interest of a loan

We shall compute the total interest on a loan when the amount of the loan is L, its duration is m months, and its annual percentage interest I_a. The monthly payment p_{mon} is determined from

$$p_{\text{mon}} = \frac{iL}{1 - (1 + i)^{-m}}$$

where

$$i = I_a/1200$$

is the monthly interest rate expressed as a decimal number. Each month, as the loan is being paid off, a portion of the payment is used to pay the interest, and the remainder is applied to the unpaid loan amount. The unpaid loan amount after each payment is called the balance. Mathematically, we express these relations as follows. If $b_0 = L$, then

$$\begin{aligned} i_n &= ib_{n-1} \\ P_n &= p_{\text{mon}} - i_n \qquad n = 1, 2, 3, \ldots, m \\ b_n &= b_{n-1} - P_n \end{aligned}$$

where i_n is the portion of p_{mon} that goes toward the payment of the interest and P_n is the portion of the payment that goes toward the reduction of the balance b_n; that is, the amount required to pay off the loan. The total interest paid at the end of the loan's duration is

$$i_T = \sum_{j=1}^{m} i_j$$

The script to compute i_T is

```
loan = input('Enter loan amount: ');
durat = input('Enter term of loan in months: ');                 ←  Input
int = input('Enter annual interest rate (%): ')/1200;
ints = zeros(durat, 1);
prins = ints;
bals = ints;                                                      ←  Initialization
pmon = (loan*int)/(1-(1+int)^(-durat));
bals(1) = loan;
for m = 2:durat+1
   ints(m) = int*bals(m-1);
   prins(m) = pmon-ints(m);                                       ←  Computation
   bals(m) = bals(m-1)-prins(m);
end
fprintf(1, 'Total interest = $%8.2f/n', sum(ints))                ←  Output
```

As noted in Section 1.2, this script follows a program's usual structure: input, initialization, computations, and output, which in this case is to display the results to the command window. Execution of the script gives

```
Enter loan amount: 100000
Enter term of loan in months: 360
Enter annual interest rate (%): 8
Total interest = $164155.25
```

The first three lines are the user's response to the script's sequentially displayed queries, where the user entered the three numerical quantities shown after each query, and the last line is the answer. Notice that no comma was used in entering the first numerical value—100000—because the comma indicates the end of an expression.

Example 4.8 Equivalent implementation of `find`

We shall assume that we are given a vector g of positive and negative numbers and of arbitrary length. The objective is to create a script that performs the same function as the expression

```
indx = find(g>a)
```

where a is specified by the user. We shall check the script with $a = 4$ and with the vector $g = [4, 4, 7, 10, -6, 42, 1, 0]$. The script is

```
g = [4, 4, 7, 10, -6, 42, 1, 0];
a = 4;   k = 0;
indx = [];
for n = 1:length(g)
   if g(n) > a
      k = k+1;
      indx(k) = n;
   end
end
disp(['Element locations for g(n)>' num2str(a)': 'num2str(indx)])
```

Upon execution, the following results are displayed to the command window

```
Element locations for g(n)>4:  3  4  6
```

Example 4.9 Equivalent implementation of **cumsum**

For a vector c of arbitrary length, we shall create a script that provides the same results as

```
Csum = cumsum(c)
```

We shall verify the script with the vector $c = [4, 4, 7, 10, -6, 42, 1, 0]$. The script is

```
c = [4, 4, 7, 10, -6, 42, 1, 0];
Csum = zeros(length(c), 1);
Csum(1) = c(1);
for k = 2:length(c)
  Csum(k) = Csum(k-1)+c(k);
end
disp(['Cumsum of c = ' num2str (Csum')])
```

Upon execution, the following results are displayed to the command window

```
Cumsum of c = 4  8  15  25  19  61  62  62
```

Example 4.10 Specification of the elements of an array

We shall create an $(n \times n)$ matrix whose elements are either $+1$ or -1 such that the sign of each element is different from its adjacent elements, both from those above and below it and those on either side of it. The selection of n is arbitrary. Thus, we want to create the matrix

$$M = \begin{bmatrix} 1 & -1 & 1 & \dots \\ -1 & 1 & -1 & \dots \\ 1 & -1 & 1 & \dots \\ \vdots & & & \end{bmatrix} \rightarrow (n \times n)$$

The script is

```
n = input('Enter the order of the square matrix: ');
k = 1:n;
M = zeros(n, n);
OddRow = (-1).^(k-1);
EvenRow = (-1).^k;
for m = 1:2:n
  M(m,:) = OddRow;
  if m+1 <= n
    M(m+1,:) = EvenRow;
  end
end
disp(M)
```

The execution of this script for $n = 3$ displays to the command window

```
Enter the order of the square matrix: 3
 1  -1   1
-1   1  -1
 1  -1   1
```

where the value 3 was entered by the user.

Example 4.11 Sorting a vector of numerical values in ascending order

We shall create a script that does the same thing that sort does, that is, produces a vector whose elements go from the most negative value to the most positive value. We shall verify our script with the following vector: [17, 12, 12, -6, 0, -14].

```
H = [17, 12, 12, -6, 0, -14];
LH = length(H);
for k = 1:(LH-1)
   smin = H(k);
   for m = (k+1):LH
      if H(m) < smin
         smin = H(m);
         M = m;
      end
   end
   temp = H(k);
   H(k) = H(M);
   H(M) = temp;
end
disp(H)
```

It is seen that the last three statements of the outer loop use the same swapping method that was used in Example 2.3. Also note that the index for the inner loop is a function of the index of the outer loop. This has been done because the previous *k* elements have already been placed in their correct order and there is no need to include them again in the magnitude evaluation process. The execution of this script gives

```
-14   -6   0   12   12   17
```

4.2.4 While Loop

The while loop repeats one or more statements an indefinite number of times, leaving the loop only when a specified condition has been satisfied. Its general form is

```
while condition
   statements
end
```

where the expression defining *condition* is usually composed of one or more of the variables evaluated by *statements*. The while loop is typically used when the number of times one is going to use the loop is not known.

We now present several examples of the use of while loops.

Example 4.12 Ensuring that data are input correctly

The following excerpt from a program asks the user to enter a number from one to eight and continues to request that from the user until the entry is in the specified range. The input function prints the message appearing in quotes to the command window and waits until the user enters a value, at which point the program sets that value to *nfnum*.

```
nfnum = 0;
while (nfnum < 1)||(nfnum > 8)
   nfnum = input('Enter a number from 1 to 8: ');
end
```

The quantity *nfnum* is initially set equal to a value (in this case zero) that causes the while test (*nfnum*<1)||(*nfnum*>8) to enter the while structure. After reaching the last expression prior to the end statement, the program returns to the while test expression to determine if it is satisfied. If it is not satisfied, it executes the next line in the structure; otherwise, it proceeds to the next statement after the end statement. Notice that we have employed the preferred practice of placing the structure's initializing value—*nfnum* in this case—just prior to its entry into the structure.

Example 4.13 Convergence of a series

Let us determine and display the number of terms that it takes for the series

$$S_N = \sum_{n=1}^{N} \frac{1}{n^2}$$

to converge to within 0.01% of its exact value, which is $S_\infty = \pi^2/6$.

The script is

```
series = 1;   k = 2;   exact = pi^2/6;
while abs((series-exact)/exact) >= 1e-4
   series = series+1/k^2;
   k = k+1;
end
disp(['Number of terms = ' int2str(k-1)])
```

which, upon execution, displays to the command window

```
Number of terms = 6079
```

The quantity *series* is initially set equal to a value (1 in this case) that causes the while test abs((*series-exact*)/*exact*) to enter the while structure. After reaching the last expression prior to the end statement, the program returns to the while test expression to determine if it is satisfied. If it is not satisfied, it executes the next line in the structure; otherwise, it proceeds to the next statement after the end statement, which in this case displays the result to the command window.

We have used the absolute value of the test condition ((*series-exact*)/*exact*) to avoid any instance when the difference *series-exact* is negative, which would be less than 10^{-4} (since it is a negative number), but whose magnitude may not be $<10^{-4}$. This avoids having to know *a priori* whether the quantity *series* approaches the limit from above or below. One must also be careful when establishing the test criterion for the termination of the while loop, for, if improperly or poorly stated, one may stay in the loop indefinitely—that is, until control and c are pressed simultaneously by the user.

Example 4.14 Approximation to π

The following expression converges to $1/\pi$ when $x_0 = 1/\sqrt{2}$ and $y_o = 1/2$:

$$y_{n+1} = y_n(1 + x_{n+1})^2 - 2^{n+1}x_{n+1} \qquad n = 0, 1, 2, \ldots$$

where

$$x_{n+1} = \frac{1 - \sqrt{1 - x_n^2}}{1 + \sqrt{1 - x_n^2}}$$

We shall show that the difference $|1/\pi - y_4|$ is less than 10^{-15}. The script is

```
xo = 1/sqrt(2);   yo = 1/2;   n = 0;
while abs(1/pi - yo) > 1e-15
   xo=(1-sqrt(1-xo^2))/(1+sqrt(1-xo^2));
   yo = yo*(1+xo)^2-2^(n+1)*xo;
   n = n+1;
end
fprintf(1, 'For n = %2.f, |1/pi-y_(n+1)| = %5.4e\n', n, abs(1/pi - yo))
```

Execution of this script results in

```
For n = 4, |1/pi-y_(n+1)| = 4.9960e-016
```

Example 4.15 Multiple root finding using interval halving[3]

MATLAB has a function fzero that is used to determine the value of x that makes $f(x)$ very closely equal to zero, provided that fzero is given a good estimate of the location of the root. See Section 5.5.1. However, it only finds one zero at a time; to find additional zeros, one must call fzero again with another estimate of where the next zero can be found.

Let us assume that we would like to find automatically a series of positive values of x ($x_1, x_2, \ldots$) that make $f(x) = 0$. This assumes, of course, that $f(x)$ has multiple zeros and, therefore, the sign of $f(x)$ alternates as x increases. One technique to find the zeros of a function is called interval halving.[4] Referring to Figure 4.1, this technique works as follows. The independent variable x is given a starting value $x = x_{\text{start}}$ and the sign of $f(x_{\text{start}})$ is determined. The variable x is then incremented by an amount Δ and the sign of $f(x_{\text{start}} + \Delta)$ is determined. The signs of these two values are compared. If the signs are the same (as they are in Figure 4.1), then x is again incremented by Δ and the sign of $f(x + 2\Delta)$ is evaluated and compared to that of $f(x_{\text{start}})$. If the signs are different, then the current value of x is decremented by half the interval size—that is, by $\Delta/2$. From Figure 4.1, we see that the sign change

[3] See, for example, S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 2nd ed., McGraw-Hill, New York, 1988, p. 128ff.

[4] Although this method will find the roots, it is not the best way to do it for the techniques that are used in fzero require two to three times fewer iterations to find a root to within a specified precision.

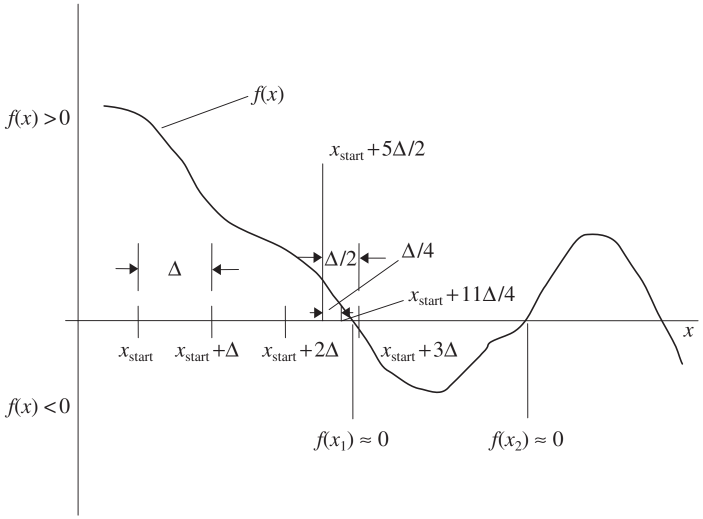

$f(x) > 0$
$f(x)$
$x_{\text{start}} + 5\Delta/2$
Δ
$\Delta/2$
$\Delta/4$
$x_{\text{start}} + 11\Delta/4$
x
x_{start}
$x_{\text{start}} + \Delta$
$x_{\text{start}} + 2\Delta$
$x_{\text{start}} + 3\Delta$
$f(x) < 0$
$f(x_1) \approx 0$
$f(x_2) \approx 0$

Figure 4.1 Interval halving scheme.

occurs at $x = x_{\text{start}} + 3\Delta$ so that after the sign change has been detected, the next value for x is $x = x_{\text{start}} + 5\Delta/2$. The sign of $f(x_{\text{start}} + 5\Delta/2)$ is then compared to $f(x_{\text{start}})$. If it is the same, then half the current interval is added to the current value of x; otherwise, it is subtracted from the current value of x. In this example, the sign of $f(x_{\text{start}})$ is the same as $f(x_{\text{start}} + 5\Delta/2)$ so that the next point at which $f(x)$ is evaluated is $x_{\text{start}} + 11\Delta/4$. This process is repeated until the incremental change in x divided by the current value of x is less than some tolerance; that is, $\Delta_{\text{current}}/x_{\text{current}} < t_o$, the tolerance. When this tolerance criterion has been satisfied, the root $x_1 = x_{\text{current}}$. The process is continued until the desired number of x_j has been determined. After each x_j has been obtained, we reset Δ to its original value, set x_{start} to a value slightly larger than x_j, say $x_{\text{start}} = 1.05x_j$, and repeat the process.

The objective is to write a script using the interval halving technique to determine the first five values of x that set the function

$$f(x) = \cos(ax) \cong 0$$

where a is a constant. These x_j are said to satisfy this equation when the incremental change in x divided by x is less than $t_o = 10^{-6}$. Also, $x_j \geq x_{\text{start}} \geq 0$ for $j = 1, 2, \ldots, n$. Thus, in general, the inputs to the root-finding portion of the program are n, x_{start}, t_o, and Δ, and for this particular $f(x)$ the quantity a.

In the present case, $n = 5$ and $t_o = 10^{-6}$ and we shall let $x_{\text{start}} = 0.2$, $\Delta = 0.3$, and $a = \pi$. The script is

```
n = 5;    a = pi;
increment = 0.3;    tolerance = 1e-6;
xstart = 0.2;    x = xstart;    dx = increment;
route = zeros(n, 1);
```

```
for m = 1:n
   s1 = sign(cos(a*x));
   while dx/x > tolerance
      if  s1 ~= sign(cos(a*(x+dx)))
         dx = dx/2;
      else
         x = x+dx;
      end
   end
   route(m) = x;
   dx = increment;
   x = 1.05*x;
end
disp(route')
```

The function sign brings back a +1, −1, or 0, depending on whether the sign of its argument is positive, negative, or zero, respectively. Upon execution, the following results are displayed in the command window.

```
0.5000    1.5000    2.5000    3.5000    4.5000
```

In Section 5.3, we shall convert this script into a function so that we can determine the roots for an arbitrary $f(x)$.

4.2.5 Early Termination of Either a `for` or a `while` Loop

The break function is used to terminate either a for or while loop. If the break function is within nested for or while loops, then it returns to the next higher level for or while loop. Consider the following portion of a script:

```
for j = 1:14
  ⋮
  b = 1
  while b < 25
    ⋮
    if n < 0              When n < 0 the while loop is exited
      break               and the script continues from the next
    end                   statement after this end statement
    ⋮
  end
  ⋮
end
```

4.3 SUMMARY OF FUNCTIONS INTRODUCED IN CHAPTER 4

Several logical and relational operators are given in Table 4.1. A summary of the additional functions introduced in the chapter is presented in Table 4.2.

TABLE 4.2 MATLAB Functions Introduced in Chapter 4

MATLAB function	Description
`break`	Terminates the execution of a `for` or `while` loop
`case`	An alternative identifier in the `switch` structure
`else`	Executes statements based on a relational or logical expression
`elseif`	Executes statements based on a relational or logical expression
`end`	Terminates a `for`, `while`, `if`, or `switch` structure
`error`	Displays error message and abort program execution
`for`	Repeats statements a specific number of times
`if`	Executes statements based on a relational or logical expression
`otherwise`	Default part of a `switch` statement
`sign`	Signum function
`switch`	Switches among several cases based on a value
`while`	Repeats statements an indefinite number of times until a condition is satisfied

EXERCISES

Section 4.2.1

4.1 Write a script that computes the day of week for the years 2010, 2011, and 2012, when its input is of the form month/day/year: xx/xx/xxxx. The following information is given: January 1, 2010, is a Thursday, January 1, 2011, is a Saturday, and January 1, 2012, is a Sunday and is a leap year. Have the script input and output the results in the following manner:

```
Enter month, day and year in the form xx/xx/xxxx for 2010, 2011, or 2012: 08/31/2011
The date 08/31/2011 is the 243 day of the year and falls on a Wednesday.
```

where 08/31/2011 was entered by the user.

Section 4.2.3

4.2 The estimate of the variance of n samples x_i is determined from

$$s_n^2 = \frac{1}{n-1}\left[\sum_{j=1}^{n} x_j^2 - n\bar{x}_n^2\right] \qquad n > 1$$

where

$$\bar{x}_n = \frac{1}{n}\sum_{j=1}^{n} x_j$$

is an estimate of the mean. The variance is determined from `var`. Write a script that determines s_n^2 as a function of $n, n > 1$, for the following data: $x = [45\ 38\ 47\ 41\ 35\ 43]$. [Answer: [24.50 22.33 16.25 24.20 19.90].]

4.3 Given a vector a of N elements a_n, $n = 1, 2, \ldots, N$. The simple moving average of m sequential elements of this vector is defined as

$$\mu_j = \mu_{j-1} + \frac{a_{m+j-1} - a_{j-1}}{m} \qquad j = 2, 3, \ldots, (N - m + 1)$$

where

$$\mu_1 = \frac{1}{m} \sum_{k=1}^{m} a_k$$

Write a script that computes these moving averages when a is given by $a = 5$*(1 + `rand`(N, 1)), where `rand` generates uniformly distributed random numbers. Assume that $N = 100$ and $m = 6$. Plot the results using `plot`(j, μ_j) for $j = 1, 2, \ldots, N - m + 1$.

Section 4.2.4

4.4 The Newton method is used to determine the value of x that makes $|f(x)| < t_o \approx 0$. The value of x is determined from the following formula, where x_0 is the initial guess:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad n = 0, 1, 2, \ldots$$

In this equation, the prime denotes the derivative with respect to x. The value of n is a function of the tolerance t_o, which is a very small value, and the choice of x_o. Write a function that obtains the value of x when

$$f(x) = \cos(x) + \sin(x)$$
$$f'(x) = -\sin(x) + \cos(x)$$

and when $x_0 = 1.5$ and $t_o < 10^{-8}$. Have the output of the program display to the command window the various quantities as shown below.

```
At x = 2.3561945, f(x) = 1.1102e-016 after 4 iterations
```

4.5 The number of years n that it takes to deplete an annuity that starts with an amount P (the principal) by withdrawing an amount A on the first day of each year when the account earns interest at the rate of I_r% annually can be determined from the following relations:

After the first day of the first year, the annuity has an amount $R_1 = P - A$.
After the first day of the second year, the annuity has an amount
$R_2 = R_1(1+I_r/100) - A$.
After the first day of the third year, the annuity has an amount $R_3 = R_2(1+I_r/100) - A$.
$\vdots$
After the first day of the nth year the annuity has an amount
$R_n = R_{n-1}(1+I_r/100) - A < A$.

Determine the number of years that one can withdraw the amount A when $P = \$250{,}000$, $I_r = 4.5\%$, and $A = \$25{,}000$. Also determine the amount left in the annuity at the beginning of the nth year after A dollars has been withdrawn. Display the results as follows:

```
Principal   Annuity   Interest    No.     Remain
   ($)      ($/yr)      (%)      years     ($)
 250000      25000      4.5       12      19112
```

4.6 The arithmetic–geometric mean process is a means that can be used to evaluate elliptic integrals. For a given α, the arithmetic–geometric mean process to obtain the complete elliptic integral is as follows

$$\begin{array}{llll} a_0 = 1 & b_0 = \cos\alpha & c_0 = \sin\alpha & \\ a_n = \dfrac{1}{2}(a_{n-1} + b_{n-1}) & b_n = \sqrt{a_{n-1}b_{n-1}} & c_n = \dfrac{1}{2}(a_{n-1} - b_{n-1}) & n = 1, 2, \ldots, N \end{array}$$

When $|c_N| < t_o$, where $t_o \ll 1$ is the tolerance to which the process is said to have converged to,

$$K(\alpha) = \frac{\pi}{2a_N}$$

where $K(\alpha)$ is the complete elliptic integral of the first kind. For $\alpha = \pi/4$ and $t_o = 10^{-5}$, show that this process produces the same value as that obtained from `ellipke`($\sin^2\alpha$). Use `format long e` to verify your result.

Sections 4.2.1 and 4.2.3

4.7 Create a script that performs the equivalent function of the logical operator introduced in Section 4.1 for any vector of values h such that the output vector v of the logical operator indicates which of its elements satisfy $h > a$ and $h < b$. Test your script with $h = [1\ 3\ 6\ {-7}\ {-45}\ 12\ 17\ 9]$, $a = 3$, and $b = 13$. [Answer: $v = [0\ 0\ 1\ 0\ 0\ 1\ 0\ 1]$.]

4.8 The elements of a $(N \times N)$ Hankel matrix are given by

$$\begin{aligned} h_{nm} &= 0 && n+m-1 > N \\ &= n + m - 1 && \text{otherwise} \end{aligned}$$

Generate a script that creates a Hankel matrix for $N < 10$.

Sections 4.2.1 and 4.2.4

4.9 Generate a script that asks the user to enter a positive integer from 1 to 19 and continues to make this request if any number other than one of these 19 integers is entered. The `rem` function should prove useful.

4.10 Given the following relation

$$\begin{aligned} x_{n+1} &= x_n/2 && \text{if } x_n \text{ is an even positive integer} \\ &= 3x_n+1 && \text{if } x_n \text{ is an odd positive integer} \end{aligned}$$

where $n = 1, 2, 3, \ldots$. Let x_1 be an arbitrary positive integer. This iteration process appears to always converge at some value of $n + 1 = N$ such that $x_N = 1$. The value of N cannot be predicted as a function of x_1 as you will discover. Print all the values of x_j for the selected value of x_1. The function `rem` should prove useful. For a test case, verify your code by obtaining the following sequence: 3, 10, 5, 16, 8, 4, 2, 1. Other numbers to experiment with are $x_1 = 24$, 17, and 27.

Sections 4.2.3 and 4.2.4

4.11 Consider the following relation

$$x_{n+1} = x_n^2 + 0.25 \qquad n = 0, 1, 2, \ldots, N$$

For $x_0 = 0$, write two scripts that plot the values of x_n for $n = 0, 5, 10, \ldots, 200$. In the first script, use a `for` loop and in the second script a `while` loop. To what value does x_N appear to converge? For the third argument of the `plot` function, use `plot(..., ..., 'ks')`, which will plot the values of x_n as squares. The x-axis values are the values of n and the y-axis values are the x_n. Note that all x_n, $n = 0, 1, 2, \ldots, 200$, must be computed, but only every fifth x_n is plotted. This exercise differs from Exercise 4.10 in that the values of x_n must be saved as elements of a vector so that the appropriate elements can be subsequently displayed.

4.12 For a given $a > 0$, the following relationship will determine the positive value of the $\sqrt{a}$ to within a tolerance t_0 for any starting value (guess) $x_0 > 0$

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \qquad n = 0, 1, 2, \ldots$$

where $x_{n+1} \cong \sqrt{a}$. (When $x_0 < 0$, the negative square root is found.) Write a script that determines $\sqrt{a}$ to within $|x_n - x_{n+1}| < 10^{-6}$ for $a = 7$. How many iterations does it take if (a) $x_0 = 3$ and (b) $x_0 = 100$. The first iteration is the determination of x_1. [Hint: Notice that the above relationship is not an explicit function of n. Here the subscript n is simply an indicator that the next (new) value x_{n+1} is a function of the previous (old) value x_n. Thus, each time through the loop the old and new values keep changing. Therefore, one has to keep track of n in order to record the number of times this relationship is used until the convergence criterion is met.] [Answer: (a) $n_{\text{iterations}} = 4$ and (b) $n_{\text{iterations}} = 10$.]

Sections 4.2.1, 4.2.2, and 4.2.3

4.13 Given two polynomials:

$$y(x) = p_1x^n + p_2x^{n-1} + \ldots + p_nx + p_{n+1}$$
$$z(x) = s_1x^m + s_2x^{m-1} + \ldots + s_mx + s_{m+1}$$

Write a script to add them, that is, $h(x) = y(x) + z(x)$, when $m = n, m < n$, and $m > n$. Polynomials are added by adding the coefficients of the terms with the same exponent. Assume that the input to the script are the vectors $p = [p_1 p_2 \ldots p_n\, p_{n+1}]$ and $s = [s_1 s_2 \ldots s_m s_{m+1}]$.

Check your script with the following data sets:

i. $p = [1, 2, 3, 4]$ and $s = [10, 20, 30, 40]$;
ii. $p = [11, 12, 13, 14]$ and $s = [101, 102]$; and
iii. $p = [43, 54, 55]$ and $s = [77, 66, 88, 44, 33]$.

[Answers: (i) $h = [11, 22, 33, 44]$; (ii) $h = [11, 12, 114, 116]$; (iii) $h = [77, 66, 131, 98, 88]$.]

Sections 4.2.1, 4.2.3, and 4.2.4

4.14 Given the following relation

$$x_{n+1} = ax_n(1 - x_n) \qquad n = 1, 2, \ldots, N$$

When $x_1 = 0.1$, determine which of the following cases converge: $a = 1.45, 2.75, 3.2$, and 4. Consider the sequence to be nonconvergent when $|(x_{n+1} - x_n)/x_{n+1}| > 10^{-4}$ and $N > 200$ and to be convergent when $|(x_{n+1} - x_n)/x_{n+1}| \leq 10^{-4}$ and $N \leq 200$. Print the value of x_N.

4.15 Create a script that asks the user to input sequentially two positive integers N and M such that $N \leq 9$ and $M \leq N^2$ and ensures that these limitations are met. The program has to ask the user for these two integers separately, N first and then M if N has the correct attributes.

Sections 4.2.1– 4.2.4

4.16 The chi-square statistic is used to perform goodness-of-fit tests. It is defined as

$$X^2 = \sum_{i=1}^{k} \frac{(x_i - e_i)^2}{e_i}$$

where e_i and x_i are independent vectors of length k.

If $e_i < 5$, then the e_i and x_i must be combined with their respective e_{i+1} and x_{i+1} values. If the sum of $e_i + e_{i+1}$ is still < 5, then e_{i+2} is added to the sum of $e_i + e_{i+1}$. This process is repeated until the sum is ≥ 5. When $e_i \geq 5$ and the sum of the remaining $e_{i+1}, e_{i+2}, \ldots, e_k$, is less than 5, then these remaining values are added to e_i.

Write a script that computes X^2 under the conditions described above. Check your results with the following vectors, which represent three different cases:

i. $x = [1, 7, 8, 6, 5, 7, 3, 5, 4]$ and $e = [2, 6, 10, 4, 3, 6, 1, 2, 3]$;
ii. $x = [7, 11, 13, 6]$ and $e = [6, 10, 15, 7]$; and
iii. $x = [3, 14, 20, 25, 14, 6, 2, 0, 1, 0]$ and $e = [4, 12, 19, 19, 14, 8, 4, 2, 1, 1]$.

Hint: The most compact script will be obtained by performing tests on the elements of `cumsum`(e), where the length of e changes as the evaluation procedure progresses. [Answers:

i. $e_{\text{modified}} = [8, 10, 7, 6, 6]$, $x_{\text{modified}} = [8, 8, 11, 7, 12]$, $X^2 = 8.8524$;
ii. $e_{\text{modified}} = [6, 10, 15, 7]$, $x_{\text{modified}} = [7, 11, 13, 6]$, $X^2 = 0.6762$; and
iii. $e_{\text{modified}} = [16, 19, 19, 14, 8, 8]$, $x_{\text{modified}} = [17, 20, 25, 14, 6, 3]$, $X^2 = 5.6349$.]

5

Function Creation and Selected MATLAB Functions

Edward B. Magrab

The creation of functions and their various uses within MATLAB are described, and several MATLAB functions that are frequently used to obtain numerical solutions to engineering problems are illustrated.

5.1 INTRODUCTION

One form of an M file is the script file. A second type of M-file is the function file. Function files are script files that create their own local and independent workspace within MATLAB. Variables defined within a function are local to that function; they neither affect nor are they affected by the same variable names being used in any script or other function file. All of MATLAB's functions are of this type. The exceptions are those variables that are designated global variables in user-created functions, which are discussed in Section 5.2.2. The first noncomment line of a function must follow a prescribed format, which is given in Section 5.2.2. Typically, user-created MATLAB programs are either scripts or functions and employ any number of user-created functions and MATLAB functions. MATLAB asserts that a function executes faster than a script. In addition, as we shall discuss, the function can contain within it additional functions called subfunctions; this is discussed in Section 5.2.3.

5.1.1 Why Use Functions

There are several reasons to create functions, besides the fact that many MATLAB functions require them, as discussed in Section 5.5. They are used to

1. Avoid duplicate code.
2. Limit the effect of changes to specific sections of a program.
3. Promote program reuse.

4. Reduce the complexity of the overall program by making it more readable and manageable.
5. Isolate complex operations.
6. Improve portability.
7. Make debugging and error isolation easier.
8. Improve performance because each function can be "optimized."

The compartmentalization brought about by the use of functions also tends to minimize the unintended use of data by portions of the program, because data to each function is provided only on a need-to-know basis.

5.1.2 Naming Functions

The names of functions should be chosen so that they are meaningful and indicate what the function does. Typical lengths of function names are between nine and twenty characters and should employ standard or consistent conventions. The proper choice of function names can also minimize the use of comments within the function itself. Recall, also, the naming conventions suggested in Section 1.2.3.

5.1.3 Length of Functions

The length of a function can vary from two lines of code to hundreds of lines of code. However, the length of a function should be governed, in part, by its functional cohesion—that is, the degree to which it does one thing and not anything else. For example, $\sin(x)$ is 100% cohesive, whereas a function that computes the sine and square root would be less cohesive because it does two separate things, each of which is unrelated to the other. A function can be created with numerous highly cohesive functions to create another cohesive function. An additional advantage of the creation of cohesive functions is their reliability—that is, lower error rate. In addition, when functions have a low degree of cohesion, one often encounters difficulty in isolating errors.

5.1.4 Debugging Functions

During the creation of functions (and scripts), the program should be independently verified to ensure that it is working correctly after every few expressions are written. MATLAB is particularly well suited to this type of procedure, which simply involves omitting the semicolon at the end of each expression. Furthermore, one incurs very little time penalty when omitting the semicolon, except in those expressions using large vectors and matrices or iterative solution techniques. The verification should be performed with some type of independent calculation or estimation. During the verification/debugging stage, any lines of code that may be inserted to provide intermediate output should be commented out, not deleted, until the entire function has been verified to be working correctly. Only after a function is working correctly should it be improved to decrease its execution time, if necessary. Creating correctly performing programs is the primary goal.

5.2 USER-DEFINED FUNCTIONS

5.2.1 Introduction

The function in MATLAB can be created in several ways. The most general form is the *function file*, which is created by the `function` keyword, saved in a file, and subsequently accessed by scripts and functions or from the command window. However, a function cannot be created in the command window. The creation of the function is given in Section 5.2.2. A second form of the function is the *subfunction*. When the `function` keyword is used more than once in a function file, then all additional functions that appear after the first `function` keyword are called subfunctions. The first usage of the `function` keyword denotes the *primary* function. The subfunctions are only accessible by the primary function and the other subfunctions within the primary function file. Subfunctions are used to reduce function file proliferation. The subfunction is discussed in Section 5.2.3. A third way to create a function is with an *anonymous function*, which provides a means of creating simple functions without having to use the function keyword. It is limited to a single MATLAB expression, but may use expressions that contain other anonymous functions. Anonymous functions are accessible from the command window, a script, function, or subfunction and can be created in any of these venues. The creation of an anonymous function is discussed in Section 5.2.4. A fourth way to create a function is with `inline`. Like the anonymous function, this function is also limited to one MATLAB expression but unlike the anonymous function it is accessible only from the command window, script, function, or subfunction in which it was created. Its creation is discussed in Section 5.2.5. The comparison of the usage of each of these ways to create functions is summarized in Section 5.2.6.

5.2.2 Function File

A function that is going to reside in a function file has at least two lines of program code, the first line having a format required by MATLAB. There is no terminating character or expression for the function program such as the end statement, which is required for the `for`, `while`, `if`, and `switch` structures. Furthermore, the name of the M-file must be the same as the name of the primary function, except that the file name has the extension ".m".

The number of variables and their type (scalar, vector, matrix, string, cell, function handle) that are brought in and out of the function are controlled by the function interface, which is the first noncomment line of the function program. In general, a function will consist of the interface line, some comments and one or more expressions as shown below. The interface has the general form given in the first line.

```
function [OutputVariables] = FunctionName(InputVariables)
% Comments
Expressions
OutputVariables = ...
```

OutputVariables is a comma-separated list of the names of the output variables, *InputVariables* is a comma-separated list of the names of the input variables, and **FunctionName** is the name of the function and in the case of a primary function, the name of the function M-file. It must begin with an upper or lower case letter and has the same restrictions as discussed for variable names in Section 1.2.3. The first word of this statement, `function`, is a reserved word that may only be used in this context. The function file may be stored in any directory to which a path has been or will be defined and has the file name *FunctionName.m*. However, in order for one to have access to this function during a session, the function and the script must have their respective paths known to the system. This is done by either having the script and function in the current directory, or by placing the function's directory in a path that has been saved using the procedure discussed regarding Figure 1.13.

The comments immediately following the function interface statement are used by MATLAB to create this function's Help information—that is, when one types at the command line

`help` **FunctionName**

all the initial contiguous comments will appear in the MATLAB command window. Any comments appearing prior to the `function` statement will not be part of the Help information. The *Help* information ends when no more contiguous comment lines are encountered, that is, when a blank line or an executable expression is encountered.

Special Case 1

Functions can also be used to create a figure, to display annotated data to the command window, or to write data to files. In these cases, no values are transferred back to the calling program (a script, function, or subfunction that uses this function in one or more of its expressions). In this case, the function interface line becomes

`function` **FunctionName**(InputVariables)

Special Case 2

When a function is used only to store data in a prescribed manner, the function does not require any input arguments. In this case, the function interface line has the form

`function` OutputVariables = **FunctionName**

This case is illustrated in Example 5.1.

Special Case 3

When a function is a primary function and the primary function is being used instead of script file, the function interface line has the form

`function` **FunctionName**

This form is the one that we shall use frequently to replace the script file, which has been employed in the previous chapters.

There are several concepts that must be understood to correctly create functions. The first is that the variable names used in the function definition do not have to match the corresponding names when the function is called from the command window, a script, or another function. Instead, it is the locations of the input variables within the comma-separated argument list inside the parentheses that govern the transfer of information—that is, the first argument in the calling statement transfers its value(s) to the first argument in the function interface line, and so on.

Second, the names selected for each argument are local to the function program and have meaning only within the context of the function program. The same names can be used in an entirely different context in the script file that calls this function or in another function used by this function. However, the names appearing for each input variable of the function statement must be of the same type: either scalar, vector, matrix, cell, string, or function handle in the calling program as in the function program in order for the function's expressions to work as intended. For example, the multiplication of two row vectors may result in an error message if the variables do not have the correct size (order). Furthermore, the names used for the input variables of the function statement are equivalent to their appearing on the left side of an equal sign. Thus, if one of the input variable names is a, then a is equivalent to $a =$ numerical value(s). The variable names are not local to the function when they have been assigned as global variables using `global`. The use of `global` is discussed subsequently.

We shall first illustrate the construction of a function, and then list several of its variations. Consider the following two equations that are to be computed in a function

$$x = \cos(at) + b$$
$$y = |x| + c$$

The values of x and y are to be returned by the function. We now create a function to compute these quantities and we call it **ComputeXY**, which is saved as a file named *ComputeXY.m*.[1]

```
function [x, y] = ComputeXY(t, a, b, c)
% Computation of -
% x = cos(at)+b
% y = |x|+c
% Scalars: a, b, c
% Vectors: t, x, y
x = cos(a*t)+b;
y = abs(x)+c;
```

When one types in the MATLAB command window

```
help ComputeXY
```

[1] The comments are included in this example to show its usage. In the large majority of scripts and functions presented in this book, the comment lines have been omitted in order to make the listings themselves more readable. However, in most cases, the important features of the programs are discussed within the text accompanying each script or function, or they are obvious from its context.

the following is displayed

```
Computation of -
   x = cos(at)+b
   y = |x|+c
Scalars: a, b, c
Vectors: t, x, y
```

Several other variations of the function interface and how they are accessed from the calling programs are given in Table 5.1. It is seen from the examples in this table that one must ensure that the number and the type of the input and output variables are correct with respect to how they are used by the function. These restrictions should be denoted in the function's comments intended for the response to the `help` request. In this case, we have assumed that t is a vector and that a, b, and c are scalars. We can now call this function by typing in the command window

```
[u, v] = ComputeXY(0:pi/4:pi, 1.4, 2, 0.75);
```

By virtue of the location within the parentheses, this means that with reference to the function, t = [0, pi/4, pi/2, 3*pi/4, pi], $a = 1.4$, $b = 2.0$, and $c = 0.75$. Upon executing this statement, we obtain

```
u =
   3.0000   2.4540   1.4122   1.0123   1.6910
v =
   3.7500   3.2040   2.1622   1.7623   2.4410
```

Functions normally return to the calling program when the last statement of the function is reached. To force an earlier return, one uses

```
return
```

Let us modify the function *ComputeXY* so that the function is only evaluated when the variable t is a vector of length two or more and when the number of arguments in

TABLE 5.1 Several Variations of the Function Statement [u, v] = **ComputeXY**(t, a, b, c)

Function	Accessing function from script or function	Comments

function z = **ComputeXY**(t, w)
x = cos(w(1)*t)+w(2);
z = [x; abs(x)+w(3)];
	t = 0:pi/4:pi;
	w = [1.4, 2, 0.75];
	q = **ComputeXY**(t, w);
		$w(1) = a = 1.4; w(2) = b = 2;$
		$w(3) = c = 0.75; q \rightarrow (2 \times 5)$
		$x(:) = q(1, 1:5); y(:) = q(2, 1:5)$

function z = **ComputeXY**(t, w)
x = cos(w(1)*t)+w(2);
z = [x abs(x)+w(3)];
	t = 0:pi/4:pi;
	w = [1.4, 2, 0.75];
	q = **ComputeXY**(t, w);
		$w(1) = a = 1.4; w(2) = b = 2;$
		$w(3) = c = 0.75; q \rightarrow (1 \times 10)$
		$x(:) = q(1:5); y(:) = q(6:10)$

function[x, y] = **ComputeXY**(t, w)
x = cos(w(1)*t)+w(2);
y = abs(x)+w(3);
	t = 0:pi/4:pi;
	w = [1.4, 2, 0.75];
	q = **ComputeXY**(t, w);
		$w(1) = a = 1.4; w(2) = b = 2;$
		$w(3) = c = 0.75; q \rightarrow (1 \times 5)$
		$x = q$; y not available[§]

§ Many of the MATLAB functions make use of this form, as will be seen in subsequent chapters.

the calling statement is four. This ensures that the user has entered the correct number of variables and that *t* is not a scalar. In order to determine the number of input variables that are actually used to call the function, we use `nargin`. To signify that inappropriate or insufficient data have been entered, the function returns `NaN`, which, we recall, is a reserved word meaning "not a number". Then, our previous function is modified as follows (the comments have been omitted for clarity):

```
function [x, y] = ComputeXY(t, a, b, c)
if (length(x) == 1)||(nargin ~= 4)
  x = NaN;
  y = NaN;
  return
end
x = cos(a*t)+b;
y = abs(x)+c;
```

In some instances, the number of different variables that are transferred to a function can become large. In these cases, it may be beneficial for the function to share the global memory of the script or function or to create access to global variables for use by various functions.[2] This access is provided by

```
global
```

To illustrate its usage, we shall transfer the values of *a*, *b*, and *c* in **ComputeXY** as global variables. The script is modified as follows:

```
function [x, y] = ComputeXY(t)
global A B C
x = cos(A*t)+B;
y = abs(x)+C;
```

where the blank spaces between the `global` variable names are required. Notice that the variables *a*, *b*, and *c* no longer appear in the function interface line.

The script required to call this function is now

```
global A B C
A = 1.4;    B = 2;    C = 0.75;
[u, v] = ComputeXY(0:pi/4:pi)
```

The same variable names must be used in both the script and the function and they must have the same context in both. Upon execution, the following values are displayed to the command window:

```
u =
  3.0000   2.4540   1.4122   1.0123   1.6910
v =
  3.7500   3.2040   2.1622   1.7623   2.4410
```

which is what we obtained previously.

[2] MATLAB suggests that global variables be used sparingly or not at all.

Since the arguments in the function definition are, in a sense, placeholders for the numerical values that will reside in their respective places when the function is executed, when appropriate, we can insert any correctly constructed MATLAB expression in the calling statement. To illustrate this, let us use **ComputeXY** to determine the values of *x* and *y* for

$$b = \sqrt{\frac{1.8}{(1 + k)^3}} \quad k = 1, \ldots, n$$

when *t* varies from 0 to π in increments of $\pi/4$, $c = 1/0.85$, *a* has *n* values that range from 1 to 1.4, and $n = 3$. Upon using **ComputeXY** of the form

```
function [x, y] = ComputeXY(t, a, b, c)
x = cos(a*t)+b;
y = abs(x)+c;
```

the script is

```
n = 3;
a = linspace(1, 1.4, n);
for k = 1:n
  [u, v] = ComputeXY(0:pi/4:pi, a(k), sqrt(1.8/(1+k)^3), 1/.85);
  disp(['For k = ', int2str(k)])
  disp(['u = ' num2str(u)])
  disp(['v = ' num2str(v)])
end
```

Upon execution, the following results are displayed in the command window:

```
For k = 1
   u =
      1.4743   1.1814     0.47434     -0.23277   -0.52566
   v =
      2.6508   2.3579     1.6508       1.4092     1.7021
For k = 2
   u =
      1.2582   0.84598   -0.050818   -0.69286   -0.55082
   v =
      2.4347   2.0225     1.2273       1.8693     1.7273
For k = 3
   u =
      1.1677   0.6217    -0.42008     -0.81998   -0.14131
   v =
      2.3442   1.7982     1.5966       1.9965     1.3178
```

If we repeat the above computation using global variables for *a*, *b*, and *c*, then the function is

```
function [x, y] = ComputeXY(t)
global A B C
x = cos(A*t)+B;
y = abs(x)+C;
```

and the script is

```
global A B C
n = 3;    C = 1/.85;
c = linspace(1, 1.4, n);
for k = 1:n
   A = c(k);
   B = sqrt(1.8/(1+k)^3);
   [u, v] = ComputeXY(0:pi/4:pi)
end
```

which upon execution produces the previously obtained results (without the annotation).

As a final remark, we illustrate the case where the results of a function are returned as a vector and are redefined in the script file as one row of a matrix. For simplification, we shall assume that we are interested only in the values of x and that these values are returned inside a `for` loop. Thus, a segment of a program could be

```
...
n = 4;
c = linspace(1, 1.4, n);
t = 0:pi/4:pi;
p = zeros(n, length(t));
for k = 1:4
   p(k,:) = ComputeXY(t, c(k), sqrt(1.8/(1+k)^3), 1/.85);
   ⋮
end
⋮
```

It is seen that $p(= u = \cos(at) + b)$ in this case will be a (4×5) matrix, since $k = 1, 2, 3, 4$ and the length of the vector t is 5. Recall that the notation $p(k,:)$ means that the kth row of matrix p is to have its column elements assigned the corresponding values of the columns of the row vector returned by **ComputeXY**. In addition to the fact that the initial assignment of p using `zeros` to pre-allocate an array is good programming practice, it is also necessary because of the way p is used in the `for` loop; otherwise, an error message will occur.

5.2.3 Subfunctions

When the `function` keyword is used more than once in a function file, all the additional functions created after the first `function` keyword are called subfunctions. The expressions comprising the first use of the `function` keyword is called the primary function. It is the only function that is accessible from the command window, scripts, and other functions residing in their own M-file. The subfunctions are accessible only to the primary function and to other subfunctions within the primary function file.

We shall illustrate the use of a primary function and subfunctions by computing the mean and standard deviation of a vector of numerical values. We shall compute this in a relatively inefficient manner in order to illustrate the properties and use of subfunctions. The mean m and the standard deviation s are given by

$$m = \frac{1}{n}\sum_{k=1}^{n} x_k$$

$$s = \left[\frac{1}{n-1}\left(\sum_{k=1}^{n} x_k^2 - nm^2\right)\right]^{1/2}$$

We shall call the primary function **MeanStdDev**, the subfunction that computes the mean m **meen**, and the subfunction that computes the standard deviation s **stdev**. The primary function and its subfunctions are saved in a file *MeanStdDev.m*. The primary function and subfunctions are then given by the following program:

```
function [m, s] = MeanStdDev(dat)          % Primary function
n = length(dat);
m = meen(dat, n);
s = stdev(dat, n);

function m = meen(v, n)                    % Subfunction
m = sum(v)/n;

function sd = stdev(v, n)                  % Subfunction
m = meen(v, n);                            % Calls a sub function
sd = sqrt((sum(v.^2)-n*m^2)/(n-1));
```

A script to illustrate the use of this function file is

```
v = [1, 2 , 3, 4];
[m, s] = MeanStdDev(v)
```

which upon execution gives

```
m =
  2.5000
s =
  1.2910
```

It is noted that **meen** and **stdev** cannot be used independently; that is, typing, for example,

```
v = [1, 2 , 3, 4];
m = meen(v, length(v))
```

in the command window will produce an error message indicating that the function file for **meen** cannot be found.

If one anticipates that the functions created in support of a script will not be used outside the immediate context, then one can convert the script to a function and make all functions subfunctions in that function file. The form of the primary function most likely will be that given by Special Case 3 in Section 5.2.2. We shall illustrate this procedure the first time in Example 5.1.

5.2.4 Anonymous Functions

Another way to create a function, either in the command window, a script, a primary function, or subfunction is by creating an anonymous function. Anonymous functions are a means of creating functions for simple expressions without having to create M-files or subfunctions. It can be composed of only one expression, and it can bring back only one variable—that is, the form [u, v] on the left-hand side of the equal sign is not allowed. Thus, any function requiring logic or multiple operations to arrive at the result cannot employ the anonymous function.

The general form of an anonymous function is

```
functionhandle = @(arguments) (expression)
```

where *functionhandle* is the function handle, *arguments* is a comma-separated list of variable names, and *expression* is a valid MATLAB expression. Any parameters that appear in *expression* and do not appear in *argument*s must be given a numerical value prior to this statement. The parentheses around *expression* are optional, but can be a visual aid. A function handle is a way of referencing a function and is used as a means of invoking the anonymous function and as a means to pass the function as an argument in functions, which then evaluates it using `feval` as shown in Section 5.3. A function handle is constructed by placing an "@" in front of the syntax as shown above or in front of a function name as shown in Section 5.3. The function handle *functionhandle* also serves as the name of the function and is used in the same manner as one does for a function created by the `function` keyword.

We illustrate the anonymous function with the following examples. Let us create an anonymous function that evaluates the expression

$$c_x = |\cos(\beta x)|$$

at $\beta = \pi/3$ and $x = 4.1$. The anonymous function is created with the following script:

```
bet = pi/3;
cx = @(x) (abs(cos(bet*x)));
disp(cx(4.1))
```

which upon execution gives

```
0.4067
```

This anonymous function could have also been created as having two arguments: β and x. In this case, the preceding script becomes

```
cx = @(x, bet) (abs(cos(bet*x)));
disp(cx(4.1, pi/3))
```

We can also use this anonymous function directly in another anonymous function. Let us create an anonymous function that determines the cube root of **cx**. Then, to illustrate the use of one anonymous function using another anonymous function, consider the following script:

```
cx = @(x, bet) (abs(cos(bet*x)));
cxrt = @(x, bet) (cx(x, bet)^(1/3));
disp(cxrt(4.1, pi/3))
```

which upon execution results in

```
0.7409
```

For a last example, consider the creation of an anonymous function that evaluates a two-element column vector v where

$$\begin{aligned} v_{11} &= x^2/4 + y^2 - 1 \\ v_{21} &= y - 4x^2 + 3 \end{aligned}$$

In this case, the anonymous function can be created in two ways. The first way is

```
v = @(x, y) ([0.25*x.^2+y.^2-1; y-4*x.^2+3]);
a = v(1, 2);
disp(['v(1,1) = 'num2str(a(1))' v(2,1) = ' num2str(a(2))])
```

which upon execution gives

```
v(1,1) = 3.25      v(2,1) = 1
```

The second way is

```
v = @(xy) ([0.25*xy(1).^2+xy(2).^2-1; xy(2)-4*xy(1).^2+3]);
a = v([1, 2]);
disp(['v(1,1) = ' num2str(a(1)) ' v(2,1) = ' num2str(a(2))])
```

where $xy(1) = x$ and $xy(2) = y$. Execution of this script gives the previous result.

5.2.5 inline

Another way to create a local function, either in the command window, a script, a primary function, or subfunction is by using inline. Like the anonymous function, this function also doesn't have to be saved in a separate file. However, it does have several limitations. It cannot call another inline function, but it can use a user-created function existing as a function file. Like the anonymous function it can be composed of only one expression and can bring back only one variable. Thus, any function requiring logic or multiple operations to arrive at the result cannot employ inline. An additional utility of inline is that it provides one way to convert a symbolic result to a function; this is shown in Section 5.6.

The general form of inline is

```
FunctionName = inline ('expression', 'v1','v2', . . . )
```

where *expression* is any valid MATLAB expression and $v1, v2, \ldots$ are the names of all the variables appearing in *expression*. The single quotation marks are required as shown.

We illustrate inline with the following example. Let us create a function **FofX** that evaluates

$$f(x) = x^2 \cos(ax) - b$$

where a and b are scalars and x is a vector. Then,

```
FofX = inline('x.^2.*cos(a*x)-b', 'x', 'a', 'b')
```

displays in the MATLAB command window

```
FofX =
Inline function:
FofX(x,a,b) = x.^2.*cos(a*x)-b
```

The dot multiplication is required, since x is a vector. If we had ended the inline expression with a semicolon, then this display would have been suppressed.

Thus, typing in the command window

```
FofX = inline('x.^2.*cos(a*x)-b', 'x', 'a', 'b');
g = FofX([pi/3, pi/3.5], 4, 1)
```

results in

```
g =
  -1.5483   -1.7259
```

The `inline` form for functions, just as for the anonymous function, has utility in many MATLAB functions that require one to first create a function that will be evaluated subsequently by that MATLAB function. Several examples of its usage are given in Section 5.5.

5.2.6 Comparison of the Usage of Subfunctions, Anonymous Functions, and `inline`

We shall now compare the usage of the subfunction, anonymous function, and the inline function. We will assume that the subfunctions, anonymous functions, and inline functions are used in a function file and that the operations to be performed are to determine the mean and standard deviation of a vector of numbers as illustrated in Section 5.2.3. The three usages are summarized in Table 5.2.

TABLE 5.2 Comparison of the Usage of Subfunctions, Anonymous Functions, and `inline`

Subfunctions	Anonymous functions	`inline`

function **Example**
dat = 1:3:52;
n = length(dat);
m = **meen**(dat, n)
s = **stdev**(dat, n)
 	
function m = **meen**(v, n)
m = sum(v)/n;
 	
function sd = **stdev**(v, n)
m = **meen**(v, n);
sd = sqrt((sum(v.^2)-
n*m^2)/(n-1));
	function **Example**
	dat = 1:3:52;
	n = length(dat);
	meen = @(dat, n)
	(sum(dat/n);
	stdev = @(dat, n, m)
	(sqrt((sum(dat.^2)
	-n*m^2)/(n-1)));
	m = **meen**(dat, n)
	s = **stdev**(dat, n, m)
		function **Example**
		dat = 1:3:52; n = length(dat);
		meen = inline('sum(dat)/n','dat','n');
		stdev = inline('sqrt((sum(dat.^2)-
		n*m^2)/(n-1))', 'dat', 'n', 'm');
		m = **meen**(dat, n)
		s = **stdev**(dat, n, m)

5.3 USER-DEFINED FUNCTIONS, FUNCTION HANDLES, AND FEVAL

Many MATLAB functions require the user to create functions in a form specified by that MATLAB function. These functions use the MATLAB function

```
feval(FunctionHandle, p1, p2, . . . , pn)
```

where *FunctionHandle* is the function handle (recall Section 5.2.4) and $p1, p2, \ldots$ are parameters that are to be passed to the function represented by *FunctionHandle*. Several examples of MATLAB functions that require the use of a function handle and `feval` are given in Section 5.5. In addition, there are situations when the user would also like to have this capability.

We will explain how this procedure works with an example based on the results of Example 4.11 where a root-finding program to determine the m lowest roots of a specific function $f(x) = 0$ was presented. We now convert this script to a function whose name is **ManyZeros**, which resides in the function M-file *ManyZeros.m*. The function $f(x)$ will now be arbitrary. In addition, it will be assumed that $f(x)$ has several parameters that are part of its definition. We recall that the root-finding program requires four inputs: m, the number of roots desired; x_s, the starting value for search; t, the computational tolerance that determines the degree of closeness to zero of $f(x_{\text{root}})$; and Δ, the initial search increment.

For this example, we shall let

$$f(x) = \cos(\beta x) - \alpha \qquad \alpha \leq 1$$

Thus, we have to transfer to the user-defined function two quantities: β and α. This user-defined function will be called **CosBeta**, and it will reside in the file *CosBeta.m*.

The function **ManyZeros** is as follows (recall Example 4.11).

```
function nRoots = ManyZeros(zname, n, xs, toler, dxx, w)
x = xs;
dx = dxx;
nRoots = zeros(n, 1);
for m = 1:n
  s1 = sign(feval(zname, x, w));
  while dx/x >toler
    if s1 ~= sign(feval(zname, x+dx, w))
      dx = dx/2;
    else
      x = x+dx;
    end
  end
  nRoots(m) = x;
  dx = dxx;
  x = 1.05*x;
end
```

The variable w is introduced because **ManyZeros** is intended to be used with an arbitrary $f(x)$, which, in general, may have any number of parameters. Therefore, w can be a vector of any length. The meaning of its individual elements will depend on the specific choice of $f(x)$. Consequently, **CosBeta** is given by

```
function d = CosBeta(x, w)
% beta = w(1);   alpha = w(2)
d = cos(x*w(1))-w(2);
```

We recall that **ManyZeros** requires the following input variables: (1) the name of the function defining $f(x)$, which is the function handle @**CosBeta**; and (2) six parameters, the first four of which correspond to m, x_s, t, and Δ and the remaining two are β and α, which are to be transferred to the function **CosBeta** as elements of the vector w. Depending on the value of m, the result *nRoots* is either a scalar ($m = 1$) or a vector of length m.

To access **CosBeta** and bring back its numerical value, we use the MATLAB function `feval`. The MATLAB function `sign` then evaluates the sign of the numerical value brought back by `feval`. Also, notice that the variable names defined in the function's input variables in the script file and the two function files are mostly different. This has been done to emphasize that only the locations and the subsequent usage of the arguments are important and not their alphanumeric descriptors, since the variable names are local to their respective functions.

The program that uses these functions is

```
function ExampleFeval
NoRoots = 5;   xStart = 0.2;
tolerance = 1e-6;   increment = 0.3;
beta = pi/3;   a = 0.5;
c = ManyZeros(@CosBeta, NoRoots, xStart, tolerance, increment, [beta, a])

function f = CosBeta(x, w)
f = cos(w(1)*x)-w(2)
```

The execution of the script displays to the command window

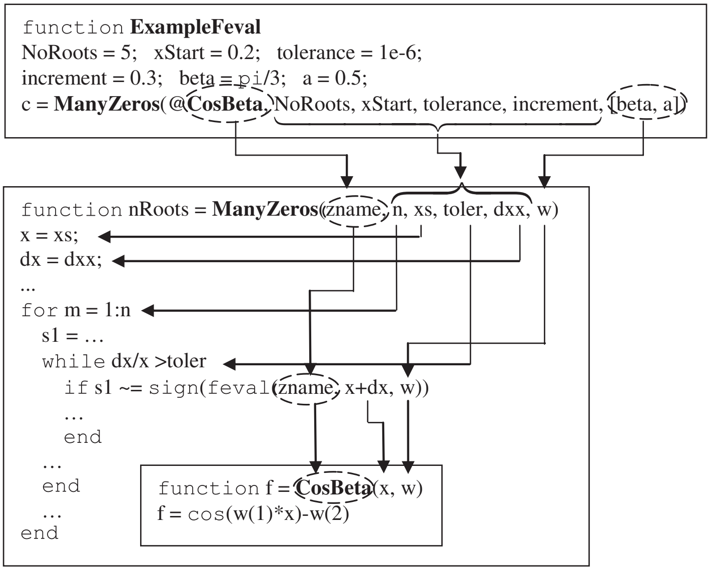
```
c =
   1.0000   5.0000   7.0000   11.0000   13.0000
```

To further illustrate the transfer of information from one function to another function, we have shown in Figure 5.1 how the various arguments are transferred in **ExampleFeval**.

5.4 MATLAB FUNCTIONS THAT OPERATE ON ARRAYS OF DATA

5.4.1 Introduction

There are many general-purpose functions in MATLAB that have a wide range of use in obtaining numerical solutions to engineering problems. We will consider a subset of them and divide them into two groups: those that operate on arrays of data



```
function ExampleFeval
NoRoots = 5;   xStart = 0.2;   tolerance = 1e-6;
increment = 0.3;   beta = pi/3;   a = 0.5;
c = ManyZeros(@CosBeta, NoRoots, xStart, tolerance, increment, [beta, a])
```

```
function nRoots = ManyZeros(zname, n, xs, toler, dxx, w)
x = xs;
dx = dxx;
...
for m = 1:n
  s1 = …
  while dx/x >toler
    if s1 ~= sign(feval(zname, x+dx, w))
    …
    end
  …
  end
  …
end
```

```
function f = CosBeta(x, w)
f = cos(w(1)*x)-w(2)
```

Figure 5.1 Graphical representation of how MATLAB transfers information from one function to another function.

and those that require user-defined functions. In this section, we consider those functions that require arrays of data as input. In Section 5.5, we consider those functions that require user-defined functions.

The functions that will be introduced in this section are:

`polyfit`	Fits a polynomial to an array of values
`polyval`	Evaluates a polynomial at an array of values
`spline`	Applies cubic spline interpolation to arrays of coordinate values
`interp1`	Interpolates between pairs of coordinate values
`trapz`	Approximates an integral from an array of amplitude values
`polyarea`	Determines the area of a polygon
`fft/ifft`	Determines the Fourier transform and its inverse from sampled data

5.4.2 Fitting Data with Polynomials—`polyfit/polyval`

Consider the general form of a polynomial

$$y(x) = c_1x^n + c_2x^{n-1} + \cdots + c_nx + c_{n+1} \tag{5.1}$$

where x is the input value and $y = y(x)$ its corresponding output. The coefficients c_k are determined from

```
c = polyfit(x, y, n)
```

where n is the order of the polynomial, $c = [c_1\, c_2\, \ldots\, c_n\, c_{n+1}]$ is a vector of length $n + 1$ representing the coefficients of the polynomial in Eq. (5.1), and x and y are each vectors of length $m \geq n + 1$; they are the data to which the polynomial is fitted, with x the input and y the output.

To evaluate Eq. (5.1) once we have determined c, we use

```
y = polyval(c, xnew)
```

where c is a vector of length $n + 1$ that has been determined from `polyfit` and *xnew* is either a scalar or a vector of points at which the polynomial will be evaluated. In general, the values of *xnew* in `polyval` can be arbitrarily selected and may or may not be the same as x.

If the coefficients c are not of interest, then we can combine `polyfit` and `polyval` as follows:

```
y = polyval(polyfit(x, y), xnew)
```

We now illustrate these functions with an example.

Example 5.1 Neuber's constant for the notch sensitivity of steel

A notch sensitivity factor q for metals can be defined in terms of Neuber's constant $\sqrt{a}$ and the notch radius r as follows:

$$q = \left(1 + \frac{\sqrt{a}}{\sqrt{r}}\right)^{-1}$$

The value of $\sqrt{a}$ is different for different metals and is a function of the ultimate strength S_u of the material. It can be estimated by fitting a polynomial to the experimentally obtained data of $\sqrt{a}$ as a function of S_u for a given metal. Once we have this polynomial, we can determine the value of q for a given value of r and S_u.

Let us consider the data given in Table 5.3 for steel. Using these data, we first determine the coefficients of a fourth-order polynomial that expresses $\sqrt{a}$ as a function of S_u, and then we use this polynomial to obtain q for a given r and S_u.

To fit the data appearing in Table 5.3 and to obtain the value of $\sqrt{a}$ for any value $0.3 \leq S_u \leq 1.7$ GPa and $0 < r < 5$ mm, we use the following script. For simplicity, we assume that we enter one set of S_u and r at a time. Furthermore, we place the data

TABLE 5.3 Neuber's Constant for Steel

S_u (GPa)	$\sqrt{a}(\sqrt{\text{mm}})$	S_u (GPa)	$\sqrt{a}(\sqrt{\text{mm}})$
0.34	0.66	1.17	0.14
0.48	0.46	1.31	0.10
0.62	0.36	1.45	0.075
0.76	0.29	1.59	0.050
0.90	0.23	1.72	0.036
1.03	0.19		

appearing in Table 5.3 in a function called **NeuberData**. In **NeuberData**, $nd(:,1) = S_u$ and $nd(:,2) = \sqrt{a}$. The generation of a set of graphs that display the values of q for a range of data is given in Figure 6.27b.

The program is

```
function Example5_1
ncs = NeuberData;
c = polyfit(ncs(:, 1), ncs(:, 2), 4);
r = input('Enter notch radius (0 < r < 5 mm): ');
Su = input('Enter ultimate strength of steel (0.3 < Su < 1.7 GPa): ');
q = 1/(1+polyval(c, Su)/sqrt(r));
disp('Notch sensitivity = ' num2str(q, 3)])

function nd = NeuberData
nd = [0.34, 0.66; 0.48, 0.46; 0.62, 0.36; 0.76, 0.29; 0.90, 0.23; 1.03, 0.19; ...
      1.17, 0.14; 1.31, 0.10; 1.45, 0.075; 1.59, 0.050; 1.72, 0.036];
```

Executing this script yields

```
Enter notch radius (0 < r < 5 mm): 2.5
Enter ultimate strength of steel (0.3 < Su < 1.7 GPa): 0.93
Notch sensitivity = 0.879
```

where, in the first two lines, the user entered sequentially the numbers 2.5 and 0.93 after each line was displayed. The program then computed the value of q and displayed the third line.

5.4.3 Fitting Data with `spline`

A very powerful way to generate smooth curves that pass through a set of discrete data values is to use splines. The function that performs this curve generation is

```
Y = spline(x, y, X)
```

where y is $y(x)$, x and y are vectors of the same length that are used to create the functional relationship $y(x)$, and X is a scalar or vector for which the values of $Y = y(X)$ are desired. In general, $x \neq X$.

We shall now illustrate the use of `spline`.

Example 5.2 Fitting data to an exponentially decaying sine wave

We shall generate some data using an exponentially decaying oscillatory function and then fit these data with a series of splines. The data will be generated by sampling the following function over a range of nondimensional times τ for $\xi < 1.0$:

$$f(\tau, \xi) = \frac{e^{-\xi\tau}}{\sqrt{1-\xi^2}}\sin\left(\tau\sqrt{1-\xi^2} + \varphi\right) \tag{5.2}$$

where

$$\varphi = \tan^{-1}\frac{\sqrt{1-\xi^2}}{\xi}$$

We shall evaluate this equation in a function called **DampedSineWave** and place it in its own M-file so that we can use it again. Thus,

```
function f = DampedSineWave(tau, xi)
r = sqrt(1-xi^2);
phi = atan(r/xi);
f = exp(-xi*tau).*sin(tau*r+phi)/r;
```

Let us sample twelve equally spaced points of $f(\tau,\xi)$ over the range $0 \le \tau \le 20$ and plot the resulting piecewise polynomial using 200 equally spaced values of τ. We shall also plot the original waveform and assume that $\xi = 0.1$. The program is

```
n = 12;   xi = 0.1;
tau = linspace(0, 20, n);
data = DampedSineWave(tau, xi);
newtau = linspace(0, 20, 200);
yspline = spline(tau, data, newtau);
yexact = DampedSineWave(newtau, xi);
plot(newtau, yspline, 'k—', newtau, yexact, 'k-')
```

which when executed produces Figure 5.2. The dashed lines are the fitted data. It is seen that the results are very good. The two curves become virtually indistinguishable from each other when fifteen equally spaced points are selected. A detailed discussion of `plot` is given in Section 6.2. The degree of closeness of the fit over the range considered could not have been obtained using `polyfit`.

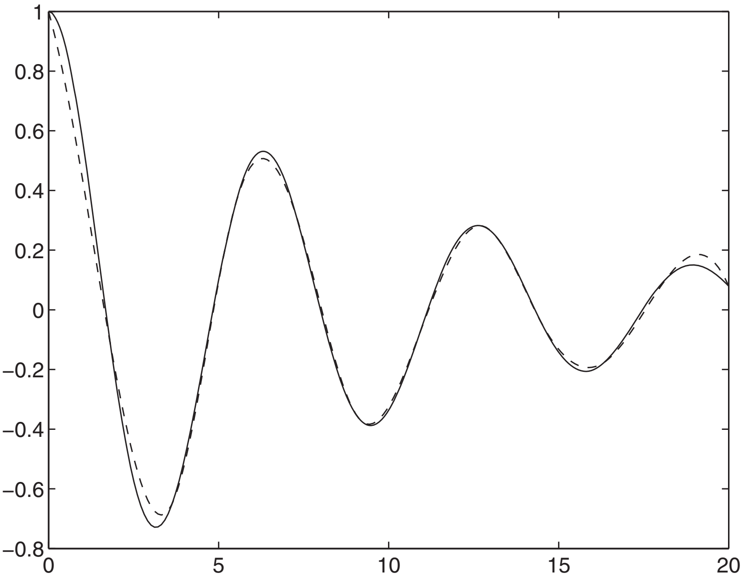


Figure 5.2 Comparison of a damped sine wave (solid line) with an approximation (dashed line) obtained with a spline using twelve equally spaced points in the range $0 \le \tau \le 20$.

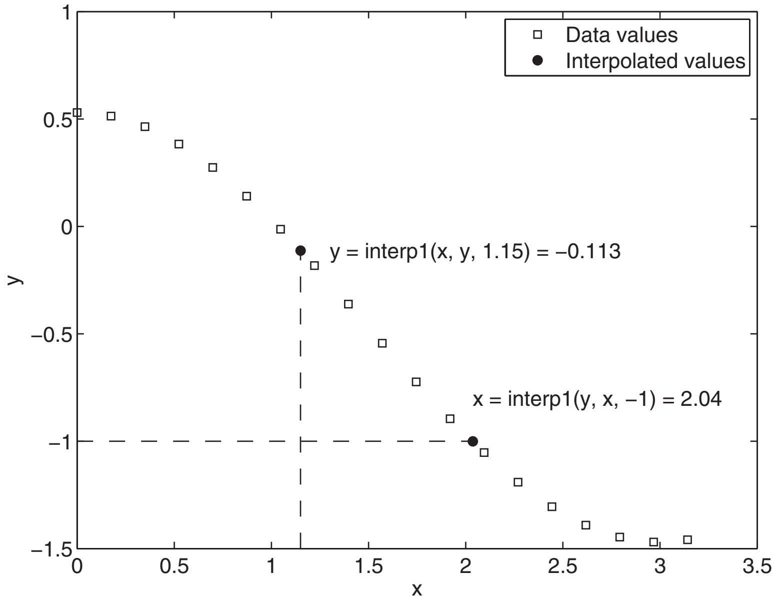


Figure 5.3 Two different usages of `interp1`.

5.4.4 Interpolation of Data—`interp1`

To approximate the location of a value that lies between a pair of data points, we must interpolate. The function that does this interpolation is

V = `interp1`(u, v, U)

where v is $v(u)$, u and v are vectors of the same length, and U is a scalar or vector of values for u for which V is desired. The array V has the same length as U and, in general, $u \neq U$.

There are two ways in which `interp1` can be used. If $y = f(x)$, then the first way that `interp1` can be used is to determine the value of y when x is given and the second way is to determine x when y is given. These two usages are summarized in Figure 5.3.

We shall now illustrate the use of `interp1`.

Example 5.3 First zero crossing of an exponentially decaying sine wave

Let us again create a data set for the exponentially decaying sine wave given by Eq. (5.2) and implemented with **DampedSineWave**. We are interested in approximating the first zero crossing from these data. From Figure 5.2, we see that this occurs before $\tau = 4.5$; that is, after this value we start to get very close to the second zero crossing. Thus, we shall create fifteen pairs of data values for the exponentially decaying sine wave in the range $0 \leq \tau \leq 4$ and use `interp1` to approximate the value of τ for which $f(\tau,\xi) \approx 0$. We assume that $\xi = 0.1$.

The script is

```
xi = 0.1;
tau = linspace(0, 4.5, 15);
data = DampedSineWave(tau, xi);
TauZero = interp1(data, tau, 0)
```

The execution of the script gives

```
TauZero =
  1.6817
```

The exact answer is obtained from Eq. (5.2) as

$$\tau = \frac{\pi - \varphi}{\sqrt{1-\xi^2}} = \frac{1}{\sqrt{1-\xi^2}}\left(\pi - \tan^{-1}\frac{\sqrt{1-\xi^2}}{\xi}\right) = 1.6794$$

5.4.5 Numerical Integration—trapz

One can obtain an approximation to a single integral in several ways. We shall introduce the function trapz in this section, which requires arrays of data. Another function, which requires the integrand to be in functional form, is quadl. This function is introduced in Section 5.5.2.

We start with

```
Area = trapz(x, y)
```

In this case, one specifies the values of *x* and the corresponding values of *y* as arrays. The function then performs the summation of the product of the average of adjacent *y* values and the corresponding *x* interval separating them.

Example 5.4 Area of an exponentially decaying sine wave

By using trapz, we shall determine the net area about the *x*-axis of the exponentially decaying sine wave given by Eq. (5.2) and plotted in Figure 5.2 and then show how to obtain the individual contributions of the positive and negative portions in the region $0 \le \tau \le 20$.

The script to obtain the area of the damped sine wave for $\xi = 0.1$ and for 200 data points is

```
xi = 0.1;
tau = linspace(0, 20, 200);
ftau = DampedSineWave(tau, xi);
Area = trapz(tau, ftau)
```

Execution of this script gives

```
Area =
  0.3021
```

To determine the area of the positive and negative portions of the waveform is a little more complicated. We start in the same way that we did to determine the total area. We first generate 200 equally spaced values of τ and obtain the corresponding values of $f(\tau, \xi)$ for $\xi = 0.1$ using **DampedSineWave**. To separate the positive and negative values, we create a logical structure and use it to set to zero all the negative values in one case and the positive values in another case. Thus, we modify the previous script as follows:

```
xi = 0.1;
tau = linspace(0, 20, 200);
ftau = DampedSineWave(tau, xi);
PosArea = trapz(tau, ftau.* (ftau >= 0));
NegArea = trapz(tau, ftau.* (ftau < 0));
disp(['Positive area = ' num2str(PosArea)])
disp(['Negative area = ' num2str(NegArea)])
disp(['Net area = ' num2str(PosArea+NegArea)])
```

Upon execution, we obtain

```
Positive area = 2.9549
Negative area = -2.6529
Net area = 0.30207
```

We see that the net area agrees very closely with the value previously obtained.

Example 5.5 Length of a line in space

Consider the following integral from which the length of a line in space can be approximated:

$$L = \int_a^b \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2 + \left(\frac{dz}{dt}\right)^2}\, dt \approx \sum_{i=1}^{N} \sqrt{(\Delta_i x)^2 + (\Delta_i y)^2 + (\Delta_i z)^2}$$

where

$$\Delta_i x = x(t_{i+1}) - x(t_i)$$
$$\Delta_i y = y(t_{i+1}) - y(t_i)$$
$$\Delta_i z = z(t_{i+1}) - z(t_i)$$

and $t_1 = a$ and $t_{N+1} = b$.

To illustrate the approximation to L, we choose the specific parametric relations

$$x = 2t$$
$$y = t^2$$
$$z = \ln t$$

for $1 \le t \le 2$ and assume that $N = 25$. The quantities $\Delta_i x$, $\Delta_i y$, and $\Delta_i z$, can each be evaluated with[3]

```
q = diff(x)
```

[3] As indicated in Section 1.4, `diff` is also used by the Symbolic toolbox to differentiate a symbolic function.

which computes the difference between successive elements of a vector; that is, for a vector $x = [x_1 x_2 \ldots x_n]$, a vector q with $n - 1$ elements of the form

$$q = [x_2 - x_1, x_3 - x_2, \ldots, x_n - x_{n-1}]$$

is created. For a vector x, `diff` is simply

```
q = x(2:end)-x(1:end-1);
```

The script to determine L is

```
t = linspace(1, 2, 25);
L = sum(sqrt(diff(2*t).^2+diff(t.^2).^2+diff(log(t)).^2))
```

which upon execution yields

```
L =
  3.6931
```

5.4.6 Area of a Polygon—polyarea

One can obtain the area of an n-sided polygon, where each side of the polygon is represented by its end points, with the use of

```
polyarea(x, y)
```

where x and y are vectors of the same length that contain the (x, y) coordinates of the endpoints of each side of the polygon. Each (x, y) coordinate pair is the end point of two adjacent sides of the polygon; hence, for an n-sided polygon, we need $n + 1$ pairs of end points.

We shall illustrate the use of this function by determining the area of a polygon whose vertices lie on the ellipse given by the parametric equations

$$x = a \cos\theta$$
$$y = b \sin\theta$$

If we let $a = 2$, $b = 5$, and we create a polygon with ten sides, then the script is

```
a = 2;   b = 5;
t = linspace(0, 2*pi, 11);
x = a*cos(t);
y = b*sin(t);
disp(['Area = ' num2str(polyarea(x, y))])
```

Upon execution of this script, we obtain

```
Area = 29.3893
```

It is noted that the area of the ellipse is $\pi ab = 10\pi = 31.42$.

5.4.7 Digital Signal Processing—`fft` and `ifft`

Discrete Fourier Transform

The Fourier transform of a real function $g(t)$ that is sampled every Δt over an interval $0 \le t \le T$ can be approximated by its discrete Fourier transform

$$G_n = G(n\Delta f) = \Delta t \sum_{k=0}^{N-1} g_k e^{-j2\pi nk/N} \quad n = 0, 1, ..., N - 1$$

where, as shown in Figure 5.4, $g_k = g(k\Delta t)$, $\Delta f = 1/T$, $T = N\Delta t$, and N is the number of samples. In general, G_n is a complex quantity. The restriction on Δt is that

$$\alpha\Delta t < \frac{1}{f_h}$$

where f_h is the highest frequency in $g(t)$ and $\alpha > 2$. The quantity G_n is called the amplitude density of $g(t)$ and has the units amplitude-second or, equivalently, amplitude/Hz. The inverse transform is approximated by

$$g_k = \Delta f \sum_{n=0}^{N-1} G_n e^{j2\pi nk/N} \quad k = 0, 1, ..., N - 1$$

In order to estimate the magnitude of the amplitude A_n corresponding to each G_n at its corresponding frequency $n\Delta f$, one multiplies G_n by Δf. Thus,

$$A_n = \Delta f G_n$$

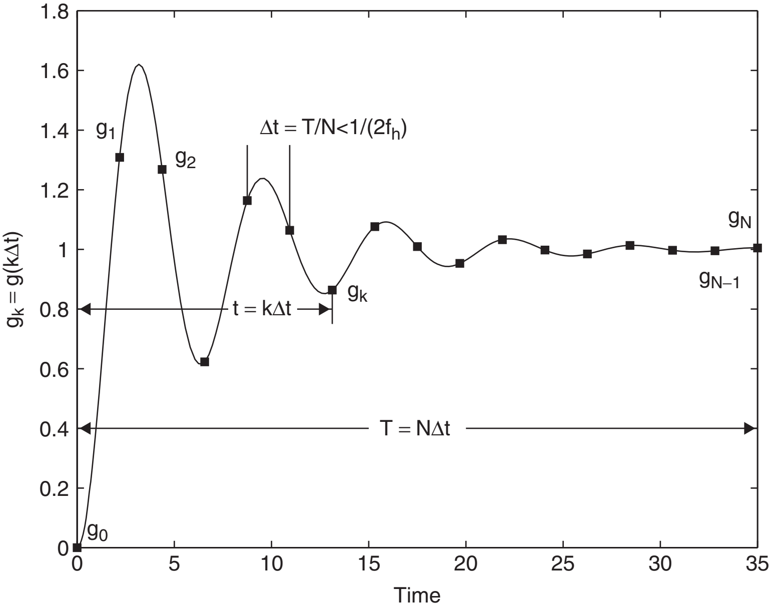


Figure 5.4 Sampled waveform.

and therefore

$$A_n = \frac{1}{N}\sum_{k=0}^{N-1} g_k e^{-j2\pi nk/N} \qquad n = 0, 1, \ldots, N-1$$

since $\Delta f \Delta t = 1/N$. The average power in the signal is

$$P_{avg} = \sum_{n=0}^{N-1} |A_n|^2$$

One often plots $|A_n|$ as a function of $n\Delta f$ to obtain an amplitude spectral plot. In this case, we have [4]

$$|A_n|_s = 2|A_n| \qquad n = 0, 1, \ldots, N/2 - 1$$

These expressions are best evaluated using the fast Fourier transform, which is a very efficient algorithm for numerically evaluating the discrete Fourier transform. It is most effective when the number of sampled data points is a power of two; that is, when $N = 2^m$, where m is a positive integer. The FFT algorithm is implemented with

G = `fft`(g, N)

and its inverse with

g = `ifft`(G, N)

where $G = G_n/\Delta t$ and $g = g_k/\Delta f$.

Weighting Functions

There are many situations when it is desirable to weight $g(t)$ by a suitable function to provide better resolution or other properties in the transformed domain. The procedure is to modify the original signal prior to performing the discrete Fourier transform in such a way that the effects of the changes caused by the windowing function to the signal's mean value and the signal's average power are removed. Thus, if the sampled values of the weighting function are $w_n = w(n\Delta t)$, then the corrected signal g_{cn} is given by[5]

$$g_{cn} = k_2 w_n (g_n - k_1) \qquad n = 0, 1, \ldots, N-1$$

where

$$k_1 = \sum_{n=0}^{N-1} w_n g_n \Big/ \sum_{n=0}^{N-1} w_n$$

[4] See, for example, J. S. Bendat and A. G. Piersol, *Engineering Applications of Correlation and Spectral Analysis*, John Wiley & Sons, New York, 1980.

[5] E. C. Ifeachor and B. W. Jervis, *Digital Signal Processing: A Practical Approach*, Addison-Wesley, Harlow, UK, 1993, p. 593.

corrects for the mean of the windowing function and

$$k_2 = \left[N \Big/ \sum_{n=0}^{N-1} w_n^2 \right]^{1/2}$$

corrects for the average power of the windowing function. One then takes the discrete Fourier transform of g_{cn}.

The Digital Signal Processing toolbox contains eight commonly used weighting functions.

Cross-Correlation

The cross-correlation of two functions $x(t)$ and $y(t)$ is given by

$$R_{xy}(\tau) = \int_{-\infty}^{\infty} x(t)y(t + \tau)dt \qquad -\infty < \tau < \infty$$

It is current practice to evaluate this quantity from the inverse Fourier transform of the cross-spectral density function $S_{xy}(\omega)$

$$R_{xy}(\tau) = F^{-1}\left[S_{xy}(\omega)\right]$$

where $F^{-1}[\dots]$ indicates the inverse Fourier transform and

$$S_{xy}(\omega) = X(\omega)Y^*(\omega)$$

The quantities $X(\omega)$ and $Y(\omega)$ are the Fourier transforms of $x(t)$ and $y(t)$, respectively, and the asterisk denotes the complex conjugate. To convert $R_{xy}(\tau)$ to its proper units requires that we multiply $S_{xy}(\omega)$ by $\Delta t = T/N$.

We shall illustrate these relationships with two examples.

Example 5.6 Fourier transform of a sine wave

Let us sample a sine wave of duration T. Thus,

$$g(t) = B_0 \sin(2\pi f_0 t) \quad 0 \le t \le T = 2^K/f_0 \qquad K = 0, 1, 2, \dots$$

and

$$t < \frac{1}{2f_0} \quad \text{or} \quad m - K > 1$$

since

$$f_h = f_0 = 2^K/T$$

and

$$\Delta t = 2^{-m}T$$

We assume that $g(t)$ is weighted by the Hamming function, which is given by

$$\begin{aligned} w(t) &= 0.54 - 0.46\cos(2\pi t/T) \qquad 0 \le t \le T \\ &= 0 \qquad \text{otherwise} \end{aligned}$$

The program to compute and plot the corrected weighted signal $g_c(t)$ and the amplitude spectrum A_n and display the average power in the signal, which is $P_{\text{avg}} = B_0^2/2$, is as follows: We assume that $A_o = 2.5$, $f_0 = 10$Hz, $K = 5$, and $m = 10$ ($N = 1024$).

```
k = 5;    m = 10;    fo = 10;    Bo = 2.5;
N = 2^m;    T = 2^k/fo;
ts = (0:N-1)*T/N;
df = (0:N/2-1)/T;
whamm = 0.54-0.46*cos(2*pi*ts/T);
SampledSignal = Bo*sin(2*pi*fo*ts);
k1 = sum(whamm.*SampledSignal)/sum(whamm);
k2 = sqrt(N/sum(whamm.^2));
CorrectedSignal = whamm.*(SampledSignal-k1)*k2;
figure(1)
plot(ts, CorrectedSignal)
figure(2)
An = abs(fft(CorrectedSignal, N))/N;
plot(df, 2*An(1:N/2))
disp(['Average power = ' num2str(sum(An.^2))])
```

Execution of the script results in Figures 5.5 and 5.6 and the following result being displayed to the command window:

```
Average power = 3.125
```

The MATLAB function `figure` is used to provide two separate figure windows, as discussed in Section 6.1. Notice in Figure 5.6 that the amplitude of the sine wave does not equal 2.5. This value is obtained, however, when the weighting function is removed.

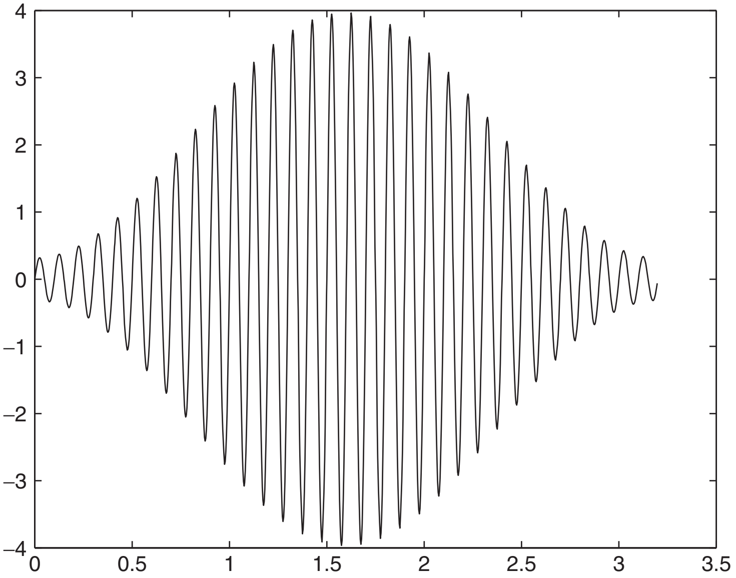


Figure 5.5 Sine wave modified by the Hamming weighting function.

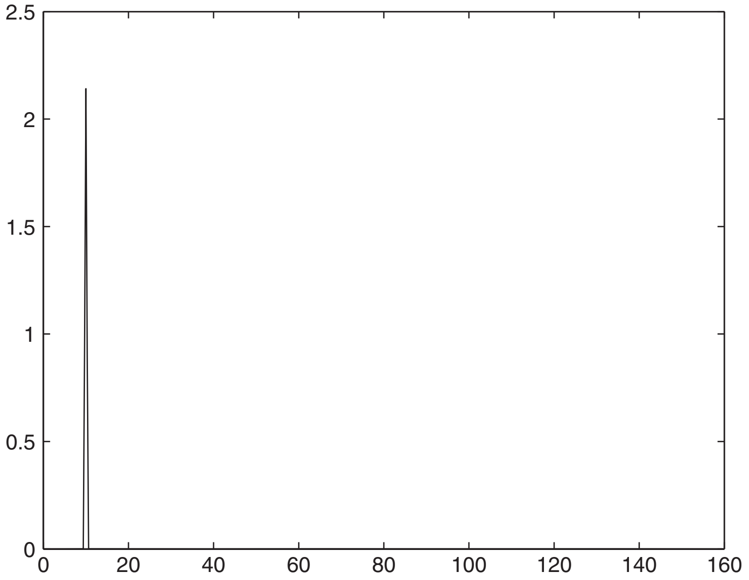


Figure 5.6 Amplitude spectrum of a sine wave using a Hamming weighting function.

Example 5.7 Cross-correlation of two pulses

We shall determine the cross-correlation function for the two rectangular pulses shown in Figure 5.7, which are expressed as

$$x(t) = A_x\,[u(t) - u(t - T_0)] \qquad t \geq 0$$

$$y(t) = A_y\,[u(t - T_1) - u(t - T_1 - T_2)] \qquad t \geq 0$$

where $u(t)$ is the unit step function. We assume that $A_x = A_y = 1$, $T_0 = 0.01$ s, $T_1 = 2T_o$, $T_2 = T_1 + T_o$, $T = T_2 + T_o$ and $N = 2^{10}$. The program is

```
To = 0.01;   T1 = 2*To;   T2 = T1+To;   Tend = T2+To;
N = 2^10;   deltaT = Tend/N;   Ax = 1;   Ay = 1;
t = linspace(0, Tend, N);
```

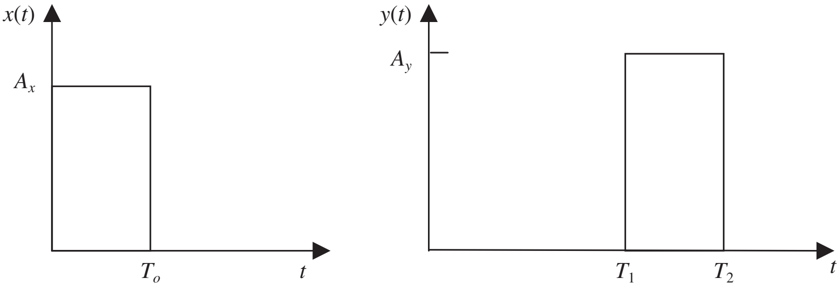


Figure 5.7 Two rectangular pulses.

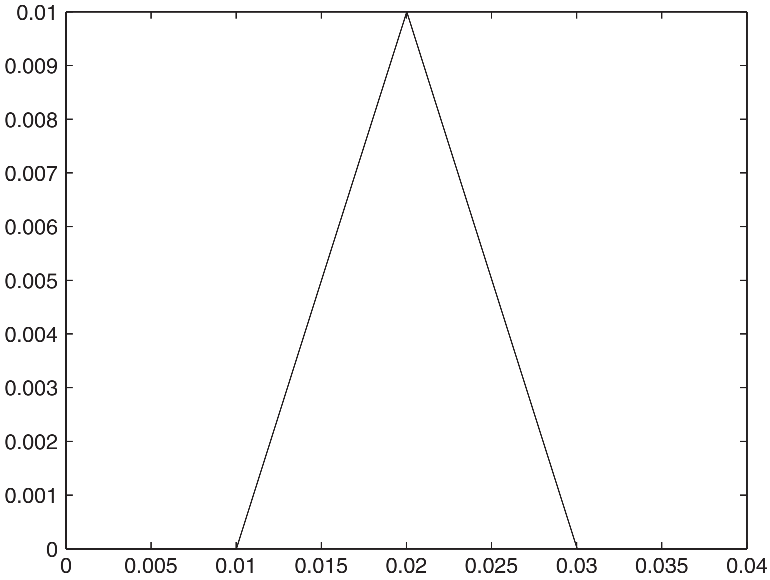


Figure 5.8 Cross-correlation of two rectangular pulses of equal duration.

```
PulseCrossCorr = inline('ampl*(((t-Ts)>=0)-((t-Te)>0))', 't', 'Ts', 'Te', 'ampl');
x = PulseCrossCorr(t, 0, To, Ax);
y = PulseCrossCorr(t, T1, T2, Ay);
X = fft(x, N);
Y = conj(fft(y, N));
Rxy = ifft(X.*Y*deltaT, N);
plot(t, real(Rxy))
```

Execution of the script results in Figure 5.8. The function `real` removes residual imaginary parts due to numerical round-off errors.

5.5 MATLAB FUNCTIONS THAT REQUIRE USER-DEFINED FUNCTIONS

MATLAB provides several functions that evaluate user-defined functions. The ones that we shall illustrate in this section are

`fzero`	Finds one root of $f(x) = 0$
`roots`	Finds the roots of a polynomial
`quadl`	Numerically integrates $f(x)$ in a specified interval
`dblquad`	Numerically integrates $f(x, y)$ in a specified region
`ode45`	Solves a system of ordinary differential equations with prescribed initial conditions

`bvp4c` Solves a system of ordinary differential equations with prescribed boundary conditions
`dde23` Solves delay differential equations with constant delays and with prescribed initial conditions
`fminbnd` Finds a local minimum of $f(x)$ in a specified interval
`fsolve` Numerically solves a system of nonlinear equations

The last function, `fsolve`, is from the Optimization toolbox.

When using these functions, the arguments of the user-defined functions and/or their output have to conform to specific requirements. These different requirements are illustrated in the following sections and are clearly specified in the *Help* file for that function.

5.5.1 Zeros of Functions—`fzero` and `roots/poly`

There are many equations for which an explicit algebraic solution to $f(x) = 0$ cannot be found. In these cases, a numerical procedure is required. The function `fzero` is a function that finds numerically one solution to the real function $f(x) = 0$ within a tolerance t_o in either the neighborhood of x_o or within the range $[x_1, x_2]$. It can also transfer $p_j, j = 1, 2, \ldots,$ parameters to the function defining $f(x)$. The function $f(x)$ must have a change of sign in the interval $[x_1, x_2]$, otherwise an error message will result. The general expression is

x = `fzero`(@**FunctionName**, x0, options, p1, p2, . . .)

where x is the value of x for which $f(x) \approx 0$, **FunctionName** is the name of the handle to a function file or subfunction, x0 = x_o or x0 = $[x_1 \, x_2]$, p1, p2, etc., are the parameters p_j required by **FunctionName**, and *options* is set by using

`optimset`

The function `optimset` is a general parameter-adjusting function that is used by several MATLAB functions, primarily from the Optimization toolbox. See the *Help* file for `optimset` for the types of attributes that can be altered; they are dependent on the MATLAB function selected.

The interface for the function required by `fzero` has the form

`function` z = **FunctionName**(x, p1, p2, . . .)
Expressions
z = . . .

where x is the independent variable that `fzero` is changing in order to find a value such that $f(x = z) \cong 0$. The independent variable must always appear in the first location. This requirement is true for most user-defined functions that are created for evaluation by MATLAB functions and is true for all functions illustrated in this chapter.

The function `fzero` can also be used with the anonymous function as

fhandle = @(x, p1, p2, . . .) (Expression);
z = `fzero`(**fhandle**, x0, options, p1, p2, . . .)

The function `fzero` can also be used with the `inline` function as

```
InlineFunctionName = inline('Expression', 'x', 'p1', 'p2', . . . );
z = fzero(InlineFunctionName, x0, options, p1, p2, . . . )
```

We shall now illustrate the use of `fzero` with the goal of pointing out how to avoid making a poor choice of $x0$. The function $f(x)$ can be either a MATLAB function or a user-defined function. Let us determine a root of $\cos(x)$ near $x = 2\pi$. Selecting a guess of $x0 = 2\pi$, the script

```
w = fzero(@cos, 2*pi)/pi
```

yields, upon execution,

```
w =
  1.5000
```

that is, $\cos(1.5\pi) = 0$. However, when we change the initial guess slightly to $x0 = 2.04\pi$, the script becomes

```
w = fzero(@cos, 2.04*pi)/pi
```

which upon execution yields

```
w =
  2.5000
```

However, when $x0 = 2.03\pi$, the script

```
w = fzero(@cos, 2.03*pi)/pi
```

upon execution yields

```
w =
  1.5000
```

Thus, for multiple-valued functions, one should use the form x0 = $[x_1 \ \ x_2]$ and specify the region explicitly. However, an error will result if the sign of $f(x_1)$ does not differ from the sign of $f(x_2)$. To show this, we rewrite the above script as

```
w = fzero(@cos, [0, 2*pi])/pi
```

When we execute this script, an error message is displayed saying that the values at the interval endpoints must differ in sign. However, when the interval is changed as given below,

```
w = fzero(@cos, [0.6*pi, 2*pi])/pi
```

we obtain upon execution

```
w =
  1.5000
```

Hence, for multi-valued functions whose properties are not known *a priori*, one should plot the function first to estimate the interval(s) where its zeros are or one should use the following function, which gives one way to determine the search regions for fzero automatically. This function determines the approximate locations of the change in signs in $f(x)$, and thereby obtains the region $[x_1, x_2]$. This method is presented as an M-file so that it may be used with an arbitrary $f(x)$. Its structure is similar to that used in **ManyZeros**, which was given in Section 5.3.

```
function Rt = FindZeros(FunName, Nroot, x, w)
f = feval(FunName, x, w);
indx = find(f(1:end-1).*f(2:end)<0);
L = length(indx);
if L<Nroot
   Nroot = L;
end
Rt = zeros(Nroot, 1);
for k = 1:Nroot
   Rt(k) = fzero(FunName, [x(indx(k)), x(indx(k)+1)], [], w);
end
```

The quantity x is a vector of N values of the independent variable for which *Nroot* zeros are expected over the range of x_1 to x_N. The quantity f is a vector of N values corresponding to $f(x)$ at each x. The spacing of x is a function of the expected closeness of the zeros; if unknown, a plot of the function will reveal this. The quantity w is a vector of parameters that are to be passed to **FunName**. It is important to note that in creating **FunName**, one must include w in its arguments even if it is not used. This is illustrated in Example 5.8.

Before giving some examples illustrating the use of fzero, we shall determine the root of $J_1(x) = 0$ near 3, where $J_1(x)$ is the Bessel function[6] of the first kind of order 1. The purpose of this illustration is to show that not all MATLAB functions can be used directly in fzero. The Bessel function is obtained from

```
besselj(n, x)
```

where n is the order (= 1 in this case) and x is the independent variable. We cannot use this function directly because the independent variable x is not the first variable in the function; n is. Hence, we create a new function using inline as follows:

```
besseljx = inline('besselj(n, x) ', 'x', 'n');
a = fzero(besseljx, 3, [], 1)
```

Upon execution, we obtain

```
a =
   3.8317
```

Notice that in order to transfer the parameter $p_1 = n = 1$ to the function *besseljx*, we had to place a value of 1 in the fourth location of fzero. In addition, since the third location of fzero is expecting the selection of an option and we are only using

[6] See, for example, Hildebrand, *Advanced Calculus for Applications*.

the default values, we have to put a null vector in that location as a separator so that we can include a parameter in the fourth location.

If we were to use an anonymous function instead of inline, the script is

```
B = @(x, n) (besselj(n, x));
a = fzero(B, 3, [], 1)
```

roots

When $f(x)$ is a polynomial of the form

$$f(x) = c_1x^n + c_2x^{n-1} + \cdots + c_nx + c_{n+1}$$

its roots can more easily be found by using

```
r = roots(c)
```

where

$$c = [c_1, c_2, \ldots, c_{n+1}]$$

and r is a vector of real and/or complex numbers.

For example, if

$$f(x) = x^4 - 35x^2 + 50x + 24$$

then, the script to find all the roots of the polynomial is

```
r = roots([1, 0, -35, 50, 24])
```

where the 0 has to be included to represent the coefficient of x^3. Executing this script gives

```
r =
  -6.4910
   4.8706
  2.0000
  -0.3796
```

Notice that the roots do not come out in any particular order. To order them, one uses sort. Thus,

```
r = sort(roots([1, 0, -35, 50, 24]))
```

upon execution gives

```
r =
  -6.4910
  -0.3796
   2.0000
   4.8706
```

The inverse of roots is

```
c = poly(r)
```

which returns *c*, the polynomial's coefficients, and *r* is a vector of roots. Thus,

```
r = roots([1, 0, -35, 50, 24]);
c = poly(r)
```

upon execution, displays

```
c =
   1.0000    0.0000    -35.0000    50.0000    24.0000
```

Polynomials can also be multiplied by using

```
h = conv(a, b)
```

where *a* and *b* are vectors containing the coefficients of the respective polynomials. For example, suppose we had, in addition to *f*(*x*), another polynomial

$$g(x) = x^2 - 4$$

Then, the product $h(x) = g(x)f(x)$ is obtained from

```
h = conv([1, 0, -4], [1, 0, -35, 50, 24])
```

which upon execution results in

```
h =
   1    0    -39    50    164    -200    -96
```

Thus, the resultant polynomial is

$$h(x) = x^6 - 39x^4 + 50x^3 + 164x^2 - 200x - 96$$

We now present several examples of the use of `fzero`.

Example 5.8 Lowest five natural frequency coefficients of a clamped beam

The characteristic equation from which the natural frequency coefficients Ω of a thin beam clamped at each end is given by

$$\cos(\Omega)\cosh(\Omega) - 1 = 0$$

We use this equation to determine the lowest five roots greater than zero using **FindZeros**. We will determine them two ways: with `inline` and with an anonymous function. The script using `inline` is

```
qcc = inline('cos(x).*cosh(x)-1', 'x', 'w');
x = linspace(0.1, 20, 50);
q = FindZeros(qcc, 5, x, []);
disp('Lowest five natural frequency coefficients are:')
disp(num2str(q'))
```

Upon execution, the following is displayed to the command window:

```
Lowest five natural frequency coefficients are:
4.73004    7.8532    10.9956    14.1372    17.2788
```

The script using an anonymous function is

```
qcc = @(x, w) (cos(x).*cosh(x)-1);
x = linspace(0.1, 20, 50);
q = FindZeros(qcc, 5, x, []);
disp('Lowest five natural frequency coefficients are:')
disp(num2str(q'))
```

Example 5.9 Zero of a function expressed as a series

We shall determine the value of *a* that satisfies the series equation

$$\sum_{j=1}^{1000} \frac{1}{j^2 - a} = 0$$

and display its annotated value to the command window. We shall use an initial guess of $\pi/2$. The script is

```
suma = inline('sum(1./([1:1000].^2-a))', 'a');
fofa = fzero(suma, pi/2);
disp('The value of a is ' num2str(fofa)])
```

Upon execution, we obtain

```
The value of a is 2.0466
```

5.5.2 Numerical Integration—`quadl` and `dblquad`

The function `quadl` numerically integrates a user-defined function *f*(*x*) from a lower limit *a* to an upper limit *b* to within a tolerance t_o. It can also transfer p_j parameters to the function defining *f*(*x*). The general expression for `quadl` is

```
A = quadl(@FunctionName, a, b, t0, tc, p1, p2, . . . )
```

where **FunctionName** is a function or a subfunction, a = a, b = b, t0 = t_o (when omitted, the default value is used), p1, p2, etc., are the parameters p_j, and when tc ≠ [], `quadl` provides intermediate output. When the function is created by `inline` or is an anonymous function then

```
A = quadl(IorAFunctionName, a, b, t0, tc, p1, p2, . . . )
```

where **IorAFunctionName** is the name of the inline function or the anonymous function.

The interface for the user-defined function has the form

```
function z = FunctionName(x, p1, p2, . . . )
Expression
z = . . .
```

where x is the independent variable that `quadl` is integrating over. The independent variable must always appear in this location. The interface for `inline` is

IorAFunctionName = `inline` ('Expression', 'x', 'p1', 'p2', . . .)

and that for an anonymous function is

IorAFunctionName = @(x, p1, p2, . . .) (Expression)

In Section 1.4, we introduced `int` from the Symbolic toolbox as one way to evaluate an integral. For those integrals that `int` cannot obtain, one can use `quadl`.

We shall now illustrate the use of `quadl`.

Example 5.10 Determination of area and centroid

Two quantities that are frequently of interest in mechanics are the area of a two-dimensional shape and the location of its centroid. Let us assume that we have two curves $y_j = f_j(x), j = 1, 2$, and that the two curves intersect at x_1 and x_2. Then the area between the two intersection points of the curves is

$$A = \int dA = \int_{x_1}^{x_2} (y_2 - y_1)dx$$

and the location of the area's centroid with respect to the origin is

$$x_c = \frac{1}{A}\int xdA = \frac{1}{A}\int_{x_1}^{x_2} x(y_2 - y_1)dx$$

$$y_c = \frac{1}{A}\int ydA = \frac{1}{A}\int_{x_1}^{x_2} \tfrac{1}{2}(y_2 + y_1)dA = \frac{1}{2A}\int_{x_1}^{x_2} (y_2^2 - y_1^2)dx$$

Suppose that $y_2 = x + 2$ and $y_1 = x^2$, as are shown in Figure 5.9. It is straightforward to show that the intersections occur at $x_1 = -1$ and $x_2 = 2$. Performing the above integrations yield: $A = 4.5$, $x_c = 0.5$ and $y_c = 1.6$. We now repeat these calculations numerically.

Since the expressions for these curves are relatively simple, we use `inline` to represent them. The script is

```
Atop = inline('x+2', 'x');
Abot = inline('x.^2', 'x');
Area = quadl(Atop, -1, 2)-quadl(Abot, -1, 2)
Mxc = inline('x.*((x+2)-x.^2)', 'x');
Myc= inline('((x+2).^2-x.^4)/2', 'x');
xc = quadl(Mxc, -1, 2)/Area
yc = quadl(Myc, -1, 2)/Area
```

The execution of the script gives

```
Area =
  4.5000
xc =
  0.5000
yc =
  1.6000
```

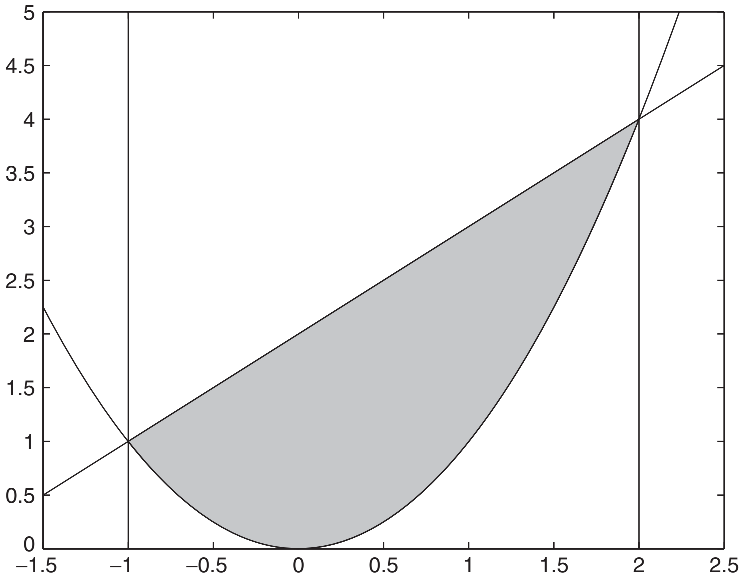


Figure 5.9 Shape for which the centroid and area are determined.

Example 5.11 Area of an exponentially decaying sine wave revisited

The damped sine wave is represented by **DampedSineWave**. If we again let $\xi = 0.1$ and integrate from $0 \le \tau \le 20$, then the script is

Area = `quadl`(@**DampedSineWave**, 0, 20, [], [], 0.1)

Upon execution, we obtain

```
Area =
  0.3022
```

which agrees with what was determined in Example 5.4.

Example 5.12 Response of a single degree-of-freedom system to a ramp force—numerical solution

The nondimensional response of a single degree-of-freedom system to a ramp force is[7]

$$y(\tau) = h(\tau)u(\tau) - h(\tau - \tau_0)u(\tau - \tau_0)$$

where $u(\tau)$ is the unit step function

$$h(\tau) = \frac{1}{\tau_0\sqrt{1 - \zeta^2}}\int_0^\tau f(t, \tau)dt \quad 0 \le \zeta < 1, \quad \tau \ge 0$$

and

$$f(t, \tau) = te^{-\zeta(\tau - t)} \sin\left[(\tau - t)\sqrt{1 - \zeta^2}\right]$$

[7] Balachandran and Magrab, *Vibrations,* pp. 311–312.

We shall determine $y(\tau)$ for $0 \leq \tau \leq 30$ when $\zeta = 0.1$ and $\tau_o = 15$ and plot the results. In addition, we shall determine the time at which $y(\tau) = 0.85$. The program is as follows:

```
function Example5_12
Nt = 100;   tzo = 15;   z = 0.1;   A = 0.85;
tau = linspace(0.1, 30, Nt);
yt = zeros(Nt,1);
for k = 1:Nt
   yt(k) = inq(tau(k), z, tzo);
   if tau(k)>tzo
      yt(k) = yt(k)-inq(tau(k)-tzo, z, tzo);
   end
end
plot(tau, yt, 'k-', [0, tzo, 30], [0, 1, 1], 'k—')
ri = interp1(yt, tau, A);
disp(['y(' num2str(ri,4) ') = ' num2str(A)])

function a = arg(t, tau, z)
a = t.*exp(-z*(tau-t)).*sin(sqrt(1-z^2)*(tau-t));

function in = inq(tau, z, tzo)
in = quadl(@arg, 0, tau, [], [], tau, z)/sqrt(1-z^2)/tzo;
```

Notice that we had to use `interp1` to determine the time at which $y(\tau) = 0.85$ because $y(\tau)$ is an array of numerical values.

Upon execution, we obtain the results shown in Figure 5.10 and the following appears in the command window:

```
y(12.99) = 0.85
```

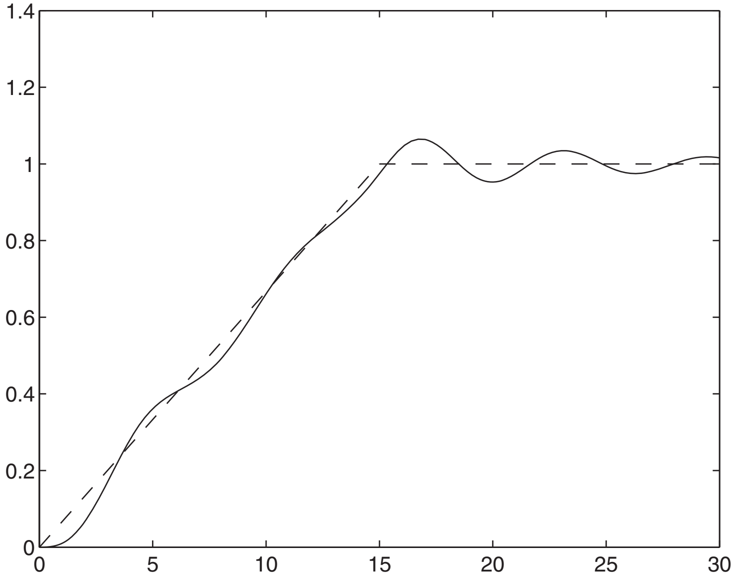


Figure 5.10 Response of a single degree-of-freedom system to a ramp forcing.

dblquad

The function dblquad numerically integrates a user-provided function $f(x, y)$ from a lower limit x_l to an upper limit x_u in the x-direction and from a lower limit y_l to an upper limit y_u in the y-direction to within a tolerance t_o. It can also transfer p_j parameters to the function defining $f(x, y)$. The general expression for dblquad is

dq = dblquad(@**FunctionName**, xl, xu, yl, yu, t0, meth, p1, p2, . . .)

where **FunctionName** is the name of the function M-file or a subfunction, xl $= x_l$, xu $= x_u$, yl $= y_l$, yu $= y_u$, $t0 = t_o$ (when omitted, the default value is used), p1, p2, etc., are the parameters p_j, and when meth = [], quadl is the method used. When an anonymous function is used or a function is created by inline, then

dq = dblquad(**IorAFunctionName**, xl, xu, yl, yu, t0, meth, p1, p2, . . .)

where **IorAFunctionName** is the name of the inline function or the anonymous function.

The interface for the function file or subfunction has the form

function z = **FunctionName**(x, y, p1, p2, ...)
Expressions
z = . . .

The interface for inline has the form

IorAFunctionName = inline('Expression', 'x', 'y', 'p1', 'p2', ...)

and that for the anonymous function is

IorAFunctionName = @(x, y, p1, p2, ...) (Expression)

We now illustrate the use of dblquad.

Example 5.13 Probability of two correlated variables

We shall numerically integrate the following expression for the probability of two random variables that are normally distributed over the region indicated:

$$P = \frac{1}{2\pi\sqrt{1 - r^2}} \int_{-2}^{2} \int_{-3}^{3} e^{-(x^2-2rxy+y^2)/2} dxdy$$

If we assume that $r = 0.5$, then the script is

```
r = 0.5;
Arg = @(x, y) (exp(-(x.^2-2*r*x.*y+y.^2)));
P = dblquad(Arg, -3, 3, -2, 2)/2/pi/sqrt(1-r^2)
```

Upon execution, we obtain

```
P =
   0.6570
```

5.5.3 Numerical Solutions of Ordinary Differential Equations—ode45

MATLAB can numerically solve several different types of systems of ordinary differential equations depending on whether one is solving an initial value problem or a boundary value problem. In the initial value problem, the conditions at $t = 0$ (or $x = 0$) are prescribed. In the boundary value problem, the conditions at both ends of the domain are prescribed, say, at $x = 0$ and $x = L$. The initial value problem is solved with

```
ode45
```

and the boundary value problem with

```
bvp4c
```

We shall discuss ode45 in this section and bvp4c in Section 5.5.4. A third type of ordinary differential equation that can be solved in MATLAB is called a delay differential equation; this equation is solved by using dde23 as discussed in Section 5.5.5. Lastly, a fourth type of equation is a one-dimensional parabolic-elliptic partial differential equation that is solved by using pdepe and is discussed in Section 5.5.6.

There are six additional ordinary differential equation solvers in MATLAB that can be used to solve initial value problems, each of which has its advantages depending on the particular properties of the differential equations. They are ode23, ode113, ode15s, ode23s, ode23t, and ode23tb. Their use is the same as described for ode45. See the MATLAB users guide and their respective *Help* files for details. MATLAB recommends that one start the solution process with ode45.

The function ode45 returns the numerical solution to a system of n first-order ordinary differential equations

$$\frac{dy_j}{dt} = f_j\,(t, y_1, y_2, ..., y_n) \quad j = 1, 2, ..., n$$

over the interval $t_0 \le t \le t_f$ subject to the initial conditions $y_j\,(t_0) = a_j$, $j = 1, 2, \ldots, n$, where a_j are constants. The arguments and outputs of ode45 are as follows:

[t, y] = ode45(@**FunctionName**, [t0, tf], [a1, a2, . . . , an], options, p1, p2, . . .)

where the output t is a column vector of the times $t_0 \le t \le t_f$ that are determined by ode45, the output y is the matrix of solutions such that the rows correspond to the times t and the columns correspond to the solutions; that is,

$$\begin{gathered} y_1(t) = y(:, 1) \\ y_2(t) = y(:, 2) \\ \cdots \\ y_n(t) = y(:, n) \end{gathered}$$

The first argument of ode45 is @**FunctionName**, which is a handle to either a function file or a subfunction. Its form must be as follows:

function yprime = **FunctionName**(t, y, p1, p2, . . .)

where t is the independent variable, y is a column vector whose elements correspond to y_j, $p1, p2, \ldots$ are parameters passed to **FunctionName**, and *yprime* is a column vector of length n whose elements are $f_j(t, y_1, y_2, \ldots, y_n)$, $j = 1, 2, \ldots, n$; that is,

$$yprime = [f_1; f_2; \ldots; f_n]$$

The variable names *yprime*, **FunctionName**, etc., are assigned by the programmer.

The second argument of `ode45` is a two-element vector giving the starting and ending times over which the numerical solution will be obtained. This quantity can, instead, be a vector of the times $[t_0\, t_1\, t_2 \ldots t_f]$ at which the solutions will be given. The third argument is a vector of initial conditions $y_j(t_o) = a_j$. The fourth argument, *options*, is usually set to null; however, if some of the solution method tolerances are to be changed, one does this with `odeset` (see the `odeset` *Help* file). The remaining arguments are those that are passed to **FunctionName**.

We now illustrate the usage of `ode45` by considering the nondimensional second-order ordinary differential equation with constant coefficients

$$\frac{d^2y}{dt^2} + 2\xi\frac{dy}{dt} + y = h(t) \tag{5.3}$$

which is subjected to the initial conditions $y(0) = a$ and $dy(0)/dt = b$. Equation (5.3) can be rewritten as a system of two first-order equations with the substitution

$$\begin{aligned} y_1 &= y \\ y_2 &= \frac{dy}{dt} \end{aligned}$$

Then, the system of equations is

$$\begin{aligned} \frac{dy_1}{dt} &= y_2 \\ \frac{dy_2}{dt} &= -2\xi y_2 - y_1 + h(t) \end{aligned}$$

with the initial conditions $y_1(0) = a$ and $y_2(0) = b$.

Let us consider the case where $\xi = 0.15$, $y(0) = 1$, $dy(0)/dt = 0$, and

$$\begin{aligned} h(t) &= \sin(\pi t/5) \quad && 0 \le t \le 5 \\ &= 0 && t > 5 \end{aligned}$$

We are interested in the solution over the region $0 \le t \le 35$. Then, $y_1(0) = 1$, $y_2(0) = 0$, $t_0 = 0$, and $t_f = 35$. We solve this system by creating a primary function called **Exampleode** and a subfunction called **HalfSine**. Then the program to solve the system of ordinary differential equations and to plot $y(t) = y_1(t)$ is

```
function Exampleode
[t, yy] = ode45(@HalfSine, [0 35], [1 0], [], 0.15);
plot(t, yy(:,1))
```

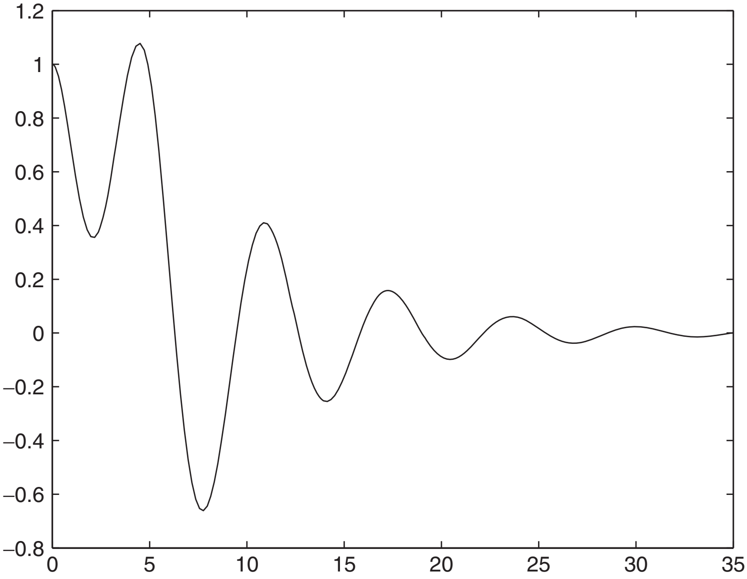


Figure 5.11 Response of Eq. (5.3) to the initial conditions $y(0) = 1$ and $dy(0)/dt = 0$, and when $h(t)$ is a half sine wave.

```
function y = HalfSine(t, y, z)
h = sin(pi*t/5).*(t<=5);
y = [y(2); -2*z*y(2)-y(1)+h];
```

The results are shown in Figure 5.11. It is to be noted that $yy(:, 1) = y_1(t) = y(t)$ and $yy(:, 2) = y_2(t) = dy/dt$.

We now give several additional examples that show the wide range of problems that ode45 can solve.

Example 5.14 Natural convection along a heated vertical plate

The equations describing the natural convection along a heated vertical plate in contact with a cooler fluid is given by (see Section 12.2.3)

$$\frac{d^3f}{d\eta^3} + 3f\frac{d^2f}{d\eta^2} - 2\left(\frac{df}{d\eta}\right)^2 + T^* = 0$$

$$\frac{d^{2^*}T^*}{d\eta^2} + 3\mathrm{Pr}\,f\frac{dT^*}{d\eta} = 0$$

where Pr is the Prandtl number. When Pr $= 0.7$, the initial conditions at $\eta = 0$ are

$$f = 0 \qquad \frac{df}{d\eta} = 0 \qquad \frac{d^2f}{d\eta^2} = 0.68$$

$$T^* = 1 \qquad \frac{dT^*}{d\eta} = -0.50$$

This coupled system of equations can be decomposed into a system of five first-order equations by introducing the following set of dependent variables:

$$y_1 = f \qquad y_4 = T^*$$

$$y_2 = \frac{df}{d\eta} \qquad y_5 = \frac{dT^*}{d\eta}$$

$$y_3 = \frac{d^2f}{d\eta^2}$$

where y_1 is the stream function, y_2 is the velocity, y_3 is the shear in the fluid stream, y_4 is the temperature, and y_5 is the heat flux. The system of first-order differential equations in terms of these new variables is

$$\frac{dy_1}{d\eta} = y_2 \qquad \frac{dy_4}{d\eta} = y_5$$

$$\frac{dy_2}{d\eta} = y_3 \qquad \frac{dy_5}{d\eta} = -3\ \mathrm{Pr}\ y_1 y_5$$

$$\frac{dy_3}{d\eta} = 2y_2^2 - 3y_1y_3 - y_4$$

and the corresponding initial conditions at $\eta = 0$ are

$$y_1(0) = 0 \quad y_4(0) = 1$$

$$y_2(0) = 0 \quad y_5(0) = -0.50$$

$$y_3(0) = 0.68$$

To solve this system of equations, we create the subfunction **NaturalConv** to specify the column vector representing the right-hand side of the five first-order differential equations. Assuming that $0 \le \eta \le 5$, the program is

```
function Example5_14
y0 = [0, 0, 0.68, 1, -0.50];
Pr = 0.7;
[eta ff] = ode45(@NaturalConv, [0 5], y0, [], Pr);

function ff = NaturalConv(x, y, Pr)
ff = [y(2); y(3); -3*y(1)*y(3)+2*y(2)^2-y(4); y(5); -3*Pr*y(1)*y(5)];
```

The results from the execution of this program are shown in Figure 12.15.

Example 5.15 Pendulum absorber

Consider the pendulum absorber shown in Figure 5.12. It can be shown that the nondimensional equations governing the motion of the system are[8]

$$\ddot{z} + 2\zeta_x\dot{z} + z + m_r\,[\varphi\sin\varphi + \varphi^2\cos\varphi] = f_0\cos\Omega\tau$$

$$\ddot{\varphi} + 2\zeta_t\dot{\varphi} + (\omega_\mathrm{r}^2 + \ddot{z})\sin\varphi = 0$$

[8] *Ibid*, p. 515.

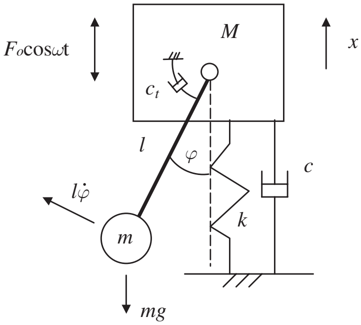


Figure 5.12 Pendulum absorber.

where

$$z = \frac{x}{l}, \quad \tau = \omega_x t, \quad \omega_r = \frac{\omega_\varphi}{\omega_x}, \quad \Omega = \frac{\omega}{\omega_x}, \quad f_0 = \frac{F_o}{(M+m)l\omega_x^2}, \quad \omega_\varphi = \sqrt{\frac{g}{l}}$$

$$2\zeta_x = \frac{c}{(M+m)\omega_x}, \quad 2\zeta_t = \frac{c_t}{ml^2\omega_x}, \quad m_r = \frac{m}{(M+m)}, \quad \omega_x = \sqrt{\frac{k}{m+M}}$$

and the dot indicates the derivative with respect to τ, g is the gravitational constant, c and c_t are the values of damping, k is the spring constant, m and M are mass of the pendulum and the main mass, respectively. To put these equations in the form of a system of first-order equations, we set

$$\begin{aligned} x_1 &= z \quad & x_3 &= \varphi \\ x_2 &= \dot{z} \quad & x_4 &= \dot{\varphi} \end{aligned}$$

Thus,

$$\dot{x}_1 = x_2 \quad \dot{x}_3 = x_4$$

To obtain the remaining two first-order equations, we use these results in the original equations to arrive at the following system of coupled equations in matrix form:

$$\begin{bmatrix} 1 & m_r \sin x_3 \\ \sin x_3 & 1 \end{bmatrix} \begin{Bmatrix} \dot{x}_2 \\ \dot{x}_4 \end{Bmatrix} = \begin{Bmatrix} A \\ B \end{Bmatrix}$$

where

$$\begin{Bmatrix} A \\ B \end{Bmatrix} = \begin{Bmatrix} f_0 \cos \Omega\tau - 2\zeta_x x_2 - x_1 - x_4^2 m_r \cos x_3 \\ - 2\zeta_t x_4 - \omega_r^2 \sin x_3 \end{Bmatrix}$$

Solving for $\dot{x}_2$ and $\dot{x}_4$, we obtain

$$\dot{x}_2 = \frac{A - Bm_r \sin x_3}{1 - m_r \sin^2 x_3}$$

$$\dot{x}_4 = \frac{B - A \sin x_3}{1 - m_r \sin^2 x_3}$$

For systems of practical interest, m_r is much less than 1 and, therefore, there are no singularities in these quantities.

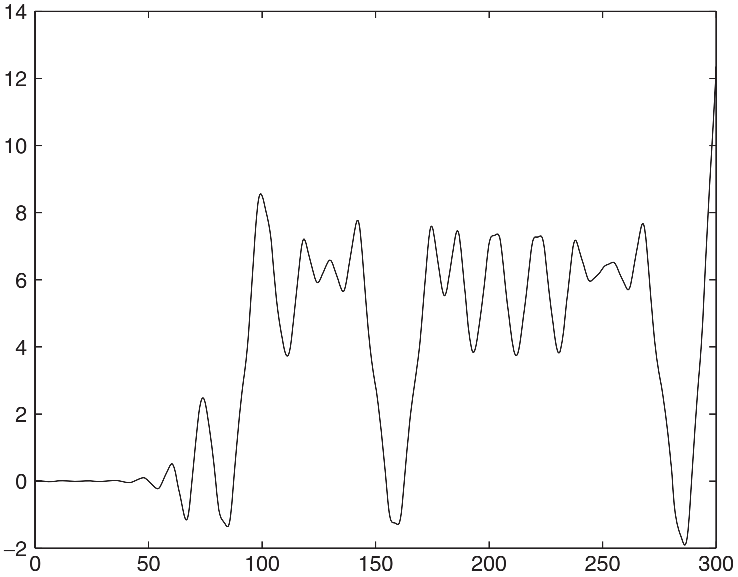


Figure 5.13 Angular rotation of the pendulum of a pendulum absorber.

We shall obtain a solution for the case when $\Omega = 1, \omega_r = 0.5, \zeta_x = 0.05$, $\zeta_t = 0.005, m_r = 0.05, f_o = 0.03$, and for $0 \leq \tau \leq 300$. We shall plot the angular rotation φ when $\varphi(0) = x_3 = 0.02$ rad. In addition, the subfunction for the interface required by `ode45` is called **PendulumAbsorber**. The program is

```
function Example5_15
mr = 0.05;    zx = 0.05;    zt = 0.005;
Om = 1;    wr = 0.5;    fo = 0.03;
[t w] = ode45(@PendulumAbsorber, [0 300], [0 0 0.02 0],[], mr, zx, zt, Om, wr, fo);
plot(t, w(:,3))

function Q = PendulumAbsorber(t, w, mr, zx, zt, Om, wr, fo)
A = fo*cos(Om*t)-w(1)-2*zx*w(2)-mr*w(4)^2*cos(w(3));
B = -2*zt*w(4)-wr^2*sin(w(3));
x4dot = (B-A*sin(w(3)))/(1-mr*sin(w(3))^2);
x2dot = (A-mr*B*sin(w(3)))/(1-mr*sin(w(3))^2);
Q = [w(2); x2dot; w(4); x4dot];
```

A plot of φ obtained from the execution of this program is given in Figure 5.13.

5.5.4 Numerical Solutions of Ordinary Differential Equations—bvp4c

The MATLAB function `bvp4c` obtains numerical solutions to the two-point boundary value problem. However, unlike `ode45, bvp4c` requires the use of several additional MATLAB functions that were specifically created to assist in initializing `bvp4c` (`bvpinit`) and in smoothing its output for plotting (`deval`). In addition, several user-defined functions are needed to implement `bvp4c`.

The function bvp4c returns the numerical solution to a system of n first-order ordinary differential equations

$$\frac{dy_j}{dx} = f_j(x, y_1, y_2, \ldots, y_n, q) \qquad j = 1, 2, \ldots, n$$

over the interval $a \leq x \leq b$ subject to the boundary conditions $y_j(a) = a_j$, $j = 1, 2, \ldots, J$ and $y_k(b) = b_k$, $k = 1, 2, \ldots, K$, where $J + K = n$; a_j and b_k are constants; and q is a vector of unknown parameters that are to be determined by bvp4c. The arguments and outputs of bvp4c are

sol = bvp4c(@**FunctionName**, @**BCFunction**, solinit, options, p1, p2 . . .)

where *sol* is a structure[9] that contains the solution at a specific number of points that have been determined as part of the solution method used in bvp4c. In order to obtain a smooth curve, values at additional intermediate points are needed. To provide these additional points, we use

sxint = deval(sol, xint)

where *xint* is a vector of locations at which the solution is to be evaluated and *sol* is the output of bcp4c. The output *sxint* is an array containing the values of y_j at the spatial location*s xint*; that is,

y_1(*xint*) = *sxint*(1, :)
y_2(*xint*) = *sxint* (2, :)
. . .
y_n(*xint*) = *sxint* (*n*, :)

The user-defined function **FunctionName** requires the following interface:

function dydx = **FunctionName**(x, y, p1, p2, ...)

where *x* is a scalar corresponding to *x*, *y* is a column vector of f_j, and p1, p2, etc., are known parameters that are needed to define f_j. The output *dxdy* is a column vector. The fourth argument, *options*, is usually set to null; however, if some of the solution method tolerances are to be changed, one does this with

options = odeset(arguments)

where the appropriate values for *arguments* are found in the help file for odeset.

The function **BCFunction** contains the boundary conditions $y_j(a) = a_j$ and $y_k(b) = b_k$ and requires the following interface:

function Res = **BCFunction**(ya, yb, p1, p2, ...)

where *ya* is a column vector of $y_j(a)$ and *yb* is a column vector of $y_j(b)$. The known parameters p1, p2, etc., must appear in this interface, even if the boundary conditions do not require them. The output *Res* is a column vector.

[9] A structure will be explained by example subsequently.

The variable *solinit* is a structure obtained from the function `bvpinit` as follows:

```
solinit = bvpinit(x, y)
```

The vector x is a guess for the initial mesh points that the solution method in `bvp4c` should initially use. The vector y is a guess of the magnitude of each of the y_j; they can be constants or functions of x. The lengths of x and y are independent of each other.

To illustrate the use of `bvp4c`, consider the following equation:

$$\frac{d^2y}{dx^2} - kxy = 0$$

subject to the boundary conditions

$$y(0) = 0.1$$
$$y(1) = 0.05$$

First, we transform the equation into a pair of first-order differential equations with the substitutions

$$y_1 = y$$
$$y_2 = \frac{dy}{dx}$$

to obtain

$$\frac{dy_1}{dx} = y_2$$
$$\frac{dy_2}{dx} = kxy_1$$

The boundary conditions for this formulation are

$$y_1(0) = 0.1$$
$$y_1(1) = 0.05$$

We now proceed to create the required primary function and subfunctions. The subfunction that expresses the system of first-order ordinary differential equations is called **OdeBvp** and the function that records the boundary conditions is called **OdeBC**. The set of initial guesses is given in `bvpinit`, where it is seen that we have selected five points between 0 and 1 and assumed that the solution for y_1 has a constant magnitude of -0.05 and that for y_1 has a constant magnitude of 0.1. Then, assuming that $k = 100$, the script is

```
function bvpExample
k = 100;
solinit = bvpinit(linspace(0, 1, 5), [-0.05, 0.1]);
exmpsol = bvp4c(@OdeBvp, @OdeBC, solinit, [], k);
x = linspace(0, 1, 50);
y = deval(exmpsol, x);
```

```
plot(x, y(1, :))

function dydx = OdeBvp(x, y, k)
dydx = [y(2); x*k*y(1)];

function res = OdeBC(ya, yb, k)
res = [ya(1)-0.1; yb(1)-0.05];
```

The results are plotted in Figure 5.14.

The outputs from the various functions in the above script are now discussed. The output *exmpsol* is a structure, which permits one to access the various quantities as follows. The structure *exmpsol*. $y(1, :)$ gives the values of y_1 at the mesh points given in the structure *exmpsol.x* and the structure *exmpsol*. $y(2, :)$ gives the values of y_2 at these same mesh points. All these quantities are generated by `bvp4c` after executing its computational procedure. In this case, the number of mesh points that `bvp4c` used was eighteen. On the other hand, the quantity y that is the output of `deval`, looks more like that which comes from `ode45`. The variable y, in this case, is a (2×50) array, where $y(1, :) = y_1 = y$ and $y(2, :) = y_2 = dy/dx$. If we plotted the eighteen values from the structure *exmpsol.y*$(1, :)$, these values would lie on the curve drawn in Figure 5.14.

We shall now give several examples of the use of `bvp4c`. In particular, we shall use an Euler beam as the means to show the wide range of solutions that can be obtained by examining the following cases: (1) an Euler beam with uniform loading; (2) an Euler beam with an overhang and uniform loading; (3) an Euler beam with a point load; (4) an Euler beam with different cross-sectional characteristics along its length; and (5) the determination of the lowest natural frequency coefficient of an Euler with prescribed

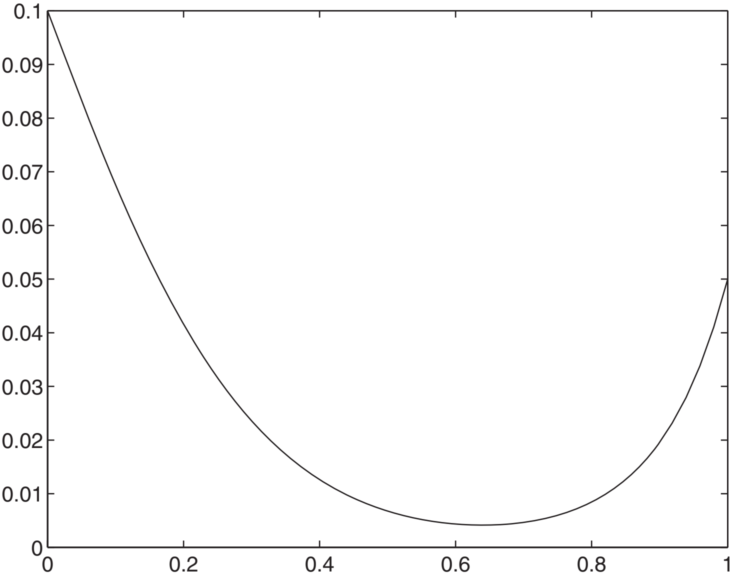


Figure 5.14 Solution for $y(x)$ from **bvpExample**.

boundary conditions.[10] Examples (2)–(5) each requires a different "trick" in order to obtain a solution and is the reason why these examples have been selected.

Since the next five examples refer to the Euler beam, we first provide the governing equation and boundary conditions and then convert these quantities to nondimensional form. The beam equation is given by

$$EI\frac{d^4w}{dx^4} = P_0q(x) \qquad 0 \le x \le L \tag{5.4}$$

where $w = w(x)$ is the transverse displacement, L is the length of the beam, E is the Young's modulus, I is the moment of inertia of the cross section, P_o is the magnitude of the applied load per unit length, and q is the shape of the load along the length of the beam. We will consider the following three different boundary conditions at each end of the beam:

Simply Supported (Hinged)

$$w = 0 \quad \text{and} \quad M = EI\frac{d^2w}{dx^2} = 0 \tag{5.5a}$$

Clamped

$$w = 0 \quad \text{and} \quad \frac{dw}{dx} = 0 \tag{5.5b}$$

Free

$$V = EI\frac{d^3w}{dx^3} = 0 \quad \text{and} \quad M = EI\frac{d^2w}{dx^2} = 0 \tag{5.5c}$$

where M is the moment, V is the shear force, and dw/dx is the slope.

If we introduce the following definitions,

$$\eta = x/L \quad y = y(\eta) = w/h_0 \quad \text{and} \quad h_0 = \frac{P_0L^4}{EI}$$

then Eq. (5.4) becomes

$$\frac{d^4y}{d\eta^4} = q(\eta) \quad 0 \le \eta \le 1 \tag{5.6}$$

and Eqs. (5.5a)–(5.5c) become, respectively,

Simply Supported (Hinged)

$$y = 0 \quad \text{and} \quad M_{nd} = \frac{M}{P_0L^2} = \frac{d^2y}{d\eta^2} = 0 \tag{5.7a}$$

[10] For additional examples involving a wide range of different differential equations, see L .F. Shampine, M. W. Reichelt, and J. Kierzenka, "Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with `bvp4c`," which is available at http://www.mathworks.com/support/solutions/files/s8314/bvp_paper.pdf.

Clamped

$$y = 0 \quad \text{and} \quad \frac{dy}{d\eta} = 0 \tag{5.7b}$$

Free

$$V_{nd} = \frac{V}{P_0 L} = \frac{d^3 y}{d\eta^3} = 0 \quad \text{and} \quad M_{nd} = \frac{M}{P_0 L^2} = \frac{d^2 y}{d\eta^2} = 0 \tag{5.7c}$$

We transform Eq. (5.6) to a series of first-order ordinary differential equations through the relations

$$y_1 = y \quad y_3 = \frac{d^2 y}{d\eta^2}$$

$$y_2 = \frac{dy}{d\eta} \quad y_4 = \frac{d^3 y}{d\eta^3}$$

Using these relations and the original differential equation, we obtain

$$\frac{dy_1}{d\eta} = y_2 \quad \frac{dy_3}{d\eta} = y_4$$

$$\frac{dy_2}{d\eta} = y_3 \quad \frac{dy_4}{d\eta} = q(\eta) \tag{5.8}$$

The boundary conditions given by Eqs. (5.7a)–(5.7c) in terms of these new variables become, respectively,

Simply Supported (Hinged)

$$y_1 = 0 \quad \text{and} \quad y_3 = 0 \tag{5.9a}$$

Clamped

$$y_1 = 0 \quad \text{and} \quad y_2 = 0 \tag{5.9b}$$

Free

$$y_3 = 0 \quad \text{and} \quad y_4 = 0 \tag{5.9c}$$

Example 5.16 Displacement of a uniformly loaded Euler beam

Let us consider a beam that is simply supported at each end, and at end $\eta = 1$, a nondimensional external moment M_r is applied; that is, $M_{nd} = M_r$. The boundary conditions at $\eta = 0$ are $y_1(0) = y_3(0) = 0$ and those at $\eta = 1$ are $y_1(1) = 0$ and $y_3(1) = M_r$. In addition, we apply a uniform load along the length of the beam; thus, $q(\eta) = 1$, and we assume that $M_r = 0.8$.

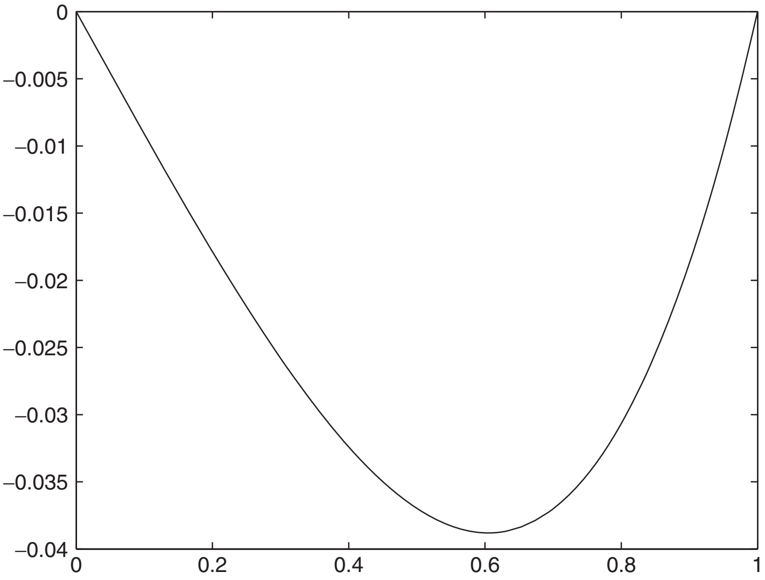


Figure 5.15 Displacement response to a uniformly loaded beam hinged at both ends with an external moment applied at $\eta = 1$.

To obtain a solution, we assume ten uniformly spaced mesh points and guess that the magnitudes of the displacement, slope, moment, and shear force each have the value of 0.5. We create **BeamODEqo** to represent the system of first-order equations and **BeamHingedBC** to represent the boundary conditions. Then the program is

```
function Example5_16
qo = 1;    Mr = 0.8;
solinit = bvpinit(linspace(0, 1, 10), [0.5, 0.5, 0.5, 0.5]);
beamsol = bvp4c(@BeamODEqo, @BeamHingedBC, solinit, [], qo, Mr);
eta = linspace(0, 1, 50);
y = deval(beamsol, eta);
plot(eta, y(1, :))

function dydx = BeamODEqo(x, y, qo, Mr)
dydx = [y(2); y(3); y(4); qo];

function bc = BeamHingedBC(y0, y1, qo, Mr)
bc = [y0(1); y0(3); y1(1); y1(3)-Mr];
```

The execution of this program results in the beam displacement shown in Figure 5.15.

Example 5.17 Displacement of a uniformly loaded Euler beam with an overhang

From Example 5.16, the governing equation for the nondimensional displacement y of a beam subjected to a nondimensional static load q is

$$\frac{d^4y}{d\eta^4} = q_1(\eta)$$

where $0 \leq \eta \leq 1$. In order to consider a beam with an overhang, we have to consider two beams: one that spans $0 \leq \eta \leq 1$ and one that spans $1 \leq \eta \leq b$, where $b > 1$. The governing equation for the nondimensional displacement w of the beam in the region $1 \leq \eta \leq b$ is

$$\frac{d^4w}{d\eta^4} = q_2(\eta)$$

At $\eta = 1$, both beams have to satisfy the continuity of displacements, slopes, moments, and shear forces. In order to be able to use `bvp4c`, we have to employ a "trick".[11] This "trick" involves a coordinate translation so that both beams span the same region $0 \leq \eta \leq 1$ after which one can consider the governing equations as a system of two differential equations coupled by their boundary conditions. For the first beam in the region $0 \leq \eta \leq 1$, we use the results of Example 5.16 directly to obtain

$$\frac{dy_1}{d\eta} = y_2 \qquad \frac{dy_3}{d\eta} = y_4$$
$$\frac{dy_2}{d\eta} = y_3 \qquad \frac{dy_4}{d\eta} = q_1$$

where y_1, y_2, y_3, and y_4, are the nondimensional displacement, slope, moment, and shear force, respectively.

The coordinate translation that we will employ for the second beam in the region $1 \leq \eta \leq b$ is $z = (\eta - 1)/(b - 1)$, where $0 \leq z \leq 1$. Thus,

$$\frac{d}{d\eta} = \frac{d}{dz}\frac{dz}{d\eta} = \frac{1}{b-1}\frac{d}{dz}$$

and, therefore, the four first-order equations for this system are

$$\frac{dy_5}{dz} = (b - 1)y_6 \qquad \frac{dy_7}{dz} = (b - 1)y_8$$
$$\frac{dy_6}{dz} = (b - 1)y_7 \qquad \frac{dy_8}{dz} = (b - 1)q_2$$

where y_5, y_6, y_7, and y_8, are the nondimensional displacement, slope, moment, and shear force, respectively, of the overhanging beam in the region $0 \leq z \leq 1$.

We assume that the beam is simply supported at $\eta = 0$, where the displacement and moment are zero and free at $\eta = b$, where the moment and shear force are zero. Thus, from Eqs. (5.9a) and (5.9c),

$$y_1(0) = 0 \quad y_7(1) = 0$$
$$y_3(0) = 0 \quad y_8(1) = 0$$

In addition, at $\eta = 1$, the displacements for both beams are zero and the slopes and moments of both beams are equal. Under these assumptions, the continuity conditions are

$$y_1(1) = 0 \qquad y_5(0) = 0$$
$$y_3(1) = y_7(0) \quad y_2(1) = y_6(0)$$

We shall assume that $b = 1.5$ and that both beams are uniformly loaded at the same magnitude; thus, $q_1 = q_2 = 1.0$. For the initial guesses, we shall assume twenty uniformly

[11] See Example 9 of Shampine et al., "Solving Boundary Value."

spaced mesh points and guess that the magnitudes of the displacement, slope, moment, and shear force of each beam have the value of 0.05. We create **BeamOverODEqo** to represent the system of first-order equations and **BeamOverBC** to represent the boundary conditions. The program is

```
function Example5_17
b = 1.5;    gues(1:8) = 0.05;
solinit = bvpinit(linspace(0, 1, 20), gues);
beamsol = bvp4c(@BeamOverODEqo, @BeamOverBC, solinit, [], b);
eta = linspace(0, 1, 50);
y = deval(beamsol, eta);
eet = [eta eta*(b-1)+1];
for k = 1:4
  subplot(2, 2, k)
  plot(eet, -[y(k,:) y(k+4,:)], 'k-')
  hold on
  plot([0 b], [0 0], 'k—')
end

function dydx = BeamOverODEqo(x, y, b)
dydx = [y(2); y(3); y(4); 1; (b-1)*y(6); (b-1)*y(7); (b-1)*y(8); b-1];

function bc = BeamOverBC(yL, yR, b)
bc = [yL(1); yL(3); yR(1); yR(3)-yL(7); yL(5); yR(2)-yL(6); yR(7); yR(8)];
```

Upon execution, we obtain the results plotted in Figure 5.16. The plotting statements within the for loop are discussed in Sections 6.1 and 6.2. Notice that in the plot

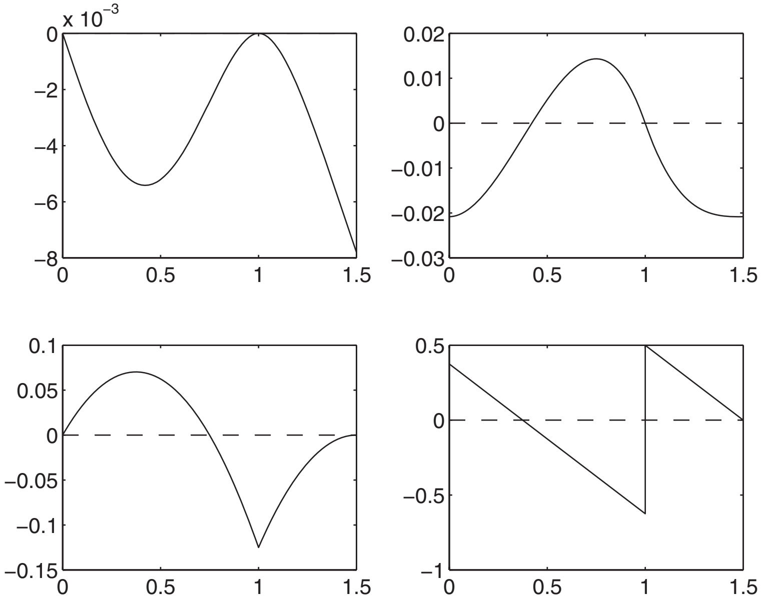


Figure 5.16 Displacement, slope, moment, and shear force for a uniformly loaded beam with an overhang.

statement there are two vectors: [eta eta*(b−1)+1], which are the locations along the beam and the vector [y(k,:) y(k+4,:)], which are the corresponding displacements, slopes, moments, and shear forces as determined by the index k. The first set of elements of these vectors is for the first beam for $0 \leq \eta \leq 1$ and the second set of vectors is for the second beam for $0 \leq z \leq 1$, or in terms of η for $1 \leq \eta \leq b$.

Example 5.18 Displacement of an Euler beam subjected to a point load

For a beam subjected to a point load of magnitude P_o located at $\eta = \eta_o$, the nondimensional equation becomes

$$\frac{d^4y}{d\eta^4} = \delta(\eta - \eta_0)$$

where $\delta(\eta)$ is the delta function. In order to solve this equation with a discontinuity at $\eta = \eta_o$, we have to make two "adjustments" to our solution procedure. The first adjustment is that we will have to approximate the point load with a uniform load that is distributed over a very small portion of the beam; that is, we will have to assume that P_o is a constant load that acts over the region $\eta_o - \varepsilon \leq \eta \leq \eta_o + \varepsilon$ where $\varepsilon << 1$ such that in this small region, $P_o \rightarrow P_o/(2\varepsilon)$. The second adjustment that we have to make is in our initial guess for the locations that bvp4c should start with. In order for the numerical procedure to "know" that the inhomogeneous term of the equation is nonzero over a very small region, we must ensure that our initial guess includes this region. To do this, we specifically include as part of our initial guess the three locations $\eta_o - \varepsilon$, η_o, and $\eta_o + \varepsilon$.

To illustrate this procedure, we assume that the beam is simply supported at both ends and that $\eta_o = 0.5$ and $\varepsilon = 0.005$. Then the program is

```
function Example5_18
e = 0.005;   etao = 0.5;
pts = [linspace(0, etao-2*e, 4), etao-e, etao, etao+e, linspace(etao+2*e, 1, 4)];
solinit = bvpinit(pts, [0.5, 0.5, 0.5, 0.5]);
beamsol = bvp4c(@BeamPointODEqo, @BeamPointHingedBC, solinit, [], etao, e);
eta = linspace(0, 1, 100);
y = deval(beamsol, eta);
for k = 1:4
  subplot(2, 2, k)
  plot(eta, y(k,:), 'k-')
  hold on
  plot([0 1], [0 0], 'k—')
end

function dydx = BeamPointODEqo(x, y, etao, e)
q = ((x > etao-e) & (x < etao+e))/(2*e);
dydx = [y(2); y(3); y(4); q];

function bc = BeamPointHingedBC(yL, yR, etao, e)
bc = [yL(1); yL(3); yR(1); yR(3)];
```

Upon execution, we obtain the results plotted in Figure 5.17.

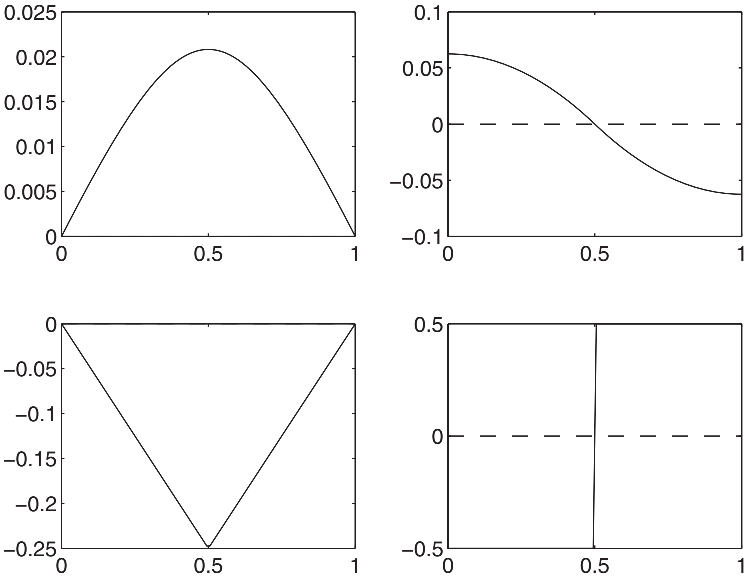


Figure 5.17 Displacement, slope, moment, and shear force for a beam with a point load at $\eta = 0.5$.

Example 5.19 Displacement of an Euler beam with a step change in cross section

Let us assume that we have a simply supported beam of length L and that the beam is composed of two sections, one of length L_1 and the other of length L_2 such that $L = L_1 + L_2$. Each section of the beam is subjected to a load per unit length $P_j q_j(x)$, where $j = 1, 2$ and P_j is the magnitude of the load. From Eq. (5.4), the governing equations of these two beams are, respectively,

$$E_1 I_1 \frac{d^4 w_1}{dx^4} = P_1 q_1(x) \qquad 0 \le x \le L_1$$

$$E_2 I_2 \frac{d^4 w_2}{dx^4} = P_2 q_2(x) \qquad L_1 \le x \le L$$

If we define $\eta = x/L$, $\beta = L_1/L$, and

$$\alpha = \frac{E_1 I_1 P_2}{E_2 I_2 P_1} \qquad h_o = \frac{P_1 L^4}{E_1 I_1}$$

then these equations can be written as

$$\frac{d^4 y_1}{d\eta^4} = q_1(\eta) \qquad 0 \le \eta \le \beta$$

$$\frac{d^4 y_2}{d\eta^4} = \alpha q_2(\eta) \qquad \beta \le \eta \le 1$$

where $y_j = w_j/h_o$.

To solve these equations, we again use the "trick" of translating the coordinates of the second beam so that it is in the same region as that of the first beam and then solving the two equations as a system of equations coupled by their boundary conditions. In this case, the coordinate translation relation is

$$z = \frac{\beta}{1-\beta}(\eta - \beta) \quad \beta \le \eta \le 1$$

so that $0 \le z \le \beta$. Therefore,

$$\frac{d}{d\eta} = \frac{\beta}{1-\beta}\frac{d}{dz} \quad \beta \le \eta \le 1$$

The system of first-order equations is

$$\begin{array}{llll}
\dfrac{dw_1}{d\eta} = w_2 & \dfrac{dw_5}{dz} = \dfrac{(1-\beta)}{\beta}w_6 \\
\dfrac{dw_2}{d\eta} = w_3 & \dfrac{dw_6}{dz} = \dfrac{(1-\beta)}{\beta}w_7 \\
\dfrac{dw_3}{d\eta} = w_4 & \dfrac{dw_7}{dz} = \dfrac{(1-\beta)}{\beta}w_8 \\
\dfrac{dw_4}{d\eta} = q_1 & \dfrac{dw_8}{dz} = \alpha\dfrac{(1-\beta)}{\beta}q_2
\end{array}$$

The continuity conditions at $\eta = \beta$ are that the displacements, slopes, moments, and shear forces are equal. The boundary conditions for a beam simply supported at $\eta = 0$ and $\eta = 1$ are that the displacements and moments are zero. We assume that $\alpha = 4.0$ and $\beta = 0.7$ and take 0.05 as an initial guess for the eight unknown quantities and select five equally spaced locations. The program is

```
function Example5_19
b = 0.7;    alpha = 4;
guess(1:8) = 0.05;
solinit = bvpinit(linspace(0, b, 5), guess);
beamsol = bvp4c(@BeamStepODEqo, @BeamStepBC, solinit, [], b, alpha);
eta = linspace(0, b, 50);
y = deval(beamsol, eta);
for k = 1:4
  subplot(2, 2, k)
  plot([eta (1-b)/b*eta+b], -[y(k,:) y(k+4,:)],'k-')
  hold on
  plot([0 1], [0 0], 'k—')
end

function dydx = BeamStepODEqo(x, y, b, alpha)
dydx=[y(2); y(3); y(4); 1; (1-b)*y(6)/b; (1-b)*y(7)/b; (1-b)*y(8)/b; (1-b)/b*alpha];

function bc = BeamStepBC(yL, yR, b, alpha)
bc = [yL(1); yL(3); yR(1)-yL(5); yR(3)-yL(7); yR(2)-yL(6); yR(4)-yL(8); yR(5); yR(7)];
```

Upon execution, we obtain the results plotted in Figure 5.18.

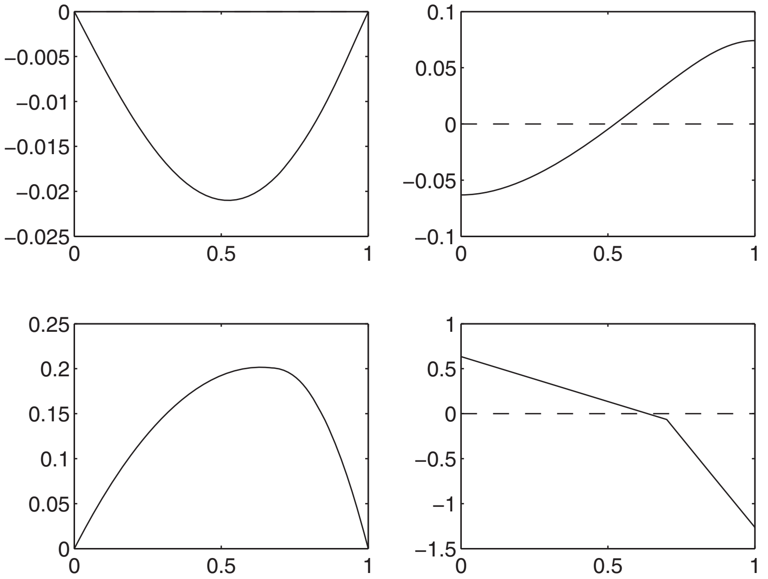


Figure 5.18 Displacement, slope, moment, and shear force for a uniformly loaded beam with a step change in cross section.

Example 5.20 Lowest natural frequency coefficient of an Euler beam clamped at both ends

The governing equation of an Euler beam undergoing harmonic oscillations at frequency ω rad/s is[12]

$$EI\frac{d^4w}{dx^4} - \rho A\omega^2 w = 0 \quad 0 \le x \le L$$

where ρ is the mass density of the beam material and A is the cross-sectional area. If we introduce the quantities

$$\eta = x/L \quad \text{and} \quad \Omega^4 = \frac{\rho AL^4\omega^2}{EI}$$

where Ω is a nondimensional frequency coefficient, then the governing equation becomes

$$\frac{d^4w}{d\eta^4} - \Omega^4 w = 0 \quad 0 \le \eta \le 1$$

We will consider a beam clamped at both ends. Then, from Eq. (5.9b), the boundary conditions are

$$w_1(0) = 0 \quad w_1(1) = 0$$

$$w_2(0) = 0 \quad w_2(1) = 0$$

[12] Balachandran and Magrab, *Vibrations*, p. 564.

The systems of first-order differential equations are similar to those given by Eq. (5.8) with q replaced by $\Omega^4 w_1$, that is,

$$\frac{dw_1}{d\eta} = w_2 \quad \frac{dw_3}{d\eta} = w_4$$

$$\frac{dw_2}{d\eta} = w_3 \quad \frac{dw_4}{d\eta} = \Omega^4 w_1$$

To use bvp4c to obtain the value of Ω requires a slightly different approach. First, bvp4c requires a "fifth" boundary condition, because there are five conditions to be specified: four homogeneous boundary conditions and a fifth condition that will permit the unknown parameter Ω to be determined. We shall choose the fifth condition as the nonzero boundary condition $y_4(0) = 0.05$. (The magnitude 0.05 is not critical, but it should be "small.") Next, bvp4c requires a guess for the parameter Ω; however, the solution is sensitive to this value. If it is too far from the desired region, one may get an answer that corresponds to different solution region. In addition, it is necessary to provide a guess that reasonably approximates the expected spatial shapes of the solutions, y_j. In this example, the function that provides these spatial distributions is **EulerBeamInit**. The spatial distribution that was chosen is $w_1 = \sin(\pi\eta)$ and $w_j, j = 2, 3, 4$, are determined by straightforward differentiation of w_1. Lastly, the value of the parameter Ω is obtained by selecting the appropriate structure of the solution; in our case, it is called *beamsol.parameters* [*beamsol* is a name chosen by the programmer and is arbitrary; the suffix *parameters* is required].

We now illustrate this procedure with the following program:

```
function Example5_20
Omguess = 4.0;
solinit = bvpinit(linspace(0,1,6), @EulerBeamInit, Omguess);
beamsol = bvp4c(@EulerBeamODE, @EulerBeamBC, solinit);
Omega = beamsol.parameters;
eta = linspace(0, 1, 50);
y = deval(beamsol, eta);
ModeShape = y(1,:)/max(abs(y(1,:))); % Normalized mode shape - not plotted
disp(' Omega/pi = ' num2str(Omega/pi,6)])

function yinit = EulerBeamInit(x)
yinit = [sin(pi*x); cos(pi*x); -sin(pi*x); -cos(pi*x)];

function bc = EulerBeamBC(w0, w1, Om)
bc = [w0(1); w0(4)-0.05; w0(2); w1(1); w1(2)];

function dydx = EulerBeamODE(x, w, Om)
dydx = [w(2); w(3); w(4); Om^4*w(1) ];
```

Execution of this program results in the following being displayed to the command window:

```
Omega/pi = 1.50567
```

5.5.5 Numerical Solutions of Delay Differential Equations—`dde23`

The function `dde23` returns the numerical solution to a system of n first-order ordinary differential equations

$$\frac{dy_j}{dt} = f_j\,(t, y_1, y_2, ..., y_n,) \qquad j = 1, 2, ..., n$$

over the interval $t_0 \le t \le t_f$ and $y_j = y_j\,(t, t - \tau_1, t - \tau_2, \ldots, t - \tau_k)$, where τ_j are the delays (lags). The initial conditions (which for a delay equation is called its history) are given by $y_j(t_0) = a_j, j = 1, 2, \ldots, n$, where a_j are either constants or functions of t, and represent the state of the system for $t \le t_0$. At least one a_j has to be nonzero. The arguments and outputs of `dde23` are as follows:

sol = `dde23`(@**FunctionName**, [t0, tf], [a1, a2, . . . , an], options, p1, p2, . . .)

where *sol* is a structure that contains the solution at a specific number of points that have been determined as part of the solution method used in `dde23`. As with `bvp4c`, in order to obtain a smooth curve one needs values at additional intermediate points. To provide these additional points, we use

st = `deval`(sol, t)

where t is a vector of times $t_0 \le t \le t_f$ at which the solution is to be determined and *sol* is the output of `dde23`. The output *st* is an array containing the values of y_j at the time*s* *t*; that is,

$$\begin{gathered} y_1(t) = st(1,:) \\ y_2(t) = st\,(2,:) \\ \cdots \\ y_n(t) = st\,(n,:) \end{gathered}$$

The first argument of `dde23` is @**FunctionName**, which is a handle to either a function file or a subfunction. Its form must be as follows:

`function` yprime = **FunctionName**(t, y, z, p1, p2, . . .)

where t is the independent variable, y is a column vector whose elements correspond to y_j, z is a vector of lags or a function of time histories, $p1, p2, \ldots$ are parameters passed to **FunctionName**, and *yprime* is a column vector of length n whose elements are $f_j(t, y_1, y_2, \ldots, y_n), j = 1, 2, \ldots, n$; that is,

$$yprime = [f_1; f_2; \ldots; f_n]$$

The variable names *yprime*, **FunctionName**, etc., are assigned by the programmer.

We now illustrate the use of `dde23` with the following example.

Example 5.21 Machine tool chatter in turning

Consider the following equation, which describes the dynamic interaction between the work piece and a cutting tool's displacement x in turning:[13]

$$\frac{d^2x}{d\tau^2} + A\frac{dx}{d\tau} + Bx - Cx(\tau - 1/\Omega) = 0$$

where

$$A = \frac{1}{Q} + \frac{K}{k\Omega}, \quad B = 1 + \frac{k_1}{k}, \quad \text{and} \quad C = \mu\frac{k_1}{k}$$

and k_1 is the cutting stiffness, μ is the overlap factor, K is the penetration rate coefficient, k is the stiffness of the tool holder support, Q is the quality factor of the tool, Ω is a nondimensional rotational speed coefficient of the work piece, and $\tau = \omega_n t$, where ω_n is the natural frequency of the tool. The quantity $1/\Omega$ is the lag (delay). It has been shown that for certain combinations of systems and cutting parameters and work piece rotational speeds, there are regions where the system becomes unstable; that is, the displacement grows without limit as time progresses. We shall determine the response x for the following parameters, which have been shown to produce an unstable response: $\mu = 1$, $K/k = 0.0029$, $k_1/k = 0.0785$, $Q = 20$, and $\Omega = 0.225$. We shall assume that prior to the start of the onset of chatter $y_1(\tau \le 0) = 0.1$ and $y_2(\tau \le 0) = 0.1$. Finally, we shall examine the solution over the range $0 \le \tau \le 300$.

The two first-order equations are

$$\frac{dy_1}{d\tau} = y_2(\tau)$$

$$\frac{dy_2}{d\tau} = -Ay_2(\tau) - By_1(\tau) + Cy_1(\tau - 1/\Omega)$$

The program to determine the response is

```
function Example5_21
Om = 0.225;    Q = 20;    mu = 1;
Kk = 0.0029;    k1k = 0.0785;
B = 1+k1k;    C = mu*k1k;
A = 1/Q+Kk/Om;
sol = dde23(@Chatterode, 1/Om, [0.1; 0.1], [0, 300], [], A, B, C);
tau = linspace(0, 300, 700);
y = deval(sol, tau);
plot(tau, y(1,:), 'k-')

function der = Chatterode(t, y, Z, A, B, C)
ylag1 = Z(:, 1);
der = [y(2); -A*y(2)-B*y(1)+C*ylag1(1)];
```

The execution of this program results in Figure 5.19.

[13] *Ibid.*, p. 165ff.

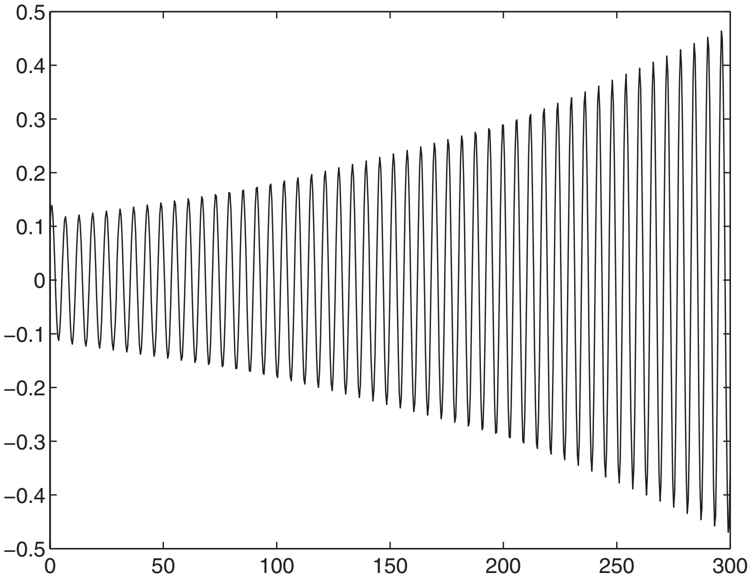


Figure 5.19 Machine tool chatter in turning: response in unstable operating region.

5.5.6 Numerical Solutions of One-Dimensional Parabolic-Elliptic Partial Differential Equations—pdepe

The function `pdepe` obtains the numerical solution to the partial differential equation

$$c(x,t,u,\partial u/\partial x)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f(x,t,u,\partial u/\partial x)\right) + s(x,t,u,\partial u/\partial x) \quad m = 0, 1, \text{or } 2 \tag{5.10}$$

where $u = u(x, t), a \le x \le b, t_o \le t \le t_{\text{end}}$, the initial condition is

$$u(x, t_0) = u_0(x) \tag{5.11}$$

and the boundary conditions at $x = a$ and $x = b$ are, respectively,

$$\begin{aligned} p_a(a,t,u) + q_a(a,t)f(a,t,u,\partial u/\partial x) &= 0 \\ p_b(b,t,u) + q_b(b,t)f(b,t,u,\partial u/\partial x) &= 0 \end{aligned} \tag{5.12}$$

The function `pdepe` is invoked with

sol = `pdepe`(m, @**pdeID**, @**pdeIC**, @**pdeBC**, x, t, options, p1, p2, ...)

where $m = 0$, 1, or 2 and defines the coordinate system with 0 corresponding to a Cartesian system, 1 to a cylindrical system, and 2 to a spherical system, x is a vector of values for which `pdepe` will provide the corresponding values of u such that $x(1) = a$ and $x(\texttt{end}) = b$, t is a vector of values for which `pdepe` will provide the corresponding values of u such that $t(1) = t_o$ and $t(\texttt{end}) = t_{\text{end}}, p1, p2, \ldots$ are parameters that are passed to **pdeID**, **pdeIC**, and **pdeBC** (they must appear in each of these functions whether or not they are used by that function), and *options* is set by `odeset`. The quantity *sol* (= $u(t,x)$) is an array $sol(i,j)$, where i is the element number of the temporal mesh and j is the element number of the spatial mesh. Thus, $sol(i,:)$ approximates $u(t,x)$ at time t_i at all mesh points from $x(1) = a$ to $x(\texttt{end}) = b$. The function **pde1D** specifies c, f, and s in Eq. (5.10) as follows:

```
function [c, f, s] = pde1D(x, t, u, dudx, p1, p2, ... )
c = ...;
f = ...;
s = ...;
```

where *dudx* is the partial derivative of u with respect to x and is provided by `pdepe`. The function **pdeIC** specifies $u_o(x)$ in Eq. (5.11) as follows:

```
function uo = pdeIC(x, p1, p2, ... )
uo = ;
```

The function **pdeBC** specifies the elements p_a, q_a, p_b, and q_b, of the boundary conditions in Eq. (5.12) as follows:

```
function [pa, qa, qb, pb] = pdeBC(xa, ua, xb, ub, t, p1, p2, ... )
pa = ...;
qa = ...;
pb = ...;
qb = ...;
```

where *ua* is the current solution at $xa = a$ and *ub* is the current solution at $xb = b$.

To illustrate the use of `pdepe`, we consider the following equation that describes one-dimensional heat conduction in a slab with a heat source:[14]

$$\frac{\partial\theta}{\partial\tau} = \frac{\partial^2\theta}{\partial\xi^2} + \Sigma \tag{5.13}$$

where all the quantities are nondimensional. The corresponding initial condition is

$$\theta(\xi, 0) = 1 - 0.45\xi \tag{5.14}$$

and the boundary conditions are

$$\left.\frac{\partial\theta}{\partial\xi}\right|_{\xi=0} = Bi\theta(0, \tau) \tag{5.15}$$
$$\theta(1, \tau) = \theta_1$$

where Bi is a constant called the Biot number and θ_1 is a constant.

Comparing Eq. (5.13) with Eq. (5.10), we see that $c = 1, f = \partial\theta/\partial\xi$, and $s = \Sigma$ and on comparing Eqs. (5.11) with (5.14), we find that $u_o(x) \rightarrow 1 - 0.45\xi$. We note

[14] See Section 12.1.3 for additional details concerning this equation and its solution. Other examples of the use of `pdepe` can be found in Sections 11.2.1, 11.5, 12.1.3, 12.2.1, 14.1.1, 14.2.4, and 14.2.6.

that the boundary conditions are specified at $a = 0$ and $b = 1$; therefore, on comparing Eq. (5.12) with (5.15), we have, $qa = 1$ and $pa = -Bi\theta(0, \tau)$ and $qb = 0$ and $pb = \theta(1, \tau) - \theta_1$. We assume that $Bi = 0.1, \theta_1 = 0.55$, and $\Sigma = 1$. We also note that in the arguments of **pdeBC**, $ua = \theta(0, \tau)$ and $ub = \theta(1, \tau)$, which are provided by pdepe.

The numerical solutions to Eqs. (5.13)–(5.15) are obtained using the following program:

```
function pdepeExample
Bi = 0.1;    T1 = 0.55;    Sigma = 1;
xi = linspace(0, 1, 25);    tau = linspace(0, 1, 101);
theta = pdepe(0, @pde1D, @pdeIC, @pdeBC, xi, tau, [], Bi, T1, Sigma);
hold on
for kk = 1:5:length(zi)
  plot(tau, theta(:, kk), 'k-')
end

function [c, f, s] = pde1D(x, t, u, DuDx, Bi, Tr, Sigma)
c = 1;    f = DuDx;    s = Sigma;

function T0 = pdeIC(x, Bi, Tr, Sigma)
T0 = 1-0.45*x;

function [pl, ql, pr, qr] = pdeBC(xl, ul, xr, ur, t, Bi, Tr, Sigma)
pr = ur-Tr;    qr = 0;
pl = -Bi*ul;    ql = 1;
```

The execution of this program results in Figure 5.20, which is a plot of θ as a function of τ for $\xi = 0.0, 0.25, 0.5, 0.75$, and 1.0; the top curve corresponds to $\xi = 0$ and the bottom curve to $\xi = 1$.

5.5.7 Local Minimum of a Function—fminbnd

The function fminbnd finds a local minimum of the real function $f(x)$ in the interval $a \leq x \leq b$ within a tolerance t_o. It can also transfer $p_j, j = 1, 2, \ldots,$ parameters to the function defining $f(x)$. The general expression for fminbnd is

```
[xmin fmin] = fminbnd(@FunctionName, a, b, options, p1, p2, . . . )
```

where **FunctionName** is the name of the function or subfunction, a = a, b = b, *options* is an optional vector whose parameters are set with optimset (see the *Help* file for optimset), and p1, etc., are the parameters p_j. The quantity *xmin* is the value of x at which **FunctionName** is a minimum and *fmin* is the value of **FunctionName** at *xmin*.

The interface for **FunctionName** has the form

```
function z = FunctionName(x, p1, p2, . . . )
Expression(s)
z = . . .
```

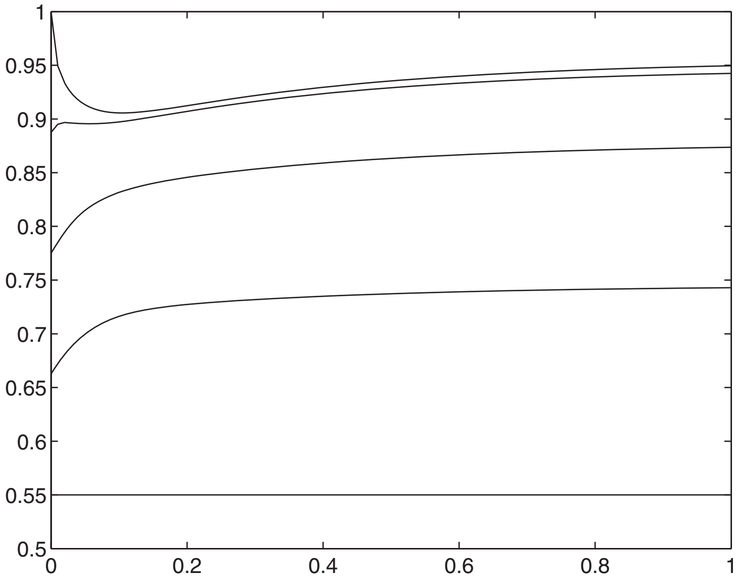


Figure 5.20 Temperature distributions in a slab as a function of time for several values of position.

where x is the independent variable that `fminbnd` is varying in order to determine where the minimum of $f(x)$ occurs. The independent variable must always appear in the first location.

When $f(x)$ is an expression that can be represented by `inline` or by an anonymous function with the name **IorAFunctionName**, `fminbnd` is accessed as follows:

[xmin, fmin] = `fminbnd`(**IorAFunctionName**, a, b, options, p1, p2, . . .)

We shall now illustrate the use of `fminbnd`. Consider the MATLAB demonstration function `humps`, which is shown in Figure 5.21. The minimum value of the function between $0.5 \le x \le 0.8$ is determined from the script

[xmin, fmin] = `fminbnd`(@`humps`, 0.5, 0.8)

Upon execution, we obtain

```
xmin =
   0.6370
fmin =
   11.2528
```

Thus, the minimum in the interval $0.5 \le x \le 0.8$ occurs at $x = 0.6370$ where the magnitude of the function is 11.253.

If, on the other hand, we want to find the maximum value of `humps` in the interval $0 \le x \le 0.5$ and where it occurs, then we have to recognize that `fminbnd` must operate on the negative or the reciprocal of the function `humps`. Thus, we use `inline` to create a function that computes the negative of `humps` in this region. The script is

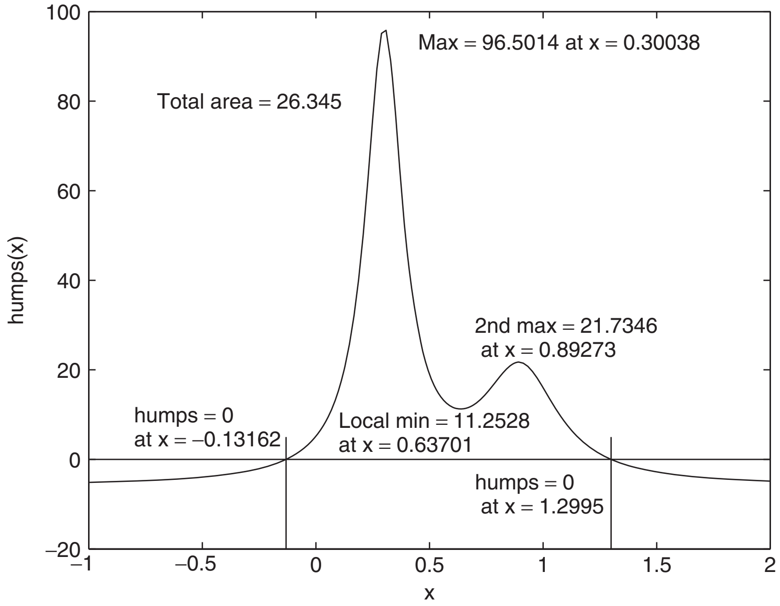


Figure 5.21 Properties of MATLAB's demonstration function humps.

```
[xmax, fmax] = fminbnd(inline('-humps(x)', 'x '), 0, 0.5);
disp(['Maximum value of humps in the interval 0 <= x <= 0.5 is '
      num2str(-fmax)])
disp('which occurs at x = ' num2str(xmax)])
```

which upon execution displays to the command window

```
Maximum value of humps in the interval 0 <= x <= 0.5 is 96.5014
which occurs at x = 0.30039
```

Notice that we had to compute the negative of *fmax* before displaying it, since fminbnd uses a function that is the negative of humps. The other quantities appearing in Figure 5.21 can be verified using the functions discussed in Sections 5.5.1 and 5.5.2.

Example 5.22 Response of a single degree-of-freedom system to a ramp force

The nondimensional displacement response of a single degree-of-freedom system to a ramp force is[15]

$$y(\tau) = h(\tau)u(\tau) - h(\tau - \tau_0)u(\tau - \tau_0)$$

where $u(\tau)$ is the unit step function and

$$h(\tau) = \frac{1}{\tau_0}\left\{-2\zeta + \tau + e^{-\zeta\tau}\left[2\zeta \cos\left(\tau\sqrt{1-\zeta^2}\right) + \frac{2\zeta^2 - 1}{\sqrt{1-\zeta^2}} \sin\left(\tau\sqrt{1-\zeta^2}\right)\right]\right\}$$

[15] Balachandran and Magrab, *Vibrations*, 2009, p. 312.

We shall determine the maximum value of $y(\tau)$ and the time at which this maximum value occurs. The program is as follows:

```
function Example5_22
z = 0.1;    tzo = 15;
MMin = @(tau, z, tzo) (-yt(tau, z, tzo));
[tm, ym] = fminbnd(MMin, 15 , 20, [], z, tzo);
disp(['y_max = ' num2str(-ym, 4) ' at tau = ' num2str(tm, 4)])

function a = yt(tau, z, tzo)
a = ht(tau, z, tzo)-ht(tau-tzo, z, tzo).*(tau-tzo>0);

function out = ht(tau, z, tzo)
r = sqrt(1-z^2);
out= (-2*z+tau+exp(-z*tau).*( 2*z*cos(tau*r)+(2*z^2-1)/r*sin(tau*r) ))/tzo;
```

Upon execution, we obtain

```
y_max = 1.065 at tau = 16.82
```

5.5.8 Numerical Solutions of Nonlinear Equations—fsolve

The function fsolve in the Optimization toolbox finds the numerical solution to a system of n nonlinear equations $f_n(x_1, x_2, \ldots, x_n) = 0$ in the x_n unknowns using a starting guess $x_s = [x_{s1}\, x_{s2} \ldots x_{sn}]$. The outputs of the function are the n solutions x_{sol}. The function fsolve can also transfer p_j parameters to the functions defining $f_n(x)$. The general expression for fsolve is

```
xsol = fsolve(@FunctionName, xs, options, p1, p2, . . . )
```

where **FunctionName** is the name of the function file or subfunction, xs = x_s, *options* is an optional vector whose parameters are set with optimset (see optimset in the *Help* file), and p1, etc., are the parameters p_j. The output of the function, *xsol*, is a vector of x_{sol}.

The interface for the function whose name is **FunctionName** has the form

```
function z = FunctionName(x, p1, p2, . . . )
z = [f1; f2; . . . ; fn];
```

where x is a vector of the n quantities to be determined, x_n, and z is a column vector composed of n MATLAB expressions for the n nonlinear equations $f_n(x_1, x_2, \ldots, x_n)$ in terms of x and the parameters p_j.

When $f_n(x_1, x_2, \ldots, x_n)$ is represented by an anonymous function or by inline with the name **IorAFunctionName**, fminbnd is accessed as follows:

```
xsol = fsolve (IorAFunctionName, a, b, options, p1, p2, . . . )
```

We shall now illustrate the use of fsolve.

Example 5.23 Inverse kinematics

Consider the following system of equations, which results from an intermediate step in the inverse kinematics solution for the three degree-of-freedom linkages, as shown in Figure 2.8.

$$r_1 - a_1 \cos(\theta_1) - a_2 \cos(\theta_1 + \theta_2) = 0$$
$$r_2 - a_1 \sin(\theta_1) - a_2 \sin(\theta_1 + \theta_2) = 0$$

To solve this system of equations, we first create the function **kinematics**, which puts these equations in the form required by fsolve. Let us assume that $r_1 = 1.8$, $r_2 = 2.1$, $a_1 = 1.0$, and $a_2 = 2$, and let our initial guesses for θ_1 and θ_2 be $\pi/6$. Then, the program is

```
function Example5_23
options = optimset('display', 'off');
z = fsolve(@kinematics, [pi/6 pi/6], options, 1, 2, 1.8, 2.1)*180/pi;
for k = 1:length(z)
  disp(['Theta(' num2str(k,1) ') = ' num2str(z(k)) ' degrees'])
end

function w = kinematics(theta, a1, a2, r1, r2)
w = [a1*cos(theta(1))+a2*cos(theta(1)+theta(2))-r1; . . .
     a1*sin(theta(1))+a2*sin(theta(1)+theta(2))-r2];
```

where *theta*(1) $= \theta_1$ and *theta*(2) $= \theta_2$. Upon execution, we obtain

```
Theta(1) = 16.6028 degrees
Theta(2) = 48.5092 degrees
```

Thus, $\theta_1 = z(1) = 16.6026°$ and $\theta_2 = z(2) = 48.5095°$. Another set of angles will be found when the initial guess is $\theta_1 = \theta_2 = \pi$. Thus, fsolve must be used with caution when more than one solution exists.

Example 5.24 Intersection of a parabola and an ellipse

Consider the intersection of an ellipse

$$g(x, y) = x^2/4 + y^2 - 1$$

with the parabola

$$f(x, y) = y - 4x^2 + 3$$

A graph of these two functions reveals that they intersect at four points. Thus, the value returned by fsolve will be sensitive to the initial guess.

The function that will be used by fsolve is created with inline, where $xy(1) = x$ and $xy(2) = y$. The script to determine the solution with the initial guesses of $x = 0.5$ and $y = -0.5$ is

```
fgsolve = inline('[0.25*xy(1).^2+xy(2).^2-1; xy(2)-4*xy(1).^2+3]', 'xy');
options = optimset('display', 'off');
xy = fsolve(fgsolve, [0.5, -0.5], options)
```

Upon execution, we obtain

```
xy =
   0.7188    -0.9332
```

Thus, $x = xy(1) = 0.7188$ and $y = xy(2) = -0.9332$. If, instead, we had chosen for our initial guess $x = -0.5$ and $y = 0.5$, we would have obtained $x = xy(1) = -0.9837$ and $y = xy(2) = 0.8707$.

5.6 SYMBOLIC SOLUTIONS AND CONVERTING SYMBOLIC EXPRESSIONS INTO FUNCTIONS

As discussed in Section 1.4, the Symbolic Math toolbox provides the capability of manipulating symbols to perform algebraic, matrix, and calculus operations symbolically. When one couples the results obtained from symbolic operations with MATLAB's ability to create functions, one has a very effective means of numerically evaluating symbolically obtained expressions. In this section, we will illustrate the two ways in which one can straightforwardly obtain numerical values from symbolically derived expressions. The first way is to employ `inline` and

`vectorize`(f)

which converts its argument *f*, a MATLAB expression, to a string and converts the multiplication, division, and exponentiation operators to their dot operator counterparts. In other words, if $f = f(x, y, z)$, say, is a symbolic expression where x, y, and z, are symbolic variables, we create an inline function in the following manner:

fnct = `inline`(`vectorize`(f), 'x', 'y', 'z')

which creates the inline function

```
fnct =
Inline function:
fnct(x,y,z) = f
```

where *f* is a MATLAB expression with *, /, and ^ operators replaced by their dot operation counterparts, that is, by .*, ./, and .^.

The second way is to use `matlabFunction` to create an anonymous function. If, again, $f = f(x, y, z)$ is a symbolic expression with symbolic variables x, y, and z, then

fnct = `matlabFunction`(f, 'vars', [x, y, z])

where 'vars' (apostrophes required) indicates that an anonymous function will be created with the parameters x, y, and z of the form

fnct = @(x, y, z) (f)

where *f* is a MATLAB expression with the *, /, and ^ operators replaced by their dot operation counterpart, that is, by .*, ./, and .^. If 'vars' is replaced with 'file', a function M-file is created.

We now illustrate this technique with several examples.

Example 5.25 Inverse Laplace transform

Consider Eq. (5.3) whose Laplace transform when $y(0) = 0$ and $dy(0)/dt = 0$, is

$$Y(s) = \frac{H(s)}{s^2 + 2\xi s + 1}$$

where $Y(s)$ is the Laplace transform of $y(t)$, $H(s)$ is the Laplace transform of $h(t)$, and $\xi < 1$ is a real constant. If we assume that $h(t) = u(t)$, where $u(t)$ is the unit step function, then $H(s) = 1/s$. The script below uses `inline` to convert the symbolically obtained quantity $y(t)$ to a function that can be evaluated numerically. It is assumed that $\xi = 0.15$.

```
syms s t
syms xi real
den = s*(s^2+2*xi*s+1);
yt = ilaplace(1/den, s, t);
yoft = inline(vectorize(yt), 't', 'xi');
t = linspace(0, 20, 200);    xi = 0.15;
plot(t, real(yoft(t, xi)))
```

An examination of the numerical results indicates that the imaginary part of the solution is virtually zero. Therefore, we use `real` to remove any residual imaginary part due to numerical round-off errors. The execution of this script results in the curve shown in Figure 5.22.

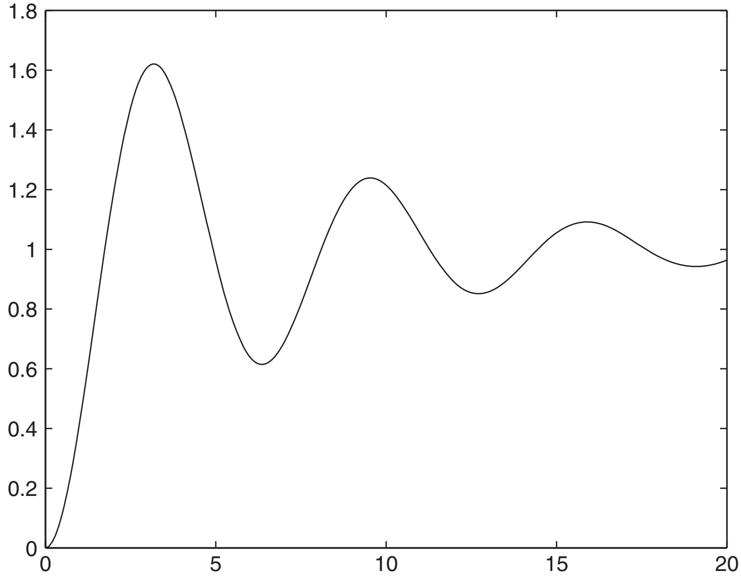


Figure 5.22 Solution to Eq. (5.3) using Laplace transforms when $h(t) = u(t)$ and $\xi = 0.15$.

Repeating the above solution using matlabFunction, we have that

```
syms s t
syms xi real
den = s*(s^2+2*xi*s+1);
yt = ilaplace(1/den, s, t);
yoft = matlabFunction(yt, 'vars', [t, xi]);
t = linspace(0, 20, 200);    xi = 0.15;
plot(t, real(yoft(t, xi)))
```

Example 5.26 Evaluation of a convolution integral and its characteristics

Consider the following convolution integral that results from the solution to Eq. (5.3), when $\xi < 1$ and $h(t) = u(t)$, where $u(t)$ is the unit step function:

$$y(t) = \frac{e^{-\xi t}}{\sqrt{1 - \xi^2}} \int_0^t e^{\xi \eta} \sin\left((t - \eta)\sqrt{1 - \xi^2}\right) d\eta$$

We shall (1) obtain a symbolic solution to this integral, convert it to a function, and plot it; (2) determine the magnitude of its maximum response from a solution that satisfies $dy/dt = 0$ and the time at which it occurs and examine the second derivative to verify that it is a maximum; and (3) determine the time it takes for $y(t)$ to go from $0.1y(t)$ to $0.9y(t)$, which is called the rise time of the response.

The script for part (1) is

```
syms t xi n r a
r = sqrt(1- xi^2);
arg = exp(xi*n)*sin(r*(t-n));
yt = exp(-xi*t)*int(arg, n, 0, t)/r;
yoft = inline(vectorize(yt), 't', 'xi');
tt = linspace(0, 20, 200); z = 0.15;
plot(tt, yoft(tt, z))
```

Upon execution, this script also produces Figure 5.22.

To obtain the maximum value, we differentiate the solution $y(t)$, convert it to a function, and then use fzero to determine the time t_{max} at which the derivative is zero. To verify that it is a maximum, we obtain the second derivative of $y(t)$. If the second derivative is negative at t_{max}, then the function is a maximum. The script for part (2) is

```
z = 0.15;
syms t xi n r a
r = sqrt(1- xi^2);
arg = exp(xi*n)*sin(r*(t-n));
yt = exp(-xi*t)*int(arg, n, 0, t)/r;
yoft = inline(vectorize(yt), 't', 'xi');
dydt = inline(vectorize(diff(yt, t)), 't', 'xi');
tmax = fzero(dydt, [3 5],[], z);
ymax = yoft(tmax, z);
disp('ymax = ' num2str(ymax) ' tmax = ' num2str(tmax)])
d2ydt2 = inline(vectorize(diff(yt, t, 2)), 't', 'xi');
secder = d2ydt2(tmax, z);
disp(['Second derivative at tmax = ' num2str(secder)])
```

The execution of this script displays the following result to the command window:

```
ymax = 1.6209    tmax = 3.1775
Second derivative at tmax = -0.62087
```

To obtain the rise time, we have to create a new `inline` function to compute $y(t) - a$, where $a = 0.1$ and 0.9. The script for part (3) is

```
z = 0.15;
syms t xi n r a
r = sqrt(1- xi^2);
arg = exp(xi*n)*sin(r*(t-n));
yt = exp(-xi*t)*int(arg, n, 0, t)/r;
ytrise = inline(vectorize(yt-a), 't', 'xi', 'a');
t9 = fzero(ytrise, [0 2], [], z, 0.9);
t1 = fzero(ytrise, [0 2], [], z, 0.1);
disp(['Rise time = ' num2str(t9-t1)])
```

Upon execution, the following is displayed to the command window:

```
Rise time = 1.1518
```

Example 5.27 Symbolic solution of algebraic equations

In this example, we shall show how to use the symbolic counterpart to `fsolve`, which is

```
z = solve('Eqn1', 'Eqn2', . . . , 'EqnN', 'Var1', 'Var2', ..., 'VarN')
```

where *EqnK* are the $K = 1, 2, \ldots, N$ homogeneous algebraic equations in terms of the N variables *VarK*, $K = 1, 2, \ldots, N$. The quantity z is a structure that contains the solutions such that

```
Var1 = z.Var1
Var2 = z.Var2
      ...
VarN = z.VarN
```

where *VarK* are symbolic quantities.

To illustrate the use of `solve`, we shall use the equations given in Example 5.24; that is,

$$x^2/4 + y^2 - 1 = 0$$
$$y - 4x^2 + 3 = 0$$

The script to solve these equations is

```
z = solve('x^2/4+y^2-1', 'y-4*x^2+3', 'x', 'y')
x = z.x
y = z.y
```

which upon execution gives the symbolic expressions

```
x =
   ((7*17^(1/2))/32 + 95/32)^(1/2)/2
  -((7*17^(1/2))/32 + 95/32)^(1/2)/2
   (95/32 - (7*17^(1/2))/32)^(1/2)/2
  -(95/32 - (7*17^(1/2))/32)^(1/2)/2
```

```
y =
   (7*17^(1/2))/32 - 1/32
   (7*17^(1/2))/32 - 1/32
  -(7*17^(1/2))/32 - 1/32
  -(7*17^(1/2))/32 - 1/32
```

To convert the symbolic expressions for *x* and *y* to numerical values, we use

```
v = double(x)
```

Then, the previous script becomes

```
z = solve('x^2/4+y^2-1', 'y-4*x^2+3', 'x', 'y');
v = [double(z.x) double(z.y)];
disp(['    x    y'])
disp(v)
```

where we have placed *x* and *y* in a (4 × 2) array *v* whose elements are accessible in the standard manner. Upon execution, we obtain

```
    x          y
  0.9837    0.8707
 -0.9837    0.8707
  0.7188   -0.9332
 -0.7188   -0.9332
```

We see that the symbolic technique found all four sets of solutions, with the third row corresponding to the solution found in Example 5.24.

Example 5.28 Symbolic solution of a differential equation

The symbolic solution to a system of coupled ordinary differential equations and their specified boundary conditions is obtained from

```
z = dsolve('Eqn1', 'Eqn2', . . . , 'EqnN', 'BC1', 'BC2', . . . , 'BCM', 'v')
```

where *EqnK* are the $K = 1, 2, \ldots, N$ differential equations in terms of the independent variable *v* and *BCK*, $K = 1, 2, \ldots, M$ are the *M* boundary conditions. The number of boundary conditions is the sum of the highest order of each of the *N* equations. The quantity *z* is a symbolic expression for the dependent variables. If we have a differential equation in the dependent variable *w*, then the derivatives in the string quantities are represented as *Dnw*, where *D* indicates the derivative, *n* indicates the order of the derivative, and *w* is the dependent variable. Thus, *D*2*w* corresponds to d^2w/dx^2, where *x* is the independent variable.

We shall illustrate how to obtain the symbolic solution by considering a cantilever Euler beam subject to a load of the form $q(\eta) = \sin(\pi\eta)$. The governing equation is given by Eq. (5.6) and the boundary conditions by Eq. (5.7b) at $\eta = 0$ and by Eq. (5.7c) at $\eta = 1$. We shall obtain the symbolic solution for this beam and then convert it to a function so that numerical results can be obtained. The script is as follows:

```
syms x
r = dsolve('D4w-sin(pi*x)','w(0) = 0', 'Dw(0) = 0', 'D3w(1) = 0', 'D2w(1) = 0', 'x');
dis = inline(vectorize(r), 'x');
disp(['Displacement at the free end = ' num2str(dis(1))])
```

Execution of this script yields

```
Displacement at the free end = 0.073852
```

Example 5.29 Symbolic solution used by different functions

In this example, we shall show how to use a symbolic result in several different functions for several different purposes. We shall do this by again considering the displacement response of the mass of a single degree-of-freedom system to a ramp input, which is given by[16]

$$h(\tau) = \frac{1}{\tau_0}\left[y(\tau)u(\tau) - y(\tau - \tau_0)u(\tau - \tau_0)\right] \quad \tau \geq 0$$

where

$$y(\tau) = \frac{1}{r}\int_0^\tau xe^{-\zeta(\tau-x)} \sin\left[r(\tau - x)\right]dx$$

where $\zeta < 1$ is a constant, $r = \sqrt{1 - \zeta^2}$, τ_o is the duration of the ramp portion of the input, and $u(t)$ is the unit step function. For this response, we shall first obtain $y(\tau)$ using the Symbolic toolbox, then plot the function, determine the magnitude of its maximum response and the time at which it occurs, and finally, determine the time that it takes to reach an amplitude of 0.8. The program is as follows:

```
function Example5_29
xi = 0.1;   to = 15;   A = 0.8;
y = SymRes;
[a, b] = fminbnd(@ResNeg, 10, 20, [], y, xi, to);
disp(['Maximum value of y(t) = ' num2str(-b) ' and occurs at t = ' num2str(a)])
t8 = fzero(@ResA, [0, 15], [], y, xi, to, A);
disp(['y(t) = 0.8 at t = ' num2str(t8) ])
t = linspace(0, 30, 200);
plot(t, Res(t, y, xi, to))

function Y = Res(t, y, xi, to)
Y = (y(t,xi)-y(t-to,xi).*(t>to))/to;

function Y = ResNeg(t, y, xi, to)
Y = -Res(t, y, xi, to);

function Y = ResA(t, y, xi, to, A)
Y = A-Res(t, y, xi, to);

function z = SymRes
syms zet x t
arg = x*exp(-zet*(t-x))*sin(sqrt(1-zet^2)*(t-x))/sqrt(1-zet^2);
z = inline(vectorize(int(arg, x, 0, t)), 't', 'zet');
```

We see that rather than evaluating the integral each time we need it, we evaluate it once by calling **SymRes** once. The output quantity y is an inline function that is then passed to the three functions: **Res**, **ResNeg**, and **ResA**. Upon execution, the following is displayed to the command window:

```
Maximum value of y(t) = 1.065 and occurs at t = 16.8152
y(t) = 0.8 at t = 11.9707
```

and the response is the same as that shown in Figure 5.10.

[16] *Ibid*, pp. 311–312.

5.7 SUMMARY OF FUNCTIONS INTRODUCED IN CHAPTER 5

A summary of the functions introduced in the chapter along with their descriptions is presented in Table 5.4, and a summary of the functions from the Symbolic toolbox is presented in Table 5.5.

TABLE 5.4 MATLAB Functions Introduced in Chapter 5

MATLAB function	Description
`besselj`	Bessel function of the first kind
`bvp4c`	Solves the two-point boundary value problem for a system of ordinary differential equations
`bvpinit`	Forms the initial guesses for `bvp4c`
`conv`	Multipies two polynomials
`dde23`	Delays differential equation solver
`deval`	Evaluates the solution from `bvp4c`
`dblquad`	Numerically evaluates a double integral
`diff`	Obtains differences of adjacent elements in an array
`error`	Displays an error message
`feval`	Evaluates a function
`fft`	Obtains the discrete Fourier transform
`fminbnd`	Minimizes a function of one variable in a specified interval
`fsolve`	Solves a system of nonlinear equations (Optimization toolbox)
`function`	Creates a function m file
`fzero`	Finds a zero of a function of one variable
`global`	Defines global variables
`help`	Accesses help comments in MATLAB functions from command window
`humps`	MATLAB demonstration function
`ifft`	Obtains the discrete inverse Fourier transform
`inline`	Constructs an inline function
`interp1`	Performs a one-dimensional interpolation
`matlabFunction`	Converts a symbolic expression to a function that can be evaluated numerically
`nargin`	Determines the number of arguments in a function interface
`ode45`	Solves the initial value problem for a system of ordinary differential equations
`odeset`	Alters options in the ordinary differential equations solvers
`optimset`	Alters options in Optimization solvers including `fzero` and `fminbnd`
`pdepe`	One-dimensional parabolic-elliptic partial differential equation solver
`poly`	Creates a polynomial from its roots
`polyarea`	Determines the area of polygon
`polyfit`	Fits data with an *n*th order polynomial
`polyval`	Evaluates a polynomial
`quadl`	Numerically evaluates a single integral
`return`	Early return from a function
`roots`	Determines the roots of a polynomial
`spline`	Fits data with splines
`trapz`	Numerically integrates a single integral using trapezoidal approximation
`vectorize`	Converts an expression to a string and replaces all mathematical operators with dot operators

TABLE 5.5 MATLAB Functions from the Symbolic Toolbox Introduced in Chapter 5

MATLAB function	Description
`dsolve`	Obtains symbolic solution to a system of ordinary differential equations
`solve`	Obtains symbolic solution to a system of nonlinear algebraic equations

EXERCISES

Section 5.4.2

5.1 The stress concentration factor for a stepped circular shaft shown in Figure 5.23 is approximated by[17]

$$K_t = c\left(\frac{D}{2d} - \frac{1}{2}\right)^{-a}$$

where c and a are given in Table 5.6. Obtain two expressions, one for c and the other for a, as a function of D/d in two ways: (1) with a fifth-order polynomial and (2) with a spline. For both methods, compare the values of K_t obtained with the two sets of fitted values to those obtained with the original values given in Table 5.6. Which is the better method to use in this case?

TABLE 5.6 Stress Concentration Factor Constants

D/d	c	a
6.00	0.88	0.33
3.00	0.89	0.31
2.00	0.91	0.29
1.50	0.94	0.26
1.20	0.97	0.22
1.10	0.95	0.24
1.07	0.98	0.21
1.05	0.98	0.20
1.03	0.98	0.18
1.01	0.92	0.17

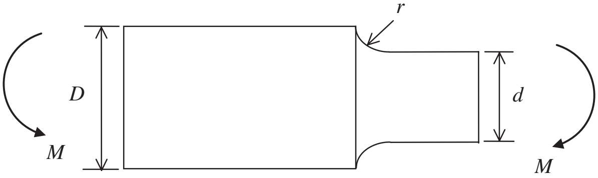


Figure 5.23 Geometry and loading for a stress concentration factor determination.

[17] R. L. Norton, *Machine Design, An Integrated Approach*, Prentice-Hall, Upper Saddle River, NJ, 1996, p. 1005ff.

TABLE 5.7 Constants Defining the Signal in Exercise 5.2

n	$\omega_n/2\pi$	ζ_n	H_n
1	5	0.1	1
2	9	0.04	1.3
3	9.4	0.04	1.3
4	20	0.03	1.8

Section 5.4.7

5.2 Consider the following signal:

$$f(t) = \sum_{n=1}^{4} H_n e^{-\zeta_n \omega_n t} \sin\left(\sqrt{1 - \zeta_n^2}\omega_n t\right) \quad 0 \le t \le T$$

where the values of the constants are given in Table 5.7. For $N = 2^{10}$ and $\Delta t = 2\pi/(4\omega_4)$,

a. Plot the amplitude spectrum for this signal with and without the Hamming weighting function. The results should look like those shown in Figure 5.24.

b. Determine the frequencies at which the peaks occur. [Hint: Use several applications of `find` and `diff`.] [Answers: No Hamming: [4.84375 9.14063 20.0781] Hz; with Hamming: [4.92188 9.0625 9.45313 20.0781] Hz.]

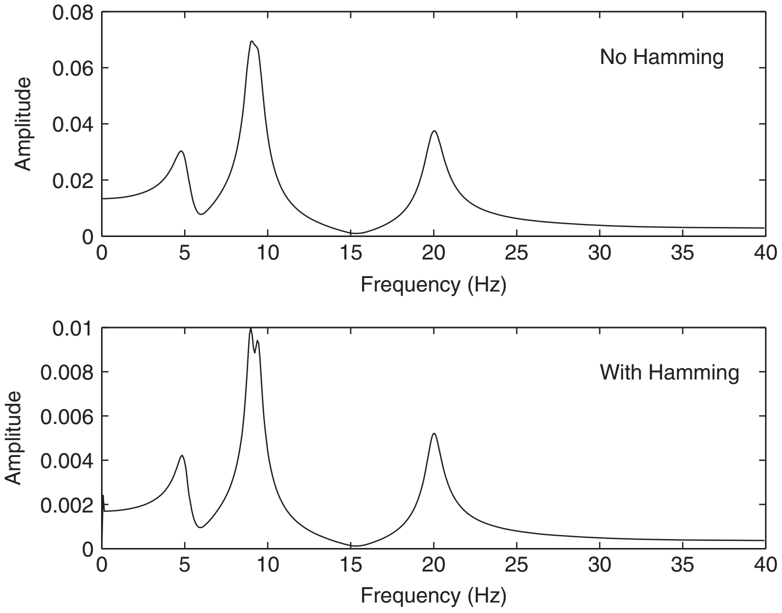


Figure 5.24 Results from the solution to Exercise 5.2a.

Section 5.5.1

5.3 The principal stresses can be determined from the roots of the polynomial[18]

$$\sigma^3 - C_2\sigma^2 - C_1\sigma - C_0 = 0$$

where

$$\begin{aligned} C_2 &= \sigma_x + \sigma_y + \sigma_z \\ C_1 &= \tau_{xy}^2 + \tau_{yz}^2 + \tau_{zx}^2 - \sigma_x\sigma_y - \sigma_y\sigma_z - \sigma_z\sigma_x \\ C_0 &= \sigma_x\sigma_y\sigma_z + 2\tau_{xy}\tau_{yz}\tau_{zx} - \sigma_x\tau_{yz}^2 - \sigma_y\tau_{zx}^2 - \sigma_z\tau_{xy}^2 \end{aligned}$$

and σ_x, σ_y, and σ_z are the applied normal stresses and τ_{xy}, τ_{yz}, and τ_{zx} are the applied shear stresses. If the roots of the equation are σ_1, σ_2, and σ_3 (the three principal stresses), where $\sigma_1 > \sigma_2 > \sigma_3$, then the principal shear stresses are

$$\tau_{12} = (\sigma_1 - \sigma_2)/2 \quad \tau_{23} = (\sigma_2 - \sigma_3)/2 \quad \tau_{13} = (\sigma_1 - \sigma_3)/2$$

where $\tau_{\max} = \tau_{13}$.

Determine the principal stresses and corresponding principal shear stresses when $\sigma_x = 100$, $\tau_{xy} = -40$, $\sigma_y = -60$, $\tau_{yz} = 50$, $\sigma_z = 80$, and $\tau_{zx} = 70$. [Answers: $\sigma_1 = 160.7444$, $\sigma_2 = 54.8980$, $\sigma_3 = -95.6424$, $\tau_{12} = 52.9232$, $\tau_{23} = 75.2702$, and $\tau_{13} = 128.1934$.]

Note: It is suggested that **FindZeros** of Section 5.5.1 be used in Exercises 5.4–5.17, except in Exercise 5.13. In Exercises 5.4–5.8, find the lowest five roots greater than zero for the given equations.

5.4 The following equation arises in the vibration of strings:[19]

$$\sin x = x \cos x$$

5.5 The following equation arises in the heat flow in slabs.[20] Obtain the roots for the two separate cases: $p = 0.1$ and 1.

$$2 \cos x = \left(\frac{x}{p} - \frac{p}{x}\right) \sin x$$

5.6 The following equation[21] arises in the vibrations of annular membranes. Assume that $b = 2$.

$$J_0(x)Y_0(xb) - J_0(xb)Y_0(x) = 0$$

Use `besselj` and `bessely`, respectively, for $J_0(x)$ and $Y_0(x)$, which are the Bessel functions of the first and second kind, respectively, of order 0.

[18] See, for example, Shigley and Mischke, *Mechanical Engineering Design*.

[19] E. B. Magrab, *Vibration of Elastic Structural Members*, Sijthoff & Noordhoff, The Netherlands, 1979, p. 58.

[20] M. N. Ozisik, *Heat Conduction*, 2nd ed., John Wiley & Sons, New York, 1993, p. 47.

[21] Magrab, *Vibrations of Elastic Structural Members*, p. 83.

5.7 The following equation[22] arises in the vibrations of a cantilever beam carrying a concentrated mass M_0 at its free end. Obtain the roots for the three separate cases: $M_0/m_0 = 0, 0.2$, and 1.

$$\frac{\Omega M_0}{m_0}(\cos\Omega\sinh\Omega - \sin\Omega\cosh\Omega) + \cos\Omega\cosh\Omega + 1 = 0$$

5.8 The following equation[23] arises in the vibrations of a beam clamped at one end and simply supported at its other end:

$$\cos\Omega\,\tanh\Omega - \sin\Omega = 0$$

5.9 The following equation[24] arises in the vibrations of a solid circular plate clamped on its outer boundary:

$$J_m(\Omega)I_{m+1}(\Omega) + I_m(\Omega)J_{m+1}(\Omega) = 0$$

where $J_m(x)$ is the Bessel function of the first kind of order m and $I_m(x)$ is the modified Bessel function of the first kind of order m. Use `besselj` and `besseli`, respectively, for $J_m(x)$ and $I_m(x)$. Find the lowest three roots for $m = 0, 1$, and 2.

5.10 The following equation[25] arises in the determination of the inplane symmetric modes of a suspended cable. Find the lowest root when $\lambda^2 = 2\pi^2, 4\pi^2$, and $8\pi^2$.

$$\sin\Omega = \left(\Omega - \frac{4\Omega^3}{\lambda^2}\right)\cos\Omega$$

5.11 In the analysis of nonuniform flow in an open channel of trapezoidal cross section, the ratio of the depth of the fluid to the height of the energy gradient x is determined from[26]

$$(1 + c_0x)^2(x^2 - x^3) = c_1$$

where $0 \le c_0 \le 11$ and $0.005 \le c_1 \le 12.3$ are functions of the geometry of the channel and the flow rate. However, not all combinations of c_0 and c_1 are appropriate. Find the pairs of real values of x between 0 and 1 that satisfy this equation for (1) $c_0 = 0.4$ and $c_1 = 0.2$; and (2) $c_0 = 7.0$ and $c_1 = 4.0$. Use two methods: `fzero` and `roots`. To use `roots`, the equation is rewritten as

$$-c_0^2x^5 + (c_0^2 - 2c_0)x^4 + (2c_0 - 1)x^3 + x^2 - c_1 = 0$$

5.12 The wave angle $\beta(0 < \beta \le \pi/2)$ of a disturbance wave on top of a fluid in an open channel in which the velocity of the fluid is greater than the wave speed in the fluid is determined from[27]

$$2N_F^2\sin^2(\beta)\tan^2(\beta - \theta) = \tan(\beta)\tan(\beta - \theta) + \tan^2(\beta) \quad \beta > \theta$$

where θ is the wall deflection angle and $1 \le N_F \le 12$ is the Froude number. Determine the values of β, in degrees, in the range $\theta < \beta \le 90°$ when $\theta = 35°$ and $N_F = 5$.

[22] *Ibid.*, p. 130.

[23] *Ibid.,* p. 120.

[24] *Ibid.*, p. 252.

[25] M. Irvine, *Cable Structures*, Dover Publications, Inc., New York, 1981, p. 95.

[26] H. W. King, *Handbook of Hydraulics*, 4th ed., McGraw-Hill, New York, 1954, p. 8–1.

[27] N. H. C. Hwang and C. E. Hita, *Fundamentals of Hydraulic Engineering Systems*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1987, p. 222.

5.13 An estimate of a parameter β appearing in the Weibull probability density function (see Section 8.2.2) requires the solution of[28]

$$\beta = \left[\sum_{i=1}^{n} x_i^{\beta} \ln(x_i) \Big/ \sum_{i=1}^{n} x_i^{\beta} - \frac{1}{n} \sum_{i=1}^{n} \ln(x_i) \right]^{-1}$$

where x_i are a random sample of size n. If $x = [72, 82, 97, 103, 113, 117, 126, 127, 127, 139, 154, 159, 199, 207]$, then determine the value of β. *Note*: **FindZeros** will not function correctly for this problem because β is an array whose size is different from the size of x and is independent of x.

5.14 In determining the surface contact shear stress between a sphere and a plane, which is a model of the effects of a bearing against a surface, the value of a ratio x is obtained from[29]

$$x \ln\left(\sqrt{x^2 - 1} + x\right) - \sqrt{x^2 - 1} - Cx = 0$$

where $x > 1$ and $C < 1$. For $C = 0.5$, determine x.

5.15 Find the three real roots of[30]

$$x^4 = 2^x$$

5.16 The computational formula for the generalized equation for the compressibility factor Z of a gas is given by[31]

$$\begin{aligned} Z(r, \tau) = 1 &+ r\sum_{i=1}^{6} A_i\tau^{i-1} + r^2\sum_{i=7}^{10} A_i\tau^{i-7} + r^3\sum_{i=11}^{13} A_i\tau^{i-11} + r^4 A_{14}\tau \\ &+ r^5(A_{15}\tau^2 + A_{16}\tau^3) + r^6 A_{17}\tau^2 + r^7(A_{18}\tau + A_{19}\tau^3) \\ &+ r^8 A_{20}\tau^3 + r^2 e^{-0.0588r^2}[A_{21}\tau^3 + A_{22}\tau^4 + r^2(A_{23}\tau^3 + A_{24}\tau^5) \\ &+ r^4(A_{25}\tau^3 + A_{26}\tau^4) + r^6(A_{27}\tau^3 + A_{28}\tau^5) + r^8(A_{29}\tau^3 + A_{30}\tau^4) \\ &+ r^{10}(A_{31}\tau^3 + A_{32}\tau^4 + A_{33}\tau^5)] \end{aligned}$$

where $\tau = T_c/T (0.4 \le \tau \le 1)$, $r = RT_c/P_c v$, R is the gas constant in (MPa-m^3)/(kg-K), T is the temperature in K, P is the pressure in MPa, v is the reciprocal of density in m^3/kg, T_c and P_c are the critical temperature and pressure, respectively, and the values of the 33 constants are given in Table 5.8.

a. Create a function to determine $Z(r, \tau)$. Check your function, using `format long e`, with the following test values:

i. $Z(1,1) = 0.70242396927$

ii. $Z(1/0.3,1) = 0.29999999985$

iii. $Z(2.5,0.5) = 0.99221853928$

b. The above quantity is used in the formula

$$Z(r, \tau) = \frac{p\tau}{r} = \frac{Pv}{RT} \tag{a}$$

[28] D. C. Montgomery and G. C. Runger, *Applied Statistics and Probability for Engineers*, John Wiley & Sons, New York, 1994, p. 299.

[29] W. Changsen, *Analysis of Rolling Element Bearings*, Mechanical Engineering Publishers, London, 1991, p. 80.

[30] Problem suggested by Prof. J. M. Cooper, Department of Mathematics, University of Maryland, College Park, MD.

[31] W. C. Reynolds, *Thermodynamic Properties in SI*, Department of Mechanical Engineering, Stanford University, Stanford, CA, 1979.

TABLE 5.8 Constants in Generalized Formula for Z

j	A_j	j	A_j	j	A_j
1	0.062432384	12	−0.000727155024313	23	−0.0845194493813
2	0.12721477	13	−0.00452454652610	24	−0.00340931311928
3	−0.93633233	14	0.00130468724100	25	−0.00195127049901
4	0.70184411	15	−0.000222165128409	26	$4.93899910978 \times 10^{-5}$
5	−0.35160896	16	−0.00198140535656	27	$-4.93264612930 \times 10^{-5}$
6	0.056450032	17	$5.97573972921 \times 10^{-5}$	28	$8.85666572382 \times 10^{-7}$
7	0.0299561469907	18	$-3.64135349702 \times 10^{-6}$	29	$5.34788029553 \times 10^{-8}$
8	−0.0318174367647	19	$8.41364845386 \times 10^{-6}$	30	$-5.93420559192 \times 10^{-8}$
9	−0.0168211055517	20	$-9.82868858822 \times 10^{-9}$	31	$-9.06813326929 \times 10^{-9}$
10	1.60204060081	21	−1.57683056810	32	$1.61822407265 \times 10^{-9}$
11	−0.00109996740746	22	0.0400728988908	33	$-3.32044793915 \times 10^{-10}$

where $p = P/P_c(1 \leq p \leq 6)$. Determine the value of r and $Z(r, \tau)$ using Eq.(a) for (i) $p = 0.6$ and $\tau = 1/1.05$ and (ii) $p = 2.18$ and $\tau = 1/1.2$. [Answer: (i) $Z = 0.8013$ at $r = 0.7131$ (ii) $Z = 0.5412$ at $r = 3.3567$.]

c. Use Eq. (a) to determine the value τ and $Z(r, \tau)$ when (i) $p = 0.6$ and $r = 1/1.4$ and (ii) $p = 2.18$ and $r = 1/0.6$. [Answer: (i) $Z = 0.8007$ at $\tau = 0.9532$ (ii) $Z = 0.8508$ at $\tau = 0.6505$.]

5.17 The pressure drop of a fluid flowing in a pipe is a function of the pipe's coefficient of friction λ, which can be estimated from the Colebrook formula[32]

$$\lambda = \left[-2\log_{10}\left(\frac{2.51}{\mathrm{Re}\sqrt{\lambda}} + \frac{0.27}{d/k}\right)\right]^{-2} \qquad \mathrm{Re} \geq 4000$$

where Re is the Reynolds number, d is the diameter of the pipe, and k is the surface roughness. For smooth pipes ($k \cong 0$ *or* $d/k > 100{,}000$),

$$\lambda = \left[2\log_{10}\left(\frac{\mathrm{Re}\sqrt{\lambda}}{2.51}\right)\right]^{-2} \qquad \mathrm{Re} \geq 4000$$

For fully developed turbulent flow, the coefficient of friction is given by

$$\lambda = \left[2\log_{10}\left(3.7\frac{d}{k}\right)\right]^{-2}$$

which is independent of Re. It is a special case of the general Colebrook formula.

If the values of λ range from 0.008 to 0.08, then find the value of λ when $\mathrm{Re} = 10^5$ and (i) $d/k = 200$ and (ii) $k = 0$. [Answer: (i) $\lambda = 0.0313$ and (ii) $\lambda = 0.0180$.]

[32] Hwang and Hita, *Hydraulic Engineering Systems*, p. 68.

5.18 Display only the real roots of

$$10x^6 - 75x^3 - 190x + 21 = 0$$

5.19 Display only the real root of

$$3\pi\sqrt{5}\eta^9 - 60\eta^8 + 20 = 0$$

that is greater than 1.

5.20 Find the value of t_o that satisfies

$$\theta(e^{-\pi^2 t_0}) = \frac{\pi}{2\sqrt{5}}$$

where

$$\theta(q) = 2q^{1/4} \sum_{k=0}^{N\rightarrow\infty} \frac{(-1)^k}{2k + 1} q^{k(k+1)}$$

It has been found that $N = 10$ is a sufficient number of terms to sum this series provided that $q < 0.9$.

5.21 Determine the value of x that satisfies

$$x = \left(1 + \frac{1}{x}\right)^x$$

Section 5.5.2

5.22 Find the area between the two sine curves shown in Figure 5.25 using `quadl` and `trapz`. The two sine waves are given by $\sin(x)$ and $|\sin(2x)|/2$, $0 \le x \le 2\pi$.

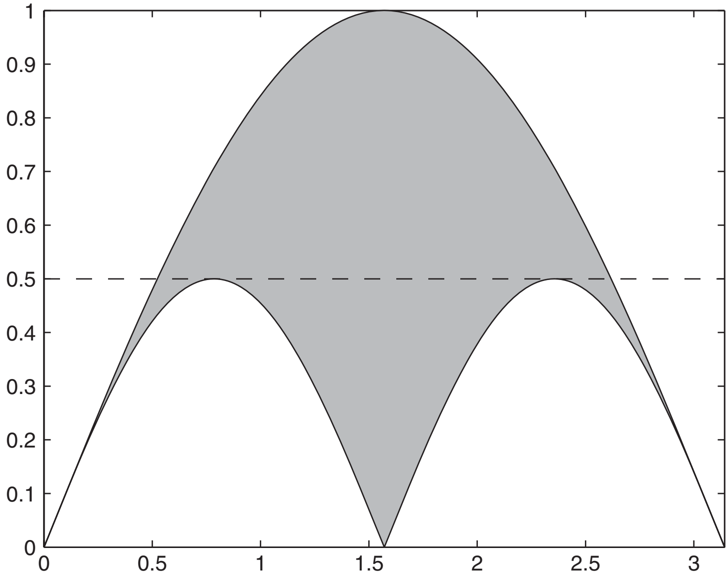


Figure 5.25 Geometry for Exercise 5.22.

5.23 In determining the load distribution in axial thrust bearings under an eccentric load, the following integral must be evaluated:[33]

$$I_m(\varepsilon) = \frac{1}{2\pi}\int_{-a}^{a} [1 - (1 - \cos(x))/2\varepsilon]^c \cos(mx)\, dx$$

where $\varepsilon > 0, m = 0$ or 1,

$$a = \cos^{-1}(1 - 2\varepsilon)$$

and $c = 1.5$ for ball bearings and $c = 1.1$ for roller bearings. Determine the value of $I_1(0.6)$ for a ball bearing. [Answer: $I_1(0.6) = 0.2416$.]

5.24 Given the integral

$$\int_0^\infty E_{\lambda,\,b}(\lambda, T) d\lambda = \sigma T^4$$

where

$$E_{\lambda,b}(\lambda,T) = \frac{C_1}{\lambda^5\left[\exp(C_2/\lambda T) - 1\right]}$$

and λ is the wavelength in μm, T is the temperature in Kelvin, $C_1 = 3.742 \times 10^8$ W-μm^4/m^2, $C_2 = 1.439 \times 10^4$ μm-K, and $\sigma = 5.667 \times 10^{-8}$ W/(m^2-K^4) is the Stephan–Boltzmann constant. Perform this integration numerically for $T = 300$K, 400K and 500K and determine the percentage error of the approximate results compared to the exact value, which is σT^4. Approximate the integral using a lower limit of 1 μm and an upper limit of 200 μm. [Answer: $\text{error}_{300} = 0.0.0611\%$, $\text{error}_{400} = 0.0243\%$, $\text{error}_{500} = 0.0109\%$.] (See also Exercise 12.22.)

5.25 Evaluate the following integral:

$$Z = \int_{\pi/4}^{\pi}\int_{0}^{\pi/2} \cos(x - y)e^{-xy/\pi^2}\, dx\, dy$$

5.26 Use `quadl` to show that

$$\int_2^{\sqrt{6}} \left(6 - u^2\right)^{-1/2} du = \cot^{-1}(\sqrt{2})$$

5.27 As part of the solution to the forced vibrations of Euler beams, one is required to determine the normalization constant[34]

$$N_n = \int_0^1 W_n^2(\eta) d\eta$$

[33] Changsen, *Rolling Element Bearings*, p. 92.

[34] See Section 9.4.2.

where $W_n(\eta)$ is the nth mode shape of the beam corresponding to Ω_n, the nth nondimensional natural frequency coefficient. For a cantilever beam,[35] $\Omega_1 = 0.5969\pi$, $\Omega_2 = 1.4942\pi$, $\Omega_3 = 2.5002\pi$, $\Omega_n = (n - 0.5)\pi$, $n = 4, 5, \ldots$, and

$$W_n(\eta) = -\frac{T(\Omega_n)}{Q(\Omega_n)} T(\Omega_n \eta) + S(\Omega_n \eta) \quad 0 \le \eta \le 1$$

where

$$Q(x) = 0.5[\cosh(x) + \cos(x)]$$
$$S(x) = 0.5[\cosh(x) - \cos(x)]$$
$$T(x) = 0.5[\sinh(x) - \sin(x)]$$

For the cantilever beam, determine the values of N_n, $n = 1, 2, \ldots, 10$.

Section 5.5.3

5.28 Consider the motion of a projectile that leaves a point (0, 0) with an initial velocity v_0 and at an angle with the horizontal of α. If the projectile lands at a location (x_e, y_e) and is subjected to a drag during flight that is proportional to the square of its velocity, then the four first-order equations governing its flight are[36]

$$\frac{dv_x}{dx} = -c_d v \qquad \frac{dv_y}{dx} = \frac{-(g + c_d v v_y)}{v_x} \qquad \frac{dy}{dx} = \frac{v_y}{v_x} \qquad \frac{dt}{dx} = \frac{1}{v_x}$$

where y is the vertical height of the projectile, x is the horizontal distance of travel, t is the time, v_x and v_y are the horizontal and vertical components of the velocity v, respectively, c_d is the drag coefficient, g is the gravity constant (9.8 m/s^2), and

$$v = \sqrt{v_x^2 + v_y^2}$$

These equations are valid only when v_0 is large enough so that v_x is greater than zero when it reaches x_e. The test for this condition can be stated as, say, $|v_x| > v_0 \times 10^{-6}$. If this condition is not satisfied, then the program's execution must be terminated. Use `error` to cause the termination. This check is placed in the beginning of the function that is called by `ode45`. The initial conditions are

$$v_{0x} = v_0 \cos(\alpha) \quad v_{0y} = v_0 \sin(\alpha) \quad y = 0 \quad t = 0$$

From the order in which the equations are written, let $y_1(x) = v_x$, $y_2(x) = v_y$, $y_3(x) = y$, and $y_4(x) = t$. For $v_0 = 180$ m/s, $c_d = 0.007$, and $\alpha = 45°$ and for $x_{\text{final}} = 300$m in `ode45`:

a. What is the value of the maximum elevation of the projectile and at what distance does this occur. Use `spline` and `fminbnd` to determine these values. [Answer: $y_{\max} = 137.26$ m at $x = 187.87$ m.]

b. How long does it take for the projectile to travel x_e when $y_e = 0$. Use `interp1` to determine these values. [Answer: $x_e = 280.77$m and the time of travel is 10.353 s.]

[35] Balachandran and Magrab, *Vibrations*, pp. 574 and 578.

[36] H. B. Wilson and L. H. Turcotte, *Advanced Mathematics and Mechanics Applications Using MATLAB*, 2nd ed., CRC Press, Boca Raton, FL, 1997, p. 294.

5.29 A bungee jumper is preparing to make a high altitude jump from a hot air balloon using a length L of bungee line. In order to do so safely, the peak acceleration, velocity, and total drop distance must be predicted so that the arresting force is not too great and the balloon is high enough so that the jumper doesn't hit the ground. Taking into account the aerodynamic drag forces, the governing equation is[37]

$$\frac{d^2x}{dt^2} + c_d\,\text{signum}(dx/dt)\left(\frac{dx}{dt}\right)^2 + \frac{k}{m_J}(x - L)u(x - L) = g$$

where $g = 9.8$ m/s^2 is the acceleration of gravity, c_d is proportional to the drag coefficient and has the unit of m^{-1}, k is the spring constant of the bungee cord in N/m, m_J is the mass of the jumper, and $u(z)$ is the unit step function—that is, $u(z) = 0$ when $z \le 0$ and $u(z) = 1$ when $z > 0$. The programming is greatly simplified if the logical operator described in Section 4.1 is used to describe $u(t)$.

If $L = 150$ m, m$_J = 70$ kg, $k = 10$ N/m, $c_o = 0.00324$ m^{-1}, and the initial conditions are zero, then show that

1. The maximum distance traveled is 308.47 m, which occurs at 11.47 s.
2. The jumper will reach 150 m in 5.988 s traveling at a velocity of −43.48 m/s.
3. The maximum acceleration will be −12.82 m/s^2(−1.308 g) at 11.18 s.

The numerical results stated above were obtained by using `spline` on the appropriate outputs from `ode45`.

5.30 Consider an inverted pendulum that is composed of a weightless rigid rod of length L to which a mass m and a linear spring of spring constant k are attached at its free end. The pendulum is initially vertical. The unstretched length of the spring is L. The rotation of the pendulum's pivot has a damping c, and the pendulum is driven by a moment $M(t)$. The governing equation describing the angular motion is[38]

$$\frac{d^2\theta}{d\tau^2} + \alpha\frac{d\theta}{d\tau} - \sin\theta + \beta\left(1 - \frac{1}{\sqrt{5 - 4\cos\theta}}\right)\sin\theta = P(t)$$

where

$$\beta = \frac{2kL}{mg} \quad P = \frac{M}{mgL} \quad \tau = t\sqrt{\frac{g}{L}} \quad \alpha = (c/m)\sqrt{L/g}$$

and t is the time.

If $M = 0$, $\beta = 10$, $\alpha = 0.1$, $\theta(0) = \pi/4$, and $d\theta(0)/d\tau = 0$, then plot the rotation θ as a function of τ for 1,000 equally spaced values of τ from $0 \le \tau \le 50$ and in a separate figure, plot $\theta(\tau)$ versus $d\theta(\tau)/d\tau$, which is called the phase portrait.

5.31 The oscillations of the height Z of the separation between the fluid levels in two rectangular prismatic reservoirs connected by a long pipeline can be determined from[39]

$$\frac{d^2Z}{dt^2} + \text{signum}(dZ/dt)\,p\left(\frac{dZ}{dt}\right)^2 + qZ = 0$$

[37] See, for example, D. M. Etter, *Engineering Problem Solving with MATLAB*, Prentice Hall, Upper Saddle River, NJ, 1997, pp. 220–221.

[38] Wilson and Turcotte, *Advanced Mathematics*, p. 279.

[39] D. N. Roy, *Applied Fluid Mechanics*, Ellis Horwood Limited, Chichester, UK, 1988, pp. 290–293.

If $p = 0.375\ \text{m}^{-1}$, $q = 7.4 \times 10^{-4} s^{-2}$ and the initial conditions are $Z(0) = Z_n$ m and $dZ(0)/dt = 0$ m/s, then determine the value of the *first* occurrence of $t_n, n = 1, 2$, for which $Z(t_n) = 0$ when $Z_1 = 10$ m and $Z_2 = 50$ m. Use interp1 to determine t_n. The quantity signum is determined with sign. *Suggestion*: Use an appropriate combination of min and find to determine the index of the first value of t at which Z is negative. Then take a small range of values of t around this value over which interp1 should perform the interpolation. [Answers: for $Z_1 = 10$ m, $t_1 = 114.2692$s andfor $Z_2 = 510$m, $t_2 = 276.1428$ s.]

5.32 Consider Eq. (5.3) and its numerical solution to a step input; that is, $h(t) = u(t)$. Determine the value of ξ that makes the following quantity a minimum:

$$f(\xi) = \sum_{n=1}^{N} (y(t_n) - 1)^2$$

where $y(t_n), n = 1, 2, \ldots, N$ are the values of y at the times t_n over the range $0 \le t \le 35$ that are determined by ode45.

5.33 Determine the solution to the following system of nonlinear ordinary differential equations:

$$\frac{dy_1}{dt} = y_2$$
$$\frac{dy_2}{dt} = -\left(\frac{2}{L}\frac{dL}{dt} + \gamma L\right) y_2 - \frac{1}{L}\sin(y_1)$$

where

$$L = 1 + \varepsilon \sin^7(\omega t + 9\pi/8)$$
$$\frac{dL}{dt} = 7\varepsilon\omega \sin^6(\omega t + 9\pi/8)\cos(\omega t + 9\pi/8)$$

The initial conditions are $y_1(0) = -1$ and $y_2(0) = 1$ and the constants have the following values: $\varepsilon = 0.16$, $\gamma = 0.4$, and $\omega = 0.97$. Plot $y_1(t)$ versus $y_2(t)$.

5.34 Lord Rayleigh modeled a clarinet reed using the following equation:

$$\frac{d^2x}{dt^2} - \alpha\frac{dx}{dt} + \beta\left(\frac{dx}{dt}\right)^3 + kx = 0$$

where the parameter k is proportional to stiffness and α and β indicate the relative contributions of damping: α for negative damping when the velocity is small and β for positive damping when the velocity is large. For $\alpha = 0.5$, $\beta = 0.6$, and $k = 1$, plot x versus dx/dt.

5.35 The mechanism shown in Figure 5.26 in its postbuckled state is undergoing harmonic excitation. When it undergoes forced harmonic excitation in the x-direction of magnitude U_o, the governing equation of motion is[40]

$$\left[\frac{1}{3} + (1 + 2rp)\sin^2\varphi\right]\frac{d^2\varphi}{d\tau^2} + (1 + 2rp)\left(\frac{d\varphi}{d\tau}\right)^2 \sin\varphi\cos\varphi$$
$$+ (1 + rp)\omega^2 u_0 \sin\omega\tau \sin\varphi + 2c\frac{d\varphi}{d\tau} + 2\varphi - p\sin\varphi = 0$$

[40] R. H. Plaut, L. A. Alloway, and L.N. Virgin, "Nonlinear Oscillations of a Buckled Mechanism Used as a Vibration Isolator," in *Chaotic Dynamics and Control of Systems and Processes in Mechanics*, G. Rega and F. Vestroni, Eds., Springer, Dordrecht, The Netherlands, 2005, pp. 242–250.

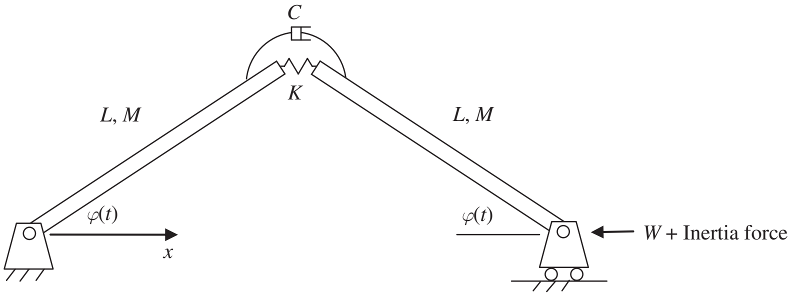


Figure 5.26 Buckled mechanism undergoing harmonic acceleration at its right end.

where

$$r = \frac{K}{MgL} \quad p = \frac{WL}{K} \quad \tau = \frac{t}{L}\sqrt{\frac{K}{M}} \quad c = \frac{C}{L\sqrt{KM}} \quad u_0 = \frac{U_0}{L}$$

are nondimensional quantities, W is the weight that caused the mechanism to buckle, and $u_o\omega^2$ is the magnitude of the axial acceleration at the nondimensional frequency ratio ω and nondimensional amplitude u_o. The axial motion of the supported weight about its equilibrium position is

$$x(\tau) = u_0 \sin\omega\tau + 2\cos\varphi - 2\cos\varphi_e$$

where φ_e is the static equilibrium position determined from

$$2\varphi_e - p\sin\varphi_e = 0$$

Plot $\varphi(\tau)$ and $x(\tau)$, $0 \le \tau \le 800$, for $u_o = 0.05$, $c = 0.02$, $r = 1$, $\omega = 0.2$, and $p = 2.01$ when the initial conditions are $\varphi(0) = \varphi_e$ and $d\varphi(0)/d\tau = 0$. To obtain the solution of this equation requires that the relative tolerance in `ode45` be set to 10^{-5}.

Section 5.5.4

5.36 Determine the solution to the system of nonlinear first-order ordinary differential equations

$$\frac{du}{dx} = \frac{u}{2v}(w - u)$$

$$\frac{dv}{dx} = -\frac{1}{2}(w - u)$$

$$\frac{dw}{dx} = \frac{1}{z}\left(0.9 - 1000(w - y) - \frac{w}{2}(w - u)\right)$$

$$\frac{dz}{dx} = \frac{1}{2}(w - u)$$

$$\frac{dy}{dx} = -100(y - w)$$

subject to the boundary conditions: $u(0) = v(0) = w(0) = 1, z(0) = -10$, and $w(1) = y(1)$. For the initial guesses for the solutions, use $[1, 1, 1, -10, 0.91]$. For the initial guesses for the domain, assume five equally spaced points. Plot u, v, and w on the same figure by using three plot commands with `hold on` following the first plot command.

5.37 Consider a uniform inextensible cable of length L_o and weight per unit length w that hangs between two fixed points $x = 0$ and $x = L$ such that $L < L_o$. If the cable has no flexural rigidity and can only support tensile forces T, then the governing equation of the nondimensional deflection $z(\eta)$ of the cable is[41]

$$\frac{d^2z}{d\eta^2} = \beta\sqrt{1 + \left(\frac{dz}{d\eta}\right)^2} \qquad \text{(a)}$$

where $\eta = x/L$, $\beta = wL/H$, H is the horizontal component of T, and a negative z indicates a downward deflection. The corresponding length L_o of the cable is equal to

$$L_0 = L\int_0^1 \sqrt{1 + \left(\frac{dz}{d\eta}\right)^2}d\eta \qquad \text{(b)}$$

from which one can determine β and hence H when w, L, and L_o are given. The boundary conditions are

$$z(0) = 0 \text{ and } z(1) = 0$$

Determine the value of β and the slope $dz(0)/d\eta$ when $L_o/L = 1.2$. The solution method uses `fzero` to satisfy Eq. (b) for the given value of L_o/L by varying β in obtaining z, the solution to Eq. (a). Also, the integration of Eq. (b) must be performed with `trapz`.

5.38 Consider the transverse displacement $y(\eta)$ of a uniform beam clamped at $\eta = 0$ and free at $\eta = 1$. The boundary conditions are $y(0) = dy(0)/d\eta = d^2y(1)/d\eta^2 = d^3y(1)/d\eta^3 = 0$. Plot the displacement of the beam when there is a uniform load of unit magnitude on the beam from $\eta = 0.5$ to $\eta = 1$.

5.39 The governing equation of motion in terms of the slope φ of a cantilever beam undergoing large deflections from a follower force $\overline{P}$ acting at an angle α at the free end of the beam in terms of its arc length s is[42]

$$\frac{d^2\varphi}{ds^2} + \overline{P}\sin(\varphi + \alpha - \varphi(0)) = 0$$

with the boundary conditions

$$\varphi(0) = 0$$
$$\frac{d\varphi(l)}{ds} = 0$$

[41] Irvine, *Cable Structures*, p. 4.

[42] B. S. Shvartsman, "Large deflections of a cantilever beam subjected to a large follower force," *Journal of Sound and Vibration*, **304**, 2007, pp. 969–973.

When $\alpha = \pi/2$, the force is normal to the beam. The vertical displacement component of the beam is determined from

$$y(s) = \int_s^1 \sin \varphi(x)dx$$

Show that when $\alpha = \pi/2$ and $P = 8$, $y(0) = -0.0738$ and $y(1) = 0.634$. [Hint: The solution is very sensitive to the initial guess for $\varphi(0)$; it should be assumed to be the region $157° - 158°$. In addition, the relative tolerance used by `bvp4c` has to be changed to 10^{-3}; use `bvpset`.]

5.40 Consider the following nonlinear differential equation:

$$\frac{d^2w(z)}{dz^2} + \frac{2z}{\sqrt{1 - \alpha w(z)}}\frac{dw(z)}{dz} = 0$$

with the following boundary conditions: $w(0) = 1$ and $w(\infty) = 0$. Since `bvp4c` cannot consider a boundary condition at ∞, the objective is to determine a finite value of z_∞ that is sufficient to provide a reasonable approximation to $z = \infty$. Thus, the second boundary condition is restated as $w(z_\infty) \cong 0$. Let us call the various trial values $z_{n,\infty}$, where $n = 1, 2, \ldots$ indicates the nth trial value. Corresponding to these trial values are the solutions $w(z, z_{n,\infty})$. We shall base our decision of when $z_{n,\infty}$ gives a reasonable approximation to $w(z, z_\infty)$ by using the following criterion: $[(w(1.1, 1.05z_{k,\infty}) - w(1.1, z_{k,\infty})]/0.05 \le 0.03$. For $\alpha = 0.8$, determine an approximate value of $z_{n,\infty}$ for which the criterion is satisfied. [Answer: $z_\infty \cong 1.9501$.]

5.41 The transverse displacement $w(r)$ of a symmetrically loaded circular plate of outer radius a, Young's modulus E, thickness h, and Poisson's ratio ν is given by[43]

$$\frac{d^4y}{d\xi^4} + \frac{2}{\xi}\frac{d^3y}{d\xi^3} - \frac{1}{\xi^2}\frac{d^2y}{d\xi^2} + \frac{1}{\xi^3}\frac{dy}{d\xi} = q(\xi)$$

where $\xi = r/a$ and $y = y(\xi) = w(\xi)/h_o$,

$$h_0 = \frac{Q_0a^4}{D} \qquad D = \frac{Eh^3}{12(1 - \nu^2)}$$

and Q_o is the magnitude of the load per unit area and $q(\xi)$ is the shape of the load as a function of ξ. The moment M per unit length and shear force V per unit length are given by, respectively,

$$M_\xi = \frac{M}{a^2Q_0} = -\left(\frac{d^2y}{d\xi^2} + \frac{\nu}{\xi}\frac{dy}{d\xi}\right)$$

$$V_\xi = \frac{V}{aQ_0} = -\left(\frac{d^3y}{d\xi^3} + \frac{1}{\xi}\frac{d^2y}{d\xi^2} - \frac{1}{\xi^2}\frac{dy}{d\xi}\right)$$

[43] Timoshenko and Woinowsky-Krieger, *Theory of Plates and Shells*, p. 282ff.

Consider an annular plate that is clamped along the inner perimeter at $\xi = 0.2$ and is free along the outer perimeter $\xi = 1$. The boundary conditions at $\xi = 0.2$ are

$$y(0.2) = \frac{dy(0.2)}{d\xi} = 0$$

and those at $\xi = 1$ are

$$M_\xi = V_\xi = 0$$

If the plate is uniformly loaded, that is, $q(\xi) = 1$, and $\nu = 0.3$, then determine the displacement, slope, moment, and shear force as a function of ξ and plot the results.

5.42 Consider an Euler cantilever beam that is subjected to a transverse loading and a compressive axial force of magnitude P. Equation (5.6) becomes

$$\frac{d^4y}{d\eta^4} + S\frac{d^2y}{d\eta^2} = q(\eta) \quad 0 \le \eta \le 1$$

where $S = PL^2/EI$. The simply supported and clamped boundary conditions given by Eqs. (5.7a) and (5.7b), respectively, remain the same and the free boundary condition becomes

$$\frac{d^3y}{d\eta^3} + S\frac{dy}{d\eta} = 0 \quad \text{and} \quad \frac{d^2y}{d\eta^2} = 0$$

For a uniformly loaded cantilever beam, determine the displacement, slope, moment, and shear force when $S = 0.4$ and plot the results.

5.43 The governing equations of a Timoshenko beam with constant cross section and undergoing harmonic oscillations at radian frequency ω are given by[44]

$$\frac{dW^2}{d\eta^2} + \gamma_{bs}R_0^2\Omega^4W - \Psi' = 0 \qquad 0 \le \eta \le 1$$

$$\gamma_{bs}R_0^2\frac{d\Psi^2}{d\eta^2} + \left(\gamma_{bs}R_0^4\Omega^4 - 1\right)\Psi + \frac{dW}{d\eta} = 0 \qquad 0 \le \eta \le 1$$

where

$$\eta = x/L \quad \Omega^4 = \omega^2t_0^2 \quad t_0 = \sqrt{\frac{\rho AL^4}{EI}} \quad R_0 = r_0/L \quad r_0 = \sqrt{I/A} \quad \gamma_{bs} = 2(1+\nu)/\kappa$$

and $W = W(\eta)$ is the transverse displacement, $\Psi = \Psi(\eta)$ is the angle of rotation of the cross section due to bending only, L is the length of the beam, ρ is the mass density, A is the cross-sectional area, E is the Young's modulus, ν is Poisson's ratio, I is the

[44] E. B. Magrab, "Natural Frequencies and Mode Shapes of Timoshenko Beams with Attachments," *Journal of Vibration and Control*, **13**, No. 7, 2007, pp. 905–934.

moment of inertia of the cross section, and $\kappa$is the shear correction actor, which is a constant relating A to an effective area over which the shear stress is constant and is a function of the cross-sectional shape. The classical boundary conditions are

Clamped

$$W = \Psi = 0$$

Simply Supported (Hinged)

$$W = \frac{d\Psi}{d\eta} = 0$$

Free

$$\frac{d\Psi}{d\eta} = 0$$

$$\frac{dW}{d\eta} - \Psi = 0$$

Determine the lowest natural frequency coefficient Ω for a cantilever beam when $\gamma_{bs} = 3.12$ and $R_o = 0.06$. Use as a guess for Ω a value of $\pi/2$. See Example 5.20 for a guess of the spatial distribution of W and Ψ. [Answer: $\Omega = 1.8444$.]

Section 5.5.6

5.44 The relationship between the lead angle of a worm gear λ, the ratio $\beta = N_1/N_2$, where N_1 and N_2 are the number of teeth on the worm gear and the driven gear, respectively, the center distance C between shafts, and the normal diametral pitch P_{dn} is[45]

$$K = \frac{2P_{dn}C}{N_2} = \frac{\beta}{\sin\lambda} + \frac{1}{\cos\lambda}$$

The ranges of practical interest are $1 \le K \le 2$, $1° \le \lambda \le 40°$, and $0.02 \le \beta \le 0.30$. For certain combinations of values, λ can have one value, two values, or no value.

a. Find the value of λ that makes K a minimum when $\beta = 0.02, 0.05, 0.08, 0.11, 0.15, 0.18, 0.23$, and 0.30.

b. For $K = 1.5$ and $\beta = 0.16$, find the value(s) of λ.

5.45 In Exercise 1.24, the mass flow rate of a gas escaping from a tank at pressure p_0 and under reversible adiabatic conditions was proportional to

$$\psi = \sqrt{\frac{k}{k-1}}\sqrt{\left(\frac{p_e}{p_0}\right)^{2/k} - \left(\frac{p_e}{p_0}\right)^{(k+1)/k}}$$

[45] M. F. Spotts and T. E. Shoup, *Design of Machine Elements*, Prentice Hall, Upper Saddle River, NJ, 1998, p. 613.

where p_e is the pressure exterior to the tank's exit and k is the adiabatic reversible gas constant. The maximum value occurs at

$$\frac{p_e}{p_0} = \left(\frac{2}{k+1}\right)^{\frac{k}{k-1}}$$

Verify this maximum value numerically for $k = 1.4$ by using `fminbnd` and by using `min` with 200 equally spaced values for $0 \le p_e/p_0 \le 1$.

Section 5.5.7

5.46 Use `fsolve` to obtain a solution to the following set of equations:

$$\begin{aligned} \sin x + y^2 + \ln z &= 7 \\ 3x + 2^y - z^3 &= -1 \\ x + y + z &= 5 \end{aligned}$$

5.47 **a.** Use `fsolve` to find the values of θ in degrees and k that satisfy the following equations when $a = 1$ and $b = 3$:

$$\begin{aligned} b &= k(1 - \cos\theta) \\ a &= k(\theta - \sin\theta) \end{aligned}$$

b. The two equations in (a) can be combined into the following one equation:

$$b(\theta - \sin\theta) - a(1 - \cos\theta) = 0$$

Use `fzero` to determine the value of θ when $a = 1$ and $b = 3$, and then use one of the equations in (a) to determine k. [Answers: $k = 6.9189$ and $\theta = 55.4999°$.]

c. Obtain the solutions to part (a) using `solve`.

5.48 **a.** Use `fsolve` to determine from the following equations the values of Q, T_A, and T_B when $\sigma = 5.667 \times 10^{-8}$, $T_1 = 373\text{K}$, and $T_2 = 293\text{K}$.

$$\begin{aligned} T_1^4 - T_A^4 &= Q/\sigma \\ T_A^4 - T_B^4 &= Q/\sigma \\ T_B^4 - T_2^4 &= Q/\sigma \end{aligned}$$

b. The equations in (a) can also be written as

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & -1 & -1 \\ 0 & 1 & -1 \end{bmatrix} \begin{Bmatrix} x \\ y \\ Q/\sigma \end{Bmatrix} = \begin{Bmatrix} T_1^4 \\ 0 \\ T_2^4 \end{Bmatrix}$$

where $x = T_A^4$ and $y = T_B^4$. Determine the values of Q, T_A, and T_B from this system of equations using matrix left division. [Answer: $T_A = 352.052$, $T_B = 326.5116$, and $Q = 226.4312$.]

Section 5.6

5.49 Use `solve` to determine the value of x between $0 \leq x \leq \pi$ of the function given below that makes $f(x)$ an extremum. Is this extremum a maximum or a minimum?

$$f(x) = e^{\sin x}$$

5.50 Determine the value of the integral given below for ten equally spaced values of b from 0 to 4π.

$$A(b) = \int_0^b \frac{2x + 5}{x^2 + 4x + 5} dx$$

5.51 For the following function:

$$f(\alpha) = (2 + \sin(10\alpha)) \int_0^{1.8} x^a \sin\left(\frac{\alpha}{2 - x}\right) dx$$

find the maximum and minimum values of $f(\alpha)$ and the values of α where these extremes occur.

5.52 The nondimensional natural frequency coefficients Ω for the nonplanar vibrations of a cylindrical shell can be obtained from the solution to[46]

$$\begin{vmatrix} -\Omega^2 + k_m^2 a^2 + \frac{1}{2}(1-\nu)n^2 & \frac{1}{2}(1+\nu)k_m a & \nu k_m a \\ \frac{1}{2}(1+\nu)k_m a & -\Omega^2 + \frac{1}{2}(1-\nu)k_m^2 a^2 + n^2 & n \\ \nu k_m a & n & -\Omega^2 + 1 + \beta^2(k_m^2 a^2 + n^2)^2 \end{vmatrix} = 0$$

where $2\pi/k_m$ is the axial wavelength, ν is Poisson's ratio, n is a positive integer, a is the inner radius of the shell, $\beta^2 = h^2/12a^2$, and h is the thickness of the shell. Find the positive values of Ω for $k_m a = \pi/4$, $\nu = 0.3$, $n = 0, 1, \ldots, 10$, and $\beta = 0.05/\sqrt{12}$ in three ways:

a. By using the Symbolic toolbox to obtain a third-order polynomial in Ω^2, and then using these results in `roots` to determine Ω.

b. By using `eig`.

c. By using `fzero` on the determinant given above.

[46] M. C. Junger and D. Feit, *Sound, Structures, and Their Interactions*, MIT Press, Cambridge, MA, 1972, pp. 242–244.

6

2D Graphics

Edward B. Magrab

The implementation of a wide range of two-dimensional plotting capabilities is presented.

6.1 INTRODUCTION: GRAPHICS MANAGEMENT

MATLAB provides a wide selection of very flexible and easy-to-implement two- and three-dimensional plotting capabilities. The plotting capabilities can be grouped into three categories: graphics management, curve and surface generation, and annotation and graph characteristics. Although there are quite a few plotting functions, for the most part their syntax is similar and they can be annotated with the same set of functions. The functions whose usage we shall illustrate in this chapter and in Chapter 7 are given in Table 6.1.

The purpose of a graphic is to communicate complex ideas with clarity, precision, and efficiency and these goals are attained when the viewer obtains the greatest

TABLE 6.1 Plotting Capabilities Categories of Graphics Management, Curve and Surface Generation, and Annotation and Graph Characteristics

Management	Generation	Annotation and characteristics
`figure`	2D	2D and 3D
`hold`	`axes`	`axis, axis equal, axis off, axis image`
`subplot`	`bar`	`box`
`zoom`	`convhull`	`clabel`
	`delauney`	`grid`
3D	`fill`	`legend`
`rotate3d`	`image`	`set`
`view`	`loglog`	`text`
	`movie`	`title`
	`patch`	`xlabel`
	`pie`	`xlim`
	`plot`	`ylabel`
	`plotyy`	`ylim`
	`polar`	
	`semilogx`	3D
	`semilogy`	`axis vis3d, axis ij`
	`stairs`	`colorbar`
	`stem`	`colormap`
	`voronoi`	`shading`
		`zlabel`
	3D	
	`contour, contour3, contourf`	
	`cylinder`	
	`ellipsoid`	
	`fill3`	
	`mesh, meshc, meshz`	
	`pie3`	
	`plot3`	
	`ribbon`	
	`sphere`	
	`surf, surfc`	
	`waterfall`	

number of ideas in the shortest time from the least amount of ink.[1] Thus, graphed entities should illustrate what is important and exhibit clarity and specificity by being fully annotated with the axes labeled, curves identified (if more than one), and important numerical values displayed. However, any devices that are used to enhance the figure, such as color, line type, symbols, and text should do so without being distracting.

A typical set of graph-creating expressions consists of management functions, followed by one or more graph-generation functions, and followed in turn by annotation functions, which may be followed by additional management functions. However, except for the management functions, the order of these functions is, in many applications, arbitrary. Also, the employment of the annotation and graph characteristic functions is optional. MATLAB scales the axes and labels the axes' magnitudes, even if more than one set of data is plotted. Thus, one can always obtain a partially annotated graph, provided that MATLAB's graphics syntax has been used correctly.

Some Graph Management Functions

A graph is created in a figure window, which is a window created by MATLAB at execution time, when any one of its graph management, generation, or annotation and characteristics functions is invoked. When a program, either a script or function, uses more than one graph-generation function, MATLAB creates a new figure window. To retain each new graph in its own figure window, one must use

```
figure(n)
```

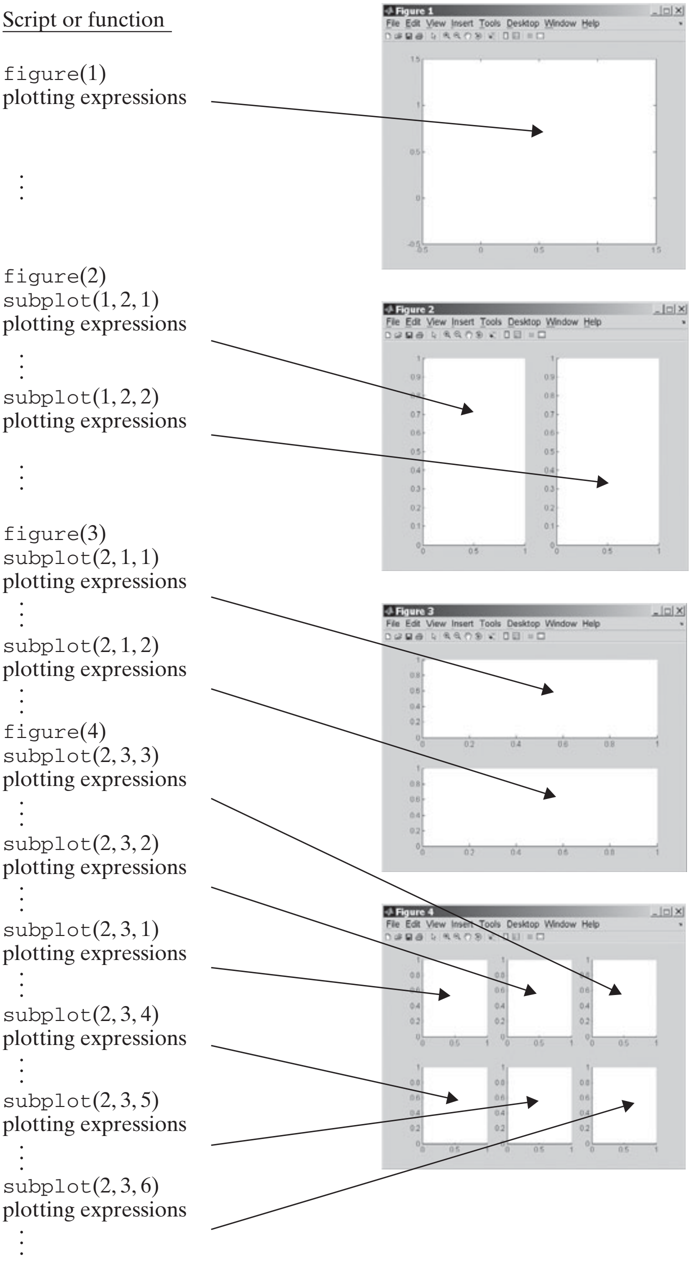
where *n* is an integer. If *n* is omitted, then MATLAB gives it the next integer value.

One can also place several independently created graphs in one figure window with

```
subplot(i, j, k)
```

The first two arguments divide the window into sectors (rows and columns), and the third argument indicates in which sector a graph is to be placed. A value of 1 for third argument indicates the upper left corner, and the product of the number of rows and number of columns indicates the lower right corner. As the numbers increase, they indicate the sectors from left to right, starting at the top row. Any annotation and management functions that appear in the program after `subplot` apply only to that sector indicated by the third argument of `subplot`. Within each sector, any compatible set of the 2D or 3D graph generation functions can be used. Refer to Figure 6.1 to see several examples of how `figure` and `subplot` can be used. It is noted that if only one `figure` window is needed, `figure` can be omitted, even if `subplot` is used.

[1] E. Tufte, *Visual Display of Quantitative Information*, Graphics Press, Cheshire, CT, 1997.



Script or function

```
figure(1)
plotting expressions
  ⋮
figure(2)
subplot(1,2,1)
plotting expressions
  ⋮
subplot(1,2,2)
plotting expressions
  ⋮
figure(3)
subplot(2,1,1)
plotting expressions
  ⋮
subplot(2,1,2)
plotting expressions
  ⋮
figure(4)
subplot(2,3,3)
plotting expressions
  ⋮
subplot(2,3,2)
plotting expressions
  ⋮
subplot(2,3,1)
plotting expressions
  ⋮
subplot(2,3,4)
plotting expressions
  ⋮
subplot(2,3,5)
plotting expressions
  ⋮
subplot(2,3,6)
plotting expressions
  ⋮
```

Figure 6.1 Examples of the use of various combinations of `figure` and `subplot`.

Since each graph-generation function creates a new figure window,[2] to draw more than one curve, surface, or line (or combination of these) on a given graph, one must use

```
hold on
```

which holds the current window or `subplot` sector active.

All figures that have been created can be copied to the Windows clipboard by selecting *Copy Figure* from the *Edit* pull-down menu within each figure window. This figure can then be transferred (pasted) to a page in a word processor program and will be in the Windows metafile format.

MATLAB provides the means to convert a figure to a format compatible with many common print devices. For example, if one wants to save the graphics appearing in the active figure window as a level-2 encapsulated postscript file for black-and-white printers with the name *FileName.eps*, then one uses

```
print('-deps2', 'c:/path/FileName.eps')
```

where *-deps2* is a keyword to indicate that a level-2 encapsulated postscript file is to be created and *path* describes the directory and subdirectory names where the file will reside. For other options, see the *Help* file for `print`. On the other hand, if one wants to insert a level-2 encapsulated postscript file into an MS Word document such that a “tiff” preview image of the figure is displayed in the document, then one uses the following expression:

```
print('-deps2', '-tiff', 'c:\path\FileName.eps')
```

where *-tiff* is a keyword indicating that a tiff preview is available.

6.2 BASIC 2D PLOTTING COMMANDS

6.2.1 Introduction

The basic 2D plotting command is

```
plot(u, v, c)
```

where u and v are the x- and y-coordinates, respectively, of a point or a series of points. Each set of u and v is a pair of numbers, vectors of the same length, matrices of the same order, or expressions that, when evaluated, result in one of these three quantities. The quantity c is a string of characters: one character specifies the line/point color, one character specifies the point type if points are to be plotted, and up to two characters are used to specify the line characteristics. These various line and point characteristics are given in Table 6.2. When a series of points are to be plotted, one of the characters of c_j can be, for example, an “s” to plot a square or an

[2] The MATLAB window look, management, and file management descriptions relate to a Windows environment. Equivalent procedures are used with other operating systems.

TABLE 6.2 Line, Point, and Fill Characteristics

Line type		Line, point, or fill color		Point type	
Symbol	Description	Symbol	Description	Symbol	Description
-	Solid	r	Red	+	Plus sign
--	Dashed	g	Green	o	Circle
:	Dotted	b	Blue	*	Asterisk
-.	Dashed-dot	c	Cyan	.	Point
		m	Magenta	x	Cross
		y	Yellow	s	Square
		k	Black	d	Diamond
		w	White	∧	Upward-pointing triangle
				∨	Downward-pointing triangle
				>	Right-pointing triangle
				<	Left-pointing triangle
				p	Pentagram
				h	Hexagram

asterisk "*" to plot an asterisk. When the points, whether or not they are to be displayed, are to be connected with straight lines, the characters of c_j can be, for example, a "-" for a solid line and a "--" for a dashed line. When both the lines and points are to be plotted with the same color, the c_j contains both descriptors. For example, if we were to plot blue dashed lines connecting blue diamonds, c_j would be "b--d". The order of the three sets of characters within the single quotes is not important.

When both lines and points are to be plotted, but the points defining the line ($u1$, $v1$) are different from the points that are to be plotted ($u2$, $v2$), we use either

```
plot(u1, v1, c1, u2, v2, c2)
```

or

```
plot(u1, v1, c1)
hold on
plot(u2, v2, c2)
```

where c_1 contains the symbols for the line type and color and c_2 contains the symbols for the point type and color. If c_j is omitted, then the system assumes that only a line is to be drawn and system's default values are used. If more than one curve is drawn, then the line colors change according to the default sequence.

The basic 2D plotting command also has the capability to change the attributes of the lines and points that are being plotted. These attributes can be changed by one of two ways. The first way is with

```
plot(u1, v1, c1,'KeyWord', KeyWordValue, . . . )
```

where 'KeyWord' is a string expression of the keyword for one of the line and point attributes and *KeyWordValue* is either a numerical value or a string expression, depending on 'KeyWord'. One may use as many pairs of keywords and their values as required. The keywords that are used to change the line and point attributes of the various characteristics of a graph are found in *Handle Graphics* file as indicated in Figure 6.2 by going from *Axes* to *Core Objects* to *Line*.

The second way to change the line and point attributes is with the combination of getting the handle to the plotted curve and to then changing the attribute using `set` as follows:

```
hdl = plot(u1, v1, c1);
set(hdl, 'KeyWord', KeyWordValue, ... )
```

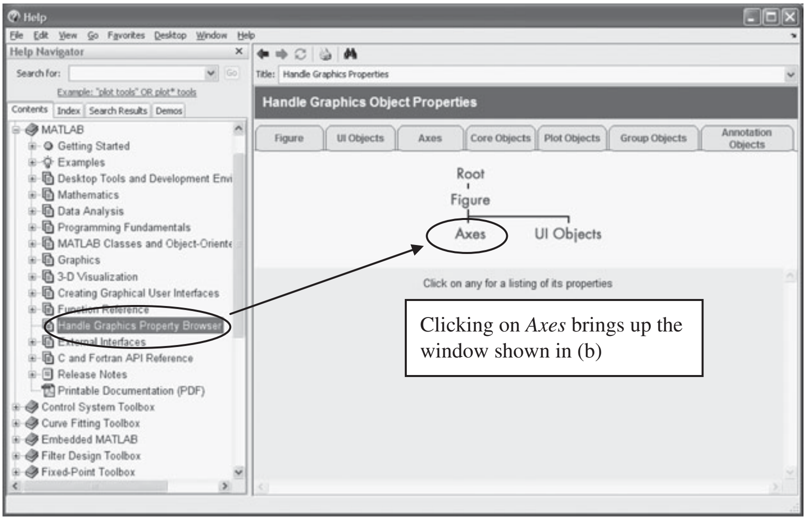
Again, one may use as many pairs of keywords and their values as required. Illustrations of how these methods are used are given subsequently along with how one goes about changing other attributes of a graph.

We now illustrate how one can draw points, lines, circles, and families of curves.

Plotting Points

To place a red asterisk at the location (2, 4), the plotting instruction is

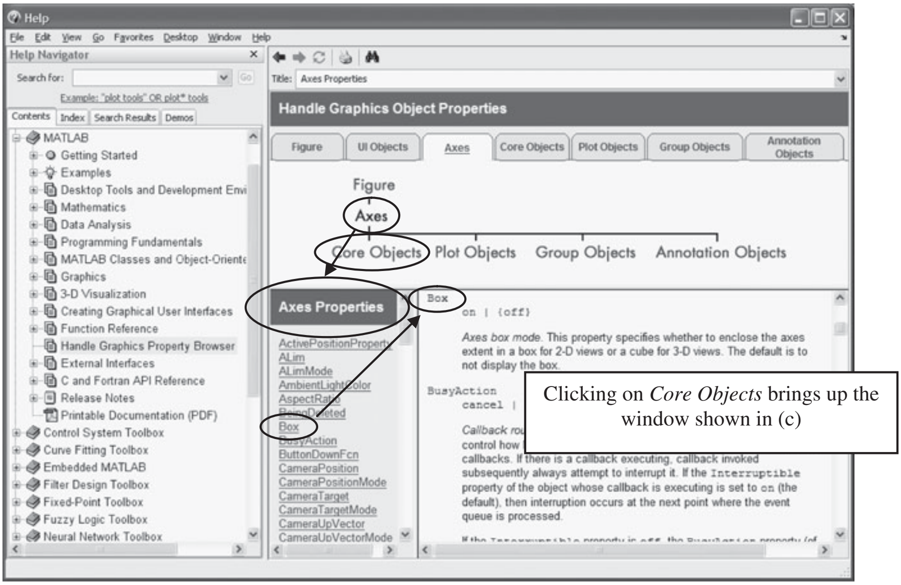
```
plot(2, 4, 'r*')
```


Help
File Edit View Go Favorites Desktop Window Help
Help Navigator
Search for:
Go
Example: "plot tools" OR plot* tools
Contents Index Search Results Demos
MATLAB
Getting Started
Examples
Desktop Tools and Development Envi
Mathematics
Data Analysis
Programming Fundamentals
MATLAB Classes and Object-Oriente
Graphics
3-D Visualization
Creating Graphical User Interfaces
Function Reference
Handle Graphics Property Browser
External Interfaces
C and Fortran API Reference
Release Notes
Printable Documentation (PDF)
Control System Toolbox
Curve Fitting Toolbox
Embedded MATLAB
Filter Design Toolbox
Fixed-Point Toolbox
Title: Handle Graphics Properties
Handle Graphics Object Properties
Figure
UI Objects
Axes
Core Objects
Plot Objects
Group Objects
Annotation Objects
Root
Figure
Axes
UI Objects
Click on any for a listing of its properties

Clicking on *Axes* brings up the window shown in (b)

(a)

Figure 6.2 Sequential use of the *Help* file to determine the appropriate keywords and their values that are needed to change specific attributes of a plot.



```
Box
      on | {off}
```

Axes box mode. This property specifies whether to enclose the axes extent in a box for 2-D views or a cube for 3-D views. The default is to not display the box.

```
BusyAction
      cancel |
```

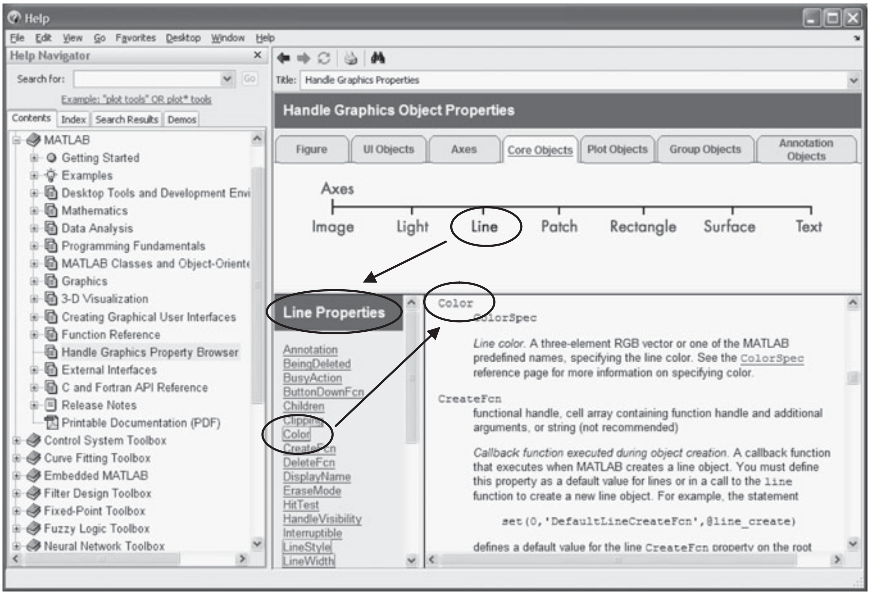
Callback rou

control how

callbacks. If there is a callback executing, callback invoked subsequently always attempt to interrupt it. If the `Interruptible` property of the object whose callback is executing is set to `on` (the default), then interruption occurs at the next point where the event queue is processed.

Clicking on *Core Objects* brings up the window shown in (c)

(b)



```
Color
      ColorSpec
```

Line color. A three-element RGB vector or one of the MATLAB predefined names, specifying the line color. See the `ColorSpec` reference page for more information on specifying color.

```
CreateFcn
```

functional handle, cell array containing function handle and additional arguments, or string (not recommended)

Callback function executed during object creation. A callback function that executes when MATLAB creates a line object. You must define this property as a default value for lines or in a call to the `line` function to create a new line object. For example, the statement

```
set(0,'DefaultLineCreateFcn',@line_create)
```

defines a default value for the line `CreateFcn` property on the root

(c)

Figure 6.2 Continued

Plotting Lines

To draw a straight line that goes from (0, 0) to (1, 2) using the default line type (solid) and the default color (blue), the plotting instruction is

```
plot([0, 1], [0, 2])
```

Notice that the first two-element vector [0, 1] represents the values of the x-coordinates, and the second two-element vector [0, 2] represents the values of the y-coordinates. Thus, the first element of each vector defines the (x, y)-coordinates of one end point of the line and the second elements of these vectors are the coordinates of the other end point.

Consider the set of unconnected straight lines shown in Figure 6.3. To draw this set of n unconnected straight lines whose end points are (x_{1n}, y_{1n}) and (x_{2n}, y_{2n}), we create four vectors

$$x_j = [x_{j1}\, x_{j2}\, \ldots\, x_{jn}] \qquad j = 1, 2$$
$$y_j = [y_{j1}\, y_{j2}\, \ldots\, y_{jn}] \qquad j = 1, 2$$

Then, the `plot` instruction is

```
x1 = [...];  x2 = [...];
y1 = [...];  y2 = [...];
plot([x1; x2], [y1; y2])
```

where [x1; x2] and [y1; y2] are each $(2 \times n)$ arrays.

To illustrate how this expression is used, we shall draw four vertical lines from $y = 0$ to $y = \cos(\pi x/20)$ when $x = 2, 4, 6$, and 8. The script is

```
x = 2:2:8;
y = [zeros(1, length(x)); cos(pi*x/20)];
plot([x; x], y, 'k')
```

where we have used the fact that $x_1 = x_2 = x$. The color is specified so that all the lines have the same color, black in this case. The function `zeros` is used to create

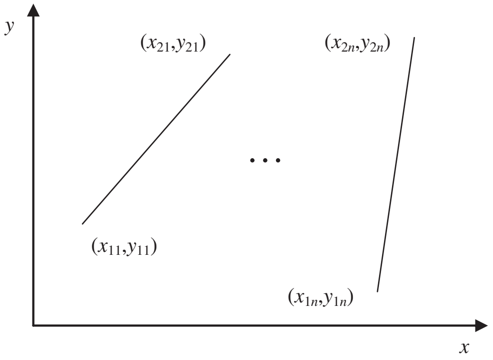
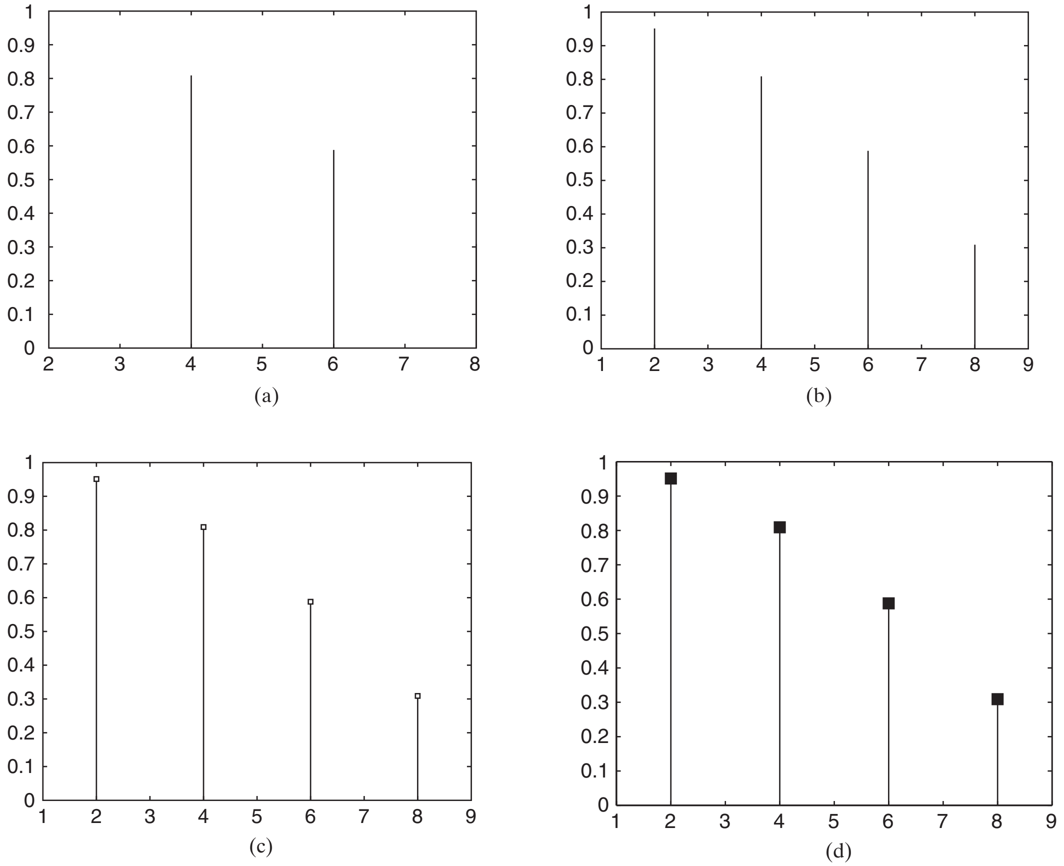


Figure 6.3 Set of unconnected straight lines and the coordinates of their end points.



(a)

(b)

(c)

(d)

Figure 6.4 (a) Situation where the figure box hides lines. (b) Use of `axis` to broaden the axis limits so that all lines can be seen. (c) Placement of red squares at uppermost ends of lines. (d) Placement of solid red squares with blue edges and increased size at uppermost ends of lines.

a vector of 0's of the same length as *x*. The result is shown in Figure 6.4a. Unfortunately, because of the automatic scaling of the axes, the first and last lines are coincident with the box around the figure. Therefore, one has to adjust the *x*-axis limits so that these lines are visible. Referring to Figure 6.5, this adjustment is done with either

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

where x_{min}, x_{max}, y_{min}, and y_{max} are the minimum and maximum values of the *x*- and *y*-axes, respectively, or with

`xlim`([xmin, xmax])

for the *x*-axis only and with

`ylim`([ymin, ymax])

for the *y*-axis only. Thus, the revised script is

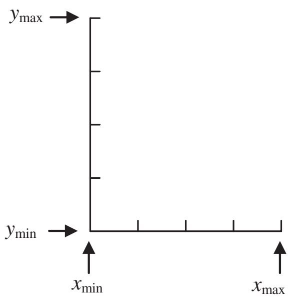


Figure 6.5 Identification of the arguments of `axis`, `xlim`, and `ylim`.

```
x = 2:2:8;
y = [zeros(1, length(x)); cos(pi*x/20)];
plot([x; x], y, 'k')
axis([1, 9, 0, 1])    % or xlim([1, 9])
```

The revised graph is shown in Figure 6.4b.

Obtaining the values of the axis limits and then redefining one or more of them can provide additional flexibility. The limits are obtained from

```
v = axis;
```

in which v is a four-element vector where $v(1) = x_{\text{min}}$, $v(2) = x_{\text{max}}$, $v(3) = y_{\text{min}}$, and $v(4) = y_{\text{max}}$. Thus, to obtain Figure 6.4b, the script could have been written as

```
x = 2:2:8;
y = [zeros(1, length(x)); cos(pi*x/20)];
plot([x; x], y, 'k')
v = axis;
v(1) = 1;   v(2) = 9;
axis(v)
```

If the figure is to be enhanced by placing a square with red edges at the end of each vertical line, then we have to add another triplet of instructions in `plot` as follows:[3]

```
x = 2:2:8;
y = [zeros(1, length(x)); cos(pi*x/20)];
plot([x; x], y, 'k', x, cos(pi*x/20), 'rs')
axis([1, 9, 0, 1])
```

The result is shown in Figure 6.4c.

[3] This `plot` expression is, in some respects, a generalization of the plotting function `stem`, which assumes that $x_1 = x_2$ and that $y_1 = 0$.

One can also change the characteristics of the square shown in Figure 6.4c. For example, to change the square with red edges to a solid red square with blue edges that is larger than the default size, we use the procedure indicated in Figure 6.2 to find that the governing keywords are *MarkerEdgeColor*, *MarkerFaceColor*, and *MarkerSize*. Hence, the script becomes

```
x = 2:2:8;
y = [zeros(1, length(x)); cos(pi*x/20)];
plot([x; x], y, 'k')
hold on
plot(x, cos(pi*x/20), 's', 'MarkerEdgeColor', 'b', 'MarkerFaceColor', 'r',
   'MarkerSize', 14)
axis([1, 9, 0, 1])
```

which upon execution results in Figure 6.4d.

Plotting Circles

To draw a circle of radius *r* whose center is located at (a, b) in a Cartesian coordinate system, one has to first transform the radial coordinates to Cartesian coordinates using (recall Figure 2.2)

$$x = a + r\cos(\theta)$$
$$y = b + r\sin(\theta)$$

where $0 \le \theta \le \theta_1 \le 2\pi$. When $\theta_1 < 2\pi$, an arc of a circle is drawn. If it is assumed that $\theta_1 = 2\pi$, $a = 1$, $b = 2$, and $r = 0.5$, then the script to draw the circle is

```
theta = linspace(0, 2*pi);
plot(1+0.5*cos(theta), 2+0.5*sin(theta))
axis equal
```

The `axis equal` function proportions the graph so that the circles appear as circles, rather than as ellipses. The execution of this script is shown in Figure 6.6.

The script to draw a family of six concentric circles whose initial radius of 0.5 increases in increments of 0.25 and whose centers are indicated by a plus sign is

```
theta = linspace(0, 2*pi, 50);        % (1×50)
rad = 0.5:0.25:1.75;                  % (1×6)
x = 1+cos(theta') *rad;               % (50×6)
y = 2+sin(theta') *rad;               % (50×6)
plot(x, y, 'k', 1, 2, 'k+')
axis equal
```

The values in the arrays are plotted column by column. Since all fifty values of *theta* are to be drawn at each value of *rad*, we formed them as (50×6) arrays. If the string ‘k’ were omitted, then each circle would have been drawn in a different color. The execution of this script yields Figure 6.7.

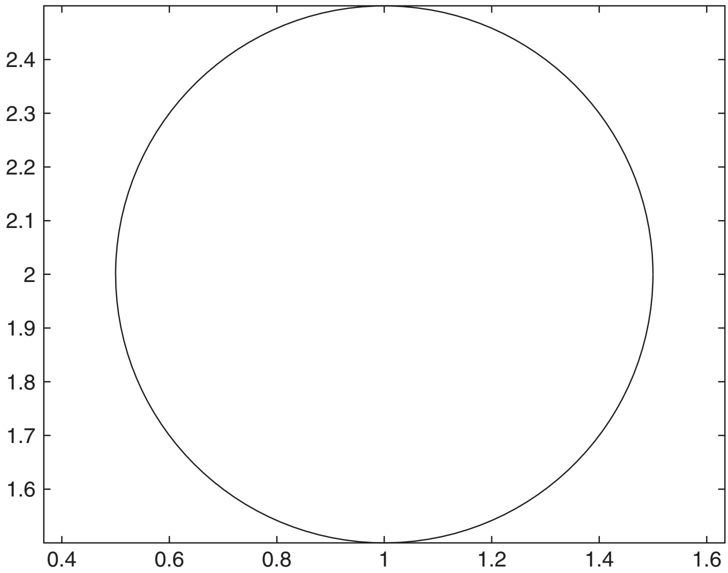


Figure 6.6 Circle drawn with `axis equal`.

Plotting Family of Curves

One way to draw a family of curves was presented for concentric circles. In general, MATLAB allows one to have the *x*-axis represented by a vector and the *y*-axis by a matrix. It will draw the curves by drawing the vector versus either the columns or

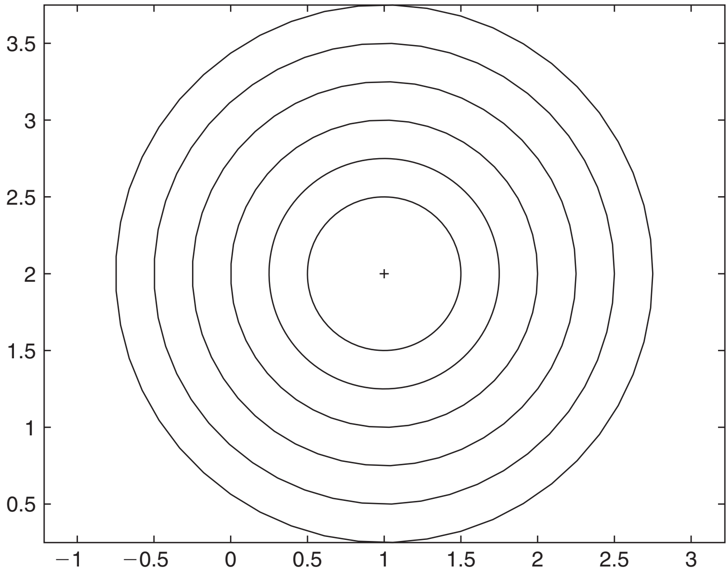


Figure 6.7 Six concentric circles.

the rows of the matrix, depending on which one matches the length of the vector. We illustrate this method with two examples. In the first example, we draw a family of parabolas given by

$$y = a^2 - x^2$$

for $-5 \le x \le 5$ and $a = 1, 2, \ldots, 5$. The script is

```
x = -5:0.2:5;                          % (1×51)
a = 1:5;                               % (1×5)
[xx, aa] = meshgrid(x.^2, a.^2);       % (5×51)
plot(x, aa-xx, 'k')
```

Upon execution, we obtain the results shown in Figure 6.8.

Now consider the visualization of the convergence of the series

$$S_N = \sum_{j=1}^{N} \frac{1}{(a + j)^2}$$

for $N = 1, 2, \ldots, 10$ and $a = 1, 2,$ and 3. In this case, we use cumsum to obtain the following script:

```
aa = 1:3;                      % (1×3)
N = 1:10;                      % (1×10)
[a, k] = meshgrid(aa, N);      % (10×3)
S = cumsum(1./(a+k).^2);       % (10×3)
plot(N, S, 'ks-')
```

which when executed results in Figure 6.9.

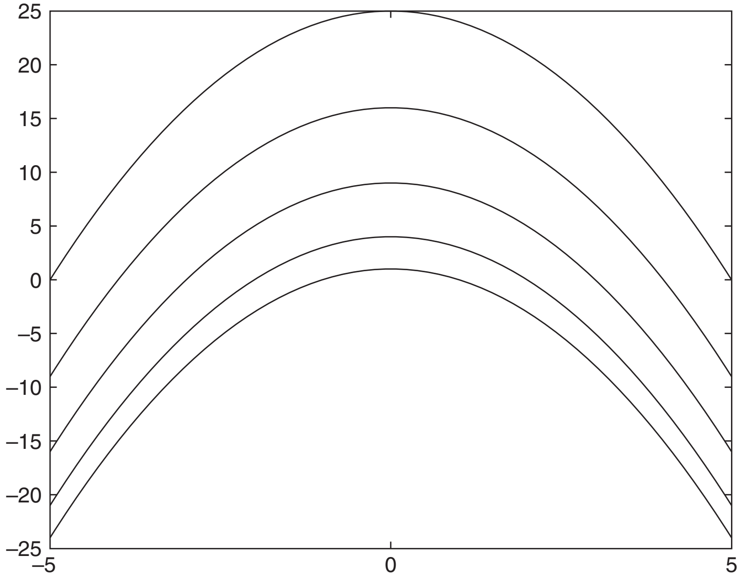


Figure 6.8 Family of parabolas.

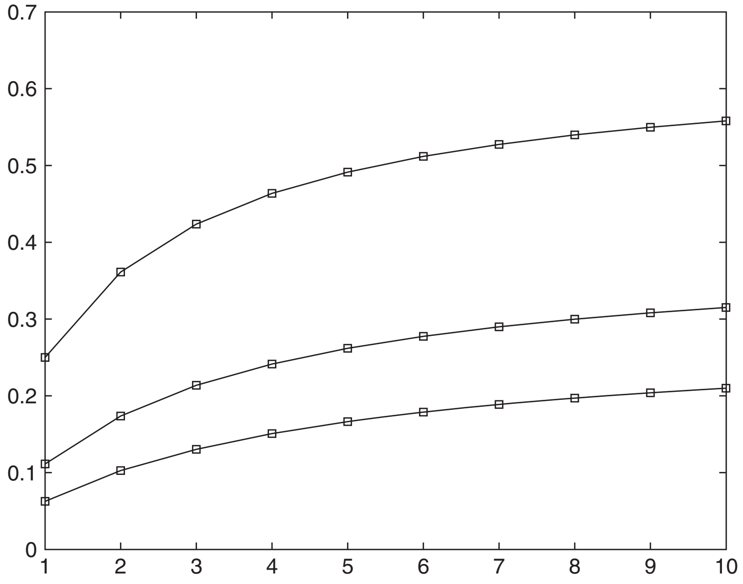


Figure 6.9 Visualization of the convergence of a series.

Plotting Multiple Mathematical Functions on One Figure[4]

Consider the three mathematical relations:

$$\begin{aligned} g_1(x) &= 0.1x^2 \\ g_2(y) &= \cos^2 y \\ g_3(z) &= e^{-0.3z} \end{aligned}$$

where $0 \le x = y = z \le 3.5$. These three relations can be drawn on one figure in either of three ways:

```
x = linspace(0, 3.5);
plot(x, [0.1*x.^2; cos(x).^2; exp(-0.3*x)], 'k')
```

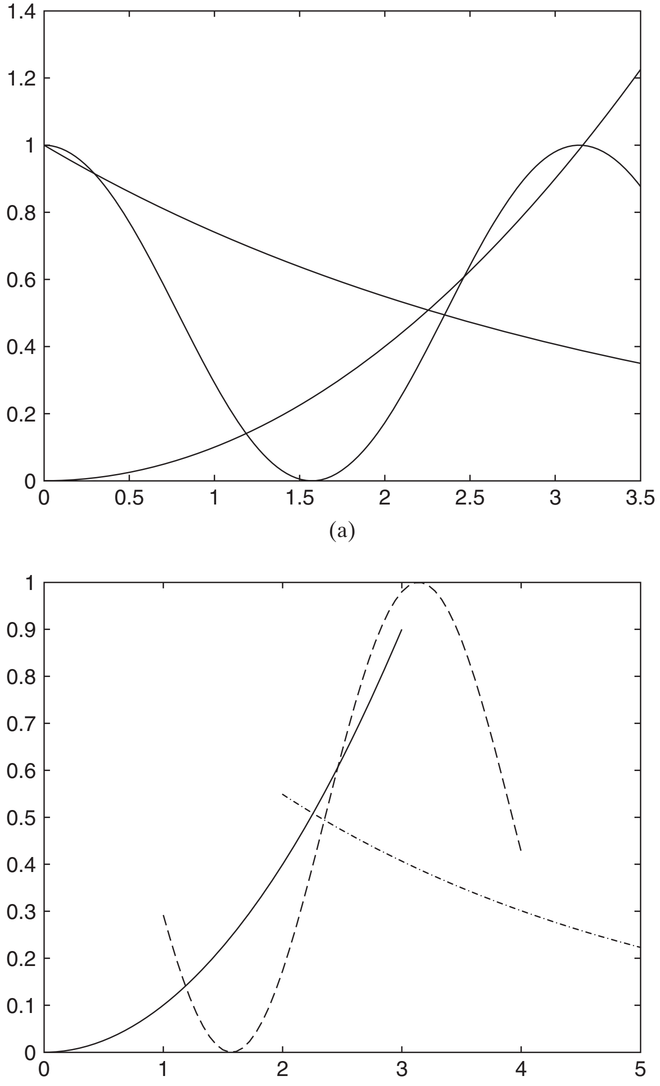
or

```
x = linspace(0, 3.5);
plot(x, 0.1*x.^2, 'k', x, cos(x).^2, 'k', x, exp(-0.3*x), 'k')
```

or

```
x = linspace(0, 3.5);
plot(x, 0.1*x.^2, 'k')
```

[4] To plot two different types of graphs with two different ordinates use plotyy. See Section 6.2.3 and Figure 6.11.



(a)

(b)

Figure 6.10 (a) Three different functions plotted over the same range. (b) Three different functions plotted over three different ranges.

```
hold on
plot(x, cos(x).^2, 'k')
plot(x, exp(-0.3*x), 'k')
```

Execution of any of these three scripts will produce Figure 6.10a, where all the curves have the same color: black.

On the other hand, if the range of the independent variable for each of these functions is different, then only the second and third scripts can be used. For example, if $0 \le x \le 3, 1 \le y \le 4$, and $2 \le z \le 5$ and the form of the second script above is used, we have

```
x = linspace(0, 3, 45);
y = linspace(1, 4, 55);
z = linspace(2, 5, 65);
plot(x, 0.1*x.^2, 'k-', y, cos(y).^2, 'k--', z, exp(-0.3*z), 'k-.')
```

which upon execution results in Figure 6.10b. Notice that we have plotted each function with a different line type and that each curve is plotted with a different number of points.

6.2.2 Changing a Graph's Overall Appearance

Several functions can be used to change the basic appearance of a graph. They are:

```
axis on or axis off     [default – on]
box on or box off       [default – on]
grid on or grid off     [default – off]
```

The function `box on` only works when `axis on` has been selected.

We shall illustrate the effects that these functions have on the graph's appearance by plotting a Lissajous figure, which is a graph of $\sin(n\theta)$ versus $\sin(m\theta + \theta_1)$, where m and n are positive numbers, $0 \le \theta \le 2\pi$, and $0 \le \theta_1 < 2\pi$. Let us consider the case where $n = 1, m = 2$, and $\theta_1 = \pi/4$ (45°). If we take 101 equally spaced values of θ, then the script is

```
th = linspace(0, 2*pi, 101);
plot(sin(th), sin(2*th+pi/4))
```

Execution of this script and its modifications with the `box`, `axis`, and `grid` are summarized in Table 6.3.

6.2.3 Special Purpose Graphs

MATLAB has a library of special purpose graphs that are applicable to a wide variety of applications. We shall illustrate several of them by having each of them plot one or both parts of the following expression:

$$F(\Omega) = H(\Omega)e^{j\theta(\Omega)} \qquad \Omega \ge 0$$

where

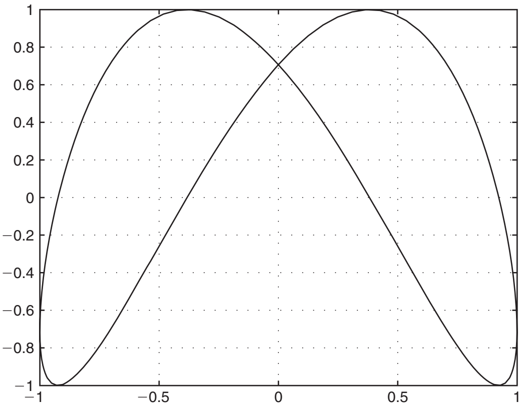
$$H(\Omega) = \frac{1}{\sqrt{\left(1 - \Omega^2\right)^2 + \left(2\zeta\Omega\right)^2}}$$

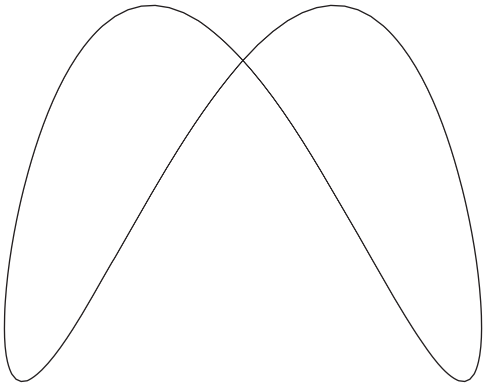
$$\theta(\Omega) = \tan^{-1}\frac{2\zeta\Omega}{1 - \Omega^2}$$

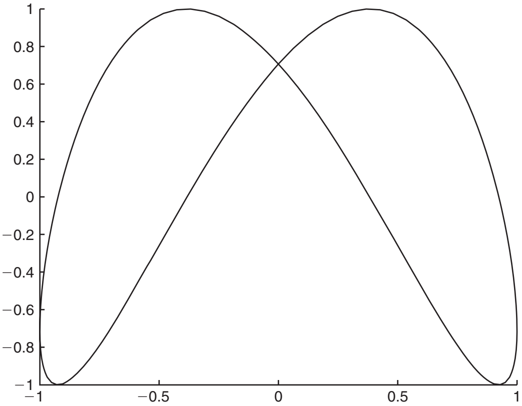
and $\zeta < 1$.

TABLE 6.3 Illustration of box, grid, and axis

Function	Script	Graph

box on
grid on
	th = linspace(0, 2*pi, 101);
	x = sin(th);
	y = sin(2*th+pi/4);
	plot(x, y, 'k-')
	box on
	grid on
		

box off
grid off
axis off
	th = linspace(0, 2*pi, 101);
	x = sin(th);
	y = sin(2*th+pi/4);
	plot(x, y, 'k-')
	box off
	grid off
	axis off
		

box off
grid off
axis on
	th = linspace(0, 2*pi, 101);
	x = sin(th);
	y = sin(2*th+pi/4);
	plot(x, y, 'k-')
	box off
	grid off
		

We first create the function M file **FOm** to represent this expression as

```
function [H,T] = FOm(Om, z)
T = atan2(2*z*Om, 1-Om.^2)*180/pi;
H = 1./sqrt((1-Om.^2).^2+(2*z*Om).^2);
```

where $T = \theta(\Omega)$ is expressed in degrees, $z = \zeta$, and we have used the two argument form of the arctangent function because of the sign change in the denominator.

semilogx, semilogy, and loglog

The first set of special purpose graphs that we consider are semilogx, semilogy, and loglog. The function semilogx plots the *x*-axis on a log to the base 10 scale, the function semilogy plots the *y*-axis on a log to the base 10 scale, and loglog plots both axes on the log to the base 10 scale. These three plotting functions are summarized in Table 6.4.

stairs, stem, and bar

Now consider the set of plotting functions stairs, stem, and bar. The plotting function stairs plots a staircase-like representation of the data points; stem plots the data as discrete values connected by straight lines from the *x*-axis; bar plots the data points connected by filled rectangles (bars) from the *x*-axis. These three plotting functions are summarized in Table 6.5. The third argument of bar specifies the width; that is, 0.6 indicates that of the total width allocated for this bar, only 60% of it has been used. This has the effect of increasing the white space between the bars, which can improve the readability of the figure. The default value is 0.8.

plotyy

To create a graph that consists of a plot of two different functions each with two different ranges of *x* and *y* values, one uses

```
plotyy(x1, y1, x2, y2, 'function_1', 'function_2')
```

where 'function_1' and 'function_2' each can be plot, semilogx, semilogy, loglog, or stem. This plot function is equivalent to function_1(x1, y1) and function_2(x2, y2), where function_n is any one of the five plotting functions mentioned previously.

To illustrate the use of plotyy, we will plot $H(\Omega)$ and $\theta(\Omega)$ on the same graph. The script is

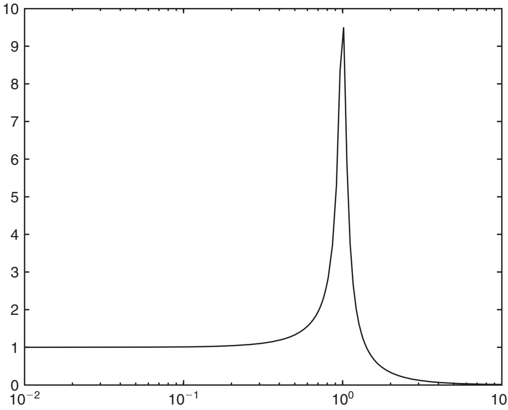
```
Om = logspace(-1, 1, 200);
[H,T] = FOm(Om, 0.05);
plotyy(Om, H, Om, T, 'loglog', 'semilogx')
```

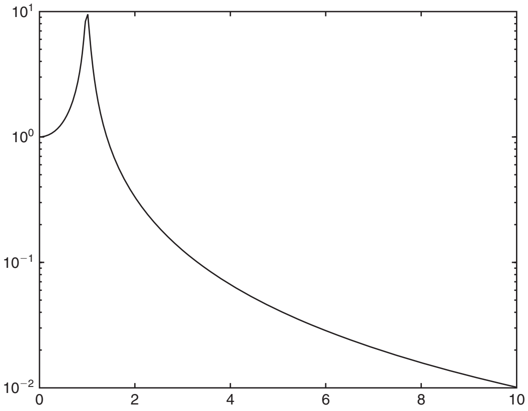
which upon execution produces Figure 6.11. The ordinate and curve corresponding to that ordinate appear in the same color. In Figure 6.11, the left-hand ordinate and the curve representing $H(\Omega)$ are given in blue and the right-hand ordinate and the curve representing $\theta(\Omega)$ are given in green.[5]

[5] For an example of how to add axis labels to a plotyy-generated graph and how to change line characteristics, see Example 10.10 and Figure 10.22.

TABLE 6.4 Illustration of `semilogx`, `semilogy`, and `loglog`

Plotting function	Script	Graph

`semilogx`	Om = `linspace`(0.01, 10, 200);
	[H, T] = **FOm**(Om, 0.05);
	`semilogx`(Om, H)
		

`semilogy`	Om = `linspace`(0.01, 10, 200);
	[H, T] = **FOm**(Om, 0.05);
	`semilogy`(Om, H)
		

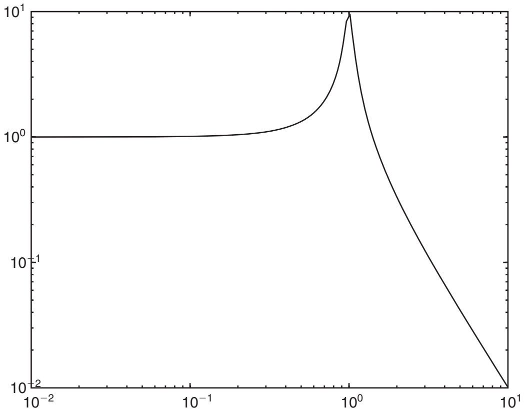
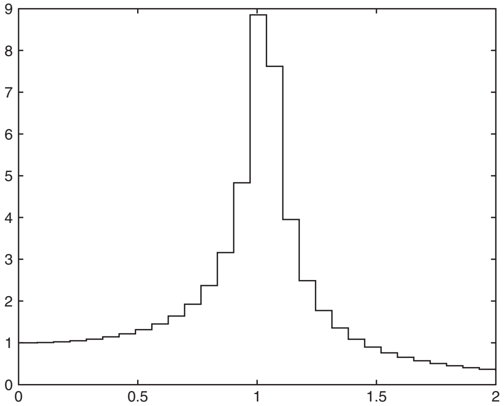
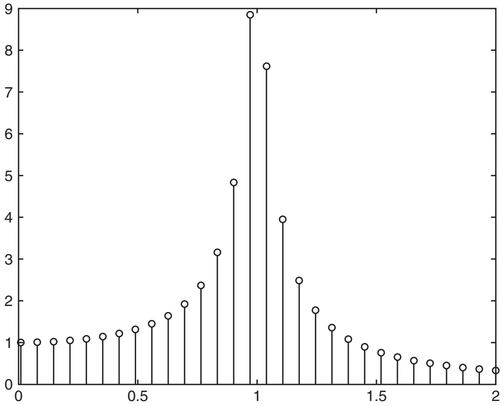
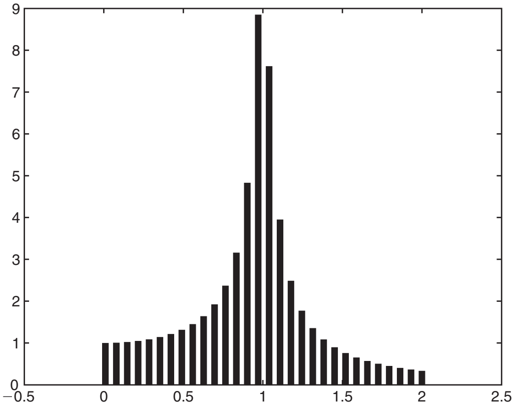
`loglog`	Om = `linspace`(0.01, 10, 200);
	[H, T] = **FOm**(Om, 0.05);
	`loglog`(Om, H)
		

TABLE 6.5 Illustration of `stairs`, `stem`, and `bar`

Plotting function	Script	Graph

`stairs`	Om = `linspace`(0.01, 2, 30);
	[H, T] = **FOm**(Om, 0.05);
	`stairs`(Om, H)
		

`stem`	Om = `linspace` (0.01, 2, 30);
	[H, T] = **FOm**(Om, 0.05);
	`stem`(Om, H)
		

`bar`	Om = `linspace` (0.01, 2, 30);
	[H, T] = **FOm**(Om, 0.05);
	`bar`(Om, H, 0.6)
		

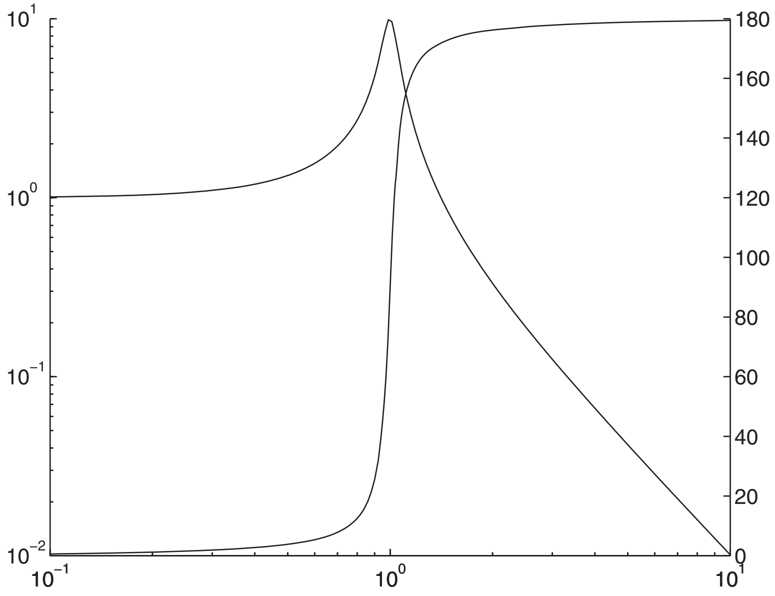


Figure 6.11 Plot of $H(\Omega)$ and $\theta(\Omega)$ using `plotyy`.

`convhull`, `delauney`, and `voronoi`

There is another group of plotting functions that has the capability of displaying clusters of points in different ways. These plotting functions are `convhull`, `delauney`, and `voronoi`. The function `convhull` plots the convex hull of a set of points. The convex hull is the smallest polygon that encloses a set of points in a plane, provided that all the points are not collinear. The function `delauney` uses a procedure on a set of points in a plane to create a set of triangles such that no points are contained in any triangle's circumscribed circle. A circumscribed circle is a circle on whose perimeter the three vertices of a triangle lie. The output of `delauney` is plotted using `triplot`, a special plotting function used to plot triangles in a plane. For each point P in a set of points in a plane, the function `voronoi` draws a convex polygon around each P such that each line segment of the polygon that separates P from each of its closest neighboring points is the perpendicular bisector between P and those nearest neighbors.

To illustrate these three functions, we create the following function M file that contains the following set of (x, y)-coordinate pairs:

```
function [x, y] = PointSet
x = [1, 3, 5, 2, 4, 6, 7, 9, 10, 8, 11];
y = [ 4, 6, 7, 1, 2, 10, 8, 3, 11, 5, 9];
```

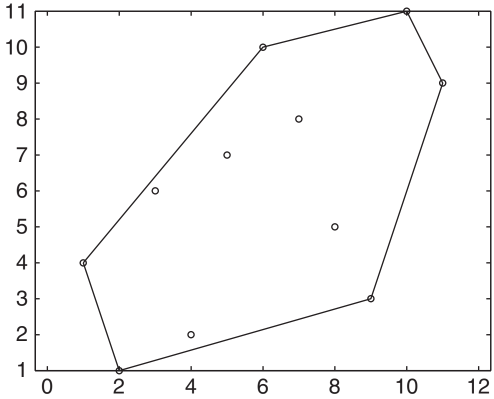
Using **PointSet**, we summarize the usage of `convhull`, `delauney`, and `voronoi` in Table 6.6.

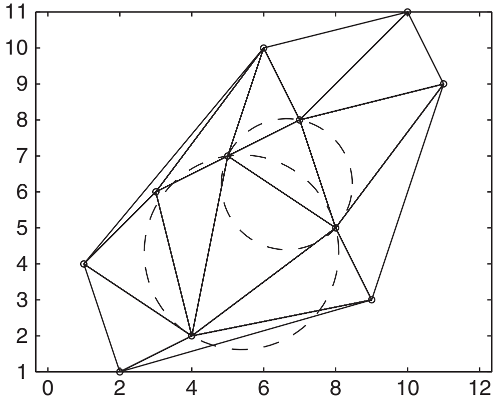
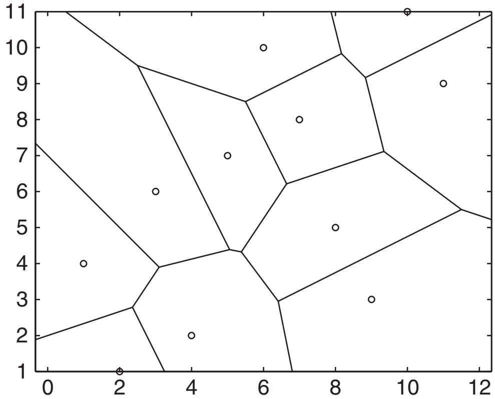
`pie` and `pie3`

For our last set of special purpose plotting functions, we consider pie charts, which are created by using `pie` and `pie3`. For `pie`, we have

TABLE 6.6 Illustration of convhull, delauney, and voronoi

Plotting function	Script	Graph

convhull	[x, y] = **PointSet**;
	n = convhull(x, y);
	plot(x(n), y(n), 'k-', x, y, 'ok')
	axis equal
		

delauney	[x, y] = **PointSet**; tr = delaunay(x, y); triplot(tr, x, y, 'k')[§] hold on plot(x, y, 'ok') axis equal	
voronoi	[x, y] = **PointSet**; voronoi(x, y, 'ko') axis equal	

[§] The MATLAB expressions for drawing the two representative circumscribed circles have been omitted. See Weisstein, *CRC Concise Encyclopedia,* p. 442.

```
pie(d, expl, label)
```

and for `pie3`, we have

```
pie3(d, expl, label)
```

where *d* is a vector of length *n* from which the pie chart will be constructed, *expl* is an optional vector of length *n* consisting of 1s and 0s to indicate which pie sectors are to be "exploded" (separated from the pie), and *label* is an optional cell of length *n* that gives the labels for each pie sector. Some of the ways that `pie` and `pie3` can be used are summarized in Table 6.7. The colors of the pie sectors can be changed by using `colormap`(c), which is discussed in Section 7.2. The argument *c* is one of thirteen keywords that change the spectrum of colors that are used by `pie` and `pie3`.

6.2.4 Reading, Displaying, and Manipulating Digital Images

MATLAB provides the capability of reading fourteen different digital image formats, some of the more common being jpeg (joint photographic experts group), bmp (Windows bit map), tiff (tagged image file format), and gif (graphics interchange format). The digital images are read with

```
A = imread('FileName', 'fmt')
```

where *FileName* is the name of the file containing the digital image in the format specified by *fmt*. If the file does not reside in the current directory, then its complete path name must be given. The array *A* is an $(N \times M \times 3)$ array where $(n \times m)$ is the location of a pixel within this array and $A(n, m, 1)$, $A(n, m, 2)$, and $A(n, m, 3)$ are components of the red-green-blue (RGB) triplet for each pixel. The values of each of these three components of the triplet vary from 0 to 255. For example, the color yellow is expressed as $A(n, m, 1) = 255$, $A(n, m, 2) = 255$, and $A(n, m, 3) = 0$.

To display the image, we use

```
A = imread('FileName', 'fmt')
image(A)
```

We shall illustrate the use of these two functions with the following script. The digital image is in the file *WindTunnel.jpg* in the jpeg format and it resides in the current directory. The script that reads and displays this image is

```
A = imread('WindTunnel.jpg', 'jpeg');
image(A)
axis image off
```

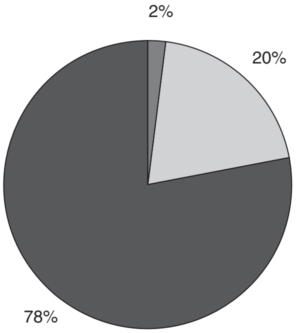
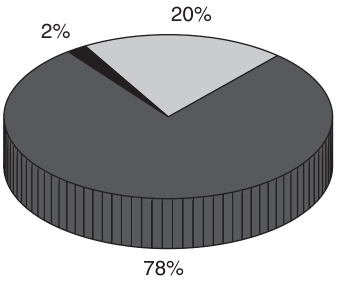
where `axis image` makes the aspect ratio of the axes the same as that of the image. The size of *A* is (2848 × 2732 × 3). The results of the execution of this script are shown in Figure 6.12a.

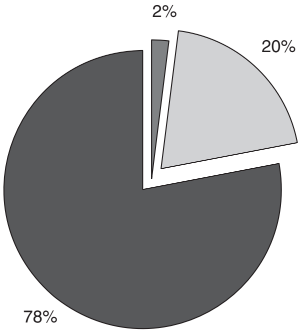
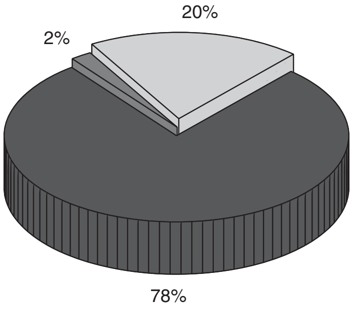
To illustrate how the digital image may be manipulated, we shall arbitrarily change a small number of pixels to yellow. The above script becomes

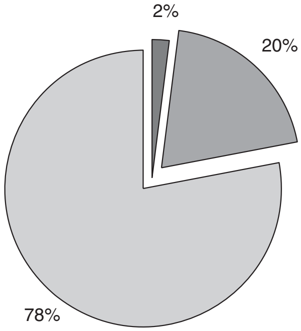
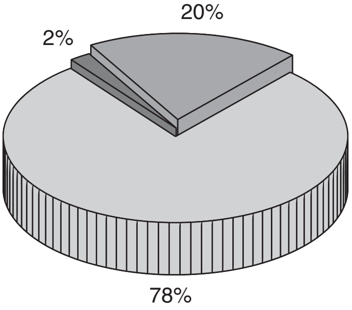
```
A = imread('WindTunnel.jpg', 'jpeg');
A(1800:2800, 1150:1500, 1) = 255;
A(1800:2800, 1150:1500, 2) = 255;
```

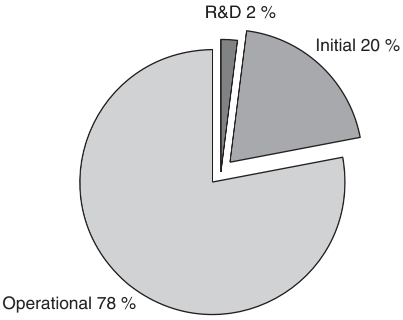
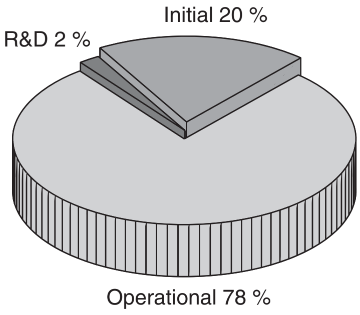
TABLE 6.7 Illustration of pie and pie3

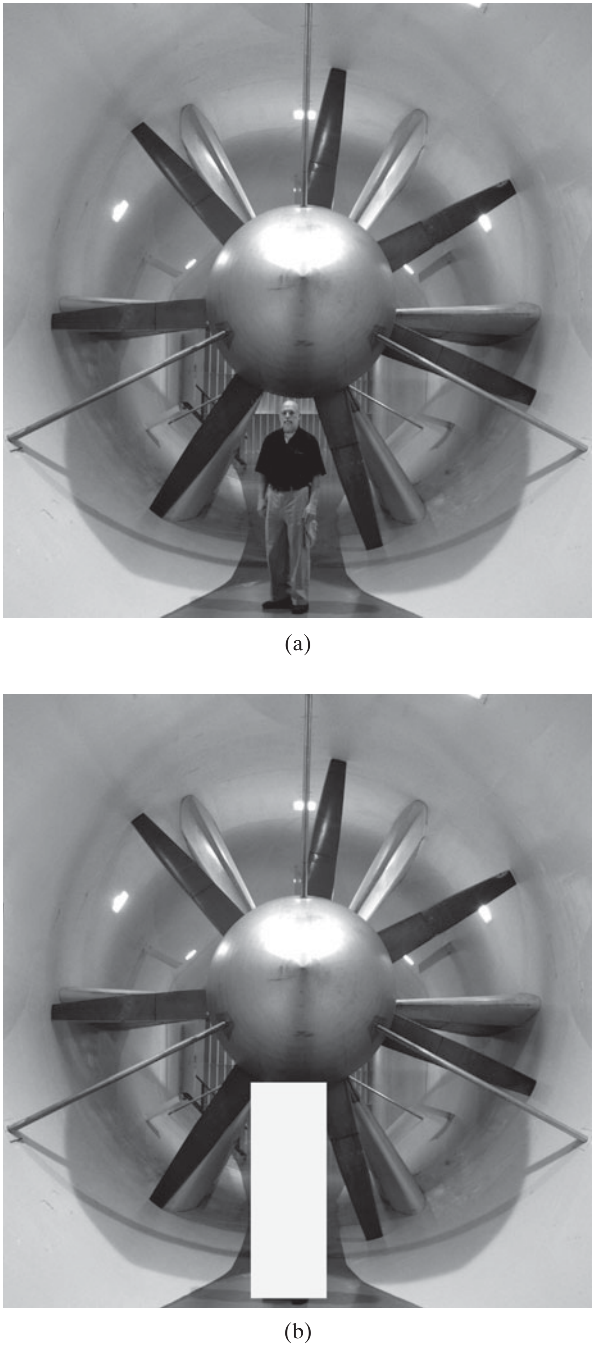
Script	Graph – pie	Graph – pie3

dat = [39, 10, 1];
pie(dat)
% or pie3(. . .)
		

dat = [39, 10, 1];
pie(dat, [1, 1, 0])
% or pie3(. . .)
		

dat = [39, 10, 1];
pie(dat, [1, 1, 0])
% or pie3(. . .)
colormap('cool')
		

dat = [39, 10, 1];
dat = 100*dat/sum(dat);
A = ['Operational '
num2str(dat(1)) ' %'];
B = ['Initial '
num2str(dat(2)) ' %'];
C = ['R&D '
num2str(dat(3)) ' %'];
colormap('cool')
pie(dat, [1, 1, 0], {A, B, C})
% or pie3(. . .)
		



(a)

(b)

Figure 6.12 (a) Display of a digital image. (b) Digital image altered with a small yellow region. (Image courtesy of Dr. J. B. Barlow, Director Glenn L. Martin Wind Tunnel, University of Maryland, College Park, Maryland.)

```
A(1800:2800, 1150:1500, 3) = 0;
image(A)
axis image off
```

The results of the execution of this script are shown in Figure 6.12b.

6.3 GRAPH ANNOTATION AND ENHANCEMENT

6.3.1 Introduction

MATLAB has extensive graphic enhancement capabilities. In this section, we shall illustrate through examples how to enhance a graph—

- With axis labels, figure titles, labeled curves, legends, filled areas, and placement of text
- By altering the attributes of the axes, curve lines, and text
- By using Greek letters, mathematical symbols, and subscripts and superscripts
- By positioning one figure inside another figure
- By using the interactive plotting tools
- By using animation

6.3.2 Axes and Curve Labels, Figure Titles, Legends, and Text Placement

The functions that are used to label the *x* and *y* axes and to place a title above the graph are, respectively,

```
xlabel(s1)
ylabel(s2)
title(s3)
```

where *s*1, *s*2, and *s*3 are strings. The function that places text anywhere in the figure window is

```
text(x, y, s4)
```

where *x* and *y* are the coordinates of where the text given by the string *s*4 will be placed.

Let us draw, label, title, and annotate the relationship of two intersecting curves, cos(x) and 1/cosh(x), over the range $0 \le x \le 6$. In this range, these two curves intersect at $x = 4.73$. Recall Example 5.8. We shall also draw a vertical line through the intersecting point and denote the value of *x* near this intersection. The script to perform these operations is

```
x = linspace(0, 6, 100);
plot(x, cos (x), 'k', x, 1./cosh (x), 'k', [4.73, 4.73], [-1, 1], 'k')
xlabel ('x')
ylabel ('Value of functions')
title ('Visualization of two intersecting curves')
text (4.8, -0.1, 'x = 4.73')
text (2.1, 0.3, '1/cosh(x)')
text (1.2, -0.4, 'cos(x)')
```

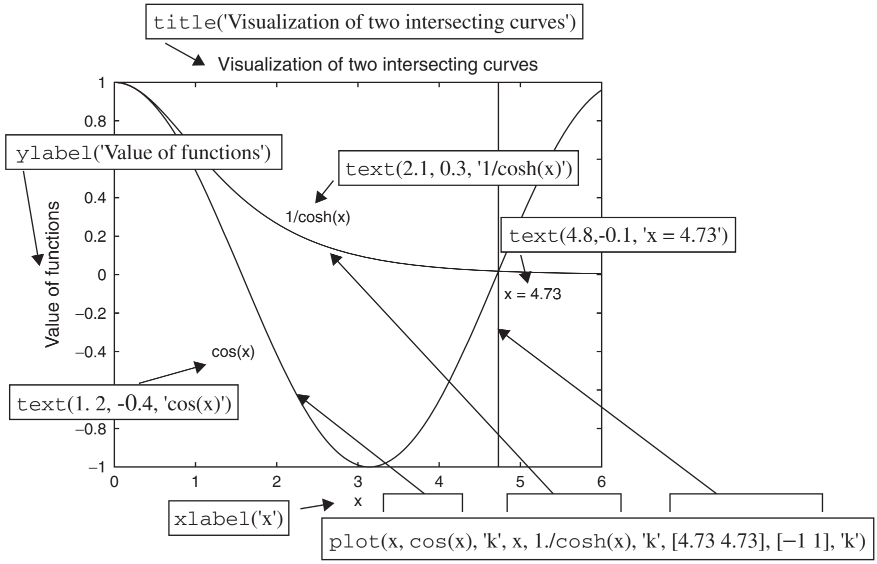


Figure 6.13 MATLAB expressions that create and annotate a figure.

Execution of the script results in Figure 6.13. The coordinate values for the location of the various texts are chosen only after the plot function is executed, that is, only after the first two lines of the script have been written and executed and the resulting figure examined to determine the appropriate coordinates for the text. Then text is added to the script.

There is another way that we can identify the curves in Figure 6.13, and that is with

```
legend (s1, s2, . . . , sn, 'Location', 'p')
```

where *s*1, etc., are the strings containing the alphanumeric identifier that will appear in the legend box and correspond to the curves *in the order that they are drawn*. The specification of the location of where the legend will appear is given by the keyword 'Location' and 'p' is the specified location. When omitted, the legend is placed in the upper right-hand corner of the graph. When the pair of strings 'Location' and 'p' is used, the string 'p' tells legend where in one of eight predetermined locations to place the legend. The string keywords and their effect are shown in Figure 6.14. Finally, the legend function differs from text in that text can be used as many times as practical, whereas legend can only be used once. In addition, legend places all the text within a box irrespective of whether the box edges are visible as discussed below.

We shall illustrate the use of the legend function by revisiting the script that produced Figure 6.13 and replace the two text statements with a legend. We shall place the legend in the lower left-hand corner of the graph, that is, in the southwest corner. Then, the script becomes

```
x = linspace(0, 6, 100);
plot(x, cos(x), 'k-', x, 1./cosh(x), 'k--', [4.73, 4.73], [-1, 1], 'k')
xlabel('x')
```

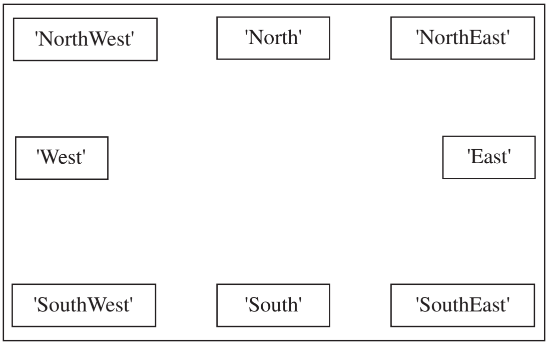


Figure 6.14 Keywords that specify the indicated placement of the legend.

```
ylabel('Value of functions')
title('Visualization of two intersecting curves')
text(4.8, -0.1, 'x = 4.73')
legend('cos(x) ', '1/cosh(x) ', 'Location', 'SouthWest')
```

The execution of this script produces Figure 6.15. Notice that we have employed `plot` to display three curves, but `legend` has only two string identifiers. Therefore, only the first two curves that were plotted are identified in the legend. The third argument of each triplet in the `plot` function specifies that the curves are to be

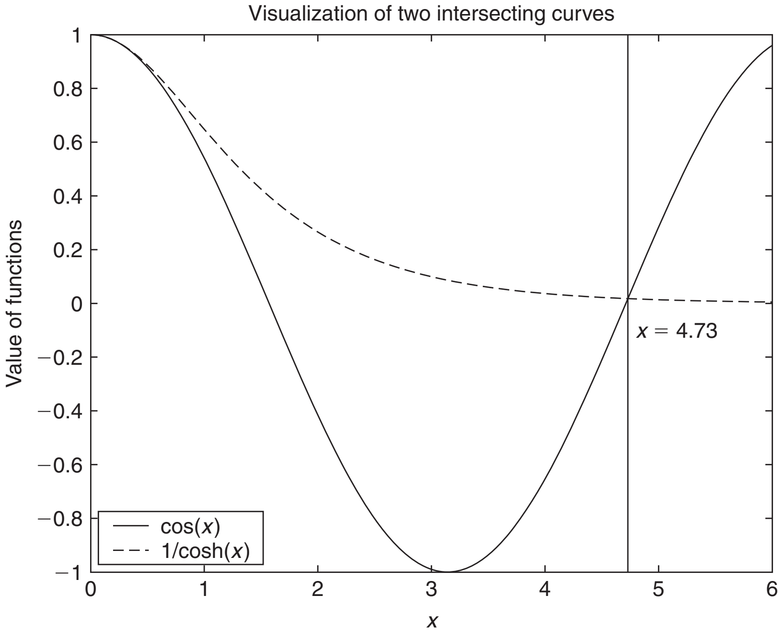


Figure 6.15 Use of `legend`.

drawn in black, with cos(x) appearing as a solid line and 1/cosh(x) as a dashed line. The legend's arguments are order dependent. The first argument of legend corresponds to the first curve drawn and the second argument of legend corresponds to the second curve drawn. If there are several plot functions used, then the order continues with the first argument of the second plot function following the last string identifying the last curve plotted in the previous plot statement. Only one legend function can be used per figure or subplot. The various attributes of the text that appear in the legend can be altered as discussed in Section 6.3.5.

The box around the legend can be toggled on and off with

```
legend('boxon')
```

and

```
legend('boxoff')
```

6.3.3 Filling Regions

A region of a graph can be highlighted by coloring it. We shall illustrate two functions that can be used to perform this highlighting: fill and patch.

To fill the area contained within a polygonal region, we use

```
fill(x, y, c)
```

where x and y are arrays of the same length that represent the end points of the lines that form a polygon. The string c is the color of the fill given by one of the letters appearing in the second column of Table 6.2.

To illustrate the use of fill, we modify the script used in the previous section so that the area between the two curves in the range $0 \leq x \leq 4.73$ is colored cyan. The polygon that has to be created is that formed by the straight line approximation to 1/cosh(x) from $0 \leq x \leq 4.73$ and that formed by the straight line approximation to cos(x) from $4.73 \geq x \geq 0$. Thus, the script becomes

```
x = linspace(0, 6, 100);
plot(x, cos(x), 'k-', x, 1./cosh(x), 'k--', [4.73, 4.73], [-1, 1], 'k')
xlabel('x')
ylabel('Value of functions')
title('Visualization of two intersecting curves')
text(4.8, -0.1, 'x = 4.73')
legend('cos(x) ', '1/cosh(x) ', 3)
xn = linspace(0, 4.73, 50);
hold on
fill([xn, fliplr(xn)], [1./ cosh(xn), fliplr(cos(xn))], 'c')
```

The execution of this script gives Figure 6.16. The connected polygon is created by forming the vector [1./ cosh(xn) fliplr(cos(xn))], which is the concatenation of the top curve 1/cosh(x) and the reversal of the elements of the vector of cos(x), the bottom curve. Corresponding to this new vector is the new x-coordinate vector [xn fliplr(xn)], which is formed by the concatenation of the new values of x and its reverse-ordered values.

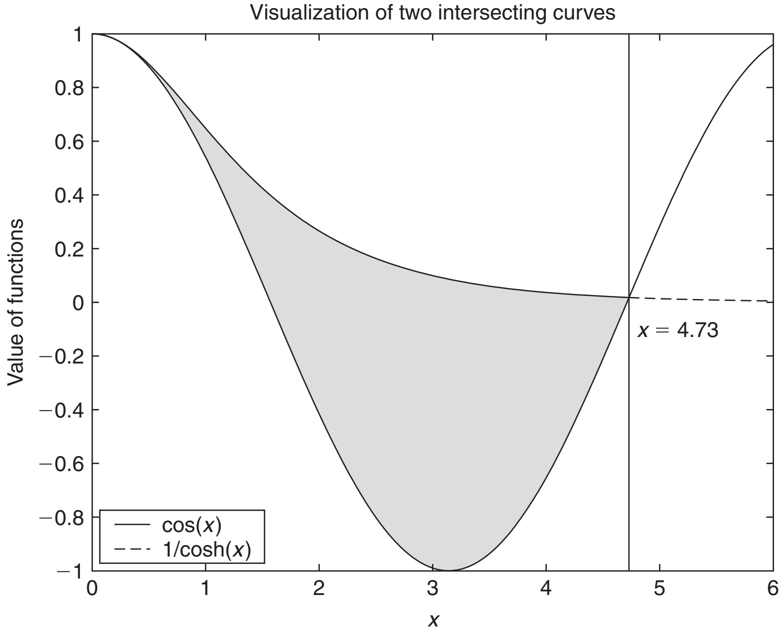


Figure 6.16 Modification of Figure 6.15 with the area between the curves filled.

The use of `fill` to plot two or more overlapping areas becomes dependent on the order of `fill` within the script. In addition, `fill` has a transparency adjustment that permits two filled overlapping areas to show through a specified amount: from 0, which is transparent (invisible) to 1, which is opaque (no transparency). To obtain transparency, two keywords are required: *FaceVertexAlphaData*, which in our case will be a number between 0 and 1 and *FaceAlpha*, which must be set to 'flat' when *FaceVertex AlphaData* is equal to a single number. These different scenarios are illustrated in Table 6.8.

Another way to fill a polygon is with `patch`, which gives a little more control over the attributes of its boundary edges and is given by

```
patch(x, y, c)
```

where its arguments have the same meaning as those for `fill`. This function also can adjust its transparency properties for overlapping patches in the same manner as is done for `fill`. We shall illustrate the use of patch by drawing two squares that are generated by the following function:

```
function [x y] = ptc(a, b, e)
x = a + [0 0 e e 0];
y = b + [0 e e 0 0];
```

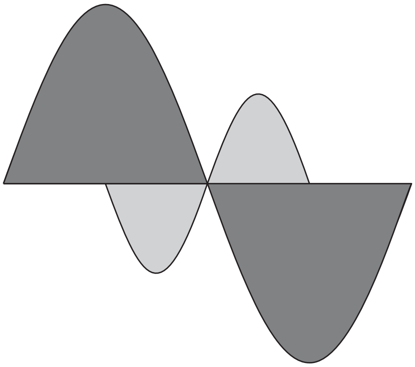
where *a* and *b* are the distances from the origin along the *x* and *y* axes, respectively, and *e* is the length of the edges of the square. The usage of patch is shown in Table 6.9.

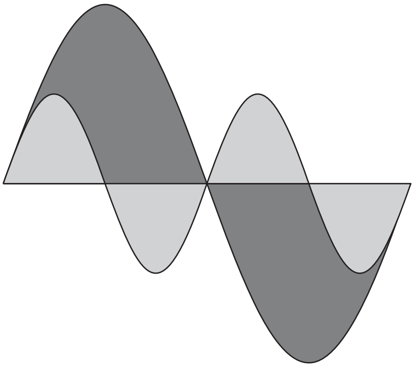
6.3.4 Greek Letters, Mathematical Symbols, Subscripts, and Superscripts

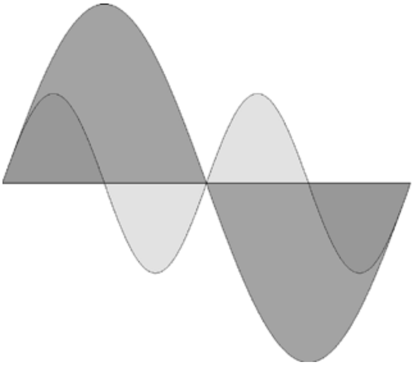
MATLAB provides the capability to annotate a graph with upper- and lower-case Greek letters, subscripts and superscripts, and a range of mathematical symbols. These annotations can be done within `xlabel`, `ylabel`, `text`, `legend`, and

TABLE 6.8 Illustration of `fill` and Its Transparency Capability

Illustration	Script	Graph

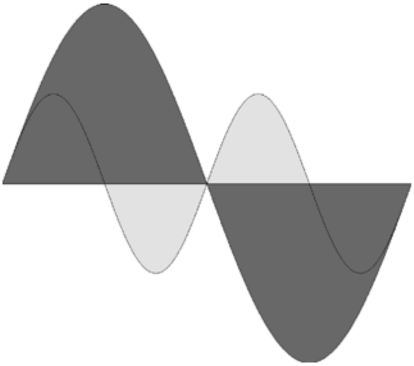
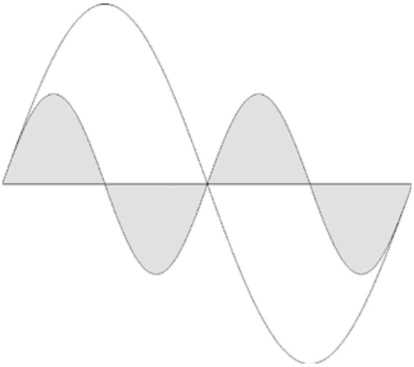
`fill` order 1	t = `linspace`(0, 2*`pi`);
	`fill`(t, 0.5*`sin`(2*t), 'y')
	`hold on`
	`fill`(t, `sin`(t), 'm')
	`axis off`
		

`fill` order 2	t = `linspace`(0, 2*`pi`);
	`fill`(t, `sin`(t), 'm')
	`hold on`
	`fill`(t, 0.5*`sin`(2*t), 'y')
	`axis off`
		

`fill` order 1 plus transparency = 0.6	t = `linspace`(0, 2*`pi`);
	`fill`(t, 0.5*sin(2*t), 'y')
	`hold on`
	`fill`(t, `sin`(t), 'm', . . .
	'FaceVertexAlphaData', . . .
	0.6, 'FaceAlpha', 'Flat')
	`axis off`
		

(*Continued*)

TABLE 6.8 Continued

Illustration	Script	Graph
fill order 1 plus transparency = opaque	t = linspace(0, 2*pi); fill(t, 0.5*sin(2*t), 'y') hold on fill(t, sin(t), 'm', . . . 'FaceVertexAlphaData', . . . 1, 'FaceAlpha', 'Flat') axis off	
fill order 1 plus transparency = invisible	t = linspace(0, 2*pi); fill(t, 0.5*sin(2*t), 'y') hold on fill(t, sin(t), 'm', . . . 'FaceVertexAlphaData', . . . 0.0, 'FaceAlpha', 'Flat') axis off	

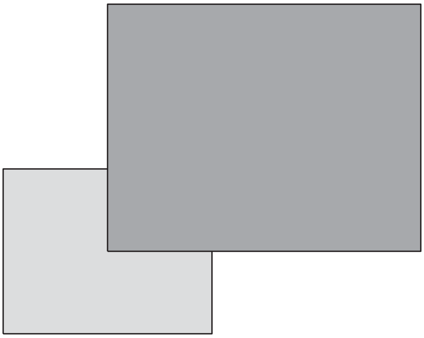
title. The formatting instructions follow the *LaTeX* language.[6] All the following techniques are only valid within a pair of apostrophes. None of these Greek letters, mathematical symbols, subscripts, and superscripts will work when displayed to the command window; that is, if, for example, disp is used.

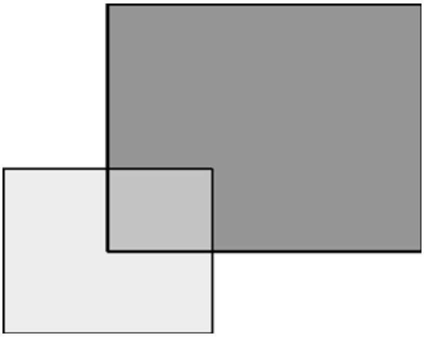
Subscripts are created with the underscore character (_) and superscripts with the exponentiation character (^). The creation of the Greek letters is obtained by the spelling of the letter and preceding the spelling by a backslash (\), as shown in Table 6.10. Upper-case Greek letters are obtained by capitalizing the first letter of the spelling of the Greek letter. However, since many of the upper-case Greek letters are the same as upper-case English letters, only those that are different are given in Table 6.10. The remaining upper-case Greek letters are obtained by using the appropriate upper-case English letters.

[6] See, for example, L. Lamport, *LaTeX: A Document Preparation System*, Addison-Wesley, Reading, MA, 1987.

TABLE 6.9 Illustration of `patch` and Its Transparency Capability

Illustration	Script	Graph

`patch` with no transparency	[x1, y1] = **ptc**(0, 0, 1);
	[x2, y2] = **ptc**(0.5, 0.5, 1.5);
	`patch`(x1, y1, 'y')
	`patch`(x2, y2, 'g')
	`axis off`
		

`patch` with transparency and altered edge attributes	[x1, y1] = **ptc**(0, 0, 1); [x2, y2] = **ptc**(0.5, 0.5, 1.5); `patch`(x1, y1, 'y', 'EdgeColor', 'b', . . . 'LineWidth', 2.5, . . . 'FaceVertexAlphaData', . . . 0.6, 'FaceAlpha', 'Flat') `patch`(x2, y2, 'g', 'EdgeColor', 'r', . . . 'LineWidth', 3.5) `axis off`	

In addition to the symbols in Table 6.10, the alphanumeric characters can be made bold by preceding the alphanumeric characters with

\bf

To make the alphanumeric characters italic, we use

\it

and to return either of these changes to normal, we use

\rm

These character-changing instructions remain in effect until they are changed again.

The creation of the mathematical symbols is obtained by their special spellings preceded by a backslash (\). Some of the more commonly used symbols are also given in Table 6.10. The general syntax is to place a set of concatenated instructions between a pair of apostrophes. When certain groups of symbols are to be kept together, such as an expression that is to appear in an exponent, they are placed between a pair of braces ({}). We shall now illustrate this procedure with an example.

TABLE 6.10 Upper- and Lower-Case Greek Letters and Some Mathematical Symbols

Lower case				Upper case		Mathematical			
Symbol	Syntax	Symbol	Syntax	Symbol	Syntax	Symbol	Syntax	Symbol	Syntax
α	\alpha	ν	\nu	Γ	\Gamma	$\leq$	\leq	$\circ$	\circ
β	\beta	ξ	\xi	Δ	\Delta	$\geq$	\geq	<<	\ll
γ	\gamma	o	o	Θ	\Theta	$\neq$	\neq	>>	\gg
δ	\delta	π	\pi	Λ	\Lambda	$\pm$	\pm	$\prime$	\prime
ϵ	\epsilon	ρ	\rho	Ξ	\xi	$\times$	\times	$\Leftarrow$	\Leftarrow
ζ	\zeta	σ	\sigma	Π	\Pi	∞	\infty	$\angle$	\angle
η	\eta	τ	\tau	Σ	\Sigma	$\sum$	\sum	$\surd$	\surd
θ	\theta	υ	\upsilon	Υ	\Upsilon	$\int$	\int	#	\#
ι	\iota	ϕ	\phi	Φ	\Phi	$\div$	\div	$	\$
κ	\kappa	χ	\chi	Ψ	\Psi	$\sim$	\sim	%	\%
λ	\lambda	ψ	\psi	Ω	\Omega	$\leftarrow$	\leftarrow	&	\&
μ	\mu	ω	\omega			$\uparrow$	\uparrow	{	\{

Let us compute and plot the function

$$g_2 = \cos(4\pi\Omega_1)e^{-(1+\Omega_1^{\beta})}$$

for $\beta = 3$ and $1 \leq \Omega_1 \leq 2$, and label the figure accordingly. The script is

```
Om1 = linspace(1, 2);    beta = 3;
plot(Om1, cos(4*pi*Om1).*exp(-(1+Om1.^beta)), 'k')
title('\itg_{\rm2} \rmversus \Omega_1 for \it\beta \rm= 3')
ylabel('\itg_{\rm2}')
xlabel('\Omega_1')
text(1.2, 0.08, '\itg_{\rm2}\rm=cos(\Omega_1)\ite^{\rm-(1+\Omega_1^ ...
    {\it\beta\rm})}')
```

The execution of this script results in Figure 6.17.

6.3.5 Altering the Attributes of Axes, Curves, Text, and Legends

MATLAB provides the capability to make changes to virtually all characteristics of the elements that comprise a graph. The ones that we shall consider are as follows. For lines, we shall discuss line width and line color. For the text in axis labels, titles, placed text, and legends, we shall discuss font, alignment, size, type, characteristics, and color. For the axes, we shall discuss line width and text attributes.

The default value of the line width for the axes and drawn curves is 0.5 pt, the default values for font size of the axis labels, placed text, and titles is 10 pt, and the default font name for the axis labels and numbering, title, and legend is Helvetica.

The manner in which changes to the attributes of lines and text are made is as follows. For `xlabel`, `ylabel`, `title`, and `text`, we add any number of pairs

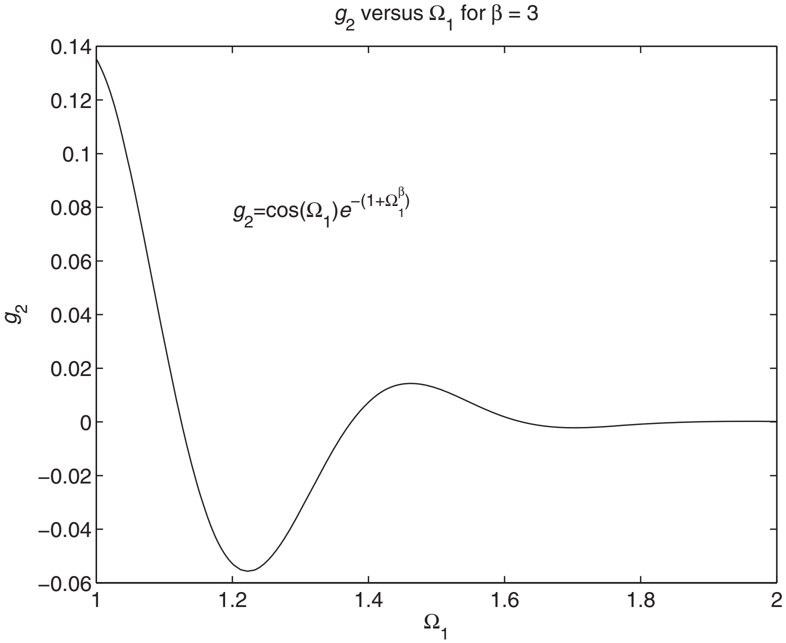


Figure 6.17 Annotation with superscripts, subscripts, and Greek letters.

of keywords and their values to the arguments of these functions as indicated below:

```
xlabel(s1, 'KeyWord', KeyWordValue, ...)
ylabel(s2, 'KeyWord', KeyWordValue, ...)
title(s3, 'KeyWord', KeyWordValue, ...)
text(x, y, s4, 'KeyWord', KeyWordValue, ...)
```

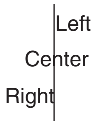
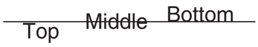
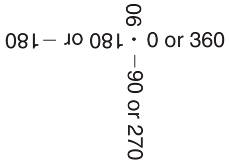
where 'KeyWord' is a string containing the keyword, *KeyWordValue* is the value expressed as a string or a numerical value, depending on the keyword. The keywords and their values for text positioning are given in Table 6.11. The additional keywords and their values that we shall consider for `text` are given in Table 6.12.

To set the attributes for curves created by `plot` and for the text appearing in `legend`, we require the function handles to `legend` and `plot`. To alter the width of the axes and the properties of their labels we use `set` and `gca`, which gets the handle for the current axis. Then `set` is used to set a specific attribute associated with a function handle. To get the function handles for `legend` and `plot`, we use

```
hdl = plot(...);
[a, b] = legend(...);
```

where `plot` and `legend` perform in the manner already discussed and *hdl* is the handle to the curve(s) plotted, and *a* and *b* are the handles to the text in the legend

TABLE 6.11 Keywords and Attributes for Text Positioning

Keyword	Keyword value	Example
'HorizontalAlignment'	'Left' 'Center' 'Right'	
'VerticalAlignment'	'Top' 'Middle' 'Bottom'	
'Rotation'	0 to 360 or −180 to +180	

and the legend box as shown in Table 6.13. Note that we have ended the expressions with a semicolon to suppress displaying to the command window the numerical values of the handles.

The `set` function is

`set`(hdl, 'KeyWord', KeyWordValue, . . .)

where *hdl* is the handle, 'KeyWord' is a string containing the keyword, and *KeyWordValue* is a string or numerical value that corresponds to the keyword. Table 6.14 contains two keywords that are used for altering the characteristics of lines.

We now illustrate how to change the attributes of the legend. We shall make the background color of the legend box yellow and the thickness of the box's edges two points. In addition, the size of the text in the legend will be increased to fourteen points, the text for the solid curve will be blue and that for the dashed curve will be red, and the independent variable *x* will be in italics. Returning to the script that generated Figure 6.15, we modify it to obtain

TABLE 6.12 Additional Keywords and Attributes for `text`

Keyword	Keyword value
'Linewidth'	Number > 0 (default: 0.5)
'FontSize'	Number > 0 (default: 10)
'FontName'	'Courier' 'Helvetica' (default) 'Times' (similar to Times roman)
'Color'	'Letter from 2nd column of Table 6.2'
'FontWeight'	'Normal' (default) 'Bold'

TABLE 6.13 Keywords and Attributes for `legend` Handles

Handle name	Keyword	Keyword value	Attribute affected
a(1)	'LineWidth'	Number > 0 (default: 0.5)	Thickness of legend box edges
a(1)	'Color'	'Letter from 2nd column of Table 6.2'	Background color of legend box
b(1), b(2)	'FontSize'	Number > 0 (default: 10)	Font size of legend text
b(1), b(2)	'FontName'	'Courier' 'Helvetica' (default) 'Times' (Times roman)	Font type of legend text
b(1), b(2)	'Color'	'Letter from 2nd column of Table 6.2'	Color of legend text

```
x = linspace(0, 6, 100);
plot(x, cos(x), 'k-', x, 1./cosh(x), 'k--', [4.73, 4.73], [-1, 1], 'k')
xlabel('x')
ylabel('Amplitude')
title('Visualization of two intersecting curves')
text(4.8, -0.1, 'x = 4.73')
[a, b] = legend('cos(\itx\rm)', '1/cosh(\itx\rm)', 'Location', 'SouthWest');
set(a(1), 'LineWidth', 2, 'Color', 'y')
set(b(1), 'fontsize', 14, 'Color', 'b')
set(b(2), 'fontsize', 14, 'Color',' r')
```

The results of the execution of this script are shown in Figure 6.18.

Again returning to the script that generated Figure 6.15, we shall make the following alterations to the figure:

Title: 14 pt Courier, bold face
x-axis label: 14 pt Times Roman, italic, and bold face
y-axis label: 14 pt Helvetica
Placed text: 12 pt standard mathematical notation
Axes lines: 1.5 pt wide
Axes text: 14 pt Helvetica
Curve for cos(*x*): 4 pt line width
Curve for cosh(*x*): 2.5 pt line width
Vertical line at *x* = 4.73: 0.25 line width, green

TABLE 6.14 Keywords and Attributes for Lines

Keyword	Keyword value
'Linewidth'	Number > 0 (default: 0.5)
'Color'	'Letter from 2nd column of Table 6.2'

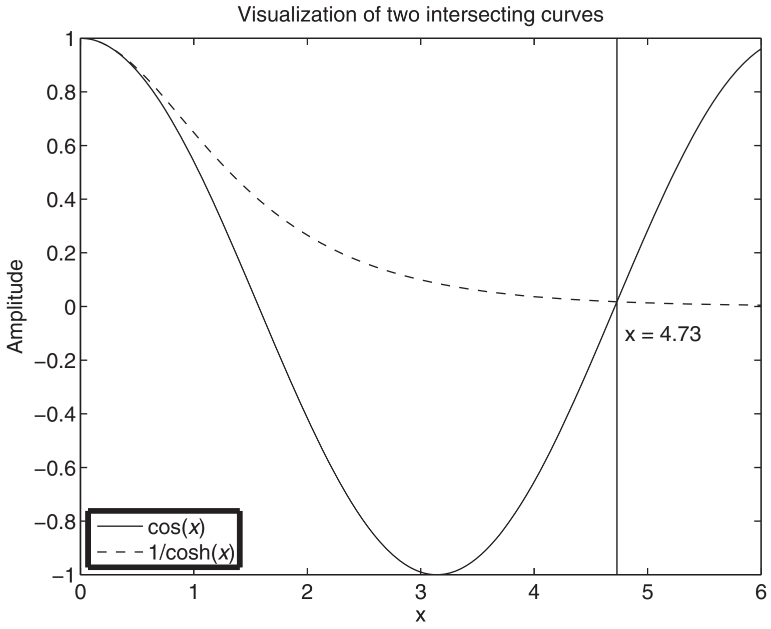


Figure 6.18 Alteration of the attributes of `legend`.

The revised script is

```
x = linspace(0, 6, 100);
hc = plot(x, cos(x), 'k-');
hold on
hch = plot(x, 1./cosh(x), 'k–');
hsl = plot([4.73, 4.73], [-1, 1], 'k');
[a, b] = legend('cos(x) ', '1/cosh(x) ', 'Location', 'SouthWest');
xlabel('\it\bfx', 'FontSize', 14, 'FontName', 'Times')
ylabel('Value of functions', 'FontSize', 14)
title('\bfVisualization of two intersecting curves', 'FontName', ...
   'Courier', 'FontSize', 14)
text(4.8, -0.1, '\itx \rm= 4.73','FontName', 'Times', 'FontSize', 12)
set(hc, 'LineWidth', 4)
set(hch, 'LineWidth', 2.5)
set(hsl, 'LineWidth', 0.25, 'color', 'g')
set(gca, 'FontSize', 14, 'LineWidth', 1.5)
set(b(1), 'FontSize', 10)
```

The execution of this script results in Figure 6.19. (Disclaimer: This graph was created to illustrate how to modify various attributes of its constitutive components. It violates the goals mentioned in Section 6.1, for the changes clearly do not enhance the clarity of the graph.)

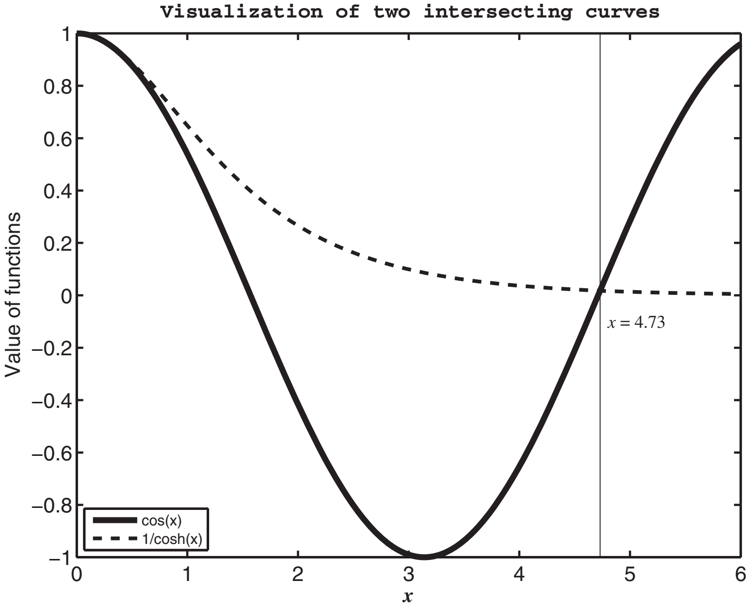


Figure 6.19 Alteration of several of the attributes of Figure 6.15.

6.3.6 Positioning One Figure Inside Another Figure

In certain situations, it may be desirable to insert one or more independent figures within a main figure. This insertion is accomplished with

`axes`('Position', [left, bottom, width, height])

where *Position* is a keyword and [*left*, *bottom*, *width*, *height*] is a four-element vector that defines the dimensions and position of a rectangular region that is inserted in the main figure as shown in Figure 6.20. The values for these four quantities range from 0 to 1 as indicated in the figure. All plotting commands that follow `axes` pertain to the inserted figure. The use of `axes` to insert an additional figure is illustrated in the following example:

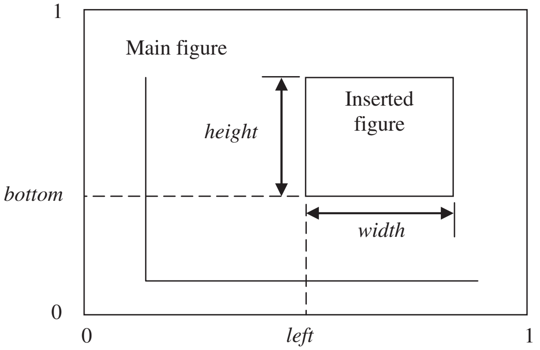


Figure 6.20 Coordinate system definitions for positioning a figure inside a main figure.

Example 6.1 Response of a single degree-of-freedom system to periodic forcing

We shall illustrate axes by displaying the nondimensional displacement response $x(\tau)$ of a single degree-of-freedom system subjected to a periodic pulse of period T and pulse duration t_d in the main figure and the amplitude of the frequency response function $H(\Omega)$ of the system in an inserted figure. If the natural frequency of the system is ω_n, then the governing relations are[7]

$$x(\tau) = \alpha\left[1 + 2\sum_{k=1}^{\infty} H(\Omega_k)\left|\frac{\sin(k\pi\alpha)}{k\pi\alpha}\right|\sin(\Omega_k\tau - \theta(\Omega_k) + \psi_k)\right]$$

where $\alpha = t_d/T < 1$, $\Omega_k = k\Omega_o$, $\Omega_o = \omega_o/\omega_n$, $\omega_o = 2\pi/T$,

$$H(\Omega_k) = \frac{1}{\sqrt{\left(1 - \Omega_k^2\right)^2 + \left(2\zeta\Omega_k\right)^2}}$$

$$\theta(\Omega_k) = \tan^{-1}\frac{2\zeta\Omega_k}{1 - \Omega_k^2}$$

$$\psi_k = \tan^{-1}\frac{\sin(k\pi\alpha)/k\pi\alpha}{0}$$

and $\zeta < 1$ is the damping factor. If we take 200 terms of the series, and assume that $\zeta = 0.1$, $\Omega_o = 0.03\sqrt{2}$, $-50 \le \tau \le 120$, and $\alpha = 0.4$, then the script is

```
k = 1:200;   alph = 0.4;   xi = 0.1;
Omo = 0.03*sqrt(2);   N = 400;
HOm = inline('1./sqrt((1-(Om*k).^2).^2+(2*xi*Om*k).^2)', 'k', 'Om', 'xi');
tau = linspace(-50, 120, N);
sn = sin(pi*k*alph)./(pi*k*alph);
thn = atan2(2*xi*Omo*k, (1-(Omo*k).^2));
psi = atan2(sn, 0);
cnt = sin(Omo*k'*tau-repmat(thn', 1, N)+repmat(psi', 1, N));
z = alph*(1+2*abs(sn).*HOm(k, Omo, xi)*cnt);
plot(tau, z, 'k-')
a = axis;    a(1) = -50;    a(2) = 120;
axis(a)
xlabel('\tau')
ylabel('x(\tau)')
axes('Position', [0.62, 0.62, 0.25, 0.25])
semilogy(k*Omo, HOm(k, Omo, xi), 'k-')
ylabel('H(\Omega)')
xlabel('\Omega')
box off
```

The execution of this script results in Figure 6.21. The determination of the values for *left* and *bottom* takes a little experimenting.

[7] Balachandran and Magrab, *Vibrations*, p. 260ff.

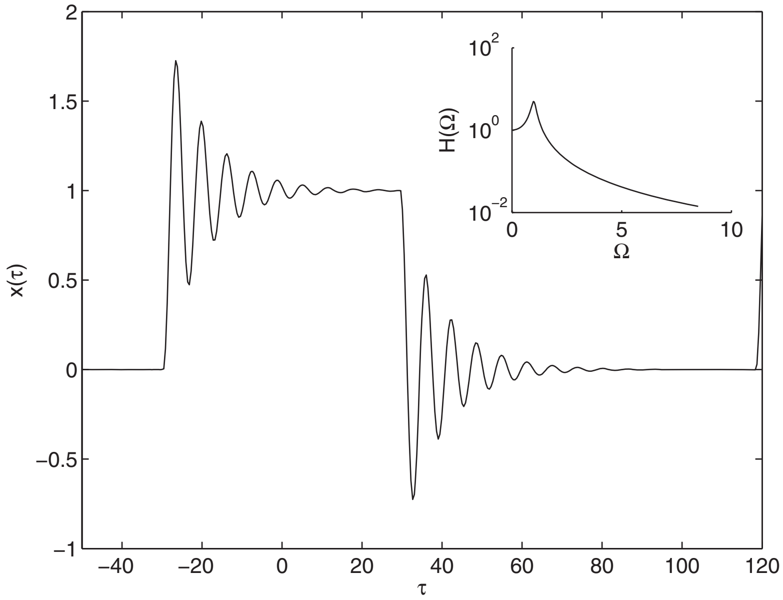


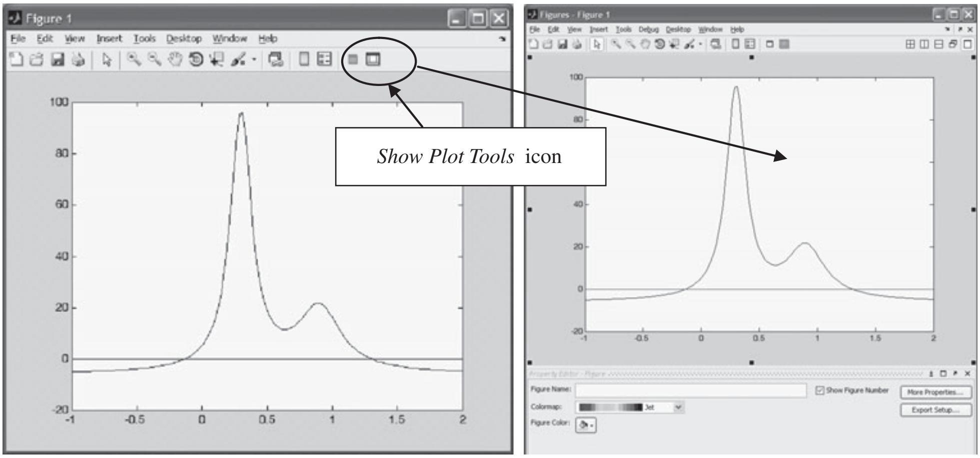
Figure 6.21 Illustration of the use of the position option of `axes`.

6.3.7 Interactive Plotting Tools

The changes to the attributes of the text, curves, and axes can also be made interactively in the figure window by selecting the appropriate operation from its *Tools* menu. After the changes have been made, the figure can be saved as a function M file. We shall illustrate how to use these interactive tools by using them to make a few modifications to the following program:

```
x = linspace(-1, 2, 150);
plot(x, humps(x), 'k-', [-1, 2], [0, 0], 'b-')
```

The execution of this program results in Figure 6.22a. By clicking on the *Show Plot Tools* icon, the figure window becomes that shown in Figure 6.22b. To insert a label for the *x*-axis, we click on *Insert* and select *X Label*. Then we type in ‘t (time)’ as shown in Figure 6.23a. The text property menus appear and they can be used to alter such text properties as font style and size and normal, bold, or italic. By clicking on the horizontal line, the line properties menus are displayed and the line properties can be altered as shown in Figure 6.23b. In a similar manner, other characteristics of the graph can be changed by adding lines, shapes, a legend, and text boxes. When completed, the resulting graph can be saved as a function M file by going to the *File* pull-down menu and selecting *Generate M-file*.



Show Plot Tools icon

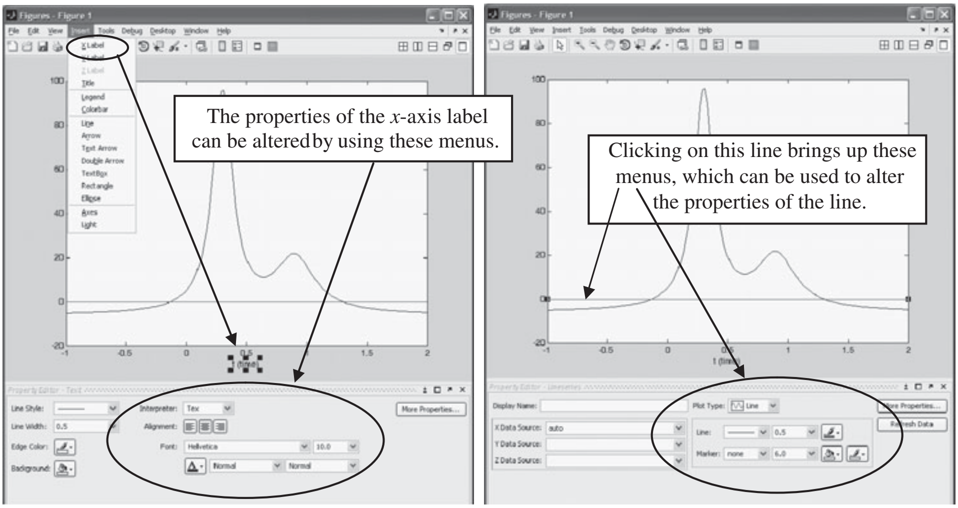
(a) (b)

Figure 6.22 (a) Window resulting from the execution of a program that plots `humps`. (b) Window resulting from clicking on the *Show Plot Tools* icon.

6.3.8 Animation

Another way to enhance a graph is through animation, which in MATLAB means creating a movie. The two functions that are used to create a movie are

A(k) = `getframe`



The properties of the *x*-axis label can be alteredby using these menus.

Clicking on this line brings up these menus, which can be used to alter the properties of the line.

(a) (b)

Figure 6.23 (a) Means by which an *x*-axis label can be inserted interactively. (b) Means by which line properties can be altered interactively.

which captures the *k*th movie frame of a total of *N* frames and

```
movie(A, nF, pbs)
```

which is used to play *nF* times the *N* frames captured in the matrix *A* by `getframe`. This function plays the movie at a playback speed *pbs*, which, if omitted, uses a default value of twelve frames per second. To create a movie in the avi (audio/video interleaved) format from the movie created by `movie`, one uses

```
movie2avi(A, 'FileName.avi', 'KeyWord', 'KeyWordValue');
```

The variable *A* is the variable used in `movie`. To create movies that can be shown in Microsoft PowerPoint, we set 'KeyWord' = 'compression' and 'KeyWordValue' = 'none'. The use of these commands is illustrated in the following example.

Example 6.2 Animation of a slider–crank mechanism

We shall illustrate the animation procedure by considering a slider–crank mechanism given in Figure 2.7 of Exercise 2.9 and, at the end, create a movie file in the avi format that can be used in, say, a Microsoft PowerPoint slide. The starting configuration of the slider–crank mechanism is shown in Figure 6.24. The procedure requires that each component of the image must be replicated, even those components that do not change. From Figure 6.24, we see that we must create the following stationary objects: the thin horizontal bar of thickness *f* that represents the ground, the dashed circle that indicates the circle traversed by the end of the crank arm *a*, and the center of the circle represented by a small circle. The moveable elements are the crank arm *a*, the crank arm *b*, the rectangle that represents the slider, and the two small circles that indicate a connection between *a* and *b* and between *b* and the slider. The horizontal distance that the slider moves as a function of the rotation angle φ of the crank arm *a* is

$$s = a\cos\varphi + \sqrt{b^2 - (a\sin\varphi - e)^2}$$

We shall assume the following numerical values: $0 \le \varphi \le 2\pi$ at $n = 40$ equally spaced positions, $a = 1, b = 2.5, e = 0.25, c = 0.5, d = 1$, and $f = 0.06$. The number of times that the forty frames is to be repeated (played over) is 5 (= *nF*).

```
n = 40;   phi = linspace(0, 2*pi, n);
a = 1;   b = 2.5;   e = 0.25;   nF = 5;
c = 0.5;   d = 1;   f = 0.06;
ax = a*cos(phi);   ay = a*sin(phi);
s = real(ax+sqrt(b^2-(ay-e).^2));
```

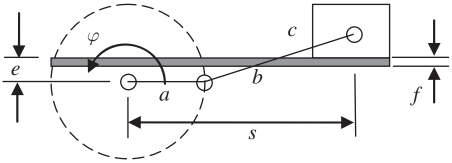


Figure 6.24 Slider–crank starting position and definitions of its components.

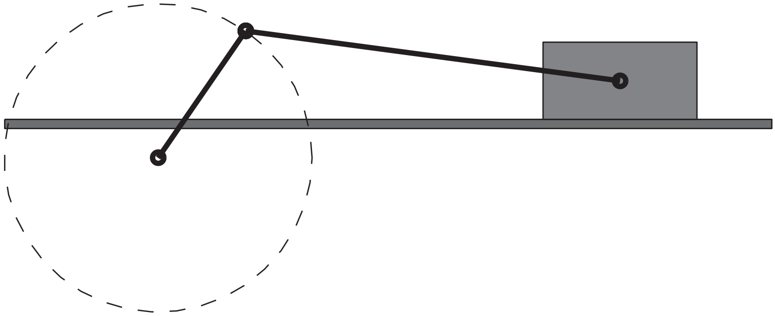


Figure 6.25 Slider–crank mechanism in it seventh position (image of the seventh frame).

```
v = [1.1*min(ax), 1.1*(max(s)+d/2) 1.1*min(ay), 1.1*max(ay)];
xgnd = [min(ax), max(s)+d/2, max(s)+d/2, min(ax), min(ax)];
ygnd = [e, e, e-f, e-f, e];
slidery = [e, e+c, e+c, e, e];    % Vertical component of slider is constant
for k = 1:n
  fill(xgnd, ygnd, 'r')    % Thin horizontal bar
  hold on
  plot(ax, ay, 'b--', 0, 0, 'ko');    % Dashed circle and center of circle
  sliderx = [s(k)-d/2, s(k)-d/2, s(k)+d/2, s(k)+d/2, s(k)-d/2];
  fill(sliderx, slidery, 'm');    % Slider position
  plot([0 ax(k)], [0;ay(k)], 'ko-', 'LineWidth', 2);
  plot([ax(k), s(k)], [ay(k), e+c/2], 'ko-', 'LineWidth', 2);
  axis(v)
  axis off equal
  SliderCrankFrame(k) = getframe;
  hold off
end
movie(SliderCrankFrame, nF, 30)
movie2avi(SliderCrankFrame, 'SliderCrankAvi.avi', 'compression', 'none')
```

The seventh frame that is produced by the execution of this program is shown in Figure 6.25.

6.4 EXAMPLES

The following examples are chosen to illustrate various plotting techniques that can be used to express a wide variety of results in different ways. They build on the various capabilities that were introduced in the preceding sections of this chapter.

Example 6.3 Polar plot: far field radiation pattern of a sound source

The normalized sound pressure at a large distance from the center of a circular piston in an infinite baffle that is oscillating at a frequency f is given by

$$p(r, \theta) = \left| \frac{J_1(ka\theta)}{ka\theta} \right| \qquad ka^2 \ll r \quad \text{and} \quad a \ll r$$

where r is the radial distance from the center of the piston, θ is the angle of r with respect to the plane of the baffle, k is the wave number, a is the radius of the piston, and $J_1(x)$ is the Bessel function of the first kind of order 1. The wave number is the reciprocal of the wavelength of the sound at frequency f; thus, ka is non dimensional. This model is a fair approximation to the angular dispersion of sound from a loudspeaker.

We shall create a polar plot of the normalized radiation pattern for $ka = 6\pi$ when θ ranges from $-\pi/2 < \theta < \pi/2$. We have chosen this solution to illustrate the use of polar, which plots results directly in polar coordinates. The script is

```
theta = linspace(-pi/2, pi/2, 300);
p = abs(besselj(1, 6*pi*theta)./(6*pi*theta));
polar(theta, p/max(p))
```

The execution of the script gives the curve shown in Figure 6.26a. Notice that the values of θ are chosen so that $ka\theta \neq 0$. The max function finds the maximum value in the vector p so that the ratio p/max(p) is the normalized radiation pattern whose maximum value is 1. To magnify the region in the vicinity of $r < 0.1$, we have two options.

The first is to depress the *Zoom In* icon in the figure window and zoom in on this region. The other option is to use axis, which is the option that we shall employ. If we type v = axis in the command window immediately after the script is run, we find that

```
v =
  -1.0000   1.0000   -1.1500   1.1500
```

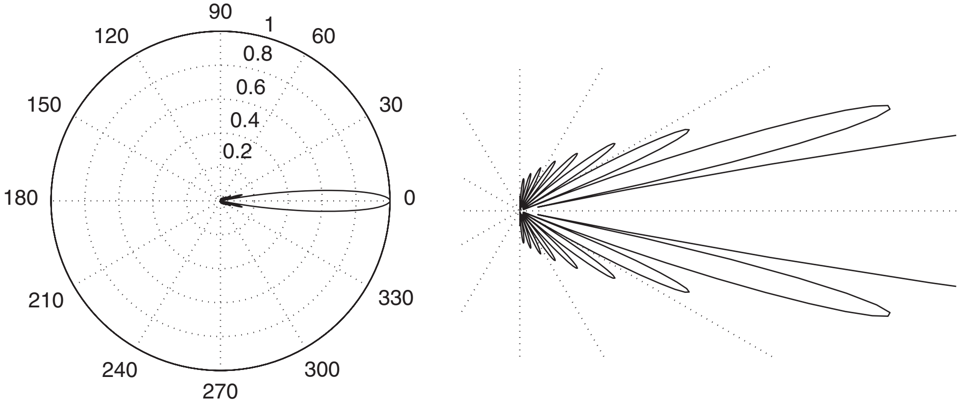
Thus, the limits of the x-axis are ±1 and those for the y-axis are ±1.15. Therefore, we can crop the view in Figure 6.25a by using

```
axis([-.02, 0.15, -0.05, 0.05])
```

Then, the modified script becomes

```
theta = linspace(-pi/2, pi/2, 300);
p = abs (besselj(1, 6*pi*theta)./(6*pi*theta));
polar(theta, p/max(p))
axis([-.02, 0.15, -0.05, 0.05])
```

which upon execution gives the result shown in Figure 6.26b.



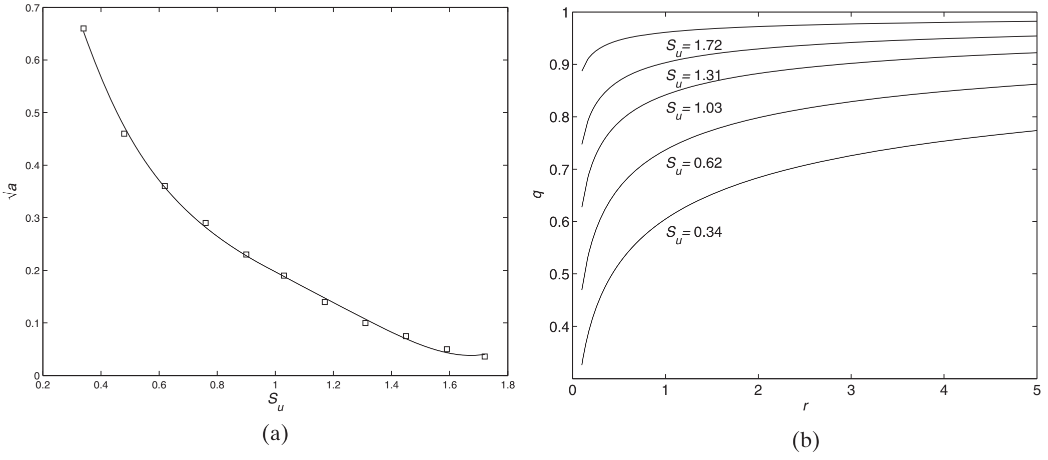
(a) (b)

Figure 6.26 (a) Polar representation of a radiation pattern. (b) Magnified region obtained by redefining axes limits with axis.

Example 6.4 Displaying and labeling multiple curves: notch sensitivity for steel

We now return to Example 5.1 and plot the notch sensitivity constant q over a range of values for $0.3 \leq S_u \leq 1.7$ GPa and $0.1 < r < 5$ mm. To make the script a little more readable, we create a function for the data that are to be fitted. We shall create a script that consists of two parts. The first part obtains the values of the coefficients of the fourth-order polynomial used to fit these data, and then displays the data points and the polynomial that fits these points. The second part uses the polynomial to generate a family of curves of notch sensitivity q versus the notch radius for several values of the ultimate strength of steel, S_u. The execution of the script results in Figures 6.27a and 6.27b.

```
function Example6_4
Su = linspace(0.34, 1.72, 50);    skip = [1, 3, 6, 8, 11];
ncs = NeuberData;    L = length(skip);
p = polyfit(ncs(:,1), ncs(:,2), 4);
figure(1)
plot(Su, polyval(p, Su), 'k', ncs(:,1), ncs(:,2), 'ks')
xlabel('\itS_u')
ylabel('\surd\ita')
figure(2)
[s, r] = meshgrid(ncs(skip,1), linspace(0.1, 5, 80));
notch = inline('1./(1+polyval(p, s)./sqrt(r))', 'p', 's', 'r');
plot(r, notch(p, s, r), 'k')
y(1:L,1) = 1;
lab = [repmat('\itS_u= \rm', L, 1) num2str(ncs(skip,1))];
text(repmat(1, 1, L), notch(p, ncs(skip,1), y)-0.03, lab)
xlabel('\itr')
ylabel('\itq')

function nd = NeuberData
nd = [0.34, 0.66; 0.48, 0.46; 0.62, 0.36; 0.76, 0.29; 0.90, 0.23; 1.03, 0.19; . . .
      1.17, 0.14; 1.31, 0.10; 1.45, 0.075; 1.59, 0.050; 1.72, 0.036];
```



(a)

(b)

Figure 6.27 (a) Neuber's constant for steel using a fourth-order polynomial fit. (b) Notch sensitivity for steel as a function of notch radius r for different values of ultimate strength.

The meshgrid function creates two (80 × 6) arrays, where the rows are the values of r and the columns are the values of S_u. Since **notch** was written using the dot notation, we can enter these arrays for their appropriate arguments. The placing of the curve's identifying labels is done with text and using the y-coordinate that is obtained from evaluating **notch** at $r = 1$ shifted down by 0.03 units. We use repmat to create vectors of values or strings of length L. We select the values of S_u with the vector of indices given in *skip*.

Example 6.5 Stability of a loaded structure

The static equilibrium position of an imperfection-sensitive structural model similar to that shown in Figure 5.26 is determined from[8]

$$p = \frac{\cot\theta\left(\sqrt{1 + \sin\theta} - \sqrt{1 + \sin\theta_0}\right)}{\sqrt{1 + \sin\theta}}$$

where θ_o is the amount of angular imperfection of the structure. For a given value of θ_o, the value of $\theta - \theta_o$ at which p has a maximum value is that value at which the system becomes unstable; that is, for $\theta - \theta_o < \theta_{\max} - \theta_o$ the system is stable and for those values for which $\theta - \theta_o > \theta_{\max} - \theta_o$ the system is unstable. We shall use these requirements to create Figure 6.28, which depicts these ideas graphically.

```
function Example6_5
tho = [0.001, 0.005, 0.01, 0.02];
th = linspace(0.01, 0.2, 200);
thmx = zeros(1, length(tho));
for k = 1:length(tho)
   thmx(k) = fminbnd(@prev, 0.01, 0.18, [], tho(k));
   indx = find(th<thmx(k));
   plot(th(indx)-tho(k), p(th(indx), tho(k)), 'k—')
   hold on
   plot(th(indx(end)+1:end)-tho(k), p(th(indx(end)+1:end), tho(k)), 'k-')
   if k == 1
      legend('Stable region', 'Unstable region')
   end
   plot(thmx(k)-tho(k), p(thmx(k), tho(k)), 'ko', 'MarkerFaceColor', 'k')
   text(thmx(k)-tho(k)+0.015, p(thmx(k), tho(k))+0.005, ['\theta_o=' numstr(tho(k))])
   text(thmx(k)-tho(k)-0.02, p(thmx(k), tho(k))-0.01, ['\theta=' num2str(thmx(k),3)])
end
plot(thmx-tho, p(thmx, tho), 'k-')
v = axis;   v(1) = 0;   v(3) = 0.32;   axis(v)
xlabel('\theta-\theta_o')
ylabel('p')

function s = p(th, tho)
s = cot(th).*(sqrt(1+sin(th))-sqrt(1+sin(tho)))./sqrt(1+sin(th));

function s = prev(th, tho)
s = -p(th, tho);
```

[8] G. J. Simitses, *Dynamic Stability of Suddenly Loaded Structures,* Springer-Verlag, NY, 1990, pp. 25–26.

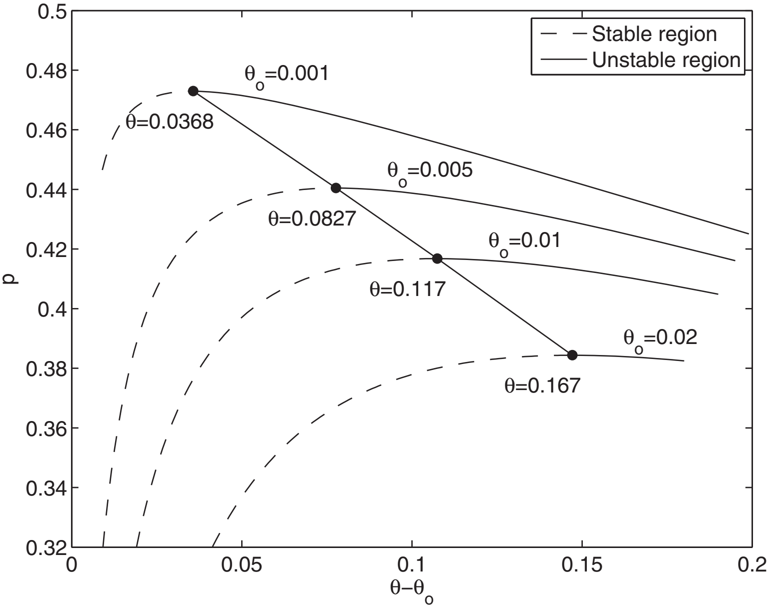


Figure 6.28 Stable regions for the static equilibrium position of an imperfection-sensitive structural model.

Example 6.6 Nontraditional histogram

We shall generate the histogram shown in Figure 6.29 and by doing so illustrate a variety of graph manipulation techniques. We first generate a set of normally distributed integers $x_n, n = 1, \ldots, N_{\mathrm{ran}}$ using randn. We then break the range of values of x_n into N_{bin} bins such that the width of each bin is $\Delta = (x_{\max} - x_{\min})/N_{\mathrm{bin}}$ and denote the center of the jth bin as B_j. The bin centered at B_j covers the range $B_{L,j} = B_j - \Delta/2 \le x_n < B_j + \Delta/2 = B_{Uj}$. The objective is to determine the number of x_n that fall into each bin and to plot the results as shown in Figure 6.29. The program has to be written so that it will work for different combinations of N_{ran}, N_{bin}, $x_{\max}$, and $x_{\min}$, the latter two parameters determined by randn. Since, in general, the number of integers that fall in the bins is different from each other, it is best to save these values in a cell array. Also, because of the nonequality of the upper limit of the bin width definition, one must make sure that $x_{\max}$ is included. For $N_{\mathrm{ran}} = 100$ and $N_{\mathrm{bin}} = 19$, the script is as follows:

```
Nran = 100;    Nbins = 19;
x = sort(round(30*randn([1, Nran])));    % Generate integer data and sort them
Delta = (x(end)-x(1))/Nbins;
BLU = x(1):Delta:x(end);
dist = cell([1, Nbins]);
Len = zeros(1, Nbins);
for k = 1:Nbins    % Determine values in each bin and N_j
   if k == Nbins
      indx = find((BLU(k)<=x) & (x<=BLU(k+1)));
   else
```

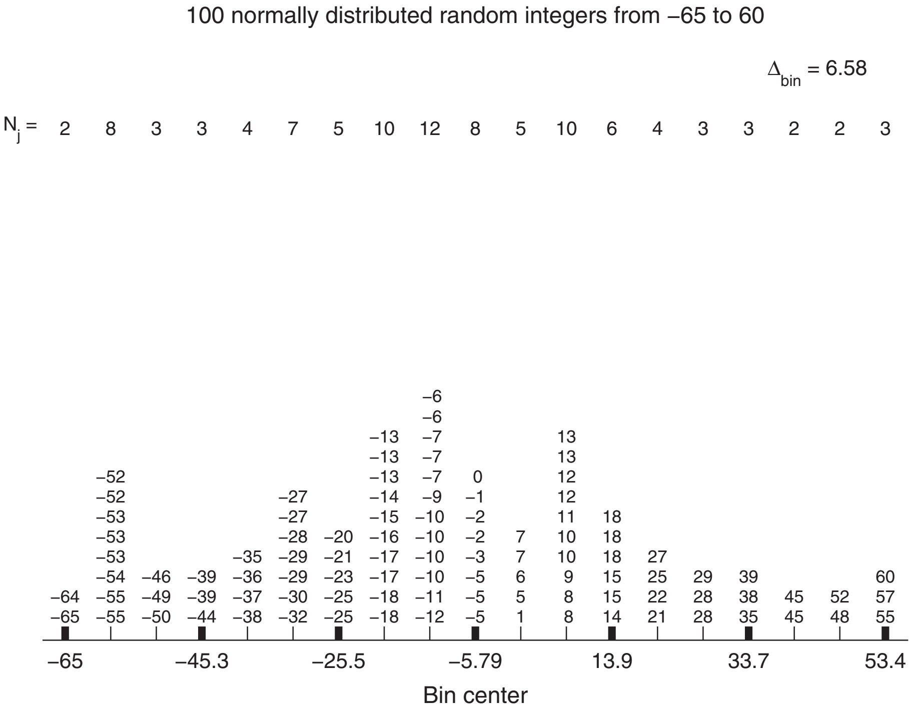


Figure 6.29 Nontraditional histogram.

```
        indx = find((BLU(k)<=x) & (x<BLU(k+1)));
      end
        dist(k) = {fliplr(x(indx))};   % Makes vertical column of values go from
                                                 smallest to largest
        Len(k) = length(indx);
    end
    Ymax = max(Len);   dY = Ymax/12;
    plot([x(1), x(end)], [0, 0], 'k-')
    hold on
    axis([x(1), x(end), 0, Ymax])
    axis off
    box off
    for k = 1:Nbins
      text(BLU(k)+Delta/2, 0.02*Ymax, num2str(dist{k}'), . . .
        'VerticalAlignment', 'Bottom', 'HorizontalAlignment', 'Center', 'fontsize', 9)
      plot([BLU(k)+Delta/2, BLU(k)+Delta/2], [0, 0.02*Ymax], 'k-')   % tick marks
      text(BLU(k)+Delta/2, 9*dY, num2str(Len(k)), . . .
        'HorizontalAlignment', 'Center', 'fontsize', 10)
    end
    text(x(1), 9*dY,'N_j = ', 'fontsize', 10, 'HorizontalAlignment', 'Right')
    for k = 1:3:Nbins
      text(BLU(k)+Delta/2, -0.03*Ymax, num2str(BLU(k),3), 'fontsize', 11, . . .
        'HorizontalAlignment','Center')
    plot([BLU(k)+Delta/2, BLU(k)+Delta/2], [0,0.02*Ymax], 'k-', 'LineWidth', 3)
                                                              % tick marks
end
text((x(1)+x(end))/2, -0.08*Ymax, 'Bin center', 'fontsize', 12, . . .
    'HorizontalAlignment', 'Center')
text((x(1)+x(end))/2, 11*dY, . . .
    [int2str(Nran) ' normally distributed random integers from ' int2str(x(1)) . . .
    'to' int2str(x(end)) ], 'fontsize', 12, 'HorizontalAlignment', 'Center')
text(0.9*x(end), 10*dY, ['\Delta = ' num2str(Delta,3)], 'fontsize', 11, . . .
    'HorizontalAlignment', 'Right')
```

Example 6.7 Frequency response functions of a two degree-of-freedom system

The frequency–response functions of a two degree-of-freedom system with masses m_1 and m_2, stiffness k_1 and k_2, and damping c_1 and c_2 are determined from[9]

$$H_{11}(j\Omega) = \frac{E(j\Omega)}{D(j\Omega)}, \qquad H_{21}(j\Omega) = \frac{C(j\Omega)}{D(j\Omega)}$$

$$H_{12}(j\Omega) = \frac{B(j\Omega)}{m_r D(j\Omega)}, \qquad H_{22}(j\Omega) = \frac{A(j\Omega)}{m_r D(j\Omega)}$$

where the terms in the numerators and the denominators are given by

$$\begin{aligned}
A(j\Omega) &= -\Omega^2 + 2(\zeta_1 + \zeta_2 m_r \omega_r) j\Omega + 1 + m_r\omega_r^2 \\
B(j\Omega) &= 2\zeta_2 m_r \omega_r j\Omega + m_r\omega_r^2 \\
C(j\Omega) &= 2\zeta_2 \omega_r j\Omega + \omega_r^2 \\
E(j\Omega) &= -\Omega^2 + 2\zeta_2\omega_r j\Omega + \omega_r^2 \\
D(j\Omega) &= \Omega^4 - j[2\zeta_1 + 2\zeta_2\omega_r m_r + 2\zeta_2\omega_r]\Omega^3 \\
&\qquad - [1 + m_r\omega_r^2 + \omega_r^2 + 4\zeta_1\zeta_2\omega_r]\Omega^2 + j[2\zeta_2\omega_r + 2\zeta_1\omega_r^2]\Omega + \omega_r^2
\end{aligned}$$

and

$$\omega_{nl}^2 = \frac{k_l}{m_l}, \qquad 2\zeta_l = \frac{c_l}{m_l\omega_{nl}} \qquad \omega_r = \frac{\omega_{n2}}{\omega_{n1}} \qquad l = 1, 2$$

$$m_r = \frac{m_2}{m_1} \qquad \Omega = \frac{\omega}{\omega_{n1}}, \qquad j = \sqrt{-1}$$

The magnitudes of the frequency–response functions are given by $|H_{il}(j\Omega)|$, $i, l = 1, 2$.

We shall now plot the four frequency–response magnitudes as a function of Ω for $\zeta_1 = \zeta_2 = 0.1$, $m_r = 0.6$, and $\omega_r = 0.7$. The script is

```
function Example6_7
z1 = 0.1;    z2 = 0.1;    wr = 0.7;    mr = 0.6;    wend = 2;
w = linspace(0, wend, 200);
kase = [11, 12, 21, 22];
```

[9] Balachandran and Magrab, *Vibrations*, p. 483.

```
for k = 1:4
  subplot(2, 2, k)
  H = tf2dof(w, mr, wr, z1, z2, kase(k));
  plot(w, H, 'k-')
  hold on
  xlabel('\Omega')
  ylabel(['|H_{' num2str(kase(k)) '}(\Omega)|'])
  v = axis;   v(2) = wend;   axis(v)
end

function H = tf2dof(w, mr, wr, z1, z2, kase)
D = w.^4-(2*z1+2*z2*wr.*mr+2*z2*wr)*1i.*w.^3 ...
     -(1+mr.*wr.^2+wr.^2+4*z1*z2*wr).*w.^2 ...
     +(2*z2*wr+2*z1*wr.^2).*w*1i+wr.^2;
switch kase
  case 11
    q1 = (-w.^2+2*z2*wr.*w*1i+wr.^2)./D;
  case 12
    q1 = (2*z2*wr.*w*1i+wr.^2)./D;
  case 21
    q1 = (2*z2*mr*wr*w*1i+mr*wr^2)./D/mr;
  case 22
    q1 = (-w.^2+2*(z1+z2*mr*wr)*w*1i+1+mr*wr^2)./D/mr;
end
H = abs(q1);
```

The execution of this script gives the results shown in Figure 6.30.

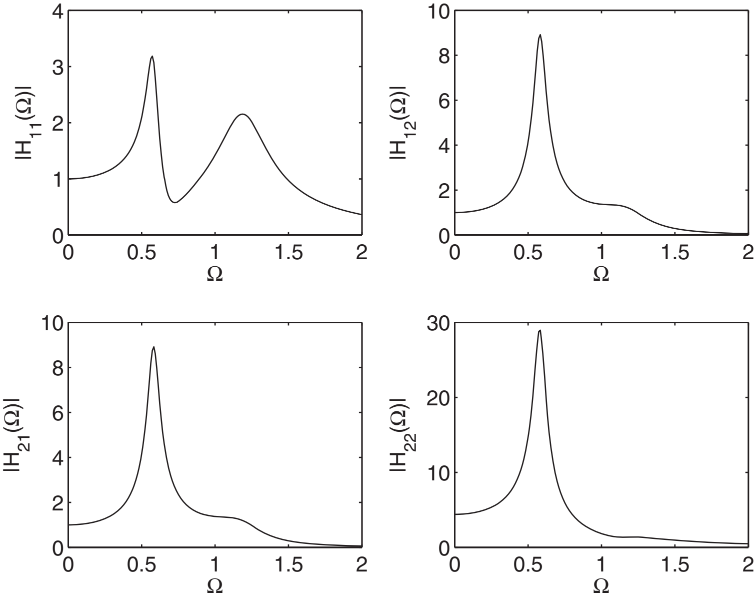


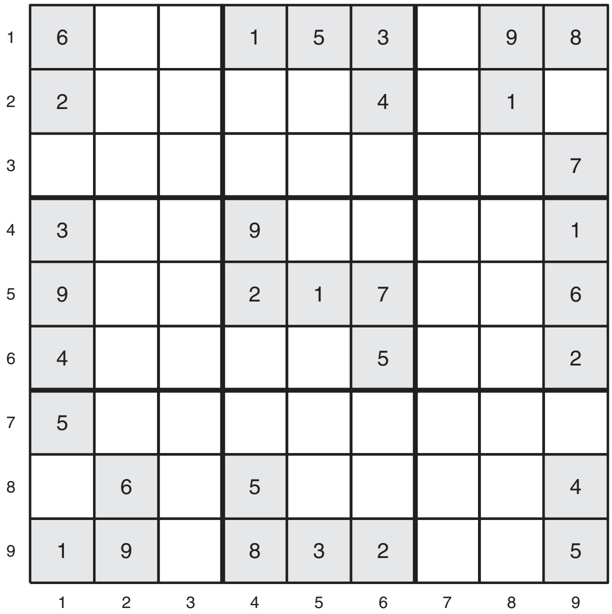
Figure 6.30 Magnitude of the frequency–response functions of a two degree-of-freedom system for $\zeta_1 = \zeta_2 = 0.1$, $m_r = 0.6$, and $\omega_r = 0.7$.

Example 6.8 Sudoku: Drawing squares

We shall give the program that replicates the grid shown in Figure 6.31 as a means to shown how one can manipulate the plotting area. We start by creating an appropriate 9 by 9 array of the values shown, using zeros to represent the squares without numbers. The fill is yellow, the numbers outside the grid are 10 point and aligned at the center of their respective adjacent squares, and the numbers inside the grid are 13 point and they are centered within each grid. The line widths are 1.5 point except for the four heavy lines, which are 3 point. These heavy lines have to be drawn after the coloring of the squares has been completed. Note that the script given below will still perform properly when the values, number of values, and locations of values in *S* are changed.

The script is

```
S = flipud([6, 0, 0 1, 5, 3, 0, 9, 8; ...
    2, 0, 0, 0, 0, 4, 0, 1, 0; ...
    0, 0, 0, 0, 0, 0, 0, 0, 7; ...
    3, 0, 0, 9, 0, 0, 0, 0, 1; ...
    9, 0, 0, 2, 1, 7, 0, 0, 6; ...
    4, 0, 0, 0, 0, 5, 0, 0, 2; ...
    5, 0, 0, 0, 0, 0, 0, 0, 0; ...
    0, 6, 0, 5, 0, 0, 0, 0, 4; ...
    1, 9, 0, 8, 3, 2, 0, 0, 5]);
hold on
axis off equal
```



1	6			1	5	3		9	8
2	2					4		1	
3									7
4	3			9					1
5	9			2	1	7			6
6	4					5			2
7	5								
8		6		5					4
9	1	9		8	3	2			5
	1	2	3	4	5	6	7	8	9

Figure 6.31 Sudoku grid.

```
for r = 1:9
  for c = 1:9
    if S(r, c)~=0
      fill([c-1, c, c, c-1, c-1], [r-1, r-1, r, r, r-1], 'y', 'LineWidth', 1.5)
      text(c-0.5, r-0.5, int2str(S(r, c)), 'fontsize', 13, . . .
        'HorizontalAlignment','center', 'VerticalAlignment','middle')
    else
      plot([c-1, c, c, c-1, c-1], [r-1, r-1, r, r, r-1], 'k-', 'LineWidth', 1.5)
    end
  end
end
for k = 1:9
  text(k-0.5, -0.3, int2str(k), 'fontsize', 10, 'HorizontalAlignment', 'center', . . .
    'VerticalAlignment', 'middle')
  text(-0.3, k-0.5, int2str(10-k), 'fontsize', 10, 'HorizontalAlignment', 'center', . . .
    'VerticalAlignment', 'middle')
end
Hev = [3, 6];
plot([Hev; Hev], [zeros(1, 2); repmat(9, 1, 2)], 'k', 'linewidth', 3);
plot([zeros(1, 2); repmat(9, 1, 2)], [Hev; Hev], 'k', 'linewidth', 3);
```

6.5 SUMMARY OF FUNCTIONS INTRODUCED IN CHAPTER 6

Attributes that can be assigned to lines and points are summarized in Table 6.2. Keywords that are used to display Greek letters and a wide range of mathematical symbols are given in Table 6.10. Keywords for positioning text are given in Table 6.11, those for additional text attributes are given in Table 6.12, and those for changing the attributes of the legend are given in Table 6.13. Keywords for specifying attributes of lines are given in Table 6.14. In Table 6.15, we have summarized the plotting functions introduced in this chapter.

TABLE 6.15 MATLAB Functions Introduced in Chapter 6

MATLAB function	Description
`axes`	Means to set the properties of the axes
`axis`	Scales and changes the appearance of the axes
`axis equal`	Sets axes so that aspect ratio units are the same in each direction
`axis image`	Makes the aspect ratio of the axes the same as those of the image
`axis on/off`	Turns on and off the visibility of the axes
`bar`	Creates a bar chart
`box on/off`	Turns on and off the display of the axes boundaries
`convhull`	Plots the convex hull of a set on points in a plane
`delauney`	Creates a set of triangles in a plane using the Delauney triangle criterion
`figure`	Creates an individual figure window
`fill`	Creates and colors an area enclosed by a polygon
`gca`	Gets current axis handle for the current figure

(*Continued*)

TABLE 6.15 Continued

MATLAB function	Description
`getframe`	Captures a movie frame
`grid on/off`	Turns on and off the grid lines in 2D and 3D plots
`hold on/off`	When 'on' system adds new graphic object to figure; when 'off' it replaces objects in the figure
`image`	Displays an image object
`imread`	Reads a digital image file
`legend`	Displays a legend on the graph
`loglog`	Creates a logarithmic *x*-axis and a logarithmic *y*-axis
`movie`	Plays movie frames
`movie2avi`	Creates avi movie from `movie`
`patch`	Creates and colors an area enclosed by a polygon
`pie`	Creates a pie chart
`pie3`	Creates a 3D pie chart
`plot`	Creates a linear 2D plot
`plotyy`	Plots a linear 2D graph with a different *y*-axis on the left- and right-hand sides
`polar`	Creates a polar coordinate plot
`print`	Creates a hardcopy of the current figure
`semilogx`	Creates a logarithmic *x*-axis and a linear *y*-axis
`semilogy`	Creates a logarithmic *y*-axis and a linear *x*-axis
`set`	Sets object properties
`stairs`	Creates a stair step plot
`stem`	Creates a discrete data plot with data values connected by straight lines emanating from the *x*-axis
`subplot`	Divides current figure into a number of panes, each with its own set of axes
`text`	Places a text object in the current axes
`title`	Places a title on the current axes
`triplot`	Plots the output from `delauney`
`voronoi`	Creates a Voronoi diagram
`xlabel`	Labels the current *x*-axis
`ylabel`	Labels the current *y*-axis

EXERCISES

Note: The actual plotting in almost all the exercises can be done using vector and dot operations and `meshgrid`. In these cases, use the `for` structure only to increment through a range of parameters as appropriate.

Section 6.2.1

6.1 Plot the following curves.[10, 11] For all figures use `axis equal off`. For those that request multiple plots, use `subplot`.

[10] D. von Seggern, *CRC Standard Curves and Surfaces*, CRC Press, Inc., Boca Raton, FL, 1993.

[11] E. W. Weisstein, *CRC Concise Encyclopedia of Mathematics,* Chapman & Hall, Boca Raton, FL, 2003.

Name	Parameter values	Equations
Cycloid	$-\pi \le \varphi \le 3\pi$ $r_a = 0.5, 1, 1.5$	$x = r_a\varphi - \sin\varphi$ $y = r_a - \cos\varphi$
Lemniscate	$-\pi/4 \le \varphi \le \pi/4$	$x = \cos\varphi\sqrt{2\cos(2\varphi)}$ $y = \sin\varphi\sqrt{2\cos(2\varphi)}$
Archimedean spiral	$0 \le \varphi \le 6\pi$	$x = \varphi\cos\varphi$ $y = \varphi\sin\varphi$
Logarithmic spiral	$0 \le \varphi \le 6\pi$ $k = 0.1$	$x = e^{k\varphi}\cos\varphi$ $y = e^{k\varphi}\sin\varphi$
Cardioid	$0 \le \varphi \le 2\pi$	$y = 2\cos\varphi - \cos 2\varphi$ $y = 2\sin\varphi - \sin 2\varphi$
Astroid	$0 \le \varphi \le 2\pi$	$x = 4\cos^3\varphi$ $y = 4\sin^3\varphi$
Epicycloid	(1) $R_r = 3, a_r = 0.59$ $0 \le \varphi \le 2\pi$ (2) $R_r = 2.5, a_r = 2$ $0 \le \varphi \le 4\pi$	$x = (R_r + 1)\cos\varphi - a_r\cos(\varphi(R_r + 1))$ $y = (R_r + 1)\sin\varphi - a_r\sin(\varphi(R_r + 1))$
Hypocycloid	$R_r = 3, a_r = 0.5, 1, 2$ $0 \le \varphi \le 2\pi$	$x = (R_r - 1)\cos\varphi + a_r\cos(\varphi(R_r - 1))$ $y = (R_r - 1)\sin\varphi - a_r\sin(\varphi(R_r - 1))$
Eight curve	$0 \le \varphi \le 2\pi, a = 2$	$x = a\sin\varphi$ $y = a\sin\varphi\cos\varphi$
Butterfly	$0 \le x \le 1$	$y^6 = x^2 - x^6$
Dumbbell	$0 \le x \le 1, a = 2$	$y^2 = a^2\left(x^4 - x^6\right)$
Bicuspid	$-1 \le x \le 1, a = 1$	$\left(x^2 - a^2\right)\left(x - a\right)^2 + \left(y^2 - a^2\right)^2 = 0$
Super ellipse	$-a \le x \le a, a = 2, b = 1$ $n = 2/3, 3, 7$	$\left\lvert\dfrac{x}{a}\right\rvert^n + \left\lvert\dfrac{y}{b}\right\rvert^n = 1$
Gear	$a = 1, b = 10, n = 12$ $0 \le \varphi \le 2\pi$ [use 600 values for φ]	$x = (a + (1/b)\tanh[b\sin(n\varphi)])\cos\varphi$ $y = (a + (1/b)\tanh[b\sin(n\varphi)])\cos\varphi$
Folium	$0 \le \varphi \le 2\pi, a = 2, b = 3$	$r^2 = \cos\varphi(4a\sin^2\varphi - b)$

6.2 Consider the following polynomial from $-12 \le x \le 7$.

$$y = 0.001x^5 - 0.01x^4 - 0.2x^3 + x^2 + 4x - 5$$

Plot only its positive values such that the positive portions of y start and end on the x-axis.

6.3 Consider two straight lines of equal length with each line having N equally spaced points. The two lines form an angle φ with respect to each other. Create a program that generates the set of straight lines connecting these points as shown in Figure 6.32. If the spacing between the points is Δ, then the coordinates of the points on the line that has been rotated an angle φ are

$$x = n\Delta \cos\varphi$$
$$y = n\Delta \sin\varphi \quad n = 0, 1, \ldots, N$$

Test your program for $\varphi = \pi/3$, $\Delta = 1$, and $N = 17$.

6.4 Write a script that produces three or more circles around a central circle of radius $r_b = 1.5$ as shown for $n = 5$ circles in Figure 6.33. The radius of the n outer circles r_s is

$$r_s = \frac{r_b \sin(\pi/n)}{1 - \sin(\pi/n)}$$

Have the script ask the user for the number of circles. The script can be written without using the `for` loop.

6.5 The piecewise linear map defined by the points with the coordinates

$$x_{n+1} = 1 - y_n + |x_n|$$
$$y_{n+1} = x_n \qquad n = 1, 2, \ldots, N$$

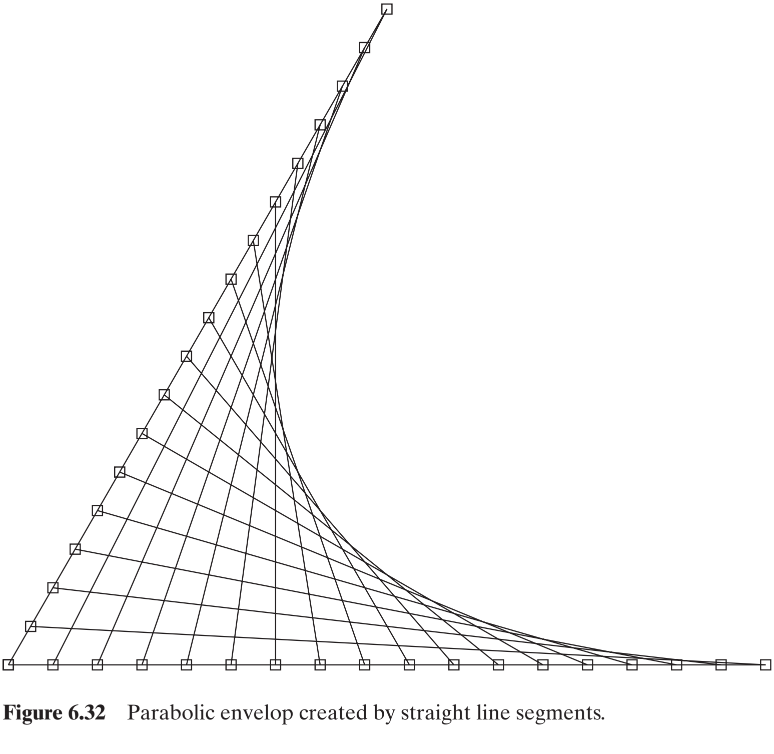


Figure 6.32 Parabolic envelop created by straight line segments.

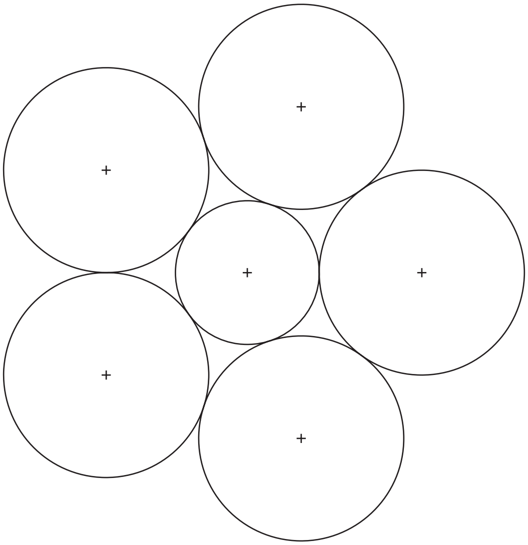


Figure 6.33 Five circles on a circle.

produces the gingerbread man shown in Figure 6.34. Reproduce this figure by using $x_1 = 1, y_1 = 3.65, N = 15{,}000$, and the point (.) as the marker.

6.6 The Cartesian locations of bi-cylinder coordinates are[12]

$$x = \frac{a \sinh\eta}{\cosh\eta - \cos\varphi}$$
$$y = \frac{a \sin\varphi}{\cosh\eta - \cos\varphi}$$

If $0 \le \eta \le 1, 0 \le \varphi \le 2\pi, a = 1.5$ and the graphics area is limited to a 4×4 square, then replicate Figure 6.35.

Sections 6.3.2–6.3.5

6.7 The probability density function of a time-varying signal is used to relate the probability that over a period of time T the signal's amplitude has a value between x and $x + dx$. In other words, it is used to obtain a measure of the fraction of time the signal spends within this amplitude range. The probability density function can be approximated by

$$P(x) = \lim_{\substack{\Delta x \to 0 \\ T \to \infty}} \left[\frac{1}{T\Delta x} \sum_{i}^{N} \Delta t_i \right]$$

[12] P. Moon and D. E. Spencer, *Field Theory Handbook*, Springer-Verlag, Berlin, 1961, p. 89.

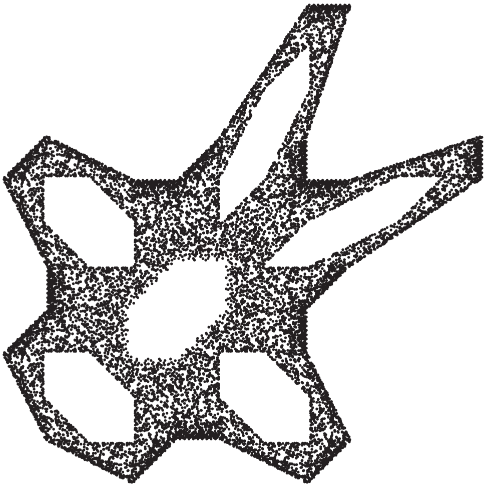


Figure 6.34 Gingerbread man.

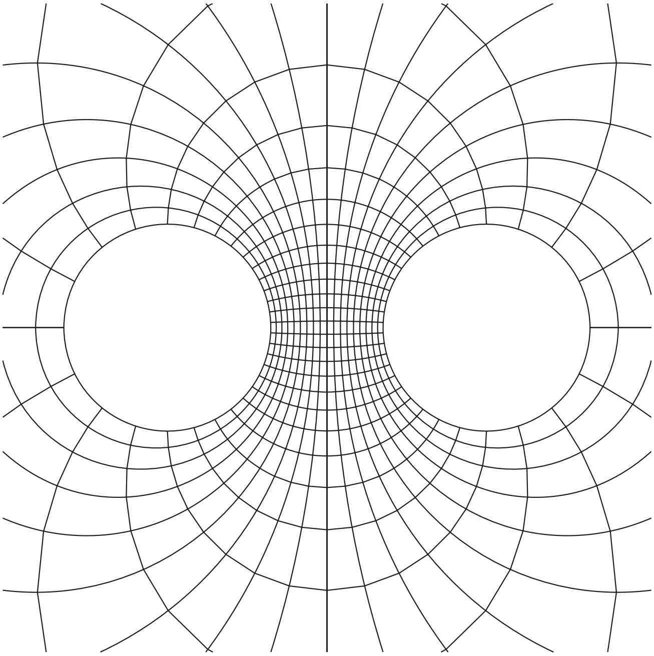


Figure 6.35 Bi-cylinder coordinate system.

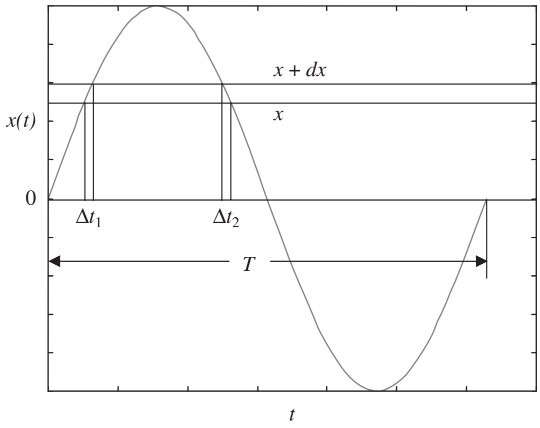


Figure 6.36 Determination of the amount of time a sine wave spends between x and $x + dx$.

where the terms in this expression are shown for one period of a sine wave in Figure 6.36. The probability density function $P(x)$ of a sine wave of amplitude A_o is given by

$$\begin{aligned} P(x) &= \frac{1}{\pi\sqrt{A_0^2 - x^2}} \qquad |x| \le A_0 \\ &= 0 \qquad\qquad\qquad |x| > A_0 \end{aligned}$$

Estimate the probability density function for

$$x = A_0 \sin t$$

for $-\pi \le t \le \pi$ and compare the results to the exact values. Let $A_o = 2$, the number of amplitude bins equal to 20, and the number data points in the time interval equal 400 $(=N)$. Plot the estimated values of $P(x)$ and the exact values. The results should look like those shown in Figure 6.37.

6.8 The force on a Belleville spring is proportional to C_1, where

$$C_1 = 0.5d_t^3 - 1.5h_t d_t^2 + (1 + h_t^2)d_t$$

and $h_t = h/t$, $d_t = \delta/t$, and δ is the deflection of the spring. Plot C_1 as a function of d_t when h_t varies from 1 to 3 in increments of 0.5 and d_t varies from 0 to 5. Label the curves and limit the y-axis to 8. The results should look like those shown in Figure 6.38.

6.9 Consider the gear tooth shown in Figure 6.39. It is seen that if the gear has n teeth, then each tooth appears every $2\pi/n$ radians. Let R_b be the radius of the base circle, R_T the radius of the tooth tip circle, and R $(R_b \le R \le R_T)$ the radius of a point on the profile of the tooth. Then the polar coordinates of the profile of one gear tooth (R, ψ), including the space between an adjacent tooth, are given in Table 6.16. In Table 6.16, φ_s is the gear pressure angle, either 14.5°, 20°, or 25°, $R_s = nm/2$ is the standard pitch radius, m is the gear module, and t_s is the tooth thickness at R_s.

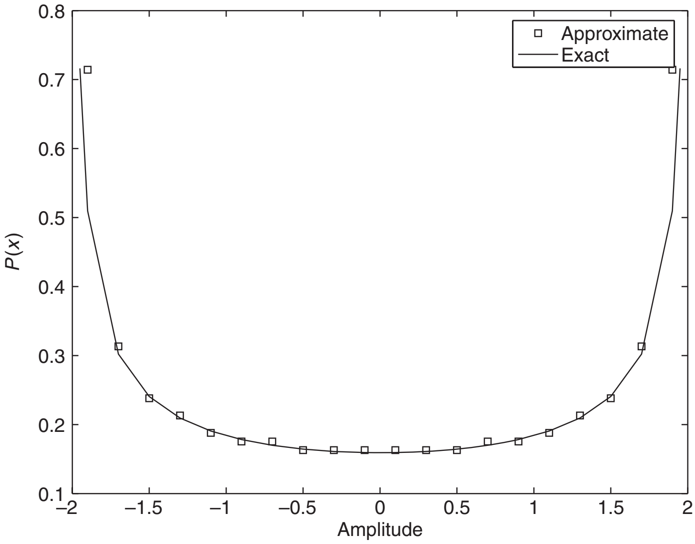


Figure 6.37 Probability density function of a sine wave.

If a gear has twenty-four teeth, a pressure angle of 20°, a module of 10 mm, a tooth thickness of 14.022 mm, a base radius of 90.21 mm, and a tip radius of 106 mm, then draw the gear in two ways: using `polar` and using `plot`.

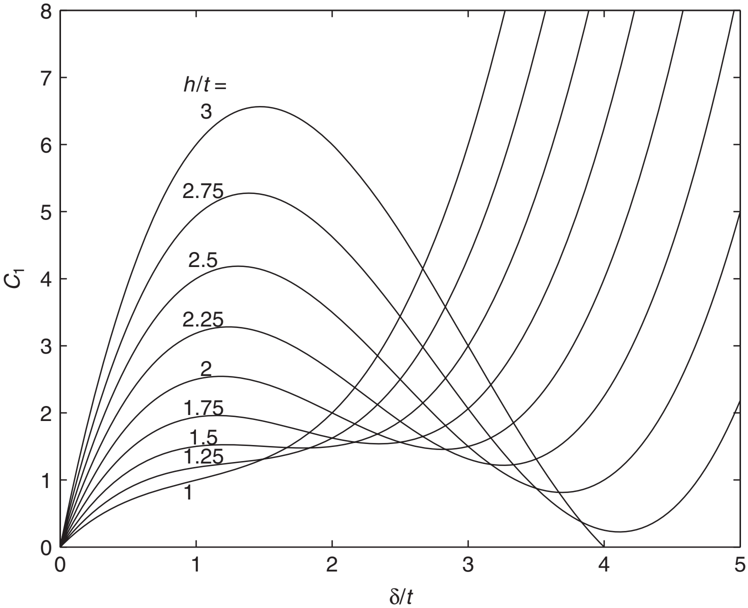


Figure 6.38 Belleville spring constant C_1.

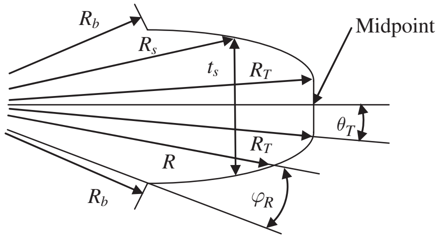


Figure 6.39 Nomenclature of a gear tooth.

6.10 The efficiency, in percent, of a power screw when the friction of the collar is ignored, is

$$e = 100\,\frac{\cos(\alpha) - \mu\tan(\lambda)}{\cos(\alpha) + \mu\cot(\lambda)}\,\%$$

where μ is the coefficient of friction, λ is the lead angle of the screw and α is the thread angle. Plot the efficiency as a function of λ for $0 < \lambda < 90°$, $\mu = 0.02, 0.05, 0.10, 0.15, 0.20$, and 0.25, and for two thread angles: $\alpha = 7°$ and 14.5°. Label the figure and the individual curves and use the `axis` function to limit the efficiency to 0 to 100%. The results should look like those shown in Figure 6.40.

6.11 Consider the rectangle shown in Figure 6.41 where it is seen that

$$r_1 = \sqrt{d^2 + (W/2)^2} \qquad \alpha = \tan^{-1}(W/2d)$$
$$r_2 = \sqrt{(d + L)^2 + (W/2)^2} \qquad \beta = \tan^{-1}\big(W/2(d + L)\big)$$

If the values of L, W, and d are given, then create a script that generates the maximum number of nonoverlapping replicated rectangles as shown in Figure 6.42. The values of $L = 1$, $W = 2$, and $d = 2$ were used to generate Figure 6.42. The maximum number of rectangles can be determined from `floor`(π/α). This script can be written without using a `for` loop.

6.12 Create the square root spiral shown in Figure 6.43.

6.13 Using the results of Exercise 1.19, plot $\sigma_x/p_{\max}$, $\sigma_z/p_{\max}$, and $\tau_{xz}/p_{\max} = \tau_{yz}/p_{\max} = 0.5 \times (\sigma_x/p_{\max}-\sigma_z/p_{\max})$ as a function of z/a for $\nu_1 = 0.3$. Label the figure and identify the curves.

6.14 Using the results of Exercise 1.20, plot $\sigma_x/p_{\max}$, $\sigma_y/p_{\max}$, $\sigma_z/p_{\max}$, and $\tau_{yz}/p_{\max}$ as a function of z/b for $\nu = 0.3$. Label the figure and identify the curves.

TABLE 6.16 Definitions of the Various Sectors of the Gear Tooth Shown in Figure 6.39

R	ψ	Definitions
$R_b \le R \le R_T$	$\mathbf{inv}(\varphi(R))$	$\varphi(R) = \cos^{-1}(R_b/R)$ $\mathbf{inv}(x) = \tan(x) - x$
R_T	$\mathbf{inv}(\varphi(R_T)) \le \psi \le \mathbf{inv}(\varphi(R_T)) + 2\theta_T$	$\theta_T = 0.5t_s/R_s + \mathbf{inv}(\varphi_s) - \mathbf{inv}(\varphi(R_T))$
$R_b \le R \le R_T$	$2[\theta_T + \mathbf{inv}(\varphi(R_T))] - \mathbf{inv}(\varphi(R))$	
R_b	$2[\theta_T + \mathbf{inv}(\varphi(R_T))] \le \psi \le 2\pi/n$	

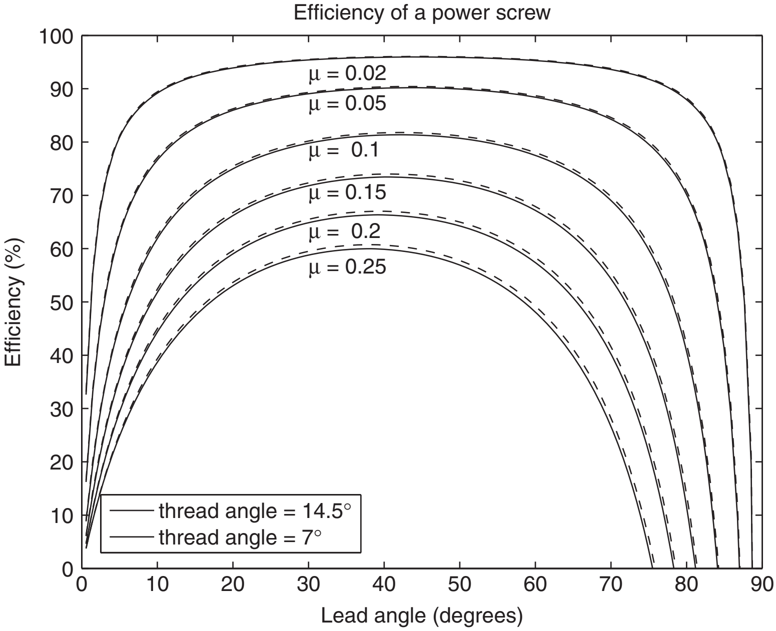


Figure 6.40 Efficiency of a power screw.

6.15 The relationship between the lead angle of a worm gear λ, the ratio $\beta = N_1/N_2$, where N_1 is the number of teeth on the worm gear and N_2 is the number of teeth on the driven gear, the center distance between shafts C, and the normal diametral pitch P_{dn} is

$$K = \frac{2P_{dn}C}{N_2} = \frac{\beta}{\sin\lambda} + \frac{1}{\cos\lambda}$$

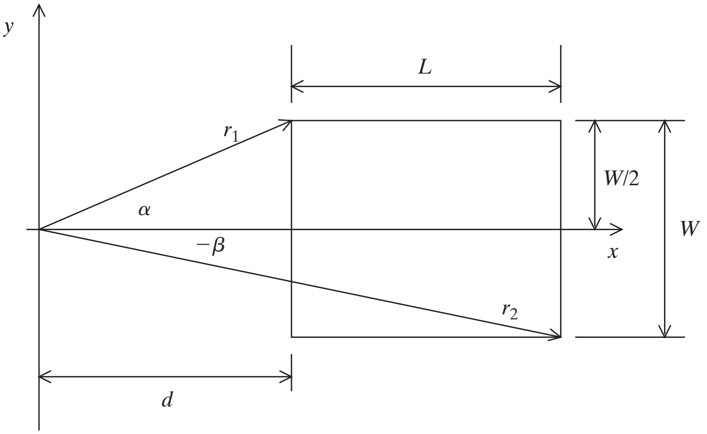


Figure 6.41 Description of a rectangle for Exercise 6.11.

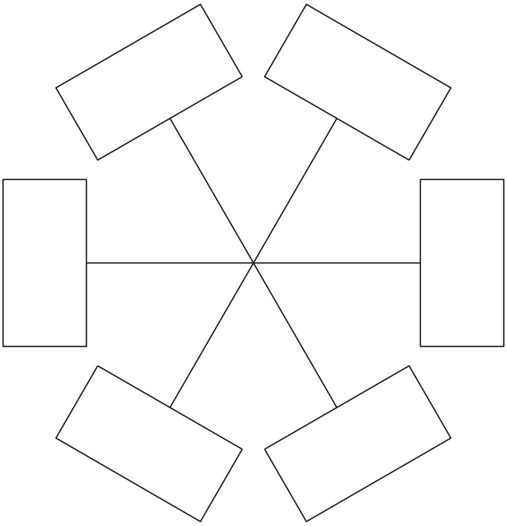


Figure 6.42 Replicated nonoverlapping rectangles.

Plot K as a function of λ for $1° \leq \lambda \leq 40°$ and $\beta = 0.02, 0.05, 0.08, 0.11, 0.15, 0.18, 0.23, 0.30$. Label the figure and the curves. Limit the range of the y-axis from 1 to 2. On the same figure, plot the results of Exercise 5.44 by drawing a line that connects the minimum values of each curve. Do this by incorporating the appropriate function(s) and portions of the script from Exercise 5.44 into the script written for this exercise. The results should look like those shown in Figure 6.44.

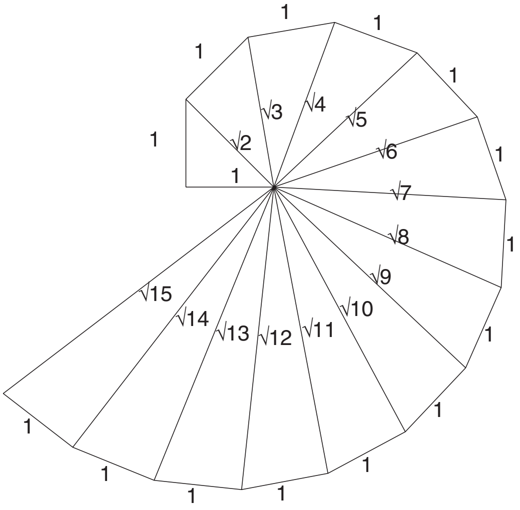


Figure 6.43 Square root spiral.

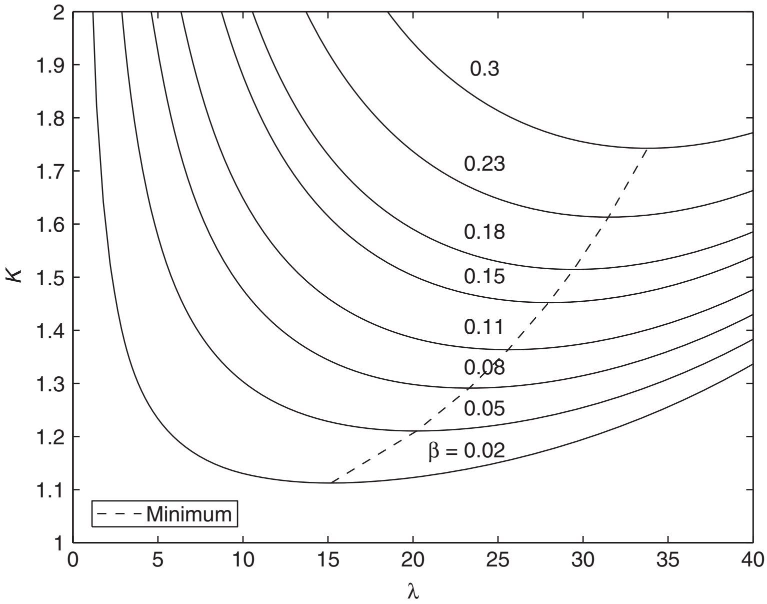


Figure 6.44 Lead angle of a worm gear.

6.16 In Exercise 5.17, we gave the following Colebrook formula from which the pipe's coefficient of friction λ could be estimated:

$$\lambda = \left[-2\log_{10}\left(\frac{2.51}{\mathrm{Re}\sqrt{\lambda}} + \frac{0.27}{d/k}\right)\right]^{-2} \qquad \mathrm{Re} \geq 4000$$

where Re is the Reynolds number, d is the diameter of the pipe, and k is the surface roughness. For smooth pipes ($k \cong 0$ or; $d/k > 100{,}000$), we have that

$$\lambda = \left[2\log_{10}\left(\frac{R_e\sqrt{\lambda}}{2.51}\right)\right]^{-2} \qquad R_e \geq 4000$$

Plot $\log_{10}(\lambda)$ as a function of $\log_{10}(\mathrm{Re}), 4 \times 10^3 \leq \mathrm{Re} \leq 10^7$, for $d/k = 20; 50; 100; 200; 500; 1{,}000; 2{,}000; 5{,}000; 10{,}000; 20{,}000; 50{,}000; 100{,}000;$ and ∞ ($k = 0$). Use `semilogx` and label the figure and the curves. Place the identifiers for the curves to the right of $R_e = 10^7$—that is, outside the figure's right-hand vertical axis. The resulting figure is known as the Moody diagram of friction factors for pipe flow. The results should look like those in Figure 6.45.

6.17 In optimization analysis, it is often beneficial to plot the function being optimized (called the objective function) and its constraints (regions in which the solution is required to reside). Consider the requirement to minimize

$$f(x_1, x_2) = (x_1 - 1)^2 + (x_2 - 1)^2$$

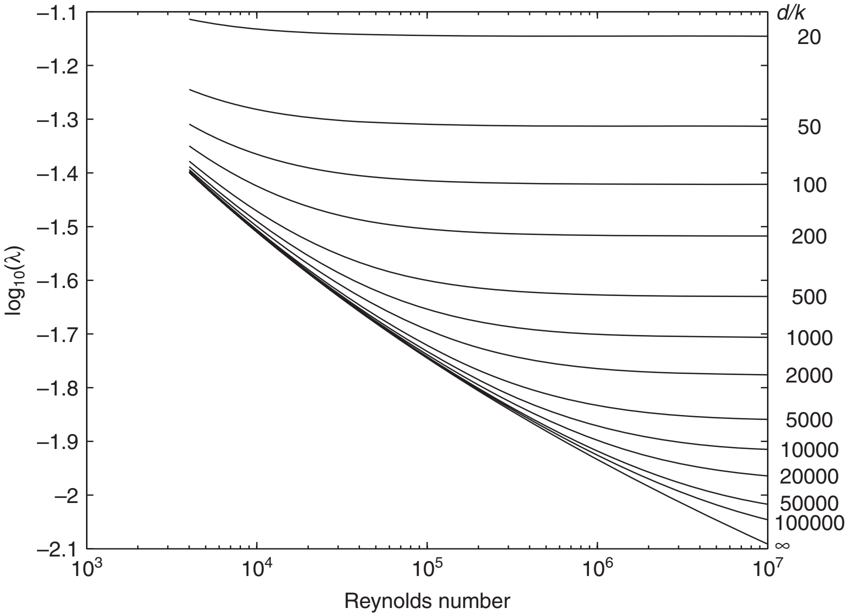


Figure 6.45 Moody diagram.

subject to the constraints

$$g_1 = (x_1 - 3)^2 + (x_2 - 1)^2 - 1 \le 0$$
$$g_2 = 2x_1 - x_2 - 5 \le 0$$

Thus, the solution x_{m1} and x_{m2} must be on the circle $f(x_1, x_2)$ and within the region specified by g_1 and g_2.

Plot the above objective function (circles) and the region in which the solution must lie. The results should be made to look like those shown in Figure 6.46. To obtain this result, `fill` will have to be used and applied in a certain order.

6.18 Write a script that creates Figure 5.25 of Exercise 5.22. The fill color is cyan.

6.19 Consider a linear array of $2N$ acoustic sources shown in Figure 6.47 , which are vibrating at a frequency ω. The magnitude of the total acoustic normalized pressure at a distance r from the array that is oriented at an angle θ as shown in Figure 6.47 is[13]

$$P(r, \theta) = \frac{\sin(2Nkd\cos\theta)}{2N\sin(kd\cos\theta)}$$

[13] Junger and Feit, *Sound, Structures, and Their Interactions,* p. 44.

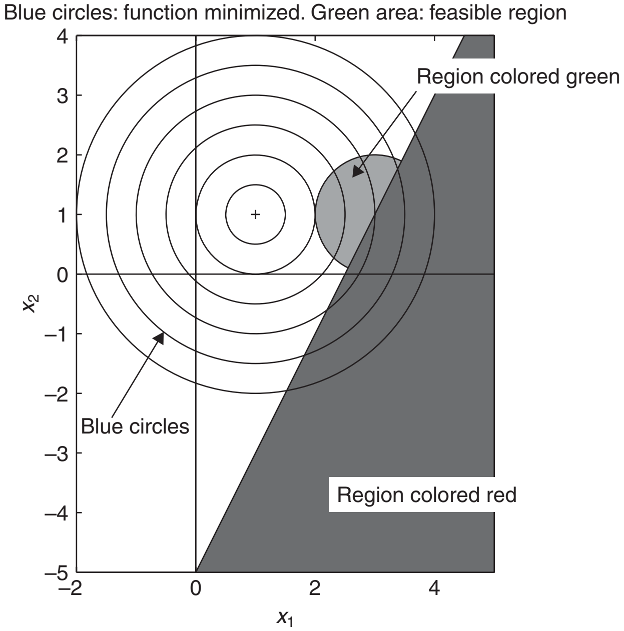


Figure 6.46 Solution to Exercise 6.17.

where $k = 2\pi\omega/c$ is the wave number and c is the wave speed in the medium. Use this equation to replicate Figure 6.48, which is for $N = 2$.

6.20 A Pappus chain is a series of N tangent circles inscribed in the area between three semicircles as shown in Figure 6.49. The radius of the outer circle is R_o

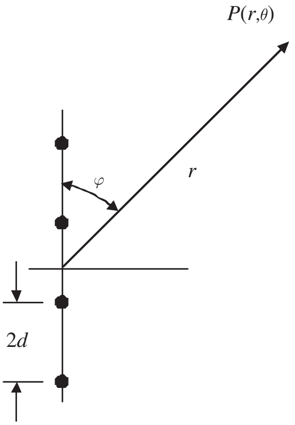
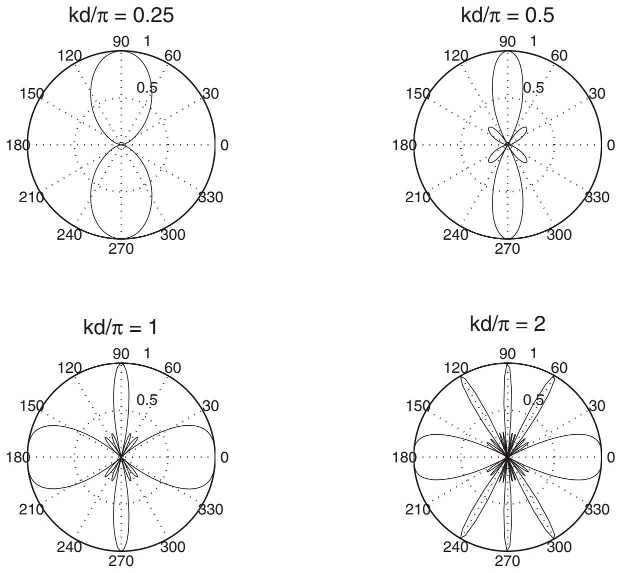


Figure 6.47 Linear array of $2N$ acoustic sources.


kd/π = 0.25

kd/π = 0.5

kd/π = 1

kd/π = 2

Figure 6.48 Radiation pattern of an acoustic array for $N = 2$.

and the radii of the other two circles are R_L and R_R, $R_L > R_R$. The location of the center of the nth circle is[14]

$$
\begin{aligned}
x_n &= \frac{r(1+r)}{2D} \\
y_n &= \frac{nr(1-r)}{D} \\
D &= n^2(1-r)^2 + r
\end{aligned}
$$

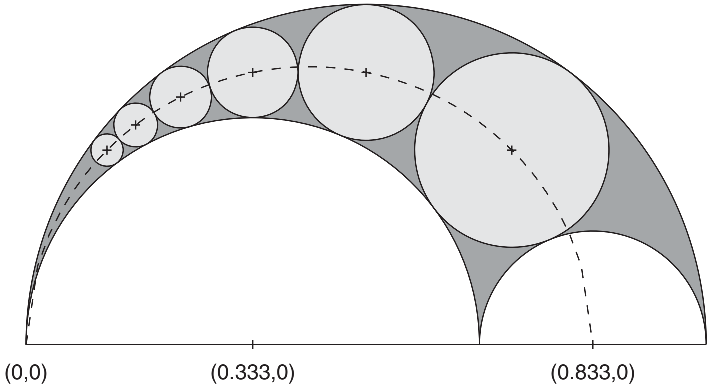


Figure 6.49 Pappus chain.

[14] Weisstein, *CRC Concise Encyclopedia,* pp. 2129–2130.

and its radius is

$$r_n = \frac{r(1 - r)}{2D}$$

where $r = 2R_L/(R_L + R_R)$. The ellipse that connects the centers of the circles is given by

$$\left[\frac{4x - (1 + r)}{1 + r}\right]^2 + \left[\frac{2y}{\sqrt{r}}\right]^2 = 1 \qquad 0 \le x \le (1 + r)/2$$

Replicate Figure 6.49 where $N = 6$, $R_o = 0.5$, $R_L = 1/3$, and $R_R = 1/6$. Color the six circles yellow, their background green; the two smaller semicircles are white.

6.21 The loss factor ratio η of a metal sheet with a damping layer ratio of metal sheet thickness to damping layer thickness ξ is given by[15]

$$\eta = \frac{\lambda\xi\left[3 + 6\xi + 4\xi^2 + 2\lambda\xi^3 + \lambda^2\xi^4\right]}{\left(1 + \lambda\xi\right)\left[1 + 2\lambda\left(2\xi + 3\xi^2 + 2\xi^3\right) + \lambda^2\xi^4\right]}$$

where λ is the ratio of the Young's modulus of the two materials. Use this relation to obtain the results in Figure 6.50.

6.22 A rectangular plate of sides of length a and b is loaded with an in-plane force in a direction parallel to the side of length a. If the plate is simply supported on all four edges, then the value at which the rectangular plate buckles is determined from[16]

$$N_{cr,m} = \left(\frac{m}{\alpha} + \frac{\alpha}{m}\right)^2$$

where $N_{cr,m}$ is the nondimensional buckling coefficient and $\alpha = a/b$. The lowest values at which $N_{cr,m} = N_{cr,m+1}$ is determined from

$$\alpha = \sqrt{m(m + 1)}$$

Using these relations, replicate Figure 6.51.

6.23 The percentage error as a function of radian frequency ω that is created when one connects a device with an output resistance R_g to the input of amplifier that has an input resistance R_i and a shunt capacitance C_i is given by[17]

$$\text{error} = 100\left\{1 - \left[\left(1 + \alpha\right)^2 + \left(\alpha\omega\tau_i\right)^2\right]^{-1/2}\right\} \%$$

where $\alpha = R_g/R_i$ and $\tau_i = R_iC_i$. Using this relation, replicate Figure 6.52.

[15] E. Skudrzyk, *Simple and Complex Vibratory Systems*, Pennsylvania State University Press, University Park, PA, 1968, p.446.

[16] S. Timoshenko and J. M. Gere, *Theory of Elastic Stability*, 2nd ed., McGraw-Hill, NY, 1961, p. 353.

[17] E. B. Magrab, *Environmental Noise Control*, John Wiley & Sons, NY, 1975, p. 96.

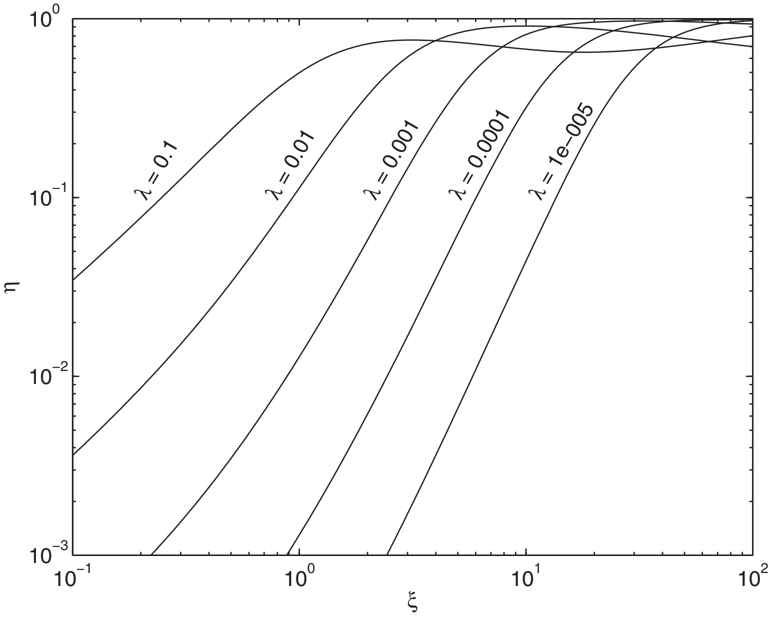


Figure 6.50 Loss factor of a layered metal sheet.

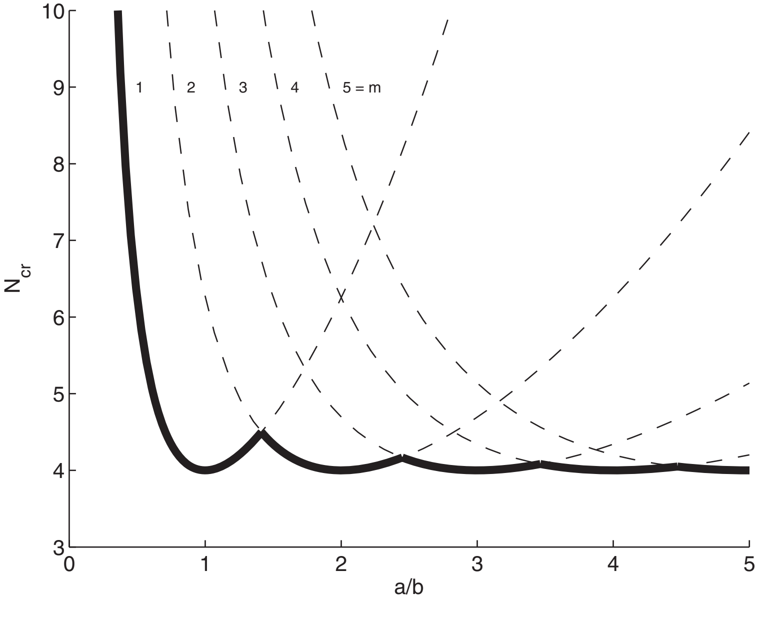


Figure 6.51 Nondimensional buckling coefficient.

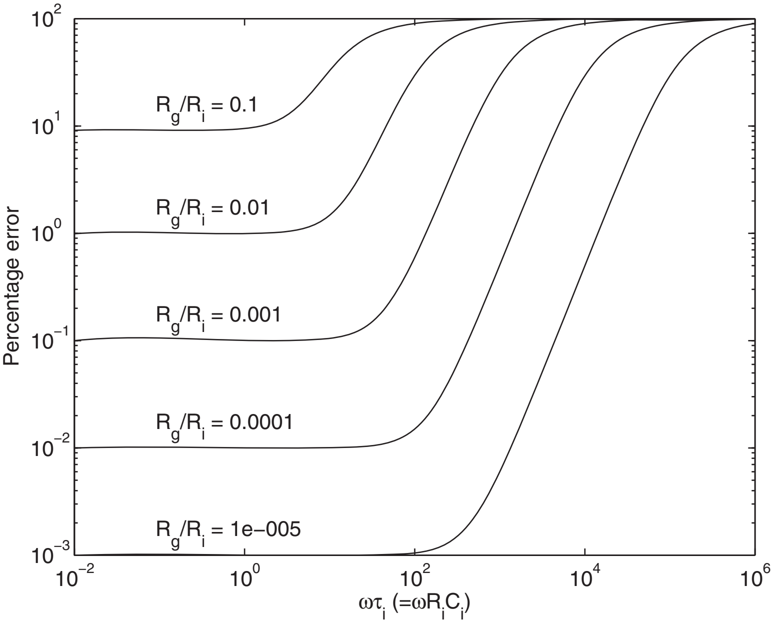


Figure 6.52 Percentage error due to an impedance mismatch.

6.24 The natural frequency coefficients for a thin circular cylindrical elastic shell of radius R, thickness h, and length L that has its ends constrained by a shear diaphragm is given by[18]

$$\Omega^6 - K_2\Omega^4 + K_1\Omega^2 - K_0 = 0$$

where

$$K_2 = 1 + 0.5(3 - \nu)(n^2 + \lambda^2) + k(n^2 + \lambda^2)^2$$
$$K_1 = 0.5(1 - \nu)\left[(3 + 2\nu)\lambda^2 + n^2 + (n^2 + \lambda^2)^2 + \frac{3 - \nu}{1 - \nu}k(n^2 + \lambda^2)^3\right]$$
$$K_0 = 0.5(1 - \nu)\left[(1 - \nu^2)\lambda^4 + k(n^2 + \lambda^2)^4\right]$$

In addition, ν is Poisson's ratio, $n = 0, 1, 2, \ldots,$

$$k = \frac{h^2}{12L^2}, \ \lambda = \frac{m\pi R}{L}, \ \Omega = R\omega\sqrt{\frac{\rho(1 - \nu^2)}{E}}$$

and E is Young's modulus, m is an integer that does not explicitly affect the results, and ω is the radian frequency. When $\nu = 0.3$ and $k = 10^{-5}$, replicate Figure 6.53.

[18] A. Leissa, "Vibrations of Shells," NASA SP-288, 1973, p. 44.

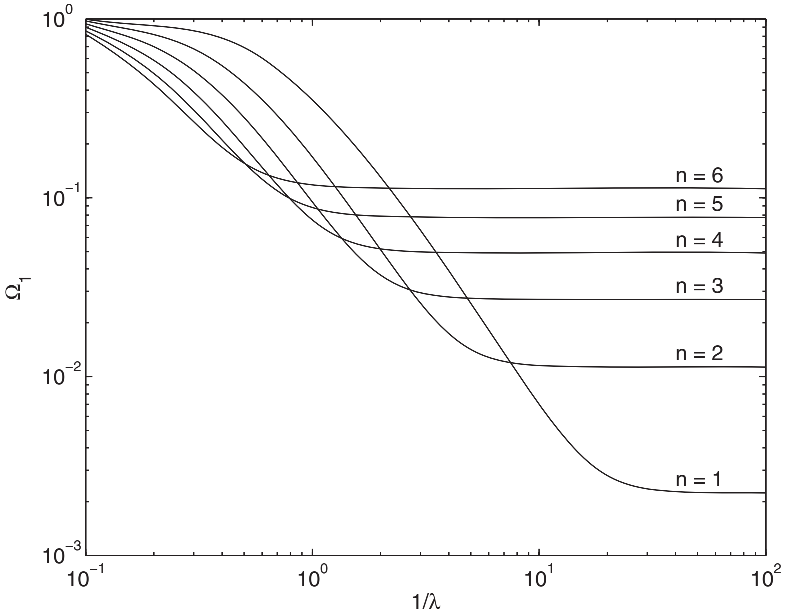


Figure 6.53 Lowest natural frequency coefficient for a cylindrical shell.

6.25 A cam with cycloidal motion moves a flat-faced translating follower up and down by an amount L, where

$$\begin{aligned} L(\varphi) &= r_b + h\left(\frac{\varphi}{\beta} - \frac{1}{2\pi}\sin\left(2\pi\varphi/\beta\right)\right) && 0 \le \varphi \le \beta \\ &= r_b + h - h\left(\frac{\varphi - \beta}{\beta} - \frac{1}{2\pi}\sin\left(2\pi(\varphi - \beta)/\beta\right)\right) && \beta \le \varphi \le 2\beta \\ &= r_b && 2\beta \le \varphi \le 2\pi \end{aligned}$$

and r_b is the base circle of the cam whose center is concentric with the cam's axis of rotation and φ is the angle of rotation of the cam about its axis. If the profile of a cam is described by the radius $R(\varphi)$, then the x and y components of the profile are given by, respectively,

$$R_x = R\cos(\theta + \varphi) \qquad R_y = R\sin(\theta + \varphi)$$

where

$$R = \frac{L}{\cos\theta} \qquad \theta = \tan^{-1}\left(\frac{dL}{Ld\varphi}\right)$$

and

$$\begin{aligned} \frac{dL}{d\varphi} &= \frac{h}{\beta}\left(1 - \cos(2\pi\varphi/\beta)\right) && 0 \le \varphi \le \beta \\ &= -\frac{h}{\beta}\left(1 - \cos\left(2\pi(\varphi - \beta)/\beta\right)\right) && \beta \le \varphi \le 2\beta \\ &= 0 && 2\beta \le \varphi \le 2\pi \end{aligned}$$

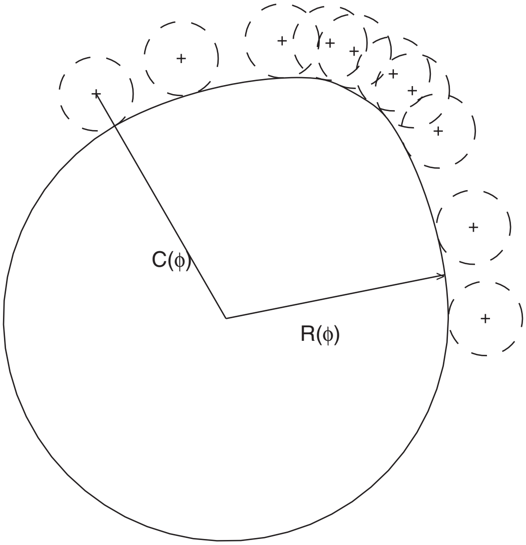


Figure 6.54 Cam profile for cycloidal motion of flat-face follower and several of its cutter's positions.

If the center of a cutter of radius r_c that produces the cam profile is at a radial distance $C(\varphi)$ from the center of rotation of the cam, then the x and y components of $C(\varphi)$ are, respectively,

$$C_x = C\cos(\gamma + \varphi) \qquad C_y = C\sin(\gamma + \varphi)$$

where

$$C = \frac{L + r_c}{\cos\gamma} \qquad \gamma = \tan^{-1}\left(\frac{1}{\left(L + r_c\right)}\frac{dL}{d\varphi}\right)$$

Using these relations, obtain the results shown in Figure 6.54 when $\beta = 60°$, $r_b = 3.0$, and $h = 0.5$. Take forty-six equally spaced points for $0 \le \varphi \le 120°$ and plot the cutter at $\varphi(1{:}5{:}46)$.

7

3D Graphics

Edward B.Magrab

The implementation of a wide selection of three-dimensional plotting capabilities is presented.

7.1 LINES IN 3D

The 3D version of `plot` is

`plot3`(u1, v1, w1, c1, u2, v2, w2, c2, . . .)

where u_j, v_j, and w_j are the x-, y-, and z-coordinates, respectively, of a point. They are scalars, vectors of the same length, matrices of the same order, or expressions that, when evaluated, result in one of these three quantities. The quantity c_j is a string of characters, where one character specifies the color, one character specifies the point characteristics, and where up to two characters specify the line type. See Table 6.2.

To draw a set of n unconnected lines whose end points are (x_{1j}, y_{1j}, z_{1j}) and (x_{2j}, y_{2j}, z_{2j}), $j = 1, 2, \ldots, n$, we create six vectors

$$\begin{aligned} x_j &= [x_{j1}\, x_{j2}\, \ldots\, x_{jn}] \\ y_j &= [y_{j1}\, y_{j2}\, \ldots\, y_{jn}] \qquad j = 1, 2 \\ z_j &= [z_{j1}\, z_{j2}\, \ldots\, z_{jn}] \end{aligned}$$

Then, the plot3 instruction is

```
x1 = [ . . . ];   x2 = [ . . . ];
y1 = [ . . . ];   y2 = [ . . . ];
z1 = [ . . . ];   z2 = [ . . . ];
plot3([x1; x2], [y1; y2], [z1; z2])
```

where [x1; x2], [y1; y2], and [z1; z2] are each ($2 \times n$) matrices. This is the 3D counterpart of the 2D procedure used with plot.

All annotation procedures discussed for 2D drawings in Section 6.3 are applicable to the 3D curve- and surface-generating functions, except that the arguments of text become

```
text(x, y, z, s)
```

where s is a string and

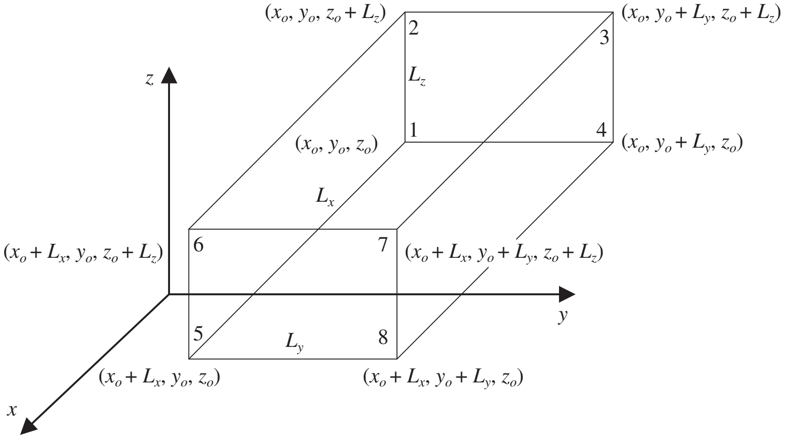
```
zlabel(s1)
```

is used to label the z-axis and $s1$ is a string.

Example 7.1 Drawing wire-frame boxes

Consider a box of dimensions $L_x \times L_y \times L_z$ as shown in Figure 7.1. We create a function M file called **BoxPlot3** to draw the four edges of each of the six surfaces of the box and then use **BoxPlot3** to draw several boxes. The location and orientation of the box are determined by the coordinates of its two diagonally opposed corners: **P**(x_o, y_o, z_o) and **P**$(x_o + L_x, y_o + L_y, z_o + L_z)$. Hence, the function M file is

```
function BoxPlot3(xo, yo, zo, Lx, Ly, Lz)
x = [x0   x0      x0      x0      x0+Lx   x0+Lx   x0+Lx   x0+Lx];
y = [y0   y0      y0+Ly   y0+Ly   y0      y0      y0+Ly   y0+Ly];
z = [z0   z0+Lz   z0+Lz   z0      z0      z0+Lz   z0+Lz   z0     ];
index = zeros(6, 5);
index(1,:) = [1 2 3 4 1];
```



$(x_o, y_o, z_o + L_z)$
$(x_o, y_o + L_y, z_o + L_z)$
2
3
z
L_z
1
4
(x_o, y_o, z_o)
$(x_o, y_o + L_y, z_o)$
L_x
6
7
$(x_o + L_x, y_o, z_o + L_z)$
$(x_o + L_x, y_o + L_y, z_o + L_z)$
y
5
L_y
8
$(x_o + L_x, y_o, z_o)$
$(x_o + L_x, y_o + L_y, z_o)$
x

Figure 7.1 Coordinates of a box and the numbering of its corners.

```
index(2,:) = [5 6 7 8 5];
index(3,:) = [1 2 6 5 1];
index(4,:) = [4 3 7 8 4];
index(5,:) = [2 6 7 3 2];
index(6,:) = [1 5 8 4 1];
for k = 1:6
  plot3(x(index(k,:)), y(index(k,:)), z(index(k,:)), 'k')
  hold on
end
```

We now use **BoxPlot3** to generate three boxes with the following dimensions and the coordinates (x_o, y_o, z_o):

Box #1

Size: $3 \times 5 \times 7$
Location: (1, 1, 1)

Box #2

Size: $4 \times 5 \times 1$
Location: (3, 4, 5)

Box #3

Size: $1 \times 1 \times 1$
Location: (4.5, 5.5, 6)

The program to create and display these three wire-frame boxes is

BoxPlot3(1, 1, 1, 3, 5, 7)
BoxPlot3(4, 6, 8, 4, 5, 1)
BoxPlot3(8, 11, 9, 1, 1, 1)

which upon execution gives Figure 7.2.

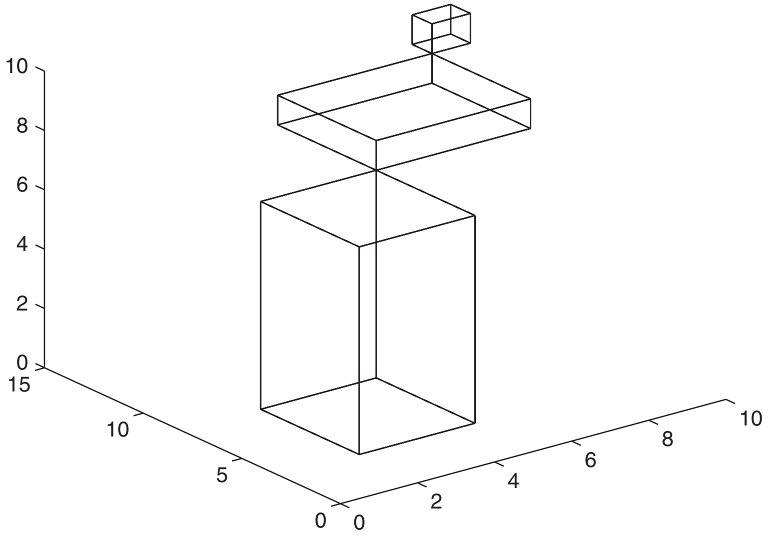


Figure 7.2 Three wire-frame boxes.

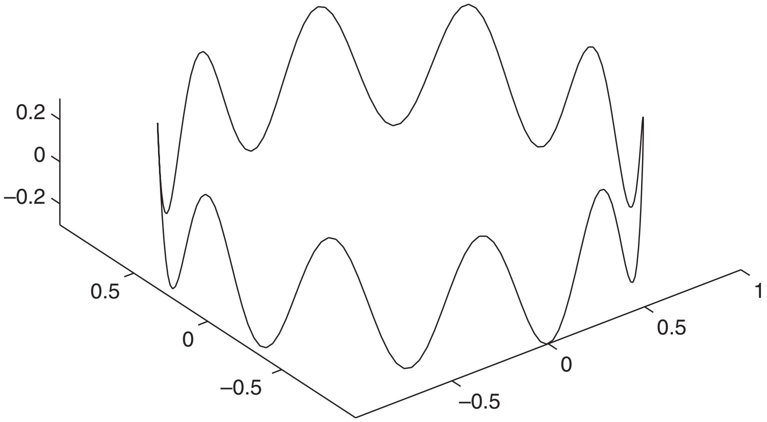


Figure 7.3 Sine wave drawn on a cylindrical surface.

Example 7.2 Sine wave drawn on the surface of a cylinder

The coordinates of a sine wave on the surface of a cylinder are obtained from the following relations[1]

$$x = b\cos(t)$$
$$y = b\sin(t)$$
$$z = c\cos(at)$$

If we assume that $a = 10.0$, $b = 1.0$, $c = 0.3$, and $0 \le t \le 2\pi$, then the script is

```
t = linspace(0, 2*pi, 200);
a = 10;   b = 1.0;   c = 0.3;
x = b*cos(t);
y = b*sin(t);
z = c*cos(a*t);
plot3(x, y, z, 'k-')
axis equal
```

The execution of this script results in Figure 7.3.

7.2 SURFACES

A set of 3D plotting functions is available to create surfaces, contours, and variations and specialization of these basic forms. A surface is defined by the expression

$$z = f(x, y)$$

where x and y are the coordinates in the xy-plane and z is the resulting height. The basic surface plotting functions are

```
surf(x, y, z)
```

and

```
mesh(x, y, z)
```

[1] von Seggern, *CRC Standard Curves and Surfaces*.

where the x, y, and z are the coordinates of the points on the surface. The function `surf` draws a surface composed of colored patches, whereas `mesh` draws white surface patches that are defined by their boundary. In `surf`, the colors of the patches are determined by the magnitude of z, whereas it is the colors of the lines in `mesh` that are determined by the magnitude of z. In both `surf` and `mesh`, hidden lines are removed. The hidden lines can be displayed in `mesh` by using

```
hidden off
```

We shall illustrate the use of these functions, and several other functions, with the plotting of the surface created by

$$z(x, y) = x^4 + 3x^2 + y^2 - 2x - 2y - 2x^2y + 6$$

over the range $-3 < x < 3$ and $-3 < y < 13$. We shall place the generation of the x-, y-, and z-coordinate values in a function M file called **SurfExample**. Thus,

```
function [x, y, z] = SurfExample
x1 = linspace(-3, 3, 15);
y1 = linspace(-3, 13, 17);
[x, y] = meshgrid(x1, y1);
z = x.^4+3*x.^2-2*x+6-2*y.*x.^2+y.^2-2*y;
```

Using **SurfExample**, the differences between `mesh`, `surf`, and `mesh` without using the hidden line removal option are shown in Table 7.1.

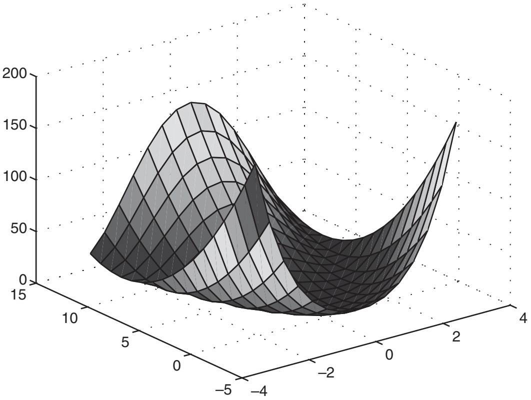
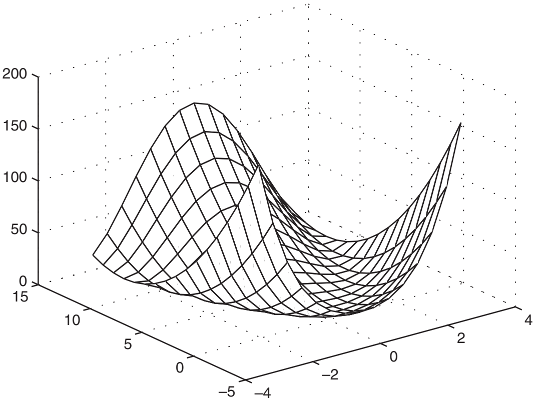
Combining Surfaces and Lines

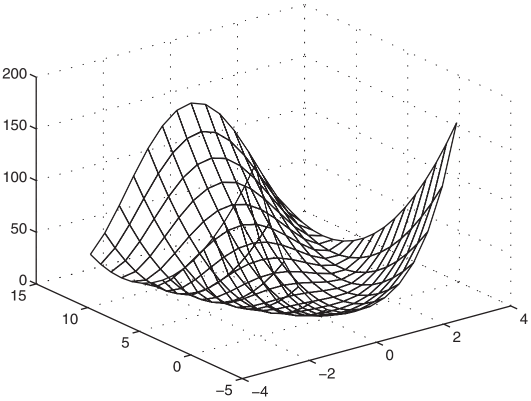
To show how one can combine 3D plotting functions to draw multiple surfaces and multiple lines, we create two function M files. The first function, called **Corners**, draws four lines connecting the corners of the surface generated by **SurfExample** to the xy-plane passing through $z = 0$. The second function, called **Disc**, creates a circular disc that intersects this surface at $z_o = 80$, has a radius of 10 units, and has its center at (0, 5). The coordinates of the corners of $z(x, y)$ are: $(-3, -3, z(-3,-3))$, $(-3, 13, z(-3, 13))$, $(3, 13, z(3, 13))$, and $(3, -3, z(3, -3))$. The script that draws the surface, the disc, and the lines to the corners of the surface is

```
function SurfDisc
[x, y, z] = SurfExample;
surf(x, y, z);
Disc(10, 80)
Corners
```

```
function Corners
xc = [-3, -3, 3, 3];
yc = [-3, 13, 13, -3];
```

TABLE 7.1 Illustration of the Difference between `surf`, `mesh`, and `mesh` without Hidden Line Removal

Plotting function	Script	Graph
`surf`	[x, y, z] = **SurfExample**; `surf`(x, y, z)	
`mesh`	[x, y, z] = **SurfExample**; `mesh`(x, y, z)	

`mesh`
`hidden off`
	[x, y, z] = **SurfExample**;
	`mesh`(x, y, z)
	`hidden off`
		

```
zc = xc.^4+3*xc.^2-2*xc+6-2*yc.*xc.^2+yc.^2-2*yc;
hold on
plot3([xc; xc], [yc; yc], [zeros(1, 4); zc], 'k-')

function Disc(R, zo)
r = linspace(0, R, 12);
theta = linspace(0, 2*pi, 50);
x = cos(theta')*r;
y = 5 + sin(theta')*r;
hold on
z = repmat(zo, size(x));
surf(x, y, z)
```

The execution of **SurfDisc** produces Figure 7.4a. The fourth line is not visible in this view.

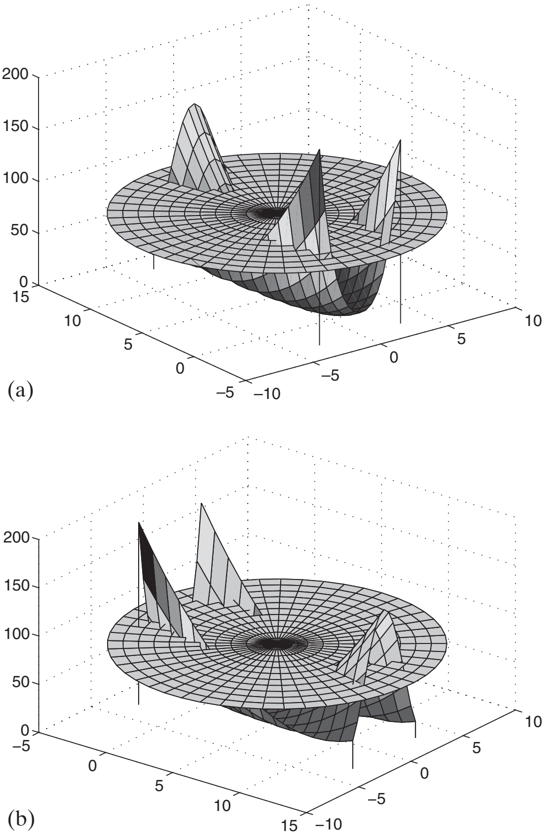


Figure 7.4 (a) Surface with lines drawn to its corners and intersecting a disc. (b) Figure in (a) redrawn using `axis ij`.

Reversing One Axis

The portions of the disc, lines, and surface that were not visible in Figure 7.4a can be viewed by reversing one of the display axes. This reversal is implemented with

```
axis ij
```

When this command is appended to **SurfDisc** and the program run, we obtain Figure 7.4b. Notice that the left-hand axis now goes from −5 to 15.

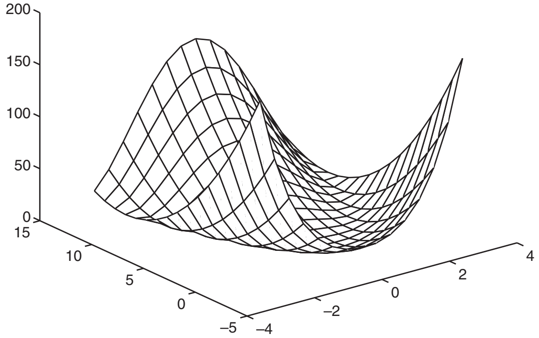
Altering Graph Appearance

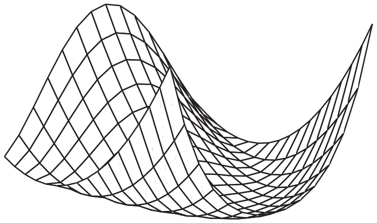
Several functions that can be used in various combinations to alter the appearance of a surface plot are

```
box on    or  box off
grid on   or  grid off
axis on   or  axis off
```

The function `box on` only draws a box if `axis on` has been selected. Some examples of combinations of these functions are given in Table 7.2.

TABLE 7.2 Illustration of `box`, `grid`, and `axis`

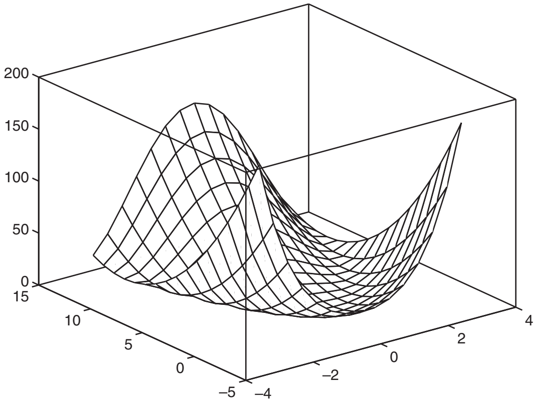
Plotting function	Script	Graph
`grid off`	[x, y, z] = **SurfExample**; mesh(x, y, z) grid off	

`axis off`
`grid off`
	[x, y, z] = **SurfExample**;
	mesh(x, y, z)
	axis off
	grid off
		

(*Continued*)

TABLE 7.2 Continued

Plotting function	Script	Graph

`box on`
`axis on`
`grid off`
	[x, y, z] = **SurfExample**;
	mesh(x, y, z)
	`box on`
	`axis on`
	`grid off`
		

As mentioned previously, the colors of the patches are generated automatically with `surf` according to their *z*-value. Similarly, the colors of the lines generated automatically with `mesh` vary according to their *z*-value. The colors of either the patches or the lines can be changed to a uniform color by using

`colormap(c)`

where *c* is a three-element vector, each of whose value varies between 0 and 1. The first element corresponds to the intensity of red, the second to the intensity of green, and the third to the intensity of blue. Some commonly used combinations are listed in Table 7.3. In addition, there are thirteen preassigned color maps. They are: *jet*, *hsv*, *hot*, *cool*, *spring*, *summer*, *autumn*, *winter*, *gray*, *bone*, *copper*, *pink*, and *lines*. Visiting *colormap* in the *Help* file will provide the color spectrum that each of these keywords produces. To use any of these predefined color maps, *c* is replaced with one of these keywords (without single quotes).

TABLE 7.3 Some Values of the Color Vector Used in `colormap(c)`

c	Color
[0 0 0]	black
[1 1 1]	white
[1 0 0]	red
[0 1 0]	green
[0 0 1]	blue
[1 1 0]	yellow
[1 0 1]	magenta
[0 1 1]	cyan
[0.5 0.5 0.5]	gray

Four additional ways to visually enhance the surface generated by **SurfExample** are illustrated in Table 7.4.

Contour Plots

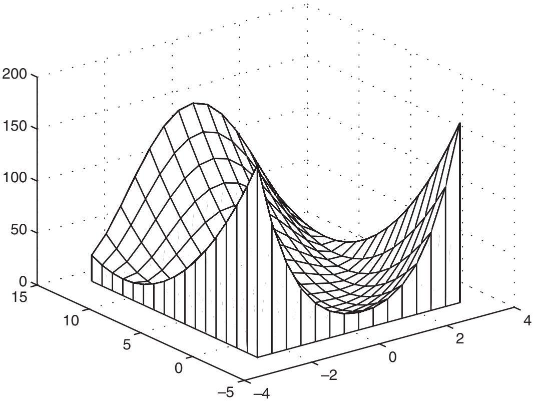
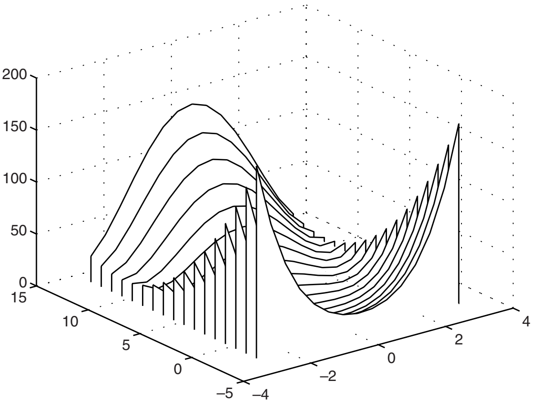
Surfaces can also be transformed into various contour plots, which are plots of the curves formed by the intersection of the surface and a plane parallel to the *xy*-plane at given values of *z*. The functions

`surfc`(x, y, z)

and

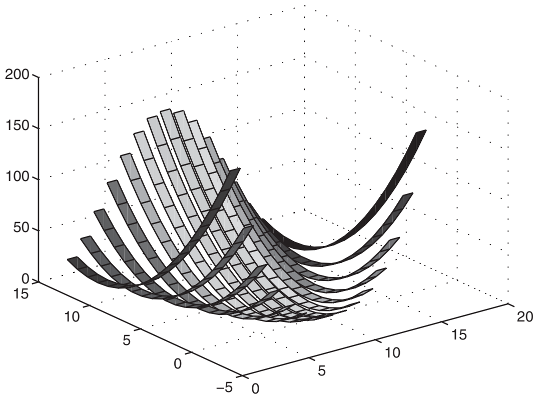
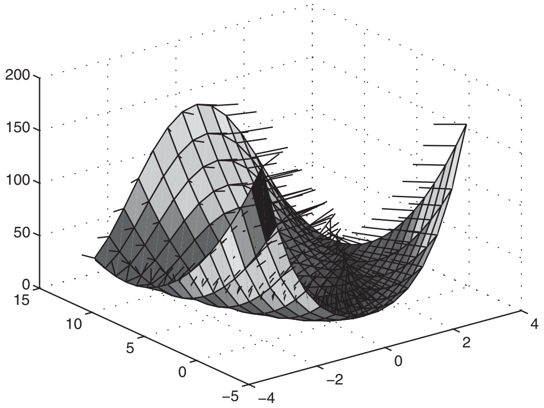
`meshc`(x, y, z)

TABLE 7.4 Illustration of `meshz`, `waterfall`, `ribbon`, and `surfnorm`

Plotting function	Script	Graph
`meshz`	[x, y, z] = **SurfExample**; `meshz`(x, y, z)	
`waterfall`	[x, y, z] = **SurfExample**; `waterfall`(x, y, z)	

(*Continued*)

TABLE 7.4 Continued

Plotting function	Script	Graph
`ribbon`	[x, y, z] = **SurfExample**; `ribbon`(y, z)	
`surfnorm`	[x, y, z] = **SurfExample**; `surfnorm`(x, y, z)	

create surfaces with contours projected beneath the surfaces. The quantities x, y, and z are the values of the coordinates of points that define the surface. These two functions are illustrated in Table 7.5.

Various contour plots without the surfaces can be created, either with labels or without labels. The function

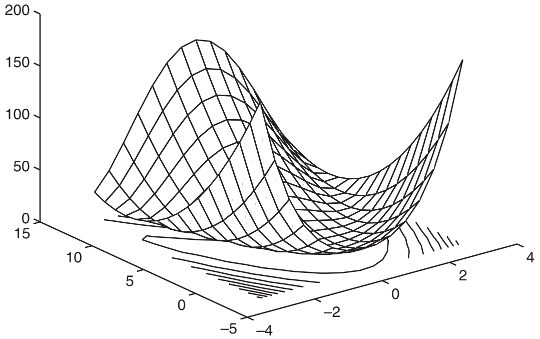
```
contour(x, y, z, v)
```

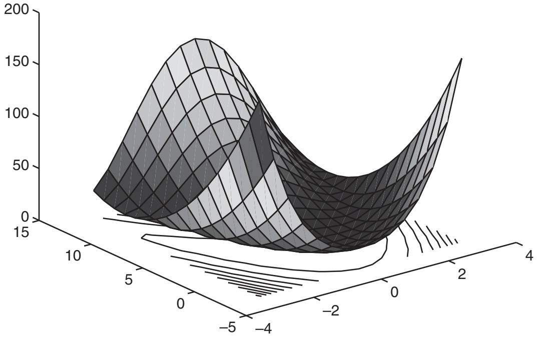
creates a 2D contour plot. The values of x, y, and z are the coordinates of the points that define the surface and v, if a scalar, is the number of contours to be displayed and, if a vector of values, the contours of the surface at those values of z. The use of v is optional. If the contour plot is to be labeled, then we use the following pair of functions:

```
[C, h] = contour(x, y, z, v)
clabel(C, h, v)
```

TABLE 7.5 Illustration of `meshc` and `surfc`

Plotting function	Script	Graph

`meshc`	[x, y, z] = **SurfExample**;
	`meshc(x, y, z)`
	`grid off`
		

`surfc`	[x, y, z] = **SurfExample**;
	`surfc(x, y, z)`
	`grid off`
		

Several ways in which these two functions can be used are illustrated in Table 7.6.

Two additional contour plots are available. The first is

```
contour3(x, y, z, v)
```

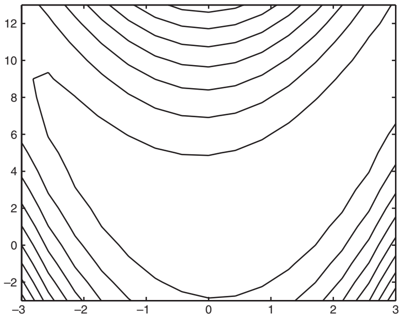
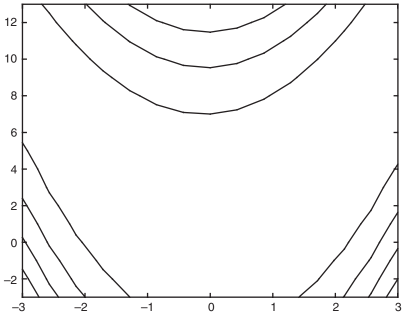
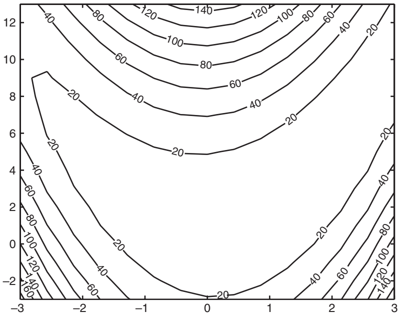
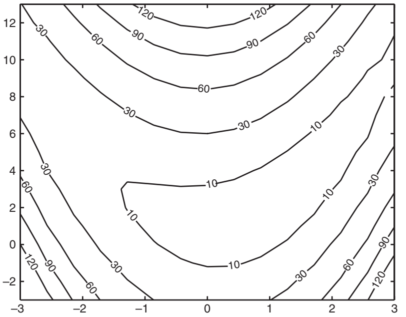
which displays contours of the surface in 3D. The values of x, y, and z are the coordinates of points that define the surface and v, if a scalar, is the number of contours to be displayed and, if a vector of values, the contours of the surface at those values of z. The use of v is optional. If the contour plot is to be labeled, then we use the following pair of functions:

```
[C, h] = contour3(x, y, z, v)
clabel(C, h, v)
```

The second contour function is

```
contourf(x, y, z, v)
```

TABLE 7.6 Illustration of `contour` and `clabel`

Plotting function	Script	Graph
`contour`	[x, y, z] = **SurfExample**; `contour`(x, y, z)	
`contour`	[x, y, z] = **SurfExample**; `contour`(x, y, z, 4)	
`contour` `clabel`	[x, y, z] = **SurfExample**; [C, h] = `contour`(x, y, z); `clabel`(C, h)	
`contour` `clabel`	[x, y, z] = **SurfExample**; v= [10, 30:30:120]; [C, h] = `contour`(x, y, z, v); `clabel`(C, h, v)	

which fills the region between the 2D contours with different colors. The values of the colors can be identified using

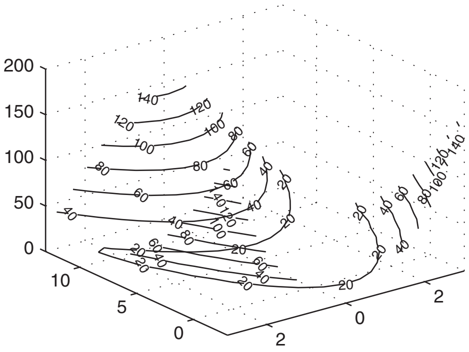
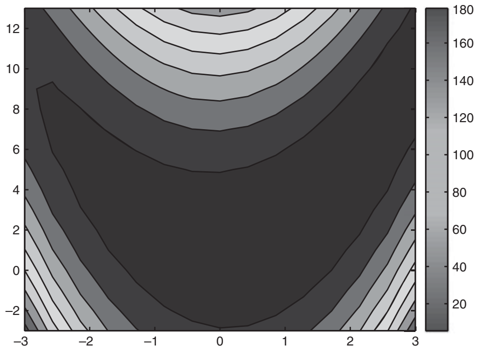
```
colorbar(s)
```

which places a bar of colors and their corresponding numerical values adjacent to the figure. The quantity *s* is a string equal to either 'horiz' or 'vert' to indicate the orientation of the bar. The default value is 'vert'. These functions are illustrated in Table 7.7.

The properties of the lines and numbering in contour can be altered in a manner similar to what was used for `plot`. For example, to enlarge the contour labels created by `contour` to 14 points and for all the contour lines to be black, we employ the following steps:

```
[x, y, z] = SurfExample;
v = [10, 30:30:120];
[C, h] = contour(x, y, z, v);
clabel(C, h, v, 'fontsize', 14);
set(h, 'LineColor', 'k')
```

TABLE 7.7 Illustration of `contour3`, `contourf`, and `colorbar`

Plotting function	Script	Graph
`contour3` `clabel`	[x, y, z] = **SurfExample**; [C, h] = `contour3`(x, y, z); `clabel`(C, h)	
`contourf` `colorbar`	[x, y, z] = **SurfExample**; `contourf`(x, y, z) `colorbar`	

On the other hand, to enlarge the contour labels created by `contour3` to 14 points and for all the contour lines to be black, we employ the following steps:

```
[x, y, z] = SurfExample;
v = [10, 30:30:120];
[C, h] = contour3(x, y, z, v);
clabel(C, h, v, 'fontsize', 14)
set(h, 'EdgeColor', 'k')
```

since `contour3` uses patches and `contour` uses lines.

Generation of Cylindrical, Spherical, and Ellipsoidal Surfaces

One can use a 2D curve as a generator to create surfaces of revolution by using

```
[x, y, z] = cylinder(r, n)
```

which returns the x-, y-, and z-coordinates of a cylindrical surface using the vector $r = r(z)$ to define a profile curve. The function `cylinder` treats each element in r as a radius at n equally spaced points around its circumference. If n is omitted, MATLAB uses a value of 20. If r is omitted, then a cylinder centered at the origin of length 1 and radius 1 is created with the bottom of the cylinder resting on the xy-plane.

To illustrate `cylinder`, consider the curve

$$r = 1.1 + \sin(z) \qquad 0 \le z \le 2\pi$$

which is rotated 360° about the z-axis. We take twenty-six equally spaced increments in the z-direction as shown in Figure 7.5a and sixteen equally spaced

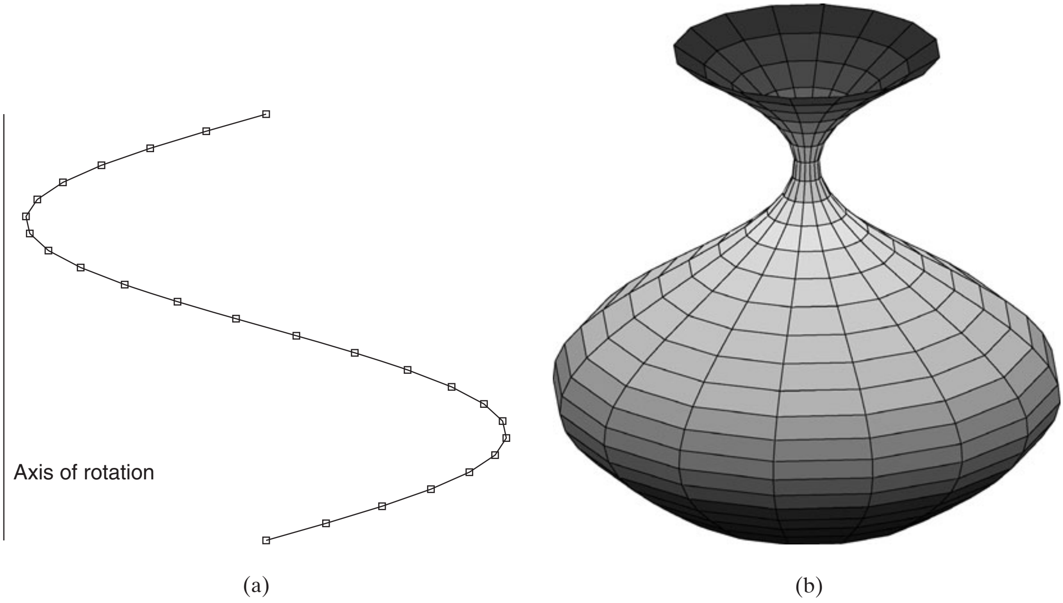


Figure 7.5 Application of `cylinder`: (a) profile $r(z)$. (b) Resulting surface of revolution.

increments in the circumferential direction. The script to plot a cylindrical surface with this profile is

```
zz = linspace(0, 2*pi, 26);
[x, y, z] = cylinder(1.1+sin(zz), 16);
surf(x, y, z)
axis off
```

which upon execution gives the results shown in Figure 7.5b.

To create a sphere, one can use

```
[x, y, z] = sphere(n);
axis equal
surf(x, y, z)
```

where n is the number of $n \times n$ elements that will comprise the sphere of radius 1 centered at the origin. If n is omitted, then $n = 20$.

To create an ellipsoid, we use

```
[x, y, z] = ellipsoid(xc, yc, zc, xr, yr, zr, n);
axis equal
surf(x, y, z)
```

which is centered at (xc, yc, zc) and has semi-axis lengths in the x-, y-, and z-directions, respectively, of xr, yr, and zr. In addition, n is the number of $n \times n$ elements that will comprise the ellipsoid. If n is omitted, then $n = 20$.

Viewing Angle

In Figure 7.5b, the viewing angles are the default values. There are instances when one wants to change the default viewing angle of the 3D image because it does not display the features of interest, several different views are to be displayed using `subplot`, or one wants to explore the surface from many different views before deciding on the final orientation. To determine the azimuth and elevation angles of the view, we use

```
[a, e] = view
```

where a is the azimuth and e the elevation. To orient the object, one depresses the *Rotate 3D* icon in the figure window and orients the object until a satisfactory orientation is obtained. Upon typing the above expression in the command window, the values of the azimuth and elevation angles will be displayed. These values are recorded and entered in the expression

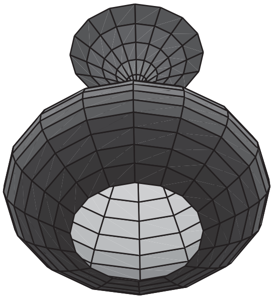
```
view(an, en)
```

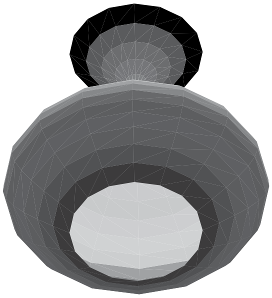
to create the desired orientation the next time that the script is executed. In this expression, *an* and *en* are the numerical values of a and e taken from the command window.

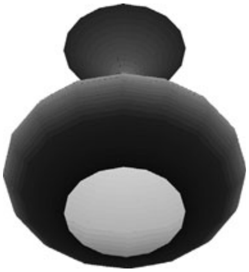
Using this procedure with the object shown in Figure 7.5b, we find that an orientation that produces the desired results is one where $a = -88.5°$ and $e = -48°$.

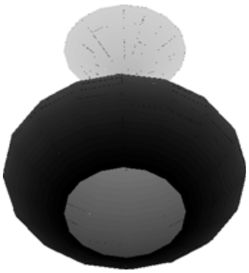
TABLE 7.8 Illustration of view, shading, and colormap

Plotting function	Script	Graph

view
shading faceted
	zz = linspace(0, 2*pi, 26);
	r = 1+sin(zz);
	[x, y, z] = cylinder(r, 16);
	surf(x, y, z)
	view(-88.5, -48)
	shading faceted
	axis off vis3d
		

view
shading flat
	zz = linspace(0, 2*pi, 26);
	r = 1+sin(zz);
	[x, y, z] = cylinder(r, 16);
	surf(x, y, z)
	view(-88.5, -48)
	shading flat
	axis off vis3d
		

view
shading interp
	zz = linspace(0, 2*pi, 26);
	r = 1+sin(zz);
	[x, y, z] = cylinder(r, 16);
	surf(x, y, z)
	view(-88.5, -48)
	shading interp
	axis off vis3d
		

view
shading interp
colormap
	zz = linspace(0, 2*pi, 26);
	r = 1+sin(zz);
	[x, y, z] = cylinder(r, 16);
	surf(x, y, z)
	view(-88.5, -48)
	shading interp
	colormap(copper)
	axis off vis3d
		

Then, the script becomes that shown in Table 7.8. Also shown in the table is the result of its execution.

When one uses `view`, the default situation is for the object to be stretched to fill the plotting area. This can be overridden by using

```
axis vis3d
```

This keyword *vis3d* freezes the aspect ratio properties so that the figure does not get distorted after each change in viewing angle.

Shading

The surfaces created with `surf` have used the default shading property called *faceted*. The function that changes the shading is

```
shading s
```

where *s* is equal to *faceted*, *flat*, or *interp*. The results obtained from using these shading options are shown in Table 7.8.

Filling Polygons

Polygons whose vertices of their connected line segments are at the coordinate locations (x_n, y_n, z_n) can have their interior regions filled by using

```
fill3(x, y, x, c)
```

where x, y, and z are arrays of the same length that represent the end points of the lines that form a polygon. The string c is the color of the fill given by one of the letters appearing in the second column of Table 6.2.

Transparency

The surfaces created with `surf` can have their opaqueness altered using `set` and assigning a numerical value to the keyword 'FaceAlpha'. The effect of this keyword on the resulting surface is dependent on the type of shading chosen. To illustrate the use of this transparency option, we create a function that generates the numerical values for the surface given by

$$\begin{aligned} x &= a^v \cos v(1 + \cos u) \\ y &= -a^v \sin v(1 + \cos u) \\ z &= -ba^v(1 + \sin u) \end{aligned}$$

If we assume that $a = 1.13$ and $b = 1.14$, then the function M file for this surface is

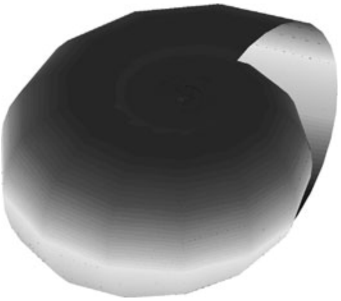
```
function [x, y, z] = Transparency
a = 1.13;    b = 1.14;
uu = linspace(0, 2*pi, 30);
vv = linspace(-15, 6, 45);
```

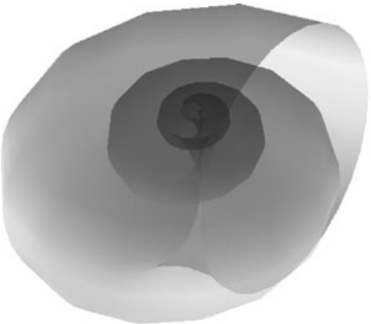
```
[u, v] = meshgrid(uu, vv);
x = a.^v.*cos(v).*(1+cos(u));
y = -a.^v.*sin(v).*(1+cos(u));
z = -b*a.^v.*(1+sin(u));
```

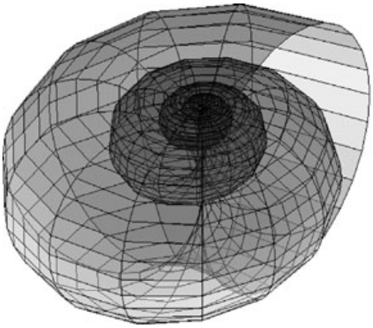
The results obtained from using the transparency option are shown in Table 7.9.

We now present several additional examples of the use of 3D plotting functions.

TABLE 7.9 Illustration of Surface Transparency

Plotting function	Script	Graph
shading interp	[x, y, z] = **Transparency**; surf(x, y, z) shading interp axis vis3d off equal view([-35 38])	

shading interp
FaceAlpha = 0.4
	[x, y, z] = **Transparency**;
	h = surf(x, y, z);
	set(h, 'FaceAlpha', 0.4)
	shading interp
	axis vis3d off equal
	view([-35 38])
		

FaceAlpha = 0.4	[x, y, z] = **Transparency**;
	h = surf(x, y, z);
	set(h, 'FaceAlpha', 0.4)
	axis vis3d off equal
	view([-35 38])
		

Example 7.3 Drawing wire-frame boxes: coloring the box surfaces

We revisit Example 7.1 and modify the function M file **BoxPlot3** so that all of the six surfaces represented by the rectangles are each filled with a different color. This modification entails using `fill3`. The revised **BoxPlot3** is renamed **BoxPlot3C** and becomes

```
function BoxPlot3C(x0, y0, z0, Lx, Ly, Lz, w)
% w = 0, wire frame; w = 1, rectangles are colored
x = [x0   x0       x0       x0       x0+Lx   x0+Lx   x0+Lx     x0+Lx];
y = [y0   y0       y0+Ly    y0+Ly    y0      y0      y0+Ly     y0+Ly ];
z = [z0   z0+Lz    z0+Lz    z0       z0      z0+Lz   z0+Lz     z0      ];
index = zeros(6,5);
index(1,:) = [1 2 3 4 1];
index(2,:) = [5 6 7 8 5];
index(3,:) = [1 2 6 5 1];
index(4,:) = [4 3 7 8 4];
index(5,:) = [2 6 7 3 2];
index(6,:) = [1 5 8 4 1];
c = 'rgbcmy';
for k = 1:6
  if w~=0
    fill3(x(index(k,:)), y(index(k,:)), z(index(k,:)), c(k))
  else
    plot3(x(index(k,:)), y(index(k,:)), z(index(k,:)), 'k')
  end
  hold on
end
```

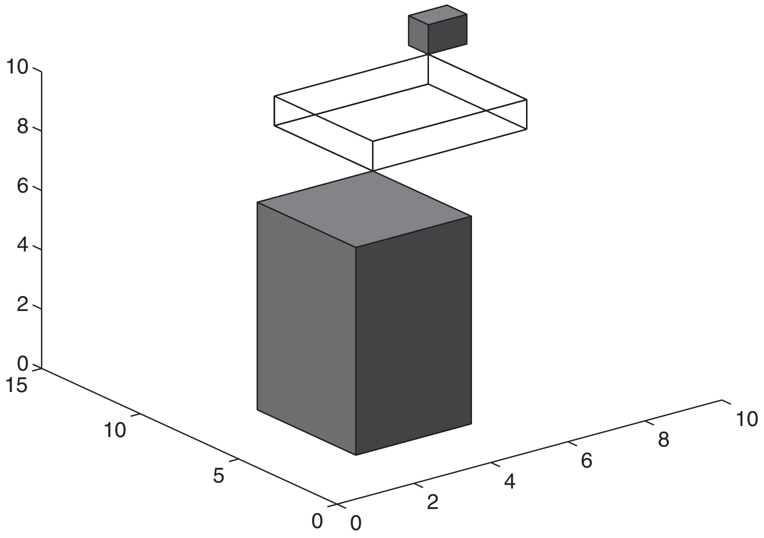


Figure 7.6 Use of `fill3` on two of the wire-frame boxes in Figure 7.2.

Thus, if we use the same coordinate values that were used in Example 7.1, we have

BoxPlot3C(1, 1, 1, 3, 5, 7, 1)
BoxPlot3C(4, 6, 8, 4, 5, 1, 0)
BoxPlot3C(8, 11, 9, 1, 1, 1, 1)

The execution of these statements results in Figure 7.6.

Example 7.4 Intersection of a cylinder and a sphere and the highlighting of their intersection

The curve that results from the intersection of a sphere of radius $2a$ centered at the origin and a circular cylinder of radius a centered at $(a, 0)$ is given by the parametric equations[2]

$$\begin{aligned} x &= a(1 + \cos\varphi) \\ y &= a\sin\varphi \\ z &= 2a\sin(\varphi/2) \end{aligned}$$

where $0 \le \varphi \le 4\pi$. We shall create a graph of the intersecting sphere and cylinder and superimpose on these two objects a yellow spatial curve given by the equations above. It will be seen that this curve indeed describes this intersection. To create a sphere of radius $2a$, we must multiply each of the coordinates from the output of `sphere` by $2a$. The coordinates that are output from `cylinder` must be altered as follows: $x \rightarrow ax + a$, $y \rightarrow ay$, and $z \rightarrow 4az - 2a$. We assume that $a = 1$. The script is

```
a = 1;
[xs, ys, zs] = sphere(30);
```

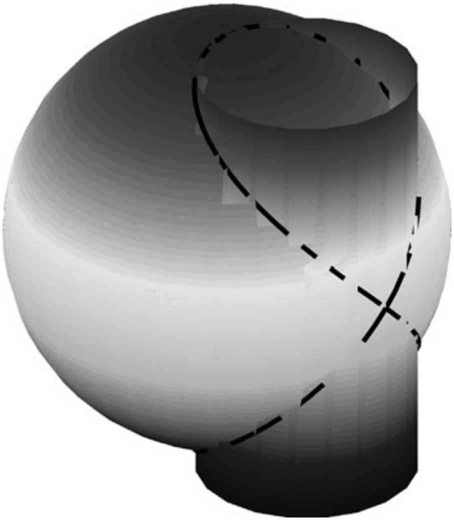


Figure 7.7 Intersection of a sphere and a cylinder.

[2] Weisstein, *CRC Concise Encyclopedia of Mathematics,* p. 3159.

```
surf(2*a*xs, 2*a*ys, 2*a*zs)
hold on
[x, y, z] = cylinder;
surf(a*x+a, a*y, 4*a*z-2*a)
shading interp
t = linspace(0, 4*pi, 100);
x = a*(1+cos(t));
y = a*sin(t);
z = 2*a*sin(t/2);
plot3(x, y, z, 'y-', 'Linewidth', 2.5);
axis equal off
view([45, 30])
```

The execution of this script gives the results that are shown in Figure 7.7.

Example 7.5 Natural frequencies of a beam hinged at both ends and restrained by a spring at an interior point

Consider the thin elastic beam discussed in Example 5.20. If we attach to this beam a linear spring of constant k_s at $\eta = \eta_1, 0 < \eta_1 < 1$, then the governing equation becomes

$$\frac{d^4w}{d\eta^4} + K_s w\delta(\eta - \eta_1) - \Omega^4 w = 0 \qquad 0 \le \eta \le 1$$

where η and Ω are given in Example 5.20, $\delta(\eta)$ is the delta function, and $K_s = k_s L^3/EI$ is a nondimensional quantity. The solution of this equation subject to the boundary conditions for a beam hinged at both ends results in the following characteristic equation from which the natural frequency coefficients Ω_n can be determined:[3]

$$K_s\left[C_{1n}T(\Omega_n\eta_1) + C_{2n}R(\Omega_n\eta_1)\right] + \Omega_n^3 = 0$$

where

$$C_{1n} = \frac{T(\Omega_n)T(\Omega_n[1 - \eta_1]) - R(\Omega_n)R(\Omega_n[1 - \eta_1])}{R^2(\Omega_n) - T^2(\Omega_n)}$$

$$C_{2n} = \frac{T(\Omega_n)R(\Omega_n[1 - \eta_1]) - R(\Omega_n)T(\Omega_n[1 - \eta_1])}{R^2(\Omega_n) - T^2(\Omega_n)}$$

and

$$R(x) = 0.5(\sinh x + \sin x)$$

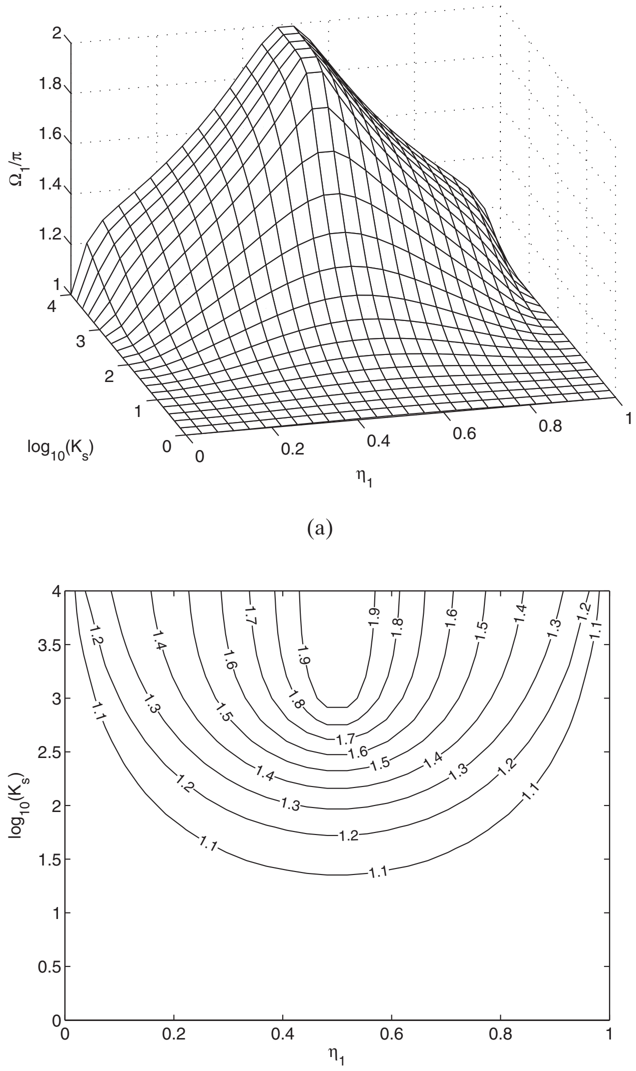
$$T(x) = 0.5(\sinh x - \sin x)$$

[3] Balachandran and Magrab, *Vibrations*, p. 597.

There are two parameters that are of interest: η_1 and K_s. We shall first generate a surface of the lowest natural frequency coefficient Ω_1 as a function of η_1 and $\log_{10}(K_s)$. Then we shall use these same natural frequency coefficients to create a contour plot of $\log_{10}(K_s)$ versus η_1 for several values of Ω_1/π. The program is

```
function Example7_5
Neta1 = 28; NKs = 21; Kend = 4;
Ks = logspace(0, Kend, NKs);
Om = linspace(0.02, 10, 50);
Omeg = zeros(Neta1, NKs);
eta1 = linspace(0, 1, Neta1);
for et = 1:Neta1
  for kss = 1:NKs
    D = NFEqnBeamWithKs(Om, Ks(kss), eta1(et));
    for k = 2:length(Om)
      if D(k)*D(k-1) < 0
        Omeg(et, kss) = fzero(@NFEqnBeamWithKs, [Om(k-1), Om(k)], [],
        Ks(kss), eta1(et));
        break
      end
    end
  end
end
figure(1)
mesh(eta1, log10(Ks), (Omeg/pi)')
xlabel('\eta_1')
ylabel('log_{10}(K_s)')
zlabel('\Omega_1/\pi')
view([-15,30])
a = axis; a(4) = Kend;
axis(a)
figure(2)
[C, h] = contour(eta1, log10(Ks), (Omeg/pi)');
clabel(C, h)
xlabel('\eta_1')
ylabel('log_{10}(K_s)')

function C = NFEqnBeamWithKs(Om, Ks, eta1)
CA = T(Om*eta1).*(T(Om).*T(Om*(1-eta1))-R(Om).*R(Om*(1-eta1)));
CB = R(Om*eta1).*(T(Om).*R(Om*(1-eta1))-R(Om).*T(Om*(1-eta1)));
C = CA+CB+(R(Om).^2-T(Om).^2).*Om.^3/Ks;

function r = R(x)
r = 0.5*(sinh(x)+sin(x));

function t = T(x)
t = 0.5*(sinh(x)-sin(x));
```

where we have used a modification of **FindZeros** described in Section 5.5.1 to determine automatically the search region for fzero. The results are shown in Figure 7.8.



(a)

(b)

Figure 7.8 Lowest natural frequency coefficient of a cantilever beam restrained by a spring K_s attached at η_1: (a) Surface plot. (b) Contour plot as a function Ω_1/π.

Example 7.6 Enhancing 2D graphs with 3D objects

This example illustrates how 3D objects can be used to enhance a 2D graph. We shall use the graphing of the comparison of the volume of a sphere to the volume of an ellipsoid as a function of its two minor diameter ratios as the means to illustrate this. For a sphere of radius a and an ellipsoid with its major axis in the x-direction equal to $2a$, minor axis in the y-direction equal to $2b$, and minor axis in the z-direction equal to $2c$, the ratio of the volume of an ellipsoid to the volume of a sphere is given by

$$V = \frac{V_{\text{ellipse}}}{V_{\text{sphere}}} = \left(\frac{b}{a}\right)\left(\frac{c}{a}\right)$$

We create the following program to enhance the understanding of a plot of V as a function b/a for several values of c/a.

```
b = [0.5, 1];    c = b;
for k = 1:2
  plot(b, b*c(k), 'k-')
  text(0.75, (b(1)*c(k)+b(2)*c(k))/2-0.02, ['c/a = ' num2str(c(k))])
  hold on
end
xlabel('b/a')
ylabel('V')
for k = 1:4
  switch k
  case 1
    axes('position', [0.12, 0.2, 0.2, 0.2])
    [xs, ys, zs] = ellipsoid(0, 0, 0, 1, b(1), c(1), 20);
    mesh(xs, ys, zs)
    text(0, 0, 1, ['b/a = ' num2str(b(1)) ' c/a = ' num2str(c(1))])
  case 2
    axes('position', [0.1, 0.5, 0.2, 0.2])
    [xs, ys, zs] = ellipsoid(0, 0, 0, 1, b(1), c(2), 20);
```

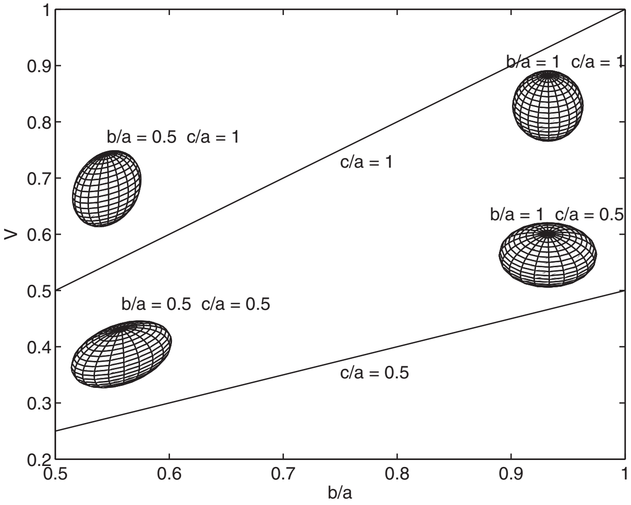


Figure 7.9 Enhancement of a 2D graph with 3D objects.

```
    mesh(xs, ys, zs)
    text(0, 0, 1.5, ['b/a = ' num2str(b(1)) ' c/a = ' num2str(c(2))])
  case 3
    axes('position', [0.7, 0.65, 0.2, 0.2])
    [xs, ys, zs] = ellipsoid(0, 0, 0, 1, b(2), c(2), 20);
    mesh(xs, ys, zs)
    text(-1.5, 0, 2, ['b/a = ' num2str(b(2)) ' c/a = ' num2str(c(2))])
  case 4
    axes('position', [0.7, 0.38, 0.2, 0.2])
    [xs, ys, zs] = ellipsoid(0, 0, 0, 1, b(2), c(1), 20);
    mesh(xs, ys, zs)
    text(-1.5, 0, 1.5, ['b/a = ' num2str(b(2)) ' c/a = ' num2str(c(1))])
  end
  axis equal off
end
```

The execution of this program results in Figure 7.9.

Example 7.7 Generation of planes and their projections

When the coordinates of three points in space, $\mathbf{P}_0(x_0, y_0, z_0)$, $\mathbf{P}_1(x_1, y_1, z_1)$, and $\mathbf{P}_2(x_2, y_2, z_2)$, have been specified, the parametric representation of any point in a plane containing these three points is given by

$$\mathbf{P} = \mathbf{P}_0 + s\mathbf{v} + t\mathbf{w}$$

where

$$\begin{aligned}
\mathbf{P} &= x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \\
\mathbf{P}_0 &= x_0\mathbf{i} + y_0\mathbf{j} + z_0\mathbf{k} \\
\mathbf{v} &= v_1\mathbf{i} + v_2\mathbf{j} + v_3\mathbf{k} = (x_1 - x_0)\mathbf{i} + (y_1 - y_0)\mathbf{j} + (z_1 - z_0)\mathbf{k} \\
\mathbf{w} &= w_1\mathbf{i} + w_2\mathbf{j} + w_3\mathbf{k} = (x_2 - x_0)\mathbf{i} + (y_2 - y_0)\mathbf{j} + (z_2 - z_0)\mathbf{k}
\end{aligned}$$

and $0 \le s \le 1$ and $0 \le t \le 1$. Thus,

$$\begin{aligned}
x &= x_0 + sv_1 + tw_1 = x_0 + s(x_1 - x_0) + t(x_2 - x_0) \\
y &= y_0 + sv_2 + tw_2 = y_0 + s(y_1 - y_0) + t(y_2 - y_0) \\
z &= z_0 + sv_3 + tw_3 = z_0 + s(z_1 - z_0) + t(z_2 - z_0)
\end{aligned}$$

If it is assumed that we can adequately display a plane as a surface using a 5 × 5 grid of patches, then we can create the following function M file called **PlanarSurface** that creates the *x*-, *y*-, and *z*-coordinates of the planar surface:

```
function [xx, yy, zz, L] = PlanarSurface(P0, P1, P2)
v = P1-P0;
w = P2-P0;
```

```
S = 0:0.2:1;
L = length(S);
[s, t] = meshgrid(S, S);
xx = P0(1)+s*v(1)+t*w(1);
yy = P0(2)+s*v(2)+t*w(2);
zz = P0(3)+s*v(3)+t*w(3);
```

where P_0, P_1, and P_2 are each three-element vectors containing the coordinates of the points on the plane. Thus, if we execute the following script:

```
[x, y, z, L] = PlanarSurface([0 0 0], [2 6 3], [7 1 5]);
surf(x, y, z)
```

we obtain the results shown in Figure 7.10.

To project this surface onto the three orthogonal coordinate reference planes, we take the appropriate vector dot products. Thus, to project the planar surface onto the xy-plane, we have

$$\mathbf{P}\cdot(\mathbf{i} + \mathbf{j} + 0\mathbf{k})$$

Similarly, for the projection onto the yz-plane, we have

$$\mathbf{P}\cdot(0\mathbf{i} + \mathbf{j} + \mathbf{k})$$

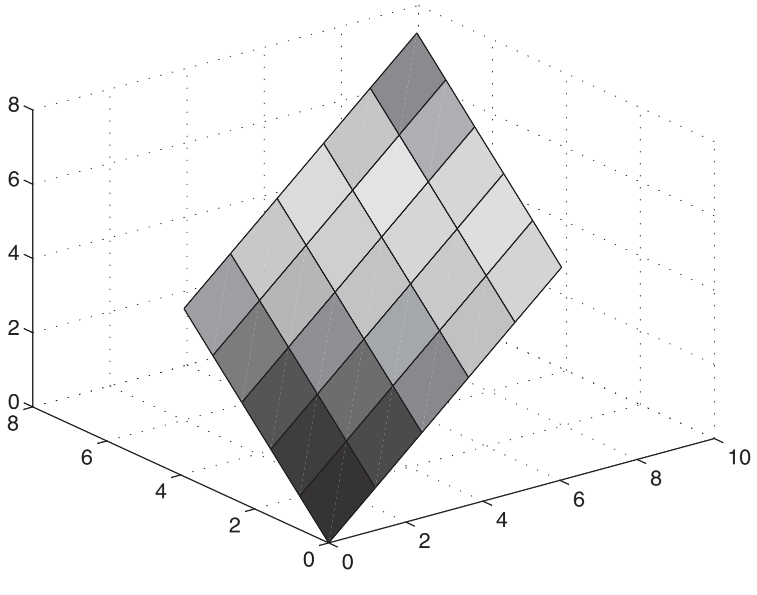


Figure 7.10 Generation of a planar surface.

and for the projection onto the *xz*-plane

$$\mathbf{P} \cdot (\mathbf{i} + 0\mathbf{j} + \mathbf{k})$$

Thus, we create a new function M file **PlanarSurfaceProj** to obtain these projections.

```
function PlanarSurfaceProj(P0, P1, P2)
[xx, yy, zz, L] = PlanarSurface(P0, P1, P2);
hold on
a = axis;
c(1:L, 1:L, 1:3) = zeros(L, L, 3);
c(:,:,1) = 1;
c(:,:,2) = 1;
c(:,:,3) = 0;
surf(xx, yy, a(5)*ones(L, L), c)
surf(xx, a(4)*ones(L, L), zz, c)
surf(a(2)*ones(L, L), yy, zz, c)
```

The array *c* is defined such that the projections are displayed in yellow. The first two indices of array *c* must be of the same order as *xx*, *yy*, and *zz*. The last index must represent exactly three elements, each of which can have a value that varies from zero to one. These last three elements specify the color of each patch at each combination of the first two indices. If we execute the script

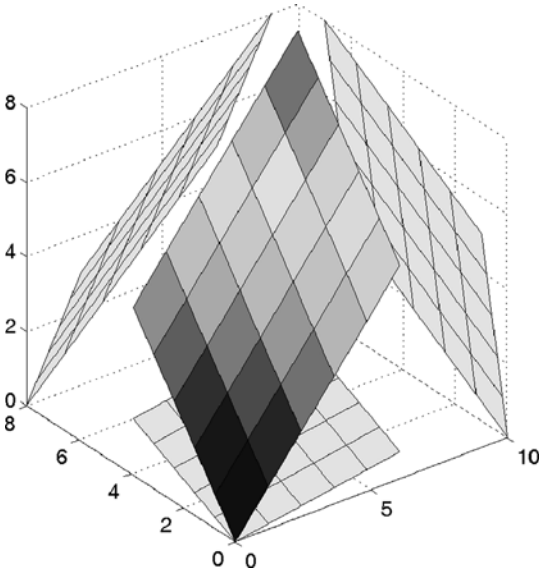


Figure 7.11 Projection of a plane onto its coordinate reference planes.

```
[x, y, z, L] = PlanarSurface([0 0 0], [2 6 3], [7 1 5]);
surf(x, y, z)
PlanarSurfaceProj([0 0 0], [2 6 3], [7 1 5])
```

we obtain the results shown in Figure 7.11. Although there is an illusion that the projections do not appear to be in their designated planes, the use of the *Rotate 3D* icon in the figure window will confirm that they are.

Example 7.8 Rotation and translation of 3D objects: Euler angles

The rotation and translation of a point $p(x, y, z)$ to another location $P(X, Y, Z)$ is given by[4]

$$\begin{aligned} X &= L_x + a_{11}x + a_{12}y + a_{13}z \\ Y &= L_y + a_{21}x + a_{22}y + a_{23}z \\ Z &= L_z + a_{21}x + a_{22}y + a_{23}z \end{aligned}$$

where L_x, L_y, and L_z are the x-, y-, and z-components of the translation, respectively, and a_{ij}, $i, j = 1, 2, 3$, are the elements of

$$a = \begin{bmatrix} \cos\psi\cos\chi & -\cos\psi\sin\chi & \sin\psi \\ \cos\phi\sin\chi + \sin\phi\sin\psi\cos\chi & \cos\phi\cos\chi - \sin\phi\sin\psi\sin\chi & -\sin\phi\cos\psi \\ \sin\phi\sin\chi - \cos\phi\sin\psi\cos\chi & \sin\phi\cos\chi + \cos\phi\sin\psi\sin\chi & \cos\phi\cos\psi \end{bmatrix}$$

The quantities ϕ, ψ, and χ are the ordered rotation angles (Euler angles) of the coordinate system about the origin: ϕ about the x-axis, ψ about the y-axis, and χ about the z-axis. In general (x, y, z) can be scalars, vectors of the same length, or matrices of the same order.

Before we apply these relations, we create the function M file **EulerAngles** to implement them.

```
function[Xrt, Yrt, Zrt] = EulerAngles(psi, chi, phi, Lx, Ly, Lz, x, y, z)
a = [cos(psi)*cos(chi), -cos(psi)*sin(chi), sin(psi);
     cos(phi)*sin(chi)+sin(phi)*sin(psi)*cos(chi), cos(phi)*cos(chi)-
     sin(phi)*sin(psi)*sin(chi), -sin(phi)*cos(psi);
     sin(phi)*sin(chi)-cos(phi)*sin(psi)*cos(chi),
     sin(phi)*cos(chi)+cos(phi)*sin(psi)*sin(chi),
     cos(phi)*cos(psi)];
Xrt = a(1,1)*x+a(1,2)*y+a(1,3)*z+Lx;
Yrt = a(2,1)*x+a(2,2)*y+a(2,3)*z+Ly;
Zrt = a(3,1)*x+a(3,2)*y+a(3,3)*z+Lz;
```

[4] W. Gellert, H. Kustner, M. Hellwich, and H. Kastner, *The VNR Concise Encyclopedia of Mathematics*, Van Nostrand Reinhold, New York, 1975, pp. 534–535.

We now illustrate the use of these transformation equations with the manipulation of a torus, whose coordinates are given by[5]

$$\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \\ z &= \pm\sqrt{a^2 - \left(\sqrt{x^2 + y^2} - b\right)^2} \end{aligned}$$

where $b - a \le r \le b + a, 0 \le \theta \le 2\pi$, and $b > a$.

We first create the following function M file to obtain the coordinates of the torus:

```
function[X, Y, Z] = Torus(a, b)
r = linspace(b-a, b+a, 10);
th = linspace(0, 2*pi, 22);
x = r'*cos(th);
y = r'*sin(th);
z = real(sqrt(a^2-(sqrt(x.^2+y.^2)-b).^2));
X = [x x];
Y = [y y];
Z = [z -z];
```

where real is used to eliminate any small imaginary parts caused by numerical round-off error.

We obtain four plots of the torus. The first plot is the torus without any rotations. In the second plot, the torus is rotated 60° about the x-axis ($\phi = 60°$) and compared to the orientation of the original torus. In the third plot, the torus is rotated 60° about the y-axis ($\psi = 60°$) and compared to the orientation of the original torus. In the final plot, the torus is rotated 60° about the x-axis ($\phi = 60°$) and 60° about the y-axis ($\psi = 60°$) and compared to the orientation of the original torus. We assume that $a = 0.2$ and $b = 0.8$. The script is

```
a = 0.2; b = 0.8;
[X, Y, Z] = Torus(a, b);
Lx = 0; Ly = 0; Lz = 0;
for k = 1:4
  subplot(2,2,k)
  switch k
    case 1
      mesh(X, Y, Z)
      v = axis;
      axis([v(1) v(2) v(3) v(4) -1 1])
      text(0.5, -0.5, 1, 'Torus')
    case 2
      psi = 0; chi = 0; phi = pi/3;
      [Xr Yr Zr] = EulerAngles(psi, chi, phi, Lx, Ly, Lz, X, Y, Z);
      mesh(X, Y, Z)
```

[5] von Seggern, *CRC Standard Curves*.

```
         hold on
         mesh(Xr, Yr, Zr)
         text(0.5, -0.5, 1,'\phi = 60\circ')
      case 3
         psi = pi/3; chi = 0; phi = 0;
         [Xr Yr Zr] = EulerAngles(psi,chi,phi,Lx,Ly,Lz,X,Y,Z);
         mesh(X, Y, Z)
         hold on
         mesh(Xr, Yr, Zr)
         text(0.5,-0.5,1,'\psi = 60\circ')
      case 4
         psi = pi/3; chi = 0; phi = pi/3;
         [Xr Yr Zr] = EulerAngles(psi,chi,phi,Lx,Ly,Lz,X,Y,Z);
         mesh(X, Y, Z)
         hold on
         mesh(Xr, Yr, Zr)
         text(0.5, -0.5, 1.35,'\psi = 60\circ')
         text(0.55, -0.5, 1,'\phi = 60\circ')
   end
   axis equal off
   grid off
   colormap([0 0 0])
end
```

The execution of this script results in Figure 7.12.

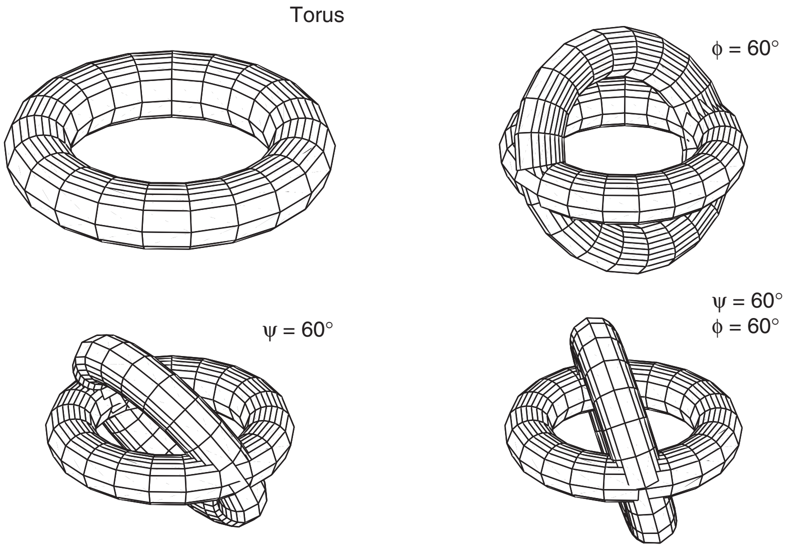


Figure 7.12 Rotations of a torus.

7.3 SUMMARY OF FUNCTIONS INTRODUCED IN CHAPTER 7

In Table 7.10, we have summarized the plotting functions introduced in Chapter 7.

TABLE 7.10 MATLAB Functions Introduced in Chapter 7

MATLAB function	Description
`axis on/off`	Turns axes on and off
`box on/off`	Places box around axes borders (`axis on` must be selected)
`axis vis3d`	Freezes aspect ratio to enable rotation of 3D objects and overrides stretch-to-fill
`clabel`	Labels elevations of a contour plot
`colorbar`	Displays color bar with values of color scale
`colormap`	Sets color map; when a three-element vector, it sets all colors to one value
`contour`	Creates a two-dimensional contour plot
`contourf`	Fills regions of a 2D contour plot with colors
`contour3`	Creates a 3D contour plot
`cylinder`	Generates the coordinates of a cylinder of a specified profile
`ellipsoid`	Generates an ellipsoid
`fill3`	Fills polygons oriented in three-dimensional space
`grid on/off`	Turns graph grid lines on and off
`hidden on/off`	Enables or disables hidden line removal
`mesh`	Plots a surface with white patches and lines; color based on their z-value
`meshc`	Plots a `mesh` generated surface with contours shown beneath it.
`meshz`	Draws vertical planes around the limits of the surface
`plot3`	Linear 3D plots
`ribbon`	Generates a ribbon plot in 3D
`shading`	Sets shading properties of surfaces created by `surf`
`sphere`	Generates a sphere
`surf`	Plots a surface with patches whose colors are based on their z-value
`surfc`	Plots a `surf` generated surface with contours shown beneath it.
`surfnorm`	Computes and displays surface normals
`text`	Creates a text object in the current axes
`view`	Changes the viewpoint specification
`waterfall`	Creates a waterfall effect of a surface generated by `mesh`
`zlabel`	Labels the z-axis.

EXERCISES

Section 7.1

7.1 Plot the following three-dimensional curves. Use `axis equal`.

Name	Parameter values	Equations
Spherical helix[6]	$c = 5.0; 0 \le \varphi \le 10\pi$	$x = \sin(\varphi/2c)\cos\varphi$ $y = \sin(\varphi/2c)\sin\varphi$ $z = \cos(\varphi/2c)$
Toroidal spiral	$a = 0.2, b = 0.8,$ $c = 20.0; 0 \le \varphi \le 2\pi$	$x = [b + a\sin(c\varphi)]\cos\varphi$ $y = [b + a\sin(c\varphi)]\sin\varphi$ $z = a\cos(c\varphi)$
Sine wave on sphere	$a = 10.0, b = 1.0,$ $c = 0.3; 0 \le \varphi \le 2\pi$	$x = \cos\varphi\sqrt{b^2 - c^2\cos^2(a\varphi)}$ $y = \sin\varphi\sqrt{b^2 - c^2\cos^2(a\varphi)}$ $z = c\cos(a\varphi)$
Concho spiral[7]	$a = 1.0, b = 1.05,$ $c = 2.0; 0 \le u \le 12\pi$	$r = ab^u$ $\varphi = u$ $z = cb^u$
Intersection of two cylinders[8]	$a = 1.0, b = 1.3; 0 \le \varphi \le 2\pi$	$x = a\cos\varphi$ $y = a\sin\varphi$ $z = \pm\sqrt{b^2 - a^2\sin^2\varphi}$
Baseball seam[9]	$a = 0.4; 0 \le \varphi \le 4\pi$	$x = \sin[\pi/2 - (\pi/2 - a)\cos(t)]$ $\cos[t/2 + a\sin(2t)]$ $y = \sin[\pi/2 - (\pi/2 - a)\cos(t)]$ $\sin[t/2 + a\sin(2t)]$ $z = \cos[\pi/2 - (\pi/2 - a)\cos(t)]$
Spherical spiral[10]	$a = 0.08; -12\pi \le \varphi \le 12\pi$	$x = \cos\varphi\cos\left(\tan^{-1}a\varphi\right)$ $y = \sin\varphi\cos\left(\tan^{-1}a\varphi\right)$ $z = -\sin\left(\tan^{-1}a\varphi\right)$

[6] *Ibid*.
[7] Weisstein, *CRC Concise Encyclopedia,* p. 501.
[8] *Ibid.,* p. 2853.
[9] http://local.wasp.uwa.edu.au/~pbourke/geometry/baseball/
[10] Weisstein, *CRC Concise Encyclopedia,* p. 2795.

Section 7.2

7.2 Plot the following surfaces. Use `axis equal vis3d`. Use the *Rotate 3D* icon to get a better understanding of the surface. In some cases, the use of `shading interp` will improve the image.

Name	Parameter values	Equations
Seashell[11]	$0 \le v \le 2\pi, 0 \le u \le 6\pi$	$x = 2\left(1 - e^{u/(6\pi)}\right)\cos(u)\cos^2(0.5v)$ $y = 2\left(-1 + e^{u/(6\pi)}\right)\sin(u)\cos^2(0.5v)$ $z = 1 - e^{u/(3\pi)} - \sin(v) + e^{u/(6\pi)}\sin(v)$
Figure eight torus[12]	$c = 1, -\pi \le u, v \le \pi$	$x = \cos(u)(c + \sin(v)\cos(u)$ $-\sin(2{*}v)\sin(u)/2);$ $y = \sin(u)(c + \sin(v)\cos(u)$ $-\sin(2{*}v)\sin(u)/2);$ $z = \sin(u)\sin(v)$ $+\cos(u)\sin(2{*}v)/2;$
Helical spring[13]	$r_1 = r_2 = 0.25, T = 2, n = 6,$ $0 \le u \le 2n\pi, 0 \le v \le 2\pi$	$x = [1 - r_1\cos(v)]\cos(u)$ $y = [1 - r_1\cos(v)]\sin(u)$ $z = r_2[\sin(v) + Tu/\pi]$
Cornucopia	$a = 0.3, b = 0.5,$ $0 \le u \le 2\pi, -3 \le v \le 3$	$x = e^{bv}\cos v + e^{av}\cos u \cos v$ $y = e^{bv}\sin v + e^{av}\cos u \sin v$ $z = e^{av}\sin u$
Astroidal ellipsoid[14]	$a = b = c = 1,$ $-\pi/2 \le u \le \pi/2,$ $-\pi \le v \le \pi$	$x = (a\cos u \cos v)^3$ $y = (b\sin u \cos v)^3$ $x = (c\sin v)^3$
Möbius strip[15]	$-0.4 \le s \le 0.4, 0 \le t \le 2\pi$	$x = s\cos(t/2)\cos t$ $y = s\cos(t/2)\sin t$ $z = s\sin(t/2)$
Bow curve[16]	$T = 0.7; 0 \le u, v \le 2\pi$	$x = (2 + T\sin u)\sin(2v)$ $y = (2 + T\sin u)\cos(2v)$ $z = T\cos u + 3\cos v$

(*Continued*)

[11] *Ibid*., p. 2644.

[12] http://local.wasp.uwa.edu.au/~pbourke/geometry/figure8torus/

[13] http://local.wasp.uwa.edu.au/~pbourke/geometry/spring/

[14] Weisstein, *CRC Concise Encyclopedia,* p. 136.

[15] *Ibid.*, p. 1928.

[16] http://local.wasp.uwa.edu.au/~pbourke/geometry/bow/

Name	Parameter values	Equations
Hyperbolic helicoid[17]	$\tau = 7; -\pi \le u \le \pi,$ $0 \le v \le 0.5$	$x = \dfrac{\sinh(v)\cos(\tau u)}{1 + \cosh(u)\cosh(v)}$ $y = \dfrac{\sinh(v)\sin(\tau u)}{1 + \cosh(u)\cosh(v)}$ $z = \dfrac{\sinh(u)\cosh(v)}{1 + \cosh(u)\cosh(v)}$
Apple surface[18]	$0 \le u \le 2\pi, -\pi \le v \le \pi$ [Use the transparency option to view interior]	$x = \cos u(4 + 3.8\cos v)$ $y = \sin u(4 + 3.8\cos v)$ $z = (\cos v + \sin v - 1)(1 + \sin v)$ $\log(1 - \pi v/10) + 7.5\sin v$

7.3 The mode shape of a solid circular plate clamped along its outer boundary $r = b$ is[19]

$$w_{mn}(r, \theta) = [C_{mn}J_m(\Omega_{mn}r/b) + I_m(\Omega_{mn}r/b)]\cos(m\theta)$$

where $m = 0, 1, 2, \ldots, J_m(x)$ is the Bessel function of the first kind of order m and $I_m(x)$ is the modified Bessel function of the first kind of order m,

$$C_{mn} = -\frac{I_m(\Omega_{mn})}{J_m(\Omega_{mn})}$$

and Ω_{mn} are the solutions to

$$J_m(\Omega_{mn})I_{m+1}(\Omega_{mn}) + I_m(\Omega_{mn})J_{m+1}(\Omega_{mn}) = 0$$

as obtained in Exercise 5.9.

Using the results of Exercise 5.9 in which the lowest three natural frequency coefficients for $m = 0, 1, 2$ have been determined, plot each of the corresponding nine mode shapes with `surfc` on one figure using `subplot`, and place at the top of each figure the value of m, n, and the frequency coefficient. Do not draw the axes. The first row is for $m = 0$, and so on. Normalize each mode shape using the `max` function twice (because the displacement field is a matrix) so that the maximum absolute value of the amplitude is 1. It is suggested that the number of radial divisions (r/b) be fifteen and those for the angular divisions be thirty. The results should look like those shown in Figure 7.13, which have been obtained with `surfc` and `colormap([0 0 0])` for clarity.

[17] Weisstein, *CRC Concise Encyclopedia*, p. 1421.

[18] http://local.wasp.uwa.edu.au/~pbourke/geometry/apple/

[19] Magrab, *Vibration of Elastic Structural Members*, p. 252.

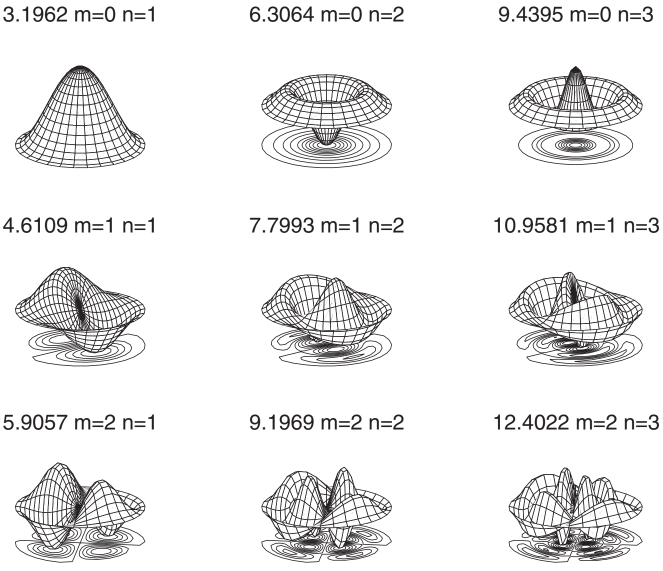


Figure 7.13 Mode shapes of a circular plate.

7.4 Consider a slab of thickness $2L$ in the x-direction and of very large dimensions in the y- and z-directions. If the slab, which is initially at a uniform constant temperature T_i at $t = 0$, is suddenly exposed to a convective environment of temperature T_∞, then the temperature distribution as a function of time and position within the slab is[20]

$$\frac{\theta}{\theta_i} = 2\sum_{n=1}^{\infty} \frac{\sin\delta_n \cos(\delta_n \eta)}{\delta_n + \sin\delta_n \cos(\delta_n)} \exp\left(-\delta_n^2 \tau\right)$$

where $\theta = \theta(\eta, \tau) = T - T_\infty$, $T = T(\eta, \tau)$ is the temperature in the slab, $\theta_i = T_i - T_\infty$, $\eta = x/L$, $\tau = \alpha^2 t/L^2$ is the nondimensional time (sometimes called the Fourier modulus), α is the thermal diffusivity, and δ_n are the solutions of

$$\cot \delta_n = \frac{\delta_n}{Bi}$$

where $Bi = \overline{h}Lk$ is the Biot number, $\overline{h}$ is the average heat transfer coefficient for convection from the entire surface, and k is the thermal conductivity of the slab.

Find the lowest twenty values of δ_n for $Bi = 0.7$, and use them to plot the surface $\theta(\eta, \tau)/\theta_i$ for $0 \le \eta \le 1$ and $0 \le \tau \le 2$. Then use the *Rotate 3D* icon interactively to obtain an acceptable view of the surface. Label the axes and title the figure.

[20] D. R. Pitts and L. E. Sissom, *Theory and Practice of Heat Transfer*, Schaum's Outline Series, McGraw-Hill, New York, 1977, p. 79.

7.5 Plot the following mode shape and its contours for a square membrane clamped on its outer boundary using twenty-five grid points in each direction for $0 \le x \le 1$ and $0 \le y \le 1$:

$$w_{23}(x, y) = \sin(2\pi x)\sin(3\pi y)$$

7.6 The mean Nusselt number for turbulent flow over a plate of length l is[21]

$$Nu = \frac{0.037\text{Re}^{0.8}\,\text{Pr}}{1 + 2.443\text{Re}^{-0.1}\left(\text{Pr}^{2/3} - 1\right)} \qquad 5 \times 10^5 \le \text{Re} \le 10^7 \quad 0.6 \le \text{Pr} \le 2000$$

where Re is the Reynolds number and Pr is the Prandtl number. Plot $\log_{10}(Nu)$ as a surface that is a function of $\log_{10}(\text{Re})$and $\log_{10}(\text{Pr})$ over the ranges indicated. Connect vertical lines from the boundary plane of the figure to the corners of the surface as shown in Figure 7.4.

7.7 The location of the neutral axis of a steel-reinforced concrete beam shown in Figure 7.14 is determined by the parameter k as defined below[22]

$$k = -\rho n + \sqrt{(\rho n)^2 + 2\rho n}$$

where $\rho = A_s/b_d$ and $n = E_s/E_c$, which is the ratio of the Young's modulus of the steel to that of concrete. Plot a surface of k as a function of n and ρ for ten values of n for $6 \le n \le 12$ and for five values of ρ for $0.001 \le \rho \le 0.009$ plus another ten values for $0.01 \le \rho \le 0.1$.

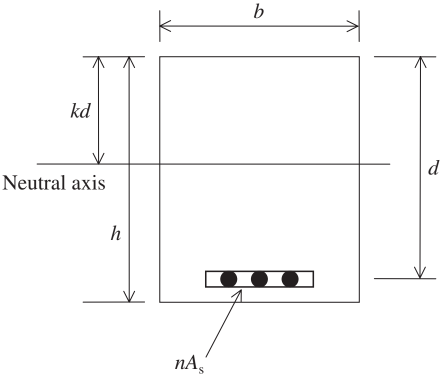


Figure 7.14 Section of a steel-reinforced concrete beam.

[21] Beitz and Kuttner, *Handbook of Mechanical Engineering,* p. C31.

[22] L. Spiegal and G. F. Limbrunner, *Reinforced Concrete Design*, 3rd ed., Prentice Hall, Upper Saddle River, NJ, 1992, p. 196.

TABLE 7.11 Deviations from Process Norms

x_1	x_2	z	x_1	x_2	z
2	50	1.5713	2	360	−0.6023
8	110	−1.1460	4	205	5.8409
11	120	−2.2041	4	400	−0.3620
10	550	−1.5968	20	600	4.3341
8	295	−2.8937	1	585	−2.0368
4	200	1.1136	10	540	−1.5415
2	375	1.9297	15	250	0.0302
2	52	1.1962	15	290	−2.1809
9	100	−3.8650	16	510	1.5587
8	300	−0.4763	17	590	0.3222
4	412	−1.3223	6	100	2.1478
11	400	−0.4619	5	400	0.1537
12	500	0.4911			

7.8 Consider the data in Table 7.11, which are the deviations of the output of a process about its mean value $z = 0$ for the given inputs x_1 and x_2. Use `stem3` to obtain Figure 7.15. You will have to plot the plane at $z = 0$ with a separate command. Use `view(-30, 7)` to obtain the orientation shown. (The function `stem3` is the 3D version of `stem`.)

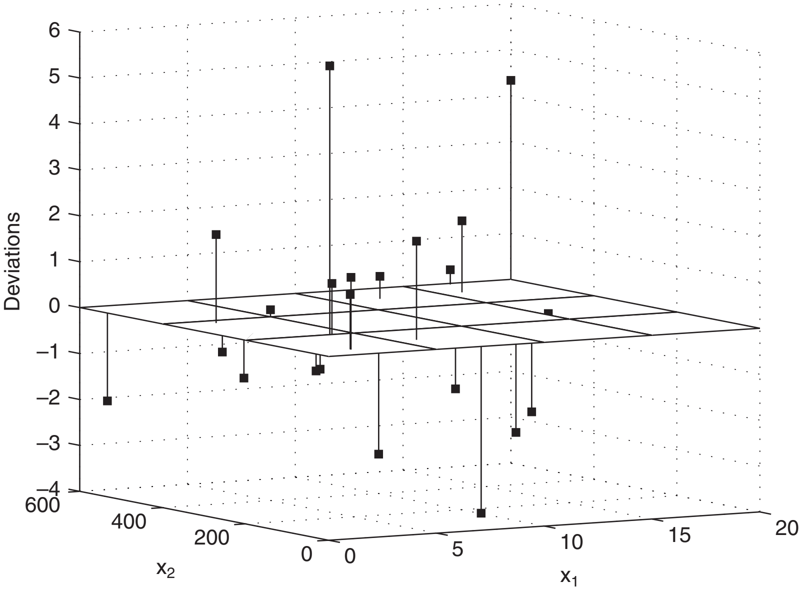


Figure 7.15 Deviations from the plane $z = 0$ using `stem3`.

7.9 Consider a thin rectangular plate with a circular hole of radius a that is subjected to a uniform tension of magnitude S in the x-direction. The radial, tangential, and shear stresses are, respectively, given by[23]

$$\overline{\sigma}_{rr} = \frac{\sigma_{rr}}{S/2} = 1 - \frac{1}{\eta^2} + \left(1 + \frac{3}{\eta^4} - \frac{4}{\eta^2}\right)\cos(2\theta)$$

$$\overline{\sigma}_{\theta\theta} = \frac{\sigma_{\theta\theta}}{S/2} = 1 + \frac{1}{\eta^2} - \left(1 + \frac{3}{\eta^4}\right)\cos(2\theta)$$

$$\overline{\tau}_{r\theta} = \frac{\tau_{rr}}{S/2} = -\left(1 - \frac{3}{\eta^4} + \frac{2}{\eta^2}\right)\sin(2\theta)$$

where $\eta = r/a \geq 1$ and the origin of the polar coordinates (r, θ) is located at the center of the hole. The von Mises equivalent tensile stress is given by

$$\overline{\sigma} = \sqrt{\frac{1}{2}\left[\left(\overline{\sigma}_{rr} - \overline{\sigma}_{\theta\theta}\right)^2 + \overline{\sigma}_{rr}^2 + \overline{\sigma}_{\theta\theta}^2\right] + 3\overline{\tau}_{r\theta}^2}$$

Obtain a contour plot of $\overline{\sigma}$ for $3 \geq \eta \geq 1$. The result should look like that shown in Figure 7.16.

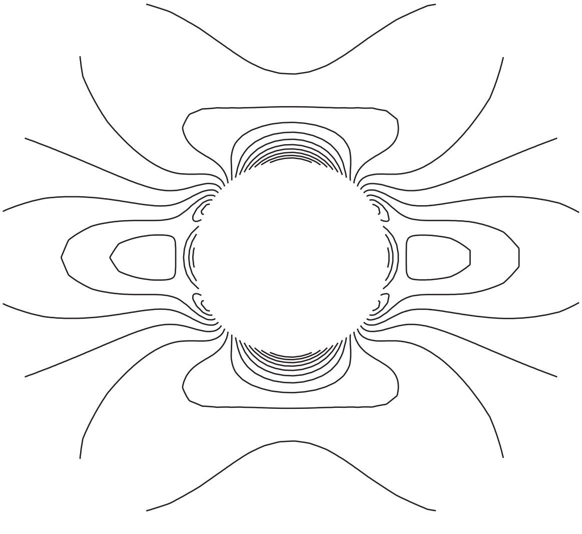


Figure 7.16 von Mises stress distribution around a hole in a plate.

[23] S. Timoshenko and J. N. Goodier, *Theory of Elasticity*, McGraw-Hill, New York, 1951, p. 80.

8

Engineering Statistics

Edward B. Magrab

The solutions to a wide range of engineering statistics applications are illustrated with the Statistics Toolbox.

8.1 DESCRIPTIVE STATISTICAL QUANTITIES

Consider a collection of measured values $x_j, j = 1, 2, \ldots, n$. The sample mean of these values is

$$\bar{x} = \frac{1}{n}\sum_{j=1}^{n} x_j \tag{8.1}$$

and sample variance is

$$s^2 = \frac{1}{n-1}\left[\sum_{j=1}^{n} x_j^2 - n\bar{x}^2\right] \tag{8.2}$$

where s is the standard deviation. These quantities are the estimates of the true mean μ and the true standard deviation σ. The mean value is determined from

```
mean(x)
```

and the standard deviation from

```
std(x)
```

where x is either a vector or matrix of values.

Histograms

Let the smallest value of x_j be denoted $x_{\min}$ and its largest value be denoted $x_{\max}$. We shall now sort these values as follows. We first divide the region $x_{\min} - x_{\max}$ into N equal segments called bins, and place each x_j into that bin whose lower limit is less than or equal to x_j and whose upper limit is greater than x_j. We denote the center of each bin b_k, $k = 1, 2, \ldots, N$. We note that $x_{\min}$ is in b_1 and $x_{\max}$ is in b_N. After all the x_j have been assigned to a bin, the number of x_j falling into each bin is counted. We denote this value n_k, which is the number of data values that fell in the bin whose center is b_k. When the number of values n_k is plotted as a function of the value of the center of each bin, and each bin is represented by a bar whose height is proportional to n_k and whose width is equal to the bin's upper and lower limits, then the resulting figure is called a histogram. The number of x_j in each bin can be determined from

```
[nk, b] = hist(x, N)
```

where *nk* is the vector of n_k, *b* is the vector of bin centers computed by `hist`, *x* are the *n* data samples, and *N* is the number of bins desired. When *N* is omitted, MATLAB uses $N = 10$. This same function without the left-hand side plots the histogram; that is,

```
hist(x)
```

One can also use

```
bar(b, nk)
```

to plot the histogram (recall Table 6.5), where `hist` is frequently used to determine n_k.

If we define $f_k = n_k/n$, then we have the fraction of the n samples that fall in the bin centered at b_k. If we let

$$c_k = \sum_{j=1}^{k} f_j \qquad k = 1, 2, \ldots, N$$

then c_k is called the cumulative distribution function and is obtained from

```
ck = cumsum(f)
```

where $f = [f_1 f_2 \ldots f_k] = [n_1/n \; n_2/n \; \ldots \; n_k/n]$. We can also plot c_k versus b_k, which is an approximation to the probability that a measurement has a value less than or equal to b_k.

Now let us sort x from its lowest value to its highest value. The lowest value can be obtained using `min`(x) and the highest value from `max`(x), where $x = [x_1, x_2, \ldots, x_n]$. The range of the values is the difference between the highest and lowest values of the samples and can be determined from either

```
range(x)
```

or

```
max(x)-min(x)
```

The center of the sorted values is called the median value. If the number of samples n is odd, then the median value is $x_{(n+1)/2}$; if it is even, then the median value is $(x_{n/2} + x_{n/2+1})/2$. The median value is determined from

```
median(x)
```

Another statistical metric that is sometimes useful is the geometric mean, which is defined as the nth root of the product of the measurements of n samples—that is,

$$\bar{x}_g = \sqrt[n]{\prod_{j=1}^{n} x_j}$$

This quantity can be determined from either

```
geomean(x)
```

or from the expression

```
prod(x)^(1/length(x))
```

We now illustrate the use of these relations with an example.

Example 8.1 Determination of several statistical quantities

Consider the data given in Table 8.1. We shall find the mean value, median value, standard deviation, geometric mean, range, minimum value, and maximum value and plot a histogram and the cumulative distribution of these data. We shall place the data in nine bins starting at 80 and ending at 240. We shall color the bars of the histogram yellow.

TABLE 8.1 Data Comprising: **DataSet81**

105	97	245	163	207	134	218	199
160	196	221	154	228	131	180	178
157	151	175	201	183	153	174	154
190	76	101	142	149	200	186	174
199	115	193	167	171	163	87	176
121	120	181	160	194	184	165	145
160	150	181	168	158	208	133	135
172	171	237	170	180	167	176	158
156	229	158	148	150	118	143	141
110	133	123	146	169	158	135	149

The data are placed in a function M file **DataSet81**. Thus,

```
function d = DataSet81
d = [105 97 245 163 207 134 218 199 160 196 221 154 228 131 180 178 . . .
    157 151 175 201 183 153 174 154 190 76 101 142 149 200 186 174 . . .
    199 115 193 167 171 163 87 176 121 120 181 160 194 184 165 145 . . .
    160 150 181 168 158 208 133 135 172 171 237 170 180 167 176 158 . . .
    156 229 158 148 150 118 143 141 110 133 123 146 169 158 135 149];
```

The script is

```
data = DataSet81;
n = length(data);
b = 80:20:240;
nn = hist(data, b);
maxn = max(nn);
cs = cumsum(nn*maxn/n);
bar(b, nn, 0.95, 'y')
axis([70, 250, 0, maxn])
box off
hold on
plot(b, cs, 'k-s')
title('\leftarrow Histogram Cumulative distribution \rightarrow')
ylabel('Number of occurrences')
xlabel('Measured values')
text(72, 0.97*maxn, ['Mean = ' num2str(mean(data))])
text(72, 0.92*maxn, ['Median = ' num2str(median(data))])
text(72, 0.87*maxn, ['Geometric mean = ' num2str(geomean(data))])
text(72, 0.82*maxn, ['Standard deviation = ' num2str(std(data))])
text(72, 0.77*maxn, ['No. of samples = ' num2str(n)])
text(72,.67*maxn, ['Maximum = ' num2str(max(data))])
text(72,.72*maxn, ['Minimum = ' num2str(min(data))])
text(72,.62*maxn, ['Range = ' num2str(range(data))])
plot([70 250], [maxn maxn], 'k', [250 250], [0 maxn], 'k')
j = 0:0.1:1;
lenj = length(j);
text(repmat(251, lenj, 1), maxn*j', num2str(j', 2))
plot([repmat(248.5, 1, lenj); repmat(250, 1, lenj)], [maxn*j; maxn*j], 'k')
```

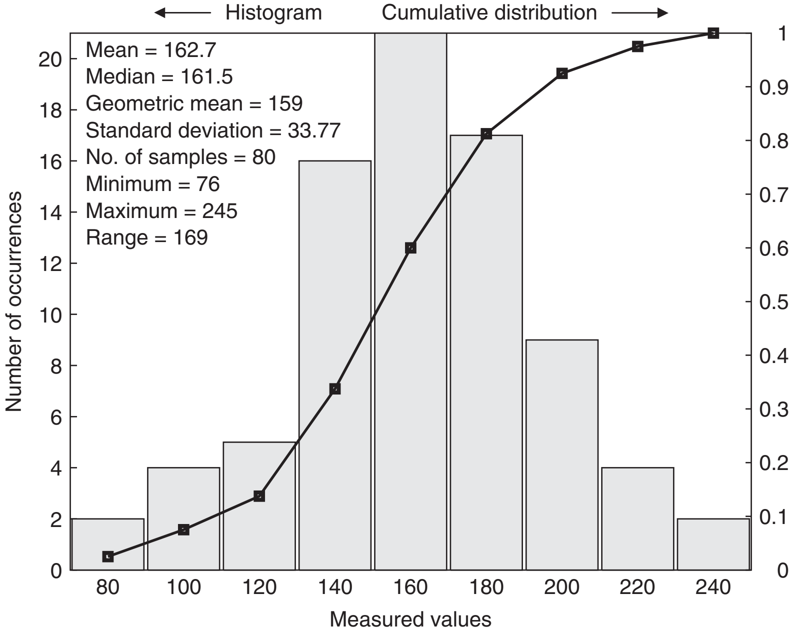


Figure 8.1 Histogram, cumulative distribution, and descriptive statistics for the data in **DataSet81**.

Execution of this program results in Figure 8.1. Although the centers of the bins can be computed by `hist`, we have chosen to specify them. This permits us to more easily control the presentation of the data. We had to turn off the `box` function because this function repeats the tic marks from the horizontal and vertical axes to the top and right-hand vertical axes, respectively. Therefore, we have to consider the labels independently from the tic marks, and we have to draw the top and right-hand figure boundaries separately. Thus, the tic marks appear at 21*j*, where, in this problem, 21 is the maximum value of the *y*-axis. The maximum value is set with the `axis` function.

If the bin centers had not been specified, then the resulting histogram would look slightly different because the centers of the bins would be different. This difference may change one or more of the n_k. Thus, the execution of the statements

```
[nn, b] = hist(DataSet81, 9);
bar(b, nn, 0.95, 'y');
axis([70, 250, 0, max(nn)])
```

results in Figure 8.2.

The differences between the histograms in Figures 8.1 and 8.2 are due to the differences in the bin centers. In the first case, the bin centers were defined as

b = [80, 100, 120, 140, 160, 180, 200, 220, 240]

whereas in the new script, the bin centers were computed by `hist` and found to be

b = [85.38, 104.16, 122.94, 141.72, 160.50, 179.27, 198.05, 216.83, 235.61]

We see that the number of x_j in several of the bins differs.

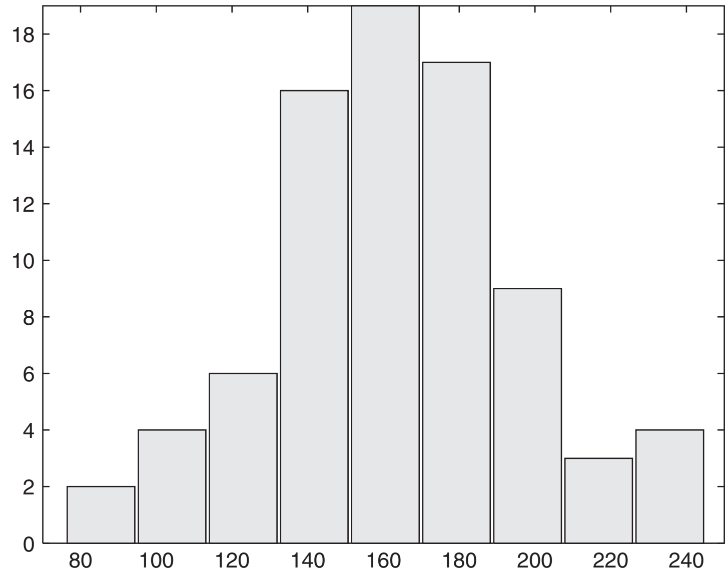


Figure 8.2 Resulting histogram for the data in **DataSet81** when `hist` computes the bin centers.

Another way of presenting these data is to use a box plot. A box plot of the data in **DataSet81** is shown in Figure 8.3, which is obtained from

`boxplot(`**DataSet81**, 'notch', 'on')

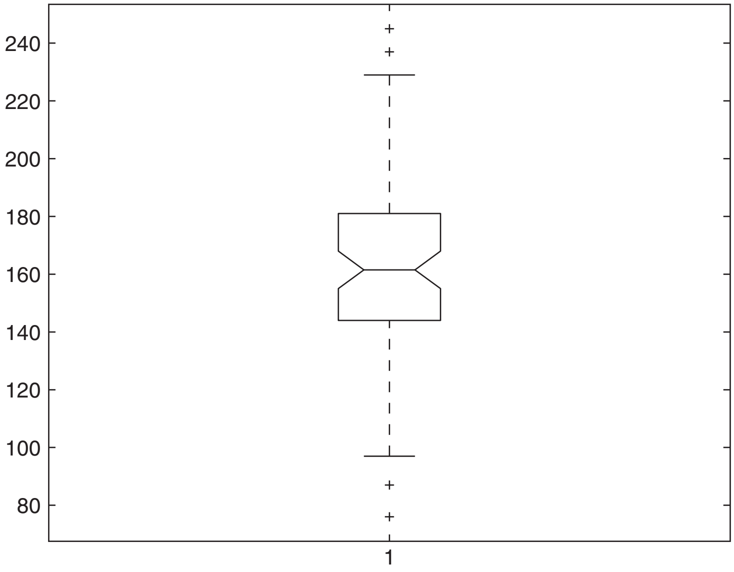


Figure 8.3 Box plot of **DataSet81**.

where we have selected to display a notched box. The notch indicates the median of the data. The region within the top and bottom limits of the box represents 50% of the data, with the bottom of the box indicating the end of the first quartile q_1 and its top the end of the third quartile q_3. Note that, in general, the box is not symmetrical about the median value. The lines (whiskers) extending from the bottom and top of the box represent the extreme values defined by the regions $q_1 - 1.5(q_3 - q_1)$ and $q_3 + 1.5(q_3 - q_1)$, respectively. Any data points that lie outside these whiskers are called outliers and are denoted in this figure by plus signs. The more general usage of a box plot is to compare several sets of data in this manner. See, for example, Figure 8.16b.

To obtain the values of q_1 and q_3 explicitly, we use, respectively,

```
q1 = prctile(DataSet81, 25)
q3 = prctile(DataSet81, 75)
```

which upon execution gives $q_1 = 144$ and $q_3 = 181$. The second argument in `prctile` specifies the percentile of interest. When the percentile equals 25%, this is referred to as the first quartile.

To determine whether the data are symmetrically distributed about the mean, we use

```
s = skewness(DataSet81)
```

which upon execution gives $s = -0.0246$. The negative sign means that the distribution is skewed to the left.

8.2 PROBABILITY DISTRIBUTIONS

MATLAB has a large family of probability distribution functions and random number generation functions. A subset of these functions is summarized in Table 8.2. We shall examine two discrete distributions, the binomial and the Poisson distributions, and two continuous distributions, the normal distribution and the Weibull distribution. Other distributions are used in a similar manner.

8.2.1 Discrete Distributions

The probability $P(X)$ that a discrete random variable $X = x$, where x is from the set of all possible values of X, is defined as

$$f(x) = P(X = x) \tag{8.3}$$

where $f(x) \geq 0$ for all x and

$$\sum_{\text{all } x_i} f(x_i) = 1 \tag{8.4}$$

The quantity $f(x)$ is called the probability mass function for a discrete random variable.

If we are interested in the probability that $X \leq x$—that is, $P(X \leq x)$—then

$$P(X \leq x) = \sum_{x_k \leq x} f(x_k) = 1 - \sum_{x_k > x} f(x_k) \tag{8.5}$$

TABLE 8.2 Several MATLAB Probability Distribution and Random Number Generation Functions

Probability distribution	Probability density function	Cumulative distribution function	Inverse cumulative distribution function	Mean and variance of distribution	Parameter estimates and confidence intervals	Random number generation
Discrete						
Binomial	binopdf	binocdf	binoinv	binostat	binofit	binornd
Poisson	poisspdf	poisscdf	poissinv	poissstat	poissfit	poissrnd
Continuous						
Chi-square	chi2pdf	chi2cdf	chi2inv	chi2stat	–	chi2rnd
f	fpdf	fcdf	finv	fstat	–	frnd
Lognormal	lognpdf	logncdf	logninv	lognstat	lognfit	lognrnd
Normal	normpdf	normcdf	norminv	normstat	normfit	normrnd
Rayleigh	raylpdf	raylcdf	raylinv	raystat	raylfit	raylrnd
Student *t*	tpdf	tcdf	tinv	tstat	–	trnd
Weibull	wblpdf	wblcdf	wblinv	wblstat	wblfit	wblrnd

which is called the cumulative distribution function. Conversely, if we are interested in the probability that $X \geq x$ — that is, $P(X \geq x)$—then

$$P(X \geq x) = \sum_{x_k \geq x} f(x_k) \tag{8.6}$$

Binomial Distribution

If we conduct n repeated trials such that (1) the trials are independent, (2) each trial results in only two possible outcomes, "success" or "failure," and (3) the probability p of a success on each trial remains constant, then the probability mass function is called the binomial distribution given by

$$f_b(x) = P(X = x) = \frac{n!}{x!(n-x)!} p^x (1-p)^{n-x} \quad x = 0, 1, \ldots, n \tag{8.7}$$

where x is the number of trials that meets with success.

The mean of this distribution is

$$\bar{x} = np \tag{8.8a}$$

and its standard deviation is

$$s = \sqrt{np(1-p)} \tag{8.8b}$$

The function that computes the probability mass function of the binomial distribution is

binopdf(x, n, p)

and that which computes its mean and variance s^2 is

[Bmean, Bvariance] = binostat(n, p)

where $x = 0, 1, 2, \ldots, n$.

Consider a die. The probability of getting any one of its sides to be the top surface is $p = 1/6$. Let the side with three dots be of interest. Then the probability that with one toss of the die, the side with three dots will appear is

$$P(X = \text{side with three dots}) = \frac{1!}{1!(0!)}(1/6)^1(1 - 1/6)^{1-1} = \frac{1}{6}$$

which can be determined from the expression

```
Pb = binopdf(1, 1, 1/6)
```

However, the probability that we can get the side with three dots to show up exactly once in two tries is

```
Pb = binopdf(1, 2, 1/6)
```

Executing this expression gives Pb $= 0.2778 < 1/3$.

Now, consider a coin toss; thus, $p = 0.5$. The probability of getting exactly four "heads" ($x = 4$) in ten tosses ($n = 10$) is determined from

```
Pb = binopdf(4, 10, 0.5)
```

which yields Pb = 0.2051.

Example 8.2 Probability of getting airplanes airborne[1]

An air force squadron of sixteen airplanes should always be ready to become airborne immediately. There is, however, a 20% chance that an aircraft will not start, at which time several minutes must elapse before another start procedure can be attempted. Thus, the probability of an aircraft starting immediately is 0.80.

We are interested in the probability that exactly twelve airplanes can successfully become airborne. The script is

```
Pb = binopdf(12, 16, 0.80)
```

Upon execution, we obtain Pb = 0.2001.

On the other hand, the probability that at least fourteen aircraft can become airborne immediately is determined from (recall Eqs. (8.5) and (8.6))

```
Pb = 1-binocdf(13, 16, 0.80)
```

or

```
Pb = sum(binopdf(14:16, 16, 0.80))
```

The execution of either expression gives Pb $= 0.3518$.

[1] A. J. Hayter, *Probability and Statistics for Engineers and Scientists*, PWS Publishing Company, Boston, 1996, p. 167.

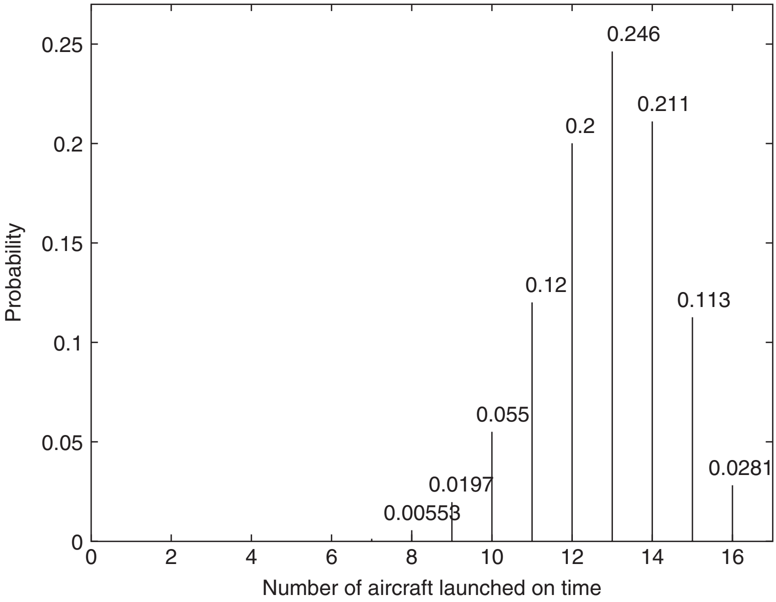


Figure 8.4 Probability mass function of launching 0 of 16 aircraft on time to launching 16 of 16 aircraft on time.

A graphical representation of this distribution can be obtained from the following script:

```
n = 1:16;
Pb = binopdf(n, 16, 0.80);
plot([n; n], [zeros(1,16); Pb], 'k')
text(8-.7:16-.7, Pb(8:16)+.005, num2str(Pb(8:16)',3))
axis([0, 17, 0, 0.27])
xlabel('Number of aircraft launched on time')
ylabel('Probability')
```

Upon execution, we obtain the results shown in Figure 8.4.

Poisson Distribution

Assume that an event occurs randomly throughout an interval and that this interval can be partitioned into smaller subintervals such that (1) the probability of more than one event in the subinterval is zero; (2) the probability of the event is the same for all subintervals and is proportional to the length of the subinterval; and (3) the number of events in each subinterval is independent of the other subintervals. Such a series of events is called a Poisson process. If the mean of the number of events in the interval is $\lambda > 0$, then the probability mass distribution is

$$f_p(x) = P(X = x) = \frac{e^{-x}\lambda^x}{x!} \quad x = 0, 1, 2, \dots \tag{8.9}$$

is a Poisson distribution for x events occurring in the interval.

The mean value of the Poisson distribution is

$$\bar{x} = \lambda \tag{8.10a}$$

and its standard deviation is

$$s = \sqrt{\lambda} \tag{8.10b}$$

The probability mass function of the Poisson distribution is obtained from

```
poisspdf(x, lambda)
```

and that which computes its mean and variance s^2 is

```
[Pmean, Pvariance] = poisstat(lambda)
```

A summary of the binomial and Poisson statistical functions is given in Table 8.2. It is seen from this table that one can also generate random numbers that will have either of these distributions.

Example 8.3 Adequacy of hospital resources

A hospital emergency room receives an average of forty-six heart attack cases per week or 46/7 per day. The hospital currently is able to handle nine such cases per day. The hospital staff is interested in knowing the probability that their current resources are adequate. Consequently, they want to know the value of $P(X \le 9)$. Thus,

```
Pp = poisscdf(9, 46/7)
```

which, upon execution, gives $Pp = 0.8712$. Thus, on 13% of the days additional resources will be required.

8.2.2 Continuous Distributions

The probability $P(X)$ that a continuous random variable X lies in the range $x_1 \le X \le x_2$, where x_1 and x_2 are from the set of all possible values of X, is defined as

$$P(x_1 \le X \le x_2) = \int_{x_1}^{x_2} f(x)dx \tag{8.11a}$$

where $f(x) \ge 0$ for all x, and

$$\int_{-\infty}^{\infty} f(x)dx = 1 \tag{8.11b}$$

The quantity $f(x)$ is called the probability distribution function (pdf) for a continuous random variable.

The cumulative distribution function (cdf) $F(x)$ is

$$F(x) = P(X \le x) = \int_{-\infty}^{x} f(u)du = 1 - \int_{x}^{\infty} f(u)du \tag{8.12}$$

and, therefore,

$$P(X \ge x) = 1 - F(x) \tag{8.13}$$

Normal Distribution

The normal probability distribution function is

$$f_n(x) = P(X = x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \quad -\infty < x < \infty \tag{8.14}$$

where $-\infty < \mu < \infty$ and $\sigma > 0$ are independent parameters. It can be shown that μ is the mean of the distribution and σ^2 its variance (σ is the standard deviation). If we have a set of data x_j, $j = 1, 2, \ldots, n$, then for the normal probability distribution function the probability of x_0 occurring is obtained from the following script.

```
mu = mean(x);
sigma = std(x);
Pn = normpdf(x0, mu, sigma)
```

where $x0 = x_0$, $mu = \mu$, $sigma = \sigma$, and the size of Pn is equal to the size of $x0$. Estimates for the values of μ and σ can also be obtained from

```
[mu, sigma] = normfit(x)
```

The cumulative normal distribution function $\Phi(x)$ is

$$\Phi(x) = P(X \le x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-(u-\mu)^2/2\sigma^2} du \qquad -\infty < x < \infty \tag{8.15}$$

$$= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{(x-\mu)/\sigma} e^{-u^2/2} du \qquad -\infty < x < \infty$$

or

$$\Phi(z) = P(Z \le z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{-u^2/2} du \qquad -\infty < z < \infty \tag{8.16}$$

where

$$z = (x - \mu)/\sigma \tag{8.17}$$

is called the standard normal random variable for which $\mu_z = 0$ and $\sigma_z = 1$. The quantity z can be obtained using

```
[Z, mu, s] = zscore(x)
```

where $Z = z$, $mu = \mu$, and $s = \sigma$. Thus, from Eq. (8.16), we see that

$$\begin{aligned} P(Z \geq z) &= 1 - \Phi(z) \\ P(Z \leq -z) &= \Phi(-z) \\ P(-z \leq Z \leq z) &= \Phi(z) - \Phi(-z) \\ P(z_L \leq Z \leq z_H) &= \Phi(z_H) - \Phi(z_L) \end{aligned} \tag{8.18}$$

The regions given by Eq. (8.18) are shown in Figure 8.5. The figure was obtained using `normspec`.

The probability that $x_o \leq x$ for a normal cdf is obtained from

```
mu = mean(x);
sigma = std(x);
Pn = normcdf(x0, mu, sigma)
```

where $x0 = x_0$, $mu = \mu$, $sigma = \sigma$, and the size of Pn is equal to the size of $x0$. If the x are converted to z using Eq. (8.17), then $mu = 0$ and $sigma = 1$. These are the default values and, therefore, when $x \rightarrow z$ these arguments can be omitted.

Referring to Figure 8.5(c) and Eq. (8.18), we see that if we are interested in $P(x_L \leq X \leq x_H)$, then

```
Pn = diff(normcdf([xL, xH], mean(x), std(x)))
```

For example, the probability of finding a measured value in **DataSet81** between 120 and 200 is obtained from

```
Pn = diff(normcdf([120, 200], mean(DataSet81), std(DataSet81)))
```

which, upon execution, yields $Pn = 0.7623$. The value compares well with the estimated value obtained from Figure 8.1 of approximately $0.93 - 0.15 = 0.78$.

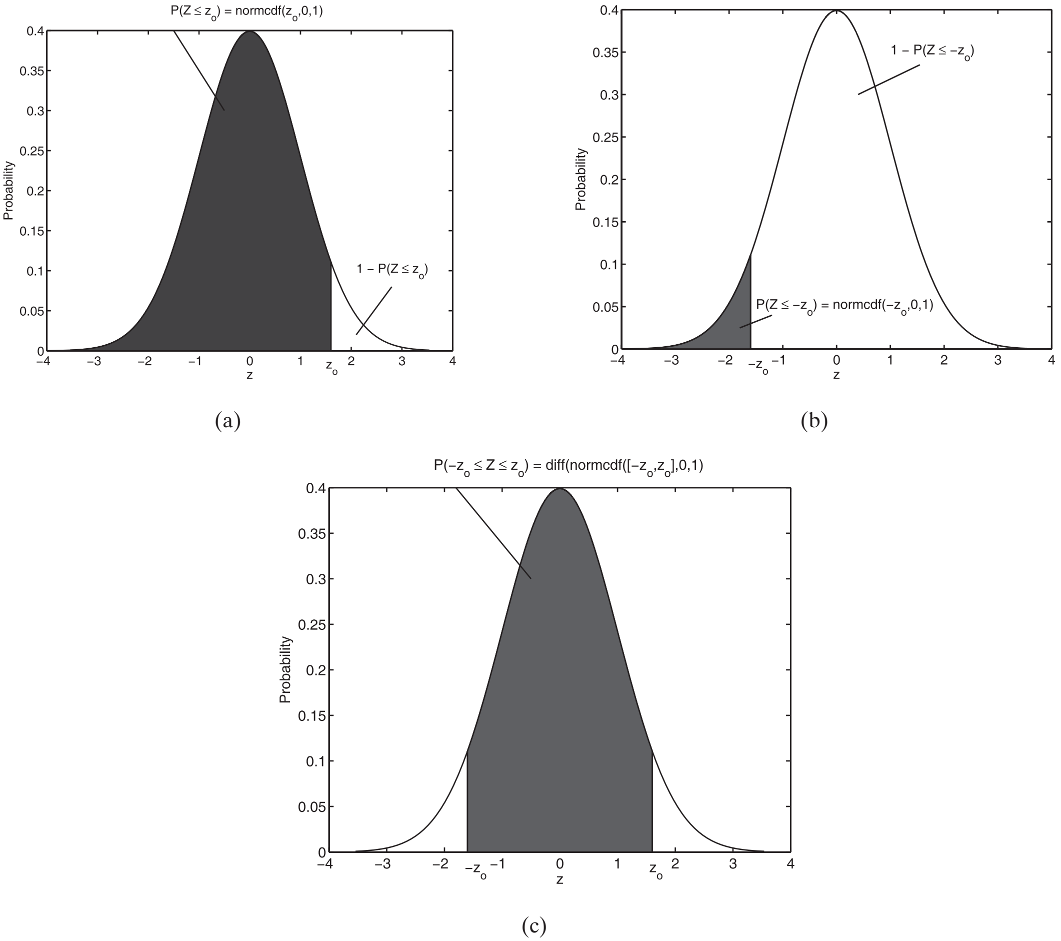
In some instances, one would like to determine the inverse of $\Phi(x)$—that is,

$$x = \Phi^{-1}[P(X \leq x)] \tag{8.19}$$

This is accomplished with

```
norminv(p, mean(x), std(x))
```

where p is the cumulative probability; that is, the shaded area in Figure 8.5a.



(a)

(b)

(c)

Figure 8.5 Relationship of cdf to `normcdf`: (a) $Z \leq z_o$ (b) $Z \leq -z_o$ (c) $-z_o \leq Z \leq z_o$.

In order to determine whether a set of data can be modeled with the normal probability distribution function, one usually plots the data on a normal probability graph in which the ordinate (y-axis) is scaled using the cumulative normal distribution function. This is analogous to plotting data on which the ordinate has been scaled by the logarithm. On a graph in which the ordinate has been scaled logarithmically, an exponential function will appear as a straight line. Similarly, on a graph in which the ordinate has been scaled with the normal cumulative probability function, a process that has its ordered values distributed normally will appear as a straight line. In other words, for a normal distribution, the cumulative probability values that are one standard deviation σ on either side of the mean μ are $P(X \leq \mu + \sigma) = 0.84$ and $P(X \leq \mu - \sigma) = 0.16$, respectively, while that of the mean is $P(X \leq \mu) = 0.5$. Thus, on a probability-transformed graph, the three sets of coordinates $(\mu - \sigma, 0.16)$,

(μ, 0.5), and ($\mu + \sigma$, 0.84) specify three points that lie on a straight line. The values of μ and σ are estimated by Eqs. (8.1) and (8.2), respectively, and computed using `mean` and `std`, respectively.

The procedure for plotting data on a probability graph is as follows. Consider a set of m data values $y_j, j = 1, 2, \ldots, m$. Order the data from the smallest (most negative) to the largest (most positive) value and assign the lowest value the number 1, and the next lowest value the number 2, and so on, with the highest value having the number m. Call these ordered data values $w_j, j = 1, 2, \ldots, m$. Corresponding to each w_j, we assign a cumulative probability of $(j-0.5)/m, j = 1, 2, \ldots, m$; that is, $P(w \leq w_j)$. The coordinates of each data value that is to be plotted on the probability distribution graph are $(w_j, (j-0.5)/m)$. When only a linear graph is available, one plots instead (w_j, z_j), where $z_j =$ `norminv` $((j-0.5)/m)$. The function that performs these computations and does the plotting is

```
normplot(y)
```

where $y = [y_1\, y_2 \ldots y_m]$. The straight line appearing in this plot is determined from the coordinate pairs of the first and third quartiles of y_j and z_j. Recall the determination of $q1$ and $q3$ in Section 8.1 and the interpretation of Figure 8.3.

Example 8.4 Verification of the normality of data

Let us revisit the data in **DataSet81**. First, we replot its histogram and superimpose on this bar graph the corresponding normal probability distribution function. This is accomplished with `histfit`. The script is

```
histfit(DataSet81, 9)
colormap([1, 1, 1,])
```

whose execution results in Figure 8.6. The function `colormap` is used to change the color of the bars to white. Next, we see whether the data are normally distributed. To do these operations, we use

```
normplot(DataSet81)
```

and obtain Figure 8.7. It is seen that a fairly large portion of the data are close to the straight line, leading one to conclude that the normal distribution is a reasonable approximation to these data.

If we accept the normal distribution as an adequate representation of these data, then we can determine the values at which, say, 90% of the data lie. Then, referring to Figures 8.5a and 8.5c and Eq. (8.19), we use `norminv` as follows:

```
m = mean(DataSet81);
s = std(DataSet81);
zh = norminv(.95, m, s)
zl = norminv(.05, m, s)
```

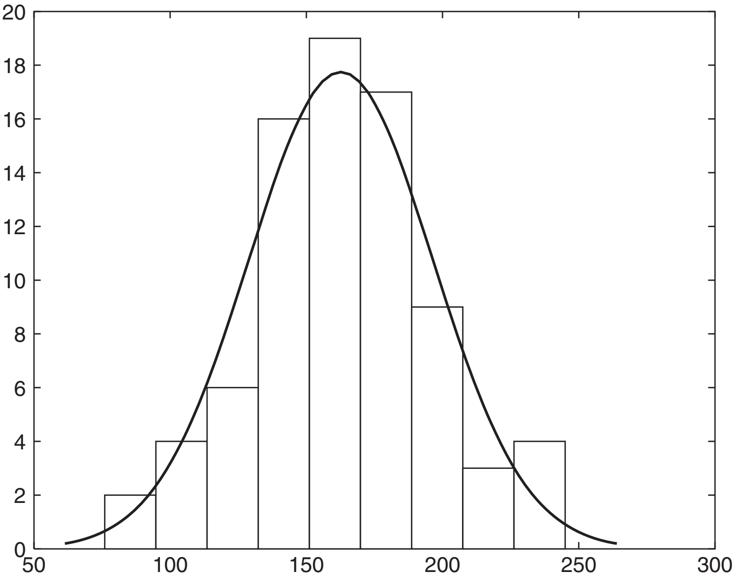


Figure 8.6 Histogram with a superimposed normal probability density function obtained with `histfit`.

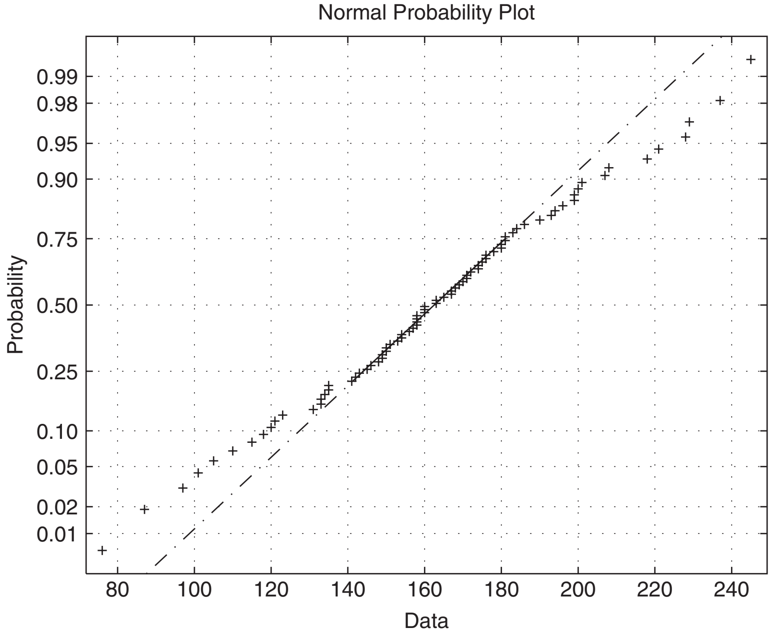


Figure 8.7 Normal cumulative probability plot of **DataSet81** using `normplot`.

Upon execution of this script, we obtain $z_h = 218.2145$ and $z_l = 107.1105$. As a check, we find the difference between the probabilities of these two limits, which are determined from the following script:

```
m = mean(DataSet81);
s = std(DataSet81);
zh = norminv(.95, m, s);
zl = norminv(.05, m, s);
ph = normcdf(zh, m, s)
pl = normcdf(zl, m, s)
```

The execution of this script gives $p_h - p_l = 0.9500 - 0.0500 = 0.90$.

Example 8.5 Normal distribution approximation to the Poisson and binomial distributions

The normal distribution is a good approximation to the binomial distribution when $np > 5$ and $n(1-p) > 5$, and is also a good approximation to the Poisson distribution to obtain $P(X \leq x)$ when $\lambda > 5$. For the case of the binomial distribution, the normal standard random variable given by Eq. (8.17) is (recall Eq. (8.8))

$$z_b = \frac{X - np}{\sqrt{np(1-p)}}$$

and that for the Poisson distribution is (recall Eq. (8.10))

$$z_p = \frac{X - \lambda}{\sqrt{\lambda}}$$

To illustrate the normal distribution approximation to the Poisson distribution, we return to Example 8.3, where we determined the probability that a hospital will treat nine or fewer heart attacks a day is 0.8712. We now solve this problem using the normal distribution. Since $\mu = 46/7$ and $\sigma = \sqrt{46/7}$, the script is

```
P = normcdf((9–46/7)/sqrt(46/7), 0, 1)
```

which gives $P = 0.8283$, a result that is in fair agreement with the correct value. In order to visualize this approximation, we create the following script, which draws the Poisson distribution from Example 8.3 and its approximation is given by the above expression.

```
x = 1:16;
y = linspace(0, 16, 50);
yPoisson = poisspdf(x, 46/7);
NormApprox = normpdf(y, 46/7, sqrt(46/7));
plot([x; x], [zeros(1,16); yPoisson], 'k', y , NormApprox, 'k')
xlabel('Number of aircraft launched on time')
ylabel('Probability')
```

The results from the execution of this script are shown in Figure 8.8.

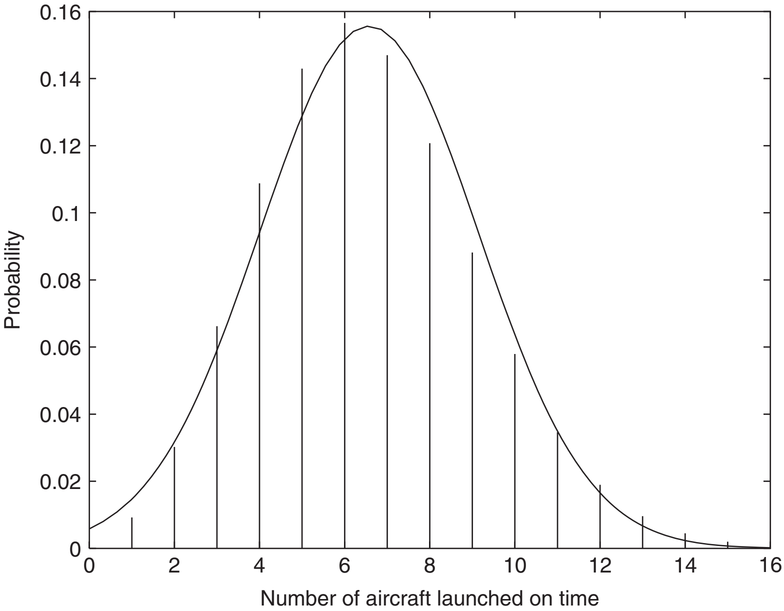


Figure 8.8 Poisson distribution of Example 8.3 and its approximation using the normal probability distribution function as indicated in Example 8.5.

Weibull Distribution

The Weibull probability distribution function is

$$f_w(x) = \alpha\beta x^{\beta-1}e^{-\alpha x^{\beta}} \quad x > 0 \tag{8.20}$$

where $\alpha > 0$ is a scale parameter and $\beta > 0$ is a shape parameter. (Another notation that is commonly used is obtained with the transformation $\alpha = \delta^{-\beta}$.) When $\beta = 1$, Eq. (8.20) reduces to the exponential distribution, and when $\beta = 2$, to the Rayleigh distribution. The mean value of the Weibull distribution is

$$\mu_W = \alpha^{-1/\beta}\Gamma\left(1 + \frac{1}{\beta}\right)$$

and its variance is

$$\sigma_W^2 = \alpha^{-2/\beta}\Gamma\left(1 + \frac{2}{\beta}\right) - \alpha^{-2/\beta}\left[\Gamma\left(1 + \frac{1}{\beta}\right)\right]^2$$

where $\Gamma(x)$ is the gamma function. The Weibull probability distribution function is obtained from

```
wblpdf(x, alpha, beta)
```

where alpha = α, beta = β, and the size of *Pw* is equal to the size of *x*. The mean and variance are obtained from

```
[muW, VarW] = wblstat(alpha, beta)
```

The Weibull cumulative distribution function $F_w(x)$ is

$$F_w(x) = P(X \le x) = 1 - e^{-\alpha x^{\beta}} \quad x > 0 \tag{8.21}$$

and is obtained from

```
Wcdf = wblcdf(x, alpha, beta)
```

where the size of *Wcdf* is equal to the size of x.

In some instances, one would like to determine the inverse of $F(x)$—that is, when

$$x = F_w^{-1}[P(X \le x)] \tag{8.22}$$

This is accomplished with the function

```
wblinv(p, alpha, beta)
```

where p is the value of the cumulative probability distribution.

Example 8.6 Verification that data can be represented by a Weibull distribution

Consider the sorted data on the longevity of a component given in Table 8.3. We create the following function M file **DataSet82** for these data.

```
function d = DataSet82
d = [72 82 97 103 113 117 126 127 127 139 154 159 199 207]';
```

We now plot these data to determine whether a Weibull distribution can be used to model it. To perform the plotting, we use wblplot as shown in the following script to obtain Figure 8.9.

```
wblplot(DataSet82)
```

It is seen that these data are fairly well represented by the Weibull distribution and, therefore, we shall adopt it as a model for these data.

To determine the values of α and β, we use

```
ab = wblfit(x)
```

where $ab(1) = \alpha$ and $ab(2) = \beta$. The script to obtain the magnitudes of α and β and the mean value and the standard deviation for **DataSet82** is

TABLE 8.3 Sorted Component Life Data: **DataSet82**

Component life	Component life
72	127
82	127
97	139
103	154
113	159
117	199
126	207

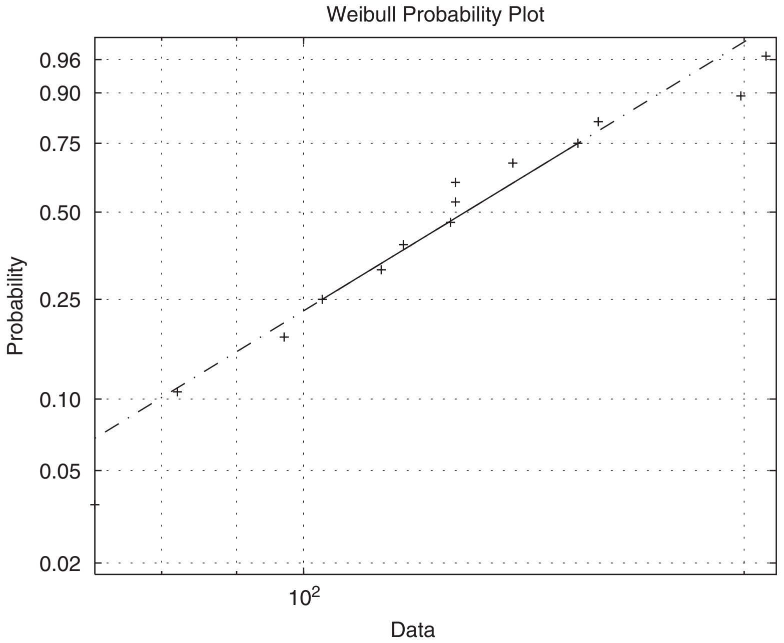


Figure 8.9 Weibull cumulative probability plot of **DataSet82** using `wblplot`.

```
ab = wblfit(DataSet82)
[muW, varW] = wblstat(ab(1), ab(2))
sigW = sqrt(varW)
```

The execution of this script gives that $\alpha = 144.27$, $\beta = 3.6437$, $\mu_W = 130.09$, and $\sigma_W = 39.70$. To compare the mean value and the standard deviation by assuming a Weibull distribution with the mean and standard deviation obtained from Eqs. (8.1) and (8.2), we use the following script:

```
muN = mean(DataSet82)
sigmaN = std(DataSet82)
```

Upon execution, we find that $\mu = 130.1429$ and $\sigma = 39.3854$.

We now plot the Weibull probability distribution function for the data in Table 8.3 and, for comparison, the normal probability density function when these data are assumed normally distributed. The script is

```
ab = wblfit(DataSet82);
xx = linspace(50, 200, 50);
yW = wblpdf(xx, ab(1), ab(2));
yN = normpdf(xx, mean(DataSet82), std(DataSet82));
plot(xx, yW, 'k-', xx, yN, 'k--')
legend('Weibull', 'Normal')
xlabel('x')
ylabel('Probability distribution functions')
```

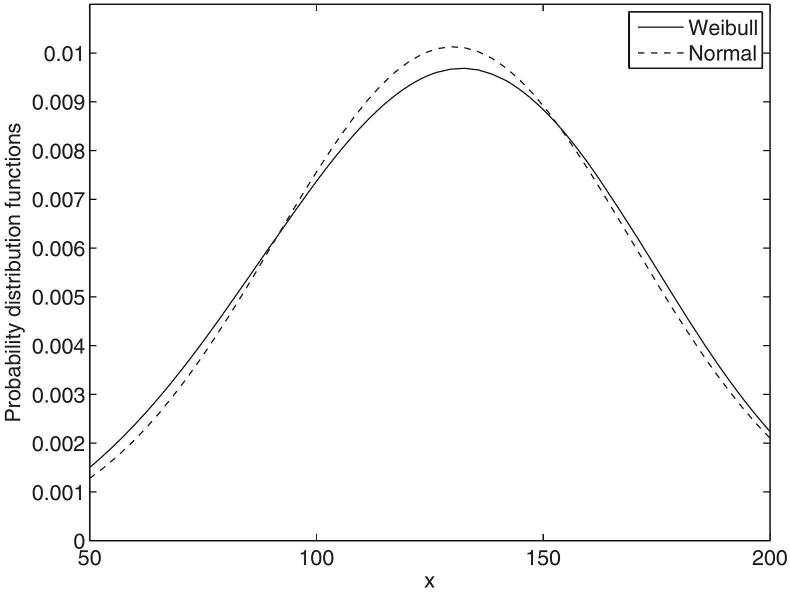


Figure 8.10 Comparison of the Weibull and normal probability density functions for **DataSet82**.

Executing this script results in Figure 8.10.

Finally, we determine the probability that a component's life is less than 100 hours. The script is

```
ab = wblfit(DataSet82);
p = wblcdf(100, ab(1), ab(2))
```

which upon execution yields $p = 0.2312$ or 23.1%.

8.3 CONFIDENCE INTERVALS

Let θ be a numerical value of a statistic (e.g., the mean, variance, or difference in means) of a collection of n samples. What we are interested in is determining the values of l and u such that the following is true

$$P(l \leq \theta \leq u) = 1 - \alpha$$

where $0 < \alpha < 1$. This means that we will have a probability of $1 - \alpha$ of selecting a collection of n samples that will produce an interval that contains the true value of θ. The interval

$$l \leq \theta \leq u$$

is called the $100(1-\alpha)\%$ two-sided confidence interval for θ. The quantities l and u are called the upper and lower confidence limits, respectively. Similarly, the $100(1-\alpha)\%$ one-sided lower confidence interval is

$$l \le \theta$$

and the $100(1-\alpha)\%$ one-sided upper confidence interval is

$$\theta \le u$$

The confidence limits depend on the distribution of the samples and on whether or not the standard deviation of the population is known. Several commonly used relationships to determine these confidence limits are summarized in Table 8.4. In this table, the following definitions are used:

μ and σ are the true mean and standard deviation, respectively.

$\bar{x}$ and s^2 are determined from Eqs. (8.1) and (8.2), respectively.

$t_{\alpha,n-1}$ is the value of the t distribution with $n-1$ degrees of freedom obtained from `tinv`.

TABLE 8.4 Summary of Several Confidence Interval Procedures

		100(1−α)% confidence interval $\hat{\theta} - q \le \theta \le \hat{\theta} + q$			
Type and conditions for σ	Statistic $\hat{\theta}$	θ	q		Case
Mean with σ^2 known	$\bar{x}$	μ	$z_{\alpha/2}\sigma/\sqrt{n}$		1
Difference in means with σ_1^2 and σ_2^2 known	$\bar{x}_1 - \bar{x}_2$	$\mu_1 - \mu_2$	$z_{\alpha/2}\sqrt{\dfrac{\sigma_1^2}{n_1} + \dfrac{\sigma_2^2}{n_2}}$		2
Mean with σ^2 unknown	$\bar{x}$	μ	$t_{\alpha/2,n-1}s/\sqrt{n}$		3
Difference in means with $\sigma_1^2 = \sigma_2^2$ unknown	$\bar{x}_1 - \bar{x}_2$	$\mu_1 - \mu_2$	$t_{\alpha/2,n_1+n_2-2}s_p\sqrt{\dfrac{1}{n_1} + \dfrac{1}{n_2}}$		4[§]
Difference in means with $\sigma_1^2 \ne \sigma_2^2$ unknown	$\bar{x}_1 - \bar{x}_2$	$\mu_1 - \mu_2$	$t_{\alpha/2,\nu}\sqrt{\dfrac{s_1^2}{n_1} + \dfrac{s_2^2}{n_2}}$		5[§]
		100(1−α)% confidence interval $q_1\hat{\theta} \le \theta \le q_2\hat{\theta}$			
	$\hat{\theta}$	θ	q_1	q_2	
Variance	s^2	σ^2	$\dfrac{n-1}{\chi^2_{\alpha/2,n-1}}$	$\dfrac{n-1}{\chi^2_{1-\alpha/2,n-1}}$	6
Ratio of variances	$\dfrac{s_1^2}{s_2^2}$	$\dfrac{\sigma_1^2}{\sigma_2^2}$	$\dfrac{1}{f_{\alpha/2,n_1-1,n_2-1}}$	$f_{\alpha/2,n_2-1,n_1-1}$	7

[§] See text for definitions of ν and s_p.

$z_{\alpha/2}$ is the value of the normal distribution obtained from `norminv`.

$\chi^2_{\alpha/2,n-1}$ is the chi-square distribution with $n - 1$ degrees of freedom obtained from `chi2inv`.

$f_{\alpha/2,n-1,m-1}$ is *f* distribution with $n - 1$ and $m - 1$ degrees of freedom obtained from `finv`.

Furthermore, for Case 4,

$$s_p = \sqrt{\frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}}$$

and for Case 5,

$$\nu = \left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2 \left[\frac{(s_1^2/n_1)^2}{n_1 + 1} + \frac{(s_2^2/n_2)^2}{n_2 + 1}\right]^{-1} - 2$$

Confidence Limits for Cases 3 and 7

For Case 3 in Table 8.4, the two-sided confidence limits are written explicitly as

$$\bar{x}-t_{\alpha/2,\, n-1}s/\sqrt{n} \le \mu \le \bar{x}+t_{\alpha/2,\, n-1}s/\sqrt{n}$$

and for Case 7, they are written explicitly as

$$\frac{s_1^2}{s_2^2}\frac{1}{f_{\alpha/2,n_1-1,n_2-1}} \le \frac{\sigma_1^2}{\sigma_2^2} \le \frac{s_1^2}{s_2^2} f_{\alpha/2,n_2-1,n_1-1}$$

Notice that the degrees of freedom in the subscripts of *f* are reversed.

We now illustrate the determination of the two-sided confidence limits for Cases 3 and 7 of Table 8.4.

Example 8.7 Two-sided confidence limits

Case 3 Consider the data in Table 8.1, which reside in **DataSet81**. If we set the confidence level to 95%, then the script to determine the confidence interval of the mean is

```
meen = mean(DataSet81);
L = length(DataSet81);
q = std(DataSet81)*tinv(0.975, L-1)/sqrt(L);
disp(['Sample mean = ' num2str(meen)])
disp('Confidence interval for sample mean at 95% confidence level –')
disp([' ' num2str(meen-q) ' < = Sample mean < = ' num2str (meen+q)])
```

Upon execution, the following is displayed to the command window

```
Sample mean = 162.6625
Confidence interval for sample mean at 95% confidence level –
   155.1466 <= Sample mean <= 170.1784
```

TABLE 8.5 Data for Case 7: **DataFci**

Set 1	Set 2
41.60	39.72
41.48	42.59
42.34	41.88
41.95	42.00
41.86	40.22
42.18	41.07
41.72	41.90
42.26	44.29
41.81	
42.04	

Another way to obtain this confidence interval is with `ttest`, which is illustrated in Section 8.4.

Case 7 We consider the two columns of data in Table 8.5, which are placed in the function M file **DataFci** shown below.

```
function [set1, set2] = DataFci
set1 = [41.60 41.48 42.34 41.95 41.86 42.18 41.72 42.26 41.81 42.04];
set2 = [39.72 42.59 41.88 42.00 40.22 41.07 41.90 44.29];
```

To determine the confidence interval of the ratio of the sample variances, we use the following script:

```
[data1, data2] = DataFci;
r = var(data1)/var(data2);
L1 = length(data1);
L2 = length(data2);
q2 = r*finv(.975, L2 -1, L1-1);
q1 = r/finv(.975, L1-1, L2-1);
disp(['Ratio of sample variances = ' num2str(r)])
disp('Confidence interval for ratio of sample variances at 95% confidence
level -')
disp([' ' num2str(q1) ' <= Ratio of sample variances <= ' num2str(q2)])
```

Upon execution, we obtain

```
Ratio of sample variances = 0.039874
Confidence interval for ratio of sample variances at 95% confidence level -
  0.0082672 <= Ratio of sample variances <= 0.16736
```

8.4 HYPOTHESIS TESTING

In engineering, there are many situations where one has to either accept or reject a statement (hypothesis) about some parameter. A statistical hypothesis can be thought of as a statement about the parameters of one or more populations. A population is the totality of the observations with which we are concerned. A sample is a subset of a population. Since we use probability distributions to represent populations, a statistical hypothesis can be thought of as a statement about the statistical distribution of the population.

Suppose that we have a parameter θ that has been obtained from n samples of a population, and we are interested in determining whether this parameter is equal to θ_o. The hypothesis testing procedure requires one to

1. Postulate a hypothesis, called the null hypothesis, H_0.
2. Form the appropriate test statistic, q_0.
3. Select a confidence level (recall that $100(1-\alpha)\%$ is the confidence level for θ).
4. Compare the test statistic to a value that corresponds to the magnitude of the test statistic that one can expect to occur naturally, q.

Based on the respective magnitudes of q_0 and q, the null hypothesis is either accepted or rejected. If the null hypothesis is rejected, then we accept an alternative one, which is denoted H_1.

There are three cases to consider:

$$
\begin{array}{lll}
H_0\colon \theta = \theta_0 & H_0\colon \theta = \theta_0 & H_0\colon \theta = \theta_0 \\
H_1\colon \theta \neq \theta_0 & H_1\colon \theta > \theta_0 & H_1\colon \theta < \theta_0
\end{array}
$$

For each case, there are corresponding test statistics $q_0(n, \alpha)$ and $q(n, \alpha)$. Several hypothesis testing procedures are summarized in Table 8.6, which parallel the confidence interval procedures in Table 8.4. The terms appearing in Table 8.6 have been defined in Section 8.3.

Two types of errors that can be made in hypothesis testing are:

Type I: Rejecting the null hypothesis H_0 when it is true.

Type II: Accepting the null hypothesis H_0 when it is false; that is, when really $\theta = \theta_1$.

The probability of making a Type I error is α and that for the Type II is β.

It is common practice to replace the confidence parameter α with a quantity called the p-value, which is the smallest level of significance that would lead to the rejection of the null hypothesis. That is, the smaller the p-value, the less plausible is the null hypothesis.

We now illustrate these concepts with three examples from Table 8.6: Cases 2, 4, and 7.

TABLE 8.6 Several Hypothesis Testing Procedures

Null hypothesis H_0	Alternative hypotheses H_1	Criteria for rejection of H_0	Test statistic	MATLAB function	Case
$\mu = \mu_0$ (σ known)	$\mu \neq \mu_0$ $\mu > \mu_0$ $\mu < \mu_0$	$\lvert z_0 \rvert > z_{\alpha/2}$ $z_0 > z_\alpha$ $z_0 < -z_\alpha$	$z_0 = \dfrac{\bar{x} - \mu_0}{\sigma/\sqrt{n}}$	`ztest`	1
$\mu = \mu_0$ (σ unknown)	$\mu \neq \mu_0$ $\mu > \mu_0$ $\mu < \mu_0$	$\lvert t_0 \rvert > t_{\alpha/2,n-1}$ $t_0 > t_{\alpha,n-1}$ $t_0 < -t_{\alpha,n-1}$	$t_0 = \dfrac{\bar{x} - \mu_0}{s/\sqrt{n}}$	`ttest`	2
$\mu_1 = \mu_2$ (σ_1 and σ_2 known)	$\mu_1 \neq \mu_2$ $\mu_1 > \mu_2$ $\mu_1 < \mu_2$	$\lvert z_0 \rvert > z_{\alpha/2}$ $z_0 > z_\alpha$ $z_0 < -z_\alpha$	$z_0 = \dfrac{\bar{x}_1 - \bar{x}_2}{\sqrt{\dfrac{\sigma_1^2}{n_1} + \dfrac{\sigma_2^2}{n_2}}}$		3
$\mu_1 = \mu_2$ ($\sigma_1 = \sigma_2$ unknown)	$\mu_1 \neq \mu_2$ $\mu_1 > \mu_2$ $\mu_1 < \mu_2$	$\lvert t_0 \rvert > t_{\alpha/2,n_1+n_2-2}$ $t_0 > t_{\alpha,n_1+n_2-2}$ $t_0 < t_{\alpha,n_1+n_2-2}$	$t_0 = \dfrac{\bar{x}_1 - \bar{x}_2}{s_p\sqrt{\dfrac{1}{n_1} + \dfrac{1}{n_2}}}$	`ttest2`	4
$\mu_1 = \mu_2$ ($\sigma_1 \neq \sigma_2$ unknown)	$\mu_1 \neq \mu_2$ $\mu_1 > \mu_2$ $\mu_1 < \mu_2$	$\lvert t_0 \rvert > t_{\alpha/2,v}$ $t_0 > t_{\alpha,v}$ $t_0 < -t_{\alpha,v}$	$t_0 = \dfrac{\bar{x}_1 - \bar{x}_2}{\sqrt{\dfrac{s_1^2}{n_1} + \dfrac{s_2^2}{n_2}}}$		5
$\sigma^2 = \sigma_0^2$	$\sigma^2 \neq \sigma_0^2$ $\sigma^2 > \sigma_0^2$ $\sigma^2 < \sigma_0^2$	$\chi_0^2 > \chi_{\alpha/2,n-1}^2$ $\chi_0^2 > \chi_{\alpha,n-1}^2$ $\chi_0^2 < \chi_{1-\alpha,n-1}^2$	$\chi_0^2 = \dfrac{(n-1)s^2}{\sigma_0^2}$	`vartest`	6
$\sigma_1^2 = \sigma_2^2$	$\sigma_1^2 \neq \sigma_2^2$	$f_0 > f_{\alpha/2,n_1-1,n_2-1}$ or $f_0 < f_{1-\alpha/2,n_1-1,n_2-1}$	$f_0 = \dfrac{s_1^2}{s_2^2}$	`vartest2`	7
	$\sigma_1^2 > \sigma_2^2$	$f_0 > f_{\alpha,n_1-1,n_2-1}$			

Example 8.8 Test for statistical significance of the mean and the variance

Case 2 Consider the data in **DataSet81**, which appear in Table 8.1. We want to know whether there is a statistically significant difference between the sample mean and a mean value of 168 ($\mu_0 = 168$) at a 95% confidence level. Thus, the hypothesis is

$$H_0\colon \mu = 168$$

$$H_1\colon \mu \neq 168$$

We use `ttest` to determine the validity of this hypothesis. The `ttest` function is

```
[h, p, ci] = ttest(Data, muzero, alpha)
```

where *Data* are the data, *muzero* $= \mu_0$, *alpha* $= \alpha$, $h = 0$ if H_0 and $h = 1$ if H_1, $p = p$-value; that is,

```
p = 2*(1-tcdf(t0, n-1));
```

for a two-sided confidence interval; $t0 = t_0$ is defined in the fourth column of case 2 of Table 8.6, and $ci(1) = l$ and $ci(2) = u$ are the lower and upper confidence limits, respectively. Thus, for the present case,

```
[h, p, ci] = ttest(DataSet81, 168, 0.05)
```

Upon execution, we find that $h = 0$; that is, we cannot reject the null hypothesis, $p = 0.1614$, $ci(1) = 155.1466$, and $ci(2) = 170.1784$. Recall that from Example 8.7 we had determined that $\bar{x} = 162.6625$ and that the confidence interval for this value at the 95% confidence level is $155.1466 \le \bar{x} \le 170.1784$. Since the hypothesized value 168 for the mean lies within this confidence interval, we should expect that the null hypothesis would not be rejected. In fact, based on its p-value, we see that we are only $100(1-0.1614) = 83.9\%$ confident, which is less than our desired confidence level of 95%.

On the other hand, if our null hypothesis is

$$H_0\text{: } \mu = 175$$

$$H_1\text{: } \mu \neq 175$$

then

```
[h, p, ci] = ttest(DataSet81, 175, 0.05)
```

gives $h = 1$; that is, we reject the null hypothesis and adopt H_1; $p = 0.0016$, $ci(1) = 155.1466$, and $ci(2) = 170.1784$. In other words, we can be $100(1-0.0016) = 99.84\%$ confident that the mean of the data in **DataSet81** is different from the mean value of 175.

Case 4 Consider the data in **DataFci**, which appear in Table 8.5. We want to determine whether there is any statistically significant difference between the means of these samples at a 95% confidence level. Thus, the hypothesis is

$$H_0\text{: } \mu_1 = \mu_2$$

$$H_1\text{: } \mu_1 \neq \mu_2$$

We use `ttest2` to determine the validity of this hypothesis. The `ttest2` function is

```
[h, p, ci] = ttest2(x1, x2, alpha)
```

where $x1$ and $x2$ are the data, $alpha = \alpha$, $h = 0$ if H_0 and $h = 1$ if H_1, $p = p$-value; that is,

```
p = 2*(1-tcdf(t0, n-1))
```

for a two-sided confidence interval; $t0 = t_0$ is defined in the fourth column of case 4 of Table 8.6, and $ci(1) = l$ and $ci(2) = u$ are the lower and upper confidence limits, respectively. Thus, the script is

```
[x1, x2] = DataFci;
[h, p, ci] = ttest2(x1, x2, 0.05)
```

Executing this script yields $h = 0$; that is, we cannot reject the null hypothesis, $p = 0.6445$, $ci(1) = -0.7550$, and $ci(2) = 1.1855$ are the lower and upper confidence limits, respectively, on the difference between the means. Based on the p-value, we see that we are only 100(1-0.6445) = 35.55% confident that there is a statistically significant difference between the means, which is substantially less than our desired confidence level of 95%. Therefore, the null hypothesis cannot be rejected.

Case 7 Consider the data in **DataFci,** which appear in Table 8.5. We want to know whether there is any statistically significant difference between the variances of these samples at a 95% confidence level. Thus, the hypothesis is

$$H_0\colon \sigma_1^2 = \sigma_2^2$$
$$H_1\colon \sigma_1^2 \neq \sigma_2^2$$

The test statistic is

$$f_0 = \frac{s_1^2}{s_2^2}$$

and the criterion for rejection of the null hypothesis is either

$$f_0 > f_{\alpha/2,n_1-1,n_2-1} \quad \text{or} \quad f_0 < f_{1-\alpha/2,n_1-1,n_2-1}$$

We use `vartest2` to determine the validity of this hypothesis; that is,

```
[h, p, ci] = vartest2(x1, x2, alpha)
```

where $x1$ and $x2$ are the data, $alpha = \alpha$, $h = 0$ if H_0 and $h = 1$ if H_1, $p = p$-value—that is,

```
p = 2*(1-fcdf(f0, n1, n2))
```

for a two-sided confidence interval; $f0 = f_0$, and $ci(1) = l$ and $ci(2) = u$ are the lower and upper confidence limits, respectively. Thus, the script is

```
[x1, x2] = DataFci;
[h, p, ci] = vartest2(x1, x2, 0.05)
```

Executing this script yields $h = 1$; that is, we reject the null hypothesis, $p = 6.5379 \times 10^{-5}$, $ci(1) = 0.0083$, and $ci(2) = 0.1674$ are the lower and upper confidence limits, respectively, on the ratio of the variances. Based on the p-value, we see that we are $100(1-6.5379 \times 10^{-5})$ = 99.993% confident that there is a statistically significant difference in their variances.

8.5 LINEAR REGRESSION

8.5.1 Simple Linear Regression

Regression analysis is a statistical technique for modeling and investigating the relationship between two or more variables. A simple linear regression model has only one independent variable. If the input to a process is x and its response y, then a linear model is

$$y = \beta_1 x + \beta_0$$

If there are n values of the independent variable x_i and n corresponding measured responses y_i, $i = 1, 2, \ldots, n$, then estimates of y are obtained from

$$\hat{y} = \hat{y}(x) = \hat{\beta}_1 x + \hat{\beta}_0 \quad x_{\min} \le x \le x_{\max} \tag{8.23}$$

where $x_{\min}$ is the minimum value of x_i, $x_{\max}$ is the maximum value of x_i, and $\hat{\beta}_1$ and $\hat{\beta}_0$ are the estimates of β_1 and β_0, respectively, and are given by

$$\hat{\beta}_1 = \frac{S_{xy}}{S_{xx}} \tag{8.24}$$

$$\hat{\beta}_0 = \overline{\mathrm{y}} - \hat{\beta}_1 \bar{x}$$

where

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{8.25}$$

$$S_{xx} = \sum_{i=1}^{n} x_i^2 - n\bar{x}^2 \quad S_{xy} = \sum_{i=1}^{n} x_i y_i - n\overline{xy}$$

The values for $\hat{\beta}_1$ and $\hat{\beta}_0$ are obtained from `polyfit` (recall Section 5.4.2). Thus,

```
[c, ss] = polyfit(x, y, 1)
```

where $c(1) = \hat{\beta}_1$ and $c(2) = \hat{\beta}_0$, and ss is a quantity needed by `polyconf`, which is described below.

The $100(1 - \alpha)\%$ confidence limits on the estimate of $y(x)$, for $x_{\min} \le x \le x_{\max}$, are

$$\hat{y}(x) - w(x) \le y(x) \le \hat{y}(x) + w(x) \tag{8.26}$$

where

$$w(x) = t_{\alpha/2,n-2}\hat{\sigma}\sqrt{1 + \frac{1}{n} + \frac{(x - \bar{x})^2}{S_{xx}}} \tag{8.27}$$

$$\hat{\sigma}^2 = \frac{\mathrm{SS}_E}{n - 2} \qquad \mathrm{SS}_E = S_{yy} - \hat{\beta}_1 S_{xy}$$

$$S_{yy} = \sum_{i=1}^{n} y_i^2 - n\bar{y}^2$$

The quantities $w(x)$ and $\hat{y}(x)$ are obtained from `polyconf` as follows:

```
[c, ss] = polyfit(x, y, 1)
[yhat, w] = polyconf(c, x, ss, alpha)
```

where *yhat* $= \hat{y}(x)$, $w = w(x)$, and *alpha* $= \alpha$. The vector x determines the values at which *yhat* and w are evaluated.

One means of determining the adequacy of the model given by Eq. (8.23) is to examine its residuals, which are determined from

$$e_i = y_i - \hat{y}(x_i) \quad i = 1, 2, \ldots, n \tag{8.28}$$

If e_i are approximately normally distributed, then the model has been correctly applied.

Another indicator of the model's representation of the data is the coefficient of determination R^2, which is given by

$$R^2 = 1 - \frac{\text{SS}_E}{S_{yy}} \tag{8.29}$$

The value $100R^2$ is the percentage of the variability of the data that is accounted for by the model. The closer this value is to 100%, the better the model. The quantity R is called the correlation coefficient.

We now illustrate the use of these relationships.

Example 8.9 Regression analysis

Consider the data given in Table 8.7. These data are placed in a function M file **DataRegress1**. Notice, however, that these data are not ordered. Since this is inconvenient when it comes time to plot them with connected straight lines, we sort them in ascending order. Neither `polyfit` nor `polyconf` requires the sorting. Thus,

```
function [x, y] = DataRegress1
xx = [2.38 2.44 2.70 2.98 3.32 3.12 2.14 2.86 3.50 3.20 2.78 2.70 2.36 2.42 . . .
      2.62 2.80 2.92 3.04 3.26 2.30];
yy = [51.11 50.63 51.82 52.97 54.47 53.33 49.90 51.99 55.81 52.93 52.87 52.36 . . .
      51.38 50.87 51.02 51.29 52.73 52.81 53.59 49.77];
[x, index] = sort(xx);
y = yy(index);
```

TABLE 8.7 Data for Simple Linear Regression: **DataRegress1**

x	y	x	y
2.38	51.11	2.78	52.87
2.44	50.63	2.70	52.36
2.70	51.82	2.36	51.38
2.98	52.97	2.42	50.87
3.32	54.47	2.62	51.02
3.12	53.33	2.80	51.29
2.14	49.90	2.92	52.73
2.86	51.99	3.04	52.81
3.50	55.81	3.26	53.59
3.20	52.93	2.30	49.77

where *index* gives the original position of each element of x prior to being sorted. This technique has to be used because the correspondence of the elements in x and y must be preserved. If two sort functions were used, one on x and the other on y, then this correspondence would be lost.

We now present a script that determines $\hat{\beta}_1$ and $\hat{\beta}_0$, plots $\hat{y}(x)$ and its confidence limits at the 95% level, plots the data and connects their values to $\hat{y}(x)$, and includes the appropriate annotation. In addition, the value for the coefficient of determination will be computed and placed on the graph.

```
[x, y] = DataRegress1;
[c, s] = polyfit(x, y, 1);
[yhat, w] = polyconf(c, x, s, 0.05);
syy = sum(y.^2)-length(x)*mean(y)^2;
sse = syy-c(1)*(sum(x.*y)-length(x)*mean(x)*mean(y));
plot(x, yhat, 'k-', x, yhat-w, 'k—', x, yhat+w ,'k—', x, y, 'ks', [x; x], [yhat; y], 'k-')
legend('Regression line', '95% confidence interval of y', 'Location', 'SouthEast')
axis([2, 3.6, 48, 57])
xlabel('x (Input)')
ylabel('y (Response)')
text(2.1, 56, ['Coefficient of determination R^2 = ' num2str(1-sse/syy,3)])
```

Upon execution, we obtain the results shown in Figure 8.11.

We proceed further and investigate the residuals. We first compute the residuals and then plot them using normplot (recall Figure 8.7) to determine whether they are normally distributed. The program is

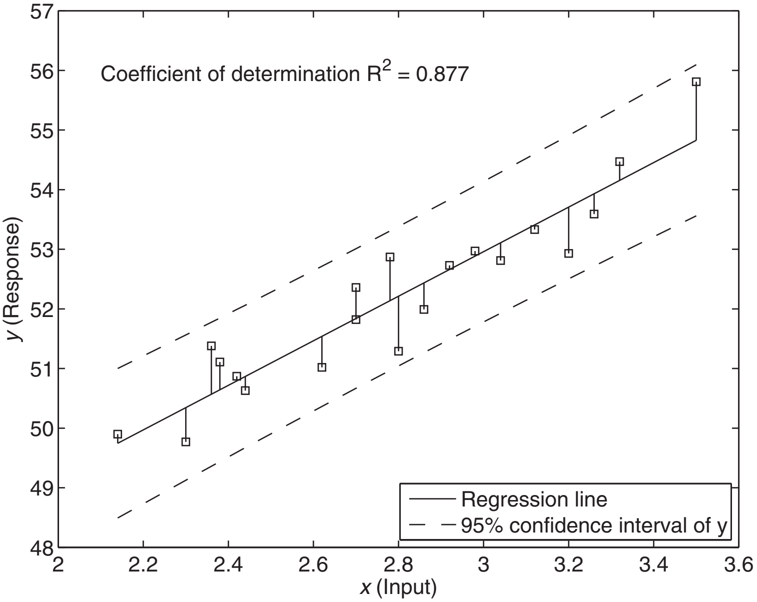


Figure 8.11 Linear regression for the data in Table 8.7 and the confidence limits on y.

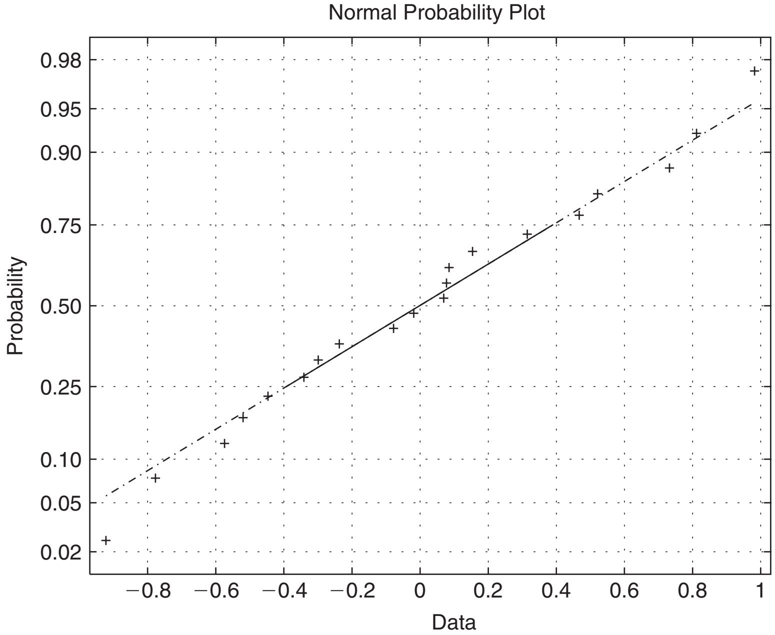


Figure 8.12 Normal cumulative distribution plot of the residuals from the fitted line appearing in Figure 8.11.

```
[x, y] = DataRegress1;
normplot(y-polyval(polyfit(x, y, 1), x))
```

which upon execution results in Figure 8.12. Since the residuals are very close to the line representing the normal distribution, we can say that the residuals are very nearly normally distributed and, therefore, our model is adequate.

8.5.2 Multiple Linear Regression

There are many applications where more than one independent factor (variable) affects the outcome of a process. In this situation, we require a multiple regression model. Consider a process that has one output y and k inputs x_j, $j = 1, 2, \ldots, k$. This process can be modeled as

$$y = \beta_o + \sum_{j=1}^{k} \beta_j x_j \tag{8.30}$$

which is called a multiple linear regression model with k independent variables. The parameters β_j, $j = 0, 1, 2, \ldots, k$ are the regression coefficients. Models that are more complex in appearance may often be analyzed with this multiple linear regression model. For example, suppose that we have a cubic polynomial in *one* independent variable x

$$y = \beta_o + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$$

If we let $x_1 = x, x_2 = x^2$, and $x_3 = x^3$, then we have the linear model shown in Eq. (8.30); that is,

$$y = \beta_o + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3$$

However, this class of models is more easily solved with `polyfit`.

Another example is

$$y = \beta_o + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1^2 + \beta_4 x_2^2 + \beta_5 x_1 x_2$$

which is of the form of Eq. (8.30) when we set $x_3 = x_1^2, x_4 = x_2^2$, and $x_5 = x_1 x_2$. Thus, we see that any regression model that is linear in the parameters β_j is a linear regression model, regardless of the shape of the surface y that it generates.

In order to estimate the parameters, we run an experiment n times, $n > k + 1$, such that for each set of $x_{ij}, i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, k$, we obtain a corresponding set of outputs y_i. In tabular form, this would appear as

y	x_1	x_2	$\ldots$	x_k
y_1	x_{11}	x_{12}	$\ldots$	x_{1k}
y_2	x_{21}	x_{22}	$\ldots$	x_{2k}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
y_n	x_{n1}	x_{n2}	$\ldots$	x_{nk}

Then, Eq. (8.30) becomes

$$y_i = \beta_o + \sum_{j=1}^{k} \beta_j x_{ij} \quad i = 1, 2, \ldots, n \tag{8.31}$$

If these data are arranged in the following matrix form

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} & \ldots & x_{1k} \\ 1 & x_{21} & x_{22} & \ldots & x_{2k} \\ \vdots & \vdots & & & \vdots \\ 1 & x_{n1} & x_{n2} & \ldots & x_{nk} \end{bmatrix} \quad y = \begin{Bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{Bmatrix} \quad \hat{\beta} = \begin{Bmatrix} \hat{\beta}_0 \\ \hat{\beta}_1 \\ \vdots \\ \hat{\beta}_k \end{Bmatrix} \tag{8.32}$$

then the solution for the estimates of β_j, denoted $\hat{\beta}_j$, are obtained from the solution of the following matrix equation:

$$\hat{\beta} = (X'X)^{-1}X'\mathrm{y} \tag{8.33}$$

The matrix X is, in general, not square. Then,

$$\hat{y}_i = \hat{\beta}_o + \sum_{j=1}^{k} \hat{\beta}_j \, x_{ij} \qquad i = 1, 2, \ldots, n \tag{8.34}$$

where $\hat{y}_i$ is the estimate of y_i.

Once the regression coefficients have been obtained, one indication of the adequacy of the model is to compute the residuals and see whether they are normally distributed. The residuals are defined as

$$e = y - \hat{y}$$

Thus,

$$e_i = y_i - \hat{y}_i = y_i - \hat{\beta}_o - \sum_{j=1}^{k} \hat{\beta}_j x_{ij} \quad i = 1, 2, \ldots, n \tag{8.35}$$

The confidence limits on the regression coefficients β_j are given by

$$\beta_{Lj} \le \beta_j \le \beta_{Uj} \quad j = 0, 1, \ldots, k \tag{8.36}$$

where

$$\begin{aligned} \beta_{Lj} &= \hat{\beta}_j - t_{\alpha/2,\, n-k-1}\, \hat{\sigma}\sqrt{C_{jj}} \\ \beta_{Uj} &= \hat{\beta}_j + t_{\alpha/2, n-k-1}\, \hat{\sigma}\sqrt{C_{jj}} \end{aligned} \tag{8.37}$$

and

$$\hat{\sigma}^2 = \frac{y'y - \hat{\beta}'X'y}{n - k - 1}$$

$$\mathrm{C} = (X'X)^{-1} = \begin{bmatrix} C_{00} & C_{01} & \cdots & C_{0k} \\ C_{10} & C_{11} & & \\ \vdots & & \ddots & \\ C_{k0} & & & C_{kk} \end{bmatrix} \tag{8.38}$$

In other words,

$$\mathrm{var}(\hat{\beta}_j) = \hat{\sigma}^2 C_{jj} \quad j = 0, 1, \ldots, k$$

is an estimate of the variance of $\hat{\beta}_j$ and

$$\mathrm{covar}(\hat{\beta}_i, \hat{\beta}_j) = \hat{\sigma}^2 C_{ij} \quad i, j = 0, 1, \ldots, k \quad i \ne j$$

is an estimate of the covariance of $\hat{\beta}_i$ and $\hat{\beta}_j$.

The multiple determination coefficient R^2 is given by

$$R^2 = 1 - \frac{y'y - \hat{\beta}'X'y}{y'y - n\bar{y}^2} \tag{8.39}$$

where

$$\bar{y} = \frac{1}{n}\sum_{j=1}^{n} y_j$$

The quantity R is the correlation coefficient.

One can also perform a hypothesis test to determine whether there exists a linear relationship between at least one regressor variable (x_i, $i = 1, 2, \ldots, k$) and the response y. The hypothesis test is

$$H_0\colon \beta_1 = \beta_2 = \cdots = \beta_k = 0$$

$$H_1\colon \beta_j \neq 0 \text{ for at least one } j$$

Rejection of H_0 implies that at least one regressor variable makes a statistically significant contribution. The test statistic is

$$F_0 = \frac{(\hat{\beta}'X'y - n\bar{y}^2)/k}{(y'y - \hat{\beta}'X'y)/(n-k-1)} \qquad n > k+1 \tag{8.40}$$

We reject H_0 if

$$F_0 > f_{\alpha, k, n-k-1}$$

The numerical evaluation of these equations can be obtained from either

beta = `regress`(y, x)

or

[beta, betacl, e, ecl, stats] = `regress`(y, X, alpha)

where

beta = $[\hat{\beta}_0\, \hat{\beta}_1 \ldots \hat{\beta}_k]$ as defined by Eq. (8.33)
betacl = $((k+1) \times 2)$ array of the lower and upper confidence limits β_L and β_U, respectively, as defined by Eq. (8.37) and whose order corresponds to that of *beta*
e = $[e_1\, e_2 \ldots e_n]$ are the residuals given by Eq. (8.35);
ecl = the confidence limits on the residuals
stats = $[R^2, F_0, p]$, where
R^2 is given by Eq. (8.39)
F_0 is given by Eq. (8.40)
p is the p-value corresponding to F_0—that is, $p = 1 -$ `fcdf`(F0, k, n-k-1)
$y = [y_1 y_2 \ldots y_n]'$ is the column vector of responses
$X = X$ as defined by Eq. (8.32)
alpha = α

We now illustrate the use of these formulas.

Example 8.10 Multiple regression analysis

Consider the data in Table 8.8. We shall fit the following model to these data:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_1 x_2 + \beta_5 x_1 x_3 + \beta_6 x_2 x_3 + \beta_7 x_1^2 + \beta_8 x_2^2 + \beta_9 x_3^2$$

First, we create the function M file **DataMultiRegress1** to create ***X*** according to Eq. (8.32).

TABLE 8.8 Data for Multiple Linear Regression: **DataMultiRegress1**

y	x_1	x_2	x_3	y	x_1	x_2	x_3
0.22200	7.3	0.0	0.0	0.10100	7.3	2.5	6.8
0.39500	8.7	0.0	0.3	0.23200	8.5	2.0	6.6
0.42200	8.8	0.7	1.0	0.30600	9.5	2.5	5.0
0.43700	8.1	4.0	0.2	0.09230	7.4	2.8	7.8
0.42800	9.0	0.5	1.0	0.11600	7.8	2.8	7.7
0.46700	8.7	1.5	2.8	0.07640	7.7	3.0	8.0
0.44400	9.3	2.1	1.0	0.43900	10.3	1.7	4.2
0.37800	7.6	5.1	3.4	0.09440	7.8	3.3	8.5
0.49400	10.0	0.0	0.3	0.11700	7.1	3.9	6.6
0.45600	8.4	3.7	4.1	0.07260	7.7	4.3	9.5
0.45200	9.3	3.6	2.0	0.04120	7.4	6.0	10.9
0.11200	7.7	2.8	7.1	0.25100	7.3	2.0	5.2
0.43200	9.8	4.2	2.0	0.00002	7.6	7.8	20.7

```
function [y, X] = DataMultiRegress1
y = [0.22200 0.39500 0.42200 0.43700 0.42800 0.46700 0.44400 0.37800 0.49400 . . .
    0.45600, 0.45200, 0.11200, 0.43200, 0.10100, 0.23200, 0.30600, 0.09230, 0.11600, . . .
    0.07640, 0.43900, 0.09440, 0.11700, 0.07260, 0.04120, 0.25100, 0.00002]';
x1 = [7.3, 8.7, 8.8, 8.1, 9.0, 8.7, 9.3, 7.6, 10.0, 8.4, 9.3, 7.7, 9.8, 7.3, 8.5, 9.5, 7.4, 7.8, 7.7, 10.3, . . .
    7.8, 7.1, 7.7, 7.4, 7.3, 7.6]';
x2 = [0.0, 0.0, 0.7, 4.0, 0.5, 1.5, 2.1, 5.1, 0.0, 3.7, 3.6, 2.8, 4.2, 2.5, 2.0, 2.5, 2.8, 2.8, 3.0, 1.7, . . .
    3.3, 3.9, 4.3, 6.0, 2.0, 7.8]';
x3 = [0.0, 0.3, 1.0, 0.2, 1.0, 2.8, 1.0, 3.4, 0.3, 4.1, 2.0, 7.1, 2.0, 6.8, 6.6, 5.0, 7.8, 7.7, 8.0, . . .
    4.2, 8.5, 6.6, 9.5, 10.9, 5.2, 20.7]';
X = [ones(length(y),1), x1, x2, x3, x1.*x2, x1.*x3, x2.*x3, x1.^2 x2.^2, x3.^2];
```

Next, we determine the estimates of the coefficients $\hat{\beta}_j$ and their confidence limits at the 95% confidence level, display the values of R^2, F_0 and its p-value, and plot the residuals to determine whether they are normally distributed. The residuals will be plotted in two ways: (1) using `normplot` to determine if they are normally distributed, and (2) using `rcoplot` to display the residuals at all twenty-six combinations of input values x_1, x_2, and x_3 and their error bounds based on their confidence limits. In addition, we shall plot the surface y over the range of x_1 and x_2 for two values of x_3. The script is

```
function Example8_10
[y, X] = DataMultiRegress1;
[b, bcl, e, ecl, stat] = regress(y, X, 0.05);
lenb = length(b);
disp('Regression coefficients and their confidence limits')
disp([num2str(bcl(:,1)) repmat(' <= beta(', lenb, 1) num2str((0:lenb-1)') . . .
   repmat(') = ', lenb, 1) num2str(b) repmat(' <= ', lenb, 1) num2str(bcl(:,2))])
disp(['Coefficient of determination R^2 = ' num2str(stat(1))])
disp(['Test statistic F0 = ' num2str(stat(2)) ' and corresponding p-value = ' . . .
   num2str(stat(3))])
```

```
figure(1)
rcoplot(e, ecl)
figure(2)
normplot(e)
figure(3)
[x1, x2] = meshgrid(linspace(7.1, 10.3, 10), linspace(0.0, 7.8, 15));
x3 = [0, 18];    x12 = [0.5, 0.2];
for k = 1:2
  mesh(x1, x2, Surface(x1, x2, x3(k), b))
  text(10, -0.7, x12(k), ['x_3=' num2str(x3(k))])
  hold on
end
view([-63 26])
xlabel('x_1')
ylabel('x_2')
zlabel('y')

function z = Surface(x1, x2, x3, beta)
z = beta(1)+ beta(2)*x1+ beta(3)*x2+ beta(4)*x3+ beta(5)*x1.*x2+ . . .
  beta(6)*x1.*x3+beta(7)*x2.*x3+ beta(8)*x1.^2+beta(9)*x2.^2+beta(10)*x3.^2;
```

Executing this script displays the following information to the command window and plots the results shown in Figures 8.13–8.15. It is seen from Figure 8.13 that all but five residuals fall close to the line representing the normal distribution. Therefore, the model is adequate.

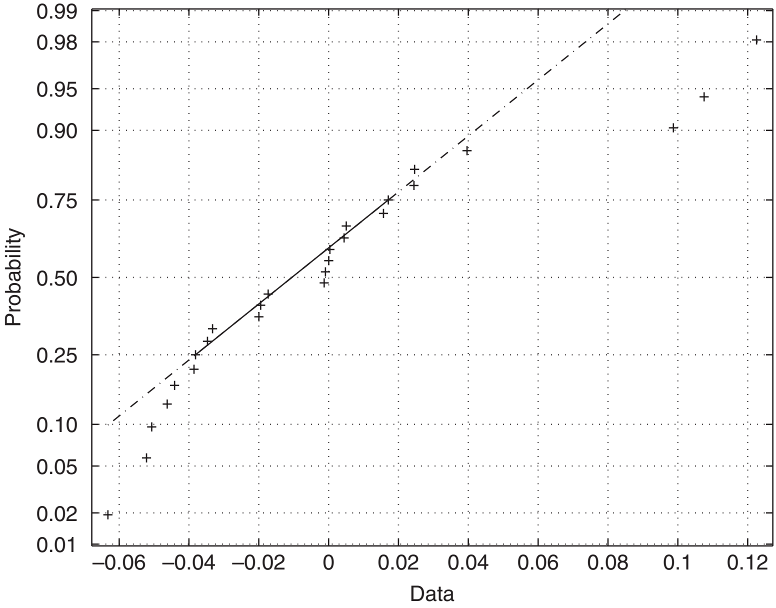


Figure 8.13 Normal cumulative distribution of the residuals from the surface modeling of the data in Table 8.8.

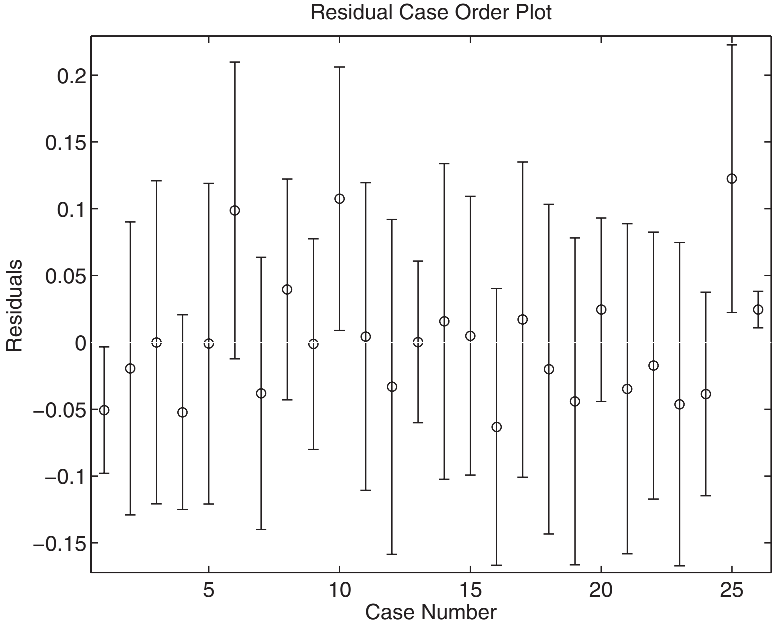


Figure 8.14 Residuals from the surface modeling of the data in Table 8.8 using `rcoplot`.

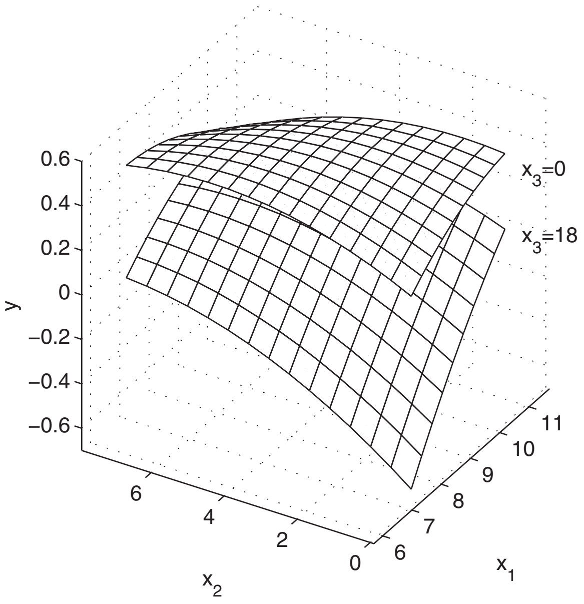


Figure 8.15 Response surfaces of the data in Table 8.8.

```
Regression coefficients and their confidence limits
−4.4976        <=beta(0) = −1.7694         <=0.9589
−0.20282       <=beta(1) = 0.4208          <=1.0444
−0.054708      <=beta(2) = 0.22245         <=0.49961
−0.27691       <=beta(3) = −0.128          <=0.020918
−0.045395      <=beta(4) = −0.019876       <=0.0056419
−0.0070049     <=beta(5) = 0.0091515       <=0.025308
−0.012346      <=beta(6) = 0.0025762       <=0.017499
−0.054932      <=beta(7) = −0.019325       <=0.016283
−0.032989      <=beta(8) = −0.0074485      <=0.018092
−0.002231      <=beta(9) = 0.00082397      <=0.003879
Coefficient of determination R^2 = 0.91695
Test statistic F0 = 19.628 and corresponding p-value = 5.0513e-007
```

8.6 DESIGN OF EXPERIMENTS

8.6.1 Single-Factor Experiments: Analysis of Variance

Consider a single-factor experiment with the factor denoted by A. We run an experiment varying A at a different levels, A_j, $j = 1, 2, \dots a$, and we repeat the experiment n times—that is, we obtain n replicates. The results are shown symbolically in Table 8.9. The results in the first column of the observations, x_{j1} in Table 8.9, would be obtained by randomly ordering the levels A_j, $j = 1, 2 \dots a$, and then running the experiment in this randomly selected order. Then the results in the second column of the observations, x_{j2}, $j = 1, 2 \dots a$, would be obtained by generating a new random order for the levels A_j and running the experiment in this new random order. This procedure is repeated until the replicates have been obtained. Running the experiment in this manner ensures that the values obtained for the x_{jk} have each been independently obtained. Thus, we can define two independent variances using the quantities μ_i and s_i^2 defined in Table 8.9 as follows. The variance of the mean of factor A is

$$s_A^2 = \frac{n}{a-1}\left(\sum_{i=1}^{a}\mu_i^2 - a\bar{x}^2\right) = \frac{SS_A}{a-1}$$

TABLE 8.9 Tabulations of the Results of a Single-Factor Experiment with $n > 1$ Replicates

Level	Observations				Average	Variance	Residuals
A_1	x_{11}	x_{12}	$\dots$	x_{1n}	$\mu_1 = \frac{1}{n}\sum_{j=1}^{n} x_{1j}$	$s_1^2 = \frac{1}{n-1}\sum_{j=1}^{n}(x_{1j} - \mu_1)^2$	$\varepsilon_{1j} = x_{1j} - \mu_1$
A_2	x_{21}	x_{22}		x_{2n}	$\mu_2 = \frac{1}{n}\sum_{j=1}^{n} x_{2j}$	$s_2^2 = \frac{1}{n-1}\sum_{j=1}^{n}(x_{2j} - \mu_2)^2$	$\varepsilon_{2j} = x_{2j} - \mu_2$
$\dots$				$\dots$	$\dots$	$\dots$	
A_a	x_{a1}	x_{a2}	$\dots$	x_{an}	$\mu_a = \frac{1}{n}\sum_{j=1}^{n} x_{aj}$	$s_a^2 = \frac{1}{n-1}\sum_{j=1}^{n}(x_{aj} - \mu_a)^2$	$\varepsilon_{aj} = x_{aj} - \mu_a$

which has $a - 1$ degrees of freedom and $\bar{x}$ is the grand mean given by

$$\bar{x} = \frac{1}{an}\sum_{i=1}^{a}\sum_{j=1}^{n}x_{ij}$$

The variance of the error is

$$s^2_{\text{error}} = \frac{1}{a}\sum_{i=1}^{a}s_i^2 = \frac{1}{a(n-1)}\left(\sum_{i=1}^{a}\sum_{j=1}^{n}x_{ij}^2 - an\bar{x}^2\right) = \frac{\text{SS}_{\text{error}}}{a(n-1)}$$

which has $a(n - 1)$ degrees of freedom.

The variances s_A^2 and s^2_{error} are related by the following identity, called the sum-of-squares identity, which has $an - 1$ degrees of freedom:

$$\begin{aligned}\text{SS}_{\text{total}} &= \sum_{i=1}^{a}\sum_{j=1}^{n}(x_{ij} - \bar{x})^2 = \sum_{i=1}^{a}\sum_{j=1}^{n}[(\mu_i - \bar{x}) + (x_{ij} - \mu_i)]^2\\ &= n\sum_{i=1}^{a}(\mu_i - \bar{x})^2 + \sum_{i=1}^{a}\sum_{j=1}^{n}(x_{ij} - \mu_i)^2\\ &= \text{SS}_A + \text{SS}_{\text{error}}\\ &= (a-1)s_A^2 + a(n-1)s^2_{\text{error}}\end{aligned}$$

The left-hand side of the equation is called the total sum of squares. The identity has partitioned the total variance into two independent components: that due to the factor A and that due to the variation in the process as expressed by the residuals ε_{ij} .

In the analysis of variance, the convention is to define a quantity called the mean square, denoted by MS, which is the sum of squares divided by the number of degrees of freedom. Thus, for the single-factor experiment, we have

$$\text{MS}_A = \frac{\text{SS}_A}{(a-1)} = s_A^2 \qquad a > 1$$

$$\text{MS}_{\text{error}} = \frac{\text{SS}_{\text{error}}}{a(n-1)} = s^2_{\text{error}} \qquad n > 1$$

The objective of the experiment is to determine whether the various levels of A have any statistically significant effect on the output x_{ij}. We now have the ability to determine this by forming the ratio of the mean square of the factor A with the independent mean square of the random error. This tells us whether the variance of A is a statistically significant portion of the total variance. Thus, the test statistic is

$$F_0 = \frac{\text{MS}_A}{\text{MS}_{\text{error}}}$$

The hypothesis is

$$H_0\colon \mu_1 = \mu_2 = \cdots = \mu_a$$

$$H_1\colon \mu_j \neq \mu_i \quad \text{for at least one } j \neq i$$

TABLE 8.10 ANOVA Table for a Single-Factor Experiment with $n > 1$ Replicates

Factor	Sum of squares	Degrees of freedom	Mean square	F_0	p-value
A	SS_A	$a - 1$	MS_A	MS_A/MS_{error}	
Error	SS_{error}	$a(n - 1)$	MS_{error}		
Total	SS_{total}	$an - 1$			

Thus, when

$$F_0 > f_{\alpha,a-1,a(n-1)}$$

the null hypothesis is rejected. The results of this analysis are usually presented in the form shown in Table 8.10.

A single-factor analysis of variance is obtained with

```
p = anova1(x)
```

where p is the p-value and x is the *transpose* of the data shown in Table 8.9. There are two additional outputs from this function: one is the ANOVA table shown in Table 8.10 and the other is a box plot of the variations in the medians of each of the a levels.

We now illustrate the analysis of variance of a single-factor experiment.

Example 8.11 Single-factor analysis of variance

Consider the data in Table 8.11. We shall write a script to generate the ANOVA table, display the p-value, and compute the residuals and determine whether they are normally distributed. We first create the function M file **DataAnova1** with the data in the form required by anova1.

```
function d = DataAnova1
d = [143 141 150 146; . . .
     152 149 137 143; . . .
     134 133 132 127; . . .
     129 127 132 129; . . .
     147 148 144 142]';
```

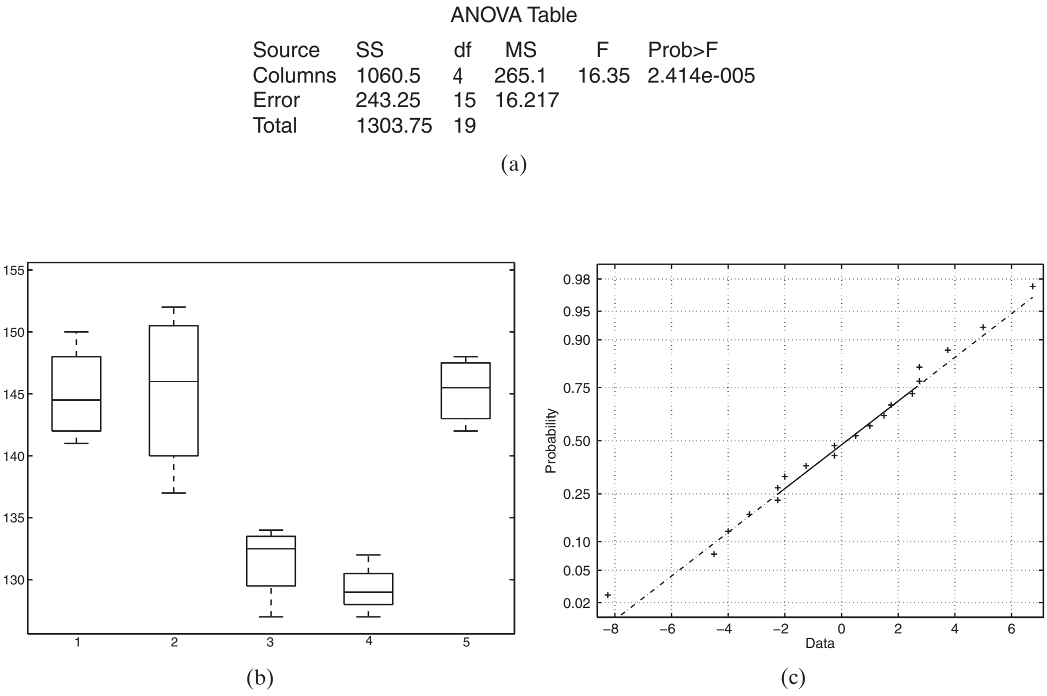
TABLE 8.11 Data for Example 8.11: **DataAnova1**

Level	Observations			
1	143	141	150	146
2	152	149	137	143
3	134	133	132	127
4	129	127	132	129
5	147	148	144	142

The program is

```
vv = DataAnova1;
[r, c] = size(vv);
pp = anova1(vv);
meen = mean(vv);
k = 0;
for n = 1:r
   for m = 1:c
      k = k+1;
      e(k) = vv(n,m)-meen(m);
   end
end
figure(3)
normplot(e)
```

The nested for loops are needed to store the residuals as a vector. The figure function is used to open another figure window, because anova1 opens two windows of its own. If the figure function weren't used, then one of the two figures generated by anova1 would be overdrawn. The function anova1 produces the table in Figure 8.16a and the box plot in Figure 8.16b; Figure 8.16c is produced independently using normplot.



```
                ANOVA Table

Source     SS        df   MS        F      Prob>F
Columns    1060.5    4    265.1     16.35  2.414e-005
Error      243.25    15   16.217
Total      1303.75   19
```

(a)

(b)

(c)

Figure 8.16 Analysis of variance of the data in Table 8.11: (a) ANOVA table (b) Box plot of the five levels (c) Normal distribution plot of the residuals using normplot.

8.6.2 Multiple-Factor Factorial Experiments

Factorial Experiments

The results for a single-factor experiment can be extended to experiments with several factors. In particular, if we run all combinations of all the levels of each factor for each replicate of the experiment, we call these factorial experiments. We illustrate the procedure for a two-factor experiment, which has the factor A at a levels and the factor B at b levels. The number of replicates is $n(>1)$ and the output is x_{ijk}, where $i = 1, 2, \ldots, a, j = 1, 2, \ldots, b$, and $k = 1, 2, \ldots, n$. The intervals between each level of each factor do not have to be equal. The tabular form of these data is given in Table 8.12.

The starting point is the sum-of-squares identity. Before proceeding with this identity, however, we introduce the following definitions for several different means:

$$\bar{x}_{ijn} = \frac{1}{n}\sum_{k=1}^{n} x_{ijk}$$

$$\bar{x}_{ibn} = \frac{1}{b}\sum_{j=1}^{b} \bar{x}_{ijn} = \frac{1}{bn}\sum_{j=1}^{b}\sum_{k=1}^{n} x_{ijk}$$

$$\bar{x}_{ajn} = \frac{1}{a}\sum_{i=1}^{a} \bar{x}_{ijn} = \frac{1}{an}\sum_{i=1}^{a}\sum_{k=1}^{n} x_{ijk}$$

and the grand mean

$$\bar{x} = \frac{1}{abn}\sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n} x_{ijk}$$

The $\bar{x}_{ijn}$ are used to plot the average output as a function of the various factors, as is shown in Example 8.12.

The sum-of-squares identity for a two-factor analysis of variance is

$$\mathrm{SS}_{\mathrm{total}} = \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n}(x_{ijk} - \bar{x})^2 = \mathrm{SS}_A + \mathrm{SS}_B + \mathrm{SS}_{AB} + \mathrm{SS}_{\mathrm{error}}$$

TABLE 8.12 Data Arrangement for a Two-Factor Factorial Experiment

		Factor B			
		1	2	. . .	b
Factor A	1	$y_{111}, y_{112}, \ldots, y_{11n}$	$y_{121}, y_{122}, \ldots, y_{12n}$	. . .	$y_{1b1}, y_{1b2}, \ldots, y_{1bn}$
	2	$y_{211}, y_{212}, \ldots, y_{21n}$	$y_{221}, y_{222}, \ldots, y_{22n}$	. . .	$y_{2b1}, y_{2b2}, \ldots, y_{2bn}$
	. . .	. . .	. . .		. . .
	a	$y_{a11}, y_{a12}, \ldots, y_{a1n}$	$y_{a21}, y_{a22}, \ldots, y_{a2n}$	. . .	$y_{ab1}, y_{ab2}, \ldots, y_{abn}$

where

$$SS_A = \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n}(\overline{x}_{ibn} - \overline{x})^2 = bn\sum_{i=1}^{a}\overline{x}_{ibn}^2 - abn\overline{x}^2$$

$$SS_B = \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n}(\overline{x}_{ajn} - \overline{x})^2 = an\sum_{j=1}^{b}\overline{x}_{ajn}^2 - abn\overline{x}^2$$

$$SS_{AB} = n\sum_{i=1}^{a}\sum_{j=1}^{b}(\overline{x}_{ijn} - \overline{x}_{ibn} - \overline{x}_{ajn} + \overline{x})^2$$

and

$$SS_{\text{error}} = \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n}(x_{ijk} - \overline{x}_{ijn})^2 = \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n}x_{ijk}^2 - n\sum_{i=1}^{a}\sum_{j=1}^{b}\overline{x}_{ijn}^2$$

The quantities SS_A, SS_B, SS_{AB}, SS_{error}, and SS_{total} have $(a-1)$, $(b-1)$, $(a-1)(b-1)$, $ab(n-1)$, and $abn-1$ degrees of freedom, respectively. The sum-of-squares term SS_{AB} indicates the interaction of factors A and B. The ANOVA table for a two-factor experiment is given in Table 8.13. The definitions of the mean square values are also given in this table. It is seen that the analysis of variance isolates the interaction effects of the two factors and provides a means of ascertaining, through the ratio $MS_{AB}/MS_{\text{error}}$, whether or not the interaction of the factors is statistically significant at a stated confidence level.

The solution to a two-factor factorial experiment is obtained from

```
anova2(y, n)
```

where n is the number of replicates. This function produces Table 8.13. The matrix y follows the form of the data in Table 8.12 as follows:

$$y = \begin{bmatrix} y_{111} & y_{121} & \cdots & y_{1b1} \\ y_{112} & y_{122} & & y_{1b2} \\ \vdots & & & \\ y_{11n} & y_{12n} & & y_{1bn} \\ y_{211} & y_{221} & & y_{2b1} \\ y_{212} & y_{222} & & y_{2b2} \\ \vdots & & & \\ y_{21n} & y_{22n} & & y_{2bn} \\ \vdots & & & \\ y_{a11} & y_{a21} & & y_{ab1} \\ y_{a12} & y_{a22} & & y_{ab2} \\ \vdots & & & \\ y_{a1n} & y_{a2n} & & y_{abn} \end{bmatrix}$$

We shall now illustrate the use of these relationships.

TABLE 8.13 ANOVA Table for a Two-Factor Experiment with $n > 1$ Replicates

Factor	Sum of squares	Degrees of freedom	Mean square	F_0	$f_{\alpha, z, ab(n-1)}$	p-value
A	SS_A	$a-1$	$MS_A = SS_A/(a-1)$	MS_A/MS_{error}	(f-table, $z = a-1$)	
B	SS_B	$b-1$	$MS_B = SS_B/(b-1)$	MS_B/MS_{error}	(f-table, $z = b-1$)	
AB	SS_{AB}	$(a-1)(b-1)$	$MS_{AB} = SS_{AB}/(a-1)(b-1)$	MS_{AB}/MS_{error}	(f-table, $z = (a-1)(b-1)$)	
Error	SS_{error}	$ab(n-1)$	$MS_{error} = SS_{error}/ab(n-1)$			
Total	SS_{total}	$abn-1$				

Example 8.12 Two-factor analysis of variance

Consider the data shown in Table 8.14. We create the following function M file **DataAnova2** to put these data in the appropriate format and to compute the average values of the four replicates at each combination of the levels A and B, that is, $\bar{x}_{ijn}$.

```
function [d, xbar] = DataAnova2
dc1 = [[130, 155, 74, 180]'; [150, 188, 159, 126]'; [138, 110, 168, 160]'];
dc2 = [[34, 40, 80, 75]'; [136, 122, 106, 115]'; [174, 120, 150, 139]'];
dc3 = [[20, 70, 82, 58]'; [25, 70, 58, 45]'; [96, 104, 82, 60]'];
d = [dc1, dc2, dc3];
xbar = zeros(3,3);
for c = 1:3
  for r = 1:3
    z = 4*(r-1);
    xbar(r,c) = sum(d(z+1:z+4,c))/4;
  end
end
```

The program is

```
[d, xbar] = DataAnova2;
anova2(d, 4);
figure(2)
```

TABLE 8.14 Data for Example 8.12: **DataAnova2**

		Factor B		
		1	2	3
Factor A	1	130, 155, 74, 180	34, 40, 80, 75	20, 70, 82, 58
	2	150, 188, 159, 126	136, 122, 106, 115	25, 70, 58, 45
	3	138, 110, 168, 160	174, 120, 150, 139	96, 104, 82, 60

```
for k = 1:3
  plot(1:3,xbar(1:3,k),'ks-')
  hold on
  text(2.1, xbar(2,k)-5, ['B_' num2str(k)])
end
ylabel('Average response')
xlabel('Levels of A','fontsize',14)
set(gca, 'XTick', [1,2,3])
set(gca, 'XTickLabel', {'A1' 'A2' 'A3'})
```

which upon execution creates Figure 8.17 and displays the following table in a figure window:

```
                              ANOVA Table
Source        SS          df      MS          F         Prob>F
Columns       39118.7     2       19559.4     28.97     0
Rows          10683.7     2       5341.9      7.91      0.002
Interaction   9613.8      4       2403.0      3.56      0.0186
Error         18230.8     27      675.2
Total         77647       35
```

Thus, based on the p-values, we see that factors A and B are statistically significant at the greater than 99.8% level and that their interaction is significant at the 98% level. The statistically significant interaction is also apparent in Figure 8.17, where we see that level B_2 strongly interacts with levels A_1 and A_3.

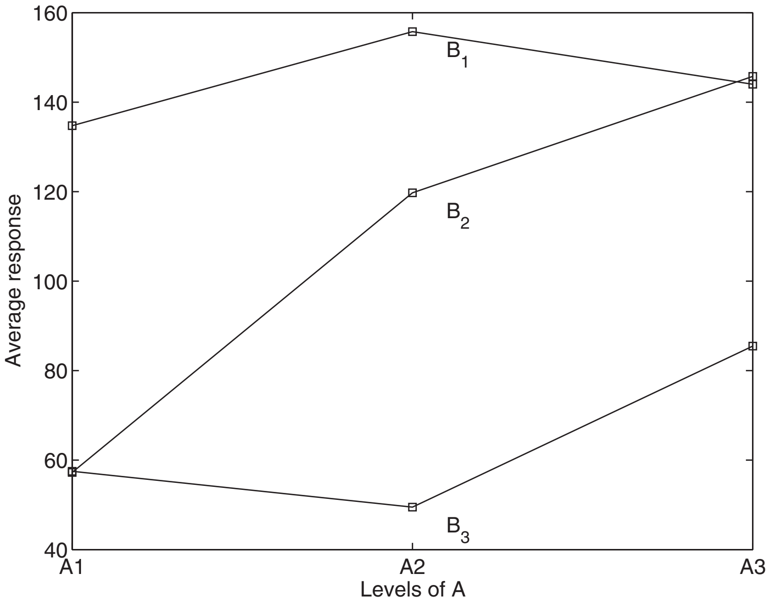


Figure 8.17 Average responses of the variables in a two-factor ANOVA using the data in Table 8.14.

Example 8.13 Three-factor analysis of variance: stiffness of fiberglass–epoxy beams

A series of tests were conducted to determine the manufacturing conditions that produce the stiffest fiberglass and epoxy composite beams when the beam is subjected to a three-point bending load. The manufacturing conditions are determined by running a three-factor, single replicate, full factorial experiment, which resulted in twenty-seven different manufacturing combinations. The three factors are (1) the fiberglass fabric orientation; (2) epoxy resins from three manufacturers, with each resin having the same nominal strength characteristics; and (3) the amount of the hardener (curing agent) provided by each manufacturer and used with their epoxy resin. Twenty-seven beams were fabricated under twenty-seven different combinations of these manufacturing conditions and the measured results are tabulated in Table 8.15. These data are analyzed using `anovan`, which creates the ANOVA table, and `maineffectsplot`, which plots the main effects irrespective of whether or not they are statistically meaningful. The arguments of these two functions are illustrated in the following script. They are somewhat self-explanatory when examined in the context of Table 8.15. The program that analyzes these results and plots the main effects is

```
EX = [ 3.16, 3.986, 0.376]*10^4;
EY = [4.82, 4.80, 4.41]*10^4;
EZ = [4.08, 3.82, 3.87]*10^4;
FX = [2.52, 2.12, 1.48]*10^4;
FY = [3.20, 3.97, 3.38]*10^4;
FZ = [2.98, 2.76, 3.79]*10^4;
GX = [2.26, 1.57, 1.99]*10^4;
GY = [2.21, 3.69, 2.54]*10^4;
GZ = [2.72, 2.45, 2.67]*10^4;
g1={'E'; 'E'; 'E'; 'E'; 'E'; 'E'; 'E'; 'E'; 'E'; . . .
    'F'; 'F'; 'F'; 'F'; 'F'; 'F'; 'F'; 'F'; 'F'; . . .
    'G'; 'G'; 'G'; 'G'; 'G'; 'G'; 'G'; 'G'; 'G'};
g2 = {'X'; 'X'; 'X'; 'Y'; 'Y'; 'Y'; 'Z'; 'Z'; 'Z'; . . .
    'X'; 'X'; 'X'; 'Y'; 'Y'; 'Y'; 'Z'; 'Z'; 'Z'; . . .
    'X'; 'X'; 'X'; 'Y'; 'Y'; 'Y'; 'Z'; 'Z'; 'Z'};
g3 = {'L'; 'M'; 'N'; 'L'; 'M'; 'N'; 'L'; 'M'; 'N'; . . .
    'L'; 'M'; 'N'; 'L'; 'M'; 'N'; 'L'; 'M'; 'N'; . . .
    'L'; 'M'; 'N'; 'L'; 'M'; 'N'; 'L'; 'M'; 'N'};
g = [EX'; EY'; EZ'; FX'; FY'; FZ'; GX'; GY'; GZ'];
anovan(g, {g1 g2 g3}, 'model', 'interaction', 'varnames', . . .
```

TABLE 8.15 Measured Stiffness (N/m) for the Twenty-Seven Manufacturing Combinations

	Fiber orientation E			Fiber orientation F			Fiber orientation G		
	Hardener ratio			Hardener ratio			Hardener ratio		
Resin system	L	M	N	L	M	N	L	M	N
X	31.6×10^3	39.8×10^3	3.7×10^3	25.2×10^3	21.2×10^3	14.8×10^3	22.6×10^3	15.7×10^3	19.9×10^3
Y	48.2×10^3	48.0×10^3	44.1×10^3	32.0×10^3	39.7×10^3	33.8×10^3	22.1×10^3	36.9×10^3	25.4×10^3
Z	40.8×10^3	38.2×10^3	38.7×10^3	29.8×10^3	27.6×10^3	37.9×10^3	27.2×10^3	24.5×10^3	26.7×10^3

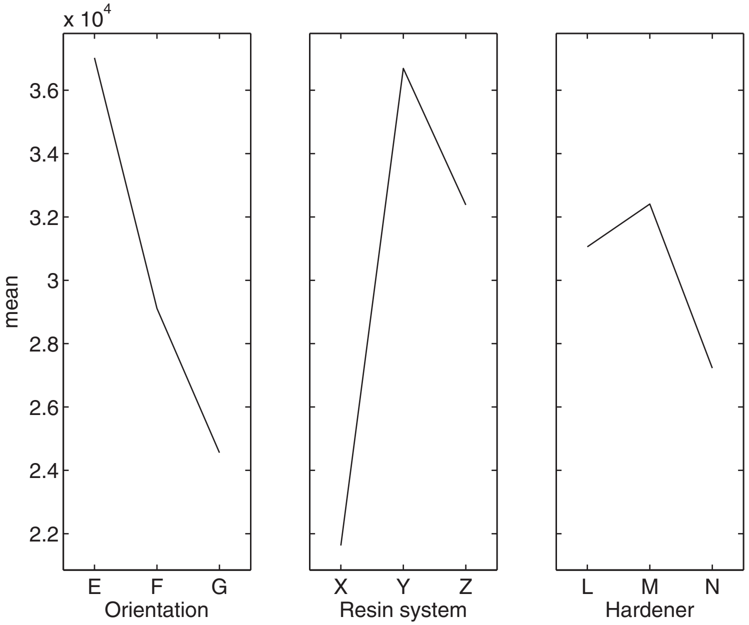


Figure 8.18 Main effects as created by `maineffectsplot` for the data in Table 8.15.

```
    {'Orientation', 'Resin system', 'Hardener'});
maineffectsplot(g, {g1 g2 g3}, 'varnames', {'Orientation', 'Resin system', 'Hardener'})
```

The execution of these results produces the following table in its own figure window and the plot of the main effects shown in Figure 8.18.

```
Source                   Sum Sq.       d.f.  Mean Sq.      F      Prob>F
Orientation              7.16541e+008  2     3.58271e+008  7.98   0.0124
Resin system             1.08347e+009  2     5.41736e+008  12.07  0.0038
Harder                   1.29835e+008  2     6.49174e+007  1.45   0.2909
Orientation*Resin system 1.2881e+008   4     3.22025e+007  0.72   0.6032
Orientation*Hardener     1.82203e+008  4     4.55508e+007  1.02   0.4543
Resin system*Hardener    3.54323e+008  4     8.85808e+007  1.97   0.1916
Error                    3.59e+008     8     4.48749e+007
Total                    2.95418e+009  26
```

We see that the fiber orientation and the resin system are the main factors that are statistically significant and that none of the interaction effects is statistically significant.

2^k *Factorial Experiments*

If the factorial experiments described above contain k factors and each factor is considered at only two levels, then the experiment is called a 2^k factorial design. It implicitly assumes that there is a linear relationship between the two levels of each factor. This assumption leads to certain simplifications in how the tests are conducted and how the results are analyzed.

The convention is to denote the value of the high level of a factor with either "1" or "+," and the value of the low level either "0" or "−." Then the 2^k combination of factors that comprise one run, which represents one replicate, is given in Table 8.16 for $k = 2, 3$, and 4. The table is used as follows. For the $2^2\,(k = 2)$ factorial experiment, only the columns labeled A and B and the first four rows ($m = 1, \ldots, 4$) are used. The four combinations of the factors are run in a random order. One such random order is shown in the column labeled 2^2. Thus, the combination in row 2 is run first, with A high (A_{high}) and B low (B_{low}). This yields the output value $y_{2,1}$. Then the combination shown in the fourth row is run, where both A and B are at their high levels (A_{high} and B_{high}, respectively). This gives the output response $y_{4,1}$. After the remaining combinations have been run, one replicate of the experiment has been completed. A newly obtained random order for the run is obtained, one that is most likely different from the one shown in the column labeled 2^2, and the four combinations are run in the new order to get the output response for the second replicate. For $k = 3$, the factors are A, B, and C, and the first eight rows of the table are used; for $k = 4$, the factors are A, B, C, and D, and all sixteen rows of the table are used. One set of a random run order is given for each of these cases in the columns labeled 2^3 and 2^4, respectively.

TABLE 8.16 Levels and Run Order of Each Factor for a 2^2, 2^3, and 2^4 Factorial Experiment

Run no.	Factors and their levels				Data ($y_{m,j}$)			Run order number*		
m	A	B	C	D	$j = 1$	$j = 2$	…	2^2	2^3	2^4
1	−	−	−	−	$y_{1,1}$	$y_{1,2}$		3	5	6
2	+	−	−	−	$y_{2,1}$	$y_{2,2}$		1	7	11
3	−	+	−	−	$y_{3,1}$	$y_{3,2}$		4	8	14
4	+	+	−	−	$y_{4,1}$	$y_{4,2}$		2	4	5
5	−	−	+	−	$y_{5,1}$	$y_{5,2}$			2	13
6	+	−	+	−	$y_{6,1}$	$y_{6,2}$			1	2
7	−	+	+	−	$y_{7,1}$	$y_{7,2}$			3	16
8	+	+	+	−	$y_{8,1}$	$y_{8,2}$			6	15
9	−	−	−	+	$y_{9,1}$	$y_{9,2}$				9
10	+	−	−	+	$y_{10,1}$	$y_{10,2}$				7
11	−	+	−	+	$y_{11,1}$	$y_{11,2}$				10
12	+	+	−	+	$y_{12,1}$	$y_{12,2}$				3
13	−	−	+	+	$y_{13,1}$	$y_{13,2}$				8
14	+	−	+	+	$y_{14,1}$	$y_{14,2}$				4
15	−	+	+	+	$y_{15,1}$	$y_{15,2}$				1
16	−	+	+	+	$y_{16,1}$	$y_{16,2}$				12

*One set of randomly ordered runs for $j = 1$ only. For $j = 2$, a new set of a randomly generated run order is used, and so on.

The combination of levels indicated in Table 8.16 as a function of k can be obtained with

```
s = ffn2(k)
```

where k $= k$ and s is the $(2^k \times k)$ array of 1's and 0's. Unfortunately, s is not in the order given in Table 8.16; it is equal to that shown in Table 8.16 when its columns are flipped using `fliplr`. Furthermore, in order to use s in subsequent examples, its 0's have to be converted to -1's as shown in **Levels**, which is created in Example 8.14.

After the data have been collected, they are analyzed as follows, provided that the number of replicates is greater than one. Consider the tabulations in Table 8.17. The + and − signs in each column represent +1 and −1, respectively.

TABLE 8.17 Definitions of Various Terms That Are Used to Calculate the Sum of Squares and Mean Square Values for a 2^2, 2^3, and 2^4 Factorial Experiment

Factors and their interactions (λ)[†]															Data[‡]				
														A					
						A				A		A	B	B					
		A		A	B	B		A	B	B	C	C	C	C					
A	B	B	C	C	C	C	D	D	D	D	D	D	D	D	$j = 1$	. . .	$j = n$	$S_m{}^{\#}$	m
−	−	+	−	+	+	−	−	+	+	−	+	−	−	+	$y_{1,1}$		$y_{1,n}$	S_1	1
+	−	−	−	−	+	+	−	−	+	+	+	+	−	−	$y_{2,1}$		$y_{2,n}$	S_2	2
−	+	−	−	+	−	+	−	+	−	+	+	−	+	−	$y_{3,1}$		$y_{3,n}$	S_3	3
+	+	+	−	−	−	−	−	−	−	−	+	+	+	+	$y_{4,1}$		$y_{4,n}$	S_4	4
−	−	+	+	−	−	+	−	+	+	−	−	+	+	−	$y_{5,1}$		$y_{5,n}$	S_5	5
+	−	−	+	+	−	−	−	−	+	+	−	−	+	+	$y_{6,1}$		$y_{6,n}$	S_6	6
−	+	−	+	−	+	−	−	+	−	+	−	+	−	+	$y_{7,1}$		$y_{7,n}$	S_7	7
+	+	+	+	+	+	+	−	−	−	−	−	−	−	−	$y_{8,1}$		$y_{8,n}$	S_8	8
−	−	+	−	+	+	−	+	+	−	+	−	+	+	−	$y_{9,1}$		$y_{9,n}$	S_9	9
+	−	−	−	−	+	+	+	−	−	−	−	−	+	+	$y_{10,1}$		$y_{10,n}$	S_{10}	10
−	+	−	−	+	−	+	+	+	+	−	−	+	−	+	$y_{11,1}$		$y_{11,n}$	S_{11}	11
+	+	+	−	−	−	−	+	−	+	+	−	−	−	−	$y_{12,1}$		$y_{12,n}$	S_{12}	12
−	−	+	+	−	−	+	+	+	−	+	+	−	−	+	$y_{13,1}$		$y_{13,n}$	S_{13}	13
+	−	−	+	+	−	−	+	−	−	−	+	+	−	−	$y_{14,1}$		$y_{14,n}$	S_{14}	14
−	+	−	+	−	+	−	+	+	−	−	+	−	+	−	$y_{15,1}$		$y_{15,n}$	S_{15}	15
+	+	+	+	+	+	+	+	+	−	+	+	+	+	+	$y_{16,1}$		$y_{16,n}$	S_{16}	16

[†] The '+' and '−' stand for +1 and −1, respectively, although they also indicate the high and low levels of the factors.

[‡] The data are obtained as indicated in Table 8.16.

[#] $S_m = \sum_{j=1}^{n} y_{m,j}$

The columns for the primary factors A, B, C, and D are the same as those given in Table 8.16, where again the + and − signs stand for +1 and −1, respectively. The columns representing all the interaction terms are obtained by multiplying the corresponding signs in the columns of the primary factors. Thus, the signs in the columns designating the interaction ABC are obtained by multiplying the signs in the columns labeled A, B, and C. For example, in row seven ($m = 7$), $A = -1, B = +1$, and $C = +1$; therefore, the sign in the seventh row of the column labeled ABC is $-1\,[=(-1)(+1)(+1)]$. Furthermore, for the 2^2 experiment, the first three columns and the rows $m = 1, 2, \ldots, 4$ are used; for the 2^3 experiment, the first seven columns and the rows $m = 1, 2, \ldots, 8$ are used; and for the 2^4 experiment all fifteen columns and the rows $m = 1, 2, \ldots, 16$ are used.

The sum of squares is obtained for $n > 1$ and for a given value of k as follows:

$$\mathrm{SS}_{\mathrm{total}} = \sum_{j=1}^{n}\sum_{m=1}^{2^k} y_{m,j}^2 - 2^k n\bar{y}^2$$

$$\mathrm{SS}_{\mathrm{error}} = \mathrm{SS}_{\mathrm{total}} - \sum_{\lambda} SS_\lambda$$

$$\mathrm{SS}_\lambda = \frac{C_\lambda^2}{n2^k} \qquad \lambda = A, B, AB, \ldots$$

where

$$C_\lambda = \sum_{m=1}^{2^k} S_m \times (\text{sign in row } m \text{ of column } \lambda) \qquad l = A, B, AB, \ldots$$

$$\bar{y} = \frac{1}{n2^k}\sum_{m=1}^{2^k} S_m$$

and S_m is defined at the bottom of Table 8.17.

The average value of the effect of the primary factors and their interactions is obtained from the relation

$$\mathrm{Effect}_\lambda = \frac{C_\lambda}{n2^{k-1}} \qquad \lambda = A, B, AB, \ldots$$

where Effect$_\lambda$ is called the effect of λ. As seen in Table 8.17, for $k = 2$, there are three λ's: A, B, and AB; for $k = 3$, there are seven λ's: A, B, C, AB, AC, BC, and ABC; and for $k = 4$, there are fifteen λ's: $A, B, C, D, AB, AC, BC, AD, BD, CD, ABC, ABD, ACD, BCD$, and $ABCD$.

The mean square values for the main effects and their interactions are simply

$$\mathrm{MS}_\lambda = \mathrm{SS}_\lambda$$

since the number of degrees of freedom for each primary factor and their interactions is one. The mean square for the error is

TABLE 8.18 ANOVA Table for a 2^k Factorial Experiment with $n > 1$ Replicates

Factor	Sum of squares	Degrees of freedom	Mean square	F_λ	$f_{\alpha,1,(n-1)2^k}$
A	SS_A	1	MS_A	$\text{MS}_A/\text{MS}_{\text{error}}$	(Value from $f -$ table)
B	SS_B	1	MS_B	$\text{MS}_B/\text{MS}_{\text{error}}$	(Value from $f -$ table)
C	SS_C	1	MS_C	$\text{MS}_C/\text{MS}_{\text{error}}$	(Value from $f -$ table)
...	...	...	...	...	...
AB	SS_{AB}	1	MS_{AB}	$\text{MS}_{AB}/\text{MS}_{\text{error}}$	(Value from $f -$ table)
AC	SS_{AC}	1	MS_{AC}	$\text{MS}_{AC}/\text{MS}_{\text{error}}$	(Value from $f -$ table)
BC	SS_{BC}	1	MS_{BC}	$\text{MS}_{BC}/\text{MS}_{\text{error}}$	(Value from $f -$ table)
...	...	...	...	...	...
ABC	SS_{ABC}	1	MS_{ABC}	$\text{MS}_{ABC}/\text{MS}_{\text{error}}$	(Value from $f -$ table)
...	...	...	...	...	...
Error	SS_{error}	$(n-1)2^k$	MS_{error}		
Total	SS_{total}	$n2^{k-1}$			

$$\text{MS}_{\text{error}} = \frac{\text{SS}_{\text{error}}}{2^k(n-1)} \qquad n > 1$$

The test statistic for each factor and their interactions is

$$F_\lambda = \frac{\text{MS}_\lambda}{\text{MS}_{\text{error}}} = \frac{\text{MS}_\lambda}{\text{SS}_{\text{error}}/[2^k(n-1)]} \qquad \lambda = A, B, AB, \ldots, \qquad n > 1$$

The ANOVA table for the 2^k factorial analysis is given in Table 8.18.

The results of ANOVA for the 2^k factorial design can be used directly to obtain a multiple regression model that estimates the output of the process as a function of the statistically significant primary factors and the statistically significant interactions. We first introduce the coded variable x_β

$$x_\beta = \frac{2\beta - \beta_{\text{low}} - \beta_{\text{high}}}{\beta_{\text{high}} - \beta_{\text{low}}}$$

where β is a primary variable; that is, $\beta = A, B, C, \ldots$. Thus, if $\beta = A$, when $\beta = A_{\text{high}}$, then $x_A = +1$, and when $\beta = A_{\text{low}}, x_A = -1$. Since $A_{\text{low}} \leq A \leq A_{\text{high}}$, then $-1 \leq x_A \leq +1$.

An estimate of the average output y_{avg} is

$$y_{\text{avg}} = \bar{y} + \frac{1}{2}\left[\sum_\lambda \mathit{Effect}_\lambda x_\lambda + \sum_\lambda \sum_\beta \text{Effect}_{\lambda\beta} x_\lambda x_\beta \right.$$
$$\left. + \sum_\lambda \sum_\beta \sum_\gamma \text{Effect}_{\lambda\beta\gamma} x_\lambda x_\beta x_\gamma \ldots \right] \tag{8.41}$$

where $\lambda, \beta, \gamma, \ldots$ have the values of $A, B, C, \ldots$ and correspond only to those combinations of subscripts that indicate statistically significant factors and interactions, and the $x_\alpha (-1 \leq x_\alpha \leq +1)$ are the coded values.

2^k *Factorial Experiments: One Replicate*

In some cases, it is more economical to perform a factorial experiment with only one replicate. When only one replicate is used, there is no estimate of the mean square error. There is, however, a graphical method that permits one to identify the significant factors and interactions for $k > 2$; but the error variance still remains unknown. The graphical method is based on the discussion in Example 8.4 with respect to `normplot` and the creation of ordered data. Here, the ordered data values are replaced by the ordered values of Effects$_\lambda$. The effects that are negligible will be normally distributed and will tend to fall on a straight line on this plot, whereas the effects that are significant will not lie on this straight line.

We now illustrate the use of these relationships.

Example 8.14 Analysis of a 2^4 factorial experiment

We generate the ANOVA table for the data in Table 8.19, which were obtained from a two-replicate 2^4 factorial experiment. We shall include in the ANOVA table the effects.

TABLE 8.19 Data for a Two-Replicate 2^4 Experiment: **FactorialData**

Run no.*	Data ($y_{m,j}$)	
m	$j = 1$	$j = 2$
1	159	163
2	168	175
3	158	163
4	166	168
5	175	178
6	179	183
7	173	168
8	179	182
9	164	159
10	187	189
11	163	159
12	185	191
13	168	174
14	197	199
15	170	174
16	194	198

*Run number corresponds to level combinations given in Table 8.16.

The run numbers correspond to those of Table 8.16. First, we create a function M file **FactorialData** for these data.

```
function dat = FactorialData
dat1 = [159 168 158 166 175 179 173 179 164 187 163 185 168 197 170 194];
dat2 = [163 175 163 168 178 183 168 182 159 189 159 191 174 199 174 198];
dat = [dat1, dat2];
```

Next, we create a function M file called **FactorialSigns**, which determines the values ±1 in Table 8.17 for *k* factors. This function uses `ff2n` to obtain the levels indicated in Table 8.16. The output of `ffn2`, however, has to be altered in two respects. In `ff2n`, the column order is reversed with respect to that appearing in Table 8.16 and instead of having ±1 as output it has +1 and 0. These manipulations are performed in the function M file **Levels**.

```
function s = FactorialSigns(k)
s = Levels(k);
s(:,3) = s(:,1).*s(:,2);
for j = 3:k
   ink = 2^(j-1);
   for m = (ink+1):(2^j-1)
      s(:,m) = s(:,ink).*s(:,m-ink);
   end
end
```

```
function s = Levels(k)
sx = fliplr(ff2n(k));
for n = 1:k
   ind = find(sx(:,n)==0);
   sx(ind,n) = -1;
end
s = zeros(2^k, 2^k-1);
M = 1/2;
for n = 1:k
   M = M*2;
   s(1:2^k,M) = sx(:,n);
end
```

The program to create the ANOVA table is as follows:

```
tag = char('A', 'B', 'AB', 'C', 'AC', 'BC', 'ABC', 'D', 'AD', 'BD', 'ABD', . . .
   'CD', 'ACD', 'BCD', 'ABCD');
k = 4;     n = 2;
fdata = FactorialData;
s = FactorialSigns(k);
Sm = sum(fdata')'
yBar = sum(Sm)/n/2^k;
SStotal = sum(sum(fdata.^2))-yBar^2*n*2^k;
for nn = 1:2^k-1
   Clambda = sum(s(:,nn).*Sm);
   SSlambda(nn) = Clambda^2/2^(k+1);
   EffectLambda(nn) = Clambda./2^k;
end
```

```
SSerror = SStotal-sum(SSlambda);
MSerror = SSerror/2^k;
f0 = SSlambda/MSerror;
pValue = 1-fcdf(f0, 1, 2^k);
disp('Factor   SS   MS   Effect   f-lambda   p-value')
disp([tag repmat(' ',15,1) num2str(SSlambda',6) repmat (' ',15,1) . . .
  num2str(SSlambda',6) repmat(' ',15,1) num2str (EffectLambda',6) . . .
  repmat(' ',15,1) num2str(f0',6) repmat(' ',15,1) num2str (pValue',6)])
disp(['SSerror ' num2str(SSerror,6) ' ' num2str (MSerror,6)])
disp(['SStotal ' num2str(SStotal,6)])
disp(['yBar = ' num2str(yBar,6)])
```

The execution of this script results in the following information being displayed to the MATLAB window:

```
Factor    SS         MS         Effect     f-lambda     p-value
A         2312       2312       17         241.778      4.45067e-011
B         21.125     21.125     -1.625     2.20915      0.156633
AB        0.125      0.125      -0.125     0.0130719    0.910397
C         946.125    946.125    10.875     98.9412      2.95785e-008
AC        3.125      3.125      -0.625     0.326797     0.575495
BC        0.5        0.5        -0.25      0.0522876    0.822026
ABC       4.5        4.5        0.75       0.470588     0.502537
D         561.125    561.125    8.375      58.6797      9.69219e-007
AD        666.125    666.125    9.125      69.6601      3.18663e-007
BD        12.5       12.5       1.25       1.30719      0.269723
ABD       2          2          -0.5       0.20915      0.653583
CD        12.5       12.5       -1.25      1.30719      0.269723
ACD       0          0          0          0            1
BCD       0.125      0.125      0.125      0.0130719    0.910397
ABCD      21.125     21.125     -1.625     2.20915      0.156633
SSerror   153        9.5625
SStotal   4716
yBar =    175.25
```

It is seen that at the considerably better than 95% confidence level, factors A, C, and D and interaction AD are statistically significant and, therefore, influence the outcome of the process. In fact, the sum of the sum of squares of these four quantities is 4,485. Thus, the sum-of-squares contribution of the quantities that are not statistically significant is $78 = 4716 - 4485 - 153$, or 1.65% of the total sum of squares.

We now use Eq. (8.41) and the above results to obtain the following regression equation that is a function of the three main effects and one interaction effect that are statistically significant at the greater than 95% confidence level.

$$y_{\text{avg}} = 175.25 + 8.50x_A + 5.44x_C + 4.10x_D + 4.56x_Ax_D$$

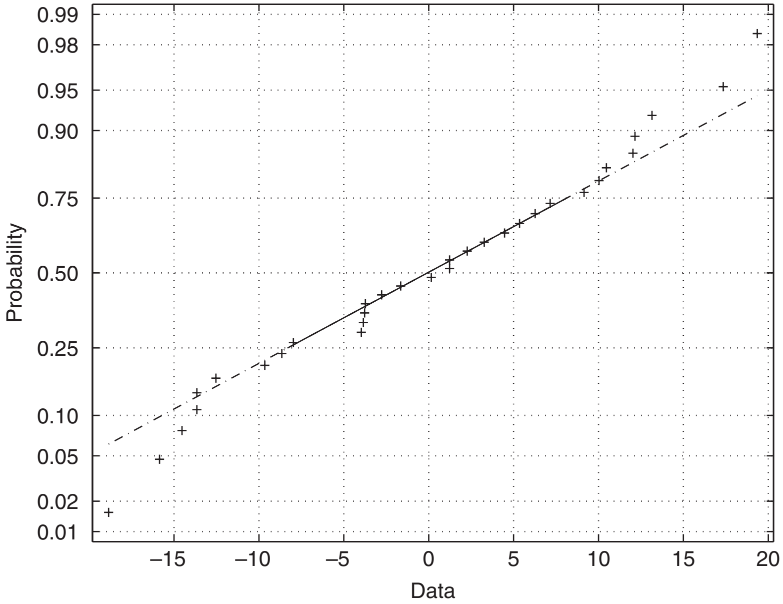


Figure 8.19 Residual plot for the data used in Example 8.14.

The residuals are the differences between the measured values $y_{m,j}$ and y_{avg} at the 2^4 combinations of coded values of x_γ shown in Table 8.16. We now use the following program to determine the residuals and plot them using normplot.

```
fdata = FactorialData;
s = FactorialSigns(4);
yAvg = 175.25+8.5*s(:,1)+5.44*s(:,3)+4.1*s(:,4)+4.56*s(:,1).*s(:,4);
normplot([fdata(:,1)-yAvg; fdata(:,2)-yAvg])
```

When the program is executed, we obtain Figure 8.19. We see that the residuals are acceptable.

Example 8.15 Analysis of a 2^4 factorial experiment with one replicate

We generate a modified ANOVA table for the data in Table 8.20, which were obtained from a single-replicate 2^4 factorial experiment. The modified ANOVA table will include the effects and the ordered effects with their corresponding cumulative distribution function $(j - 0.5)/(2^4 - 1), j = 1, 2, \ldots, 15$. The task is broken into two parts. The first part determines the ordered effects and their corresponding cumulative distribution function and the second part plots the results. The second part of the program was written after the results of the first part were analyzed. In plotting the results, we had to create our own version of normplot; that is, we had to transform the ordinate into cumulative normal distribution function. This conversion is performed in **VertHorizAxis**. The straight line fit

TABLE 8.20 Data for a Single-Replicate 2^4 Experiment

Run no.*	Data (y_j)
1	86
2	200
3	90
4	208
5	150
6	172
7	140
8	192
9	90
10	142
11	96
12	130
13	136
14	120
15	160
16	130

*Run number corresponds to level combinations given in Table 8.16.

to the data was determined as the line that represents those effects that are within the 25%–75% quartiles. The script is given below. It is noted that we have used several portions of the program that were employed in Example 8.14.

```
function Example8_15
y = [86, 200, 90, 208, 150, 172, 140, 192, 90, 142, 96, 130, 136, 120, 160, 130];
tag = char('A', 'B', 'AB', 'C', 'AC', 'BC', 'ABC', 'D', 'AD', 'BD', 'ABD', 'CD',
   'ACD', 'BCD', 'ABCD');
k = 4;
s = FactorialSigns(k);
Sm = sum(y);
yBar = Sm/2^k;
SStotal = sum(y.^2)-yBar^2*2^k;
SSlambda = zeros(1, 2^k-1);
EffectLambda = zeros(1, 2^k-1);
for nn = 1:2^k-1
   Clambda = sum(s(:,nn).*y');
   SSlambda(nn) = Clambda^2/2^k;
   EffectLambda(nn) = Clambda./2^(k-1);
end
[OrdLam, ixx] = sort(EffectLambda, 'descend');
n = 15:-1:1;
```

```
Cdf = (n-0.5)/15;
disp('Lambda   SS   Effect   Ordered   Ordered   Cumulative')
disp('   Lambda   Effect   Distribution')
disp([tag repmat(' ',15,1) num2str(SSlambda',6) repmat(' ',15,1) . . .
   num2str(EffectLambda',6) repmat(' ',15,1) tag(ixx,:) . . .
   repmat(' ',15,1) num2str(OrdLam') repmat(' ',15,1) num2str(Cdf',3)])
disp(['SStotal ' num2str(SStotal,6)])
% Second part
plot(OrdLam, norminv(Cdf, 0, 1), 'ko')
hold on
y25 = prctile(OrdLam,25);
y75 = prctile(OrdLam,75);
ind = find((OrdLam >= y25) & (OrdLam <= y75));
mu = mean(OrdLam(ind));
sig = std(OrdLam(ind));
slope = norminv(0.84, 0, 1)/sig;
b =-slope*mu;
plot([-5, 9], [-5*slope+b, 9*slope+b], 'k—')
Lab = char('A', 'C', 'D',' AD', 'AC');
n = [1, 2, 13, 14, 15];
for k = 1:5
   text(OrdLam(n(k))+2.5, norminv(Cdf(n(k)), 0, 1), Lab(k,:))
end
VertHorizAxis

function VertHorizAxis
axis off
Xax = -40:20:60;
con = [0.01, 0.02, 0.05, 0.1:0.1:0.9, 0.95, 0.98, 0.99];
y = norminv(con, 0, 1);
v(3) = y(1); v(4) = y(end);
v(1 ) = Xax(1); v(2) = Xax(end);
axis(v)
plot([v(1), v(1), v(2), v(2), v(1)], [v(3), v(4), v(4), v(3), v(3)], 'k-')
for k = 1:length(con)
   text(v(1)-1, y(k), num2str(con(k)), 'horizontalalignment', 'right')
   plot([v(1), v(1)+2], [y(k), y(k)], 'k-')
end
for m = 1:length(Xax)
   text(Xax(m), v(3)-0.1, num2str(Xax(m)), 'verticalalignment', 'top',
   'horizontalalignment', 'center')
   plot([Xax(m), Xax(m)], [v(3), v(3)+0.1], 'k-')
end
text(v(1)-13, -1, 'Cumulative probability', 'fontsize', 14, 'rotation', 90)
text(10, v(3)-0.5, 'Effects_\lambda', 'horizontalalignment', 'center')
```

The execution of this program displays the following table to the command window and creates the graph shown in Figure 8.20. The column spacing of the output below has been realigned manually for clarity. The labeling of the five statistically significant points in the figure was done after the figure was created. These points were not known prior to their initial display.

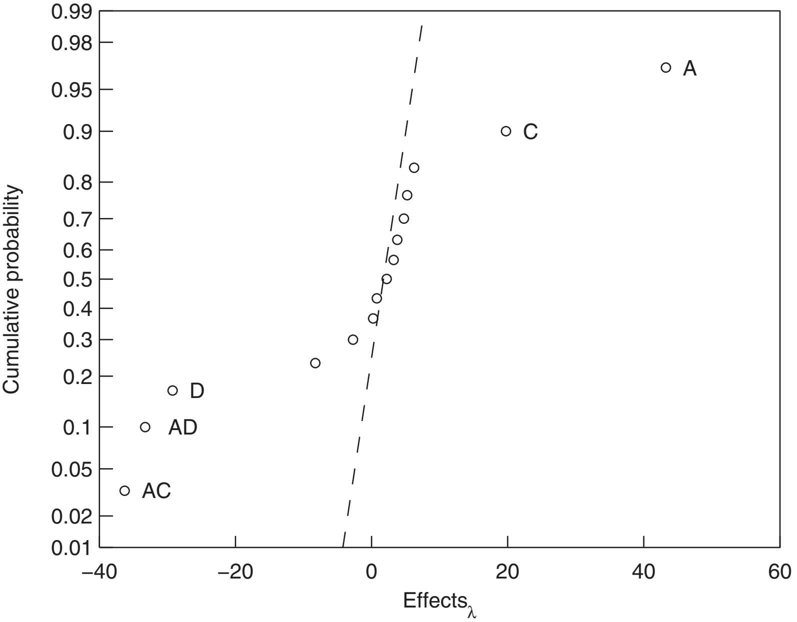


Figure 8.20 Normal distribution plot of a single-replicate 2^4 factorial experiment to determine the statistically significant factors and their interactions.

Lambda	SS	Effect	Ordered Lambda	Ordered Effect	Cumulative Distribution
A	7482.25	43.25	A	43.25	0.967
B	156.25	6.25	C	19.75	0.9
AB	0.25	0.25	B	6.25	0.833
C	1560.25	19.75	BCD	5.25	0.767
AC	5256.25	-36.25	BC	4.75	0.7
BC	90.25	4.75	ABC	3.75	0.633
ABC	56.25	3.75	ACD	3.25	0.567
D	3422.25	-29.25	CD	2.25	0.5
AD	4422.25	-33.25	BD	0.75	0.433
BD	2.25	0.75	AB	0.25	0.367
ABD	272.25	-8.25	ABCD	-2.75	0.3
CD	20.25	2.25	ABD	-8.25	0.233
ACD	42.25	3.25	D	-29.25	0.167
BCD	110.25	5.25	AD	-33.25	0.1
ABCD	30.25	-2.75	AC	-36.25	0.0333
SStotal	22923.8				

From Figure 8.20, it is seen that effects A, C, D, AD, and AC are statistically significant.

8.7 SUMMARY OF FUNCTIONS INTRODUCED IN CHAPTER 8

A summary of the functions introduced in Chapter 8 and their descriptions are presented in Tables 8.2 and 8.21.

TABLE 8.21 MATLAB Functions Introduced in Chapter 8

MATLAB function	Description
`anova1`	One-way analysis of variance
`anova2`	Two-way analysis of variance
`anovan`	*N*-way analysis of variance
`boxplot`	Box plot of statistical data
`ffn2`	Two-level full factorial design
`geomean`	Geometric mean
`hist`	Plot a histogram
`histfit`	Plot a histogram with a superimposed normal distribution function
`maineffectsplot`	Plots main effects for grouped data
`mean`	Mean
`median`	Median
`normplot`	Normal probability plot
`polyconf`	Polynomial evaluation and confidence interval estimation
`prctile`	Percentiles of a sample
`prod`	Product of an array of elements
`range`	Range of data
`rcoplot`	Residuals plot with confidence intervals
`regress`	Multiple linear regression
`skewness`	Sample skewness
`std`	Standard deviation
`ttest`	Hypothesis test for a single sample mean
`ttest2`	Hypothesis test for the difference in means of two samples
`var`	Variance
`vartest`	Chi-square variance test
`vartest2`	Two-sample *F*-test for equal variances
`wblplot`	Displays a weibul probability plot of the data
`zscore`	Centers and scales data using its mean and standard deviation.

EXERCISES

Section 8.2.1

8.1 **a.** A company's telephone helpline receives an average of five calls per minute during its working hours. What is the probability that it could receive: (i) eight calls per minute; (ii) two calls per minute? (Answers: (i) 0.065278 (ii) 0.084224)

b. The telephone system can handle ten calls per minute; if there is more than this number, the caller gets a busy signal. What is the probability of getting a busy signal? (Answer: 0.013695)

8.2 The probability that a structural member can withstand a load L_o is 0.7. If fifteen of these members are to be used, then what is the probability that at least twelve of them can withstand L_o? (Answer: 0.29687)

8.3 Taguchi defines the average loss factor of a process as being proportional to

$$L_{\text{avg}} = s^2 + (\bar{x} - \tau)^2$$

TABLE 8.22 Data for Exercise 8.3

Process #1		Process #2	
88.4	89.0	92.6	93.2
93.2	90.5	93.2	91.7
87.4	90.8	89.2	91.5
94.3	93.1	94.8	92.0
93.0	92.8	93.3	90.7
94.3	91.9	94.0	93.8

where τ is the target mean. In other words, when comparing two processes, the one whose mean is closest to τ and whose variance is the smallest is the process with the lowest loss factor. For the data in Table 8.22, determine which process has the lowest average loss factor when $\tau = 92.0$ (Answer: $L_1 = 5.5904$ and $L_2 = 2.6936$).

8.4 A manufacturer found that 20% of one of its products was underweight. There are twenty-four of these items in a case. If we assume that the weight of each item is independent of the weight of another item, then one can apply the binomial distribution.

a. What is the expected number of underweight items in a case and its variance?

b. What is the probability that there are no more than two underweight items in a case?

c. What is the probability that none of the items in the case is underweight?

d. Plot on the same figure the probability mass function and the cumulative distribution function as a function of the number of underweight items in a case.

(Answers: (a) expected value = 4.8 and variance = 3.84 (b) 0.11452 (c) 0.0047224.)

Section 8.2.2

8.5 The reliability of a component $R(t)$ is the probability that it operates without failure for a length of time t. If the probability distribution function of the life of the component is $f(t)$, then its cumulative distribution is

$$F(t) = P(T \leq t) = \int_{-\infty}^{t} f(u)du = \int_{0}^{t} f(u)du$$

which is the probability of the time to failure. Thus,

$$R(t) = 1 - F(t)$$

The hazard rate function $h(t)$ is the chance of a component, which has not yet failed at time t, suddenly failing. It is given as

$$h(t) = \frac{f(t)}{R(t)} = \frac{f(t)}{1 - F(t)}$$

a. Plot the hazard-rate function and the reliability on the same graph when $f(t)$ is the exponential distribution given by

$$f(t) = \frac{1}{\mu} e^{-t/\mu}$$

Assume $\mu = 1$ and use `exppdf` and `expcdf`.

b. Plot the hazard-rate function and the reliability when $f(t)$ is the Weibull distribution with $\alpha = 1$ and $\beta = 0.\,5, 1, 2$, and 4. Use `subplot` to create a 2×2 array of four figures, each with a pair of curves corresponding to a β.

8.6 The cumulative distribution function for the lognormal distribution is given by

$$F(t) = \Phi\left(\frac{\ln(t) - \bar{x}_L}{s_L}\right) \qquad \text{(a)}$$

where

$$\bar{x}_L = \frac{1}{n}\sum_{i=1}^{n} \ln(t_i) \quad s_L^2 = \frac{1}{n-1}\left[\sum_{i=1}^{n}\left(\ln(t_i)\right)^2 - n\bar{x}_L^2\right] \qquad \text{(b)}$$

If we take the inverse of Eq. (a), we obtain

$$y = \beta_0 + \beta_1 x$$

where

$$y = \Phi^{-1}\left(F(t)\right) \quad x = \ln(t) \quad \beta_0 = -\frac{\bar{x}_L}{s_L} \quad \beta_1 = \frac{1}{s_L}$$

and $\Phi^{-1}(\ldots)$ is obtained from `norminv`. The mean and variance of t (not $\ln(t)$, which has a normal distribution) are given, respectively, by

$$\bar{x}_t = e^{\bar{x}_L + s_L^2/2} \qquad \text{and} \qquad s_t^2 = \left(e^{s_L^2} - 1\right)e^{2\bar{x}_L + s_L^2}$$

which can be obtained from `lognstat`.

a. For the sorted data in Table 8.23, determine whether or not they are distributed lognormally using the technique outlined prior to Example 8.3. In other words,

TABLE 8.23 Data for Exercise 8.6

1.55	15.70
3.05	16.35
3.65	17.70
5.20	17.95
7.75	19.45
10.45	19.80
10.85	20.05
10.90	32.75
12.65	35.45
15.25	49.35

plot $F(t)$ as a function of $\ln(t)$ and the fitted line; also use `normplot` to display the residuals.

b. Compare the values of x_L and s_L obtained by the graphical method to those obtained from Eq. (b). (Answer: From curve fit, $x_L = 2.5072$ and $s_L = 0.88841$; from Eq. (a), $x_L = 2.5072$ and $s_L = 0.85441$.)

Section 8.3

8.7 The process capability ratio (PCR) is a measure of the ability of a process to meet specifications that are given in terms of a lower specification limit LSL and an upper specification limit USL. It is defined as

$$\text{PCR} = \frac{\text{USL} - \text{LSL}}{6\hat{\sigma}}$$

for a centered process and as

$$\text{PCR}_k = \min\left[\frac{\text{USL} - \bar{x}}{3\hat{\sigma}}, \frac{\bar{x} - \text{LSL}}{3\hat{\sigma}}\right]$$

for a noncentered process. The quantity $\hat{\sigma}$ is the estimate of the standard deviation of the process and $\bar{x}$ an estimate of its mean. When PCR $>$ 1, very few defective or nonconforming units are produced; when PCR $=$ 1, then 0.27% (or 2,700 parts per million) nonconforming units are produced; and when PCR $<$ 1, a large number of nonconforming units are produced. The quantity 100/PCR is the percentage of the specification width used by the process. When PCR $=$ PCR$_k$, then the process is centered.

The number of nonconforming parts is Np, where N is the total number of parts produced, and

$$p = 1 - \Phi\left(\frac{\text{USL} - \bar{x}}{\hat{\sigma}}\right) + \Phi\left(\frac{\text{LSL} - \bar{x}}{\hat{\sigma}}\right)$$

where Φ is given by Eq. (8.16). Also recall Figure 8.5c and Eq. (8.18).

For the data in Table 8.24, use the MATLAB function `capable` to determine p, PCR, and PCR$_k$ when LSL = 2.560 and $USL = 2.565$. Is the process centered? (Answer: $p = 1.5351e - 004$, PCR $=$ 1.3103, and PCR$_k$ $=$ 1.2099.)

Section 8.4

8.8 Consider the data shown in Table 8.25. We shall assume two scenarios: (1) All the data in Table 8.25 comprise one set denoted by S_0, and (2) the data in each of the five pairs of columns represent five separate sets denoted by S_j, $j = 1, 2, \ldots, 5$.

TABLE 8.24 Data for Exercise 8.7

2.5629	2.5630
2.5630	2.5628
2.5628	2.5623
2.5634	2.5631
2.5619	2.5635
2.5613	2.5623

TABLE 8.25 Data for Exercise 8.8

1		2		3		4		5	
1115	1567	1223	1782	1055	798	1016	2100	910	1501
1310	1883	375	1522	1764	1020	1102	1594	1730	1238
1540	1203	2265	1792	1330	865	1605	2023	1102	990
1502	1270	1910	1000	1608	2130	706	1315	1578	1468
1258	1015	1018	1820	1535	1421	2215	1269	758	1512
1315	845	1452	1940	1781	1109	785	1260	1416	1750
1085	1674	1890	1120	1750	1481	885	1888	1560	1642

a. Determine the harmonic mean of data set S_0 and compare it with the mean and the geometric mean. The harmonic mean is determined from `harmmean`.

b. What are the mean values and standard deviations of the six data sets $S_j, j = 0, 1, 2, \ldots, 5$.

c. Display a vertical box plot of the data sets $S_j, j = 1, 2, \ldots, 5$.

d. Determine the confidence limits on the differences in the mean values of S_0 and each $S_j, j = 1, 2, \ldots, 5$, at the 95% confidence level assuming that the standard deviations are unknown but equal (Case 4 in Tables 8.4 and 8.6). What are the p-values for each of the data sets? Are any of the mean values of the data sets S_j statistically significantly different from the mean of S_0? Do these conclusions qualitatively agree with the results displayed in (c) above?

8.9 To determine whether or not one should use Case 4 or Case 5 in Table 8.6, an F-test is first performed on the ratio of the variances as denoted in Case 7 of the table. If the variances are statistically significantly different, then Case 5 is used; otherwise, Case 4 is used. Write a script to determine whether there is a difference between the means of the data given in Table 8.26, and then based on the results determine whether the means are different. Also create a box plot to visualize the data and qualitatively support your conclusions. (Answer: From F-test on the ratio of variances, $p = 0.47092$;

TABLE 8.26 Data for Exercise 8.9

Group 1		Group 2	
88	81	76	79
79	83	83	85
84	90	78	76
89	87	80	80
81	78	84	82
83	80	86	78
82	87	77	78
79	85	75	77
82	80	81	81
85	88	78	80

therefore, there is no difference in the variances. From a t-test on differences in means, $p = 0.0009342$; therefore, the means are different.)

Section 8.5.2

8.10 **a.** For the model below, determine β_j for the data in Table 8.27 and show that this model is a good fit to these data.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1^2 + \beta_4 x_2^2 + \beta_5 x_1 x_2$$

b. Using the values found for β_j, plot its surface and use `contour` to plot the contours of the projection of this surface onto the (x_1, x_2)-plane.

c. Determine the coordinates of the maximum value of this fitted surface (Answer: $x_1 = 18.7635$ and $x_2 = 38.0156$).

8.11 The correlation coefficient R for a simple linear regression analysis can be obtained from Eq. (8.29). We can test the hypothesis that

$$H_0: R = 0$$
$$H_1: R \neq 0$$

by forming the test statistic

$$t_0 = \frac{R\sqrt{n-2}}{\sqrt{1-R^2}}$$

TABLE 8.27 Data for Exercise 8.10

y	x_1	x_2
144	18	52
142	24	40
124	12	40
64	30	48
96	30	32
74	26	56
136	26	24
54	22	64
92	22	16
96	14	64
92	10	56
82	10	24
76	6	48
68	6	32

and comparing it with $t_{\alpha/2,n-2}$. If $t_0 > t_{\alpha/2,n-2}$, then we reject H_0. In practice, we examine the p-value corresponding to t_0. The confidence limits on the correlation coefficient r, for $n \geq 25$, can be estimated from

$$\tanh\left(\tanh^{-1}(R) - \frac{z_{\alpha/2}}{\sqrt{n-3}}\right) \leq r \leq \tanh\left(\tanh^{-1}(R) + \frac{z_{\alpha/2}}{\sqrt{n-3}}\right)$$

where $z_{\alpha/2} = \texttt{norminv}(1 - \alpha/2)$ [recall Eq. (8.19)].

For the data given in Table 8.28,

a. Determine the regression coefficients when the model is of the form $y = \beta_0 + \beta_1/x$.

b. Plot the fitted line and the data points.

c. Determine if the residuals are normally distributed.

d. Determine (i) whether the correlation coefficient is different than 0; and (ii) its confidence limits at the 95% confidence level.

(Answer: (a) $\beta_0 = 8.9366$, $\beta_1 = -41.6073$; (d) $0.976998 \leq 0.98996 \leq 0.99564$.)

8.12 In multiple linear regression analysis, one of two types of residuals are frequently examined: (a) the standardized residuals, which are defined as

$$d_i = \frac{e_i}{\hat{\sigma}}$$

where $\hat{\sigma}^2$ is given by Eq. (8.38), and (b) the studentized residuals, which are defined as

$$r_i = \frac{e_i}{\hat{\sigma}\sqrt{1 - h_{ii}}}$$

TABLE 8.28 Data for Exercise 8.11

x	y	x	y
10.0	4.746	11.6	5.211
12.0	5.466	14.8	6.264
6.8	3.171	7.2	3.411
5.4	1.500	15.7	6.537
20.0	6.708	17.6	6.336
19.4	7.158	14.0	5.400
19.1	6.882	10.9	4.503
6.1	1.674	18.2	6.909
16.3	6.498	20.4	6.930
12.4	5.598	8.2	3.582
5.8	1.959	7.9	3.432
12.7	5.790	4.9	0.369
9.2	4.686		

TABLE 8.29 Data for Exercise 8.13

x	y
95	85
105	76
115	114
125	143
135	164
145	281
155	306
165	358
175	437
185	470
195	649
205	702

where h_{ii} is the i th diagonal element of

$$H = X\left(X'X\right)^{-1}X'$$

and X is given by Eq. (8.32).

Using the model in Example 8.10 and the corresponding data in Table 8.8, determine the standardized and studentized residuals. Plot these residuals as a function of the average output $\hat{y}_i$ ($= y_i - e_i$, y_i are the output values given in Table 8.8) on the same graph using two different symbols to differentiate them. Are any of these residuals outliers; that is, do any of them exceed three. Label the figure and identify the two different sets of residuals with `legend`.

8.13 Consider the data in Table 8.29. Fit these data with two models: (1) one that assumes a linear fit; that is, $y = \beta_0 + \beta_1 x$; and (2) one that assumes a quadratic fit; that is, $y = \beta_0 + \beta_1 x + \beta_2 x^2$. For each model, obtain a plot of the residuals to determine if they are normally distributed.

BIBLIOGRAPHY

T. B. Barker, *Quality by Experimental Design*, Marcel Dekker, New York, 1985.

G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters*, John Wiley & Sons, New York, 1978.

F. W. Breyfogle III, *Statistical Methods for Testing, Development and Manufacturing*, John Wiley & Sons, New York, 1992.

N. Draper and H. Smith, *Applied Regression Analysis*, 2nd ed., John Wiley & Sons, New York, 1981.

E. A. Elsayed, *Reliability Engineering*, Addison-Wesley Longman, Inc., Reading, MA, 1996.

N. L. Frigon and D. Mathews, *Practical Guide to Experimental Design*, John Wiley & Sons, New York, 1997.

A. J. Hayter, *Probability and Statistics for Engineers and Scientists*, PWS Publishing Company, Boston, 1996.

E. E. Lewis, *Introduction to Reliability Engineering*, 2nd ed., John Wiley & Sons, New York, 1996.

D. C. Montgomery, *Design and Analysis of Experiments*, 3rd ed., John Wiley & Sons, New York, 1991.

D. C. Montgomery, and G. C. Runger, *Applied Statistics and Probability for Engineers*, John Wiley & Sons, New York, 1994.

R. H. Myers and D. C. Montgomery, *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*, John Wiley & Sons, New York, 1995.

R. E. Walpole, R. H. Myers, and S. L. Myers, *Probability and Statistics for Engineers and Scientists*, 6th ed., Prentice Hall, Upper Saddle River, NJ, 1998.

9

Dynamics and Vibrations

Balakumar Balachandran

Various methods are presented to analyze the dynamics of particles and rigid bodies, the free and forced oscillations of linear and nonlinear systems with one and more degrees of freedom, and the vibrations of Euler–Bernoulli and Timoshenko beams.

9.1 DYNAMICS OF PARTICLES AND RIGID BODIES

9.1.1 Planar Pendulum

The equation of motion of an undamped planar pendulum shown in Figure 9.1 is given by[1]

$$\frac{d^2\theta}{d\tau^2} + \sin\theta = 0 \tag{9.1}$$

where θ is the angular coordinate describing the pendulum motion, $\tau = t\omega_n$ is the nondimensional time, $\omega_n = \sqrt{g/L}$, g is the acceleration due to gravity, and L is the length of the pendulum.

The first integral of motion of Eq. (9.1) leads to

$$F(\theta_1, \theta_2) = \frac{1}{2}\theta_2^2 - \cos\theta_1 \tag{9.2}$$

where

$$\theta_1 = \theta \quad \text{and} \quad \theta_2 = \frac{d\theta}{d\tau}$$

The function given by Eq. (9.2) is plotted in Figure 9.2 using the program below. The lowest point in the valley corresponds to a stable equilibrium position of the system and the peaks located at $\theta = \pm\pi$ correspond to unstable equilibrium positions of the system.

```
theta1 = linspace(-2.0*pi, 2.0*pi, 35);
theta2 = linspace(-2.0*pi, 2.0*pi, 35);
[T1, T2] = meshgrid(theta1, theta2);
```

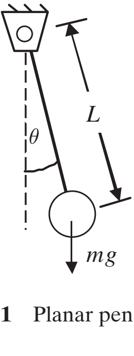


Figure 9.1 Planar pendulum.

[1] D. T. Greenwood, *Principles of Dynamics*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1988, Chapter 5.

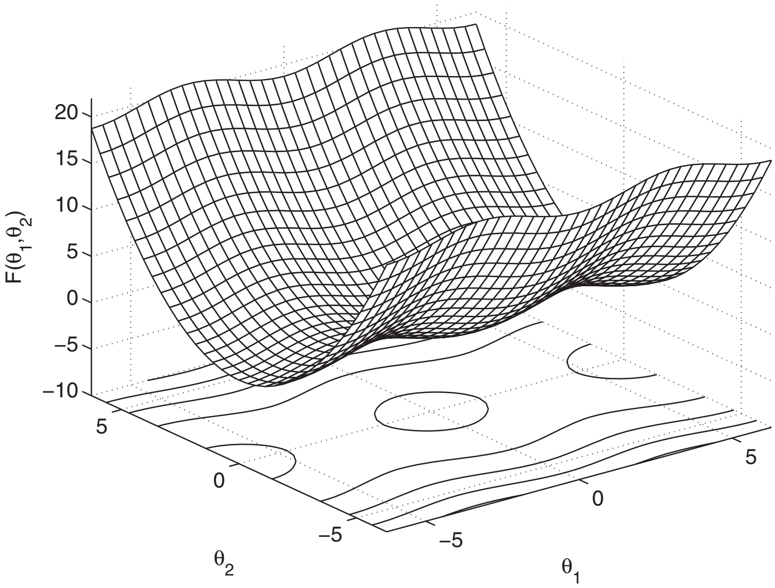


Figure 9.2 Surface $F(\theta_1, \theta_2)$ for a planar pendulum.

```
F = T2.^2/2-cos(T1);
meshc(T1, T2, F)
axis([-2.0*pi, 2.0*pi, -2.0*pi, 2.0*pi, -5, 20])
xlabel('\theta_1')
ylabel('\theta_2')
zlabel('F(\theta_1,\theta_2)')
```

9.1.2 Orbital Motions

Consider a system of two bodies in a gravitational field, shown in Figure 9.3, where the mass m_2 is fixed and the mass m_1 orbits m_2 in a plane. The coordinates describing the orbiting mass are the radial distance $R(t)$ and the angular variable $\theta(t)$, where t is the time.

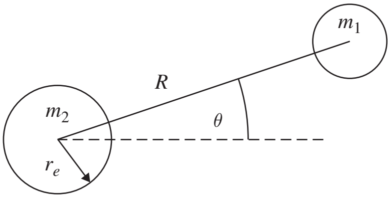


Figure 9.3 A two-body system.

If m_1 is a satellite orbiting about the earth of radius r_e and mass m_2, then the governing equations of motion are given by[2,3]

$$\frac{d^2r}{d\tau^2} - r\left(\frac{d\theta}{d\tau}\right)^2 = -\frac{4\pi^2}{r^2} \tag{9.3}$$
$$r\frac{d^2\theta}{d\tau^2} + 2\frac{dr}{d\tau}\frac{d\theta}{d\tau} = 0$$

where $\tau = t/T_c$ is the nondimensional time, T_c is the period of a circular orbit at the earth's surface, and $r(t) = R(t)/r_e$. At the surface of m_2, $r = 1$.

Although a closed-form solution of the nonlinear system given by Eq. (9.3) is available,[4] numerical solutions will be sought for a given set of initial conditions:

$$r(0), \quad r'(0) = dr(0)/d\tau$$
$$\theta(0), \quad \theta'(0) = d\theta(0)/d\tau$$

The system described by Eq. (9.3) is an example of a differential dynamical system, which describes the evolution of states r and θ with respect to τ. A solution of these equations for a given set of initial conditions can be determined through numerical integration with respect to the independent variable.

Example 9.1 Orbital motions for different initial conditions

We shall determine the orbits of mass m_1 for the three sets of initial conditions shown in Table 9.1. Although the type of orbit realized in each case is not known *a priori*, the orbit type that was determined after analyzing the numerical results is also shown in the last column of the table.

Using the methods discussed in Section 5.5.3, we introduce the variables

$$x_1 = r, \quad x_3 = \theta$$
$$x_2 = \frac{dr}{d\tau}, \quad x_4 = \frac{d\theta}{d\tau}$$

and rewrite Eq. (9.3) as the following set of four first-order differential equations:

$$\frac{dx_1}{d\tau} = x_2, \quad \frac{dx_2}{d\tau} = x_1x_4^2 - \frac{4\pi^2}{x_1^2} \tag{9.4}$$
$$\frac{dx_3}{d\tau} = x_4, \quad \frac{dx_4}{d\tau} = -\frac{2x_2x_4}{x_1}$$

TABLE 9.1 Initial Conditions for Orbital Motions

Set	$x_1(0)$	$x_2(0)$	$x_3(0)$	$x_4(0)$	Orbit type
1	2.0	0.0	0.0	1.5	Elliptical
2	1.0	0.0	0.0	2π	Circular
3	2.0	0.0	0.0	4.0	Hyperbolic

[2] *Ibid.*

[3] F. C. Moon, *Applied Dynamics with Applications to Multibody and Mechatronic Systems*, John Wiley & Sons, NY, 1998, Chapter 7.

[4] Greenwood, *Principles of Dynamics*.

The orbits corresponding to the three sets of initial conditions of Table 9.1 are obtained and plotted using the following script:

```
function Example9_1
initcond = [2.0, 0.0, 0.0, 1.5; 1.0, 0.0, 0.0, 2.0*pi; 2.0, 0.0, 0.0, 4.0];
tspan = [1.5, 1, 0.3];
options = odeset('RelTol', 1e-6, 'AbsTol', [1e-6 1e-6 1e-6 1e-6]);
for n = 1:3
   [t, x] = ode45(@orbit, [0, tspan(n)], [initcond(n,:)]', options);
   polar(x(:,3), x(:,1), 'k-');
   hold on
end
text(0.50, -1.30, 'Elliptical orbit');
text(-1.80, 1.00, 'Circular orbit');
text(1.75, 2.00, 'Hyperbolic orbit');

function xdot = orbit(t, x)
xdot = [x(2); x(1)*x(4)^2-4.0*pi^2/x(1)^2; x(4); -2.0*x(2)*x(4)/x(1)];
```

The three sets of initial conditions are provided in the array labeled *initcond*. The specifier odeset is used to set the relative tolerance and the absolute tolerance to 10^{-6} for each of the four states, namely, x_j, $j = 1, 2, 3$, and 4.

Execution of the script results in Figure 9.4. The first, second, and third set of initial conditions, respectively, lead to an elliptical orbit, a circular orbit, and a hyperbolic orbit. The last orbit, which is an open orbit, is associated with unbounded motion.[5] Open orbits represent escape trajectories from earth and are not considered for satellite motions. In addition, the elliptical orbit is not realistic, since $r < 1$ indicates that the satellite has crashed into the earth's surface.

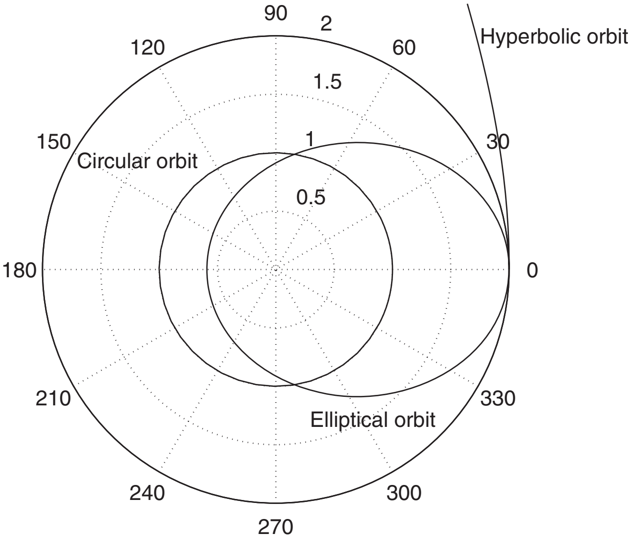


Figure 9.4 Orbits from Eq. (9.3) for three sets of initial conditions given in Table 9.1.

[5] *Ibid*, p. 211.

When systems such as those given by Eq. (9.3) are numerically integrated, one has to be aware that spurious solutions may be obtained during the integration. Usually in problems such as the present one, there is a constant of motion (here, the angular momentum per unit mass $r^2 d\theta/d\tau$) that does not change with time. If a spurious solution were obtained, this constant would vary with time. For other systems without damping, one can determine whether the sum of the kinetic energy and potential energy remains constant during the numerical integration. It is good practice, therefore, to determine if a different result is obtained when the step size and/or the tolerances are changed (see Exercise 9.1).

9.1.3 Principal Moments of Inertia

Consider the rigid body shown in Figure 9.5, which has three rotational degrees of freedom. The associated rotational inertia matrix has the form

$$I_{\text{rot}} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \tag{9.5}$$

where the various moments of inertia are defined with respect to the coordinate system shown in Figure 9.5.

Example 9.2 Principal moments of inertia

We shall determine a new set of orthogonal axes such that the inertia matrix is diagonal. These axes are called the principal axes, and the associated moments of inertia are called the principal moments of inertia. The eigenvalues of Eq. (9.5) provide the principal moments of inertia, and the associated eigenvectors define the principal axes.[6] These two quantities are determined with `eig` as shown below. We also note that the sum of the eigenvalues of a matrix is equal to the trace of the matrix. The trace of a matrix is the sum of the diagonal elements of the matrix, and it is determined with `trace`.

Consider the inertia matrix:

$$[I] = \begin{bmatrix} 150 & 0 & -100 \\ 0 & 250 & 0 \\ -100 & 0 & 500 \end{bmatrix} \text{kg} \cdot \text{m}^2$$

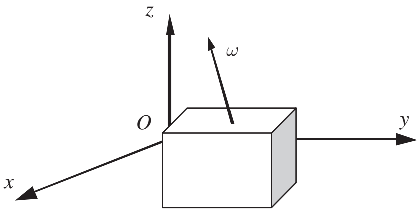


Figure 9.5 Rotation of a rigid body in a frame with Cartesian axes.

[6] *Ibid*.

The script that is used to determine the principal moments of inertia and the trace of the various matrices is as follows:

```
Irot = [150, 0, -100; 0, 250, 0; -100, 0, 500];
[PrincipalDirections, PrincipalMoments] = eig(Irot)
TraceIrot = trace(Irot)
TracePM = trace(PrincipalMoments)
```

Upon execution, we obtain

```
PrincipalDirections =
  -0.9665        0         -0.2567
      0       1.0000           0
  -0.2567        0          0.9665
PrincipalMoments =
  123.4436       0             0
      0       250.0000         0
      0          0          526.5564
TraceIrot =
  900
TracePM =
  900
```

Although the first eigenvector corresponds to the first eigenvalue, the second eigenvector corresponds to the second eigenvalue, and so forth, the eigenvalues (principal moments, in this case) are not in any particular order. This is typical of the results obtained from eig. When the inertia matrix is examined, it is found that in this case, the y-axis is a principal axis and hence one of the eigenvalues is equal to I_{yy}. The matrix of principal directions defines a direction cosine matrix that can be used to transform the (x, y, z) axes to the principal axes.

9.1.4 Stability of a Rigid Body

Again, consider the rigid body shown in Figure 9.5. Let I_1, I_2, and I_3, respectively, represent the second mass moments of inertia about the x-, y-, and z-axes that are chosen to be along the respective principal axes of the body—that is, the principal moments of inertia in the previous example. Let ω_1, ω_2, and ω_3 represent the respective angular velocities about these axes, and let M_1, M_2, and M_3 represent the respective external moments about these axes. The equations of motion, which are known as Euler's equations, are of the form[7]

$$\begin{aligned} I_1\dot{\omega}_1 + (I_3 - I_2)\omega_2\omega_3 &= M_1 \\ I_2\dot{\omega}_2 + (I_1 - I_3)\omega_3\omega_1 &= M_2 \\ I_3\dot{\omega}_3 + (I_2 - I_1)\omega_1\omega_2 &= M_3 \end{aligned} \tag{9.6}$$

[7] *Ibid.*, p. 392; Moon, *Applied Dynamics*, p. 192.

where

$$\dot{\omega}_j = \frac{d\omega_j}{dt} \quad j = 1, 2, 3$$

In the moment-free case—that is, when $M_1 = M_2 = M_3 = 0$—there are three types of solutions where ω_i are constant with respect to time. These solutions, which are called the constant solutions, are as follows:

1. $(\omega_{10} \neq 0, \quad \omega_{20} = 0, \quad \omega_{30} = 0)$
2. $(\omega_{10} = 0, \quad \omega_{20} \neq 0, \quad \omega_{30} = 0)$
3. $(\omega_{10} = 0, \quad \omega_{20} = 0, \quad \omega_{30} \neq 0)$

Each of these solutions corresponds to pure rotational motions about one of the principal axes. We are interested in determining the stability of these three types of motions. To this end, we let ξ_j, $j = 1, 2, 3$, represent the disturbances provided to the system about the respective axes—that is,

$$\begin{aligned} \omega_1(t) &= \omega_{10} + \xi_1(t) \\ \omega_2(t) &= \omega_{20} + \xi_2(t) \\ \omega_3(t) &= \omega_{30} + \xi_3(t) \end{aligned} \tag{9.7}$$

Substituting Eq. (9.7) into Eq. (9.6) and assuming that the magnitudes of the disturbances are "small," one can linearize[8] Eq. (9.6) and study the associated eigenvalue problem. This results in the following system of equations:

$$\begin{bmatrix} 0 & (I_3-I_2)\omega_{30}/I_1 & (I_3-I_2)\omega_{20}/I_1 \\ (I_1-I_3)\omega_{30}/I_2 & 0 & (I_1-I_3)\omega_{10}/I_2 \\ (I_2-I_1)\omega_{20}/I_3 & (I_2-I_1)\omega_{10}/I_3 & 0 \end{bmatrix} \begin{Bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{Bmatrix} = \lambda \begin{Bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{Bmatrix} \tag{9.8}$$

If one or more of the three eigenvalues of Eq. (9.8) has a positive real part, then the disturbances will grow in magnitude and the associated motion will be unstable. Since the trace of Eq. (9.8) is zero, the sum of its eigenvalues will also be zero.

Example 9.3 Stability of a rigid body

Let us consider a rigid body with $I_1 = 150\ \text{kg}\cdot\text{m}^2$, $I_2 = 50\ \text{kg}\cdot\text{m}^2$, and $I_3 = 300\ \text{kg}\cdot\text{m}^2$ and determine the stability of each of the three constant solutions. The following script is used to determine the eigenvalues λ for disturbances provided to the rotational motions along the axis of maximum inertia, the axis of minimum inertia, and the other axis.

```
I = [150, 50, 300];
omega = [1, 0, 0; 0, 1, 0; 0, 0, 1];
```

[8] A. H. Nayfeh and B. Balachandran, *Applied Nonlinear Dynamics: Analytical, Computational, and Experimental Methods*, John Wiley & Sons, New York, 1995.

```
for n = 1:3
   A = [0, (I(3)-I(2))*omega(3,n)/I(1), (I(3)-I(2))*omega(2,n)/I(1); . . .
         (I(1)-I(3))*omega(3,n)/I(2), 0, (I(1)-I(3))*omega(1,n)/I(2); . . .
         (I(2)-I(1))*omega(2,n)/I(3), (I(2)-I(1))*omega(1,n)/I(3), 0];
   disp(['Case ', num2str(n)])
   disp([repmat('lambda(', 3, 1) int2str((1:3)') repmat(') = ', 3,1) . . .
           num2str(eig(A), '%1.0f')])
   disp(['Sum of lambda(j) = ' num2str(sum(eig(A)))])
   disp(' ')
end
```

Execution of the script displays the following in the command window:

```
Case 1
  lambda(1) = 1
  lambda(2) = -1
  lambda(3) = 0
  Sum of lambda(j) = -0
Case 2
  lambda(1) = 0+0.74536i
  lambda(2) = 0-0.74536i
  lambda(3) = 0+0i
  Sum of lambda(j) = 0
Case 3
  lambda(1) = 0+2.2361i
  lambda(2) = 0-2.2361i
  lambda(3) = 0+0i
  Sum of lambda(j) = 0
```

In each of the three cases, one of the eigenvalues is always zero and the sum of the eigenvalues is always zero. In the first case, where the initial rotational motion is along the axis with the intermediate value of inertia, one of the eigenvalues has a positive real part indicating that the motion is unstable. The associated physical motions are wobbly. In the second case, which corresponds to an initial rotational motion about the axis of minimum rotational inertia, two of the eigenvalues form a purely imaginary pair. In the third case, which corresponds to an initial rotational motion about the axis of maximum rotational inertia, two of the eigenvalues form a purely imaginary pair. In the second and third cases, the respective disturbances to the system do not grow because none of the eigenvalues has a positive real part. Hence, the motions in this case are stable. If one further explores the solutions of Eq. (9.6) by numerically integrating the last two cases, it will be found that the motions correspond to circular orbits in the three-dimensional space defined by the three states ω_1, ω_2, and ω_3. These orbits lie on an ellipsoid called Poinsot's ellipsoid.[9]

[9] Greenwood, *Principles of Dynamics*, Section 8.4.

9.2 SINGLE-DEGREE-OF-FREEDOM VIBRATORY SYSTEMS

9.2.1 Introduction

Consider a spring–mass–damper system with a mass m, a spring with linear spring constant k, and a damper with damping coefficient c. In addition, we include a general nonlinear element $g(x, \dot{x})$ to represent a general nonlinearity that is a function of displacement x and velocity $\dot{x}$. The mass is subjected to an excitation $F(t) = F_o f(t)$. This system, which is illustrated in Figure 9.6, is a prototypical model used to study mechanical systems ranging from washing machines to vehicles.[10]

The governing equation of motion is of the form

$$\frac{d^2y}{d\tau^2} + 2\zeta\frac{dy}{d\tau} + y + g(y, \dot{y}) = f(\tau) \tag{9.9}$$

where $y = x/\delta_{st}$, $\delta_{st} = F_o/k$, $\tau = \omega_n t$ is the nondimensional time, and the damping factor ζ is given by

$$\zeta = \frac{c}{2m\omega_n} \tag{9.10}$$

In Eq. (9.10), the natural frequency ω_n is given by

$$\omega_n = \sqrt{k/m} \tag{9.11}$$

When we are considering free oscillations, $F_o = 0$; in this case, we set $\delta_{st} = mg/k$.

When the system is subjected to an initial displacement X_o and an initial velocity V_o, then in terms of the nondimensional coordinates, we have

$$\begin{aligned} y(0) &= X_o/\delta_{st} = y_o \\ \frac{dy(0)}{d\tau} &= V_o/(\delta_{st}\omega_n) = v_o \end{aligned} \tag{9.12}$$

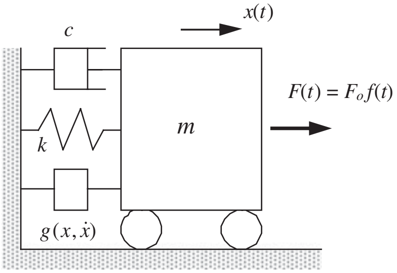


Figure 9.6 Spring–mass–damper system with a general nonlinear element $g(x, \dot{x})$.

[10] B. Balachandran and E. B. Magrab, *Vibrations*, 2nd ed., Cengage, Toronto, ON, 2009.

For a linear system, that is, when $g(x, \dot{x}) = 0$, the damped period of oscillation of this system T_d is defined as

$$T_d = \frac{2\pi}{\omega_n\sqrt{1 - \zeta^2}} \qquad \zeta < 1 \tag{9.13a}$$

or in terms of the nondimensional time τ_d

$$\tau_d = \omega_n T_d = \frac{2\pi}{\sqrt{1 - \zeta^2}} \qquad \zeta < 1 \tag{9.13b}$$

Logarithmic Decrement

Considering free oscillations, the damping factor ζ can be obtained from the displacement response of an underdamped system $(0 < \zeta < 1)$ on the basis of the logarithmic decrement δ, which is given by[11]

$$\delta = \frac{1}{n} \ln\left(\frac{x_j}{x_{j+n}}\right) \tag{9.14}$$

The damping is related to δ by

$$\zeta = \frac{\delta}{\sqrt{4\pi^2 + \delta^2}} \tag{9.15}$$

The quantities x_j and x_{j+n} are the displacements at t_j and $t_j + nT_d$, respectively, where T_d is the period of the underdamped oscillation given by Eq. (9.13a). One way to determine δ is first to determine the period T_d of the system's free response and then to determine the magnitudes of the response x_j and x_{j+n} in the vicinity of the various maxima (minima). From these results, one can use Eq. (9.14) and (9.15) to determine ζ.

State–Space Form

Equation (9.9) is put into state–space form by introducing the variables

$$y_1 = y$$
$$y_2 = \frac{dy}{d\tau}$$

Then Eqs. (9.9) and (9.12), respectively, become

$$\begin{aligned} \frac{dy_1}{d\tau} &= y_2 \\ \frac{dy_2}{d\tau} &= -2\zeta y_2 - y_1 - g(y, \dot{y}) + f(\tau) \end{aligned} \tag{9.16}$$

and

$$\begin{aligned} y_1(0) &= y_o \\ y_2(0) &= v_o \end{aligned} \tag{9.17}$$

Equations (9.16) and (9.17) are in the form required by `ode45`.

[11] *Ibid*, p. 145.

Laplace Transform

To consider a linear system, we set $g(x, \dot{x}) = 0$. Taking the Laplace transform of Eq. (9.9) and setting the initial conditions to zero, we arrive at

$$Y(s) = \frac{F(s)}{s^2 + 2\zeta s + 1} \tag{9.18}$$

where $Y(s)$ and $F(s)$ are the Laplace transforms of $y(\tau)$ and $f(\tau)$, respectively. The transfer function $G(s)$ of a linear time-invariant system (i.e., a system described by a differential equation with constant coefficients) can be obtained from Eq. (9.18) as

$$G(s) = \frac{Y(s)}{F(s)} = \frac{1}{s^2 + 2\zeta s + 1} = \frac{N(s)}{D(s)} \tag{9.19}$$

where the numerator $N(s)$ and the denominator $D(s)$, respectively, are

$$\begin{aligned} N(s) &= 1 \\ D(s) &= s^2 + 2\zeta s + 1 \end{aligned} \tag{9.20}$$

Frequency–Response Function

The frequency–response function for the system described by Eq. (9.19) can be obtained by setting $s = j\omega/\omega_n = j\Omega$. Thus, Eq. (9.19) becomes

$$G(j\Omega) = H(\Omega)e^{-j\theta(\Omega)} \tag{9.21}$$

where

$$\begin{aligned} H(\Omega) &= \frac{1}{\sqrt{(1 - \Omega^2)^2 + (2\zeta\Omega)^2}} \\ \theta(\Omega) &= \tan^{-1}\frac{2\zeta\Omega}{1 - \Omega^2} \end{aligned} \tag{9.22}$$

We now illustrate the use of these results by considering separately the free and forced oscillations of linear and nonlinear single-degree-of-freedom systems.

9.2.2 Linear Systems: Free Oscillations

For free oscillations of linear systems, we set $g(y, \dot{y}) = f(\tau) = 0$ and Eq. (9.9) becomes

$$\frac{d^2y}{d\tau^2} + 2\zeta\frac{dy}{d\tau} + y = 0 \tag{9.23}$$

and Eq. (9.16) simplifies accordingly.

We use Eq. (9.16) to create the following function M file that will be used by `ode45` in several of the examples to follow.

```
function y = LinearOscillatorFree(t, y, zeta)
y = [y(2); -2*zeta*y(2)-y(1)];
```

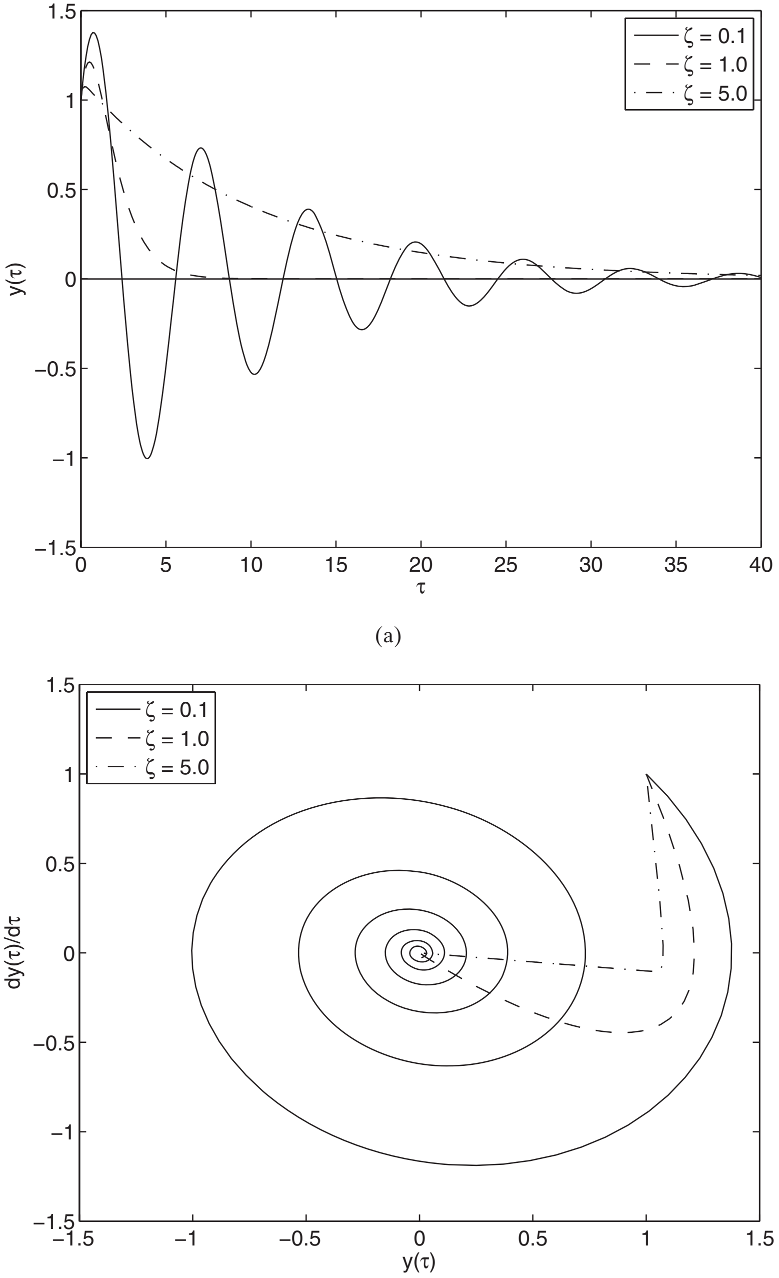
Example 9.4 Oscillations of a single-degree-of-freedom system for given initial velocity and initial displacement

We now determine the free response of the linear spring–mass–damper system given by Eq. (9.23) for the initial conditions $y_o = 1$ and $v_o = 1$ and for the three values of the damping factor ζ: (1) $\zeta = 0.1$, an underdamped case; (2) $\zeta = 1$, the critically damped case; and (3) $\zeta = 5.0$, an overdamped case.

To determine the free response over the range $0 \le \tau \le 40$, we use the following script:

```
zeta = [0.1, 1.0, 5.0];   lintyp = char('-k', '--k', '-.k');
tspan = linspace(0, 40, 400);
for n = 1:3
   [t, y] = ode45(@LinearOscillatorFree, tspan, [1 1]', [], zeta(n));
   figure(1);
   plot(t, y(:,1), lintyp(n,:));
   hold on
   figure(2);
   plot(y(:,1), y(:,2), lintyp(n,:));
   hold on
end
figure(1)
xlabel('\tau');
ylabel('y(\tau)');
axis([0, 40, -1.5, 1.5]);
plot([0, 40], [0, 0], 'k-')
legend('\zeta = 0.1', '\zeta = 1.0', '\zeta = 5.0')
figure(2)
xlabel('Displacement');
ylabel('Velocity');
legend('\zeta = 0.1', '\zeta = 1.0', '\zeta = 5.0', 2)
axis([-1.5, 1.5, -1.5, 1.5]);
```

The results shown in Figure 9.7 are obtained by executing this script. The displacement responses are shown in Figure 9.7a, and the displacements-versus-velocity plots are shown in Figure 9.7b. The space $(x, \dot{x})$ which is formed by using the displacement and velocity as coordinates, is called the phase space. The collections of trajectories that are initiated in this space from different sets of initial conditions constitute a phase portrait. The equilibrium position of the linear system given by Eq. (9.23) corresponds to the location $y_1 = 0$ and $y_2 = 0$ [or (0, 0)] in the phase space. It is seen from the time histories that there are oscillations only in the underdamped case, which correspond to the spiral trajectory in the phase portrait. As time unfolds, this trajectory is attracted to the location (0, 0), which is the equilibrium position and is an example of a point attractor. No oscillations are observed as the system approaches the equilibrium position in the critically damped and overdamped cases. When the system is critically damped, it is seen that the trajectory reaches the equilibrium position in the shortest time.



(a)

(b)

Figure 9.7 (a) Displacement histories. (b) Phase portraits for the free oscillations of a damped, linear oscillator.

Example 9.5 Estimate of damping factor from the logarithmic decrement

If we assume that $y(\tau)$ is the free response of a system to an initial condition, then the damping factor of the system can be determined from the logarithmic decrement given by Eqs. (9.14) and (9.15). In order to use Eq. (9.14), one needs to know τ_d. Therefore, we create a function M file called **ZeroCrossing** to determine all the times at which y equals zero. From these times, one can determine the value(s) of τ_d; that is, from the difference between the third and first zero-crossing times, the fifth and third times, and so on. **ZeroCrossing** is similar to **FindZeros** given in Section 5.5.1, except that in this case we use `interp1` instead of `fzero`.

```
function T = ZeroCrossing (t, y)
cnt = 0;
for n = 3:length(t)-4
   if y(n-1)*y(n) < 0
      N = n-3:n+4;
      cnt = cnt+1;
      T(cnt) = interp1(y(N), t(N), 0);
   end
end
```

For a linear system, the period of damped oscillation does not change over different cycles; however, this is not so for nonlinear systems. In the following script, we assume a value for ζ and use `ode45` to obtain the solution for $y(\tau)$. We use $y(\tau)$ in **ZeroCrossing** to obtain an estimate of the logarithmic decrement from Eqs. (9.14) and (9.15). This estimate should result in a value of ζ that is very closely equal to the value that was used to obtain $y(\tau)$. We assume that $\zeta = 0.3$ and the response is initiated from $y_o = 0.0$ and $v_o = -10.0$. We shall determine the estimates of ζ for the first three periods starting at the approximate time at which $y(\tau)$ has its first maximum.

```
zeta = 0.30;
[t, y] = ode45(@LinearOscillatorFree, linspace(0, 35, 400), [0, -10]', [], zeta);
Tp = ZeroCrossing(t, y(:,1));
[ymx, Imax] = max(y(:,1));
Per = Tp(3:2:7)-Tp(1:2:5);
Q = [t(Imax), t(Imax)+(1:3).*Per];
yspline = spline(t, y(:,1), Q);
delta = log(yspline(1:3)./yspline(2:4));
zetalog = delta./sqrt(4*pi^2+delta.^2);
disp('Period Damping Factor')
disp([num2str((1:3)') repmat('        ', 3, 1) num2str(zetalog')])
```

When one executes this script, the following results are displayed in the command window:

```
Period Damping Factor
  1     0.30011
  2     0.30014
  3     0.30013
```

As expected for a linear system in which energy is dissipated through viscous damping, the decay is exponential and the logarithmic decrement remains constant over each cycle of the damped oscillation.

Example 9.6 Machine Tool Chatter

The vibration of the tool shown in Figure 9.8 can be described by[12]

$$\frac{d^2x}{d\tau^2} + \left(\frac{1}{Q} + \frac{K}{k\Omega}\right)\frac{dx}{d\tau} + \left(1 + \frac{k_1}{k}\right)x - \mu\frac{k_1}{k}x(\tau - 1/\Omega) = 0 \tag{9.24a}$$

where

$$\Omega = \frac{N}{2\pi\omega_n}, \quad Q = \frac{1}{2\zeta} \tag{9.24b}$$

and τ, ω_n, and ζ are defined in Eqs. (9.10) and (9.11). In Eqs. (9.24), N is the rotational speed of the workpiece in revolutions per second, μ is the overlap factor ($0 \le \mu \le 1$), and there is a time delay term in Eq. (9.24a). This is an example of a delay differential equation.

To determine when chatter can occur, a solution of the form

$$x = Ae^{\lambda\tau}$$

is assumed and introduced into Eq. (9.24a) to obtain the characteristic equation

$$\lambda^2 + \left(\frac{1}{Q} + \frac{K}{k\Omega}\right)\lambda + 1 + \frac{k_1}{k}\left(1 - \mu e^{-\lambda/\Omega}\right) = 0 \tag{9.25}$$

To find the stability boundary, we let $\lambda = j\omega$ and substitute into Eq. (9.25) to obtain the equations

$$\frac{1}{Q} + \frac{K}{k\Omega} + \frac{\mu k_1}{k}\frac{\sin(\omega/\Omega)}{\omega} = 0 \tag{9.26}$$

$$\omega^2 = 1 + \frac{k_1}{k}\left(1 - \mu\cos(\omega/\Omega)\right)$$

In Eq. (9.26), the quantities K/k, μ, and k_1/k are known, and the values of the nondimensional spindle speed Ω are varied over a specified range. At each value of Ω, the value of ω is determined numerically from the second of the equations in Eq. (9.26) by using `fzero`. The values for Ω and ω are then used in the first of the equations in Eq. (9.26) to determine the positive values of Q that satisfy the equation; that is, those values of Ω and ω for which

$$\frac{1}{Q} = -\frac{K}{k\Omega} - \frac{\mu k_1}{k}\frac{\sin(\omega/\Omega)}{\omega}$$

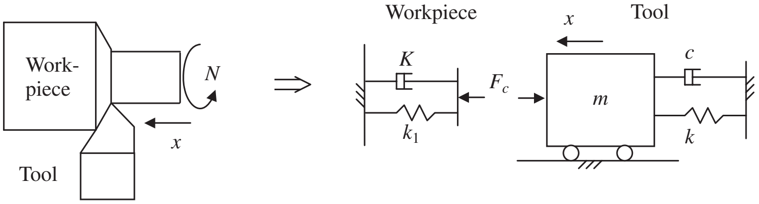


Figure 9.8 Model of a tool and workpiece during turning.

[12] *Ibid*, Section 4.5.

In the plot of Ω versus Q, we can show the regions for which the system is either stable or unstable. For $K/k = 0.0029$, $k_1/k = 0.0785$, and $\mu = 1$, the system of equations given by Eq. (9.26) is solved numerically using the following script:

```
chat = inline('1-w.^2+k1k*(1-u*cos(w/Ob))', 'w', 'u', 'k1k', 'Ob');
k1k = 0.0785;   Kk = 0.0029;   u = 1;
Ob = linspace(0.03, 0.35, 300);
L = length(Ob);   w = zeros(L,1);
for n = 1:L
  w(n) = fzero(chat, [0.8 1.2], [], u, k1k, Ob(n));
end
xx = -1./(Kk./Ob'+u*sin(w./Ob')./w*k1k);
indx = find(xx >= 0);
hold on
a = axis;   a(4) = 50;   a(2) = 0.35;   axis(a)
fill(Ob(indx), xx(indx), 'c')
B = sqrt(2)*sqrt(1+k1k-sqrt(1+2*k1k+(k1k^2)*(1-u^2)));
Qm = 1./(B-Kk./Ob);
ind = find(Ob > 0.05);
plot(Ob(ind), Qm(ind), 'k--')
xlabel('\Omega')
ylabel('Q')
text(.22, 40, 'Unstable')
text(.15, 5, 'Stable')
text(0.02, 10, ['K/k=' num2str(Kk, 5)])
```

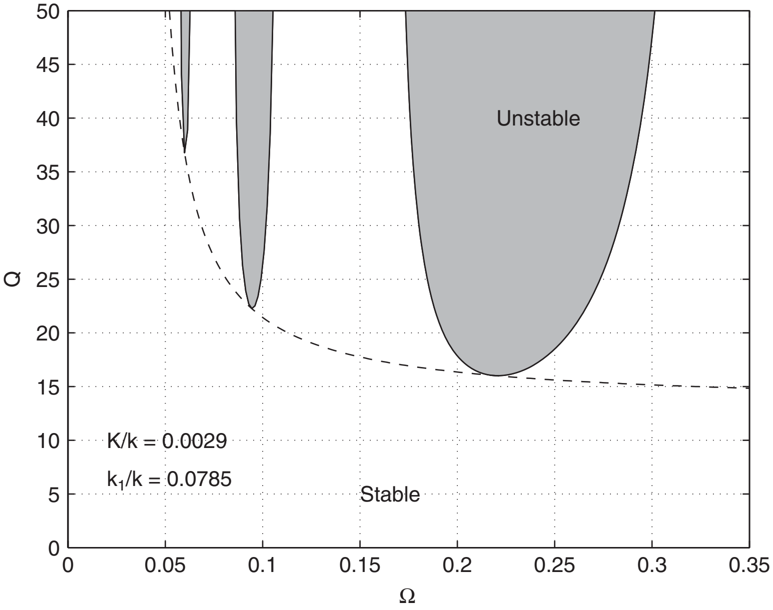


Figure 9.9 Stability chart for $K/k = 0.0029$, $k_1/k = 0.0785$, and $\mu = 1$. Chatter is expected to occur in the shaded regions marked unstable in the figure. In between the stability lobes, as well as below the stability lobes, the cutting operations are expected to be stable.

```
text(0.02, 6, ['k_1/k=' num2str(k1k, 5)])
box on
grid on
```

The execution of the script results in Figure 9.9. The shaded regions or lobes in this figure are the unstable regions where the machine tool can chatter. The time-domain solution of this model is given in Example 5.12.

9.2.3 Linear Systems: Forced Oscillations

For forced oscillations of linear systems, we set $g(y,\dot{y}) = y(0) = \dot{y}(0) = 0$ and Eq. (9.9) becomes

$$\frac{d^2y}{d\tau^2} + 2\zeta\frac{dy}{d\tau} + y = f(\tau) \tag{9.27}$$

We now illustrate two ways to determine and display the displacement response of a linear system subjected to an impulse force and to a step force and two ways to determine the frequency response. One way is to use the functions `step`, `impulse`, `bode`, and `tf` from the Controls toolbox.[13] The transfer function of Eq. (9.27) is given by Eq. (9.17); that is,

$$G(s) = \frac{1}{s^2 + 2\zeta s + 1} \tag{9.28}$$

In order to create the transfer function, we use

```
sys = tf(N, D)
```

where N and D are vectors containing the coefficients of their respective polynomials in the same way that the coefficient vectors were defined for `roots` in Section 5.5.1. In this case, N and D are given by Eq. (9.20).

The frequency response function $G(j\Omega)$ can be computed and plotted with

```
bode(tf(N, D), Om)
```

which plots the magnitude and phase of $G(j\Omega)$, or with

```
[magnitude, phase] = bode(tf(N, D), Om)
```

which provides arrays of numerical values for the magnitude and phase. The quantity *Om* is either a two-element cell that specifies the minimum and maximum values of the frequency range of interest or an array of frequency values.

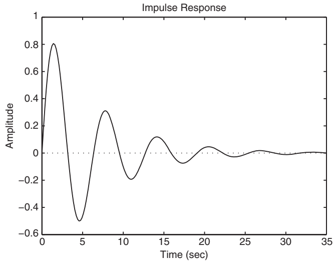
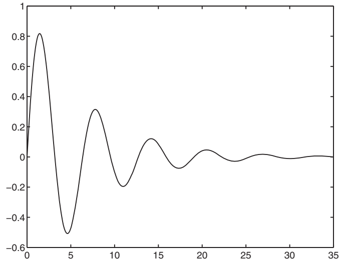
The functions `impulse` and `step` can be used to determine the impulse and step responses, respectively, of linear time-invariant systems set into motion from rest. Thus,

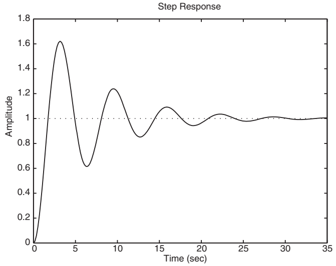
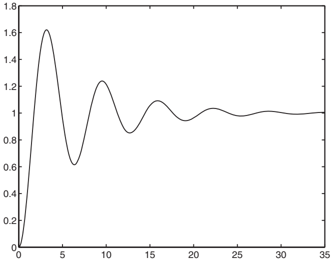
```
impulse(tf(N, D))
```

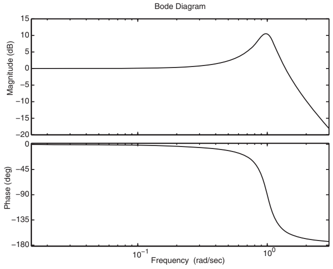
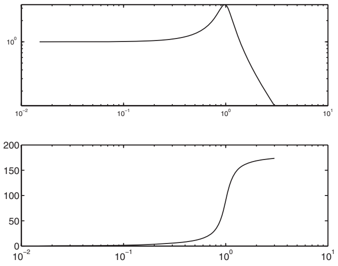
[13] See Chapter 10 for additional applications of these functions.

TABLE 9.2 Various Response Functions of a Single-Degree-of-Freedom System Determined by Two Different Methods

Response	Control toolbox	Output	MATLAB	Output

Impulse	T = linspace(0, 35, 200);
	z = 0.15;
	N = [0, 0, 1]; D = [1, 2*z, 1];
	impulse(tf(N, D), T)
			function **Impulz**
			T = linspace(0, 35, 400);
			z = 0.15; IC = [0,0];
			[t, yy] = ode45(@**Imp**, T, . . .
			            IC, [], z);
			plot(t, yy(:,1), 'k-')
			
			function dd = **Imp**(t, y, z)
			h = (1/0.01).*(t <= 0.01);
			dd = [y(2); -2*z*y(2)-y(1)+h];
				

Step	T = linspace(0, 35, 200);
	z = 0.15;
	N = [0, 0, 1]; D = [1, 2*z, 1];
	step(tf(N, D), T)
			function **StepResponse**
			T = linspace(0, 35, 400);
			z = 0.15; IC = [0,0];
			[t, yy] = ode45(@**Stp**, T, IC, [], z);
			plot(t, yy(:,1), 'k-')
			
			function dd = **Stp**(t, y, z)
			dd = [y(2); -2*z*y(2)-y(1)+1];
				

Amplitude and phase	Om = linspace(0, 3, 200);
	z = 0.15;
	N = [0, 0, 1]; D = [1, 2*z, 1];
	bode(tf(N, D), Om)
			Om = linspace(0, 3, 200);
			z = 0.15;
			[a, p] = **H**(Om, z);
			subplot(2, 1, 1)
			loglog(Om, a, 'k-')
			subplot(2, 1, 2)
			semilogx(Om, p*180/pi, 'k-')
			
			function [a, p] = **H**(Om, z)
			a = 1./sqrt((1-Om.^2).^2 . . .
			    +(2*z*Om).^2);
			p = atan2(2*z*Om, 1-Om.^2);
				

plots the impulse response of the system described by `tf` and

```
step(tf(N,D))
```

plots the response of a system to a unit step function applied at $\tau = 0$.

The function

```
[wn,zeta] = damp(tf(N,D))
```

is used to determine the damping factors ζ and natural frequencies ω_n of a linear time-invariant system from its transfer function.

The second way to obtain the impulse and step response is to use `ode45`. The second way to get the amplitude–response function is to use Eq. (9.22). An evaluation of Eq. (9.22) will produce the Bode plot.

A summary of the two solution methods is given in Table 9.2. The functions from the Controls toolbox can also be used to study multidegree-of-freedom systems, as illustrated in Example 9.14.

Example 9.7 Estimation of natural frequency and damping factor for a damped oscillator

We shall determine the natural frequency and the damping for the system described by Eq. (9.18) for $\zeta = 0.3$. First, we model the system with $\zeta = 0.3$ and then we use `damp` to determine this value. The function `damp` determines the damping factors and associated natural frequencies from the poles of the transfer function. The script is

```
[wn,zeta] = damp(tf([0,0,1],[1,2*0.3,1]))
```

which upon execution gives $wn = \omega_n = 1$ and $zeta = \zeta = 0.3$.

Example 9.8 Curve fitting of the amplitude–response function

We generalize the magnitude of the amplitude–response function given by Eq. (9.22) to include an amplitude scale factor A_o. Then, the amplitude–response function becomes

$$H(\omega) = \frac{A_o}{\sqrt{(1 - (\omega/\omega_n)^2)^2 + (2\zeta\,\omega/\omega_n)^2}} \tag{9.29}$$

We shall simulate the acquisition of experimentally obtained data using `randn` to create twenty-five equally spaced points from $0 \le \omega \le 3$ as shown in the following script in the definition of the variable *hfit*. We restrict these simulated values to vary approximately ±18% around the value obtained from Eq. (9.29). The objective is to determine the parameters A_o, ω_n, and ζ that make Eq. (9.29) fit these data the "best." The MATLAB function `lsqcurvefit` from the Optimization toolbox will be used to perform this fit.[14]

[14] See Section 13.3.2 for additional applications of this function.

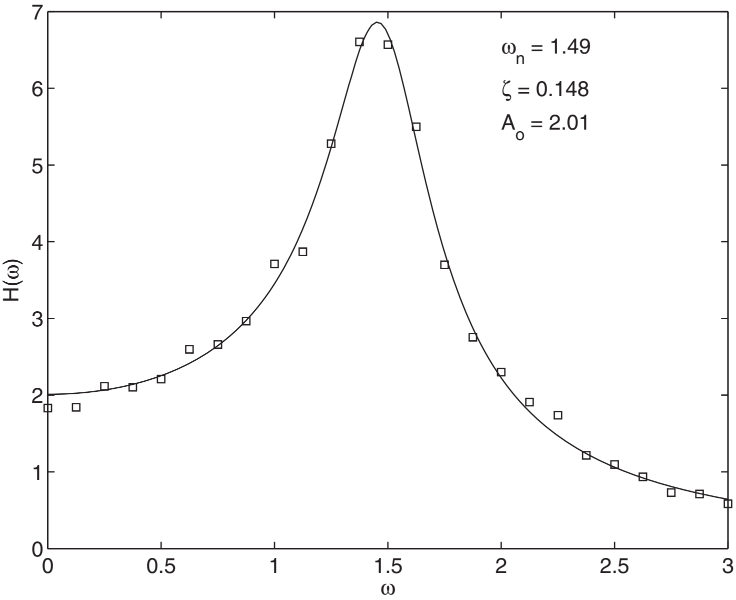


Figure 9.10 Curve-fit to simulated experimental data.

In order to generate the simulated data, we use Eq. (9.29) with $A_o = 2$, $\omega_n = 1.5$, and $\zeta = 0.15$. The estimated parameters that are found by `lsqcurvefit` should be very close to these values. The script that is used to carry out the parameter estimation through curve fitting is given below. The quantity *xo* is a vector of initial guesses for ω_n, ζ, and A_o, respectively, and is required by `lsqcurvefit`.

```
hh = inline('x(3)./sqrt((1-(w/x(1)).^2).^2+(2*x(2)*w/x(1)).^2)', 'x', 'w');
M = 25;    wfit = linspace(0, 3, M);
z = 0.15; wn = 1.5;    Ao = 2;
x = [wn, z, Ao];
hfit = hh(x, wfit).*(1+0.06*randn(1, M));
xo = [1.3, 0.1, 1];
x = lsqcurvefit(hh, xo, wfit, hfit);
wplt = linspace(0, 3, 150);
plot(wplt, hh(x,wplt), 'k-', wfit, hfit, 'ks')
text(2,6.5, ['\omega_n = ' num2str(x(1), 3)])
text(2,6.0, ['\zeta = ' num2str(x(2), 3)])
text(2,5.5, ['A_o = ' num2str(x(3), 3)])
xlabel('\omega')
ylabel('H(\omega)')
```

The execution of this script results in Figure 9.10, where it is seen that the values obtained from the curve-fitting procedure are very close to those that were used to create them.

Example 9.9 Single-degree-of-freedom system subjected to periodic pulse train forcing

We shall consider the response of a single-degree-of-freedom system to a periodic pulse train of period T and pulse duration per period t_d shown in Figure 9.11. The Fourier series representation is given by

$$f(\tau) = F_o\left[\alpha + \sum_{l=1}^{\infty} b_l \cos(\Omega_l \tau)\right] \tag{9.30}$$

where

$$b_l = \frac{2\alpha \sin(l\pi\alpha)}{(l\pi\alpha)}$$

and $\Omega_l = l\Omega_o$, $\Omega_o = \omega_o/\omega_n$, $\alpha = t_d/T$, $T = 2\pi/\omega_o$, and $\tau = \omega_n t$. The solution to Eq. (9.27) with the forcing given by Eq. (9.30) is[15]

$$y(\tau) = \alpha\left[1 + \sum_{l=1}^{\infty} c_l(\Omega_l) \sin(\Omega_l \tau - \theta(\Omega_l) + \psi_l)\right] \tag{9.31a}$$

where

$$\begin{aligned} c_l(\Omega_l) &= \frac{2}{\sqrt{(1-\Omega_l^2)^2 + (2\zeta\Omega_l)^2}}\left|\frac{\sin(l\pi\alpha)}{l\pi\alpha}\right| \\ \theta(\Omega_l) &= \tan^{-1}\frac{2\zeta\Omega_l}{1-\Omega_l^2} \\ \psi_l &= \tan^{-1}\frac{\sin(l\pi\alpha)/(l\pi\alpha)}{0} \end{aligned} \tag{9.31b}$$

and the form of ψ_l is required in order to obtain the proper quadrant.

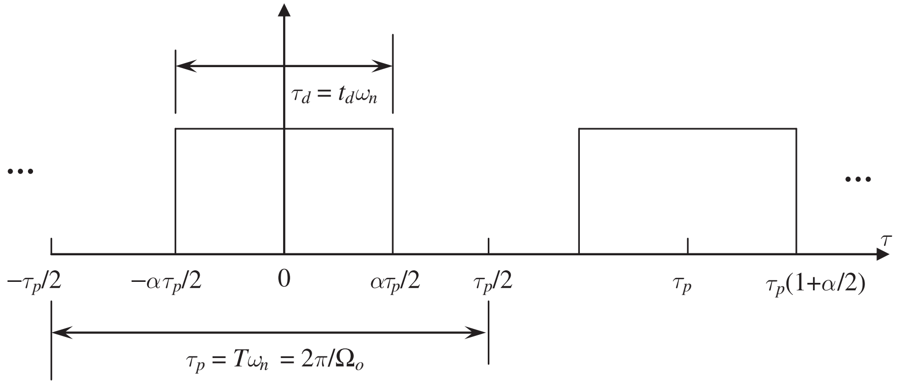


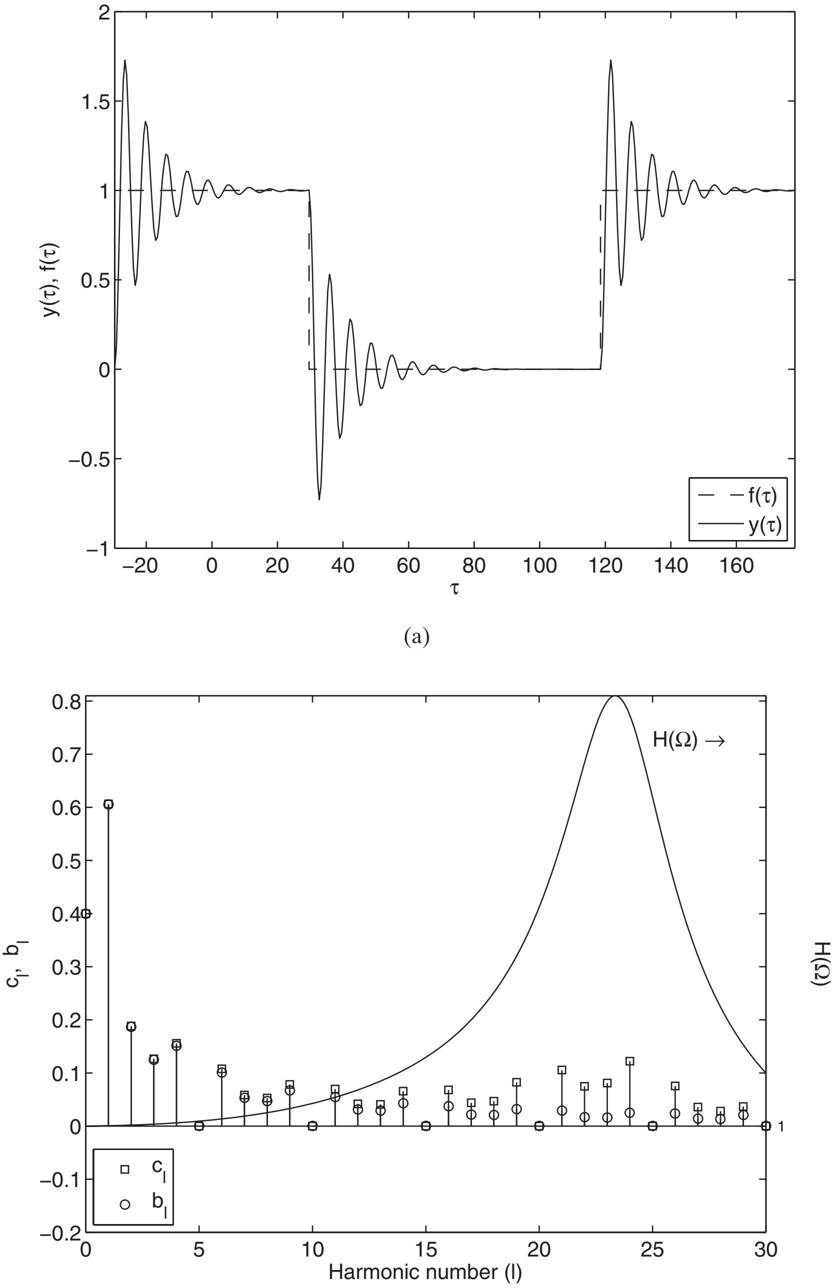
Figure 9.11 Nomenclature for a periodic pulse train.

[15] Balachandran and Magrab, *Vibrations*, Section 5.9.

We shall create a program that generates two separate graphs, one that plots Eqs. (9.30) and (9.31a) and another that plots b_l and c_l, $l = 1, 2, \ldots, 30$, at the discrete frequencies Ω_l and the system's amplitude–response function given by Eq. (9.22). In Eq. (9.22), it is noted that $\omega/\omega_n = \Omega$, which includes Ω_l as specific values. We shall use 200 terms in the summation appearing in Eqs. (9.30) and (9.31a) and the following values for the various parameters: $\Omega_o = 0.0424$, $\alpha = 0.4$, and $\zeta = 0.1$. In addition, we shall display the response from $-t_d/2 \le t \le T + t_d/2$ or in terms of the nondimensional quantities $-\alpha\tau_p/2 \le \tau \le \tau_p(1 + \alpha/2)$, where $\tau_p = T\omega_n = 2\pi/\Omega_o$. The program that performs the necessary operations to obtain these graphs is as follows.

```
n = 1:200;    alph = 0.4;    z = 0.1;    Omo = 0.0424;    Nt = 400;
Hr = inline('1./sqrt((1-(Omo*n).^2).^2+(2*z*Omo*n).^2)', 'n', 'Omo', 'z');
LL = pi/Omo*alph;    UL = pi/Omo*(2+alph);
tau = linspace(-LL, UL, Nt);
sn = 2*alph*sin(pi*n*alph)./(pi*n*alph);
Xnsn = abs(sn).*Hr(n, Omo, z);
thn = atan2(2*z*Omo*n, 1-(Omo*n).^2);
psi = atan2(sn, 0);
cnt = sin(Omo*n'*tau-repmat(thn', 1, Nt)+repmat(psi', 1, Nt));
y = alph+Xnsn*cnt;
figure(1)
plot([-LL -LL LL LL 2*LL/alph*(1-alph/2) 2*LL/alph*(1-alph/2) UL], . . .
         [0 1 1 0 0 1 1], 'k--')
hold on
plot(tau, y, 'k-')
legend('f(\tau)','y(\tau)', 'location', 'SouthEast');
r = axis;    r(1) = -LL;    r(2) = UL;    axis(r)
xlabel('\tau')
ylabel('y(\tau), f(\tau)')
figure(2)
M = 30;
plot(n(1:M), abs(Xnsn(1:M)), 'ks', n(1:M), abs(sn(1:M)), 'ko')
hold on
plot(0, alph,'ks', 0, alph, 'ko')
plot([n(1:M); n(1:M)], [zeros(1,M); abs(Xnsn(1:M))], 'k-')
plot([n(1:M); n(1:M)], [zeros(1,M); abs(sn(1:M))], 'k-')
r = axis;    r(3) = -0.2;    r(4) = 0.81;    axis(r)
nn = linspace(0, M, 300);
H = Hr(nn, Omo, z)-1;
plot(nn, H/max(H)*r(4), 'k-', [0 M], [0 0], 'k-')
ylabel('c_l, b_l')
xlabel('Harmonic number (l)')
text(M+.5, 0, num2str(1,3))
text(M+2.5, (r(3)+r(4))/2, 'H(\Omega)', 'rotation', -90, 'HorizontalAlignment'. . .
        'center')
text(25, 0.9*r(4), 'H(\Omega) \rightarrow')
legend('c_l','b_l', 'location', 'SouthWest');
```

The execution of this program results in Figure 9.12. In Figure 9.12a, we see that the normalized output displacement response overshoots the input pulse's amplitude



(a)

(b)

Figure 9.12 Comparison of a periodic pulse train in the (a) time domain and (b) frequency domain.

and then exhibits a decaying oscillation about the pulse's normalized height. This response is equivalent to the step response of a single-degree-of-freedom system as shown in the second row, third and fifth columns of Table 9.2. We see from Figure 9.12b that the amplitudes of the harmonics in the vicinity of the system's natural frequency are amplified and are responsible for the decaying oscillations of Figure 9.12a.

9.2.4 Nonlinear Systems: Free Oscillations

Through the use of separate examples, we shall now explore the free oscillations of three nonlinear systems; that is, systems for which $f(\tau) = 0$ and $g(y, \dot{y}) \neq 0$ in Eq. (9.9). The different types of nonlinearities considered are the following: (1) cubic spring; (2) dry friction (Coulomb) damping; and (3) piecewise linear springs.

Example 9.10 System with nonlinear spring

For this system, there is a nonlinear cubic spring with spring constant α, thus,

$$g(y, \dot{y}) = \alpha_o y^3 \tag{9.32a}$$

where

$$\alpha_o = \frac{\alpha \delta_{st}^2}{k} \tag{9.32b}$$

The nonlinear stiffness given by Eq. (9.32b) is said to have a softening spring when α has a negative value and a hardening spring when α has a positive value.[16] Thus, Eq. (9.9) becomes

$$\frac{d^2y}{d\tau^2} + 2\zeta\frac{dy}{d\tau} + y + \alpha_o y^3 = 0 \tag{9.33}$$

Then, from Eqs. (9.14) and (9.15), we have

$$\begin{aligned} \frac{dy_1}{d\tau} &= y_2 \\ \frac{dy_2}{d\tau} &= -2\zeta y_2 - y_1 - \alpha_o y_1^3 \end{aligned}$$

and

$$\begin{aligned} y_1(0) &= y_o \\ y_2(0) &= v_o \end{aligned}$$

We shall compare the responses of this system for the following three sets of initial conditions and values of α_o.

1. $y_o = -2.00$, $v_o = 2.00$, and $\alpha_o = 0.00$
2. $y_o = -2.00$, $v_o = 2.00$, and $\alpha_o = -0.25$
3. $y_o = -2.00$, $v_o = 2.31$, and $\alpha_o = -0.25$.

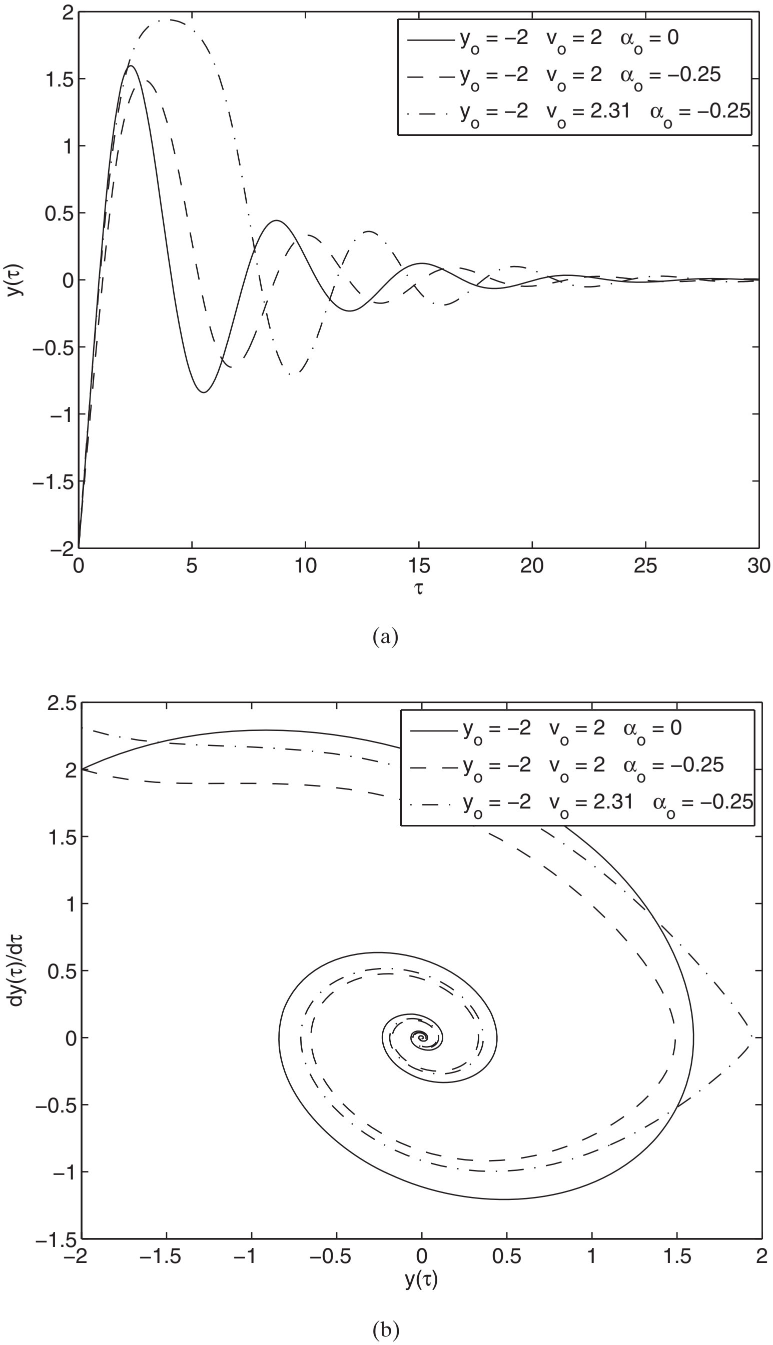
[16] Nayfeh and Balachandran, *Applied Nonlinear Dynamics*.

The first case corresponds to a linear system, while the second and third cases correspond to a nonlinear system with a softening spring. The initial conditions are the same in the first two cases and different for the third case.

The following script generates the responses for the three cases. In addition, the periods of the first three cycles of the responses are determined by using **ZeroCrossing**, which is given in Example 9.5.

```
function Example9_10
z = 0.2;   alphao = [0.00, -0.25, -0.25];
yo = [-2.00, -2.00, -2.00];   vo = [ 2.00, 2.00, 2.31];
d = char('Linear ', 'Nonlinear ', 'Nonlinear ');
lintyp = char('-k', '--k', '-.k');
for n = 1:3
  A{n} = ['y_o = ' num2str(yo(n)) ' v_o = ' num2str(vo(n)) ' \alpha_o = ' ...
           num2str(alphao(n))];
  [t, y] = ode45(@NonLinearOscillatorFree, linspace(0, 30, 401), ...,
          [ ], z, alphao(n));
  figure(1)
  plot(t, y(:,1), lintyp(n,:))
  hold on
  if n == 3
    legend(A{1}, A{2}, A{3})
    xlabel('\tau')
    ylabel('y(\tau)')
  end
  figure(2)
  plot(y(:,1), y(:,2), lintyp(n,:))
  hold on
  if n == 3
    legend(A{1}, A{2}, A{3})
    xlabel('y(\tau)')
    ylabel('dy(\tau)/d\tau')
  end
  disp (['Case ' num2str(n) ' ' d(n,:)])
  disp(A{n})
  Tp = ZeroCrossing(t, y(:,1));
  [ymx, Imax] = max(y(:,1));
  Per = Tp(3:2:7)-Tp(1:2:5);
  disp(' ')
  disp('Period')
  disp(num2str(Per'))
  disp(' ')
end

function y = NonLinearOscillatorFree(t, y, z, alphao)
y = [y(2); -2*z*y(2)-y(1)-alphao*y(1)^3];
```

When this script is executed, the output to the command window consists of the periods for three consecutive periods. Graphs of the free oscillations for these three cases are shown in Figure 9.13.



(a)

(b)

Figure 9.13 Free responses of damped, linear, and nonlinear oscillators: (a) Displacement histories. (b) Phase portraits. The solid line is the linear case; the other two lines are for the nonlinear cases.

```
Case 1 Linear
y_o = -2   v_o = 2   \alpha_o = 0
  Period
  6.4131
  6.4126
  6.4128
Case 2 Nonlinear
y_o = -2   v_o = 2   \alpha_o = -0.25
  Period
  7.5969
  6.4564
  6.4157
Case 3 Nonlinear
y_o = -2   v_o = 2.31   \alpha_o = -0.25
  Period
  10.4388
  6.4641
  6.41658
```

In Case 1, which corresponds to the linear system, the period of the damped oscillation remains essentially constant over each cycle. In Cases 2 and 3, which correspond to a nonlinear system, the periods of the damped oscillation from the first cycle are significantly different from those of the subsequent periods of motion. The effect of the nonlinearity is typically pronounced when the amplitudes of motion are "large," as it is in the first cycle of oscillation in Case 3. The behavior of the systems in Cases 2 and 3 approaches that of the linear system (Case 1) as the amplitudes of motion become "smaller."

As in the corresponding linear case, the orbits of the nonlinear system initiated from these sets of the initial conditions are attracted toward the stable equilibrium position $(0, 0)$ in the phase portrait. The spirals in the phase portrait indicate that the corresponding motions of the nonlinear system are underdamped. For "small'' oscillations around the stable equilibrium position, the nonlinear system should behave like a linear system. Examining the responses initiated from the two sets of initial conditions, the feature observed for Case 3 around the first extremum in the time history is not typical of the response of a linear system. In this case, the trajectory comes "close" to the unstable equilibrium position $(2, 0)$ of the system and the system motion is affected. Unlike a linear system, a nonlinear system can have multiple equilibrium positions, not all of which are necessarily stable.[17]

Example 9.11 System with Coulomb damping

In this example, we consider a spring–mass system with nonlinear damping known as Coulomb damping or dry friction. In this case, we set $\zeta = 0$ and

$$g(y, \dot{y}) = \mu \, \mathrm{signum}(dy/d\tau)$$

[17] *Ibid.*

where the constant μ is called the friction coefficient. Thus, Eq. (9.9) becomes

$$\frac{d^2y}{d\tau^2} + y + \mu \operatorname{signum}(dy/d\tau) = 0 \tag{9.34}$$

The dry friction force, which is a piecewise constant function of the velocity, is described by the signum function in Eq. (9.34). It has a value of +1 when the velocity is positive and a value of −1 when the velocity is negative. Since the dry friction force varies nonlinearly with respect to velocity, the system is nonlinear. When the system is set into motion, the system comes to rest when the spring force is no longer able to overcome the dry friction force. This means that the system will stop when

$$\frac{dy}{d\tau} = 0 \quad \text{and} \quad |y| \le \mu \tag{9.35}$$

The nonlinear system described by Eq. (9.34) has multiple equilibrium positions, and the locus of these equilibriums in the phase space is the straight line joining the points $(-\mu, 0)$ and $(\mu, 0)$. A closed-form solution of Eq. (9.34) can be obtained since the system is linear in the region $dy/d\tau > 0$ and linear in the region $dy/d\tau < 0$. However, here, we shall find the solution numerically.

From Eqs. (9.16) and (9.17), we have

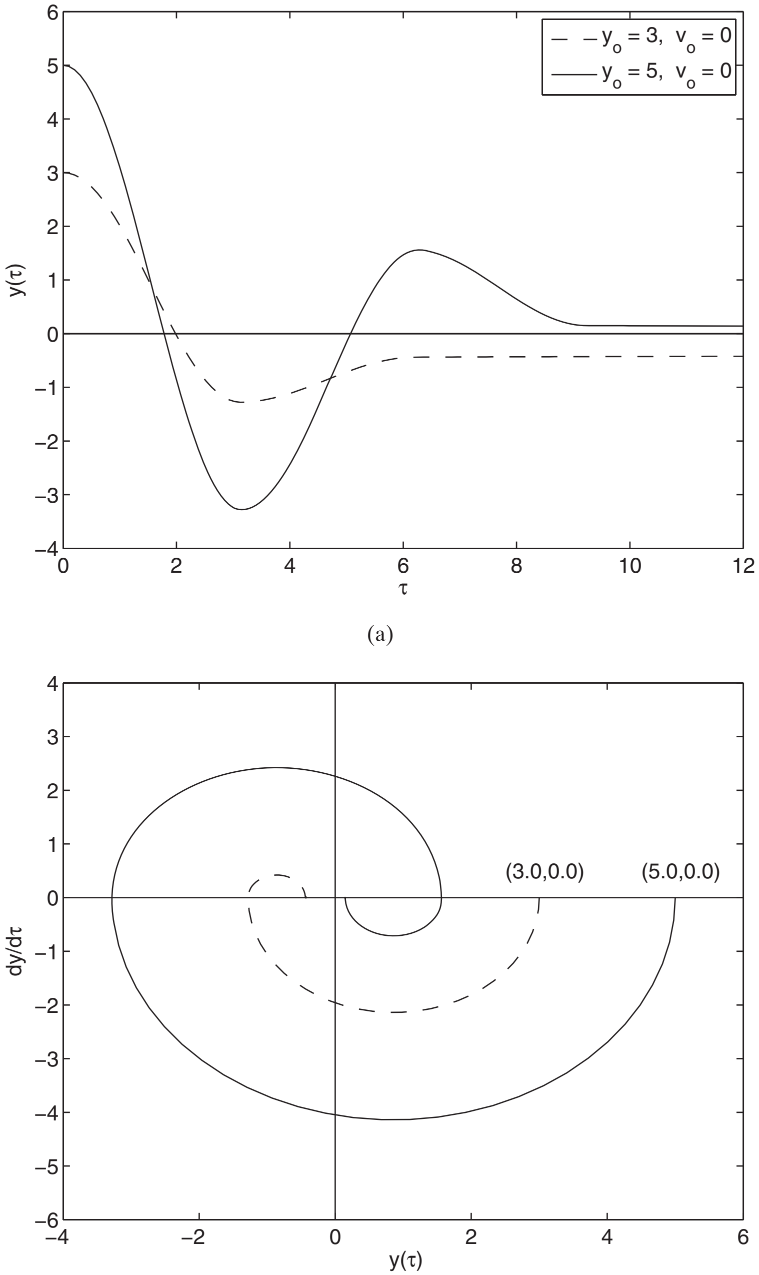
$$\begin{aligned} \frac{dy_1}{d\tau} &= y_2 \\ \frac{dy_2}{d\tau} &= -y_1 - \mu \operatorname{signum}(dy/d\tau) \end{aligned} \tag{9.36}$$

and

$$\begin{aligned} y_1(0) &= y_o \\ y_2(0) &= v_o \end{aligned}$$

A subfunction called **CoulombDamping** is created to represent Eqs. (9.35) and (9.36). The numerical solutions of Eq. (9.34) are obtained for $\mu = 0.86$ and the following two sets of initial conditions: (a) $x(0) = 3.0$ and $\dot{x}(0) = 0$; that is, $y_1(0) = 3.0$ and $y_2(0) = 0.0$; and (b) $x(0) = 5.0$ and $\dot{x}(0) = 0$; that is, $y_1(0) = 5.0$ and $y_2(0) = 0.0$. The program is as follows.

```
function Example9_11
mu = 0.86;    yo = [3.0, 5.0];    vo = [0.0, 0.0];
tspan = linspace(0, 12, 120);
options = odeset('AbsTol', [1e-3, 1e-3]);
lintyp = char('--k', '-k');    lab = ', v_o = 0';
for n = 1:2
   [t, y] = ode45(@CoulombDamping, tspan, [yo(n) vo(n)]', options, mu);
   figure(1)
   plot(t, y(:,1), lintyp(n,:))
   hold on
```



(a)

(b)

Figure 9.14 Free response of an oscillator with dry friction: (a) Displacement histories. (b) Phase portraits.

```
  figure(2)
  plot(y(:,1), y(:,2), lintyp(n,:))
  hold on
end
figure(1)
xlabel('\tau')
ylabel('y(\tau)')
axis([0.0, 12.0, -4.0, 6.0])
plot([0 12], [0 0], 'k-')
legend(['y_o = ' num2str(yo(1)) lab], ['y_o = ' num2str(yo(2)) lab])
plot([0, 12], [0, 0], 'k-')
figure(2)
xlabel(' y(\tau)')
ylabel('dy/d\tau')
text(2.5, 0.5, '(3.0,0.0)')
text(4.5, 0.5, '(5.0,0.0)')
plot([-4 6], [0 0], 'k-', [0 0], [-6 4], 'k-')
axis([-4.0, 6.0, -6.0, 4.0])

function xdot = CoulombDamping(t, x, mu)
if (abs(x(1)) <= mu) && (x(2) == 0.0)
  xdot = [0; 0];
else
  xdot = [x(2); -mu*sign(x(2))-x(1)];
end
```

The absolute tolerance specified for each state is larger than the default value of 10^{-6} to speed up the computations. Execution of the script produces the results shown in Figure 9.14. The systems come to rest at two different positions, and the respective rest positions are reached at two different times. This example illustrates the fact that the long time response of a nonlinear system depends upon the initial conditions. By contrast, the asymptotic response of a damped linear system is independent of the initial conditions.

Example 9.12 System with piecewise linear springs

Consider the free oscillations of the nonlinear system shown in Figure 9.15. All of the springs are linear; however, the mass is straddled by two linear elastic spring-stops that are not contacted until the mass has been displaced by an amount d in either direction. The stiffness of the springs is proportional to the attached spring by a constant of proportionality μ ($\mu \geq 0$). When $\mu = 0$, we have the standard linear single-degree-of-freedom system, and when $\mu > 1$, the elastic spring-stops are stiffer than the spring that is permanently attached to the mass. For this system,

$$g(y, \dot{y}) \rightarrow g(w, \dot{w}) = \mu h(w)$$

where

$$\begin{aligned} h(w) &= 0 \quad |w| \leq 1 \\ &= w - \text{signum}(w) \quad |w| > 1 \end{aligned} \tag{9.37}$$

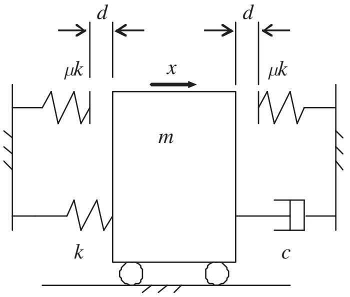


Figure 9.15 Single-degree-of-freedom system with additional springs that are not contacted until the mass displaces a distance d in either direction.

and the signum function is $+1$ when $w > 0$ and is -1 when $w < 0$ and $w = w(\tau) = x(\tau)/d$. Then Eq. (9.9), with $f(\tau) = 0$, becomes[18]

$$\frac{d^2w}{d\tau^2} + 2\zeta\frac{dw}{d\tau} + w + \mu h(w) = 0 \tag{9.38}$$

From Eqs. (9.16) and (9.17), we have

$$\frac{dw_1}{d\tau} = w_2$$

$$\frac{dw_2}{d\tau} = -w_1 - \mu h(w)$$

and

$$w_1(0) = X_o/d = w_o$$

$$w_2(0) = V_o/(d\omega_n) = v_o$$

Although it is possible to find an analytical solution for this piecewise linear system, here we obtain a numerical solution using `ode45`. We shall determine the response of this system when the damping factor $\zeta = 0.15$, $w_o = 0$, $v_o = 10$, and the values of μ are 0, 1, and 10. From this response, we shall determine the first four periods for each value of μ. The script that is used to determine a numerical solution to Eq. (9.38) for these parameters is as follows:

```
function Example9_12
c = char('k-', 'k--', 'k-.');
mu = [0, 1, 10];    z = 0.15;    wo = 10;
```

[18] H. Y. Hu. "Primary Resonance of a Harmonically Forced Oscillator with a Pair of Symmetric Set-up Elastic Stops," *Journal of Sound Vibration*, **207**, No. 3, 1997, pp. 393–401.

```
disp('Periods')
for n = 1:length(mu)
   [t, w] = ode45(@sdofstops, [0, 30], [0 wo]', [], z, mu(n));
   plot(t, w(:,1), c(n,:))
   hold on
   Tp = ZeroCrossing(t, w(:,1));
   Per = Tp(3:2:9)-Tp(1:2:7);
   disp(['mu = ' num2str(mu(n)) ': ' num2str(Per)])
end
legend(['\mu = ' num2str(mu(1))], ['\mu = ' num2str(mu(2))], . . .
            ['\mu = ' num2str(mu(3))]);
plot([0 t(end)], [0 0], c(1,:))
plot(t, wo*exp(-t*z), 'k:', t, -wo*exp(-t*z), 'k:')
xlabel('\tau')
ylabel('w(\tau)')

function q = sdofstops(t, y, z, mu)
if abs(y(1)) <= 1
   h = 0;
else
   h = y(1)-sign(y(1));
end
q = [y(2); -2*z*y(2)-y(1)-mu*h];
```

The function **ZeroCrossing** is described in Example 9.5.

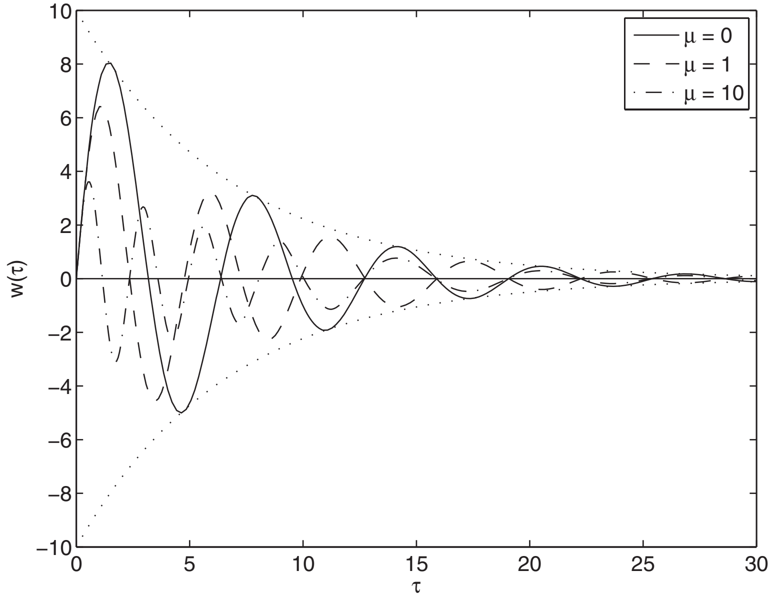


Figure 9.16 Response of system shown in Figure 9.15 with prescribed initial velocity $w_o = 10$.

The execution of this program gives the results shown in Figure 9.16 and displays to the command window the following values for the periods:

```
Periods
  mu = 0:  6.3539   6.355     6.3549   6.3544
  mu = 1:  4.9081   5.4775   6.3419   6.3559
  mu = 10:   2.4537   2.8021   3.6293   5.8175
```

We see that the introduction of the spring-stops decreases the amplitude of the mass. In addition, it has the effect of initially decreasing the period of oscillation, which is equivalent to increasing the system's natural frequency.

9.2.5 Nonlinear Systems: Forced Oscillations

The response of a system with a cubic spring to harmonic excitations will be determined in the time domain using `ode45`. In addition, the corresponding information in the frequency domain will be determined with `fft` by using the procedures illustrated in Section 5.4.7. The initial conditions are zero and the harmonic forcing at frequency ω is given by

$$F(\tau) = F_o \cos(\omega t) = F_o \cos(\Omega\tau) \tag{9.39}$$

where $\Omega = \omega/\omega_n$ is the nondimensional excitation frequency ratio. The nonlinearity is given by Eq. (9.32). Then, Eq. (9.9) becomes

$$\frac{d^2y}{d\tau^2} + 2\zeta\frac{dy}{d\tau} + y + \alpha_o y^3 = \cos(\Omega\tau) \tag{9.40}$$

From Eqs. (9.16) and (9.17), we have

$$\begin{aligned}\frac{dy_1}{d\tau} &= y_2\\ \frac{dy_2}{d\tau} &= -2\zeta y_2 - y_1 - \alpha_o y_1^3 - \cos(\Omega\tau)\end{aligned}$$

and

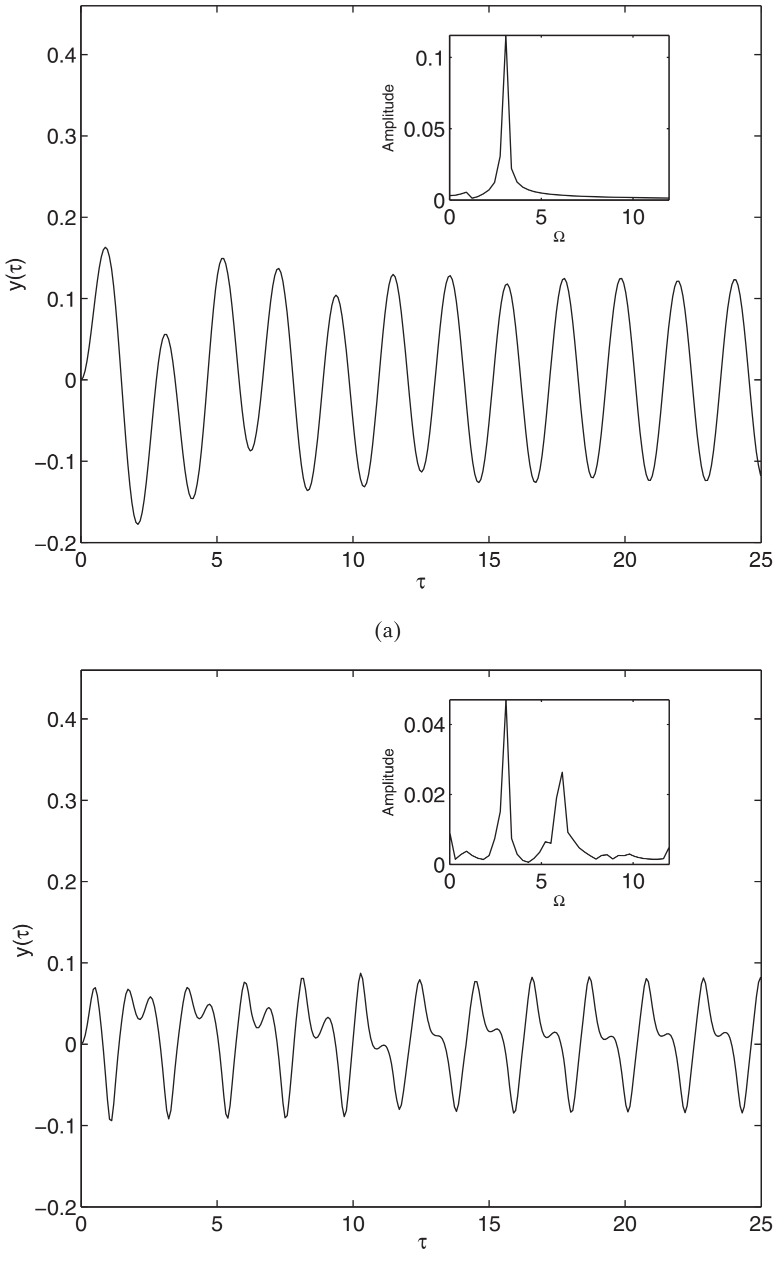
$$\begin{aligned}y_1(0) &= 0\\ y_2(0) &= 0\end{aligned}$$

At a given excitation frequency, we shall determine the steady-state response of the system given by Eq. (9.40)—that is, after the transients have died out—and examine the spectral content of this steady-state response employing `fft` for the following cases: (1) $\alpha_o = 0$ (linear system) and (2) $\alpha_o = 6{,}250$ (nonlinear system).

Equation (9.40) is numerically integrated using `ode45`. In addition, we assume that $\zeta = 0.2$ and $\Omega = 3.0$. The excitation frequency Ω has been chosen to be three times the natural frequency of the system. We shall examine the time histories for $0 \le \tau \le 30$ and take 12,000 samples in this region. This is equivalent to the data being acquired every $\tau_s = 30/12{,}000 = 0.0025$. Therefore, the (dimensionless) sampling frequency is $2\pi/\tau_s = 800\pi$. This is far in excess of what is necessary to sample the response based on the excitation frequency $\Omega = 3$. The consequence of this oversampling is that we have to truncate the spectrum plot; thus, we shall display only the first forty values. Also, we let $N_{\text{start}} = 3{,}200$ and the number of samples used by fft is $N = 2^{13} = 8{,}192$. The justification for the choice of N_{start} will be discussed subsequently. The script is as follows:

```
function Example9_2_5
z = 0.2;   alphao = [0, 6250];
Omega = 3;   M = 12000;
tspan = linspace(0, 30, M);
N = 2^13;   Nstart = 3200;   Fs = M/30;
f = (Fs*(0:N-1)/N)*2.0*pi;
for m = 1:2
  [t, y] = ode45(@ForcedNLOscillator, tspan, [0 0]', [], z, alphao(m), Omega);
  figure(m);
  plot(t, y(:,1), 'k-');
  axis([0, 25, -0.2, 0.46]);
  xlabel('\tau');
  ylabel('y(\tau)');
  axes('position', [0.55, 0.63, 0.25, 0.25])
  Amp = abs(fft(y(Nstart:Nstart+N, 1), N))/N;
  plot(f(1:40), 2*Amp(1:40), 'k-');
  v = axis;   v(2) = f(40);   v(4) = max(2*Amp(1:40));   axis(v)
  xlabel('\Omega');
  ylabel('Amplitude');
end

function xdot = ForcedNLOscillator(t, x, zeta, alphao, Omega)
xdot = [x(2); -2*zeta*x(2)-x(1)- alphao*x(1)^3+cos(Omega*t)];
```

Execution of the program results in Figure 9.17, where it is seen that the responses of both the linear and nonlinear systems reach steady state when $\tau \ge 8$. This time corresponds to an index $N_{\text{start}} = 3{,}200$. Although both of the steady-state responses have a period equal to the period of the harmonic forcing function, they have different characteristics that can be more clearly distinguished in the frequency domain. The corresponding spectral response appears in the upper right-hand corner of each figure where it is seen that the amplitude spectrum of the displacement response in the nonlinear case shows spectral peaks at the forcing frequency Ω and at integer multiples of it. The additional peaks are due to the cubic nonlinearity of the spring. In the linear case, there is only one spectral peak, which corresponds to the excitation frequency. This example illustrates that the response of a nonlinear system can have spectral components different from the excitation (input) frequency.



(a)

(b)

Figure 9.17 (a) Linear system subjected to a harmonic forcing at $\Omega = 3$.
(b) Nonlinear system subjected to a harmonic forcing at $\Omega = 3$ and $\alpha_o = 6{,}250$.

9.3 SYSTEMS WITH MULTIPLE DEGREES OF FREEDOM

9.3.1 Two-Degree-of-Freedom Systems: Free and Forced Oscillations[19]

The governing equations of motion for the two-degree-of-freedom system shown in Figure 9.18 about its static equilibrium position are

$$m_1\frac{d^2x_1}{dt^2} + (c_1 + c_2)\frac{dx_1}{dt} + (k_1 + k_2)x_1 - c_2\frac{dx_2}{dt} - k_2x_2 = f_1(t) \qquad (9.41)$$
$$m_2\frac{d^2x_2}{dt^2} + c_2\frac{dx_2}{dt} + k_2x_2 - c_2\frac{dx_1}{dt} - k_2x_1 = f_2(t)$$

subject to the initial conditions

$$x_1(0) = X_1 \quad \dot{x}_1(0) = \frac{dx_1(0)}{dt} = V_1 \qquad (9.42)$$
$$x_2(0) = X_2 \quad \dot{x}_2(0) = \frac{dx_2(0)}{dt} = V_2$$

If we introduce the following definitions,

$$\omega_{nj}^2 = \frac{k_j}{m_j}, \quad m_r = \frac{m_2}{m_1}, \quad \tau = \omega_{n1}t \quad j = 1, 2 \qquad (9.43)$$
$$\omega_r = \frac{\omega_{n2}}{\omega_{n1}} = \frac{1}{\sqrt{m_r}}\sqrt{\frac{k_2}{k_1}}, \quad 2\zeta_j = \frac{c_j}{m_j\omega_{nj}} \quad j = 1, 2$$

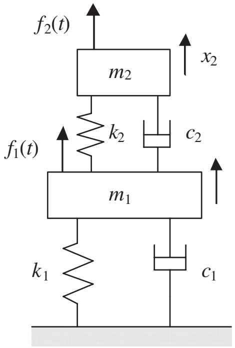


Figure 9.18 Two-degree-of-freedom system subjected to external forces.

[19] Balachandran and Magrab, *Vibrations*, Chapter 8.

then Eq. (9.41) can be written as

$$
\begin{aligned}
&\frac{d^2x_1}{d\tau^2} + (2\zeta_1 + 2\zeta_2 m_r\omega_r)\frac{dx_1}{d\tau} + (1 + m_r\omega_r^2)x_1 - 2\zeta_2 m_r\omega_r\frac{dx_2}{d\tau} - m_r\omega_r^2 x^2 = \frac{f_1(\tau)}{k_1} \\
&\qquad\qquad \frac{d^2x_2}{d\tau^2} + 2\zeta_2\omega_r\frac{dx_2}{d\tau} + \omega_r^2 x_2 - 2\zeta_2\omega_r\frac{dx_1}{d\tau} - \omega_r^2 x_1 = \frac{f_2(\tau)}{k_1 m_r}
\end{aligned}
\tag{9.44}
$$

and the initial conditions given by Eq. (9.42) become

$$
\begin{aligned}
x_1(0) &= X_{o1} \quad \dot{x}_1(0) = \frac{dx_1(0)}{d\tau} = \frac{V_{o1}}{\omega_{n1}} = \hat{V}_{o1} \\
x_2(0) &= X_{o2} \quad \dot{x}_2(0) = \frac{dx_2(0)}{d\tau} = \frac{V_{o2}}{\omega_{n1}} = \hat{V}_{o2}
\end{aligned}
\tag{9.45}
$$

Notice that $\hat{V}_{oj}$ has the units of length.

Laplace Transform

If $X_1(s)$ and $X_2(s)$ designate the Laplace transform of $x_1(\tau)$ and $x_2(\tau)$, respectively, and $F_1(s)$ and $F_2(s)$ designate the Laplace transform of $f_1(\tau)$ and $f_2(\tau)$, respectively, then taking the Laplace transform of Eq. (9.44), we can solve for $X_1(s)$ and $X_2(s)$ to obtain

$$
\begin{aligned}
X_1(s) &= \frac{K_1(s)E(s)}{D(s)} + \frac{K_2(s)B(s)}{D(s)} \\
X_2(s) &= \frac{K_1(s)C(s)}{D(s)} + \frac{K_2(s)A(s)}{D(s)}
\end{aligned}
\tag{9.46}
$$

where

$$
\begin{aligned}
A(s) &= s^2 + 2(\zeta_1 + \zeta_2 m_r\omega_r)s + 1 + m_r\omega_r^2 \\
B(s) &= 2\zeta_2 m_r\omega_r s + m_r\omega_r^2 \\
C(s) &= B(S)/m_r = 2\zeta_2\omega_r s + \omega_r^2 \\
E(s) &= s^2 + 2\zeta_2\omega_r s + \omega_r^2 \\
D(s) &= s^4 + 2\big(\zeta_1 + \zeta_2\omega_r(1 + m_r)\big)s^3 + \big(1 + \omega_r^2(1 + m_r) + 4\zeta_1\zeta_2\omega_r\big)s^2 \\
&\qquad + 2\big(\zeta_2\omega_r + \zeta_1\omega_r^2\big)s + \omega_r^2
\end{aligned}
\tag{9.47a}
$$

and

$$
\begin{aligned}
K_1(s) &= \frac{F_1(s)}{k_1} + \hat{V}_{o1} + (s + 2\zeta_1 + 2\zeta_2 m_r\omega_r)\,X_{o1} - 2\zeta_2 m_r\omega_r X_{o2} \\
K_2(s) &= \frac{F_2(s)}{k_1 m_r} + \hat{V}_{o2} + (s + 2\zeta_2\omega_r)X_{o2} - 2\zeta_2\omega_r X_{o1}
\end{aligned}
\tag{9.47b}
$$

Transfer Functions

To determine the transfer functions of the system, denoted $G_{ij}(s)$, we consider two cases: (1) the initial conditions are zero and $f_2(t) = 0$ and a force impulse of magnitude F_o is applied to m_1; that is $f_1(t) = F_o\delta(t)$; and (2) the initial conditions are zero and $f_1(t) = 0$ and a force impulse of magnitude F_o is applied to m_2; that is, $f_2(t) = F_o\delta(t)$. Then for case 1, $F_1(s) = F_o$ and for case 2, $F_2(s) = F_o$. Using these results in Eqs. (9.46) and (9.47b), we find for the first case that

$$\begin{aligned} G_{11}(s) &= \frac{X_1(s)}{F_1(s)} = \frac{E(s)}{k_1D(s)} \\ G_{21}(s) &= \frac{X_2(s)}{F_1(s)} = \frac{C(s)}{k_1D(s)} \end{aligned} \tag{9.48a}$$

and for the second case that

$$\begin{aligned} G_{12}(s) &= \frac{X_1(s)}{F_2(s)} = \frac{B(s)}{k_1m_rD(s)} \\ G_{22}(s) &= \frac{X_2(s)}{F_2(s)} = \frac{A(s)}{k_1m_rD(s)} \end{aligned} \tag{9.48b}$$

where, in the subscripts of $G_{il}(s)$, i indicates the response of mass m_i when an impulse force is applied to mass m_l.[20]

Frequency–Response Functions

To obtain the frequency–response functions, we replace s by $j\omega/\omega_{n1} = j\Omega$ and define the frequency–response functions as

$$\begin{aligned} k_1G_{11}(j\Omega) &= \frac{E(j\Omega)}{D(j\Omega)} \\ k_1G_{21}(j\Omega) &= \frac{C(j\Omega)}{D(j\Omega)} \\ k_1G_{12}(j\Omega) &= \frac{B(j\Omega)}{m_rD(j\Omega)} \\ k_1G_{22}(j\Omega) &= \frac{A(j\Omega)}{m_rD(j\Omega)} \end{aligned} \tag{9.49a}$$

[20] Equations (9.46) and (9.48) are in a form that can be used directly in `tf` and then in `impulse`, `step`, and `bode` as described in Section 9.2.3.

where

$$\begin{aligned}
A(j\Omega) &= -\Omega^2 + 2j(\zeta_1 + \zeta_2 m_r\omega_r)\Omega + 1 + m_r\omega_r^2 \qquad (9.49b)\\
B(j\Omega) &= m_r\omega_r^2 + j2\zeta_2 m_r\omega_r\Omega\\
C(j\Omega) &= B(j\Omega)/m_r = \omega_r^2 + j2\zeta_2\omega_r\Omega\\
E(j\Omega) &= -\Omega^2 + j2\zeta_2\omega_r\Omega + \omega_r^2\\
D(j\Omega) &= \Omega^4 - 2j\big(\zeta_1 + \zeta_2\omega_r(1 + m_r)\big)\Omega^3 - \big(1 + \omega_r^2(1 + m_r) + 4\zeta_1\zeta_2\omega_r\big)\Omega^2\\
&\quad + 2j\big(\zeta_2\omega_r + \zeta_1\omega_r^2\big)\Omega + \omega_r^2
\end{aligned}$$

The magnitude of the frequency response function is given by

$$H_{il}(\Omega) = k_1|G_{il}(j\Omega)| \qquad i, l = 1, 2 \tag{9.50}$$

Natural Frequencies

The natural frequencies of the undamped two-degree-of-freedom system is obtained by setting $\zeta_1 = \zeta_2 = 0$ and finding the positive values of $\Omega_{1,2}$ that satisfy $D(j\Omega) = 0$; that is,

$$\Omega^4 - \big(1 + \omega_r^2(1 + m_r)\big)\Omega^2 + \omega_r^2 = 0 \tag{9.51}$$

Therefore,

$$\Omega_{1,2} = \sqrt{\frac{1}{2}\left(1 + \omega_r^2(1 + m_r) \mp \sqrt{\big(1 + \omega_r^2(1 + m_r)\big)^2 - 4\omega_r^2}\right)} \tag{9.52}$$

State–Space Form

Equations (9.44) and (9.45) can be put into state–space form by introducing the variables

$$\begin{aligned}
y_1 &= x_1 \qquad & y_3 &= x_2\\
y_2 &= \frac{dx_1}{d\tau} \qquad & y_4 &= \frac{dx_2}{d\tau}
\end{aligned}$$

Then Eqs. (9.44) and (9.45), respectively, become

$$\begin{aligned}
\frac{dy_1}{d\tau} &= y_2\\
\frac{dy_2}{d\tau} &= -(2\zeta_1 + 2\zeta_2 m_r\omega_r)y_2 - (1 + m_r\omega_r^2)y_1 + 2\zeta_2 m_r\omega_r y_4 + m_r\omega_r^2 y_3 + \frac{f_1(\tau)}{k_1}\\
\frac{dy_3}{d\tau} &= y_4\\
\frac{dy_4}{d\tau} &= -2\zeta_2\omega_r y_4 - \omega_r^2 y_3 + 2\zeta_2\omega_r y_2 + \omega_r^2 y_1 + \frac{f_2(\tau)}{k_1 m_r} \qquad (9.53)
\end{aligned}$$

and

$$\begin{aligned} y_1(0) &= X_{o1} & y_2(0) &= \hat{V}_{o1} \\ y_3(0) &= X_{o2} & y_4(0) &= \hat{V}_{o2} \end{aligned} \tag{9.54}$$

We shall now use these results in the following examples.

Example 9.13 Two-degree-of-freedom system subjected to an initial velocity

We assume that each mass of the two-degree-of-freedom system is subjected to an initial velocity $\hat{V}_{o1} = \hat{V}_{o2} = 1$ and that $X_{o1} = X_{o2} = f_1(\tau) = f_2(\tau) = 0$. Then Eq. (9.53) simplifies to

$$\begin{aligned} \frac{dy_1}{d\tau} &= y_2 \\ \frac{dy_2}{d\tau} &= -(2\zeta_1 + 2\zeta_2 m_r\omega_r)y_2 - (1 + m_r\omega_r^2)y_1 + 2\zeta_2 m_r\omega_r y_4 + m_r\omega_r^2 y_3 \\ \frac{dy_3}{d\tau} &= y_4 \\ \frac{dy_4}{d\tau} &= -2\zeta_2\omega_r y_4 - \omega_r^2 y_3 + 2\zeta_2\omega_r y_2 + \omega_r^2 y_1 \end{aligned}$$

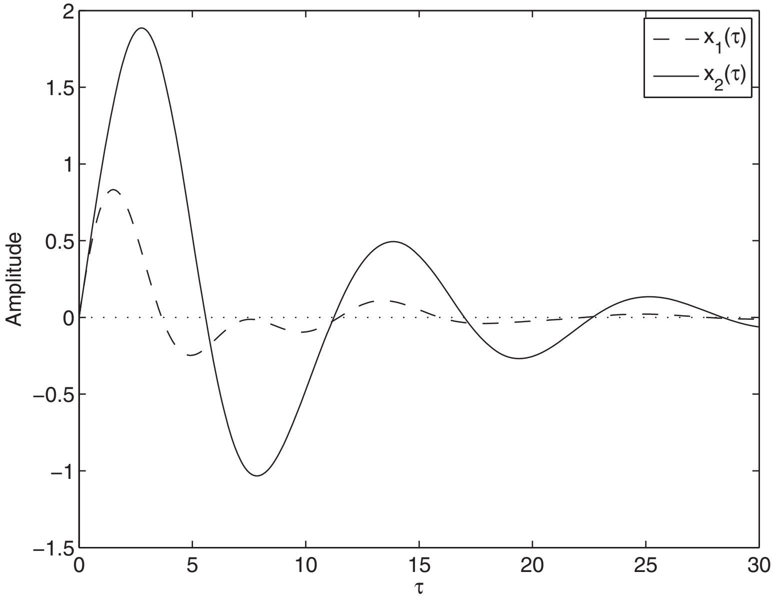


Figure 9.19 Displacement responses of the masses shown in Figure 9.18 when each mass is subjected to the same initial velocity.

and Eq. (9.54) becomes

$$y_1(0) = 0 \qquad y_2(0) = 1$$

$$y_3(0) = 0 \qquad y_4(0) = 1$$

We shall use these simplified equations and ode45 to obtain the time-varying displacement response of each mass. We assume that $m_r = 0.3$, $\omega_r = 0.6$, $\zeta_1 = 0.15$, and $\zeta_2 = 0.25$. The program is as follows:

```
function Example9_13
mr = 0.3;   wr = 0.6;   z1 = 0.15;   z2 = 0.25;
tt = linspace(0, 30, 300);
[t, y] = ode45(@InitialVelocity, tt, [0 1 0 1]', [], z1, z2, mr, wr);
plot(t, y(:,1), 'k--', t, y(:,3), 'k-', [0 30], [0 0], 'k:')
xlabel('\tau')
ylabel('Amplitude')
legend('x_1(\tau)', 'x_2(\tau)')

function dd = InitialVelocity(t, y, z1, z2, mr, wr)
A = -2*(z1+z2*mr*wr)*y(2)-(1+mr*wr^2)*y(1)+2*z2*mr*wr*y(4) . . .
    +mr*wr^2*y(3);
B = -2*z2*wr*y(4)-wr^2*y(3)+2*z2*wr*y(2)+wr^2*y(1);
dd = [y(2); A; y(4); B];
```

Execution of this program results in Figure 9.19.

Example 9.14 Impulse and step responses of a two-degree-of-freedom system

In this example, we shall compute the time-varying response of the displacements of the two masses shown in Figure 9.18 using three different methods: (a) impulse and step from the Controls toolbox; (b) ode45; and (c) ilaplace from the Symbolic toolbox. For each of these methods, we shall assume that $m_1 = 100$ kg, $m_2 = 10$ kg, $k_1 = 10{,}000$ N/m, $k_2 = 1{,}960$ N/m, $c_1 = 400$ Ns/m, $c_2 = 56$ Ns/m and $0 \le \tau \le 30$. In each method, we shall provide the choice of the mass to be forced and the type of response sought; that is, impulse response or step response. As indicated previously, for an impulse force, $f(\tau) = F_o\delta(\tau)$ and its Laplace transform is $F(s) = F_o$. For a step force, $f(\tau) = F_o u(\tau)$, where $u(\tau)$ is the unit step function; the Laplace transform is $F(s) = F_o/s$.

(a) Using the Control toolbox

Using Eqs. (9.46) and (9.47) and the definitions employed in Eqs. (9.19) and (9.20), we construct the following program:

```
m1 = 100;   m2 = 10;   k1 = 10000;   k2 = 1960;
c1 = 400;   c2 = 56;
mr = m2/m1;   wn1 = sqrt(k1/m1);
wn2 = sqrt(k2/m2);   wr = wn2/wn1;
z1 = c1/(2*m1*wn1);   z2 = c2/(2*m2*wn2);
D = [1, 2*(z1+z2*wr*(1+mr)), (1+wr^2*(1+mr)+4*z1*z2*wr), 2*wr*(z2+z1*wr), wr^2];
M = 2; % =1: Impulse applied to m_1; =2: Impulse applied to m_2
R = 'I'; % ='I': Obtain impulse response; ='S': Obtain step response
Tend = 30;
tt = linspace(0, Tend, 300);
```

```
Yval = [0, 0];
if strcmp(R, 'S') == 1
   Yval = [1, 1];
end
if M==1
   E = [1, 2*z2*wr, wr^2];
   C = [2*z2*wr, wr^2];
   if strcmp (R, 'I') == 1
      y1= impulse(tf(E, D), tt);
      y2 = impulse(tf(C, D), tt);
   else
      y1= step(tf(E, D), tt);
      y2 = step(tf(C, D), tt);
   end
else
   B = [2*z2*wr, wr^2];
   A = [1, 2*(z1+z2*mr*wr), 1+mr*wr^2]/mr;
   if strcmp (R, 'I') == 1
      y1 = impulse(tf(B, D), tt);
      y2 = impulse(tf(A, D), tt);
   else
      y1 = step(tf(B, D), tt);
      y2 = step(tf(A, D), tt);
   end
end
plot(tt, y1, 'k--', tt, y2, 'k-', [0 Tend], Yval, 'k:')
xlabel('\tau')
ylabel('Amplitude')
legend('x_1(\tau)/(F_o/k_1)', 'x_2(\tau)/(F_o/k_1)')
```

(b) Using ode45

Using Eqs. (9.53) and (9.54) with $X_{oj} = \hat{V}_{oj} = 0, j = 1, 2$, we develop the following program:

```
function Example9_14b
m1 = 100;   m2 = 10;   k1 = 10000;   k2 = 1960;
c1 = 400;   c2 = 56;
mr = m2/m1;   wn1 = sqrt(k1/m1);
wn2 = sqrt(k2/m2);   wr = wn2/wn1;
z1 = c1/(2*m1*wn1);   z2 = c2/(2*m2*wn2);
D = [1, 2*(z1+z2*wr*(1+mr)), (1+wr^2*(1+mr)+4*z1*z2*wr), . . .
      2*wr*(z2+z1*wr), wr^2];
M = 2; % =1: Impulse applied to m_1; =2: Impulse applied to m_2
R = 'I'; % ='I': Obtain impulse response; ='S': Obtain step response
Tend = 30;
tt = linspace(0, Tend, 300);
Yval = [0, 0];
if strcmp(R, 'S') == 1
   Yval = [1, 1];
end
[t, y] = ode45(@StateSpaceOde45, tt, [0 0 0 0]', [], z1, z2, mr, wr, M, R);
```

```
plot(t, y(:,1), 'k--', t, y(:,3), 'k-', [0 Tend], Yval, 'k:')
xlabel('\tau')
ylabel('Amplitude')
legend('x_1(\tau)/(F_o/k_1)', 'x_2(\tau)/(F_o/k_1)')

function dd = StateSpaceOde45(t, y, z1, z2, mr, wr, M, R)
A = -2*(z1+z2*mr*wr)*y(2)-(1+mr*wr^2)*y(1)+2*z2*mr*wr*y(4)+mr*wr^2*y(3);
B = -2*z2*wr*y(4)-wr^2*y(3)+2*z2*wr*y(2)+wr^2*y(1);
if strcmp(R, 'I') == 1
   h = (1/0.01).*(t <= 0.01);
else
   h = 1;
end
if M == 1
   A = A+h;
else
   B = B+h/mr;
end
dd = [y(2); A; y(4); B];
```

(c) Using the Symbolic toolbox and `ilaplace`

Using Eqs. (9.46) and (9.47), we construct the following program:

```
m1 = 100;    m2 = 10;    k1 = 10000;    k2 = 1960;
c1 = 400;    c2 = 56;
mr = m2/m1;    wn1 = sqrt(k1/m1);
wn2 = sqrt(k2/m2);    wr = wn2/wn1;
z1 = c1/(2*m1*wn1);    z2 = c2/(2*m2*wn2);
D = [1, 2*(z1+z2*wr*(1+mr)), (1+wr^2*(1+mr)+4*z1*z2*wr), . . .
       2*wr*(z2+z1*wr), wr^2];
M = 2; % =1: Impulse applied to m_1; =2: Impulse applied to m_2
R = 'I'; % ='I': Obtain impulse response; ='S': Obtain step response
Tend = 30;
tt = linspace(0, Tend, 300);
Yval = [0, 0];
if strcmp(R, 'S') == 1
   Yval = [1, 1];
end
syms s t
A = s^2+2*(z1+z2*mr*wr)*s+1+mr*wr^2;
B = 2*z2*mr*wr*s+mr*wr^2;
C = B/mr;
E = s^2+2*z2*wr*s+wr^2;
D2 = s^4+2*(z1+z2*wr*(1+mr))*s^3+(1+wr^2*(1+mr)+4*z1*z2*wr)*s^2 . . .
       +2*wr*(z2+z1*wr)*s+wr^2;
if M==1
   K1 = 1;    K2 = 0;         % Impulse applied to m_1 only
else
   K1 = 0;    K2 = 1/mr;    % Impulse on mass m_2 only
end
if strcmp(R, 'I') == 1
   si = 1;
```

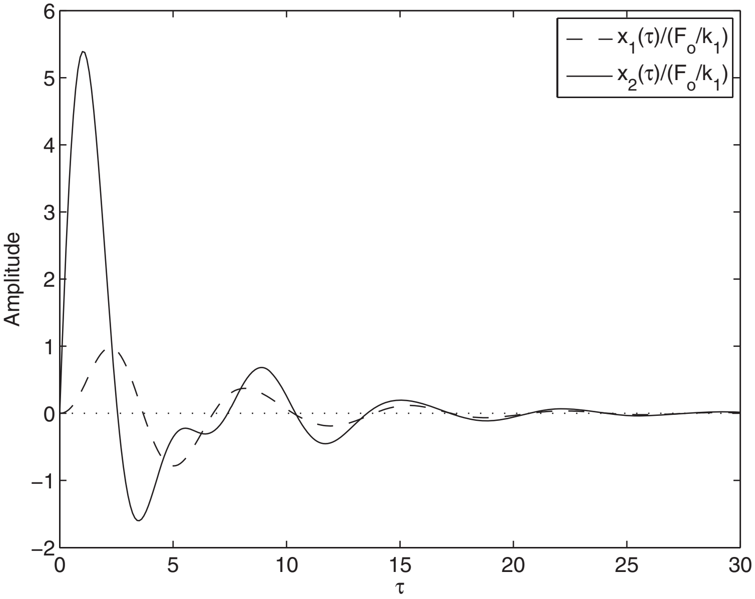


Figure 9.20 Displacement response of the two degree-of-freedom system of Figure 9.18 when mass m_2 is subjected to an impulse force.

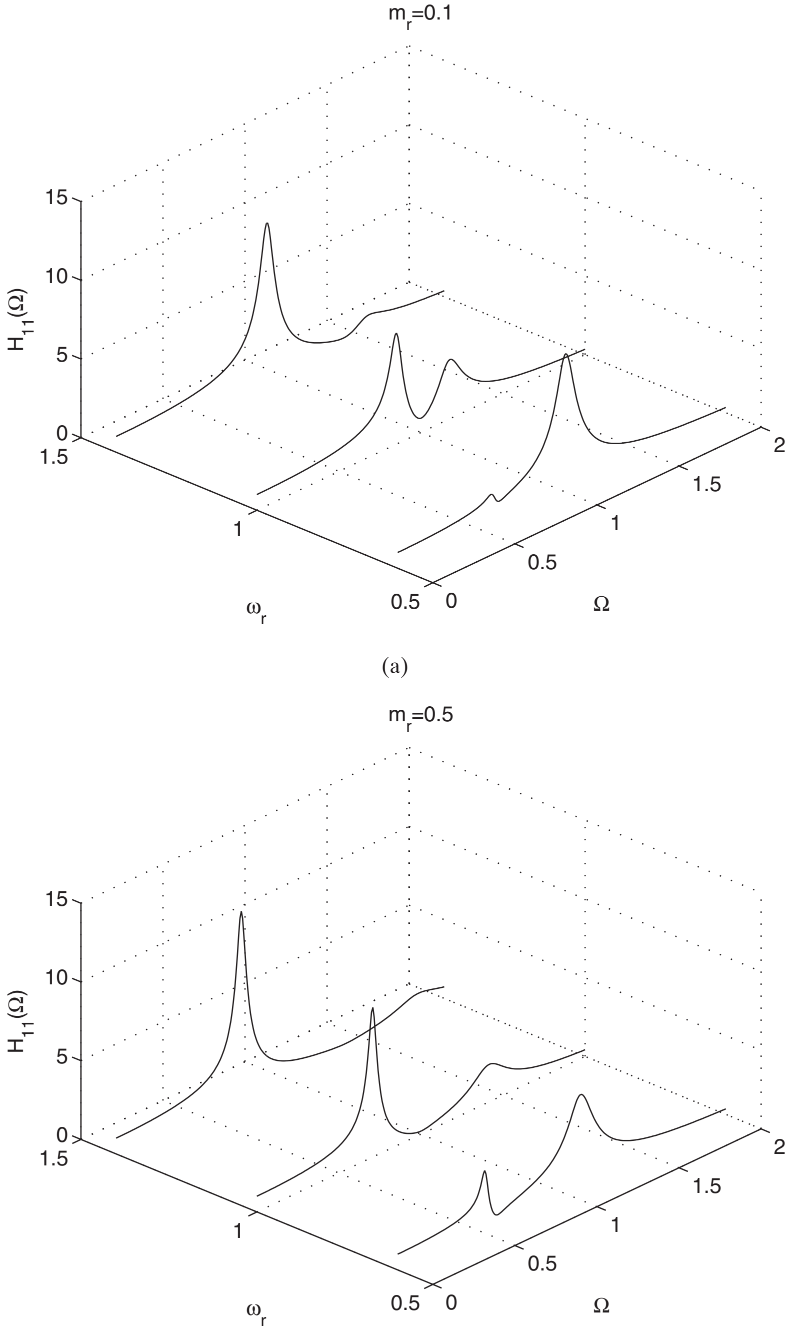
```
else
   si = 1/s;
end
X1 = vpa(si*(K1*E+K2*B)/D2, 5);
X2 = vpa(si*(K1*C+K2*A)/D2, 5);
xt1 = inline(vectorize(vpa(ilaplace(X1, s, t), 5)), 't');
xt2 = inline(vectorize(vpa(ilaplace(X2, s, t), 5)), 't');
plot(tt, xt1(tt), 'k--', tt, xt2(tt), 'k-', [0 Tend], Yval, 'k:')
xlabel('\tau')
ylabel('Amplitude')
legend('x_1(\tau)/(F_o/k_1)', 'x_2(\tau)/(F_o/k_1)')
```

After execution of these programs, the results given in Figure 9.20 are obtained for the impulse response of the system. In a similar manner, one can also obtain the step response of the system by setting $R = 'S'$.

Example 9.15 Amplitude–response function of a two-degree-of-freedom system

We shall determine the magnitude of the frequency response function $H_{ij}(\Omega)$ as given by Eqs. (9.49) and (9.50) for $i = l = 1$, $m_r = 0.1$ and 0.5, and $\omega_r = 0.6$, 1.0, and 1.4. For each of these cases, we assume that $\zeta_1 = \zeta_2 = 0.05$ and that $0 \leq \Omega \leq 2$. The script is as follows:

```
function Example9_15
z1 = 0.05;    z2 = z1;
wrr = [0.6, 1, 1.4];
```



(a)

(b)

Figure 9.21 Magnitude of the frequency-response function $H_{11}(\Omega)$ for $\omega_r = 0.6, 1.0,$ and $1.4, \zeta_1 = \zeta_2 = 0.05,$ and (a) $m_r = 0.1$ and (b) $m_r = 0.5$.

```
mr = [0.1, 0.5];   NOm = 250;
Om = linspace(0, 2, NOm);
for nm = 1:length(mr)
  figure(nm)
  for nwr = 1:length(wrr)
    H = H11(Om, wrr(nwr), mr(nm), z1, z2);
    plot3(Om, repmat(wrr(nwr), NOm, 1), H, 'k-')
    hold on
  end
  grid on
  xlabel('\Omega')
  ylabel('\omega_r')
  zlabel('H_{11}(\Omega)')
  title(['m_r=' num2str(mr(nm))])
  view([-47, 42])
end

function h = H11(Om, wr, mr, z1, z2)
D = Om.^4-1j*2*(z1+z2*wr*mr+z2*wr)*Om.^3 . . .
    -(1+mr*wr^2+wr^2+4*z1*z2*wr)*Om.^2+1j*2*(z2*wr+z1*wr^2)*Om+wr^2;
E = wr^2+1j*2*z2*wr*Om-Om.^2;
h = abs(E./D);
```

The execution of this program results in Figure 9.21.

Example 9.16 Optimal parameters for a vibration absorber

A representative graph of the function $H_{11}(\Omega)$ given by Eq. (9.50) is shown in Figure 9.22. The objective is to determine the values of the parameters for the absorber composed of m_2, k_2, and c_2 so that there is an operating region including $\Omega = 1$ where the variation of the amplitude of the primary system m_1 with respect to the frequency is minimal. The parameters that are chosen to optimize the response of m_1 are $\zeta_2 \rightarrow \zeta_{2,\text{opt}}$ and $\omega_r \rightarrow \omega_{r,\text{opt}}$. With respect to Figure 9.22, the goal is to find these absorber parameters for which the peak amplitudes $H_{11}(\Omega_A)$ and $H_{11}(\Omega_B)$ are equal and are as "small" as possible, while the minimum between these peaks $H_{11}(\Omega_C)$ is as close to $H_{11}(\Omega_A)$ and $H_{11}(\Omega_B)$. In other words, we would like to find the system parameters that minimize each of the following three maximum values simultaneously: $H_{11}(\Omega_A)$, $H_{11}(\Omega_B)$, and $1/H_{11}(\Omega_C)$. This can be stated as follows:

$$
\begin{aligned}
&\min_{\omega_r,\zeta_2}\left\{H_{11}(\Omega_A)\right\}\\
&\min_{\omega_r,\zeta_2}\left\{H_{11}(\Omega_B)\right\}\\
&\min_{\omega_r,\zeta_2}\left\{1/H_{11}(\Omega_c)\right\}\\
&\text{subject to}: \omega_r > 0\\
&\qquad\qquad\quad \zeta_2 \geq 0
\end{aligned}
$$

The peaks occur approximately at $\Omega_{A,B}$, where $\Omega_{A,B}$ are given by Eq. (9.52); that is, $\Omega_A = \Omega_1$ and $\Omega_B = \Omega_2$. The minimum between the two peaks is specified to occur

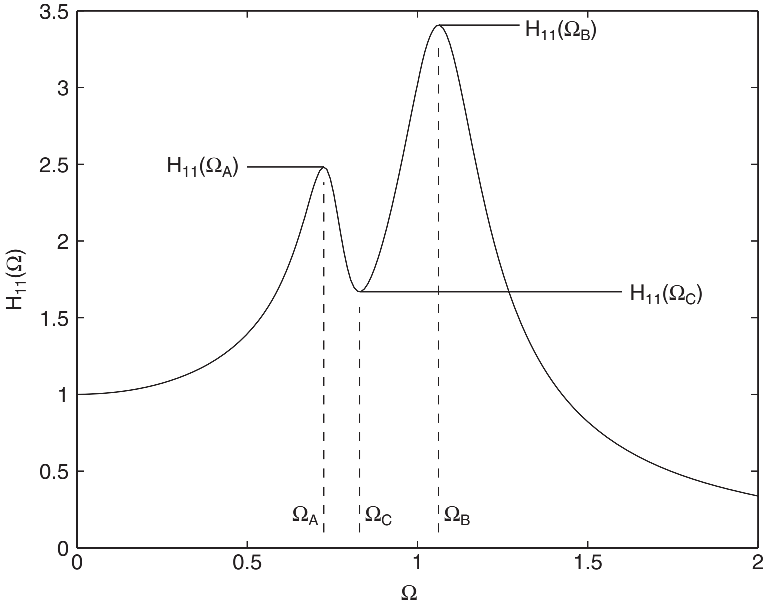


Figure 9.22 Representative amplitude response of a two-degree-of-freedom system.

at $(\Omega_A + \Omega_B)/2$. We shall use two functions from the Optimization toolbox: fminsearch and fminimax. The use of these functions is discussed in Sections 13.4.1 and 13.6, respectively.

For $\zeta_1 = 0.1$ and mass ratio $m_r = 0.1$, the following program is used to determine the optimum values $\zeta_{2,\text{opt}}$ and $\omega_{r,\text{opt}}$. The magnitude of the amplitude–response function corresponding to these optimal values is shown in Figure 9.23.

```
function Example9_16
Om = linspace(0, 2, 100);
Lbnd = [0.1, 0];    Ubnd = [2 1];
xo = [.8, 0.35];
opt = optimset('Display', 'off');
mr = .1;    z1 = .1;
[xopt, fopt] = fminimax(@objfun2doflinconstr, xo, [], [], [], [], Lbnd, ...
                        Ubnd, [], opt, mr, z1);
plot(Om, H11(Om, xopt(1), mr, z1, xopt(2)), 'k-')
ax = axis;
text(0.2*ax(2), 0.3*ax(4), ['\omega_{r,opt} = ' num2str(xopt(1), 3)])
text(0.2*ax(2), 0.2*ax(4), [' \zeta_{2,opt} = ' num2str(xopt(2), 3)])
xlabel('\Omega')
ylabel('H_{11}(\Omega)')

function [z, xx] = objfun2doflinconstr(x, mr, z1)
wr = x(1);    z2 = x(2);
opt = optimset('Display', 'off');
a1 = 1+(1+mr)*wr^2;
```

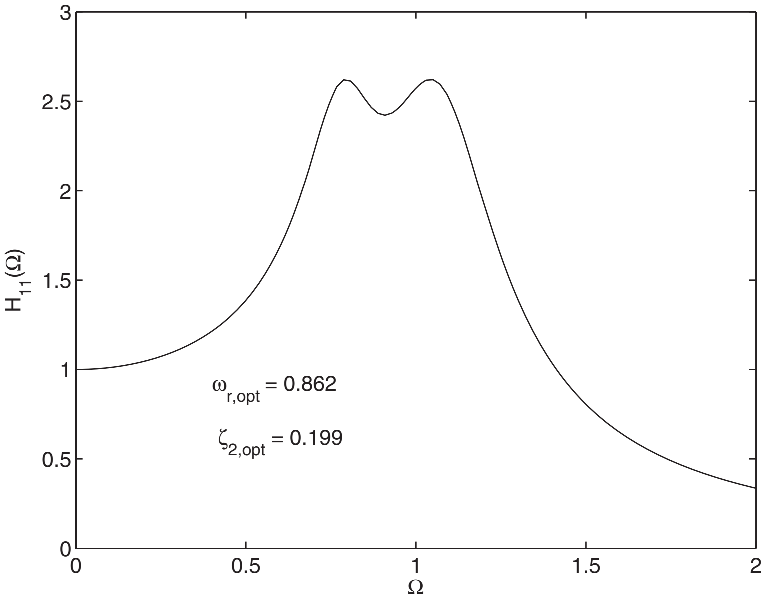


Figure 9.23 Optimum values for the parameters of a vibration absorber and the resulting amplitude response of m_1 for $\zeta_1 = 0.1$ and $m_r = 0.1$.

```
OmA = sqrt(0.5*(a1-sqrt(a1^2-4* wr^2)));
OmB = sqrt(0.5*(a1+sqrt(a1^2-4* wr^2)));
[x1, f1] = fminsearch(@Min2dof, OmA, opt, wr, mr, z1, z2);
[x2, f2] = fminsearch(@Min2dof, OmB, opt, wr, mr, z1, z2);
[x3, f3] = fminsearch(@H11, (OmA+OmB)/2, opt, wr, mr, z1, z2);
z = [1/f1, 1/f2, f3];
xx = [x1, x2, x3];

function h = H11(Om, wr, mr, z1, z2)
D2 = Om.^4-1j*2*(z1+z2*wr*mr+z2*wr)*Om.^3 . . .
      -(1+mr*wr^2+wr^2+4*z1*z2*wr)*Om.^2+1j*2*(z2*wr+z1*wr^2)*Om+wr^2;
E2 = wr^2+1j* 2*z2*wr*Om-Om.^2;
h = abs(E2./D2);

function m = Min2dof(Om, wr, mr, z1, z2)
m = 1./H11(Om, wr, mr, z1, z2);
```

Example 9.17 Half sine wave base excitation of a two-degree-of-freedom system

We shall consider the case where the base to which spring k_1 and damper c_1 are fixed now moves a prescribed amount $x_3(t)$. It is assumed that the initial conditions are zero and that $f_2(t) = 0$. The externally applied force $f_1(t)$ can be replaced by the reaction force of spring k_1 and damper c_1 to a displacement $x_3(t)$; thus,

$$f_1(t) = c_1 \frac{dx_3}{dt} + k_1 x_3$$

or in terms of the nondimensional quantities

$$\frac{f_1(\tau)}{k_1} = 2\zeta_1 \frac{dx_3}{d\tau} + x_3$$

Then, Eqs. (9.44) become

$$\frac{d^2x_1}{d\tau^2} + (2\zeta_1 + 2\zeta_2 m_r \omega_r)\frac{dx_1}{d\tau} + (1 + m_r\omega_r^2)x_1 - 2\zeta_2 m_r \omega_r \frac{dx_2}{d\tau} - m_r\omega_r^2 x_2 = 2\zeta_1 \frac{dx_3}{d\tau} + x_3$$

$$\frac{d^2x_2}{d\tau^2} + 2\zeta_2\omega_r \frac{dx_2}{d\tau} + \omega_r^2 x_2 - 2\zeta_2\omega_r \frac{dx_1}{d\tau} - \omega_r^2 x_1 = 0$$

and Eqs. (9.53) and (9.54), respectively, become

$$\frac{dy_1}{d\tau} = y_2$$

$$\frac{dy_2}{d\tau} = -(2\zeta_1 + 2\zeta_2 m_r\omega_r)y_2 - (1 + m_r\omega_r^2)y_1 + 2\zeta_2 m_r\omega_r y_4 + m_r\omega_r^2 y_3 + 2\zeta_1 \frac{dx_3}{d\tau} + x_3$$

$$\frac{dy_3}{d\tau} = y_4$$

$$\frac{dy_4}{d\tau} = -2\zeta_2\omega_r y_4 - \omega_r^2 y_3 + 2\zeta_2\omega_r y_2 + \omega_r^2 y_1$$

and

$$y_1(0) = 0 \qquad y_2(0) = 0$$
$$y_3(0) = 0 \qquad y_4(0) = 0$$

We shall assume that the base is subjected to a half sine wave of frequency ω_o and duration $t_o = \pi/\omega_o$. Then, in terms of the nondimensional quantities

$$x_3(\tau) = X_o \sin(\Omega_o\tau)[u(\tau) - u(\tau - \tau_o)]$$

and

$$\frac{dx_3}{d\tau} = X_o\Omega_o \cos(\Omega_o\tau)[u(\tau) - u(\tau - \tau_o)]$$

where $\tau_o = t_o\omega_{n1} = \pi/\Omega_o$, $\Omega_o = \omega_o/\omega_{n1}$, and $u(\tau)$ is the unit step function.

If we assume that $m_r = 0.1$, $\omega_r = 0.2$, $\zeta_1 = \zeta_2 = 0.1$, and $\Omega_o = 0.05$, then the response of the system is determined from the following program:

```
function Example9_17
mr = 0.1;   wr = 0.2;   z1 = 0.1;   z2 = 0.1;   Omo = 0.05;
tt = linspace(0, 150, 400);
[t, y] = ode45(@MovingBase, tt, [0 0 0 0]', [], z1, z2, mr, wr, Omo);
indx = find(tt<=pi/Omo);
hh = sin(Omo*tt(indx));
subplot(2,1,1)
```

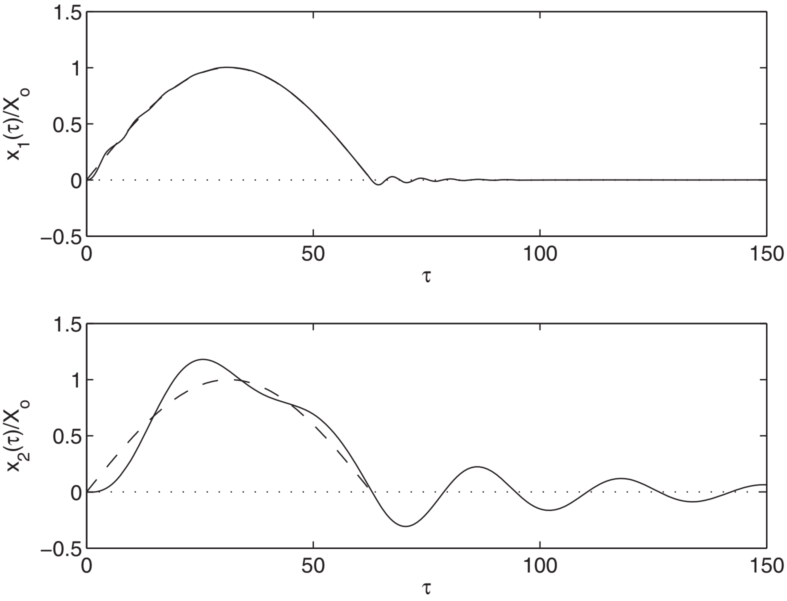


Figure 9.24 Response of a two-degree-of-freedom system whose base is subjected to a half sine wave for $m_r = 0.1, \omega_r = 0.2, \zeta_1 = \zeta_2 = 0.1$, and $\Omega_o = 0.05$.

```
plot(t, y(:,1), 'k-', tt(indx), hh, 'k--', [0, 150], [0, 0],'k:')
xlabel('\tau')
ylabel('x_1(\tau)/X_o')
subplot(2,1,2)
plot(t, y(:, 3), 'k-', tt(indx), hh, 'k--', [0, 150], [0, 0],'k:')
xlabel('\tau')
ylabel('x_2(\tau)/X_o')

function dd = MovingBase(t, y, z1, z2, mr, wr, Omo)
A = -2*(z1+z2*mr*wr)*y(2)-(1+mr*wr^2)*y(1)+2*z2*mr*wr*y(4)+mr*wr^2*y(3);
B = -2*z2*wr*y(4)-wr^2*y(3)+2*z2*wr*y(2)+wr^2*y(1);
h = (2*z1*Omo*cos(Omo*t)+sin(Omo*t)).*(t<=pi/Omo);
dd = [y(2); A+h; y(4); B];
```

Executing this program produces the results in Figure 9.24.

9.3.2 Natural Frequencies and Mode Shapes

Free oscillations of vibratory undamped vibratory systems that are modeled by discrete spring–mass elements have invariant characteristics called natural frequencies and mode shapes. These characteristics can be determined from the algebraic equations

$$\left[[K] - \omega^2[M]\right]\{X\} = \{0\} \tag{9.55}$$

where $[K]$ and $[M]$ are the $(n \times n)$ stiffness and mass matrices, respectively, and $\{X\}$ is a column vector of the displacements of the masses. Equation (9.55) constitutes an eigenvalue problem, where the values ω_l, $l = 1, 2, \ldots, n$, are the eigenvalues and at each eigenvalue there is a corresponding eigenvector $\{X\}_l$. The quantity ω_l is the natural frequency of the system and $\{X\}_l$ is the corresponding mode shape; both are determined using eig.

We shall illustrate the use of Eq. (9.55) with two examples.

Example 9.18 Natural frequencies and mode shapes of a three-degree-of-freedom system

Consider the system shown in Figure 9.25. Let the displacements x_1, x_2, and x_3 be measured from the static-equilibrium position of the system. The governing system of equations for harmonic motions $x_l = X_l e^{j\omega t}$ is given by[21]

$$\left[\begin{bmatrix} k_1 & -k_1 & 0 \\ -k_1 & (k_1 + k_2) & -k_2 \\ 0 & -k_2 & k_2 \end{bmatrix} - \omega^2 \begin{bmatrix} m_1 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & m_3 \end{bmatrix}\right] \begin{Bmatrix} X_1 \\ X_2 \\ X_3 \end{Bmatrix} = 0$$

We assume that $k_1 = 100$ N/m, $k_2 = 50$ N/m, and $m_1 = m_2 = m_3 = 100$ kg. The script that determines the eigenvalues and associated eigenvectors is

```
K = [100, -100, 0; -100, 150, -50; 0, -50, 50];
M = diag([100, 100, 100]);
[VibrationModes, Eigenvalues] = eig(K, M)
```

Execution of the script displays to the command window

```
VibrationModes =
  -0.0577    -0.0577    -0.0577
  -0.0577    -0.0211     0.0789
  -0.0577     0.0789    -0.0211
Eigenvalues =
  -0.0000          0          0
        0     0.6340          0
        0          0     2.3660
```

When the system shown in Figure 9.25 is examined, it is found that since the masses at each end are not restrained, a rigid-body mode in which all masses move in the same

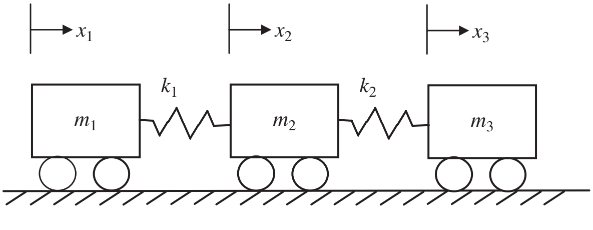


Figure 9.25 System with three degrees of freedom.

[21] Balachandran and Magrab, *Vibrations*, Section 7.2.2.

direction by the same amount is possible. This is reflected in the corresponding vibration mode depicted by the first column of the matrix of vibration modes. The springs are neither stretched nor compressed in this case. This motion is associated with the zero eigenvalue.

When a square matrix has a zero eigenvalue, the determinant of the matrix is zero. In order to ascertain whether or not a matrix has zero eigenvalues, the rank of a matrix can be determined. The rank of a matrix, which is the order of the largest square matrix for which the determinant is nonzero, can be determined from

```
rank(K)
```

where K is a matrix. The program for determining whether the stiffness matrix in Eq. (9.55) has a zero eigenvalue is

```
K = [100, -100, 0; -100, 150, -50; 0, -50, 50];
rnk = rank(K);
[m, n] = size(K);
disp(['Number of zero eigenvaules is ' int2str(m-rnk)])
```

Execution of the script produces the following output:

```
Number of zero eigenvaules is 1
```

Here, the rank of the stiffness matrix is two, indicating that one can form a (2×2) matrix with a nonzero determinant from the (3×3) stiffness matrix.

Example 9.19 Natural frequencies and mode shapes of a four-degree-of-freedom system

Consider the system shown in Figure 9.26. Let the displacements x_1, x_2, and x_3 and the rotation θ about the center of mass be measured from the static-equilibrium position of the system. The governing system of equations for harmonic motions $x_l = X_l e^{j\omega t}$ and $\theta = \Theta e^{j\omega t}$ is given by[22]

$$\left[\begin{bmatrix} 2k_1L & 0 & -k_1 & k_1 \\ 0 & 2k_1 & -k_1 & -k_1 \\ -k_1L & -k_1 & k_1 + k_2 & 0 \\ k_1L & -k_1 & 0 & k_1 + K_2 \end{bmatrix} - \omega^2 \begin{bmatrix} J_G/L & 0 & 0 & 0 \\ 0 & m_1 & 0 & 0 \\ 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & m_2 \end{bmatrix}\right] \begin{Bmatrix} L\Theta \\ X_1 \\ X_2 \\ X_3 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix}$$

where we have formed the quantity $L\Theta$ so that it is dimensionally similar to X_l and represents the displacement of the attachment points of springs k_1 to mass m_1. In other words, one end rotates upward $L\Theta \pm X_1$ and the other end rotates downward $-L\Theta \pm X_1$, where X_1 is the translation of the center of mass m_1 and the sign of X_1 indicates the direction of the translation.

We assume that $m_1 = 800$ kg, $m_2 = 25$ kg, $k_1 = 60$ kN/m, $k_2 = 20$ kN/m, $L = 1.4$ m, and $J_G = 180 \text{ kg} \cdot \text{m}^2$. The following program determines the natural frequencies and plots the corresponding mode shapes. In plotting the mode shapes, we place the masses

[22] *Ibid.*, solution to Exercise 7.45.

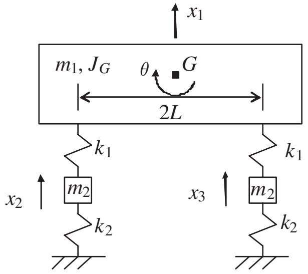


Figure 9.26 System with four-degrees-of-freedom.

on a 1 × 1.8 unit grid. Each mode shape is normalized to the maximum absolute value of the displacement. Since the eigenvalues from eig can be in any order, they are sorted and the corresponding mode matrix is rearranged to reflect their new order.

```
k1 = 60000;    k2 = 20000;    L = 1.4;
M = diag([180/L 800 25 25]);
K = [2*k1*L 0 -k1 k1
       0 2*k1 -k1 -k1
       -k1*L -k1 k1+k2 0
       k1*L -k1 0 k1+k2];
[mode E] = eig(K, M);
[w indx] = sort(diag(sqrt(E)));
modes = mode(:, indx);
x = [2 1 3 1 3];
y = 1.8*[2 1 1 2 2];
for n = 1:4
   modes(:, n) = modes(:, n)/max(abs(modes(:, n)));
   subplot(2, 2, n)
   plot(x(1), y(1), 'ks', x(2), y(2), 'ks', x(3), y(3), 'ks')
   hold on
   plot(x(4), y(4), 'ko', x(5), y(5), 'ko', [x(4), x(5)], [y(4), y(5)],'k--')
   axis([0 4 -0.2 5])
   plot(x(1), y(1)+modes(2,n), 'ks', x(2), y(2)+modes(3,n), 'ks', x(3), . . .
          y(3)+modes(4,n), 'ks', 'MarkerFaceColor', 'k')
   plot(x(4), y(4)+modes(2,n)+modes(1,n), 'ko', x(5), y(5)+modes(2,n)- . . .
          modes(1,n), 'ko', 'MarkerFaceColor', 'k')
   plot([x(4), x(5)], [y(4)+modes(2,n)+modes(1,n), y(5)+modes(2,n)-modes(1,n)],'k-', . . .
          'MarkerFaceColor', 'k')
   title(['\omega_{' int2str(n) '} = ' num2str(w(n), 4) ' rad/s'])
   axis off
end
```

The execution of this program results in Figure 9.27.

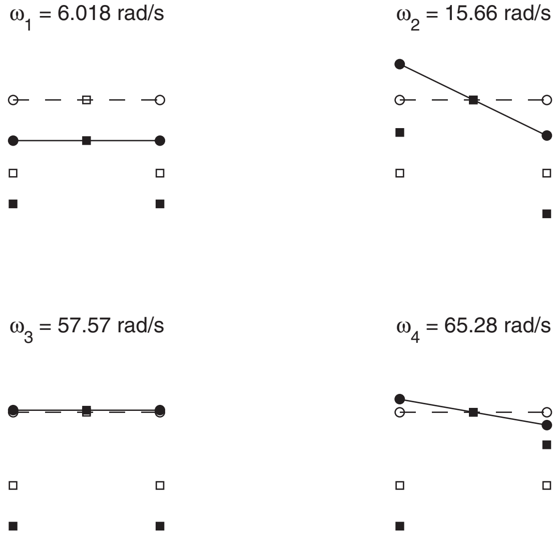


Figure 9.27 Natural frequencies and mode shapes of the system of Figure 9.26. The open symbols correspond to the static equilibrium positions of the masses and the closed symbols correspond to the displaced positions during oscillations in a particular mode. The circles indicate the attachment points of the springs k_1 to mass m_1.

9.4 FREE AND FORCED VIBRATIONS OF EULER–BERNOULLI AND TIMOSHENKO BEAMS

9.4.1 Natural Frequencies and Mode Shapes of Euler–Bernoulli and Timoshenko Beams

In this section, we shall show how to use bvp4c to determine the natural frequencies and mode shapes for Euler–Bernoulli and Timoshenko beams with uniform cross sections for a wide range of boundary conditions and for three independent and arbitrarily positioned in-span attachments: translation spring, torsion spring, and mass. As will be shown subsequently, we need only the equations for the Timoshenko beam; the results for the Euler–Bernoulli beam can be obtained by simply setting a parameter to an appropriate value. The analytical solutions to these classes of problems can be found in the literature.[23]

Consider a Timoshenko beam of length L shown in Figure 9.28 that has attached at different in-span locations a translational spring with stiffness k at $x = L_k$, a torsion spring with stiffness k_t at $x = L_t$, and a mass m having a rotational inertia j_m at $x = L_m$. We shall consider the general boundary conditions shown in Figure 9.28. Each end of the beam is restrained by a translation spring and a torsion

[23] For Euler beams see Balachandran and Magrab, *Vibrations*, Chapter 9, and for Timoshenko beams see E. B. Magrab, "Natural Frequencies and Mode Shapes of Timoshenko Beams with Attachments," *Journal of Vibration and Control*, **13**, No. 7, 2007, pp. 905–934.

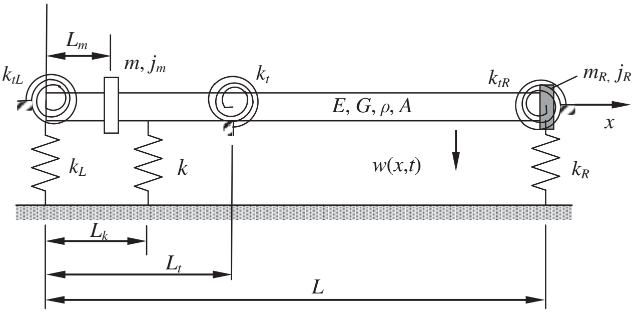


Figure 9.28 Notation and locations of beam attachments.

spring. At the right end of the beam there is an attached mass m_R with a rotational inertia j_R. The motions of a Timoshenko beam are described by a transverse displacement $w(x, t)$ and a rotation of the cross section $\psi(x, t)$ due to bending. The governing equations of motion of an undamped and unforced Timoshenko beams that are undergoing harmonic oscillations at frequency ω of the form $w(x, t) = LW(x)e^{j\omega t}$ and $\psi(x, t) = \Psi(x)e^{j\omega t}$ are given by[24]

$$\kappa AG\left(L\frac{d^2W}{dx^2} - \frac{d\Psi}{dx} \right) + \rho AL\omega^2 W + \omega^2 mW\delta(x - L_m) - kW\delta(x - L_k) = 0 \quad (9.56)$$

$$EI\frac{d^2\Psi}{dx^2} + \kappa AG\left(L\frac{dW}{dx} - \Psi \right) + \rho I\omega^2\Psi + \frac{j_m}{L}\omega^2\Psi\delta(x - L_m) - \frac{k_t}{L}\Psi\delta(x - L_t) = 0$$

The boundary conditions at $x = 0$ are[25]

$$k_L LW(0) = \kappa AG\left(L\frac{dW(0)}{dx} - \Psi \right) \quad (9.57a)$$

$$k_{tL}\Psi(0) = EI\frac{d\Psi(0)}{dx}$$

and those at $x = L$ are

$$-k_R LW(L) = \kappa AG\left(L\frac{dW(L)}{dx} - \Psi \right) - m_R\omega^2 LW(L) \quad (9.57b)$$

$$-k_{tR}\Psi(L) = EI\frac{d\Psi(L)}{dx} - j_R\omega^2\Psi(L)$$

where the various parameters appearing in these equations are given in Figure 9.28 and in Table 9.3. It is noted that the shear force is proportional to $dW/dx - \Psi$ and the bending moment is proportional to $d\Psi/dx$.

[24] Magrab, *Journal of Vibration and Control*, 2007.

[25] Magrab, *Vibrations of Elastic Structural Members*, Chapter 5.

TABLE 9.3 Nomenclature for Timoshenko Beam Formulation: Dimensional Quantities

Quantity	Units	Description
General		
x	m	x-axis
$w(x, t) = W(\eta)Le^{j\omega t}$	m	Transverse displacement of Timoshenko beam
L	m	Length of beam
ρ	kg/m^3	Beam density
A	m^2	Beam cross-sectional area
E	N/m^2	Young's modulus
$G = E/(2(1 + \nu))$	N/m^2	Shear modulus
ν		Poisson's ratio
ω	rad/s	Radian frequency
I	m^4	Moment of inertia of beam cross section
$r_o = \sqrt{I/A}$	m	Radius of gyration of beam cross section
$t_o = \sqrt{(\rho AL^4)/(EI)}$	s	Characteristic time of beam
$m_b = \rho AL$	kg	Mass of beam
$\psi(x,t) = \Psi(\eta)e^{j\omega t}$	rad	Angle of rotation of cross section of Timoshenko beam due to bending only
κ		Shear correction factor, which is a constant relating A to an effective area over which the shear stress is constant; it is a function of the cross section shape
$\gamma_{bs} = 2(1 + \nu)/\kappa$		A constant that relates the shear correction factor and the wave propagation speed at high frequencies
Boundary attachments		
k_{tL}, k_{tR}	N · m	Boundary torsion spring constants
k_L, k_R	N/m	Boundary transverse spring constants
j_R	kg · m^2	Rotational inertia of concentrated mass attached at $x = L$
m_R	kg	Concentrated mass attached to the beam at $x = L$
In-span attachments		
m	kg	Mass attached to the beam at $x = L_m\,(0 < L_m < L)$
j_m	kg · m^2	Rotational inertia of mass attached at $x = L_m\,(0 < L_m < L)$
L_α	m	Locations of various in-span attachments shown in Figure 9.28
k	N/m	Spring constant of transverse spring located at $x = L_k\,(0 < L_k < L)$
k_t	N m/rad	Spring constant of torsion spring attached at $x = L_t\,(0 < L_t < L)$

Introducing the nondimensional parameters given in Table 9.4, Eq. (9.56) becomes

$$\frac{d^2W}{d\eta^2} - \frac{d\Psi}{d\eta} + \gamma_{bs}R_o^2\left(\Omega^4 + \Omega^4 M\delta(\eta - \eta_m) - K\delta(\eta - \eta_k)\right)W = 0 \tag{9.58}$$

$$\gamma_{bs}R_o^2\frac{d^2\Psi}{d\eta^2} + \frac{dW}{dx} - \Psi + \gamma_{bs}R_o^2\left(R_o^2\Omega^4 + J\Omega^4\delta(\eta - \eta_m) - K_t\delta(\eta - \eta_t)\right)\Psi = 0$$

TABLE 9.4 Nomenclature for Timoshenko Beam Formulation: Nondimensional Quantities

General	Boundary attachments		In-span attachments	
$\eta = x/L$	$M_R = \dfrac{m_R}{m_b}$	$K_R = \dfrac{k_R L^3}{EI}$	$M = \dfrac{m}{m_b}$	$K = \dfrac{kL^3}{EI}$
$\eta_\alpha = L_\alpha/L$				
$\Omega^4 = \omega^2 t_o^2$	$J_R = \dfrac{j_R}{m_b L^2}$	$K_{tL} = \dfrac{k_{tL} L}{EI}$	$J = \dfrac{j_m}{m_b L^2}$	$K_t = \dfrac{k_t L}{EI}$
$R_o = r_o/L$				
	$K_L = \dfrac{k_L L^3}{EI}$	$K_{tR} = \dfrac{k_{tR} L}{EI}$		

The boundary conditions given by Eq. (9.57a) become

$$\frac{dW(0)}{dx} - \Psi(0) - \gamma_{bs} R_o^2 K_L W(0) = 0 \qquad (9.59a)$$

$$\frac{d\Psi(0)}{d\eta} - K_{tL}\Psi(0) = 0$$

and those given by Eq. (9.57b) become

$$\frac{dW(1)}{d\eta} - \Psi(1) + \left(K_R - M_R\Omega^4\right)W(1) = 0 \qquad (9.59b)$$

$$\frac{d\Psi(1)}{d\eta} + \left(K_{tR} - J_R\Omega^4\right)\Psi(1) = 0$$

Equation (9.58) can be expressed as a system of four first-order equations by using the substitutions

$$W(\eta) = y_1(\eta) \qquad \Psi(\eta) = y_3(\eta) \qquad (9.60)$$

$$\frac{dW(\eta)}{d\eta} = y_2(\eta) \qquad \frac{d\Psi(\eta)}{d\eta} = y_4(\eta)$$

Then, Eq. (9.60) is manipulated to obtain

$$\frac{dy_1(\eta)}{d\eta} = y_2(\eta) \qquad \frac{dy_3(\eta)}{d\eta} = y_4(\eta) \qquad (9.61a)$$

and

$$\frac{d^2W(\eta)}{d\eta^2} = \frac{dy_2(\eta)}{d\eta} \qquad \frac{d^2\Psi(\eta)}{d\eta^2} = \frac{dy_4(\eta)}{d\eta} \qquad (9.61b)$$

Using Eqs. (9.60) and (9.61) in Eq. (9.58), we obtain

$$\frac{dy_2(\eta)}{d\eta} = y_4(\eta) - \gamma_{bs} R_o^2\left(\Omega^4 + \Omega^4 M\delta(\eta - \eta_m) - K\delta(\eta - \eta_k)\right)y_1(\eta) \tag{9.62}$$

$$\frac{dy_4(\eta)}{d\eta} = \left(-y_2(\eta) + y_3(\eta)\right)\Big/\left(\gamma_{bs} R_o^2\right) - \left(R_o^2\Omega^4 + J\Omega^4\delta(\eta - \eta_m) - K_t\delta(\eta - \eta_t)\right)y_3(\eta)$$

The boundary conditions at $\eta = 0$ become

$$\begin{aligned} y_2(0) - y_3(0) - \gamma_{bs}R_o^2K_Ly_1(0) &= 0 \\ y_4(0) - K_{tL}y_3(0) &= 0 \end{aligned} \tag{9.63a}$$

and those at $\eta = 1$ become

$$\begin{aligned} y_2(1) - y_3(1) + \gamma_{bs}R_o^2\left(K_R - M_R\Omega^4\right)y_1(1) &= 0 \\ y_4(1) + \left(K_{tR} - J_R\Omega^4\right)y_3(1) &= 0 \end{aligned} \tag{9.63b}$$

TABLE 9.5 Boundary Conditions for a Timoshenko Beam in a Form Suitable for `bvp4c`

Boundary condition	$\eta = 0$[a]	$\eta = 1$
Hinged	$y_1(0) = 0$ $y_4(0) = 0$ $\boldsymbol{y_2(0)} - \mathbf{0.05} = 0$	$y_1(1) = 0$ $y_4(1) = 0$
Clamped	$y_1(0) = 0$ $y_3(0) = 0$ $\boldsymbol{y_4(0)} - \mathbf{0.05} = 0$	$y_1(1) = 0$ $y_3(1) = 0$
Free	$y_2(0) - y_3(0) = 0$ $y_4(0) = 0$ $\boldsymbol{y_1(0)} - \mathbf{0.02} = 0$	$y_2(1) - y_3(1) = 0$ $y_4(1) = 0$
Hinged with torsion spring	Not considered	$y_1(1) = 0$ $y_4(1) + K_{tR}y_3(1) = 0$
Free with translation spring	Not considered	$y_2(1) - y_3(1) + \gamma_{bs}R_o^2K_Ry_1(1) = 0$ $y_4(1) = 0$
Free with torsion spring	Not considered	$y_2(1) - y_3(1) = 0$ $y_4(1) + K_{tR}y_3(1) = 0$
Free with mass	Not considered	$y_2(1) - y_3(1) - \gamma_{bs}R_o^2M_R\Omega^4y_1(1) = 0$ $y_4(1) - J_R\Omega^4y_3(1) = 0$
Free with mass and translation and torsion springs	Not considered	$y_2(1) - y_3(1) + \gamma_{bs}R_o^2(K_R - M_R\Omega^4)y_1(1) = 0$ $y_4(1) + (K_{tR} - J_R\Omega^4)y_3(1) = 0$

[a] Boundary conditions in bold are the “fifth” boundary condition.

Then, Eqs. (9.61a) and (9.62) form a set of four first-order equations that can be used by bvp4c to find a numerical solution. The boundary conditions given by Eq. (9.63) are also in a form that can be used by bvp4c.

It is pointed out that Eq. (9.63) represents a wide rage of boundary conditions as determined by the limits of K_L, K_R, K_{tL}, and K_{tR}. For example, if we were to divide the first of Eq. (9.63a) by K_L and the second of Eq. (9.63a) by K_{tL} and let $K_L \rightarrow \infty$ and $K_{tL} \rightarrow \infty$, we have the case of a clamped end. Conversely, when $K_L = 0$ and $K_{tL} = 0$, we have the case of a free end. Various special cases are summarized in Table 9.5.

We shall now illustrate these results with an example. Two other examples regarding the determination of the natural frequency coefficient of Euler–Bernoulli beams are given in Examples 5.20 and 7.5.

Example 9.20 Natural frequencies and mode shapes of Euler–Bernoulli and Timoshenko beams with attachments

Before we create a program that can deal with many combinations of these boundary conditions and in-span attachments, we have to consider some of the implementation aspects of bvp4c. As discussed in Example 5.20, to solve the system of homogeneous differential equations and homogeneous boundary conditions, we have to specify a "fifth" boundary condition, which is not homogeneous. We shall always choose it to be one from the $\eta = 0$ end and restrict our general results to the three cases given in Table 9.5. Regarding the in-span attachments, we have to use the technique that was introduced in Example 5.18. Here, we have to make sure that our initial guess for the mesh points includes the locations where each attachment has been placed. For programming simplicity, we shall limit the placement of an in-span attachment to only one: either a translation spring, a torsion spring, or a mass. In order for bvp4c to successfully determine the natural frequency and mode shape, it has to be used twice in succession. First it is used with very loose tolerances and with bvpinit. Then it is used again, this time with closer tolerances and employing the output of the just-executed bvp4c in place of the output of bvpinit.

To be able to create a program that can transfer a large number of different values in a readable form, we form the following three vectors:

$$a = [M, J, K, K_t]$$

$$b = [M_R, J_R, K_R, K_{tR}]$$

$$e = [\eta_m, \eta_k, \eta_t]$$

The values selected are recorded to the command window using **DisplayParameters**. In addition, we consider three basic boundary conditions at each end: clamped, hinged, and free. At the right end ($n = 1$), we add the various attachments as indicated in Table 9.5, depending on which boundary condition has been chosen. The initial guesses for the solutions are selected as $y_1 = \sin(n\pi x)$, $y_2 = \cos(n\pi x)$, $y_3 = \sin(n\pi x)$, $y_4 = \cos(n\pi x)$, where n is the natural frequency number; that is, for $n = 1$, we have the lowest natural frequency, for $n = 2$, we have the second natural frequency, and so on. The node points are determined using **ZeroCrossing** from Example 9.5.

The program is implemented assuming a cantilever beam with the following parameters: $M = 0$, $J = 0$, $K = 100$ at $\eta_t = 0.5$, $K_t = 0$, $M_R = 0.3$, $J_R = 0.1$, $K_R = 100$, and $K_{tR} = 5$. For these parameters, we assume that $\gamma_{bs} = 3.12$ and that $R_o = 0.005$, which very closely approximates an Euler beam and $R_o = 0.04$, which requires the Timoshenko model. A cantilever beam is one that is clamped at one end and free at the other.

```
function Example9_20
a = [0, 0, 100, 0];           % a = [M, J, K, Kt]
b = [0.3, 0.1, 100, 5];       % b = [MR, JR, KR, KtR]
e = [0, 0.5, 0];              % e = [etaMJ, etaK, etat]
eta = linspace(0, 1, 101);
Ro = [0.005, 0.04];   gbs = 3.12;
BCL = 'clamped';      % 'hinged';  'clamped';  'free';
BCR = 'free';         % 'hinged';  'clamped';  'free';
Spac = InitialMesh(e);
Omguess = [1, 5, 8];
for m = 1:length(Ro)
  DisplayParameters(b, a, e, BCL, BCR, Ro(m), gbs)
  figure(m)
  for n = 1:length(Omguess)
    opt = bvpset('RelTol', 1e-2, 'AbsTol', 1e-3);
    solinit = bvpinit(Spac, @TimoModeGuess, Omguess(n), n);
    beamsol = bvp4c(@TimoODE, @TimoBC, solinit, opt, a, b, e, Ro(m), . . .
                    gbs, BCL, BCR);
    opt = bvpset('RelTol', 1e-5, 'AbsTol', 1e-6);
    beamsol = bvp4c(@TimoODE, @TimoBC, beamsol, opt, a, b, e, Ro(m), . . .
                    gbs, BCL, BCR);
    Omega = beamsol.parameters;
    disp(['Omega/pi = ' num2str(Omega/pi)])
    y = deval(beamsol, eta);
    ModeShape = y(1,:)/max(abs(y(1,:)));
    Nodes = ZeroCrossing(eta, ModeShape);
    subplot(3, 1, n)
    plot(eta, ModeShape, 'k-', [0 1], [0 0], 'k--');
    if length(Nodes) == 1 && Nodes == 0
      No = '0';
    else
      No = num2str([0, Nodes], 3);
    end
    text(0, -1, ['Node locations: ' No])
    title(['R_o = ' num2str(Ro(m)) ' \Omega _' int2str(n) '/\pi = '
      num2str(Omega/pi)])
    axis([0 1 -1 1])
    axis off
  end
end

function DisplayParameters(b, a, e, BCL, BCR, Ro, gbs)
disp(' ')
disp(['Timoshenko beam: Ro = ' num2str(Ro) ' gbs = ' num2str(gbs)])
disp(['Boundary conditions: Left end - ' BCL ' Right end - ' BCR])
disp(' ')
if isempty(find(b> 0, 1)) == 0
  disp('Attachments at right end - ')
  disp(['Translation spring (KR) = ' num2str(b(3)) ' Torsion spring (KtR) = '...
         num2str(b(4))])
  disp(['Mass (MR) = ' num2str(b(1)) ' Rotational inertia (JR) = '...
         num2str(b(2))])
  disp(' ')
end
if isempty(find(a> 0, 1)) == 0
  disp('In-span attachments: ')
  disp(['Translation spring (K) = ' num2str(a(3)) ' at eta = ' num2str(e(2))])
  disp(['Torsion spring (Kt) = ' num2str(a(4)) ' at eta = ' num2str(e(3))])
  disp(['Mass (M) = ' num2str(a(1)) ' and Rotational Mass (J) = ' . . .
         num2str(a(2)) ' at eta = ' num2str(e(1))])
  disp(' ')
end

function Spac = InitialMesh(e)
ep = 0.005;
indx = find(e~=0);
if isempty(indx)
  Spac = linspace(0, 1, 10);
else
  Spac = [linspace(0, e(indx)-0.01, 5), e(indx)-ep, e(indx), e(indx)+ep, . . .
          linspace(e(indx)+0.01, 1, 5)];
end

function yinit = TimoModeGuess(x, n)
yinit = [sin(n*x*pi); cos(n*x*pi); sin(n*x*pi); cos(n*x*pi)];

function dydx = TimoODE(x, y, Om, a, b, e, Ro, gbs, BCL, BCR)
% a = [M, J, K, Kt]
% e = [etaMJ, etaK, etat]
ep = 0.004;   dm = 0;   dk = 0;   dt = 0;
if a(1) > 0
  dm = ((x>=(e(1)-ep)&(x<=(e(1)+ep))))/(2*ep);
end
if a(3) > 0
  dk = ((x>=(e(2)-ep)&(x<=(e(2)+ep))))/(2*ep);
end
if dt > 0
  dk = ((x>=(e(3)-ep)&(x<=(e(3)+ep))))/(2*ep);
end
A = y(4)-(Om^4+a(1)*Om^4*dm-a(3)*dk)*y(1)*gbs*Ro^2;
B = (-y(2)+y(3))/(gbs*Ro^2)-(Ro^2*Om^4+a(2)*Om^4*dm-a(4)*dt)*y(3);
dydx = [y(2); A; y(4); B];

function bc = TimoBC(y0, y1, Om, a, b, e, Ro, gbs, BCL, BCR)
% b = [MR, JR, KR, KtR]
```

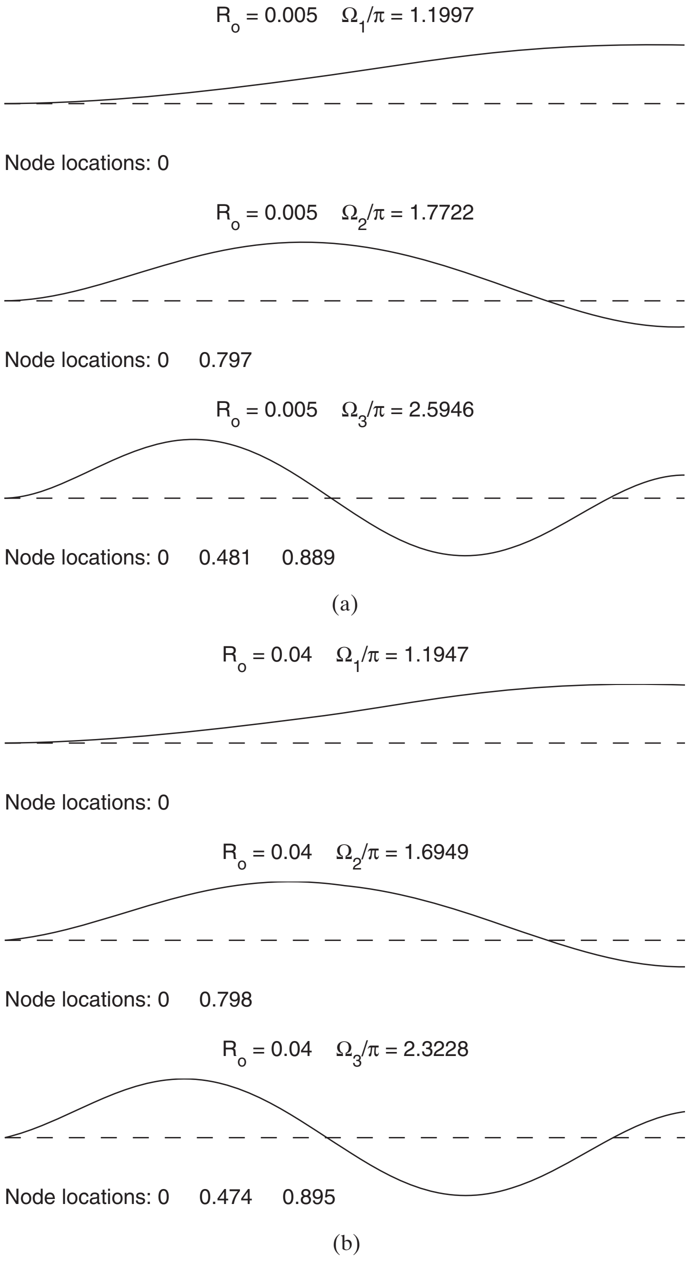


Figure 9.29 Lowest three natural frequencies and corresponding mode shapes for $M = 0$, $J = 0$, and $K = 100$ at $\eta_t = 0.5$, $K_t = 0$, $M_R = 0.3$, $J_R = 0.1$, $K_R = 100$, $K_{tR} = 5$, and $\gamma_{bs} = 3.12$: (a) $R_o = 0.005$, a beam geometry that can be approximated by the Euler beam theory.
(b) $R_o = 0.04$, a beam geometry that requires the Timoshenko beam theory.

```
switch BCL
   case 'clamped'
      bc = [y0(1); y0(4)-0.05; y0(3)];
   case 'hinged'
      bc = [y0(1); y0(2)-0.05; y0(4)];
   case 'free'
      bc = [(y0(2)-y0(3)); y0(1)-0.02; y0(4)];
end
switch BCR
   case 'clamped'
      bc = [bc; y1(1); y1(3);];
   case 'hinged'
      bc = [bc; y1(1); y1(4)+b(4)*y1(3)];
   case 'free'
      bc = [bc; y1(1)*(b(3)-b(1)*Om^4)*gbs*Ro^2+y1(2)-y1(3); y1(4)+(b(4)
             -b(2)*Om^4)*y1(3)];
end
```

Executing this program results in Figure 9.29, with the following information being displayed in the command window.

```
Timoshenko beam: Ro = 0.005 gbs = 3.12
Boundary conditions: Left end - clamped    Right end - free
Attachments at right end -
Translation spring (KR) = 100 Torsion spring (KtR) = 5
Mass (MR) = 0.3 Rotational inertia (JR) = 0.1

In-span attachments:
Translation spring (K) = 100 at eta = 0.5
Torsion spring (Kt) = 0 at eta = 0
Mass (M) = 0 and Rotational Mass (J) = 0 at eta = 0

Omega/pi = 1.1997
Omega/pi = 1.7722
Omega/pi = 2.5946

Timoshenko beam: Ro = 0.04 gbs = 3.12
Boundary conditions: Left end - clamped Right end - free

Attachments at right end -
Translation spring (KR) = 100 Torsion spring (KtR) = 5
Mass (MR) = 0.3 Rotational inertia (JR) = 0.1

In-span attachments:
Translation spring (K) = 100 at eta = 0.5
Torsion spring (Kt) = 0 at eta = 0
Mass (M) = 0 and Rotational Mass (J) = 0 at eta = 0

Omega/pi = 1.1947
Omega/pi = 1.6949
Omega/pi = 2.3228
```

9.4.2 Forced Oscillations of Euler–Bernoulli Beams

In this section, we consider the forced oscillations of an undamped Euler–Bernoulli beam of constant cross section A, area moment of inertia I, length L, Young's modulus E, and mass density ρ that is subjected to a time-dependent external force per unit length $F_o f(x, t)$. The governing equation for this system is given by[26]

$$EI\frac{\partial^4 w}{\partial x^4} + \rho A\frac{\partial^2 w}{\partial t^2} = F_o f(x, t)$$

where $w = w(x, t)$ is the transverse displacement of the beam, x is the spatial variable, and t is the time. The beam is constrained by a set of boundary conditions at each end of the beam. If we consider the boundary conditions shown in Figure 9.30, then at $x = 0$, we have

$$\begin{aligned} EIw''(0, t) &= k_{t1}w'(0, t) \\ EIw'''(0, t) &= -k_1 w(0, t) \end{aligned}$$

and at $x = L$, we have

$$\begin{aligned} EIw''(L, t) &= -k_{t2}w'(L, t) \\ EIw'''(L, t) &= k_2 w(L, t) \end{aligned}$$

where the prime denotes the derivative with respect to x.

Before proceeding, we introduce the following notations

$$\eta = x/L, \quad \Omega^4 = \frac{\omega^2 \rho A L^4}{EI} = \frac{\omega^2 L^4}{c_b^2 r^2} = \omega^2 t_o^2$$

$$c_b^2 = E/\rho, \quad r^2 = I/A, \quad t_o = \frac{L^2}{c_b r}, \quad \tau = t/t_o$$

$$G_o = \frac{F_o L^4}{EI} \quad K_j = \frac{k_j L^3}{EI}, \quad B_j = \frac{k_{tj} L}{EI} \quad j = 1, 2$$

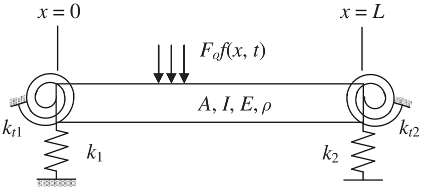


Figure 9.30 Boundary conditions and properties for an Euler–Bernoulli beam.

[26] The material in this section is based, in part, on Balachandran and Magrab, *Vibrations*, Sections 9.2.4, 9.3.3, and 9.4.

into the governing equation and boundary conditions to obtain, respectively,

$$\frac{\partial^4 w}{\partial \eta^4} + \frac{\partial^2 w}{\partial \tau^2} = G_o g(\eta, \tau)$$

and at the boundary condition at $\eta = 0$

$$\begin{aligned} \frac{d^2 w(0,\tau)}{d\eta^2} &= B_1 \frac{dw(0,\tau)}{d\eta} \\ \frac{d^3 w(0,\tau)}{d\eta^3} &= -K_1 w(0,\tau) \end{aligned} \tag{9.64}$$

and at the boundary condition at $\eta = 1$

$$\begin{aligned} \frac{d^2 w(1,\tau)}{d\eta^2} &= -B_2 \frac{dw(1,\tau)}{d\eta} \\ \frac{d^3 w(1,\tau)}{d\eta^3} &= K_2 w(1,\tau) \end{aligned} \tag{9.65}$$

It is mentioned that these boundary conditions have been chosen because they can be specialized to wide range of boundary conditions by independently varying the values of B_j and K_j, $j = 1, 2$, from zero to infinity.

If we assume that the initial conditions are zero, then the solution to the governing equation and boundary conditions is of the form

$$w(\eta, \tau) = \sum_{n=1}^{N} \frac{G_o W_n(\eta)}{N_n} \left[\frac{1}{\Omega_n^2} \int_0^{\tau} \left[\int_0^1 g(\eta, \tau - \tau') W_n(\eta) \sin(\Omega_n^2 \tau') d\eta \right] \right] d\tau' \tag{9.66}$$

where Ω_n are the natural frequency coefficients that satisfy the characteristic equation

$$D(\Omega_n) = 0$$

the function $W_n(\eta)$ is the corresponding orthogonal mode shape that satisfies the boundary conditions, and

$$N_n = \int_0^1 W_n^2(\eta) d\eta$$

Explicit expressions for $W_n(\eta)$ and $D(\Omega_n)$ for several combinations of boundary conditions are available. See reference in footnote to this section.

Example 9.21 Impulse response of an Euler–Bernoulli beam

To illustrate the evaluation and display of $w(\eta,\tau)$, we consider a cantilever beam subjected to an impulse force applied at $\tau = 0$ and at $\eta = \xi$. For this case, the applied force is given by

$$g(\eta, \tau) = \delta(\eta - \xi)\delta(\tau)$$

Substituting this expression in Eq. (9.66), we obtain

$$\frac{w(\eta, \tau)}{G_o} = \sum_{n=1}^{\infty} \frac{W_n(\xi)}{N_n \Omega_n^2} W_n(\eta) \sin(\Omega_n^2 \tau) \tag{9.67}$$

The characteristic equation and mode shape, respectively, are

$$D(\Omega_n) = \cos(\Omega_n) \cosh(\Omega_n) + 1 = 0$$

and

$$W_n(\eta) = -\frac{T(\Omega_n)}{Q(\Omega_n)} T(\Omega_n \eta) + S(\Omega_n \eta)$$

where

$$Q(x) = 0.5[\cosh(x) + \cos(x)]$$

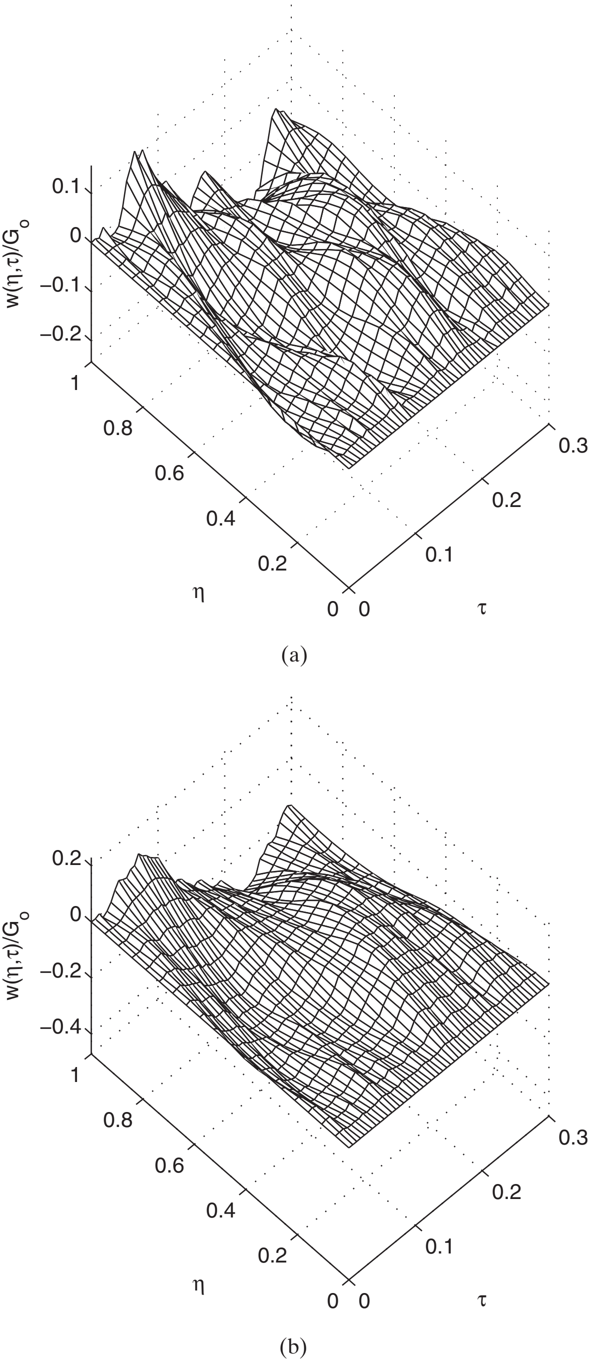
$$S(x) = 0.5[\cosh(x) - \cos(x)]$$

$$T(x) = 0.5[\sinh(x) - \sin(x)]$$

These expressions were obtained as a special case of the boundary conditions given by Eqs. (9.64) and (9.65) as follows. At $\eta = 0$, the first of Eq. (9.64) is divided by B_1 and $B_1 \to \infty$ and the second of Eq. (9.64) is divided by K_1 and $K_1 \to \infty$. The boundary conditions at $\eta = 1$ are obtained by setting $B_2 = K_2 = 0$ in Eq. (9.65).

The numerical evaluation of these expressions requires that we first obtain the lowest N values of Ω_n that satisfy the transcendental equation $D(\Omega_n) = 0$. Once the Ω_n are known, we determine N_n and $W_n(\xi)$ for a specific value of ξ. The values for these quantities are then used in Eq. (9.67), which is evaluated at specific values of η and τ. For illustration, we shall determine $w(\eta, \tau)$ for $\xi = 0.25$ and 0.4, $N = 8$, and for a range of values $0 < \eta < 1$ and $0 < \tau < 0.3$. The program that is used to perform these calculations and that produce the results shown in Figure 9.31 is as follows:

```
function Example 9_21
Nroot = 8;   w = [];   Ne= 20;   Ntau = 52;
CantRoot = @(x,w) (cosh(x).*cos(x)+1);
xi = [0.25, 0.4];   eta = linspace(0, 1, Ne);   tau = linspace(0, 0.3, Ntau);
Om = linspace(1.5, 35, 100);   Nn = zeros(Nroot, 1);
Omega = FindZeros(CantRoot, Nroot, Om, w);
for k = 1:Nroot
  Nn(k) = quadl(@W2, 0, 1, [], [], Omega(k));
end
for m = 1:2
  figure(m)
  Cn = W(xi(m), Omega)./Nn./Omega.^2;
  w = zeros(Ne,Ntau);
  for c = 1:Ntau
```



(a)

(b)

Figure 9.31 Response of an Euler–Bernoulli cantilever beam to an impulse at (a) $\xi = 0.25$ and (b) $\xi = 0.4$.

```
      for r = 1:Ne
        w(r,c) = sum(Cn.*W(eta(r), Omega).*sin(Omega.^2*tau(c)));
      end
    end
    mesh(tau, eta, -w)
    colormap([0 0 0 ])
    axis vis3d
    view([-44, 58])
    xlabel('\tau')
    ylabel('\eta')
    zlabel('w(\eta,\tau)/G_o')
    a = axis;   a(1) = 0;   a(2) = 0.3;
    axis(a)
  end

  function f = Q(x)
  f = 0.5*(cosh(x)+cos(x));

  function f = S(x)
  f = 0.5*(cosh(x)-cos(x));

  function f = T(x)
  f = 0.5*(sinh(x)-sin(x));

  function f = W(x,Om)
  f = -T(Om)./Q(Om).*T(Om.*x)+S(Om.*x);

  function f = W2(x, Om)
  f = W(x,Om).^2;
```

In this script, the function M file **FindZeros** introduced in Section 5.5.1 is used.

9.5 SUMMARY OF FUNCTIONS INTRODUCED IN CHAPTER 9

A summary of the functions introduced in Chapter 9 is presented in Table 9.6.

TABLE 9.6 MATLAB Functions Introduced in Chapter 9

MATLAB function	Description
bode	Frequency response of linear time-invariant systems
damp	Damping factors and natural frequencies of linear time-invariant systems
fminimax	Solves minimax problem
fminsearch	Finds a minimum of an unconstrained multivariable function
impulse	Impulse response of linear time-invariant system
rank	Estimates the number of linearly independent rows or columns of a full matrix
step	Step response of linear time-invariant system
tf	Specify transfer function of a linear time-invariant system
trace	Sum of the diagonal elements of a matrix

EXERCISES

Section 9.1.2

9.1 For the two-mass system described by Eq. (9.3), numerically determine the corresponding orbit in the (r, θ) plane for the initial conditions $r(0) = 2.0$, $dr(0)/d\tau = 0.0$, $\theta(0) = 0.0$, and $d\theta(0)/d\tau = 0.5$. Use `ode45` with the following parameter values: (a) time span of 20 units, step size of 20/400, relative tolerance of 10^{-3}, and absolute tolerance of 10^{-3} for each of the states; and (b) time span of 20 units, step size of 20/4,000, relative tolerance of 10^{-6}, and absolute tolerance of 10^{-6} for each of the states. Determine whether or not the angular momentum per unit mass $r^2\dot{\theta}$ is conserved in each case throughout the time span and display graphs of the following: (i) orbits and (ii) angular momentum per unit mass versus time.

9.2 If we let $r(0) = r_o$ and $d\theta(0)/d\tau = \theta_o$, then Eq. (9.3) can be reduced to the single equation:

$$\frac{1}{2}\left(\frac{dr}{d\tau}\right)^2 + \frac{r_o^2\theta_o^2}{2r^2} - \frac{4\pi^2}{r} = \frac{\theta_o^2}{2} - \frac{4\pi^2}{r_o}$$

The maximum and minimum values of the radial distance r can be determined by setting $dr/d\tau = 0$ in the above equation, which results in the following relation:

$$\left(\theta_o^2 - \frac{8\pi^2}{r_o}\right)r^2 + 8\pi^2 r - r_o^2\theta_o^2 = 0$$

Determine if a satellite orbiting the earth will crash on the earth's surface for the following initial conditions: (1) $r(0) = 2.0$, $dr(0)/d\tau = 0.0$, $\theta(0) = 0.0$, and $d\theta(0)/d\tau = 2.0$; and (2) $r(0) = 2.0$, $dr(0)/d\tau = 0.0$, $\theta(0) = 0.0$, and $d\theta(0)/d\tau = 0.2$. A crash will occur when $r = 1$.

Section 9.2.1

9.3 The ratio of the measured amplitude to the true acceleration amplitude for an accelerometer is

$$\frac{A_m}{A_t} = \frac{1}{\sqrt{(1 - \Omega^2)^2 + 4\zeta^2\Omega^2}}$$

where $\Omega = \omega/\omega_n$, ω is the acceleration frequency, ω_n is the accelerometer's natural frequency, and ζ is the accelerometer's damping factor. Obtain the following:

a. A surface plot of A_m/A_t as a function of Ω and ζ.

b. The damping factor of an accelerometer with a mass $m = 0.01$ kg and natural frequency of 150 Hz that is to measure accelerations at 6,000 rpm with an error e of ±2.0 %.[27] The percentage error is determined from

$$e = 100\left(1 - \frac{A_m}{A_t}\right)\%$$

[27] *Ibid*, Section 5.6.

Hence,

$$\zeta = \frac{1}{2\Omega}\left[\frac{1}{1-e/100} - \left(1-\Omega^2\right)^2\right]^{1/2}$$

9.4 The ratio of the measured amplitude to the true displacement amplitude of a seismometer is[28]

$$\frac{d_m}{d_t} = \frac{\Omega^2}{\sqrt{(1-\Omega^2)^2 + 4\zeta^2\Omega^2}}$$

where $\Omega = \omega/\omega_n$, ω is the acceleration frequency, ω_n is the seismometer's natural frequency, and ζ is the seismometer's damping factor. Obtain the following:

a. A surface plot of d_m/d_t as a function of Ω and ζ.

b. Determine the maximum natural frequency of the seismometer if the seismometer is to measure vibrations at 1,500 rpm with an error less than ±2.0% when $\zeta = 0.1$. In other words, since the percentage error is determined from

$$e = 100\left(1 - \frac{d_m}{d_t}\right)\%$$

Ω must be a solution to

$$\left(1 - 1/(1-e/100)\right)\Omega^4 + \left(-2+4\zeta^2\right)\Omega^2 + 1 = 0$$

Section 9.2.2

9.5 Compare the results of Example 9.4, which were obtained using `ode45`, with the results obtained from the following analytical solutions.[29] For the underdamped case, set $\zeta = 0.1$; for the critically damped case, $\zeta = 1$; and for the overdamped case, set $\zeta = 2$. In addition, let $0 \le \tau \le 20$ and $x_o = 1$ and $v_o = 1$.

Underdamped ($\zeta < 1$)

$$y(\tau) = x_o e^{-\zeta\tau}\cos\left(\sqrt{1-\zeta^2}\tau\right) + \frac{v_o + \zeta x_o}{\sqrt{1-\zeta^2}} e^{-\zeta\tau}\sin\left(\sqrt{1-\zeta^2}\tau\right)$$

Critically damped ($\zeta = 1$)

$$y(\tau) = [x_o + (v_o + x_o)\tau]e^{-\tau}$$

Overdamped ($\zeta > 1$)

$$y(\tau) = x_o e^{-\zeta\tau}\cosh\left(\tau\sqrt{\zeta^2-1}\right) + \frac{\zeta x_o + v_o}{\sqrt{\zeta^2-1}} e^{-\zeta\tau}\sinh\left(\tau\sqrt{\zeta^2-1}\right)$$

[28] *Ibid*, p. 220.

[29] *Ibid.*, Appendix D.

TABLE 9.7 Free Oscillation Data for Exercise 9.6

Time	Amplitude	Time	Amplitude	Time	Amplitude
0.000	0.801	10.00	−0.0151	20.77	0.0379
0.692	0.365	10.77	−0.00688	21.54	0.0167
1.538	−0.386	11.54	0.118	22.31	−0.0184
2.308	−0.562	12.31	0.0882	23.08	−0.0259
3.077	−0.114	13.07	−0.028	23.85	−0.0141
3.846	0.349	13.85	−0.0871	24.61	0.0149
4.615	0.338	14.15	−0.0551	25.38	0.0115
5.385	−0.301	15.85	0.0220	26.15	0.00367
6.154	−0.204	16.23	0.0687	26.92	−0.0148
6.923	0.104	17.92	0.0376	27.69	−0.0125
7.692	0.228	18.46	−0.040	28.46	0.0157
8.461	0.008	19.23	−0.0514	29.23	0.00263
9.231	−0.010	20.00	−0.00641	30.00	−0.00727

9.6 Consider the free oscillation response data given in Table 9.7. Curve fit these data by assuming that the response is of the form

$$x(t) = \frac{X_o e^{-\zeta\omega_n t}}{\sqrt{1-\zeta^2}} \sin\left(\omega_n t\sqrt{1-\zeta^2} + \varphi\right)$$

where

$$\varphi = \tan^{-1}\frac{\sqrt{1-\zeta^2}}{\zeta}$$

and determine X_o, ζ, and ω_n.

9.7 A mass m slides along a rough rod of length l, which is pivoted at the end O. The angular orientation of the rod with respect to the horizontal is given by $\theta(t)$ and the location of the mass along the rod from its pivot point is given by r. When the rod rotates in the horizontal plane with a constant angular speed $d\theta/dt = \omega$, the equation of motion of mass m is

$$\ddot{r} + 2\mu\omega\dot{r} - \omega^2 r = 0$$

where μ is the coefficient of friction between the rod and the mass. For $\mu = 0.2$, $l = 3.0\text{ m}$, $\omega = 6\text{ rad/s}$, and initial conditions $r(0) = 1.0\text{ m}$, $\theta(0) = 0.0\text{ rad/s}$, and $\dot{r}(0) = 0.0\text{ m/s}$, obtain a graph of the path of the mass in the (r, θ) plane up to the time it leaves the rod; that is, the time when $r > l$.

Section 9.2.3

9.8 The transfer function for a mechanical system shown in Figure 9.32 when subjected to base excitation $x_b(t)$ is

$$G(s) = \frac{x_m(s)}{x_b(s)} = \frac{2\zeta\omega_n s + \omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

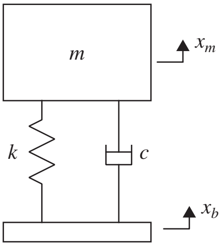


Figure 9.32 Spring–mass–damper system excited at its base.

Let $\zeta = 0.1$ and $\omega_n = 4$ rad/s. Use `bode` to determine the amplitude and phase response of this system for $0 \le \omega \le 10$. Compare these results with those obtained from the following analytical solution:

$$G(\omega) = \sqrt{\frac{\omega_n^4 + \left(2\zeta\omega\omega_n\right)^2}{\left(\omega_n^2 - \omega^2\right)^2 + \left(2\zeta\omega\omega_n\right)^2}}$$

$$\phi(\omega) = \tan^{-1}\left(\frac{2\zeta\omega}{\omega_n}\right) - \tan^{-1}\left(\frac{2\zeta\omega\omega_n}{\omega_n^2 - \omega^2}\right)$$

9.9 Consider a vibratory system whose motion is described by Eq. (9.9) with $g(x, \dot{x}) = 0$ Determine the response of the mass when

$$f(\tau) = \frac{1}{\tau_o}\left[\tau u(\tau) - (\tau - \tau_o)u(\tau - \tau_o)\right]$$

for $0 \le \tau \le 30$, $\zeta = 0.1$, and $\tau_o = 15, 6$, and 0.7; $u(\tau)$ is the unit step function. Include in each graph $f(\tau)$.

Section 9.2.5

9.10 A single-degree-of-freedom system is shown in Figure 9.33 with a dead zone of width $2b$ centered on its equilibrium position. The governing equations of the system are

$$\begin{aligned} m\ddot{x} + k(x + b) + 2c\dot{x} &= F(t) \qquad x < -b \\ m\ddot{x} + 2c\dot{x} &= F(t) \qquad -b \le x \le b \\ m\ddot{x} + k(x - b) + 2c\dot{x} &= F(t) \qquad x > b \end{aligned}$$

a. Determine the free response of a system with $m = 10.0$ kg, $k = 150 \times 10^3$ N/m, and $c = 50$ Ns/m when the motion is initiated from $x(0) = 0$ m and $dx(0)/dt = 2$ m/s in the following cases: (a) dead zone $b = 1.0\,\mu$m and (b) dead zone $b = 1{,}000\,\mu$m.

b. For $b = 5.0\,\mu$m and $x(0) = 0$ m and $dx(0)/dt = 0$ m/s, determine the response of the system when $F(t) = 20\cos(12t)u(t)$ N, where $u(t)$ is the unit step function.

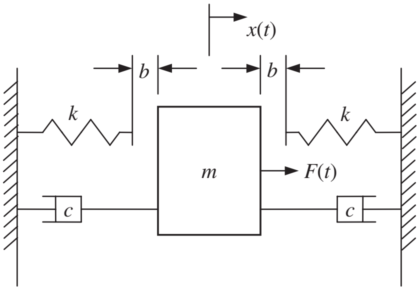


Figure 9.33 Spring–mass–damper system with dead zone.

9.11 An overhead crane's trolley is carrying, via a cable, a load of mass m as shown in Figure 9.34. When the trolley is moved with an acceleration $b(t)$, the governing equation of motion of the crane load is

$$L\frac{d^2\theta}{dt^2} + g\sin\theta = -b(t)\cos\theta$$

where $g = 9.8\ \text{m/s}^2$ is the gravity constant. If the cable length is 2 m, then graph the swing motion $\theta(t)$ for the following accelerations of the trolley over the time interval $0 \le t \le 10$ s:

a. $b(t) = 10u(t)\ \text{m/s}^2$, $\theta(0) = 0.2$ rad, and $d\theta(0)/dt = 0$ rad/s, where $u(t)$ is the unit step function.

b. $b(t) = 0.2u(t)\ \text{m/s}^2$, $\theta(0) = 0.2$ rad, and $d\theta(0)/dt = 0$ rad/s, where $u(t)$ is the unit step function.

9.12 Consider a single-degree-of-freedom system with a linear spring of stiffness k (N/m) and a cubic spring of stiffness αk (N/m^3) that is subjected to a base excitation $w(\tau)$. If the displacement of the mass is $x(\tau)$, the static deflection of the system is δ_{st}, and

$$w(\tau) = w_{\max}\left(1-(1-\gamma\tau)e^{-\gamma\tau}\right)$$

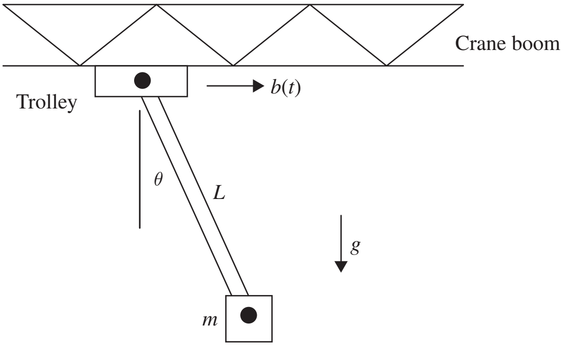


Figure 9.34 Trolley on an overhead crane carrying a swinging load m.

then the governing equation of the system is[30]

$$\frac{d^2z}{d\tau^2} + 2\zeta\frac{dz}{d\tau} + z + \alpha_o\left(z - \delta_o\right)^3 = -\alpha_o\delta_o^3 - \gamma^2(1 - \gamma\tau)e^{-\gamma\tau}$$

where $z(\tau) = (x(\tau) - w(\tau))/w_{\max}$, $\delta_o = \delta_{st}/w_{\text{amx}}$, and $\alpha_o = \alpha w^2_{\max}$. The absolute displacement of the mass is

$$x(\tau) = w_{\max}\,z(\tau) + w_{\max}\left(1 - (1 - \gamma\tau)e^{-\gamma\tau}\right)$$

Determine $x(\tau)/w_{\max}$ for $\delta_o = 0.3$, $\alpha_o = 30$, $\gamma = 1$, $\zeta = 0.15$, and $0 \le \tau \le 25$.

Section 9.3.1

9.13 Consider the amplitude–response function $H_{11}(\Omega)$ given by Eq. (9.50). Set $\zeta_1 = 0$, $m_r = 0.05$, and $\omega_r = 1.0$. Vary ζ_2 from 0.05 to 0.3 in increments of 0.05 and plot $H_{11}(\Omega)$ at each of these values on the same graph for $0.6 \le \Omega \le 1.4$. It will be found that each of the graphs for the different values of ζ_2 intersect at two frequency locations. Verify that these two frequencies are the same as those given by

$$\Omega_{1,2} = \sqrt{\frac{1}{2 + m_r}\left[1 + \omega_r^2(1 + m_r) \mp \sqrt{(\omega_r^2 - 1)^2 + m_r(2 + m_r)}\right]}$$

9.14 A model that is frequently used to study the bounce-pitch motion of a vehicle is shown in Figure 9.35. The equations governing the undamped system when undergoing free harmonic oscillations of the from $x = Xe^{j\omega t}$ and $\theta = \Theta e^{j\omega t}$ are

$$-\omega^2 mX + (k_1 + k_2)X + (L_2k_2 - L_1k_1)\Theta = 0$$
$$-\omega^2 I_c\Theta + (k_1L_1^2 + k_2L_2^2)\Theta + (L_2k_2 - L_1k_1)X = 0$$

If $k_1 = 14{,}600$ N/m, $k_2 = 21{,}900$ N/m, $L_1 = 1.52$ m, $L_2 = 1.22$ m, $m = 730$ kg, and $I_c = 1{,}360$ kg m^2, then find the natural frequencies and mode shapes.

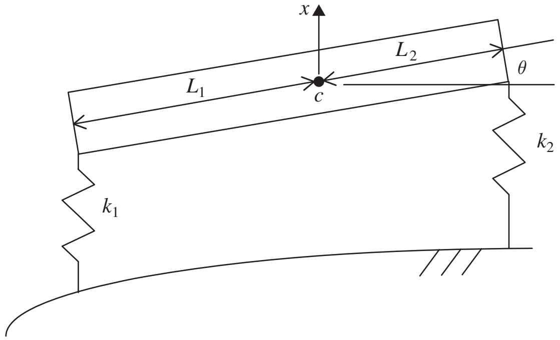


Figure 9.35 Two-degree-of-freedom model of a vehicle.

[30] *Ibid*, Example 6.9.

9.15 Consider the two-degree-of-freedom system shown in Figure 9.18. If an impulse force is applied to m_2, then the Laplace transform of the ratio of the force transmitted to the ground to the force applied to mass m_2 is

$$T_R(s) = \frac{\omega_r\big(2\zeta_1 s + 1\big)\big(2\zeta_2 s + \omega_r\big)}{D(s)}$$

where $D(s)$ is given by Eq. (9.47a). For $\zeta_1 = \zeta_2 = 0.1$ and $m_r = 1.0$, determine $T_R(\tau)$ for $\omega_r = 0.05$ and 0.25 over the range $0 \leq \tau \leq 100$.

Section 9.3.2

9.16 Consider the three-degree-of-freedom system shown in Figure 9.36. When the system undergoes harmonic oscillations of the form $x_l = X_l e^{j\omega t}$, $l = 1, 2$, and $\theta = \Theta e^{j\omega t}$ and the displacements and rotation are small, the resulting equation from which the natural frequencies and mode shapes can be determined is

$$\big[[K] - \omega^2[M]\big]\big\{Y\big\} = \big\{0\big\}$$

where

$$\big\{Y\big\} = \begin{Bmatrix} X_1 \\ X_2 \\ \Theta \end{Bmatrix}, \qquad \big[M\big] = \begin{bmatrix} m_1 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & J_G \end{bmatrix}$$

$$[K] = \begin{bmatrix} k_1 + k_2 + k_3 & -k_3 & (k_1L_1 - k_2L_2 + k_3L_3)\cos\theta_o \\ -k_3 & k_3 & -k_3L_3\cos\theta_o \\ (k_1L_1 - k_2L_2 + k_3L_3)\cos\theta_o & -k_3L_3\cos\theta_o & k_{33} \end{bmatrix}$$

$$\begin{aligned} k_{33} = {} & -(k_1L_1 - k_2L_2 + k_3L_3)x_{1o}\sin\theta_o + k_3L_3x_{2o}\sin\theta_o \\ & + \big(k_1L_1^2 + k_2L_2^2 + k_3L_3^2\big)\big(\cos^2\theta_o - \sin^2\theta_o\big) \end{aligned}$$

$$L_3 = (L_1 - L_2)/2$$

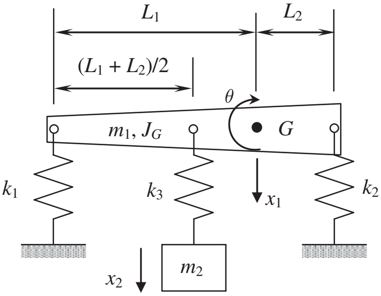


Figure 9.36 Three-degree-of-freedom system.

The static equilibrium positions x_{1o}, x_{2o}, and θ_o are determined from

$$\begin{bmatrix} k_1+k_2+k_3 & -k_3 & k_1L_1-k_2L_2+k_3L_3 \\ -k_3 & k_3 & -k_3L_3 \\ k_1L_1-k_2L_2+k_3L_3 & -k_3L_3 & k_1L_1^2+k_2L_2^2+k_3L_3^2 \end{bmatrix} \begin{Bmatrix} x_{1o} \\ x_{2o} \\ Z_o \end{Bmatrix} = \begin{Bmatrix} m_1g \\ m_2g \\ 0 \end{Bmatrix}$$

and $\theta_o = \sin^{-1}Z_o$.

Determine the natural frequencies and mode shapes when $m_1 = 10$ kg, $m_2 = 40$ kg, $J_G = 8$ kg m^2, $k_1 = 10{,}000$ N/m, $k_2 = 35{,}000$ N/m, $k_3 = 8{,}100$ N/m, $L_1 = 1$ m, and $L_2 = 0.3$ m.

9.17 A six-cylinder, four-cycle engine driving a generator is modeled as an eight-degree-of-freedom system, which for harmonic motions is of the form[31]

$$\left[[K_t] - \omega^2[J]\right]\left\{\Phi\right\} = \left\{0\right\}$$

where

$$[J] = \begin{bmatrix} 21 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 21 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 21 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 21 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 21 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 21 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 98 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 49 \end{bmatrix} \text{kg} \cdot \text{m}^2$$

$$[K_t] = \begin{bmatrix} 51 & -51 & 0 & 0 & 0 & 0 & 0 & 0 \\ -51 & 102 & -51 & 0 & 0 & 0 & 0 & 0 \\ 0 & -51 & 102 & -51 & 0 & 0 & 0 & 0 \\ 0 & 0 & -51 & 102 & -51 & 0 & 0 & 0 \\ 0 & 0 & 0 & -51 & 102 & -51 & 0 & 0 \\ 0 & 0 & 0 & 0 & -51 & 117 & -66 & 0 \\ 0 & 0 & 0 & 0 & 0 & -66 & 81 & -15 \\ 0 & 0 & 0 & 0 & 0 & 0 & -15 & 15 \end{bmatrix} \times 10^6 \quad \text{Nm/rad}$$

and Φ is a vector of the angular motion of the masses of the system. Determine the natural frequencies and mode shapes associated with the system and plot the mode shapes as shown in Figure 9.37.

9.18 Consider a spinning rigid circular shaft that is elastically supported at each end, as shown in Figure 9.38. The rotor is spinning at an angular speed of ω rad/s about its axis. Furthermore, the rotor has a polar moment of inertia J_p about the axis of rotation, a transverse moment of inertia J_t about any axis in the plane of rotation, and support stiffness k_1 and k_2 in their respective horizontal directions. The free whirling speeds Ω can be determined from the solution to the eigenvalue problem

$$[[K^*] - \Omega^2[M^*]]\{w\} = \{0\}$$

[31] G. Genta, *Vibration of Structures and Machines: Practical Aspects*, Springer-Verlag, New York, 1993.

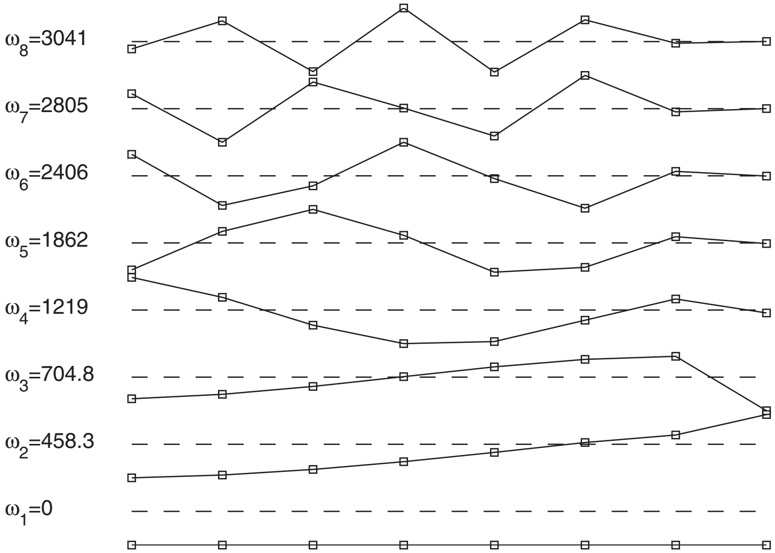


Figure 9.37 Natural frequencies and mode shapes of an eight-degree-of-freedom system.

where

$$K^* = \begin{bmatrix} KM^{-1}K & KM^{-1}G \\ G'M^{-1}K & K + G'M^{-1}G \end{bmatrix} \quad M^* = \begin{bmatrix} K & 0 \\ 0 & M \end{bmatrix}$$

and

$$M = \begin{bmatrix} m & 0 & 0 & 0 \\ 0 & J_t & 0 & 0 \\ 0 & 0 & m & 0 \\ 0 & 0 & 0 & J_t \end{bmatrix} \qquad G = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -J_p\omega \\ 0 & 0 & 0 & 0 \\ 0 & J_p\omega & 0 & 0 \end{bmatrix}$$

$$K = \begin{bmatrix} k_1 & 0 & 0 & 0 \\ 0 & k_2 & 0 & 0 \\ 0 & 0 & k_1 & 0 \\ 0 & 0 & 0 & k_2 \end{bmatrix}$$

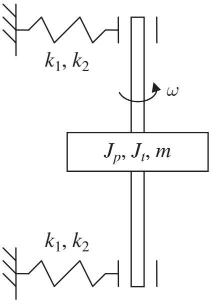


Figure 9.38 Rigid spinning rotor on an elastic support.

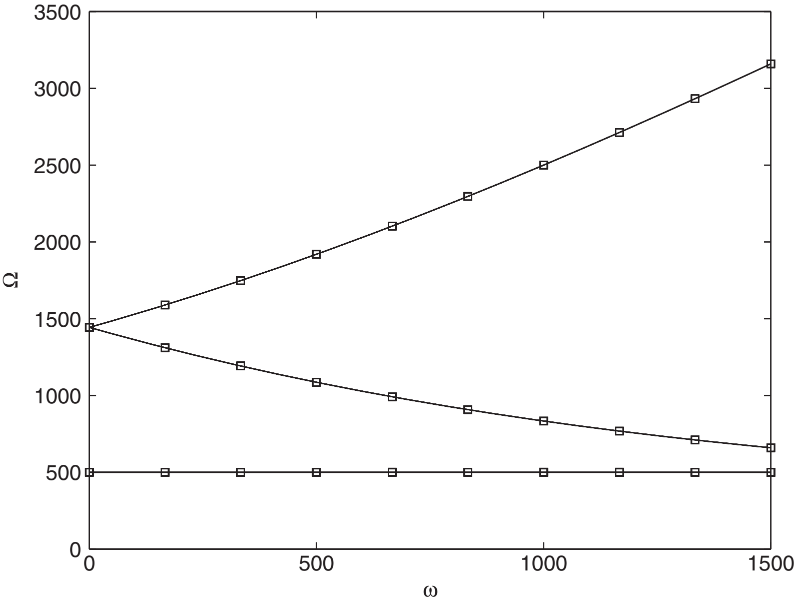


Figure 9.39 Campbell diagram for a spinning rigid rotor on an elastic support.

If $m = 10$ kg, $J_p = 2$ kg m^2, $J_t = 1.2$ kg m^2, $k_1 = k_2 = 2.5 \times 10^6$ N/m, then plot the value of Ω as a function of ω in the range $0 \leq \omega \leq 1{,}500$ rad/s. This graph is an example of a Campbell diagram.[32] The speed at which $\Omega = \omega$ is called the critical speed. The results should look like those shown in Figure 9.39.

BIBLIOGRAPHY

B. Balachandran and E. B. Magrab, *Vibrations*, 2nd ed., Cengage, Toronto, ON, 2009.

D. T. Greenwood, *Principles of Dynamics*, 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1988.

F. J. Hale, *Introduction to Space Flight*, Prentice Hall, Englewood Cliffs, NJ, 1994.

D. J. Inman, *Engineering Vibration*, Prentice Hall, Englewood Cliffs, NJ, 1994.

E. B. Magrab, *Vibrations of Elastic Structural Members*, Sijthoff & Noordhoff, The Netherlands, 1979.

L. Meirovitch, *Elements of Vibration Analysis*, McGraw Hill, New York, 1986.

F. C. Moon, Applied Dynamics with Applications to Multibody and Mechatronic Systems, John Wiley & Sons, New York, 1998.

A. H. Nayfeh and B. Balachandran, *Applied Nonlinear Dynamics: Analytical, Computational, and Experimental Methods*, John Wiley & Sons, New York, 1995.

S. S. Rao*, Mechanical Vibrations*, 3rd ed., Addison-Wesley, Reading, MA, 1995.

B. H. Tongue, *Principles of Vibration*, Oxford University Press, New York, 1996.

[32] *Ibid*.

10

Control Systems

Gregory C. Walsh

The representation, design, and evaluation of control systems using MATLAB's Controls Toolbox and SIMULINK are presented.

10.1 INTRODUCTION TO CONTROL SYSTEM DESIGN

Many products contain control systems and engineers are often faced with the challenge of implementing them. MATLAB offers tools to assist a designer in this task. In this chapter, we discuss some of these tools and provide examples on how to employ them. There are many different examples of controls, but for the purpose of illustration, in this chapter we will focus on the use of MATLAB to design controllers for the servomotor shown in Figure 10.1.

The motor shown in Figure 10.1 has an optical encoder attached to the shaft, an amplifier to drive the motor, and a joystick to command the motor. Between the joystick and the motor is an embedded computer. The computer can read the position of the joystick, track the angle of the motor shaft, and from this information apply an algorithm that computes the appropriate voltage that is applied to the motor. The embedded computer typically will make this computation many times a second.

In the servomotor example, the design goal is for the angle of the motor to track the position of the joystick. If the joystick is moved quickly from one position to another, the motor should follow swiftly without overshoot or without oscillating around its new location. In addition, the position of the shaft should be very close to the position commanded by the joystick.

Engineers solve many variations of control system design problems. The joystick might command the velocity of the motor instead of its position. The motor might be replaced by a heater and the angle encoder replaced by a temperature sensing device. If the controller adjusts the height of an elevator or the position of a driverless airport train, the design goal of being swift might not be nearly as important as not overshooting or oscillating about the final destination. If the controller is to point a telescope, then the final position becomes a dominant concern. Also, as a system's components wear, the system should preserve as much as possible its performance, and in the event of failure it should fail safely.

Control system design objectives fall into three categories: performance, safety, and robustness. In this chapter, we shall focus on quantifiable performance goals. Objectives verified over short time periods such as overshoot and swiftness (rise time) are called transient response design goals. Those tested over longer periods of time such as the eventual pointing accuracy of a telescope are called steady-state design goals.

Other important controller design goals include insuring that a system does not enter a dangerous state nor do damage to itself or others; insulating the performance against disturbances and noise; and designing the controller so that the performance

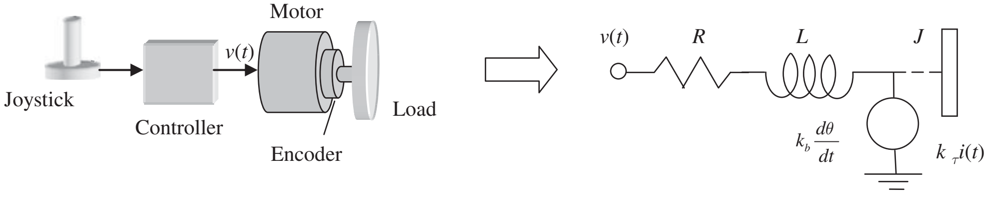


Figure 10.1 Common electric servomotor.

is maintained even in the face of changes to the plant itself. Safety and graceful failure objectives are clear for such systems as the driverless airport train. Disturbances in control systems refer to uncontrolled and possibly unmeasured inputs to the system, such as wind buffeting a servomotor's load. Noise in control systems typically enters in the measurement of its output.

Although control systems and control design problems wear many guises, they contain certain key elements shown in the block diagram of Figure 10.2. The system being controlled is called the plant. In the servomotor example, the plant is composed of the motor, amplifier, and encoder. A plant generally has one or more inputs $u(t)$, such as the power applied to the motor, and one or more outputs $y(t)$, such as the motor angle measured by the optical encoder. The joystick in the servomotor example is a source of commands and these commands are called the reference $r(t)$. The difference between the output of the plant $y(t)$ and the commanded output reference $r(t)$ is called the error signal $e(t) = r(t) - y(t)$. The controller uses the error signal $e(t)$, and possibly the output $y(t)$ and reference $r(t)$, to compute the needed $u(t)$. The concept of using the output $y(t)$ to compute the input $u(t)$ is called feedback and is the hallmark of a control problem. The output is literally fed back into the input.

MATLAB provides a tool called SIMULINK, which allows the engineer to literally draw the block diagram in Figure 10.2 and simulate the resulting behavior of the control system. The signals connecting the blocks such as $r(t)$ and $y(t)$ become time sequences represented by vectors. The contents of the blocks such as the plant or the controller may be filled with transfer-function and state–space models representing differential or difference equations. The engineer can also place a program in the controller block to run on the embedded processor through the MATLAB–MEX interface, or conversely, there are tools that instantiate the controller block in code for the embedded processor.

In this chapter, we demonstrate how one may use the tools available in MATLAB for designing controllers and analyzing control problems. In Section 10.2, different representations of plants and controllers are reviewed and include block diagrams, transfer functions, and state–space models. The representation of models in discrete time, that is, as an embedded controller would view them, is also reviewed. With different representations in hand, we show in Section 10.3 how to compute the response $y(t)$ given the plant and the input $u(t)$. Section 10.3 also considers the inverse problem of system identification; that is, identifying the plant given the response $y(t)$ and the input $u(t)$. In Section 10.4, design tools such as bode plots, root locus, and

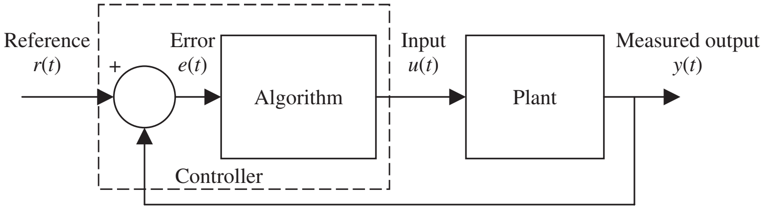


Figure 10.2 Schematic diagram of a feedback loop.

LQR/LQG are presented. Finally, Section 10.5 is devoted to the detailed discussion of the four different applications.

10.1.1 Tools for Controller Design

In control systems, the designer knows from the outset what the desired behavior of the systems is. The control design problem, therefore, involves changing the physical system so that the desired behavior occurs. This requires not only the ability to predict what will happen to a given model of the system but also what changes in the system model are needed to obtain the desired behavior. Thus, control system design is an inverse problem and design demands a good understanding of the solutions of ordinary differential equations. Ordinary differential equations are used to describe the behavior of many physical systems. Prior to the advent of computational tools like MATLAB, the understanding had to be gained by solving hundreds of differential equations by hand. Now one can direct MATLAB to solve the differential equations provided they know how to present them to the program.

In this chapter, we consider linear, time-invariant ordinary differential equations. MATLAB has three representations of linear time-invariant ordinary differential equations that are convenient for the solution of control problems:

1. State–space equations
2. Transfer functions
3. Block diagrams

State–space representations are time domain based and use matrices. Transfer functions are Laplace domain based and use polynomials of the complex variable *s*. Block diagram representations, available through the SIMULINK toolbox in MATLAB, visually depict the input and output connections. Conversion between the various representations is facilitated by built-in functions provided by MATLAB.

In this chapter, only the performance design objectives in the following three categories are considered.

1. Transient
2. Steady state
3. Stability

Transient design requirements focus on the short-term behavior of the system and address concerns such as responsiveness and stiffness. Steady-state requirements focus on the long-term behavior of the system, answering questions about how the system will perform over long periods of time. Standard input signals such as steps, ramps, and sinusoids are applied to test whether or not the system meets the transient and steady-state design requirements. MATLAB provides functions for finding the response of systems to the standard test signals. Transient and steady-state requirements are performance oriented, while feedback stability has more to do with safety. Feedback has the potential both to remove and to introduce instability into otherwise well-behaved physical processes, and instability must always be avoided.

Transient and steady-state design requirements are typically in conflict, forcing one to make a design trade-off. In practice, limitations in control performance come from inherent limitations in the sensors, actuators, and the plant.

Graphical tools used to solve control problems include Bode plots, Nyquist plots, and root locus plots. Linear algebra-based tools are used in more advanced design techniques such as LQG (linear quadratic Gaussian), H_∞, and μ-synthesis. For most single-input and single-output control designs, one of five controllers that are presented will provide a means to control the system.

10.1.2 Naming and File Conventions

In the course of this chapter, we shall use a standard set of naming conventions. Because the description of even a simple differential equation requires multiple vectors and matrices, MATLAB has provided a method for gathering the necessary matrices and vectors under a single name. These collections of matrices, vectors, and even strings are called systems. We will use the name *Plant* to label systems whose structure is fixed during the controller design and the name *Control* to label the part that we will be able to choose. The final closed-loop system consisting of the *Plant* connected with the *Control* will be labeled *clSys*. If the feedback connection is broken, we will call the resulting open loop system *olSys*. The MATLAB functions used in this chapter to assemble and analyze control systems take systems as arguments, instead of vectors and matrices.

Several example systems are considered in this chapter. For convenience, we shall create function M files that return these model systems. The functions will return a system object. The examples covered include the following:

- A permanent magnet motor with a load (**MotorSS.m**)
- A pointer with a flexible shaft (**Pointer.m**)
- A magnetic levitator (**MagLev.m**)
- An inverted pendulum (**Pend.m**)
- A flywheel (**Fly.m**).

Controllers such as lead, lag, PI, and PD are generated as needed.

10.2 REPRESENTATION OF SYSTEMS IN MATLAB

We now return to the servomotor shown in Figure 10.1 to illustrate the different ways a set of differential equations may be represented. The controller, at any particular time, may select the voltage $v(t)$ applied to the windings and may read the angular position of the rotor $\theta(t)$. By reading through the manufacturer's data sheets, one can find values of parameters describing the behavior of the motor such as the conversion factor k_τ which when multiplied by the current in the motor coils $i(t)$ gives the torque applied to the motor shaft. Other parameters include the electrical resistance of the motor coils R and the motor coil's inductance L. The motor as it is spun will act as a generator and produce a voltage proportional to the angular velocity of the rotor that is proportional to k_b. The inertia J seen by the rotor is the sum of the rotor inertia J_m and the inertia of the load J_{load}.

With the values of these parameters known, one can write down a set of coupled linear ordinary differential equations by using torque balance and circuit analysis. The resulting set of equations constitutes a model of the motor behavior. In the case of the servomotor, the following coupled ordinary differential equations[1] describe the relationship between the input voltage $v(t)$ and the output angle $\theta(t)$:

$$L\frac{di(t)}{dt} + k_b\frac{d\theta(t)}{dt} + Ri(t) = v(t) \tag{10.1}$$

$$J\frac{d^2\theta(t)}{dt^2} + b\frac{d\theta(t)}{dt} - k_\tau i(t) = 0$$

The electrical constants are R, L, k_τ, and k_b, where R is the motor resistance, L is the winding inductance, k_τ is the conversion factor from current to torque, and k_b is the back electromotive force (emf) generator constant. The total inertia J is usually dominated by the load inertia J_{load}. The motor friction b is generally small if there is no gearbox.

The model described by Eq. (10.1) allows the engineer to predict the output motor angle $\theta(t)$ given the input voltages $v(t)$ applied over time. MATLAB provides the tools needed to estimate solutions to these differential equations. In general, the input to a control system or control system component is described by a real-valued function of time $u(t)$. This quantity typically represents some physical variable under control, such as a force, voltage, or temperature. The output of a control system is also described by a real-valued function of time $y(t)$. The value of this function is some measured quantity, such as position, pressure, or velocity. In this chapter, we will limit the discussion primarily to models where the relationship between the input function $u(t)$ and the output function $y(t)$ is represented by a linear time-invariant ordinary differential equation of the general form

$$a_n\frac{d^ny(t)}{dt^n} + a_{n-1}\frac{d^{n-1}y(t)}{dt^{n-1}} + \cdots + a_0y(t) = b_m\frac{d^mu(t)}{dt^m} + b_{m-1}\frac{d^{m-1}u(t)}{dt^{m-1}} + \cdots + b_0u(t) \tag{10.2}$$

where $n \geq m$. The coefficients of the equation, a_j and b_j, are constant real-valued numbers. In the servomotor example, these constants depend on the performance characteristics of the motor, which are either given in data sheets or are measured.

System models in MATLAB are stored as objects, and much like the graphics objects of Chapters 6 and 7, the properties of these models are accessible though the use of `get` and `set`. Introductory control topics typically focus on differential

[1] D. K. Anand and R. B. Zmood, *Introduction to Control Systems,* 3rd ed., Butterworth-Heinemann Ltd, Oxford, England, 1995.

equations of the form shown in Eq. (10.2), and MATLAB provides three classes to represent this type of input–output relationship:

- Transfer-function representation (class `tf`)
- State–space representation (class `ss`)
- Zero-pole-gain representation (class `zpk`).

Discrete-time linear systems are also of great practical interest, since control loops are often implemented on computers. All three representations also have discrete-time versions, where the additional information concerning the sampling time is appended. Having the system models encapsulated as objects allows the user to attach auxiliary data to the representation. Examples of data fields attached to system objects include `InputName`, `OutputName`, and `Notes`.

10.2.1 State–Space Models

State–space models use matrices to represent the ordinary differential equations and have gained popularity with the widespread use of computers. They can be more numerically reliable to solve than models represented by transfer functions. State–space models are models composed of a system of first-order-coupled differential equations. In order to represent the motor as a state–space model, we first have to convert Eq. (10.1) to a system of first-order equations as shown in Section 5.5.4. If we let

$$\begin{aligned} x_1(t) &= \theta(t) \\ x_2(t) &= \frac{d\theta(t)}{dt} = \omega(t) \\ x_3(t) &= i(t) \end{aligned}$$

then Eq. (10.1) becomes

$$\begin{aligned} \frac{dx_1}{dt} &= x_2 \\ \frac{dx_2}{dt} &= -\frac{b}{J}x_2 + \frac{k_\tau}{J}x_3 \\ \frac{dx_3}{dt} &= -\frac{k_b}{L}x_2 - \frac{R}{L}x_3 + \frac{v(t)}{L} \end{aligned}$$

If we define

$$\begin{aligned} x(t) &= [x_1(t), x_2(t), x_3(t)]' \\ u(t) &= v(t) \\ y(t) &= \theta(t) = x_1(t) \end{aligned}$$

then the state–space representation for the motor system is

$$\begin{aligned} \frac{dx(t)}{dt} &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned} \tag{10.3}$$

where

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & -b/J & k_r/J \\ 0 & -k_b/L & -R/L \end{bmatrix} \qquad B = \begin{Bmatrix} 0 \\ 0 \\ 1/L \end{Bmatrix}$$

$$C = [1\ 0\ 0\,] \qquad\qquad D = [0]$$

The matrices A, B, C, and D are the essential data needed to describe the differential equations in MATLAB. Notice that even though D is zero, the format demands all four matrices be provided. In general, state–space models of linear, time-invariant ordinary differential equations will take the form of Eq. (10.3), with four matrices A, B, C, and D of appropriate dimensions. Many linear algebra tools are implemented in MATLAB to compute and understand the nature of the solutions of equations described by these four matrices.

We now illustrate these results with several examples.

Example 10.1 State–space model of a servomotor

We shall create a function M file **MotorSS** that will return a state–space system model of the servomotor shown in Figure 10.1. The values assumed for the constants are $L = 5$ mH (motor inductance), $R = 5\ \Omega$ (motor resistance), $k_b = 0.125$ V/rad/s (back emf), $k_\tau = 15$ Nm/A (motor torque), $J = 0.03\ \text{kg}\cdot\text{m}^2$ (rotor inertia), and $B = 0.01$ Nm/rad/s (rotor friction).

The state–space model is created in the following function M file:

```
function Plant = MotorSS(Jload)
if nargin < 1
  Jload = 0;
end;
L = 5e-3;   R = 5;   kb = 12.5e-2;
ki = 15;   J = 3e-2 + Jload;   b = 1e-2;
A = [0, 1, 0; 0, -b/J, ki/J; 0, -kb/L, -R/L];
B = [0; 0; 1/L];
C = [1, 0, 0];
D = 0;
Plant = ss(A, B, C, D);
set(Plant, 'InputName', 'volts', 'OutputName', '\theta')
set(Plant, 'StateName', {'\theta', '\omega','i'})
set(Plant, 'Notes', 'Small DC servomotor')
```

The function ss collects the matrices A, B, C, and D into a single system object. Typing

MotorSS

in the command window displays

```
a =
            \theta   \omega       i
  \theta       0         1        0
  \omega       0   -0.33333     500
       i       0       -25    -1000
```

```
b =
              volts
   \theta        0
   \omega        0
      i        200
c =
              \theta    \omega     i
   \theta        1         0       0
d =
              volts
   \theta        0
Continuous-time model.
```

The default input label is $u1$, the default output label is $y1$, and the interior states are labeled $x1$, $x2$, and $x3$. We chose to label these quantities using our own labels, which was done with `set`. Thus, we have named the state–space variables θ, ω, and i. In addition, we have included a note to remind us what the model represents. Without labeling the model, the following would have been displayed in the command window.

```
a =
          x1         x2          x3
   x1      0          1           0
   x2      0   -0.33333         500
   x3      0        -25       -1000
b =
          u1
   x1      0
   x2      0
   x3    200
c =
          x1         x2          x3
   y1      1          0           0
d =
          u1
   y1      0
Continuous-time system.
```

When the function **MotorSS** is called, only the system object is returned and not any of the constants. If we wish to recover the matrices A, B, C, and D, then we use

[A, B, C, D] = `ssdata`(**MotorSS**)

which displays to the command window the matrices A, B, C, and D as

```
A =
  1.0e+003 *
     0    0.0010         0
     0   -0.0003    0.5000
     0   -0.0250   -1.0000
B =
     0
     0
   200
```

```
C =
  1  0  0
D =
  0
```

MATLAB provides functions that take system objects such as **MotorSS** as an argument. Suppose that we wish to examine the behavior of the motor when it is connected in a simple feedback configuration, as shown in Figure 10.3. From the schematic, we have that

$$v(t) = r(t) - y(t) = r(t) - Cx(t)$$

and, consequently, Eq. (10.3) becomes

$$\begin{aligned}\frac{dx(t)}{dt} &= (A - BC)x(t) + Br(t)\\ y(t) &= Cx(t) + Dr(t)\end{aligned}$$

which amounts to replacing A in Eq. (10.3) with $A - BC$. MATLAB provides a function for the operation we have just described mathematically. The command is

```
clSys = feedback(MotorSS, 1);
```

which returns the closed-loop system. The number 1 in the second argument describes the transfer function of the feedback loop, which we have assumed is 1. Note that A of *clSys* is equal to $A - BC$ of **MotorSS**. One can verify this by typing in the command window

```
clSys = feedback(MotorSS, 1);
Plant = MotorSS;
clSys.a-(Plant.a - Plant.b*Plant.c)
```

which returns a (3×3) matrix of zeros.

The function `feedback` performed the algebra necessary to connect the motor system into a new configuration. Other MATLAB functions, which take systems as arguments, solve the differential equations when inputs are applied. In the next example, we use `step` to compute the response of the system to a step input signal.

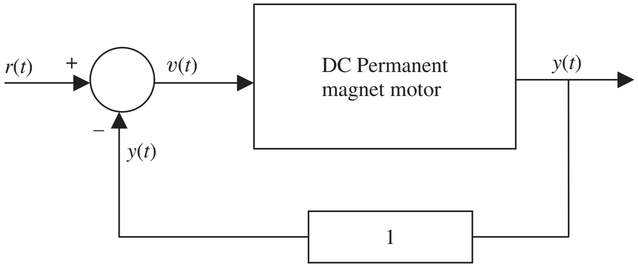


Figure 10.3 Simple unity gain feedback control system for controlling the servomotor.

Example 10.2 Step response of a servomotor

We shall determine the response of the servomotor shown in Figure 10.1, connected with unity feedback as shown in Figure 10.3, to a step input. The script is

```
[y, t] = step(feedback(MotorSS, 1));
plot(t, y, 'k-')
xlabel('Time (s)')
ylabel('Rotor angle\theta(t)(radians)')
```

Upon execution, we obtain the results shown in Figure 10.4.

We now extend the previous script to generate the step response of the servomotor for a variety of load inertias J_{load}. The script is as follows:

```
t = 0:0.05:2;
Jload = 0:0.01:0.1;
data = zeros(length(t), length(Jload));
for i = 1:length(Jload)
   data(:,i) = step(feedback(MotorSS(Jload(i)), 1), t);
end
mesh(Jload, t, data)
view([45, 30])
xlabel('Load inertia J_{load} (kg m^2)')
ylabel('Time (s)')
zlabel('Rotor angle \theta(t) (radians)')
```

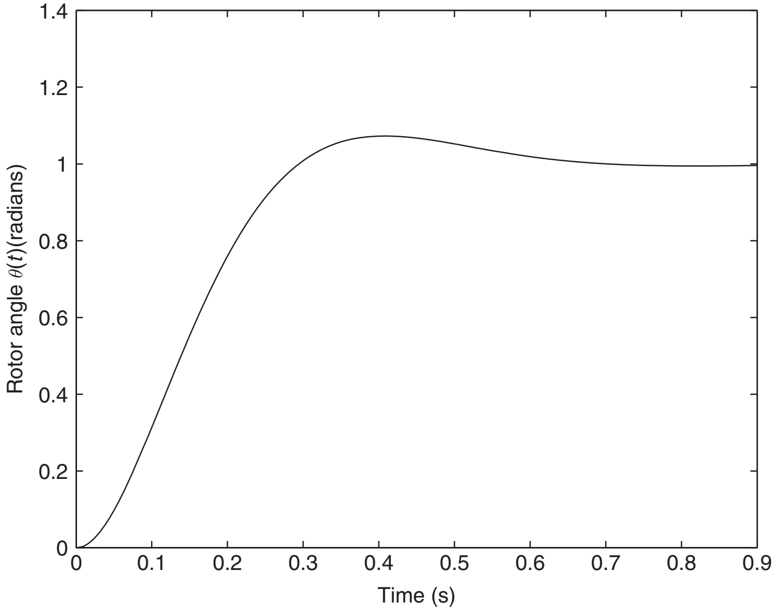


Figure 10.4 Step response of the servomotor control system of Figure 10.3.

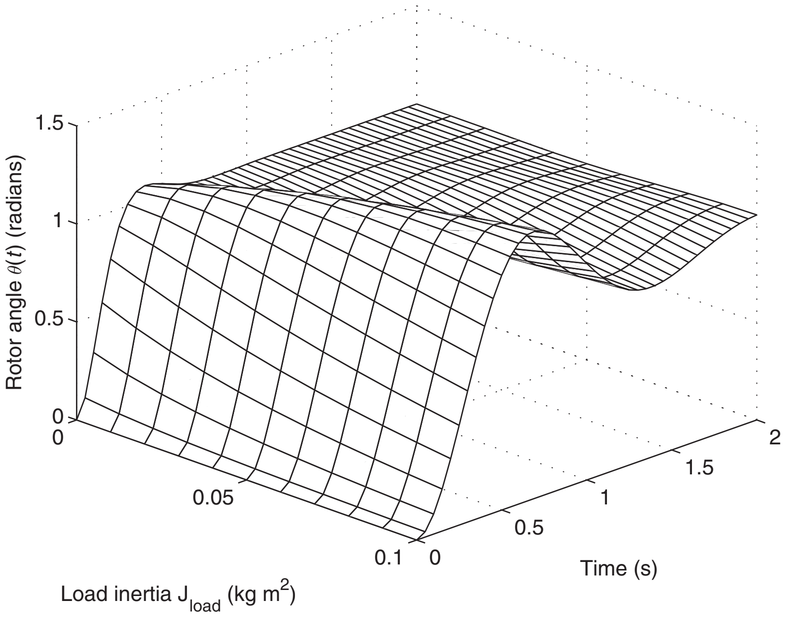


Figure 10.5 Response of the closed-loop servomotor to a step command in position as a function of load inertia.

Upon execution, we obtain the results shown in Figure 10.5. The response overshoots the goal more as the load inertia J_{load} is increased. In Section 10.3, we shall develop a similar set of expectations for system responses as a function of pole and zero locations.

10.2.2 Transfer-Function Representation

The transfer-function representation is obtained from the Laplace transform of the output divided by the Laplace transform of the input, assuming that the initial conditions are zero. Although this representation is less general and more sensitive to numerical errors than the state–space approach,[2] it remains popular for the intuition it provides. MATLAB supports transfer-function representations as well as state–space representations.

To find the transfer function of the servomotor, we take the Laplace transform of Eq. (10.1) and set the initial conditions to zero to obtain

$$k_b s\Theta(s) + (sL - R)I(s) = V(s)$$

$$(Js^2 + bs)\Theta(s) - k_\tau I(s) = 0$$

Solving these equations for $\Theta(s)$ gives

$$\Theta(s) = \frac{N(s)}{D(s)} V(s) \tag{10.4}$$

[2] *MATLAB Control System Toolbox*, *User's Guide*, Version 4, The MathWorks, Natick, MA, 1992.

where

$$N(s) = k_\tau \tag{10.5}$$

$$D(s) = JLs^3 + (JR + bL)s^2 + (bR + k_\tau k_b)s + 0$$

MATLAB represents these polynomials as vectors containing the polynomial coefficients, from highest power to the constant. In the case of the servomotor, the constant or zero-order coefficient of the denominator is zero. The format demands, however, that it be included in the vector even though it is zero, as was the case in the state–space representation for the matrix D.

The Laplace transform of the general ordinary differential equation of the form given by Eq. (10.2) yields the following transfer function:

$$H(s) = \frac{Y(s)}{R(s)} = \frac{b_m s^m + b_{m-1}s^{m-1} + \cdots + b_0}{a_n s^n + a_{n-1}s^{n-1} + \cdots + a_1 s + a_0} \tag{10.6}$$

where $a_n \neq 0$ and $n \geq m$. To represent this transfer function in MATLAB, two vectors are formed, one for the coefficients of the numerator polynomial and one for the coefficients of the denominator polynomial. Hence, a transfer function is represented in MATLAB as two vectors, each containing the coefficients of s. Thus, the coefficients of the numerator polynomial are

```
Num = [bm, ... b1, b0]
```

and those of the denominator polynomial are

```
Den = [an, . . . a1, a0]
```

The coefficient a_n is often set to 1 by dividing all coefficients by it, but doing so is not required.

The servomotor transfer-function model is now generated with the following function M file. It is the transfer-function representation of the state–space model generated by **MotorSS**.

```
function PlantTF = MotorTF
L = 5e-3;   R = 5;   kb = 12.5e-2;
ki = 15;   J = 3e-2;   b = 1e-2;
Num = ki;
Den = conv([L, R], [J, b, 0]) + [0, 0, kb*ki, 0];
PlantTF = tf(Num, Den);
```

When one types

```
MotorTF
```

in the command window, the following is displayed:

```
Transfer function:
            15
------------------------------------------------------
  0.00015 s^3 + 0.15 s^2 + 1.925 s
```

The coefficients of the numerator and denominator polynomials, *Num* and *Den*, are the essential data for the transfer-function realization, just as the matrices A, B, C, and D were the essential data for the state–space realization. Names of the input, output, and other descriptive fields can be set as before by using `set` and `get`. To extract the coefficients of the numerator and denominator polynomials, we use

```
[Num, Den] = tfdata(MotorTF, 'v')
```

The character *v* tells the function to return *Num* and *Den* as row vectors. If *v* is omitted, the numerator will be returned in a cell array, which is used to represent multiple-input multiple-output (MIMO) systems. For the most part, we shall confine the discussion to single-input single-output (SISO) systems. Executing the above statement, we obtain

```
Num =
  0   0   0   15
Den =
  0.0001   0.1500   1.9250   0
```

As with state–space systems, MATLAB functions accept transfer-function systems as input arguments. For example, the command

```
step(feedback(MotorTF, 1))
```

computes and plots the response of the controlled motor subjected to a step input command. Notice that `step` and `feedback` are the same commands used for the state–space models. These functions will work with any representation of a system object.

The roots of the denominator's polynomial are called the poles of the system, and the roots of the numerator are called the zeros. The functions `pole` and `tzero` will find the poles and zeros of a given transfer function. The function `pole` returns the poles (roots of the denominator polynomial in the case of transfer functions). For example,

```
p = pole(feedback(MotorTF, 1))
```

displays

```
p =
1.0e+02 *
-9.8744
-0.0645 + 0.0773i
-0.0645 - 0.0773i
```

which are the closed-loop poles of the simple unity feedback control system. The function tzero returns the transmission zeros of a plant. Thus, the execution of

```
z = tzero(feedback(MotorTF, 1))
```

returns z as an empty matrix; that is, the system has no zeros. Both pole and tzero may be applied to state–space systems as well. The transfer function is related to the state–space model through the equation

$$H(s) = C(sI - A)^{-1}B$$

where I is the identity matrix. From this equation, it can be shown that the roots of the denominator equation, the poles, are equal to the roots of the determinant of $(sI - A)$, which are the eigenvalues of the matrix A. The concepts of poles, zeros, and eigenvalues are frequently used to develop an understanding of the behavior of a control system.

Poles and zeros so strongly characterize the behavior of a control system that MATLAB provides an additional format based on poles and zeros called the zero-pole-gain format. In general, because the transfer function is a rational polynomial function, its numerator and denominator can both be factored to give

$$H(s) = k\frac{(s - z_1)(s - z_2)\cdots(s - z_m)}{(s - p_1)(s - p_2)\cdots(s - p_n)} \tag{10.7}$$

The transfer function is uniquely defined by its list of poles and zeros together with the constant gain k. We now generate a zero-pole-gain model of the closed-loop control system shown in Figure 10.3. First, we generate the lists of the poles and zeros and then set the gain to 1 by dividing by the DC gain. The script is

```
Poles = pole(feedback(MotorTF, 1));
Z = tzero(feedback(MotorTF, 1));
PlantZPK = zpk(Z, Poles, 1);
PlantZPK = dcgain(feedback(MotorTF, 1))/dcgain(PlantZPK)*PlantZPK;
step(PlantZPK)
```

which upon execution also yields Figure 10.4. We see that the functions feedback, pole, and tzero also apply to this model.

The unity gain controller step response shown in Figure 10.4 with these examples is slow and has a poor performance. We will see shortly that increasing the gain of the controller for the servomotor; that is, using a feedback gain greater than one, provides only a slight improvement. The MATLAB function feedback allows transfer functions and state–space models for controllers to be used in the place of the unity gain controller of this example. The performance of the servomotor system with controllers described by differential equations can be vastly superior to the simple feedback of this example.

10.2.3 Discrete-Time Models

Although the motor current $i(t)$ and the motor angle $\theta(t)$ might operate continuously and a set of differential equations describe how $v(t)$ and $\theta(t)$ are causally related to each other, an embedded computer will only observe the angle and change the voltage $v(t)$ at fixed times. To design a controller for an embedded computer, it is helpful to have a set of tools for studying the servomotor and the controller in a way that emulates the discrete nature of the embedded computer. MATLAB provides such a set of tools and methods to convert between continuous and discrete-time representations. Discrete-time versions of the state–space, transfer-function, and zero-pole-gain models can be generated by using

```
c2d
```

The function `c2d` uses, by default, the zero-order-hold approximation. Functions such as `step`, `impulse`, and `feedback` support discrete-time system models. The sampling time of the various components must be the same when combining elements. If

```
c2d(MotorSS, 0.001)
```

is typed in the command window, the following system description is displayed:

```
a =
              \theta       \omega         i
   \theta        1       0.00099818   0.00018374
   \omega        0       0.99507      0.31535
     i           0      -0.015768     0.36458
b =
               volts
   \theta    1.3203e-05
   \omega    0.036748
     i       0.12617
c =
              \theta       \omega         i
   \theta        1          0             0
d =
               volts
   \theta        0
Sampling time: 0.001
Discrete-time model.
```

Note that the matrices *a* and *b* of the discrete-time model are substantially different from those of the continuous time models. The state equations for discrete-time models evolve using difference equations, not differential equations. In the case of state–space models, we have for time indexed by *k*,

$$x[k+1] = A_d x[k] + B_d u[k]$$
$$y[k] = Cx[k] + Du[k]$$

which are matrix multiplications. These matrix multiplications match the behavior of the continuous-time differential equations given by Eq. (10.3) at sample times $t = k\Delta$, where Δ is the sampling interval as long as the matrices A_d and B_d are chosen properly. With a zero-order hold approximation, the discrete time matrices are given by[3]

$$A_d = e^{A\Delta}$$

$$B_d = \int_0^{\Delta} e^{A(\Delta-\tau)} B d\tau$$

which maps all left-half complex plane eigenvalues into the unit circle. The triangle approximation, the bilinear approximation (Tustin), the prewarped Tustin approximation, and the matched approximation are also available in MATLAB.

If the command

```
c2d(MotorTF, 0.001)
```

is typed in the command window, then the following polynomial representation of a discrete-time system is displayed.

```
Transfer function:
1.32e-005 z^2 + 4.191e-005 z + 8.023e-006
-----------------------------------------------------------------
  z^3 - 2.36 z^2 + 1.727 z - 0.3678

Sampling time: 0.001
```

Example 10.3 Conversion of a continuous-time model to a discrete-time model

We shall convert a continuous-time model to a discrete-time model with a sampling time of 1 ms. The script is as follows:

```
[ydiscrete, time] = step(feedback(c2d(MotorSS, 0.001), 1));
ycontinous = step(feedback(MotorSS, 1), time);
plot(time, ycontinous, 'k-.', time, ydiscrete, 'k-')
grid on
xlabel('Time (s)')
ylabel('Response')
legend('Continuous', 'Discrete', 'Location', 'SouthEast')
```

The result of executing this script is shown in Figure 10.6, where it is seen that there is no observable difference between the two responses. This virtually identical response is due to the very small sampling time chosen. As the sampling time increases, however, one would find that these responses would start to diverge. In other words, a lot happens to the system in between sampling times when the embedded computer is not paying attention. An investigation of how to keep the responses the same would show that one should adjust the voltage on the motor at a higher rate than the change of the motor itself and that in the conversion to discrete time some information is lost in the operation. For this reason, design will be considered primarily using continuous time in this chapter, and some implementation will be done in discrete time because embedded controllers are typically used to implement a control loop.

[3] See, for example, T. Kailith, *Linear Systems Theory*, Prentice Hall, Englewood Cliffs, NJ, 1980 and K. Astrom and B. Wittenmark, *Computer Controlled Systems*, 3rd ed., Prentice Hall, Upper Saddle River, NJ, 1997.

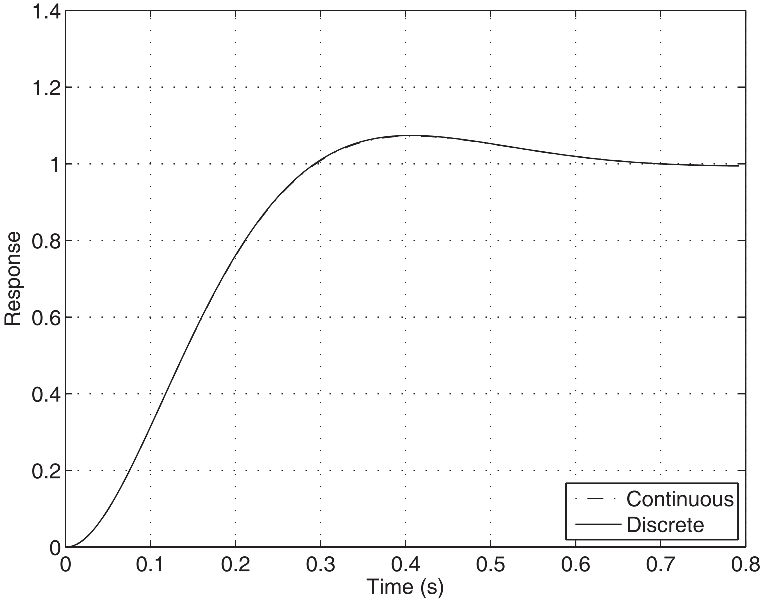


Figure 10.6 Comparison of the discrete-time step response and the continuous-time step response. (The continuous-time step response and the discrete-time response overlap.)

In discrete time, transfer functions are polynomials of the complex variable z instead of the complex variable s. The variable z is used to represent one sample delay. Consider the following generic discrete-time transfer function:

$$H(z) = \frac{Y(z)}{R(z)} = \frac{b_m z^m + b_{m-1} z^{m-1} + \cdots + b_0}{a_n z^n + a_{n-1} z^{n-1} + \cdots + a_1 z + a_0} \tag{10.8}$$

Multiplying both sides of Eq. (10.8) by $R(z)$, we obtain

$$\begin{aligned} &a_n z^n Y(z) + a_{n-1} z^{n-1} Y(z) + \cdots + a_0 Y(z) \\ &\quad = b_m z^m R(z) + b_{m-1} z^{m-1} R(z) + \cdots + b_0 R(z) \end{aligned}$$

This can be converted to discrete sample times

$$\begin{aligned} &a_n y[j + n] + a_{n-1} y[j + n - 1] + \cdots + a_0 y[j] \\ &\quad = b_m r[j + m] + b_{m-1} r[j + m - 1] + \cdots + b_0 r[j] \end{aligned}$$

Setting $k = j + n$ and $p = n - m$, we solve for $y[k]$ and obtain

$$\begin{aligned} y[k] = \frac{1}{a_{n-1}} (&-a_{n-1} y[k-1] - \cdots - a_0 y[k-n] + b_m r[k-p] \\ &+ b_{m-1} r[k-p-1] + \cdots + b_0 r[k-p-m]) \end{aligned} \tag{10.9}$$

Implementing control and filtering algorithms in an embedded system often involve a computation such as Eq. (10.9), which is known as an infinite impulse response (IIR) filter. A computer can implement this type of algorithm at each sample time with a series of multiplications and additions as long as the computer stores previous values of the computed output and measured inputs. Special-purpose computer architectures for embedded control and filtering systems, called digital signal processors, have been developed to implement this equation and other similar equations quickly. For many control implementations, however, special processors are not needed because the time scales of physical systems are much slower than the computational times of general-purpose microcontrollers.

10.2.4 Block Diagrams and SIMULINK

A typical control system is composed of several distinct units, such as the plant and the controller. More complicated structures involving many distinct subsystems are often studied, and these input–output maps are sketched by system designers using block diagrams, the basic elements of which are shown in Figure 10.7. MATLAB has provided a collection of built-in functions and operators to compute their transfer functions, a process referred to as block-diagram algebra. MATLAB supplies

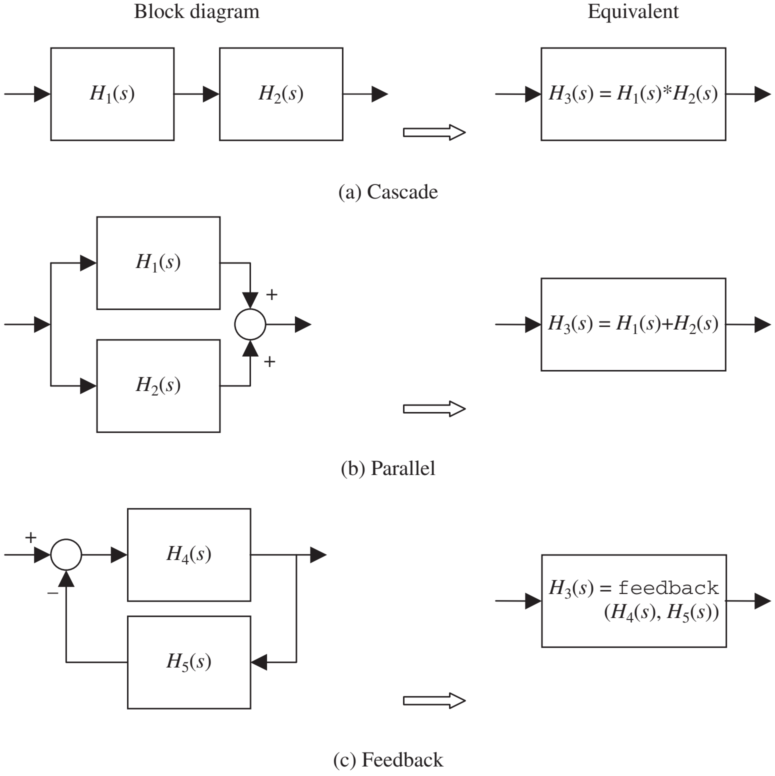


Figure 10.7 Most common block-diagram algebra operations: (a) Series (cascade). (b) Parallel. (c) Feedback.

standard operators+, −, *, / and provides feedback, series, and connect, among others, so that from the command line the block diagrams may be implemented and simulated. The SIMULINK toolbox provides a graphical user interface from which one can literally sketch the block diagram and simulate the characteristics of the resulting system.

The cascade connection of two systems shown in Figure 10.7a can be determined using the multiplication operator. For example, if system objects $H1$ and $H2$ represent the transfer functions $H_1(s)$ and $H_2(s)$, respectively, then the resulting cascaded system $H_3(s)$ can be obtained by

```
H3 = H1*H2
```

Division is also supported, but is applicable only to strictly proper models and will not be considered here.

The parallel connection of two systems illustrated in Figure 10.7b can be found using the addition and subtraction operators. If systems $H1$ and $H2$ are connected in parallel, then the resulting system is

```
H3 = H1 + H2
```

Subtraction is obtained by changing the sign in the above equation to minus.

The feedback connection shown in Figure 10.7c differs from the other two operations in that a function, instead of an operator, implements the operation. If systems $H4$ and $H5$ are connected in feedback, then the resulting closed loop system is given by

```
H3 = feedback(H4, H5);
```

Negative feedback is assumed; if positive feedback is desired, then $H_3(s)$ is obtained from

```
H3 = feedback(H4, H5,+1);
```

One common control configuration is called cascade feedback, which was shown in Figure 10.2. This configuration is the combination of Figure 10.7a, the cascade, with Figure 10.7c, feedback. In this case, the output of the *Plant* (H_2) is subtracted from the input, and the resulting error signal is fed to the *Controller* (H_1). To build the closed-loop model, we set

```
H4 = Controller*Plant
```

and

```
H5 = 1
```

The closed-loop system *clSys* is then obtained from

```
clSys = feedback(Controller*Plant, 1);
```

The functions feedback and series and the operators *, +, and − also support MIMO systems by allowing vector-valued inputs and outputs. In addition, matrix operations allow the quick construction of MIMO systems. For example, a two-input, one-output system $H3$ can be created from SISO plants, $H1$ and $H2$, using

```
H3 = [H1, H2]
```

A one-input, two-output system can be created with

```
H3 = [H1; H2]
```

The subsystems $H1$ and $H2$ can be extracted from $H3$ using the same operations used to extract submatrices discussed in Section 2.4. Cascade, parallel, and feedback connections require specifications as to which inputs are connected to which outputs when working with MIMO systems. The functions `series` and `parallel` are provided for these purposes. More complicated MIMO structures may be formed using `ift` and `connect.` We consider SISO control systems almost exclusively.

If the systems $H1$ and $H2$ are of different types, then an implicit conversion is performed so that the resulting model is homogeneous. MATLAB favors state–space models above all others and, therefore, if any one of the models in a computation is a state–space model, the result will be in state–space form. Between transfer-function and zero-pole-gain models, transfer-function models are favored.

SIMULINK

SIMULINK allows a designer to model and simulate systems by constructing them from a large library of components. The components are selected from the library, dragged to a modeling window, and connected and their individual parameters are specified. Then the model is run and the results displayed. SIMULINK is invoked by typing

```
simulink
```

in the command window. Variables defined in the command window are accessible from the SIMULINK window.

We shall illustrate how to model the DC servomotor with SIMULINK. The final result is shown in Figure 10.8. To generate the plant model, we first define it in the command window and extract the system matrices. Then we start SIMULINK. Thus,

```
[A, B, C, D] = ssdata(MotorSS);
simulink
```

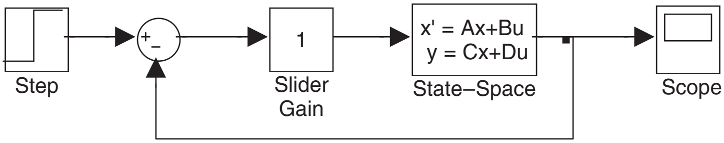


Figure 10.8 Block-diagram model of a DC servomotor created in SIMULINK.

This brings up the SIMULINK library browser window, which displays a list of available libraries and includes the SIMULINK main library. First, we click on the new page icon (white rectangle) to open a SIMULINK modeling window. Then, we return to the browser window and double-click on SIMULINK, which displays the following directory of SIMULINK component libraries:

Commonly Used Blocks
Continuous
Discontinuities
Discrete
Logic and Bit Operations
Lookup Tables
Math Operations
Model Verification
Model Wide Utilities
Ports & Subsystems
Signal Attributes
Signal Routing
Sinks
Sources
User-Defined Functions
Additional Math & Discrete

To open the components (blocks) of any of these libraries, one clicks on the library name of interest. We start by displaying the *Continuous* library. Next, we click on *State–Space* and, keeping the mouse button depressed, drag a copy of this library component to the modeling window. We place the component at its desired location and release the mouse button. Next, we go to the *Math Operations* library and sequentially select *Slider Gain* and *Sum* and drag them one at a time to the model window, placing each, respectively, to the left of the *State–Space* component. Then, we go to the *Sources* library and select *Step* and place it in the modeling window to the left of all components placed so far. The last component we select is *Scope* from the *Sinks* library. It is placed to the right of all components selected.

The next steps are to specify the parameters of the components. We start with the *State–Space* block and double-click on it. This brings up a *State–Space* parameter window with five places to enter data: A, B, C, D, and *Initial condition*. Since these matrices have been defined in the command window, we type A, B, C, and D, respectively, for each of the four quantities. Had we selected different names in the command window, we would have entered those names in their appropriate places. We leave the initial condition at 0. The signs of the summation are changed in the same manner; the second sign is changed to negative. The range of gains for the slider may also be selected. A range of 0 for the smallest and 15 for the largest is recommended. The center selection of 1 is left as is. We use the default values for the *Step* block except for *Step time* (time offset or delay), which we set to 0. Double-clicking on *Scope* brings up a simulation of an oscilloscope display. There are no parameters to select at this point.

We now connect the components to form a feedback control system. Each of the components in the project window may be moved and resized using the mouse, if desired. Connections are made at the small protrusions on the exterior of the blocks; by default, inputs are on the left and outputs on the right. Connections are made using the mouse and dragging a line from an output to an input. Place the crosshairs at the output of the *Step* block and, with the left button depressed, move the crosshairs to the plus (+) input of the *Sum* component. Continue this process until the connected block diagram looks like that shown in Figure 10.8. The line that goes from the middle of the *State–Space* block and the *Scope* to the negative input of the summing device is created as follows. Place the cursor (arrowhead) on the existing line between these two blocks and, while maintaining this position, depress the *Ctrl* key on the keyboard and then the left mouse button. While holding both of these down, move the crosshairs to the negative input of the summing device and release the mouse button and the *Ctrl* key. To adjust the line, simply click on it and, with the mouse button depressed, move it up or down to its desired position.

The placement and size of each component is not material, only the connections. There will be models in which, prior to making these connections, we will have to flip one or more of the blocks around so that the inputs are on the right and the outputs on the left. This operation is performed by selecting (clicking on) the block and then going to the *Format* pull-down menu and selecting *Flip Block*. Also from the format menu one can suppress the display of the block identifier beneath the block. Select the block and then select *Hide Name*. To put the identifier back, select the block and then select *Show Name*. These two choices do not appear together.

To run the simulation, we go to the *Simulation* menu and select *Start*. After the simulation has executed, double-click on *Scope* to see the results. Use the *x*- and *y*-axis zoom icons to obtain the desired resolution for the image. Note that under the *Simulation* menu, one may adjust the parameters of the simulation, such as the integration method used, by selecting *Parameters*. If the diagram is not completely or correctly connected, the simulation will not run, and MATLAB will send error messages to that effect to the command window and to a special pop-up window. When rendered as indicated in Figure 10.8, the results shown in *Scope* are those given in Figure 10.4.

Another example of using SIMULINK is given in Section10.5.2.

10.2.5 Conversion Between Representations

MATLAB provides functions for conversion between the three representations by using the system constructor functions `ss`, `tf`, and `zpk`. For example, the transfer-function model can be generated from the state–space model using the state–space model **MotorSS** in `tf`.

PlantTF = `tf`(**MotorSS**)

Slight differences in the coefficients of system models result from numerical errors in the conversion process.

The state–space model can be generated from the transfer-function model by using `ss`. Thus,

```
PlantTF = tf(MotorSS);
PlantSS = ss(PlantTF)
```

The resulting system matrices A, B, C, and D are not the same matrices that were defined in **MotorSS**, which points out that unlike the transfer-function model, there is no unique state–space representation for a given system. One typically uses

```
ssbal
```

to scale the input, state, and output quantities to make the simulation as well conditioned as possible. Thus,

```
PlantTF = tf(MotorSS);
PlantSS = ss(PlantTF)
PlantBal = ssbal(PlantSS)
```

attempts to find the best conditioned representation of the system.

The zero-pole-gain model can also be converted from the state–space or transfer-function models. For example, the zero-pole-gain model could have been generated from the state–space model by using

```
PlantTF = tf(MotorSS);
PlantSS = ss(PlantTF)
PlantZ = zpk(PlantSS)
```

This and the other conversion methods make use of the numerical root-finding algorithms within MATLAB and are sometimes subject to large numerical errors, especially for systems of order 10 and higher.[4] In practical situations, it is recommended that one resist changing representations too often.

10.3 RESPONSE OF SYSTEMS

In this section, we shall illustrate the use of a number of tools available for estimating the response of a system and conversely, for estimating a system given its response. Functions for computing the response of a system applicable to all three representations in both continuous and discrete time include `step`, `impulse`, `initial`, and `lsim`. The SIMULINK toolbox also supplies a number of built-in signal sources that simplify simulation when using block diagrams. For estimating a system from its response, we will use the function `arx`.

The locations of the poles and zeros, either in a transfer-function or state–space representation, are a shorthand notation that control engineers use to

[4] N. E. Leonard and W. S. Levine, *Using MATLAB to Analyze and Design Control Systems*, Benjamin/Cummings, Redwood City, CA, 1992.

estimate the responses of a system. Many design specifications can be translated into pole and zero location constraints. The controller design problem often becomes one of designing feedback so that the closed-loop poles lie in desired regions of the complex plane. Zeros cannot be moved by feedback.

10.3.1 Estimating Response from Systems

The response of a control system to a step input is the most commonly used benchmark to compare different controller designs. The MATLAB function `step` computes the step response and, if the return values are not requested, it plots the results. The function automatically determines a suitable range of times in which to compute the simulation if a time vector is not given. Executing the script

```
t = linspace(0, 0.8, 100);
theta = step(MotorSS, t);
plot(t, theta)
xlabel('Time (s)')
ylabel('\theta(t)')
```

we obtain Figure 10.9. From this figure, it is seen that the step response of the motor is a ramp, because if a constant voltage is applied to the windings, eventually a constant speed will be reached.

The MATLAB function `impulse` is used in the same manner as `step`, except that it computes the response of the system to an impulse.

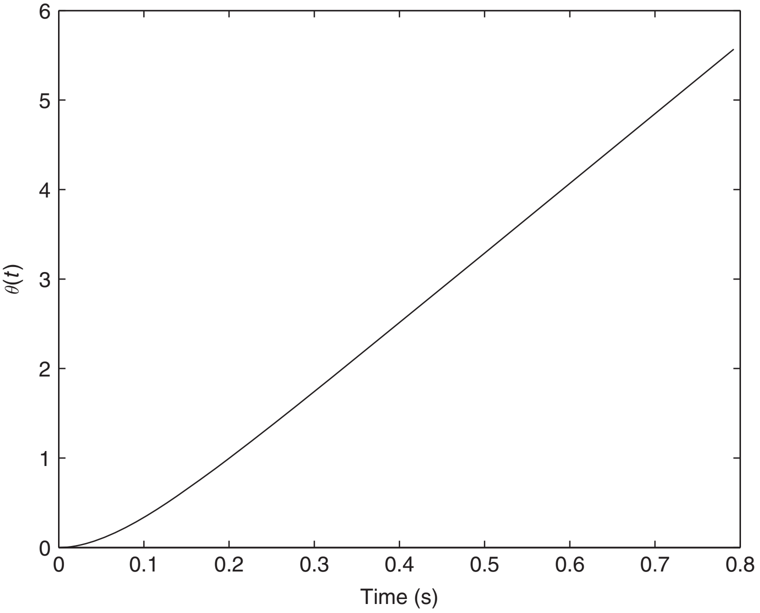


Figure 10.9 Step response of the motor without feedback.

MATLAB does not supply functions for all standard test inputs. However, MATLAB does provide the capability of determining the response of a system to an arbitrary input by using

```
lsim(sys, u, t)
```

where *sys* is the system under consideration and *u* is a vector representing the amplitude of the input as a function of time *t*. The lengths of *u* and *t* must be equal.

Example 10.4 Tracking error of a motor control system

We shall determine the steady-state tracking error of the motor control system given in Figure 10.3. The steady-state tracking error is the eventual difference between the desired position and the actual position when the input is a ramp. We will use a ramp with a slope of 1 over the range $0 \le t \le 1$. For the DC servomotor, the error is $e(t) = \theta(t) - t$ for $0 \le t \le 1$. The script is as follows:

```
t = linspace(0, 1, 100);
theta = lsim(feedback(MotorSS, 1), t, t);
plot(t, t'-theta, 'k-')
error = t(end) - theta(end);
hold on
plot([0, t(end)], [error, error], 'k--')
xlabel('Time (s)')
ylabel('Error e(t)')
```

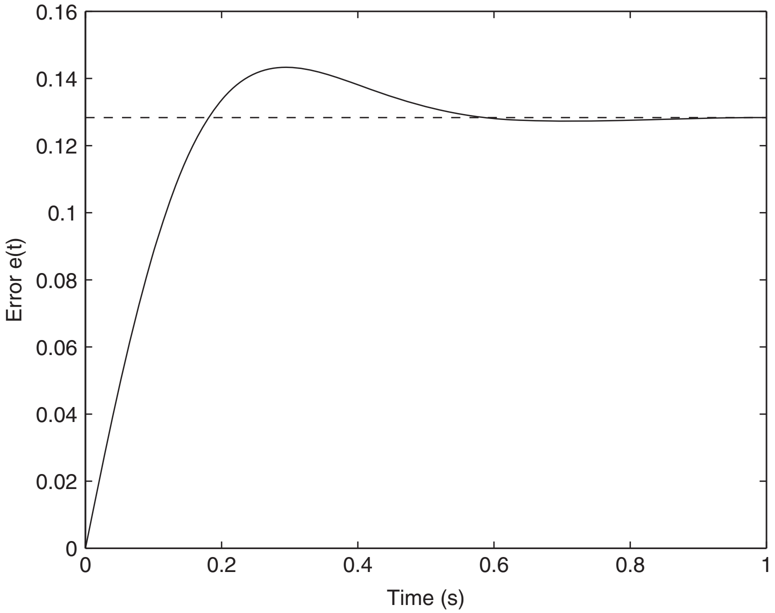


Figure 10.10 Tracking error of a control system as a function of time.

Upon execution, we obtain Figure 10.10. Note that after an initial transient, the error settles to approximately 0.13. A control system design objective could be to reduce this steady-state error to less than 0.05.

For a state–space system, the MATLAB function

```
initial(sys, x0)
```

runs a simulation with nonzero initial conditions, where *sys* is the system under consideration and *x*0 is a vector of initial conditions.

Example 10.5 Response of a DC motor to initial conditions

Consider the DC motor given in Figure 10.3 with an initial position $\theta(0) = 0$, an initial current $i(0) = 0$, and an initial angular velocity of $\omega(0) = 5$ rad/s. In addition, we assume a gain of 2. If the voltage across the windings is kept at zero (possibly by shorting them), then the following script will compute the response of the motor to these initial conditions.

```
x0 = [0; 5; 0];
[theta, t, x] = initial(feedback(2*MotorSS, 1), x0);
plot(x(:,1), x(:,2), 'k-')
grid on
xlabel('\theta(t)   (radians)')
ylabel('\omega(t)   (radians/s)')
```

where $x(:,1) = \theta(t)$, $x(:,2) = \omega(t)$, and $x(:,3) = i(t)$. The result of executing this script is shown in Figure 10.11.

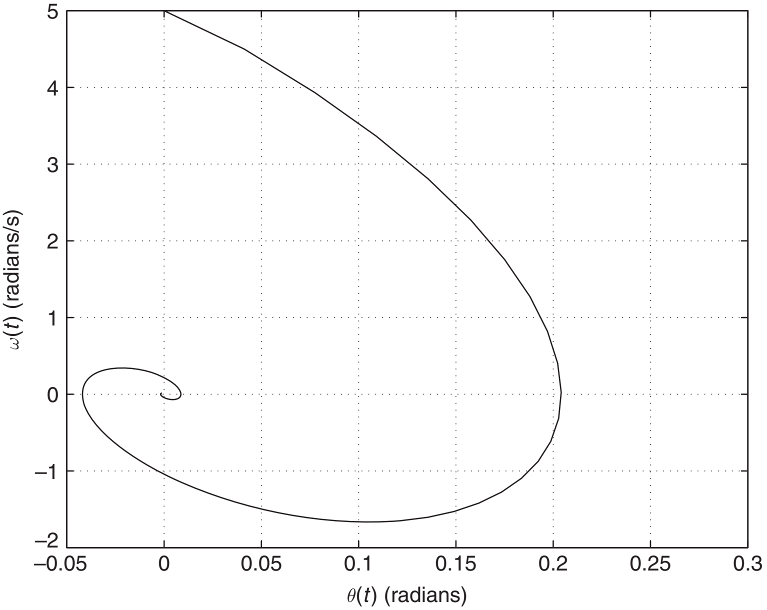


Figure 10.11 Phase plot of the rotor with an initial angular velocity $\omega(t) = 5$ rad/s.

10.3.2 Estimating Response from Poles and Zeros

When using MATLAB to solve control problems, it is important to have a qualitative understanding of the solution of differential equations. Since controller design is an inverse process, having this qualitative understanding allows one to know what the system should be like in order to achieve some desired behavior. MATLAB can be used to help develop this important qualitative understanding of the solutions to differential equations by solving many equations of particular types.

In this section, we shall compute the step response to a number of systems characterized by their pole and zero locations. First, we consider a first-order system with one pole and no zeros. We assume that the pole is located at $-\sigma$; therefore, the transfer function is

$$H(s) = \frac{\sigma}{s + \sigma}$$

The numerator is set to σ to keep the DC gain of the system at 1.

We now obtain the response of this system to a step input.

Example 10.6 Step response of first-order system to a range of pole locations

We shall generate a series of step responses for a series of pole locations ranging from slow ($\sigma = 0.1$) to quick ($\sigma = 2$). The script is

```
t = 0:0.1:10;
polevect = 0.1:0.1:2;
hold on
for i = 1:length(polevect)
```

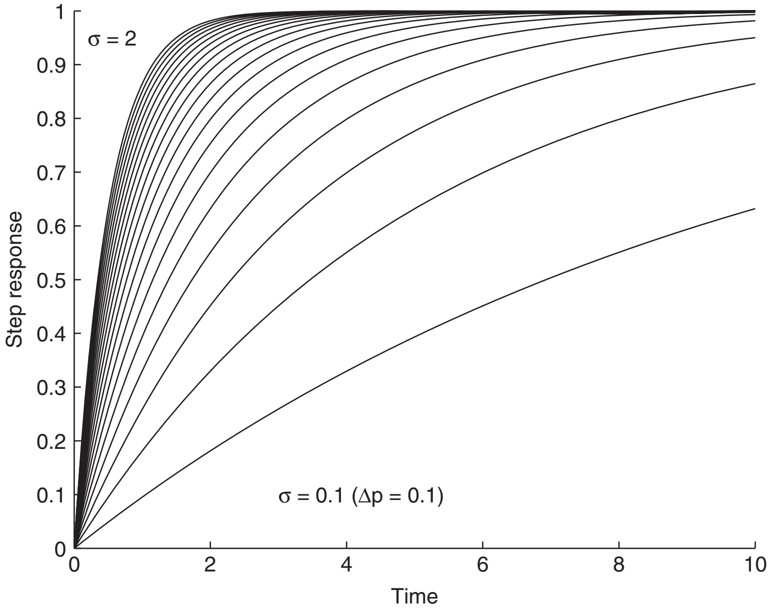


Figure 10.12 Response of a first-order system for a variety of pole locations σ.

```
  y = step(tf([polevect(i)], [1, polevect(i)]), t);
  plot(t, y, 'k-')
end
xlabel('Time')
ylabel('Step response')
text(0.2, 0.95, '\sigma = 2')
text(3, 0.1 ,'\sigma = 0.1 (\Delta\sigma = 0.1)')
```

The results from executing this script are shown in Figure 10.12.

As the pole of a first-order system approaches the imaginary axis, that is, as σ becomes small, the control system becomes more sluggish. Sluggishness is usually not a good characteristic of a control system, but sometimes, if the slow response is with respect to some disturbance that needs to be rejected, this slowness is a good characteristic. The best location of the system poles will depend on the objective of the control system.

If two first-order systems are cascaded, then the system becomes a second-order one. Many mechanical systems behave as second order, so a good understanding of second-order systems is important. Consider the following second-order system:

$$H(s) = \frac{\sigma^2 + \omega^2}{s^2 + 2s\sigma + \sigma^2 + \omega^2} \tag{10.10}$$

The system has complex poles located at $-\sigma \pm j\omega$, where σ is the real part of the pole locations and ω is the complex part. In general, σ represents the amount of damping in the system and ω specifies the strength of the storage mechanism or spring. However, since we are interested in relating the system's response to the location of the pole, the transfer function is parameterized by the pole location.

We shall now examine Eq. (10.10) as a function of its pole locations.

Example 10.7 Step response of second-order system to a range of pole locations

We shall explore the response of the system represented by Eq. (10.10) to a variety of pole locations. First, we plot the step response as a function of σ with $\omega = 1.0$. Then, we plot the step response as a function of ω with $\sigma = 0.5$. We also sketch the locus of pole locations next to each response plot. The script is as follows:

```
t = 0:0.4:10;   sigma = linspace(0.05, 1.0, 10);
data = zeros(length(t), length(sigma));
omega = 1.0;
for i = 1:length(sigma)
  data(:,i) = step(tf([sigma(i)^2 + omega^2], [1, 2*sigma(i) sigma(i)^2+omega^2]), t);
end
subplot(2,2,1)
mesh(t, -sigma, data')
ylabel('\sigma')
xlabel('Time')
zlabel('Response')
title('Response as a function of \sigma: \omega = 1.0')
data = zeros(length(t), length(omega));
sigma = 0.5; omega = linspace(0.3,2.0,10);
```

```
for i = 1:length(omega)
  data(:,i) = step(tf([sigma^2 + omega(i)^2], [1 2*sigma sigma^2+omega(i)^2]), t);
end
subplot(2,2,3)
mesh(t, omega, data')
ylabel('\omega')
xlabel('Time')
zlabel('Response')
title('Response as a function of \omega: \sigma = 0.5')
subplot(2,2,2)
hold on
plot([-0.1, -0.1], [1.0, -1], 'x')
plot([-0.1, -1], [1.0, 1.0])
plot([-0.1, -1], [-1.0, -1.0])
plot ([-1, -1], [1.0, -1], '<')
plot ([-2, 1], [0, 0], 'k')
plot ([0, 0], [-2, 2], 'k')
axis([-2, 1, -2, 2])
xlabel('Real axis')
ylabel('Imaginary axis')
title('Pole location')
subplot(2,2,4)
hold on
plot ([-0.5, -0.5], [0.3, -0.3], 'x')
plot ([-0.5, -0.5], [0.3, 1.5])
plot ([-0.5, -0.5], [-0.3, -1.5])
plot (-0.5, 1.5, '^')
plot (-0.5, -1.5, 'v')
plot ([-2, 1], [0, 0], 'k')
plot ([0, 0], [-2, 2], 'k')
axis([-2, 1, -2, 2])
xlabel('Real axis')
ylabel('Imaginary axis')
title('Pole location')
```

The results from executing this script are shown in Figure 10.13.

Like the first-order system, the steady-state response is $H(0) = 1$. As seen in Figure 10.13, when σ is fixed and ω is increased, the system response becomes less damped. When ω is held constant and σ is increased, the response becomes more damped. If the real part of the pole approaches the imaginary axis, then the system become less damped. If the pole crosses into the right half of the complex plane, then the response becomes unbounded.

Systems of higher order, like the electric motor, often behave like a second-order and sometimes a first-order system. The system poles with the greatest real part dominate the input–output behavior of the system if they are much greater than the real part of the closest poles, as long as there are no transmission zeros near them. This allows the designer to approximate the closed-loop behavior of a system with that of either a first- or second-order system. To understand why this is true, consider the velocity control of a DC permanent magnet motor. The equations of motion are the

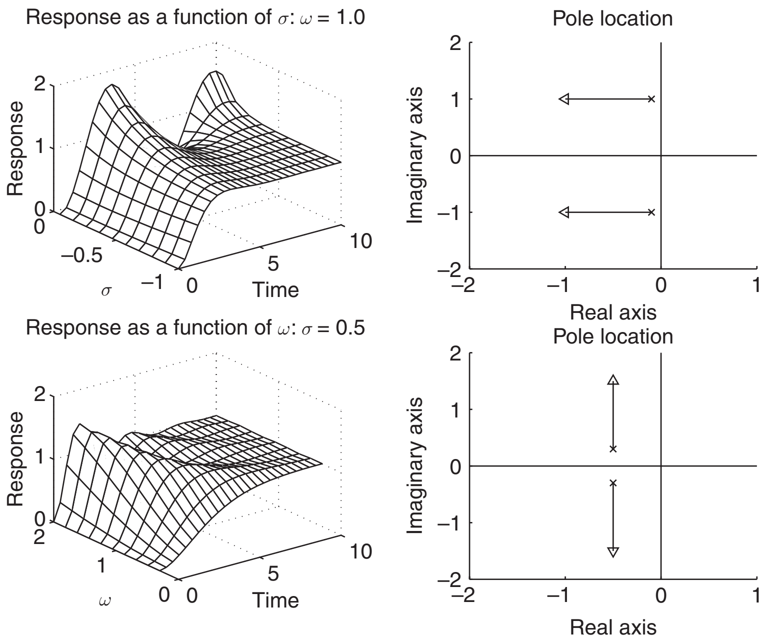


Figure 10.13 Response of a second-order system as a function of pole location.

same as Eq. (10.1), except the output is now rotor angular velocity $d\theta(t)/dt = \omega(t)$ instead of rotor angle $\theta(t)$. The following script makes the needed changes:

```
PlantRPM = MotorSS;
PlantRPM.c = [0, 1, 0];
PlantRPM = minreal(PlantRPM);
set(PlantRPM, 'OutputName', '\omega')
pole(PlantRPM)/(2*pi)
```

The first line of the script creates a system *PlantRPM* and then sets the readout matrix C to select $\omega(t)$ instead of $\theta(t)$. Since the rotor angle $\theta(t)$ is now unobservable, `minreal` is called to eliminate that state from the equations. The last line returns the poles of the system: one fast pole at approximately 157 Hz, which is related to the electronic coil, and one slow pole at approximately 2 Hz, which is related to the rotor dynamics. The slow pole due to the rotor dynamics dominates the open-loop response of the system, since the fast pole is more than ten times the speed of the slow pole. The following script runs simulations with initial conditions for the rotor speed $\omega(t)$ and coil current i ranging between -1 and 1 and plots the results together in a phase plot; that is, a graph of $\omega(t)$ versus $i(t)$. Time is not explicitly marked in a phase plot, so a circle is placed on the trajectory when 5% of the total trajectory time has passed.

```
PlantRPM = MotorSS;
PlantRPM.c = [0, 1, 0];
PlantRPM = minreal(PlantRPM);
set(PlantRPM, 'OutputName', '\omega')
t = linspace(0, 0.05, 500);
```

```
hold on
for i = 0:13
   x0 = [-1.4+0.2*i, 1];
   [y, t, x] = initial(PlantRPM, x0, t);
   plot(x(:,2), x(:,1), 'k-')
   k = floor(0.05*length(t));
   plot(x(k,2),x(k,1),'ro')
end
for i = 0:13
   x0 = [-1+0.2*i, -1];
   [y, t, x] = initial(PlantRPM, x0, t);
   plot(x(:,2), x(:,1), 'k-')
   k = floor(0.05*length(t));
   plot(x(k,2), x(k,1), 'ko')
end
plot([1, -1], [0, 0], 'k--')
plot([0 0], [,1 -1], 'k--')
axis([-1, 1, -1, 1])
ylabel('i(t) (A)')
xlabel('\omega(t) (radians/s)')
```

The resulting phase portrait of the system is shown in Figure 10.14. Note that the vast majority of the time for every trajectory is spent about a one-dimensional subspace. The subspace is associated with the slow pole. In general, those poles with the larger real part will dominate the step response of a system.

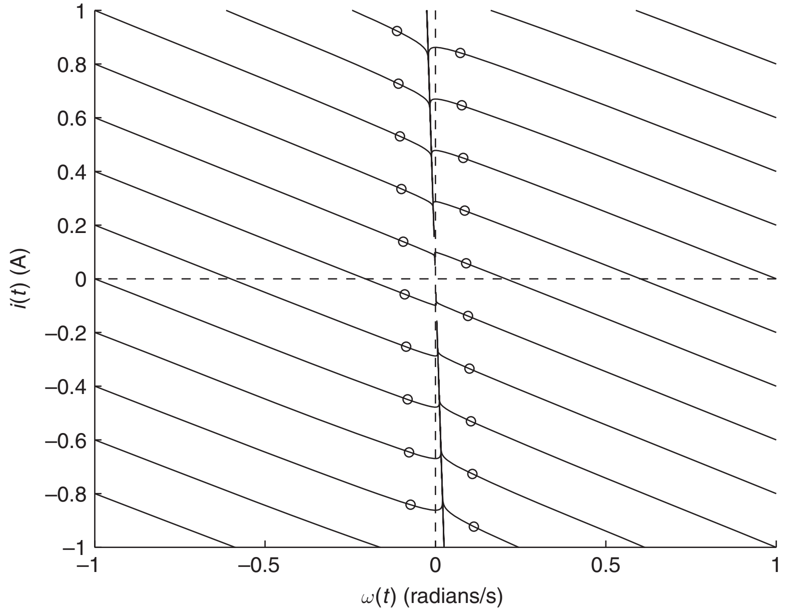


Figure 10.14 Phase portrait of a DC electric motor.

Example 10.8 Effects of zeros near poles of a second-order system

Consider the second-order plant:

$$H(s) = \frac{-(s - z)}{z(s^2 + 0.5s + 1)}$$

where z is the location of the zero. Although zeros cannot be moved by feedback, the effect of their positions can be profound if they are near poles or near the imaginary axis. To show this, we examine the step response of a family of plants with a zero approaching and crossing the imaginary axis. We select four values of z: ±5 and ±1. The script is as follows:

```
t = linspace(0, 25, 200);
Z = [-5, -1, 1, 5];
Den = [1, 0.5, 1];
y = zeros(length(t), length(Z));
for i = 1:length(Z)
   y(:,i) = step(-1/Z(i)*tf([1, -Z(i)], Den), t);
end
plot(t, y(:,1), 'k-')
hold on
plot(t, y(:,2), 'k--')
plot(t, y(:,3), 'k-o')
plot(t, y(:,4), 'k-+')
```

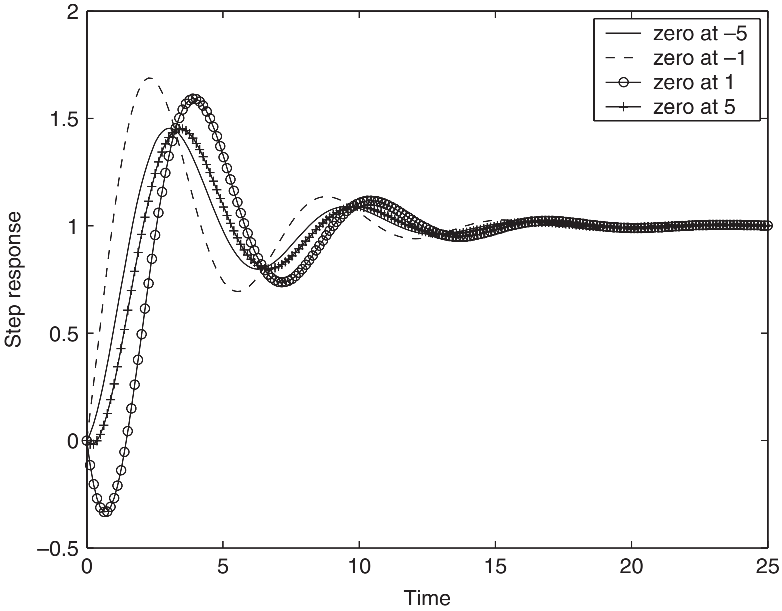


Figure 10.15 Effect of a zero approaching and crossing the imaginary axis on the step response.

```
legend('zero at -5', 'zero at -1', 'zero at 1', 'zero at 5')
xlabel('Time')
ylabel('Step response')
```

Executing this script results in Figure 10.15. The zeros are stable at $z = -5$ and $z = -1$; however, for the cases where $z = 1$ and $z = 5$, the zero is unstable. Although unstable zeros do not destabilize a system, they limit the amount of feedback that can be applied. The hallmark of an unstable zero is the system's tendency to go the wrong way initially, as seen with the plot with the zero at 1. A system with one or more unstable zeros is called nonminimum phase.

Example 10.9 Masking of modal dynamics

Consider the first- through fourth-order systems given by

$$G_1 = \frac{1}{s+1} \qquad G_3 = G_1(s)*G_2(s)$$

$$G_2 = \frac{100}{s^2 + 10s + 100} \qquad G_4 = \frac{19.8s + 20}{s + 20}G_3(s)$$

The step response of the plant G_3 is very similar to the response of the first-order plant G_1, because the pole at -1 dominates the complex poles, as discussed previously. However, if a zero were near the pole -1, then that pole would no longer dominate. The system given by G_4 places a zero near the dominant pole at -1 in G_3, thereby

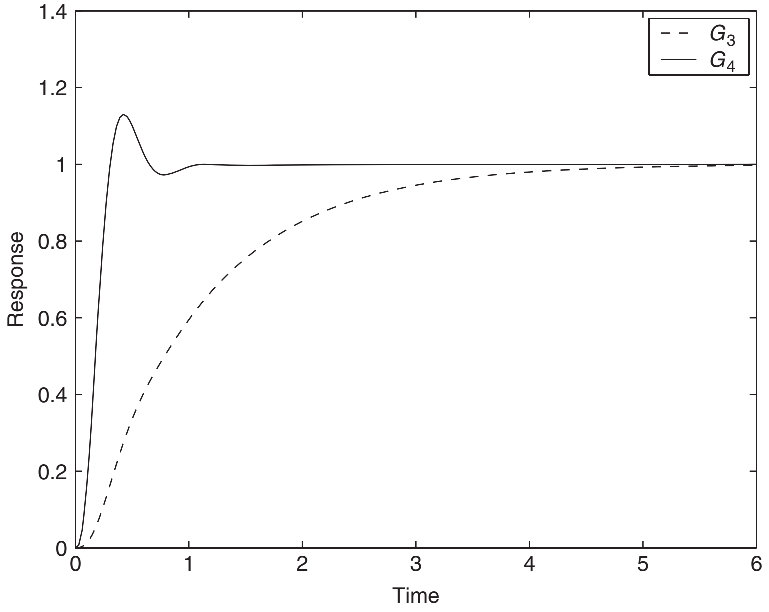


Figure 10.16 Effect of hiding a system's slow dynamics with a zero.

masking its effect. However, this is not generally practical to do, since in order to force G_3 to move quickly, G_4 will initially produce a large output, which may saturate or damage the actuators. This can be observed in the step response of G_4. The step responses of G_3 and G_4 are obtained with the following script and the results are shown in Figure 10.16.

```
G1 = tf([1], [1, 1]);
G2 = tf([100], [1, 10, 100]);
G3 = G1*G2;
G4 = tf([19.8, 20], [1, 20])*G3;
t = linspace(0, 6, 200);
yG3 = step(G3, t);
yG4 = step(G4, t);
plot(t, yG3, 'k--', t, yG4, 'k-')
legend('G3', 'G4')
xlabel('Time')
ylabel('Response')
```

The distance between a zero and a pole measures how perpendicular the input or output matrices are to the mode eigenvector. If a zero is directly on top of a pole, then the mode is either not excitable or not seeable in the output. The terms controllability and observability are also used to describe these phenomena.

10.3.3 Estimating Systems from Response

While some of the parameters used for the servomotor model of Eq. (10.1) can be found in data sheets, some, such as the friction constant b, might be difficult to find or be unavailable. Fortunately, one can obtain some of the parameters directly. A multimeter can be used to determine the values of L and R. Experiments can be run by applying different voltages $v(t)$ to the motor windings and observing what happens to the output angle $\theta(t)$. Estimating the system transfer function from the input and output responses is known as system identification. MATLAB provides a set of tools for this type of analysis in the System Identification toolbox, which is accessed with `ident`. To introduce this topic, we focus on parametric identification using an autoregressive model with external input (ARX).

We shall create a script that illustrates the techniques as follows. Experimentally obtained data are simulated using the motor model example system under feedback control. We generate a random input for the discrete-time version of the system and determine the response. The experiment generates a series of inputs $u[k]$ (commands) and the resulting responses are $y[k]$ (angles). The sampling interval is T_s. In practice, noise and unmeasured disturbances could contaminate the signals. The data are then collected into an identification data object using `iddata` from the System Identification toolbox and split into an identification data set and a validation data set. The results are plotted in Figure 10.17. Three different `arx` models of differing degrees are created and their ability to predict the response is shown in Figure 10.18. The ARX estimation function requires the user to supply the orders of the numerator and denominator polynomials, as well as a

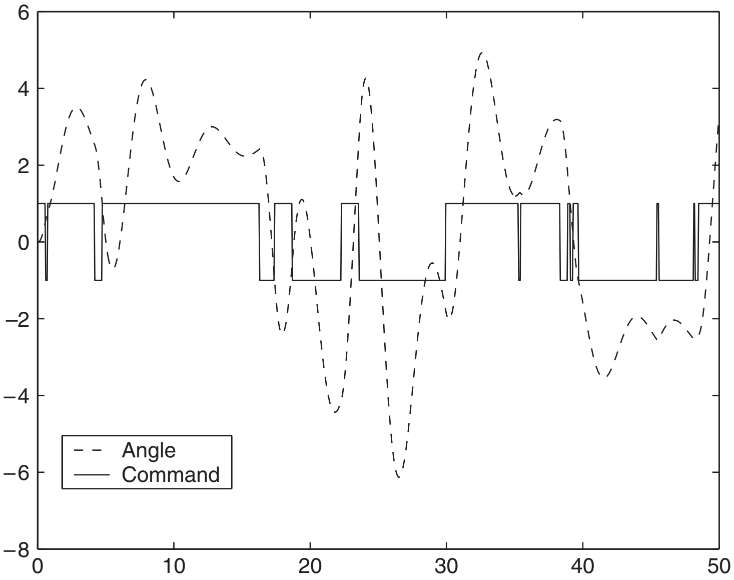


Figure 10.17 System identification data set. The *x*-axis is time.

net delay. Because we have simulated the data, we know in advance that the system has three poles; in practice, however, this is not certain. The command `detrend` is used to set the mean of the outputs and inputs to zero in order to satisfy linearity and initial condition assumptions of the ARX technique. The results

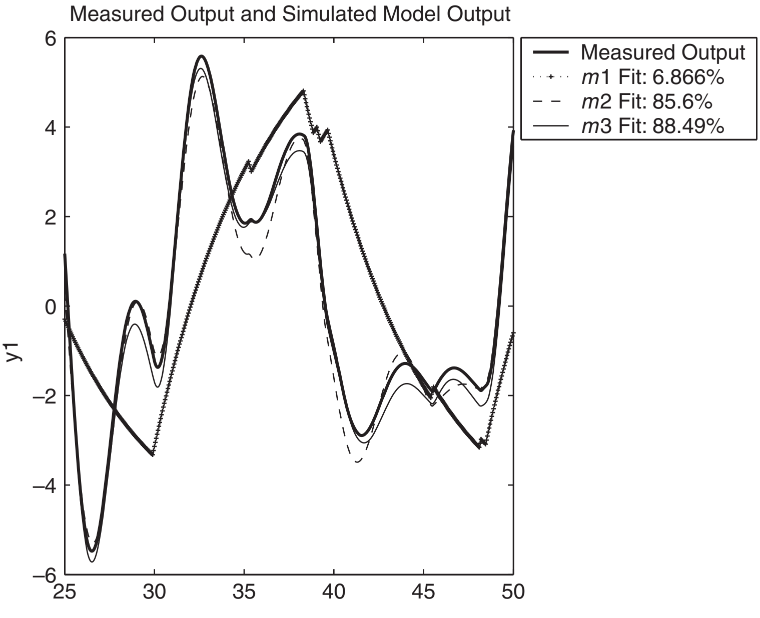


Figure 10.18 Comparison of the three system responses to the validation data set. The *x*-axis is time.

of the three models are then compared using the System Identification toolbox function compare.

```
Ts = 5e-2; N = 1000;
u = zeros(N, 1);
Jmodel = rand;
clSys = feedback(c2d(MotorSS(Jmodel), Ts), 0.4);
r = 1;
for i = 1:length(u)
  if rand < 2.5e-2
    r = -1*r;
  end
  u(i) = r;
end
t = Ts*(1:N);
y = lsim(clSys, u);
figure(1)
plot(t, y, 'k--', t, u, 'k-')
legend('Angle', 'Command', 'Location', 'SouthWest')
motorExp = iddata(y, u, Ts);                % Package data set
motorId = detrend(motorExp(1:N/2));         % Detrend 0–25 s
motorVal = detrend(motorExp(N/2:N));        % Detrend 25–50 s
m1 = arx(motorId, [1, 1, 0]);
m2 = arx(motorId, [2, 1, 0]);
m3 = arx(motorId, [3, 1, 0]);
figure(2)
compare(motorVal, m1, m2, m3)
```

It is seen that there is little improvement between the second-order model *m*2 and the third-order model *m*3. The system itself has three poles; however, two are dominant in determining the response and are, hence, first to be identified.

A number of parametric model identification techniques such as Box-Jenkins, Output Error, and Prediction Error are implemented in the MATLAB toolbox; each model differentiated by the model structure and its treatment of noise. The frequency response of systems can be estimated directly from experimental data by using nonparametric identification methods.

10.4 DESIGN TOOLS

In this section, we shall consider design tools in MATLAB and the criteria by which designs are evaluated. Many design techniques are graphical, as they were developed before computers were in general use. The graphical design tools include

bode—creates Bode plots
nyquist—creates Nyquist plots
rlocus—creates root locus plots

A matrix-based design tool employing lqr and lqe is also introduced.

Design criteria for control systems involve three requirements:

- Stability
- Transient response
- Steady-state response

The consequences of instability are clarified when solving the differential equations. Each root of the denominator polynomial corresponds to a component of the solution, and any root with a positive real part will contribute a term that grows exponentially. A system with a pole on the imaginary axis is labeled marginally stable. The other two categories of design criteria assume that the system is stable. Transient response requirements are measured by examining the short-term response of the system to a unit step input. Steady-state response requirements look at the long-term error in tracking either a step or ramp input and, on rare occasion, a more complex input such as a parabola. The stability criteria must always be met; some systems are initially unstable and must be stabilized. Examples of such systems include an inverted pendulum and a magnetic bearing. Many systems can be destabilized by the application of feedback. Stability of a closed-loop control system can easily be checked using `pole` on the closed-loop transfer function. Any roots with a positive real part indicate instability of the closed-loop system.

10.4.1 Design Criteria

We return to the DC motor to study design tools and consider the stability of a closed-loop system with the DC motor as the plant. A proportional controller, which takes the error between the desired position and the actual position and multiplies it by its gain, is used. When using a proportional controller, one must choose the gain. The following script generates Figure 10.19, which is a plot of the real part of the right-most pole as a function of the proportional controller gain, which ranges from 1 to 200. For small gains, the system is stable. By the time the gain is greater than approximately 128, however, the response of the closed-loop system is unbounded, and the motor might be damaged.

```
L = 50;   gains = linspace(1, 200, L);
y = zeros(L,1);
for i = 1:L
   y(i) = max(real(pole(feedback(gains(i)*MotorSS, 1))));
end
plot(gains, y, 'k-', [0, 200], [0, 0], 'k--')
xlabel('Proportional controller gain')
ylabel('Real part of right-most pole')
```

The Bode plot in Figure 10.20 is generated by

```
bode(MotorSS)
```

One can also determine at which gain the system becomes unstable by using

```
[gm, pm, wgm, wpm] = margin(MotorSS)
```

which computes the gain margin (*gm*) and phase margin (*pm*) and the frequencies at which they occur—*wgm* and *wpm*, respectively. In this case, the gain margin (the

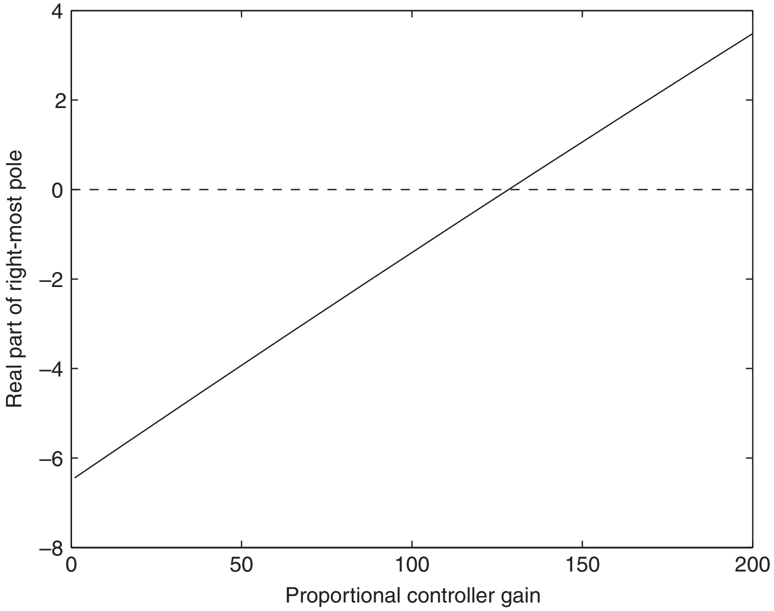


Figure 10.19 Real part of the right-most system pole of the closed-loop system as a function of controller gain.

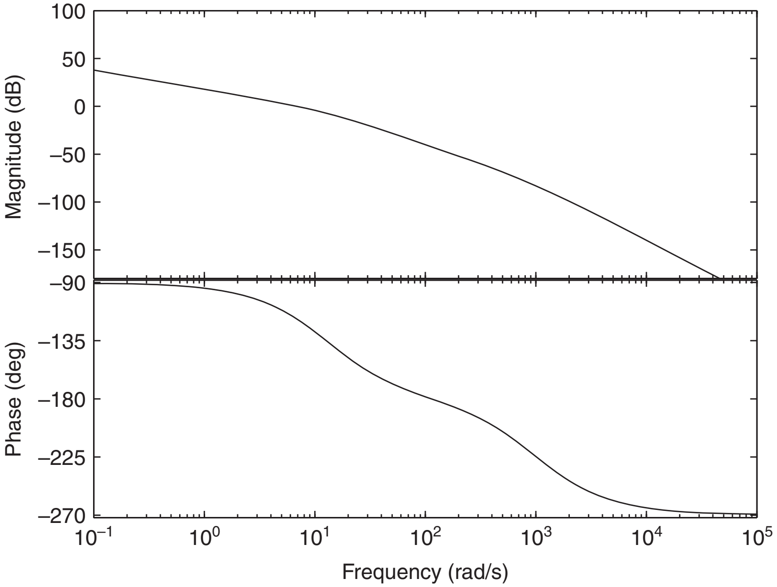


Figure 10.20 Bode plot indicating the crossover frequencies for the gain and phase margin.

amount of gain that may be applied before the system becomes unstable) is gm = 128.38 or 42.2 dB.

Even if the closed-loop system is stable, the behavior may not be acceptable. Consider the step response of the DC permanent magnet motor system for a range of stable gains, which is shown in Figure 10.21. Time ranges between 0 and 1 s, and four gains are chosen at equally spaced points between 1 and 10 on a logarithmic scale. The script to obtain Figure 10.21 is as follows:

```
t = linspace(0, 1, 100);
gains = logspace(0, 1, 4);
hold on
for i = 1:length(gains)
  [y, t] = step(feedback(gains(i)*MotorSS, 1));
  plot(t, y, 'k-')
end
xlabel('Time')
ylabel('Response')
text(0.16, 1.5, 'Gain = 10')
text(0.16, 0.4,'Gain = 1')
```

The response to commands quickens as the gain is increased, but the high-gain controllers overshoot the goal angle and tend to oscillate about the target position of one radian. There are many metrics by which these observations are quantified; here, we will consider rise time, overshoot, and settling time.

Transient requirements are measured from the response of the system to a step input. Rise time is the amount of time the system takes to go from 10% to 90%

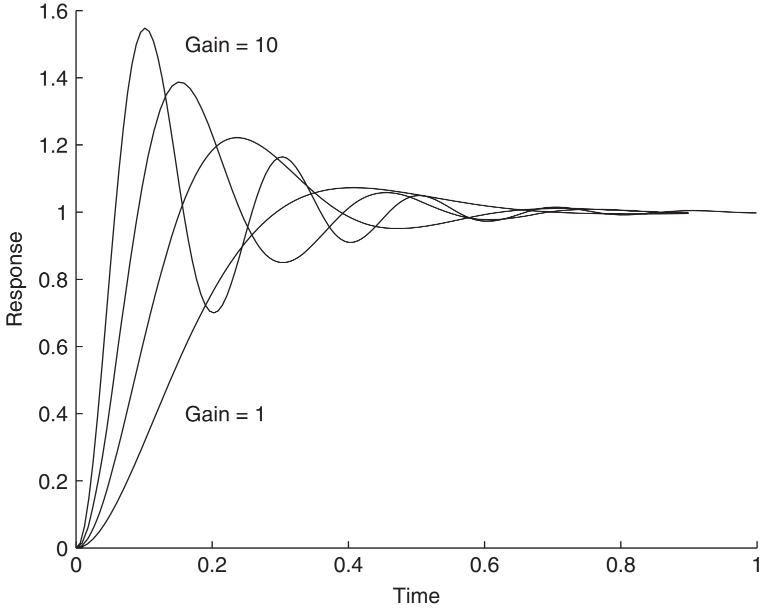


Figure 10.21 Step response of a motor system under proportional control.

of its final value. A related value is peak time, the time of the first maximum. In this particular system, rise time and peak time are in conflict with the next criteria, which is overshoot. Percentage overshoot is the amount by which the system overshoots its goal. Settling time is typically defined as the amount of time it takes for the system to come to and stay within a 2% envelope of the final value. These quantities are calculated in the function M file **transient** given below. The function returns $[-1, -1, -1]$ if the system is not stable.

```
function criteria = transient(system)
criteria = [-1, -1, -1];
maxP = max(real(pole(system)));
if maxP>= 0
  return
end
MaxTime = -6*(1/maxP);
Time = linspace(0, MaxTime, 500);
Response = step(system, Time);
[ResponseMax, IndexMax] = max(Response);
FinalValue = Response(end);
TimeLow = interp1(Response(1:IndexMax), Time (1:IndexMax), . . .
                        0.1*FinalValue);
TimeHigh = interp1(Response(1:IndexMax), Time (1:IndexMax), . . .
                        0.9*FinalValue);
criteria(1) = TimeHigh - TimeLow;
k = length(Time);
while (k>0) && (0.02 > abs((FinalValue - Response(k))/FinalValue))
  k = k-1;
end
criteria(2) = Time(k);
criteria(3) = 100*(max(Response) - FinalValue)/FinalValue;
```

where *criteria*(1) = rise time, *criteria*(2) = settling time, and *criteria*(3) = percentage overshoot.

We use this function to evaluate controllers for the DC motor. Thus,

```
v = transient(feedback(MotorSS, 1))
```

displays the vector $v = [0.1935, 0.6080, 7.6136]$, where 0.1935 is the rise time, 0.6080 the settling time, and 7.6136 is the percentage overshoot.

10.4.2 Design Tools

For the design criteria discussed in Section 10.4.1, we introduce a collection of design tools and illustrate their application to the motor controller. Typically, one of three different tools will be applied:

- Frequency based
- Root locus
- LQG based

Frequency-based design does not require an explicit model, only the results from a collection of experiments. The last two methods require very good models of the plant.

In the course of illustrating the use of these design tools, the controllers designed will not include a proportional-integral-derivative (PID) controller. The PID algorithm is typically applied without any analysis. Tuning a control loop without analysis can take a great deal of time and very often results in mediocre performance. Such an approach does not take advantage of the computational tools available in, say, MATLAB. The PID algorithm is an excellent general-purpose controller, however, and will be reviewed in the examples section.

Example 10.10 Controller design to meet rise time and percentage overshoot criteria

We again consider the motor controller. The design criteria require that we keep the overshoot under 20%; thus, many of the design gains shown in Figure 10.21 are unacceptable. Furthermore, we want the closed-loop system to be very quick, having a rise time under 0.05 s. Using a straight proportional controller, we see that we are in a deadlocked situation. To obtain a rise time under 0.05 s, a gain greater than 3 must be used. To have an overshoot under 20%, we must use a gain smaller than 2. The following script generates Figure 10.22, which graphs the overshoot and rise time as functions of gain for the proportional controller.

```
L = 20;   kp = 0.4*logspace(0, 1, L);
result = zeros(L,3);
for i = 1:L
  result(i,:) = transient(feedback(kp(i)*MotorSS, 1));
end
[ax, h1, h2] = plotyy(kp, result(:,3), kp, result(:,1));
```

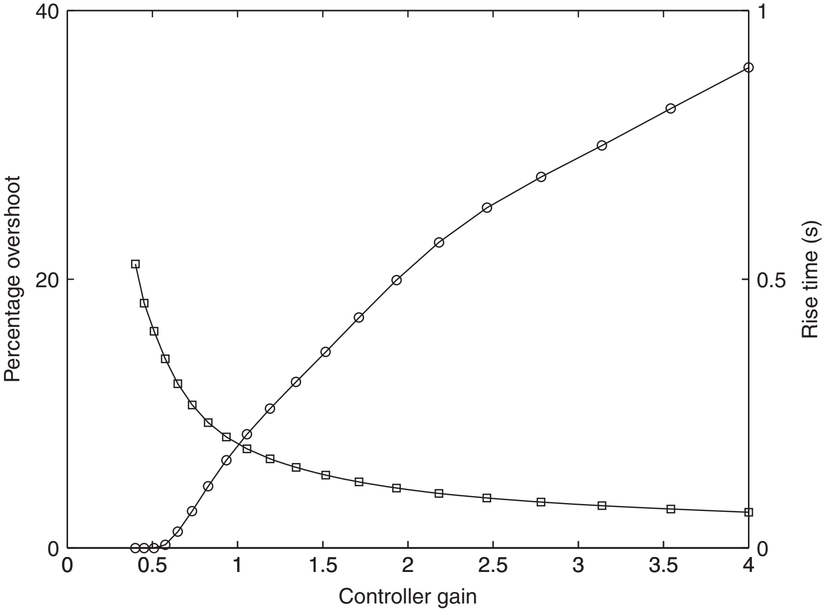


Figure 10.22 Percentage overshoot (circles) and rise time (squares) as a function of controller gain.

```
xlabel('Controller gain')
ylabel('Percentage overshoot')
set(get(ax(2), 'Ylabel'), 'String', 'Rise time (s)')
set(h2, 'Marker', 's')
set(h1 , 'Marker', 'o')
```

Frequency Based Design

First, we attempt a frequency-based design. The overshoot requirement can be translated into a phase-margin minimum. For this system, a phase margin of 45° is needed to meet the overshoot requirements. Looking at the Bode plot in Figure 10.20, we see that at a phase of $-135°$ (= 45° phase margin), a gain of approximately 2.0 is allowed. In order to compute the gain more precisely, we use `fzero` with **transient**. The script is

```
function Gain
gain = fzero(@PEcontrol, 2)
transresp = transient(feedback(gain*MotorSS, 1))

function s = PEcontrol(gain)
rval = transient(feedback(gain*MotorSS, 1));
s = rval(3)-20;
```

Executing this program, we obtain $gain = 1.9384$ for a 20% overshoot and that $transresp(1) = \text{rise time} = 0.1111\text{s}$, more than twice as slow as the design specification.

A lead controller is typically used to improve the transient response of a system. A properly designed lead controller increases the phase for a short range of frequencies; this boost in phase allows more gain to be applied. The zero of the lead controller is chosen to be at -15, just to the left of the second open-loop pole at -13. This ensures that the lead controller's phase boost starts about where the phase of the DC motor starts to roll off. The lead controller pole is at -100, nearly ten times the zero location. In theory, the further to the left, the better, but having the pole very far to the left makes the controller sensitive to noise. As a rule of thumb, the pole should not be located further left than ten times the location of the zero. The resulting controller transfer function is

$$C(s) = \frac{100}{15}\frac{s + 15}{s + 100} = \frac{6.667s + 100}{s + 100}$$

where we have multiplied the transfer function by 100/15 to set the DC gain to 1.

The following script generates Figure 10.23, which compares the frequency response of the uncompensated and compensated systems.

```
Control = tf([6.667 100], [1 100]);
bode(MotorSS, 'k-', Control*MotorSS, 'k--')
```

Note how the phase roll-off is delayed to higher frequencies. Again, we use the Bode plot to find an initial guess for the correct feedback gain, approximately 10. To compute the exact value of the gain at which an overshoot of 20% is reached and the corresponding rise time, we use the following program:

```
function Gain2
gain = fzero(@LDcontrol, 10)
transresp = transient(feedback(13.0108*tf([6.667, 100], [1, 100])*MotorSS, 1))
```

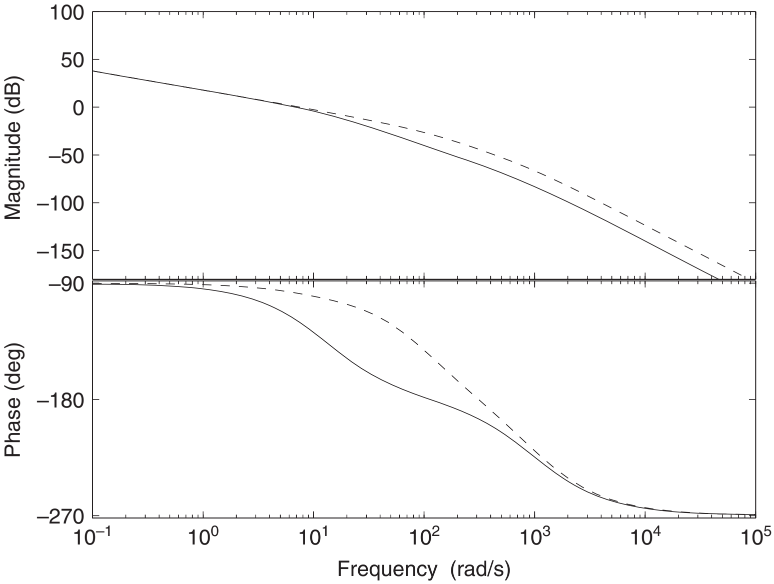


Figure 10.23 Bode plot of the lead compensated system (dashed line) and that of the uncompensated system (solid line).

```
function s = LDcontrol(gain)
rval = transient(feedback(gain*tf([6.667, 100], [1, 100])*MotorSS, 1));
s = rval(3)-20;
```

Upon execution, we find that *gain* = 13.01 for a 20% overshoot and that *transresp*(1) = rise time = 0.017 s, nearly three times faster than the target value. The graphs of the step responses for both the proportional and lead-controlled systems are shown subsequently in Figure 10.26.

Using the lead compensator, one is able to meet both the overshoot and the rise-time requirements. Frequency-based design requires data from a Bode plot, but does not depend on an explicit model. The other frequency-based tools include the Nyquist plot and the Nichols plot, which are used in a similar manner.

Root Locus Based Dseign

The root locus is another commonly applied tool. The same lead compensator may be applied, but the approach differs. Given that the complex poles subtend an angle ξ from the imaginary axis and have radius ω_n, we have[5]

$$M_p = \exp\left(-\frac{\pi\xi}{\sqrt{1 - \xi^2}}\right)$$

$$T_r \approx \frac{1}{\omega_n}(1 + 1.4\xi)$$

[5] Anand and Zmood, *Control Systems*, 1995.

where M_p is the peak magnitude and T_r is the rise time. Inverting these formulas constrains where the closed-loop poles may be located in the complex plane in order to meet the transient design requirements. Hence, an overshoot requirement of less than 20% constrains ξ, requiring that the dominant poles be located within a 120° wedge centered along the negative imaginary axis. A 0.05 s rise time corresponds roughly to a minimum pole radius ω_n of 20. These formulas are rules of thumb, but they serve as a good starting point. In the following script, we shade the region of the complex plane in which all closed-loop poles must lie. Then, we plot the root locus to see if this condition is met at any gain.

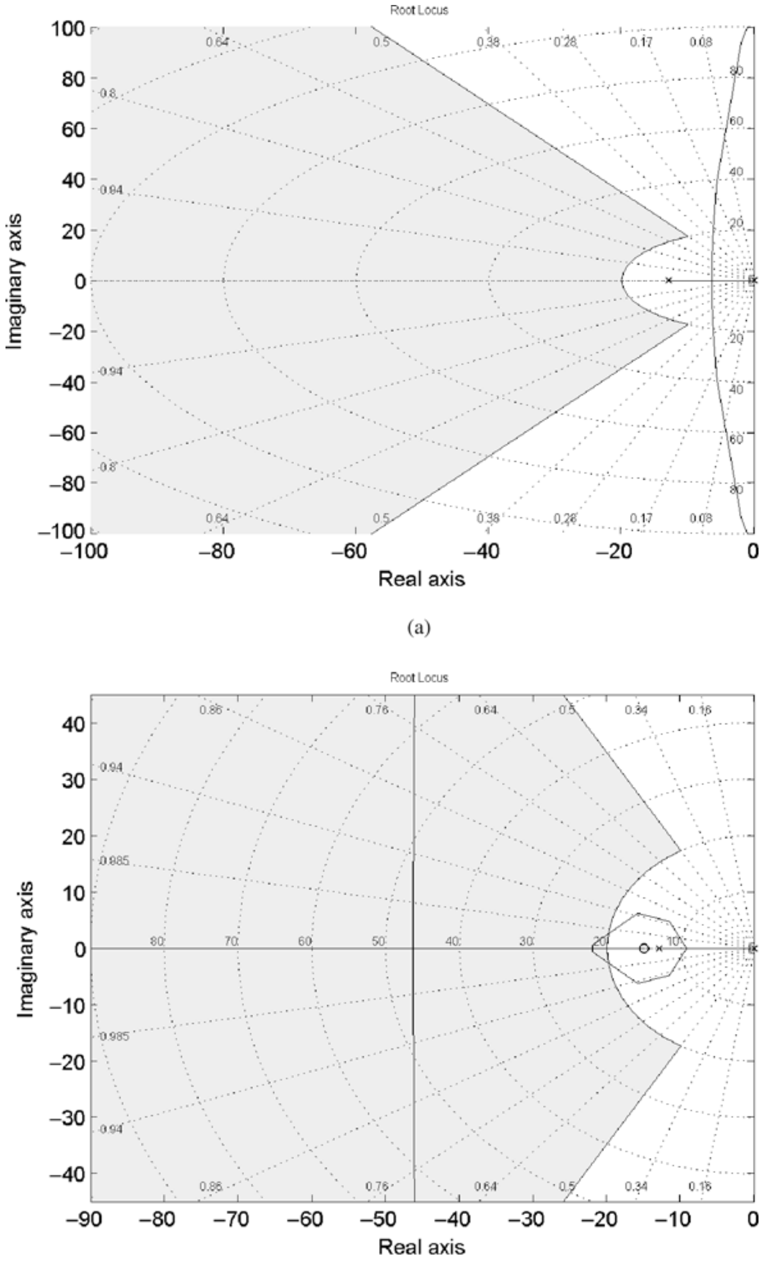
```
theta = linspace(-2/3*pi, -4/3*pi, 15);
X = [20*cos(theta), 200*cos(-4/3*pi), 200*cos(-2/3*pi), 20*cos(-2/3*pi)];
Y = [20*sin(theta), 200*sin(-4/3*pi), 200*sin(-2/3*pi), 20*sin(-2/3*pi)];
hold on
h = fill(X, Y, 'c');
alpha(h, 0.2)
sgrid
rlocus(MotorSS)
axis(100*[-1, 0, -1, 1])
ylabel('Imaginary axis')
xlabel('Real axis')
```

Executing this program gives the results shown in Figure 10.24a. Notice that the controller and plant, **MotorSS**, have had their order switched. In theory, this produces the same root-locus plot. However, when the root-locus command with the lead controller precedes the plant, the calculations run much slower and may cause some computers to lock. This is due to the controller's zero being near the plant pole. The transformation to controller form, which simplifies the computation of the closed-loop poles, is nearly singular when the lead controller precedes the plant; that is, the cascade is nearly uncontrollable. By switching the order, we make the system nearly unobservable and do not compromise the transformation.

A similar plot can be generated for the lead-controlled system using the following script:

```
theta = linspace(-2/3*pi, -4/3*pi, 15);
X = [20*cos(theta), 200*cos(-4/3*pi), 200*cos(-2/3*pi), 20*cos(-2/3*pi)];
Y = [20*sin(theta), 200*sin(-4/3*pi), 200*sin(-2/3*pi), 20*sin(-2/3*pi)];
hold on
h = fill(X, Y, 'c');
alpha(h, 0.2)
sgrid
rlocus(MotorSS*tf([6.667, 100] ,[1, 100]))
axis(90*[-1, 0, -0.5, 0.5])
ylabel('Imaginary axis')
xlabel('Real axis')
```

Executing the program gives the results shown in Figure 10.24b. Notice that for the proportional controller, there is no gain where all of the closed-loop roots are located



(a)

(b)

Figure 10.24 Root-locus plots of the motor positioning system: (a) proportional controlled, and (b) lead controlled

in the acceptable region. The net effect of the lead controller is to bend the root-locus lines back into the acceptable region.

To find the correct gain, we use

```
rlocus(MotorSS*tf([6.667, 100], [1 100]))
[k, p] = rlocfind(MotorSS*tf([6.667, 100], [1 100]))
```

and place the crosshairs where the root locus crosses the edge of the acceptable region. The function `rlocus` precedes `rlocfind` because `rlocfind` does not draw the root locus. This procedure yields a gain of 12. These regions are approximate, so it is a good practice to fine-tune the gain by simulation. Thus, from a simulation, it will be found that a gain as high as 13 can be applied.

LQG Based Design

In the root-locus design, the objective is to place the poles of the closed-loop system into the acceptable region. With the linear-algebra tools developed for state–space models, such a problem can be solved by directly placing the poles in the desired locations. The two methods introduced are pole placement with

```
place
```

and

```
acker
```

and LQG design with

```
lqr
lqe
```

and

```
reg
```

In order to meet the design specifications, we choose the pole locations $-30, -20+30i, -20-30i$. These pole locations are arbitrarily chosen, but are safely inside the acceptable shaded region of Figures 10.24a and b. The following script uses `place` to compute the gain matrix K so that the matrix $A - BK$ has eigenvalues in the desired locations. The quantities A and B are those given by Eq. (10.3).

```
DesiredPoles = [-30, -20+30*i, -20-30*i];
[A, B, C, D] = ssdata(MotorSS);
K = place(A, B, DesiredPoles)
```

Executing the script gives $K = [0.3900, -0.1002, -4.6517]$. The feedback needed is then $u = Kx$, which assumes that we have available the internal state of the system given by x. Since only the output is available, a state estimator needs to be designed. A complete script computing both the controller and the observer is as follows:

```
DesiredPoles = [-30, -20+30*i, -20-30*i];
[A, B, C, D] = ssdata(MotorSS);
K = place(A, B, DesiredPoles);
```

```
L = (place(A', C', 3*DesiredPoles))';
ControlSS = reg(MotorSS, K, L);
clSys = feedback(MotorSS, ControlSS, +1);
clSys = 1/dcgain(clSys)*clSys;
step(clSys)
```

The result of this script is plotted subsequently in Figure 10.26 as the state–space controller. The first three lines of the script generate the feedback gain matrix *K*. Using duality through the transpose between observability and controllability, `place` is used to compute the observer feedback gain matrix *L*. This matrix depends on the system matrices *A* and *C* and a set of desired observer pole locations, which we set to three times the feedback pole locations. The function `reg` then creates the estimator which, using the plant output, estimates the internal state and outputs the corrective command. Since the command signal has the correct sign, we employ positive feedback in `feedback.`

The pole locations in the previous example were chosen to satisfy the transient requirements. These requirements are inequalities in nature, so a range of pole locations is acceptable; we arbitrarily chose a set of pole locations within the acceptable region. The linear quadratic Gaussian controller design method follows similar steps, but offers the designer a systematic method for assigning pole locations. Poles are chosen to minimize the integral[6]

$$J = \int \left[x'(t) Q x(t) + u'(t) R u(t) \right] dt \tag{10.11}$$

where $x(t)$ is the internal system state at time t, $u(t)$ is the input vector at time t, Q is a positive semidefinite matrix, and R is a positive definite matrix. The matrix Q is often chosen as $C'C$, so that the first term reduces to the square of the output error.

The designer may adjust the relative importance of the state error to the input by modifying the relative magnitudes of Q and R. In our particular case, the plant has only one input, so R is a positive scalar. With $Q = C'C$, the following script plots the location of the optimal poles as a function of R and, in addition, the region in which the poles must lie to satisfy the transient design requirements.

```
[A, B, C, D] = ssdata(MotorSS);
L = 60;   clPoles = zeros(L,3);
R = logspace(-4, 1, L);
for i = 1:L
   [K, S, E] = lqr(A, B, C'*C, R(i));
   clPoles(i,:) = E;
end
theta = linspace(-2/3*pi, -4/3*pi, 15);
X = [20*cos(theta), 200*cos(-4/3*pi), 200*cos(-2/3*pi), 20*cos(-2/3*pi)];
Y = [20*sin(theta), 200*sin(-4/3*pi), 200*sin(-2/3*pi), 20*sin(-2/3*pi)];
h = fill(X, Y, 'c');
alpha(h, 0.2)
```

[6] Kailith, *Linear Systems Theory*, 1980.

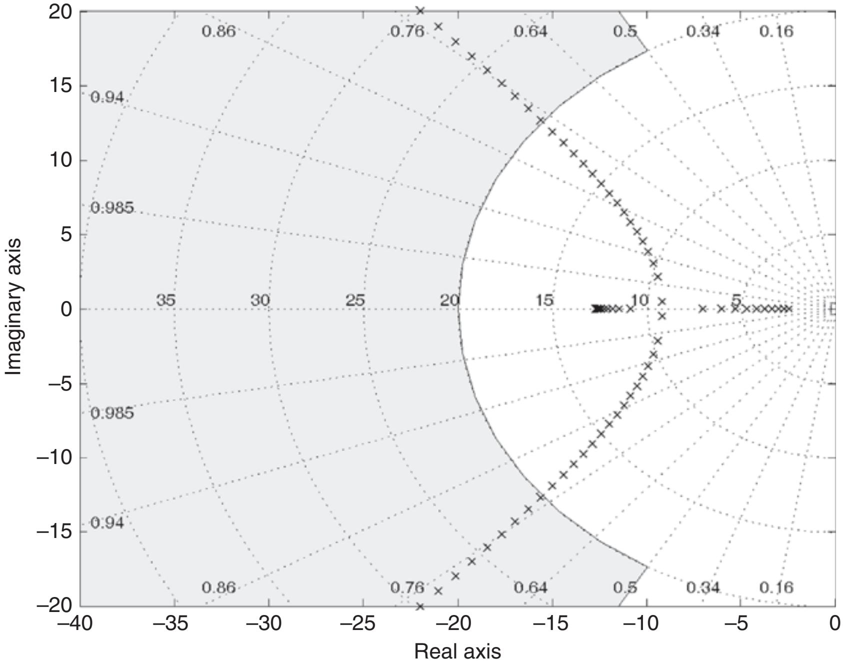


Figure 10.25 Closed-loop poles of the optimal controller as a function of the input cost weight *R*.

```
hold on
plot(real(clPoles), imag(clPoles), 'kx')
axis(40*[-1, 0, -0.5, 0.5])
sgrid
ylabel('Imaginary axis')
xlabel('Real axis')
```

Executing this script results in Figure 10.25, where the discrete root locations have been plotted with an 'x'. For large values of *R*, the cost of the input is large relative to the cost of the output error and, consequently, little control effort is applied. The closed-loop poles are close to the open-loop poles. As the cost of the input is made less expensive, the optimal closed-loop poles move further into the left half of the complex plane. More control action is being applied, and the response of the system is much faster. Since we want to meet both an optimality condition and the transient requirements, we must select *R* so that the optimal closed-loop poles lie within the shaded region.

The step responses of the three major controller designs discussed so far are generated in the following script and are compared in Figure 10.26. The proportional controller fails to meet the rise-time design criteria. Using pole placement, both the lead controller and the state–space controller meet the design criteria. The state–space controller has the additional benefit of having almost no overshoot.

```
t = linspace(0, 1, 200);
yp = step(feedback(1.9416*MotorSS, 1), t);
```

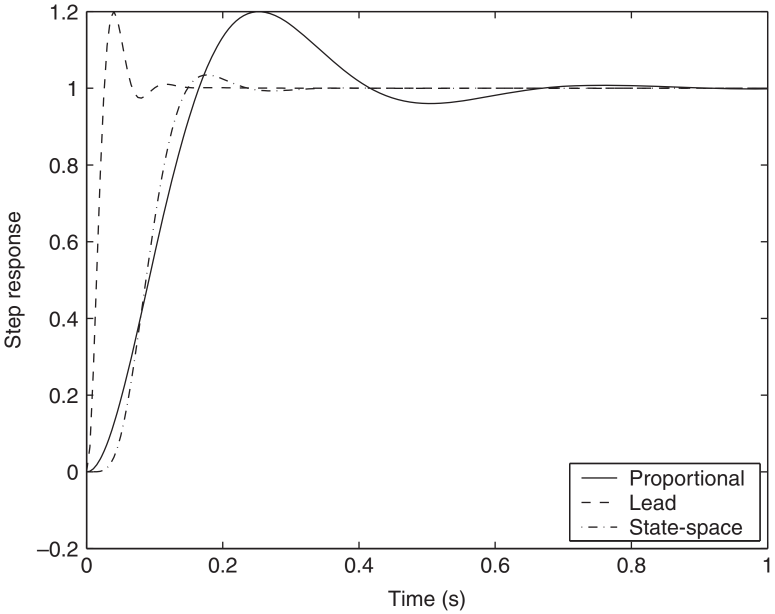


Figure 10.26 Comparison of a proportional, lead, and state–space controller to a step input.

```
yl = step(feedback(13.0108*tf([6.667, 100], [1 100])*MotorSS, 1), t);
DesiredPoles = [-30, -20+30*i, -20-30*i];
[A, B, C, D] = ssdata(MotorSS);
K = place(A, B, DesiredPoles);
L = (place(A', C', 3*DesiredPoles))';
ControlSS = reg(MotorSS, K, L);
clSys = feedback(MotorSS, ControlSS, +1);
clSys = clSys/dcgain(clSys);
ys = step(clSys, t);
plot(t, yp, 'k-', t, yl, 'k--', t, ys, 'k-.')
xlabel('Time (s)')
ylabel('Step response')
legend('Proportional', 'Lead', 'State-space', 'Location', 'SouthEast')
```

10.5 DESIGN EXAMPLES

In the previous section, we discussed several design approaches. In this section, we shall apply these techniques to four different physical systems:

1. *DC motor with flexible shaft*—design a notch controller in order not to excite the flexible shaft's vibration mode.
2. *Single-axis magnetic suspension system*—design a PID controller to keep the mass positioned at its equilibrium location.

3. *Inverted pendulum*—design multi-input single-output controller to keep a pendulum vertical.
4. *Magnetically suspended flywheel*—design a multi-input, multi-output controller to keep a flywheel suspended.

There are four steps in the controller design process:

1. Specify the controller requirements.
2. Develop a model of the plant.
3. Design the controller to meet the requirements.
4. Simulate and test the controller design.

Plant models in the following sections are derived from first principles, but are not tested against experimental data. It cannot be overemphasized that validating and refining the model is a step that is of great importance in controller design and one that must not be ignored. The latter three plant models described above are nonlinear, but only slightly; each can be linearized about an operating point, which is stabilized. Frequency-based design using open-loop data from Bode plots is possible only for the DC motor with flexible shaft, since all the other systems are open-loop unstable.

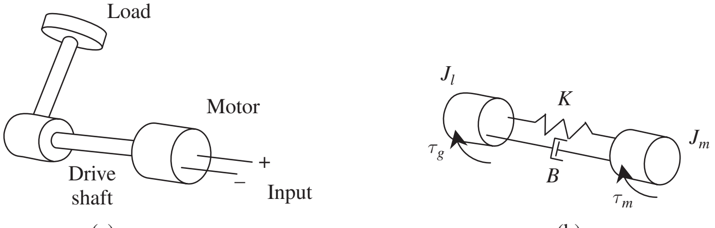
Controllers used in the subsequent sections include

- Lead (Lag)
- Notch
- PID
- LQG

With each of these methods, the root locus is the primary design tool. In practice, the PID controller is by far the most common type of controller used for single-input, single-output control systems.

10.5.1 Notch Control of a Flexible Pointer

Consider the read–write head on a hard-disk drive. The objective is for the head to move as fast as possible to a desired location, and once there, to provide a steady platform for the read or write operation. With limited actuation, typically a voice coil, the way to go faster is to remove material from the swing arm holding the read–write head. Removing material tends to reduce the stiffness of the arm, and hence moving quickly will more likely excite the vibration modes of the arm.



(a) (b)

Figure 10.27 (a) Pointer with a flexible drive shaft, and (b) its equivalent model.

As a model of this design, we consider the DC motor mounted with a flexible shaft, as shown in Figure 10.27. The user specifies a desired angle θ_d and the control system, measuring the angle of inclination of the rod θ, attempts to match the command in as quick a manner as possible. However, if the pointer gets to the desired position and then oscillates for a long time, the head will not function properly.

The equations of motion for the flexible pointer are[7]

$$L\frac{di(t)}{dt} + k_b\frac{d\theta(t)}{dt} + Ri = v(t)$$

$$J_m\frac{d^2\theta(t)}{dt^2} - k_\tau i(t) = b\left(\frac{d\phi(t)}{dt} - \frac{d\theta(t)}{dt}\right) + k\left[\phi(t) - \theta(t)\right] \tag{10.12}$$

$$J_l\frac{d^2\phi(t)}{dt^2} = -b\left(\frac{d\phi(t)}{dt} - \frac{d\theta(t)}{dt}\right) - k\left[\phi(t) - \theta(t)\right]$$

where θ is the angle of the rotor, ϕ is the orientation of the pointer, $J_m = 0.03\ \text{kg}\cdot\text{m}^2$ is the rotor inertia, $J_l = 0.015\ \text{kg}\cdot\text{m}^2$ is the load inertia, $b = 0.01$ Nm/rad/s is the flexible shaft damping, and $k = 10$ Nm/rad is the flexible shaft spring constant. The values for the inductance L, resistance R, back emf constant k_b, and the motor torque constant k_τ are those defined in Example 10.1.

The coupled second-order differential equations can be converted to a set of linear coupled first-order equations by introducing the state variables

$$x_1 = \theta \qquad x_4 = \phi$$

$$x_2 = \frac{d\theta}{dt} \qquad x_5 = \frac{d\phi}{dt}$$

$$x_3 = i$$

Then, Eq. (10.12) becomes

$$\frac{dx_1}{dt} = x_2$$

$$\frac{dx_2}{dt} = -\frac{k}{J_m}x_1 - \frac{b}{J_m}x_2 + \frac{k_2}{J_m}x_3 + \frac{k}{J_m}x_4 + \frac{b}{J_m}x_5$$

$$\frac{dx_3}{dt} = -\frac{k_b}{L}x_2 - \frac{R}{L}x_3 + v$$

$$\frac{dx_4}{dt} = x_5$$

$$\frac{dx_5}{dt} = \frac{k}{J_1}x_1 + \frac{b}{J_1}x_2 - \frac{k_2}{J_1}x_4 - \frac{b}{J_1}x_5$$

[7] R. C. Dorf and R. H. Bishop, *Modern Control Systems*, Addison-Wesley, Reading, MA, 1998.

The following function M file **Pointer** computes the system matrices for the coupled first-order equations and returns a state–space system object model.

```
function Plant = Pointer
L = 5e-3;   R = 5;   kb = 0.125;
ki = 15;   Jm = 3e-2;
Jl = 0.5*Jm;   k = 10;   b = 0.01;
A = [0, 1, 0, 0, 0;
      -k/Jm, -b/Jm, ki/Jm, k/Jm, b/Jm;
      0, -kb/L, -R/L, 0, 0;
      0, 0, 0, 0, 1;
      k/Jl, b/Jl, 0, -k/Jl, -b/Jl];
B = [0; 0; 1/L; 0; 0];
C = [0, 0, 0, 1, 0];
D = 0;
Plant = ss(A, B, C, D);
```

The output of the system is the angular position of the pointer $\phi(t)$. Typing

```
pole(Pointer)
```

in the command window gives

```
 1.0e+002 *
-9.8734
-0.0248 + 0.3104i
-0.0248 - 0.3104i
 0.0000
-0.0871
```

which shows that the flexible pointer system has five poles, three of which are on the real axis: one at the origin, one at nearly -1000 rad/s due to the motor coil electronics, and a pole at -8.7 rad/s due to the rotor dynamics. The flexible attachment adds a pair of complex poles at $-2.5 \pm 31.0i$. As before, a lead controller could be applied to improve the transient response; however, the poorly damped poles will frustrate this approach.

To illustrate the limitation of lead control for this system, we shall generate the root locus of the proportional and lead-controlled systems. We place the lead zero at -6, which is just to the right of the first stable pole. The lead pole is placed at -50, nearly ten times the location of the zero. Then the transfer function is

$$H(s) = \frac{s + 6}{s + 50}$$

The script is as follows:

```
rlocus(Pointer);
figure(1)
axis(70*[-1, 1, -0.5, 0.5])
sgrid
```

```
xlabel('Real axis')
ylabel('Imaginary axis')
figure(2)
rlocus(tf([1, 6], [1, 50])*Pointer)
axis(70*[-1, 1, -0.5, 0.5])
sgrid
xlabel('Real axis')
ylabel('Imaginary axis')
```

The result of executing this script is shown in Figure 10.28. The allowable gain for both designs is limited not by the real-axis open-loop poles, as it was in the case of the DC motor, but by the complex poles due to the flexible shaft. The performance of the lead controller will be only a little better than the performance of the proportional controller. By entering

```
rlocus(tf([1, 6], [1, 50])*Pointer)
[k, p] = rlocfind(tf([1, 6], [1, 50])*Pointer)
```

in the command window, we can pick the best gains for the lead-controlled system and, similarly, the proportional-controlled system. The crosshairs must be placed near the complex poles that are due to the flexible shaft, because they primarily limit the gain. A gain of 3 for each is stable and has fair performance, as shown subsequently by their closed-loop step responses in Figure 10.30. The undamped complex poles block further performance improvement because standard controllers will unwittingly excite the flexible mode.

Not exciting the flexible mode is the key to further improving the closed-loop performance. Recall from Section 10.3.2 that if a zero happens to be near a pole, then that mode is difficult to excite. We will use a notch controller whose zeros are chosen close to the flexible mode locations at $-3 \pm 30i$. To keep the transfer strictly proper, we choose both notch poles at -60. There is no realistic hope of being directly on top of the poles and canceling them, for to be close requires a good model that has most likely been derived from a set of experiments on the system.

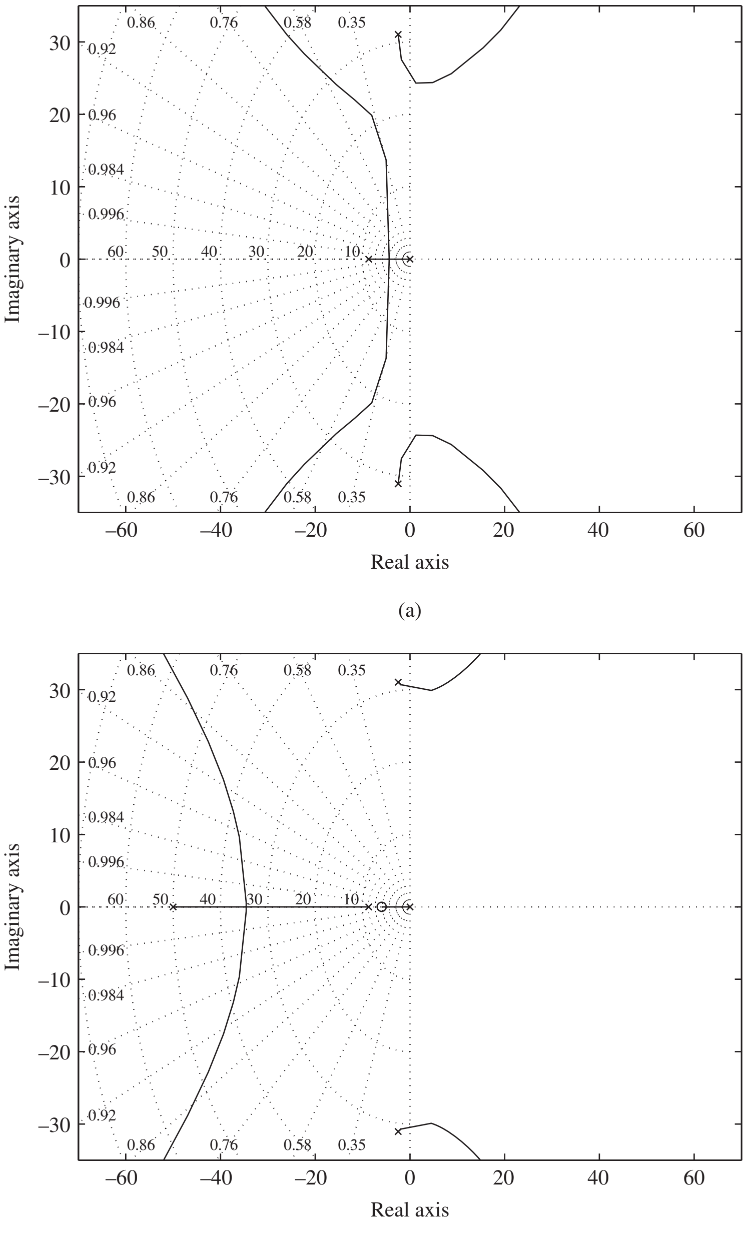
The following script plots the root locus of the notch-controlled system:

```
Notch = zpk([-3+30i, -3-30i], [-60, -60], 1);
rlocus(tf([1 6], [1 50])*Notch*Pointer)
axis(70*[-1, 1, -0.5, 0.5])
sgrid
xlabel('Real axis')
ylabel('Imaginary axis')
```

The result of executing this script is shown in Figure 10.29. The zeros are very close to the poles due to the flexible mode and the root-locus plot is very similar to a system without the flexible modes.

Using the following set of commands in the command window,

```
Notch = zpk([-3+30i, -3-30i], [-60, -60], 1);
rlocus(tf([1, 6], [1, 50])*Notch*Pointer)
rlocfind(tf([1, 6], [1, 50])*Notch*Pointer)
```



(a)

(b)

Figure 10.28 Root locus of the flexible pointer under (a) proportional control, and (b) lead control.

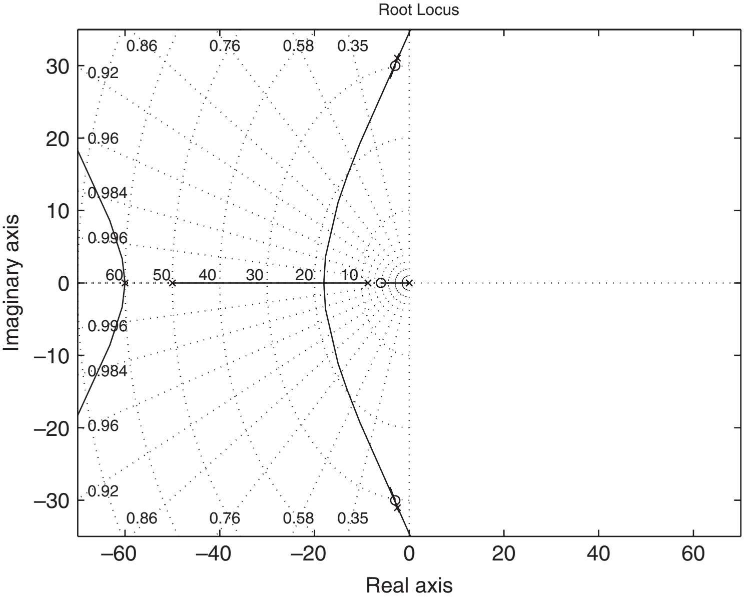


Figure 10.29 Root locus of the notch-lead compensated flexible pointer.

we find that a gain of 40.0 produces a stable closed-loop system. The crosshairs are placed along the root-locus lines that cross the imaginary axis. Placing them inside a 120° wedge centered about the negative real axis will yield good transient performance.

The following script computes the step response of all three types of control schemes previously described:

```
Lead = tf([1, 6], [1, 50]);
Notch = zpk([-3+30i, -3-30i], [-60, -60], 1);
t = linspace(0, 3, 200);
yp = step(feedback(3.0*Pointer, 1), t);
yl = step(feedback(3.0*Lead*Pointer, 1), t);
yn = step(feedback(40.0*Notch*Lead*Pointer, 1), t);
plot(t, yp, 'k--', t, yl, 'k-.', t, yn, 'k-')
legend('Proportional', 'Lead', 'Notch')
xlabel('Time')
ylabel('Step response')
```

The results of executing this script are shown in Figure 10.30. The notch controller has the rise time of the proportional controller, but it doesn't excite the flexible mode of the shaft.

The controllers resulting from the preceding design are analog; however, the final controller will most likely be implemented using an embedded controller, which is a small, inexpensive computer. The embedded controller, using perhaps an optical

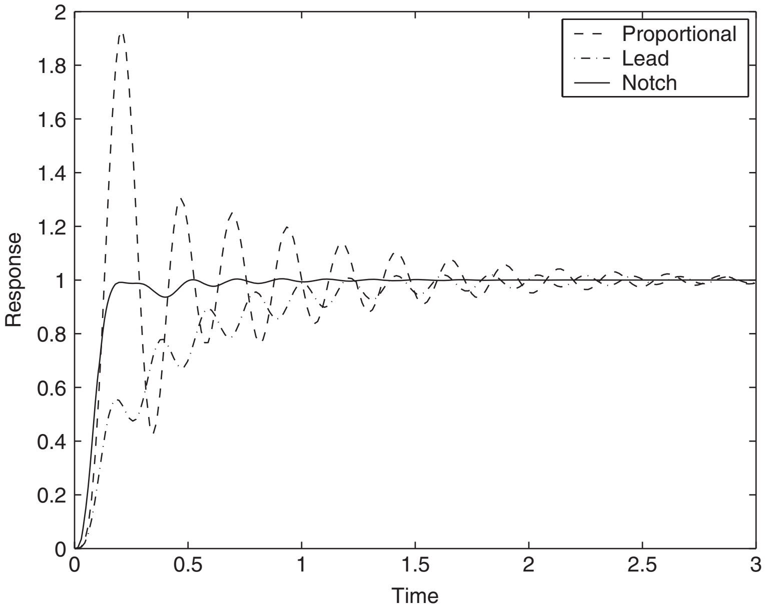


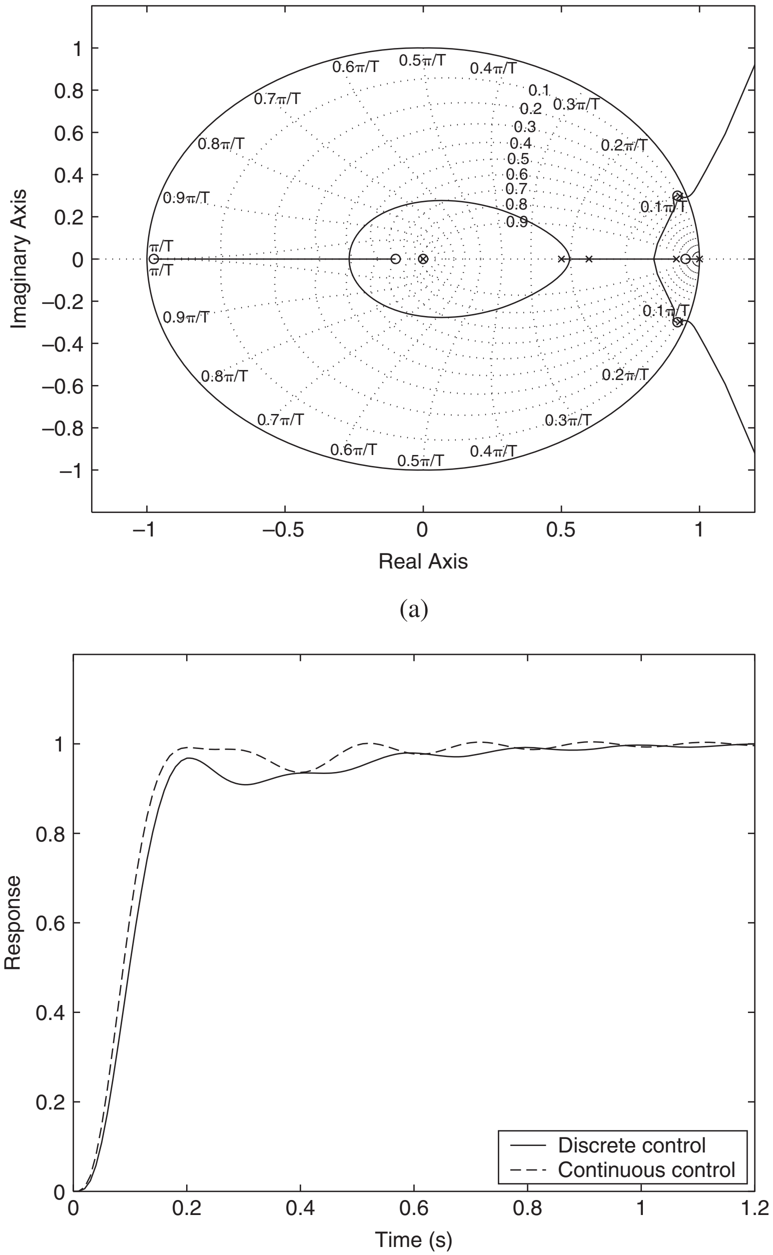
Figure 10.30 Comparison of the step responses of the proportional, lead, and notch controllers.

encoder, periodically measures the position of the pointer and compares to its desired position. After some computation, a digital-to-analog converter or a pulse-width modulator sets the effective amplifier voltage. In the following example, the computer reads the encoder and updates the output of the voltage amplifier 100 times a second. Because of the small sampling interval, the computation done at every sampling instance must be kept to a minimum. The following script designs the controller for the discretized version of the plant and then compares the digital design to that obtained from the previous continuous notch design. The zeros of the digital notch filter are set at $0.92 \pm 0.3i$, close to the poles of the discrete version of the plant, and the poles are placed at 0.6. A digital transfer function does not have to be strictly proper to be implemented. The zero of the digital lead is placed at 0.95, just to the right of the first stable pole of the discrete plant, the pole at 0.5.

Executing the following script

```
Ts = 0.01;
DNotch = zpk([0.92+0.3i, 0.92-0.3i], [0.6 0.6], 1, Ts);
DLead = tf([1, -0.95], [1, -0.5], Ts);
rlocus(DNotch*DLead*c2d(Pointer, Ts))
rlocfind(DNotch*DLead*c2d(Pointer, Ts))
```

and then placing the crosshairs on the root-locus line that leaves the unit circle, one finds a gain of 15. Using this gain and executing the following program produces Figure 10.31. It is seen that the performance of both systems is very close.



(a)

(b)

Figure 10.31 Digital implementation of the flexible pointer system: (a) Root locus. (b) Step response.

```
Ts = 0.01;
DNotch = zpk([0.92+0.3i, 0.92-0.3i], [0.6 0.6], 1, Ts);
DLead = tf([1, -0.95], [1, -0.5], Ts);
figure(1)
rlocus(DNotch*DLead*c2d(Pointer, Ts))
axis(1.2*[-1, 1, -1, 1]);
zgrid
figure(2)
[yd, t] = step(feedback(15*DNotch*DLead*c2d(Pointer, Ts), 1));
Lead = tf([1, 6], [1, 50]);
Notch = zpk([-3+30i, -3-30i], [-60, -60], 1);
yn = step(feedback(40.0*Notch*Lead*Pointer, 1), t);
plot(t, yd, 'k-', t, yn, 'k--')
legend('Discrete control', 'Continuous control', 'Location', 'SouthEast')
xlabel('Time (s)')
ylabel('Response')
axis(1.2*[0, 1, 0, 1])
```

10.5.2 PID Control of a Magnetic Suspension System

Consider the magnetic suspension system[8] shown in Figure 10.32. The objective is to keep the ball floating at a desired height when it is subjected to external disturbances. The height of the ball is $h(t)$ and the current in the coil is $i(t)$. The equations of motion for the magnetic suspension are

$$\begin{aligned} m\frac{d^2h(t)}{dt^2} &= mg - k\frac{i^2(t)}{h^2(t)} \\ L\frac{di(t)}{dt} &= v(t) - Ri(t) \end{aligned} \tag{10.13}$$

where m is the mass of the ball, $g = 9.82$ m/s^2 is the gravitational constant, L is the inductance of the coil, R the coil resistance, and k the coupling factor between the magnetic fields and the ball. The input to the system is the coil voltage $v(t)$ and the measured output is the height of the ball $h(t)$. The equations are nonlinear.

Ideally, the ball should be located far enough from the magnet so that the magnetic force cancels the pull of gravity. If the ball drops too far, then the magnetic fields are weaker and the ball drops away completely. If the ball is too close to the magnet, then the magnetic fields are stronger and the ball will be pulled to the magnet. Our first step is to compute the point where the gravitational pull equals the attractive magnetic force. This point is called an equilibrium point. The current that is required

[8] B. Friedland, *Advanced Control System Design*, Prentice Hall, Englewood Cliffs, NJ, 1996.

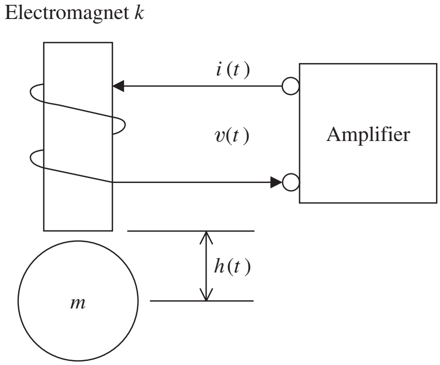


Figure 10.32 Magnetic suspension system.

to maintain desired position h_0 can be found by setting the acceleration equal to zero. Hence, from Eq. (10.13),

$$i_0^2 = \frac{mg}{k} h_0^2$$

Knowing the equilibrium point, the model can be linearized about it. This linearization simplifies Eq. (10.13) to give a set of linear equations.

First we introduce the state variables

$$x_1 = h \quad x_2 = \frac{dh}{dt} \quad x_3 = i$$

Then, Eq. (10.13) becomes

$$\begin{aligned} \frac{dx_1}{dt} &= x_2 \\ \frac{dx_2}{dt} &= g - \frac{k}{m}\left(\frac{x_3}{x_1}\right)^2 \\ \frac{dx_3}{dt} &= \frac{v}{L} - \frac{R}{L}x_3 \end{aligned} \tag{10.14}$$

Equation (10.14) can be linearized by taking a Taylor's series expansion around the operating points $x_3 = i_0$ and $x_1 = h_0$. The linearization results in dx_2/dt being modified to yield

$$\begin{bmatrix} dx_1/dt \\ dx_2/dt \\ dx_3/dt \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ \dfrac{2k}{m}\dfrac{i_0^2}{h_0^3} & 0 & \dfrac{-2k}{m}\dfrac{i_0^2}{h_0^2} \\ 0 & 0 & -R/L \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + v \begin{bmatrix} 0 \\ 0 \\ 1/L \end{bmatrix}$$

We assume the mass of the ball is 100 gm, the resistance of the coil is 5 Ω, the inductance of the coil is 40 mH, the coupling constant is 0.01 Nm^2/A, and the desired height is 2 cm. We first create the function M file **MagLev** to represent the state–space model of the system.

```
function Plant =MagLev
m = 0.1;   g = 9.82;   R = 5;
L = 0.04;   k = 0.01;   h0 = 0.02;
i0 = h0*sqrt(m*g/k);
A = [0, 1, 0; 2*k*i0^2/(m*h0^3), 0, -2*k*i0/(m*h0^2); 0, 0, -R/L];
B = [0; 0; 1/L];
C = [1, 0, 0];
D = 0;
Plant = ss(A, B, C, D);
```

Typing in the command window

```
MagPoles = pole(MagLev)
```

we obtain

```
MagPoles =
     31.3369
    -31.3369
   -125.0000
```

We see that the poles of the linearized system are located at ±31.3 for the suspension and at -125 for the amplifier. Thus, a proportional derivative (PD) controller to stabilize the system is needed. Theoretically, the transfer function for a PD controller is given by

$$C_0(s) = k_p + sk_d$$

where k_p is the proportional gain and k_d is the derivative gain. The controller output involves the derivative of the input, which is hard to realize in practice because of high-frequency noise. Typically, the derivative is approximated and then filtered to remove the noise, resulting in

$$C_1(s) = k_p + k_d \frac{s}{\tau_f s + 1} = k_p \frac{(\tau_f + k_d/k_p)s + 1}{\tau_f s + 1}$$

The transfer function $C_1(s)$ is equivalent to a lead controller with the zero time constant $\tau_f + k_d/k_p$ and the pole (filter) time constant τ_f. The controller is a lead controller because the zero is always slower than the pole. We select the controller zero at -20, which is just to the right of the first stable pole of the magnetic levitation system, and the filter pole at -50, which results in $\tau_f = 20$ ms. This system requires positive feedback to be stabilized, so we include the sign change in the controller. Thus, the transfer function is

$$C_1(s) = -\frac{s + 20}{s + 50}$$

The script is as follows:

```
PD = tf(-1*[1,20],[1,50]);
rlocus(PD*MagLev);
sgrid
xlabel('Real axis')
ylabel('Imaginary axis')
```

Upon execution, we obtain the results shown in Figure 10.33. The plot shows that at some low gain, the unstable pole is pulled into the left-half plane, and at a higher gain, a complex conjugate pair crosses over into the right-half plane. A stabilizing gain can be determined from this figure by typing in the command window

```
rlocus(tf(-1*[1,20],[1,50])*MagLev)
rlocfind(tf(-1*[1,20],[1,50])*MagLev)
```

and then placing the crosshairs on the real axis root-locus line between the unstable pole and the controller zero. A point approximately halfway between the imaginary axis and the controller zero yields a gain of 150. Hence, $k_p = -60$, $k_d = -1.8$, and the filter time constant is 20 ms.

Measurement and modeling errors are likely to result in errors in determining $v_0 = Ri_0$. This will cause a steady-state error in the position of the ball, $h(t)$. While the PD portion of the control shapes the instability and transient behavior of the system, the PI portion is typically used to improve the steady-state behavior. Consider the PI controller:

$$C_2(s) = k_p + \frac{k_i}{s} = \frac{k_p}{k_i}\frac{s + k_i/k_p}{s}$$

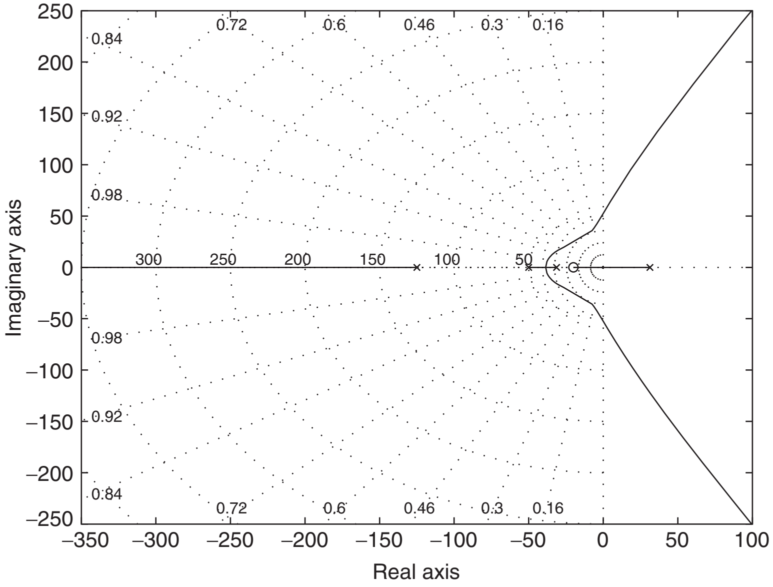


Figure 10.33 Root-locus plot of the PD-cascade control magnetic-levitation system.

with parameters k_p and k_i. The PI controller has one pole at the origin and a zero at $-k_i/k_p$. If the zero is close to the pole relative to the locations of the other poles and zeros of the system, then the effect of the PI controller on the closed-loop transient behavior is negligible when it is cascaded with the PD controller to form a PID controller. Hence, the feedback gain of 150 may still be used. The effect of the PI controller on the steady-state error is large. For this control system, we choose $k_p = k_i = 1$. The following script simulates the impulse response of the closed-loop system with the linearized model of the magnetic levitator.

```
PD = tf(-1*[1, 20], [1, 50]);
PI = tf([1, 1], [1, 0]);
[y, t] = impulse(feedback(150*PI*PD*MagLev, 1));
plot(t, y, 'k-')
grid
xlabel('Time')
ylabel('Impulse response')
```

The result of executing this script is shown in Figure 10.34, which is the closed-loop system of the linearized plant model. However, determining the stability of the system with the nonlinear model in the loop is of far greater interest.

The nonlinear system will now be simulated using SIMULINK. We will require a block called *S-Function*, which is a user-defined function. We will call this user-defined function **MagModel**, which is given by

```
function [sys, x0] = MagModel(t, x, u, flag)
m = 0.1;   g = 9.82;   R = 5;   L = 0.040;
k = 0.01;   h0 = 0.02;
```

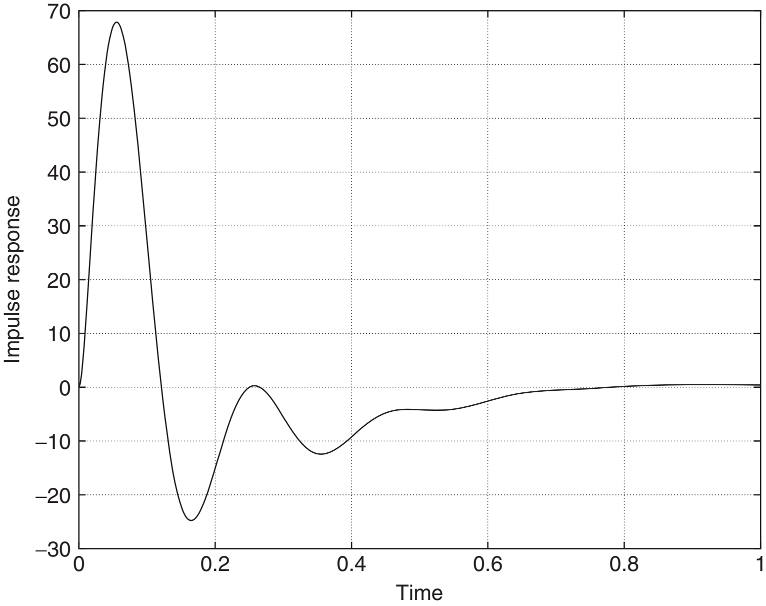


Figure 10.34 Impulse response of the linear approximation to the magnetic-levitation system.

```
i0 = h0*sqrt(m*g/k);
switch flag
  case 1
    xdot = zeros(3,1);
    xdot(1) = x(2);
    xdot(2) = m*g-k*x(3)^2/x(1)^2;
    xdot(3) = -R/L*x(3) + 1/L*u(1);
    sys  = xdot;
  case 3
    sys = x(1);
  case 0
    sys = [3 0 1 1 0 0];
    x0 = [h0+0.1*h0; 0; i0];
  otherwise
    sys = [];
end
```

The four parameters to be passed to **MagModel** are in the order specified by SIMULINK's *S-Function*: time t, state variables x, inputs u, and an integer *flag*. Hybrid models with both discrete and continuous states may be constructed using *S-Function*; we consider those parts that enable continuous nonlinear models. SIMULINK queries the user function to determine everything about the nonlinear model; *flag* determines the purpose of the query. When $flag = 1$, the function returns the derivatives of x using time t, states x, and input u given by Eq. (10.12). When $flag = 3$, the function returns the outputs. Finally, when $flag = 0$, the function returns a vector *sys* whose components are, in order, the number of continuous states, the number of discrete states, the number of outputs, the number of inputs, the number of roots, and a final flag that is set to 1 if the system has direct feedthrough. In the case of the magnetic levitator, $sys = [3\ 0\ 1\ 1\ 0\ 0]$, which indicates three continuous states, no discrete states, one input, one output, no roots, and no feedthrough. When $flag = 0$, we also return the initial conditions of the continuous states—that is, the state of the system when it is first started. The equilibrium of the system is $x(0) = [h(0)\ 0\ i(0)]'$. We will start the system near but not at the equilibrium position, by setting the initial value $h(0)$ 10% larger than the equilibrium value. With such an initial condition, the controller must take action or the ball will drop away from the magnet. The deviation for the equilibrium position must be small, because the system is nonlinear and the linear behavior on which the controller is based is only valid for small perturbations. In addition, if the ball drops away too far, the coil might not be strong enough to pull it back to the equilibrium position.

The controller will be specified using variable names rather than numerical values; therefore, these variables must be defined in the command window prior to running the simulation in SIMULINK. To generate these variables and start SIMULINK, we run the following script in the command window:

```
PD = tf(-1*[1, 20], [1, 50]);
PI = tf([1, 1], [1, 0]);
v0 = 0.991;   h0 = 0.02;
[num, den] = tfdata(150*PD*PI, 'v');
simulink
```

The values of *v*0 and *h*0 were computed from the values given.

We now use SIMULINK to generate the block diagram shown in Figure 10.35. From the *Math* library, we use *Sum* and *Gain*. For *Gain*, we enter a gain of 2. From the *User Defined Functions* library, we use *S-Function*, which, in turn, will use **MagModel**. This assignment is done by double-clicking on the *S-Function* block in the modeling window and entering **MagModel** for the *S-Functio*n name.

From the *Continuous* library, we use *Transfer Fnc*. Then we double-click on this block and enter the variable names *num* and *den* for the numerator and denominator, respectively, since these two quantities have been determined in the fourth line in the script that was run prior to entering SIMULINK.

From the *Sources* library, we use *Constant* to enter a voltage offset and a height offset. These quantities are used to represent errors. Their values are defined by double-clicking on the block and entering either numerical values or variable names, if the variable names have been or will be assigned numerical values in the command window. We choose the latter method since *v*0 and *h*0 have been defined in our previously run script.

For the connectivity shown, we go to the *Format* pull-down menu and apply *Flip Block* sequentially on the right-hand *Sum* block, on the *Gain* block, on the *Transfer Fcn* block, and on the right-hand *Constant* block.

Finally, we use *To Workspace* from the *Sinks* library to record the bearing's height (*Height*) and the magnitude of the magnet's coil voltage (*Vapp*) as a function of time. The values of time are stored in the array *Time* by *Clock*, which is from the *Sources* library. These variable names are entered in each of these three blocks by double-clicking on them and entering the appropriate name as indicated in Figure 10.35. In addition, *Array* in *Save Format* was selected for each of these variables. The shadows around these three blocks are obtained using *Show Drop Shadow* from the *Format* pull-down menu. From this same menu, we have also selected

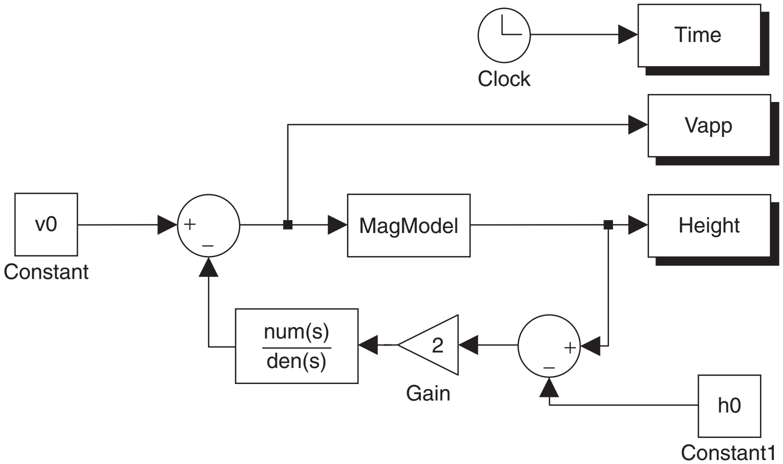


Figure 10.35 SIMULINK block diagram for the nonlinear magnetic-levitation system.

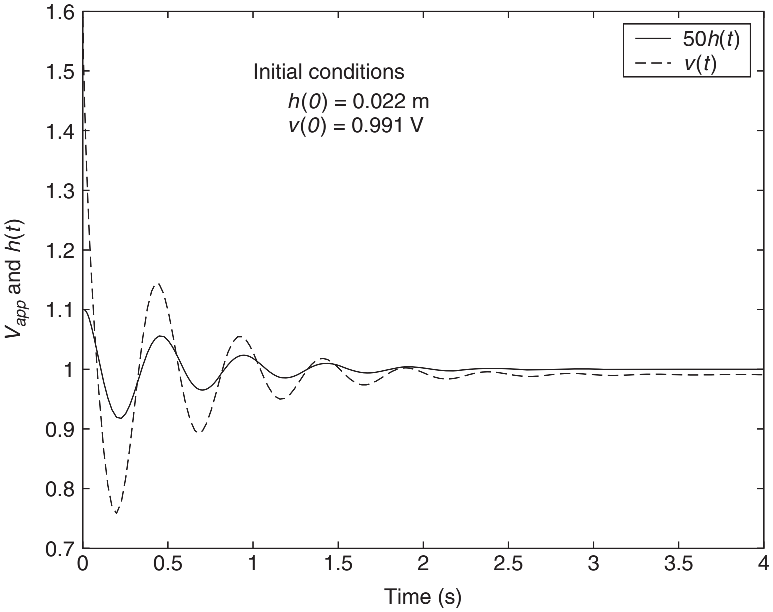


Figure 10.36 Response under PID control of the nonlinear magnetic suspension system to initial conditions and modeling errors.

Hide Name for each of them. The quantities *Vapp*, *Height*, and *Time* are saved to the workspace each time the simulation is run and can then be displayed with `plot`. In Figure 10.36, we show the quantities *Vapp* and *Height* as a function of *Time*. These quantities were obtained with the following script run in the command window after the SIMULINK entity was executed. The values of *Height* have been scaled by a factor of 50.

```
plot(Time, 50*Height, 'k-', Time, Vapp, 'k--.')
legend('50h(t)', 'v(t)')
text(1, 1.5, 'Initial conditions')
text(1.2, 1.45, 'h(0) = 0.022 m')
text(1.2, 1.4, 'v(0) = 0.991 V')
xlabel('Time (s)')
ylabel('V_{app} and h(t)')
axis([0 4 .7 1.6])
```

10.5.3 Lead Control of an Inverted Pendulum

We shall design a lead controller for an inverted pendulum by using root-locus techniques. Consider an inverted pendulum mounted on a disk as shown in Figure 10.37. The objective of the control system is to command the position of the disk while

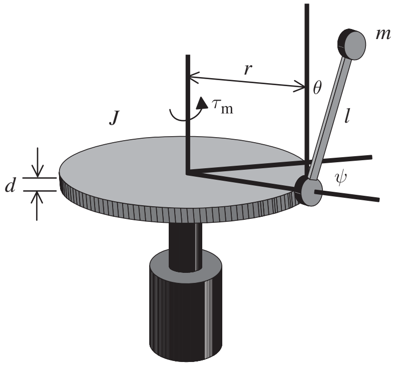


Figure 10.37 Inverted pendulum on a disk.

keeping the pendulum upright. Both the angle of the disk ψ and the angle of the pendulum θ are measured. The equations of motion are

$$ml^2 \frac{d^2\theta}{dt^2} + mrl\cos(\theta)\frac{d^2\psi}{dt^2} = mgl\sin(\theta) + b_1\frac{d\theta}{dt} \tag{10.15}$$

$$mrl\cos(\theta)\frac{d^2\theta}{dt^2} + (J + mr^2)\frac{d^2\psi}{dt^2} = mrl\sin(\theta)\left(\frac{d\theta}{dt}\right)^2 + b_2\frac{d\psi}{dt} + \tau_m$$

where m is the mass of the bob, l is the length of the pendulum, r is the radius of the disk and the radius of the bob attachment, d is the thickness of the disk, $J = \rho\pi dr^4/4$ is the inertia of the disk, b_1 is the friction coefficient of the revolute joint of the pendulum, b_2 is the friction in the revolute joint of the disk, and τ_m is the torque applied by the motor attached to the base of the disk. As with the magnetic bearing, the equations of motion may be linearized about the operating point when θ and $d\theta/dt$ are small.

We define the elements of the state vector x as

$$x_1(t) = \theta(t) \quad x_3(t) = \frac{d\theta}{dt}$$

$$x_2(t) = \psi(t) \quad x_4(t) = \frac{d\psi}{dt}$$

Substituting these equations into Eq. (10.15) and assuming that θ and $d\theta/dt$ are very small and, hence, we can neglect all terms of order 2 and higher, we obtain

$$\frac{dx_1}{dt} = x_3$$

$$\frac{dx_2}{dt} = x_4$$

$$ml^2 \frac{dx_3}{dt} + mlr \frac{dx_4}{dt} = mglx_1 + b_1 x_3$$

$$mlr \frac{dx_3}{dt} + (J + mr^2) \frac{dx_4}{dt} = b_2 x_4 + \tau_m$$

or in matrix form

$$M\dot{x} = Qx + Wu$$

where

$$W = [0 \quad 0 \quad 0 \quad 1]' \quad x = [x_1 \quad x_2 \quad x_3 \quad x_4]' \quad \dot{x} = dx/dt$$

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & ml^2 & mlr \\ 0 & 0 & mlr & J + mr^2 \end{bmatrix} \quad Q = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ mgl & 0 & b_1 & 0 \\ 0 & 0 & 0 & b_2 \end{bmatrix} \quad u = \tau_m$$

The upright position corresponding to $x(t) = 0$ is an equilibrium point of the system at which $u = \tau_m = 0$. The linearized equations of the inverted pendulum system about the upright position are represented as a state–space system in the following function M file **Pendulum**. This system has the angle of the pendulum θ and the angle of the disk ψ as the outputs. Therefore, the output *Plant* of **Pendulum** is a system with two outputs and one input, which can be accessed as follows: *Plant*(1, 1) is the transfer function from τ_m to θ and *Plant*(2, 1) is the transfer function from τ_m to ψ. We assume the length of the pendulum is 30 cm, the mass of the bob 0.2 kg, the radius of the disk 15 cm, the thickness of the disk 1 cm, and its density 2,500 kg/m^3. The friction in the systems is set to zero; thus, $b_1 = b_2 = 0$.

```
function Plant = Pendulum
l = 0.3;    g = 9.81;    m = 0.2;    r = 0.15;
d = 0.01;    rho = 2500;    b1 = 0;    b2 = 0;
J = 0.25*pi*rho*d*r^4;
M = [1, 0, 0, 0; 0, 1, 0, 0; 0, 0, m*l^2 m*r*l; 0, 0, m*r*l, J+m*r^2];
Q = [0, 0, 1, 0; 0, 0, 0, 1; m*g*l, 0, b1, 0; 0, 0, 0, b2];
W = [0; 0; 0; 1];
A = inv(M)*Q;
B = inv(M)*W;
C = [1, 0, 0, 0; 0, 1, 0, 0];
D = [0];
Plant = ss(A, B, C, D);
```

The poles of the system are found by typing

```
pole(Pendulum)
```

in the command window, which then displays

```
0
0
 6.8923
-6.8923
```

Thus, there are two poles at the origin and a pair on the real axis mirrored about the imaginary axis at ±6.9 rad/s. The system is, therefore, open-loop unstable. The transmission zeros of the inverted pendulum from the perspective of θ, the angle of the pendulum, is found by typing in the command window

```
Plant = Pendulum;
tzero(Plant(1, 1))
```

which displays two zeros. The system with only the first output θ and first input τ_m may be addressed using matrix notation; hence, the $(1, 1)$ subscript on *Plant*. Thus, there are two zeros right on top of the poles. This indicates that using only the output θ to control the inverted pendulum will ignore some of the dynamics. In particular, the angle of the pendulum contains insufficient information to discern the position and velocity of the disk. Thus, the outputs of a sensor measuring this angular motion can be zero even when the disk is rotating at a uniform angular velocity. The dynamics, which is not at rest while the output is zero, is sometimes referred to as zero dynamics. Such (unobservable) zero dynamics will not be stabilized by feedback from θ alone. The zeros of the pendulum from the perspective of the disk angle ψ are found by typing

```
Plant = Pendulum;
tzero(Plant(2, 1))
```

in the command window. Since the plant has two outputs, we address the subsystem with the second output and the first input with the subscript $(2, 1)$. It is found from these results that the zeros are located at ± 5.72, which are close to the open-loop poles of the pendulum, which are ± 6.9. This indicates that while the behavior of the pendulum is observable from the disk position, it is just barely so. A single-input, single-output controller designed using either output will perform badly.

A multiple-input, single-output (MISO) controller that depends on both outputs would perform much better than a SISO controller and, thus, this will be the focus of the rest of our effort. We shall design this controller shown in Figure 10.38 in two steps. First, we shall design a controller using the output θ to keep the pendulum upright. Wrapped around this control loop will be a controller that uses ψ to keep the angle of the disk at the commanded position.

We start the design process with a lead controller that keeps the pendulum upright. Such a controller will use the output θ to determine which correction to apply. Recall that the open-loop poles are located at ± 6.9. We put the lead zero at -5, just to the right of the stable pole in the pair in order to pull the unstable

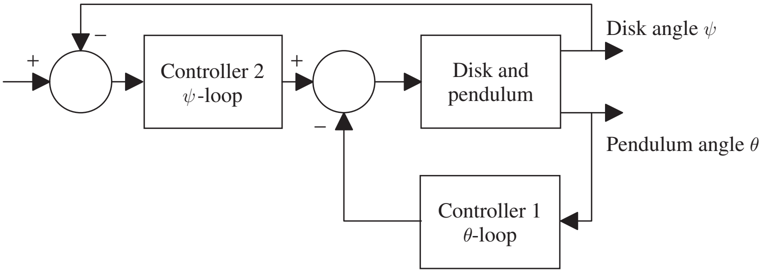


Figure 10.38 Block diagram for the inverted pendulum control system.

pole into the left half of the complex plane. The lead pole is placed at −10. Hence,

$$C_\theta(s) = -\frac{s + 5}{s + 10}$$

Since positive feedback is required, a negative sign appears in $C_\theta(s)$.

The following script generates the root-locus plot of the θ control loop. MATLAB automatically picks a range of gains to apply, but in this case we select a range of $0.1 \leq \theta \leq 10$ equally spaced on a logarithmic scale.

```
Plant = Pendulum;
PlantTheta = minreal(Plant(1,1));
ControlTheta = tf(-1*[1, 5], [1, 10]);
r = rlocus(ControlTheta*PlantTheta, logspace(-1, 1, 60));
plot(real(r), imag(r), 'kx')
sgrid
xlabel('Real axis')
ylabel('Imaginary axis')
```

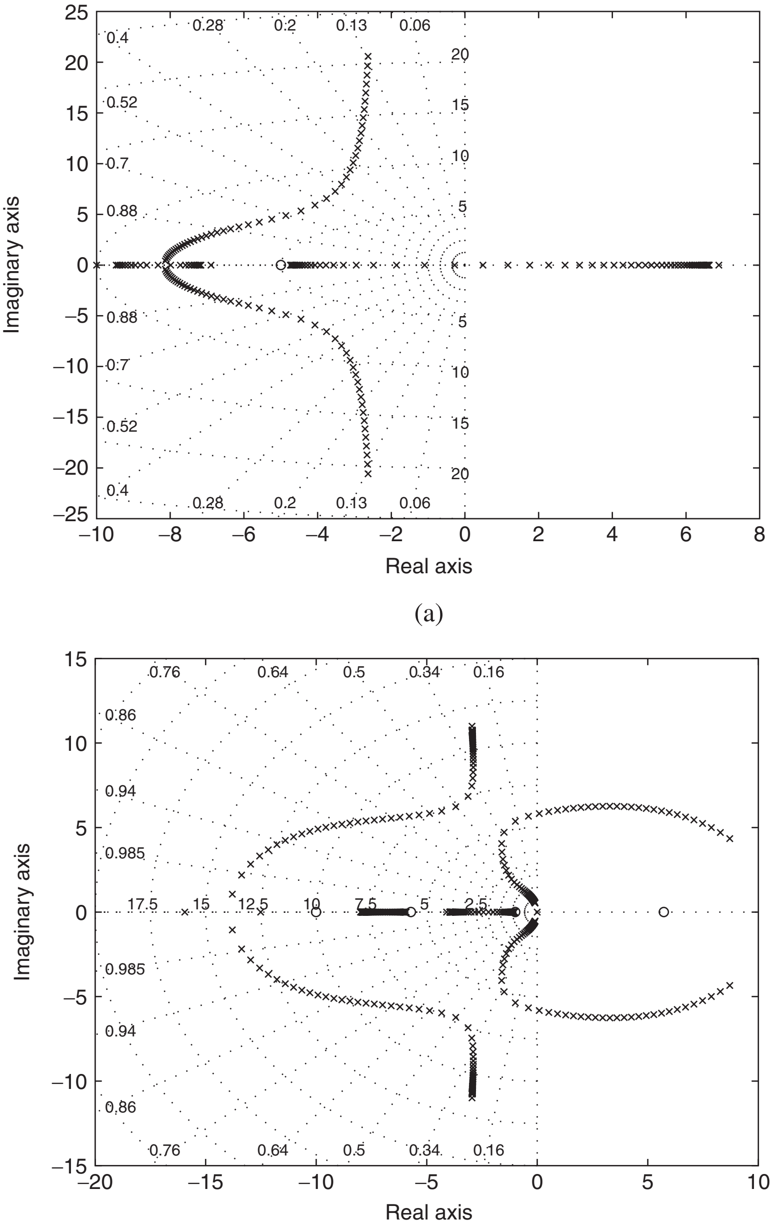
Executing the script gives the results shown in Figure 10.39a.

A suitable gain can be found with

```
Plant = Pendulum;
PlantTheta = minreal(Plant(1, 1));
ControlTheta = tf(-1*[1, 5], [1, 10]);
rlocus(ControlTheta*PlantTheta)
rlocfind(ControlTheta*PlantTheta)
```

Placing the crosshairs on the real axis between the imaginary axis and the lead zero, we find that a gain of 4 stabilizes the θ-loop. The resulting closed-loop poles of the theta control system are found from the following script:

```
Plant = Pendulum;
PlantTheta = minreal(Plant(1, 1));
ControlTheta = tf(-1*[1, 5], [1, 10]);
pole(feedback(4*ControlTheta*PlantTheta,1))
```



(a)

(b)

Figure 10.39 Root locus of the control loops in feedback for (a) the θ control loop and (b) the ψ control loop.

It is found from the execution of this script that the closed-loop poles are approximately -4, $-3 \pm 11i$, which, although stable, are not well damped.

Now we design the outer feedback loop for the disk position ψ. The outer feedback loop is designed with the inner θ loop in place, so we first form the closed-loop system by wrapping the first controller inside a loop employing θ. We do this with the following script:

```
ControlTheta = tf(-[1, 5], [1, 10]);
PlantPsi = feedback(Pendulum, 4*ControlTheta, [1], [1]);
pole(PlantPsi(2,1))
tzero(PlantPsi(2,1))
```

Note that we have to specify which inputs and outputs to use since the plant is MIMO and the controller is SISO. The results show that *PlantPsi* has two additional poles at the origin, which is representative of a double integrator. These need to be moved to the left; however, complicating that objective is an unstable zero at 5.72. The inverted pendulum is an example of a system that is nonminimum phase. In order to move the disk, the controller must first move in the opposite direction to keep the pendulum upright during the transition from its current position to the desired position. The unstable zero attracts one of the poles of the double integrator. To solve this problem, we again use a lead controller whose zero is just inside the left half of the complex plane. The following script plots the root locus of the ψ control loop:

```
ControlTheta = tf(-[1, 5], [1, 10]);
PlantPsi = feedback(Pendulum, 4*ControlTheta, [1], [1]);
ControlPsi = tf(-[1, 1], [1, 8]);
r = rlocus(ControlPsi*PlantPsi(2,1), 0.35*logspace(-1, 1, 60));
plot(real(r), imag(r), 'kx')
sgrid;
xlabel('Real axis')
ylabel('Imaginary axis')
```

Executing this script results in Figure 10.39b. To find the appropriate gain, we type

```
ControlTheta = tf(-[1, 5], [1, 10]);
PlantPsi = feedback(Pendulum, 4*ControlTheta, [1], [1]);
ControlPsi = tf(-[1, 1], [1, 8]);
rlocus(ControlPsi*PlantPsi(2, 1))
rlocfind(ControlPsi*PlantPsi(2, 1))
```

in the command window and place the crosshairs near the lower half of the root locus lines that loop into the complex plane. It is found that a gain of 0.3 places all the poles inside the left-half complex plane.

The step response of the final control system is computed from the following script:

```
ControlTheta = tf(-[1, 5], [1, 10]);
PlantPsi = feedback(Pendulum, 4*ControlTheta, [1], [1]);
```

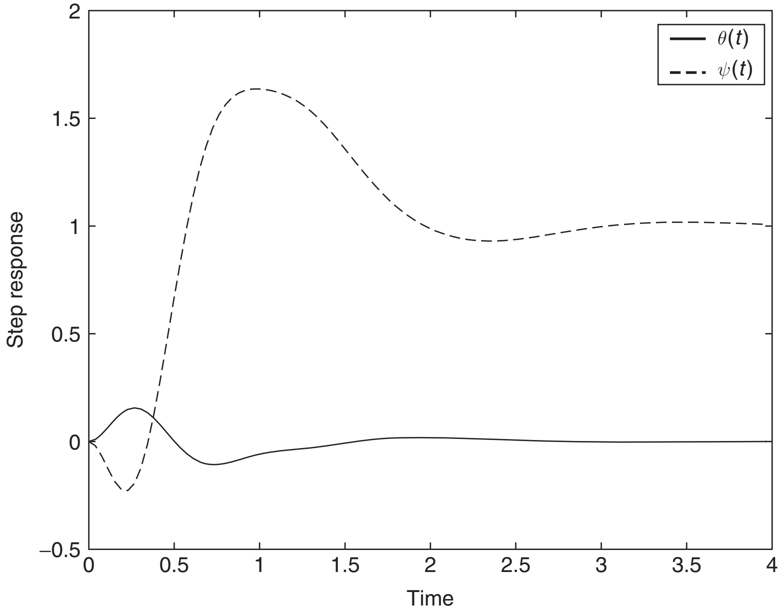


Figure 10.40 Step response of the inverted pendulum. Note that the response of the disk (broken line) initially goes the wrong way. This initial action tilts the pendulum (solid line) into the direction of the move.

```
ControlPsi = tf(-[1, 1], [1, 8]);
[y, t] = step(feedback(0.3*ControlPsi*PlantPsi, 1, [1], [2]));
plot(t, y(:,1), 'k-', t, y(:,2), 'k--')
xlabel('Time')
ylabel('Step response')
legend('\theta(t)', '\psi(t)')
```

The resulting step-response plot is shown in Figure 10.40, which reveals the nonminimum phase behavior of the controller and the plant.

10.5.4 Control of a Magnetically Suspended Flywheel

Consider the magnetically suspended flywheel system shown in Figure 10.41. Magnetic coils are used to float the wheel so that the wheel can be run at high speeds without the losses associated with friction. The objective of the control system is to keep the wheel suspended. Like the magnetic suspension described in Section 10.5.2, the system is naturally unstable. There are four distances that are measured as outputs. These distances correspond to the *x* and *y* positions of the top and bottom shafts as measured in the inertial frame. Four coil currents may be selected to control the magnetic fields about the shaft; these coils are colocated with the sensors. The linearized equations of motion are

$$\frac{d^2 x_{\text{cm}}}{dt^2} = \frac{f_1 + f_3}{m} \quad \frac{d^2 y_{\text{cm}}}{dt^2} = \frac{f_2 + f_4}{m}$$

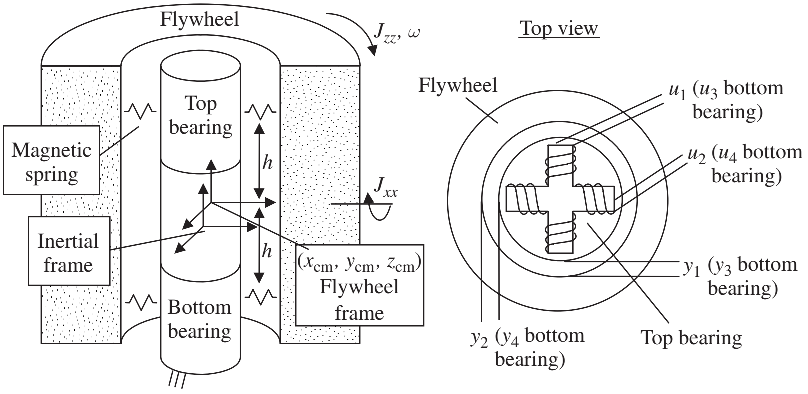

Flywheel
J_{zz}, ω
Top bearing
Magnetic spring
h
J_{xx}
Inertial frame
h
$(x_{\text{cm}}, y_{\text{cm}}, z_{\text{cm}})$ Flywheel frame
Bottom bearing
Top view
Flywheel
u_1 (u_3 bottom bearing)
u_2 (u_4 bottom bearing)
y_1 (y_3 bottom bearing)
y_2 (y_4 bottom bearing)
Top bearing

Figure 10.41 Magnetically suspended flywheel.

$$\frac{d^2\varphi}{dt^2} = -\beta\omega\frac{d\psi}{dt} + \frac{h}{J_{xx}}(f_4 - f_2)$$

$$\frac{d^2\psi}{dt^2} = \beta\omega\frac{d\varphi}{dt} + \frac{h}{J_{xx}}(f_1 - f_3)$$

where x_{cm} and y_{cm} are the location of the center of mass of the flywheel as measured in the inertial frame, (ϕ, ψ) give the orientation of the flywheel frame with respect to the inertial frame using roll (ϕ), pitch (ψ), and yaw orientation,[9] m is the mass of the flywheel, J_{xx} is the rotational inertia of the flywheel about the nonspinning axis, $\beta = J_{zz}/J_{xx}$, and h is the distance from the center of mass to the actuators. The inputs f_i are the forces applied by the magnetic bearings and obey the following relationships:

$$f_i = k_1 y_i + k_2 u_i$$

where y_i is the distance from the wheel to the actuator and is the system's output. The bearings are composed of a permanent magnet surrounded by coils. The negative spring constant k_1 is due to the permanent magnet, and the gain k_2 is due to the field generated by the current u_i in the coils. The operating speed of the wheel is ω rad/s.

The output values y_i for small ϕ and ψ are given by

$$y_1 = x_{\text{cm}} + h\psi$$

$$y_2 = y_{\text{cm}} - h\varphi$$

$$y_3 = x_{\text{cm}} + h\psi$$

$$y_4 = y_{\text{cm}} - h\varphi$$

If we let

$$q(t) = [x_{\text{cm}}, y_{\text{cm}}, \varphi(t), \psi(t)]'$$

[9] R. M. Murray, X. Li, and S. S. Sastry, *A Mathematical Introduction to Robotic Manipulation*, CRC Press, Boca Raton, FL, 1994.

and

$$u(t) = [u_1 \quad u_2 \quad u_3 \quad u_4]'$$

then the linearized equations can be written as

$$\begin{aligned} \ddot{q} &= \omega P_a \dot{q} + k_1 B_a C_a q + k_2 B_a u \\ y &= C_a q \end{aligned}$$

where

$$P_a = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \beta \\ 0 & 0 & -\beta & 0 \end{bmatrix}$$

$$B_a = \begin{bmatrix} 1/m & 0 & 1/m & 0 \\ 0 & 1/m & 0 & 1/m \\ 0 & -h/J_{xx} & 0 & h/J_{xx} \\ h/J_{xx} & 0 & -h/J_{xx} & 0 \end{bmatrix} \tag{10.16}$$

$$C_a = \begin{bmatrix} 1 & 0 & 0 & h \\ 0 & 1 & -h & 0 \\ 1 & 0 & 0 & -h \\ 0 & 1 & h & 0 \end{bmatrix}$$

Hence, if we set

$$z = \begin{bmatrix} q \\ \dot{q} \end{bmatrix}$$

we have the following matrix equations

$$\begin{aligned} \dot{z} &= \begin{bmatrix} 0 & I \\ k_1 B_a C_a & \omega P_a \end{bmatrix} z + \begin{bmatrix} 0 \\ k_2 B_a \end{bmatrix} u \\ y &= [C_a \quad 0] z + [0] u \end{aligned}$$

where I is the identity matrix.

The following function M file **Flywheel** generates the linearized model as a function of rotational speed in revolutions per minute (rpm) and is used in the subsequent design evaluations. If no arguments are supplied to this function, it is assumed that the operating speed is 0 rpm. For the flywheel under consideration, the rotational inertia about the x- and y-body axes of the flywheel is $1.563 \times 10^{-4}\,\mathrm{Nm} \cdot \mathrm{s}^2$, and the rotational inertia about the z-axis is $1.141 \times 10^{-4}\,\mathrm{Nm} \cdot \mathrm{s}^2$. The dimensionless shape factor β is approximately 1. The mass of the wheel is 340 gm and the height from the center of mass is 3 cm. The coil constants are $k_1 = 4.8 \times 10^4$ N/m and $k_2 = 3.75$ N/A.

```
function Plant = Flywheel(rpm)
if nargin < 1
   rpm = 0;
end;
Jxx = 1.563e-4;    Jzz = 1.141e-4;
beta = Jzz/Jxx;    m = 0.34;
h = 0.03;    k1 = 4.8e4;    k2 = 3.75;
omega = rpm/60*2*pi;
Pa = [zeros(1,4); zeros(1,4); 0, 0, 0, -beta; 0, 0, beta, 0];
Ba = [1/m, 0, 1/m, 0; 0, 1/m, 0, 1/m; 0, -h/Jxx, 0, h/Jxx; h/Jxx, 0, -h/Jxx, 0];
Ca = [1, 0, 0, h; 0, 1, -h, 0; 1, 0, 0, -h; 0, 1, h, 0];
A = [zeros(4), eye(4); k1*Ba*Ca, omega*Pa];
B = [zeros(4); k2*Ba];
C = [Ca, zeros(4)];
D = [zeros(4)];
Plant = ss(A, B, C, D);
```

The open-loop poles change as a function of *rpm*. Executing the following script plots the open-loop poles for a range of typical operating speeds.

```
rpm = [linspace(0, 16000, 15), linspace(16100, 20000, 15)];
result = zeros(8, length(rpm));
for j = 1:length(rpm)
   result(:,j) = pole(Flywheel(rpm(j)));
end
plot(real(result'), imag(result'), 'kx')
grid
xlabel('Real axis')
ylabel('Imaginary axis')
```

The result of executing this script is shown in Figure 10.42. There are two sets of poles (but no zeros) mirrored about the imaginary axis. One set, located at ±530 rad/s (85 Hz), is due to the translational modes and does not change with operating speed. The other set, located at ±740 rad/s (120 Hz), is due to the two rotational modes and is affected by the gyroscopic coupling.

In summary, there are four unstable and four stable poles, and half of the poles change location with the operating speed. From an examination of Eq. (10.16), we see that at $\omega = 0$ rpm, P_a does not affect the solution. This suggests that the flywheel at 0 rpm may be statically decoupled, resulting in two rotational and two translational SISO systems. Thus, traditional lead-control design techniques, such as a root locus, can now be used to generate a stabilizing controller.

Consider the matrix T, which forms the sums and differences between input channels:

$$T = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

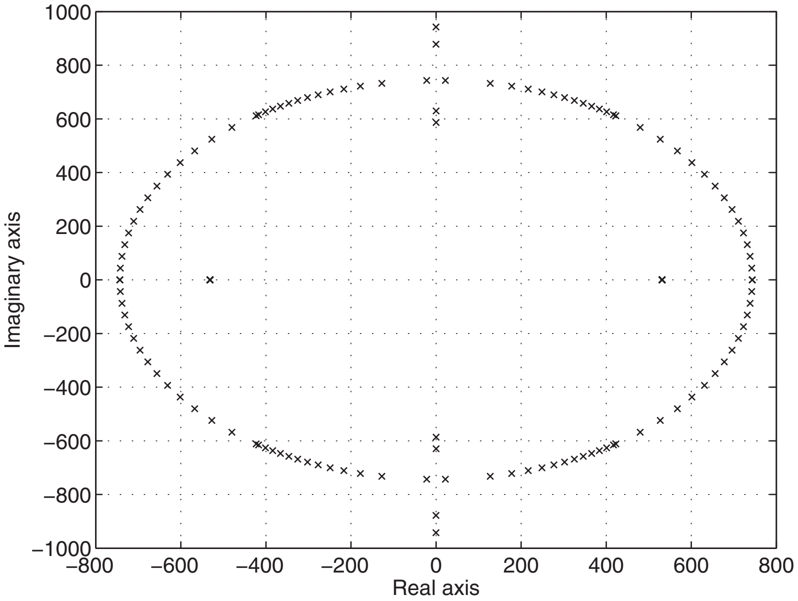


Figure 10.42 Root locus of the flywheel as a function of operating speed from 0 to 20,000 rpm.

Note that the matrix T diagonalizes the input and output matrices of Eq. (10.16) as shown in the following equations:

$$B_aT = \begin{bmatrix} 2/m & 0 & 0 & 0 \\ 0 & 2/m & 0 & 0 \\ 0 & 0 & 2h/J_{xx} & 0 \\ 0 & 0 & 0 & 2h/J_{xx} \end{bmatrix}$$

$$T^{-1}C_a = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & h & 0 \\ 0 & 0 & 0 & h \end{bmatrix}$$

After performing a static decoupling using T, we can perform our lead design on the four resulting SISO plants. Figure 10.43 shows how the matrices T would be inserted in practice. The following script generates the decoupled SISO plants and computes a root locus of the result:

```
T = [1, 0, 0, 1;0, 1, -1, 0;1, 0, 0, -1;0, 1, 1, 0];
decoupFly = inv(T)*Flywheel(0)*T;
transFly = minreal(decoupFly(1,1));
rotFly = minreal(decoupFly(3,3));
Lead = tf([1, 400], [1, 1000]);
rlocus(Lead*transFly)
```

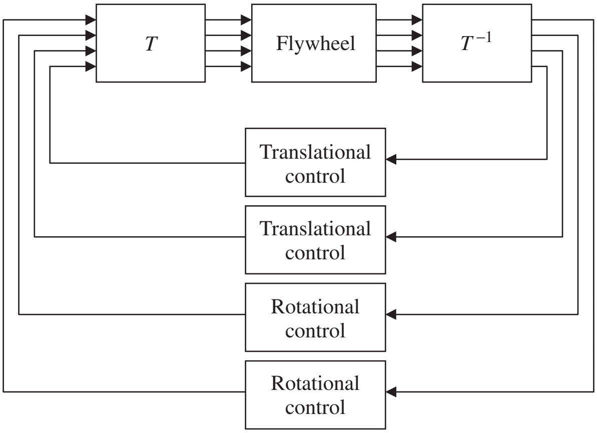


Figure 10.43 Static decoupling of the flywheel.

```
sgrid
xlabel('Real axis')
ylabel('Imaginary axis')
axis([-1000, 800, -2500, 2500])
figure(2)
rlocus(Lead*rotFly)
sgrid
axis([-1000, 800, -2500, 2500])
xlabel('Real axis')
ylabel('Imaginary axis')
```

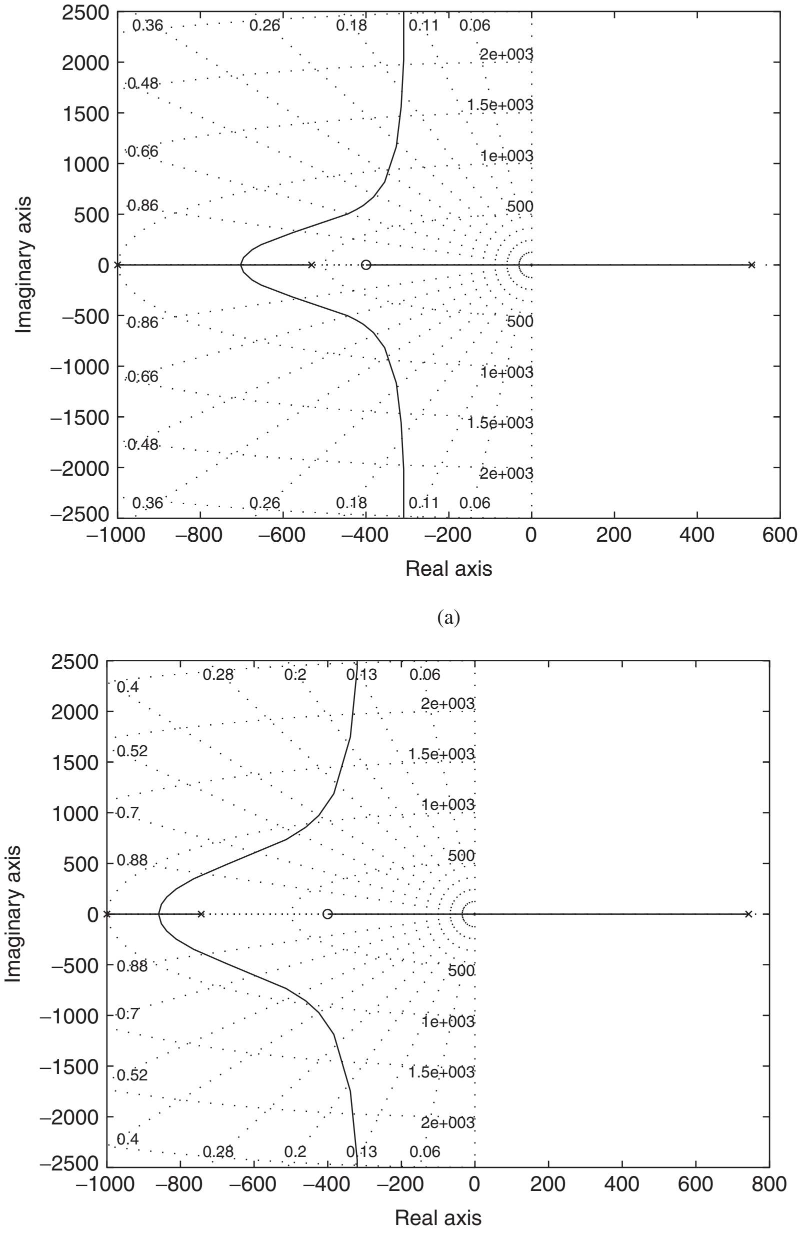
The result of executing this script is shown in Figure 10.44.

Upon typing the following commands in the command window

```
T = [1, 0, 0, 1; 0, 1, -1, 0; 1, 0, 0, -1; 0, 1, 1, 0];
decoupFly = inv(T)*Flywheel(0)*T;
transFly = minreal(decoupFly(1,1));
Lead = tf([1, 400], [1, 1000]);
rlocus(Lead*transFly)
rlocfind(Lead*transFly)
```

one can find a stabilizing gain for the translational system. Similarly, by typing the following commands in the command window

```
T = [1, 0, 0, 1; 0, 1, -1, 0; 1, 0, 0, -1; 0, 1, 1, 0];
decoupFly = inv(T)*Flywheel(0)*T;
rotFly = minreal(decoupFly(3,3));
Lead = tf([1, 400], [1, 1000]);
rlocus(Lead*rotFly)
rlocfind(Lead*rotFly)
```



(a)

(b)

Figure 10.44 Root locus of the control loops for (a) the translational and (b) the rotational motion.

one can find the stabilizing gain for the rotational system. By design, the same gain (5) stabilizes both the translational and rotational components of the system. Since the gain for the translational and rotational control is the same, the controllers are identical, and decoupling is not needed in the implementation because the matrix T commutes with the transfer function of the controller. This fact makes the implementation easier.

We now design an LQG controller for the flywheel. The design involves choosing the appropriate penalty R for the cost function. The following script computes the optimal closed-loop poles for $10^{-10} \leq R \leq 10^{-6}$. The result of its execution is shown in Figure 10.45.

```
[A, B, C, D] = ssdata(Flywheel);
L = 60;    clPoles = zeros(L,8);
R = logspace(-10, -6, L);
for i = 1:L
   [K, S, E] = lqr(A, B, C'*C, R(i)*eye(4));
   clPoles(i,:) = E;
end
plot(real(clPoles), imag(clPoles), 'kx')
sgrid
ylabel('Imaginary axis')
xlabel('Real axis')
text(-700,0,'R = 10^{-10}')
text(-1000,1000,'R = 10^{-6}')
```

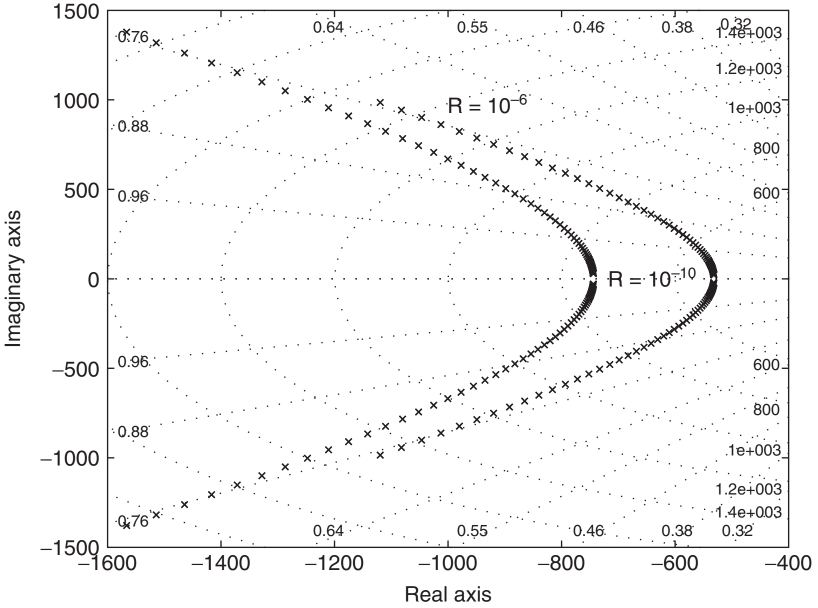


Figure 10.45 Optimal root locus for the flywheel for $10^{-10} \leq R \leq 10^{-6}$.

To compare the two different control schemes, the following script computes the response of the two different controllers with the same initial conditions. The initial conditions are all zero, except that the flywheel is slightly tilted such that $\phi(0) = 0.001$ radians. The flywheel spins at 10,000 rpm, even though the controllers were designed for 0 rpm. The LQG controller is designed with penalty factor $R = 10^{-6}$.

```
Control = 4e4*eye(4)*tf([1, 400], [1, 1000]);
x0 = zeros(12, 1); x0(3) = 1e-3; t = linspace(0, 0.25, 1000);
yl = initial(feedback(Flywheel(10000), Control), x0, t);
[A, B, C , D] = ssdata(Flywheel);
K = lqr(A, B, C'*C, 1e-6*eye(4));
L = (lqr(A', C', B*B', 1e-6*eye(4)))';
ControlSS = reg(Flywheel, K, L);
x0 = zeros(16, 1); x0(3) = 1e-3;
ys= initial(feedback(Flywheel(10000), ControlSS, +1), x0, t);
plot(1000*yl(:,1), 1000*yl(:,2), 'k-', 1000*ys(:,1), 1000*ys(:,2), 'k--')
grid on
xlabel('x upper bearing')
ylabel('y upper bearing')
legend('Lead Control', 'LQG Control', 'Location', 'SouthWest')
```

The results from the execution of the script are the phase plots shown in Figure 10.46. Both controllers keep the flywheel suspended with comparable levels of performance.

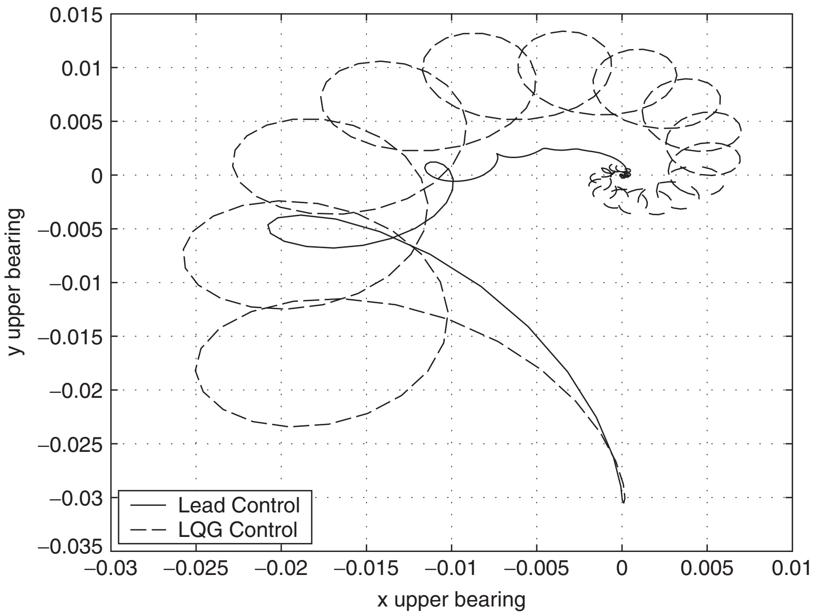


Figure 10.46 Phase response of the state–space and lead-controlled flywheels.

10.6 SUMMARY OF FUNCTIONS INTRODUCED IN CHAPTER 10

A summary of the functions introduced in Chapter 10 is presented in Table 10.1.

TABLE 10.1 MATLAB Functions Introduced in Chapter 10

MATLAB function	Description
`acker`	Pole placement for single-input systems
`alpha`	Sets transparency properties for objects
`arx`*	Estimates parameters of an ARX model
`bode`	Bode frequency response
`c2d`	Makes continuous-time systems discrete
`compare`*	Compares measured outputs with model outputs
`connect`	Obtains state–space model from block diagram description
`conv`	Convolution and polynomial multiplication
`dcgain`	DC gain
`detrend`*	Removes trends from output–input data
`feedback`	Feedback connection of two LTI models
`iddata`*	Packages input–output data into the iddata object
`ident`*	Opens the GUI
`ift`	Redheffer star product of two LTI models
`impulse`	Impulse response of LTI model
`initial`	Response of state–space models to initial conditions
`lqe`	Kalman estimator design for continuous-time systems
`lqr`	Linear-quadratic state-feedback regulator for continuous plant
`lsim`	Simulate response of LTI model to arbitrary input
`margin`	Gain and phase margins and associated crossover frequencies
`minreal`	Minimal realization of pole-zero cancellation
`nyquist`	Nyquist frequency response of LTI model
`parallel`	Parallel connection of two LTI models
`place`	Pole placement
`pole`	Poles of an LTI system
`reg`	Forms regulator given the state feedback and estimator gain
`rlocfind`	Finds root locus gains for a given set of roots
`rlocus`	Root locus
`series`	Series connection of two LTI models
`sgrid`	*s*-plane grid of constant damping factors and natural frequencies
`simulink`	Starts SIMULINK
`ss`	State–space model
`ssbal`	Balances state–space models using a diagonal similarity transformation
`ssdata`	Accesses state–space model data
`step`	Step response of LTI systems
`tf`	Transfer-function model
`tfdata`	Accesses transfer function data
`tzero`	Transmission zeros of LTI system
`zgrid`	*z*-plane grid of constant damping factors and natural frequencies
`zpk`	Specifies zero-pole-gain model

*System Identification toolbox

EXERCISES

Sections 10.2 and 10.3

10.1 Construct a fifth-order system with five equally spaced poles on a circle of radius $2\pi k$ in the left half of the complex plane, and five zeros in the right-half plane at the mirror-image locations. Set the DC gain to one and $k = 1$.

a. Cascade the fifth-order system with a simple first-order plant with a pole at -1 and a DC gain of 1. Plot the step response of the system for 4 s and the step response of the plant without the fifth-order system on the same graph.

b. Compute the percentage overshoot and the rise time.

c. Repeat parts (a) and (b) for $k = 0.5$ and $k = 2.0$. Notice how the response delay grows with increasing k.

d. Repeat parts (a) and (b) for three and seven equally spaced poles. Notice how the oscillations increase with an increase in the number of poles.

Sections 10.3

10.2 The suspension system[10] shown in Figure 10.47 has the level of the road surface as the input and absolute position of as the output . The transfer function of the system is

$$\frac{y(s)}{r(s)} = \frac{sbk_2 + k_1k_2}{m_1m_2s^4 + b(m_1 + m_2)s^3 + k_1(m_1 + m_2)s^2 + k_2m_1s^2 + k_2bs + k_1k_2}$$

Assume that $m_1 = 500$ kg, $m_2 = 100$ kg, $b = 1000$ Ns/m, $k_1 = 2000$ N/m, and $k_2 = 10^4$ N/m.

a. A washboard-like road surface can be approximated by

$$r(t) = \varepsilon \sin(2\,\pi f t)$$

Determine the value of f such that the amplitude of the induced response amplitude of the ride $y(t)$ is 10% of the amplitude of $r(t)$.

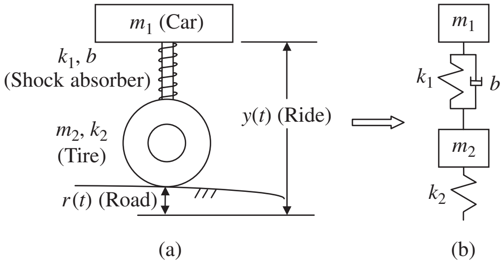


Figure 10.47 (a) Simplified model of an automotive wheel suspension system. (b) Mass spring equivalent.

[10] U. Ozguner, H. Goktas, and H. Chan, "Automotive Suspension Control Through a Computer Communication Network," *Proceedings of 1st IEEE Conference on Control Application*, Vol. 2, 1992, pp. 895–900.

b. Depending on its usage, the mass of the car may double. Generate a meshed surface of the magnitude of the Bode plot as a function of m_1 for $500 \le m_1 \le 1000$ kg.

c. Plot the value of ω for which the attenuation of the road variation $r(t)$ is 90% as a function of b and m_1.

10.3 Suppose a controller is designed using a simple nominal plant model such as[11]

$$G_0(s) = \frac{1}{s+1}$$

The plant may not be accurately modeled. For each of the following alternate models, plot the open-loop step response of the nominal plant $G_0(s)$ with the open-loop step response of the alternate plant. Also plot the closed-loop step response of both systems with proportional error control and a gain of 20. Use a maximum time of 2 s. Notice that while the open-loop responses are very different, the closed-loop responses are nearly identical.

a.
$$G_1(s) = \frac{3.7}{0.75s + 0.6}$$

b.
$$G_2(s) = \frac{1.63}{0.94s + 0.92}$$

c.
$$G_3(s) = \frac{0.7s^2 + 7s + 17}{s^3 + 2s^2 + 5.2s + 4}$$

10.4 The appearances of nonlinear characteristics in systems are very common in practice. Consider a feedback system with an input nonlinearity as shown in Figure 10.48. The nonlinearity obeys the relationship $u(t) = f(e(t))$ where $e(t)$ is the input signal and $u(t)$ is the nonlinear output signal. The plant is given by

$$G_0(s) = \frac{1}{s+1}$$

Find the steady-state response of the system as a function of the magnitude of a step input signal for the three controller gains $k = 1, 10,$ and 100 for the following nonlinear functions:

a.
$$f(e) = 0.2(e^3 - e)$$

b.
$$\begin{aligned} f(e) &= e + \sin(e) \quad & |e| > 1 \\ &= 0 & |e| \le 1 \end{aligned}$$

c.
$$f(e) = \tan^{-1}(e)$$

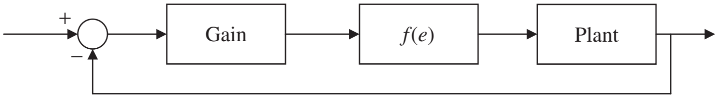


Figure 10.48 Block diagram for simple system with a nonlinear input.

[11] R. Jurgen, *Electronic Engine Control Technologies*, SAE International, Troy, MI, 1999.

Sections 10.3 and 10.4

10.5 Consider the flexible drive-shaft system discussed in Section 10.5.1 with configuration shown in Figure 10.27. As mentioned in that section, it is difficult to know the exact frequency of the resonant mode. The question is whether it is better to overestimate or underestimate this frequency. To investigate this, plot together the Bode plots of the uncompensated pointer with the following two compensators:

$$\text{Notch filter 1:} \quad \frac{(s + 3 + 28i)(s + 3 - 28i)}{(s + 60)^2}$$

$$\text{Notch filter 2:} \quad \frac{(s + 3 + 34i)(s + 3 - 34i)}{(s + 60)^2}$$

Set the gain of each notch filter to 1 before plotting and draw your conclusions from the plotted results.

Section 10.4

10.6 Consider the guided missile in Figure 10.49. The lateral force of the air rotates a missile about its center of gravity. The force applied by the air can be considered a point force applied to the missile's center of pressure. If this center of pressure is ahead of the center of mass, the guided missile is unstable.

The input to the system is the angle of thrust $\psi(t)$ and the output of the system is $\theta(t)$. The force applied by the air drag can be modeled as $F_d = k_d \sin(\theta)$, where k_d depends on the shape and velocity of the rocket. The off-axis force applied by the rocket engines is given by $F_r \sin(\psi)$. The other relevant parameters are l_2, the distance from the rocket engine to the center of mass of the missile; l_1, the distance from the center of mass of the missile to the center of pressure; and J, the rotational inertia of the rocket.

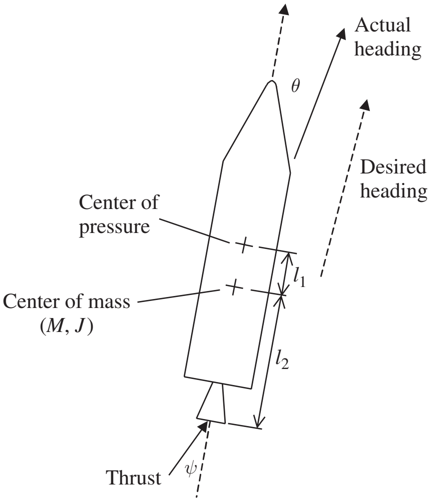


Figure 10.49 Attitude control of a missile.

For a fixed k_d and F_r, the effective transfer function from ψ to θ, when ψ to θ are small, is[12]

$$G(s) = \frac{l_2 F_r/J}{s^2 - l_1 k_d/J}$$

Assume that $l_2 F_r/J = l_1 k_d/J = 9$ is an operating point of interest.

a. Using a lead-control structure

$$C(s) = k\,\frac{s + z}{z + p}$$

find the values of k, z, and p such that the closed-loop system response is stable.

b. As the velocity of the rocket changes, k_d changes. For the fixed controller designed in (a), use a root-locus plot to determine the range of k_d for which the system remains stable.

10.7 Consider the design of a cruise controller for an automobile. The car is modeled as a mass with a damper that limits the forward motion. The open-loop transfer function $G(s)$ from the engine throttle angle to the car's velocity is

$$G(s) = \frac{1}{ms + b}$$

The controller $C(s)$ is PI, with a transfer function

$$C(s) = k_p + k_i/s$$

Assume that the car's mass m is 1,200 kg and the friction coefficient b is 70 Ns/m. The grade (slope) of the road acts as a disturbance into the plant. The block diagram for this system is shown in Figure 10.50. The desired speed is $r(t)$, the input for the motor (a throttle angle) is $u(t)$, and the speed of the vehicle is $y(t)$ as measured by a speedometer. The disturbance $d(t)$ represents the effect of the road grade.

a. The transfer function from the disturbance $d(t)$ to the speed $y(t)$ is

$$G_{dy}(s) = \frac{G(s)}{1 + C(s)G(s)}$$

where $G(s)$ is the open-loop response and $C(s)$ is the PI transfer function defined previously. Plot the step response of the open-loop system and closed-loop system on the same figure for $k_p = 100$ and $k_i = 0$. Note that as the gain is increased, the disturbance is reduced.

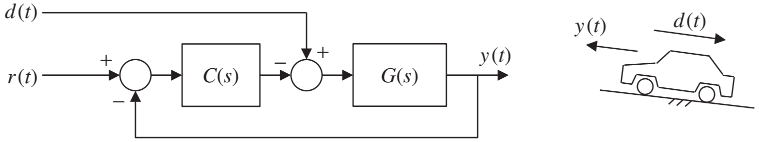


Figure 10.50 Block diagram of the automobile cruise control.

[12] M. Driels, *Linear Control Systems*, McGraw-Hill, New York, 1996.

b. The transfer function from the velocity command $r(t)$ to speed $y(t)$ is

$$G_{ry}(s) = \frac{C(s)G(s)}{1 + C(s)G(s)}$$

Plot the step response for $k_p = 100$ and $k_i = 0$.

c. Plot the steady-state output for the systems in (a) and (b) to a step response as a function of the proportional gain P in the range 50–150. It is seen that the integrator eliminates the steady-state error, and its speed is determined by the gain k_i.

10.8 An automotive suspension is typically passive, being composed of springs and dampers. To improve the suspension of cars, active suspension systems have been proposed. In Figure 10.51, we have depicted a simplified model of an active automobile suspension system in which $r(t)$ represents the input of the road surface and $y(t)$ is the vertical position of the passenger compartment. Assume that the mass of the tire is negligible and that velocity feedback is used so that $u(t) = Cdy(t)/dt$. The actuator applies a force to the rod and the passenger compartment that is proportional to their relative velocity: $C(dy/dt - dr/dt)$. The velocity transfer function from the road $r(t)$ to the ride $y(t)$ is

$$G(s) = \frac{(c + b)s + k}{ms^2 + (c + b)s + k}$$

a. Use `rlocus` and `rlocfind` to determine the value of c needed to set the damping ratio to 1, that is, to make the system critically damped. To use `rlocus`, notice that the denominator can be written as $ms^2 + bs + k + cs$. Hence, it is now in standard from for `rlocus`, with the numerator [1 0] and the denominator [m b k]. The value determined by `rlocfind` will be c. Place the crosshairs at the point where the closed-loop poles first cross the real axis.

b. Plot the step response and Bode diagrams for the system with and without active control ($c = 0$). Assume that $m = 5{,}000$ kg, $k = 8 \times 10^5$ N/m, and $b = 12{,}000$ Ns/m.

c. In this problem, the spring is accounted for in k. For k ranging between 4×10^5 and 10×10^5 N/m, create a meshed surface of the step response using the design in (b). Which car would you prefer to ride in?

10.9 Consider the task of designing an autopilot for a large, slowly moving ship in which the output of a compass provides the feedback. The controller sends commands to a rudder mechanism, which, with delay, turns it to the desired position, thereby turning the ship. The

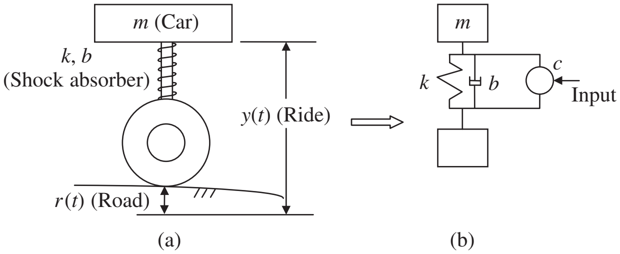


Figure 10.51 (a) A simplified model of the quarter-car model with an active suspension, and (b) mass and spring equivalent.

following equations have been linearized from Nomoto's equation[13] for a ship at cruising speed. The open-loop transfer function of the steering system without the controller is[14]

$$G(s) = \frac{s + 0.03}{s(s + 0.09)(s + 0.04)(s - 0.0004)}$$

a. The plant has an unstable pole. Plot the root locus of the steering system.

b. Using the lead-control structure

$$C(s) = k\frac{s + z}{z + p}$$

find the values of k, z, and p that stabilize the closed-loop system response while maintaining less than 30% overshoot.

c. Using a new sensor that provides velocity information, one can now use PD control. Thus,

$$C(s) = k_p + sk_d$$

Find k_p and k_d such that the closed-loop response is stable, has less than 5% overshoot, and has a settling time less than 275 s.

10.10 A recent model automobile has a catalytic converter to meet the exhaust-emission-performance standards. The catalytic converter requires tight control of the engine air/fuel ratio (A/F), the ignition-spark timing, and exhaust-gas recirculation. We consider the air/fuel ratio regulation task. The transfer function of the carburetor with the effective A/F ratio as output is[15]

$$G(s) = \frac{4e^{-Ts}}{s + 4}$$

where the time delay T is 0.2 s. The function `pade` may be used to generate an approximation of the time delay, or one may set the output delay property of the transfer function to 0.2. However, using `pade` is less difficult, because it is one of MATLAB's few functions that supports time delay.

a. Suppose that the time delay is neglected in the design of the controller and we let the controller be a PI controller so that

$$C(s) = k_p + \frac{k_i}{s}$$

Set $k_i = 2$ and select the value of k_p so that the rise time for the unit-step response is smaller than 0.4 s. Determine the step response of the system.

b. Consider feeding a time-delayed signal back into the controller as shown in Figure 10.52. The extra compensation element in the controller contains a

[13] *Ibid*.

[14] C. L. Phillips and R. D. Harbor, *Feedback Control Systems*, 3rd ed., Prentice Hall, Englewood Cliffs, NJ, 1996.

[15] B. Kuo, *Automatic Control Systems*, 7th ed., Prentice Hall, Englewood Cliffs, NJ, 1995, p. 815.

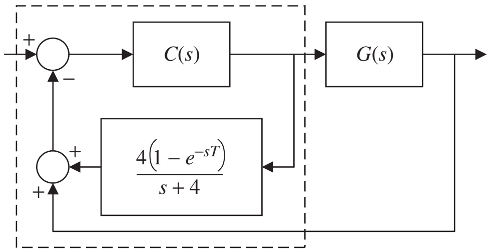


Figure 10.52 Exhaust emission control system using a Smith predictor.

model of the plant and its time delay and is called a Smith predictor. Using a lead-control structure

$$C(s) = k\frac{s + z}{z + p}$$

find the values of k, z, and p that stabilize the closed-loop system response while having no overshoot and a rise time less than 0.2 s. Compare the results to the PI controller in (a).

c. Suppose that the time delay and the plant are not modeled correctly. Determine the step responses for system time delays of 0.3 and 0.1 s with the controller generated in (b) and a plant model with a DC gain of 1.2 and a pole at -5 instead of -4. From the results, is it better to overestimate or underestimate the time delay?

10.11 Consider an automatic motorized bicycle whose block diagram is shown in Figure 10.53. An inclinometer detects the angle of the bicycle body from vertical, $y(t)$. The inclinometer's output is compared to the desired angle from vertical $r(t)$ and the error is input to the controller to generate a steering signal. Any disturbances to the system $d(t)$ are modeled as entering with the input to the bicycle. The transfer functions for the blocks shown in Figure 10.53 are

$$G(s) = \frac{9}{s^2 + 9}$$

$$F(s) = \frac{\omega^2}{s^2 + 2\xi\omega s + \omega^2}$$

$$C(s) = k\frac{s + z}{s + p}$$

where $C(s)$ is a lead controller.

a. A micromachined inclinometer has a settling time of 0.2 s and a bandwidth of 125 Hz. The sensor's parameters are $\omega = 250\pi$ and $\xi = 20/\omega$. Find k, z, and p such that the overshoot for a unit step response is less than 20% and the setting time is less than 4 s.

b. Suppose that there is another type of inclinometer to choose, one that is based on the principle of a pendulum. Its resonant frequency is 7.4 Hz ($\omega = 14.8\pi$) and its damping coefficient $\xi = 0.4$. Can this inclinometer also be used?

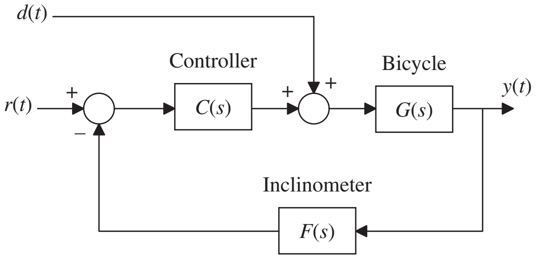


Figure 10.53 Block diagram of an automatic motorized bicycle.

BIBLIOGRAPHY

D. K. Anand and R. B. Zmood, *Introduction to Control Systems*, Butterworth and Heinmann, Ltd., Oxford, England, 1995.

E. Chowanietz, *Automobile Electronics*, SAE International, Troy, MI, 1995.

R. Dorf and R. Bishop, *Modern Control Systems*, Addison-Wesley Publishing, Reading, MA, 1997.

M. Driels, *Linear Control System Engineering*, McGraw-Hill, New York, 1996.

G. Franklin, J. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, 3rd ed., Addison-Wesley, Reading, MA, 1994.

B. Friedland, *Advanced Control System Design*, Prentice Hall, Englewood Cliffs, NJ, 1996.

R. Jurgen, Electronic Engine Control Technologies, SAE International, Troy, MI, 1999.

B. Kuo, *Automatic Control Systems*, Prentice Hall, Englewood Cliffs, NJ, 1995.

W. Levine, *The Control Handbook*, CRC Press, Boca Raton, FL, 1996.

N. Nise, *Control Systems Engineering*, Addison-Wesley, Reading, MA, 1995.

U. Ozguner, H. Goktas, and H. Chan, “Automotive Suspension Control Through a Computer Communication Network,” *Proceedings of 1st IEEE Conference on Control Application*, vol. 2, 1992, pp. 895–900.

C. Phillips and R. Harbor, *Feedback Control Systems*, Prentice Hall, Englewood Cliffs, NJ, 1996.

11

Fluid Mechanics

James H. Duncan

Several types of applications in fluid mechanics and aerodynamics are analyzed, and several flow fields are presented using a variety of visualization techniques.

11.1 HYDROSTATICS

In hydrostatics, the pressure is constant on any surface of constant height in a single fluid, and the pressure varies with height according to

$$\frac{dP}{dz} = -\rho g \tag{11.1}$$

where $\rho = \rho(z)$ is the density distribution, $g = 9.81$ m/s^2 is the acceleration of gravity, P is the pressure, and z is the vertical Cartesian coordinate, with positive being up. We now apply Eq. (11.1) to two hydrostatics applications.

11.1.1 Pressure Distribution in the Standard Atmosphere

If the atmosphere is a perfect gas, then

$$\rho = P/(RT)$$

where $R = 287.13$ J/(kg K) is the perfect gas constant, and T is the temperature in degrees Kelvin. With this assumption, Eq. (11.1) can be integrated to obtain

$$P(z) = P_0 \exp\left[-\frac{g}{R}\int_0^z \frac{dz}{T}\right] \tag{11.2}$$

where $P_0 = 101{,}330$ Pa is the pressure at $z = 0$, ground level.

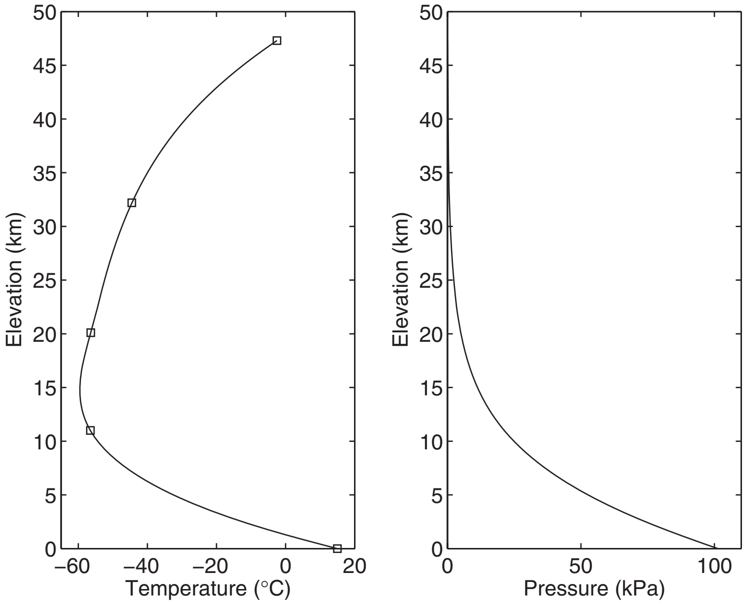
Example 11.1 Temperature and pressure variation as a function of altitude

We assume that the temperature distribution is that represented by fitting a spline to the temperatures at the elevations given in Table 11.1. We shall use Eq. (11.2) and the fitted spline to obtain graphs of P and T as a function of z. The program is as follows.

```
g = 9.81;   P0 = 101330;   R = 287.13;   np = 60;
TC = [15, -56.5, -56.4, -44.5, -2.5];
z = [0, 11000, 20100, 32200, 47300];
zz = linspace(z(1), z(end), np);
t = spline(z, TC+273.15, zz);
subplot(1,2,1)
plot(t-273.15, zz/1000, 'k-', TC, z/1000, 'ks')
axis([-65, 20, 0, 50])
xlabel('Temperature (\circC)')
ylabel('Elevation (km)')
subplot(1,2,2)
P = zeros(np,1);   gR = g/R;
gg = @(h, z, TC) (1./spline(z, TC+273.15, h));
```

TABLE 11.1 Temperature of the Standard Atmosphere as a Function of Elevation

Elevation (m)	Temperature (°C)
0.0	15.0
11,000	–56.5
20,100	–56.4
32,200	–44.5
47,300	–2.5



(a) (b)

Figure 11.1 The standard atmosphere: (a) Temperature versus elevation from Table 11.1 and a spline fitted to these data. (b) Pressure from Eq. (11.2) versus elevation.

```
for k = 1:np;
   Intg = quadl(gg, 0.1, zz(k), [], [], z, TC);
   P(k) = P0*exp(-gR*Intg);
end
plot(P/1000, zz/1000, 'k-')
axis([0, 110, 0, 50])
ylabel('Elevation (km)')
xlabel('Pressure (kPa)')
```

The results of executing this program are shown in Figure 11.1.

11.1.2 Force on a Planar Gate

Consider the reservoir shown in Figure 11.2. One wall of the reservoir is a tiltable metal gate that is hinged at the bottom and has weight W and length L. The width of the reservoir in the direction normal to the page is B. Initially, the gate is vertical and the water level reaches the top of the gate. The total volume of water is $V_w = aLB$. A rod holds the gate closed, and the force of the rod on the gate F_{rod} is directed along the rod. A stop holds the opposite end of the rod in place. This stop can be moved to the right, thus letting the gate rotate clockwise about its hinge. For

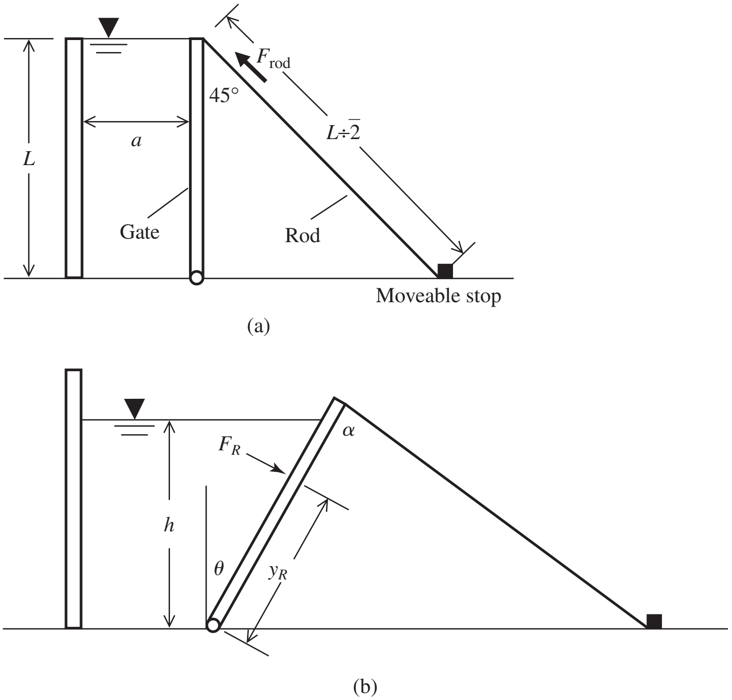


Figure 11.2 Reservoir with tiltable gate: (a) Gate vertical. (b) Gate opened to an angle θ.

gate angle θ less than or equal to some critical angle $\theta_{\max}$, the water level is at or below the top of the gate; however, for $\theta > \theta_{\max}$, the water spills over the top.

The volume V bounded by the bottom, the fixed walls, the gate, and a level surface at the top of the gate is

$$\frac{V}{V_w} = \cos\theta + \frac{L}{2a}\cos\theta\sin\theta \tag{11.3}$$

The maximum volume $V_{\max}/V_w$ occurs at θ_V. The water will spill over the dam when

$$\frac{V}{V_w} < 1.0$$

which corresponds to the critical angle $\theta_{\max}$ and is determined from

$$\cos\theta_{\max} + \frac{L}{2a}\cos\theta_{\max}\sin\theta_{\max} - 1 = 0$$

An expression relating the water level h to the gate angle θ is obtained by equating V_w to the water volume at any θ. Thus,

$$\frac{V_w}{B} = aL = ah + 0.5h^2\tan\theta \tag{11.4}$$

or

$$h = \frac{-a}{\tan\theta}\left[1 \mp \sqrt{1 + \frac{2L}{a}\tan\theta}\right]$$

for $\theta < \theta_{\max}$. Only the positive root is of interest.

The magnitude of F_{rod} is obtained by taking moments about the hinge. Thus,

$$F_{\text{rod}} = \frac{F_R y_R + 0.5WL\sin\theta}{L\sin\alpha} \tag{11.5}$$

where F_R is the total force of the water on the gate and y_R is the distance from the hinge to the center of pressure. The angle α shown in Figure 11.2b is given by

$$\alpha = \theta + \cos^{-1}\left(\cos\theta/\sqrt{2}\right).$$

From the hydrostatics equations, we find that

$$F_R = \frac{B\rho g h^2}{2\cos\theta}$$

$$y_R = \frac{h}{2\cos\theta} - \frac{2I_{xx}\cos^2\theta}{Bh^2}$$

where

$$I_{xx} = \frac{Bh^3}{12\cos^3\theta}$$

is the second moment of area of the submerged portion of the gate about its centroid. Thus, Eq. (11.5) becomes

$$F_{\text{rod}} = \frac{1}{L\sin\alpha}\left[\frac{B\rho g h^3}{6\cos^2\theta} + \frac{WL}{2}\sin\theta\right] \tag{11.6}$$

The angle at which F_{rod} is a minimum is denoted θ_F.

We shall now use these results to determine $\theta_{\max}$, θ_V, θ_F, and F_{rod}.

Example 11.2 Properties of a reservoir

We shall determine $\theta_{\max}$, θ_V, and θ_F and plot the volume ratio V/V_w, the depth of the water h, and the force F_{rod} for $0 \le \theta \le \theta_{\max}$. We assume that $L = 10$ m, $a = 5$ m, $B = 10$ m, and $W = 100{,}000$ N. We use `fzero` to determine $\theta_{\max}$ and `fminbnd` to determine the values of θ_V, and θ_F. The program that performs these calculations is as follows.

```
function Example11_2
a = 5.0;    L = 10.0;    B = 10.0;    rho = 1000;
```

```
g = 9.81;    W = 100000;    La = L/a;
theta = linspace(0.0, pi/2, 100);
VVw = @(theta,La) (cos(theta)+0.5*La*cos(theta).*sin(theta));
figure(1)
subplot(1,2,1)
plot(theta*180/pi, VVw(theta,La), 'k')
axis([0.0, 90.0, 0.0, 1.5])
ylabel('V/V_w')
xlabel('\theta (\circ)')
grid on
hold on
MaxTheta = @(theta, La) (1-VVw(theta, La));
ThetaMaxDeg = fzero(MaxTheta, [0.01, pi/2.0], [], La)*180/pi;
plot(ThetaMaxDeg, 1.0, 'sk')
text(10, 0.95, ['\theta_{max}= ', num2str(ThetaMaxDeg, 4) ' circ'])
VVwNeg = @(theta, La) (-VVw(theta, La));
[ThetaMaxVol, VVwMax] = fminbnd(VVwNeg, 0.0, ThetaMaxDeg*pi/180, [], La);
plot(ThetaMaxVol*180/pi, -VVwMax, 'ks')
text(10, -VVwMax+0.13, ['V_{max}/V_w = ' num2str(-VVwMax, 4)])
text(10, -VVwMax+0.05, [' \theta_V = ' num2str(ThetaMaxVol*180/pi, 4) . . .
       ' \circ '])
subplot(1,2,2)
theta = linspace(0.01, ThetaMaxDeg*pi/180);
plot(theta*180.0/pi, h(theta, La, a), 'k-')
ylabel('h (m)')
xlabel('\theta (\circ)')
grid on
figure(2)
plot(theta*180.0/pi, Frod(theta, L, B, rho, g, W, La, a)*10^-6, 'k-')
hold on
[FrodThetaMin, FrodMin] = fminbnd(@Frod, 0.0, ThetaMaxDeg*pi/180, [], . . .
             L, B, rho, g, W, La, a);
plot(FrodThetaMin*180/pi, FrodMin*1e-6, 'ks')
text(FrodThetaMin*180/pi, FrodMin*1e-6-0.1, ['F_{rod, min} = ' . . .
       num2str(FrodMin*1e-6, 4) ' MN'])
text(FrodThetaMin*180/pi, FrodMin*1e-6-0.22, ['\theta_F = ' . . .
       num2str(FrodThetaMin*180/pi, 4) ' \circ'])
ylim([0.4, 2.4])
ylabel('F_{rod} (MN)')
xlabel('\theta (\circ)')
grid on

function Fr = Frod(theta, L, B, rho, g, W, La, a)
D = (L*sin(theta+acos(cos(theta)/sqrt(2))));
Fr = ((B*rho*g*h(theta, La, a).^3)./(6*cos(theta).^2)+0.5*W*L*sin(theta))./D;

function f = h(theta, La, a)
f = (-a./tan(theta).*(1-sqrt(1+2*La*tan(theta))));
```

The execution of this program results in Figures 11.3 and 11.4.

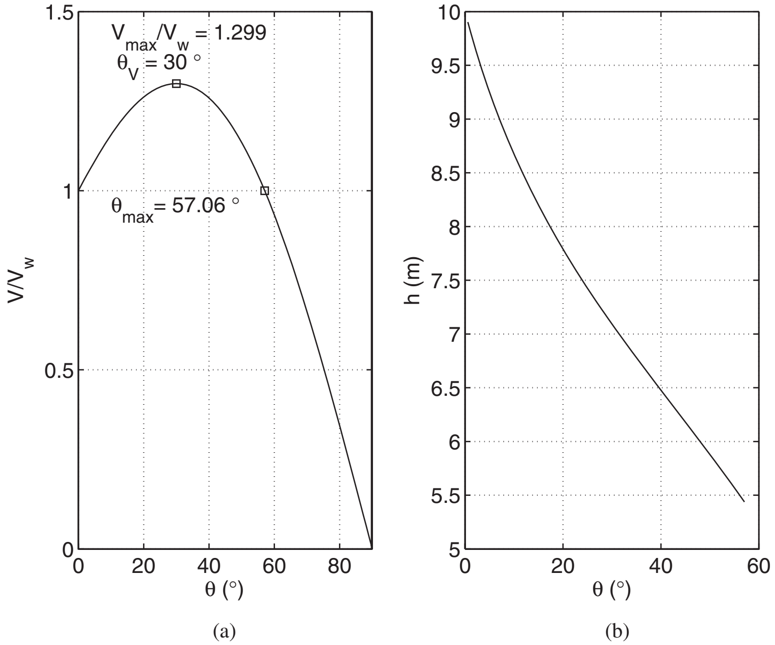


Figure 11.3 Results for the hinged gate configuration: (a) V/V_w versus θ. (b) h versus θ.

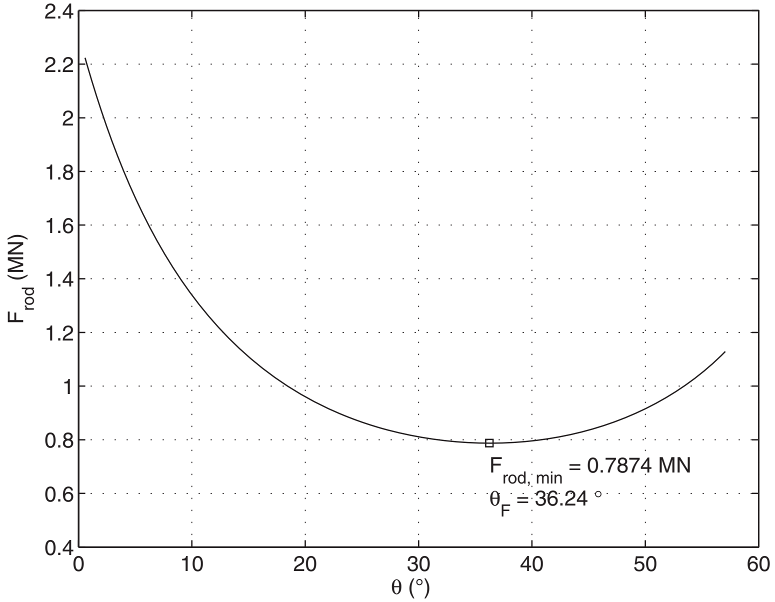


Figure 11.4 Force needed to keep the gate closed versus θ.

11.2 INTERNAL VISCOUS FLOW

There is a large class of flow applications that concern laminar and turbulent viscous flow in pipes and ducts. Several of these applications are given below. For low Reynolds numbers, the flow is laminar, and pdepe is used to compute the flow field in the pipe. For higher Reynolds numbers, the flow is turbulent, and the flow and pressure drop are computed with the aid of the Colebrook equation.[1]

11.2.1 Laminar Flow in a Horizontal Pipe with Circular Cross Section

The differential equation for unsteady, axially symmetric, incompressible, fully developed, laminar flow in a circular pipe is

$$\frac{\partial u_z}{\partial t} = -\frac{1}{\rho}\frac{dP}{dz} + \nu\left(\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u_z}{\partial r}\right)\right) \tag{11.7}$$

where r and z are the radial and axial coordinates, respectively; t is the time; $u_z = u_z(r, t)$ is the axial velocity; ρ is the fluid density; dP/dz is the axial pressure gradient, which can be function of time; and ν is the kinematic viscosity of the fluid. The axial velocity must satisfy two conditions: (1) the no-slip condition at the pipe wall $r = R$, that is, $u_z(R, t) = 0$; and (2) the symmetry condition at the pipe centerline $r = 0$, that is, $\partial u_2(0, t)/\partial r = 0$.

Equation (11.7) is solved by using pdepe, which is described in Section 5.5.6.

Example 11.3 Laminar flow in a pipe that is started from rest

Unsteady laminar pipe flow is obtained by starting the flow from rest and imposing a constant pressure gradient thereafter. The effect of the no-slip condition on the axial velocity diffuses inward from the pipe walls and reaches the center of the pipe with a timescale of $T = R^2/\nu$. In the long-time limit, the solution for steady flow in the pipe is obtained. We assume that $R = 5.0$ mm, $\nu = 0.00038$ m^2/s, $\rho = 1{,}000$ kg/m^3, and $dP/dz = 1.0 \times 10^6$ Pa/m. We use pdepe with $m = 1$, which signifies a cylindrical coordinate system. We shall determine the solution for $0 \le t \le 0.07$ s. The program is as follows.

```
function Example11_3
nu = 0.00038;   rho = 1000;
dPdz = -1e6;   nr = 100;   rmax = 0.005;
nt = 15;   tmax = 0.07;
r = linspace(0, rmax, nr);
t = linspace(0, tmax, nt);
u = pdepe(1, @pdPipe, @pdPipeIC, @pdPipeBC, r, t, [], nu, rho, dPdz);
hold on
for ijd = [2, 3, 4, 6, nt]
  plot(u(ijd,:), r*1000, 'k')
  if ijd == nt
    text(u(ijd,20), rmax*0.25*1000, [num2str(tmax*1000, 4) ' ms'])
  elseif ijd == 2
    text(u(ijd,20), rmax*0.25*1000, ['t = ' num2str(t(ijd)*1000, 4) ' ms'])
  else
    text(u(ijd,20), rmax*0.25*1000, [num2str(t(ijd)*1000, 4) ' ms'])
  end
```

[1] C. E. Colebrook, "Turbulent flow in pipes with particular reference to the transition region between smooth and rough pipe laws," *Journal of the Institute of Civil Engineers*, London, **11**, 1939, pp. 133–156.

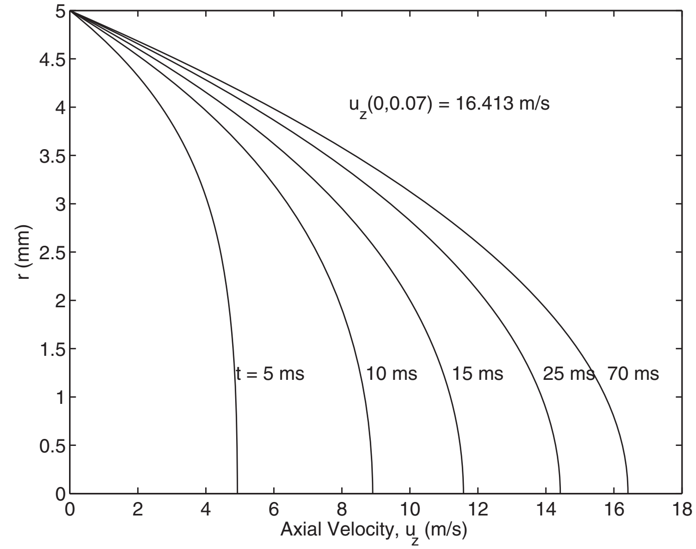


Figure 11.5 Development of the velocity profile for laminar flow started from rest by an applied pressure gradient in a pipe.

```
end
xlabel('Axial Velocity, u_z (m/s)')
ylabel('r (mm)')
text(0.5*u(nt,1), 0.8*rmax*1000, ['u_z(0,' num2str(t(nt)) ') = '...
    num2str(u(nt,1),5)' m/s'])

function [c,f,s] = pdPipe(r, t, u, DuDr, nu, rho, dPdz)
c = 1.0/nu;
f = DuDr;
s = -dPdz/(rho*nu);

function u0 = pdPipeIC(r, nu, rho, dPdz)
u0 = 0;

function [pl, ql, pr, qr] = pdPipeBC(rl, ul, rr, ur, t, nu, rho, dPdz)
pl = 1;   ql = 0;
pr = 0;   qr = ur;
```

Upon execution, we obtain the results shown in Figure 11.5. In steady flow, the analytical solution for the centerline velocity is $R^2|dP/dz|/(4\rho\nu) = 16.45$ m/s; from Figure 11.5, it is seen that at $t = 0.07$, s = 1.064T, the value is 16.41 m/s. If we had increased the upper limit of the time, the numerically determined centerline velocity would be still closer to the theoretical steady-flow value.

11.2.2 Downward Turbulent Flow in a Vertical Pipe

Consider a vertically oriented smooth pipe of length L and diameter D in which a fluid of density ρ and kinematic viscosity ν is flowing downward, as shown in Figure 11.6. For a particular flow rate, the pressure drop due to the downward flow of the fluid is balanced by the pressure gain due to gravity—that is, at this flow rate, the static pressure in the pipe is independent of the distance along the pipe.

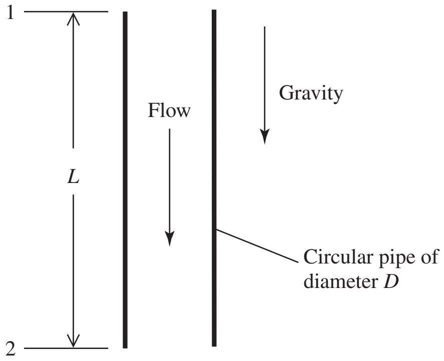


Figure 11.6 Flow in a vertically oriented pipe of diameter D.

The head loss equation is

$$\frac{P_1}{\rho g} + \frac{V_1^2}{2g} + z_1 = \frac{P_2}{\rho g} + \frac{V_2^2}{2g} + z_2 + \frac{\lambda L V^2}{2gD} \tag{11.8}$$

where P is the pressure, V is the average flow speed, z is the height, and λ is the friction factor. In the present problem, $P_1 = P_2$, $V_1 = V_2$, and $z_1 - z_2 = L$; therefore, Eq. (11.8) reduces to

$$\lambda = \frac{2gD^3}{\nu^2 \mathrm{Re}^2} \tag{11.9}$$

where Re is the Reynolds number, defined as

$$\mathrm{Re} = \frac{VD}{\nu} \tag{11.10}$$

The Colebrook formula for λ versus Re in pipes of varying roughness factors k/D is given by (recall Exercise 5.17)

$$\frac{1}{\sqrt{\lambda}} = -2\log_{10}\left(\frac{2.51}{\mathrm{Re}\sqrt{\lambda}} + \frac{k/D}{3.7}\right) \qquad \mathrm{Re} \geq 4000 \tag{11.11}$$

In the present case, $k = 0$. Substituting λ from Eq. (11.9) into Eq. (11.11) yields a transcendental equation for the Reynolds number of the pipe flow and, therefore, the desired flow rate.

We shall now illustrate these results with the following example.

Example 11.4 Flow rate in a pipe

Consider a vertically oriented smooth pipe of diameter $D = 4.0$ cm in which water of density $\rho = 1000.0$ kg/m^3 and kinematic viscosity $\nu = 1.2 \times 10^{-6}$ m^2/s is flowing downward. We shall find the flow rate $Q = \pi D^2 V/4$ for which the pressure drop due to the downward flow is balanced by the pressure gain due to gravity. The Colebrook expression is evaluated in **ColebrookFriction**. The program is as follows.

```
function Example11_4
D = 0.04;   g = 9.81;   nu = 1.2e-6;
```

```
Re = fzero(@ColebrookFriction, [1e3, 1e7], [], nu, g, D);
disp(['Re = ', num2str(Re, 7)])
disp(['Flow Rate = ' num2str(pi*D*Re*nu/4, 4) ' m^3/s'])

function value = ColebrookFriction(Re, nu, g, D)
lambda = 2*g*D^3/(nu*Re)^2;
value = 1/sqrt(lambda)+2*log10(2.51/(Re*sqrt(lambda)));
```

which when executed displays to the command window

```
Re = 240405.8
Flow Rate = 0.009063 m^3/s
```

11.2.3 Three Connected Reservoirs

Consider the classical three-reservoir problem[2] in which three reservoirs of different elevations are connected to a common junction at location J, as shown in Figure 11.7. If we are given the length L_j, diameter d_j, and roughness k_j of the pipes meeting at J and the elevations of each reservoir h_j, then we can determine the corresponding flow rates Q_j and direction of flow in each pipe. The method is as follows. If an open-ended tube were installed at the junction, then the water's elevation in the tube would rise to h_p, which is unknown. The difference between the elevations at P and J is the pressure head at the junction. Secondly, at J, the sum of the flows from each pipe must be zero—that is,

$$\sum_{j=1}^{3} Q_j = 0 \tag{11.12}$$

with a positive value of Q_j indicating flow toward the junction and a negative value indicating flow out of the junction.

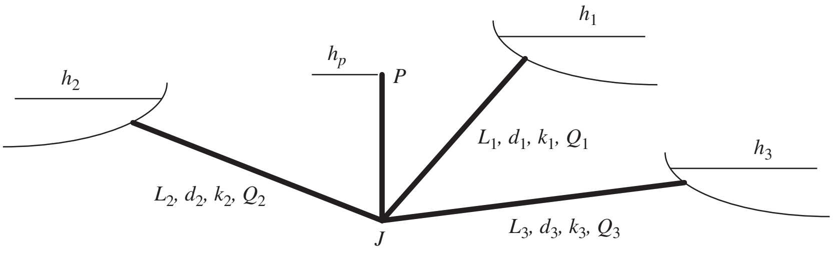


Figure 11.7 Pipes connecting three reservoirs at junction J.

[2] N. H. C. Hwang and C. E. Hita, *Fundamentals of Hydraulic Engineering Systems*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1987, pp. 106–110.

The flow in each pipe is determined from

$$Q_j = 0.25\pi d_j^2 V_j s_j \quad j = 1, 2, 3$$

where

$$V_j = s_j\sqrt{\frac{2gd_j|\Delta h_j|}{\lambda_j L_j}} \qquad \Delta h_j = h_j - h_p \quad j = 1, 2, 3$$

and s_j is the sign of Δh_j, $g = 9.81$ m/s^2 is the gravity constant, and λ_j is the pipe friction coefficient as determined from Eq. (11.11) and is a function of Re$_j$; that is [recall Eq. (11.10)],

$$\text{Re}_j = \frac{V_j d_j}{\nu} \qquad j = 1, 2, 3$$

where $\nu = 1.002 \times 10^{-6}$ m^2/s is the kinematic viscosity of water at 20°C. Thus, the objective is to determine the value of h_p that satisfies Eq. (11.12).

The solution is obtained as follows. We assume a value for h_p, which we know lies somewhere between the minimum and maximum values of h_j. Then we compute a value for each V_j by first assuming a value for λ_j, which is obtained from Eq. (11.11) for very large Re; that is,

$$\lambda_j = \left[2\log_{10}\left(3.7\frac{d_j}{k_j}\right)\right]^{-2}$$

We use these values of V_j to determine values for Re$_j$, which are then used to determine λ_j from the general Colebrook formula given by Eq. (11.11). We continue this process until the values of V_j are within an acceptable tolerance.

After each V_j has been determined, we compute each Q_j and determine whether Eq. (11.12) has been satisfied. If it hasn't, then another value of h_p is selected, and new values of V_j are computed as just described. It should be noted that when $\Delta h_j = 0$, $Q_j = 0$ and the value of λ_j cannot be computed since Re$_j = 0$. In the program that implements this procedure, we use nested applications of `fzero`, an inner one to determine V_j and an outer one to determine h_p.

We now illustrate this procedure with the following example.

Example 11.5 Flow rates from three connected reservoirs

Using the solution method described above, we shall determine the flow rates for the values that are given in Table 11.2. We create three subfunctions: **ReservoirSumQ**, which determines Q_j and evaluates Eq. (11.12); **PipeFrictionCoeff**, which evaluates

TABLE 11.2 Parameters for Reservoir in Figure 11.7

Reservoir	d_j(m)	L_j(m)	k_j(m)	h_j(m)
1	0.30	1000	0.00060	120
2	0.50	4000	0.00060	100
3	0.40	2000	0.00060	80

Eq. (11.11); and **ResFriction**, which determines λ_j at each value of V_j. The script is as follows

```
function Example11_5
d = [0.3, 0.5, 0.4];    el = [1000, 4000, 2000];
k = [0.6, 0.6, 0.6]*1e-3;    h = [120, 100, 80];
hg = fzero(@ReservoirSumQ, 110, [], d, el, k, h);
[sq, q] = ReservoirSumQ(hg, d, el, k, h);
disp(['Elevation h_sub_p = ' num2str(hg) ' m'])
disp(['Q1 = ' num2str(q(1)) ' m^3/s Q2 = ' num2str(q(2)) . . .
        ' m^3/s    Q3 = ' num2str(q(3)) ' m^3/s'])

function [sq, q] = ReservoirSumQ(hg, d, el, k, h)
cv = 2*9.81*d./el;    ro = d/1.002e-6;
dk = d./k;    qd = 0.25*pi*d.^2;
frictguess = (2*log10(3.7*dk)).^-2;
hh = h-hg;
for n = 1:length(d)
  if hh(n) == 0
     q(n) = 0;
   else
     lambda = fzero(@ResFriction, frictguess(n), [], dk(n), hh(n), cv(n), ro(n));
     q(n) = sign(hh(n))*sqrt(cv(n)*abs(hh(n))/lambda)*qd(n);
   end
end
sq = sum(q);

function x = PipeFrictionCoeff(el, re, dk)
if dk>100000|dk == 0
   x = el-(2*log10(re*sqrt(el)/2.51))^-2;
else
   x = el-(2*log10(2.51/re/sqrt(el)+0.27/dk))^-2;
end

function lamb = ResFriction(lambda, dk, dh, cv, ro)
ren = sqrt(cv*abs(dh)/lambda)*ro;
lamb = PipeFrictionCoeff(lambda, ren, dk);
```

Executing the program gives

```
Elevation h_sub_p = 98.904 m
Q1 = 0.16185 m^3/s    Q2 = 0.068728 m^3/s    Q3 = -0.23058 m^3/s
```

11.3 EXTERNAL FLOW

11.3.1 Boundary Layer on an Infinite Plate Started Suddenly from Rest

Consider a layer of liquid of thickness h that extends to infinity in the x-z plane and is bounded by a rigid plate at $y = 0$ and a free surface at $y = h$. The plate and the fluid are initially at rest. At $t = 0$, the plate begins to move in the x direction. The resulting fluid motion is only in the x-direction, and is a function of only time and the y coordinate, that is, $u = u(y, t)$.

The solution is obtained by solving the x-component of the Navier-Stokes equations, which in the present case reduces to

$$\frac{\partial u}{\partial t} = \nu_{\text{vis}} \frac{\partial^2 u}{\partial y^2} \tag{11.13}$$

The initial condition is

$$u(y, 0) = 0$$

The boundary condition at the surface of the plate is the no-slip condition,

$$u(0, t) = U$$

while the boundary condition at the free surface $y = h$ is zero shear stress—that is,

$$\nu_{\text{vis}} \left.\frac{du}{dy}\right|_{y=h} = 0$$

A variant of this problem is obtained by prescribing the stress, rather than the velocity, at $y = 0$; that is,

$$\tau(0, t) = \rho\nu_{\text{vis}} \left.\frac{\partial u}{\partial y}\right|_{y=0}$$

Example 11.6 Acceleration of a liquid layer

Consider a case with $h = 10.0$ cm and $\nu_{\text{vis}} = 1.0$ cm^2/s, and where the plate and the fluid are initially at rest. At $t = 0$, the plate is instantaneously accelerated to a speed $U = 5.0$ cm/s in the positive x-direction. The following program uses pdepe with $m = 0$, which indicates that a Cartesian coordinate system is used.

```
function Example11_6
nu = 1.0;    Uplate = 5.0;
ny = 100;    ymax = 10.0;
nt = 40;    tmax = 100.0;
A = tmax/((nt+1)^2);
y = linspace(0, ymax, ny);
t = A*((1:nt)+1).^2;
u = pdepe(0, @pdfslpde, @pdfslic, @pdfslbc, y, t, [], nu, Uplate);
hold on
for ijd = 2:nt
  plot(u(ijd,:), y, 'k-')
end
xlabel('Horizontal Velocity (cm/s)')
ylabel('y (cm)')
text(y(ny/2), u(nt,ny/2), ['t = ' num2str(tmax, 4) ' s'])
xlim([0, 6])
```

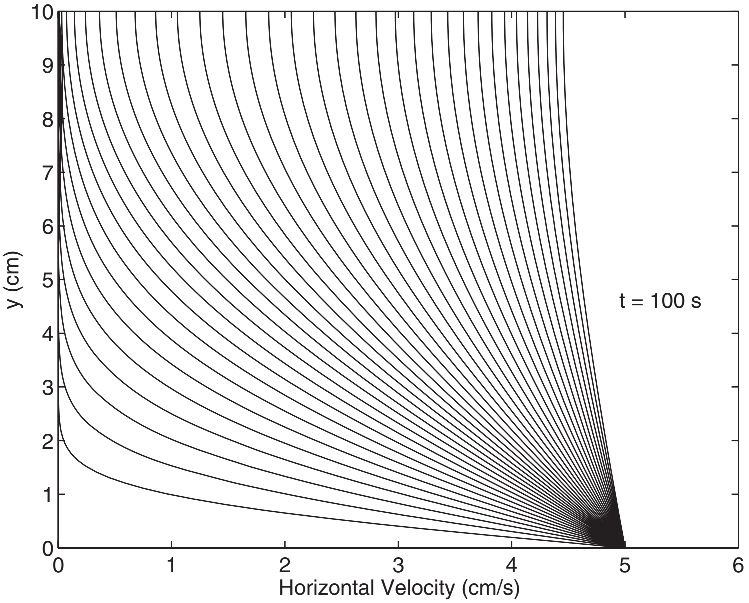


Figure 11.8 Horizontal velocity in a fluid layer of depth 10.0 cm that is suddenly accelerated to a speed $U = 5.0$ cm/s.

```
function [c, f, s] = pdfslpde(x, t, u, DuDy, nu, Uplate)
c = 1/nu;    f = DuDy;
s = 0;
function u0 = pdfslic(y, nu, Uplate)
u0 = 0;
function [pl, ql, pr, qr] = pdfslbc(yl ,ul, yr, ur, t, nu, Uplate)
pl = ul-Uplate;    ql = 0;
pr = 0;    qr = 1;
```

Executing this program results in Figure 11.8. As seen from the figure, after a long time ($t > h^2/\nu_{\text{vis}} = 100$ s), the velocity distribution of the liquid layer tends to be a constant value equal to the velocity of the plate.

11.3.2 Blasius Boundary Layer

The incompressible flow field in a laminar boundary layer on a flat plate is given by the solution to the boundary layer equations

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{11.14}$$

$$u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \nu_{\text{vis}}\frac{\partial^2 u}{\partial y^2}$$

where x and y are the coordinates parallel to and perpendicular to the plate surface, respectively; u and v are the corresponding fluid velocity components, respectively; and ν_{vis} is the kinematic viscosity. The boundary conditions are that u and v are zero on the plate surface $y = 0$ and that $u \rightarrow U$ as $y \rightarrow \infty$.[3]

A similarity solution is proposed where

$$\frac{df}{d\eta} = \frac{u}{U} \tag{11.15a}$$

and

$$\eta = y\sqrt{\frac{U}{x\nu_{\text{vis}}}} \tag{11.15b}$$

The quantity f is proportional to the stream function of the flow, and $d^2f/d\eta^2$ is proportional to the shear. The similarity solution transforms the boundary layer equation into the ordinary nonlinear differential equation

$$2\frac{d^3f}{d\eta^3} + f\frac{d^2f}{d\eta^2} = 0 \tag{11.16}$$

where at $\eta = 0$

$$f = 0, \qquad \frac{df}{d\eta} = 0 \tag{11.17a}$$

and as $\eta \rightarrow \infty$

$$\frac{df}{d\eta} \rightarrow 1 \tag{11.17b}$$

To solve Eq. (11.16), we transform it to three first-order equations using the definitions

$$f_1 = f, \qquad f_2 = \frac{df}{d\eta}, \qquad f_3 = \frac{d^2f}{d\eta^2} \tag{11.18}$$

[3] Full details can be found in R. L. Panton, *Incompressible Flow*, 2nd ed., John Wiley and Sons, New York, 1996, p. 516.

to obtain

$$\begin{aligned}\frac{df_1}{d\eta} &= f_2 \\ \frac{df_2}{d\eta} &= f_3 \\ \frac{df_3}{d\eta} &= -0.5f_1f_3\end{aligned} \tag{11.19}$$

The boundary conditions at $\eta = 0$ become

$$f_1(0) = f_2(0) = 0 \tag{11.20a}$$

and as $\eta \to \infty$

$$f_2(\eta \to \infty) \to 1 \tag{11.20b}$$

We shall now illustrate these results with the following example.

Example 11.7 Laminar boundary layer on a flat plate

We solve the differential equations in Eq. (11.19) by using bvp4c and approximating infinity with $\eta = 8$. Following the procedure discussed in Section 5.5.4, the program is

```
function Example11_7
solinit = bvpinit(linspace(0, 8, 9), @Blasiusguess);
sol = bvp4c(@Blasius, @Blasiusbc, solinit);
x = linspace(0, 8, 100);
y = deval(sol, x);
plot(x, y(1,:), 'k-', x, y(2,:), 'k--', x, y(3,:), 'k-.')
xlabel('\eta')
ylabel('f, df/d\eta, d^2f/d\eta^2')
legend('f', 'df/d\eta', 'd^2f/d\eta^2')
ylim([0 2.5])
disp(['d^2f(0)/dn^2 = ' num2str(y(3,1))])

function dydx = Blasius(x, y)
dydx = [y(2); y(3); -0.5*y(1)*y(3)];

function res = Blasiusbc(ya, yb)
res = [ya(1); ya(2); yb(2)-1];

function y = Blasiusguess(x)
y(1) = x;
y(2) = x^0.5;
y(3) = 5-x;
```

When this program is executed, we obtain the results shown in Figure 11.9. The shear stress on the plate is given by $\rho U^2 f''(0)/\sqrt{\mathrm{Re}_x}$, where from the program, we find that $f''(0) = 0.3321$.

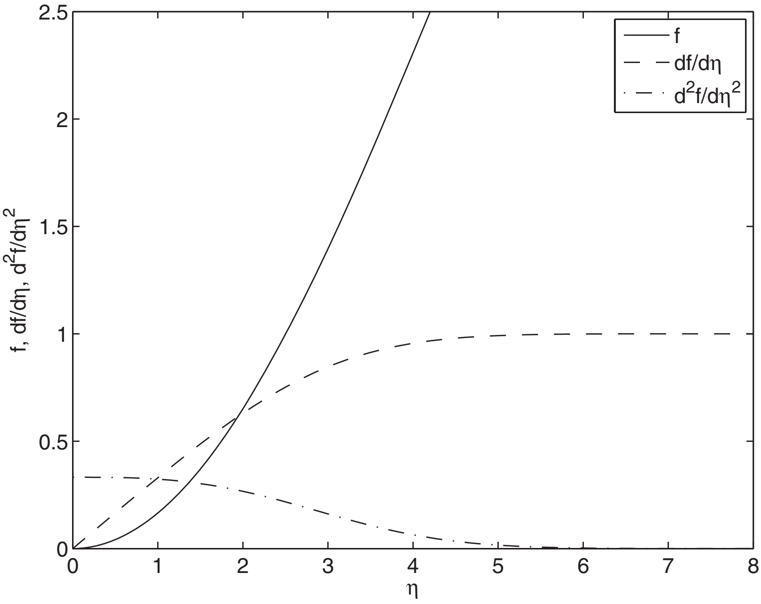


Figure 11.9 Blasius boundary layer profiles of the stream function $f_1 = f$, stream-wise component of velocity $f_2 = df/d\eta$, and the shear $f_3 = d^2f/d\eta^2$.

11.3.3 Potential Flow

In incompressible potential flows, the velocity fields $\vec{u}$ are governed by

$$\nabla \cdot \vec{u} = 0$$

and

$$\nabla \times \vec{u} = 0$$

These conditions dictate that the velocity can be expressed as the gradient of a potential field φ as

$$\vec{u} = \nabla\varphi$$

where φ satisfies Laplace's equation,

$$\nabla^2\varphi = 0 \tag{11.21}$$

An alternative mathematical description for two-dimensional flows is obtained using the stream function ψ, where

$$u = \frac{\partial \psi}{\partial y}$$

$$v = -\frac{\partial \psi}{\partial x}$$

The stream function also satisfies Laplace's equation. Boundary conditions consist of the Neumann conditions, where the component of the velocity normal to a boundary is specified, or the Dirichlet conditions, where the value of φ is specified. At solid boundaries, the Neumann condition is $\vec{u} \cdot \vec{n} = 0$, where $\vec{n}$ is the unit normal to the boundary. In the following examples, several methods for obtaining flow fields for two-dimensional potential flows are discussed. In two of these methods, the flows are constructed by adding together known potentials or stream functions. We now give four such quantities.

Sources and Sinks

$$\varphi_M = \frac{m}{2\pi} \ln r_M \qquad \psi_M = \frac{m}{2\pi}\theta_M$$

$$r_M^2 = (x - x_M)^2 + (y - y_M)^2 \quad \theta_M = \tan^{-1}\frac{y - y_M}{x - x_M}$$

where (x_M, y_M) is the location of the source or sink and m is the source strength.

Doublets (Dipoles)

$$\varphi_M = \frac{K \cos\theta_k}{r_k} \qquad \psi_k = -\frac{K \sin\theta_k}{r_k}$$

$$r_K^2 = (x - x_K)^2 + (y - y_K)^2 \quad \theta_K = \tan^{-1}\frac{y - y_K}{x - x_K}$$

where (x_K, y_K) is the location of the dipole and K is the dipole strength.

Vortices

$$\varphi_\Gamma = \frac{\Gamma}{2\pi}\theta_\Gamma \qquad \psi_\Gamma = -\frac{\Gamma \ln r_\Gamma}{2\pi}$$

$$r_\Gamma^2 = (x - x_\Gamma)^2 + (y - y_\Gamma)^2 \quad \theta_\Gamma = \tan^{-1}\frac{y - y_\Gamma}{x - x_\Gamma}$$

where (x_Γ, y_Γ) is the location of the vortex and Γ is the vortex strength.

Uniform Flow Field

For flow in the positive x-direction, we have

$$\varphi_U = Ux \qquad \psi_U = Uy$$

and for flow in the positive y-direction, we have

$$\varphi_U = Uy \qquad \psi_U = -Ux$$

where U is the flow speed.

Thus, in general, one can form an additive combination of these different stream functions (or velocity potentials) to simulate different flows around different shapes. Then, if ψ_s is the new streamline function,

$$\psi_s = \psi_M + \psi_K + \psi_\Gamma + \psi_U$$

Example 11.8 Streamline pattern using `contour`

To illustrate the creation of ψ_s, we choose a uniform flow of velocity $U = 5$ m/s in the positive x-direction with a dipole K and a vortex Γ located at $(-1, -1)$, that is, $(x_K, y_K) = (-1, -1)$ and $(x_\Gamma, y_\Gamma) = (-1, -1)$, and give the strengths of each of these component flow fields the following values: $K = 5.0$ and $\Gamma = 8\pi$. The easiest method of determining the streamline pattern of a flow is to plot the streamlines with `contour`. The following script plots the streamline ψ_s for this flow. As illustrated in Figure 11.10, the resulting streamlines for the parameters selected show flow about a cylinder with circulation. The main difficulty in obtaining these results is in how to choose the contour levels to obtain a complete description of the flow. This is accomplished by using the value of ψ_s at the lower left corner of the domain for the minimum value and the value of ψ_s at the top middle of the domain for the maximum value. In Figure 11.10, the surface of a cylinder, which is also a streamline, has been superimposed on the streamlines.

The program that generates Figure 11.10 is as follows.

```
nx = 100;    xmin = -3.5;    xmax = 1.5;
ny = 100;    ymin = -3.0;    ymax = 1.5;
[x, y] = meshgrid(linspace(xmin, xmax, nx), linspace(ymin, ymax, ny));
U0 = 5.0;    Gamma = 8*pi;    xGamma = -1.0;    yGamma = -1.0;
K = 5.0;    xK = -1.0;    yK = -1.0;
radius = inline('sqrt((x-x1).^2+(y-y1).^2)', 'x', 'y', 'x1', 'y1');
PsiK = K*sin(atan2(y-yK, x-xK))./radius(x, y, xK, yK);
```

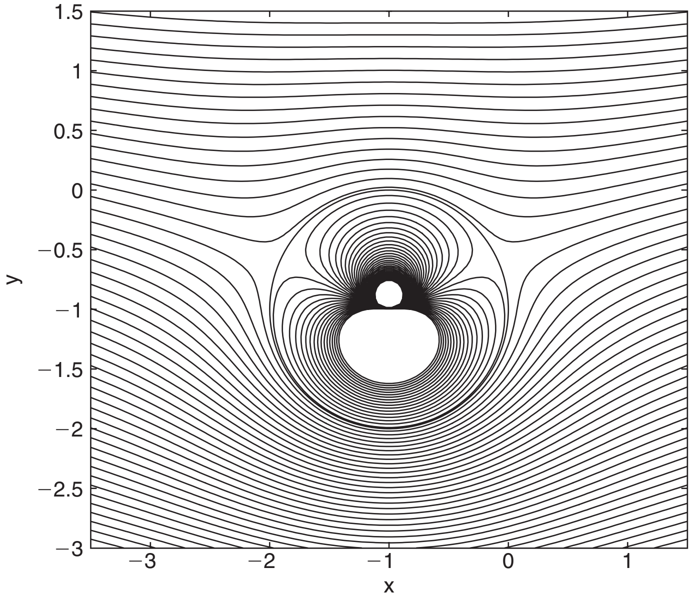


Figure 11.10 Streamlines for a cylinder with circulation in cross flow, which have been obtained from a contour plot of ψ_s.

```
PsiGamma = Gamma*log(radius(x, y, xGamma, yGamma))/2/pi;
StreamFunction = U0*y-PsiGamma-PsiK;
levmin = StreamFunction(1, nx);
levmax = StreamFunction(ny, nx/2);
levels = linspace(levmin, levmax, 50)';
contour(x, y, StreamFunction, levels)
hold on
theta = linspace(0, 2*pi);
plot(xGamma+cos(theta), yGamma+sin(theta), 'k-')
axis equal
axis([xmin, xmax, ymin, ymax])
ylabel('y')
xlabel('x')
```

Example 11.9 Direct calculation of streamlines

A second method to obtain flow patterns is to use fzero to find specific streamlines. In this case, we assume that the flow consists of a uniform stream of $U = 1$ in the positive y-direction, a source of strength $m = 4.0$ at $(0, -1)$, and a source of strength $m = -4.0$ at $(0, 1)$. Thus,

$$\psi_{\text{oval}} = \psi_U + \psi_{M_1} + \psi_{M_2}$$

These components produce a uniform flow over an oval-shaped body given by[4]

$$\frac{2xa}{x^2 + y^2 - a^2} = \tan \frac{xU}{m/2\pi} \tag{11.22}$$

where U is the flow speed, m is the source strength, and a is a characteristic dimension.

A difficulty in using fzero in this example is the need to find a good starting guess. In the following program, this is done by finding the value of the stream function ψ at a set of x locations along $y = -2.0a$. Given this initial data, a streamline is computed by marching along it, starting at $y = -2.0a$. At each successive y location, fzero uses function **StreamFun** to determine the x location of the stream function; the value of x at the previous y location is used as the initial guess. A plot from the output of the script is given in Figure 11.11, where we have assumed that $a = 1$. The graph has been rotated 90° so that the flow is horizontal, which is the traditional way of presenting it. The streamline that coincides with the boundary of the oval was not computed in this manner; it is plotted directly from Eq. (11.22). In the following program, *nPsi* is the number of streamlines and *n* is the number of points computed along each streamline.

```
U = 1.0;    a = 1.0;    m = 4.0;    co = m/(2*pi);
nPsi = 15;    n = 30;    yStart = -2.0*a;
```

[4] L. M. Milne-Thomson, *Theoretical Hydrodynamics*, Dover, Mineola, NY, 1996, p. 216.

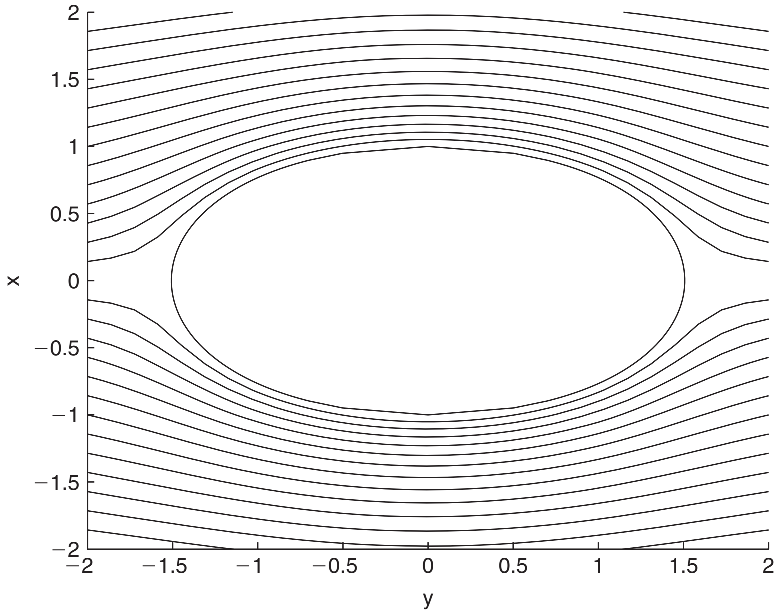


Figure 11.11 Streamlines for the oval given by Eq. (11.22).

```
xStart = linspace(0, 2*a, nPsi);
y = linspace(-2*a, 2*a, n);
x = zeros(1, n);
StreamFun = inline('-U*x-co*(atan2(x, y+a)-atan2(x, y-a))-psi', 'x', 'y', . . .
                    'psi', 'U', 'co', 'a');
Psi = StreamFun(xStart, yStart, 0, U, co, a);
for j = 1:nPsi
  guess = xStart(j);
  for i = 1:n
    x(i) = fzero(StreamFun, guess, [], y(i), Psi(j), U, co, a);
    guess = x(i);
  end
  if j>1
    plot(y, x, 'k-', y, -x, 'k-')
  end
  hold on
end
axis([-2*a, 2*a, -2*a, 2*a])
ylabel('x')
xlabel('y')
xx = linspace(-1, 1, 100);
yy = sqrt(1-xx.^2+2*xx./tan(xx/co));
plot(yy, xx, 'k-', -yy, xx, 'k-')
```

11.3.4 Joukowski Airfoils

The potential flow over a Joukowski airfoil in the complex z-plane ($z = x + iy$, where $i = \sqrt{-1}$) is obtained by conformal transformation of the flow over a cylinder with circulation in the ζ-plane ($\zeta = \xi + i\eta$). For a uniform incoming flow with speed Q and angle α relative to the ξ axis over a cylinder with radius R, center at $\zeta_{off} = \xi_{off} + i\eta_{off}$ and circulation Γ, the complex potential is given by[5]

$$F(\zeta) = \phi + i\psi = Qe^{-i\alpha}(\zeta - \zeta_{off}) + Qe^{i\alpha} R^2/(\zeta - \zeta_{off}) + \frac{i\Gamma}{2\pi} \ln\left[\left(\zeta - \zeta_{off}\right)/R\right]$$

where ϕ and ψ are the velocity potential and the stream function, respectively.

Referring to Figure 11.12, the Joukowski transformation is given by

$$z = \zeta + \lambda^2/\zeta \tag{11.23}$$

where

$$\lambda = \xi_{off} + \sqrt{R^2 - \eta_{off}^2}$$

is a real parameter determined by the position of the center of the cylinder relative to the origin of the (ξ, η) coordinate system.

The Joukowski transformation maps each point in the ζ-plane to a point in the z-plane and transfers the value of F according to $F(\zeta(z))$; that is, the value of F is the same at ζ and the corresponding point z. The transformation leaves both the circulation and the uniform flow far from the cylinder/airfoil unchanged. The circulation is adjusted so that in the ζ-plane the stagnation point on the downwind side of the cylinder is moved to the point $(\lambda, 0)$, labeled as TE in Figure 11.13, which becomes the trailing edge of the airfoil in the z plane. This value of Γ can be found either by trial and error from plots of the streamlines in the cylinder plane or by using the theoretical value

$$\Gamma = 4\pi QR \sin(\alpha - \theta_{TE})$$

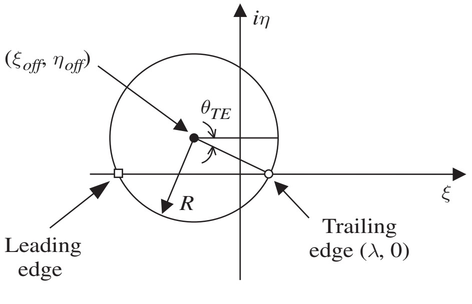


Figure 11.12 Geometry for flow over a cylinder in the ζ-plane.

[5] Panton, *Incompressible Flow*, p. 516.

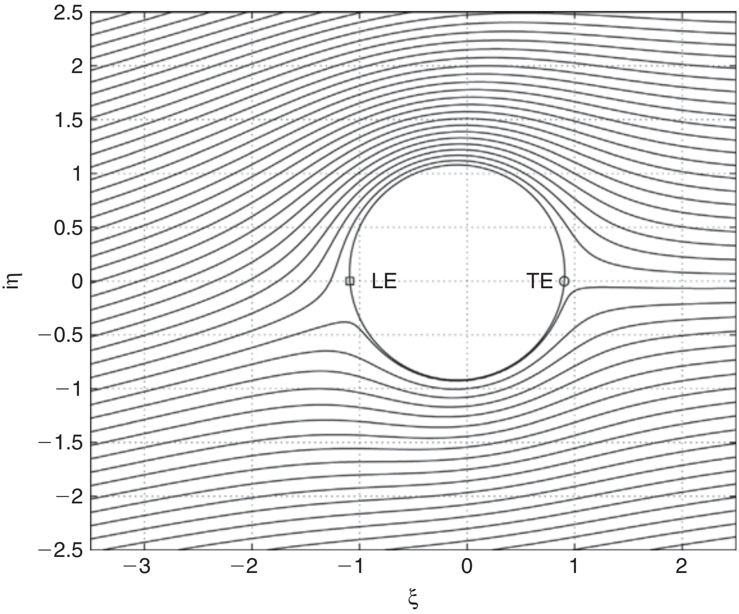


Figure 11.13 Streamline pattern around a circular cylinder with circulation Γ in the ζ-plane. Cylinder radius $R = 1\,\text{m}$, $\alpha = 8°$, and $(\xi_{off}, \eta_{off}) = (-0.093R, 0.08R)$.

as is done in the program given below. See Figure 11.12 for the definition of θ_{TE}. The lift per unit span on either the cylinder or the airfoil is $F_L = \rho Q \Gamma$, where ρ is the density of the fluid.

An important quantity in the airfoil flow field is the pressure, which is usually represented by the pressure coefficient

$$C_p = \frac{P - P_\infty}{\rho Q^2/2} = 1 - \frac{q^2}{Q^2} \tag{11.24}$$

where P is the local pressure, P_∞ is the pressure at infinity, ρ is the density of the fluid, and

$$q = \sqrt{u^2 + v^2}$$

is the local flow speed, where u and v are the x and y components of the fluid velocity, respectively. The local flow speed in the z-plane is computed from the complex velocity

$$w = u - iv$$

Hence,

$$ww^* = u^2 + v^2 = q^2$$

where the * denotes the complex conjugate. The complex velocity in the airfoil flow field is given by the derivative of F with respect to z; that is,

$$w = \frac{dF}{dz} = \frac{dF}{d\zeta}\frac{d\zeta}{dz} = \left[Qe^{-i\alpha} - Qe^{i\alpha}R^2/\left(\zeta - \zeta_{off}\right)^2 + \frac{i\Gamma}{2\pi\left(\zeta - \zeta_{off}\right)}\right]\frac{d\zeta}{dz}$$

where, from the above Joukowski transformation given by Eq. (11.23),

$$\frac{d\zeta}{dz} = \frac{1}{1 - \lambda^2/\zeta^2}$$

Example 11.10 Flow over a Joukowski airfoil

We shall calculate the streamlines and pressure field around a Joukowski airfoil. We assume that the cylinder has a radius $R = 1.0$ m and the flow comes from the lower left with an angle of 8° relative to the horizontal. The offset of the cylinder is given by $(\xi_{off}, \eta_{off}) = (-0.093R, 0.08R)$. The following program first evaluates the complex potential F in the ζ-plane and plots the streamlines (lines of constant ψ) as shown in Figure 11.13. The points on the cylinder corresponding to the leading and trailing edges of the airfoil are marked in the figure. The coordinates are then transformed to the z-plane and the streamlines are plotted in the z-plane, as shown in Figure 11.14. Finally, the pressure contours are calculated and plotted in Figure 11.15. In all cases, the

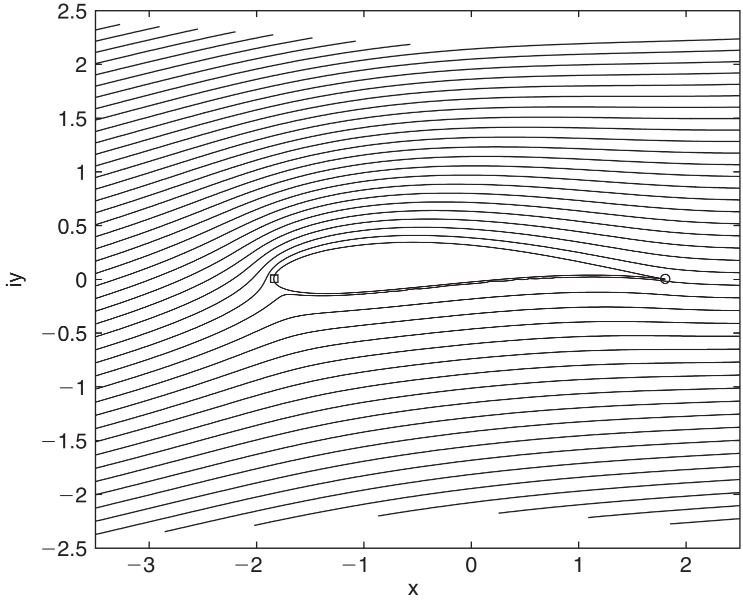


Figure 11.14 The Joukowski airfoil and streamlines obtained by transforming the cylinder and flow pattern shown in Figure 11.13. The square and circle correspond to the square and circle in Figure 11.13. This case is for $R = 1$ m, $\alpha = 8°$, and $(\xi_{off}, \eta_{off}) = (-0.093R, 0.08R)$.

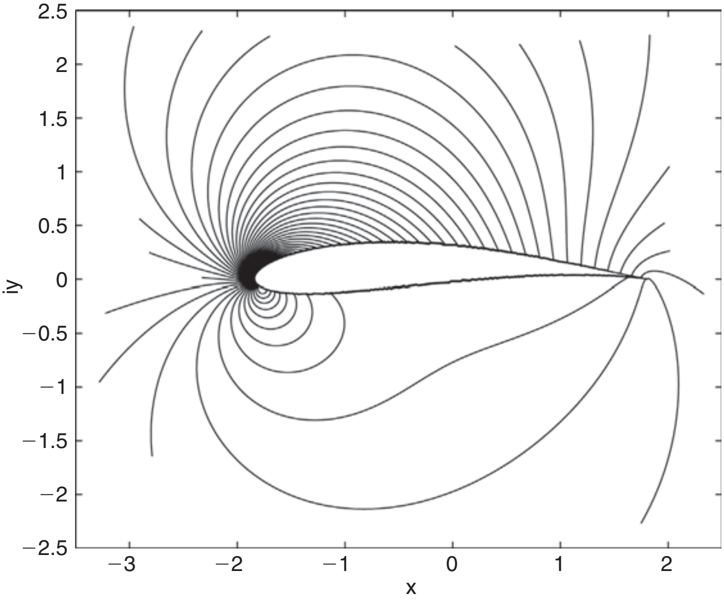


Figure 11.15 Contours of constant pressure coefficient in the vicinity of the Joukowski airfoil shown in Figure 11.14. This case is for $R = 1$ m, $\alpha = 8°$, and $(\xi_{off}, \eta_{off}) = (-0.093R, 0.08R)$.

contours are obtained using contour. The solution is complicated by the fact that while Eq. (11.23) transforms the flow outside the cylinder to the flow outside the airfoil, it transforms the flow inside the cylinder to the flow over the entire z-plane. Thus, in order to obtain only the desired streamlines and pressure contours outside the airfoil, the following procedure is used. For the streamlines, the values of the stream function inside the cylinder were set to the constant value of the stream function on its surface—the surface of the cylinder is a streamline so the stream function is constant there. For the pressure, this procedure must be modified since the value of the pressure varies along the surface of the cylinder. In this case, the value of the pressure inside was set arbitrarily to zero.

Figures 11.13, 11.14, and 11.15 are obtained with the following program.

```
R = 1.0;    Q = 1.0;    alpha = 8.0*pi/180;
ksioff = -0.093*R;    etaoff = 0.08*R;    zetaoff = complex(ksioff, etaoff);
nksi = 800;    ksimin = -3.5*R;    ksimax = 2.5*R;
neta = 800;    etamin = -2.5*R;    etamax = 2.5*R;
[ksi, eta] = meshgrid(linspace(ksimin, ksimax, nksi), . . .
                      linspace(etamin, etamax, neta));
zeta = complex(ksi, eta);
figure(1)    % Flow over cylinder
thetaTE = -asin(etaoff/R);
Gamma = 4*pi*Q*R*sin(alpha-thetaTE);
F = Q*exp(-1i*alpha)*(zeta-zetaoff)+ Q*exp(1i*alpha)*R^2./(zeta-zetaoff)+ . . .
          1i*Gamma/(2*pi).*log((zeta-zetaoff)/R);
```

```
StreamFunction = imag(F);
zetapt = complex(R+ksioff, etaoff);
Fpt = Q*exp(-1i*alpha)*(zetapt-zetaoff)+Q*exp(1i*alpha)*R^2./
          (zetapt-zetaoff)+1i*Gamma/(2*pi).*log((zetapt-zetaoff)/R);
StreamFunctionpt = imag(Fpt);
rad = sqrt((ksi-ksioff).^2+(eta-etaoff).^2);
indx = find(rad<=R);
StreamFunction(indx) = StreamFunctionpt;
levmin = StreamFunction(1,nksi);
levmax = StreamFunction(neta,1);
levels = linspace(levmin, levmax, 50);
contour(ksi, eta, StreamFunction, levels)
axis equal
grid
axis([ksimin, ksimax, etamin, etamax])
xlabel('\xi')
ylabel('i\eta')
hold on
theta = linspace(0, 2*pi, 1000);
zetac = R*exp(1i*theta)+zetaoff;
plot(zetac, 'k-')
hold on
ksiTE = ksioff + sqrt(R^2-etaoff^2);
ksiLE = ksioff - sqrt(R^2-etaoff^2);
plot(ksiTE, 0 ,'or')
plot(ksiLE, 0, 'sr')
text(ksiTE-0.35*R, 0, 'TE')
text(ksiLE+0.2*R,0, 'LE')
figure(2)   %Joukowski airfoil
lambda = ksioff + sqrt(R^2-etaoff^2);
zeta = complex(ksi, eta);
z = zeta + lambda^2./zeta;
x = real(z);
y = imag(z);
contour(x, y, StreamFunction, levels)
axis equal
axis([ksimin, ksimax, etamin, etamax])
xlabel('x')
ylabel('iy')
hold on
zair = zetac+lambda^2./zetac;
xair = real(zair);
plot(zair, 'k')
zetaTE = complex(ksiTE, 0);
zetaLE = complex(ksiLE, 0);
zTE = zetaTE+lambda^2/zetaTE;
zLE = zetaLE+lambda^2/zetaLE;
plot(zTE, 0, 'or')
plot(zLE, 0, 'sr')
```

```
figure(3)    % Pressure field around airfoil
w = (Q*exp(-1i*alpha)-Q*exp(1i*alpha)*R^2./(zeta-zetaoff).^2+ ...
      1i*Gamma/(2*pi)./(zeta-zetaoff))./(1.0-lambda^2./zeta.^2);
Cp = 1.0-w.*conj(w)/Q^2;
Cp(indx) = 0.0;
levels = linspace(-10, 1, 150);
contour(x, y, Cp, levels)
hold on
zair = zetac+lambda^2./zetac;
xair = real(zair);
plot(zair, 'k-')
axis equal
axis([ksimin, ksimax, etamin, etamax])
xlabel('x')
ylabel('iy')
```

11.4 OPEN CHANNEL FLOW

Consider a gradually varying flow in a channel with constant prismatic cross section, as shown in Figure 11.16. The water depth above the bottom of the channel at any streamwise location z is given by y. The cross-sectional area of the water in the channel is

$$A = by(2 + y/(bm))$$

and the wet portion of the perimeter of the channel is

$$P = 2(b + y\sqrt{1 + 1/m^2})$$

Thus, the hydraulic radius is

$$R_h = A/P = \frac{by(2 + y/(bm))}{2(b + y\sqrt{1 + 1/m^2})}$$

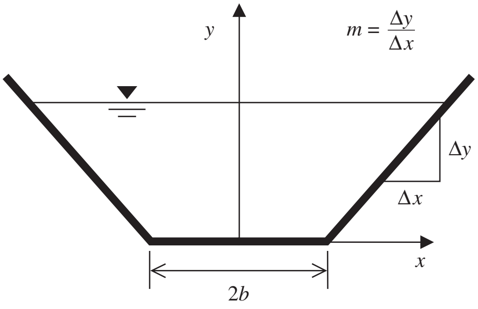


Figure 11.16 Cross section of a prismatic channel with water depth y.

Let the volume flow rate in the channel be Q. Two physically important water depths are the uniform flow depth y_0 and the critical flow depth y_c.[6] The uniform flow depth is the depth at which the water depth and flow conditions do not vary along the length of the channel. This depth is obtained by solving the Manning equation

$$Q = \frac{AR_h^{2/3}S_0^{0.5}}{n} \tag{11.25}$$

where S_0 is the stream-wise slope of the channel and n is a constant.[7] The critical flow depth y_c is the depth at which the Froude number is equal to unity. The Froude number equals $v/\sqrt{gA/B}$, where v is the average flow velocity, $g = 9.81$ m/s^2 is the gravity constant, and $B = 2b + 2y_c/m$ is the surface width. In terms of the flow rate, this condition is written as

$$Q^2 = g\frac{A^3}{B} = g\frac{by_c(2 + y_c/(bm))^3}{2b + 2y_c/m} \tag{11.26}$$

For constant flow rate, but nonuniform flow conditions along a channel of uniform slope S_0, the water depth y is given by the first-order differential equation[8]

$$\frac{dy}{dx} = S_0\frac{1-(y_0/y)^{10/3}}{1-(y_c/y)^3} \tag{11.27}$$

We shall now illustrate these results with two examples.

Example 11.11 Uniform channel with an overfall

Consider a uniform channel with $Q = 30$ m^3/s, $n = 0.025$, $m = 0.6667$, $b = 3.0$ m, and with four different slopes: $S_0 = 0.0010, 0.0015, 0.0020$, and 0.0025. In each case, the outlet of the channel is a free overfall. The presence of the overfall at the downstream end of the channel requires that the flow reach the critical condition $y = y_c$ at that point. The water surface height in the channel will increase with distance upstream of the overfall, eventually reaching the uniform flow depth y_0. For each channel slope, we shall find the water surface height profile along the channel upstream of the overfall to the point where $y = 0.975y_0$. This problem is solved by integrating Eq. (11.27); however, since the right-hand side tends to infinity at the overfall, where $y \rightarrow y_c$, it is most convenient to invert the equation and integrate dx/dy over the interval from $0.975y_0$ to y_c. This is accomplished with the following program, which produces the results shown in Figure 11.17.

```
function Example11_11
g = 9.81;   m = 0.6667;   b = 3.0;
n = 0.025;   Q = 30.0;
hold on
```

[6] J. B. Franzini and E. J. Finnemore, *Fluid Mechanics with Engineering Applications*, McGraw-Hill Companies, Inc., 1997, pp. 427–449.

[7] Equation (11.25) is valid for metric units; for English units, the right-hand side of the equation is multiplied by 1.486.

[8] F. M. Henderson, *Open Channel Flow*, MacMillan Publishing Company, Inc., 1966, p. 131.

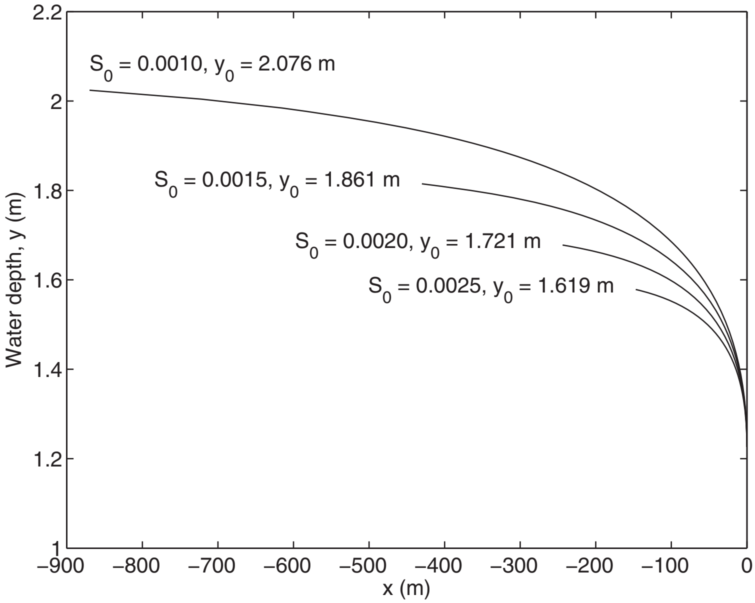


Figure 11.17 Water height y versus distance along a prismatic channel for $m = 0.6667$, $b = 3.0$ m, $n = 0.025$, $Q = 30.0$ m^3/s and for several values of the slope S_0. There is a free overfall at $x = 0$, where the flow reaches the critical condition.

```
 for slope = [0.0010:0.0005:0.0025]
   y0 = fzero(@Manning, 6, [], Q, n, b, m, slope);
   yc = fzero(@Q26, 3, [], Q, g, b, m);
   [y, x] = ode45(@dchannel, [yc, 0.975*y0], 0, [], yc, y0, slope);
   plot(x, y, 'k-')
   if slope == 0.001
     text(x(end), y(end)+0.05, ['S_0 = ' num2str(slope, '%6.4f') ', y_0 = ' ...
          num2str(y0, 4) ' m'], 'HorizontalAlignment', 'Left')
   else
     text(x(end)-30, y(end), ['S_0 = ' num2str(slope, '%6.4f') ', y_0 = ' ...
          num2str(y0, 4) ' m'], 'HorizontalAlignment', 'Right')
   end
end
axis([-900, 0, 1, 2.2])
xlabel('x (m)')
ylabel('Water depth, y (m)')

function dydx = dchannel(y, x, yc, y0, slope)
dydx(1) = 1/slope*(1-(yc/y(1))^3)/(1-(y0/y(1))^(10/3));

function A = Manning(y, Q, n, b, m, slope)
A = Q-1.0/n*b*y*(2+y/(b*m))*(b*y*(2+y/(b*m))/(2*(b+y*sqrt(1+1/m^2))))
  ^0.667*slope^0.5;

function B = Q26(y, Q, g, b, m)
B = Q^2-g*(b*y*(2+y/(b*m)))^3/(2*b+2*y/m);
```

Example 11.12 Reservoir discharge

A reservoir is connected to a long uniform prismatic channel with constant slope via a weir, as shown in Figure 11.18. The change in elevation between the top of the weir and the water-free surface in the center of the reservoir is y_R. We shall calculate the flow rate in the channel by making use of the specific energy

$$E = y + v^2/(2g)$$

which, by Bernoulli's equation, is constant along the free-surface streamline from the center of the reservoir, where the flow speed is zero, to the point on the free surface directly over the lip of the weir, where the flow speed is v_w and the water depth above the weir is y_w. In the center of the reservoir, we have $E = E_0 = y_R$. Thus, the specific energy equation reduces to

$$y_R = y_w + v_w^2/(2g)$$

In terms of the flow rate Q,

$$y_R = y_w + Q^2/\left(2gA^2\right) = y_w + Q^2/\left(2gb^2y_w^2\left(2 + y_w/(bm)^2\right)\right) \qquad (11.28)$$

The critical depth over the weir y_{wc} occurs when the Froude number $v/\sqrt{gA/B}$ equals one at that point. Thus, at the critical condition, the specific energy equation becomes

$$y_{wc} + \frac{A_{wc}}{2B_{wc}} = y_R$$

where

$$A_{wc} = by_{wc}\left(2 + y_{wc}/(bm)\right)$$

$$B_{wc} = 2b + 2y_{wc}/m$$

The value of Q that satisfies both the Manning equation, Eq. (11.25), and the specific energy equation, Eq. (11.28), is the flow rate for all $y_w > y_{wc}$. As the slope of the channel increases, the flow rate Q increases until the critical condition is reached at the weir. For higher channel slopes, the flow rate remains constant at the critical value.

We shall now calculate Q and y_w for $y_R = 3.0$ m, $n = 0.014$, $m = 30.0$, $b = 1.5$ m, and $S_0 = 0.001$. The execution of the program below results in Figure 11.19, where both

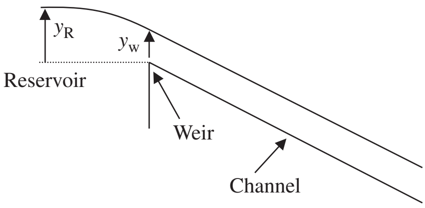


Figure 11.18 Schematic of a reservoir and a discharge channel.

the Manning equation and the specific energy equation are plotted. In this case, the flow condition at the weir is subcritical and the intersection of the two curves gives the flow rate and the water depth.

```
function Example11_12
yr = 3.0;    g = 9.81;    b = 1.5;    m = 30;    S0 = 0.001;    n = 0.014;
yw = linspace(0, yr, 100);
Qbernoulli = sqrt(2*g*b^2*(yr-yw).*(2*yw+yw.^2/(b*m)).^2);
plot(Qbernoulli, yw, 'k-')
hold on
axis([0, 30, 0, 3.0])
xlabel('Volume flow rate, Q (m^3/s)')
ylabel('y_w (m)')
A = b*yw.*(2+yw/(b*m));
P = 2*(b+yw*sqrt(1+1/m^2));
Rh = A./P;
Qmanning = A.*(Rh.^0.6666*sqrt(S0))/n;
plot(Qmanning, yw, 'k--');
legend('Bernoulli Eqn.', 'Manning Eqn.', 'Location', 'SouthEast')
ywflow = fzero(@Q2526, [8.5*0.3048, 9.5*0.3048], [], b, m, S0, n, g, yr);
Awflow = b*ywflow*(2+ywflow/(b*m));
Qflow = Awflow*sqrt(2*g*(yr-ywflow));
text(5, 2.7, 'Subcritical Flow')
text(5, 2.5, ['Q = ' num2str(Qflow, 4) ' m^3/s'])
```

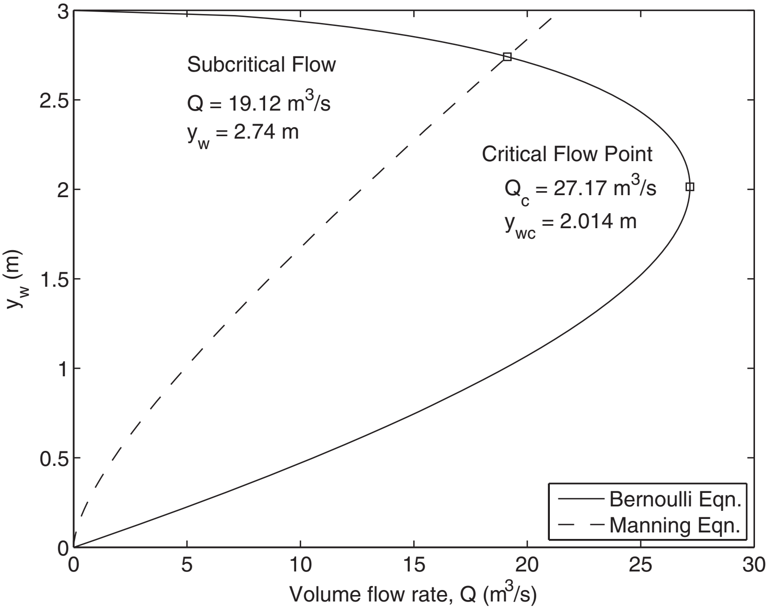


Figure 11.19 Reservoir discharge calculation for $y_r = 3.0$ m, $b = 1.5$ m, $m = 30.0$, $S_0 = 0.001$, and $n = 0.014$. The solution is the intersection of the dashed and solid curves, which are given by the values labeled subcritical flow.

```
text(5, 2.3, ['y_{w} = ' num2str(ywflow, 4) ' m'] )
plot(Qflow, ywflow, 'ks')
ywcrit = fzero(@Q12, 6, [], b, m, yr);
Awcrit = b*ywcrit*(2+ywcrit/(b*m));
Qcrit = Awcrit*sqrt(2*g*(yr-ywcrit));
text(18, 2.2, 'Critical Flow Point')
text(19, 2.0, ['Q_{c} = ' num2str(Qcrit, 4) ' m^3/s'])
text(19, 1.8, ['y_{wc} = ' num2str(ywcrit, 4) ' m'] )
plot(Qcrit, ywcrit, 'ks')

function A = Q2526(y, b, m, S0, n, g, yr)
A = (b*y*(2+y/(b*m)))*(((b*y*(2+y/(b*m))) ...
  /(2*(b+y*sqrt(1+1/m^2))))^0.6666*sqrt(S0))/n- ...
  sqrt(2*g*b^2*(yr-y)*(2*y+y^2/(b*m))^2);

function B = Q12(ywc, b, m, yr)
B = ywc+0.5*b*ywc*(2+ywc/(b*m))/(2*b+2*ywc/m)-yr;
```

11.5 BIOLOGICAL FLOWS

A simple approximation in modeling blood flow is to assume that it can be represented by a fluid undergoing laminar, fully developed flow in a straight pipe with an axial pressure gradient that varies sinusoidally in time. This model can be described by Eq. (11.7) by replacing the term $1/\rho dP/dz$ with $A\sin(\omega t)$ where A is the magnitude of the sinusoidal oscillation and ω is its radian frequency. Thus,

$$\frac{\partial u_z}{\partial t} = -A\sin(\omega t) + \nu\left(\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u_z}{\partial r}\right)\right) \tag{11.29}$$

In this model, there are two timescales, the viscous timescale $T_{\text{vis}} = R^2/\nu$ and the timescale associated with the period of the pressure gradient term $T_p = 2\pi/\omega$.

Example 11.13 Laminar pulsatile flow in a pipe

We shall consider blood flow in a large artery with $R = 0.004$ m, $\nu = 5.0 \times 10^{-6}$ m^2/s, $\rho = 1{,}000$ kg/m^3, and $\omega = 2\pi$ rad/s. Thus, the timescales are $T_{\text{vis}} = 3.2$ s and $T_p = 1$ s. The velocity distribution u_z is determined from the following program.

```
function Example11_13
omega = 2*pi;    nu = 5e-6;    A = -1e6;
nr = 30;    rmax = 0.004;
r = linspace(0, rmax, nr);
nt = 16;    tmax=2*2*pi/omega;
t = linspace(0, tmax, nt);
u = pdepe(1, @Pulse, @PulseIC, @PulseBC, r, t, [], omega, A, nu);
hold on
```

```
for ijd = 1:nt
  plot(u(ijd,:)/(A/omega)+(ijd-1), r/rmax, 'k')
  plot(u(ijd,:)/(A/omega)+(ijd-1), -r/rmax, 'k')
  plot([(ijd-1), (ijd-1)], [-1,1], ':k')
end
xlabel('u_z/(A/\omega)')
ylabel('r/R')
ylabel('t')
function [c, f, s] = Pulse(r, t, u, DuDr, omega, A, nu)
c = 1.0;   f = nu*DuDr;
s = A*sin(omega*t);
function u0 = PulseIC(r, omega, A, nu)
u0 = 0;
function [pl, ql, pr, qr] = PulseBC(rl, ul, rr, ur, t, omega, A, nu)
pl = 0;   ql = 1;
pr = 0;   qr = ur;
```

Executing this program gives the results shown in Figure 11.20. As can be seen in the figure, consistent with $T_{\text{vis}} > T_p$, there is a large central core of relatively uniform flow and a thick viscous boundary layer on the pipe wall.

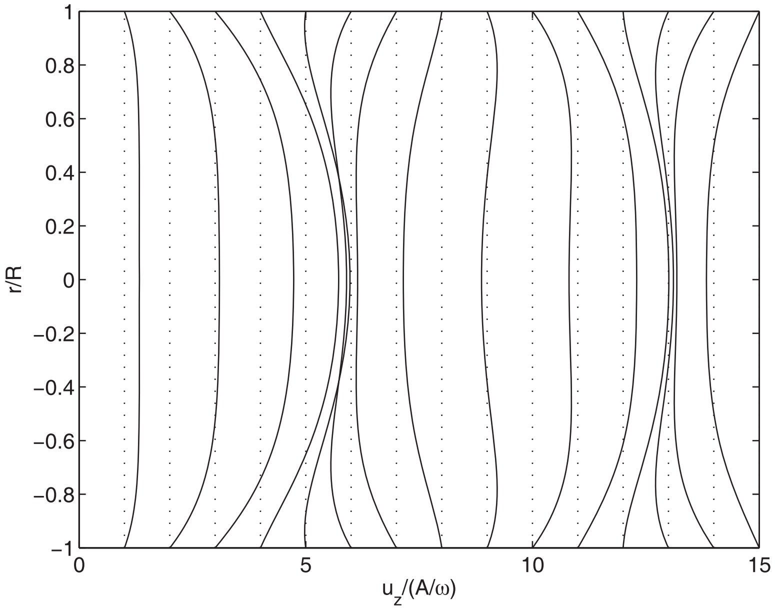


Figure 11.20 Sixteen velocity profiles corresponding to equally spaced times over the first two periods of the sinusoidally varying pressure gradient in a tube. The fluid is initially at rest. The vertical dotted lines are the zero lines for each of the profiles. The results are shown over the diameter ($-1 \leq r/R \leq 1$) of the vessel.

EXERCISES

Section 11.2.2

11.1 Water is to flow from reservoir A to reservoir B through the piping system shown in Figure 11.21. The flow rate when the valve is completely open is to be 0.003 m^3/s. The generalized head loss equation becomes

$$z_A - z_B = \frac{\lambda L V^2}{2gD} + \sum_{m=1}^{5} K_{Lm} \frac{V^2}{2g}$$

where $z_A - z_B$ is the change in height between the free surfaces of the water in the two reservoirs, K_{Lm} are the minor loss coefficients at the locations shown in Figure 11.21, λ is the pipe friction coefficient, and $V = 4Q/\pi D^2$ is the average velocity in the pipe. If $\nu_{\text{vis}} = 1.3 \times 10^{-6}\ \text{m}^2/\text{s}$ and $\rho = 1{,}000\ \text{kg/m}^3$, then determine the pipe diameter (Answer: $D = 0.04698$ m).

11.2 The oscillations caused by a suddenly released fluid from a height Z that separates the fluid levels in two rectangular prismatic reservoirs connected by a long pipeline of length L shown in Figure 11.22 can be determined from[9]

$$\frac{d^2Z}{dt^2} + \text{signum}\left(\frac{dZ}{dt}\right) p \left(\frac{dZ}{dt}\right)^2 + qZ = 0$$

where

$$p = \frac{fA_1A_2L_e}{2DaL(A_1 + A_2)} \qquad q = \frac{ga(A_1 + A_2)}{A_1A_2L}$$

Moreover, it has been assumed that the motion of the liquid is mostly turbulent so that the head loss is proportional to the square of the velocity. The quantity L_e is the equivalent length of the pipe incorporating minor losses, g is the gravitational constant, f is the friction coefficient in the pipe, A_1 and A_2 are the surface areas of the two reservoirs, a is the area of the pipeline, and D is its diameter.

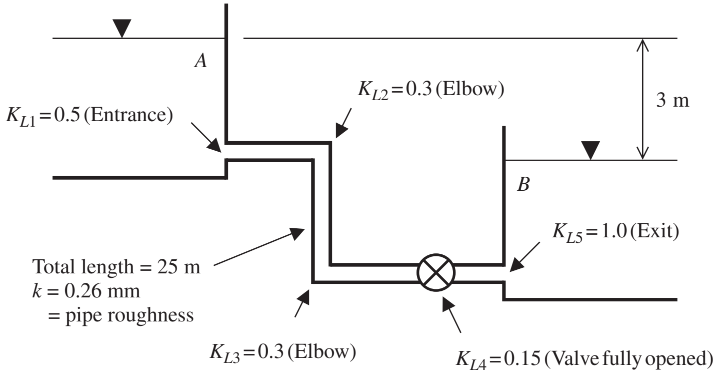


Figure 11.21 Piping system between two reservoirs.

[9] D. N. Roy, *Applied Fluid Mechanics*, Ellis Horwood Limited, Chichester, England, 1988, pp. 290–293.

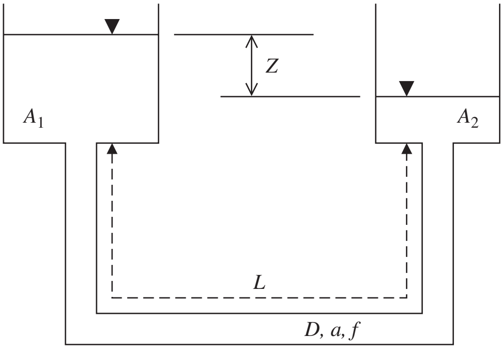


Figure 11.22 Interconnected reservoirs.

If $p = 0.375\ \mathrm{m}^{-1}$ and $q = 7.4 \times 10^{-4}\ \mathrm{s}^{-2}$ and the initial conditions are $Z(0) = Z_n$ m and $dZ(0)/dt = 0$ m/s, then determine the value of the first occurrence of t_n for which $Z(t_n) = 0$ when $Z_n = 5, 10, \ldots, 50$. Plot the results, which should look like those shown in Figure 11.23. Use `interp1` to determine t_n.

11.3 Consider a belt whose surface is moving vertically upward with velocity $V_b = 1.5$ m/s. There is a thin layer of oil on the surface whose thickness is held constant. Gravity acts to pull the oil downward, while viscous forces drag the oil upward. This flow is governed by the equation

$$\frac{\partial w}{\partial t} = \nu \frac{\partial^2 w}{\partial x^2} - g$$

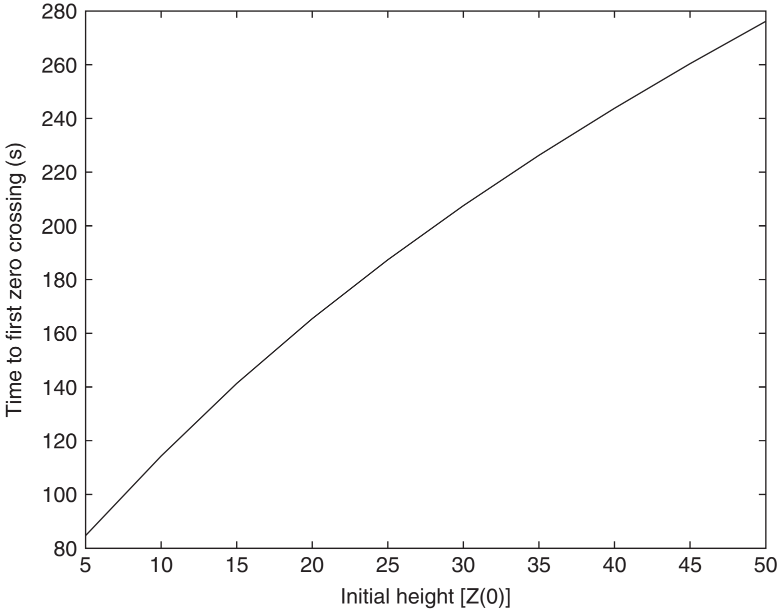


Figure 11.23 Values of the first occurrence of t_n for which $Z(t_n) = 0$ as a function of $Z(0) = Z_n$.

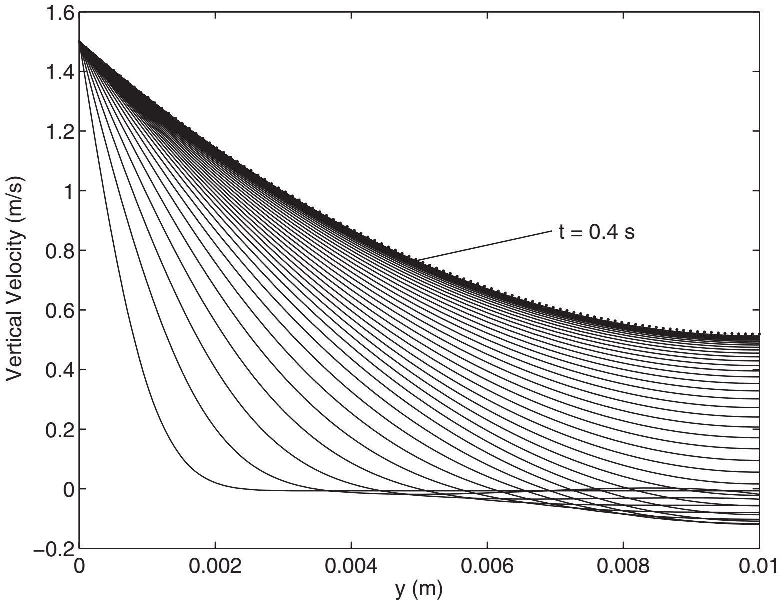


Figure 11.24 Velocity distribution in an oil layer on a belt surface at $x = 0$ moving vertically at speed 1.5 m/s. The velocity distributions are shown at equally spaced intervals in time up to $t = 0.4$ s. The dotted line is the theoretical steady-state solution to the problem. It is seen that this curve is virtually coincident with the curve at $t = 0.4$ s.

where $w(x, t)$ is the velocity distribution through the thickness of the oil layer, ν is the kinematic viscosity, and $g = 9.81$ m/s^2 is the acceleration of gravity. At the surface of the belt, the no-slip condition is enforced, $w(0, t) = U_b$, while at the free surface of the oil, the stress is zero,

$$\rho\nu\left[\frac{\partial w}{\partial x}\right]_{x=h} = 0$$

Let $h = 0.01$ m, $\rho = 900$ kg/ m^3, and $\nu = 0.0005$ m^2/s. Calculate the velocity profiles for times up to the point where the velocity profile reaches a steady state, which occurs at approximately $t = h^2/\nu = 0.2$ s. The results are shown in Figure 11.24 along with the theoretical steady-state solution

$$w = \frac{\rho g}{2\mu}x^2 - \frac{\rho g h}{\mu}x + V_b$$

11.4 Consider a horizontal layer of fluid with $\rho = 800$ kg/m^3, $\mu = 0.02$ Ns/m^2, and $h = 0.02$ m that is initially at rest. There is a free surface $y = h$ and at $y = 0$ the fluid is subjected to a constant stress ($\tau = 0.01$ N/m^2) that is turned on at $t = 0$. Compute the velocity profiles in the layer for times from $t = 0$ to $t = 1.0$ s. The results are shown in Figure 11.25.

Section 11.3.1

11.5 Consider a viscous fluid with $\mu = 0.02$ kg/m·s and $\rho = 800$ kg/m^3 in a channel with height 0.3 cm. The fluid is initially at rest. At $t = 0$, a pressure gradient of magnitude

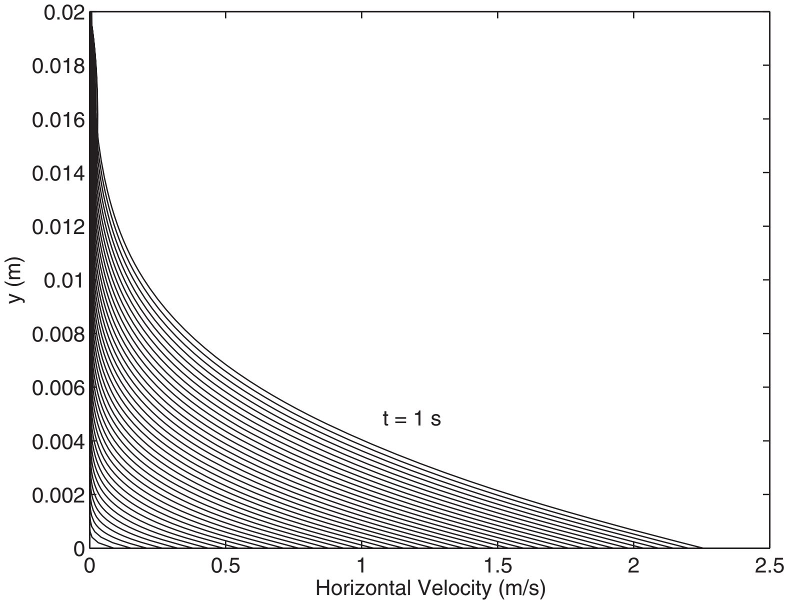


Figure 11.25 Velocity profiles in a horizontal layer with a free surface at $y = 0.02$ and a constant stress condition $\tau = 0.01$ N/m^2 at $y = 0$. The flow is started from rest.

10 kPa/m in the direction along the axis of the channel is turned on, and at the same time the upper plate begins moving at 1.5 m/s in the direction from low to high pressure. For times from 0 to 0.04 s in steps of 0.005 s, plot the velocity profile. The results should look like those shown in Figure 11.26.

Section 11.3.3

11.6 The flow about a thin symmetric airfoil can be approximated by potential flow theory.[10] The chord of the airfoil extends along the x axis from $x = 0$ to $x = c$ and is represented by a vortex sheet whose strength $\gamma(x)$ is given by

$$\gamma(\theta) = 2\alpha V_\infty \frac{1 + \cos\theta}{\sin\theta}$$

where

$$x = \frac{c}{2}(1 - \cos\theta) \qquad 0 \leq \theta \leq \pi$$

α is the angle of attack (in radians) of the incoming flow relative to the x axis, and V_∞ is the flow speed. Consider the vortex sheet to be approximated by a set of N discrete vortices separated by a distance $\Delta x = c/N$ with strength $\Gamma_i = \gamma(\theta_i)\Delta x$. Using the procedure employed in Example 11.8, draw the streamlines of this flow for $\alpha = 10°$, $c = 2$ m, and $V_\infty = 100$ m/s. The results should look like those shown in Figure 11.27.

[10] See, for example, J. D. Anderson, *Fundamentals of Aerodynamics*, McGraw-Hill, New York, 1991, Chapter 4.

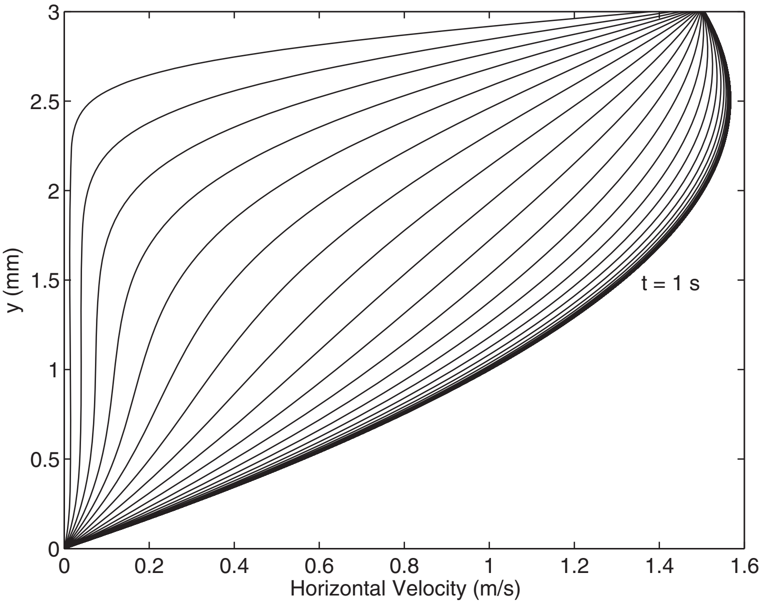


Figure 11.26 Velocity profiles in a channel of height $h = 0.003$ m at equally spaced times from 0 to 1 s. The flow is started from rest.

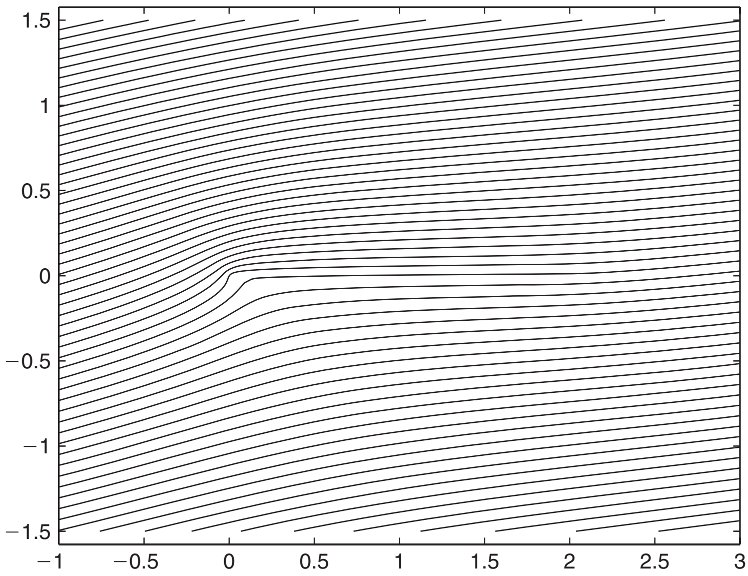


Figure 11.27 Streamlines for a thin airfoil with a 2-m chord and an angle of attack of 10°. The flow is from left to right and the foil extends horizontally from $(0, 0)$ to $(2, 0)$.

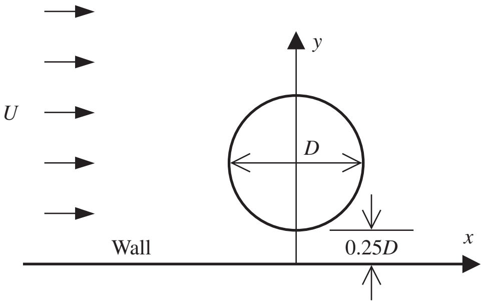


Figure 11.28 Cylinder near wall with uniform horizontal flow upstream.

11.7 Consider a potential flow over a cylinder that is placed near a wall as shown in Figure 11.28. Represent the flow over the cylinder and wall with a uniform flow of speed $U = 1.0$ m/s and two dipoles of strength $m = 2\pi UD^2/4$ located at $(x, y) = (0, 0.75D)$ and $(0, -0.75D)$. Plot the streamline pattern. Note that the closed streamline around each dipole is not circular.

11.8 The flow about a thin cambered airfoil shown in Figure 11.29 can be approximated by potential flow theory.[11] The chord of the airfoil extends along the x-axis from $x = 0$ to $x = c$ and is represented by a vortex sheet placed along the chord. The strength $\gamma(x)$ of this vortex sheet is given by

$$\gamma(\theta) = 2V_\infty\left(A_0 \frac{1 + \cos\theta}{\sin\theta} + \sum_{n=1}^{\infty} A_n \sin(n\theta)\right)$$

where

$$\frac{x}{c} = \frac{1}{2}(1 - \cos\theta) \qquad 0 \le \theta \le \pi$$

α is the angle of attack (in radians) of the incoming flow relative to the x axis and V_∞ is the flow speed. The constants A_n are given by

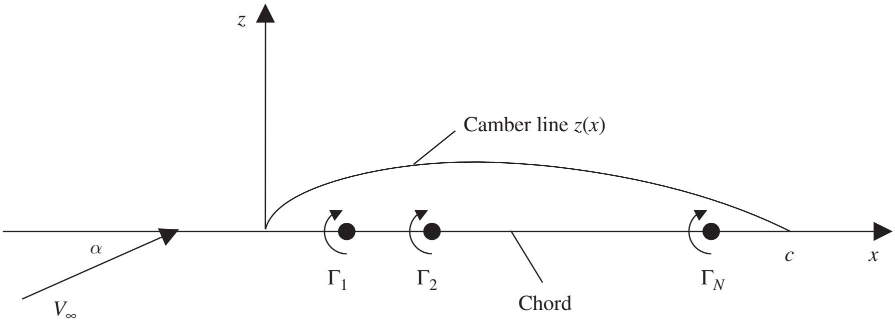


Figure 11.29 Placement of a vortex sheet on the chord line.

[11] *Ibid.*

$$A_0 = \alpha - \frac{1}{\pi}\int_0^\pi \frac{dz}{dx}\,d\theta_o$$

$$A_n = \frac{2}{\pi}\int_0^\pi \frac{dz}{dx}\cos(n\theta_o)d\theta_o$$

where $z(x)$ is the vertical distance between the chord line and the camber line.

Let the camber line be

$$\frac{z}{c} = 2.6595\frac{x}{c}\left[\left(\frac{x}{c}\right)^2 - 0.6075\frac{x}{c} + 0.1147\right] \qquad 0 \le x/c \le 0.2025$$

$$= 0.02208\left(1 - \frac{x}{c}\right) \qquad 0.2025 \le x/c \le 1.0$$

where the distance z is normal to the chord. Compute A_n for $n = 0, 1, \ldots, 20$. Let the vortex sheet be approximated by a set of N_v discrete vortices (see Section 11.3.3) along the x-axis in the region $0 \le x/c \le 1$. The vortices are separated by a distance $\Delta x = c/N$ and have the strengths $\Gamma_i = \gamma(\theta_i)\Delta x$. Use `contour` to draw the streamlines of this flow for $\alpha = 4°$, $c = 2$ m, and $V_\infty = 100$ m/s. In the streamline plot, the camber line should closely follow a streamline (Partial answers: $A_0 = 0.0412$, $A_1 = 0.0955$, $A_2 = 0.0792$, $A_3 = 0.0568$).

Section 11.3.4

11.9 Let the chord L of the Joukowski airfoil be the distance following a straight line along the x axis from leading edge to trailing edge in the z plane. Let the thickness t of the foil be the maximum vertical distance along the y-axis between the lower and upper surfaces of the foil, as shown in Figure 11.30. Let the camber b of the foil be the maximum distance between the chord line (the x axis) and the midline of the foil. (A point on the midline is located with equal vertical distance to the upper and lower surfaces of the foil.) It is found that t/L is primarily a function of ξ_{off} and b/L is primarily a function of η_{off}. Plot t/L versus ξ_{off} over the range $-0.2R$ to $0R$ for $\eta_{off}/R = [0.01, 0.02, 0.03, 0.04]$, and plot b/L versus η_{off} over the range $0.0R$ to $0.04R$ for $\xi_{off}/R = [-0.2, -0.15, -0.10, -0.05, 0]$. The plots should look like those in Figure 11.31.

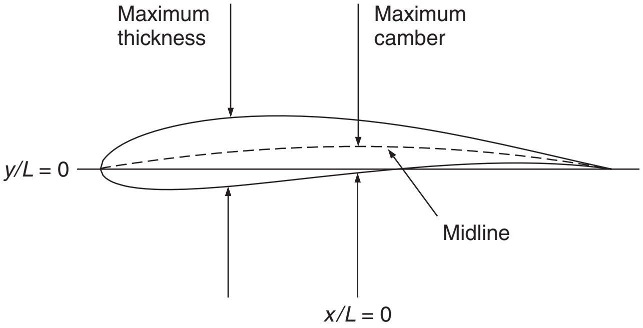


Figure 11.30 Definitions of maximum foil thickness and maximum foil camber.

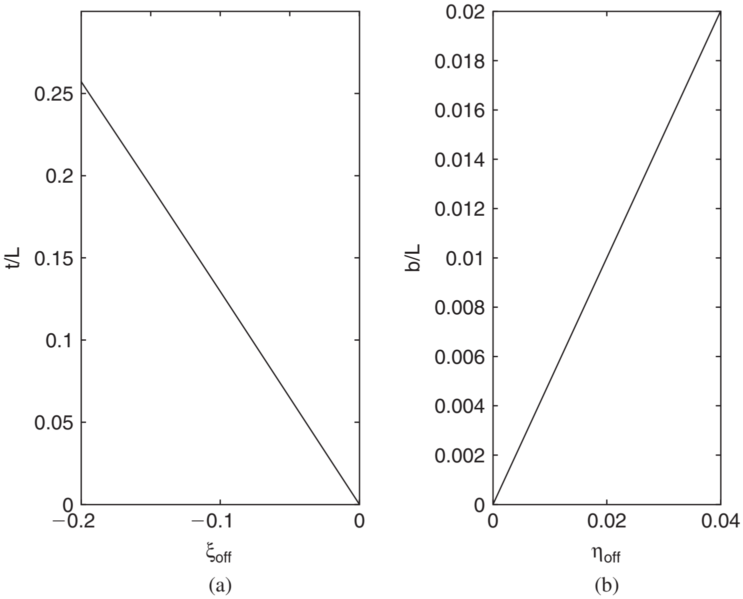


Figure 11.31 (a) Foil thickness versus ξ offset. (b) Foil camber versus η offset. In figure (a), five curves are drawn for $\eta_{off}/R = 0, 0.01, 0.02, 0.03,$ and 0.04. In figure (b), five curves are drawn for $\xi_{off}/R = -0.2, -0.15, -0.10, -0.05,$ and 0.0. Note that in each plot the five curves are virtually on top of one another.

11.10 Plot the pressure coefficient distribution on the upper and lower surfaces of a Joukowski airfoil as given by Eq. (11.24) for $R = 1.0$, $\alpha = 6°$, $\xi_{off} = -0.093R$, $\eta_{off} = 0.08R$, $t/L = 0.1215$, and $b/L = 0.0401$. The results should look like those given in Figure 11.32.

11.11 The lift coefficient of the foil is given by

$$C_L = \frac{F_L}{\rho Q^2 L/2}$$

where $F_L = \rho Q\Gamma$ is the lift force per unit span. Plot C_L versus angle of attack α, which ranges from $-10°$ to $10°$ for $\xi_{off} = -0.093R$ and $\eta_{off} = [0, 0.02R, 0.04R, 0.06R, 0.08R]$. The results should look like the one shown in Figure 11.33. For convenience, let $R = 1$.

Section 11.4

11.12 Find the reservoir discharge flow rate as a function of channel slope when the vertical distance between the top of the weir and the free surface in the reservoir is $y_r = 3.0$ m and the long prismatic channel is described by $n = 0.014$, $b = 1.5$ m, and $m = 1.0$. The result should look like that shown in Figure 11.34.

11.13 Flow discharges from a reservoir into a prismatic channel with conditions $y_r = 3.0$ m, $n = 0.014$, $b = 1.5$ m, $m = 1.0$, and $S_0 = 0.003$. Note from Figure 11.34 that S_0 is above the critical slope, indicating that the flow rate is $45.3336\ \text{m}^3/\text{s}$, which is the maximum value. Compute the surface height profile in the channel assuming that the channel is long enough for the flow to gradually reach the supercritical normal flow depth. The results should look like those given in Figure 11.35.

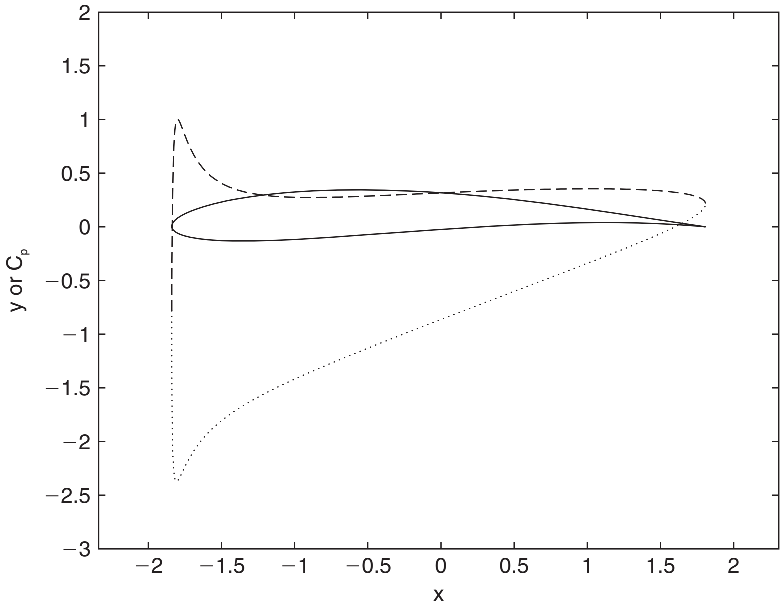


Figure 11.32 Foil and pressure coefficient distribution on upper surface (dotted line) and lower surface (dashed line) for $R = 1.0$, $Q = 1.0$, $\alpha = 6°$, $\xi_{off} = -0.093R$, $\eta_{off} = 0.08R$, $t/L = 0.1215$, and $b/L = 0.0401$.

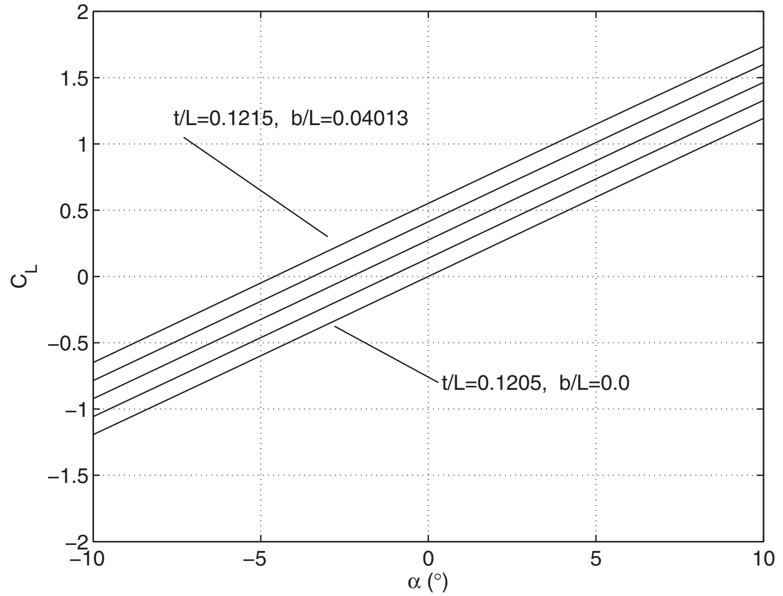


Figure 11.33 Lift coefficient C_L versus angle of attack α for $\xi_{off} = -0.093$ and $\eta_{off} = 0, 0.02, 0.04, 0.06,$ and 0.08.

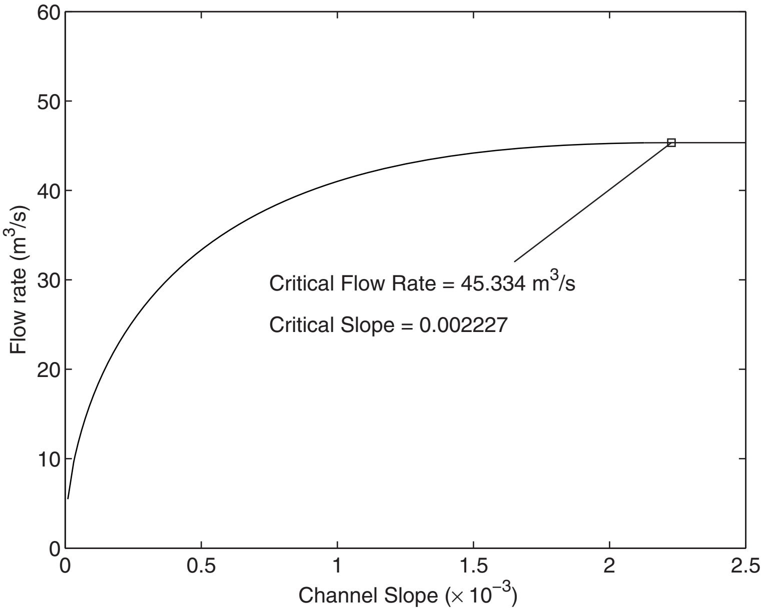


Figure 11.34 Flow rate versus channel slope for a reservoir discharge problem with $y_r = 3.0$ m, $n = 0.014$, $b = 1.5$ m, and $m = 1.0$.

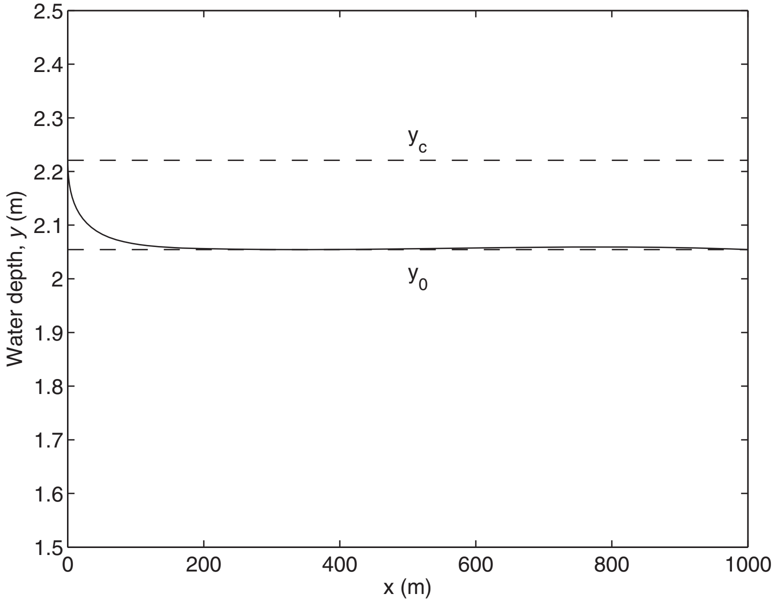


Figure 11.35 Surface profile in a channel with a critical flow inlet condition at $x = 0$ and a slope that creates a supercritical flow for $n = 0.014$, $b = 1.5$ m, $m = 1.0$, $Q = 45.3336$ m^3/s, and $S_0 = 0.003$.

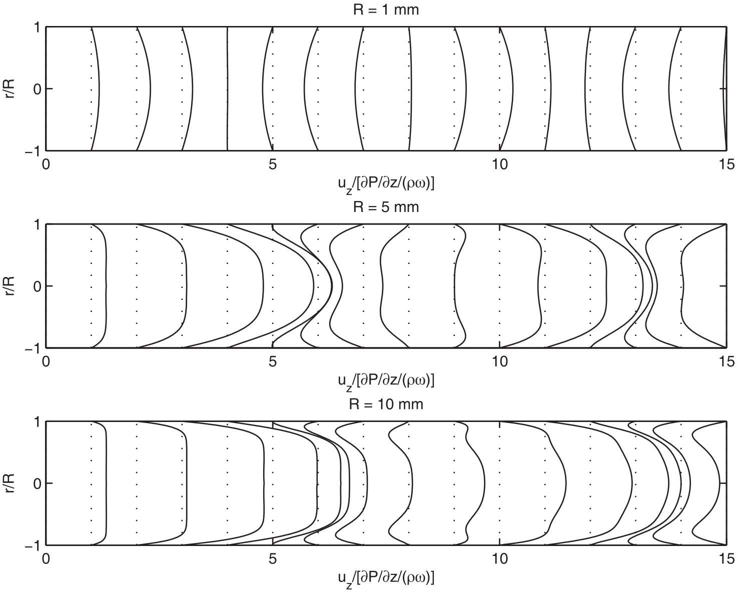


Figure 11.36 Flow profiles in three tubes with radii of 0.001, 0.005, and 0.01 m and the same oscillatory pressure gradient and fluid viscosity. In each case, sixteen velocity profiles corresponding to equally spaced times over the first two periods of the sinusoidally varying pressure gradient are shown. The fluid is initially at rest. The vertical dotted lines are the zero lines for each of the profiles. The results are shown over the diameter of the vessel ($-1 \leq r/R \leq 1$).

Section 11.5

11.14 Consider flow in a tube driven by an oscillatory pressure gradient as described in Section 11.5. Let $\nu = 5.0 \times 10^{-6}\,\text{m}^2/\text{s}$, $\rho = 1{,}000\,\text{kg/m}^3$, and $\omega = 2\pi$ rad/s. Calculate and plot the flow in the tube for three different radii: $R = 0.001, 0.005,$ and 0.01 m. This results in viscous timescales of 0.2, 5.0, and 20.0 s, respectively. The results should look like those given in Figure 11.36.

BIBLIOGRAPHY

B. R. Munson, D. F. Young, and T. H. Okiishi, *Fundamentals of Fluid Mechanics*, 3rd ed., John Wiley & Sons, New York, 1998.

J. D. Anderson, *Fundamentals of Aerodynamics*, 2nd ed., McGraw-Hill, New York, 1991.

V. L. Streeter, E. B. Wylie, and K. W. Bedford, *Fluid Mechanics*, 9th ed., McGraw-Hill, New York, 1998.

R. W. Fox and A. McDonald, *Introduction to Fluid Mechanics*, 5th ed., John Wiley & Sons, New York, 1998.

12

Heat Transfer

Keith E. Herold

Several techniques for analyzing and visualizing conduction, convection, and radiation heat transfer are presented.

12.1 CONDUCTION HEAT TRANSFER

12.1.1 Transient Heat Conduction in a Semi-Infinite Slab with Surface Convection

The transient temperature distribution $T(x, t)$ in a semi-infinite solid is represented by the governing equation[1]

$$\frac{\partial^2 T}{\partial x^2} = \frac{1}{\alpha}\frac{\partial T}{\partial t}$$

where x is the spatial coordinate measured from the free surface into the solid, t is the time, and α is the thermal diffusivity of the solid. We assume that the solid is initially at a uniform temperature T_i; that is, $T(x, 0) = T_i$. In addition, there is convection at the boundary surface $x = 0$, which is given by

$$-k\frac{\partial T(0, t)}{\partial x} = h\big(T_\infty - T(0, t)\big)$$

where h is the heat transfer coefficient, k is the thermal conductivity of the solid, and T_∞ is the ambient air temperature. Upon introducing the nondimensional variables

$$\theta(\eta, \tau) = \frac{T(\eta, \tau) - T_i}{T_\infty - T_i}, \qquad \tau = \frac{h}{k}\sqrt{\alpha t}, \qquad \eta = \frac{hx}{k}$$

the governing equation becomes

$$\frac{\partial^2 \theta}{\partial \eta^2} = \frac{1}{2\tau}\frac{\partial \theta}{\partial \tau}$$

and the convection boundary condition at $\eta = 0$ becomes

$$\frac{\partial \theta}{\partial \eta} = \theta - 1$$

The solution to this system can be written as[1]

$$\theta(\eta, \tau) = \operatorname{erfc}[\eta/(2\tau)] - e^{(\eta+\tau^2)}\operatorname{erfc}[\eta/(2\tau) + \tau]$$

where erfc is the complementary error function. If the initial heat flux at the surface is expressed as $q''_o = h(T_\infty - T_i)$, then the heat flux at the surface $\eta = 0$ is given by

$$q'' = q''_o(1 - \theta) = -q''_o\frac{\partial \theta}{\partial \eta}$$

[1] F. P. Incropera and D. P. DeWitt, *Fundamentals of Heat and Mass Transfer*, 4th ed., John Wiley & Sons, New York, 1996, p. 239.

Example 12.1 Transient heat conduction time and temperature distributions in a semi-infinite solid

We shall examine the temperature in the semi-infinite solid in two ways. In the first way, the temperature is plotted over the range $0 \leq \eta \leq 5$ and $0.01 \leq \tau \leq 3$; these results are shown in Figure 12.1. In the second way, the temperature is plotted as a function of τ for $0.01 \leq \tau \leq 4$ at $\eta = 0, 1, 2, 3, 4,$ and 5. These results are shown in Figure 12.2. The script is

```
tau = linspace(0.01, 3, 30);    eta = linspace(0, 5, 20);
[x, t] = meshgrid(eta, tau);
theta = inline('erfc(0.5*x./t)-exp(x+t.^2).*erfc(0.5*x./t+t) ', 'x', 't');
figure(1)
mesh(x, t, theta(x, t))
xlabel('\eta')
ylabel('\tau')
zlabel('\theta')
figure(2)
eta = 0:5;
tau = linspace(0.01, 4, 40);
for k = 1:length(eta)
  thet = theta(eta(k), tau);
  plot(tau, thet, 'k-')
  text(.92*4,1.02*thet(end), ['\eta = ' num2str(eta(k))])
  hold on
end
xlabel('\tau')
ylabel('\theta')
```

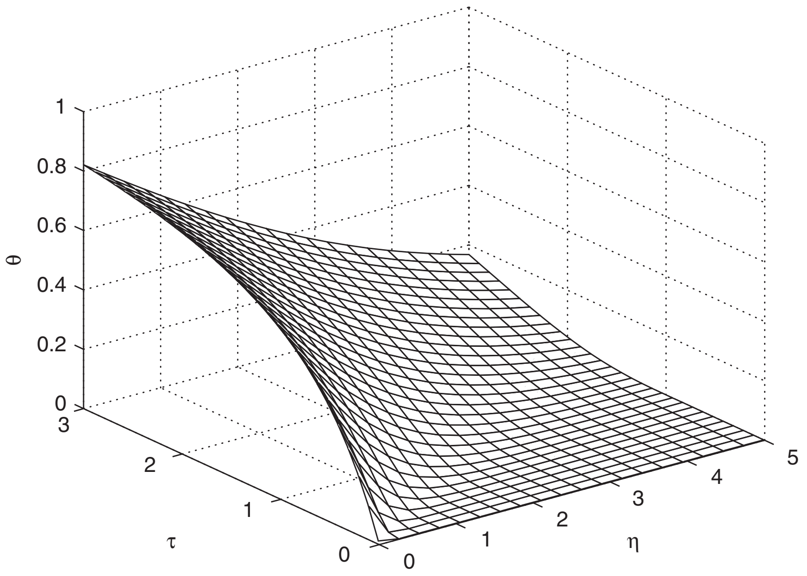


Figure 12.1 Temperature in a semi-infinite solid as a function of nondimensional position η and nondimensional time τ.

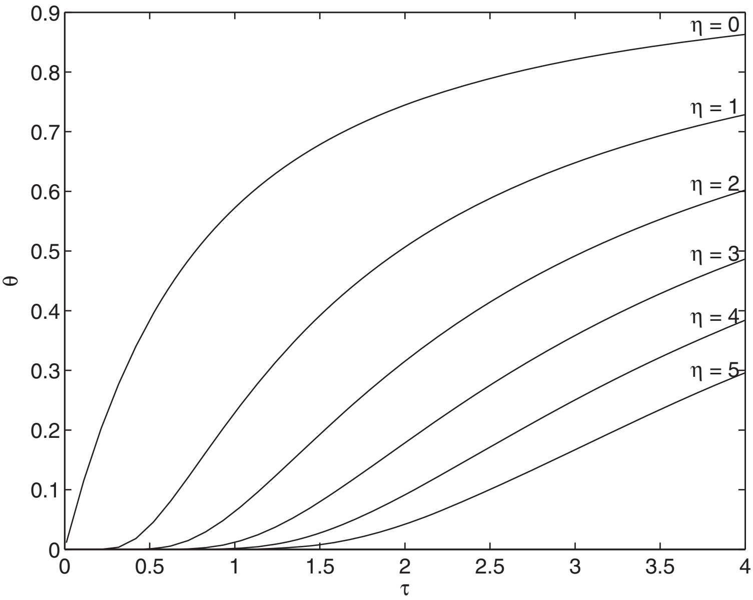


Figure 12.2 Temperature in a semi-infinite solid as a function of nondimensional time τ at several nondimensional locations η.

12.1.2 Transient Heat Conduction in an Infinite Solid Cylinder with Convection

The transient temperature distribution $T(r, t)$ in an infinitely long solid circular cylinder that is initially at a uniform temperature T_i and has convection at the surface is represented by the governing equation[2]

$$\frac{\partial^2 T}{\partial r^2} + \frac{1}{r}\frac{\partial T}{\partial r} = \frac{1}{\alpha}\frac{\partial T}{\partial t}$$

where r is the radial coordinate measured from the center of the cylinder, t is the time, and α is the thermal diffusivity of the cylinder. If the cylinder has a radius R, then the convection boundary condition at $r = R$ is of the form

$$-k\frac{\partial T}{\partial r} = h(T - T_\infty)$$

where T_∞ is the ambient air temperature, h is the heat transfer coefficient, and k is the thermal conductivity of the cylinder. We introduce the following dimensionless variables: $\xi = r/R$, $\mathrm{Bi} = hR/k$ is the Biot number, $\tau = \alpha t/R^2$, and

$$\theta(\xi, \tau) = \frac{T(\xi, \tau) - T_\infty}{T_i - T_\infty}$$

[2] *Ibid.*, p. 229.

Then the governing equation becomes

$$\frac{\partial^2\theta}{\partial\xi^2} + \frac{1}{\xi}\frac{\partial\theta}{\partial\xi} = \frac{\partial\theta}{\partial\tau}$$

and the convection boundary condition at $\xi = 1$ becomes

$$\frac{\partial\theta}{\partial\xi} + \text{Bi}\,\theta = 0$$

The solution can be written as[3]

$$\theta(\xi, \tau) = \sum_{n=1}^{\infty} C_n \exp(-\zeta_n^2\tau)J_0(\zeta_n\xi)$$

where

$$C_n = \frac{2}{\zeta_n}\frac{J_1(\zeta_n)}{J_0^2(\zeta_n) + J_1^2(\zeta_n)}$$

The quantity $J_m(x)$ is the Bessel function of the first kind of order m and ζ_n are the positive roots of

$$\frac{J_1(\zeta_n)}{J_0(\zeta_n)} - \frac{\text{Bi}}{\zeta_n} = 0$$

It is noted that the heat transfer rate in this type of problem can be approximated by a lumped parameter formulation leading to a first-order time constant $\tau_{\text{tc}} = 1/(2\text{Bi})$; this approximation becomes more accurate as the Biot number decreases. In the case considered in the following example, a time constant of $\tau_{\text{tc}} = 1$ is predicted by the lumped parameter model.[4] When a more accurate model is used, as illustrated in Example 12.2, the value of τ for which $\theta = 1/e$ is $\tau_{\text{tc}} = 1.252$ indicating that the simplified model is reasonably accurate for the case where Bi = 0.5.

Example 12.2 Transient heat conduction in an infinite solid cylinder with convection

We shall plot $\theta(\xi, \tau)$ for $0 \le \xi \le 1, 0 \le \tau \le 1.5$, and $\text{Bi} = 0.5$ using the lowest fifteen positive roots of ζ_n. The roots are determined by using **FindZeros**, which was introduced in Section 5.5.1. The script is

```
Bi = .5;    Nroot = 15;
CylinderRoots = inline('x.*besselj(1, x)-Bi*besselj(0, x)', 'x', 'Bi');
r = FindZeros(CylinderRoots, Nroot, linspace(0, 50, 200), Bi);
tau = linspace(0, 1.5, 20);
[t, rt] = meshgrid(tau, r);
```

[3] *Ibid.*

[4] In Exercise 12.9, the value of τ_{tc} for the centerline temperature is determined for the case where $\theta = 1/e = 0.3679$ when a lumped parameter model is used.

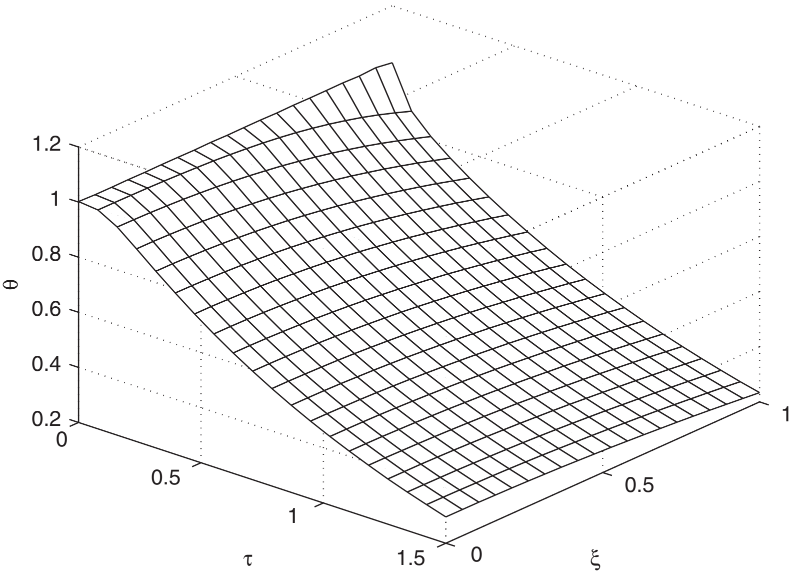


Figure 12.3 Temperature distribution in an infinite cylinder as a function of nondimensional time τ and nondimensional radial position ξ for Bi = 0.5.

```
Fn = exp(-t.*rt.^2);
cn = 2*besselj(1, r)./(r.*(besselj(0, r).^2+besselj(1, r).^2));
ccn = meshgrid(cn, tau);
pro = ccn'.*Fn;
rstar = linspace(0, 1, 20);
[R, rx] = meshgrid(rstar, r);
Jo = besselj(0, rx.*R);
the = Jo'*pro;
[rr, tt] = meshgrid(rstar, tau);
mesh(rr, tt, the')
xlabel('\xi')
ylabel('\tau')
zlabel('\theta')
view(49.5, -34)
```

Execution of this program results in Figure 12.3.

12.1.3 Transient One-Dimensional Conduction with a Heat Source

One-dimensional, transient conduction is governed by the following equation:

$$\frac{1}{\alpha}\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{q}{k}$$

where $T = T(x, t)$ is the temperature, t is the time, x is the spatial coordinate, α is the thermal diffusivity, k is the thermal conductivity, and q is the volumetric heat source.

We convert this equation to a dimensionless form by introducing the following nondimensional quantities:

$$\xi = \frac{x}{L} \quad \tau = \frac{\alpha t}{L^2} \quad \mathrm{Bi} = \frac{hL}{k}$$

$$\theta = \frac{T - T_\infty}{T_i - T_\infty} \quad \Sigma = \frac{L^2 q}{k(T_i - T_\infty)} \quad \chi = \frac{-q'' L}{k(T_i - T_\infty)}$$

where $\theta = \theta(\xi, \tau)$, L is the length of the domain, T_i is an arbitrary temperature that usually represents the initial temperature, q'' is the heat flux, and T_∞ is the fluid temperature for a convective boundary. For cases where a convective boundary condition is not used, T_∞ is an arbitrary temperature with the requirement that it must be different from T_i. In terms of these variables, the governing equation becomes

$$\frac{\partial \theta}{\partial \tau} = \frac{\partial^2 \theta}{\partial \xi^2} + \Sigma \tag{12.1}$$

The domain is illustrated in Figure 12.4.

Typical boundary conditions at each end of the domain are as follows:

Fixed Temperature

$$\theta = \theta_w$$

Specified Flux

$$\frac{\partial \theta}{\partial \xi} = \chi_w$$

Convective

$$\frac{\partial \theta}{\partial \xi} = -\mathrm{Bi}\theta_w$$

where the negative sign in the convective boundary condition is used for the boundary at $\xi = 1$ and the subscript w represents the values at the wall.

Equation (12.1) is a partial differential equation of the parabolic type with one spatial dimension. One method of solution that is applicable to a restricted set of boundary conditions is separation of variables. An example of this type of solution

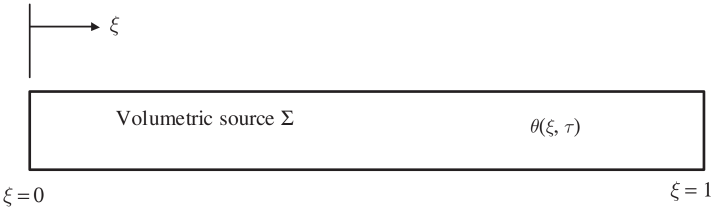


Figure 12.4 Geometry of a one-dimensional transient heat transfer with source Σ.

was used to obtain the result given in Section 12.1.2. In the present case, we shall seek a numerical solution by using pdepe, which solves parabolic partial differential equations with one spatial variable. Recall Section 5.5.6.

Example 12.3 One-dimensional transient heat transfer with source

Consider the data in Table 12.1, which describes a one-dimensional system subjected to a transient source. We shall determine the temperature as a function of time at five locations in the solid: $\xi = 0.0, 0.25, 0.5, 0.75$, and 1.0. In obtaining the solution, we note that pdepe requires three functions: (1) one to define the partial differential equation, which we call **pde1D**; (2) one to define the initial conditions, which we call **pdeIC**; and (3) one to define the boundary conditions, which we call **pdeBC**. The boundary and initial conditions are summarized in Table 12.1.

```
function Example12_3
Bi = 0.1;   Tr = 0.55;   Sigma = 1;
xi = linspace(0, 1, 41);   tau = linspace(0, 1, 101);
theta = pdepe(0, @pde1D, @pdeIC, @pdeBC, xi, tau, [], Bi, Tr, Sigma);
z = 0:0.25:1;
figure(1)
for k = 1:length(z)
  kk = find(xi == z(k));
  plot(tau, theta(:, kk), 'k-')
  hold on
  if k == 1
    text(0.5, 1.02*theta(end, kk), '\xi = 0.0 and 0.25')
  elseif k > 2
    text(0.5, theta(end, kk)+.02, ['\xi = ' num2str(xi(kk))])
  end
end
axis([0 1 0.5 1])
xlabel('\tau')
ylabel('\theta')
figure(2)
[thmin imin] = min(theta(:,1));
plot(xi, theta(1,:), 'k-', 'LineWidth', 2)
hold on
```

TABLE 12.1 Input Values for Example 12.3

Parameter	Value
Boundary Conditions and Source	
Dimensionless source strength:	$\Sigma = 1$
At $\xi = 0$:	$\mathrm{Bi} = 0.1$
At $\xi = 1$:	$\theta(1, \tau) = \theta_w = 0.55$
Initial Conditions	
Linear distribution:	$\theta(\xi, 0) = 1 - 0.45\xi$
Numerical Grid Parameters	
Number of equally spaced grid points in $0 < \xi < 1$	41
Number of equally spaced time steps $0 < \tau < 1$	101

```
plot(xi, theta(2,:) , 'k', xi, theta(imin,:), 'k:')
solinit = bvpinit(linspace(0, 1, 20), [1 1]);
sol = bvp4c(@barode, @barbc, solinit, [], Bi, Sigma);
x = linspace(0, 1, 100);
y = deval(sol, x);
plot(x, y(1,:), 'k--');
xlabel('\xi')
ylabel('\theta')
legend(['\tau = 0 (Initial condition)'], ['\tau = ' num2str(tau(2))], . . .
   ['\tau = ' num2str(tau(imin)) ' (Minimum at \xi = 0)'], . . .
   '\tau > 2 (Steady state)', 'Location', 'SouthWest')

function dydx = barode(x, y, Bi, Sigma)
dydx = [y(2), -Sigma]';

function res = barbc(ya, yb, Bi, Sigma)
res = [ya(2)-Bi*ya(1), yb(1)-0.55]';

function [c, f, s] = pde1D(x, t, u, DuDx, Bi, Tr, Sigma)
c = 1;    f = DuDx;    s = Sigma;

function T0 = pdeIC(x, Bi, Tr, Sigma)
T0 = 1-0.45*x;

function [pl, ql, pr, qr] = pdeBC(xl, ul, xr, ur, t, Bi, Tr, Sigma)
pr = ur-Tr;    qr = 0;
pl = -Bi*ul;    ql = 1;
```

The execution of this program results in Figures 12.5 and 12.6. The initial condition, which is a linear temperature profile, is shown in Figure 12.6 as a straight line and in Figure 12.5

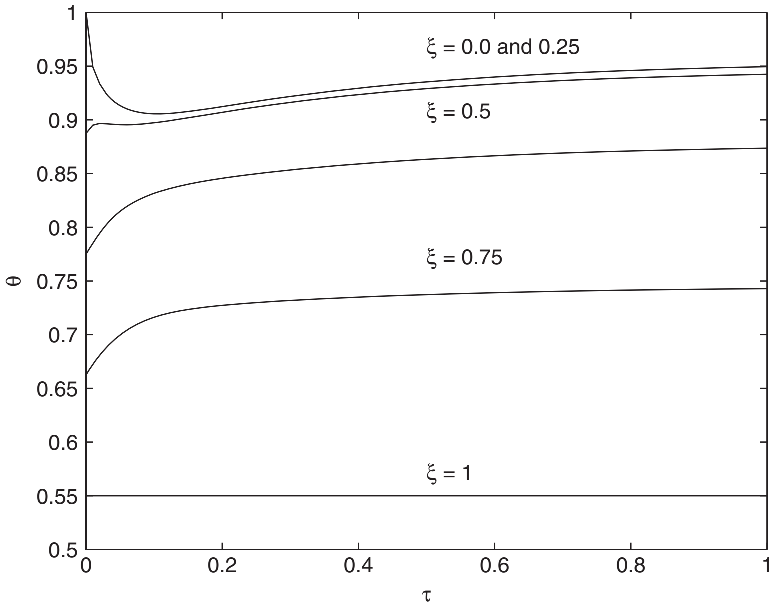


Figure 12.5 One-dimensional heat conduction using the data in Table 12.1. For the same data plotted against the spatial coordinate with time as a parameter, see Figure 12.6.

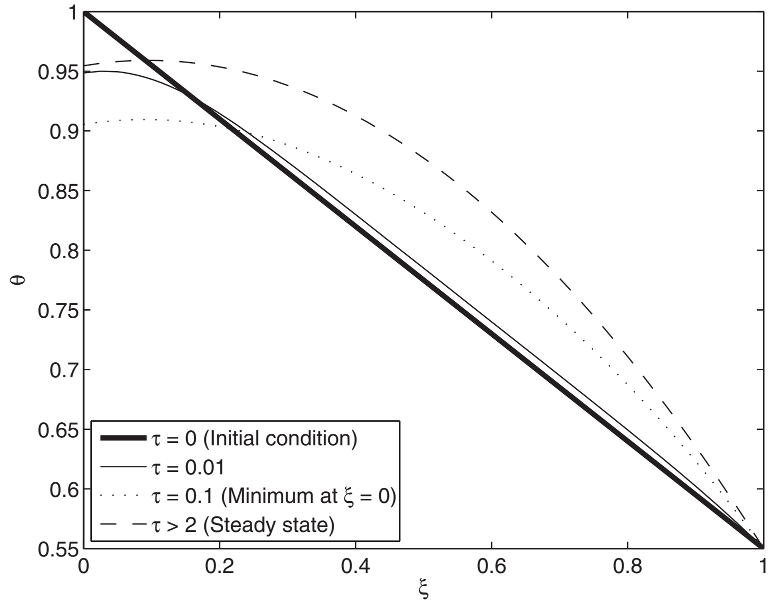


Figure 12.6 One-dimensional heat conduction using the data in Table 12.1. For the same data plotted against time with location as a parameter, see Figure 12.5.

by the equal spacing between the curves at $\tau = 0$. The top curve in Figure 12.5 is the temperature at $\xi = 0$ where convection is occurring. At small values of τ, the temperature drops rapidly due to convection and conduction. After about $\tau = 0.1$, the temperature of the surface begins to rise as the steady-state temperature profile, defined by the energy source, is approached. In Figure 12.6, the profile at the time of lowest temperature at $\xi = 0$ is shown as a dotted line. The steady-state solution could be obtained by running `pdepe` for large τ. To confirm that result, the steady-state curve plotted in Figure 12.6 was computed using `bvp4c`.

12.2 CONVECTION HEAT TRANSFER

12.2.1 Internal Flow Convection: Pipe Flow

The energy equation for flow in a pipe, where axial conduction is assumed to be negligible compared to the advection and symmetry about the longitudinal axis is assumed, can be written as

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial T}{\partial r}\right) = \frac{u}{\alpha}\frac{\partial T}{\partial z}$$

where $T = T(r, z)$ is the fluid temperature, r is the radial coordinate measured from the centerline, z is the axial coordinate, u is the fluid velocity, and α is the thermal diffusivity. We introduce the dimensionless coordinates $\xi = r/R$ and $\zeta = z/L$, where R is the pipe radius and L is the pipe length. In addition, we introduce the fluid velocity as $u = 2V(1 - \xi^2)$ where V is the average velocity in the pipe and u has the parabolic profile expected in fully developed laminar flow. Then, the governing equation becomes

$$\frac{1}{\xi}\frac{\partial}{\partial \xi}\left(\xi \frac{\partial T}{\partial \xi}\right) = \text{Re Pr } \frac{R}{L}(1 - \xi^2)\frac{\partial T}{\partial \zeta}$$

where $T = T(\xi, \zeta)$, $\text{Re} = 2RV/\nu$ is the Reynolds number, $\text{Pr} = \nu/\alpha$ is the Prandtl number, and ν is the momentum diffusivity. The auxiliary conditions needed to solve this equation include an entry condition on T at $\zeta = 0$; that is, $T(\xi, 0)$, and two boundary conditions. The boundary condition at the centerline is symmetry. Two wall boundary conditions of interest are the constant wall temperature and constant wall heat flux. The wall heat flux can be written as

$$q'' = -k\frac{\partial T}{\partial r} = -\frac{k}{R}\frac{\partial T}{\partial \xi}$$

The mean temperature is defined at a pipe cross section at a specified z as

$$T_m(z) = 2\pi \int_0^R uTrdr \Big/ 2\pi \int_0^R urdr \tag{12.2a}$$

or

$$T_m(\zeta) = 4\int_0^1 (1 - \xi^2)\xi T(\xi, \zeta)d\xi \tag{12.2b}$$

The mean temperature can be used to define a useful dimensionless temperature θ as

$$\theta(\xi, \zeta) = \frac{T(\xi, \zeta) - T_w}{T_m(\zeta) - T_w}$$

where T_w is the wall temperature. With this definition, Eq. (12.2b) becomes

$$4\int_0^1 \theta(\xi, \zeta)(1 - \xi^2)\xi d\xi = 1$$

The temperature θ is used to define the concept of a thermally fully developed state, which is required to satisfy

$$\frac{d\theta}{d\zeta} = 0$$

The heat transfer coefficient is defined as

$$h = \frac{q''}{T_m - T_w}$$

which is a constant in the thermally fully developed region. It is common practice to express the heat transfer coefficient in dimensionless form as a Nusselt number, denoted Nu,

$$\mathrm{Nu} = \frac{h2R}{k}$$

It is well known[5] that for fully developed flow, Nu = 3.66 for a constant wall temperature and Nu = 4.36 for a constant heat flux.

Example 12.4 Heat transfer coefficient for laminar flow in a pipe

In this example, we shall use pdepe to solve the governing equation for the standard boundary conditions: (i) constant wall temperature and (ii) constant heat flux. It is assumed that we have laminar flow with Re = 40 and Pr = 5, and the pipe radius $R = 0.01$ m and the pipe length $L = 0.5$ m. In addition, we assume that the boundary condition for constant heat flux is $q_w = 10\ \mathrm{W/m^2}$, the boundary condition for constant wall temperature is $T_w = 40°\mathrm{C}$, the entry condition is $T(\xi, 0) = 20°\mathrm{C}$, and $k = 0.6$ W/(m K). The program is as follows:

```
function Example12_4
Tw = 40;   qw = 10;   Re = 40;   Pr = 5;
R = 0.01;   L = 0.5;   k = 0.6;
Rt = 401;   zt = 50;   dxi = 1/(Rt-1);
xi = linspace(0, 1, Rt);
zeta = linspace(0, 1, zt);
solT = pdepe(1, @pdepde, @pdeic, @pdebcT, xi, zeta, [], Tw, qw, Re, Pr, R, L, k);
solF = pdepe(1, @pdepde, @pdeic, @pdebcF, xi, zeta, [], Tw, qw, Re, Pr, R, L, k);
NuT = zeros(zt,1);   NuF = NuT;
figure(1)
for i = 1:zt
   TmT = 4*trapz(xi, xi.*(1-xi.^2).*solT(i,:));
   dThdxiT = (solT(i,Rt)-solT(i,Rt-1))/(dxi*(TmT-solT(i,Rt)));
   NuT(i) = -2*dThdxiT;
   TmF = 4*trapz(xi, xi.*(1-xi.^2).*solF(i,:));
   dThdxiF = (solF(i,Rt)-solF(i,Rt-1))/(dxi*(TmF-solF(i,Rt)));
   NuF(i) = -2*dThdxiF;
end
ThT = (solT(end,:)-ones(1,Rt)*Tw)/(TmT-Tw);
ThF = (solF(end,:)-ones(1,Rt)*solF(end,Rt))/(TmF-solF(end,Rt));
plot(xi, ThT, 'k-', xi, ThF, 'k--')
xlabel('\xi')
ylabel('\theta')
legend('Constant wall temperature', 'Constant wall heat flux')
figure(2)
plot(zeta, NuT, 'k-', zeta, NuF, 'k--')
xlabel('\zeta')
```

[5] W. M. Kays and M. E. Crawford, *Convective Heat and Mass Transfer*, 2nd ed., McGraw Hill, New York, 1980, p. 96.

```
ylabel('Nu')
ylim([0 6])
legend('Constant wall temperature', 'Constant wall heat flux')
function [c, f, s] = pdepde(xi, zeta, T, DTDxi, Tw, qw, Re, Pr, R, L, k)
c = Re*Pr*R/L*(1-xi^2);
f = DTDxi;    s = 0;
function T0 = pdeic(xi, Tw, qw, Re, Pr, R, L, k)
T0 = 20;
function [pl, ql, pr, qr] = pdebcT(xil, Tl, xir, Tr, zeta, Tw, qw, Re, Pr, R, L, k)
pl = 0;    ql = 1;
pr = Tr-Tw;    qr = 0;
function [pl, ql, pr, qr] = pdebcF(xil, Tl, xir, Tr, zeta, Tw, qw, Re, Pr, R, L, k)
pl = 0;    ql = 1;
pr = -qw;    qr = k/R;
```

Execution of the program results in Figures 12.7 and 12.8. In Figure 12.7, the temperature as a function of the nondimensional radial position in the pipe outlet is given. It is noted that the different boundary conditions result in a different temperature slope at the wall. This temperature slope, and the accompanying differences in shape of the temperature profiles, translates into a different heat transfer coefficient as shown in Figure 12.8. The asymptotic behavior of the heat transfer coefficient indicates the approach to the thermally fully developed state. The values of the fully developed Nusselt numbers computed are within 1% of published values; the calculation can be made more accurate by increasing the number of spatial grid points *Rt*.

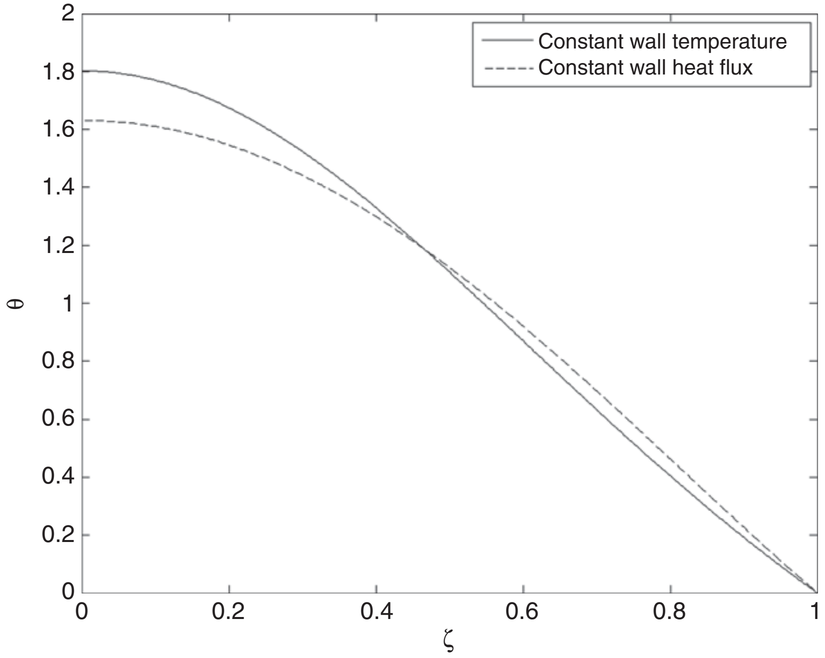


Figure 12.7 Temperature versus radial position for laminar pipe flow with different boundary conditions: (i) constant wall temperature and (ii) constant wall heat flux.

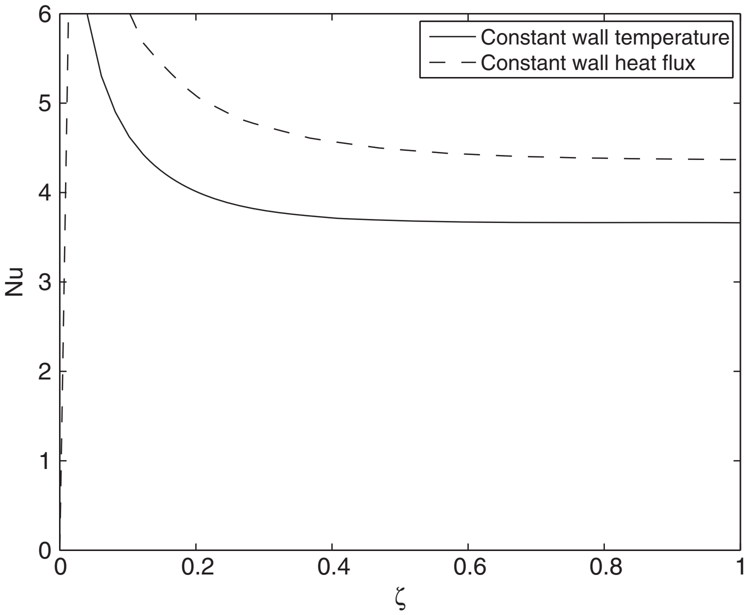


Figure 12.8 Nusselt number versus axial position for laminar pipe flow with different boundary conditions: (i) constant wall temperature and (ii) constant wall heat flux.

12.2.2 Thermal Boundary Layer on a Flat Plate: Similarity Solution

The velocity profiles for laminar boundary layer flow over a flat plate shown in Figure 12.9 are obtained from the solution of the following Blasius equation:[6]

$$\frac{d^3f}{d\eta^3} + \frac{f}{2}\frac{d^2f}{d\eta^2} = 0 \qquad (12.3)$$

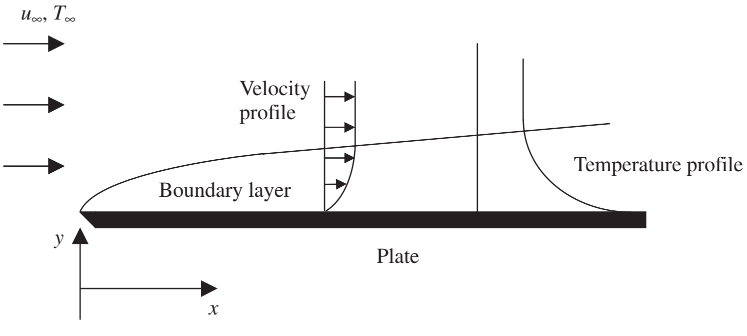


Figure 12.9 Flow over a flat plate.

[6] Incropera and DeWitt, *Fundamentals of Heat*, pp. 350–352.

where f is the modified stream function given by

$$f = \frac{\psi}{u_\infty \sqrt{\nu_{\text{vis}} x / u_\infty}}$$

The stream function ψ is defined such that

$$u = \partial\psi/\partial y$$
$$v = -\partial\psi/\partial x$$

where u and v are the velocities in the x and y directions, respectively, and η is the similarity variable

$$\eta = y\sqrt{u_\infty/(\nu_{\text{vis}} x)}$$

The free stream velocity is u_∞ and ν_{vis} is the kinematic viscosity of the fluid. Solution of the Blasius equation gives the velocity at any location within the boundary layer. See also Section 11.3.2.

Under conditions of constant fluid properties and certain boundary layer assumptions, the thermal energy equation for the fluid can be expressed in terms of the similarity variable as[7]

$$\frac{d^2T^*}{d\eta^2} + \Pr\frac{f}{2}\frac{dT^*}{d\eta} = 0 \tag{12.4}$$

where T^* is a dimensionless temperature given by

$$T^* = \frac{T - T_s}{T_\infty - T_s}$$

and T is the fluid temperature, T_s is the plate surface temperature, T_∞ is the fluid-free stream temperature, and the Prandtl number is $\Pr = \nu_{\text{vis}}/\alpha$, where α is the thermal diffusivity of the fluid. Note that T^* is coupled to the velocity solution through the presence of f in the thermal energy equation.

The boundary conditions are

$$\begin{aligned} &f(0) = 0 \qquad \left.\frac{df}{d\eta}\right|_{\eta=0} = 0 \qquad \left.\frac{df}{d\eta}\right|_{\eta\to\infty} \to 1 \\ &T^*(0) = 0 \qquad T^*(\eta\to\infty)\to 1 \end{aligned} \tag{12.5}$$

We shall now use these results in the following example.

[7] *Ibid.*

Example 12.5 Heat transfer from a flat plate: Blasius formulation

We shall obtain a solution to the extended Blasius formulation of heat transfer from a flat plate for Pr = 0.07, 0.7, and 7.0. The solution is obtained with bvp4c, with the boundary conditions at $\eta \rightarrow \infty$ being approximated by assuming a large value for η called $\eta_{\max}$. The two coupled nonlinear equations given by Eqs. (12.3) and (12.4) are decomposed into a set of five coupled first-order ordinary differential equations by introducing the following set of dependent variables:

$$
\begin{aligned}
y_1 &= f & y_4 &= T^* \\
y_2 &= \frac{df}{d\eta} & y_5 &= \frac{dT^*}{d\eta} \\
y_3 &= \frac{d^2f}{d\eta^2}
\end{aligned}
$$

where y_1 represents the stream function, y_2 the velocity, y_3 the shear, y_4 the temperature, and y_5 the heat flux. These quantities are used in Eqs. (12.3) and (12.4) to obtain the five first-order differential equations:

$$
\begin{aligned}
\frac{dy_1}{d\eta} &= y_2 & \frac{dy_4}{d\eta} &= y_5 \\
\frac{dy_2}{d\eta} &= y_3 & \frac{dy_5}{d\eta} &= -\frac{\text{Pr}}{2}y_1y_5 \\
\frac{dy_3}{d\eta} &= -\frac{1}{2}y_1y_3
\end{aligned}
$$

The boundary conditions given by Eq. (12.5) can be expressed in terms of the new variables as

$$
\begin{aligned}
y_1(0) &= 0 & y_4(\eta \rightarrow \infty) &\rightarrow 1 \\
y_2(0) &= 0 & y_4(0) &= 0 \\
y_2(\eta \rightarrow \infty) &\rightarrow 1
\end{aligned}
$$

We create the following program, which includes two subfunctions that support bvp4c. The function **BlasiusT** defines the five first-order differential equations and the function **BlasiusTbc** defines the boundary conditions. To approximate $\eta \rightarrow \infty$, we use a value of $\eta_{\max} = 8$ for Pr = 0.7 and 7.0, and a value of $\eta_{\max} = 15$ for Pr = 0.07.

```
function Example12_5
Pr = [0.07, 0.7, 7.0];    etaMax = [15, 8, 8];    xm = [15, 5, 5];
for k=1:3
  figure(k)
  solinit = bvpinit(linspace(0, etaMax(k), 8), [0, 0, 0, 0, 0]);
  sol = bvp4c(@BlasiusT, @BlasiusTbc, solinit, [], Pr(k));
```

```
   eta = linspace(0, etaMax(k));
   y = deval(sol, eta);
   subplot(2, 1, 1)
   plot(eta, y(1,:), '-.k', eta, y(2,:), '-k', eta, y(3,:), '--k')
   xlabel('\eta')
   ylabel('y_1, y_2, y_3')
   legend('Stream function, f = y_1', 'Velocity, df/d\eta = y_2', . . .
            'Shear, d^2f/d\eta^2= y_3')
   axis([0 xm(k) 0 2])
   subplot(2,1,2)
   plot(eta, y(4,:), '-k', eta, y(5,:), '--k')
   axis([0 xm(k) 0 2])
   legend('Temperature, T^* = y_4', 'Heat flux, dT^*/d\eta = y_5')
   xlabel('\eta')
   ylabel('y_4, y_5')
end

function F = BlasiusT(eta, y, Pr)
F = [y(2); y(3); -0.5*y(1)*y(3); y(5); -Pr*0.5*y(1)*y(5)];

function res = BlasiusTbc(ya, yb, Pr)
res = [ya(1); ya(2); ya(4); yb(2)-1; yb(4)-1];
```

The results from executing this program are shown in Figures 12.10–12.12.

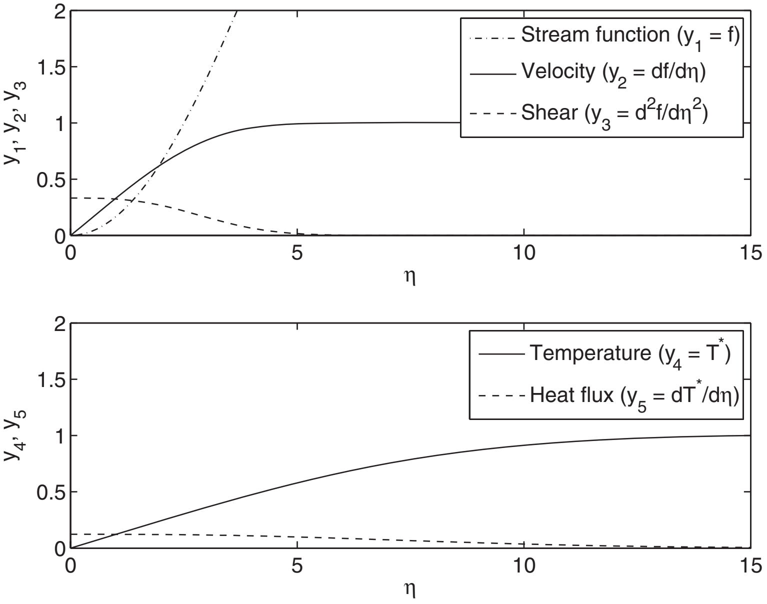


Figure 12.10 Extended Blasius solution for Pr = 0.07.

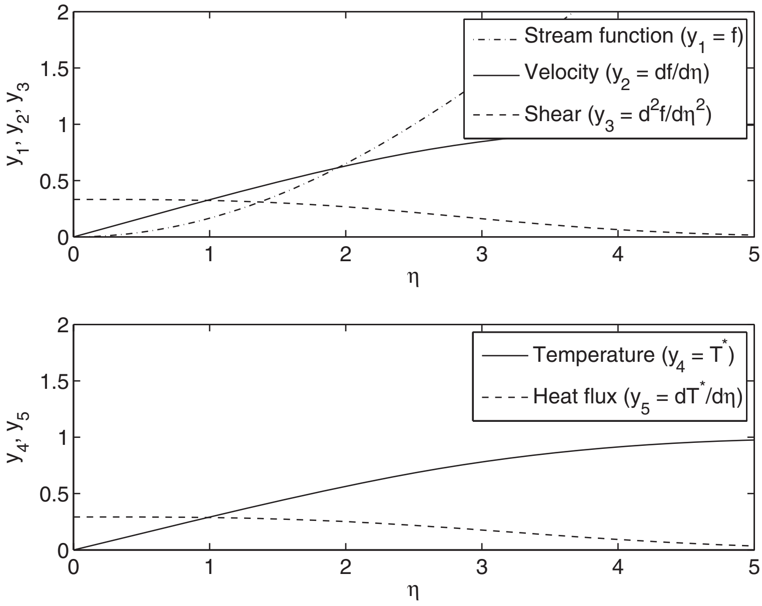


Figure 12.11 Extended Blasius solution for Pr = 0.7.

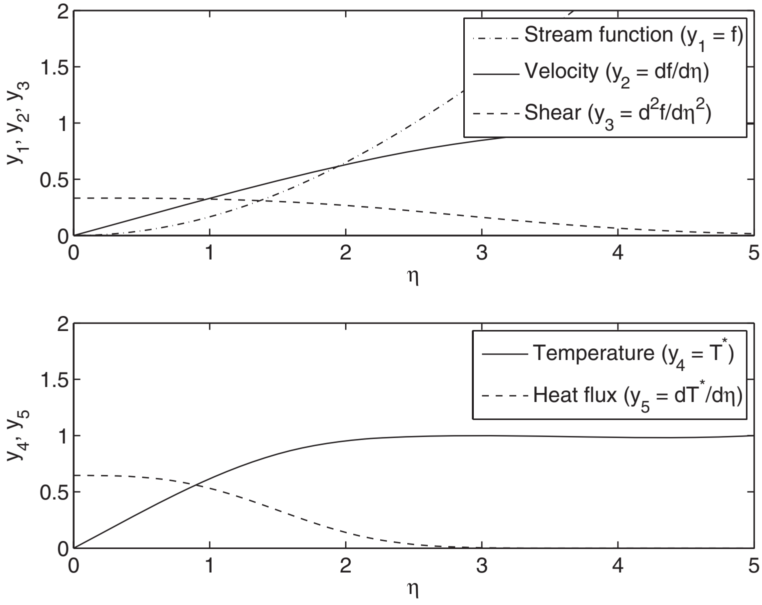


Figure 12.12 Extended Blasius solution for Pr = 7.

12.2.3 Natural Convection Similarity Solution

Natural convection along a heated vertical plate in contact with a cooler fluid is shown in Figure 12.13. The bulk fluid is quiescent, but the heat transfer from the plate causes buoyancy-driven flow. This flow is described by the following two coupled nonlinear ordinary differential equations:[8]

$$\frac{d^3f}{d\eta^3} + 3f\frac{d^2f}{d\eta^2} - 2\left(\frac{df}{d\eta}\right)^2 + T^* = 0$$

$$\frac{d^{2*}T^*}{d\eta^2} + 3\mathrm{Pr}\, f\frac{dT^*}{d\eta} = 0$$

where f is the modified stream function

$$f = \frac{\psi}{4\nu_{\mathrm{vis}}(\mathrm{Gr}_x/4)^{0.25}}$$

The stream function ψ is defined such that

$$u = \partial\psi/\partial y$$

$$v = -\partial\psi/\partial x$$

where u and v are the velocities in the x and y directions, respectively. The quantity η is the similarity variable

$$\eta = \frac{y}{x}\left(\frac{\mathrm{Gr}_x}{4}\right)^{0.25}$$

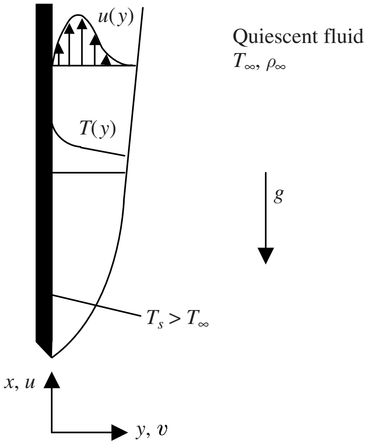


Figure 12.13 Natural convection plume along a heated plate.

[8] *Ibid.*, pp. 487–490.

defined in terms of the Grashof number

$$\mathrm{Gr}_x = g\beta(T_s - T_\infty)x^3/\nu_{\mathrm{vis}}^2$$

where T_∞ is the far-field temperature, T_s is the plate surface temperature, g is the acceleration of gravity, β is the coefficient of thermal expansion

$$\beta = -\frac{1}{\rho}\left(\frac{\partial\rho}{\partial T}\right)_p$$

and ν_{vis} is the kinematic viscosity. The quantity Pr is the Prandtl number defined previously, and the quantity T^* is the dimensionless temperature given by

$$T^* = \frac{T - T_\infty}{T_s - T_\infty}$$

The boundary conditions for this system are

$$\eta = 0: f = 0, \quad \frac{df}{d\eta} = 0, \qquad T^* = 1$$

$$\eta \to \infty: \frac{df}{d\eta} \to 0, \qquad T^* \to 0$$

This system can be decomposed into a system of five first-order equations by introducing the following set of dependent variables:

$$y_1 = f \qquad y_4 = T^*$$

$$y_2 = \frac{df}{d\eta} \qquad y_5 = \frac{dT^*}{d\eta}$$

$$y_3 = \frac{d^2f}{d\eta^2}$$

Then, the set of first-order differential equations in terms of these new variables are

$$\frac{dy_1}{d\eta} = y_2 \qquad \frac{dy_4}{d\eta} = y_5$$

$$\frac{dy_2}{d\eta} = y_3 \qquad \frac{dy_5}{d\eta} = -3\mathrm{Pr}\, y_1 y_5$$

$$\frac{dy_3}{d\eta} = 2y_2^2 - 3y_1y_3 - y_4$$

and the corresponding boundary conditions become

$$y_1(0) = 0 \quad y_4(0) = 1$$

$$y_2(0) = 0 \quad y_4(\eta \to \infty) \to 0$$

$$y_2(\eta \to \infty) \to 0$$

We shall now use these results in the following example.

Example 12.6 Natural convection along a heated plate

We shall obtain a solution to the system of equations describing the natural convection along the heated plate for $\mathrm{Pr} = 0.07, 0.7$, and 7.0. The value of η that is used to approximate the condition as $\eta \to \infty$ is denoted $\eta_{\max}$. The appropriate value of $\eta_{\max}$ depends on the Prandtl number, with larger Prandtl numbers requiring smaller values of $\eta_{\max}$. A reasonably accurate solution is found when the solution becomes independent of the choice of $\eta_{\max}$. In this case, $\eta_{\max}$ was set equal to 8 for $\mathrm{Pr} = 0.7$ and 7 and equal to 11 for $\mathrm{Pr} = 0.07$.

To solve this boundary value problem, we use bvp4c, which requires two functions: **NaturalConv**, which defines the five ordinary differential equations; and **NaturalConvbc**, which defines the boundary conditions. The primary function and subfunctions to obtain the solution are as follows:

```
function Example12_6
Pr = [.07 .7 7];   etaMax = [11, 8, 8];   xm = [10, 5, 5];   ym = [2, 0.8, 0.5];
guess = [0 0 0 0 0];
for k = 1:3
  figure(k)
  solinit = bvpinit(linspace(0, etaMax(k), 5), guess);
  sol = bvp4c(@NatConv, @NatConvBC, solinit, [], Pr(k));
  eta = linspace(0, etaMax(k), 300);
  y = deval(sol, eta);
  subplot(2, 1, 1)
  plot(eta, y(1,:), '-.k', eta, y(2,:), '-k', eta, y(3,:), '--k')
  legend('Stream function, f = y_1', 'Velocity, df/d\eta = y_2', . . .
         'Shear, d^2f/d\eta^2 = y_3')
  axis([0 xm(k) -0.2 ym(k)])
  xlabel('\eta')
  ylabel('y_1, y_2, y_3')
  subplot(2, 1, 2)
  plot(eta, y(4,:), '-k', eta, y(5,:), '--k')
  legend('Temperature, T^* = y_4', 'Heat flux, dT^*/d\eta = y_5')
  axis([0 xm(k) -1.2 1])
  xlabel('\eta')
  ylabel('y_4, y_5')
end

function ff = NatConv(eta, y, Pr)
ff = [y(2); y(3); -3*y(1)*y(3)+2*y(2)^2-y(4); y(5); -3*Pr*y(1)*y(5)];

function res = NatConvBC(ya, yb, Pr)
res = [ya(1); ya(2); ya(4)-1; yb(2); yb(4)];
```

The results of the execution of this script are shown in Figures 12.14–12.16. Referring to these figures, which are for Prandtl numbers 0.07, 0.7, and 7.0, respectively, we see that in all three cases there is a wall plume where the velocity attains a maximum value in the vicinity of $\eta = 1$. The shear stress in the fluid in a direction parallel to the wall is

$$\tau_s = \frac{\sqrt{2}\nu^2\rho}{x^2}\,\mathrm{Gr}_x^{0.75}\,\frac{d^2f}{d\eta^2}$$

which goes to zero at the location where the velocity is maximum.

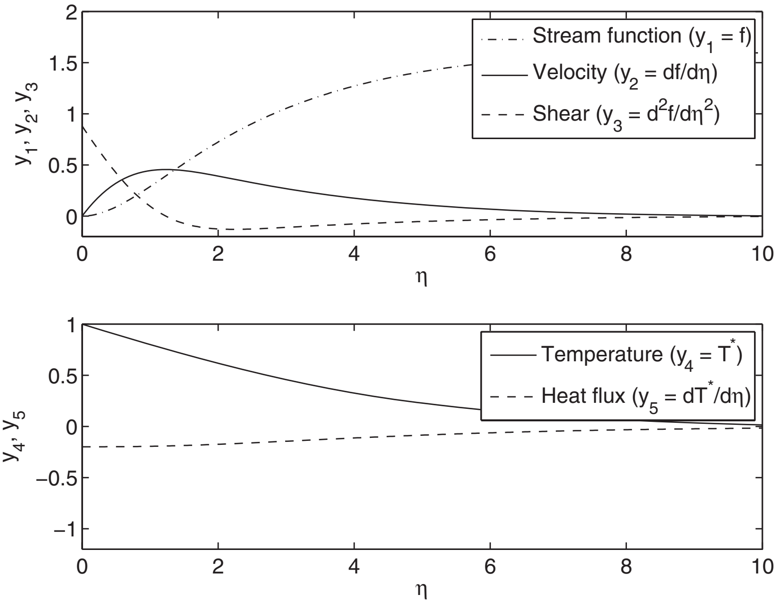


Figure 12.14 Natural convection solution for Pr = 0.07.

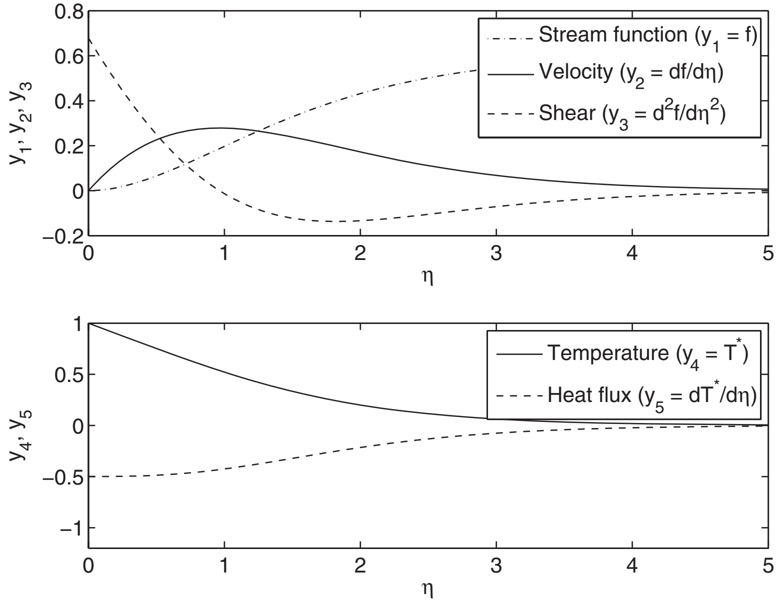


Figure 12.15 Natural convection solution for Pr = 0.7.

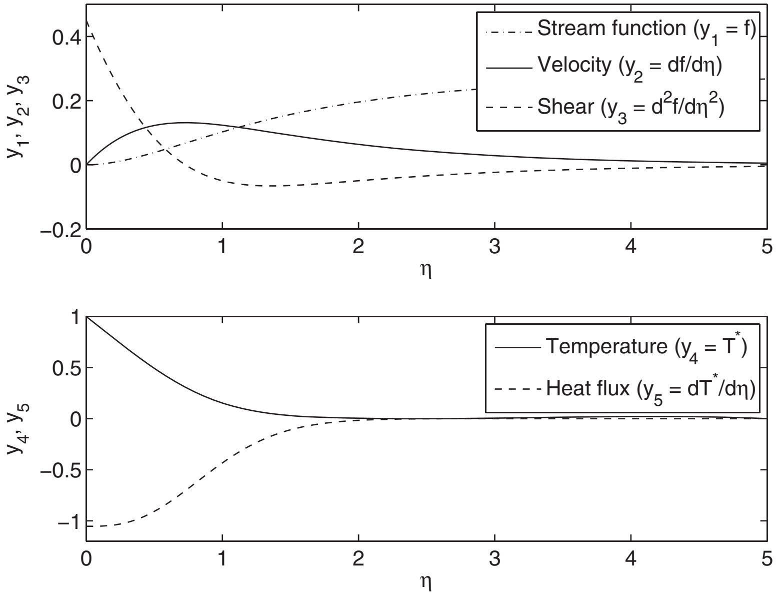


Figure 12.16 Natural convection solution for Pr = 7.

The thermal effects drive the flow. As a result, for a thermally driven wall plume, the velocity boundary layer thickness is never less than the temperature boundary layer thickness. This is different from the result obtained for the corresponding case for forced flow over a flat plate in Section 12.2.2, where the thickness of the velocity boundary layer is much less than the thickness of the temperature boundary layer for the case Pr = 0.07.

The maximum value of the stream function is a measure of the pumping action provided by the heating of the fluid, which is a strong function of Pr. High values of Pr yield low values of the modified stream function *f*. The maximum value of the modified stream function is related to the total volumetric flow rate in the plume. However, to interpret this for a particular fluid, one must compute the dimensional stream function using

$$\psi(x, y) = 4\nu_{\text{vis}} f \left(\frac{\text{Gr}_x}{4} \right)^{0.25}$$

When this is done, it is found that the volumetric flow rate for air at the same temperature difference is significantly greater than it is for water.

Since the flow carries energy away from the surface, a similar analysis is of interest for the heat flux, which is determined from

$$q_s'' = -\frac{k(T_s - T_\infty)}{x} \left(\frac{\text{Gr}_x}{4} \right)^{0.25} \frac{dT^*}{d\eta}\bigg|_{\eta=0}$$

When the heat flux is computed for both air and water, it is found that the heat flux for water is on the order of 100 times greater than that for air. This is primarily due to the roles of thermal conductivity, specific heat, and density, which determine the magnitude of the heat flux. At atmospheric pressure and 300K, the thermal conductivity is approximately 30 times greater for water as compared with air, the specific heat is 4 times greater, and the density is 1,000 times greater. Thus, even though the volumetric flow rate in the plume is larger for air, the heat flux is larger for water.

12.3 RADIATION HEAT TRANSFER

12.3.1 Radiation View Factor: Differential Area to Arbitrary Rectangle in Parallel Planes

The computation of radiation view factors is required when analyzing the radiation in enclosures with diffuse surfaces. There are numerous techniques for evaluating these factors, many of which apply to specific geometries. A more general approach is to start from the relations that define the view factor and integrate them numerically. Consider first the general expression[9] for the view factor between a differential area element dA_1 and a finite area A_2:

$$dF_{2-d_1} = \frac{dA_1}{A_2} \int\limits_{A_2} \frac{\cos\theta_1 \cos\theta_2}{\pi S^2} dA_2 \tag{12.6}$$

where S is the line-of-sight distance between dA_1 and some position on A_2, as shown in Figure 12.17. The angles $\theta_j, j = 1, 2$, are measured between the normal to the surface and S. The reciprocity relation for the view factors is

$$A_2 dF_{2-d_1} = dA_1 F_{d_1-2} \tag{12.7}$$

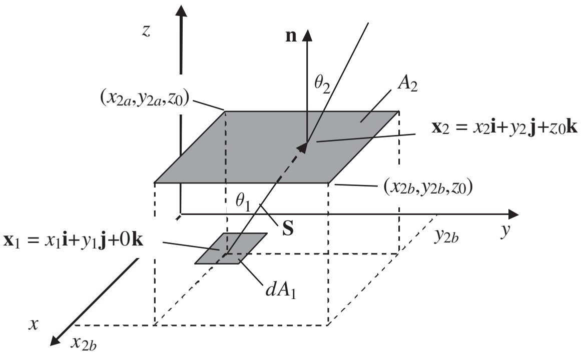

z
n
θ_2
A_2
(x_{2a}, y_{2a}, z_0)
$\mathbf{x}_2 = x_2\mathbf{i} + y_2\mathbf{j} + z_0\mathbf{k}$
(x_{2b}, y_{2b}, z_0)
θ_1
S
y
y_{2b}
$\mathbf{x}_1 = x_1\mathbf{i} + y_1\mathbf{j} + 0\mathbf{k}$
dA_1
x
x_{2b}

Figure 12.17 Geometry when the differential area and the finite rectangle are in parallel planes.

[9] R. Siegel and J. R. Howell, *Thermal Radiation Heat Transfer*, 3rd ed., Hemisphere Publishing, Washington, 1992, pp. 189–252.

Thus, F_{d_1-2} can be written as

$$F_{d_1-2} = \int_{A_2} \frac{\cos\theta_1 \cos\theta_2}{\pi S^2} dA_2 \tag{12.8}$$

Consider the case where dA_1 and A_2 are in parallel planes and A_2 is a rectangle. Both of these restrictions could be removed with additional programming effort. For this case, Eq. (12.8) can be written as

$$F_{d_1-2} = \frac{1}{\pi}\int_{y_{2a}}^{y_{2b}}\int_{x_{2a}}^{x_{2b}} \frac{\cos\theta_1 \cos\theta_2}{S^2} dx_2 dy_2$$

where the line-of-sight vector **S** is

$$\mathbf{S} = \mathbf{x}_2 - \mathbf{x}_1 = (x_2 - x_1)\mathbf{i} + (y_2 - y_1)\mathbf{j} + (z_2 - z_1)\mathbf{k}$$

and $S = |\mathbf{S}|$. Since the two surfaces are parallel, $\theta_1 = \theta_2 = \theta$ and the angles can be expressed in terms of the line-of-sight vector and a normal vector to the rectangle $\mathbf{n} = \mathbf{k}$ as

$$\cos\theta_1 = \cos\theta_2 = \frac{\mathbf{n}\cdot\mathbf{S}}{|\mathbf{S}|}$$

Then,

$$F_{d_1-2} = \frac{1}{\pi}\int_{y_{2a}}^{y_{2b}} I_{x2}(y_2) dy_2 \tag{12.9}$$

where

$$I_{x2}(y_2) = \int_{x_{2a}}^{x_{2b}} f(x_2, y_2) dx_2$$

and

$$f(x_2, y_2) = \frac{\cos^2\theta}{|\mathbf{S}|^2} = \frac{(\mathbf{n}\cdot\mathbf{S})^2}{|\mathbf{S}|^4}$$

We shall now use these results in the following example.

Example 12.7 View factor for a differential area and a finite rectangle in parallel planes

To illustrate the numerical integration of Eq. (12.9), we consider the two sets of data given in Table 12.2. In the first case, we shall obtain the view factors shown in the last row of the table. In the second case, we shall obtain a plot of the view factor for data Set 1 as a function of the separation distance of the surfaces for $0.1 \le z_o \le 5$.

TABLE 12.2 View Factors for Two Plate Configurations and View Factor Results

Parameter	Set 1	Set 2
Geometry of A_2		
X coordinate of first corner point, x_{2a}	-1	-1
Y coordinate of first corner point, y_{2a}	-1	-1
X coordinate of opposite corner point, x_{2b}	0	1
Y coordinate of opposite corner point, y_{2b}	0	1
Separation distance between planes, z_0	5	1
Computed view factor F_{d_1-2}	0.0121	0.5541

We create two subfunctions **Fd1_2** and **kernel2**, which are used to determine the kernel $f(x_2, y_2)$ at any location on surface A_2. The function **kernel2** uses a vector-based formulation to determine the length S and $\cos\theta$. It is noted that for compatibility with dblquad, **kernel2** must return a vector whose length equals the length of the input vector x. This allows dblquad to minimize the number of calls to the integrand function while still providing the needed data. This capability is implemented in **kernel2** by using length to determine the number of elements in x. We now use these functions to evaluate the two data sets given in Table 12.2. The program is as follows:

```
function Example12_7
Set1 = Fd1_2(-1, 0, -1, 0, 5)
Set2 = Fd1_2(-1, 1, -1, 1, 1)
N = 100;    dz = linspace(0.1, 5, N);
Fd12 = zeros(N,1);
for i = 1:N
   Fd12(i) = Fd1_2(0, 1, 0, 1, dz(i));
end
plot(dz, Fd12, 'k-')
xlabel('Separation distance of surfaces')
ylabel('View factor')

function F = Fd1_2(x_2a, x_2b, y_2a, y_2b, dz)
F = dblquad(@kernel2, x_2a, x_2b, y_2a, y_2b, [], [], dz)/pi;

function f = kernel2(x, y, dist)
L = length(x);
S = [x; repmat(y, 1, L); dist*ones(1, L)];
n = repmat([0, 0, 1]', 1, L);
f = dot(n, S).^2./dot(S, S).^2;
```

Upon execution, we obtain the results shown in the last row of Table 12.2 and the result shown in Figure 12.18. It is noted that F_{d_1-2} goes to a limiting value of 0.25 as the separation distance between the two parallel planes goes to zero. This is because the point dA_1 is aligned with one of the corners of the square area A_2. Thus, as the two parallel planes approach each other, A_2 cuts off one-quarter of the total hemispherical view from dA_1. In the geometry of Set 2, dA_1 is aligned with the center point of A_2 and the limiting value on F_{d_1-2} as the planes approach each other is 1.0.

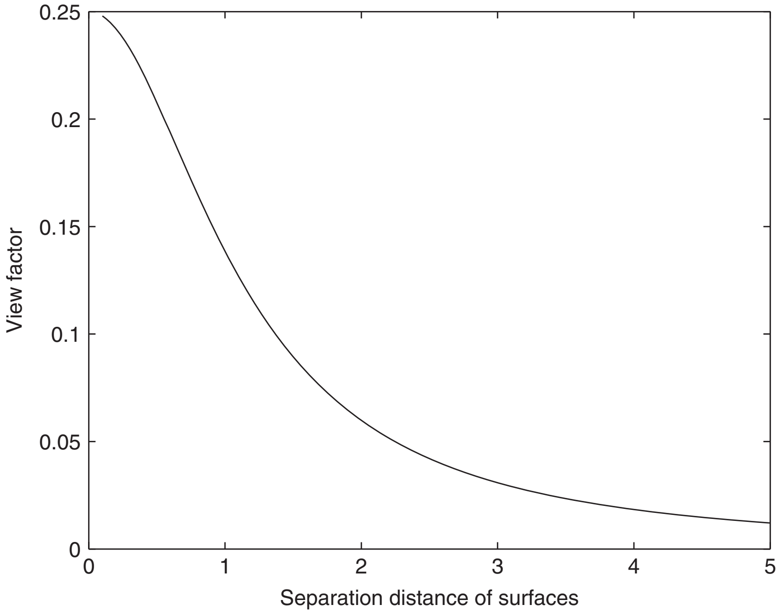


Figure 12.18 View factor versus separation distance between two parallel planes for the geometry shown in Figure 12.17. The areas are in parallel planes and the differential area is aligned with a corner of the finite area.

12.3.2 View Factor Between Two Rectangles in Parallel Planes

The view factor computed in Section 12.3.1 was for an infinitesimal area to a finite area. The infinitesimal area can be integrated over a second finite area to obtain the view factor between two finite areas. The equation defining such a view factor is

$$\begin{aligned} F_{2-1} &= \frac{1}{\pi A_2}\int\limits_{A_1}\int\limits_{A_2}\frac{\cos\theta_1\cos\theta_2}{S^2}\,dA_2dA_1 \\ &= \frac{1}{\pi A_2}\int\limits_{y_{1a}}^{y_{1b}}\int\limits_{x_{1a}}^{x_{1b}}\int\limits_{y_{2a}}^{y_{2b}}\int\limits_{x_{2a}}^{x_{2b}}\frac{\cos\theta_1\cos\theta_2}{S^2}\,dx_2dy_2dx_1dy_1 \end{aligned} \tag{12.10}$$

where the variables are defined in Section 12.3.1 and those variables specific to this quadruple integral are defined in Figure 12.19. There are numerous methods to evaluate this integral. The approach used here is direct integration. Since the two plates are in parallel planes and their edges are parallel, Eq. (12.10) can be written as

$$F_{2-1} = \frac{1}{\pi A_2}\int\limits_{y_{1a}}^{y_{1b}} I_{x1}(y_1)dy_1$$

$$I_{x1}(y_1) = \int\limits_{x_{1a}}^{x_{1b}} I_{y2}(x_1, y_1)dx_1$$

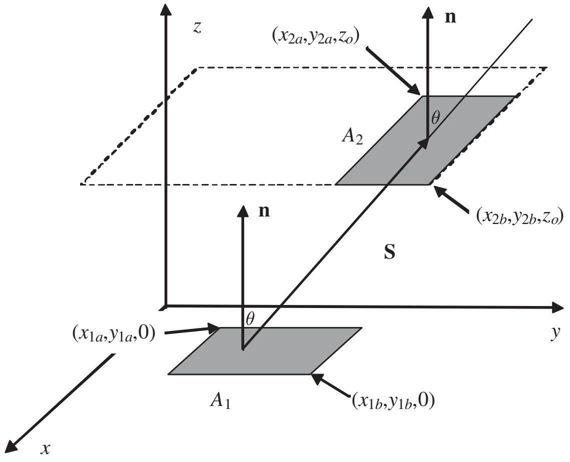


Figure 12.19 Geometry for the determination of the view factors between two finite rectangles in parallel planes.

$$I_{y2}(x_1, y_1) = \int_{y_{2a}}^{y_{2b}} I_{x2}(x_1, y_1, y_2)dy_2$$

$$I_{x2}(x_1, y_1, y_2) = \int_{x_{2a}}^{x_{2b}} f(x_1, y_1, x_2, y_2)dx_2$$

$$f(x_1, y_{1,} x_2, y_2) = \frac{\cos^2\theta}{|\mathbf{S}|^2} = \frac{(\mathbf{n}\cdot\mathbf{S})^2}{|\mathbf{S}|^4}$$

$$\mathbf{S} = (x_1 - x_2)\mathbf{i} + (y_1 - y_2)\mathbf{j} + (z_1 - z_2)\mathbf{k}$$

and $\mathbf{n} = \mathbf{k}$, $z_1 = 0$, and $z_2 = z_0$.

We shall now use these results in the following example.

Example 12.8 View factor between two parallel rectangles

We shall obtain the view factors between two arbitrarily located rectangles in parallel planes for the two sets of data shown in Table 12.3. The method uses two nested calls to dblquad and is implemented in the function **F1_2**, which performs the integration over all the points on A_1. The function **InnerKernel** evaluates $f(x_1, y_1, x_2, y_2)$ and is called by **OuterKernel** to evaluate $I_{y2}(x_1, y_1)$; that is, to integrate f over A_2 for a particular point on A_1.

```
function Example12_8
Set1 = F1_2(-1, 1, -1, 1, -1, 1, -1, 1, 2)
Set2 = F1_2(-2, 0, -2, 0, 2, 0, 2, 0, 2)

function F12 = F1_2(x1a, x1b, y1a, y1b, x2a, x2b, y2a, y2b, dz)
A2 = abs(x1a-x1b)*abs(y1a-y1b);
F12 = dblquad(@OuterKernel, x1a, x1b, y1a, y1b, [], [], x2a, x2b, y2a, y2b, dz)/(A2*pi);
```

TABLE 12.3 Data Used to Compute View Factors and View Factor Results

Parameter	Set 1	Set 2
Geometry of A_1		
x coordinate of first corner point, x_{1a}	−1	−2
y coordinate of first corner point, y_{1a}	−1	−2
x coordinate of opposite corner point, x_{1b}	1	0
y coordinate of opposite corner point, y_{1b}	1	0
Geometry of A_2		
x coordinate of first corner point, x_{2a}	−1	2
y coordinate of first corner point, y_{2a}	−1	2
x coordinate of opposite corner point, x_{2b}	1	0
y coordinate of opposite corner point, y_{2b}	1	0
Separation distance between planes, z_o	2	2
Computed view factor (F_{2-1})	0.1998	0.0433

```
function f = OuterKernel(x1, y1, x2a, x2b, y2a, y2b, dz)
f = zeros(length(x1), 1);
for i = 1:length(x1)
  f(i) = dblquad(@InnerKernel, x2a, x2b, y2a, y2b, [], [], dz, x1(i), y1);
end

function f = InnerKernel(x, y, dz, x2, y2)
L = length(x);
S = [x-x2*ones(1, L); (y-y2)*ones(1, L); dz*ones(1, L)];
n = repmat([0, 0, 1]', 1, L);
f = dot(n, S).^2./dot(S, S).^2;
```

Upon execution, we obtain the results shown in the last row of Table 12.3. The areas of the two rectangles and the spacing between the parallel planes are the same for both data sets. However, the rectangles in Set 1 are directly opposed to each other, whereas the rectangles of those in Set 2 are offset from each other. The result of Set 1 in Table 12.3 was verified by comparison with an available analytical solution for the case of directly opposed rectangles[10] and was found to agree with these results to four significant digits.

12.3.3 Enclosure Radiation with Diffuse Gray Walls

A common task in radiation heat transfer is to determine the temperatures and heat transfer rates due to radiation in an enclosure with diffuse, gray surfaces enclosing a nonparticipating medium. These situations occur in ovens, rooms, and other enclosed spaces. Making the diffuse, gray surface assumption considerably reduces the complexity of the model compared with the general radiation model. The diffuse specification means that the intensity of the radiation leaving and arriving at all surfaces is

[10] *Ibid*, pp. 1030.

independent of direction. The gray specification means that the emissivity and absorptivity are independent of wavelength. However, even with these simplifications, enclosure problems still require considerable effort to set up and solve. Such problems are expressed in matrix notation and, thus, MATLAB provides an ideal environment for their solution. The equations that result from such analyses are[11]

$$\frac{Q_k}{A_k} = q_k = \frac{\varepsilon_k}{1 - \varepsilon_k}\left(\sigma T_k^4 - q_{0,\,k}\right) \tag{12.11}$$

$$\frac{Q_k}{A_k} = q_k = q_{0,\,k} - \sum_{j=1}^{N} F_{k-j} q_{0,\,j} = \sum_{j=1}^{N} F_{k-j}(q_{0,\,k} - q_{0,\,j}) \tag{12.12}$$

where ε_k is the emissivity of surface k, Q_k is the heat transfer rate from surface k, A_k is its area, q_k is its heat flux, $q_{0,\,k}$ is its radiosity, F_{k-j} is the view factor representing the fraction of the energy leaving surface k that is intercepted by surface j, N is the number of surfaces in the enclosure, and $\sigma = 5.67 \times 10^{-8}\,\mathrm{W/(m^2\,K^4)}$ is the Stefan–Boltzmann constant. The formulation of these equations assumes that both the incoming and the outgoing radiation from each of the surfaces is uniform over that surface, and that the intensity is independent of direction. This assumption should be evaluated for a given problem by subdividing the surfaces of the enclosure until the results become independent of the area subdivision scheme.

For a general enclosure problem, one must specify either the heat transfer rate or the temperature of each of the surfaces. Once such a specification has been made, Eqs. (12.11) and (12.12) yield a single independent relation for each surface. For a specified temperature, Eqs. (12.11) and (12.12) are equated to yield

$$q_{0,\,k} - \sum_{j=1}^{N} F_{k-j} q_{0,\,j} = \frac{\varepsilon_k}{1 - \varepsilon_k}(\sigma T_k^4 - q_{0,\,k}) \tag{12.13}$$

When the heat transfer rate is specified, Eq. (12.12) is written as

$$\frac{Q_k}{A_k} = q_{0,\,k} - \sum_{j=1}^{N} F_{k-j} q_{0,\,j} \tag{12.14}$$

These equations can be written in matrix form as

$$\begin{bmatrix} d_1 - F_{1-1} & -F_{1-2} & \cdots & -F_{1-N} \\ -F_{2-1} & d_2 - F_{2-2} & & -F_{1-2} \\ \vdots & & & \vdots \\ -F_{N-1} & -F_{N-2} & & d_N - F_{N-N} \end{bmatrix} \begin{bmatrix} q_{0,\,1} \\ q_{0,\,2} \\ \vdots \\ q_{0,\,N} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix}$$

where, when the temperatures are specified,

$$d_k = 1/(1 - \varepsilon_k)$$

$$b_k = \frac{\varepsilon_k}{1 - \varepsilon_k}\sigma T_k^4$$

[11] *Ibid.*, pp. 189–252.

and when the heat transfer rates are specified,

$$d_k = 1$$

$$b_k = \frac{Q_k}{A_k}$$

Each row of the matrix represents one surface whose form depends on whether the temperature or the heat transfer rate is specified for that surface. In either case, the radiosity associated with the surface is the unknown. Thus, the resulting system of equations consists of N equations in the N unknown radiosities. Once the radiosities are known, the unknown temperature or heat transfer rate for a given surface can be determined from either Eq. (12.11) or Eq. (12.12). Equation (12.11) is somewhat simpler to evaluate except for the special case where the emissivity equals one, in which case Eq. (12.12) is used.

We now illustrate these results with the following example.

Example 12.9 Total heat transfer rate of a rectangular enclosure

Consider an oven that is assumed to be infinitely long into the plane of the page and that has a rectangular cross section. The geometry of the cross section is defined in Figure 12.20 and the corresponding view factors are defined in Table 12.4. These view factors can be computed using Hottel's crossed-string method.[12] The values shown in boldface type are those that were computed by this method and the remaining values are those that were computed from view-factor algebra based on the boldfaced values and their areas.

The following script makes use of an identifier c_k, $k = 1, 2, \ldots, N$, which equals 0 when the temperature is given and equals 1 when the heat transfer rate is given. It is then used to select the appropriate values for d_k and b_k. Furthermore, all vectors must be of length N and the matrix F is $(N \times N)$. The program is as follows:

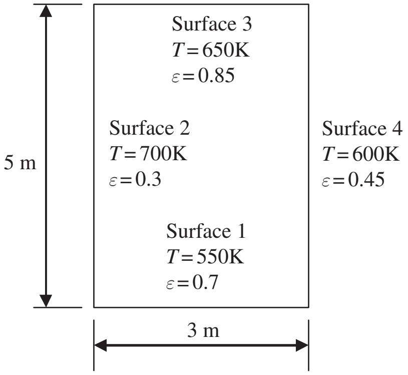


Figure 12.20 Enclosure geometry and surface properties for radiation in an enclosure with gray walls.

[12] *Ibid*.

TABLE 12.4 View Factors F_{i-j} for the Enclosure Shown in Figure 12.20

$i\downarrow \backslash j\rightarrow$	1	2	3	4
1	**0**	**0.3615**	**0.2770**	0.3615
2	0.2169	**0**	**0.2169**	0.5662
3	0.2770	0.3615	**0**	0.3615
4	0.2169	0.5662	0.2169	0

```
sigma = 5.6693e-8;
A = [3, 5, 3, 5];    epsilon = [0.7, 0.3, 0.85, 0.45];
T = [550, 700, 650, 600];
F = -[0, 0.3615, 0.277, 0.3615; . . .
     0.2169, 0, 0.2169, 0.5662; . . .
     0.277, 0.3615, 0, 0.3615; . . .
     0.2169, 0.5662, 0.2169, 0];
Q = zeros(1, length(A));
c = zeros(1, length(A));
b = sigma*epsilon./(1-epsilon).*(1-c).*T.^4+c.*Q./A;
d = (1-c).*1./(1-epsilon)+c;
F = F+diag(d);
q0 = F\b';
Q = A.*epsilon./(1-epsilon).*(1-c).*(sigma*T.^4-q0')
q = Q./A
```

Execution of the script gives $Q = [-8627.9, 8061.1, 4525.9, -3959.1]$ W and $q = [-2876, 1612.2, 1508.6, -791.8]$ W/m^2. It is seen that the heat transfer rates Q correctly sum to zero.

12.3.4 Transient Radiation Heating of a Plate in a Furnace[13]

Consider a vertically suspended flat plate in a furnace. The furnace walls contain heating elements. The furnace and the plate are initially at room temperature. The amount of heating power Q required in the heating elements to raise the plate's temperature to T_e in time t_h can be determined from energy balances on the plate and the furnace walls, which yield the following coupled equations

$$\frac{dT_w}{dt} = P_1 Q - P_2(T_w^4 - T_p^4)$$

$$\frac{dT_p}{dt} = -P_3(T_p^4 - T_w^4)$$

[13] Topic suggested by Professor Yogendra Joshi, Department of Mechanical Engineering, Georgia Institute of Technology, Atlanta, GA.

where the plate and furnace walls are modeled as lumped masses, T_w is the temperature of the furnace walls, and T_p is the temperature of the plate. If we assume that this system can be modeled as a two-surface enclosure with gray diffuse surfaces, then it is found that[14]

$$P_1 = \frac{1}{m_w c_w}$$

$$P_2 = \frac{\sigma}{m_w c_w}\left[\frac{1-\varepsilon_p}{\varepsilon_p A_p} + \frac{1}{A_w F_{wp}} + \frac{1-\varepsilon_w}{\varepsilon_w A_w}\right]^{-1}$$

$$P_3 = \frac{\sigma}{m_p c_p}\left[\frac{1-\varepsilon_p}{\varepsilon_p A_p} + \frac{1}{A_p F_{pw}} + \frac{1-\varepsilon_w}{\varepsilon_w A_w}\right]^{-1}$$

where m_p and m_w are the masses of the plate and wall, respectively; c_p and c_w are the specific heats of the plate and wall, respectively; ε_p and ε_w are the emissivities of the plate and wall, respectively; A_p and A_w are the areas of the plate and wall, respectively; F_{pw} and F_{wp} are the view factors; and $\sigma = 5.67 \times 10^{-8}$ W/(m^{2}K^4) is the Stefan–Boltzmann constant.

Example 12.10 Transient radiation heating of a plate in a furnace

Let us assume that for a certain configuration we have determined that $P_1 = 1.67 \times 10^{-5}$ K/J, $P_2 = 8.8 \times 10^{-14}$ s^{-1}K^{-3}, and $P_3 = 6.3 \times 10^{-13}$ s^{-1}K^{-3}. We shall determine the value of Q required by the heating elements to raise the plate's temperature to $T_e = 1{,}100$K in $t_h = 10$ min (600 s). It is assumed that both the plate and the furnace are initially at 300K; that is, $T_w(0) = T_p(0) = 300$K.

To obtain a solution, we create the following program with two subfunctions: **RadTemp**, which is used by `ode45` to solve the two coupled first-order ordinary differential equations, and **Qgen**, which is used by `fzero` to determine the value of Q. In **QGen**, we have used `interp1` to locate the time at which the temperature $T_e = 1{,}100$K.

```
function Example12_10
P1 = 1.67e-5;   P2 = 8.8e-14;   P3 = 6.3e-13;
Qguess = 1e5;   Te = 1100;   th = 600;
tend = 660;   Two = 300;   Tpo = 300;
Q = fzero(@QGen, Qguess, [], Te, th, Two, Tpo, tend, P1, P2, P3);
[t, T] = ode45(@RadTemp, [0, tend], [Two; Tpo], [], Q, Te, th, Two, Tpo,
   tend, P1, P2, P3);
plot(t, T(:,1), 'k-', t, T(:,2), 'k--')
z = axis;
hold on
plot([0, z(2)], [Te, Te], 'k.:', [th, th], [z(3), z(4)], 'k.:')
xlabel('Time (s)')
text(0.05*z(2), 0.85*z(4), ['Q = ' num2str(Q,6) ' W'])
```

[14] Incropera and DeWitt, *Fundamentals of Heat*, Chapter 13.

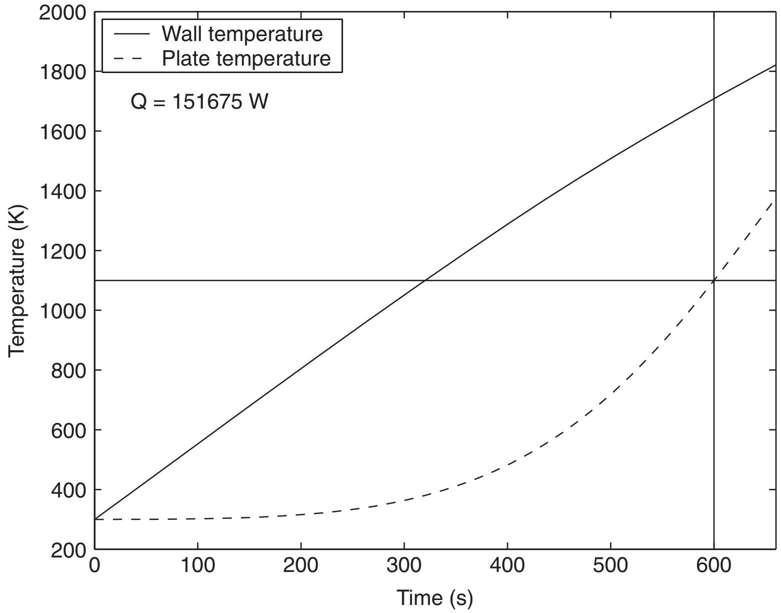


Figure 12.21 Temperature as a function time in the plate and in the furnace.

```
ylabel('Temperature (K)')
legend('Wall temperature', 'Plate temperature', 'Location', 'NorthWest')

function dTdt = RadTemp(t, T, Q, Te, th, Two, Tpo, tend, P1, P2, P3)
dTdt = [P1*Q-P2*(T(1)^4-T(2)^4); -P3*(T(2)^4-T(1)^4)];

function PlateTempDev = QGen(Q, Te, th, Two, Tpo, tend, P1, P2, P3)
[t, T] = ode45(@RadTemp, [0, tend], [Two;Tpo], [], Q, Te, th, Two, Tpo, tend, P1, P2, P3);
PlateTempDev = Te-interp1(t, T(:,2), th, 'spline');
```

When the function is executed, we obtain Figure 12.21.

EXERCISES

Section 12.1.1

12.1 Referring to Section 12.1.1, demonstrate analytically that the expression for $\theta(\eta, \tau)$ satisfies both the governing equation and the boundary condition by using the Symbolic toolbox. *Note*: When using the Symbolic toolbox, erfc must be replaced by $1 - \text{erfc}$.

12.2 From the results of Section 12.1.1, it can be shown that the value of τ when the heat flux at $\eta = 0$ is one-half its initial value is determined from the relation

$$\exp(\tau_{1/2}^2)\text{erfc}(\tau_{1/2}) = 0.5$$

Determine $\tau_{1/2}$.

Section 12.1.2

12.3 A standard plastic milk jug can be represented as a lumped capacitance for the purpose of estimating the time required to heat (or to cool) the milk. The governing equation for such a situation is

$$\frac{dT}{dt} = \frac{Q}{mc_v} = \frac{hA}{mc_v}(T_{\text{amb}}-T) = \frac{1}{\tau_{\text{tc}}}(T_{\text{amb}}-T)$$

where $Q = hA(T_{\text{amb}}-T)$ is the heat transfer to the jug from the surroundings, h is the heat transfer coefficient, A is the surface area, T is the jug temperature, T_{amb} is the ambient temperature, $\tau_{\text{tc}} = mc_v/(hA)$ is the time constant, c_v is the specific heat, and m is the mass of the jug.

For a simple radiation model,

$$Q = A\sigma\varepsilon(T^4 - T_{\text{amb}}^4)$$

where $\sigma = 5.667 \times 10^{-8}\,\text{W/m}^2 \cdot K^4$ is the Stephan–Boltzmann constant and ε is the emissivity. Thus, the governing equation with radiation included is

$$\frac{dT}{dt} = \frac{1}{\tau_{\text{tc}}}\left[(T_{\text{amb}}-T) - \frac{\sigma\varepsilon}{h}(T^4-T_{\text{amb}}^4)\right]$$

Determine the time constant for this system with and without radiation. The time constant τ_{tc} is the time required for the temperature difference between the jug's temperature and the ambient temperature to decrease by 63.2% from its initial value, that is,

$$\frac{T_{\text{amb}} - T(\tau)}{T_{\text{amb}} - T(0)} = 0.368$$

The jug has a mass of $m = 3.5$ kg and specific heat of $c_v = 4.2$ J/g · K. The surface area of the jug is 0.3 m^2 and the initial temperature is $T(0) = 5$°C. The jug interacts with an environment at $T_{\text{amb}} = 30$°C. With no radiation, natural convection occurs from the jug surface, with a heat-transfer coefficient of $h = 2$ W/K · m^2. With combined convection plus radiation, assume that $\varepsilon = 0.5$ (Answers: For the case without radiation $\tau_{\text{tc}} = 6.8$ h; with radiation present $\tau_{\text{tc}} = 2.78$ h).

12.4 The steady-state temperature distribution in a triangular fin shown in Figure 12.22 is determined from

$$(1 - \eta)\frac{d^2\theta}{d\eta^2} - \frac{d\theta}{d\eta} - M^2\theta = 0$$

where $\eta = x/L$,

$$\theta(\eta) = \frac{T(\eta) - T_\infty}{T_b - T_\infty}$$

$$M^2 = \frac{2hL^2}{kt}\sqrt{1 + \left(\frac{t}{2L}\right)^2}$$

h is the heat transfer coefficient, k is the thermal conductivity, and it is assumed that $t/w << 1$.

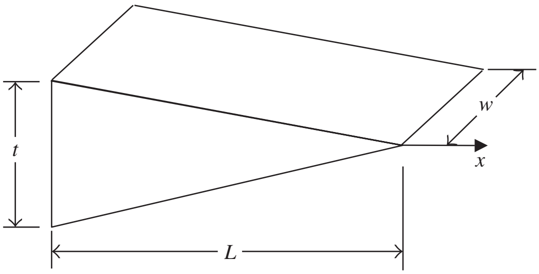


Figure 12.22 Triangular fin geometry.

Assume that the boundary condition at $\eta = 0$ is $T(0) = T_b$, that is,

$$\theta(0) = 1$$

and that at $\eta = 1$, the boundary condition is

$$\left.\frac{d\theta}{d\eta}\right|_{\eta=1} = 0$$

The fin efficiency η_f for this fin is obtained from

$$\eta_f = -\frac{1}{M^2}\left.\frac{d\theta}{d\eta}\right|_{\eta=0}$$

Determine the fin efficiency for ten logarithmically equally spaced values from $0.01 < M^2 < 100$ and plot the results using `semilogx`. Compare these results with those from the analytically obtained expression[15]

$$\eta_f = \frac{1}{M}\frac{I_1(2M)}{I_0(2M)}$$

where $I_n(x)$ in the modified Bessel function of the first kind of order n and is determined by using `besseli`.

12.5 The heat loss by convection from the outer surface of the insulation of an insulated pipe is determined from

$$q = \frac{2\pi L(T_i - T_\infty)}{\dfrac{1}{k}\ln(r_o/r_i) + \dfrac{1}{r_o h}}$$

where L is the pipe length, r_o is the outer radius of the insulation, r_i is the inner radius, k is the thermal conductivity, and h is the heat transfer coefficient. For small values of r_o, additional insulation has the effect of increasing the heat transfer rate. Demonstrate this effect by plotting q as a function of r_o for a range of r_o that spans $r_o = k/h$. Let $h = 5$ W/m$^2\cdot$K, $k = 0.1$ W/m$\cdot$K, $r_i = 0.01$ m, $L = 1$ m, $T_i = 100$°C , and $T_\infty = 20$°C. Although usual experience is that adding insulation reduces the heat transfer rate, this is an exception.

[15] *Ibid.*, p. 125.

12.6 From the results in Section 12.1.2, it can be shown that the value of τ when the surface heat flux is one-half of its initial value is determined from the following expression:

$$0.5 = \sum_{n=1}^{15} C_n \exp(-\zeta_n^2 \tau) J_0(\zeta_n)$$

where we have selected to sum 15 terms. The definitions of C_n and ζ_n are given in Section 12.1.2. Determine the value of τ that satisfies this equation for Bi = 0.5.

12.7 The transient temperature distribution in a solid sphere, which is initially at a uniform temperature and has convection at the boundary surface, is given by[16]

$$\theta(\xi, \tau) = \sum_{n=1}^{\infty} C_n \exp(-\zeta_n^2 \tau) \frac{\sin(\zeta_n \xi)}{\zeta_n \xi}$$

where

$$\theta(\xi, \tau) = \frac{T(\xi, \tau) - T_\infty}{T(\xi, 0) - T_\infty}$$

and $\tau = \alpha t/a^2$, α is the thermal diffusivity, a is the radius of the sphere, t is the time, $\xi = r/a$, r is the radial location in the sphere, T_∞ is the ambient air temperature,

$$C_n = \frac{4[\sin\zeta_n - \zeta_n \cos\zeta_n]}{2\zeta_n - \sin 2\zeta_n}$$

and ζ_n are the positive roots of

$$1 - \zeta_n \cot\zeta_n = \text{Bi}$$

where Bi = ha/k is the Biot number, h is the heat transfer coefficient, and k is the thermal conductivity of the sphere.

Plot $\theta(\xi, \tau)$ as a function of ξ and τ for $0 \le \xi \le 1$, $0 \le \tau \le 0.5$, Bi = 0.5, and $n = 1, 2, \ldots, 20$.

12.8 For the cylinder in Section 12.1.2, compute and plot the temperature throughout the cylinder at $\tau = 0, 0.001$, and 0.005. This exercise highlights the accuracy of this type of infinite series solution. In particular, the truncation of higher order terms causes some error in the solution at $\tau = 0$ but these artifacts are largely gone by $\tau = 0.001$.

12.9 A lumped parameter analysis of the cylinder problem of Section 12.1.2 yields a temperature equation

$$\frac{dT}{dt} + \frac{1}{\tau^*} T = \frac{1}{\tau^*} T_\infty$$

where

$$\tau^* = \frac{R^2}{2\alpha \text{Bi}}$$

If we convert this to the nondimensional constant as given in Section 12.1.2, then

$$\tau_{\text{tc}} = \frac{\tau^* \alpha}{R^2} = \frac{1}{2\text{Bi}}$$

[16] *Ibid.*, p.229.

From the first-order solution, we know that the temperature difference between the initial temperature and T_∞ will be reduced to 36.79% of its original value after one time constant. Using the Bessel function solution in Section 12.1.2, determine the time when the centerline temperature changes to 36.79% of its initial value; that is, when $\theta(0, \tau) = 0.3679$.

Section 12.1.3

12.10 One-dimensional conduction in a plane wall can be represented by

$$-k(T)\frac{dT}{dx} = q$$

where T is the temperature, q is the heat flux, $k(T)$ is the temperature-dependent thermal conductivity, and x is the spatial coordinate. If we assume that the wall is composed of mineral wool insulation, then the thermal conductivity varies as

$$k(T) = -k_0 + k_s T \quad 240\text{K} < T < 365\text{K}$$

where $k_0 = 0.048$, $k_s = 0.00032$, k is in W/m · K, and T is in K. Determine the temperature at $x = 0$ when the heat flux is $q = 12.5$ W/m^2, the wall thickness is 0.1 m, and the temperature of the surface at $L = 0.1$ m is 300K. Compare the results with the analytical solution

$$k_0(T(x) - T(L)) - \frac{k_s}{2}\left(T(x)^2 - T(L)^2\right) = q(x - L)$$

It is noted that the heat flux computed from an average value of the thermal conductivity will give an excellent prediction of the true value.

Section 12.2.1

12.11 A temperature sensor is used to measure the temperature of a flowing fluid. The sensor is mounted on a small-diameter cylindrical probe that protrudes through a duct wall into the flow normal to the direction of flow. The probe can be modeled as a fin whose temperature distribution is given by

$$\frac{T(x) - T_\infty}{T_b - T_\infty} = \frac{\cosh m(L - x)}{\cosh mL}$$

where

$$m^2 = \frac{hP}{kA_c}$$

and T_b is the wall temperature, T_∞ is the fluid temperature, $P = \pi d$ is the perimeter of the probe of diameter d, $A_c = \pi d^2/4$ is the cross-sectional area, k is the thermal conductivity, h is the heat transfer coefficient, and L is the length of the probe.

The error e in the sensed temperature due to conduction along the probe is, therefore,

$$e = T(L) - T_\infty = \frac{T_b - T_\infty}{\cosh mL}$$

Plot the error as a function of probe length for $0.005 \leq L \leq 0.1$ m for several values of k in the range $20 \leq k \leq 400$ W/m · K. Assume that probe diameter is 0.005 m, the fluid temperature is 100°C, and the wall temperature is 80°C. The heat transfer coefficient between the fluid and the probe is 25 W/m^2 · K.

12.12 Apply the analysis of Example 12.4 for a velocity profile that is uniform over the cross section.

Section 12.2.2

12.13 Use the solution method introduced in Section 12.2.2 to determine the thickness of the temperature and velocity boundary layers for $\mathrm{Pr} = 0.07, 0.7$, and 7.0. In the notation of Section 12.2.2, the boundary layer thickness for the temperature δ_T is defined as that value of η for which $y_4 = T^*(\eta = \delta_T) = 0.99$, and the boundary layer thickness for the velocity δ_u is defined as that value of η for which $y_2 = u(\eta = \delta_u) = 0.99$. At each value of the Prandtl number, compare the ratio δ_u/δ_T to that predicted by the relation $\mathrm{Pr}^{1/3}$ (Answers are given in Table 12.5).

12.14 Consider air flowing over a flat plate at $u_\infty = 1$ m/s for which $\mathrm{Pr} = 0.7$ and $\nu_{\mathrm{vis}} = 1.5 \times 10^{-5}$ m^2/s. Use the formulation of Section 12.2.2 to compute $T(x, y)$ over the full laminar domain $0 < x < x_{\mathrm{crit}}$ and $0 < y < 10x_{\mathrm{crit}}/(\mathrm{Re}_{\mathrm{crit}})^{1/2}$ where

$$\mathrm{Re}_{\mathrm{crit}} = \frac{u_\infty x_{\mathrm{crit}}}{\nu_{\mathrm{vis}}} = 5 \times 10^{-5}$$

Create a contour plot of $T(x, y)$ over this domain. In the notation of Section 12.2.2, $T(x, y) = y_4$ and

$$\eta = y\sqrt{u_\infty/(\nu_{\mathrm{vis}}x)}$$

12.15 For the same conditions defined in Exercise 12.14, create a contour plot of the stream function

$$\psi = f\sqrt{\nu_{\mathrm{vis}}xu_\infty}$$

Section 12.2.3

12.16 For the natural convection solution of Section 12.2.3, find the value of η at which the velocity u is a maximum for $\mathrm{Pr} = 0.07, 0.7$, and 7.0. In the notation of Section 12.2.3, $u = y_2$. Use `fminbnd` and `spline` on the negative of u and select the search region based on the curves in Figures 12.14–12.16 (Answers are given in Table 12.6).

12.17 For the natural convection solution of Section 12.2.3, determine the thickness of the thermal and velocity boundary layers for $\mathrm{Pr} = 0.07, 0.7$, and 7.0. In the notation of

TABLE 12.5 Answers to Exercise 12.13

Pr	δ_u	δ_T	δ_u/δ_T	$\mathrm{Pr}^{1/3}$
0.07	4.92	13.66	0.36	0.41
0.7	4.92	5.63	0.87	0.89
7.0	4.92	2.45	2.01	1.91

TABLE 12.6 Answers to Exercises 12.16 and 12.17

Pr	0.07	0.7	7.0
$\eta_{\max}$	1.231	0.965	0.728
$u_{\max}$	0.455	0.278	0.131
δ_u	9.48	5.65	6.41
δ_T	10.27	4.47	1.78

Section 12.2.3, the boundary layer thickness for the temperature δ_T is defined as that value of η for which $y_4 = T^*(\eta = \delta_T) = 0.01$, and the boundary layer thickness for the velocity δ_u is defined as that value of η for which $y_2 = u(\eta = \delta_u)/u_{\max} = 0.01$, where $u_{\max}$ are those values found in Exercise 12.16 (Answers are given in Table 12.6).

12.18 Consider the natural convection of air over a heated plate, as described in Section 12.2.3. The dimensional velocities can be expressed in terms of the nondimensional solution as

$$u = \frac{2\nu_{\text{vis}}}{x}\mathrm{Gr}_x^{1/2}\frac{df}{d\eta}$$

$$v = \frac{\nu_{\text{vis}}}{x}\left(\frac{\mathrm{Gr}_x}{4}\right)^{1/4}\left(\eta\frac{df}{d\eta} - 3f\right)$$

For quiescent air at 300K, compute the velocity components $u(x, y)$ and $v(x, y)$ over the domain $0 < x < 1$ m and $0 < y < 0.25$ m. Plot u and v at five evenly spaced x locations from $x = 0.1$ to $x = 0.5$. Refer to Figure 12.13 for the definition of x and y.

12.19 The steam function was defined in Section 12.2.3 as

$$\psi = 4\nu f\left(\frac{\mathrm{Gr}_x}{4}\right)^{1/4}$$

For air at 300K, create a contour plot of the stream function over the domain $0 < x < 1$ m and $0 < y < 0.25$ m. Refer to Figure 12.13 for the definition of x and y.

Section 12.3.1

12.20 Based on the analysis given in Section 12.3.1, alter the formulation for the view factors to evaluate the case where the rectangles are in two perpendicular planes. As shown in Figure 12.23, the differential area element is located at a distance from a perpendicular plane in which a finite rectangular area exists. For simplicity, consider only the case where the differential element can see the entire finite rectangle. In other words, the line of intersection of the perpendicular planes cannot pass through the finite rectangle. Using the data given in Table 12.7, determine the view factor for the surfaces shown in Figure 12.23. Note that in this case $\cos\theta_1 \neq \cos\theta_2$.

Section 12.3.3

12.21 Because of the nonlinear nature of radiation heat transfer, the placement of shields in the radiation path reduces heat transfer. Thus, for insulation applications, the use of

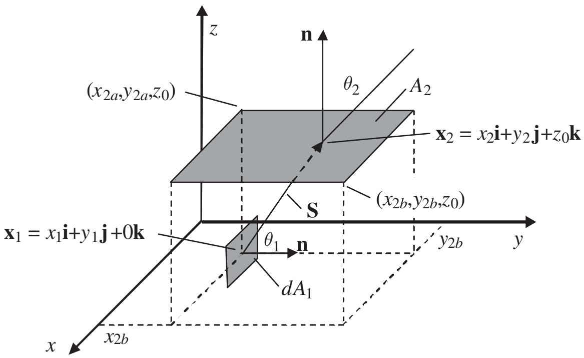


Figure 12.23 Geometry when the differential area and the finite rectangle are in perpendicular planes.

radiation shields is important. The effect of radiation shields can be illustrated by considering the heat transfer between two infinite parallel plates at temperatures T_1 and T_2 that are separated by an evacuated space. When the surfaces radiate as black bodies, the heat transfer rate q with no shield is obtained from

$$q = \sigma(T_1^4 - T_2^4)$$

where $\sigma = 5.667 \times 10^{-8}\,\mathrm{W/m^2 \cdot K^4}$ is the Stephan–Boltzmann constant. When we have one shield, the heat transfer rate is determined from

$$q = \sigma(T_1^4 - T_m^4)$$
$$q = \sigma(T_m^4 - T_2^4)$$

where T_m is the temperature of the shield. The heat transfer rate with two shields is obtained from

$$q = \sigma(T_1^4 - T_{m1}^4)$$
$$q = \sigma(T_{m1}^4 - T_{m2}^4)$$
$$q = \sigma(T_{m2}^4 - T_2^4)$$

TABLE 12.7 Parameters and Answer for Exercise 12.20

Parameter	Value
Geometry of finite rectangle	
x coordinate of first corner point, x_{1a}	−1
y coordinate of first corner point, y_{1a}	−1
x coordinate of opposite corner point, x_{1b}	0
y coordinate of opposite corner point, y_{1b}	0
Separation distance between the differential element and the perpendicular plane	5
Computed view factor (F_{d_1-2})	0.0012

where T_{m1} and T_{m2} are the temperatures of shields 1 and 2, respectively. If the temperatures of the two plates are 100°C and 20°C, then determine the heat transfer rate with no shield, one shield, and two shields (Answers: With no shield, $q = 680\ \text{W/m}^2$; with one shield, $q = 340.1\ \text{W/m}^2$ and $T_m = 340.15$ K; and with two shields, $q = 226.7\ \text{W/m}^2$, $T_{m1} = 352.2$ K, and $T_{m2} = 326.7$ K).

12.22 The Planck distribution represents the power spectral density of black-body radiation at a particular temperature and is given by

$$E_{\lambda,b}(\lambda, T) = \frac{C_1}{\lambda^5[\exp(C_2/\lambda T) - 1]}$$

where λ is the wavelength, T is the temperature, $E_{\lambda,b}$ is the spectral emissive power, $C_1 = 3.742 \times 10^8\ \text{W}\cdot\mu\text{m}^4/\text{m}^2$, and $C_2 = 1.439 \times 10^4\ \mu\text{m}\cdot\text{K}$. A common need in radiation calculations is to integrate this function over some range of wavelengths. When integrated over all wavelengths, we have that

$$\int_0^\infty E_{\lambda,b}d\lambda = \sigma T^4$$

where $\sigma = 5.667 \times 10^{-8}\ \text{W/m}^2\cdot\text{K}^4$ is the Stephan–Boltzmann constant. Perform this integration numerically for $T = 300$K, 400K, and 500K and compare the result with the exact value. A note of caution: both integration limits give considerable difficulty numerically. Approximate the integral by using a nonzero lower limit, such as 0.5 μm. For the upper limit, use a value of 300 μm.

12.23 Consider Example 12.9 with surface 4 split into two equal size surfaces—that is, the system will now have five surfaces. Assume that the surface properties are the same as those used in the original example. Calculate the heat transfer rate from each of the surfaces and compare it with the more coarse calculation done originally. Using the surface numbering scheme given in Section 12.3.3, surface 4 is now split into two, and surface 4 refers to the upper half and surface 5 to the lower half. The view factors for this geometry were calculated and are given in Table 12.8. These values were obtained from Hottel's crossed-string method. The results of the enclosure calculation are summarized in Table 12.9.

It is noted that the energy balance is satisfied—that is, the Q's sum to zero. Comparing these results with those in Example 12.9, it is seen that the heat transfer rates Q and the fluxes q are similar between the two calculations for surfaces 1, 2, and 3. However, for the surface that was split, it is seen that the flux varies considerably over the length. It is interesting to note that the overall heat transfer rate (the sum for surfaces 4 and 5) matches reasonably closely with the originally calculated heat transfer rate.

TABLE 12.8 View Factors for Exercise 12.23

$i\downarrow\backslash j\rightarrow$	1	2	3	4	5
1	**0**	**0.3615**	**0.2770**	**0.0957**	0.2658
2	0.2169	**0**	**0.2169**	**0.2831**	0.2831
3	0.2770	0.3615	**0**	**0.2658**	0.0957
4	0.1148	0.5662	0.3190	**0**	0
5	0.3190	0.5662	0.1148	0	0

TABLE 12.9 Answers to Exercise 12.23

Surface #	1	2	3	4	5
Q (W)	−8560	8064	4451	−2373	−1582
T (K)	550	700	650	600	600
q (W/m^2)	−2853	1613	1484	−949	−633

BIBLIOGRAPHY

J. P. Holman, *Heat Transfer*, 7th ed., McGraw Hill, New York, 1990.

F. P. Incropera and D. P. DeWitt, *Fundamentals of Heat and Mass Transfer*, 4th ed., John Wiley & Sons, New York, 1996.

S. Kakaç and H. Liu, *Heat Exchangers: Selection, Rating and Thermal Design*, CRC Press, Boca Raton, FL, 1998.

W. M. Kays and M. E. Crawford, *Convective Heat and Mass Transfer*, 2nd ed., McGraw Hill, New York, 1980.

F. Krieth and M. S. Bohn, *Principles of Heat Transfer*, 5th ed., West Publishing Company, New York, 1993.

A. F. Mills, *Heat Transfer*, Irwin, Boston, 1992.

R. Siegel and J. R. Howell, *Thermal Radiation Heat Transfer*, 3rd ed., Hemisphere Publishing Company, Washington, DC, 1992.

N. V. Suryanarayana, *Engineering Heat Transfer*, West Publishing Company, New York, 1995.

13

Optimization

Shapour Azarm

The solutions to a wide range of engineering optimization applications are illustrated using the Optimization toolbox and Genetic Algorithm and Direct Search toolbox.

13.1 DEFINITION, FORMULATION, AND GRAPHICAL SOLUTION

13.1.1 Introduction

Optimization in engineering refers to the process of finding the "best" possible values for a set of variables for a system while satisfying various constraints. The term "best" indicates that there are one or more design objectives that the decision maker wishes to optimize by either minimizing or maximizing them. For example, one might want to design a product by maximizing its reliability while minimizing its weight and cost. In an optimization process, variables are selected to describe the system such as size, shape, material type, and operational characteristics. An objective refers to a quantity that the decision maker wants to be made as high (a maximum) or as low (a minimum) as possible. A constraint refers to a quantity that indicates a restriction or limitation on an aspect of the system's capabilities.

Generally speaking, an optimization problem involves minimizing one or more objective functions subject to some constraints, and is stated as

$$\underset{x\in D}{\text{minimize}}\ \{f_1(x), f_2(x), \ldots, f_M(x)\} \tag{13.1}$$

where f_i, $i = 1, \ldots, M$, are each a scalar objective function that maps a vector x into the objective space. The n-dimensional design (or decision) variable vector x is constrained to lie in a region D, called the feasible domain. Constraints to the above problem are included in the specification of the feasible domain. In general, the feasible domain is constrained by J-inequality and/or K-equality constraints as

$$D = \{x : g_j(x) \le 0, h_k(x) = 0, \qquad j = 1, \ldots, J, \qquad k = 1, \ldots, K\} \tag{13.2}$$

An optimization problem in which the objective and constraint functions are linear functions of their variables is referred to as a linear programming problem. On the other hand, if at least one of the objective or constraint functions is nonlinear, then it is referred to as a nonlinear programming problem.

The classes of optimization problems, the MATLAB solution functions, and the examples appearing in this chapter and several other chapters are summarized in Table 13.1.

13.1.2 Graphical Solution

Solutions to optimization problems with two variables can be visualized with MATLAB's plotting capabilities. This is demonstrated with the following example.

Example 13.1 Equilibrium position of a two-spring system

Consider a two-spring system[1] shown in Figure 13.1. The system in Figure 13.1 shows a spring system before and after loads are applied at joint A. We shall determine the equilibrium state of the loaded system—that is, the location (x_1, x_2) of joint A. The equilibrium state of the system is obtained by obtaining the potential energy (PE) for the system and

[1] G. Vanderplaats, *Numerical Optimization Techniques for Engineering Design*, McGraw-Hill, New York, 1984, pp. 72–73.

TABLE 13.1 Classification of Optimization Problems, MATLAB Functions, and Examples

Problem class	MATLAB function	Example
Linear programming	`linprog`	13.2 Production planning
		13.3 Oil refinery profits
Binary integer programming	`bintprog`	13.4 Loading of a knapsack
Nonlinear programming		
Single-objective unconstrained:		
Multivariable	`fminunc`	13.5 Equilibrium position of a two-spring system
	`fminsearch`	13.6 Bottom of a bottle
		9.16 Vibration absorber parameters
Curve fitting	`lsqcurvefit`	13.7 Stress–strain relationship
		9.8 Estimation of system parameters
Least squares	`lsqnonlin`	13.8 Stress–strain relationship
		13.9 Semiempirical $P-v-T$ relationship
		13.10 Mineral exploration
Single-objective constrained:		
Single variable	`fminbnd`	13.11 Piping cost in a plant
		13.12 Closed box
		5.22 Response of an oscillatory system
Multivariable	`fmincon`	13.13 Two-bar truss
		13.14 Helical compression spring
		13.15 Gear reducer
Quadratic	`quadprog`	13.16 Production planning
Semi-infinite	`fseminf`	13.17 Planar two-link manipulator
Multiobjective:	`fminimax`	13.18 Vibrating platform
		9.16 Vibration absorber parameters
	`fgoalattain`	13.19 Production planning
Genetic Algorithm		
Single objective	`ga`	13.20 Loading of a knapsack revisited
		13.21 Two bar truss revisited: single objective
Multiobjective	`gamultiobj`	13.22 Two bar truss revisited: multiple objective
		13.23 Two bar truss revisited: single objective with continuous and discrete variables

then minimizing it with respect to the design variables x_1 and x_2 to obtain the new location of joint A. The potential energy is computed from the difference between the strain energies of the springs, which is given by the first two terms in Eq. (13.3), and the work done by external forces, which is given by the last two terms in Eq. (13.3). The quantities $k_1, k_2, L_1, L_2, F_1,$ and F_2 are constants whose values are shown in Figure 13.1. Hence, the objective function of the unconstrained optimization problem is

$$\underset{x_1,\, x_2}{\text{minimize}} \quad \text{PE}(x_1, x_2) = 0.5k_1\left(\sqrt{x_1^2 + (L_1 - x_2)^2} - L_1\right)^2 + 0.5k_2\left(\sqrt{x_1^2 + (L_2 + x_2)^2} - L_2\right)^2 - F_1x_1 - F_2x_2 \quad (13.3)$$

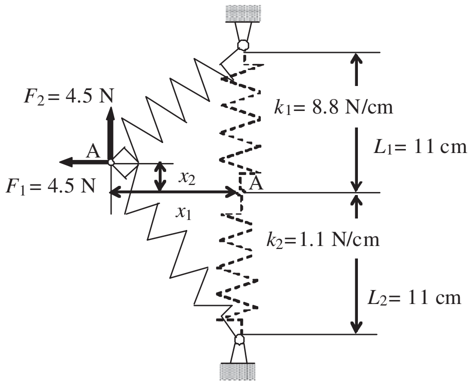


Figure 13.1 Two-spring system.

The two variables x_1 and x_2 in the objective function can have their values estimated graphically. The program that generates these graphical displays is as follows:

```
k1 = 8.8;    k2 = 1.1;    L1 = 11;
L2 = 11;    F1 = 4.5;    F2 = 4.5;
[x1, x2] = meshgrid(linspace(-5, 15, 20), linspace(-5, 15, 20));
PE1 = 1/2*k1*(sqrt(x1.^2+(L1-x2).^2)-L1).^2;
PE2 = 1/2*k2*(sqrt(x1.^2+(L2+x2).^2)-L2).^2;
PE = PE1+PE2-F1*x1-F2*x2;
subplot(1,2,1);
h = contour(x1, x2, PE, [-40:20:20, 50:90:500], 'k');
clabel(h);
axis([-5, 15, -5, 15])
xlabel('x_1');
ylabel('x_2');
subplot(1, 2, 2);
surfc(x1, x2, PE);
axis([-10, 15, -10, 15, -100, 500]);
zlabel('PE');
xlabel('x_1');
ylabel('x_2')
```

The execution of the above script produces the results shown in Figure 13.2, with Figure 13.2a showing a contour plot of PE. The contours are labeled with their numerical values so that the approximate location of the minimum/maximum point can be visually located. Figure 13.2b shows the surface plot of PE with the contours shown below the surface. It also shows the approximate location of the minimum or maximum point, that is, the point where the potential energy function reaches its minimum or maximum values. The more accurate location of the minimum or maximum point for this example is obtained in Example 13.5 of Section 13.4.1 by using an unconstrained minimization technique as implemented in fminunc.

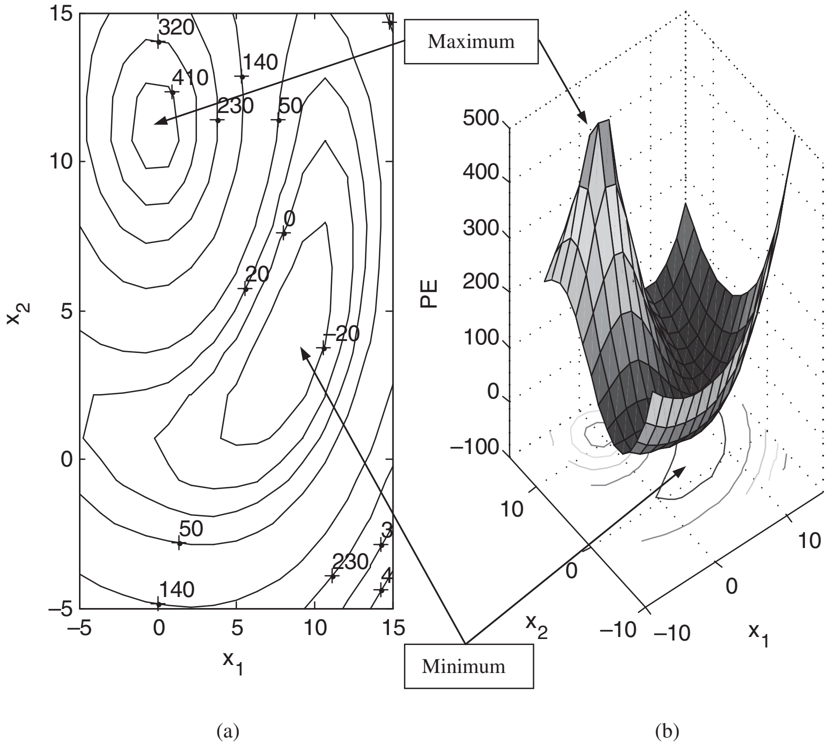


Figure 13.2 (a) Contour and (b) surface plots of the PE function for the two-spring system shown in Figure 13.1.

13.2 LINEAR PROGRAMMING

Linear programming refers to an optimization method applicable to the solution of problems in which the objective and constraint functions are linear functions of the design variables. A linear programming problem can be stated as

$$\begin{aligned} \text{minimize} \quad & f^T x \\ \text{subject to:} \quad & Ax \leq b \\ & A_{eq}x = b_{eq} \\ & lb \leq x \leq ub \end{aligned} \tag{13.4}$$

where f, b, b_{eq}, lb, and ub are vectors and A and A_{eq} are matrices. The quantity x is a vector of design variables and the superscript T indicates the transpose. The matrix A and the vector b are the coefficients of the linear inequality constraints, and A_{eq} and b_{eq} are the coefficients of the equality constraints. The MATLAB linear programming solver is `linprog`, which is used to solve Eq. (13.4). The basic command is

```
[xopt, fopt] = linprog (f, A, b, Aeq, beq, lb, ub, x0, options)
```

which returns the vector *xopt* = $[x_{1\text{opt}}, x_{2\text{opt}}, \ldots]$ of the design variables and the scalar *fopt*, which is *f*(*xopt*). The quantity x_0 sets the starting points (initial guess) of *x*, and *options* sets the parameters described in `optimset`. In the basic command, if *lb*, *ub*, and *options* are not specified, one uses []; similarly for the case when *A*, *b*, *Aeq*, and *beq* are not specified.

We now demonstrate the use of `linprog` with the following example.

Example 13.2 Production planning[2]

Consider two liquid products A and B that require production time in two departments. Product A requires 1 h in the first department and 1.25 h in the second department. Product B requires 1 h in the first department and 0.75 h in the second department. The available number of hours in each department is 200. Furthermore, there is a maximum market potential of 150 units for product B. Assume that the profits are $4 and $5 per unit of products A and B, respectively. We shall determine the number of units of products A and B that should be produced so that the producer's profit is maximized.

We denote x_1 to represent the number of units of product A to be produced and x_2 to represent the number of units of product B. Then the objective function and the constraints are

$$\text{minimize } f(x_1, x_2) = -4x_1 - 5x_2 \tag{13.5}$$

$$\begin{aligned} \text{subject to:}\quad & x_1 + x_2 \le 200 \\ & 1.25\,x_1 + 0.75\,x_2 \le 200 \\ & x_2 \le 150 \\ & (x_1, x_2) \ge 0 \end{aligned}$$

Thus,

$$f^T = [-4, -5] \tag{13.6}$$

and the inequality constraints are expressed as

$$\begin{bmatrix} 1 & 1 \\ 1.25 & 0.75 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \le \begin{bmatrix} 200 \\ 200 \\ 150 \end{bmatrix} \tag{13.7}$$

The program that solves Eqs. (13.5)–(13.7) is as follows:

```
f = [-4, -5];    A = [1, 1; 1.25, 0.75; 0, 1];
x = linprog(f, A, [200, 200, 150], [], [],[0, 0]);
disp(['No. of units A = ' num2str(x(1)) ' No. of units B = ' num2str(x(2))])
```

Upon execution, we obtain the following:

```
No. of units A = 50    No. of units B = 150
```

[2] A. Osyczka, *Multicriterion Optimization in Engineering with Fortran Programs*, Ellis Horwood Limited, West Sussex, UK, 1984, p. 4.

Example 13.3 Oil refinery profits

A refinery has three types of crude oil, C_1, C_2, and C_3. Crude oil C_1 costs \$0.40/L and there are at most 10,000 L/day of it available. Crude oil C_2 costs \$0.20/L and there are at most 12,000 L/day of it available. Crude oil C_3 costs \$0.10/L and there are at most 15,000 L/day of it available. The refinery can convert each type of crude oil to gasoline and can produce three types of gasoline: regular, plus, and premium. The maximum market demand for the regular, plus, and premium gasoline is 9,000 L/day, 8,000 L/day, and 7,000 L/day, respectively. The refinery can sell its gasoline to a distributor for \$0.70/L for regular, \$0.80/L for plus, and \$0.9/L for premium gasoline. It is found that (1) 1 L of crude oil C_1 yields 0.2 L of regular, 0.3 L of plus, and 0.5 L of premium gasoline, (2) 1 L of crude oil C_2 yields 0.5 L of regular, 0.3 L of plus gasoline, and 0.2 L of premium gasoline, and (3) 1 L of crude oil C_3 yields 0.7 L of regular, 0.3 L of plus, and no premium gasoline. We shall determine the number of liters of each of the crude oils C_1, C_2, and C_3 that the refinery should purchase to maximize its daily profit.

Let x_1, x_2, and x_3 represent the number of liters for crude oils C_1, C_2, and C_3, respectively, that is to be purchased. Then, the objective function f and constraints on gasoline demands and crude oil availability are

$$
\begin{aligned}
\text{minimize } f(x_1, x_2, x_3) &= -[0.7(0.2\,x_1 + 0.5\,x_2 + 0.7\,x_3) + 0.8(0.3\,x_1 + 0.3\,x_2 + 0.3\,x_3) \\
&\qquad +0.9(0.5x_1 + 0.2x_2) - 0.4x_1 - 0.2x_2 - 0.1x_3] \\
&= -0.43x_1 - 0.57x_2 - 0.63x_3 \\
\text{subject to: } & 0.2x_1 + 0.5x_2 + 0.7x_3 \le 9{,}000 \\
& 0.3x_1 + 0.3x_2 + 0.3x_3 \le 8{,}000 \qquad (13.8) \\
& 0.5x_1 + 0.2x_2 \le 7{,}000 \\
& x_1 \le 10{,}000 \\
& x_2 \le 12{,}000 \\
& x_3 \le 15{,}000 \\
& (x_1, x_2, x_3) \ge 0
\end{aligned}
$$

Notice that we have used the negative of the profit, since we are minimizing a positive function, which is the same as maximizing profit. This process of considering the negative of an objective function is used in several of the subsequent examples.

The program to determine the maximum daily profit is as follows:

```
f = [-0.43, -0.57, -0.62];
A = [0.2, 0.5, 0.7; 0.3, 0.3, 0.3; 0.5, 0.2, 0; 1, 0, 0; 0, 1, 0; 0, 0, 1];
b = [9000, 8000, 7000, 10000, 12000, 15000];
lb = [0, 0, 0];
[x, f] = linprog(f, A, b, [], [], lb, []);
disp(['C1 = ' num2str(x(1)) ' liters/day'])
disp(['C2 = ' num2str(x(2)) ' liters/day'])
disp(['C3 = ' num2str(x(3)) ' liters/day'])
disp(['Profit/day = $' num2str(-f,'%7.2f')])
```

Upon execution, we obtain

```
C1 = 9200 liters/day
C2 = 12000 liters/day
C3 = 1657.1429 liters/day
Profit/day = $11823.43
```

13.3 BINARY INTEGER PROGRAMMING

Binary integer programming refers to a linear programming method in which the variables are binary; that is, they can only take the value of 0 or 1. The formulation for a binary integer programming problem is similar to a linear programming problem given by Eq. (13.4), except that the bounds on the variables are replaced with the requirement that the variables are binary. The MATLAB binary integer programming solver is `bintprog`, whose basic command is

[xopt, fopt] = `bintprog`(f, A, b, Aeq, beq, x0, options)

where the arguments of `bintprog` have the same meaning as those in Eq. (13.4).

We now demonstrate the use of `linprog` with the following example.

Example 13.4 Loading of a knapsack[3]

The weight capacity of a knapsack container is limited to 35 kg. There are seven different objects that can be placed in the knapsack. Each object has a weight and a monetary value as given in Table 13.2. We shall determine the combination of objects that should be placed in the knapsack so as to maximize the total value of the placed objects subject to the weight limit. We assume that only one of each object can be placed in the knapsack.

We select x_1 to x_7 as the binary variables that determine the objects such that x_j, $j = 1, 2, \ldots, 7$, when the object is placed in the knapsack and $x_j = 0$ when it is

TABLE 13.2 Weight and Value for Knapsack Objects

Object number	Weight (kg)	Value ($)
1	3	12
2	4	12
3	3	9
4	3	15
5	15	90
6	13	26
7	16	112

[3] K. Murty, *Linear and Combinatorial Programming*, John Wiley & Sons, New York, 1976, p. 446.

not. The objective function f, which is the negative of the total value of the objects placed in the knapsack, and the capacity constraint are

$$\text{minimize } f(x) = -(12x_1 + 12x_2 + 9x_3 + 15x_4 + 90x_5 + 26x_6 + 112x_7) \qquad (13.9)$$

$$\text{subject to:} \quad 3x_1 + 4x_2 + 3x_3 + 3x_4 + 15x_5 + 13x_6 + 16x_7 \le 35$$

$$x_i = 0 \text{ or } 1 \qquad i = 1, \dots, 7$$

The program to evaluate Eq. (13.9) is as follows:

```
f = [-12, -12, -9, -15, -90, -26, -112];
A = [3, 4, 3, 3, 15, 13, 16];
b = 35;
[x, V] = bintprog(f, A, b);
disp([repmat('x(', 7,1) num2str((1:7)') repmat(') = ', 7,1) int2str(x)])
disp(['Total value of placed objects = $' num2str(-V, '%6.2f')])
```

Upon execution, we obtain

```
x(1) = 0
x(2) = 0
x(3) = 0
x(4) = 1
x(5) = 1
x(6) = 0
x(7) = 1
Total value of placed objects = $217.00
```

13.4 NONLINEAR PROGRAMMING: UNCONSTRAINED AND CURVE FITTING

Nonlinear programming refers to an optimization method in which the objective and/or constraint function is a nonlinear function of the design variables. Nonlinear programming problems and the corresponding methods are divided into two classes: the unconstrained methods and the constrained methods. We shall now discuss each of these methods.

13.4.1 Unconstrained Optimization

Unconstrained nonlinear programming methods find the minimum of an unconstrained multivariable function formulated as

$$\underset{x}{\text{minimize}} \quad f(x) \qquad (13.10)$$

where x is a vector of design variables and f is a scalar objective function. There are two functions that can be used to solve Eq. (13.10): `fminunc` and `fminsearch`, which are based on derivative and nonderivative optimization solution techniques, respectively. The command to invoke `fminunc` is

```
[xopt, fopt] = fminunc(@UserFunction, x0, options, p1, p2, . . . )
```

where **UserFunction**[4] is a function that computes the objective function $f(x)$. The quantity $x0$ is the vector of starting values for x and *options* sets the parameters described in `optimset`. The quantities $p1, p2, \ldots$ are parameters that are passed to **UserFunction**.

The command to invoke `fminsearch` is

```
x = fminsearch(@UserFunction, x0, options, p1, p2, . . . )
```

where the definitions of its arguments are the same as for `fminunc`.

In both `fminunc` and `fminsearch`, the default solution method is a large-scale optimization algorithm. When a medium-scale optimization algorithm is desired, one uses `optimset` with 'LargeScale' set to 'off'; that is,

```
options = optimset('LargeScale', 'off');
```

We now illustrate the use of `fminunc` and `fminsearch`.

Example 13.5 Equilibrium position of a two-spring system revisited

The two-spring system of Example 13.1 is now solved numerically as an unconstrained optimization problem using both `fminunc` and `fminsearch` to determine the location of the minimum and maximum values. To obtain the maximum value of the PE function, one can either minimize the inverse 1/PE or the negative PE. The unconstrained objective function is created with the design variables x_1 and x_2 in the subfunction **SpringEquilibrium**. The program is as follows:

```
function Example13_5
x0 = [0.5, 5];    k1 = 8.8;    k2 = 1.1;    L1 = 11;
L2 = 11;    F1 = 4.5;    F2 = 4.5;
options = optimset('LargeScale', 'off', 'display', 'off');
H = char('Minimum', 'Maximum');
for neg = 0:1
   [xc, fc] = fminunc(@SpringEquilibrium, x0, options, k1, k2, L1, L2, F1, F2, neg);
   [xs, fs] = fminsearch(@SpringEquilibrium, x0, options, k1, k2, L1, L2, F1, F2, neg);
   disp(H(neg+1,:))
   disp(['From fminunc x1 = ' num2str(xc(1)) ' x2 = ' num2str(xc(2)) . . .
        ' PE = ' num2str(fc*(-1)^neg)])
   disp(['From fminsearch x1 = ' num2str(xs(1)) ' x2 = ' num2str(xs(2)) . . .
        ' PE = ' num2str(fs*(-1)^neg)])
end

function PE = SpringEquilibrium(x, k1 ,k2, L1, L2, F1, F2, neg)
PE1 = 1/2*k1*(sqrt(x(1)^2+(L1-x(2))^2)-L1)^2;
PE2 = 1/2*k2*(sqrt(x(1)^2+(L2+x(2))^2)-L2)^2;
PE = PE1+PE2-F1*x(1)-F2*x(2);
if neg == 1
   PE = -PE;
end
```

[4] When **UserFunction** is created either as an anonymous function or with `inline`, the '@' symbol is omitted.

Upon execution, we obtain

```
Minimum
From fminunc x1 = 8.4251    x2 = 3.6331    PE = -35.0507
From fminsearch x1 = 8.4251    x2 = 3.6331    PE = -35.0507
Maximum
From fminunc x1 = 2.4185e-006    x2 = 11    PE = 549.4498
From fminsearch x1 = 1.6304e-006    x2 = 11    PE = 549.4498
```

We see that these values agree with those that can be estimated from Figure 13.2.

It is pointed out that both `fminunc` and `fminsearch`, and all other techniques in the Optimization toolbox except `linprog` and `bintprog`, obtain only a local optimum solution and depend upon the choice of the initial point x0. In some cases, depending on the problem, if the location of the initial point is changed, the location of the solution might also change. We illustrate this occurrence in the following example.

Example 13.6 Bottom of a bottle[5]

Consider the following two-variable function:

$$f(x_1, x_2) = 0.125\big(0.5(D + 3) + \sin(D + 2) + 1\big)$$

where

$$D = (6x_1 - 3)^2 + (6x_2 - 3)^2$$

The optimization statement for this example is

$$\begin{gathered}\text{minimize} \quad f(x_1, x_2) \\ 0 \le (x_1, x_2) \le 1\end{gathered} \tag{13.11}$$

We create the following function M file so that we can use it to plot the surface and to visualize the locations of the local maxima:

```
function y = Bottle(x)
[r, c] = size(x);
x1 = x(1:r,1:c/2);
x2 = x(1:r,(c/2+1):c);
D = (6*x1-3).^2+(6*x2-3).^2;
y = -0.125*((D+3)/2+sin(D+2)+1);
```

We now use this function in the following script to create Figure 13.3:

```
[x1, x2] = meshgrid(linspace(0, 1, 30), linspace(0, 1, 30));
mesh(x1, x2, -Bottle([x1, x2]));
zlabel('f')
xlabel('x_1')
ylabel('x_2');
view(-10, 45)
```

[5] Based on D. A. Van Veldhuizen and G. B. Lamont, "Multi-Objective Evolutionary Algorithm Research: A History and Analysis," Technical Report TR-98-03, Air Force Institute of Technology, Wright Patterson AFB, OH, 1998.

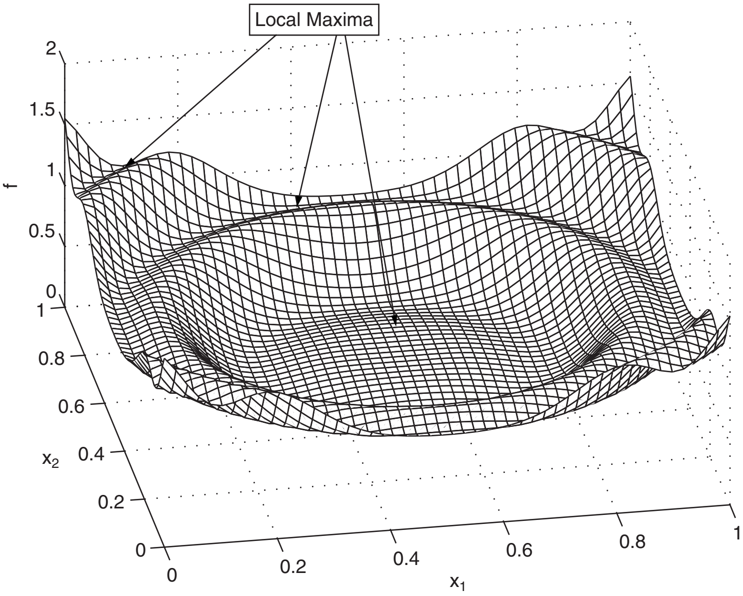


Figure 13.3 Surface plot of the bottom-of-a-bottle function.

As seen in Figure 13.3, the function is axially symmetric and has a dome-like region in the center whose shape resembles the bottom of a bottle. Also, there are three regions each having an infinite number of local maxima located on a circle: the central dome-like local region, the middle maximum region, and the one further outside these two regions. These three regions lie on a circle.

We now use fminsearch in the following script to obtain a local maximum for the above function with three sets of starting values of *x*0:

```
x0 = [0.2, 0.8; 0.2, 0.9; 0.3, 0.6];
options = optimset('Large', 'off');
for n = 1:3
   [x, f] = fminsearch(@Bottle, x0(n,:), options);
   disp(['For x0 = [' num2str(x0(n,:)) '], x = [' num2str(x) '] and f = ' num2str(-f)])
end
```

Upon execution, we obtain the following values:

```
For x0 = [0.2  0.8],   x = [0.20677  0.80195]   and   f = 0.81935
For x0 = [0.2  0.9],   x = [0.18034  0.9995]    and   f = 1.2121
For x0 = [0.3  0.6],   x = [0.4526   0.48062]   and   f = 0.42665
```

This problem, in fact, has an infinite number of local maxima and the location of the local optimum obtained from fminsearch, therefore, depends upon the initial point *x*0.

13.4.2 Curve Fitting: One Independent Variable

Given a set of *N* input values $x_{\text{data}} = [x_1, x_2, \ldots, x_N]$ for an input parameter and a corresponding set of output values $y_{\text{data}} = [y_1, y_2, \ldots, y_N]$ for an output parameter. Consider a function $f(x, a_1, \ldots, a_M)$ where *x* is the independent variable and the a_m

are the M coefficients that we are trying to determine by using a curve fitting procedure. We define the difference

$$f_n = f(x_n, a_1, \ldots, a_M) - y_n$$

The least squares procedure finds the coefficients a_m that best satisfies

$$\underset{x}{\text{minimize}} \quad \sum_n^N f_n^2 \tag{13.12}$$

This least squares nonlinear curve fitting method can be performed with `lsqcurvefit`. The basic command is

[xopt, resnorm] = `lsqcurvefit` (@**UserFunction**, x0, xdata, ydata, lb, ub, options, p1, p2, ...)

where *xopt* is the vector of optimum values of a, *resnorm* is the Euclidean norm of the residual given by

$$\sum_n^N f_n^2$$

at these values of a, and **UserFunction** is the name of the function that computes the objective function. The quantity $x0$ is the vector of starting values, *xdata* and *ydata* are, respectively, vectors of the input and output data, and *options* set the parameters described in `optimset`. The quantities *lb* and *ub* are vectors representing the lower and upper bounds on x, respectively, that is, $lb \le x \le ub$. An empty matrix [] is used for *lb* and *ub* when they are not used and for *options* when the default quantities are used. The quantities $p1, p2, \ldots$ are parameters that are passed to **UserFunction**.

We now demonstrate the use of `lsqcurvefit`.

Example 13.7 Stress–strain relationship

Consider the stress–strain data for a plastic material that are given in Table 13.3, where σ is the stress and ε is the strain. Assume that the relationship between the stress and strain is of the form

$$\varepsilon = a + b \ln \sigma \tag{13.13}$$

TABLE 13.3 Stress–Strain Data for a Plastic Material

σ (MPa)	ε
6.38	0.11
7.76	0.16
11.20	0.35
14.65	0.48
18.1	0.61
21.55	0.71
25.0	0.85

The objective is to find the coefficients *a* and *b* that produce the best fit to the data values given in Table 13.3. The function required by `lsqcurvefit` is given by the anonymous function **SigmaEpsilonFit**. The program is

```
sigma = [6.38, 7.76, 11.20, 14.65, 18.1, 21.55, 25.0];
epsilon = [0.11, 0.16, 0.35, 0.48, 0.61, 0.71, 0.85];
x0 = [0.1, 0.1];
SigmaEpsilonFit = @(x, sigma) (x(1)+x(2)*log(sigma));
[x, resnorm] = lsqcurvefit(SigmaEpsilonFit, x0, sigma, epsilon);
disp(['a = ' num2str(x(1)) ' b = ' num2str(x(2)) ' Residual = ' num2str(resnorm)])
```

Upon execution, we obtain

```
a = -0.92156    b = 0.53448    Residual = 0.0063323
```

Thus, the best fit function is given by

$$\varepsilon = -0.92156 + 0.53448 \ln \sigma \tag{13.14}$$

13.4.3 Curve Fitting: Several Independent Variables

Nonlinear least squares data fitting with multiple sets of input data $x_{1,n}, x_{2,n}, \ldots,$ and a single set of corresponding output data y_n, $n = 1, 2, \ldots, N$, are obtained with `lsqnonlin`. The `lsqnonlin` function finds *x* such that

$$\underset{x}{\text{minimize}} \quad \sum_{n=1}^{N} h_n^2 \tag{13.15}$$

where

$$h_n = h(x_{1,n}, x_{2,n}, x_{3,n}, \ldots, a_1, \ldots, a_M) - y_n$$

and $x_{k,n}$, $k = 1, 2, \ldots,$ are the independent input variables and a_m are the *M* coefficients that we are trying to determine by using the least squared procedure. The `lsqnonlin` function is

[xopt, resnorm] = `lsqnonlin`(@**UserFunction**, x0, lb, ub, options, p1, p2, . . .)

where *resnorm* is the Euclidean norm of the residual given by

$$\sum_{n}^{N} h_n^2$$

and **UserFunction** is the name of the function that computes the objective functions h_n, not h_n^2. The quantity *x*0 is the vector of starting values, *lb* and *ub* are the lower and upper bounds on *x*, *options* sets the parameters described in `optimset`, and *p*1, *p*2, . . . are parameters passed to **UserFunction.** Use the pair of brackets [] when *lb* and *ub* are not specified and when the default values are used for *options*.

The function `lsqnonlin` can also be used for a single set of input data with the corresponding single set of output data, as shown in Example 13.8. The `lsqnonlin` function for three independent variables is demonstrated in Example 13.9.

Example 13.8 Stress–strain relationship revisited

The stress–strain relationship of Example 13.7 is now solved with `lsqnonlin`. The design variables *a* and *b* are determined by minimizing

$$\underset{a,\,b}{\text{minimize}} \quad \sum_{i=1}^{7} [\varepsilon_i - (a + b \text{ In } \sigma_i)]^2 \tag{13.16}$$

where ε_i and σ_i correspond to the experimentally obtained values in Table 13.3.

The program to solve Eq. (13.16) is as follows:

```
sigma = [6.38, 7.76, 11.20, 14.65, 18.1, 21.55, 25.0];
epsilon = [0.11, 0.16, 0.35, 0.48, 0.61, 0.71, 0.85];
SigmaEpsilonFit = @(x, sigma, epsilon) (x(1)+x(2)*log(sigma)-epsilon);
[x, resnorm] = lsqnonlin(SigmaEpsilonFit, [0.1, 0.1], [], [], [], sigma, epsilon);
disp(['a = ' num2str(x(1)) ' b = ' num2str(x(2)) ' Residual = ' ...
        num2str(resnorm)])
```

Upon execution, we obtain

```
a = -0.92156    b = 0.53448    Residual = 0.0063323
```

which are the same as those obtained in Example 13.7.

Example 13.9 Semiempirical $P-v-T$ relationship

It is well known that the $P-v-T$ relationship of real gases deviates from that estimated by the ideal gas formula

$$Pv = RT \tag{13.17}$$

where P is the pressure in atmosphere (atm), v is the molar volume in cm^3/g mol, T is the temperature in K, and R is the gas constant equal to 82.06 atm cm^3/g mol K. A semiempirical relationship used to correct the departure from the ideal gas is[6]

$$P = \frac{RT}{v - b} - \frac{a}{v(v + b)\sqrt{T}} \tag{13.18}$$

where the values of a and b are the constants that will be determined from a fit to the experimental data. Listed in Table 13.4 are the $P-v-T$ experimental measurements obtained for a gas. The design variables a and b are determined by minimizing the following least squares objective function:

$$\underset{a,\,b}{\text{minimize}} \quad \sum_{i=1}^{8} \left[P_i - \frac{RT_i}{v_i - b} + \frac{a}{v_i(v_i + b)\sqrt{T_i}} \right]^2 \tag{13.19}$$

where P_i, v_i, and T_i correspond to the values at the ith experimental run shown in Table 13.4.

[6] G. V. Reklaitis, A. Ravindran, and K. M. Ragsdell, *Engineering Optimization*, John Wiley & Sons, New York, 1983, pp. 20–22.

TABLE 13.4 $P{-}v{-}T$ Data for a Gas

Run number	P (atm)	v (cm^3/g mol)	T (K)
1	32.7	480	283
2	42.6	480	313
3	44.5	576	375
4	25.7	672	283
5	36.6	576	313
6	38.6	672	375
7	37.6	384	283
8	63.0	384	375

The program to determine the optimum values of *a* and *b* is as follows:

```
x0 = [8000, 40];    R = 82.06;
T = [283, 313, 375, 283, 313, 375, 283, 375];
v = [480, 480, 576, 672, 576, 672, 384, 384];
P = [32.7, 42.6, 44.5, 25.7, 36.6, 38.6, 37.6, 63.0];
pvt = @(x, R, T, v, P) (P-R*T./(v-x(2))+x(1)./(sqrt(T).*v.*(v+x(2))));
options = optimset('MaxFunEvals', 600);
[x, resnorm] = lsqnonlin(pvt, x0, [], [], options, R, T, v, P);
disp(['a = ' num2str(x(1)) ' b = ' num2str(x(2)) ' Residual = ' num2str(resnorm)])
```

Upon execution, we obtain

```
a = 74223871.6748    b = 30.6819    Residual = 13.8143
```

Thus, the best fit function is

$$P = \frac{RT}{v - 30.682} - \frac{7.422 \times 10^7}{v(v + 30.682)\sqrt{T}} \tag{13.20}$$

For this example, the solution is sensitive to the initial point *x*0, which affects the value of *resnorm*. We are always looking for a solution that gives the smallest value for *resnorm*.

Example 13.10 Mineral exploration

A 2 km × 2 km site is believed to overlay a thick layer of mineral deposits. To create a model of the mineral deposit profile and to establish the economic viability of mining the site, a preliminary subsurface exploration consisting of sixteen boreholes is conducted. Each borehole is drilled to a depth of approximately 45 m, with the upper and lower boundaries of mineral deposits being recorded. The borehole data[7] are given in Table 13.5. We wish to create a three-dimensional computer model of the subsurface

[7] M. Austin and D. Chancogne, *Engineering Programming in C, MATLAB and JAVA*, John Wiley & Sons, New York, 1998, p. 461.

TABLE 13.5 Subsurface Mineral Exploration Data

Bore hole number	Coordinates (x, y)	Depth z (top, bottom)
1	(10, 10)	(−30.5, −40.5)
2	(750, 10)	(−29, −39.8)
3	(1250, 10)	(−28, −39.3)
4	(1990, 10)	(−26.6, −38.5)
5	(10, 750)	(−34.2, −41.4)
6	(750, 750)	(−32.8, −40.6)
7	(1250, 750)	(−31.8, −40.1)
8	(1990, 750)	(−30.3, −39.4)
9	(10, 1250)	(−36.7, −42)
10	(750, 1250)	(−35.2, −41.2)
11	(1250, 1250)	(−34.2, −40.7)
12	(1990, 1250)	(−32.8, −40)
13	(10, 1990)	(−40.4, −42.8)
14	(750, 1990)	(−39, −42.1)
15	(1250, 1990)	(−38, −41.6)
16	(1990, 1990)	(−36.5, −40.9)

mineral deposits. The four vertical sides are defined by the 2 km × 2 km boundaries of the site. The top and bottom planes are defined by the surface

$$z(x, y) = a_0 + a_1 x + a_2 y \tag{13.21}$$

where a_0, a_1, and a_2 are to be determined.

We use `lsqnonlin` in the following script to obtain the best fit and to create the top and bottom planes for the data given in Table 13.5.

```
xb = [10.0, 750.0, 1250.0, 1990.0, 10.0, 750.0, 1250.0, 1990.0 . . .
      10.0, 750.0, 1250.0, 1990.0, 10.0, 750.0, 1250.0, 1990.0];
yb = [10.0, 10.0, 10.0, 10.0, 750.0, 750.0, 750.0, 750.0, 1250.0 . . .
      1250.0, 1250.0, 1250.0, 1990.0, 1990.0, 1990.0, 1990.0];
zbtop = [-30.5,-29.0,-28.0,-26.6,-34.2,-32.8,-31.8,-30.3, . . .
         -36.7,-35.2,-34.2,-32.8,-40.4,-39.0,-38.0 -36.5];
zbbot =[-40.5,-39.8,-39.3,-38.5,-41.4,-40.6,-40.1,-39.4, . . .
         -42.0,-41.2,-40.7,-40.0,-42.8,-42.1,-41.6,-40.9];
x0 = [1, 1, 1];
z = @(x, xb, yb) (x(1)+x(2)*xb+x(3)*yb);
MinDepError =@ (x, xb, yb, zb) (zb-z(x, xb, yb));
[xtop, Errornormtop] = lsqnonlin(MinDepError, x0, [], [], [], xb, yb, zbtop);
[xbot, Errornormbot] = lsqnonlin(MinDepError, x0, [], [], [], xb, yb, zbbot);
[xb, yb] = meshgrid(linspaces(0, 2000, 12), linspace(0, 2000, 12));
disp(['Residual top = ' num2str(Errornormtop) ' Residual bottom = ' . . .
      num2str(Errornormbot)])
mesh(xb, yb, z(xtop, xb, yb));
hold on
mesh(xb, yb, z(xbot, xb, yb));
```

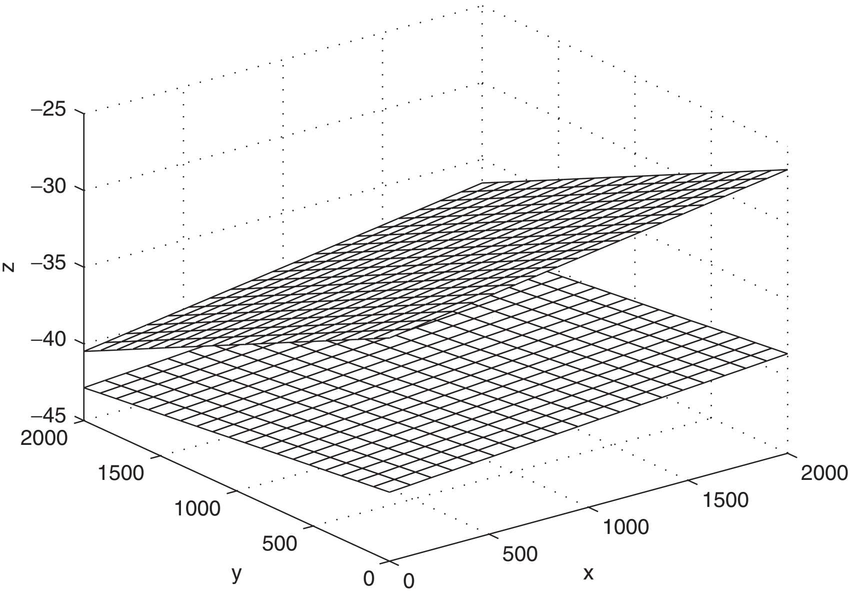


Figure 13.4 Top and bottom fitted planes for Example 13.10.

```
xlabel('x (m)')
ylabel('y (m)')
zlabel('z (m)');
```

Upon execution, we obtain the two best fit planes shown in Figure 13.4 and the following results are displayed to the command window:

```
Residual top = 0.017805    Residual bottom = 0.01219
```

13.5 NONLINEAR PROGRAMMING: CONSTRAINED SINGLE OBJECTIVE

Constrained nonlinear optimization methods find a local minimum of a constrained function as formulated by Eqs. (13.1) and (13.2) for the case of a single-objective function, that is, for $M = 1$ in Eq. (13.1).

13.5.1 Constrained Single-Variable Method

The constrained single-variable method finds the minimum of a function of one variable on a fixed interval, that is,

$$\begin{aligned} &\underset{x}{\text{minimize}}\ f(x) \\ &\text{subject to: } x_1 \le x \le x_2 \end{aligned} \tag{13.22}$$

The MATLAB function that performs this minimization is[8]

```
[xopt, fxopt] = fminbnd(@UserFunction, x1, x2, options, p1, p2, . . . )
```

where $xopt = x_{\mathrm{opt}}$ is the optimum value of x, $fxopt = f(x_{\mathrm{opt}})$, and **UserFunction** is the name of the function that computes the objective function. The quantities $x1$ and $x2$ define the interval over which **UserFunction** is minimized with respect to x, *options* sets the parameters described in `optimset`, and $p1$, $p2$, ... are parameters passed to **UserFunction**.

The `fminbnd` function is now demonstrated.

Example 13.11 Piping cost in a plant[9]

Piping costs are important considerations in the design of a chemical plant. Consider the design of a pipeline that is L meters long and is to carry fluid at the rate of Q L /min. The objective is to determine the pipe diameter D in millimeters that minimizes the annual pumping cost. For a standard pump, the annual pumping cost can be estimated from

$$f(D) = 1.476L + 0.0063LD^{1.5} + 325(hp)^{0.5} + 61.6(hp)^{0.925} + 102 \qquad (13.23)$$

where

$$hp = 0.0281\frac{LQ^3}{D^5} + 6.677 \times 10^{-4}\frac{LQ^{2.68}}{D^{4.68}} \qquad (13.24)$$

We shall now obtain the pipe diameter D for a minimum cost of a pipe with a length of 300 m and a flow rate of 76 L/min by using the following program:

```
function Example13_11
L = 300;   Q = 76;
[D, fD] = fminbnd(@PipeLineCost, 15, 50, [], L, Q);
disp(['Pipe diameter = ' num2str(D) ' mm Annual pumping cost = $' ...
      num2str(fD, '%6.2f')])
function f = PipeLineCost(D, L, Q)
hp = 0.0281*L*Q^3./D.^5+(6.6768e-004*L*Q^2.68)./(D.^4.68);
f = 1.476*L+0.0063*L*D.^1.5+325*hp.^0.5+61.6*hp.^0.925+102;
```

Upon execution, we obtain

```
Pipe diameter = 28.4403 mm    Annual pumping cost = $991.70
```

Example 13.12 Maximum volume of a closed box

We shall determine the dimensions of a closed box with a maximum volume V. The box is constructed from one piece of cardboard 90 cm × 90 cm by cutting four squares from its four corners, as shown in Figure 13.5. The lower and upper values of x and the

[8] See also Section 5.5.7.

[9] Adopted from Reklaitis, et al., *Engineering Optimization*, pp. 66–67.

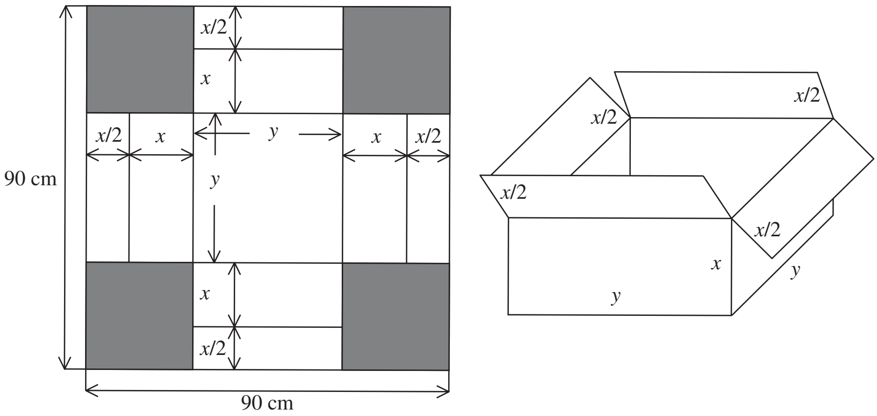


Figure 13.5 Construction of a closed box.

height of the box are 8 cm and 12 cm, respectively. From Figure 13.5, we see that $y = 90 - 3x$ and, therefore, the volume of the closed box is

$$V(x) = xy^2 = x(90 - 3x)^2$$

The optimization statement is

$$\begin{aligned} &\text{maximize} \;\; V(x) \\ &\text{subject to: } 8 \le x \le 12 \end{aligned} \tag{13.25}$$

The program to determine x is as follows:

```
function Example13_12
L = 90;
Volume = inline('-(L-3*x)^2*x', 'x', 'L');
[x, V] = fminbnd(Volume, 8, 12, [], L);
disp(['x = ' num2str(x) ' cm V = ' num2str(-V) ' cm^3'])
```

Upon execution, we obtain

```
x = 10 cm   V = 36000 cm^3
```

13.5.2 Constrained Multivariable Method

The constrained multivariable method finds the minimum of a nonlinear multivariable constrained optimization problem. Both equality and inequality constraints can be considered. Also, both the objective and/or the constraint functions can be nonlinear. A nonlinear multivariable constrained optimization problem is stated as

$$\begin{aligned} &\underset{x}{\text{minimize}}\; f(x) \\ &\text{subject to:} \quad Ax \le b \quad &&\text{(linear inequality constraints)} \\ &\qquad A_{\text{eq}}x = b_{\text{eq}} &&\text{(linear equality constraints)} \\ &\qquad C(x) \le 0 &&\text{(nonlinear inequality constraints)} \\ &\qquad C_{\text{eq}}(x) = 0 &&\text{(nonlinear equality constraints)} \\ &\qquad lb \le x \le ub \end{aligned} \tag{13.26}$$

The basic command to solve Eq. (13.26) is

[xopt, fxopt] = `fmincon`(@**UserFunction**, x0, A, b, Aeq, beq, lb, ub, . . .
@**NonLinConstr**, options, p1, p2, . . .)

where *xopt* $= x_{\text{opt}}$ is the optimum value of x, *fxopt* $= f(x_{\text{opt}})$, and **UserFunction** is the name of the function that computes the objective function. The quantity $x0$ is the vector of starting values, the matrix A and the vector b are the coefficients of the linear inequality constraints, the matrix *Aeq* and the vector *beq* are the coefficients of the equality constraints, *lb* and *ub* are the vectors of the lower and upper bounds on x, respectively, *options* sets the parameters described in `optimset`, and $p1, p2, \ldots$ are the additional arguments that are passed to **UserFunction** and **NonLinConstr**. The quantity **NonLinConstr** is a function that defines the nonlinear constraints. The arguments $p1, p2, \ldots$ of **UserFunction** and **NonLinConstr** *must be identical*, even if only one of the functions uses these values. If *lb*, *ub*, and *options* are not specified, one uses []; similarly for the case when A, b, *Aeq*, and *beq* are not specified.

There are three solution methods for this function: active set, interior point, and trust region reflective (default). The desired solution method is selected by using `optimset`. For example, to select 'active-set', one uses

options = `optimset`('Algorithm', 'active-set');

We now demonstrate the use of `fmincon`.

Example 13.13 Two-bar truss[10]

Consider the two-bar truss shown in Figure 13.6. The objective is to minimize the volume V of the two bars AC and BC. We denote x_1 and x_2 as the cross-sectional areas of the bars AC and BC, respectively, and we denote y as the vertical position of joint C. The constraints of the system are that the tensile stresses on the two bars must be less than or equal to $\sigma = 10^5$ kPa and that the vertical position y remains between 1 m and 3 m. Finally, x_1 and x_2 are nonnegative. The optimization statement is

$$\begin{aligned}
&\text{minimize } V = x_1\sqrt{16 + y^2} + x_2\sqrt{1 + y^2} \\
&\text{subject to:} \quad \left(20\sqrt{16 + y^2}\right)/(yx_1) - \sigma \le 0 \\
&\qquad \left(80\sqrt{1 + y^2}\right)/(yx_2) - \sigma \le 0 \qquad (13.27) \\
&\qquad 1 \le y \le 3 \\
&\qquad (x_1, x_2) \ge 0
\end{aligned}$$

Therefore, $A = b = A_{\text{eq}} = b_{\text{eq}} = C_{\text{eq}} = 0$, $lb = [1, 0, 0]$, and $ub = [3, \infty, \infty]$.

[10] U. Kirsch, *Optimal Structural Design*, McGraw-Hill, New York, 1981.

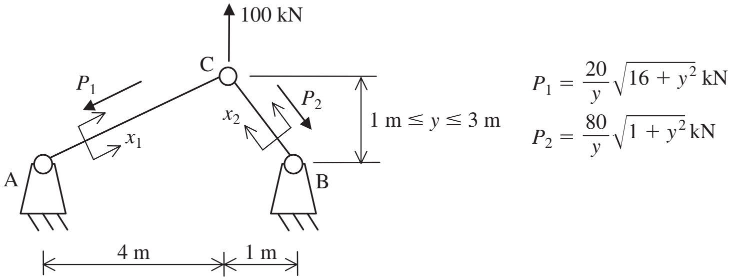

100 kN
C
P_1
P_2
x_2
x_1
A
B
1 m ≤ y ≤ 3 m
4 m
1 m
$P_1 = \frac{20}{y}\sqrt{16 + y^2}$ kN
$P_2 = \frac{80}{y}\sqrt{1 + y^2}$ kN

Figure 13.6 Two-bar truss: x_1 and x_2 are cross-sectional areas.

The program that solves Eq. (13.27) is as follows. We have arbitrarily selected the solution algorithm as "active set."

```
function Example13_13
x0 = [1, 1, 1];    sigma = 1e5;
lb = [1, 0, 0];    ub = [3, inf, inf];
options = optimset('Algorithm', 'active-set');
[x, f] = fmincon(@TrussNonLinF, x0, [], [], [], [], lb, ub, @TrussNonLinCon, . . .
                    options, sigma);
disp(['y = ' num2str(x(1)*100) ' cm x1 = ' num2str(x(2)*1e4) ' cm^2 x2 = ' . . .
        num2str(x(3)*1e4) ' cm^2 Volume = ' num2str(f*1e6) ' cm^3'])

function f = TrussNonLinF(x, sigma)
f = x(2)*sqrt(16+x(1)^2)+x(3)*sqrt(1+x(1)^2);

function [C,Ceq] = TrussNonLinCon(x, sigma)
C(1) = 20*sqrt(16+x(1)^2)-sigma*x(1)*x(2);
C(2) = 80*sqrt(1+x(1)^2)-sigma*x(1)*x(3);
Ceq = [];
```

Upon execution, we obtain

```
y = 195.039 cm    x1 = 4.5634 cm^2    x2 = 8.9902 cm^2    Volume = 4001.2619 cm^3
```

Example 13.14 Helical compression spring

Helical compression springs can be found in numerous mechanical devices. They are used to exert force, to provide flexibility, and to either store or absorb energy. To design a helical compression spring, such criteria as fatigue, yielding, surging, and buckling may have to be taken into consideration. To obtain a solution that meets the various mechanical requirements, an optimization study is performed.

The design objective is to maximize the safety factor, which is equivalent to minimizing the inverse of the safety factor. Referring to Figure 13.7, the two design variables are c and d, where $c = D/d$ is the spring index, D is the mean coil diameter, and d is the wire diameter. The design objective is to minimize the inverse of the safety factor for fatigue or yielding, whichever is critical. For fatigue, the inverse of the safety factor SF_f is obtained from

$$\frac{1}{\text{SF}_f} = \frac{\tau_a}{S_{\text{ns}}} + \frac{\tau_m}{S_{\text{us}}} \tag{13.28}$$

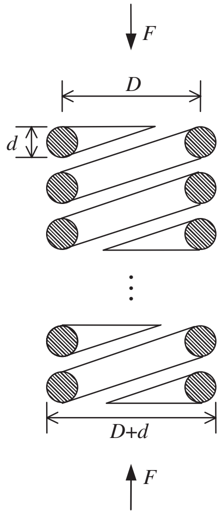


Figure 13.7 Helical compression spring.

where τ_a and τ_m are, respectively, the alternating and mean components of the shear stress, S_{ns} is the material's fatigue strength, and S_{us} is the ultimate strength.

For yielding, the inverse of the safety factor SF_y is obtained from

$$\frac{1}{\text{SF}_y} = \frac{\tau_a + \tau_m}{S_{ys}} \tag{13.29}$$

where S_{ys} is the shear yield strength. If the following condition is satisfied

$$\frac{\tau_a}{\tau_m} \geq \frac{S_{\text{ns}}(S_{\text{ys}} - S_{\text{us}})}{S_{\text{us}}(S_{\text{ns}} - S_{\text{ys}})} \tag{13.30}$$

then the inverse of the safety factor for fatigue given by Eq. (13.28) will be the objective function, otherwise, the inverse of the safety factor for yielding, given by Eq. (13.29), will be the objective function. The mean and alternating components of the shear stress are, respectively,

$$\begin{aligned} \tau_m &= \frac{8F_mK_w}{\pi}\frac{c}{d^2} \\ \tau_a &= \frac{8F_aK_w}{\pi}\frac{c}{d^2} \end{aligned} \tag{13.31}$$

where

$$\begin{aligned} K_w &= \frac{4c - 1}{4c + 4} + \frac{0.615}{c} \\ F_a &= (F_U - F_L)/2 \\ F_m &= (F_U + F_L)/2 \end{aligned} \tag{13.32}$$

and F_U and F_L are, respectively, the maximum and minimum applied compressive forces along the spring's axis, F_a and τ_a are the alternating force and shear stress, respectively, F_m and τ_m are the mean force and shear stress, respectively, and K_w is the Wahl correction factor for the curvature and direct shear effects on the spring.

The overall design optimization formulation for the helical compression spring is as follows. The design objective is to

$$\text{minimize} \quad \frac{1}{\text{SF}_f} = \frac{\tau_a}{S_{\text{ns}}} + \frac{\tau_m}{S_{\text{us}}} \tag{13.33}$$

for fatigue or to

$$\text{minimize} \quad \frac{1}{\text{SF}_y} = \frac{\tau_a + \tau_m}{S_{\text{ys}}} \tag{13.34}$$

for yielding, depending on whether or not Eq. (13.30) is satisfied. The objective function is subjected to the following constraints:

$$\begin{aligned}
&K_1 d^2 - c \leq 0 \leq && \text{Surging} \\
&K_2 - c^5 \leq 0 && \text{Buckling} \\
&K_3 c^3 - d \leq 0 && \text{Minimum number of coils} \\
&K_4 d^2/c^3 + K_8 d - 1 \leq 0 && \text{Pocket length} \\
&K_5(cd + d) - 1 \leq 0 && \text{Maximum coil diameter} \\
&1/c + K_6/(dc) - 1 \leq 0 && \text{Minimum coil diameter} \\
&K_7 c^3 - d^2 \leq 0 && \text{Clash allowance}
\end{aligned} \tag{13.35}$$

where

$$\begin{aligned}
&K_1 = \frac{G f_r \Delta}{2.865 \times 10^6 (F_U - F_L)} && K_2 = \frac{G f_U (1 + A)}{22.3 k^2} && K_3 = \frac{8 k N_{\min}}{G} \\
&K_4 = \frac{G(1 + A)}{8 k L_m} && K_5 = \frac{1}{OD} && K_6 = ID \\
&K_7 = \frac{0.8(F_U - F_L)}{AG} && K_8 = \frac{Q}{L_m} && k = \frac{F_U - F_L}{\Delta} \\
&S_{\text{ns}} = C_1 d^{A_1} \overline{NC}^{B_1} && S_{\text{us}} = C_2 d^{A_1} && S_{\text{ys}} = C_3 d^{A_1}
\end{aligned} \tag{13.36}$$

The values for the parameters of this system are as follows:

$A = 0.4$	dimensionless clearance constant
$f_r = 500$ Hz	minimum allowable natural frequency
$G = 79{,}290$ N/mm^2	shear modulus for steel
ID = 19.0 mm	minimum allowable inside diameter of the spring

OD = 38.1 mm	maximum allowable outside diameter of the spring
$N_{\min} = 3$	minimum allowable number of coils
$L_m = 31.8$ mm	maximum spring length under maximum load
$Q = 2$	number of inactive coils
$\overline{NC} = 10^6$ cycles	number of cycles to failure
$\Delta = 6.35$ mm	spring deflection
$F_L = 66.7$ N	minimum applied compressive force along the spring's axis
$F_U = 133.4$ N	maximum applied compressive force along the spring's axis

The spring material is piano wire for which $A_1 = 0.14$, $B_1 = -0.2137$, $C_1 = 683.72$, $C_2 = 1735.1$, and $C_3 = 938.6$. The lower and upper bounds on the spring index c and wire diameter d are

$$4 \le c \le 20$$

and

$$0.1 \le d \le 6$$

respectively. Thus, we have one design objective, two design variables, seven constraints, and upper and lower bounds on the design variables.

The script given below uses three subfunctions: **SpringParameters**, which computes the various spring constants K_j, $j = 1, 2, \ldots, 8$, given by Eq.(13.36), **SpringNLConstr**, which computes the nonlinear constraints given by Eq. (13.35), and **SpringObjFunc**, which computes the objective function given by Eqs. (13.28)–(13.34).

```
function Example13_14
A = 0.4;    FL = 66.7;    FU = 133.4;    G = 79290;
fr = 500;    ID = 19.0;    OD = 38.1;    Lm = 31.8;
NC = 10^6;    Nmin = 3;    Q = 2;    Delta = 6.35;
A1 = -0.14;    B1 = -0.2137;
C1 = 6837.2;    C2 = 1735.1;    C3 = 938.6;
[K, Fa, Fm] = SpringParameters(A, FL, FU, G, fr, ID, OD, Lm, Nmin, Q, Delta);
options = optimset('Algorithm', 'active-set');
[x, f] = fmincon(@SpringObjFunc, [10, 2], [], [], [], [],[4, 0.1], [20, 6], . . .
        @SpringNLConstr, options, K, Fa, Fm, NC, A1, B1, C1, C2, C3);
disp(['c = ' num2str(x(1)) ' d = ' num2str(x(2)) ' mm Safety factor = ' num2str(1/f)])

function [K, Fa, Fm] = SpringParameters(A, FL, FU, G, fr, ID, OD, Lm, Nmin, Q,
Delta)
Fa = (FU-FL)/2;
Fm = (FU+FL)/2;
k = (FU-FL)/Delta;
K(1) = G*fr*Delta/(2865000*(FU-FL));
K(2) = G*FU*(1+A)/(22.3*k^2);
K(3) = 8*k*Nmin/G;
K(4) = G*(1+A)/(8*k*Lm);
K(5) = 1/OD;
K(6) = ID;
K(7) = 0.8*(FU-FL)/(A*G);
K(8) = Q/Lm;
```

```
function [C, Ceq] = SpringNLConstr(x, K, Fa, Fm, NC, A1, B1, C1, C2, C3)
c = x(1);    d = x(2);
C(1) = K(1)*d^2-c;
C(2) = K(2)-c^5;
C(3) = K(3)*c^3-d;
C(4) = K(4)*d^2/c^3+K(8)*d-1;
C(5) = K(5)*(c*d+d)-1;
C(6) = 1/c+K(6)/c/d-1;
C(7) = K(7)*c^3-d^2;
Ceq = [];

function f = SpringObjFunc(x, K, Fa, Fm, NC, A1, B1, C1, C2, C3)
c = x(1);    d = x(2);
Sns = C1*d^A1*NC^B1;
Sus = C2*d^A1;
Sys = C3*d^A1;
Kw = (4*c-1)/(4*c+4)+0.615/c;
Temp = 8*c*Kw/(pi*d^2);
TauA = Fa*Temp;
TauM = Fm*Temp;
Ratio = TauA/TauM;
SS = Sns*(Sys-Sus)/(Sus*(Sns-Sys));
if (Ratio-SS)>=0
   f = TauA/Sns+TauM/Sus;
else
   f = (TauA+TauM)/Sys;
end
```

Upon execution, we obtain

```
c = 8.4861    d = 2.538 mm    Safety factor = 1.8411
```

Example 13.15 Gear reducer[11]

Consider the design of a gear train with a gear and a pinion as shown in Figure 13.8. We shall minimize the volume of these two gears and their corresponding shafts. There are seven design variables as follows:

x_1 = gear face width
x_2 = module
x_3 = number of teeth of the pinion
x_4 = distance between bearing set 1
x_5 = distance between bearing set 2
x_6 = diameter of shaft 1
x_7 = diameter of shaft 2

[11] Based on J. Golinski, "Optimum synthesis problems solved by means of nonlinear programming and random methods," *Journal of Mechanisms*, **5**, 1970, pp. 287–309.

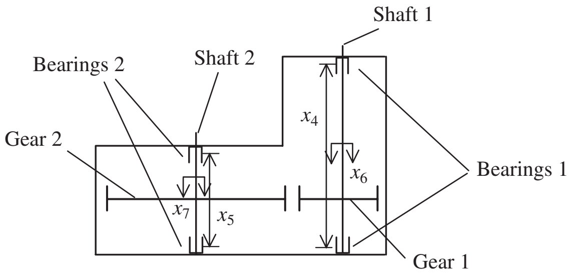


Figure 13.8 Gear reducer.

The application is such that the lower and upper limits on these variables are

$$
\begin{aligned}
2.6 &\le x_1 \le 3.6\\
0.7 &\le x_2 \le 0.8\\
17 &\le x_3 \le 28\\
7.3 &\le x_4 \le 8.3\\
7.3 &\le x_5 \le 8.3\\
2.9 &\le x_6 \le 3.9\\
5.0 &\le x_7 \le 5.5
\end{aligned}
\tag{13.37}
$$

where all the dimensions are in centimeters, except for the two nondimensional quantities x_2 and x_3.

The design objective is to minimize the overall volume of the shafts, which is given by

$$
\begin{aligned}
\text{minimize } V = 0.7854 x_1 x_2^2\,(3.3333\,x_3^2 + 14.933\,x_3 - 43.0934) - 1.508\,x_1\,(x_6^2 + x_7^2)\\
+ 7.477\,(x_6^3 + x_7^3) + 0.7854\,(x_4 x_6^2 + x_5 x_7^2)
\end{aligned}
\tag{13.38}
$$

and are subject to the constraints given in Table 13.6. Constraints 1–7 in the table form the nonlinear inequality constraints, and constraints 8–11 form the linear inequality constraints. From the linear inequality constraints, we see that

$$
A = \begin{bmatrix}
-1 & 5 & 0 & 0 & 0 & 0 & 0\\
1 & -12 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & -1 & 0 & 1.5 & 0\\
0 & 0 & 0 & 0 & -1 & 0 & 1.1
\end{bmatrix}
$$

$$
b = [0 \quad 0 \quad -1.9 \quad -1.9]'
$$

and $A_{\text{eq}} = b_{\text{eq}} = C_{\text{eq}} = 0$. From Eq. (13.37), we find that $L_{\text{bound}} = [2.6, 0.7, 17, 7.3, 7.3, 2.9, 5]$ and $U_{\text{bound}} = [3.6, 0.8, 28, 8.3, 8.3, 3.9, 5.5]$.

The program that solves Eq. (13.38) subject to the constraints of Table 13.6 is as follows:

```
function Example13_15
x0 = [2.6, 0.7, 17, 7.3, 7.3, 2.9, 5];
lb = [2.6, 0.7, 17, 7.3, 7.3, 2.9, 5];
ub = [3.6, 0.8, 28, 8.3, 8.3, 3.9, 5.5];
```

TABLE 13.6 Constraints for the Gear Reducer of Exercise 13.15

Constraint number	Constraint	Constraint description
1	$\dfrac{1}{(x_1x_2^2x_3)} - \dfrac{1}{27} \le 0$	Bending stress of gear tooth
2	$\dfrac{1}{(x_1x_2^2x_3^2)} - \dfrac{1}{397.5} \le 0$	Contact stress of gear tooth
3	$\dfrac{x_4^3}{(x_2x_3x_6^4)} - \dfrac{1}{1.93} \le 0$	Shaft 1 deflection
4	$\dfrac{x_5^3}{(x_2x_3x_7^4)} - \dfrac{1}{1.93} \le 0$	Shaft 2 deflection
5	$\dfrac{1}{0.1x_6^3}\sqrt{\left(\dfrac{745x_4}{x_2x_3}\right)^2 + 16.9 \times 10^6} - 1100 \le 0$	Shaft 1 stress
6	$\dfrac{1}{0.1x_7^3}\sqrt{\left(\dfrac{745x_5}{x_2x_3}\right)^2 + 157.5 \times 10^6} - 850 \le 0$	Shaft 2 stress
7	$x_2x_3 - 40 \le 0$	Space restriction
8	$5x_2 - x_1 \le 0$	Space restriction
9	$x_1 - 12x_2 \le 0$	Space restriction
10	$1.9 - x_4 + 1.5x_6 \le 0$	Shaft requirement
11	$1.9 - x_5 + 1.1x_7 \le 0$	Shaft requirement

```
A = zeros(4, 7);
A(1,1) = -1;   A(1,2 )= 5;
A(2,1) = 1;   A(2,2) = -12;
A(3,4) = -1;   A(3,6) = 1.5;
A(4,5) = -1;   A(4,7) = 1.1;
b = [0, 0, -1.9, -1.9];
options = optimset('Algorithm', 'active-set');
[x, V] = fmincon(@GearObjFunc, x0, A, b , [], [], lb, ub, @GearNonLinConstr, options);
for n = 1:length(x)
  if n == 2 | n == 3
    a = [];
  else
    a = ' cm';
  end
  disp(['x(' int2str(n) ') = ' num2str(x(n)) a])
end
disp(['Volume = ' num2str(V) ' cm^3'])

function f = GearObjFunc(x)
f = 0.7854*x(1)*x(2)^2*(3.3333*x(3)^2+14.9334*x(3)-43.0934) . . .
    -1.508*x(1)*(x(6)^2+x(7)^2)+7.477*(x(6)^3+x(7)^3) . . .
    +0.7854*(x(4)*x(6)^2+x(5)*x(7)^2);
```

```
function [C, Ceq] = GearNonLinConstr(x)
C(1) = 1/(x(1)*x(2)^2*x(3))-1/27;
C(2) = 1/(x(1)*x(2)^2*x(3)^2)-1/397.5;
C(3) = x(4)^3/(x(2)*x(3)*x(6)^4)-1/1.93;
C(4) = x(5)^3/(x(2)*x(3)*x(7)^4)-1/1.93;
C(5) = sqrt((745*x(4)/(x(2)*x(3)))^2+16.9*10^6)/(0.1*x(6)^3)-1100;
C(6) = sqrt((745*x(5)/(x(2)*x(3)))^2+157.5*10^6)/(0.1*x(7)^3)-850;
C(7) = x(2)*x(3)-40;
Ceq = [];
```

Upon execution, we obtain

```
x(1) = 3.5 cm
x(2) = 0.7
x(3) = 17
x(4) = 7.3 cm
x(5) = 7.7153 cm
x(6) = 3.3502 cm
x(7) = 5.2867 cm
Volume = 2994.3413 cm^3
```

13.5.3 Quadratic Programming

Quadratic programming refers to a special class of constrained optimization problems in which the objective function is quadratic and the constraints are linear, that is,

$$\begin{aligned} &\underset{x}{\text{minimize}}\ f = 0.5x^T Hx + c^T x \\ &\text{subject to:} \qquad Ax \le b \\ &\qquad\qquad\qquad A_{\text{eq}}x = b_{\text{eq}} \\ &\qquad\qquad\quad lb \le x \le ub \end{aligned} \tag{13.39}$$

where H, A, and A_{eq} are matrices and b, b_{eq}, c, x, lb, and ub are column vectors. The MATLAB function used to obtain a solution to this class of problems is

```
[xopt, fopt] = quadprog(H, c, A, b, Aeq, beq, lb, ub, x0, options)
```

where $xopt = x_{\text{opt}}$ is the vector of optimum values of x, $fopt = f(\mathrm{x}_{\text{opt}})$ is the symmetric matrix H and the vector c are the set of coefficients of the quadratic objective function f, the matrix A and the vector b are the coefficients of the linear inequality constraints, the matrix *Aeq* and the vector *beq* are the coefficients of the linear equality constraints, the vectors *lb* and *ub* specify the lower and upper bounds on the design variables x, the vector $x0$ sets the starting point, and *options* sets the parameters described in `optimset`.

It is noted that the objective function given by Eq. (13.39) is a quadratic polynomial that can be written as[12]

$$\begin{aligned} f = 0.5\{&[h_{11}x_1^2 + h_{22}x_2^2 + \cdots + h_{nn}x_n^2] + [(h_{12} + h_{21})x_1x_2 + \cdots \\ &+ (h_{1n} + h_{n1})x_1x_n] + [(h_{23} + h_{32})x_2x_3 + \cdots + (h_{2n} + h_{n2})x_2x_n] \\ &+ \cdots + [(h_{(n-1)n} + h_{n(n-1)})x_{(n-1)}x_n]\} + (c_1x_1 + \cdots + c_nx_n) \end{aligned} \tag{13.40}$$

where h_{ij} is the ith row and jth column of H and c_i is the ith element of c. The matrix H can be written in a symmetric form, that is, in a form where $h_{ij} = h_{ji}$, $i \neq j$.

We now demonstrate the use of `quadprog`.

Example 13.16 Production planning revisited

The production planning scenario discussed in Example 13.2 is revised. It is now assumed that the profits of products A and B are functions of the number of units of each product. For product A, the dollar profit per unit varies according to

$$4 + 2x_1 + 3x_2$$

and for product B the dollar profit varies according to

$$5 + 5x_1 + 4x_2$$

where x_1 and x_2 are the number of units produced for A and B, respectively. The remaining aspects of the problem are the same as those given in Example 13.2. Accordingly, the objective function to be minimized is

$$\begin{aligned} f(x_1, x_2) &= -(4 + 2x_1 + 3x_2)x_1 - (5 + 5x_1 + 4x_2)x_2 \\ &= 0.5([-4x_1^2 - 8x_2^2] + [-16x_1x_2]) + (-4x_1 - 5x_2) \end{aligned}$$

We see from Eq. (13.40) that $h_{11} = -4$, $h_{22} = -8$, $h_{12} + h_{21} = 2h_{12} = 2h_{21} = -16$, $c_1 = -4$, and $c_2 = -5$. Thus, the objective function can be written in the vector format of the objective function in Eq. (13.39) as

$$f = \frac{1}{2}[x_1 \quad x_2]\begin{bmatrix} -4 & -8 \\ -8 & -8 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + [-4 \quad -5]\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

The program is as follows:

```
H = [-4, -8; -8, -8];    c = [-4;-5];
A = [1, 1; 1.25, 0.75; 0, 1];
b = [200; 200; 150];
options = optimset('LargeScale', 'off');
[x, P] = quadprog(H, c, A, b, [], [], [0, 0], [], [], options);
disp(['x1 = ' num2str(x(1)) '    x2 = ' num2str(x(2)) '    Profit = $' num2str(-P)])
```

Upon execution, we obtain

```
x1 = 50    x2 = 150    Profit = $155950
```

[12] J. S. Arora, *Introduction to Optimum Design*, 2nd ed., Elsevier Academic Press, San Diego, 2004, p. 97.

13.5.4 Semi-Infinitely Constrained Method

The semi-infinitely constrained method finds the solution of optimization problems expressed as

$$\begin{aligned} &\underset{x}{\text{minimize}}\ f(x) \\ \text{subject to:}\quad & Ax \le b \\ & A_{\text{eq}}x = b_{\text{eq}} \\ & C(x) \le 0 \\ & C_{\text{eq}}(x) = 0 \\ & K_1(x, w_1) \le 0 \\ & K_2(x, w_2) \le 0 \\ & \vdots \\ & K_n(x, w_n) \le 0 \\ & \forall(w_1, \ldots, w_n) \end{aligned} \tag{13.41}$$

where x is the design variable vector, f is the scalar objective function, A is a matrix representing the linear inequality constraints, b is a vector representing the right-hand side of the linear inequality constraints, A_{eq} is a matrix representing the linear equality constraints, b_{eq} is a vector of linear equality constraints, C is a vector representing the nonlinear inequality constraints, and C_{eq} is a vector of nonlinear equality constraints. The quantity $K_n(x, w_n)$ is a vector or a matrix of semi-infinite functions, a function of vectors x and w_n, with the free variable w_n representing a range of values over which the solution for the design variable x is sought. In addition, we use $\forall(w_1, \ldots, w_n)$, with $\forall$ representing "for all," to indicate that the problem is considered for all values of the free variables $w_1, \ldots, w_n$ within their corresponding ranges. For example, w_n may represent a temperature range in a heat transfer problem or a frequency range in a vibration problem over which the K_n-constraint has to be satisfied for all temperature or frequency values in the range. The variables $w_1, \ldots, w_n$ are vectors of, at most, length two.

The MATLAB function to solve Eq. (13.41) is

[xopt, fxopt] = `fseminf`(@**UserFunction**, x0, n, @**SemiConstr**, A, b, . . .
Aeq, Beq, lb, ub, options, p1, p2, . . .)

where $xopt = x_{\text{opt}}$ is the optimum value of x, $fopt = f(x_{\text{opt}})$, and **UserFunction** is the name of the function that computes the scalar function f. The quantity $x0$ sets the starting point, n is the number of semi-infinite constraints in Eq. (13.41), the matrix A and the vector b are the coefficients of the linear inequality constraints, the matrix *Aeq* and the vector *beq* are the coefficients of the linear equality constraints, *lb* and *ub* specify the lower and upper bounds, respectively, on the design variables x, *options* sets the parameters described in `optimset`, and $p1, p2, \ldots$ are the additional parameters that are passed to **UserFunction** and **SemiConstr**. The arguments

$p1, p2, \ldots$ of **UserFunction** and **SemiConstr** must be identical, even if only one of the functions uses these values. If *lb*, *ub*, and *options* are not specified, one uses []; similarly for the case when A, b, Aeq, and beq are not specified. **SemiConstr** is a function that defines the nonlinear constraints and has the following form:

```
function [C, Ceq, K1, K2, . . . , Kn, s] = SemiConstr(x, s, p1, p2, . . . )
% Initial sampling interval
If isnan(s(1,1)),
   s = . . .  % s has n rows and 2 columns
end
w1 = . . .   % computes sample set
 . . .
wn = . . .   % computes sample set
K1 = . . .   % 1st semi-infinite constraint at x and w
 . . .
Kn = . . .   % nth semi-infinite constraint at x and w
C = . . .    % computes nonlinear inequalities
Ceq = . . .  % computes nonlinear equalities
```

where $K1, K2, \ldots, Kn$, are the semi-infinite constraints evaluated for a range of sampled values of the free variables $w_1, \ldots, w_n$, respectively. The rows of the two-column matrix s have the sampling interval for the corresponding values of $\omega_1, \ldots, \omega_n$, which are used to calculate $K1, K2, \ldots, Kn$; that is, the ith row of s contains the sampling interval for evaluating K_i. When K_i is a vector, use only $s(i, 1)$ (the second column can be zero). When K_i is a matrix, $s(i, 2)$ is used for sampling of the rows in K_i and $s(i, 1)$ is used for sampling of the columns in K_i. In the first iteration, s is set to NaN so that some initial sampling interval is determined. If C and/or Ceq do not exist, then set them to [].

We now demonstrate the use of `fseminf`.

Example 13.17 Planar two-link manipulator

Consider a planar two-link manipulator shown in Figure 13.9. This manipulator is capable of positioning to a point in its plane. The design objective is to maximize the workspace area covered by the manipulator. There are two design variables a and b, which represent the lengths of the two links. The constraints are as follows: (1) a lower bound and an upper bound on the ratio a/b, (2) an upper bound on a quantity κ, which is a measure of dexterity, and (3) a lower and an upper bound on the design variables a and b. The dexterity refers to the ease with which the manipulator can either move or exert force or torque along arbitrary directions within its workspace. The condition number κ of the Jacobian matrix for the manipulator is used as a metric for dexterity.[13] It is desired that this condition number be as close to unity as possible. The condition number κ is given by

$$\kappa = (a^2 + 2b^2 + 2ab\cos\theta)/2ab\cos\theta \tag{13.42}$$

[13] C. Gosselin and J. Angeles, "A global performance index for the kinetic optimization of robotic manipulators," *ASME Journal of Mechanical Design*, **113**, 1991, p. 222.

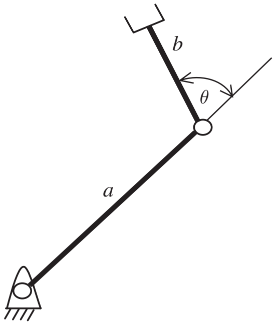


Figure 13.9 Planar two-link manipulator.

The problem statement is

$$\begin{aligned} &\text{minimize } f(a,b) = -\pi[(a+b)^2 - (a-b)^2] \\ &\text{subject to:} \quad a/b \geq 1.1 \\ &\qquad\qquad\quad a/b \leq 2 \\ &\qquad\qquad\quad k \leq 1.26 \\ &\qquad\qquad\quad 0.1 \leq a \leq 2 \\ &\qquad\qquad\quad 0.1 \leq b \leq 2 \\ &\qquad\qquad\quad \forall\,\theta \in [100^\circ, 150^\circ] \end{aligned} \tag{13.43}$$

and the semi-infinite constraints are to be satisfied for the range of $100^\circ \leq \theta \leq 150^\circ$ with a sampling interval of 2°.

The program that solves Eq. (13.43) is as follows:

```
function Example13_17
[x, A] = fseminf(@TwoLinkObjFunc, [1, 1], 1, @TwoLinkConstr, . . .
          [], [], [], [],[.1, 0.1], [2, 2]);
[xx, kap] = fminbnd(@Kappa, 100*pi/180, 150*pi/180, [], x);
disp(['a = ' num2str(x(1)) ' b = ' num2str(x(2)) ' Workspace area = ' num2str(-A)])
disp(['kappa = ' num2str(kap+1.26) ' at theta = ' num2str(xx*180/pi) ' degrees'])
text(120, 1.17, 'Initial')
text(110, 1.05, 'Optimum')
ylabel('Condition number \kappa')
xlabel('\theta (\circ)')
text(120, 1.25, ['\kappa_{min} occurs at \theta = ' num2str(xx*180/pi,5) '\circ'])
axis([100 150 1 1.30])

function [C, Ceq, K1, s] = TwoLinkConstr(x, s)
if isnan(s(1,1))
   s = [2, 0];
end
ab = x(1)/x(2);
C(1) = -ab+1.1;
```

```
C(2) = ab-2;
Ceq = [];
theta = (100:s(1,1):150);
K1 = Kappa(theta*pi/180, x);
plot(theta, K1+1.26, 'k')
hold on

function f = TwoLinkObjFunc(x)
f = -pi*((x(1)+x(2))^2-(x(1)-x(2))^2);

function k = Kappa(theta, x)
k = (x(1)^2+2* x(2)^2+2*(x(1)*x(2))*cos(theta))./(2*(x(1)*x(2))*sin(theta))-1.26;
```

Upon execution, we obtain Figure 13.10 and the following values are displayed to the command window:

```
a = 2   b = 1.4433   Workspace area = 36.274
kappa = 1.0004 at theta = 134.9882 degrees
```

In Figure 13.10, we have plotted the condition number κ as a function of θ for each iteration to show the improvement in the condition number from its initial range to its optimum range. In this optimum range of the condition number, we see from the figure and from the output to the command window that the condition number is unity when $\theta = 135°$. Hence, this angle is the most desirable configuration for the manipulator in terms of its dexterity.

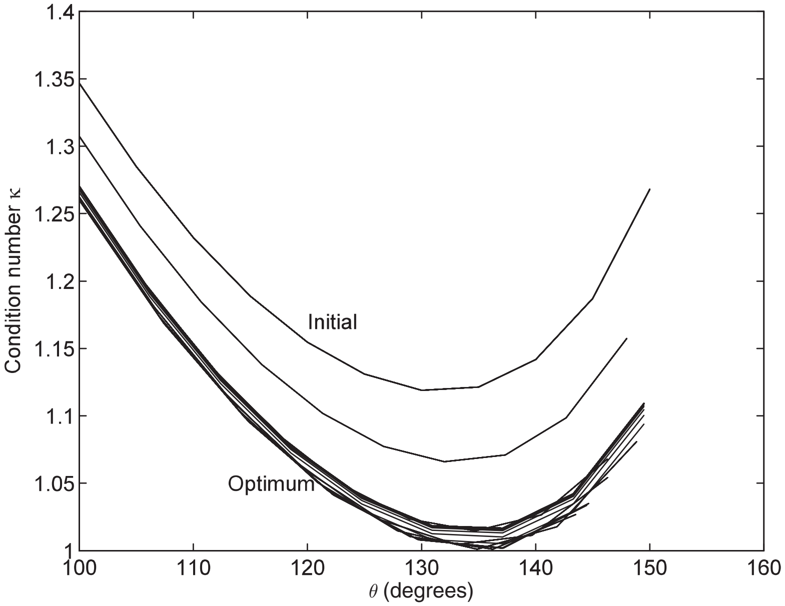


Figure 13.10 The condition number of the planar two-link manipulator as a function of θ as the `fseminf` constraint K_1 progresses from the initial values of a and b to their optimum.

13.6 MULTIOBJECTIVE OPTIMIZATION

Multiobjective optimization refers to the solution of problems in which there is more than one design objective. The objectives in such problems are at least partly in conflict with each other. The conflict arises because of the inherent properties of the problem. For example, consider a structural member in tension in which the objective is to minimize weight and stress. These two objectives conflict with each other; that is, as the weight of the member is reduced, the stress is increased and vice versa. During the optimization process for such a problem, one reaches a point where it may not be possible to simultaneously improve all such objectives. Hence, the term "optimize" in a multiobjective problem generally refers to a solution point for which there is no way to improve further any objective without worsening at least one other objective. Such a solution point is referred to as a Pareto point or noninferior point. Many such Pareto solutions may exist in a multiobjective optimization problem, and these solutions collectively form a Pareto frontier. Consider the representation of the two-variable two-objective problem shown in Figure 13.11. We see that the feasible domain in the objective space is obtained as a result of a mapping from the variable space. The Pareto frontier corresponds to the line where both f_1 and f_2 are minimized and is the "best" that can be achieved. Trade-offs exist between the solutions in the Pareto set; that is, as one objective is improved in the set, the other is worsened. The final preferred solution to a multiobjective problem is selected from the Pareto set according to the decision maker's preference.

MATLAB has two functions to solve multiobjective problems: `fminimax` and `fgoalattain`. The minimax method solves the problem

$$\min_{x} \max_{f} \{f_1, f_2, \ldots, f_m\} \tag{13.44}$$

$$\begin{aligned}
\text{subject to:}\quad Ax &\le b \quad &&\text{(linear inequality constrainsts)}\\
A_{\text{eq}}x &= b_{\text{eq}} &&\text{(linear equality constraints)}\\
C(x) &\le 0 &&\text{(nonlinear inequality constraints)}\\
C_{\text{eq}}(x) &= 0 &&\text{(nonlinear equality constraints)}\\
lb \le x &\le ub
\end{aligned}$$

where x is the design variable vector; $f_1, f_2, \ldots, f_m$, are the objective functions; the matrix A and the vector b are the coefficients of the linear inequality constraints; the matrix A_{eq} and the vector b_{eq} are the coefficients of the linear equality constraints; C contains the nonlinear inequality constraints; C_{eq} contains the nonlinearequality constraints; and lb and ub specify the lower and upper bounds, respectively, on the design variables x. The `fminimax` method iteratively minimizes the worst-case value of the objective functions subject to the constraints.

The MATLAB function that solves Eq. (13.44) is

```
[xopt, fxopt] = fminimax(@UserFunction, x0, A, b, Aeq, beq, lb, ub, ...
                   @NonLinConstr, options, p1, p2, ... )
```

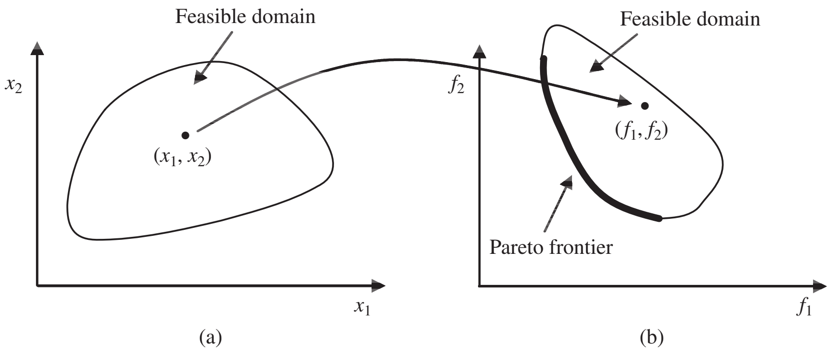


Figure 13.11 Feasible domains in (*a*) the variable space and (*b*) the objective space with its Pareto frontier.

where *xopt* $= x_{\text{opt}}$ is the optimum value of x and *fxopt* $= f_i(x_{\text{opt}})$; **UserFunction** is the name of the function that computes the objective function, the quantity $x0$ is the vector of starting values; the matrix A and the vector b are the coefficients of the linear inequality constraints; the matrix Aeq and the vector beq are the coefficients of the equality constraints; lb and ub are the vectors of the lower and upper bounds on x, respectively; *options* sets the parameters described in `optimset`; and $p1, p2, \ldots$ are the parameters that are passed to **UserFunction** and **NonLinConstr**. **NonLinConstr** is a user-created function that defines the nonlinear inequality and equality constraints as follows:

`function` [C, Ceq] = **NonLinConstr**(x, p1, p2, . . .)

The arguments $p1, p2, \ldots$ of **UserFunction** and **NonLinConstr** *must be identical*, even if only one of the functions uses these values or one of the functions uses only some of the quantities. If lb, ub, and *options* are not specified, then one uses []; similarly for the case when A, b, Aeq, and beq are not specified.

The function `fgoalattain` solves the following multiobjective problem:

$$\begin{aligned}
&\underset{x,\,y}{\text{maximize}}\ \ \gamma \\
&\text{subject to:}\quad f_i(x) - \omega_i\gamma \le (\text{goal})_i \quad && i = 1, \ldots, m \\
&\qquad Ax \le b && \text{(linear inequality constraints)} \\
&\qquad A_{\text{eq}}x = b_{\text{eq}} && \text{(linear equality constraints)} \qquad (13.45) \\
&\qquad C(x) \le 0 && \text{(nonlinear inequality constraints)} \\
&\qquad C_{\text{eq}}(x) = 0 && \text{(nonlinear equality constraints)} \\
&\qquad lb \le x \le ub
\end{aligned}$$

where γ is a scalar variable unrestricted in sign, f_i is the ith objective function, and ω_i and $(\text{goal})_i$ are, respectively, the weighting coefficient and the target values for the ith objective function. The weighting coefficient controls the relative degree of underattainment or overattainment of the goal. The term $\omega_i\gamma$ provides an element of slackness in the formulation. For instance, setting all of the weighting coefficients equal to the initial goals indicates that the same percentage of the underattainment or overattainment of the goals is desired.

The MATLAB function that solves Eq. (13.45) is

x = `fgoalattain`(@**UserFunction**, x0, Goal, Weight, A, b, Aeq, beq, lb, ub, . . .
@**NonLinConstr**, options, p1, p2, . . .)

where the vector *Weight* contains the elements ω_i, the vector *Goal* contains the elements (goal)$_i$, and the remaining quantities are as defined for `fminimax`.

We now demonstrate the use of `fminimax` and `fgoalattain`.

Example 13.18 Vibrating platform

Consider the system shown in Figure 13.12. A motor is mounted on a beam-type platform composed of three layers of materials. It is assumed that the beam is simply supported at both ends. A vibratory disturbance is imparted from the motor to the beam. The design objectives are to minimize the following:

1. The negative of the fundamental natural frequency of the beam, f_1.
2. The cost of the material comprising the beam, f_2

Referring to Figure 13.12, the constraints are an upper bound on the mass of the beam, upper bounds on the thickness of layer 2 and the thickness of the outer layer 3, and upper and lower bounds on the design variables. The five design variables are the beam's length L, its thickness b, and the distances d_1, d_2, and d_3. The mass density ρ, the Young's modulus E, and the cost per unit volume c for the material of each of the three layers are given in Table 13.7 The problem statement is

$$\begin{aligned}
&\text{minimize} \quad f_1(d_1, d_2, d_3, b, L) = -(\pi/2L^2)\sqrt{EI/\mu} \\
&\text{minimize} \quad f_2(d_1, d_2, d_3, b) = 2b\big(c_1 d_1 + c_2(d_2 - d_1) + c_3(d_3 - d_2)\big) \\
&\text{subject to:} \quad \mu L - 2800 \le 0 \qquad \text{beam mass} \\
&\qquad d_1 - d_2 \le 0 \qquad \text{layer thickness} \\
&\qquad -d_1 + d_2 \le 0.15 \qquad \text{layer thickness} \\
&\qquad d_2 - d_3 \le 0 \qquad \text{layer thickness}
\end{aligned}$$

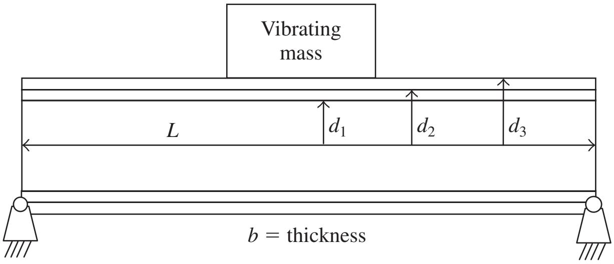


Figure 13.12 Vibrating multilayered simply supported platform.

TABLE 13.7 Material Properties and Cost for the Vibrating Platform

Layer i	ρ_i (kg/m^3)	E_i (N/m^2)	c_i ($/volume)
1	100	$1.6 * 10^9$	500
2	2,770	$70 * 10^9$	1,500
3	7,780	$200 * 10^9$	800

$$
\begin{aligned}
-d_2 + d_3 &\le 0.01 \quad \text{layer thickness}\\
0.05 \le d_1 &\le 0.5\\
0.2 \le d_2 &\le 0.5\\
0.2 \le d_3 &\le 0.5\\
0.35 \le b &\le 0.5\\
3 \le L &\le 6
\end{aligned}
$$

where

$$
\begin{aligned}
EI &= (2b/3)\left(E_1 d_1^3 + E_2(d_2^3 - d_1^3) + E_3(d_3^3 - d_2^3)\right)\\
\mu &= 2b\left(\rho_1 d_1 + \rho_2(d_2 - d_1) + \rho_3(d_3 - d_2)\right)
\end{aligned}
$$

We see that the beam mass is a nonlinear inequality constraint and layer thickness constraints are linear inequality constraints. Thus,

$$
A = \begin{bmatrix} 1 & -1 & 0 & 0 & 0\\ -1 & 1 & 0 & 0 & 0\\ 0 & 1 & -1 & 0 & 0\\ 0 & -1 & 1 & 0 & 0 \end{bmatrix}
$$

$$
b = [0 \quad 0.15 \quad 0 \quad 0.01]'
$$

and since there are no linear and nonlinear equality constraints, $C_{\text{eq}} = A_{\text{eq}} = b_{\text{eq}} = 0$.

In order to have the same order of magnitude of the computed functions, the design objectives are scaled according to

$$
S_v = \frac{R - G}{B - G} \tag{13.46}
$$

where the quantity R refers to the actual value of the function before scaling, G refers to the target (or desired) value of the function, and B refers to the undesirable value of the function. We see from Eq. (13.46) that when R is equal to G, $S_v = 0$ and when R is equal to B, $S_v = 1$.

The following program obtains a solution to this problem and uses three subfunctions: **BeamProperties** in which $[x_1, x_2, x_3, x_4, x_5] = [d_1, d_2, d_3, b, L]$; **VibPlatNLConstr**, which computes the nonlinear inequality constraint; and **VibPlatformObj**, which computes the objective functions.

```
function Example13_18
x0 = [0.3, 0.35, 0.4, 5,0.4];
lb = [0.05, 0.2, 0.2, 0.35, 3];
ub = [0.5, 0.5, 0.6, 0.5, 6];
E = [1.6, 70, 200]*10^9;
Rho = [100, 2770, 7780];
c = [500, 1500, 800];
G = [500, 100];
A = [1 -1 0 0 0; -1 1 0 0 0; 0 1 -1 0 0; 0 -1 1 0 0];
b = [0 0.15 0 0.01]';
for k = 1:5
   B = [100+k*10, 500-k*50];
   [xopt, fxopt] = fminimax(@VibPlatformObj, x0, A, b, [], [], lb, ub, . . .
                        @VibPlatNLConstr, [], E, Rho, c, G, B);
   ff = fxopt.*(B-G)+G;
   f1(k) = ff(1);
   f2(k) = ff(2);
end
[f2sort, indxf2] = sort(f2);
f1sort = f1(indxf2);
plot(-f1sort, f2sort, ' ko-');
xlabel('Negative frequency (Hz)');
ylabel('Cost ($)');

function [EI, mu] = BeamProperties(x, E, Rho)
EI = (2*x(4)/3)*(E(1)*x(1)^3+E(2)*(x(2)^3-x(1)^3)+E(3)*(x(3)^3-x(2)^3));
mu = 2*x(4)*(Rho(1)*x(1)+Rho(2)*(x(2)-x(1))+Rho(3)*(x(3)-x(2)));
```

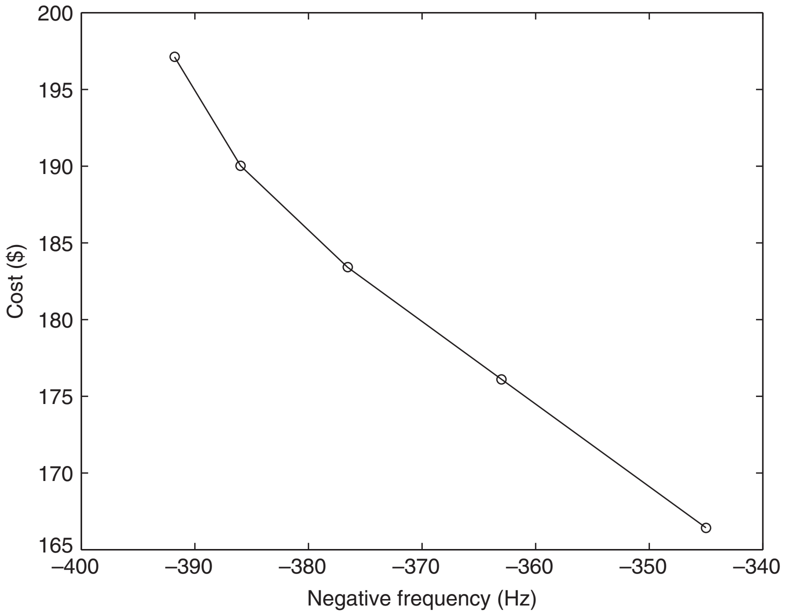


Figure 13.13 Pareto frontier of the vibrating platform.

```
function [C, Ceq] = VibPlatNLConstr(x, E, Rho, c, G, B)
[EI, mu] = BeamProperties(x, E, Rho);
C(1) = mu*x(5)-2800;
Ceq = [];

function f = VibPlatformObj(x, E, Rho, c, G, B)
[EI, mu] = BeamProperties(x, E, Rho);
f1 = pi/(2*x(5)^2)*sqrt(EI/mu);
f(1) = (f1-G(1))/(B(1)-G(1));
f2 = 2*x(4)*(c(1)*x(1)+c(2)*(x(2)-x(1))+c(3)*(x(3)-x(2)));
f(2) = (f2-G(2))/(B(2)-G(2));
```

Upon execution, we obtain the Pareto frontier shown in Figure 13.13.

Example 13.19 Production planning revisited

We now reconsider the production planning problem given in Example 13.2 and introduce a second design objective, which is to maximize (or minimize the negative of) the production units of product A. The objective function and the constraints now become

$$\begin{aligned}\text{minimize}\quad & f_1(x_1, x_2) = -4x_1 - 5x_2\\ & f_2(x_1) = -x_1\\ \text{subject to:}\quad & x_1 + x_2 \le 200\\ & 1.25x_1 + 0.75x_2 \le 200\\ & x_2 \le 150\\ & (x_1, x_2) \ge 0\end{aligned}$$

Thus, for the linearity inequality constraint

$$A = \begin{bmatrix} 1 & 1 \\ 1.25 & 0.75 \\ 0 & 1 \end{bmatrix}$$

$$b = [200 \quad 200 \quad 150]'$$

and $lb = [0, 0]$ and $ub = [\infty, \infty]$. Since there are no equality constraints and no nonlinear constraints, $C_{\text{eq}} = A_{\text{eq}} = b_{\text{eq}} = C = 0$.

The program is as follows:

```
A = [1, 1; 1.25, 0.75; 0, 1];
b = [200, 200, 150]';
goal = [-950, -50];   x0 = [50, 50];
lb = [0, 0];   ub = [inf, inf];
Weight = abs(goal);
options = optimset('GoalsExactAchieve', 2);
ProdPlanObj = inline('[-4*x(1)-5*x(2), -x(1)]', 'x');
[x, fxopt] = fgoalattain(ProdPlanObj, x0, goal, Weight, A, b, [], [], lb, ub, [], options);
disp(['x1 = ' num2str(x(1)) '   x2 = ' num2str(x(2)) '   f1 = ' num2str(fxopt(1)) . . .
            ' f2 = ' num2str(fxopt(2))])
```

where we have set the option *GoalsExactAchieve* to 2, which tells the algorithm to try to satisfy the goals exactly, that is, not over- or underachieve them. If the default value of *GoalsExactAchieve* is used, then the solution becomes a function of $x0$. The execution of the program gives the following Pareto solution:

```
x1 = 50   x2 = 150   f1 = -950   f2 = -50
```

13.7 GENETIC ALGORITHM-BASED OPTIMIZATION

Single-objective and multiobjective genetic algorithm (GA) optimization methods have capabilities that go beyond the classical methods discussed previously in this chapter. The GA methods are applicable to optimization problems in which the objective functions and/or constraint functions are highly nonlinear, nondifferential, or discontinuous. In addition, the variables can be continuous, discrete, or a combination of the two. The genetic algorithm method only requires the values of the objective functions and/or constraint functions and not their derivatives to reach the optimum. In addition, this method can converge to the global optimum, however, the convergence is not guaranteed as it is, for example, in gradient-based methods where there is a convergence proof.

Genetic algorithms are based on the principles of biological evolution or survival of the fittest. Unlike the previous methods in this chapter, which start with a single point and generate a single point at each iteration, the genetic algorithm starts with a random population of points and generates a random population of points (called individuals) at each iteration (generation) of the algorithm. Genetic algorithms use three operators to improve a population and to converge to a final population of solutions. These operators are selection, crossover, and mutation. The selection operator identifies parents in the population. The crossover operator applies to two parents that are randomly selected and matched in order to produce children. The mutation operator is applied to individuals to produce diversity in the population. These operations are performed during each generation and the algorithm stops when the population can no longer be improved or a maximum number of iterations has been reached. The genetic algorithm method is a stochastic optimization approach and, thus, each run of the approach may produce a slightly different solution.

MATLAB has two functions that are based on the genetic algorithm: `ga` and `gamultiobj`. The `ga` function solves the single-objective optimization problems of the form

$$\begin{aligned}
&\underset{x}{\text{minimize}}\ f(x) \\
&\text{subject to:} \quad Ax \le b \quad &&\text{(linear inequality constraints)} \\
&\qquad A_{\text{eq}}x = b_{\text{eq}} \quad &&\text{(linear equality constraints)} \\
&\qquad C(x) \le 0 \quad &&\text{(nonlinear inequality constraints)} \\
&\qquad C_{\text{eq}}(x) = 0 \quad &&\text{(nonlinear equality constraints)} \\
&\qquad lb \le x \le ub
\end{aligned} \tag{13.47}$$

The basic command to solve Eq. (13.47) is

[x, f] = ga(@**UserFunction**, nvars, A, b, Aeq, beq, lb, ub, @**NonLinConstr**, options)

where **UserFunction** is the name of the function that computes the objective function. The quantity *nvars* is the number of variables, the matrix A and the vector b are the coefficients of the linear inequality constraints, the matrix *Aeq* and the vector *beq* are the coefficients of the equality constraints, *lb* and *ub* are the vectors of the lower and upper bounds on x, respectively, and *options* sets the parameters described in `gaoptimset`. The quantity **NonLinConstr** is a function that defines the nonlinear inequality and equality constraints. If *lb*, *ub*, and *options* are not specified, one uses []; similarly for the case when A, b, *Aeq*, and *beq* are not specified.

Function `gamultiobj` solves the multiobjective optimization problems of the form

$$\begin{aligned} &\underset{x}{\text{minimize}}\ \{f_1, f_2, \ldots, f_m\} \\ &\text{subject to: } Ax \le b \qquad \text{(linear inequality constraints)} \\ &\qquad A_{\text{eq}}x = b_{\text{eq}} \qquad \text{(linear equality constraints)} \\ &\qquad lb \le x \le ub \end{aligned} \tag{13.48}$$

where x is the design variable vector, $f_1, f_2, \ldots, f_m$ are the objective functions, the matrix A and the vector b are the coefficients of the linear inequality constraints, the matrix A_{eq} and the vector b_{eq} are the coefficients of the linear equality constraints, and *lb* and *ub* specify the lower and upper bounds, respectively, on the variables x. The method used by `gamultiobj` is based on the genetic algorithm and iteratively evolves a population of individuals toward a Pareto solution set.

The MATLAB function that solves Eq. (13.48) is

[x, f] = gamultiobj(@**UserFunction**, nvars, A, b, Aeq, beq, lb, ub, options)

where x is the obtained Pareto solutions and f is the corresponding objective function values, **UserFunction** is the name of the function that computes the objective function, the quantity *nvars* is the number of variables, the matrix A and the vector b are the coefficients of the linear inequality constraints, the matrix *Aeq* and the vector *beq* are the coefficients of the equality constraints, *lb* and *ub* are the vectors of the lower and upper bounds on x, respectively, and *options* sets the parameters described in `gaoptimset`.

In both `ga` and `gamultiobj`, the variables can be continuous or binary (0 or 1). However, in the current version of the MATLAB,[14] the linear and nonlinear constraints are not satisfied when the option 'PopulationType' is set to 'bitstring' (binary). To resolve this, constraints can be added to the objective function in the form of an infeasibility constraint. That is, when the constraints are violated, a large penalty for

[14] *Genetic Algorithm and Direct Search Toolbox™ 2, User's Guide*, 2009.

their violation is added to the objective function, otherwise, the penalty is set equal to zero. This penalty approach forces an infeasible point to eventually move into the feasible domain. The scalar penalty parameter P is a large number that is applied to the constraint infeasibility as

$$\underset{x}{\text{minimize}}\; f + P[\max(0, (Ax - b)) + \max(0, C(x)) + (A_{\text{eq}}x - b_{\text{eq}})^2 + C_{\text{eq}}^2]$$

We now demonstrate the use of `ga` and `gamultiobj`. Example 13.20 considers a single objective with binary variables, Example 13.21 considers a single objective with continuous variables, and Example 13.22 considers two objectives with continuous variables. The last example, Example 13.23, has two objectives with both continuous and discrete variables.

Example 13.20 Loading of a knapsack revisited: single objective with binary variables

Example 13.4 is now solved with `ga`. The seven binary variables are determined by minimizing the following:

$$\text{minimize } f(x) = -(12x_1 + 12x_2 + 9x_3 + 15x_4 + 90x_5 + 26x_6 + 112x_7) + P\max[0, (3x_1 + 4x_2 + 3x_3 + 3x_4 + 15x_5 + 13x_6 + 16x_7 - 35)]$$

$$x_i = 0 \text{ or } 1 \quad i = 1, \ldots, 7$$

The parameter P is set to a large number, in this case, we have selected it to be 100. The program is as follows:

```
function Example13_20
nvars = 7;
options = gaoptimset;
options = gaoptimset(options, 'PopulationType', 'bitstring');
options = gaoptimset(options, 'PopulationSize', 100);
options = gaoptimset(options, 'PlotFcns', {@gaplotbestf, @gaplotbestindiv});
[x, f] = ga(@KnapsackObj, nvars, [], [], [], [], [], [], [], options);
disp([repmat('x(', 7,1) num2str((1:7)') repmat(') = ', 7,1) int2str(x')])
disp(['Total value of placed objects = $' num2str(-f, '%6.2f')])

function f = KnapsackObj(x)
P = 100;
c = 3*x(1)+4*x(2)+3*x(3)+3*x(4)+15*x(5)+13*x(6)+16*x(7)-35;
f = -(12*x(1)+12*x(2)+9*x(3)+15*x(4)+90*x(5)+26*x(6)+112*x(7))+P*max(0,c);
```

The execution of the program produces Figure 13.14. The upper pane shows the iteration history of the best and mean values of the fitness function in the population for each generation, and the lower pane shows the best values of the variables in the final population when the algorithm stopped. As shown in the upper figure, the mean value of the objective function (or fitness) for individuals in the population approaches the best value in the population, which is an indication of uniformity of the objective function values for individuals in the final population. The lower figure shows the best

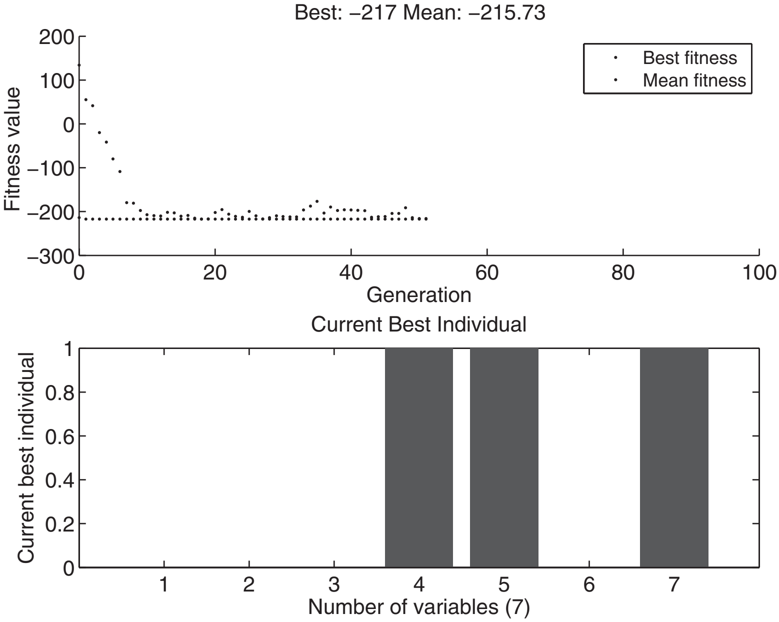


Figure 13.14 Iteration history for the best and mean fitness values in the population per generation (upper figure) and the best values for the variables in the final population (lower figure) for Example 13.20.

value of the binary variables in the final population when the algorithm stopped. In addition, the following information is displayed to the command window:

```
x(1) = 0
x(2) = 0
x(3) = 0
x(4) = 1
x(5) = 1
x(6) = 0
x(7) = 1
Total value of placed objects = $217.00
```

We see that these results agree with those obtained in Example 13.4.

Example 13.21 Two-bar truss revisited: single objective with continuous variables

Example 13.13 is now solved with ga but with the finite upper bounds indicated in the program below. In addition, to improve the numerical accuracy of the solution, the variables x_1 and x_2 are scaled by 100 and 10,000, respectively. The program is as follows:

```
function Example13_21
global sigma
```

```
nvars = 3;   lb = [100, 0, 0];   ub = [300, 20, 20];   sigma = 1e5;
options = gaoptimset;
options = gaoptimset(options, 'PopulationSize', 60);
options = gaoptimset(options, 'Generations', 200);
options = gaoptimset(options, 'Display', 'final');
options = gaoptimset(options, 'PlotFcns', {@gaplotbestindiv, ...
          @gaplotscorediversity});
[x, V] = ga(@TwoBarTrussObj, nvars, [], [], [], [], lb, ub, ...
          @TwoBarTrussNonLinCon, options);
disp(['y = ' num2str(x(1)) ' cm x1 = ' num2str(x(2)) ' cm^2 x2= ' ...
            num2str(x(3)) ' cm^2 Volume = ' num2str(V*1e6) ' cm^3'])

function f = TwoBarTrussObj(x)
x(1) = x(1)/1e2;   x(2:3) = x(2:3)/1e4;   % Values scaled
f = x(2)*sqrt(16+x(1)^2)+x(3)*sqrt(1+x(1)^2);

function [C, Ceq] = TwoBarTrussNonLinCon(x)
global sigma
x(1) = x(1)/1e2;   x(2:3) = x(2:3)/1e4;   % Values scaled
C(1) = 20*sqrt(16+x(1)^2)-sigma*x(1)*x(2);
C(2) = 80*sqrt(1+x(1)^2)-sigma*x(1)*x(3);
Ceq = [];
```

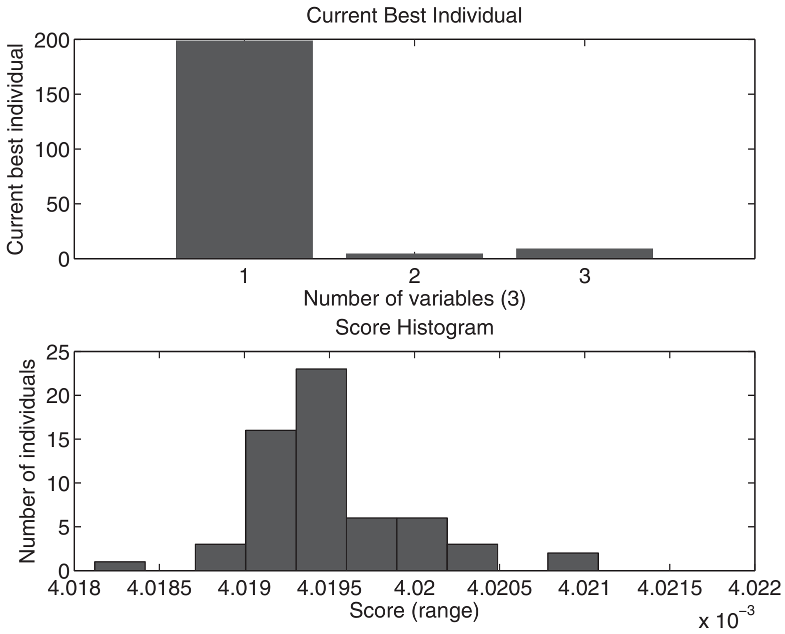


Figure 13.15 Current best individuals (upper figure) and the histogram of the score in the final population (lower figure) for Example 13.21.

The execution of the program produces Figure 13.15, where the upper pane shows the value of the best individual (solution point) in the final population and the lower pane shows the histogram of the objective function values in the final population. This lower pane gives a measure of diversity of points in the final population. In addition, the following information is displayed to the command window:

```
y = 196.0471 cm    x1 = 4.5597 cm^2    x2= 9.0389 cm^2    Volume = 4020.4176 cm^3
```

We see that these results are in fairly good agreement with those obtained in Example 13.13. As mentioned previously, due to the stochastic nature of the genetic algorithm, each run of ga produces somewhat different results. In general, it is a good practice to run ga several times and select the solution for which the objective function, *Volume* in this case, has the smallest value.

Example 13.22 Two-bar truss revisited: multiobjectives with continuous variables

Example 13.13 is now converted into a multiobjective form and solved with gamultiobj. The Pareto optimum solutions are determined by solving

$$\begin{aligned}
&\text{minimize } \{f_1, f_2\}\\
&\text{where} \quad f_1 = x_1\sqrt{16 + y^2} + x_2\sqrt{1 + y^2}\\
&\qquad\qquad f_2 = 20\sqrt{16 + y^2}/(\sigma y x_1) - 1\\
&\text{subject to:} \quad 80\sqrt{1 + y^2}/(\sigma y x_2) - 1 \le 0\\
&\qquad\qquad 1 \le y \le 3\\
&\qquad\qquad 0.05 \le (x_1, x_2) \le 0.25
\end{aligned}$$

The program is as follows:

```
function Example13_22
global P sigma
P = 100;    sigma = 1e5;
nvars = 3;    lb = [0.05, 0.05, 1];    ub = [0.25, 0.25, 3];
options = gaoptimset;
options = gaoptimset(options, 'Vectorized', 'on');
[x, V] = gamultiobj(@TwoBarTrussTwoObj, nvars, [], [], [], [], lb, ub, options);
[f , C] = TwoBarTrussTwoObj(x);
disp(C)
plot(V(:,1), V(:,2), 'k*')
xlabel('f_1')
ylabel('f_2')
grid on
function [f, C] = TwoBarTrussTwoObj(x)
global P sigma
f1(:,1) = x(:,1).*sqrt(16+x(:,3).^2)+x(:,2).*sqrt(1+x(:,3).^2);
f1(:,2) = 20*sqrt(16+x(:,3).^2)./(sigma*x(:,1).*x(:,3))-1;
C (:,1) = 80*sqrt(1+x(:,3).^2)./(sigma*x(:,2).*x(:,3))-1;
```

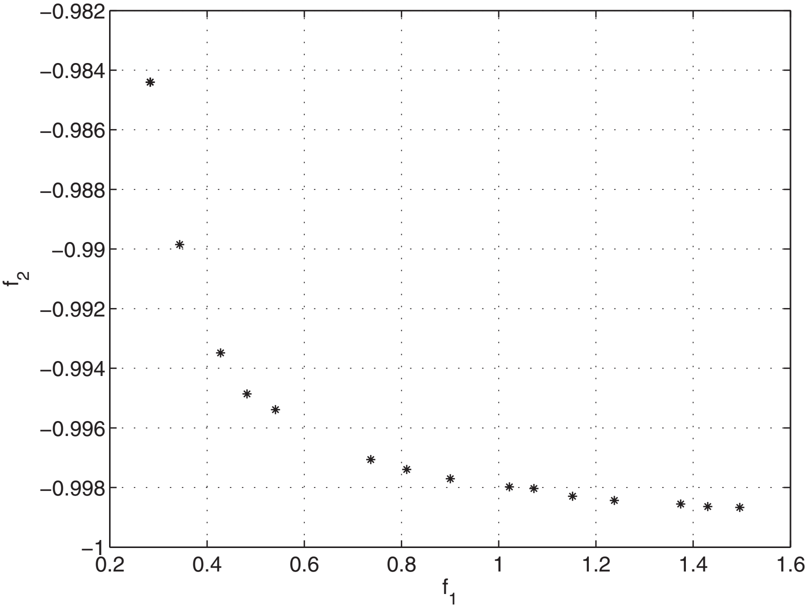


Figure 13.16 Pareto frontier for Example 13.22.

```
f(:,1) = f1(:,1)+P*max(0,C(:,1));
f(:,2) = f1(:,2)+P*max(0,C(:,1));
```

The execution of the program produces Figure 13.16 where the values of the Pareto optimum solutions for objective f_2 versus objective f_1 are shown. It is good practice for this type of formulation in which a penalty parameter is introduced to verify that the constraints have been satisfied. In the present case, the quantity C as given in **TwoBarTrussTwoObj** is less than zero. For the results shown, it was found that $C < 0$ for each pair of objective functions appearing in Figure 13.16.

Example 13.23 Two-bar truss revisited: single objective with continuous and discrete variables

Example 13.22 is now assumed to have mixed continuous–discrete variables given by

$$\begin{aligned}
\text{minimize} \quad & f = x_1\sqrt{16 + y^2} + x_2\sqrt{1 + y^2} \\
\text{subject to:} \quad & 20\sqrt{16 + y^2}/(\sigma y x_1) - 1 \le 0 \\
& 80\sqrt{1 + y^2}/(\sigma y x_2) - 1 \le 0 \\
& 0 \le (x_1, x_2) \le 0.25 \\
& y \in \{1, 1.3, 1.6, 1.9, 2.2, 2.5, 2.8\}
\end{aligned}$$

The variables x_1 and x_2 are assumed to be continuous in their range and are required to have an accuracy of seven significant decimal digits. Variable y is assumed to be discrete and can have any of the following seven values stated above. In general, any continuous or integer variable x that is defined in a closed interval $lb \leq x \leq ub$ can be expressed through binary variables denoted by s as[15]

$$x = lb + D(s) \tag{13.49}$$

where

$$D(s) = \sum_{i=0}^{N-1} 2^i s_i \qquad s_i = 0 \text{ or } 1$$

and N is the minimum number of binary variables needed. To determine N, we note that $2^N > (ub - lb)10^d$ and, therefore, one obtains

$$N = 1 + \texttt{floor}\left[\frac{d + \log_{10}(ub - lb)}{\log_{10} 2}\right]$$

where d is the number of significant decimal places required for x and the `floor` function truncates its real argument to an integer value in the direction of $-\infty$ (recall Table 1.8). Note that this approach will increase the number of variables significantly and may not be practical when the range is too large. For this example, we have $d = 7$, $lb = 0$, and $ub = 0.25$ and we obtain from the above equation $N = 22$. This means that for each real variable x_1 and x_2, we need twenty-two binary variables to represent them. For the discrete variable y, we have seven choices; that means $2^N > 7$, and thus $N = 3$. Therefore, we require a total of forty-seven (= 22 + 22 + 3) binary variables in a binary string format (bitstring) with the first twenty-two bits for x_1, the next twenty-two bits for x_2, and the last three bits for y. The actual value of the variable can then be obtained from[16]

$$x = lb + \frac{ub - lb}{2^N - 1} D(s) \tag{13.50}$$

where the binary string s of length N is given by $(s_{N-1}s_{N-2} \ldots s_2 s_1 s_0)$. For example, when s is represented by (00010110), then

$$D(s) = 0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 + 1 \times 2^4 + 0 \times 2^5 + 0 \times 2^6 + 0 \times 2^7 = 22$$

In the program below, **GetRealFromBinary** maps the value of the forty-four binary variables representing x_1 and x_2 from Eq. (13.49) to the corresponding real (or actual) values for x_1 and x_2 using Eq. (13.50). The function **GetDiscreteFromBinary** maps the value of the three binary variables representing y to one of its seven discrete (or actual) alternatives. The objective and constraint functions are then calculated with the actual values of these three variables. The program is as follows:

```
function Example13_23
nvars = 47;   PopulationSize = 100;   Generations = 100;
options = gaoptimset;
```

[15] C. A. Floudas, *Nonlinear and Mixed-Integer Optimization*, Oxford University Press, New York, 1995, p. 111.

[16] K. Deb, *Multi-Objective Evolutionary Optimization Using Evolutionary Algorithms*, John Wiley & Sons, New York, 2001, p. 82.

```
options = gaoptimset(options, 'PopulationType' , 'bitstring');
options = gaoptimset(options, 'PopulationSize' , PopulationSize);
options = gaoptimset(options, 'Generations' , Generations);
options = gaoptimset(options, 'PlotFcns', {@gaplotbestindiv, ...
            @gaplotscorediversity});
[x, fval] = ga(@TwoBarTrussTwoObjGa, nvars, [], [], [], [], [], [], [], options);
finalx1 = GetRealFromBinary(x, 1, 22, 0, 0.25);
finalx2 = GetRealFromBinary(x, 23, 44, 0, 0.25);
finaly = GetDiscreteFromBinary(x, 45, 47);
disp(['x1 = ' num2str(finalx1*1e4) ' cm^2 x2 = ' num2str(finalx2*1e4) . . .
            ' cm^2 y = ' num2str(finaly*100) ' cm'])
disp(['V = ' num2str(fval*1e6)])

function f = TwoBarTrussTwoObjGa(x)
P = 100;   sigma = 1e5;
var(1) = GetRealFromBinary(x, 1, 22, 0, 0.25);
var(2) = GetRealFromBinary(x, 23,44, 0, 0.25);
var(3) = GetDiscreteFromBinary(x, 45,47);
f1 = var(1)*sqrt(16+var(3)^2)+var(2)*sqrt(1+var(3)^2);
C1 = 80*sqrt(1+var(3)^2)/(sigma*var(2)*var(3))-1;
C2 = 20*sqrt(16+var(3)^2)/(sigma*var(1)*var(3))-1;
f = f1+P*(max(0,C1)+max(0,C2));

function x = GetRealFromBinary(y, startbit, endbit, lowerlimit, ...
                        upperlimit)
temp = y(startbit:endbit);
ii = length(temp);   newint =0;
for i = 1:ii
  newint = newint + temp(i)*2^(i-1);
end
maxbitvalue = 2^(length(temp))-1;
x = lowerlimit+(upperlimit-lowerlimit)/maxbitvalue*newint;

function x = GetDiscreteFromBinary(y, startbit, endbit)
choice = [1, 1.3, 1.6, 1.9, 2.2, 2.5, 2.8];
temp = y(startbit:endbit);   ii = length(temp);
newint = 0;
for i = 1:ii
  newint = newint+temp(i)*2^(i-1);
end
if newint == 0
  x = choice(1);
else
  x = choice(newint);
end
```

The execution of the program produces Figure 13.17 and displays the following actual values of the variables x_i and the corresponding objective function value to the command window:

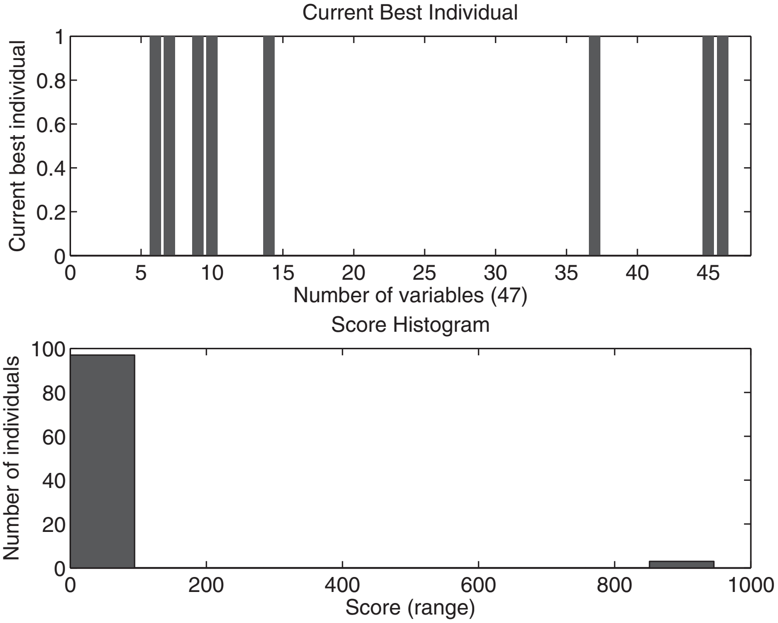


Figure 13.17 Current best individuals (upper figure) and the histogram of the score in the final population (lower figure) for Example 13.23.

```
x1 = 5.3978 cm^2    x2 = 9.7656 cm^2    y = 160 cm
V = 4168.0174
```

The upper pane shows the iteration history of the best and mean values of the objective function for all individuals in the population for each generation (or iteration). As shown in the upper pane, the mean value of the fitness (or objective function plus penalty) of the individuals in the population approaches the best value in the population, which is an indication of uniformity of the individuals in terms of their fitness value (or objective function value since the penalty is zero) in the final population. The lower pane shows the best value of the variables in the final population when the algorithm stopped. This solution is expected to be somewhat worse than that obtained from Example 13.13 because *y* is restricted to being discrete.

13.8 SUMMARY OF FUNCTIONS INTRODUCED IN CHAPTER 13

A summary of the Optimization toolbox functions and the Genetic Algorithm and Direct Search toolbox functions introduced in Chapter 13 are presented in Table 13.8.

TABLE 13.8 MATLAB Functions from Optimization Toolbox Introduced in Chapter 13

MATLAB function	Description
	Optimization Toolbox
`bintprog`	Binary integer programming solver
`fgoalattain`	Multiobjective goal attainment solver
`fminbnd`	Minimum of a function of one variable on a fixed interval
`fmincon`	Minimum of a constrained nonlinear multivariable function
`fminimax`	Minimax solver
`fminsearch`	Minimum of an unconstrained multivariable function
`fminunc`	Minimum of an unconstrained multivariable function
`fseminf`	Minimum of a semi-infinitely constrained multivariable nonlinear function
`linprog`	Linear programming solver
`lsqcurvefit`	Nonlinear least squares curve fitting solver
`lsqnonlin`	Nonlinear least squares solver
`quadprog`	Quadratic programming solver
	Genetic Algorithm and Direct Search Toolbox
`ga`	Implements the genetic algorithm to find minimum of a single-objective function
`gamultiobj`	Implements the genetic algorithm to find minima of multiple objective functions

EXERCISES

Sections 13.1.2 and 13.4.1

13.1 Consider a two-pair bar mechanism, shown in Figure 13.18 in which links OA and AC have the same length $L_1 = 0.5$ m and links CB and BD have the same length $L_2 = 0.3$ m. Joint O is a fixed joint, and joints C and D are pin connections on two slides that can move without friction along a horizontal line. The mechanism is subjected

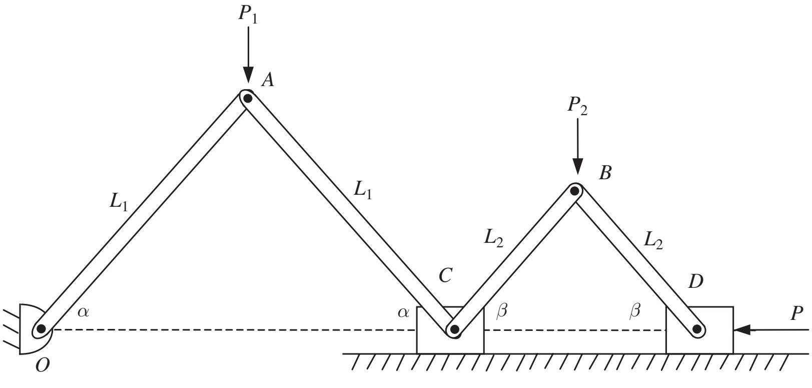


Figure 13.18 Two-pair bar mechanism of Exercise 13.1.

to three external forces. They are the vertical forces $P_1 = 3$ kN and $P_2 = 1$ kN at joints A and B, respectively, and a horizontal force $P = 3$ kN at joint D. We assume that before the loads are applied, the bars are lined up (stretched out) along the horizontal line. We also assume that the weight of the mechanism is negligible. The equilibrium of the mechanism under the applied loads is obtained by minimizing the potential energy function

$$\text{minmize PE} = -P_1 L_1 \sin\alpha - P_2 L_2 \sin\beta + 2P[L_1(1 - \cos\alpha) + L_2(1 - \cos\beta)]$$

a. Create Figure 13.19.

b. Use `fminunc` with an initial point $(\alpha, \beta) = (1, 1)$ radians to verify the graphical solution (Answer: $\alpha = 0.4636$ rad, $\beta = 0.1651$ rad, and PE $= -0.3789$).

13.2 Consider the cross section of a water canal[17] shown in Figure 13.20. The water canal has a fixed cross-sectional area and is to be designed so that its discharge flow rate is maximized. The design variables are the height h, the width of base c, and the side angle θ. It can be shown that the flow rate is proportional to the inverse of the wetted perimeter p, which is given by

$$p = c + 2h/\sin\theta$$

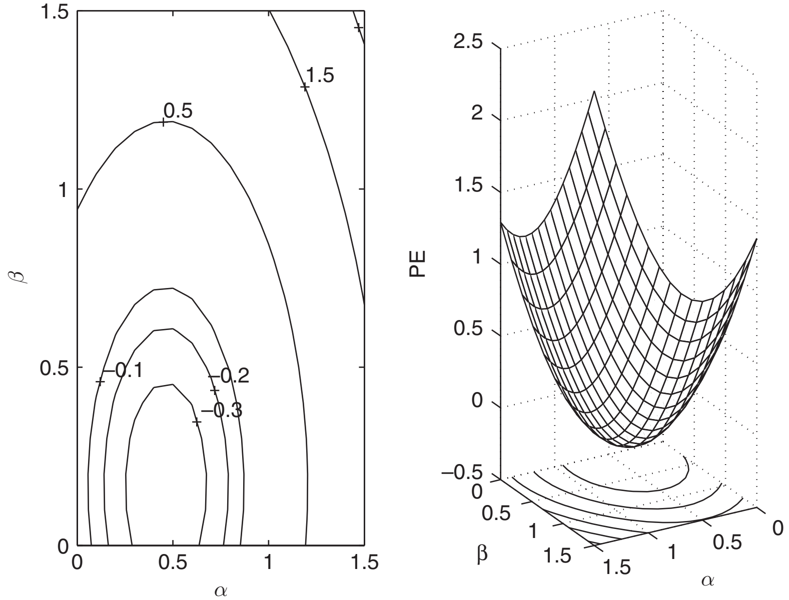


Figure 13.19 Contour and surface plots of Exercise 13.1.

[17] P. Y. Papalambros and D. J. Wilde, *Principles of Optimal Design: Modeling and Computation*, Cambridge University Press, New York, 1988, p. 151.

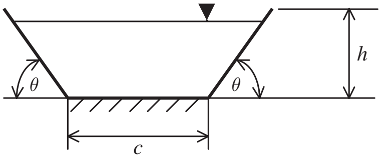


Figure 13.20 Water canal of Exercise 13.2.

and the cross-sectional area A, which is given by

$$A = ch + h^2 \cot\theta$$

Then, the function to be minimized is

$$\text{minmize}\ \frac{1}{p} = \left(\frac{A}{h} - h\cot\theta + \frac{2h}{\sin\theta}\right)^{-1}$$

If $A = 9.3\ \text{m}^2$, then

a. Create Figure 13.21.

b. Use `fminsearch` to validate the graphical solution with an initial point $(h, \theta) = (1, 1)$ rad (Answer: $h = 2.3172$ and $\theta = 60°$).

13.3 Figure 13.22 shows an unloaded and loaded two-spring system. After the load F is applied at point A, the system is deformed until it is in equilibrium at point B, that

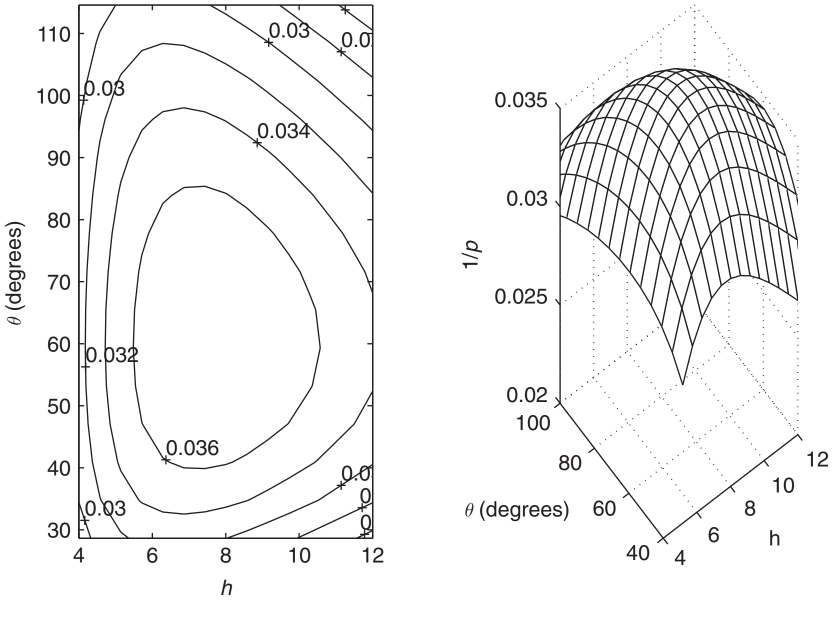


Figure 13.21 Contour and surface plots of Exercise 13.2.

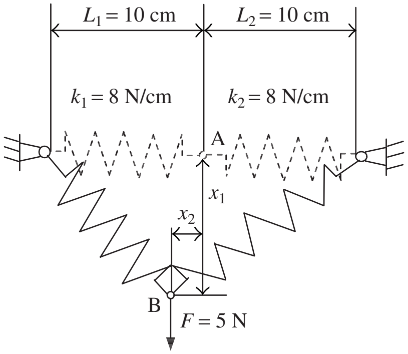


Figure 13.22 Two-spring system of Exercise 13.3.

is, when the potential energy of the system is minimum. The potential energy function is given by Eq. (13.3) with $F_2 = 0$. Find the location (x_1, x_2) of joint B in two ways:

a. Create the contour and surface plots shown in Figure 13.23.

b. Use `fminsearch` with an initial point $(x_1, x_2) = (1, 1)$ (Answers: PE $= -15.2802$, $x_1 = 4.1289$, and $x_2 = 0$).

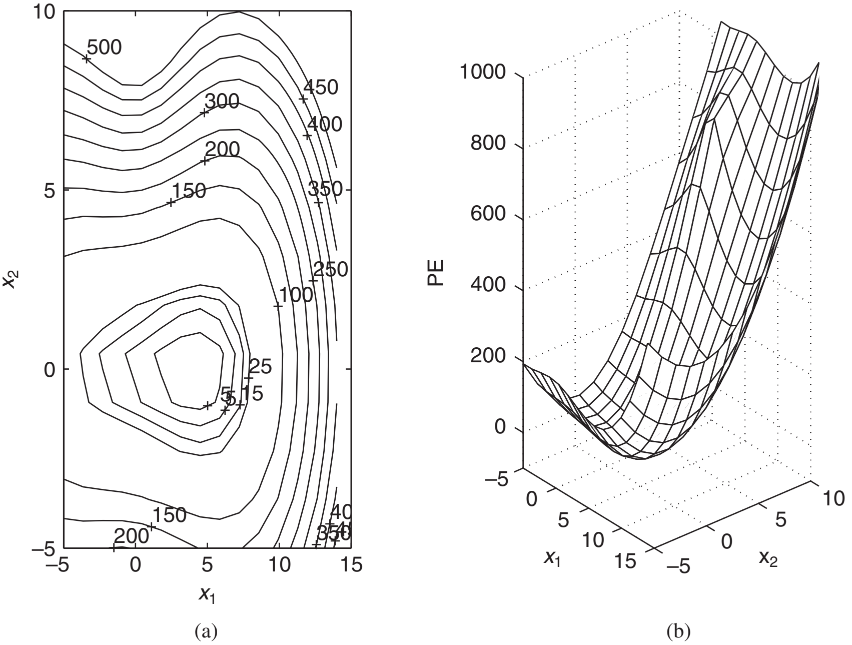


Figure 13.23 (a) Contour and (b) surface plots of the PE function for the two-spring system shown in Figure 13.22.

13.4 The average total production time per workpiece for a machining operation is given by[18]

$$T = t_m + \frac{t_m t_c}{T_l} + t_{\text{aux}}$$

where t_m is the cutting time, T_l is the tool life, t_c is the time it takes to change the tool, and t_{aux} is the auxiliary time. For a turning operation, the cutting time is obtained by

$$t_m = \frac{\pi DL}{1000Vf}$$

where the D is the diameter of the machined surface, L is the length of the workpiece to be machined, V is the cutting surface speed, and f is the feed rate of the cutting tool. The influence of the cutting speed, the feed rate, and the depth of cut d on the tool life are estimated from the extended Taylor equation, which can be used to obtain T_l as

$$T_l = \left(\frac{K_t}{Vf^a d^b}\right)^{1/n}$$

where $n = 0.17$, $a = 0.77$, $b = 0.37$, and $K_t = 200$ are empirical constants. If V and f are the design variables, then the optimization problem is

$$\text{minimize } T(V, f)$$
$$\text{subject to: } f \leq 2$$

Assume that $t_c = 7$, $t_{\text{aux}} = 3$, $d = 0.3$, $D = 100$ mm, and $L = 500$ mm. For these parameter values,

a. Create contour and surface plots for the system.

b. Use `fmincon` with an initial point of $(V, f) = (10, 1)$ to obtain the optimum solutions (Answers: $T = 3.942$, $V = 100.46$, and $f = 2.000$).

13.5 Figure 13.24 shows two frictionless rigid carts A and B connected by three linear elastic springs having spring constants $k_1 = 5$ N/m, $k_2 = 10$ N/m, and $k_3 = 8$ N/m.[19] The

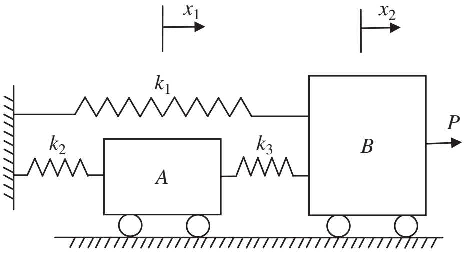
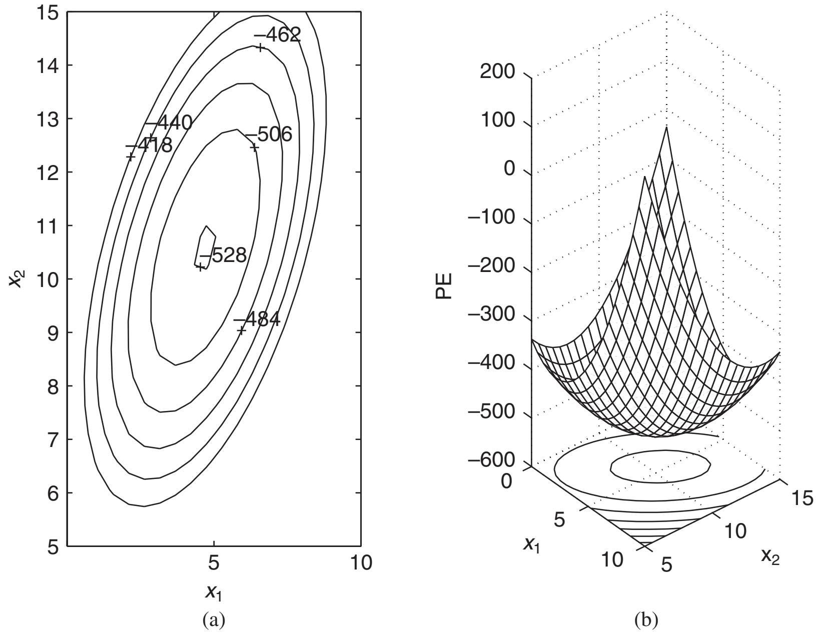


Figure 13.24 Spring-mass system of Exercise 13.5.

[18] D. A. Stephenson and J. S. Agapiou, *Metal Cutting Theory and Practice*, Marcel Dekker, New York, 1997.

[19] S. S. Rao, *Engineering Optimization, Theory and Practice*, 3rd ed., John Wiley & Sons, New York, 1996.



(a) (b)

Figure 13.25 (a) Contour and (b) surface plots of the PE function for the two-spring system shown in Figure 13.24.

springs are in their undeformed positions when the applied force is zero. Use the following unconstrained optimization problem for the potential energy function:

$$\text{minimize PE} = 0.5k_2x_1^2 + 0.5k_3(x_2 - x_1)^2 + 0.5k_1x_2^2 - Px_2$$

a. Obtain the contour and surface plots as shown in Figure 13.25.

b. Obtain the optimized displacements x_1 and x_2 by using `fminunc` with an initial point $(x_1, x_2) = (1, 1)$ and with $P = 100\,\text{N}$ (Answers: PE $= 529.41, x_1 = 4.7059$, and $x_2 = 10.5882$).

13.6 Three carts, interconnected by springs and initially at an unstressed equilibrium state, are subjected to the loads P_1, P_2, and P_3 as shown in Figure 13.26.[20] The displacements of the carts from their original equilibrium position ($x_i = 0$, for all i) are sought by minimizing the potential energy of the system

$$\text{PE} = 0.5X^TKX - X^TP$$

where

$$K = \begin{bmatrix} k_1 + k_3 + k_4 & -k_3 & -k_4 \\ -k_3 & k_2 + k_3 + k_5 & -k_5 \\ -k_4 & -k_5 & k_4 + k_5 + k_6 \end{bmatrix}$$

[20] *Ibid.,* p. 420.

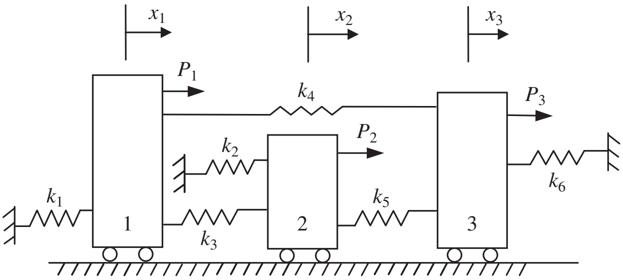


Figure 13.26 Spring-mass system of Exercise 13.6.

$$P = [P_1 \quad P_2 \quad P_3]^T$$
$$X = [x_1 \quad x_2 \quad x_3]^T$$

The parameter values for the system are as follows:

$k_1 = 4500$ N/m	$k_4 = 2250$ N/m	$P_1 = 1100$ N
$k_2 = 1650$ N/m	$k_5 = 550$ N/m	$P_2 = 1800$ N
$k_3 = 1100$ N/m	$k_6 = 9300$ N/m	$P_3 = 3300$ N

Using these values, find the equilibrium position of the carts by using `fminunc` with the initial point: $(x_1, x_2, x_3) = (0, 0, 0)$ (Answers: $x_1 = 0.348$, $x_2 = 0.723$, and $x_3 = 0.370$).

13.7 Figure 13.27 shows a spring-weight system in its undeformed position with no supporting weights, and in its deformed position with supporting weights at the joints between the springs.[21] The stiffness of the spring i is k_i and is defined by

$$k_i = 450 + 225(M/3 - i)^2 \text{ N/m} \qquad i = 1, \ldots, 6$$

where $M = 5$ is the number of weights. Weight W_j is defined by

$$W_j = 60j \text{ N} \qquad j = 1, \ldots ,5$$

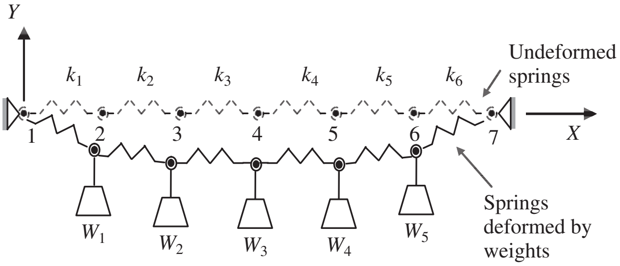


Figure 13.27 Spring-weight system of Exercise 13.7.

[21] Vanderplaats, *Numerical Optimization,* p. 94.

The length of each spring before the weights are applied is $L_i = 7.5$ m. The coordinates of the spring joints (points 2–6) are represented by ten design variables: (X_i, Y_i), $i = 2, \ldots, 6$ To solve for the equilibrium, the following PE function is minimized:

$$\text{PE} = 0.5\sum_{i=1}^{6} K_i \Delta L_i^2 + \sum_{j=1}^{5} W_j Y_{j+1}$$

where

$$\Delta L_i = \sqrt{(X_{i+1} - X_i)^2 + (Y_{i+1} - Y_i)^2} - L_i$$

Determine the equilibrium positions by using `fminunc`, that is, the joint positions for the deformed system shown in Figure 13.23. Use the initial point $(X_2, X_3, X_4, X_5, X_6, Y_2, Y_3, Y_4, Y_5, Y_6) = (7.5, 15, 22.5, 30, 37.5, 0, 0, 0, 0, 0)$ (Answer: $X_2 = 7.9168$, $X_3 = 16.2538$, $X_4 = 24.3557$, $X_5 = 32.223$, $X_6 = 39.303$, $Y_2 = -3.8577$, $Y_3 = -7.1458$, $Y_4 = -8.8364$, $Y_5 = -8.2859$, $Y_6 = -5.1603$).

Section 13.4.2 and 13.4.3

13.8 The buckling load F for a tubular column shown in Figure 13.28 may be expressed as the following equation with unknown constants a and b:

$$F = \frac{\pi^a E R^b t^{4-b}}{4L^2}$$

where E is the modulus of elasticity, R is the mean radius, t is the thickness, and L is the length of the column. It is assumed that the exact relation for the buckling is unknown and the constants will be determined through curve fitting of experimental data. To do this, an experiment is conducted where columns of different sizes, with $E = 1{,}724$ N/mm^2, $L = 127$ mm, $t = 25.4$ mm, are loaded until they buckle. The loads at which the buckling occurs for different values of R are recorded in Table 13.9. Determine a and b by using `lsqcurvefit` with an initial point $(a, b) = (1, 1)$ and create Figure 13.29 where the experimental data and the fitted curve are shown (Answer: $a = -3.3182$ and $b = 3.1588$).

13.9 Suppose that one of the ingredients in a pharmaceutical drug is to be kept at a certain percentage of the drug volume and that this percentage decreases over time. In the

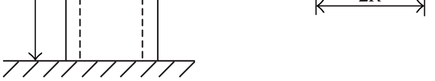


Figure 13.28 Column geometry for Exercise 13.8.

TABLE 13.9 Values for Radius R and Load F of Exercise 13.8

Experiment	R (mm)	F (kN)
1	27.9	385
2	43.2	536
3	53.3	2,317
4	73.7	7,820
5	99.1	18,230

weeks before the drug reaches the market, a decline in the percentage may occur. Since many other uncontrolled factors may also arise, theoretical calculations are not reliable for making a prediction of the decrease of this ingredient at later times. To assist the management in deciding when the drug should be replaced after it has been stored in a warehouse for an extended period of time, cartons of the drug are analyzed to measure their ingredient content as a function of time. The results of one such measurement is shown in Table 13.10. It is postulated that a nonlinear model of the form

$$Y = a + (0.51 - a)e^{-b(t-8)}$$

accounts for the variation observed in the data. Estimate the parameters a and b of the nonlinear model by using `lsqcurvefit` with an initial point $(a, b) = (1, 1)$ (Answer: $a = 0.3918$ and $b = 0.1394$).

13.10 Suppose that you are observing a vehicle at a stop sign. The vehicle stops, and then it rapidly accelerates past five houses whose distances from the stop sign are known. As

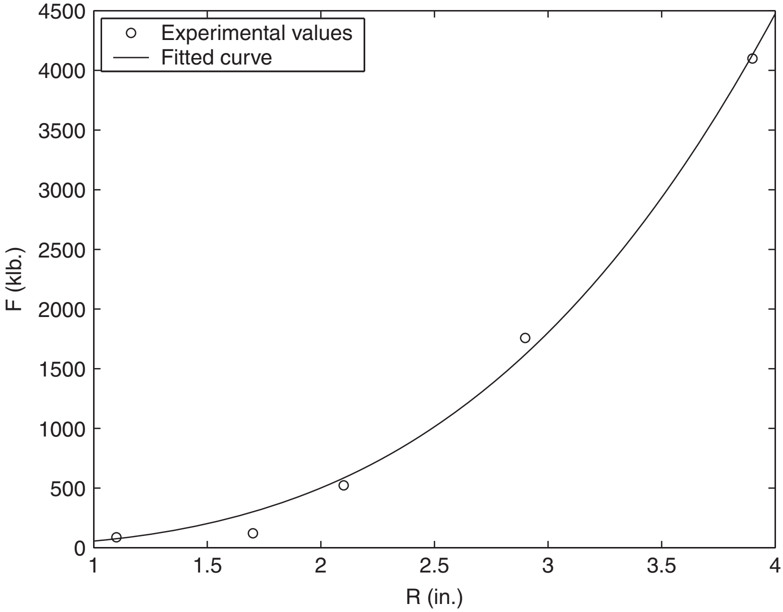


Figure 13.29 Tubular column results of Exercise 13.8.

TABLE 13.10 Data for Exercise 13.9

Length of time since production (weeks), t	Amount of ingredient, Y	Length of time since production (weeks), t	Amount of ingredient, Y
7	0.488	25	0.405
9	0.473	27	0.403
11	0.448	29	0.391
13	0.435	31	0.403
15	0.431	33	0.398
17	0.453	35	0.393
19	0.421	37	0.398
21	0.405	39	0.388
23	0.405	41	0.388

the vehicle starts from rest, you time the vehicle with your stopwatch as it passes each house. The data collected from such observations are shown in Table 13.11. The acceleration as a function of time is given by

$$a(t) = Ct^2 + Dt + 2a_0$$

where t is travel time, a_0 is initial acceleration, and C and D are constants. Hence, the position of the vehicle as a function of time is given by

$$x(t) = At^4 + Bt^3 + a_0t^2$$

Estimate the equation for the car's velocity $v(t)$ as a function of time using `lsqnonlin` with an initial point $(A, B) = (0, 0)$. Assume that the initial position, velocity, and acceleration of the vehicle are, respectively, $x_0 = 0$ m, $v_0 = 0$ m/s, and $a_0 = 0.61$ m/s^2. Determine the error (the second output of `lsqcurvefit`) of the least square estimation (Answer: $v(t) = -0.032693t^3 + 0.2515t^2 + 1.22t$ and error $= 5.6533$).

13.11 A coordinate measuring machine (CMM) is used to determine the actual dimensions of features of a manufactured part by using a sensor that indicates when the sensor has initially contacted a surface of the part. When the sensor indicates that a part's surface has been contacted, the CMM records sensor's coordinates. When the feature is a hole in the part, the N recorded values of different locations of the surface of the hole are used in the following relation to determine the location of the center of the hole and its radius:

$$r_n = \sqrt{(x_c - x_n)^2 + (y_c - y_n)^2} \qquad n = 1, 2, \ldots, N$$

TABLE 13.11 Position and Time for the Vehicle of Exercise 13.10

Position (m)	Time (s)
0	0
2.74	2.05
6.1	3.1
18.3	4.8
27.4	5.6
36.6	6.8

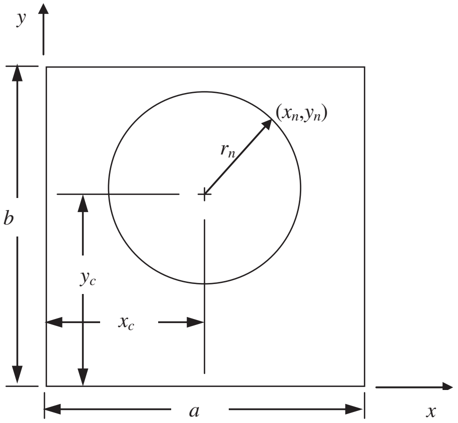


Figure 13.30 Square plate with a circular hole of Exercise 13.11.

The terms appearing in the equation are defined in Figure 13.30. Use this relation to determine the radius and center of the circular hole by letting $x_c = 1.25$ and $y_c = 1.5$ and by using simulated data generated from the following MATLAB expression:

```
r = ro+0.02*randn(N,1);
```

where $ro = 0.075$ is the nominal radius of the hole and $N = 40$. The result should look similar to that shown in Figure 13.31.

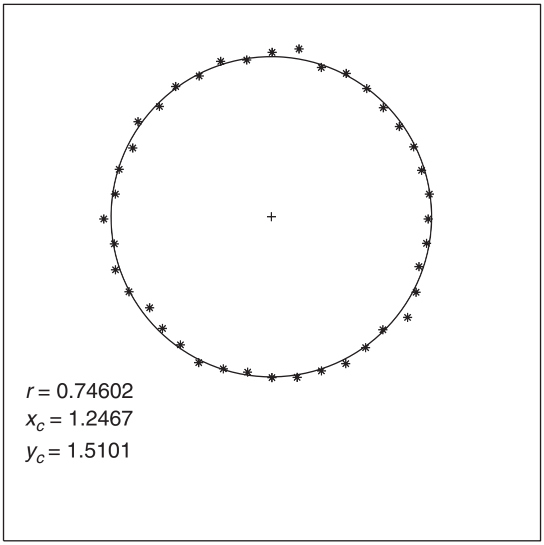


Figure 13.31 Circle fitted to CMM data.

Section 13.5.2

13.12 Find the location of the center of the smallest sphere that contains on its boundary or in its interior the following four points: $A = (1, 1, 1)$, $B = (-1, 2, 4)$, $C = (2, 3, 4)$, and $D = (-3, -4, 1)$, where the numbers in the parenthesis are the x-, y-, and z-coordinates of the point, respectively. The radius of a sphere whose center is located at (x_c, y_c, z_c) is given by

$$R^2 = (x_c - x)^2 + (y_c - y)^2 + (z_c - z)^2$$

where R is the radius of the sphere and (x, y, z) are the coordinates of A, B, C, and D. The design variables are the (x_c, y_c, z_c) coordinates of the center of the sphere and its radius R. Solve the problem with `fmincon` with an initial point $(x_c, y_c, z_c, R) = (4, 4, 4, 4)$ (Answer: $(x_c, y_c, z_c, R) = (-0.5, -0.5, 2.5, 4.5552)$).

13.13 Consider the three-bar truss shown in Figure 13.32. The vertical deflection of its loaded joint gives the objective function[22]

$$\text{minimize } f = \frac{Ph}{E}\frac{1}{x_1 + \sqrt{2}x_2}$$

where the cross-sectional areas of its members are $A_1 = x_1$ and $A_2 = x_2$, hence, x_1 and x_2 are the design variables. Load P is applied in the direction shown in Figure 13.33. The constraints are the applicable stresses on the three members and the lower and upper bounds on the design variables, which are given by

$$\frac{P(x_2 + \sqrt{2}x_1)}{\sqrt{2}x_1^2 + 2x_1x_2} - \sigma^{(u)} \le 0$$

$$\frac{P}{x_1 + \sqrt{2}x_2} - \sigma^{(u)} \le 0$$

$$\frac{Px_2}{\sqrt{2}x_1^2 + 2x_1x_2} + \sigma^{(l)} \le 0$$

$$x_i^{(l)} \le x_i \le x_i^{(u)} \qquad i = 1, 2$$

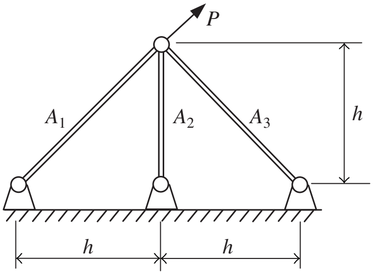


Figure 13.32 Three-bar truss of Exercise 13.13.

[22] Rao, *Engineering Optimization*, p. 530.

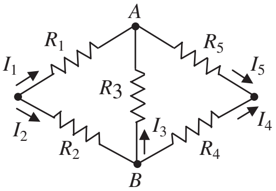


Figure 13.33 Bridge network of Exercise 13.14.

where $\sigma^{(u)}$ is the maximum permissible stress in tension, $\sigma^{(l)}$ is the maximum permissible stress in compression, $x_i^{(l)}$ is the lower bound on x_i, and $x_i^{(u)}$ is the upper bound on x_i. The values for the parameters are $\sigma^{(u)} = 17.5$, $\sigma^{(l)} = -12$, $x_i^{(l)} = 0.1$, $x_i^{(u)} = \infty$ $(i = 1, 2)$, $P = 1$, $E = 2$, and $h = 2$. Use `fmincon` to obtain the optimized values for the cross-sectional areas and the vertical deflection. Assume an initial value of $(x_1, x_2) = (0, 0)$ (Answer: $x_1 = 8.3177$, $x_2 = 26.924$, and $f = 0.021554$).

13.14 Consider the resistor bridge network shown in Figure 13.29, which consists of five resistors R_i, each carrying a current I_i, $i = 1, \ldots, 5$. The voltage drop across each resistor is $V_i = R_i I_i$. Suppose that $V_1 = 3$ V, $V_3 = 1$ V, and $V_5 = 1$ V for R_1, R_3, and R_5, respectively. Also assume that the current I_i varies between a lower limit of 1 A and an upper limit of 2 A for all resistors. The total power dissipated in the five resistors is given by

$$P = \sum_{n=1}^{5} I_n^2 R_n = \sum_{n=1}^{5} V_n^2 / R_n$$

Then, the optimization statement is

$$\begin{aligned} &\text{minimize } P \\ &\text{subject to: } 1 \le I_n \le 2 \\ &\qquad V_1/R_1 + V_3/R_3 - V_5/R_5 = 0 \\ &\qquad V_2/R_2 - V_3/R_3 - V_4/R_4 = 0 \end{aligned}$$

where the two equality constraints are obtained from setting the sum of the currents at the points A and B equal to zero. The values of V_2 and V_4 can be obtained from Kirchhoff's voltage law: the sum of the voltages for the two closed circuits in Figure 13.33 is equal to zero. This yields that $V_2 = V_4 = 2$.

Use `fmincon` to obtain the optimum values for each of the five resistors and the total power dissipation using an initial point $(R_1, R_2, R_3, R_4, R_5) = (1, 1, 1, 1, 1, 1)$ (Answer: $[R_1, R_2, R_3, R_4, R_5] = [3, 1, 1, 2, 0.5]$ ohms and total power dissipation $= 12$ W).

Section 13.5.3

13.15 A company has m manufacturing facilities to produce a product. The product is shipped to n warehouses. The warehouse at the jth location requires at least b_j units of the product to satisfy its demand. The manufacturing facility at the ith location has a capacity to produce a_i units of the product. The cost of shipping x_{ij} units of the product from manufacturing facility i to warehouse j is represented by $c_{ij}x_{ij} + d_{ij}x_{ij}^2$ where c_{ij}

and d_{ij} are constants. Thus, the problem can be formulated in a quadratic programming form as

$$\text{minimize} \sum_{i=1}^{m} \sum_{j=1}^{n} (\mathrm{c}_{ij} x_{ij} + d_{ij} x_{ij}^2)$$

$$\text{subject to: } \sum_{i=1}^{m} x_{ij} \geq b_j \qquad j = 1, \ldots, m$$

$$\sum_{j=1}^{n} x_{ij} \geq a_j \qquad i = 1, \ldots, m$$

$$x_{ij} \geq 0 \qquad \text{for all } i, j$$

Assume that $m = 6, n = 4, a = [8, 24, 20, 24, 16, 12]', b = [29, 41, 13, 21]'$,

$$c = \begin{bmatrix} 300 & 270 & 460 & 800 \\ 740 & 600 & 540 & 380 \\ 300 & 490 & 380 & 760 \\ 430 & 250 & 390 & 600 \\ 210 & 830 & 470 & 680 \\ 360 & 290 & 400 & 310 \end{bmatrix} \qquad d = \begin{bmatrix} -7 & -4 & -6 & -8 \\ -12 & -9 & -14 & -7 \\ -13 & -12 & -8 & -4 \\ -7 & -9 & -16 & -8 \\ -4 & -10 & -21 & -13 \\ -17 & -9 & -8 & -4 \end{bmatrix}$$

and an initial point of

$$x_0 = \begin{bmatrix} 5 & 5 & 0 & 0 \\ 5 & 5 & 0 & 10 \\ 10 & 0 & 0 & 0 \\ 5 & 20 & 0 & 0 \\ 5 & 0 & 15 & 0 \\ 5 & 5 & 0 & 0 \end{bmatrix}$$

Solve this problem using `quadprog` to obtain the optimum number of units that should be produced at manufacturing facility i and shipped to warehouse j, x_{ij}. Answer:

$$x = \begin{bmatrix} 0 & 8 & 0 & 0 \\ 0 & 3 & 0 & 21 \\ 20 & 0 & 0 & 0 \\ 0 & 24 & 0 & 0 \\ 3 & 0 & 13 & 0 \\ 6 & 6 & 0 & 0 \end{bmatrix}$$

Section 13.5.4

13.16 The two-bar truss shown in Figure 13.34 is symmetric about the y-axis. The ratio of the position of links 1 and 2 x/h and the ratio of the cross-sectional area of the links A/A_{ref} are treated as the design variables x_1 and x_2, respectively, A_{ref} is the reference value of the area A, and h is the height of the truss. The direction of the applied load P is subject

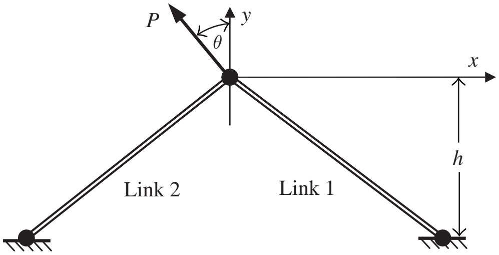


Figure 13.34 Two-bar truss of Exercise 13.16.

to change within the range $-90° \leq \theta \leq 90°$. The weight of the truss w is to be minimized. Thus,

$$\text{minimize } w = 2\rho h x_2 A_{\text{ref}}\sqrt{1 + x_1^2}$$

$$\text{subject to: } -\sigma_0 \leq \frac{P\sqrt{1 + x_1^2} \leq (x_1 \cos\theta + \sin\theta)}{2x_1x_2A_{\text{ref}}} \leq \sigma_0$$

$$-\sigma_0 \leq \frac{P\sqrt{1 + x_1^2} \leq (x_1 \cos\theta - \sin\theta)}{2x_1x_2A_{\text{ref}}} \leq \sigma_0$$

where ρ is the density. The constraints corresponding to the stresses induced in links 1 and 2 are given in the above equations where P is the applied load. In addition, the following upper and lower bounds are imposed on the design variables x_1 and x_2:

$$x_i^{\min} \leq x_i \leq x_i^{\max} \qquad i = 1, 2$$

The values of the parameters are as follows: $x_1^{\min} = 0.15, x_2^{\min} = 0.15, x_1^{\max} = 3.0$, and $x_2^{\max} = 3.5$, $\rho = 7833.4$ kg/m^3, $P = 35{,}585.8$ N, $\sigma_0 = 127.55 \times 10^6$ Pa, $h = 2.159$ m, and $A_{\text{ref}} = 6.452 \times 10^{-4}$ m^2. Use `fseminf` to obtain the optimum design variables and create a plot of the stresses as a function of θ sampled in 5° intervals as the optimum is approached. The result should look like that shown in Figure 13.35 when an initial design $(x_1^0, x_2^0) = (0.1, 0.1)$ is assumed (Answer: $x_1 = 0.8088, x_2 = 0.45058$, and $w = 12.6431$).

Section 13.7

13.17 Solve the optimization problem in Exercise 13.6 using `ga`. Use the default values of the genetic algorithm and choose the option for the hybrid function `fmincon` that runs after the genetic algorithm terminates. Assume $lb = [0\ 0\ 0]$ and $ub = [1\ 1\ 1]$. Create the plot of the *Currentbestindividual* and display the numerical value of the variables in the command window at the end of the program's execution.

13.18 Solve the optimization problem in Exercise 13.13 with `ga`. Use the default values of the genetic algorithm. Create the plot of the *Currentbestindividual* and produce the numerical value of the design variables at the end of its execution.

13.19 Solve the optimization problem in Example 13.18 with `gamultiobj`. Use the penalty approach of Section 13.7 to add the nonlinear constraints as a penalty function to each objective function using a penalty parameter $P = 100$ and the option *PlotFcns* in `gaplotpareto` to produce the Pareto front as it evolves during the optimization convergence.

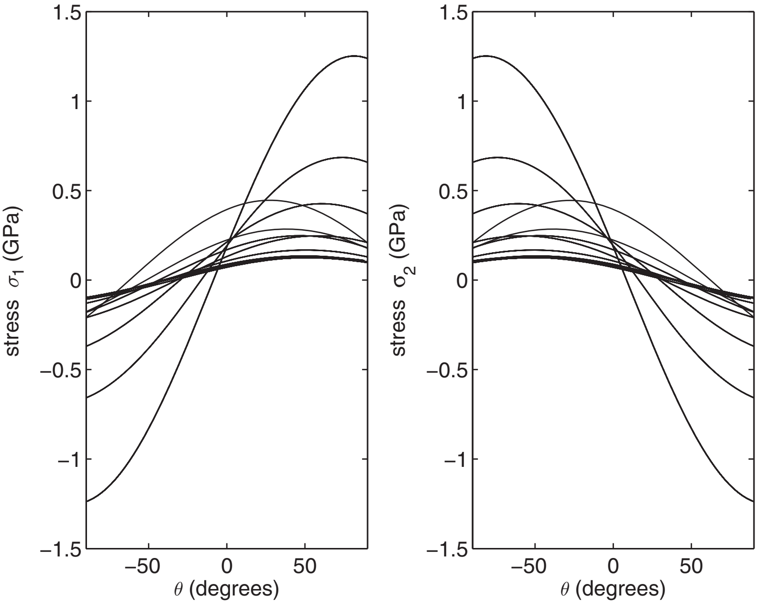


Figure 13.35 Stresses in links 1 and 2 of Figure 13.34 as the optimum is approached.

13.20 Solve the optimization problem in Example 13.15 with (a) `ga` and (b) `gamultiobj`. For part (b), use the same constraints as that of part (a) except that the second objective function is the stress of shaft 1, that is,

$$\text{minimize } \frac{1}{0.1x_6^3}\sqrt{\left(\frac{745x_4}{x_2x_3}\right)^2 + 16.9 \times 10^2}$$

For part (b), use the penalty approach discussed in Section 13.7 to add the seven nonlinear constraints as a penalty to each objective function. Use a penalty parameter $P = 100$ and select the *PlotFcns* option in `gaplotpareto` to produce the Pareto frontier as it evolves to the final solution while the optimizer converges.

BIBLIOGRAPHY

J. S. Arora, *Introduction to Optimum Design*, McGraw-Hill, New York, 1989.

M. Austin and D. Chancogne, *Engineering Programming in C, MATLAB and JAVA*, John Wiley & Sons, New York, 1998.

V. Changkong and Y. Y. Haimes, *Multiobjective Decision Making: Theory and Methodology*, Elsevier Science Publishing Co., New York, 1983.

K. Deb, *Multi-Objective Evolutionary Optimization Using Evolutionary Algorithms*, John Wiley & Sons, New York, 2001.

N. Draper and H. Smith, *Applied Regression Analysis*, John Wiley & Sons, New York, 1966.

H. Eschenauer, J. Koski, and A. Osyczka, Eds, *Multicriteria Design Optimization*, Springer-Verlag, New York, 1990.

J. Golinski, "Optimum synthesis problems solved by means of nonlinear programming and random methods," *Journal of Mechanisms*, **5**, 1970, pp. 287–309.

C. Gosselin and J. Angeles, "A global performance index for the kinetic optimization of robotic manipulators," *ASME Journal of Mechanical Design*, **113**, 1991, p. 222.

U. Kirsch, *Optimal Structural Design*, McGraw-Hill, New York, 1981.

A. Messac, "Physical programming: effective optimization for computational design," *AIAA Journal*, **34**, 1996, pp. 149–158.

A. Osyczka, *Multicriterion Optimization in Engineering with Fortran Programs*, Ellis Horwood Limited, West Sussex, UK, 1984.

P. Y. Papalambros and D. J. Wilde, *Principles of Optimal Design*, Cambridge University Press, Cambridge, UK, 1988.

S. S. Rao, *Engineering Optimization, Theory and Practice*, 3rd ed., John Wiley & Sons, New York, 1996.

G. V. Reklaitis, A. Ravindran, and K. M. Ragsdell, *Engineering Optimization*, John Wiley & Sons, New York, 1983.

J. Shigley and C. Mischke, *Mechanical Engineering Design*, McGraw-Hill, New York, 1989.

D. A. Stephenson and J. S. Agapiou, *Metal Cutting Theory and Practice*, Marcel Dekker, New York, 1997.

G. N. Vanderplaats, *Numerical Optimization Techniques for Engineering Design*, McGraw-Hill, New York, 1984.

D. A. Van Veldhuizen and G. B. Lamont, "Multi-Objective Evolutionary Algorithm Research: A History and Analysis," Technical Report TR-98-03, Air Force Institute of Technology, Wright Patterson AFB, OH, 1998.

D. J. Wilde, *Globally Optimal Design*, John Wiley & Sons, New York, 1978.

14

Biological Systems: Transport of Heat, Mass, and Electric Charge

Keith E. Herold

Several models of transport phenomena in biological systems are analyzed.

14.1 HEAT TRANSFER IN BIOLOGICAL SYSTEMS

14.1.1 Heat Transfer in Perfused Tissue

Transient heat conduction in biological material is represented by the bioheat equation,[1] which is often written as

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T + \frac{1}{\rho c}[S + S_p + S_m]$$

where T is the temperature in the tissue, α is the thermal diffusivity of the material, c is the specific heat, and ρ is the density. The term S represents energy from a source, S_m accounts for metabolic energy release, and S_p accounts for the energy carried away by blood flow (perfusion). The term S is used to include any artificially imposed sources such as radio frequency (RF) heating or cryogenic cooling. The metabolic rate of energy release varies with tissue type, activity level, and nutrition. A representative value for the metabolic rate is 145 W/m^3. The perfusion term is usually a sink, carrying away the metabolic energy. In its simplest form, the perfusion term can be modeled as

$$S_p = \dot{m}_b c_b (T_{\text{ab}} - T)$$

where $\dot{m}_b$ is the blood mass flow rate per unit volume, c_b is the blood specific heat, and T_{ab} is the temperature of the arterial blood entering the tissue. This term assumes that the blood leaves the tissue at the local tissue temperature T.

The bioheat equation can be used to model the temperature in biological tissue in a wide range of applications, including the prediction of normal tissue temperature, the prediction of transient temperature under exposure to temperature extremes, and the prediction of temperature changes during therapy, ranging from RF ablation to cryosurgery.

We shall now apply this equation in the example that follows.

Example 14.1 Ablation of a spherical tumor

One approach for treatment of a tumor is to use RF ablation, where a high-frequency electric current is dissipated in the tumor's tissue causing a rise in temperature to the point where the cells die. Ablation therapy usually employs a narrow-gauge needle that is inserted into the tumor to deliver the RF energy. In general, a temperature increase of 10K or higher will cause cell death. The objective with this therapy is to kill all the tumor cells without killing large numbers of cells in the healthy tissue surrounding the tumor. Thus, one must have an understanding of the thermal transport in the tissue in order to properly employ thermal ablation therapy.

We shall analyze this type of therapy by modeling the tumor as a sphere and by making the following assumptions. We assume that the tumor has a radius of 1 cm and that it is perfused at the same level as healthy tissue. The tumor is located at the center of

[1] H. H. Pennes, "Analysis of tissue and arterial blood temperature in the resting human forearm," *Journal of Applied Physiology*, **1**, 1948, pp. 93–102.

a sphere of healthy tissue that has a radius of 5 cm and all of the tissue has a metabolic energy release rate of 145 W/m^3. In practical terms, the metabolic energy release has little effect on this particular example because the RF source has considerably higher energy intensity: 4×10^5 W/m^3. In addition, the RF energy of the source is assumed to dissipate only within the tumor and its duration is 400 s. The power is delivered locally to the tumor by positioning the electrode in the center of the tumor. The initial temperature of the tumor and surrounding tissue is uniform at 37°C. In addition, it is assumed that because of symmetry there is zero flux at the center of the tumor and that at the outer surface of the healthy tissue the flux is also zero. The following numerical values for the parameters are used: $\dot{m}_b = 0.18$ kg/m^3/s, $c_b = 3{,}300$ J/kg/K, $S_m = 145$ W/m^3, $T_{\text{ab}} = 37$°C, $\alpha = 10^{-7}$ m^2/s, $c = 3{,}800$ J/kg/K, $S = 4 \times 10^5$ W/m^3, and $\rho = 850$ kg/m^3.

A numerical solution to this model is obtained by using pdepe with $m = 2$, which indicates that a spherical coordinate system is to be used. The use of pdepe is outlined in more detail in Section 5.5.6. We shall determine the radial temperature profiles at $t = 300$ s, 400 s, 600 s, and 1,500 s and the temperature responses at radial locations $r = 0$ cm, 1 cm, and 1.5 cm. The program is as follows:

```
function Example14_1
rho = 850;   cp = 3800;   Sm = 145;   mb = 0.18;
cpb = 3300;   Tb = 37;   alpha = 1e-7;   S = 4e5;   R = 0.05;
r = linspace(0, R, 51);
t = linspace(0, 3000, 1001);
T = pdepe(2, @tumorPDE, @tumorIC, @tumorBC, r, t, [], alpha, S, rho, cp, mb, . . .
              cpb, Tb, Sm);
figure(1)
plot(t, T(:,1), 'k-', t, T(:,11), 'k—', t, T(:,16), 'k-.', [400 400], [30 70], 'k--')
xlabel('t (s)')
ylabel('Temperature (\circC)')
legend('r = 0', 'r = 1 cm', 'r = 1.5 cm')
ylim([30 70])
figure(2)
r = r*100;
plot(r, T(61,:), 'k-.', r, T(134,:), 'k-', r, T(201,:), 'k--', r, T(501,:), 'k:', [1, 1], . . .
      [30, 70], 'k--')
xlabel('r (cm)')
ylabel('Temperature (\circC)')
legend('t = 300 s', 't = 400 s', 't = 600 s', 't = 1500 s')
ylim([30 70])

function [c, f, s] = tumorPDE(r, t, T, DTDr, alpha, S, rho, cp, mb, cpb, Tb, Sm)
c = 1;
f = DTDr*alpha;
s = (S*(r<0.01)*(t<400)+Sm+mb*cpb*(Tb-T))/(rho*cp);

function T0 = tumorIC(r, alpha, S, rho, cp, mb, cpb, Tb, Sm)
T0 = 37;

function [pl, ql, pr, qr] = tumorBC(rl, Tl, rr, Tr, t, alpha, S, rho, cp, mb, cpb, Tb, Sm)
pl = 0;   ql = 1;
pr = 0;   qr = 1;
```

The result of the execution of this program is shown in Figures 14.1 and 14.2. Figure 14.1 shows a plot of temperature versus radial location for four different times.

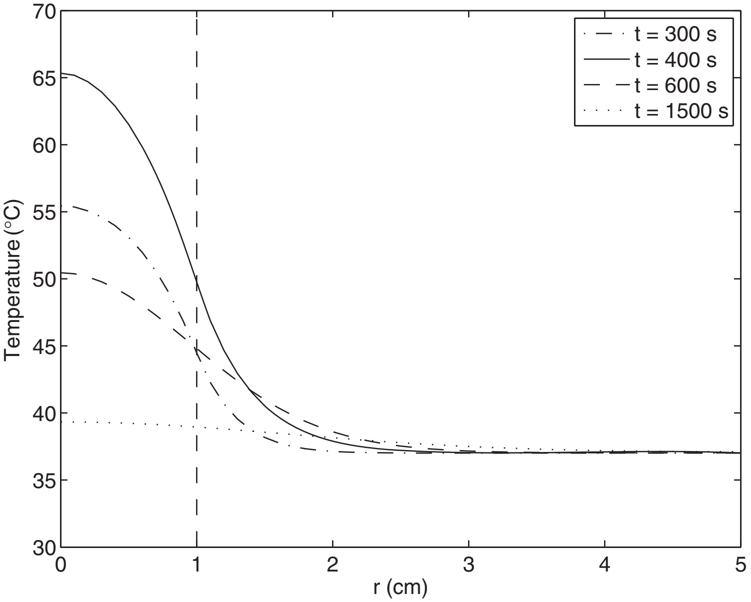


Figure 14.1 Radial temperature profiles at four instances of time. The point $r = 0$ is the center of the tumor and the outer surface of the tumor at $r = 1$ cm is represented by the vertical line. The RF energy source is assumed to dissipate only within the tumor and is applied for 400 s.

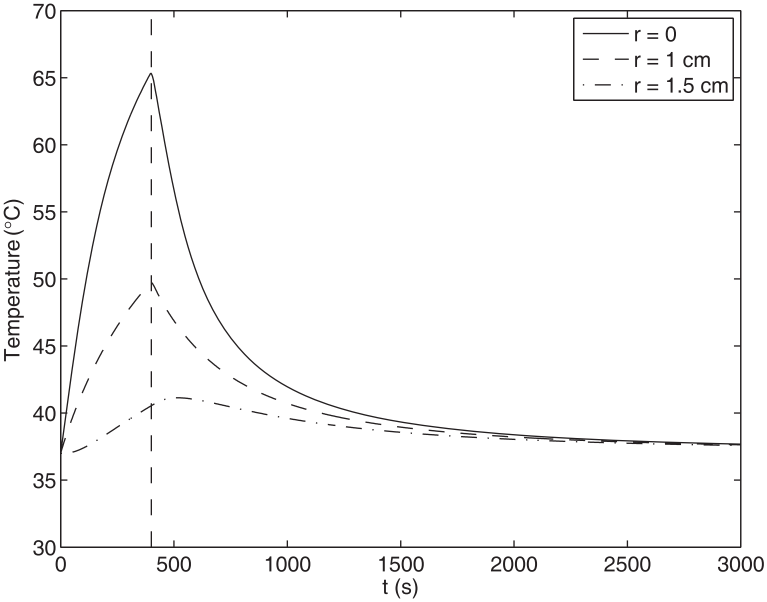


Figure 14.2 Temperature versus time at three radial locations. The RF source turns on at $t = 0$ and is turned off at $t = 400$ s. Note the conduction time lag experienced by the surrounding tissue.

The highest temperature occurs at the end of the heating phase at 400 s, where it is seen that the temperature at the center of the tumor reaches approximately 66°C. The temperature at the tumor boundary at $r = 1$ cm, which is marked by a vertical line, is rather high and may be expected to damage some healthy tissue surrounding the tumor. After the RF source is turned off, the temperature drops off rapidly due to perfusion and conduction to the surrounding tissue. It can be seen that the thermal profile penetrates a few centimeters into the healthy tissue, but the temperature rise is within healthy limits for most of this tissue.

In Figure 14.2, we have a plot of temperature versus time at the tumor center $r = 0$, at the tumor boundary $r = 1$ cm, and at $r = 1.5$ cm. The curves show the rapid rise in temperature due to the RF source and the subsequent decrease due to conduction and perfusion. The tumor center experiences the highest temperature in the system and the surrounding tissue experiences a temperature rise after a conduction time lag.

14.1.2 Thermal Conductivity Determination

To enable prediction of heat conduction in a material, the thermal conductivity of that material must be known. Tabulated sources of thermal conductivity data generally originated from measurements. One measurement technique that has been widely used is a transient line source technique. It involves embedding a heating wire in the material of interest, providing the wire constant power, and measuring the temperature as a function of time. This procedure is described by the transient conduction equation for a cylindrical geometry; thus,

$$\frac{\partial^2 T}{\partial r^2} + \frac{1}{r}\frac{\partial T}{\partial r} = \frac{1}{\alpha}\frac{\partial T}{\partial t}$$

where $T = T(r, t)$ is the temperature, r is the radial coordinate measured from the centerline, t is the time, and α is the thermal diffusivity. If the material surrounding the line source is uniform, then we expect cylindrical symmetry. Based on a Green's function solution procedure, it is found that[2]

$$T(r, t) = T_0 + \frac{Q}{4\pi k}\int_{z(r,t)}^{\infty} \frac{e^{-y}}{y}\,dy \tag{14.1}$$

where

$$z(r, t) = \frac{r^2}{4\alpha t}$$

Q is the strength of the line source in W/m, T_o is the initial temperature, and k is the thermal conductivity. The integral, which is called the exponential integral, can be written in series form as

[2] H. S. Carslaw and J. C. Jaeger, *Conduction of Heat in Solids*, 2nd ed., Oxford University Press, Oxford, 1986.

$$\int_{z(r,t)}^{\infty} \frac{e^{-y}}{y}\, dy = -\gamma - \ln z(r,t) - \sum_{n=1}^{\infty} \frac{(-1)^n (z(r,t))^n}{nn!}$$

where $\gamma = 0.57721$ is Euler's constant. For small nonzero values of $z(r, t)$; that is, at sufficiently large values of t, the integral can be approximated as

$$\int_{z(r,t)<<1}^{\infty} \frac{e^{-y}}{y}\, dy \approx -\gamma - \ln z(r,t) = -\gamma - \ln \frac{r^2}{4\alpha t} = -\gamma - \ln \frac{r^2}{4\alpha} + \ln t$$

Thus, the temperature near the centerline of the line source can be approximated as

$$T(r,t) = T_0 + \frac{Q}{4\pi k}\left[-\gamma - \ln \frac{r^2}{4\alpha} + \ln t\right] = C_0 + \frac{Q}{4\pi k} \ln t$$

Differentiation with respect to $\ln t$ and rearrangement of this expression gives

$$k = \frac{Q}{4\pi}\left(\frac{dT}{d \ln t}\right)^{-1} \tag{14.2}$$

which is valid at a fixed radial location near the centerline of the line source. We see that the thermal conductivity can be determined from the slope of the line of temperature versus the logarithm of time since the power to the wire Q is known.

Hence, the line source solution can be used along with experimental data to determine the thermal conductivity of a substance, such as a biological tissue. The data collection involves recording temperature versus time followed by a curve fit analysis to determine the thermal conductivity using the approximate solution given above.

Example 14.2 Determination of the thermal conductivity of a biological material

We shall use simulated data to show how one can determine the thermal conductivity of a biological material by using the procedure described above. We shall generate the data using Eq. (14.1) and approximate the thermal conductivity by using a curve-fit procedure to these data using Eq. (14.2). For Eq. (14.1), we assume that $T_o = 20°\text{C}$, $r = 0.001$ m, $\alpha = 10^{-6}$ m^2/s, and $k = 0.6$ W/m/K. In addition, we add a small random temperature component to the value computed from Eq. (14.1). In order to use the approximate expression given by Eq. (14.2), $z(r, t) << 1$. Using the assumed values, we find that this constraint is satisfied when $t > 10$ s. The script is as follows:

```
Npt = 60;   t = logspace(-1, 3, Npt);
k = 0.6;   Q = 1;   r =0.001;   alpha = 1e-6;   To = 20;
T = To+Q/(4*pi*k)*expint(r^2./(4*alpha.*t));
T1 = T+0.05*(rand(1, Npt)-0.5);
zz = polyfit(log(t(31:60)), T1(31:60), 1);
semilogx(t, T1, 'k+', 'MarkerSize', 5)
hold on
```

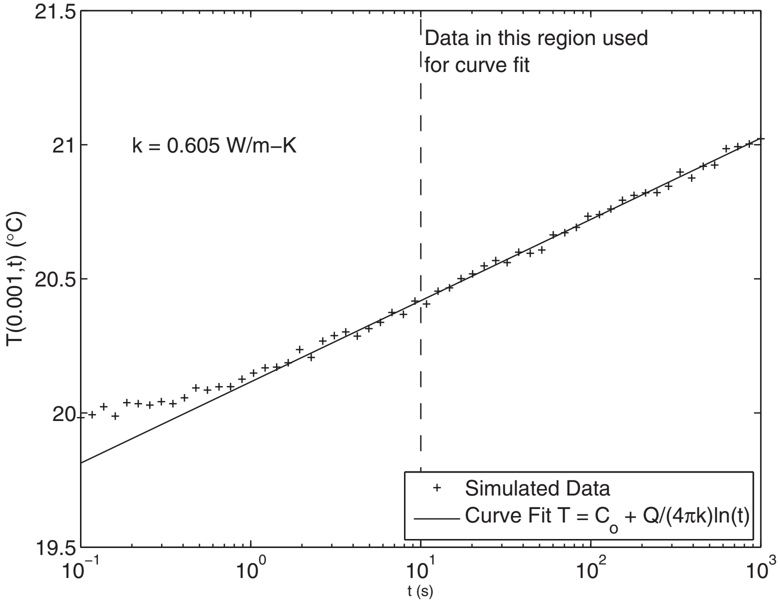


Figure 14.3 Curve fit to line source probe data to determine thermal conductivity in a biological material.

```
semilogx(t, zz(2)+zz(1)*log(t), 'k-')
legend('Simulated Data', 'Curve Fit T = C_o + Q/(4\pik)ln(t)', 'Location', 'SouthEast')
z = axis;
semilogx([10 10], [z(3) z(4)], 'k--')
xlabel('t (s)')
ylabel(['T(' num2str(r) ',t) (\circC)'])
text(0.2, 21, ['k = ' num2str(Q/(4*pi*zz(1)),4) ' W/m-K'])
text(10.5, z(4)-0.1, 'Data in this region used')
text(10.5, z(4)-0.2, 'for curve fit')
```

Execution of the program results in Figure 14.3.

14.2 MASS TRANSFER IN BIOLOGICAL SYSTEMS

14.2.1 Bicarbonate Buffer System

A major component of the pH buffer system in mammalian blood is the bicarbonate buffer. The bicarbonate (HCO_3^-) ion is one of the equilibrium species of carbonic acid (H_2CO_3), which is diprotic; that is, it has two protons. In addition, carbonic acid can condense into carbon dioxide (CO_2) and water. An understanding of the bicarbonate buffer system can be obtained by obtaining a titration curve for carbonic acid over the entire pH range. Such a curve can be obtained experimentally by starting from a concentrated solution of carbonate and adding incremental quantities of a strong acid such as HCl while monitoring the pH.

The bicarbonate buffer reactions can be written as

$$
\begin{array}{lll}
CO_2 + H_2O \leftrightarrow H_2CO_3 & K_c = 1.7 \times 10^{-3} & x_1 \\
H_2CO_3 \leftrightarrow HCO_3^- + H^+ & K_1 = 2.5 \times 10^{-4} & x_2 \\
HCO_3^- \leftrightarrow CO_3^{2-} + H^+ & K_2 = 5.6 \times 10^{-11} & x_3
\end{array}
$$

where K_c, K_1, and K_2 are the equilibrium constants. The variables x_1, x_2, and x_3 that are listed at the right are the extents of each reaction in the forward direction. The equilibrium for this set of reactions shifts as the titration proceeds, that is, as more H^+ is added: the amount of H^+ that is added is denoted y. The equilibrium point can be computed by relating the amount of each species to the extent of reactions, as shown in Table 14.1. The starting amount of carbonate is assumed to be 1 mol. The concentration of each species can be determined by assuming a particular volume for the reaction; here we assume 1 L so that the concentration is equal to the amount in mol.

The equilibrium equations for each reaction can be written in terms of the extents of reaction as

$$
\begin{aligned}
K_c &= \frac{[H_2CO_3]}{[CO_2]} = \frac{x_2 - x_1}{x_1} \\
K_1 &= \frac{[HCO_3^-][H^+]}{[H_2CO_3]} = \frac{(x_2 - x_3)(x_2 + x_3 + y)}{x_1 - x_2} \\
K_2 &= \frac{[CO_3^{2-}][H^+]}{[HCO_3^-]} = \frac{(1 + x_3)(x_2 + x_3 + y)}{x_2 - x_3}
\end{aligned}
\tag{14.3}
$$

This is a set of three nonlinear algebraic equations in the three unknown extent of reactions x_1, x_2, and x_3. As the titration proceeds; that is, as y changes, a new solution is found at each step and we can calculate the pH from

$$
pH = -\log_{10}[H^+] = -\log_{10}(x_2 + x_3 + y) \tag{14.4}
$$

TABLE 14.1 Carbonic Acid Equilibrium Variables

Species	Starting amount (mol)	Add (mol)	Ending amount (mol)
CO_2	0	0	$-x_1$
H_2CO_3	0	0	$x_1 - x_2$
HCO_3^-	0	0	$x_2 - x_3$
CO_3^{2-}	1	0	$1 + x_3$
H^+	0	y	$x_2 + x_3 + y$

We can solve Eq. (14.3) for x_1, x_2, and x_3 to obtain

$$\begin{aligned} x_1 &= -H^2/D \\ x_2 &= -(H^2 K_c + H^2)/D \\ x_3 &= -(H^2 K_c + H^2 + HK_1 K_c)/D \\ D &= K_c H^2 + H^2 + HK_1 K_c + K_1 K_2 K_c \end{aligned} \qquad (14.5)$$

From Eq. (14.4), we see that $H = 10^{-\mathrm{pH}}$ and, therefore,

$$H = y - (2H^2 K_c + 2H^2 + HK_1 K_c)/D \qquad (14.6)$$

Thus, for a given value of y, one can solve for H.

We shall now illustrate these results with an example.

Example 14.3 Carbonic acid titration curve

We shall use Eq. (14.6) to determine a graph of pH versus the amount of H^+ added as follows:

```
function Example14_3
K1 = 2.5e-4;   Kc = 1.7e-3;   K2 = 5.6e-11;
N = 400;   y = linspace(0.0001, 2.5, N);
pHg = 15;
pH = zeros(N,1);
for n = 1:N
   pH(n) = fzero(@carbon, pHg, [], y(n), K1, K2, Kc);
   pHg = pH(n);
end
```

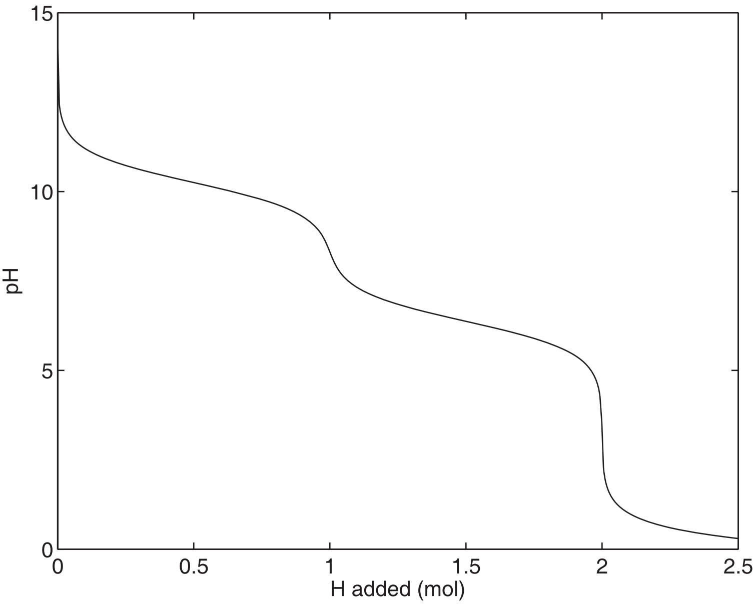


Figure 14.4 Titration curve for carbonc acid.

```
plot(y, pH, 'k-')
xlabel('H added (mol)')
ylabel('pH')
function f = carbon(pH, y, K1, K2, Kc)
H = 10^(-pH);
D = (H^2*Kc+H^2+H*K1*Kc+K1*K2*Kc);
x2x3 = (2*H^2*Kc+2*H^2+H*K1*Kc)/D;
f = H+x2x3-y;
```

The execution of this program produces the result in Figure 14.4, which represents carbonic acid equilibrium.

In the mammalian system, additional control mechanisms act to maintain pH control. Two mechanisms that act on the bicarbonate system are (1) CO_2 concentration control by mass transfer between the blood and air in the lungs, and (2) excretion of excess bicarbonate from the blood in the kidneys. Of these control mechanisms, the respiratory mechanism is fast acting (on the order of minutes) and the renal mechanism is much slower (on the order of hours). We can simulate the effect in the lung by incorporating a CO_2 mass transfer model of the lung. The simplest version of this is to assume a fixed carbon dioxide concentration $[CO_2]$ in the blood. This is normally done by starting from a partial pressure of CO_2, denoted P_{CO_2}, and then using a Henry's law model to calculate the corresponding aqueous CO_2 concentration as

$$[CO_2] = \frac{P_{CO_2}}{H_{CO_2}} \tag{14.7}$$

where H_{CO_2} is the Henry's law constant for CO_2 in water, which has the value $H_{CO_2} = 29.76$ atm/M (or 22617.6 mmHg/M). In the pH range of mammalian blood, the bicarbonate chemistry can be summarized as

$$CO_2 + H_2O \leftrightarrow HCO_3^- + H^+ \quad K_{c1} = K_cK_1 = 4.25 \times 10^{-7}$$

with the corresponding equilibrium relation

$$K_{c1} = \frac{[HCO_3^-][H^+]}{[CO_2]} \tag{14.8}$$

which is a combination of the first two equations of Eq. (14.3). In the absence of the respiratory CO_2 control mechanism, this would provide a buffering capability centered about $-\log_{10}K_c = 6.37$, which is much more acidic than blood whose typical pH is 7.4. However, when the concentration of CO_2 is held essentially constant, the pH depends only on the bicarbonate concentration, which is controlled within a range by the kidneys. This detail is reflected in the blood calculations given in Example 14.4.

14.2.2 Carbon Dioxide Transport in Blood

Carbon dioxide is generated by cellular respiration as the ultimate waste product of oxidation of organic species. All cells produce carbon dioxide continuously. As a small molecule, it can diffuse readily through cellular membranes to arterial blood flowing through the capillaries. When the CO_2 is picked up by the blood, it typically exists as a dissolved fraction in the aqueous serum. Although the carbonic acid form is thermodynamically preferred, the kinetics of the conversion is so slow that little conversion occurs except in the presence of carbonic anhydrase enzyme present inside the red blood cells. Thus, CO_2 exists in the serum in dissolved form, but it readily crosses the red blood cell membrane where it is rapidly converted to carbonic acid (H_2CO_3) and then bicarbonate ($[HCO_3^-]$), depending on the local equilibrium conditions. The equilibrium constants imply that the majority of the CO_2 that enters the red blood cell gets stored as bicarbonate. If we assume equilibrium conditions, we can compute the amount of CO_2 stored in each form. This is important when we want to determine how much CO_2 gets transported as the blood flows toward the lungs. We have to account for all of the forms of storage since the various storage modes are reversible, as described in Section 14.2.1. In the lung, the CO_2 will be transported from the blood to the air as long as the partial pressure of the CO_2 in the blood is greater than the partial pressure of the CO_2 in the air. As the dissolved fraction in the blood decreases due to mass transfer to the air, bicarbonate in the red blood cells gets converted back to carbonic acid and CO_2, replenishing the dissolved fraction. Thus, although the dissolved fraction of CO_2 in the blood is relatively small, the total CO_2 stored in the blood is much larger and the advective transport of CO_2 to the lungs depends strongly on the stored fraction. The storage of CO_2 in the blood can be related to the partial pressure of CO_2 (P_{CO_2}) and the pH as follows.

The concentration of dissolved CO_2 is indicated by subscript d and is related to the partial pressure of CO_2 through Henry's law as

$$[CO_2]_d = \frac{P_{CO_2}}{H_{CO_2}} \tag{14.9}$$

The concentration of protons in solution is obtained from the definition of pH as

$$[H^+] = 10^{-pH} \tag{14.10}$$

The primary bound form of CO_2 in the pH range of interest is bicarbonate, which has an equilibrium equation given in Section 14.2.1, that is,

$$[HCO_3^-] = \frac{K_{c1}[CO_2]_d}{[H^+]} \tag{14.11}$$

The total CO_2 is found by summing the dissolved and bound fractions as

$$[CO_2]_T = [CO_2]_d + [HCO_3^-] \tag{14.12}$$

The volume of the CO_2 in the blood expressed as a percentage of the liquid volume is given by

$$V_{CO_2} = RT_K[CO_2]_T \tag{14.13}$$

where $R = 8.314$ J/mol/K and $T_K = 310.15$ K.

These relations will be used in the Example 14.4.

14.2.3 Oxygen Transport in Blood

Oxygen is transported in the blood in both dissolved and bound forms. The primary bound form of oxygen is oxyhemoglobin where up to four oxygen molecules can bind to each molecule of hemoglobin, which is a major constituent in the cytosol of red blood cells. The binding to hemoglobin is cooperative in that the presence of bound oxygen changes the equilibrium constant for additional binding. Thus, the relationship between the dissolved oxygen concentration and the amount of bound oxygen is nonlinear. The Hill model describing this equilibrium can be written as[3]

$$Y = \frac{P_{O_2}^n}{P_{50}^n + P_{O_2}^n} \tag{14.14}$$

where P_{O_2} is the partial pressure of dissolved oxygen, n is the Hill exponent, P_{50} is the partial pressure that gives a 50% saturation, and Y is the saturation fraction, that is, the fraction of binding sites on hemoglobin that are occupied. Oxygen diffuses easily back and forth through the outer membrane of red blood cells. For oxygen, the Henry's law constant is quite high compared to CO_2, which means that the amount of oxygen stored in dissolved form is quite low. We can compute the dissolved concentration from the partial pressure as

$$[O_2]_d = \frac{P_{O_2}}{H_{O_2}} \tag{14.15}$$

The bound concentration is computed as

$$[O_2]_{Hb} = 4[Hb]Y \tag{14.16}$$

where Hb is the hemoglobin. Finally, the total oxygen concentration is computed as the sum of the dissolved and bound fractions, that is,

$$[O_2]_T = [O_2]_d + [O_2]_{Hb} \tag{14.17}$$

At a particular point in the circulation system, say the pulmonary artery, the blood has a set of properties such as pH, CO_2 concentration, and O_2 concentration. At another location, such as the pulmonary vein, the properties are different due to mass transfer in the lung. For situations like blood calculations, it is convenient to use the MATLAB data-type called a structure, which is a convenient way to organize information. Using a structure, all of the properties of the blood can be stored in a single-named variable with extensions. In addition, structure arrays can be created. In the following program, the structure is first populated with the properties of blood at two locations and then the difference calculations are performed to determine the transport.

Example 14.4 Blood calculations

We shall determine the total transport of oxygen into the blood and the total transport of carbon dioxide out of the blood as it passes through the lung for a specified set of the input parameters. These parameters can be obtained from appropriate measurements of the blood. In this example, the input parameters are P_{CO_2}, P_{O_2}, and pH and the output will be the various properties of the blood that are listed in Table 14.2. The values for the various constants for blood are as follows: $K_{c1} = 4.25 \times 10^{-7}$ M, $P_{50} = 26$ mmHg,

[3] R. L. Fournier, *Basic Transport Phenomena in Biomedical Engineering*, 2nd ed., Taylor and Francis, New York, 2007, p. 227.

TABLE 14.2 Components of the Structure *blood* in Example 14.4

Structure element	Quantity	Equation	Units
blood.CO2_dissolved	$[CO_2]_d$	(14.9)	M
blood.H	$[H^+]$	(14.10)	M
blood.bicarb	$[HCO_3^-]$	(14.11)	M
blood.CO2_total	$[CO_2]_T$	(14.12)	M
blood.CO2_V	V_{CO_2}	(14.13)	% Liquid volume
blood.O2_dissolved	$[O_2]_d$	(14.15)	M
blood.O2_Hb	$[O_2]_{Hb}$	(14.16)	M
blood.O2_total	$[O_2]_T$	(14.17)	M

$n = 2.34$, $H_{CO_2} = 22{,}617.6$ mmHg, $H_{O_2} = 740{,}012$ mmHg, and $[Hb] = 0.0022$ M. We shall determine the change in both $[CO_2]_T$ and $[O_2]_T$ between the venous and arterial sides of the lung for the input parameters given in Table 14.3. The program is as follows:

```
function Example14_4
blood(1) = bloodcalc(45, 7.4, 40);    %Venous blood
blood(2) = bloodcalc(40, 7.4, 95);    %Arterial blood
dCO2 = blood(2).CO2_total-blood(1).CO2_total
dO2 = blood(2).O2_total-blood(1).O2_total

function blood = bloodcalc(CO2pp, pH, O2pp)
TK = 310.15;   H_CO2 = 22617.6;    Kc = 4.25e-7;    R = 8.314;
H_O2 = 740012;    n = 2.34;    P50 = 26;    CHb = 0.0022;
blood.pH = pH;
blood.H = 10^(-blood.pH);
blood.CO2_pp = CO2pp;
blood.CO2_dissolved = blood.CO2_pp/H_CO2;
blood.bicarb = Kc*blood.CO2_dissolved/blood.H;
blood.CO2_total = blood.CO2_dissolved+blood.bicarb;
blood.CO2_V = blood.CO2_total*R*TK;
blood.O2_pp = O2pp;
blood.O2_dissolved = blood.O2_pp/H_O2;
blood.O2_Hb = CHb*4*O2pp^n/(P50^n+O2pp^n);
blood.O2_total = blood.O2_dissolved+blood.O2_Hb;
```

Execution of this program gives that the change in $[CO_2]_T$ across the lung is –0.0026 M and the change in $[O_2]_T$ is 0.0020 M.

TABLE 14.3 Blood Input Parameters for Example 14.4

Input parameter	Venous blood	Arterial blood
P_{CO_2} (mmHg)	45	40
pH	7.4	7.4
P_{O_2} (mmHg)	40	95

14.2.4 Perfusion Bioreactor

A perfusion bioreactor is a cell culture system with a continuous flow of a perfusion medium, as shown in Figure 14.5. The inlet flow to the perfusion bioreactor supplies nutrients and oxygen, usually in an aqueous solution; the flow also carries away waste products. In this section, we shall model only the oxygen transport. Referring to Figure 14.5, the governing equation for oxygen concentration C can be written as[4]

$$\mathrm{Pe}(\hat{y})\frac{\partial C}{\partial \hat{x}} = \beta\frac{\partial^2 \mathrm{C}}{\partial \hat{y}^2} \tag{14.18}$$

where $C = C(\hat{x}, \hat{y})$, $\hat{x} = x/L$, $\hat{y} = y/h$, $\beta = L/h$, $\mathrm{Pe}(\hat{y}) = V(\hat{y})h/D$ is the Peclet number, and $V(\hat{y})$ is the velocity of the fluid.

The boundary conditions for the reactor are

$$\begin{aligned} C(\hat{y}, 1) &= C_{\mathrm{in}} \\ \frac{\partial C(0, \hat{x})}{\partial \hat{y}} &= \mathrm{Da}C_{\mathrm{in}} \\ \frac{\partial C(1, \hat{x})}{\partial \hat{y}} &= 0 \end{aligned} \tag{14.19}$$

where C_{in} is a constant,

$$\mathrm{Da} = \frac{Sh\delta}{DC_{\mathrm{in}}}$$

is the Damkohler number, and S is the metabolic oxygen demand expressed as a volumetric sink with units mol/s/m^3.

Using this model, one can determine if the cells will receive the amount of oxygen that they need for respiration. It is expected that if the oxygen concentration is too low, it will reduce the cellular respiration. We wish to include a term that will model this effect. One function that has the general characteristics that we seek is a Michaelis–Menten term of the form

$$F_{\mathrm{MM}} = \frac{C}{C + C_{\mathrm{MM}}}$$

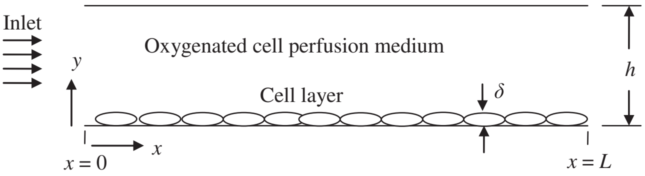


Figure 14.5 Perfusion bioreactor geometry.

[4] *Ibid*, p. 237.

where C_{MM} is the Michaelis–Menten constant that represents the concentration where the term drops to 50% of its original value. The term is combined with the maximum metabolic rate $S_{\max}$ to get the metabolic sink used in the model as

$$S = S_{\max} F_{\mathrm{MM}} = \frac{S_{\max} C}{C + C_{\mathrm{MM}}} \tag{14.20}$$

Then, the boundary condition at $\hat{y} = 0$ becomes

$$\frac{\partial C(0, \hat{x})}{\partial \hat{y}} = \frac{h\delta}{D} \frac{S_{\max} C(0, \hat{x})}{C(0, \hat{x}) + C_{\mathrm{MM}}} = \mathrm{Da}_{\max} \frac{C_{in} C(0, \hat{x})}{C(0, \hat{x}) + C_{\mathrm{MM}}} \tag{14.21}$$

where

$$\mathrm{Da}_{\max} = \frac{S_{\max} h\delta}{DC_{\mathrm{in}}}$$

When the concentration is much higher than C_{MM}, the respiration approaches the maximum value. When the concentration approaches zero, the respiration rate approaches zero and the cells die. This simplified model of cellular respiration is also used in the following two sections.

The perfusion bioreactor model can be solved using `pdepe`. Simplified analytical solutions of this problem are available;[5] however, such solutions cannot include realistic factors such as a parabolic velocity profile and the nonlinear sink term.

Since we shall be using `pdepe` to solve Eq. (14.18), we identify in Tables 14.4 and 14.5 the various elements of Eq. (14.18) and the boundary conditions given by Eqs. (14.19) and (14.21) with those parameters given in Section 5.5.6. In addition, the initial condition given by Eq. (5.11) in the present case is $u_0(x) = C_{\mathrm{in}}$. We shall be considering two types of velocity distributions: (i) constant, called plug flow, and (ii) parabolic. For plug flow, we have that $V(\hat{y}) = V_{\mathrm{avg}}$. For the parabolic velocity distribution, we assume that

$$V(\hat{y}) = V_{\mathrm{avg}} \frac{3}{2}(1 - (2\hat{y} - 1)^2) \tag{14.22}$$

We shall now illustrate these results.

TABLE 14.4 Parameters Used by `pdepe` in Defining the Governing Equation for the Perfusion Bioreactor as Indicated in Section 5.5.6

`pdepe` parameter	Plug flow	Parabolic flow
c	$V_{\mathrm{avg}}h/D$	$V(\hat{y})h/D$, where $V(\hat{y})$is given by Eq. (14.22)
f	$\beta\partial C/\partial\hat{y}$	$\beta\partial C/\partial\hat{y}$
s	0	0

[5] *Ibid,* p. 238.

TABLE 14.5 Parameters Used by `pdepe` in Defining the Boundary Conditions for the Perfusion Bioreactor as Indicated in Section 5.5.6[§]

Boundary	`pdepe` parameter p	`pdepe` parameter q
$\hat{y} = 1$	$\mathrm{pb} = 0$	$\mathrm{qb} = 1$
$\hat{y} = 0$	$\mathrm{pa} = -\mathrm{Da}_{\max} \dfrac{C_{\mathrm{in}} C(0, \hat{x})}{C(0, \hat{x}) + C_{\mathrm{MM}}}$	$\mathrm{qa} = 1/\beta$

[§] `pdepe` defines flux boundary conditions in terms of f as defined in Table 14.4. The parameter q in this table multiplies f and that product is set equal to $-p$ to get the flux boundary condition in Eq. (14.21).

Example 14.5 Perfusion bioreactor

We assume water as the perfusion medium with $L = 5.5$ cm, $h = 0.01$ cm, $V_{\mathrm{avg}} = 0.3$ cm/s, $D = 2 \times 10^{-5}$ cm^2/s, $\mathrm{Da}_{\max} = 0.45$, $C_{\mathrm{in}} = 150 \times 10^{-6}$ M, and $C_{mm} = 10^{-5}$ M. The program that obtains the concentration C for the two different velocity profiles and for the boundary conditions given by Eqs. (14.19) and (14.21) is as follows:

```
function Example14_5
L = 5.5;    h = 0.01;    Vavg = 0.3;    D = 2e-5;
Da = 0.45;    Cin = 150e-6;    Cmm = 10e-6;
Lh = L/h;    hD = h/D;
y = linspace(0, 1, 101);
x = linspace(0, 1, 101);
C = pdepe(0, @pdepb, @pbIC, @pbBC, y, x, [], Lh, hD, Vavg, Da, Cin, Cmm, 1)*1e6;
C2 = pdepe(0, @pdepb, @pbIC, @pbBC, y, x, [], Lh, hD, Vavg, Da, Cin, Cmm, 0)*1e6;
figure(1)
plot(x, C(:,1), 'k-', x, C2(:,1), 'k--')
legend('Plug flow', 'Parabolic flow')
hold on
plot(x, C(:,end), 'k-', x, C2(:,end), 'k--')
xlabel('Distance along reactor (x/L)')
ylabel('Concentration (\muM)')
text(0.19, 60, 'y/h = 0')
text(0.39, 84, 'y/h = 1')
axis([0 1 0 160])
figure(2)
plot(y, C(10,:),'k-', y, C2(10,:),'k--')
legend('Plug flow', 'Parabolic flow', 'Location', 'SouthEast')
hold on
plot(y, C(40,:), 'k-', y, C2(40,:), 'k--')
plot(y, C(1,:), 'k-', y, C(2,:), 'k-', y, C2(2,:), 'k--')
xlabel('Transverse coordinate (y/h)')
ylabel('Concentration (\muM)')
text(0.7, 60, ['x/L = ' num2str(y(41))])
text(0.7, 130, ['x/L = ' num2str(y(11))])
text(0.15, 135, ['x/L = ' num2str(y(2))])
axis([0, 1, 0, 160])
```

```
function [c, f, s] = pdepb(y, x, C, DCDy, Lh, hD, Vavg, Da, Cin, Cmm, plug)
if plug == 0
  c = Vavg*1.5*(1-(2*y-1).^2)*hD;
else
  c = Vavg*hD;
end
f = DCDy*Lh;
s = 0;

function Czero = pbIC(y, Lh, hD, Vavg, Da, Cin, Cmm, plug)
Czero = Cin;

function [pa, qa, pb, qb] = pbBC(yl, Cl, yr, Cr, x, Lh, hD, Vavg, Da, Cin, Cmm, plug)
pb = 0;   qb = 1;
pa = -Da*Cin*Cl/(Cl+Cmm);
qa = 1/Lh;
```

When the program is executed, we obtain Figures 14.6 and 14.7. Figure 14.6 is a plot of oxygen concentration in the perfusion media versus the distance into the reactor at both walls of the reactor. The solution was run for a uniform flow velocity profile across the reactor and for a parabolic velocity, which might be expected in fully developed flow. It is noted that the velocity profile does not have a large impact on the concentration distribution. In Figure 14.7, we have plotted concentration across the reactor. As expected, the slope of the concentration profile is higher for the parabolic profile, but the overall effect is minimal.

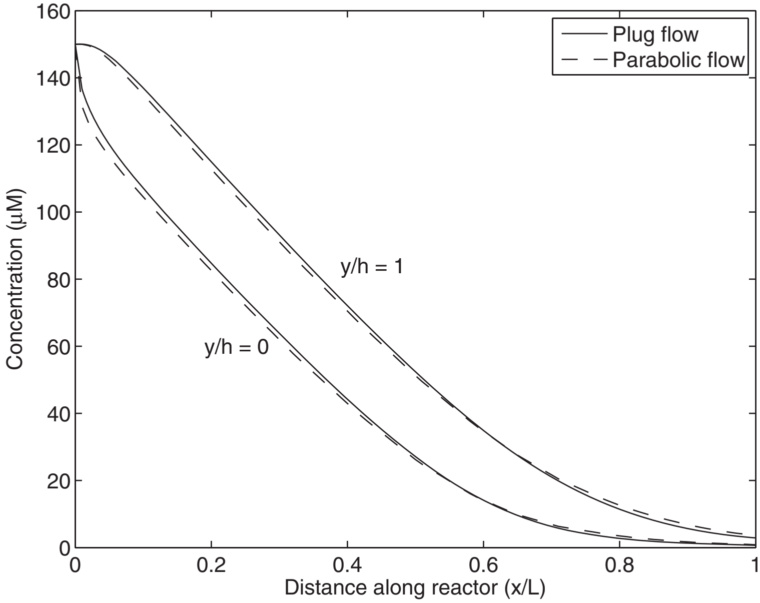


Figure 14.6 Oxygen concentration versus position along reactor at both walls for two different velocity profiles. The upper pair of curves represents the concentration at the upper boundary, that is, away from the cells, while the lower curves represent the fluid adjacent to the cells.

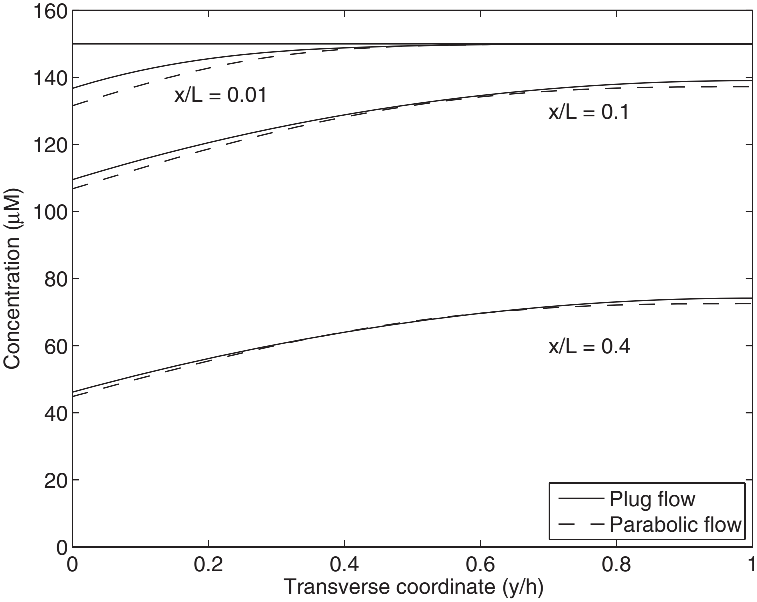


Figure 14.7 Oxygen concentration versus the transverse coordinate $\hat{y}$ at $\hat{x}$ = 0.01, 0.1, and 0.4. The horizontal line at $C = 150\ \mu$M represents the inlet concentration C_{in}.

14.2.5 Supply of Oxygen to a Spherical Tumor

When a tumor first forms, it is a bundle of cells without any vasculature. Thus, the oxygen that the tumor cells need to grow must be supplied by diffusion. We shall model this diffusion mass transfer as occurring in a symmetrical spherical geometry that can be represented by the following equation:

$$\frac{d^2X}{dr^2} + \frac{2}{r}\frac{dX}{dr} = \frac{S}{\rho D} \tag{14.23}$$

where $X = X(r)$ is the mass fraction of oxygen, r is the radial coordinate of a sphere, D is the mass diffusion coefficient, ρ is the density, and S is the volumetric sink of oxygen, that is, the rate of oxygen consumption by cell metabolism. The metabolic sink term represents the rate at which cells utilize oxygen. If the oxygen concentration goes to zero, the cells will die and the metabolic rate goes to zero. Thus, it seems reasonable to model the metabolic rate as

$$S = S_{\max}\frac{X}{K + X}$$

where $S_{\max}$ is the maximum metabolic rate. The form of this expression follows Michaelis–Menten,[6] that is, it goes to zero as the concentration goes to zero and achieves 50% of the maximum rate when $X = K$.

The boundary condition at the outer surface of the tumor $r = R$ is that the oxygen mass fraction is equal to the oxygen mass fraction in the blood. The boundary condition at $r = 0$ is that the slope of oxygen mass fraction with respect to the spatial coordinate must be zero. Thus,

$$X(R) = X_b \tag{14.24}$$
$$\left.\frac{dX}{dr}\right|_{r=0} = 0$$

where X_b is oxygen mass fraction at the boundary. Equations (14.23) and (14.24) can be transformed into a nondimensional form by introducing the following quantities:

$$\chi = \frac{r}{R}, \quad \psi = \frac{X}{X_b}, \quad \Sigma = \frac{S_{\max} R^2}{\rho D X_b}$$

Thus, Eq. (14.23) becomes

$$\frac{d^2\psi}{d\chi^2} + \frac{2}{\chi}\frac{d\psi}{d\chi} = \frac{\psi\Sigma}{K/X_b + \psi} \tag{14.25}$$

and Eq. (14.24) becomes

$$\psi(1) = 1 \quad \left.\frac{d\psi}{d\chi}\right|_{\chi=0} = 0 \tag{14.26}$$

Equations (14.25) and (14.26) can be numerically solved with `bvp4c`.

Example 14.6 Oxygen diffusion in a small tumor

We shall consider a tumor with an outer radius $R = 0.02$ cm, a diffusion coefficient $D = 2 \times 10^{-5}\,\text{cm}^2/\text{s}$, a density $\rho = 1\,\text{gm/cm}^3$, and a maximum metabolic rate $S_{\max} = 15\,\mu\text{M/s}$ ($15 \times 10^{-9}\,\text{mol/s/cm}^3$). At the outer surface, we assume for O_2 that $X_b = 5.234 \times 10^{-8}$, which corresponds to an oxygen partial pressure of 40 mmHg. The value of K in the Michaelis–Menten expression is[7] $K = 25 \times 10^{-10}$.

[6] *Ibid*, p. 324.

[7] Obtained from G. A. Truskey, Y. Fan, and D. F. Katz, *Transport Phenomena in Biological Systems*, Pearson Prentice Hall, Upper Saddle River, NJ, 2004, p. 622, and converted to mass fraction of O_2.

We decompose Eq. (14.25) into two coupled first-order equations by the introduction of two new dependent variables:

$$y_1 = \psi, \quad y_2 = \frac{d\psi}{d\chi}$$

which leads to

$$\frac{dy_1}{d\chi} = y_2$$

$$\frac{dy_2}{d\chi} = -\frac{2}{\chi}y_2 + \Sigma$$

The corresponding boundary conditions are

$$y_1(1) = 1$$

$$y_2(0) = 0$$

The second boundary condition causes some difficulties in the numerical solution because the governing equation has terms that are divided by χ. To overcome this difficulty, we instead consider a hollow sphere with a very small inner radius of $\chi = 0.0005$. The solution to this model is as follows:

```
function Example14_6
R = 0.02;    Xb = 5.234e-8;    Smax = 15e-9;
D = 2e-5;    rho = 1;    K = 25e-10;
S = Smax*R^2/(D*Xb*rho);
KKb = K/Xb;
chiR = 1.0;    chi0 = 0.0005;
solinit = bvpinit(linspace(chi0, chiR, 5), [0 0]);
sol = bvp4c(@TumorODE, @TumorBC, solinit, [], S, KKb);
chi = linspace(chi0, chiR, 50);
y = deval(sol, chi);
subplot(2,1,1)
plot(chi, y(1,:), 'k-')
ylabel('\psi')
xlabel('\chi')
subplot(2,1,2)
plot(chi, y(2,:), 'k-')
ylabel('d\psi/d\chi')
xlabel('\chi')

function dy = TumorODE(chi, y, S, KKb)
dy = [y(2); S*y(1)/(KKb+y(1))-2/chi*y(2)];

function res = TumorBC(ya, yb, S, KKb)
res = [ya(2);    yb(1)-1];
```

When the program is executed, we obtain Figure 14.8. The top pane is the dimensionless oxygen mass fraction plotted against the radial position. As is seen, the oxygen level drops until it reaches the lowest value at the center of the sphere. The fact that the oxygen level is close to zero indicates that the tumor is close to the largest dimension that can be sustained without an integrated vasculature.

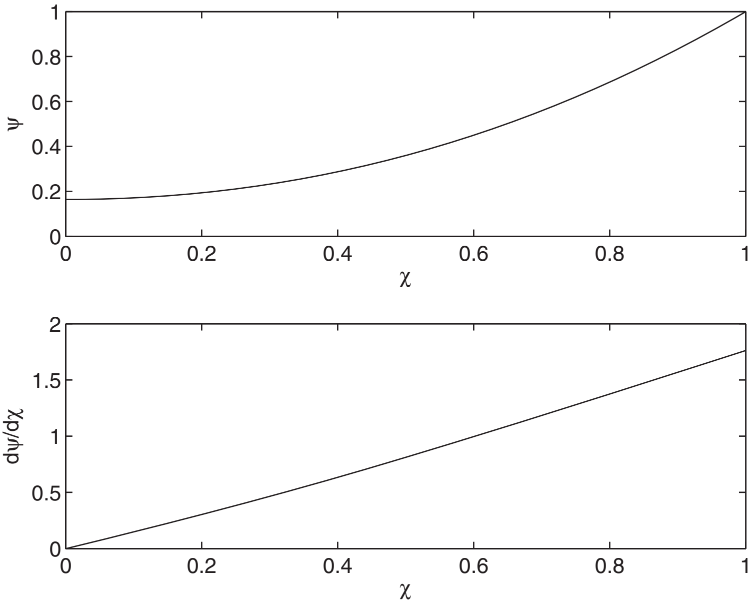


Figure 14.8 Oxygen mass fraction and slope in small spherical tumor without vasculature.

14.2.6 Krogh Cylinder Model of Tissue Oxygenation

The oxygen requirements of tissue are similar to those of the tumor described in Section 14.2.5. In the more general case, oxygen is provided to tissue through a complex branched vascular structure cascading down in scale to capillaries, which have typical dimensions of 10 μm inner diameter and lengths of 1 mm. Capillaries have a porous wall structure that allows nutrients and waste products to transfer between cells and the blood. Due to the relatively low-mass diffusivity of oxygen in the tissue and the mass transfer resistance of the capillary wall, there is only a limited region around the capillary that can be effectively supplied with oxygen. The cylindrical geometry of the capillary can supply a tissue cylinder with the capillary forming the axis. This simplified geometry of tissue vascularization, shown schematically in Figure 14.9, is called the Krogh cylinder model.

Blood flows into the capillary from an artery and then empties into the venous system. The inflowing blood is oxygenated after passing through the lungs. By way of the capillaries, the blood with high oxygen concentration is brought in contact with tissue with a low oxygen concentration. This difference in concentration drives diffusion mass transfer from the blood through the capillary wall and to the tissue. Thus, a model of this process should include both convective supply of oxygen to the capillary and transport by diffusion through the capillary wall and tissue.

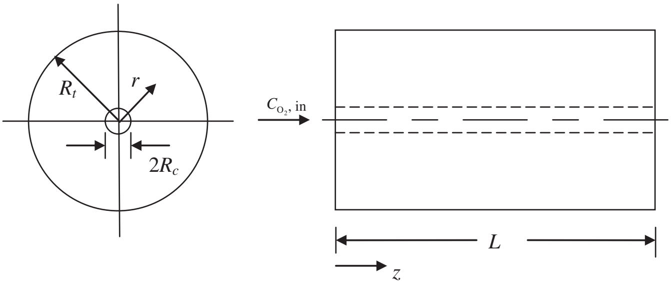


Figure 14.9 Krogh cylinder with a capillary at the center that is surrounded by a cylinder of tissue.

The convective process in the capillary can be modeled as[8]

$$V\frac{\partial C_{T,O_2}}{\partial C_{O_2}}\frac{\partial C_{O_2}}{\partial z} = D_c\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial C_{O_2}}{\partial r}\right) \quad 0 \le r \le R_c \tag{14.27}$$

where $C_{O_2} = C_{O_2}(r, z)$ is the concentration of dissolved oxygen, $C_{T,\,O_2}$ is the concentration of total oxygen including that bound to hemoglobin, $V = V(r)$ is the axial fluid velocity, D_c is the mass diffusion coefficient in the capillary, z is the axial coordinate, and r is the radial coordinate. It should be noted that axial diffusion is ignored in this model.

The concentration of total oxygen $C_{T,\,O_2}$, including that bound to hemoglobin, is given by Eq. (14.17) with the following change in notation. The concentration C_{O_2} is given by Eq. (14.15), that is,

$$C_{O_2} = [O_2] = \frac{P_{O_2}}{H_{O_2}}$$

Then, Eq. (14.14) can be written as

$$Y = \frac{(P_{O_2}/H_{O_2})^n}{(P_{50}/H_{O_2})^n + (P_{O_2}/H_{O_2})^n} = \frac{C_{O_2}^n}{C_{50}^n + C_{O_2}^n}$$

where

$$C_{50} = \frac{P_{50}}{H_{O_2}}$$

Therefore, Eq. (14.17) becomes

$$C_{T,O_2} = [O_2]_T = [O_2]_d + [O_2]_{\mathrm{Hb}} = C_{O_2} + \frac{4[\mathrm{Hb}]C_{O_2}^n}{C_{50}^n + C_{O_2}^n} \tag{14.28}$$

Differentiating Eq. (14.28) with respect to C_{O_2}, we obtain

$$\frac{\partial C_{T,O_2}}{\partial C_{O_2}} = 1 + \frac{4n[\mathrm{Hb}]Y^2C_{50}^n}{C_{O_2}^{n+1}} \tag{14.29}$$

[8] Fournier, *Basic Transport Phenomena*, p. 240.

This relationship between the total oxygen concentration, including both dissolved and bound oxygen, and the dissolved oxygen is taken from the discussion in Section 14.2.3. The large storage capacity of the hemoglobin means that a small flow rate of blood carries a large oxygen supply. Equation (14.29) is used in Eq. (14.27).

The oxygen concentration at the inlet is assumed to be a known constant value $C_{O_2,\text{in}}$ so that

$$C_{O_2}(r, 0) = C_{O_2,\text{in}} \tag{14.30a}$$

The centerline of the capillary is assumed to be an axis of symmetry such that

$$\frac{\partial C_{O_2}(0, z)}{\partial r} = 0 \tag{14.30b}$$

A boundary condition at the outer surface of the capillary is needed, but its specification is deferred until after the tissue model is introduced.

The diffusion of oxygen in the tissue is modeled as radial diffusion in a cylinder with a volumetric sink S, which represents the metabolic rate of oxygen uptake by the cells. Axial diffusion is ignored for simplicity, guided by a symmetry argument that predicts zero diffusion flux between adjoining Krogh cylinders. The governing equation in the tissue is given by[9]

$$\frac{d^2\hat{C}_{O_2}}{dr^2} + \frac{1}{r}\frac{d\hat{C}_{O_2}}{dr} = \frac{S}{D_t} \qquad R_c \leq r \leq R_t \tag{14.31}$$

where $\hat{C}_{O_2} = \hat{C}_{O_2}(r)$, D_t is the mass diffusion coefficient in the tissue, and from Eq. (14.20)

$$S = \frac{S_{\text{max}}\hat{C}_{O_2}}{\hat{C}_{O_2} + C_{\text{MM}}}$$

The boundary condition at the outer limit of the tissue cylinder $r = R_t$ is that the flux is zero, that is,

$$\left.\frac{\partial \hat{C}_{O_2}}{\partial r}\right|_{r=R_t} = 0 \tag{14.32}$$

We now couple these two regions by specifying what is taking place at the capillary wall $r = R_c$, which is common between the tissue and the blood. Oxygen can diffuse readily through the capillary wall. Thus, for simplicity, we ignore the resistance of the wall and require that there is continuity of concentration and flux at the boundary, that is,

$$\begin{gathered} C_{O_2}(R_c) = \hat{C}_{O_2}(R_c) \\ D_c\left.\frac{\partial C_{O_2}}{\partial r}\right|_{r=R_c} = D_t\left.\frac{\partial \hat{C}_{O_2}}{\partial r}\right|_{r=R_c} \end{gathered} \tag{14.33}$$

[9] *Ibid*, p. 243.

It is seen from the formulation above that we are solving a conjugate problem with two different domains each with its own physics and coupled by the continuity conditions at the capillary wall $r = R_c$.

Example 14.7 Krogh cylinder model with a parabolic blood velocity profile

Consider a Krogh cylinder model of a capillary with length 1 mm and diameter 10 μm, thus, $L = 0.001$ m and $R_c = 0.00001$ m. In addition, the outer diameter of the tissue cylinder is 100 μm; thus, $R_t = 0.0001$ m. We shall assume an oxygen mass diffusion coefficient $D_c = 2 \times 10^{-9}\,\text{m}^2/\text{s}$ in the capillary blood and $D_t = 1 \times 10^{-9}\,\text{m}^2/\text{s}$ in the tissue. The blood velocity is assumed to have a parabolic profile of the form

$$V(r) = V_{\max}\left(1 - (r/R_c)^2\right) \tag{14.34}$$

with a centerline velocity $V_{\max} = 0.0005$ m/s. In addition, we assume that $n = 2.34$, $C_{50} = 35.14 \times 10^{-6}$ M, [Hb] = 0.0022 M, $C_{\text{in}} = 120 \times 10^{-6}$ M, $S_{\max} = 20 \times 10^{-6}$ M, and $C_{\text{MM}} = 10 \times 10^{-6}$ M.

We use pdepe to solve Eq. (14.27) and bvp4c to solve Eq. (14.31). The matching conditions of concentration and flux, as given by Eq. (14.33), are imposed by utilizing the current value of wall concentration as an input to bvp4c, which returns the flux; the flux, in turn, is then fed back to pdepe. In this way, both conditions are imposed on the common boundary. The function pdepe calls the boundary condition function (which contains bvp4c) many times as it iterates to solve the boundary value problem in r at each step of the integration along z. Iteration by the pdepe solver is required because the final solution must match two conditions at the boundary; however, the two boundary value representations, one for each side of the common boundary, require a single condition at that location. On the first call to the boundary condition function an initial guess is used for the temperature; this allows computation of the tissue problem using bvp4c resulting in a heat flux at the capillary wall. This heat flux is then used by pdepe to solve the boundary value problem in r, which results in an updated value for the boundary temperature. This iterative process continues until both conditions are approached within the solution tolerances. The end result of this coupled solution of the conjugate problems in the blood and the tissue is that the governing equations are satisfied in both domains and that the matching conditions at the capillary wall are satisfied. Finally, it was found that the convergence criteria for pdepe had to be tightened to obtain a smooth and reasonably accurate solution.

The program that performs the calculations as described above is as follows. The various functions that are used by this program and their respective purposes are listed in Table 14.6.

```
function Example14_7
Rc = 10e-6;   Rt=100e-6;   Dc = 2e-9;   Dt = 1e-9;
Vmax = 0.0005;   CHb = 0.0022;   Cin = 120e-6;
Smax = 20e-6;   Cmm = 10e-6;
z = linspace(0, 0.001, 101);
r = linspace(0, 10e-6, 101);
options = odeset('RelTol', 1e-6, 'AbsTol', 1e-9);
C = pdepe(1, @CapillPDE, @CapillIC, @CapillBC, r, z, options, . . .
            Rc, Rt, Dc, Dt, Vmax, CHb, Cin, Smax, Cmm);
```

TABLE 14.6 Functions Used in Krogh Cylinder Model

Function name	Description
CapillPDE	Defines coefficients of the partial differential equation for pdepe for the blood flow model, as given by Eqs. (14.27) and (14.29)
CapillIC	Defines the inlet condition for the blood concentration for pdepe, as given by Eq. (14.30a)
CapillBC	Defines the boundary conditions for pdepe and calls bvp4c to implement the tissue model. Sets the mass flux at the capillary wall equal to the flux calculated by bvp4c in the tissue, which is given by Eq. (14.33)
TissueODE	Defines the ordinary differential equation for bvp4c for the tissue model, as given by Eq. (14.31)
TissueBC	Defines the boundary conditions for bvp4c and sets the tissue concentration equal to the blood concentration at the capillary wall, as given by Eqs (14.32) and (14.33)

```
figure(1)
plot(1000*z, 1e6*C(:,1), 'k-', 1000*z, 1e6*C(:,end),'k--')
legend('Centerline','Wall')
xlabel('z (mm)')
ylabel('C (\muM)')
ylim([0 130])
figure(2)
zv = [1 26 101];
solinit = bvpinit(linspace(Rc, Rt, 10), [0 0]);
rt = linspace(Rc, Rt, 101);
for n = 1:length(zv)
   plot(1000*r, C(zv(n),:)*1e6, '--k')
   hold on
   sol = bvp4c(@TissueODE, @TissueBC, solinit, [], C(zv(n), end), Dt, Smax, Cmm);
   y = deval(sol, rt);
   plot(1000*rt, 1e6*y(1,:), 'k-')
   text(0.001, C(zv(n),1)*1e6+3, ['z = ' num2str(1000*z(zv(n))) ' mm'])
   if n == 1
      legend('Capillary', 'Tissue')
      flux = Dt*y(2,1);
   end
end
plot([0.01 0.01], [0 130], 'k:')
xlabel('r (mm)')
ylabel('C (\muM)')
ylim([0 130])
figure(3)
for n = 1:101
   [Cc(n) dCdr(n)] = pdeval(1, r, C(n,:), Rc);
end
plot(1000*z, [-flux, -Dc*dCdr(2:end)]*1E9, 'k-')
ylabel('Flux (nmol/s-m^2)')
```

```
xlabel('z (mm)')
ylim([0 7])

function [c, f, s] = CapillPDE(r, z, C, DCDr, Rc, Rt, Dc, Dt, Vmax, CHb, Cin, ...
                               Smax, Cmm)
n = 2.34;    C50 = 35.14e-6;
Y = C^n/(C50^n+C^n);
dCtdC = 1+4*n*CHb*Y^2*C50^n/C^(n+1);
c = dCtdC*Vmax*(1-(r/Rc)^2);
f = Dc*DCDr;
s = 0;

function C0 = CapillIC(r, Rc, Rt, Db, Dt, Vmax, CHb, Cin, Smax, Cmm)
C0 = Cin;

function [pa, qa, pb, qb] = CapillBC(rl, Cl, rr, Cr, z, Rc, Rt, Dc, Dt, Vmax, CHb, ...
                                     Cin, Smax, Cmm)
solinit = bvpinit(linspace(Rc, Rt, 10), [0 0]);
sol = bvp4c(@TissueODE, @TissueBC, solinit, [], Cr, Dt, Smax, Cmm);
pb = -Dt*sol.y(2,1);    qb = 1;
pa = 0;    qa = 1;

function res = TissueBC(ya, yb, Cr, Dt, Smax, Cmm)
res = [ya(1)-Cr;  yb(2)];

function dydx = TissueODE(r, y, Cr, Dt, Smax, Cmm)
dydx = [y(2); Smax/Dt*y(1)/(y(1)+Cmm)-y(2)/r];
```

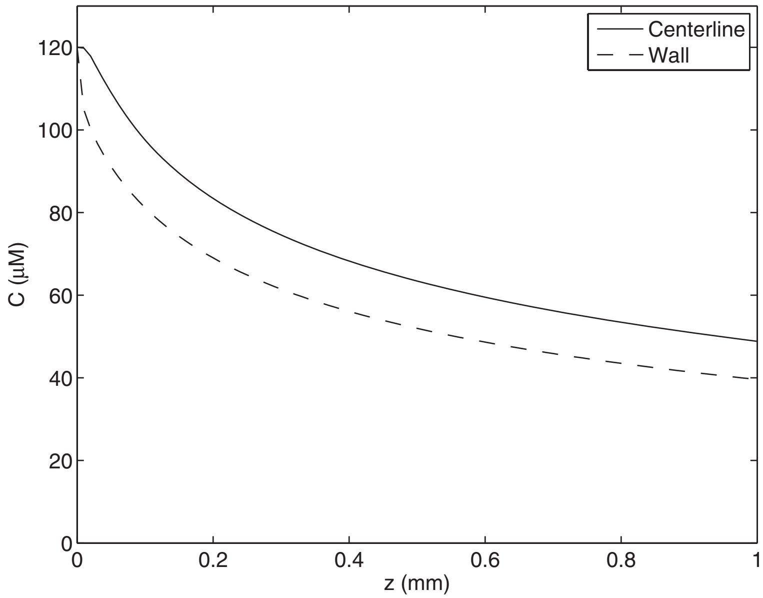


Figure 14.10 Oxygen concentration in blood versus axial position in capillary at centerline and at the wall for the Krogh cylinder model. The concentration difference shown is the mass transfer potential that drives the oxygen out of the blood.

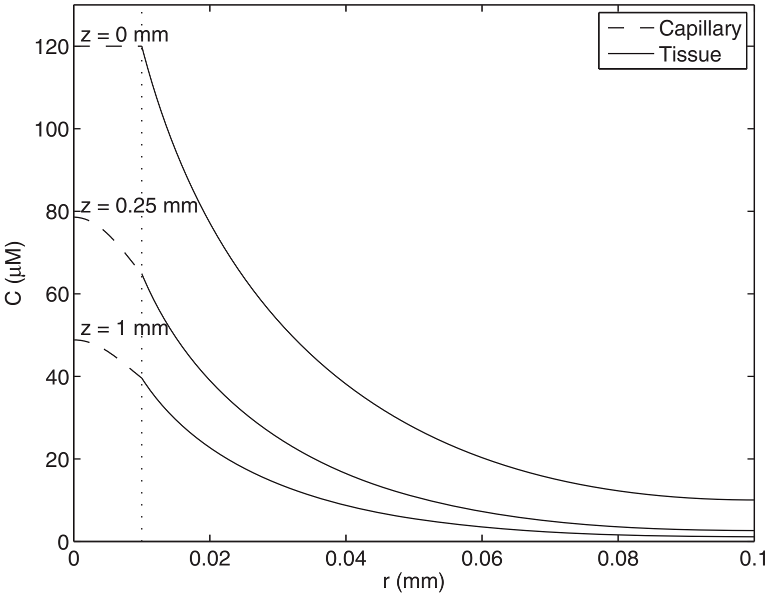


Figure 14.11 Oxygen concentration versus radial position in Krogh cylinder model at three axial locations. In the range $0 < r < 0.01$ mm, the data represent blood concentration. In the range $0.01 < r < 0.1$ mm, the data represent tissue concentration. At the capillary wall, the two concentrations are equal. The vertical line at $r = 0.01$ mm is the location of the capillary wall.

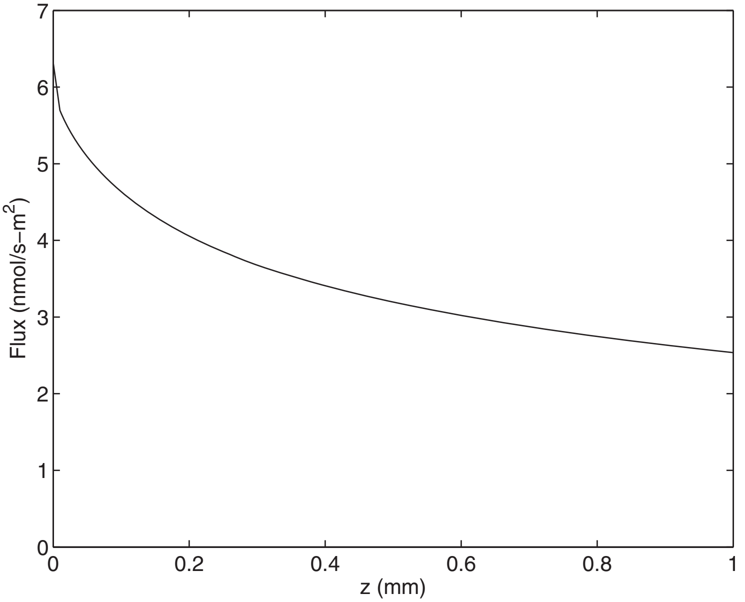


Figure 14.12 Mass flux versus axial position for the Krogh cylinder model.

Execution of the program produces Figures 14.10–14.12. As shown in Figure 14.10, the oxygen concentration falls as the blood travels through the capillary due to the oxygen transferring out of the blood to meet the oxygen requirements in the tissue. The concentration falls more rapidly near the inlet of the capillary because the flux is higher there due to the higher concentration.

The radial concentration profiles at several axial locations are shown in Figure 14.11. The continuity of concentration across the capillary wall is evident at $r = 0.01$ mm . It is noted that there is a change in the slope of the concentration at the wall due to the different values of mass diffusivity in the two adjoining domains. It is also evident that parts of the tissue are not getting an adequate supply of oxygen. When the oxygen concentration drops below a certain value, the metabolism of the cells slows down. This effect is approximated in the model by a Michaelis–Menten modification of the metabolic sink term with $C_{\mathrm{MM}} = 10\ \mu\mathrm{M}$. Thus, when the oxygen concentration drops to 10 μM, the metabolic rate goes to one-half of its maximum value. The model allows one to obtain an estimate of the effective diameter of the Krogh cylinder for a particular capillary operating point, including flow rate, diameter, and inlet concentration. The mass flux through the capillary wall is plotted as a function of axial position in Figure 14.12.

14.3 CHARGE TRANSPORT IN BIOLOGICAL SYSTEMS

14.3.1 Hodgkin–Huxley Neuron Model

Hodgkin and Huxley[10] performed a series of groundbreaking measurements on the giant axon of a squid. These experiments led to a model that explained the physics of communication along neurons. The model is based on a membrane circuit diagram, shown in Figure 14.13. The path at the far right represents the capacitance of the membrane. It is here that energy is stored when the membrane potential difference changes. Energy is stored in the form of a layer of cations on the extracellular side and a layer of anious on the cytosolic side of the membrane. Each of the other three branches in Figure 14.13 represents a separate ion flow.

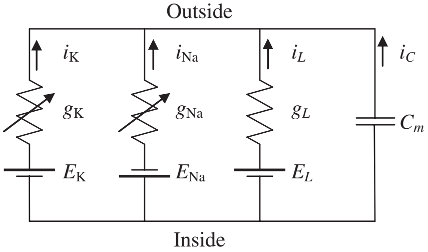


Figure 14.13 Hodgkin–Huxley circuit model.

[10] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *Journal of Physiology*, **117**(4), 1952, pp. 500–544.

The one on the far left is potassium, the second from the left is sodium, and the remaining one is termed the leakage current that represents chloride and all other ion flows combined. In each ion current path, there is a conductance and a source, which is represented by a battery symbol.

For the potassium and sodium currents, the conductance is variable. The ion channels that allow these currents to flow are voltage-gated channels that open and close in response to changes in the potential difference across the membrane. Each type of channel has a unique gating characteristic, as described in Section 14.3.2.

The sources that are shown in each of the ion current legs represent the diffusion potential of each of the ions. The biological membrane has active transporters that create ion concentration differences across the membrane, resulting in a membrane potential difference that is approximately −65 mV for a resting cell. The ion channels are very specific to individual ions such that a potassium channel will not pass sodium (or vice versa). Thus, when a specific ion channel opens, ions flow through the channel driven by a combination of two forces: the electrical driving force caused by the potential difference and the diffusion driving force. The source accounts for the diffusion driving force.

The leakage term is included to account for ion currents other than potassium and sodium. The leakage current is generally small and can be neglected in a first-order analysis. The values of the leakage conductance and the source were determined empirically as a best match to a wide range of experiments.

14.3.2 Hodgkin–Huxley Gating Parameters

The gating parameters in the Hodgkin–Huxley model[11,12] are given the symbols n, m, and h; these parameters can adopt values between 0 and 1. The potassium channel conductance depends on the parameter n as

$$g_{\mathrm{K}} = \bar{g}_{\mathrm{K}} n^4 \tag{14.35}$$

where $\bar{g}_{\mathrm{K}}$ is the maximum potassium conductance. The fourth power dependence on n was chosen by the developers to best match the data. However, in the light of recent advances in understanding the structure of the potassium channel, the fourth power dependence is often thought to be associated with a separate gating function associated with each of the four amino terminal ends of the protein molecules in the quaternary structure of the channel. The sodium channel gating was modeled empirically as

$$g_{\mathrm{Na}} = \bar{g}_{\mathrm{Na}} m^3 h \tag{14.36}$$

The three gating parameters n, m, and h are assumed to follow first-order kinetics. Using n as an example, we first note that n can be interpreted as the probability

[11] *Ibid.*

[12] B. Hille, *Ion Channels of Excitable Membranes*, 3rd ed., Sinauer Associates, 2001.

that the gate will be in the open state. A first-order reaction between the open and closed states can be written as

$$(1 - n) \underset{\beta_n}{\overset{\alpha_n}{\rightleftarrows}} n$$

where α_n is the forward rate constant and β_n the reverse rate constant. This results in a first-order equation for n of the form

$$\frac{dn}{dt} = \alpha_n(1 - n) - \beta_n n \tag{14.37}$$

where t is the time. This is the equation that is used to determine the dependence of n on t. Similar equations are used for m and h by replacing n with m or h.

An alternative form to this first-order equation can be obtained by algebraic manipulation. First, we define alternative parameters n_∞ and τ_n in terms of to α_n and β_n as

$$\begin{aligned} n_\infty &= \frac{\alpha_n}{\alpha_n + \beta_n} \\ \tau_n &= \frac{1}{\alpha_n + \beta_n} \end{aligned} \tag{14.38}$$

Equations similar to Eq. (14.38) can be written for m and h by replacing n with m or h. Then the differential equation for n becomes

$$\frac{dn}{dt} = \frac{n_\infty - n}{\tau_n} \tag{14.39}$$

Equations (14.37) and (14.39) are equivalent so that the choice of which one to use depends only on convenience. Historically, the parameters extracted from experiments were n_∞ and τ_n, but these were converted to α_n and β_n for purposes of the following correlations of the three gating parameters. The correlations for n, m, and h are given in terms of the membrane depolarization potential defined as

$$V_d = V - V_r \tag{14.40}$$

where V is the membrane potential difference, which is the difference between the cytosol potential and the extracellular potential, and V_r is the resting value of V, typically around –65 mV. The correlations are tabulated in Table 14.7. Since we will use the expressions given in Table 14.7, we shall create two function M files for the α's and β's as follows:

```
function a = alph(Vd, g)
switch g
  case 'n'
    if Vd == 10
      a = 0.1;
    else
```

TABLE 14.7 Correlation of the Gating Parameters

Gating parameter	α	β
n	$\alpha_n(V_d) = \dfrac{0.01(10 - V_d)}{e^{(10-V_d)/10} - 1}$	$\beta_n(V_d) = 0.125e^{-V_d/80}$
m	$\alpha_m(V_d) = \dfrac{0.1(25 - V_d)}{e^{(25-V_d)/10} - 1}$	$\beta_m(V_d) = 4e^{-V_d/18}$
h	$\alpha_h(V_d) = 0.07e^{-V_d/20}$	$\beta_h(V_d) = \dfrac{1}{e^{(30-V_d)/10} + 1}$

```
            a = 0.01*(10-Vd)./(exp((10-Vd)/10)-1);
        end
    case 'm'
        if Vd == 25
            a = 1;
        else
            a = 0.1*(25-Vd)./(exp((25-Vd)/10)-1);
        end
    case 'h'
            a = 0.07*exp(-Vd/20);
end

function b = bet(Vd, g)
switch g
    case 'n'
        b = 0.125*exp(-Vd/80);
    case 'm'
        b = 4*exp(-Vd/18);
    case 'h'
        b = 1./(exp((30-Vd)/10)+1);
end
```

Example 14.8 Display of Hodgkin–Huxley gating parameters

We shall evaluate and display the expressions given in Table 14.7 and the expression that results from their substitution in Eq. (14.38). The program is as follows:

```
Vr = -65;   V = linspace(-100, 75, 100);
Vd = V-Vr;   gate = char('n', 'm', 'h');
mul = [0.1, 1, 0.1];
for k = 1:3
    figure(k)
    subplot(2,1,1)
    a = alph(Vd, gate(k));
    b = bet(Vd, gate(k));
    plot(V, a./(a+b), 'k-', V, mul(k)./(a+b),'k--')
    legend([gate(k) '_\infty'], [num2str(mul(k)) '*\tau_' gate(k) ' (ms)'], ...
        'Location', 'East')
    subplot(2,1,2)
    plot(V, a, 'k—', V, b,'k-')
    xlabel('Membrane potential (mV)')
    legend(['\alpha_' gate(k) ' (ms)^{-1}'], ['\beta_' gate(k) ' (ms)^{-1}'], ...
            'Location', 'East')
    if strcmp(gate(k), 'm') == 1
      ylim([0 10])
    else
      ylim([0 1])
    end
  end
```

Upon execution, we obtain the graphs shown in Figures 14.14–14.16. As can be seen in these figures, the parameters n and m are activated by depolarization, that is, when the membrane voltage increases. On the other hand, h is deactivated by depolarization. This can be understood by noting that n_∞ and m_∞ increase asymptotically to one as the membrane potential increases, whereas h_∞ decreases toward zero. Thus, the parameter n causes the potassium channel to open upon membrane depolarization; it remains open until repolarization occurs. In contrast, the sodium channel is dependent on both m and h. Parameter m behaves similarly to n, although the sodium channel opens later than the potassium channel. However, the sodium channel shuts down after a short open duration through the influence of h. The combination of these

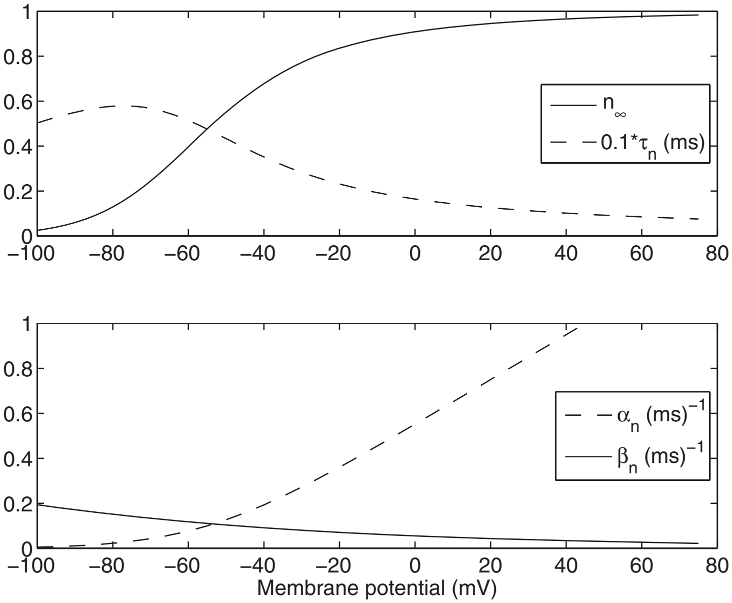


Figure 14.14 Gating parameters for the potassium channel gating function n plotted as a function of membrane potential V with $V_r = -65$ mV. The representations in the upper and lower graphs are related by Eq. (14.38).

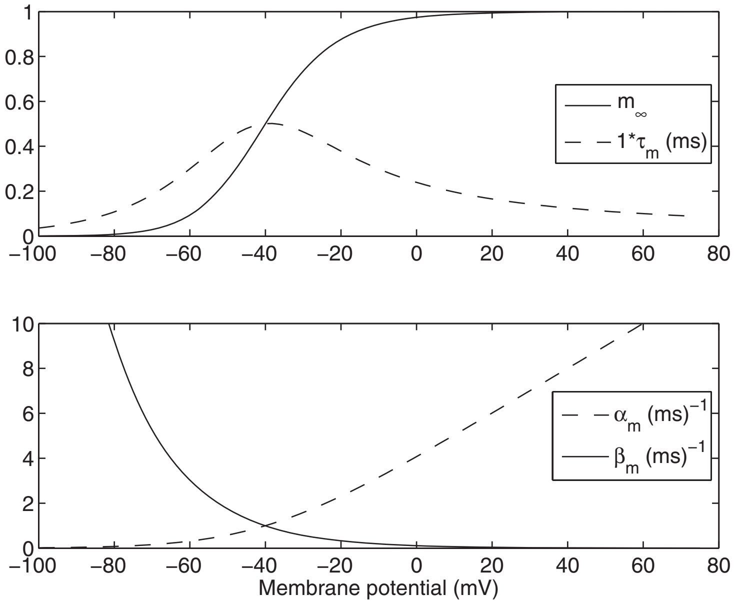


Figure 14.15 Gating parameters for the sodium channel gating function m plotted as a function of membrane potential V with $V_r = -65$ mV. The equivalent representations in the upper and lower graphs are related by Eq. (14.38) with n replaced by m.

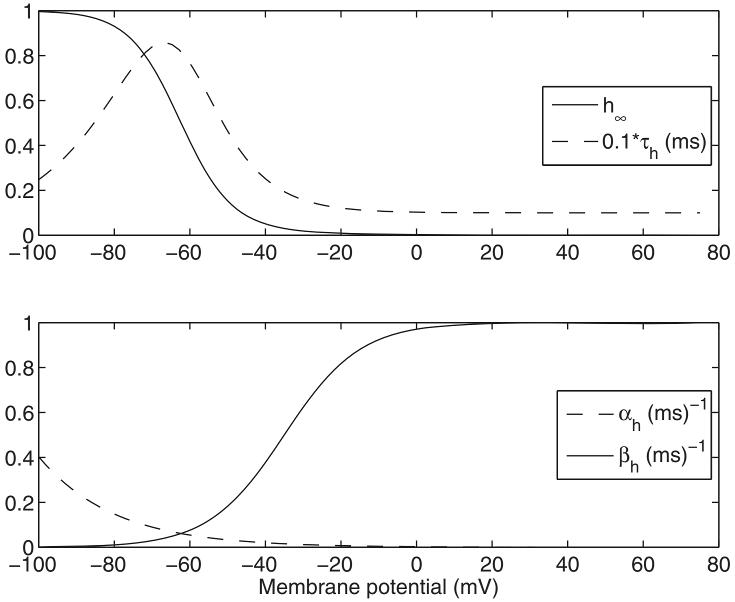


Figure 14.16 Gating parameters for the sodium channel gating function h plotted as a function of membrane potential V with $V_r = -65$ mV. The representations in the upper and lower graphs are related by Eq. (14.38) with n replaced by h.

effects produces complex electrical phenomena such as the action potential, which is discussed in Section 14.3.4. The first step to understanding these phenomena is to understand the gating parameters in Figures 14.14–14.16. It is noted that these gating parameters were obtained by empirical matching to a series of experimentally obtained results on the squid giant axon, which is much larger than a typical axon and thus provided the best experimental vehicle for early researchers. Subsequently, miniaturized instrumentation has been developed and similar characteristics have been observed in other systems.

14.3.3 Hodgkin–Huxley Model with Step Function Input

The gating parameters introduced in Section 14.3.2 define the membrane conductance in terms of the membrane potential difference. Thus, if the membrane potential difference is known as a function of time, then the conductance can be determined by integrating the time-dependent differential equations for the gating parameters, that is, equations of the form of Eqs. (14.37) and (14.39). This calculation is important since in the laboratory it is possible to set the membrane potential difference with an experimental apparatus called a voltage clamp. Such experiments formed the core empirical data that led to the development of the Hodgkin–Huxley model.

One such experiment involves a step change in the membrane potential. In this experiment, the membrane potential is controlled and the membrane current is measured. The contributions to the current from individual ion channels can be determined by using various means to block or disable each of the channels. This blocking can be done by chemical or physical means. The end result is the measurement of the total current and the contributions to that current from each of the ion channels. That information enabled Hodgkin and Huxley to curve fit the parameters n, m, and h given in Section 14.3.2. Once those parameters are known, one can mimic the voltage step experiment computationally. The equations that govern these parameters are (recall Eq. (14.37))

$$\begin{aligned}\frac{dn}{dt} &= \alpha_n - (\alpha_n + \beta_n)n \\ \frac{dm}{dt} &= \alpha_m - (\alpha_m + \beta_m)m \\ \frac{dh}{dt} &= \alpha_h - (\alpha_h + \beta_h)h\end{aligned} \tag{14.41}$$

Example 14.9 Step input to Hodgkin–Huxley model

We shall solve Eq. (14.41) and plot the results when the excitation is the change in the membrane potential V, which for this example is a potential that is held for 2 ms at –65 mV and then suddenly increased to 0 mV for 3 ms before returning to –65 mV. The initial conditions for n, m, and h are, respectively, $n(0) = 0.3177$, $m(0) = 0.0529$, and $h(0) = 0.5961$. These initial conditions are the steady-state values obtained from the model with a membrane potential of –65 mV. The program is as follows:

```
function Example14_9
gate = char('n', 'm', 'h');
tspan = linspace(0, 10, 300);
IC = [0.3177, 0.0529, 0.5961];
subplot(4,1,1)
plot([0, 2, 2, 5, 5, 10], [-65, -65, 0, 0, -65, -65], 'k-')
ylim([-100, 50])
for k = 1:3
  [t, y] = ode45(@HH, tspan, [IC(k)], [], gate(k));
  subplot(4,1,k+1)
  switch gate(k)
    case 'n'
      c = y(:,1).^4;
      L = 'n^4';
    case 'm'
      c = y(:,1).^3;
      L= 'm^3';
    case 'h'
      c = c.*y(:,1);
      L = 'm^3h';
  end
```

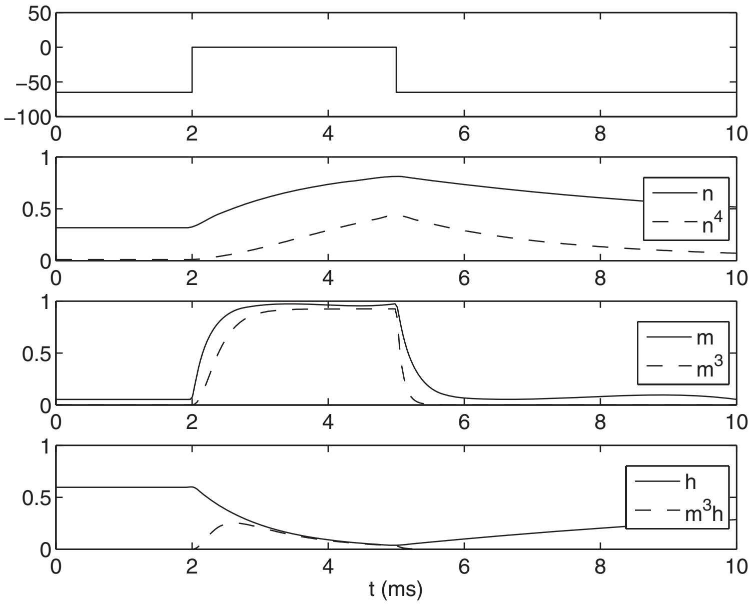


Figure 14.17 Response of the gating parameters to a step change in membrane potential. The upper pane shows the membrane voltage in mV. The lower three panes show gating parameters as indicated in the legends.

```
    plot(t, y(:,1), 'k-', t, c, 'k--')
    legend(gate(k), L, 'Location', 'East')
    if k == 3
      xlabel('t (ms)')
    end
  end

  function dy = HH(t, y, g)
  V = -65+65*((t>2)&(t<5));
  dy = alph(V+65, g)-(alph(V+65, g)+bet(V+65, g))*y(1);
```

Upon execution of the program, we obtain the results shown in Figure 14.17. The top pane shows the membrane voltage, which was held for 2 ms at −65 mV and then suddenly increased to 0 mV for 3 ms before returning to its original state. Those step changes in the membrane potential cause the gating parameters to respond as shown in the other three panes. The second pane from the top shows how n changes with time as well as the potassium gating function n^4. The third pane shows m and m^3, which are the sodium channel activation parameters. Finally, the bottom pane shows h and m^3h, which are the sodium gating functions.

14.3.4 Action Potential

The Hodgkin–Huxley model can be used to predict changes in the membrane potential that occur during an action potential in a neuron. The action potential is a communication pulse that traverses a neuron to deliver information. At one location along a neuron, the action potential can be observed as a rapid rise and then fall in the membrane potential. This change in potential is accompanied by ion currents that flow across the membrane, as well as those flowing longitudinally along the neuron, on both sides of the neuronal membrane. Thus, there is considerable complexity in the details of the action potential. Here we will simplify things by considering a special form of an action potential that is stationary in space. In the laboratory, such a stationary action potential can be created through an experimental configuration that forces the internal potential throughout a neuron to be uniform. This is accomplished by running a conductor internally along the entire length of the neuron. Then, if the potential on the outside of the cell is also uniform, all of the membrane components will see the same potential and the potential will be a function of time only.

In Section 14.3.3, the Hodgkin–Huxley model was used with the membrane potential as an input. In that mode, a net ion current flows across the membrane when a change in membrane potential is imposed. To model the action potential, we assume that the net membrane current is zero and this additional constraint allows us to compute the membrane potential. This is consistent with the fact that the action potential is a communication pulse whose primary role is to transfer information down the axon. Since a typical axon must be able to transfer many such pulses at high frequency, minimal net transport should occur during such a pulse. In addition to the equations for the gating parameters given by

Eq. (14.41), the equation requiring that the sum of the membrane currents be zero is given by

$$\begin{aligned}\frac{dV}{dt} &= \frac{-1}{C_m}\left[i_{\mathrm{K}} + i_{\mathrm{Na}} + i_L\right] \\ &= \frac{-1}{C_m}\left[g_{\mathrm{K}}\left(V - E_{\mathrm{K}}\right) + g_{\mathrm{Na}}\left(V - E_{\mathrm{Na}}\right) + g_L\left(V - E_L\right)\right]\end{aligned} \tag{14.42}$$

where the terms in Eq. (14.42) are defined in Figure 14.13. Using Eqs. (14.35) and (14.36), Eq. (14.42) becomes

$$\frac{dV}{dt} = \frac{-1}{C_m}\left[\overline{g}_{\mathrm{K}} n^4 (V - E_{\mathrm{K}}) + \overline{g}_{\mathrm{Na}} m^3 h (V - E_{\mathrm{Na}}) + g_L (V - E_L)\right] \tag{14.43}$$

Example 14.10 Hodgkin–Huxley action potential

To model the action potential, we use Eqs. (14.41) and (14.43), which are coupled through the voltage dependence of n, m, and h, as given in Table 14.7. The initial conditions are the steady-state values for this model, which are $n(0) = 0.3177$, $m(0) = 0.0529$, $h(0) = 0.5961$, and $V(0) = -62.607$ mV. To illustrate the determination of the action potential, we have assumed that the parameters are those given in Table 14.8.

The physical input that triggers an action potential is a local change in membrane potential termed a depolarization. For normal neuronal communication, this depolarization occurs at the neuron cell body due to a stimulus such as signals from other neurons, or stimulus from primary sensors such as light sensors in the eye. In the action potential model, we impose a small membrane current for a short time period as the trigger, which creates the membrane depolarization. The current trigger is a

TABLE 14.8 Parameters Used in Hodgkin–Huxley Action Potential Model[§]

Parameter	Value	Units[a]
Conductance		
Potassium $\overline{g}_{\mathrm{k}}$	0.036	S/cm^2
Sodium $\overline{g}_{\mathrm{Na}}$	0.120	S/cm^2
Leakage g_L	0.0003	S/cm^2
Nernst potential		
Potassium E_{k}	−77	mV
Sodium E_{Na}	50	mV
Leakage E_L	−49	mV
Membrane capacitance, C_m	0.001	mF/cm^2
Membrane resting potential, V_r	−62.607	mV

[§] Values represent the squid giant axon described in footnote 10.

[a] S = siemens = 1/ohm.

rectangular pulse of current density equal to 0.2 mA/cm^2, which starts at 2 ms and lasts for 0.1 ms. This depolarization trigger causes the ion channels to open slightly, which initiates an avalanche of ion flow into and out of the axon. Although the net current flow is zero, there is a net ion flow. This ion flow is reversed by a separate system of ion pumps, operating on a much longer timescale that maintains ion concentrations at normal physiological levels. The ion pumps are not addressed in this model.

The program for the action potential is as follows. In this program, we set the maximum step size in ode45 to force a small time step even when the slope of the dependent variables is small.

```
function Example14_10
Vr = -62.607;   gKbar = 0.036;   gNabar = 0.120;
gL = 0.0003;   EL = -49;   EK = -77;   ENa = 50;   Cm = 0.001;
IC = [0.3177, 0.0529, 0.5961, -62.607];
options = odeset('MaxStep', 0.1);
[t, y] = ode45(@HodHux, [0, 10], IC, options, gKbar, gNabar, gL, EL, EK, ...
                    ENa, Cm, Vr);
subplot(4,1,1)
plot(t, y(:,4), 'k-')
ylabel('V (mV)')
INa = gNabar*y(:,2).^3.*y(:,3).*(y(:,4)-ENa);
IK = gKbar*y(:,1).^4.*(y(:,4)-EK);
IL = gL*(y(:,4)-EL);
IC = -(INa+IK+IL-0.2*((t>2) & (t<2.1)));
subplot(4,1,2)
plot(t, INa, 'k-.', t, IK, 'k—', t, IL, 'k:', t, IC, 'k-')
legend('Na', 'K', 'L', 'C')
ylabel('I (mA/cm^2)')
subplot(4,1,3)
plot(t, y(:,1), 'k-', t, y(:,2), 'k—', t, y(:,3), 'k-')
legend('n', 'm', 'h')
subplot(4,1,4)
plot(t, y(:,1).^4, 'k-', t, y(:,2).^3.*y(:,3), 'k—')
xlabel('Time (ms)')
legend('n^4', 'm^3h')

function hh = HodHux(t, y, gKbar, gNabar, gL, EL, EK, ENa, Cm, Vr)
% y(1) = n,   y(2) = m,   y(3) = h,   y(4) = V
Vd = y(4)-Vr;
dn = alph(Vd, 'n')-(alph(Vd, 'n')+bet(Vd, 'n'))*y(1);
dm = alph(Vd, 'm')-(alph(Vd, 'm')+bet(Vd, 'm'))*y(2);
dh = alph(Vd, 'h')-(alph(Vd, 'h')+bet(Vd, 'h'))*y(3);
dV = -(gKbar*y(1)^4*(y(4)-EK)+gNabar*y(2)^3*y(3)*(y(4)-ENa) ...
        +gL*(y(4)-EL))/Cm;
dV = dV+200*((t>2) & (t<2.1));
hh = [dn; dm; dh; dV];
```

Execution of the program produces Figure 14.18, which plots various parameters versus time. The top pane of Figure 14.18 is a plot of membrane voltage, which is the action potential. The plot shows the initial depolarization at 2 ms, which is followed by a period of about 0.3 ms where the potential does not change very much. However, there is

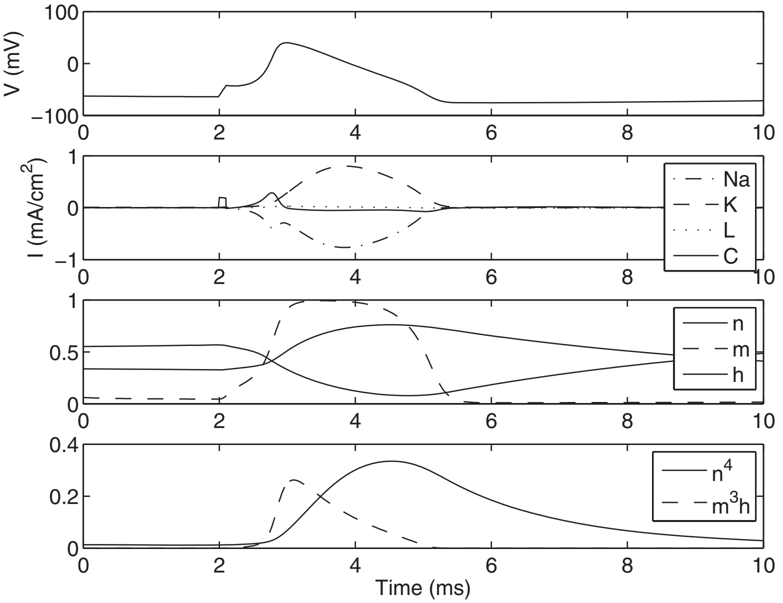


Figure 14.18 Hodgkin–Huxley action potential. Upper pane shows the membrane voltage resulting from the stimulus shown in the second pane from the top as a small step change in current starting at 2 ms; the second pane also shows the other membrane currents. The lower two panes show the time sequence of the gating parameters during the action potential.

activity behind the scenes as can be observed in the second pane from the top where individual ion currents and the capacitive current are plotted. The stimulus is a rectangular-shaped current pulse starting at 2 ms and lasting for 0.1 ms. The sodium channels respond earliest, which can be seen most clearly in the two lower panes showing the gating parameters. The sodium channel responds to the product m^3h and this function rises very steeply starting around 2.5 ms. The choreography of the various gating parameters and the resulting membrane potential changes are largely independent of the size or duration of the stimulus. This characteristic of the action potential is sometimes described as being "all or nothing." Although not fully descriptive of the physics, this characteristic does impose a largely digital character on neuron communication.

EXERCISES

Section 14.1.1

14.1 In an attempt to lower the temperature in the healthy cell region of Example 14.1, let $S = 8 \times 10^5\,\text{W/m}^3$ and apply it for 200 s in the region $0 \leq r \leq 0.008$ m. Reproduce results similar to those of Figure 14.1 for these new conditions.

14.2 Use the basic method of Example 14.1 with $S = 8 \times 10^4\,\text{W/m}^3$ to determine the steady-state energy release of the source (Q_1), the energy release of the metabolic source (Q_2), the energy removed by the perfusion term (Q_3), and the energy leaving the tumor by conduction (Q_4). These quantities are defined as follows:

$$Q_1 = 4\pi S \int_0^{R_t} r^2 dr = \frac{4}{3}\pi S R_t^3$$

$$Q_2 = 4\pi \int_0^{R_t} S_p r^2 dr = 4\pi \dot{m}_b c_b \int_0^{R_t} \big(T_{\text{ab}} - T(t_\infty, r)\big) r^2 dr$$

$$Q_3 = 4\pi R_t^2 \alpha\rho c \left.\frac{\partial T}{\partial r}\right|_{t=t_\infty,\, r=R_t}$$

$$Q_4 = 4\pi S_m \int_0^{R_t} r^2 dr = \frac{4}{3}\pi S_m R_t^3$$

where R_t is the radius of the tumor (= 0.01 m), t_∞ is the time at which steady-state conditions have been reached (= 80,000 s). Show that the energy balance is satisfied by summing these energy terms so that

$$\sum_{n=1}^{4} Q_n \approx 0$$

Section 14.1.2

14.3 Plot on the same graph the temperature as a function of time over the range $0 \le t \le 10$ s as determined by Eq. (14.1) and that as determined by the approximation given by the equation preceding Eq. (14.2). Assume that the heating element has a diameter of 1 mm and a power of 1 W/m. In addition, assume $k = 0.6$ W/m/K, $T_o = 20°\text{C}$, and $\alpha = 1.5 \times 10^{-7}\ \text{m}^2/\text{s}$.

14.4 Using the parameters in Example 14.2, create a model of the heat transfer using `pdepe` and compare the results under the same conditions as in Exercise 14.3. Let the radius of the line source be 0.00005 m. The governing equation is rewritten as

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial T}{\partial r}\right) = \frac{1}{\alpha}\frac{\partial T}{\partial t}$$

and the boundary conditions are given by

$$-k\left.\frac{\partial T}{\partial r}\right|_{r=R} = \frac{Q}{2\pi R}$$

$$T(r \to \infty, t) = T_0$$

Section 14.2.1

14.5 Using the Symbolic toolbox, obtain Eq. (14.5) from Eq. (14.3).

14.6 Compute the titration curve for bicarbonate, that is, a curve of pH versus bicarbonate concentration, assuming a fixed dissolved concentration of CO_2 as is expected in mammals with normal pulmonary function. Assume that the partial pressure of CO_2 is 45 mmHg and a starting pH = 7.4. Titrate by adding acid and by adding base from that starting point and plot pH versus bicarbonate concentration. Based on mammalian experience, consider the pH range $7 < \text{pH} < 8$.

For the acid titration, that is, when we are adding H^+, we use Eq. (14.11) and express the proton concentration as $[H^+] = [H^+]_i + y$ where $[H^+]_i$ is the initial proton concentration and y is the amount of acid added, usually HCl, which dissociates completely in water. In this context, we let $0 < y < 10^{-7}$. Thus, from Eq. (14.11)

$$[HCO_3^-] = \frac{K_{c1}[CO_2]_d}{[H^+]_i + y} \qquad 0 < y < 10^{-7}$$

For the base titration, we use Eq. (14.11) again, but combine it with the dissociation equation for water

$$K_w = [H^+][OH^-]$$

where $K_w = 10^{-14}$. For the proton concentration, we obtain

$$[H^+] = \frac{K_w}{[OH^-]} = \frac{K_w}{[OH^-]_i + y}$$

where $[OH^-]_i$ is the initial hydroxyl concentration and y is the amount of base added, usually NaOH, which dissociates completely in water. In this context, we let $0 < y < 10^{-6}$. Thus, from Eq. (14.11)

$$[HCO_3^-] = \frac{K_{c1}}{K_w}[CO_2]_d\big([OH^-]_i + y\big) \qquad 0 < y < 10^{-6}$$

For the remaining parameter values, use the values given in Example 14.4.

Section 14.2.3

14.7 Plot the saturation curve for oxygen binding to hemoglobin as given by either Eq. (14.14) (or equivalently by the equation preceding Eq. (14.28)) and the derivative of the total O_2 concentration with respect to the dissolved O_2 concentration as given by Eq. (14.29). Let $n = 2.34$, $C_{50} = 35.14 \times 10^{-6}$ M, and $[Hb] = 0.0022$ M.

14.8 Typical values for blood are $P_{CO_2} = 33 - 45$ mmHg, $[HCO_3^-] = 18 - 23$ mM, and $pH = 7.34 - 7.45$. For $P_{CO_2} = 45$ mmHg and $pH = 7.4$, use Eqs. (14.7) and (14.8) to compute the bicarbonate concentration and compare it to the normal limits (Answer: $[HCO_3^-] = 21.24$ mM, which is within the normal range).

Section 14.2.4

14.9 Compare the numerical results from Example 14.5 for plug flow to the approximate solution given by

$$C = C_{\text{in}}\left[1 + \text{Da}\left\{\frac{1 - 3(h - \hat{y})^2}{6} - \frac{\beta}{\text{Pe}}\hat{x}\right\}\right]$$

14.10 In Example 14.5, replace the water with blood as the perfusion medium and keep the same flow rate and consider plug flow only. The storage of bound oxygen influences the advection term such that Eq. (14.18) becomes

$$\text{Pe}\,\frac{\partial C_{T,O_2}}{\partial C_{O_2}}\frac{\partial C_{O_2}}{\partial \hat{x}} = \beta\frac{\partial^2 C_{O_2}}{\partial \hat{y}^2}$$

where Pe is a constant and $\partial C_{T,O_2}/\partial C_{O_2}$ is given by Eq. (14.29). For the parameters appearing in Eq. (14.29), use those given in Example 14.7. It will be found that the ability of blood to store oxygen in bound form cause the decrease in concentration along the bioreactor to be much less when using blood as compared to water.

Section 14.2.5

14.11 **a.** Starting from Example 14.6, run **Example_6** with $K = 0$ and $K = 25 \times 10^{-10}$ and compare the results. Note that with $K = 0$, the concentration is lower at the center of the tumor because in this case the metabolic sink term is constant throughout. With $K = 25 \times 10^{-10}$, the metabolic sink term is reduced when the oxygen concentration is low.

b. When $K = 0$, the solution to Eq. (14.25) subject to the boundary conditions given by Eq. (14.26) is given by

$$\psi(X) = 1 + \frac{\Sigma}{6}(\chi^2 - 1)$$

Compare the numerical solution obtained in part (a) with this result.

Section 14.2.6

14.12 **a.** Referring to Example 14.7, the mean outlet concentration is defined as

$$C_m(z) = \frac{2\pi}{V_m A}\int_0^{R_c} VC(z,r)rdr = \frac{4}{R_c^2}\int_0^{R_c}\left(1 - (r/R_c)^2\right)C(z,r)rdr$$

where A is the cross-sectional area of the capillary and we have used the fact that $V_{\max}/V_m = 2$. Using the results of Example 14.7, determine the mean outlet concentration at $z = L$ (Answer: 45.2942 μM).

b. Perform a parametric study on the effect of the mass diffusivity parameters D_c and D_t to determine the sensitivity of the mean outlet concentration to these parameters. Run the model with 10% changes in D_c and D_t using the combinations shown in Table 14.9 and determine the mean output concentration at each of these concentrations. The results are also shown in the table. It is seen that the total transport is more sensitive to the tissue diffusivity. This is an example where there are two transport resistances in series with one larger than the other such that the transport rate is "controlled" by one of the resistances. In this case, the diffusion through the tissue

TABLE 14.9 Parameter Values for Exercise 14.12

D_c (m^2/s)	D_t (m^2/s)	C_m (μM)
2.2×10^{-9}	1×10^{-9}	45.1111
1.8×10^{-9}	1×10^{-9}	45.5170
2×10^{-9}	1.1×10^{-9}	44.0989
2×10^{-9}	0.9×10^{-9}	46.6450

controls the overall transport rate. Thus, changes in that resistance have a larger overall effect on the total transport.

Section 14.3

14.13 Consider Eq. (14.41), which is used to describe an axon subjected to a membrane potential. Assume a membrane potential of −65 mV and zero initial conditions for each gating parameter. In addition, assume that the membrane potential is held for 2 ms at −65 mV and then suddenly increased to 0 mV for 3 ms before returning to −65 mV. For these conditions, determine the steady-state values for n, m, and h, which are those values that these quantities have at 100 s (Answers: $n = 0.31768$, $m = 0.05294$, and $h = 0.59611$).

14.14 Using Example 14.9 as the starting point with a resting potential of −65 mV, compute the ion current I_{ion} for step changes to a membrane potential of −50 mV, −25 mV, 0 mV, and 25 mV and plot the results. Apply the resting potential for 2 ms before applying the step changes. Let each step change remain constant for 16 ms before returning to the resting potential at 18 ms. The entire simulation is to last for 20 ms. The ion current I_{ion} is determined from Eq. (14.43) as

$$I_{\text{ion}} = \bar{g}_k n^4(V - E_{\text{K}}) + \bar{g}_{\text{Na}} m^3 h(V - E_{\text{Na}}) + g_L(V - E_L)$$

Use the values in Table 14.8 for the other parameters appearing in Eq. (14.43).

14.15 The action potential is often described as “all or nothing.” Test this description by varying the stimulus and comparing the shape and magnitude of the resulting action potentials. Start from the solution given in Example 14.10 and change the stimulation potential to 100 mV, 150 mV, 200 mV, and 250 mV. For each of these cases, plot the resulting action potential on the same graph.

14.16 One of the characteristics of the sodium channels is that they exhibit a refractory period where they will not respond until a certain time has elapsed. To test this statement, start from the program given in Example 14.10 and modify it to provide the stimuli that are separated in time as follows. Consider the following stimulus pairs: (1) a pulse that starts at 2 ms and lasts for 0.1 ms and followed by another pulse that starts at 11 ms and lasts for 0.1 ms, and (2) a pulse that starts at 2 ms and lasts for 0.1 ms followed by another pulse that starts at 12 ms and lasts for 0.1 ms. The results should show that at 11 ms the response to the second pulse is severely stunted, whereas for the case where the second pulse is initiated at 12 ms, the response to the second pulse looks very similar to that of the first pulse.

14.17 Some neurons fire repeated action potentials and encode information in the frequency of the firing. An interesting characteristic of the Hodgkin–Huxley model of Section 14.3 is that under certain conditions, the equations will fire in a self-sustained pulse train after a single stimulation. This can be shown by changing the maximum sodium conductance to one-half of its current value, that is, $\bar{g}_k = 0.018$ S/cm^2, and running a simulation for 100 ms with a single stimulus at 2 ms and duration of 0.1 ms. The result should show a continuous train of action potentials firing with a period of approximately 18 ms.

Index