MATLAB® for Engineers

Holly Moore

Sixth Edition



MATLAB® for Engineers

This page intentionally left blank

MATLAB® for Engineers

Sixth Edition

HOLLY MOORE
Salt Lake Community College
Salt Lake City, Utah



Content Management: Dawn Murrin, Erin Ault
Content Production: Devanshi Tyagi
Product Management: Holly Stark
Product Marketing: Krista Clark
Rights and Permissions: Rimpy Sharma

Please contact https://support.pearson.com/getsupport/s/ with any queries on this content.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided "as is" without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services.

The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screen shots may be viewed in full within the software version specified.

Microsoft and Windows are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

Copyright © 2023, 2018, 2015 by Pearson Education, Inc. or its affiliates, 221 River Street, Hoboken, NJ 07030. All Rights Reserved. Manufactured in the United States of America. This publication is protected by copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights and Permissions department, please visit www.pearsoned.com/permissions/.

Acknowledgments of third-party content appear on the appropriate page within the text.

PEARSON is an exclusive trademark owned by Pearson Education, Inc. or its affiliates in the U.S. and/or other countries.

Unless otherwise indicated herein, any third-party trademarks, logos, or icons that may appear in this work are the property of their respective owners, and any references to third-party trademarks, logos, icons, or other trade dress are for demonstrative or descriptive purposes only. Such references are not intended to imply any sponsorship, endorsement, authorization, or promotion of Pearson's products by the owners of such marks, or any relationship between the owner and Pearson Education, Inc., or its affiliates, authors, licensees, or distributors.

MATLAB® and Simulink® are registered trademarks of The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098.

Library of Congress Cataloging-in-Publication Data
Names: Moore, Holly, author.
Title: MATLAB® for Engineers / Holly Moore, Salt Lake Community College.
Description: Sixth edition. | Hoboken : Pearson, [2023] | Includes index.
Identifiers: LCCN 2021051256 | ISBN 9780137627981
Subjects: LCSH: Engineering mathematics—Data processing. | MATLAB.
Classification: LCC TA345 .M585 2022 | DDC 620.001/51—dc23/eng/20211117
LC record available at https://lccn.loc.gov/2021051256

ScoutAutomatedPrintCode



Rental:
ISBN-10: 0-13-762798-X
ISBN-13: 978-0-13-762798-1

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity, depth, and breadth of all learners' lived experiences.

We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, sex, sexual orientation, socioeconomic status, ability, age, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational content accurately reflects the histories and lived experiences of the learners we serve.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content prompts deeper discussions with students and motivates them to expand their own learning (and worldview).

Accessibility

We are also committed to providing products that are fully accessible to all learners. As per Pearson's guidelines for accessible educational Web media, we test and retest the capabilities of our products against the highest standards for every release, following the WCAG guidelines in developing new products for copyright year 2022 and beyond.

You can learn more about Pearson's commitment to accessibility at **https://www.pearson.com/us/accessibility.html**

Contact Us

While we work hard to present unbiased, fully accessible content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

Please contact us with concerns about any potential bias at **https://www.pearson.com/report-bias.html**

For accessibility-related issues, such as using assistive technology with Pearson products, alternative text requests, or accessibility documentation, email the Pearson Disability Support team at **disability.support@pearson.com**

Pearson

This page intentionally left blank

Contents

ABOUT THIS BOOK xiii
DEDICATION AND ACKNOWLEDGMENTS xvii

1 • ABOUT MATLAB 1

1.1 What Is MATLAB? 1
1.2 Student Edition of MATLAB 2
1.3 How is MATLAB Used in Industry? 3
1.4 Problem Solving in Engineering and Science 5

2 • MATLAB ENVIRONMENT 9

2.1 Getting Started 9
2.2 MATLAB Windows 11
2.3 Solving Problems with MATLAB 17
2.4 Saving Your Work 41
Summary 53
Key Terms 56
Problems 56

3 • BUILT-IN MATLAB FUNCTIONS 64

Introduction 64
3.1 Using Built-In Functions 64
3.2 Using the Help Feature 66
3.3 Elementary Math Functions 67
3.4 Trigonometric Functions 75
3.5 Data Analysis Functions 80
3.6 Random Numbers 100
3.7 Complex Numbers 104
3.8 Computational Limitations 108
3.9 Special Values and Miscellaneous Functions 109
Summary 111
Key Terms 113
Problems 114

4 • MANIPULATING MATLAB ARRAYS 121

Introduction 121
4.1 Manipulating Numeric Arrays 121
4.2 Problems with Two Variables—Using Meshgrid 128
4.3 Special Arrays 135
4.4 Introduction to Character and String Arrays 141
Summary 146
Key Terms 147
Problems 148

5 • PLOTTING 156

Introduction 156
5.1 Two-Dimensional Plots 156
5.2 Subplots—Tiled Chart Layouts 173
5.3 Other Types of Two-Dimensional Plots 176
5.4 Three-Dimensional Plotting 193
5.5 Editing Plots from the Menu Bar 200
5.6 Creating Plots from the Workspace Window 202
5.7 Saving Your Plots 203
5.8 Other Plotting Options 203
Summary 204
Key Terms 206
Problems 207

6 • USER-DEFINED FUNCTIONS 218

Introduction 218
6.1 Creating Function Files 218
6.2 Subfunctions 238
6.3 Creating Your Own Toolbox of Functions 243
6.4 Anonymous Functions and Function Handles 245
6.5 Function Functions 246
Summary 247
Key Terms 249
Problems 249

7 • USER-CONTROLLED INPUT AND OUTPUT 257

Introduction 257
7.1 User-Defined Input 257
7.2 Output Options 262
7.3 Graphical Input 276
7.4 Reading and Writing Data from Files 277
7.5 Debugging Your Code 280
Summary 285
Key Terms 286
Problems 287

8 • LOGICAL FUNCTIONS AND SELECTION STRUCTURES 291

Introduction 291
8.1 Relational and Logical Operators 292
8.2 Flowcharts and Pseudocode 294
8.3 Logical Functions 296
8.4 Logical Indexing 303
8.5 Selection Structures 307
8.6 Debugging 324
Summary 325
Key Terms 326
Problems 326

9 • REPETITION STRUCTURES 341

Introduction 341
9.1 `for` Loops 342
9.2 `while` Loops 350
9.3 `break` and `continue` 358
9.4 Midpoint Break Loops 359
9.5 Nested Loops 363
9.6 Improving the Efficiency of Loops 364
Summary 367
Key Terms 368
Problems 369

10 • MATRIX ALGEBRA 374

Introduction 374
10.1 Matrix Operations and Functions 374
10.2 Solutions of Systems of Linear Equations 396
10.3 Special Matrices 409
Summary 410
Key Terms 412
Problems 412

11 • OTHER KINDS OF ARRAYS 420

Introduction 420
11.1 Numeric Data Types 421
11.2 Character and String Data 427
11.3 Symbolic Data 432
11.4 Logical Data 432
11.5 Sparse Arrays 434
11.6 Categorical Arrays 434
11.7 Time Arrays 435
11.8 Multidimensional Arrays 439
11.9 Cell Arrays 441
11.10 Structure Arrays 443

11.11 Table Arrays 449
11.12 Timetable Arrays 458
Summary 462
Key Terms 464
Problems 464

12 • SYMBOLIC MATHEMATICS 472

Introduction 472
12.1 Symbolic Algebra 472
12.2 Solving Expressions and Equations 480
12.3 Symbolic Plotting 490
12.4 Units of Measurement 498
12.5 Calculus 503
12.6 Differential Equations 516
12.7 Converting Symbolic Expressions to Anonymous Functions 519
Summary 520
Key Terms 522
Problems 523

13 • NUMERICAL TECHNIQUES 532

13.1 Interpolation 532
13.2 Curve Fitting 542
13.3 Using the Interactive Fitting Tools 554
13.4 Differences and Numerical Differentiation 557
13.5 Numerical Integration 565
13.6 Solving Differential Equations Numerically 570
Summary 578
Key Terms 581
Problems 581

14 • ADVANCED GRAPHICS 589

Introduction 589
14.1 Images 589
14.2 Graphics Objects 604
14.3 Animation 611
14.4 Other Visualization Techniques 618
14.5 Introduction to Volume Visualization 620
Summary 625
Key Terms 627
Problems 627

15 • SIMULINK®— A BRIEF INTRODUCTION 630

Introduction 630
15.1 Applications 630
15.2 Getting Started 631
15.3 Solving Differential Equations with Simulink 639
Summary 646
Key Terms 646
Problems 646

APPENDIX A • SPECIAL CHARACTERS, COMMANDS, AND FUNCTIONS 651

APPENDIX B • SCALING TECHNIQUES 666

APPENDIX C • ANNUAL CLIMATOLOGICAL SUMMARY 669

APPENDIX D • SOLUTIONS TO PRACTICE EXERCISES
(Available at **www.pearsonhighered.com/moore**)

INDEX 671

This page intentionally left blank

About This Book

This book grew out of my experience teaching MATLAB and other computing languages to freshmen engineering students at Salt Lake Community College. I was frustrated by the lack of a text that "started at the beginning." Although there were many comprehensive reference books, they assumed a level of both mathematical and computer sophistication that my students did not possess. Also, because MATLAB was originally adopted by practitioners in the fields of signal processing and electrical engineering, most of these texts provided examples primarily from those areas, an approach that didn't fit with a general engineering curriculum. This text starts with basic algebra and shows how MATLAB can be used to solve engineering problems from a wide range of disciplines. The examples are drawn from concepts introduced in early chemistry and physics classes and freshman and sophomore engineering classes. A standard problem-solving methodology is used consistently.

The text assumes that the student has a basic understanding of college algebra and has been introduced to trigonometric concepts; students who are mathematically more advanced generally progress through the material more rapidly. Although the text is not intended to teach subjects such as statistics or matrix algebra, when the MATLAB techniques related to these subjects are introduced, a brief background is included. In addition, sections describing MATLAB techniques for solving problems by means of calculus and differential equations are introduced near the end of appropriate chapters. These sections can be assigned for additional study to students with a more advanced mathematics background, or they may be useful as reference material as students progress through an engineering curriculum.

The book is intended to be a "hands-on" manual. My students have been most successful when they read the book while sitting beside a computer and typing in the examples as they go. Numerous examples are embedded in the text, with more complicated numbered examples included in each chapter to reinforce the concepts introduced. Practice exercises are included in each chapter to give students an immediate opportunity to use their new skills.

The material is grouped into three sections. The first, ***An Introduction to Basic MATLAB Skills***, gets the student started and contains the following chapters:

- Chapter 1 shows how MATLAB is used in engineering and introduces a standard problem-solving methodology.
- Chapter 2 introduces the MATLAB environment and the skills required to perform basic computations. It also introduces MATLAB program files and the concept of organizing code into sections. Doing so early in the text makes it easier for students to save their work and develop a consistent programming strategy.
- Chapter 3 details the wide variety of problems that can be solved with built-in MATLAB functions. Background material on many of the functions is provided to help the student understand how they might be used. For example, the difference between Gaussian random numbers and uniform random numbers is described, and examples of each are presented.

- Chapter 4 demonstrates the power of formulating problems by using arrays in MATLAB and expanding on the techniques employed to define those arrays. The `meshgrid` function is introduced in this chapter and is used to solve problems with two variables. The difficult concept of meshing variables is revisited in Chapter 5 when surface plots are introduced. Chapter 4 also introduces how arrays are used for storing and manipulating text, and how the same techniques employed to manipulate numeric arrays apply to text arrays as well.
- Chapter 5 describes the wide variety of both two-dimensional and three-dimensional plotting techniques available in MATLAB. Creating plots via MATLAB commands, either from the command window or from within a MATLAB program, is emphasized. The extremely valuable techniques of interactively editing plots and creating plots directly from the workspace window are also introduced.

MATLAB is a powerful programming language that includes the basic constructs common to most programming languages. Because it is a scripting language, creating programs and debugging them in MATLAB is often easier than in traditional programming languages such as C++. This makes MATLAB a valuable tool for introductory programming classes. The second section of the text, ***Programming in MATLAB***, introduces students to programming and consists of the following chapters:

- Chapter 6 describes how to create and use user-defined functions. It also teaches students how to create a "toolbox" of functions to use in their own programming projects.
- Chapter 7 introduces functions that interact with the program user, including user-defined input, formatted output, and graphical input techniques. The use of MATLAB's debugging tools is also introduced.
- Chapter 8 describes logical functions such as `find`, the use of logical indexing, and demonstrates how these approaches vary from the `if` and `if/else` structures. The `switch/case` structure is also introduced. The use of logical functions and logical indexing over control structures is emphasized, partly because students (and teachers) who have previous programming experience often overlook the advantages of using MATLAB's built-in matrix functionality.
- Chapter 9 introduces repetition structures, including `for` loops, `while` loops, and midpoint break loops that utilize the `break` command. Numerous examples are included because students find these concepts particularly challenging.

Chapters 1 through 9 should be taught sequentially, but the chapters in Section 3, ***Advanced MATLAB Concepts***, do not depend upon each other. Any or all of these chapters could be used in an introductory course or could serve as reference material for self-study. Most of the material is appropriate for freshmen. A two-credit course might include Chapters 1 through 9 plus Chapter 10, while a three-credit course might include Chapters 1 through 14, but eliminate Sections 12.5, 12.6, 13.4, 13.5, and 13.6, which describe differentiation techniques, integration techniques, and solution techniques for differential equations. The skills developed in these chapters will be especially useful as students become more involved in solving engineering problems:

- Chapter 10 discusses problem solving with matrix algebra, including dot products, cross products, and the solution of linear systems of equations. Although matrix algebra is widely used in all engineering fields, it finds early application in the statics and dynamics classes taken by most engineering majors.

- Chapter 11 is an introduction to the wide variety of data types available in MATLAB.
- Chapter 12 introduces MATLAB's symbolic mathematics package. My students tell me that the package is one of the most valuable sets of techniques introduced in the course. It is something they start using immediately.
- Chapter 13 presents numerical techniques used in a wide variety of applications, especially curve fitting and statistics. Students value these techniques when they take laboratory classes such as chemistry or physics or when they take the labs associated with engineering classes such as heat transfer, fluid dynamics, or strengths of materials.
- Chapter 14 examines graphical techniques used to visualize data. These techniques are especially useful for analyzing the results of numerical analysis calculations, including results from structural analysis, fluid dynamics, and heat transfer codes.
- Chapter 15 introduces Simulink, which is a simulation package built on top of the MATLAB platform. Simulink uses a graphical user interface that allows programmers to build models of dynamic systems. It has found significant acceptance in the field of electrical engineering but has wide application across the engineering spectrum.

Appendix A lists all of the functions and special symbols (or characters) introduced in the text. Appendix B describes strategies for scaling data, so that the resulting plots are linear. Appendix C includes the Asheville, North Carolina, weather data used in a number of the example problems.

All instructor resources are available for download at www.pearson.com. If you are in need of a login and password for this site, please contact your local Pearson representative. The instructor resources include the following material:

- Solutions to practice exercises (These files are also available on the student version of the website.)
- Solutions to example problems
- Solutions to homework problems
- PowerPoint slides for each chapter
- All of the figures used in the text, suitable for inclusion in your own PowerPoint presentations
- A series of lectures (including narration) suitable for use with online classes or as reviews

Appendix D Solutions to Practice Exercises can be found at the following website:
www.pearsonhighered.com/moore

WHAT'S NEW IN THIS EDITION

New versions of MATLAB are rolled out every six months, which makes keeping any text up-to-date a challenge. The changes in this edition reflect software updates up through R2021b, which include the following:

- Screen shots shown in the book are updated to reflect the 2021b release.
- The use of subfunctions (also called local functions) in MATLAB programs is updated.
- The use of Live Scripts is emphasized. Both Scripts (M-files) and Live Scripts (MLX-files) are used to illustrate how to perform tasks. All of the examples,

practice exercises, and homework problems are available as both scripts and live scripts on the Pearson website at www.pearsonhighered.com/moore.

- The use of the string data type instead of character arrays is emphasized. The benefits of this change in programming strategy is discussed. Cell arrays of character arrays are still taught, so that students will recognize and understand the program structures when they see it in 'legacy' code.
- Tables are emphasized over cell and structure arrays as the primary storage medium for data sets containing multiple data types.
- A number of new functions are introduced throughout the book.
- Additional problems are added and some problems are modified, based on the feedback from both instructors and students who have used the book. Historic data used in the problems has been updated to current values—for example, ACE hurricane information now includes data through 2021.

Dedication and Acknowledgments

This project would not have been possible without the support of both my family and colleagues. Thanks to Mike, Heidi, Meagan, and David, and to my husband, Dr. Steven Purcell. I also benefited greatly from the suggestions for problems related to electricity from Lee Brinton and Gene Riggs of the SLCC Electrical Engineering Department. Their cheerful efforts to educate me on the mysteries of electricity are much appreciated. I'd also like to thank Quentin McRae, also at SLCC, who made numerous suggestions that improved the homework problems. And finally, Art Fox has been my tireless colleague and collaborator for almost 20 years and is responsible in large part for the success of our MATLAB computing courses at SLCC—especially the online versions.

This book is dedicated to my father, Professor George E. Moore, who taught in the Department of Electrical Engineering at the South Dakota School of Mines and Technology for almost 20 years. Professor Moore earned his college degree at the age of 54 after a successful career as a pilot in the United States Air Force and was a living reminder that you are never too old to learn. My mother, Jean Moore, encouraged both him and her two daughters to explore outside the box. Her loving support made it possible for both my sister and I to enjoy careers in engineering—something few women attempted in the early 1970s. I hope that readers of this text will take a minute to thank those people in their lives who've helped them make their dreams come true. Thanks Mom and Dad!

This page intentionally left blank

CHAPTER

1 About MATLAB

Objectives

After reading this chapter, you should be able to:

- Describe what MATLAB is and why it is widely used in engineering and science.
- Describe the advantages and limitations of the student edition of MATLAB.
- Identify example applications where MATLAB is used in industry to solve problems.
- Formulate problems using a structured problem-solving approach.

1.1 WHAT IS MATLAB?

KEY IDEA
MATLAB is a multiparadigm programming language.

MATLAB is a **"multiparadigm"** programming language, which means it can be used in several different ways to perform programming tasks. Those tasks can range from something as simple as balancing your checkbook to complex engineering modeling applications. The way you approach the calculations required to complete these tasks may be different, depending on your experience and personal preferences, the number of calculations required, the amount of data involved, mathematical limitations, the audience for the results, and many other considerations.

MATLAB is a versatile language, but it can't do everything. It is a large **application program** designed to optimize mathematical calculations; number-crunching is where it shines. You wouldn't want to use it to write a **word-processing program**, for example. If you wanted to write an **operating system** or design software, C++, Java, or some other higher-level language would be a better choice. In fact, MATLAB was originally written in Fortran and then later rewritten in C. So why use MATLAB instead of the underlying **high-level language** for any programming task? It is easier to formulate mathematical problems in MATLAB, the use of array calculations makes repetitive calculations run faster in MATLAB than in the underlying language, and there is easy access to graphics to help you display your results.

KEY IDEA
MATLAB is optimized for array calculations.

KEY IDEA
MATLAB offers easy access to graphics.

There are several different ways to access MATLAB, which is produced by the MathWorks. It is available in professional, student, and home versions. The professional version is probably installed in your college or university computer laboratory,

but you may be able to download a copy if your institution has a **site license**. It is also available online if your institution has a license or if you have purchased either a student license or a home license. With the online version, you can use MATLAB in a **web browser** without installing any software on your personal computer. MATLAB is updated regularly; this textbook is based on MATLAB 9.10. If you are using a different version, you may notice a difference in the layout of the **graphical user interface;** however, the differences in the coding approaches should be minor.

KEY IDEA
MATLAB toolboxes offer additional capability and are purchased separately.

The standard installation of MATLAB can solve a wide variety of technical problems and is sufficient for many applications. However, additional capability is available in the form of **function** toolboxes. These **toolboxes** are purchased separately. If you have access to the complete suite of toolboxes, you may be tempted to download everything—however, the complete R2021b version of the software required almost 28 gigabytes of memory. Future versions are sure to be larger. It would be wise to carefully consider your computer's memory storage limits and your internet download speed before deciding which toolboxes to download. You can find a complete list of the MATLAB product family at the MathWorks website, www.mathworks.com.

1.2 STUDENT EDITION OF MATLAB

The professional and student editions of MATLAB are very similar. Beginning students probably won't be able to tell the difference. Student editions are available for Microsoft Windows, Mac, and Linux operating systems, and can be purchased from college bookstores or online from MathWorks at www.mathworks.com.

KEY IDEA
MATLAB is regularly updated.

MathWorks packages its software in groups called *releases,* and MATLAB 9.10 is featured, along with other products, such as Simulink, in Release R2021b. New versions are released every six months. Students may purchase just MATLAB, or a bundle that includes the following products:

- A single-user license, limited to students for use in their classwork (The professional version is licensed either singly or to a group.)
- Control Systems Toolbox
- Curve Fitting Toolbox
- DSP System Toolbox
- Full MATLAB
- Image Processing Toolbox
- Instrument Control Toolbox
- Optimization Toolbox
- Parallel Computing Toolbox
- Signal Processing Toolbox
- Simulink
- Statistics and Machine Learning Toolbox
- Symbolic Math Toolbox

Toolboxes other than those included with the student edition may be purchased separately. If you are using a professional installation of MATLAB, all of the toolboxes available in the student edition may not be available to you.

The biggest difference you should notice between the professional and student editions is the command prompt, which is

```
>>
```

in the professional version, and

```
EDU>>
```

in the student edition.

1.3 HOW IS MATLAB USED IN INDUSTRY?

The ability to use tools such as MATLAB is quickly becoming a requirement for many engineering positions. A recent job search on Monster.com found the following advertisement:

> . . . is looking for a System Test Engineer with **Avionics** experience. Responsibilities include modification of MATLAB **scripts,** execution of Simulink simulations, and analysis of the results data. Candidate MUST be very familiar with MATLAB, Simulink, and C++…

This advertisement isn't unusual. The same search turned up 771 different companies that specifically required MATLAB skills for entry-level engineers. Widely used in engineering and science fields, MATLAB is particularly popular for electrical engineering applications. The sections that follow outline a few of the many applications currently using MATLAB.

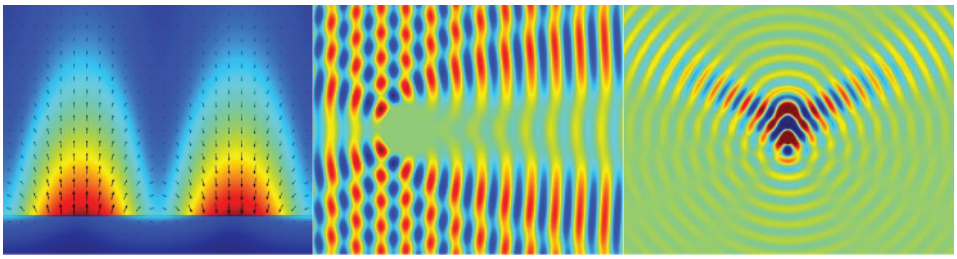
KEY IDEA
MATLAB is widely used in engineering.

1.3.1 Electrical Engineering

MATLAB is used in electrical engineering for a wide variety of applications. For example, Figure 1.1 includes several images created to help visualize the arrangements of **electromagnetic fields** in space and time. These images represent real physical situations with practical applications. Cellular communications, medical diagnostics, and home computers are just a few of the technologies that exist thanks to our understanding of this beautiful phenomenon.

1.3.2 Biomedical Engineering

Medical images are usually saved as dicom files (the Digital Imaging and Communications in Medicine standard). Dicom files use the file extension .dcm. The MathWorks offers an Image Processing Toolbox that can read these files, making their data available to MATLAB. (The Image Processing Toolbox is included with the student edition, and is optional with the professional edition.) The Image Processing Toolbox also includes a wide range of functions, many of them especially appropriate



(a) (b) (c)

Figure 1.1
Arrangements of electromagnetic fields. (a) **surface plasmon** polariton; (b) light scattering by a circular metal cylinder; (c) beam forming by a six-element dipole array. (Used with permission of Dr. James R. Nagel, University of Utah Department of Electrical and Computer Engineering.)

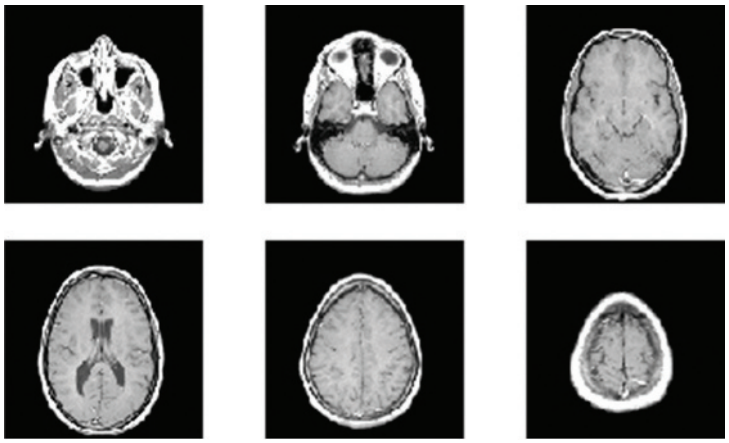


Figure 1.2
Horizontal slices through the brain, based on the sample data file included with MATLAB.

for medical imaging. A limited **MRI** data set that has already been converted to a format compatible with MATLAB ships with the standard MATLAB program. This data set allows you to try out some of the imaging functions available both with the standard MATLAB installation and with the expanded imaging toolbox, if you have it installed on your computer. Figure 1.2 shows six images of horizontal slices through the brain based on the MRI data set.

The same data set can be used to construct a three-dimensional image, such as either of those shown in Figure 1.3. Detailed instructions on how to create these images are included in the MATLAB tutorial, accessed from the help button on the MATLAB toolbar.

1.3.3 Fluid Dynamics

Calculations describing fluid velocities (speeds and directions) are important in a number of different fields. Aerospace engineers in particular are interested in the behavior of gases, both outside an aircraft or space vehicle and inside the combustion chambers. Visualizing the three-dimensional behavior of fluids is tricky, but MATLAB offers a number of tools that make it easier. Figure 1.4 represents the flow-field calculation results for a **thrust-vector control** device as a quiver plot. Thrust-vector

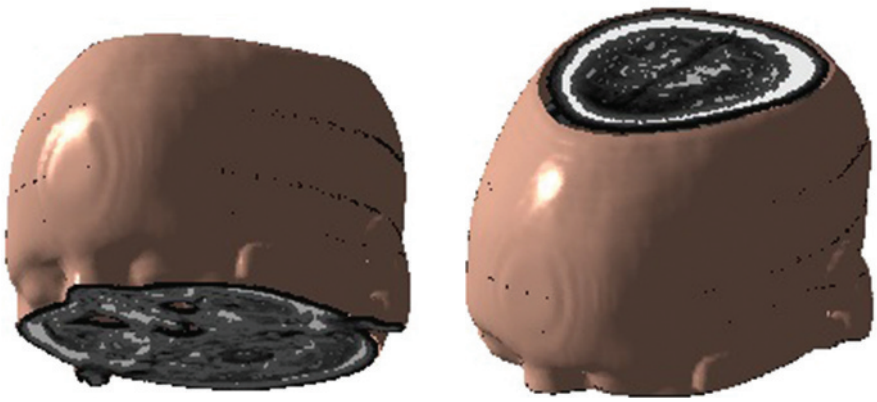


Figure 1.3
Three-dimensional visualization of MRI data, based on the sample data set included with MATLAB.

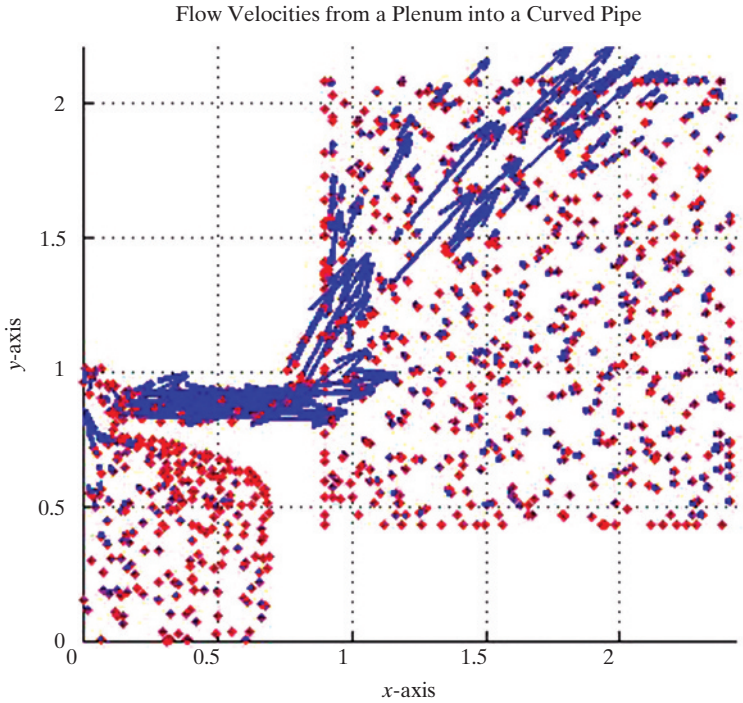


Figure 1.4
Quiver plot of gas behavior in a thrust-vector control device.

control is the process of changing the direction in which a nozzle points (and hence the direction a rocket travels) by pushing on an actuator (a piston-cylinder device). The model in the figure represents a high-pressure reservoir of gas (a plenum) that eventually feeds into the piston and thus controls the length of the actuator.

1.4 PROBLEM SOLVING IN ENGINEERING AND SCIENCE

KEY IDEA
Always use a systematic problem-solving strategy.

A consistent approach to solving technical problems is important throughout engineering, science, and computer programming disciplines. The approach we outline here is useful in courses as diverse as chemistry, physics, thermodynamics, and engineering design. It also applies to the social sciences, such as economics and sociology. Different authors may formulate their problem-solving schemes differently, but they all have the same basic format:

- *State the problem.*
  - Drawing a picture is often helpful in this step.
  - If you do not have a clear understanding of the problem, you are not likely to be able to solve it.
- *Describe the input* values (knowns) and the required *outputs* (unknowns).
  - Be careful to include units as you describe the input and output values. Sloppy handling of units often leads to wrong answers.
  - Identify constants you may need in the calculation, such as the ideal gas constant and the acceleration due to gravity.
  - If appropriate, label a sketch with the values you have identified, or group them into a table.

- *Develop an* **algorithm** to solve the problem. In computer applications, this can often be accomplished with a *hand example*. You'll need to:
  - Identify any equations relating the knowns and unknowns.
  - Work through a simplified version of the problem by hand or with a calculator.
- *Solve* the problem. In this book, this step involves creating a *MATLAB solution*.
- *Test the solution*.
  - Do your results make sense physically?
  - Do they match your sample calculations?
  - Is your answer really what was asked for?
  - Graphs are often useful ways to check your calculations for reasonableness.

If you consistently use a structured problem-solving approach, such as the one just outlined, you'll find that "story" problems become much easier to solve. Example 1.1 illustrates this problem-solving strategy.

EXAMPLE 1.1

THE CONVERSION OF MATTER TO ENERGY

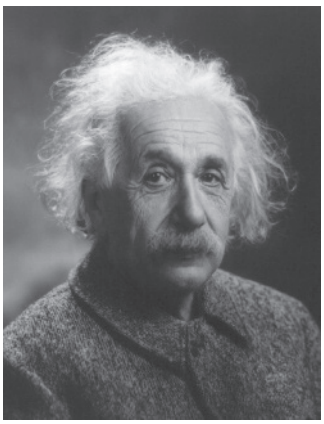


Figure 1.5
Albert Einstein.
(GL Archive/Alamy Stock Photo)

Albert Einstein (Figure 1.5) is arguably the most famous physicist of the 20th century. He was born in Germany in 1879 and attended school in both Germany and Switzerland. While working as a patent clerk in Bern, he developed his famous theory of relativity. Perhaps the best-known physics equation today is his

$$E = mc^2.$$

This astonishingly simple equation links the previously separate worlds of matter and energy, and can be used to find the amount of energy released as matter is changed in form in both natural and human-made nuclear reactions.

The sun radiates 385×10^{24} J/s of energy, all of which is generated by nuclear reactions converting matter to energy. Use MATLAB and Einstein's equation to determine how much matter must be converted to energy to produce this much radiation in one day.

1. State the problem
Find the amount of matter necessary to produce the amount of energy radiated by the sun every day.

2. Describe the input and output

Input

Energy: $E = 385 \times 10^{24}$ J/s, which must be converted into the total energy radiated during one day

Speed of light: $c = 3.0 \times 10^8$ m/s

Output

Mass m in kg

3. Develop a hand example
The energy radiated in one day is

$$385 \times 10^{24}\ \text{J/s} \times 3600\ \text{s/h} \times 24\ \text{h/day} \times 1\ \text{day} = 3.33 \times 10^{31}\ \text{J}.$$

The equation $E = mc^2$ must be solved for m and the values for E and c substituted. Thus,

$$m = \frac{E}{c^2}$$

$$m = \frac{3.33 \times 10^{31}\,\text{J}}{(3.0 \times 10^{8}\,\text{m/s})^2}$$

$$= 3.7 \times 10^{14}\,\text{J}/(\text{m}^2/\text{s}^2).$$

We can see from the output criteria that we want the mass in kg, so what went wrong? We need to do one more unit conversion:

$$1\text{J} = 1\,\text{kg m}^2/\text{s}^2$$

$$= 3.7 \times 10^{14}\,\frac{\text{J}}{\text{m}^2/\text{s}^2} \times \frac{\text{kg m}^2/\text{s}^2}{\text{J}} = 3.7 \times 10^{14}\,\text{kg}$$

4. Develop a MATLAB solution

At this point, you have not learned how to create MATLAB code. However, you should be able to see from the following sample code that MATLAB syntax is similar to that used in most algebraic scientific calculators. MATLAB commands are entered at the prompt (>>), and the results are reported on the next line. The code is:

```
>> E=385e24      The user types in this information
E =
   3.8500e+026      This is the computer's response
>> E=E*3600*24
E =
   3.3264e+031
>> c=3e8
c =
   300000000
>> m=E/c^2
m =
   3.6960e+014
```

From this point on, we will not show the prompt when describing interactions in the command window.

5. Test the solution
The MATLAB solution matches the hand calculation, but do the numbers make sense? Anything times 10^{14} is a really large number. Consider, however, that the mass of the sun is 2×10^{30} kg. We can calculate how long it would take to consume the mass of the sun completely at a rate of 3.7×10^{14} kg/day. We have

$$\text{Time} = \frac{\text{Mass of the sun}}{\text{Rate of consumption}}$$

$$\text{Time} = \frac{2 \times 10^{30}\ \text{kg}}{3.7 \times 10^{14}\ \text{kg/day}} \times \frac{\text{year}}{365\ \text{days}} = 1.5 \times 10^{13}\ \text{years}.$$

That's 15 trillion years! We don't need to worry about the sun running out of matter to convert to energy in our lifetimes.

KEY TERMS

algorithm
application program
avionics
electromagnetic fields
function
graphical user interface
high-level computer language
MRI
multiparadigm
operating system
script
site license
surface plasmon
thrust vector control
toolbox
web browser
word-processing program

CHAPTER

2 MATLAB Environment

Objectives

After reading this chapter, you should be able to:

- Start the MATLAB program and solve simple problems in the command window.
- Identify and use the various MATLAB windows.
- Identify and use the basic data types used in MATLAB.
- Define and use simple arrays.
- Name and use variables.
- Use the correct order of operation rules to solve problems using MATLAB.
- Identify the differences between scalar, array, and matrix calculations in MATLAB.
- Express numbers in either floating-point or scientific notation.
- Adjust the format used to display numbers in the command window.
- Save the value of variables used in a MATLAB session.
- Save a series of commands in a MATLAB program, either as a script (M-file) or as a live script (MLX-file).
- Use Section mode.

2.1 GETTING STARTED

Using MATLAB for the first time is easy; mastering it can take years. In this chapter, we will introduce you to the MATLAB environment and show you how to perform basic mathematical computations. After reading this chapter, you should be able to start using MATLAB for homework assignments or on the job. Of course, you will be able to do more as you complete the rest of the chapters.

Because the procedure for installing MATLAB depends upon your operating system and your computing environment, we will assume that you have already installed MATLAB on your computer, that you are working in a computing laboratory with MATLAB already installed, or are using MATLAB online. To start MATLAB in either the Windows or Mac environment, double-click on the icon on the desktop, or use

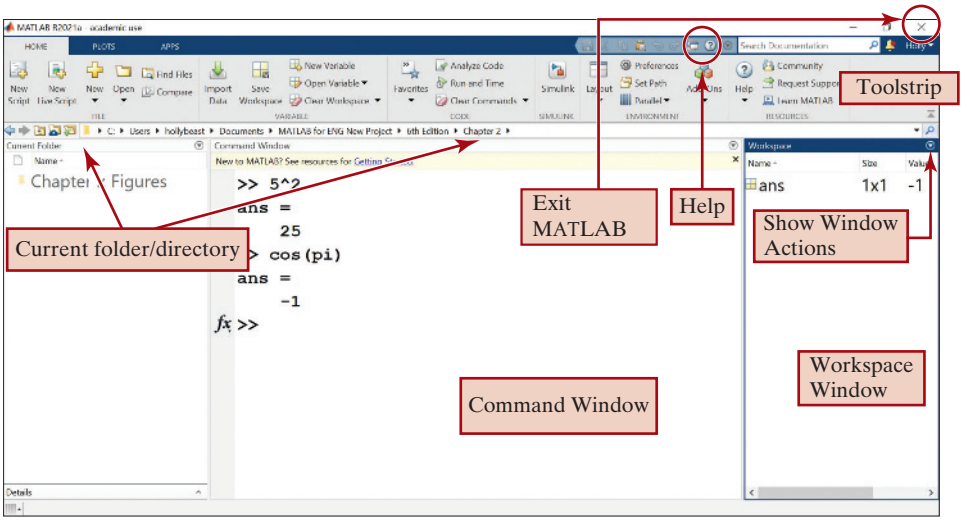


Figure 2.1
MATLAB 2021a opening window. The MATLAB environment consists of a number of windows, three of which open in the default view. Others open as needed during a MATLAB session.

the start menu to find the program. In the UNIX environment, type `Matlab` at the shell **prompt**. No matter how you start it, once MATLAB opens, you should see the MATLAB prompt (>> or `EDU`>>), which tells you MATLAB is ready for you to enter a command. When you have finished your MATLAB session, you can exit MATLAB by typing `quit` or `exit` at the MATLAB prompt. MATLAB also uses the standard Windows menu bar, so you can exit the program by selecting the close icon (**x**) at the upper right-hand corner of the screen. The default MATLAB screen, which opens each time you start the program, is shown in Figure 2.1. (There are minor differences in the MATLAB desktop, depending on the release.)

To start using MATLAB, you only need to be concerned with the command window (in the center of the screen). You can perform calculations in the command window in a manner similar to the way you perform calculations on a scientific calculator. Even most of the syntax is the same. For example, to compute the value of 5 squared, type the command

```
5^2
```

The following output will be displayed:

```
ans =
        25
```

Or, to find the value of cos(π), type

```
cos(pi)
```

which results in the output

```
ans =
       -1
```

KEY IDEA
MATLAB uses the standard algebraic rules for order of operation.

MATLAB uses the standard algebraic rules for order of operation, which becomes important when you chain calculations together. These rules will be discussed in Section 2.3.3. Notice that the value of pi is built into MATLAB, so you do not have to enter it yourself.

HINT

You may think some of the examples are too simple to type in yourself—that just reading the material is sufficient. However, you will remember the material better if you both read it and type it!

Before going any further, try Practice Exercise 2.1.

PRACTICE EXERCISE 2.1

Type the following expressions into MATLAB at the command prompt, and observe the results. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. 5 + 2
2. 5*2
3. 5/2
4. 3 + 2*(4 + 3)
5. 2.54*8/2.6
6. 6.3 − 2.1045
7. 3.6^2
8. 1+2^2
9. sqrt(5)
10. cos(pi)

HINT

You may find it frustrating to learn that when you make a mistake, you cannot just overwrite your command after you have executed it. This occurs because the command window is creating a list of all the commands you have entered. You cannot "un-execute" a command, or "un-create" it. What you can do is enter the command correctly, then execute your new version. MATLAB offers several ways to make this easier for you. One way is to use the arrow keys, usually located on the right-hand side of your keyboard. The up arrow, ↑, allows you to move through the list of commands you have executed. Once you find the appropriate command, you can edit it, then execute your new version.

2.2 MATLAB WINDOWS

MATLAB uses several display windows. The three-column view, shown in Figure 2.1, includes in the middle, a large **command window**; located on the right, the **workspace window**, and located on the left the **current folder window**. In addition, the **command history window; document windows, graphics windows,** and **editing windows** will automatically open when needed. Each will be described in the sections that follow. MATLAB also includes a built-in help tutorial that can be accessed from the **toolstrip** by selecting the question mark icon, as shown in Figure 2.1. To personalize your desktop, you can resize any of these windows, stack them on top of each other, close the ones you are not using, or "undock" them from the desktop by

using the "Show Workspace Actions" menu located in the upper right-hand corner of each window. You can restore the default configuration by selecting Layout on the toolstrip, then selecting Default, or you can add additional windows by selecting Layout and choosing the windows you would like to see.

2.2.1 Command Window

The command window is located in the center pane of the three-column view of the MATLAB screen, as shown in Figure 2.1. The command window offers an environment similar to a scratch pad. Using it allows you to save the values you calculate, but not the commands used to generate those values. If you want to save the command sequence, you will need to use the editing window to create a MATLAB program. MATLAB programs can be stored as either **scripts** (in **M-files**) or **live scripts** (in **MLX-files**). Both approaches are valuable. Before we introduce MATLAB programs, we will concentrate on using the command window.

KEY IDEA
The command window is similar to a scratch pad.

2.2.2 Command History

The command history window records the commands issued in the command window. It does not open in the default view but you can add it to your desktop by selecting Layout → Command History and checking "docked." The examples in this book will show the Command History in a docked configuration. When you exit MATLAB, or when you issue the `clc` command, the command window is cleared, but the command history retains a list of all your commands. You may clear the command history from the "Show Command History Actions" drop-down menu, located in the upper right-hand corner of the window. If you work on a public computer, as a security precaution, MATLAB's defaults may be set to clear the history when you exit MATLAB. If you entered the earlier sample commands listed in this book, notice that they are repeated in the command history window. This window is valuable for a number of reasons, including allowing you to review previous MATLAB sessions. It can also be used to transfer commands to the command window. For example, first clear the contents of the command window by typing

KEY IDEA
The command history records all of the commands issued in the command window.

```
clc
```

This action clears the command window but leaves the data in the command history window intact. You can transfer any command from the command history window to the command window by double-clicking (which also executes the command), or by clicking and dragging the line of code into the command window. Try double-clicking

```
cos(pi)
```

in the command history window. The command is copied into the command window and executed. It should return

```
ans =
      -1
```

Now click and drag

```
5^2
```

from the command history window into the command window. The command will not execute until you hit Enter, and then you will get the result:

```
ans =
      25
```

You will find the command history useful as you perform more complicated calculations in the command window.

2.2.3 Workspace Window

KEY IDEA
The workspace window lists information describing all the variables created by the program.

The workspace window keeps track of the **variables** you have defined as you execute commands in the command window. These variables represent values stored in the computer memory, which are available for you to use. If you have been doing the examples, the workspace window should show just one variable, `ans`, and indicate that it has a value of 25, and is a double **array**:

Name	Size	Class	Value
ans	1 × 1	double	25

(Your view of the workspace window may be slightly different, depending on how your installation of MATLAB is configured.)

Set the workspace window to show more about the displayed variables by right-clicking on the bar with the column labels. Check `bytes`, in addition to `name`, `value`, `class`, and `size`. Your workspace window should now display the following information, although you may need to resize the window to see all the columns:

Name	Size	Class	Bytes	Value
ans	1 × 1	double	8	25

KEY IDEA
The default data type is double-precision floating-point numbers stored in an array.

The yellow grid-like symbol indicates that the variable `ans` is an array. The size, 1 × 1, tells us that it is a single value (one row by one column) and therefore a **scalar**. The array uses 8 bytes of memory. MATLAB was written in C, and the **class** designation tells us that in the C language, `ans` is a double-precision floating-point array. For our needs, it is enough to know that the variable `ans` can store a floating-point number (a number with a decimal point). MATLAB considers every number you enter to be a floating-point number, whether you insert a decimal point or not.

In addition to information about the size of the arrays and type of data stored in them, you can also choose to display statistical information about the data. Once again, right-click the bar in the workspace window that displays the column headings. Notice that you can select from a number of different statistical measures, such as the max, min, and standard deviation.

You can define additional variables in the command window, and they will be listed in the workspace window. For example, typing

```
A = 5
```

returns

```
A =
     5
```

Notice that the variable `A` has been added to the workspace window, which lists variables in alphabetical order. Variables beginning with uppercase letters are listed first, followed by variables starting with lowercase letters.

Name	Size	Class	Bytes	Value
A	1 × 1	double	8	5
ans	1 × 1	double	8	25

In Section 2.3.2, we will discuss in detail how to enter arrays into MATLAB. For now, you can enter a simple one-dimensional array by typing

```
B = [1, 2, 3, 4]
```

This command returns

```
B =
     1 2 3 4
```

The commas are optional; you would get the same result with

```
B = [1 2 3 4]
```

```
B =
     1 2 3 4
```

Notice that the variable `B` has been added to the workspace window, and that it is a 1×4 array:

Name	Size	Class	Bytes	Value
A	1×1	double	8	5
ans	1×1	double	8	25
B	1×4	double	32	[1, 2, 3, 4]

You can define two-dimensional arrays in a similar fashion. Semicolons are used to separate rows. For example,

```
C = [1 2 3 4; 10 20 30 40; 5 10 15 20]
```

returns

```
C =
     1    2    3    4
     10   20   30   40
     5    10   15   20
```

Name	Size	Class	Bytes	Value
A	1×1	double	8	5
ans	1×1	double	8	25
B	1×4	double	32	[1, 2, 3, 4]
C	1×4	double	96	<3 × 4 double>

Notice that `C` appears in the workspace window as a 3×4 array. To conserve space, the values stored in the array are not listed.

You can recall the values for any variable by typing in the variable name. For example, entering

```
A
```

returns

```
A =
     5
```

Although the only variables we have introduced are arrays containing numbers, other types of variables are possible.

In describing the command window, we introduced the `clc` command. This command clears the command window, leaving a blank page for you to work on.

However, it does not delete from memory the actual variables you have created. The `clear` command deletes all of the saved variables, which is reflected in the workspace window. Try it out by typing

```
clear
```

in the command window. The workspace window is now empty:

Name	Value	Size	Bytes	Class

If you suppress the workspace window (closing it either from the Layout menu on the toolstrip, or from the Window Action menu in the upper right-hand corner of the window), you can still find out which variables have been defined by using the `whos` command:

```
whos
```

If executed before we entered the `clear` command, `whos` would have returned

Name	Size	Bytes	Class
A	1 × 1	8	double
B	1 × 4	32	double
C	1 × 4	96	double
ans	1 × 1	8	double

2.2.4 Current Folder Window

The current folder window lists all the files in the active directory. When MATLAB either accesses files or saves information, it uses the current folder (also called the current directory) unless told differently. The default for the location of the current folder varies with your version of the software and the way it was installed. However, the current folder is listed at the top of the main window. The current folder can be changed by selecting another directory from the drop-down list located next to the directory listing, or by browsing through your computer files. Browsing is performed with the browse button, located next to the drop-down list (see Figure 2.2).

2.2.5 Document Window

Double-clicking on any variable listed in the workspace window automatically launches a document window, containing the variable editor. Values stored in the variable are displayed in a spreadsheet format. You can change values in the variable editor, or you can add new values. For example, if you have not already entered the two-dimensional array C, enter the following command in the command window:

```
C = [1 2 3 4; 10 20 30 40; 5 10 15 20];
```

KEY IDEA
A semicolon suppresses the output from commands issued in the command window.

Placing a semicolon at the end of the command suppresses the output so that it is not repeated in the command window. However, `C` should now be listed in the workspace window. If you double-click on it, a document window will open above the command window, as shown in Figure 2.3. You can now add more values to the `C` array or change existing values. Notice that the toolstrip automatically opens the variable tab, which includes options that allow you to manipulate the data in the editor.

The document window/**array editor** can also be used in conjunction with the workspace window action menu to create entirely new arrays. Right-click on the workspace title bar, select new from the menu, and a new variable called `unnamed` should

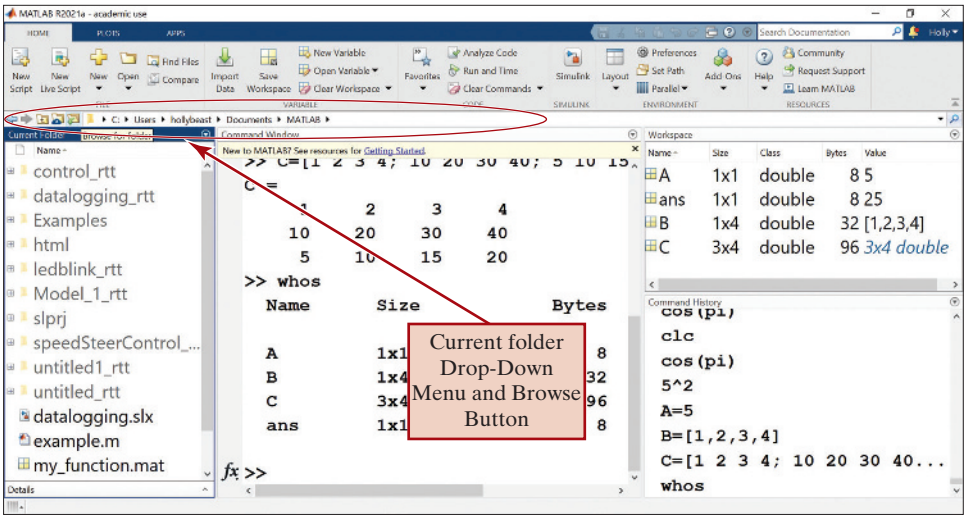


Figure 2.2
The current folder window lists all the files in the active directory. You can change the current folder by using the drop-down menu or the browse button.

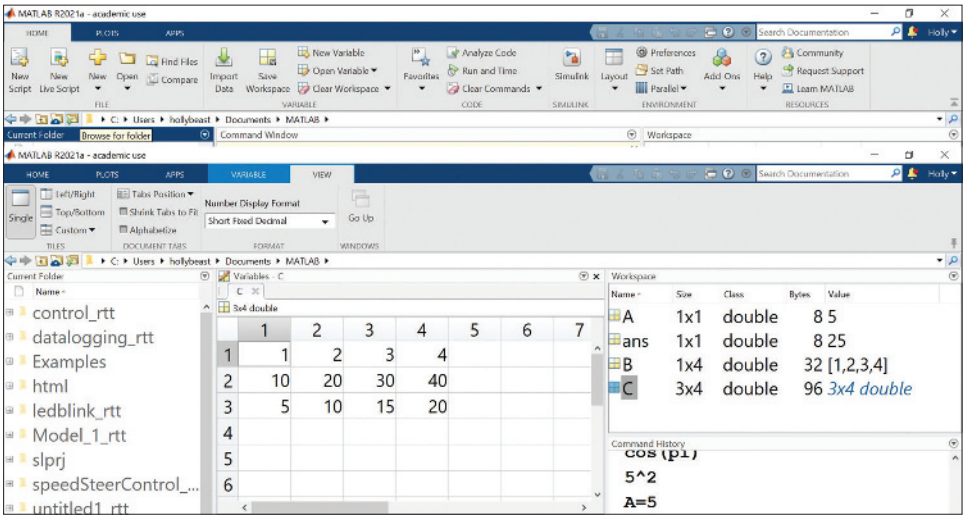


Figure 2.3
The document window displays the *variable editor*.

appear on the variable list. You can change its name by right-clicking and selecting *rename* from the pop-up menu. To add values to this new variable, double-click on it and add your data from the variable editor window. When you are finished creating new variables, close the array editor by selecting the close window icon in the upper right-hand corner of the window.

2.2.6 Graphics Window

The graphics window launches automatically when you request a graph. To demonstrate this feature, first create an array of *x* values in the command window:

```
x = [1 2 3 4 5];
```

(The semicolon suppresses the output from this command; however, a new variable, x, appears in the workspace window.)

Now create a list of *y* values:

```
y = [10 20 30 40 50];
```

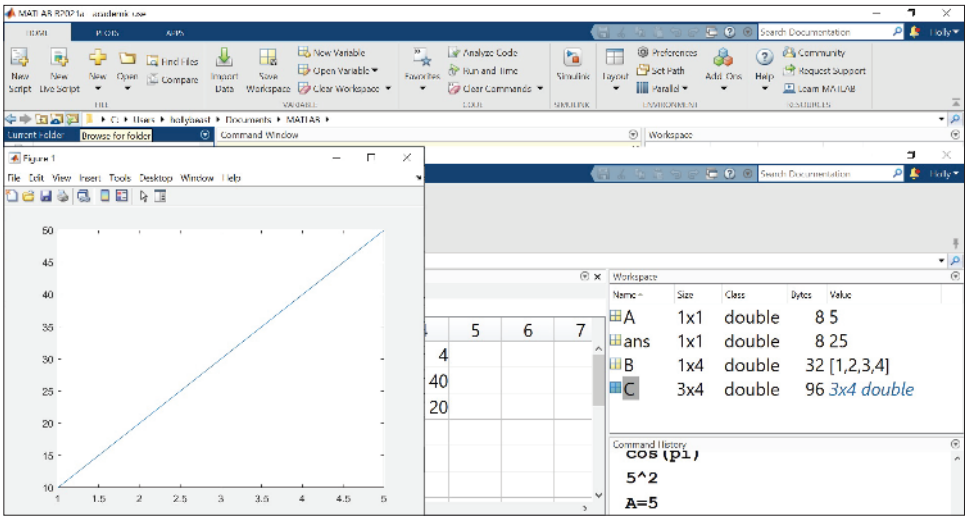


Figure 2.4
MATLAB makes it easy to create graphs. The Graphics Window opens on top of the desktop, and is not automatically docked.

To create a graph, use the plot command:

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

The graphics window opens automatically (see Figure 2.4), but is not docked with the main desktop. Notice that a new MATLAB window appears on the task bar at the bottom of the screen. Any additional graphs you create will overwrite Figure 1, unless you specifically command MATLAB to open a new graphics window. You can dock the figure by selecting the arrow in the upper right-hand corner of the window.

KEY IDEA
Always add a title and axis labels to graphs.

MATLAB makes it easy to modify graphs by adding titles, *x* and *y* labels, multiple lines, and so on. Annotating graphs will be covered in a separate chapter on plotting. Engineers and scientists **never** present a graph without labels!

2.2.7 Edit Window

The MATLAB editors allow you to type and save a series of commands without executing them. Much of your work in MATLAB will be done in an editing window. There are two types of MATLAB editors: the script editor and the live script editor. To open either editor, choose New Script or New Live Script from the toolstrip. You may also open the edit window by typing `edit` at the command prompt. Notice that the editor tab automatically opens on the toolstrip. Live Scripts are similar to regular scripts, but allow you to include formatted text in your documents. The computer code created in either editor is exactly the same.

2.3 SOLVING PROBLEMS WITH MATLAB

The command window and editing window environments are powerful tools for solving engineering problems. To use them effectively, you will need to understand more about how MATLAB works.

2.3.1 Using Variables

Although you can solve many problems by using MATLAB like a calculator, it is usually more convenient to give names to the values you are using. MATLAB uses the naming conventions that are common to most computer programs:

- All names must start with a letter. The names can be of any length, but only the first 63 characters are used in MATLAB. (Use the `namelengthmax` command

to confirm this.) Although MATLAB will let you create long variable names, excessive length creates a significant opportunity for error. A common guideline is to use lowercase letters and numbers in variable names, and to use uppercase letters for the names of constants. However, if a constant is traditionally expressed as a lowercase letter, feel free to follow that convention. For example, in physics textbooks, the speed of light is always lowercase *c*. Names should be short enough to remember and should be descriptive.

KEY IDEA
Variable names should be short and meaningful.

- The only allowable characters are letters, numbers, and the underscore. You can check to see if a variable name is allowed by using the `isvarname` command. As is standard in computer languages, the number 1 means that something is true, and the number 0 means false. Hence,

```
isvarname time
ans =
       1
```

indicates that `time` is a legitimate variable name, and

```
isvarname cool-beans
ans =
       0
```

KEY IDEA
Variable names must start with a letter.

tells us that `cool-beans` is not a legitimate variable name. (Recall that the dash is not an allowed character.)

- Names are case sensitive. The variable `x` is different from the variable `X`.
- MATLAB reserves a list of keywords for use by the program, which you cannot assign as variable names. The `iskeyword` command causes MATLAB to list these reserved names:

```
iskeyword
ans =
    20×1 cell array
    {'break'      }
    {'case'       }
    {'catch'      }
    {'classdef'   }
    {'continue'   }
    {'else'       }
    {'elseif'     }
    {'end'        }
    {'for'        }
    {'function'   }
    {'global'     }
    {'if'         }
    {'otherwise'  }
    {'parfor'     }
    {'persistent' }
    {'return'     }
    {'spmd'       }
    {'switch'     }
    {'try'        }
    {'while'      }
```

- MATLAB allows you to reassign built-in **function** names as variable names. For example, you could create a new variable called `sin` with the command

```
sin = 4
```

which returns

```
sin =
     4
```

This is clearly a dangerous practice, since the `sin` (i.e., sine) function is no longer available. If you try to use the overwritten function, you'll get an error statement:

```
sin(3)
Index exceeds the number of array elements (1).
```

and a message with a suggested solution.

`sin` appears to be both a function and a variable. If this is unintentional, use `clear sin` to remove the variable `sin` from the workspace.
You can reset `sin` back to a function by typing

```
clear sin
```

Once you have reset `sin` you can check again to see if it is a function or a variable.

Now when you ask

```
which sin
```

the response is

```
built-in (C:\ProgramFiles\MATLAB\R2011a\toolbox\matlab\elfun\
@double\sin)
% double method
```

which tells us the location of the built-in function.

PRACTICE EXERCISE 2.2

Which of the following names are allowed in MATLAB? Make your predictions, then test them with the `isvarname`, `iskeyword`, and `which` commands. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. test
2. Test
3. if
4. my-book
5. my_book
6. Thisisoneverylongnamebutisitstillallowed?
7. 1stgroup
8. group_one
9. zzaAbc
10. z34wAwy?12#
11. sin
12. log

KEY IDEA
All data in MATLAB is stored in arrays.

2.3.2 Data Types

The basic **data type** used in MATLAB is the array. A single value, called a scalar, is represented as a 1×1 array. A list of values, arranged in either a column or a row, is a one-dimensional array called a **vector**. A table of values is represented as a two-dimensional array. Although we'll limit ourselves to scalars, vectors, and two-dimensional arrays in this chapter, MATLAB can handle higher order arrays. The terms **matrix** and array are used interchangeably by most MATLAB users, even though they are technically different in a mathematical context. An array is really only a matrix if it is used in matrix algebra calculations, which are discussed in a later chapter.

KEY IDEA
MATLAB arrays can be scalars (a single value), vectors (a row or column), two-dimensional arrays or multi-dimensional arrays.

The array data type is further divided into categories, which in MATLAB are confusingly also simply called data types, or classes. Thus far we have used MATLAB with numeric information. Numbers in MATLAB default to the double-precision floating-point representation (`double`), which is only one of the possible numeric data types available. Other less commonly used numeric data types are discussed in later chapters. Another kind of data that is widely used in engineering and scientific applications is textual information. MATLAB offers two data types for storing text: character arrays (`char`) and string arrays (`string`). Logical arrays store information about whether things are true or false and are useful in making decisions in programming situations. There are many other kinds of MATLAB arrays that store just a single type of information, but these four basic types (doubles, character, strings, and logical arrays) are enough for a beginning programmer to get started.

There are also data types that can store multiple data types in the same array. The most useful for beginners is the `table`, especially because it is the default data type when data is imported from files. Tables look a lot like spreadsheets. They are used to both store information and to display results.

KEY IDEA
MATLAB arrays can hold numeric as well as other types of information.

The basic rules for manipulating information in arrays are the same, independent of the data type. In this chapter, calculations with numeric arrays will be explored. The next few chapters will introduce textual information, logical information, and tables. Later chapters will explore the wide variety of data types available for you to use, once you have mastered basic programming concepts.

2.3.3 Calculations with Numeric Arrays

In mathematical nomenclature, arrays are represented as rows and columns inside square brackets:

$$A = [5] \quad B = [2 \quad 5] \quad C = \begin{bmatrix} 1 & 2 \\ 5 & 7 \end{bmatrix}$$

In this example, A is a 1×1 array, B is a 1×2 array, and C is a 2×2 array. The advantage in using array representation is that whole groups of information can be represented with a single name. Most people feel more comfortable assigning a name to a single value, so we'll start by explaining how MATLAB handles scalars, then move on to more complicated arrays.

Scalar Operations

MATLAB handles arithmetic operations between two scalars much the same as other computer programs and handheld calculators. The syntax for addition, subtraction, multiplication, division, and exponentiation is shown in Table 2.1. The command

```
a = 1 + 2
```

Table 2.1 Arithmetic Operations Between Two Scalars (Binary Operations)

Operation	Algebraic Syntax	MATLAB Syntax
Addition	$a + b$	**a + b**
Subtraction	$a - b$	**a - b**
Multiplication	$a \times b$	**a*b**
Division	$\frac{a}{b}$ or $a \div b$	**a/b**
Exponentiation	a^b	**a^b**

should be read as "a is assigned a value of 1 plus 2," which is the addition of two scalar quantities. Arithmetic operations between two scalar variables use the same syntax. Suppose, for example, that you have defined a in the previous statement and that b has a value of 5:

```
b = 5
```

Then

```
x = a + b
```

returns the following result:

```
x =
     8
```

A single equals sign (=) is called an **assignment operator** in MATLAB. The assignment operator causes the result of your calculations to be stored in a computer memory location. In the preceding example, x is assigned a value of 8. If you enter the variable name

```
x
```

into MATLAB, you get the following result:

```
x =
     8
```

KEY IDEA
The assignment operator is different from an equality.

The assignment operator is significantly different from an equality. Consider the statement

```
x = x + 1
```

This is not a valid algebraic statement, since x is clearly not equal to $x + 1$. However, when interpreted as an assignment statement, it tells us to replace the current value of x stored in memory with a new value that is equal to the old x plus **1**.

Since the value stored in x was originally 8, the statement returns

```
x =
     9
```

indicating that the value stored in the memory location named x has been changed to 9. The assignment statement is similar to the process of saving a file. When you first save a word-processing document, you assign it a name. Subsequently, after you've made changes, you resave your file, but still assign it the same name. The first and second versions are not equal: You've just assigned a new version of your document to an existing memory location.

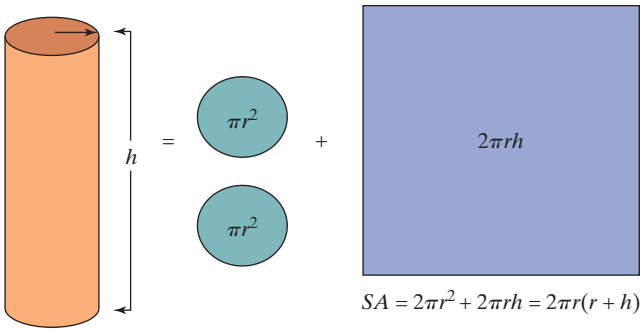


Figure 2.5
Finding the surface area of a right circular cylinder involves addition, multiplication, and exponentiation.

Order of Operations

In all mathematical calculations, it is important to understand the order in which operations are performed. MATLAB follows the standard algebraic rules for the order of operation:

- First, perform calculations inside parentheses, working from the innermost set to the outermost.
- Next, perform exponentiation operations.
- Then, perform multiplication and division operations, working from left to right.
- Finally, perform addition and subtraction operations, working from left to right.

To better understand the importance of the order of operations, consider the calculations involved in finding the surface area of a right circular cylinder.

The surface area is the sum of the areas of the two circular bases and the area of the curved surface between them, as shown in Figure 2.5. If we let the height of the cylinder be 10 cm and the radius 5 cm, the following MATLAB code can be used to find the surface area:

```
radius = 5;
height = 10;
surface_area = 2*pi*radius^2 + 2*pi*radius*height
```

The code returns

```
surface_area =
            471.2389
```

In this case, MATLAB first performs the exponentiation, raising the radius to the second power. It then works from left to right, calculating the first product and then the second product. Finally, it adds the two products together. You could instead formulate the expression as

```
surface_area = 2*pi*radius*(radius + height)
```

which also returns

```
surface_area =
        471.2389
```

In this case, MATLAB first finds the sum of the radius and height then performs the multiplications, working from left to right. If you forgot to include the parentheses, you would have

```
surface_area = 2*pi*radius*radius + height
```

in which case the program would have first calculated the product of `2*pi*radius*radius` and then added `height`—resulting in the wrong answer.

Note that it was necessary to include the multiplication operator before the parentheses, because MATLAB does not assume any operators and would misinterpret the expression

```
radius(radius + height)
```

as follows. The value of radius plus height is 15 (`radius` = 10 and `height` = 5) so MATLAB would have looked for the 15th value in an array called radius. This interpretation would have resulted in the following error statement:

```
??? Index exceeds the number of array elements (1).
```

It is important to be extra careful in converting equations into MATLAB statements. There is no penalty for adding extra parentheses, and they often make the code easier to interpret, both for the programmer and for others who may use the code in the future. Here's another common error that could be avoided by liberally using parentheses. Consider the following mathematical expression:

$$e^{\frac{-Q}{RT}}$$

In MATLAB, the mathematical constant e is evaluated using the function, `exp`, so the appropriate syntax is

```
exp(-Q/(R*T))
```

Unfortunately, leaving out the parentheses as in

```
exp(-Q/R*T)
```

gives a very different result. Since the expression is evaluated from left to right, first Q is divided by R, then the result is multiplied by T—not at all what was intended.

Another way to make computer code more readable is to break long expressions into multiple statements. For example, consider the equation

$$f = \frac{\log(ax^2 + bx + c) - \sin(ax^2 + bx + c)}{4\pi x^2 + \cos(x - 2)*(ax^2 + bx + c)}$$

It would be very easy to make an error keying in this equation. To minimize the chance of that happening, break the equation into several pieces. For example, first assign values for `x`, `a`, `b`, and `c` (you can do it all on the same line):

```
x = 9; a = 1; b = 3; c = 5;
```

Then define a polynomial and the denominator:

```
poly = a*x^2 + b*x + c;
denom = 4*pi*x^2 + cos(x - 2)*poly;
```

Combine these components into a final equation:

```
f = (log(poly) - sin(poly))/denom
```

The result is

```
f =
    0.0044
```

This approach minimizes your opportunity for error. Instead of keying in the polynomial three times (and risking an error each time), you need to key it in only once. Your MATLAB code is more likely to be accurate, and it's easier for others to understand.

KEY IDEA
Try to minimize opportunity for error.

HINT

MATLAB does not read "white space," so you may add spaces to your commands without changing their meaning. A long expression is easier to read if you add a space before and after plus (+) signs and minus (–) signs but not before and after multiplication (*) and division (/) signs.

PRACTICE EXERCISE 2.3

Predict the results of the following MATLAB expressions, then check your predictions by keying the expressions into the command window. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. 6/6 + 5
2. 2*6^2
3. (3 + 5)*2
4. 3 + 5*2
5. 4 * 3/2 * 8
6. 3 – 2/4 + 6^2
7. 2^3^4
8. 2^(3^4)
9. 3^5+4
10. 3^(5+4)

Create and test MATLAB syntax to evaluate the following expressions, and then check your answers with a handheld calculator.

11. $\dfrac{5+3}{9-1}$
12. $2^3 - \dfrac{4}{5+3}$
13. $\dfrac{5^{2+1}}{4-1}$
14. $4\dfrac{1}{2} * 5\dfrac{2}{3}$
15. $\dfrac{5 + 6*\dfrac{7}{3} - 2^2}{\dfrac{2}{3} * \dfrac{3}{3*6}}$

EXAMPLE 2.1

SCALAR OPERATIONS

Wind tunnels (see Figure 2.6) play an important role in our study of the behavior of high-performance aircraft. In order to interpret wind tunnel data, engineers need to understand how gases behave. The basic equation describing the properties of gases is the ideal gas law, a relationship studied in detail in freshman chemistry classes. The law states that



Figure 2.6
Wind tunnels are used to test aircraft designs. Kitty O'Brien Joyner was an American Electrical Engineer with the National Advisory Committee for Aeronautics (NACA) and then with NASA. She was the first woman engineer at NASA and managed several of its wind tunnels. She is shown analyzing the operation of a wind tunnel turbine at NACA Langley in 1952. (Courtesy of NASA)

$$PV = nRT$$

where P = pressure in kPa,
V = volume in m^3,
n = number of kmoles of gas in the sample,
R = ideal gas constant, 8.314 kPa m^3/kmol K, and
T = temperature, expressed in kelvins (K).

In addition, we know that the number of kmoles of gas is equal to the mass of the gas divided by the molar mass (also known as the molecular weight) or

$$n = m/MW$$

where
m = mass in kg and
MW = molar mass in kg/kmol.

Different units can be used in the equations if the value of R is changed accordingly.

Now, suppose you know that the volume of air in the wind tunnel is 1000 m^3. Before the wind tunnel is turned on, the temperature of the air is 300 K, and the pressure is 100 kPa. The average molar mass (molecular weight) of air is approximately 29 kg/kmol. Find the mass of the air in the wind tunnel.

To solve this problem, use the following problem-solving methodology:

1. State the problem
When you solve a problem, it is a good idea to restate it in your own words: Find the mass of air in a wind tunnel.

2. Describe the input and output

Input

Volume	$V = 1000\ m^3$
Temperature	$T = 300$ K
Pressure	$P = 100$ kPa
Molecular weight	$MW = 29$ kg/kmol
Gas constant	$R = 8.314$ kPa m^3/kmol K

Output

Mass	$m = ?$ kg

(*continued*)

3. Develop a hand example
Working the problem by hand (or with a calculator) allows you to outline an algorithm, which you can translate to MATLAB code later. You should choose simple data that make it easy to check your work. In this problem, we know two equations relating the data:

$$PV = nRT \quad \text{ideal gas law}$$
$$n = m/MW \quad \text{relationship between mass and moles}$$

Solve the ideal gas law for n, and plug in the given values:

$$\begin{aligned} n &= PV/RT \\ &= \frac{100 \text{ kPa} \times 1000 \text{ m}^3}{8.314 \text{ kPa m}^3/\text{kmol K} \times 300\text{K}} \\ &= 40.0930 \text{ kmol} \end{aligned}$$

Convert moles to mass by solving the conversion equation for the mass m and plugging in the values:

$$m = n \times MW = 40.0930 \text{ kmol} \times 29 \text{ kg/mol}$$
$$m = 1162.70 \text{ kg}$$

4. Develop a MATLAB solution
First, clear the screen and memory:

```
clear, clc
```

Now perform the following calculations in the command window:

```
P = 100
P =
    100
T = 300
T =
    300
V = 1000
V =
       1000
MW = 29
MW =
       29
R = 8.314
R =
     8.3140
n = (P*V)/(R*T)
n =
     40.0930
m = n*MW
m =
         1.1627e+003
```

There are several things you should notice about this MATLAB solution. First, because no semicolons were used to suppress the output, the values of the variables are repeated after each assignment statement. Notice also the use of parentheses in the calculation of n. They are necessary in the denominator, but not in the numerator. However, using parentheses in both makes the code easier to read.

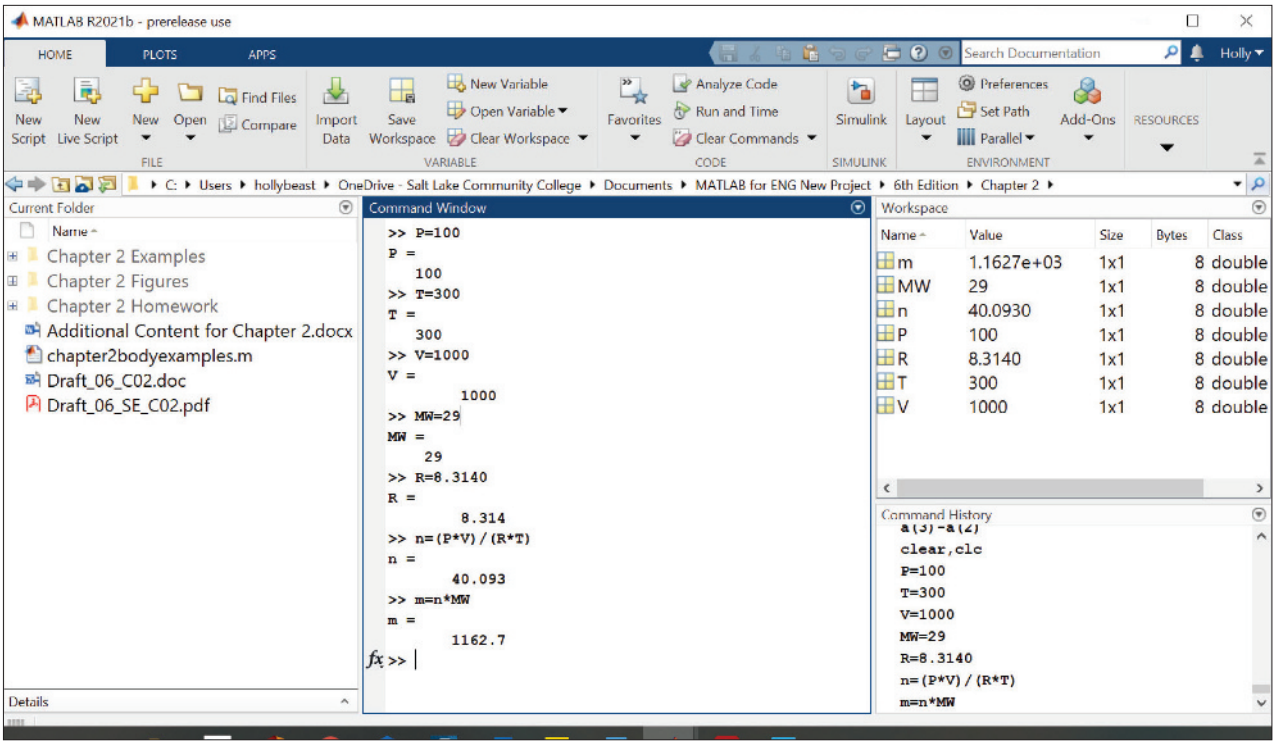


Figure 2.7
MATLAB screen used to solve the ideal gas problem.

5. Test the solution

In this case, comparing the result with that obtained by hand is sufficient. More complicated problems solved in MATLAB should use a variety of input data, to confirm that your solution works in a variety of cases. The MATLAB screen used to solve this problem is shown in Figure 2.7.

You may wonder why you should bother to do a hand calculation if you have MATLAB to do the calculation for you. Once you are proficient with the program, a quick approximation is all that is really necessary. However, new users often make simple mistakes like forgetting to use parentheses in the denominator, such as:

$$n = (P*V)/R*T$$

which results in an incorrect answer. While you are learning, you should check your work with a tool with which you are comfortable, such as your handheld calculator.

Notice that the variables defined in the command window are listed in the workspace window. Notice also that the command history lists the commands executed in the command window. If you were to scroll up in the command history window, you would see commands from previous MATLAB sessions. All of these commands are available for you to move to the command window.

Array Operations

KEY IDEA
MATLAB's strength is in array calculations.

Using MATLAB as a glorified calculator is fine, but its real strength is in array manipulations. As described previously, the simplest way to define an array is to use a list of numbers, called an **explicit list**. The command

```
x = [1 2 3 4]
```

returns the row vector

```
x =
     1 2 3 4
```

Recall that, in defining this vector, you may list the values either with or without commas. A new row is indicated by a semicolon, so a column vector is specified as

```
y = [1; 2; 3; 4]
```

and an array that contains both rows and columns is created with the statement

```
a = [1 2 3 4; 2 3 4 5; 3 4 5 6]
```

and will return

```
a =
     1 2 3 4
     2 3 4 5
     3 4 5 6
```

HINT

It's easier to keep track of how many values you've entered into an array if you enter each row on a separate line—the semicolon is optional.

```
a = [1 2 3 4;
2 3 4 5;
3 4 5 6]
```

While a complicated array might have to be entered by hand, evenly spaced arrays can be entered much more readily. The command

```
b = 1:5
```

and the command

```
b = [1:5]
```

are equivalent statements. Both return a row array

```
b =
     1 2 3 4 5
```

(The square brackets are optional.) The default increment is 1, but if you want to use a different increment, put it between the first and final values on the right side of the command. For example,

```
c = 1:2:5
```

indicates that the increment between values will be 2 and returns

```
c =
       1   3   5
```

If you want MATLAB to calculate the spacing between elements, you may use the `linspace` command. Specify the initial value, the final value, and how many total values you want. For example,

```
d = linspace(1, 10, 3)
```

returns a vector with three values, evenly spaced between 1 and 10:

```
d =
       1  5.5  10
```

If you don't specify the number of values in the third field, the function defaults to 100 values, so that

```
d = linspace(1, 10);
```

creates an array with 100 values. In this example a semicolon was used to suppress the output, but by referring to the workspace window you can see that `d` is a 1×100 vector.

You can create logarithmically spaced vectors with the `logspace` command, which also requires three inputs. The first two values are powers of 10 representing the initial and final values in the array. The final value is the number of elements in the array. Thus,

```
e = logspace(1, 3, 3)
```

returns three values:

```
e =
   10 100 1000
```

Notice that the first element in the vector is 10^1, and the last element in the array is 10^3.

HINT

New MATLAB users often err when using the `logspace` command by entering the actual first and last values requested, instead of the corresponding power of 10. For example,

```
logspace (10,100,3)
```

is interpreted by MATLAB as: Create a vector from 10^{10} to 10^{100} with three values. The result is

```
ans =
       1.0e+100 *
       0.0000 0.0000 1.0000
```

A common multiplier (1×10^{100}) is specified for each result, but the first two values are so small in comparison to the third, that they are effectively 0.

HINT

You can include mathematical operations inside an array definition statement. For example, you might have *a* = [*0:pi/10:pi*]

Arrays can be used in many calculations with scalars. If

```
a = [ 1 2 3 ]
```

we can add 5 to each value in the array with the syntax

```
b = a + 5
```

which returns

```
b =
      6   7   8
```

KEY IDEA
Matrix multiplication is different from element-by-element (array) multiplication.

This approach works well for addition and subtraction; however, multiplication and division are different. In matrix mathematics, the multiplication operator (*) indicates matrix multiplication. Because all MATLAB operations can involve arrays, we need a different operator to indicate element-by-element multiplication. That operator is `.*` (called *dot multiplication* or *array multiplication*). For example,

```
a.*b
```

results in element 1 of array `a` being multiplied by element 1 of array `b`,

element 2 of array `a` being multiplied by element 2 of array `b`,

element *n* of array `a` being multiplied by element *n* of array `b`.

For the particular case of our `a` (which is **[1 2 3]**) and our `b` (which is **[6 7 8]**),

```
a.*b
```

returns

```
ans =
       6     14     24
```

(Do the math to convince yourself that these are the correct answers.)

When you multiply a scalar times an array, you may use either operator (`*` or `.*`), but when you multiply two arrays together, they mean something quite different. Just using `*` implies a matrix multiplication, which in this case would return an error message, because `a` and `b` here do not meet the rules for multiplication in matrix algebra. The moral is, be careful to use the correct operator when you mean element-by-element multiplication.

Similar syntax holds for exponentiation (`.^`) and element-by-element division (`./`) of individual elements:

```
a.^2
a./b
```

Unfortunately, when you divide a scalar by an array you still need to use the ./ syntax, because the / operator means taking the matrix inverse to MATLAB. As a rule, unless you specifically are doing problems involving linear algebra (matrix mathematics), you should use the **dot operators**.

As an exercise, predict the values resulting from the preceding two expressions, then test your predictions by executing the commands in MATLAB.

PRACTICE EXERCISE 2.4

As you perform the following calculations, recall the difference between the * and .* operators, as well as the / and ./ and the ^ and .^ operators. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Define the array $a = [2.3 \quad 5.8 \quad 9]$ as a MATLAB variable.
2. Find the sine of a.
3. Add 3 to every element in a.
4. Define the array $b = [5.2 \quad 3.14 \quad 2]$ as a MATLAB variable.
5. Add together each element in array a and in array b.
6. Multiply each element in a by the corresponding element in b.
7. Square each element in array a.
8. Create an array named c of evenly spaced values from 0 to 10, with an increment of 1.
9. Create an array named d of evenly spaced values from 0 to 10, with an increment of 2.
10. Use the `linspace` function to create an array of six evenly spaced values from 10 to 20.
11. Use the `logspace` function to create an array of five logarithmically spaced values between 10 and 100.

KEY IDEA
The array capability of MATLAB makes it easy to do repetitive calculations.

The array capability of MATLAB makes it easy to do repetitive calculations. For example, suppose you have a list of angles in degrees that you would like to convert to radians. First put the values into an array. For angles of 10, 15, 70, and 90, enter

```
degrees = [10 15 70 90];
```

To change the values to radians, you must multiply by $\pi/180$:

```
radians = degrees*pi/180
```

This command returns an array called `radians`, with the values in radians. (Try it!) In this case, you could use either the `*` or the `.*` operator, because the multiplication involves a single array (`degrees`) and two scalars (pi and 180). Thus, you could have written

```
radians = degrees.*pi/180
```

HINT

The value of π is built into MATLAB as a floating-point number called `pi`.

Because π is an irrational number, it cannot be expressed *exactly* with a floating-point representation, so the MATLAB constant pi is really an approximation. You can see this when you find `sin(pi)`. From trigonometry, the answer should be 0. However, MATLAB returns a very small number, 1.2246e–016. In most calculations, this won't make a difference in the final result.

Another useful **array operator** is transposition. The **transpose** operator changes rows to columns and vice versa. For example,

```
degrees'
```

returns

```
ans =
      10
      15
      70
      90
```

This makes it easy to group information together in arrays with multiple columns. For example, to create an array that includes the information to convert degrees to radians, enter

```
Deg_to_R = [degrees', radians']
```

which tells MATLAB to create an array named `Deg_to_R`, in which column 1 is degrees and column 2 is radians:

```
Deg_to_R =
         10.0000 0.1745
         15.0000 0.2618
         70.0000 1.2217
         90.0000 1.5708
```

If you transpose a two-dimensional array, all the rows become columns and all the columns become rows. For example, the command

```
Deg_to_R
```

results in

```
10.0000    15.0000    70.0000    90.0000
 0.1745     0.2618     1.2217     1.5708
```

EXAMPLE 2.2

ARRAY CALCULATIONS WITH SCALARS

Scientific data, such as data collected from wind tunnels, is usually in SI (Système International) units. However, much of the manufacturing infrastructure in the United States has been tooled in English (sometimes called American Engineering or American Standard) units. Engineers need to be fluent in both systems, and should be especially careful when sharing data with other engineers. Perhaps the most notorious example of unit confusion problems is the Mars Climate Orbiter (see Figure 2.8), which was the second flight of the Mars Surveyor Program. The spacecraft burned up in the orbit of Mars in September of 1999 because of a lookup table embedded in the craft's software. The table, probably generated from wind-tunnel testing, used pounds force (lbf) when the program expected values in newtons (N).

In this example, we'll use MATLAB to create a conversion chart of pounds force to newtons. The chart, stored as an array, will start at 0 and go to 1000-lbf, at 100-lbf intervals. The conversion factor is

$$1 \text{ lbf} = 4.4482216 \text{ N}$$

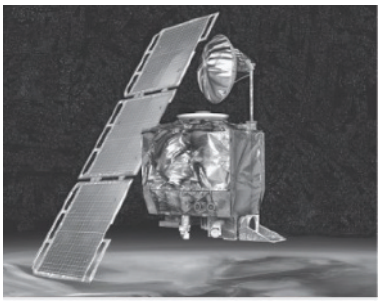


Figure 2.8
Climate Orbiter. (Courtesy of NASA/Jet Propulsion Laboratory.)

1. State the problem
Create a chart in the form of an array converting pounds force (lbf) to newtons (N).

2. Describe the input and output

Input

The starting value in the array is:	0 lbf
The final value in the array is:	1000 lbf
The increment between values is:	100 lbf
The conversion from lbf to N is:	1 lbf = 4.4482216 N

Output
Array listing pounds force (lbf) and newtons (N)

3. Develop a hand example
Since we are creating an array, it makes sense to check a number of different values. Choosing numbers for which the math is easy makes the hand example simple to complete, but still valuable as a check:

0	*	4.4482216 = 0
100	*	4.4482216 = 444.82216
1000	*	4.4482216 = 4448.2216

4. Develop a MATLAB solution

```
clear, clc
lbf = [0:100:1000];
N = lbf * 4.44822;
[lbf',N']
ans =
  1.0e+003 *
    0          0
0.1000       0.4448
0.2000       0.8896
0.3000       1.3345
0.4000       1.7793
0.5000       2.2241
0.6000       2.6689
0.7000       3.1138
0.8000       3.5586
0.9000       4.0034
1.0000       4.4482
```

(*continued*)

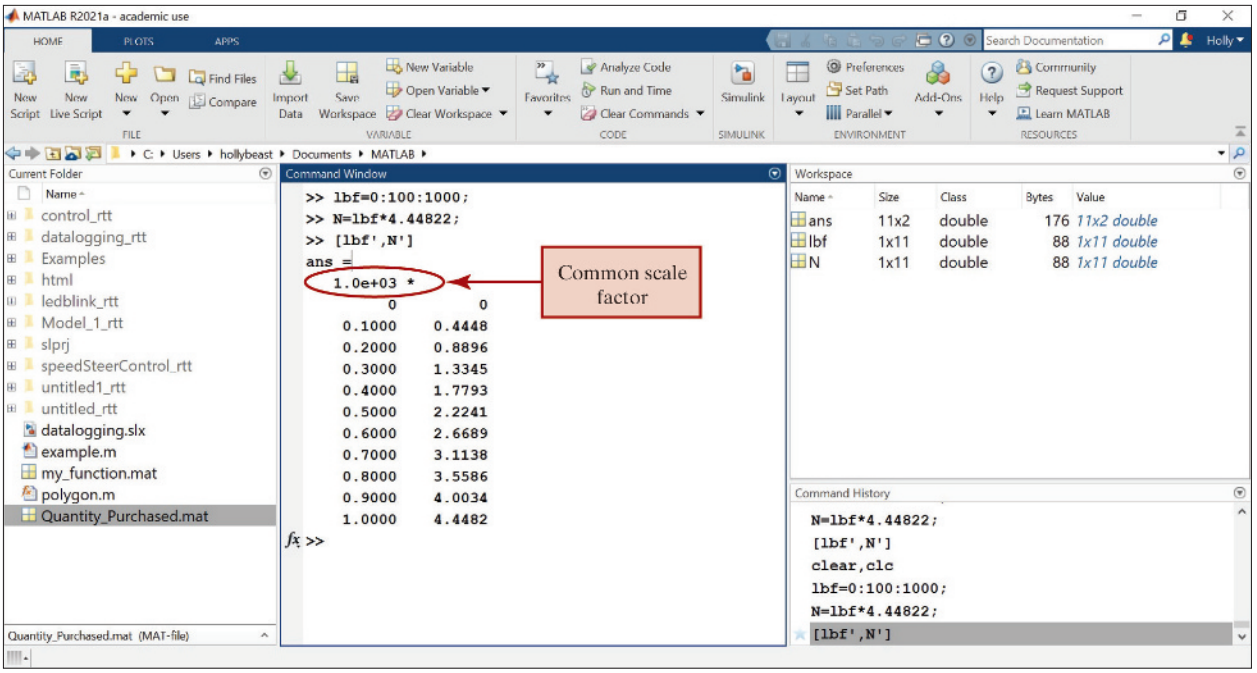


Figure 2.9
The MATLAB workspace window shows the variables as they are created.

It is always a good idea to clear both the workspace and the command window before starting a new problem. Notice in the workspace window (see Figure 2.9) that `lbf` and `N` are 1×11 arrays, and that `ans` (which is where the array we created is stored) is an 11×2 array. The output from the first two commands was suppressed by adding a semicolon at the end of each line. It would be very easy to create an array with more entries by changing the increment to 10 or even to 1. Notice also that you'll need to multiply the results shown in the command window by 1000 to get the correct answers. MATLAB tells you that this is necessary directly above the array, where the common scale factor is shown.

5. Test the solution
   Comparing the results of the MATLAB solution with the hand solution shows that they are the same. Once we've verified that our solution works, it's easy to use the same algorithm to create other conversion charts. For instance, modify this example to create a chart in the form of an array that converts newtons (N) to pounds force (lbf), with an increment of 10 N, from 0 N to 1000 N.

EXAMPLE 2.3

CALCULATING DRAG

One performance characteristic that can be determined in a wind tunnel is drag. The friction related to drag on the Mars Climate Observer (caused by the atmosphere of Mars) resulted in the spacecraft's burning up during course corrections. Drag is extremely important in the design of terrestrial aircraft as well (see Figure 2.10).

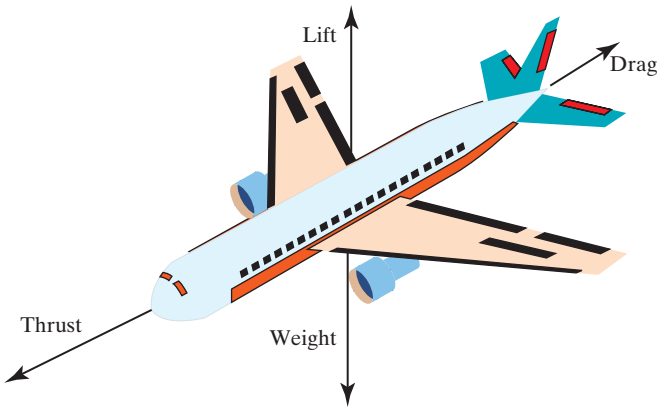


Figure 2.10
Drag is a mechanical force generated by a solid object moving through a fluid.

Drag is the force generated as an object, such as an airplane, moves through a fluid. In the case of a wind tunnel, air moves past a stationary model, but the equations are the same. Drag is a complicated force that depends on many factors. One factor is skin friction, which is a function of the surface properties of the aircraft, the properties of the moving fluid (air in this case), and the flow patterns caused by the shape of the aircraft (or, in the case of the Mars Climate Observer, by the shape of the spacecraft). Drag can be calculated with the drag equation

$$\text{drag} = C_\text{d} \frac{\rho V^2 A}{2}$$

where C_d = drag coefficient, which is determined experimentally, usually in a wind tunnel,
ρ = air density,
V = velocity of the aircraft, and
A = reference area (the surface area over which the air flows).

Although the drag coefficient is not a constant, it can be taken to be constant at low speeds (less than 200 mph). Suppose the following data were measured in a wind tunnel:

drag	200 N
ρ	1.225 kg/m^3
V	100 mph (you'll need to convert this to meters per second)
A	1 m^2

Calculate the drag coefficient, and use this experimentally determined value to predict how much drag will be exerted on the aircraft at velocities from 0 mph to 200 mph.

1. State the problem
Calculate the drag coefficient on the basis of the data collected in a wind tunnel. Use the drag coefficient to determine the drag at a variety of velocities.

2. Describe the input and output

Input

Drag	200 N
Air density ρ	1.225 kg/m^3
Velocity V	100 mph
Surface area A	1 m^2

(*continued*)

Output

Drag coefficient

Drag at velocities from 0 mph to 200 mph

3. Develop a hand example

First, find the drag coefficient from the experimental data. Notice that the velocity is in miles/h and must be changed to units consistent with the rest of the data (m/s). The importance of carrying units in engineering calculations cannot be overemphasized!

$$\begin{aligned} C_{\mathrm{d}} &= \frac{\text{drag} \times 2}{\rho \times V^2 \times A} \\ &= \frac{(200\ \mathrm{N} \times 2)}{1.225\ \mathrm{kg/m^3} \times \left(100\ \text{miles/h} \times 0.4470 \dfrac{\text{m/s}}{\text{miles/h}}\right)^2 \times 1\ \mathrm{m^2}} \\ &= 0.163 \end{aligned}$$

Since a newton is equal to a kg $\mathrm{m/s^2}$, the drag coefficient is dimensionless. Now use the drag coefficient to find the drag at different velocities:

$$\text{drag} = C_{\mathrm{d}} \times \rho \times V^2 \times A/2$$

Using a calculator, find the value of the drag with $V = 200$ mph:

$$\text{drag} = \frac{0.163 \times 1.225\ \mathrm{kg/m^3} \times \left(200\ \text{miles/h} \times 0.4470 \dfrac{\text{m/s}}{\text{miles/h}}\right)^2 \times 1\ \mathrm{m^2}}{2}$$

$$\text{drag} = 800\ \mathrm{N}$$

4. Develop a MATLAB solution

```
drag = 200;
density = 1.225;
velocity = 100*0.4470;
area = 1;
```

Define the variables, and change V to SI units.

```
cd = drag*2/(density*velocity^2*area)
cd =
 0.163
```

Calculate the coefficient of drag.

```
velocity = 0:20:200;
velocity = velocity*0.4470;
```

Redefine V as an array. Change it to SI units and calculate the drag.

```
drag = cd*density*velocity.^2*area/2;
results = [velocity', drag']
results =
1.0e+004 *
             0            0
          8.94            8
         17.88           32
         26.82           72
         35.76          128
          44.7          200
```

```
53.64          288
62.58          392
71.52          512
80.46          648
 89.4          800
```

Notice that the equation for drag, or

```
drag = cd * density * velocity.^2 * area/2;
```

uses the .^ operator, because we intend that each value in the array `velocity` be squared, not that the entire array `velocity` be multiplied by itself. Using just the exponentiation operator (^) would result in an error message. We could have used the .* operator as well in places where * was used, but since all the other quantities are scalars it doesn't matter. Unfortunately, it is possible to compose problems in which using the wrong operator does not give us an error message, but does give us a wrong answer. This makes step 5 in our problem-solving methodology especially important.

5. Test the solution
Comparing the hand solution with the MATLAB solution (see Figure 2.11), we see that they give the same results. Once we have confirmed that our algorithm works with sample data, we can substitute new data and be confident that the results will be correct. Ideally, the results should also be compared with experimental data, to confirm that the equations we are using accurately model the real physical process.

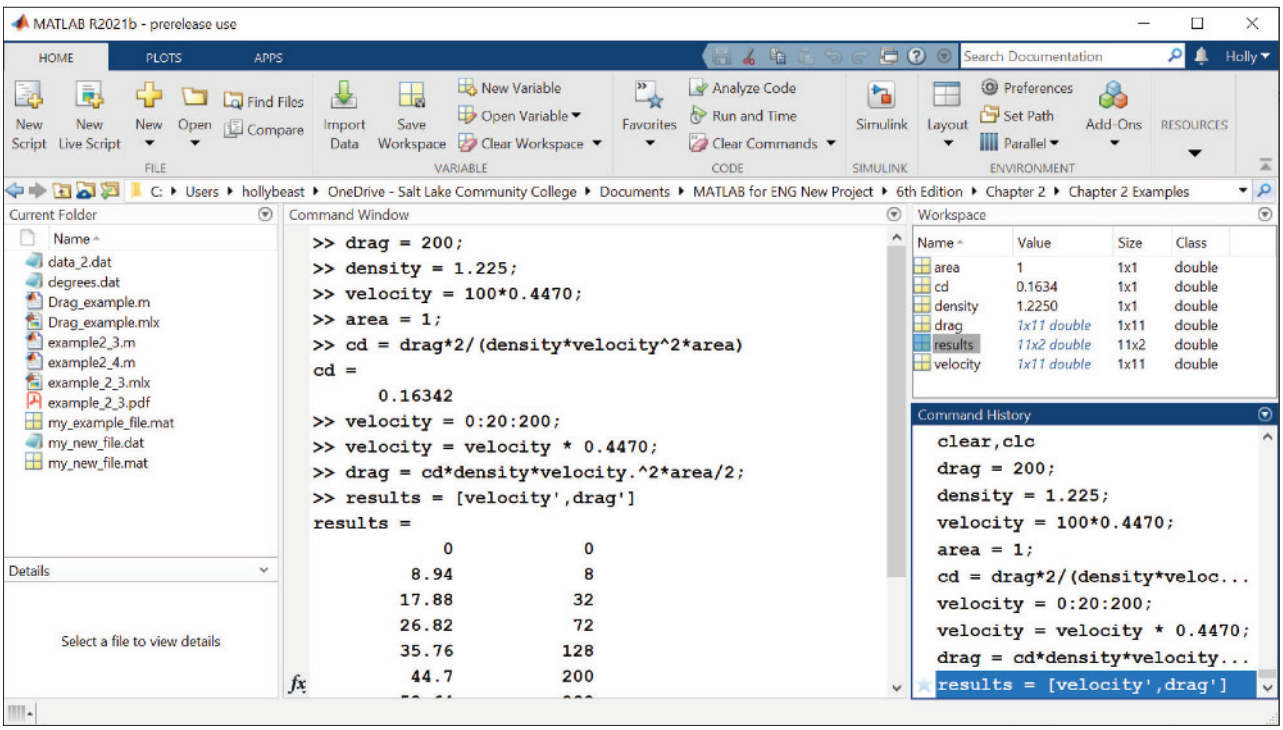


Figure 2.11
The command history window creates a record of previous commands.

(*continued*)

6. Comment

The drag coefficient calculated in this problem is quite high for today's high performance aircraft. Measured values range from 0.021 for an F-4 Phantom II to 0.031 for a Boeing 747 and 0.048 for an F-104 Starfighter. Automobile drag coefficients are higher and have changed from between 0.6 and 0.95 in the 1920s to between 0.2 and 0.4 in today's cars.

2.3.4 Number Display

KEY IDEA
Scientific notation allows very large and very small numbers to be represented efficiently.

Scientific Notation

Although you can enter any number in decimal notation, that isn't always the best way to represent very large or very small numbers. For example, a number that is used frequently in chemistry is Avogadro's constant, whose value, to four significant digits, is 602,200,000,000,000,000,000,000. Similarly, the diameter of an iron atom is approximately 140 picometers, which is 0.000000000140 m. **Scientific notation** expresses a value as a number between 1 and 10, multiplied by a power of 10 (the exponent). Thus, Avogadro's number becomes 6.022×10^{23}, and the diameter of an iron atom, 1.4×10^{-10} m. In MATLAB, values in scientific notation are designated with an **e** between the decimal number and the exponent. (Your calculator probably uses similar notation.) For example, you might have

```
Avogadro's_constant = 6.022e23;
Iron_diameter = 140e-12; or
Iron_diameter = 1.4e-10
```

It is important to omit blanks between the decimal number and the exponent. For instance, MATLAB will interpret

```
6.022 e23
```

as two values (6.022 and 10^{23}). Since putting two values in an assignment statement is an error, MATLAB will generate an error message.

HINT

Although it is a common convention to use *e* to identify a power of 10, students (and teachers) sometimes confuse this nomenclature with the mathematical constant *e*, which is equal to 2.7183. To raise *e* to a power, use the `exp` function, for example `exp(3)` is equivalent to e^3.

KEY IDEA
MATLAB does not differentiate between integers and floating-point numbers, unless special functions are invoked.

KEY IDEA
No matter what display format is selected, calculations are performed using double-precision floating-point numbers.

Display Format

A number of different display formats are available in MATLAB. No matter which display format you choose, MATLAB uses double-precision floating-point numbers in its calculations, which results in approximately 16 decimal digits of precision. Changing the display format does not change the accuracy of the results. Unlike some other computer programs, by default MATLAB handles both integers and decimal numbers as floating-point numbers.

When elements of an array are displayed in MATLAB, integers are always printed without a decimal point. However, values with decimal fractions are printed in the default short format that shows four digits after the decimal point. Thus,

```
A = 5
```

returns

```
A =
      5
```

but

```
A = 5.1
```

returns

```
A =
      5.1000
```

and

```
A = 51.1
```

returns

```
A =
      51.1000
```

MATLAB allows you to specify other formats that show additional digits. For example, to specify that you want values to be displayed in a decimal format with 15 digits after the decimal point, use the command

```
format long
```

which changes all subsequent displays. Thus, with `format long` specified,

```
A
```

now returns

```
A =
      51.100000000000001
```

Notice that the final digit in this case is 1, which represents a round-off error. Two decimal digits are displayed when the format is specified as `format bank`:

```
A =
      51.10
```

The bank format displays only real numbers, so it's not appropriate when complex numbers need to be represented. Thus, the command

```
A = 5+3i
```

returns the following using bank format

```
A =
      5.00
```

Using `format long` the same command returns

```
A =
      5.000000000000000 + 3.000000000000000i
```

You can return the format to four decimal digits with the command

```
format short
```

To check the results, recall the value of `A`:

```
A
A =
     5.0000 + 3.0000i
```

When numbers become too large or too small for MATLAB to display in the default format, it automatically expresses them in scientific notation. For example, if you enter Avogadro's constant into MATLAB in decimal notation as

```
a = 602000000000000000000000
```

the program returns

```
a =
    6.0200e+023
```

You can force MATLAB to display all numbers in scientific notation with `format short e` (with four decimal digits) or `format long e` (with 15 decimal digits). For instance,

```
format short e
x = 10.356789
```

returns

```
x =
     1.0357e+001
```

Another pair of formats that are often useful to engineers and scientists, `format short eng` and `format long eng`, are similar to scientific notation but require the power of 10 to be a multiple of three. This corresponds to common naming conventions. For example:

$$\begin{aligned} 1 \text{ millimeter} &= 1 \times 10^{-3} \text{ meters} \\ 1 \text{ micrometer} &= 1 \times 10^{-6} \text{ meters} \\ 1 \text{ nanometer} &= 1 \times 10^{-9} \text{ meters} \\ 1 \text{ picometer} &= 1 \times 10^{-12} \text{ meters} \end{aligned}$$

Consider the following example. First, change to engineering format and then enter a value for *y*.

```
format short eng
y = 12000
```

which gives the result

```
y =
    12.0000e+003
```

When an array of values is sent to the screen, and if the elements become very large or very small, a common scale factor is often applied to the entire array. This scale factor is printed along with the scaled values. For example, when the command window is returned to

```
format short
```

the results from Example 2.3 are displayed as follows:

```
results  =
1.0e+005 *
                0           0
           0.0002      0.0400
           0.0004      0.1602
           0.0006      0.3603
           0.0008      0.6406     etc...
```

Two other formats that you may occasionally find useful are `format +` and `format rat`. When an array is displayed in `format +`, the only characters printed are plus and minus signs. If a value is positive, a plus sign will be displayed; if a value is negative, a minus sign will be displayed. If a value is zero, nothing will be displayed. This format allows us to view a large array in terms of its signs:

```
format +
B = [1, -5, 0, 12; 10005, 24, -10,4]
B =
       +- +
       ++-+
```

The `format rat` command displays numbers as **rational numbers** (i.e., as fractions). Thus,

```
format rat
x = 0:0.1:0.5
```

returns

```
x =
    0   1/10   1/5   3/10   2/5   1/2.
```

KEY IDEA

The `format shortg` or `format longg` are the most popular formats used by engineers.

If you're not sure which format is the best for your application, you may select `format short g` or `format long g` (`format shortg` or `format longg` is an alternate syntax). This format selects the best of fixed-point or floating-point representations. Many people find this to be the most generally useful of the format styles, since it eliminates the use of the common multiplier.

The `format` command also allows you to control how tightly information is spaced in the command window. The default (`format loose`) inserts a line feed between user-supplied expressions and the results returned by the computer. The `format compact` command removes those line feeds. The examples in this text use the compact format to save space. Table 2.2 shows how the values of π and of 123.456 are displayed in each format.

If none of these predefined numeric display formats is right for you, you can control individual lines of output with the `fprintf` function, described in a later chapter.

2.4 SAVING YOUR WORK

Working in the command window is like performing calculations on your scientific calculator. When you turn off the calculator or when you exit the program, your work is gone. There are two different ways you may be interested in saving your work; saving the *results* of the calculations you performed or saving the actual code

Table 2.2 Numeric Display Formats

MATLAB Command	Display	Example
format short	4 decimal digits	**3.1416**
format long	14 decimal digits	**3.14159265358979**
format short e	4 decimal digits scientific notation	**3.1416e+000** **1.2346e+002**
format long e	14 decimal digits scientific notation	**3.141592653589793e+000** **1.234567890000000e+002**
format bank	2 decimal digits, only real values are displayed	**3.14**
format short eng	4 decimal digits engineering notation	**3.1416e+000** **123.4568e+000**
format long eng	14 decimal digits engineering notation	**3.141592653589793e+000** **123.456789000000e+000**
format +	+,−, blank	**+**
format rat	fractional form	**355/113**
format short g	MATLAB selects the best format	**3.1416** **123.46**
format long g	MATLAB selects the best format	**3.14159265358979** **123.456789**

entered into MATLAB so you can repeat the calculations in the future. In other words, creating a computer program to solve a problem.

2.4.1 Saving Variables

To preserve the values stored in variables you create in the command window (check the workspace window on the left-hand side of the MATLAB screen for the list of variables), you must save the contents of the workspace window to a file. The default format is a binary file called a **MAT-file**. To save the workspace to a file, type

```
save <file_name>
```

KEY IDEA
The MAT-file uses a proprietary format to save variables stored in the MATLAB workspace.

at the prompt. Although `save` is a MATLAB command, *file_name* is a user-defined file name. It can be any name you choose, as long as it conforms to the naming conventions for variables in MATLAB. Actually, you don't even need to supply a file name. If you don't, MATLAB names the file *matlab.mat*. You could also choose

```
Save Workspace
```

from the toolstrip, which will then prompt you to enter a file name for your data. To restore a workspace, type

```
load <file_name>
```

Again, `load` is a MATLAB command, but *file_name* is the user-defined file name. If you just type `load`, MATLAB will look for the default *matlab.mat* file.

The file you save will be stored in the current folder.

For example, type

```
clear, clc
```

This command will clear both the workspace and the command window. Verify that the workspace is empty by checking the workspace window or by typing

```
whos
```

Now define several variables—for example,

```
a = 5;
b = [1, 2, 3];
c = [1, 2; 3, 4];
```

Check the workspace window once again to confirm that the variables have been stored. Now, save the workspace to a file called *my_example_file*:

```
save my_example_file
```

Confirm that a new file has been stored in the current folder. If you prefer to save the file to another directory (for instance, onto a flash drive), browse (see Figure 2.2) to the directory of your choice. Remember that in a public computer lab, the current folder is probably purged after each user logs off the system.

Now, clear the workspace and command window by typing

```
clear, clc
```

The workspace window should be empty. You can recover the missing variables and their values by loading the file *(my_example_file.mat)* back into the workspace:

```
load my_example_file
```

The file you want to load must be in the current folder, or MATLAB won't be able to find it. In the command window, type

```
a
```

which returns

```
a =
    5
```

Similarly you can retrieve any of the variables now listed in the workspace window.

MATLAB can also store individual arrays or lists of arrays into a file in the current folder with the command

```
save <file_name> <variable_list>
```

where **file_name** is the user-defined file name designating the location in memory at which you wish to store the information, and *variable_list* is the list of variables to be stored in the file. For example,

```
save my_new_file a b
```

would save just the variables `a` and `b` into *my_new_file.mat*.

KEY IDEA
The ASCII file format is an industry standard and can be used to share information between dissimilar programs.

If your saved data will be used by a program other than MATLAB (such as C or C++), the .mat format is not appropriate, because .mat files are unique to MATLAB. The **ASCII** format is standard between computer platforms, and is more appropriate if you need to share files. MATLAB allows you to save files as ASCII files by modifying the save command to

```
save <file_name> <variable_list> -ascii
```

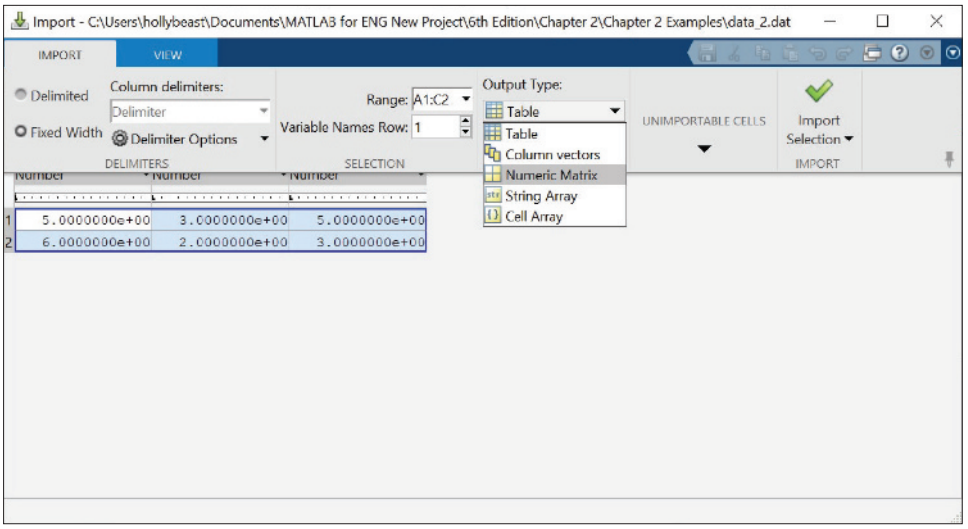


Figure 2.12
Double-clicking the file name in the command directory launches the Import Wizard.

The command `-ascii` tells MATLAB to store the data in a standard eight-digit text format. ASCII files should be saved into a .dat file or .txt file instead of a .mat file; be sure to add the extension to your file name:

```
save my_new_file.dat a b -ascii
```

If you don't add .dat, MATLAB will default to .mat.
If more precision is needed, the data can be stored in a 16-digit text format:

```
save file_name variable_list -ascii -double
```

You can retrieve the data from the current folder with the load command:

```
load <file_name>
```

For example, to create the array `z` and save it to the file *data_2.dat* in eight-digit text format, use the following commands:

```
z = [5 3 5; 6 2 3];
save data_2.dat z -ascii
```

Together, these commands cause each row of the array `z` to be written to a separate line in the data file. You can view the *data_2.dat* file by double-clicking the file name in the current folder window (see Figure 2.12). If your file contains a single array, the easiest way to retrieve data from an ASCII .dat file is to enter the `load` command followed by the file name. This causes the information to be read into an array with the same name as the data file. This won't work if your file contains multiple variables. However, it is quite easy to use MATLAB's interactive Import Wizard to load the data. When you double-click a data file name in the current folder to view the contents of the file, the Import Wizard will automatically launch. Just follow the directions to load the data into the workspace, with the same name as the data file. Be sure to specify the data output type from the drop-down menu on the toolstrip. (The default data type is a table, which is discussed in a later chapter.) You can use this same technique to import data from other programs, including Excel spreadsheets, or you can select Import Data from the toolstrip.

KEY IDEA
When you save the workspace, you save only the variables and their values; you do not save the commands you've executed.

It is important to realize that only the variable arrays listed in the workspace window are stored in the data files created using the techniques described in this section. The commands used to create the arrays are not stored.

2.4.2 Programs

KEY IDEA
MATLAB programs can be stored either as scripts or live scripts.

Using the command window for calculations is an easy and powerful tool. However, once you close the MATLAB program, your calculations are gone. Fortunately, MATLAB contains a powerful programming language. As a programmer, you can create and save code in program files. These files can be reused anytime you wish to repeat your calculations. A MATLAB program is stored as an ASCII text file like a C or FORTRAN source-code file. MATLAB offers two built-in editors for you to use from the MATLAB desktop; the script and the live script editor. They can be created and edited with the MATLAB editor/debugger (the edit window discussed in Section 2.2.7).

Scripts

The original MATLAB programs were all written in scripts and stored in files with a .m extension and were therefore often called M-files. Most of the existing MATLAB code you are likely to see will have been written in a script. The *code* is exactly the same for a script program as for a live script. The only difference is the editor.

To open the editing window for a script, select

```
New Script
```

from the MATLAB toolstrip. The MATLAB edit window is shown in Figure 2.13 with the code used to solve Example 2.3. Many programmers prefer to dock the editing window onto the MATLAB desktop, using the show window actions drop-down menu from the upper right-hand corner of the window (select dock editor). This allows you to see both the contents of the script and the results displayed when the program is executed. The results are displayed in the command window.

When you save a script, it is stored in the current folder. You'll need to name your file with a valid MATLAB variable name—that is, a name starting with a letter and containing only letters, numbers, and the underscore (_). Spaces are not allowed (see Section 2.3.1).

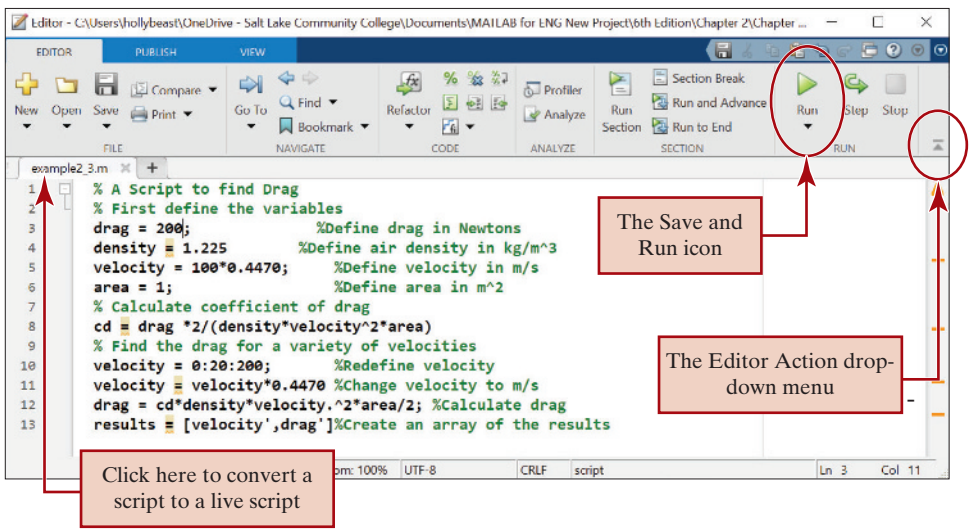


Figure 2.13
The MATLAB edit window, also called the editor/debugger.

Table 2.3 Approaches to Executing a Script from the Command Window

MATLAB Command	Comments
myscript	Type the file name, for example **myscript**. The .m file extension is assumed.
run myscript	Use the run command with the file name.
run('myscript')	Use the functional form of the run command.

There are three types of M-files, called scripts, functions, and classes. We'll start out by learning about simple scripts and add functions and classes later. A script M-file is a list of MATLAB statements that are saved in a file with a .m file extension. The script can use any variables that have been defined in the workspace, and any variables created in the script are added to the workspace when the script executes. You can execute a script created in the MATLAB edit window by selecting the *Save and Run* icon from the menu bar, as shown in Figure 2.13. You can also execute a script by typing a file name, or by using the run command from the command window as shown in Table 2.3. No matter how you do it, you can only run a MATLAB program if it is in the current folder or you use the functional form of the run command with the complete file address.

KEY IDEA
The three types of M-files are scripts, functions, and classes.

You can find out what MATLAB programs, functions, and MAT files are in the current folder by typing

```
what
```

into the command window. You can also browse through the current folder by looking in the current folder window.

Using MATLAB programs allows you to work on a project and to save the list of commands for future use. Because you will be using these programs in the future, it is a good idea to sprinkle them liberally with comments. The comment operator in MATLAB is the percentage sign, as in

```
% This is a comment.
```

MATLAB will not execute any code on a commented line.

You can also add comments after a command, but on the same line:

```
a = 5     % The variable a is defined as 5
```

Here is an example of MATLAB code that could be entered into a script and used to solve Example 2.3:

```
clear, clc
% A Script to find Drag
% First define the variables
drag = 200;                        % Define drag in Newtons
density = 1.225;                   % Define air density in kg/m^3
velocity = 100*0.4470;             % Define velocity in m/s
area = 1;                          % Define area in m^2
% Calculate coefficient of drag
cd = drag *2/(density*velocity^2*area)
% Find the drag for a variety of velocities
velocity = 0:20:200;               % Redefine velocity
velocity = velocity*0.4470         % Change velocity to m/s
drag = cd*density*velocity.^2*area/2;      % Calculate drag
results = [velocity',drag']      % Create an array of the results
```

This code can be executed either from the script (by clicking the *Save and Run* icon) or from the command window (by typing in the file name). The results will appear in the command window in either case, and the variables will be stored in the workspace.

HINT

You can execute a portion of a script by highlighting a section and then right-clicking and selecting *Evaluate Section*. You can also comment or "uncomment" whole sections of code from this menu; doing so is useful when you are creating programs while you are still debugging your work.

Example 2.4 uses a script to find the velocity and acceleration that a spacecraft might reach in leaving the solar system.

EXAMPLE 2.4

CREATING A SCRIPT TO CALCULATE THE ACCELERATION OF A SPACECRAFT

In the absence of drag, the propulsion power requirements for a spacecraft are determined fairly simply. From basic physical science

$$F = ma$$

In other words, force (F) is equal to mass (m) times acceleration (a). Work (W) is force times distance (d), and since power (P) is work per unit time, power becomes force times velocity (v):

$$W = Fd$$

$$P = \frac{W}{t} = F \times \frac{d}{t} = F \times v = m \times a \times v$$

This means that the power requirements for the spacecraft depend on its mass, how fast it's going, and how quickly it needs to speed up or slow down. If no power is applied, the spacecraft just keeps traveling at its current velocity. As long as we don't want to do anything quickly, course corrections can be made with very little power. Of course, most of the power requirements for spacecraft are not related to navigation. Power is required for communication, for housekeeping, and for science experiments and observations.

The *Voyager 1* and *Voyager 2* spacecraft explored the outer solar system during the last quarter of the 20th century (see Figure 2.14). *Voyager 1* encountered both Jupiter and Saturn; *Voyager 2* not only encountered Jupiter and Saturn, but continued on to Uranus and Neptune. The *Voyager* program was enormously successful, and the *Voyager* spacecraft continue to gather information as they leave the solar system. The power generators (low-level nuclear reactors) on each spacecraft are expected to function until at least 2025. The power source is a sample of plutonium-238, which, as it decays, generates heat that is used to produce electricity. At the launch of each spacecraft, its generator produced about 470 watts of power. Not only is the plutonium decaying, the onboard thermocouples are degrading. The power production had decreased to about 270 watts in 2011, almost 35 years after launch. This power is used to operate the science package, but if it were diverted to propulsion, how much acceleration would it produce in the spacecraft? *Voyager 1* is currently traveling at a

(*continued*)

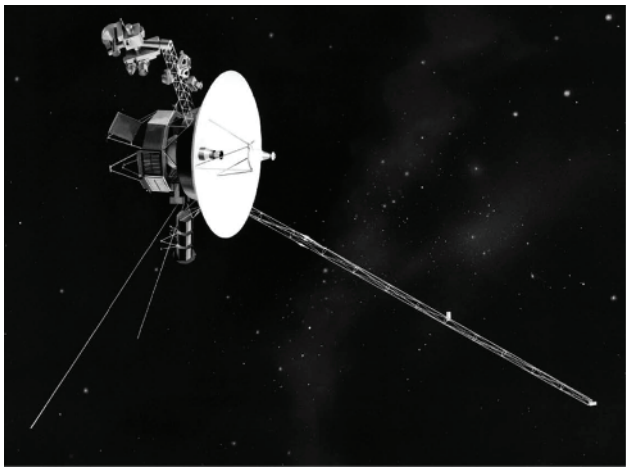


Figure 2.14
The *Voyager 1* and *Voyager 2* spacecraft were launched in 1977 and have since left the solar system. (NASA Headquarters)

velocity of 3.50 AU/year (an AU is an astronomical unit), and *Voyager 2* is traveling at 3.15 AU/year. Each spacecraft weighs 721.9 kg.

1. State the problem
Find the acceleration that is possible with the power output from the spacecraft power generators.

2. Describe the input and output
Input
Mass = 721.9 kg
Power = 270 watts = 270 J/s
Velocity = 3.50 AU/year (*Voyager 1*)
Velocity = 3.15 AU/year (*Voyager 2*)

Output
Acceleration of each spacecraft, in m/s/s

3. Develop a hand example
We know that

$$P = m \times a \times v$$

which can be rearranged to give

$$a = \frac{p}{m \times v}$$

The hardest part of this calculation will be keeping the units straight. First let's change the velocity to m/s. For *Voyager 1*,

$$v = 3.50\frac{\text{AU}}{\text{year}} \times \frac{150 \times 10^9 \text{m}}{\text{AU}} \times \frac{\text{year}}{365 \text{ days}} \times \frac{\text{day}}{24 \text{ h}} \times \frac{\text{h}}{3600 \text{ s}} = 16{,}650 \text{ m/s}$$

Then, we calculate the acceleration:

$$a = \frac{270 \text{ J/s} \times 1 \text{ kg} \times \text{m}^2/\text{s}^2 \text{ J}}{721.9 \text{ kg} \times 16{,}650 \text{ m/s}} = 2.7 \times 10^{-5} \text{ m/s}^2$$

4. Develop a MATLAB solution

```
clear, clc
% Example 2.4
% Find the possible acceleration of the Voyager 1
% and Voyager 2 Spacecraft using the on board power
% generator
```

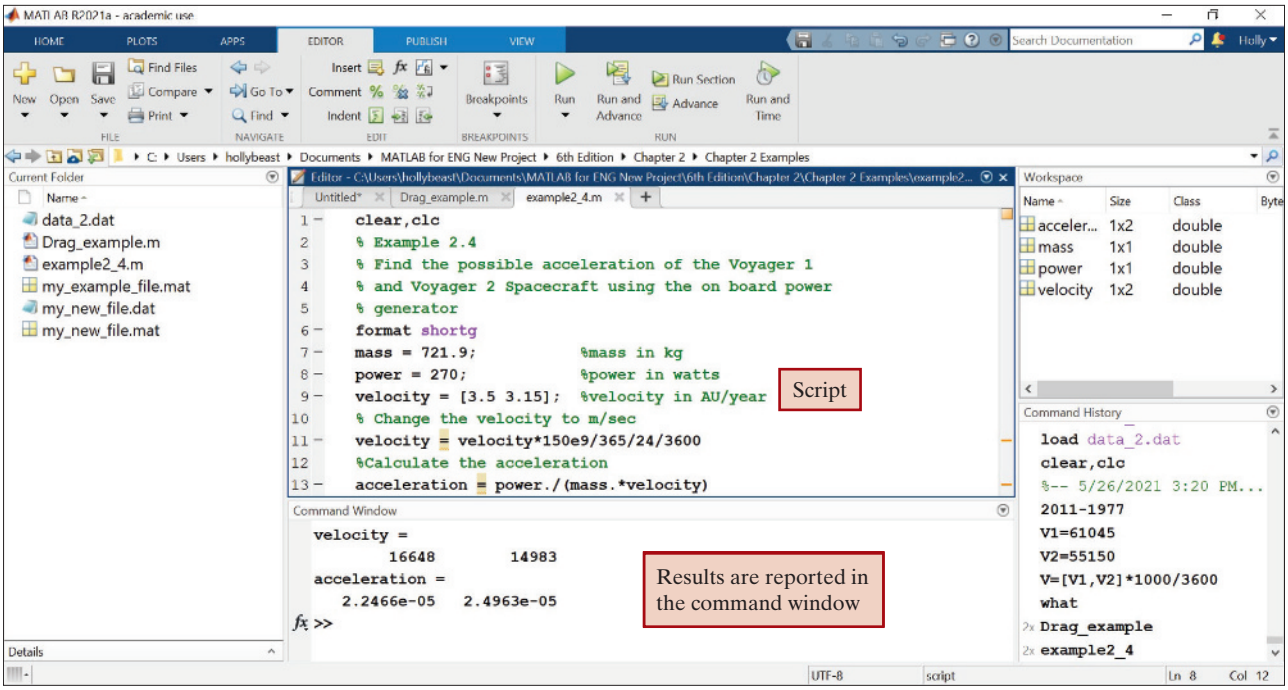


Figure 2.15
The results of a script execution are reported in the command window. The variables created are reflected in the workspace and the script is listed in the current folder window. The commands issued in the script are not mirrored in the command history.

```
format shortg
mass = 721.9;                % mass in kg
power = 270;                 % power in watts
velocity = [3.5 3.15];       % velocity in AU/year
% Change the velocity to m/sec
velocity = velocity*150e9/365/24/3600
% Calculate the acceleration
acceleration = power./(mass.*velocity)
```

To evaluate the program, click the *Save and Run* icon. The results are printed in the command window, as shown in Figure 2.15.

5. Test the solution

Compare the MATLAB results with the hand example results. Notice that the velocity and acceleration calculated from the hand example and the MATLAB solution for *Voyager 1* match. The acceleration seems quite small, but applied over periods of weeks or months such an acceleration can achieve significant velocity changes. For example, a constant acceleration of 2.25×10^{-5} m/s^2 results in a velocity change of about 58 m/s over the space of a month:

$$2.25 \times 10^{-5} \text{ m/s}^2 \times 3600 \text{ s/h} \times 24 \text{ h/day} \times 30 \text{ days/month} = 58.3 \text{ m/s}$$

Now that you have a MATLAB program that works, you can use it as the starting point for other, more complicated calculations.

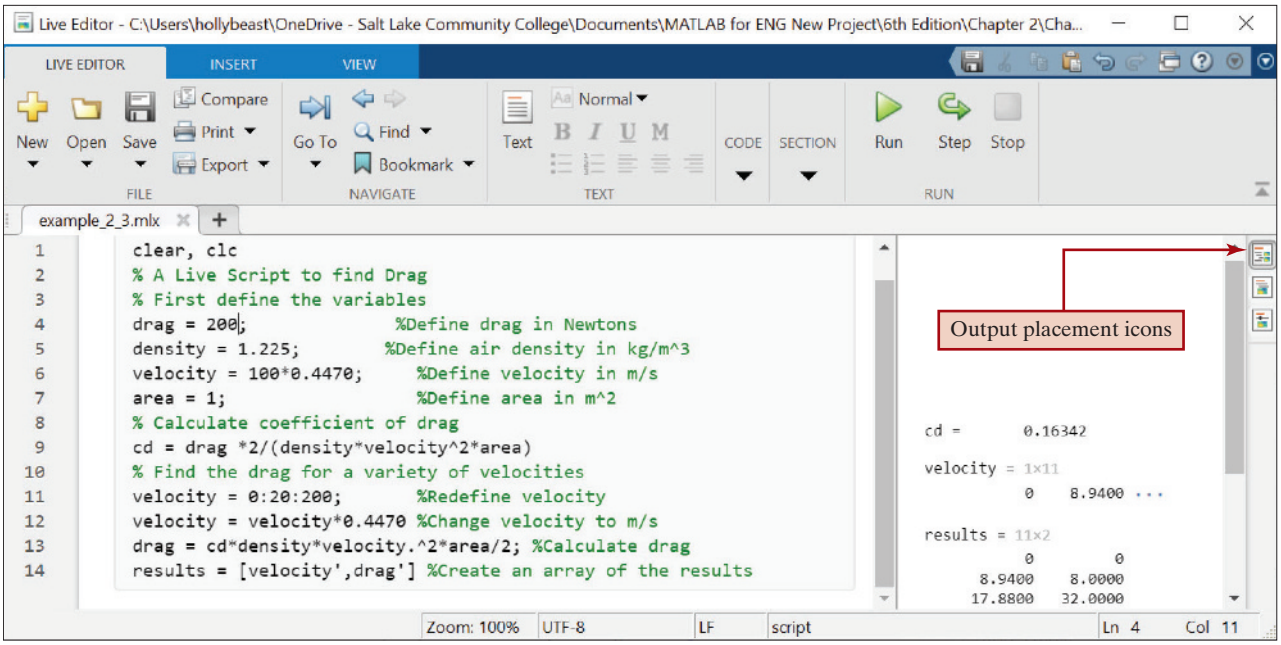


Figure 2.16
Live Scripts display the program output in the same window as the program code.

Live Scripts

Live scripts are interactive program files that contain not only your code, but also text, images, formulas, and output, including the graphics created by the program. The MATLAB code is the same as the code entered into a script M-file. Live scripts are stored in files with the .mlx extension. In addition to live scripts, MLX-files can also store functions, but unlike M-files cannot store methods (classes). These other types of MLX-files will be introduced in later chapters.

KEY IDEA
The two types of MLX-files are live scripts and functions.

If you already have an existing script, you can convert it to a live script by right-clicking the filename on the tab in the editor window, as shown in Figure 2.13, and selecting Open as Live Script. This first example program won't look much different, but once the code is executed the results are displayed by default to the right of the code segment (Figure 2.16). You can select different display output options by selecting one of the output placement icons on the far right of the window. In addition to viewing the code on the right as shown, you can choose to view the output immediately below the line of code that generated it (inline), or you can hide the code and only show the output.

Live scripts can be saved not only as MLX-files, but can be saved in several formats suitable for incorporation into reports or other documents including PDFs, Word documents, and HTML files for web publication. They can also be converted back to M-files by selecting MATLAB Code files from the Save As menu.

2.4.3 Section Mode

KEY IDEA
Section mode allows you to execute portions of the code incrementally.

Section mode is a utility that allows the user to divide both scripts and live scripts into sections that can be executed one at a time. This feature is particularly useful as you develop MATLAB programs.

To divide M-file programs into sections, create section dividers by using a double percentage sign followed by a space or select the Section Break icon from the

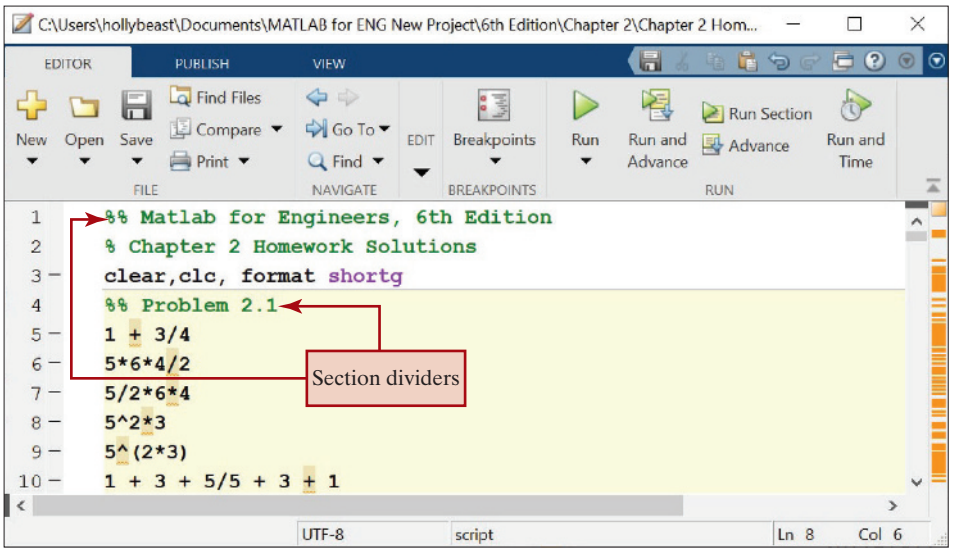


Figure 2.17
Section mode allows the user to execute one cell, or section, at a time.

toolstrip. If you want to name the section, just add a name on the same line as the section divider:

```
%% Section Name
```

It's important to include the space after the double percentage sign (%%). If you don't, the line is recognized as a comment, not a section divider.

Once the section dividers are in place, if you position the cursor anywhere inside the section, the entire section turns pale yellow. For example, in Figure 2.17, the first four lines of the program make up the first section. Now we can use the run icons on the editing tab of the toolstrip to evaluate a single section, evaluate the current section and move on to the next section, or evaluate the entire file. The Navigate menu lists all the sections in the script and allows you to jump to a particular section or function (functions are covered in another chapter), as shown in Figure 2.18.

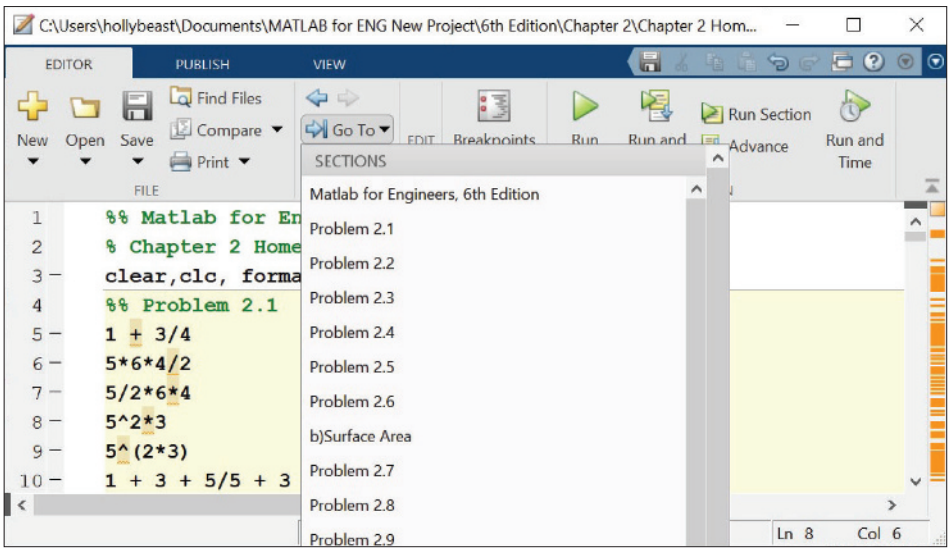


Figure 2.18
The Navigate drop-down menu lists all the sections in the M-file.

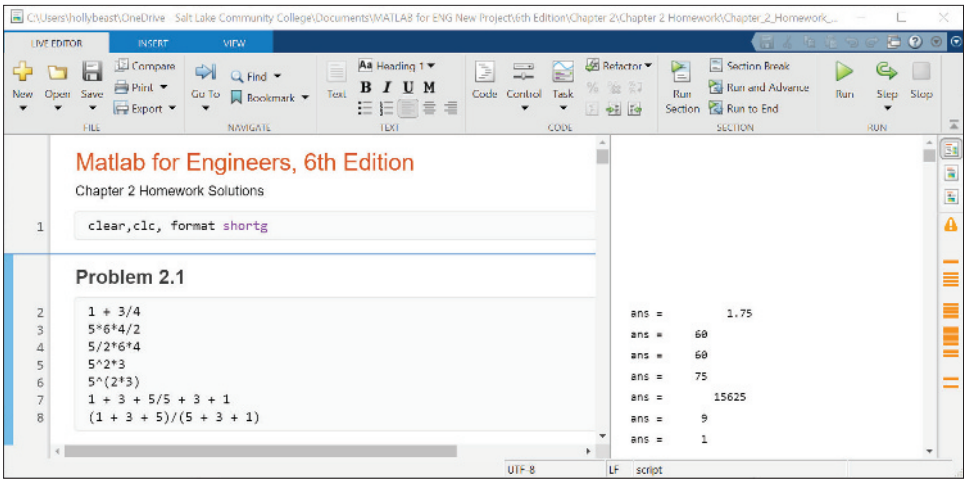


Figure 2.19
When a MATLAB script that has used sections is converted to a live script, automatic formatting is implemented.

When the program shown in Figures 2.17 and 2.18 is converted to a live script, each section is separated with a section divider and a title is automatically added as are section headings and text. Additional text or code can be added much like you would work with any word processor, making the appropriate selections on the toolstrip (Figure 2.19). Selecting the Insert tab allows you to insert images, hyperlinks, equations, and even a table of contents based on the section headings as illustrated in Figure 2.20. Finally, selecting the View tab (Figure 2.21) allows you to adjust the view, including whether to display line numbers. Especially useful is the option to clear the output after running the program.

To create a program as a live script without starting from a script, select New Live Script from the MATLAB toolstrip. You can add text or code by making the appropriate selections from the toolstrip. All the guidelines for creating code in a MATLAB script apply to creating code in a MATLAB live script.

The previous figures show the first few lines of a program written to solve some homework problems. By dividing the program into sections, it was possible to work on each problem separately.

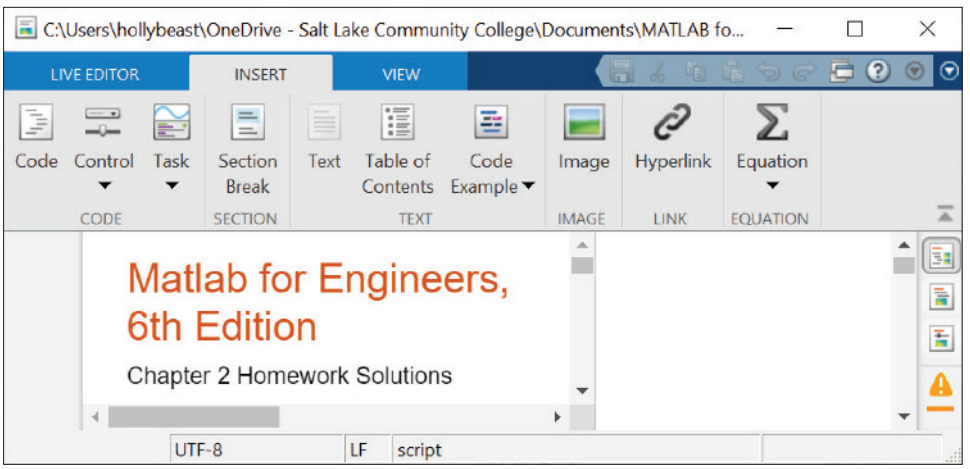


Figure 2.20
The Insert tab allows programmers to add images, hyperlinks, equations, and even a table of contents to a live script, in addition to the text and code that are the basic elements of the program.

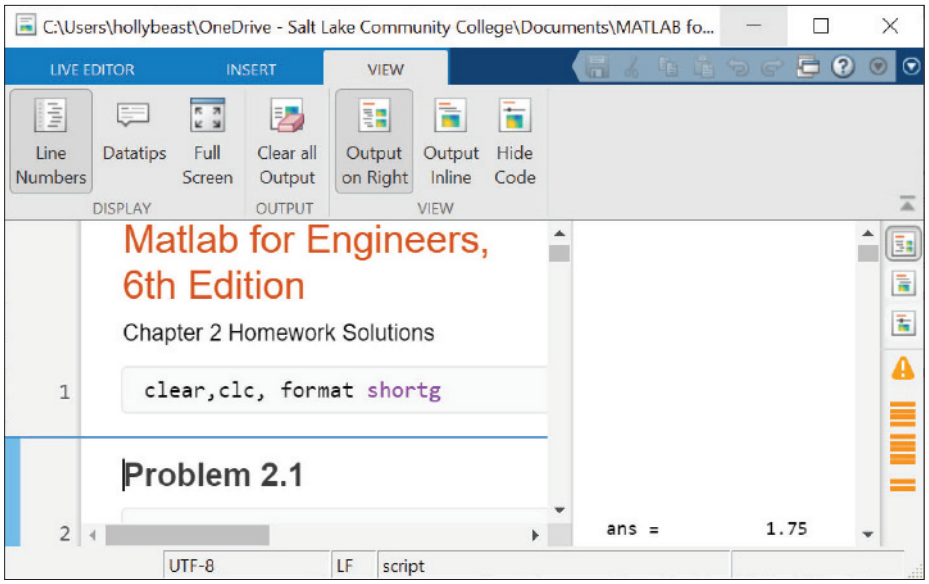


Figure 2.21
The View tab allows programmers to adjust the way the program and its output is viewed by the user, including resetting the output after the program has been executed.

Dividing a program into sections offers a big advantage to the person who must evaluate it. By using the `Run and Advance` function (Figures 2.17 and 2.19), the evaluator can step through the program one section at a time. Even more importantly, the programmer can divide a complicated project into manageable sections and evaluate these sections independently.

SUMMARY

In this chapter, we introduced the basic MATLAB structure. The MATLAB environment includes multiple windows, three of which are open in the default view:

- Command window
- Workspace window
- Current folder window

In addition, the environment includes the

- Command history window
- Document window
- Graphics window
- Edit window

which open as needed during a MATLAB session.

Variables defined in MATLAB follow common computer naming conventions:

- Names must start with a letter.
- Letters, numbers, and the underscore are the only characters allowed.
- Names are case sensitive.
- Names may be of any length, although only the first 63 characters are used by MATLAB.

- Some keywords are reserved by MATLAB and cannot be used as variable names.
- MATLAB allows the user to reassign function names as variable names, although doing so is not good practice.

The basic computational unit in MATLAB is the array. Arrays may be:

- Scalars (1×1 array)
- Vectors ($1 \times n$ or $n \times 1$ array, either a row or a column)
- Two-dimensional arrays ($m \times n$ or $n \times m$)
- Multidimensional arrays

Arrays often store numeric information, although they can store other kinds of information as well.

The kind of information stored is called a data type or class. Numeric, character, string, logical, and table data types were introduced in this chapter, although many additional data types exist in the MATLAB programming language. Data can be entered into a numeric array manually or can be retrieved from stored data files. When entered manually, a numeric array is enclosed in square brackets, elements in a row are separated by either commas or spaces, and a new row is indicated by a semicolon:

```
a = [1 2 3 4; 5 6 7 8]
```

Evenly spaced arrays can be generated with the colon operator. Thus, the command

```
b = 0:2:10
```

creates an array starting at 0, ending at 10, and with an increment of 2. The `linspace` and `logspace` functions can be used to generate an array of specified length from given starting and ending values, spaced either linearly or logarithmically. The `help` function or the MATLAB Help menu can be used to determine the appropriate syntax for these and other functions.

MATLAB follows the standard algebraic order of operations.

MATLAB supports both standard (decimal) and scientific notation. It also supports a number of different display options. No matter how values are displayed, they are stored as double-precision floating-point numbers.

MATLAB variables can be saved or imported from either .mat or .dat files. The .mat format is proprietary to MATLAB and is used because it stores data more efficiently than other file formats. The .dat format employs the standard ASCII format, and is used when data created in MATLAB will be shared with other programs.

Collections of MATLAB commands can be saved in program files as script M-files or in live script MLX-files. This is the best way to save the list of commands used to solve a problem so that they can be reused at a later time. Section mode allows the programmer to group code into sections and to run each section individually. It is especially convenient when one program is used to solve multiple problems.

MATLAB SUMMARY

The following MATLAB summary lists all the special characters, commands, and functions that were defined in this chapter:

Special Characters	
`[]`	forms arrays
`()`	used in statements to group operations; used with an array name to identify specific elements
`,`	separates array elements
`;`	separates rows in an array definition; suppresses output when used in commands
`:`	used to generate arrays; indicates all rows or all columns
`=`	assignment operator assigns a value to a memory location; not the same as an equality
`%`	indicates a comment
`%%`	section divider
`+`	scalar and array addition
`–`	scalar and array subtraction
`*`	scalar multiplication and multiplication in matrix algebra
`.*`	array multiplication (dot multiply)
`/`	scalar division and division in matrix algebra
`./`	array division (dot divide)
`^`	scalar exponentiation and matrix exponentiation in matrix algebra
`.^`	array exponentiation (dot power or dot caret)

Commands and Functions	
`ans`	default variable name for results of MATLAB calculations
`ascii`	indicates that data should be saved in standard ASCII format
`clc`	clears command window
`clear`	clears workspace
`exit`	terminates MATLAB
`format +`	sets format to plus and minus signs only
`format compact`	sets format to compact form
`format long`	sets format to 14 decimal places
`format long e`	sets format to scientific notation with 14 decimal places
`format long eng`	sets format to engineering notation with 14 decimal places
`format long g`	allows MATLAB to select the best format (either fixed point or floating point), using 14 decimal digits
`format loose`	sets format to the default, noncompact form
`format short`	sets format to the default, 4 decimal places
`format short e`	sets format to scientific notation with 4 decimal places
`format short eng`	sets format to engineering notation with 4 decimal places
`format short g`	allows MATLAB to select the best format (either fixed point or floating point), using 4 decimal digits
`format rat`	sets format to rational (fractional) display
`help`	invokes help utility
`linspace`	linearly spaced vector function
`load`	loads arrays from a file

(*continued*)

Commands and Functions	
`logspace`	logarithmically spaced vector function
`namelengthmax`	Finds the maximum variable name length
`pi`	numeric approximation of the value of π
`quit`	terminates MATLAB
`save`	saves variables in a file
`who`	lists variables in memory
`whos`	lists variables and their sizes

KEY TERMS

array
array editor
array operators
ASCII
assignment
class
command history
command window
current folder
data type
document window
dot operators
edit window
explicit list
function
graphics window
live script
M-file
matrix
MAT-file
MLX-file
operator
prompt
rational number
scalar
scientific notation
script
section
toolstrip
transpose
variable
vector
workspace

PROBLEMS

You can either solve these problems in the command window, using MATLAB as an electronic calculator, or you can create a program using a script (M-file) or a live script (MLX = file). If you are solving these problems as a homework assignment or if you want to keep a record of your work, the best strategy is to use a program file, divided into sections with section dividers.

Getting Started

2.1 Predict the outcome of the following MATLAB calculations:

```
1 + 3/4
5*6*4/2
5/2*6*4
5^2*3
5^(2*3)
1 + 3 + 5/5 + 3 + 1
(1 + 3 + 5)/(5 + 3 + 1)
```

Check your results by entering the calculations into the command window.

Using Variables

2.2 Identify which name in each of the following pairs is a legitimate MATLAB variable name:

fred	fred!
book_1	book-1
2ndplace	Second_Place
#1	No_1
vel_5	vel.5
tan	while

Test your answers by using `isvarname`—for example,

```
isvarname fred
```

The function `isvarname` returns a 1 if the name is valid, and a 0 if it is not. Although it is possible to reassign a function name as a variable name, doing so is not a good idea. Use `which` to check whether the preceding names are function names—for example,

```
which sin
```

In what case would MATLAB tell you that sin is a variable name, not a function name?

Scalar Operations and Order of Operations

2.3 Create MATLAB code to perform the following calculations:

5^2

$\dfrac{5+3}{5 \cdot 6}$

$\sqrt{4+6^3}$ (*Hint:* A square root is the same thing as a 1/2 power.)

$9\dfrac{6}{12} + 7 \cdot 5^{3+2}$

$1 + 5 \cdot 3/6^2 + 2^{2-4} \cdot 1/5.5$

Check your code by entering it into MATLAB and performing the calculations on your scientific calculator.

2.4 **(a)** As shown in Figure P2.4(a), the area of a circle is πr^2. Define r as 5, then find the area of a circle, using MATLAB.

(b) As shown in Figure P2.4(b), the surface area of a sphere is $4\pi r^2$. Find the surface area of a sphere with a radius of 10 ft.

(c) As shown in Figure P2.4(c), the volume of a sphere is $4/3\pi r^3$. Find the volume of a sphere with a radius of 2 ft.

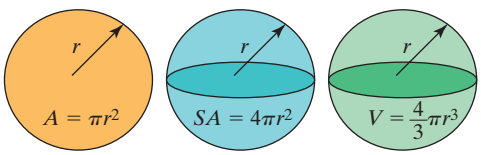


Figure 2.4
(a) Area of a circle, (b) Surface area of a sphere, (c) Volume of a sphere with radius r.

2.5 **(a)** As shown in Figure P2.5(a), the area of a square is the edge length squared ($A = edge^2$). Define the edge length as 5, then find the area of a square, using MATLAB.

(b) As shown in Figure P2.5(b), the surface area of a cube is 6 times the edge length squared (SA $= 6 \times edge^2$ Find the surface area of a cube with edge length 10.

(c) As shown in Figure P2.5(c), the volume of a cube is the edge length cubed ($V = edge^3$). Find the volume of a cube with edge length 12.

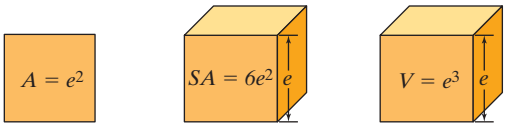


Figure P2.5
(a) Area of a square, (b) Surface area of a cube, (c) Volume of a cube with edge length e.



Figure P2.6
The geometry of a barbell can be modeled as two spheres and a cylindrical rod.

2.6 Consider the barbell shown in Figure P2.6.

(a) Find the volume of the figure, if the radius of each sphere is 10 cm, the length of the bar connecting them is 15 cm, and the diameter of the bar is 1 cm. Assume that the bar is a simple cylinder.

(b) Find the surface area of the figure.

2.7 The ideal gas law was introduced in Example 2.1. It describes the relationship between pressure (P), temperature (T), volume (V), and the number of moles of gas (n).

$$PV = nRT$$

The additional symbol, R, represents the ideal gas constant. The ideal gas law is a good approximation of the behavior of gases when the pressure is low and the temperature is high. (What constitutes low pressure and high temperature varies with different gases.) In 1873, Johannes Diderik van der Waals, (see Figure P2.7) proposed a modified version of the ideal gas law that better models the behavior of real gases over a wider range of temperature and pressure.

$$\left(P + \frac{n^2 a}{V^2}\right)(V - nb) = nRT$$

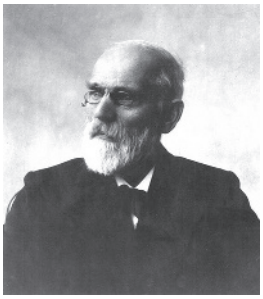


Figure P2.7
Johannes Diderik van der Waals. (Bygone Collection/Alamy Stock Photo)

In this equation, the additional variables a and b represent values characteristic of individual gases.

Use both the ideal gas law and van der Waals' equation to calculate the temperature of water vapor (steam), given the data in Table P2.7.

Table P2.7 Steam Data

Pressure, P	220 bar	
Moles, n	2 mol	
Volume, V	1 L	
a	5.536 L^2 bar/mol^2	*
b	0.03049 L/mol	*
Ideal gas constant, R	0.08314472 L bar/K mol	

*Source: Weast, R. C. (Ed.), *Handbook of Chemistry and Physics (53rd Edn.)*, Cleveland: Chemical Rubber Co., 1972.

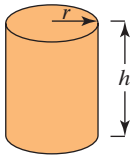


Figure P2.8(a)

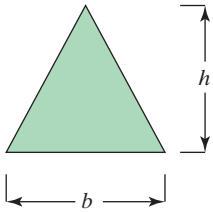


Figure P2.8(b)

Array Operations

2.8 **(a)** The volume of a cylinder is $\pi r^2 h$. Define r as 3 and h as the array

```
h = [1, 5, 12]
```

Find the volume of the cylinders (see Figure P2.8(a)).

(b) The area of a triangle is 1/2 the length of the base of the triangle, times the height of the triangle. Define the base as the array

```
b = [2, 4, 6]
```

and the height h as 12, and find the area of the triangles (see Figure P2.8(b)).

(c) The volume of any right prism is the area of the base of the prism, times the vertical dimension of the prism. The base of the prism can be any shape—for example, a circle, a rectangle, or a triangle.

Find the volume of the prisms created from the triangles of part (b). Assume that the vertical dimension of these prisms is 6 (see Figure P2.8(c)).

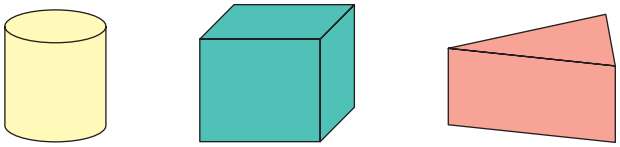


Figure P2.8(c)

2.9 The response of circuits containing resistors, inductors, and capacitors depends upon the relative values of the resistors and the way they are connected. An important intermediate quantity used in describing the response of such circuits is s. Depending on the values of R, L, and C, the values of s will be either both real values, a pair of complex values, or a duplicated value.

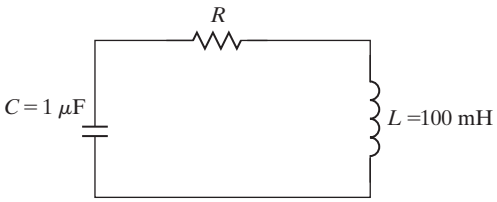


Figure P2.9
Series circuit.

The equation that identifies the response of a particular series circuit (Figure P2.9) is

$$s = -\frac{R}{2L} \pm \sqrt{\left(\frac{R}{2L}\right)^2 - \frac{1}{LC.}}$$

(a) Determine the values of s for a resistance of 800 Ω.

(b) Create a vector of values for R ranging from 100 Ω to 1000 Ω and evaluate s. Refine your values of R until you find the approximate size of resistor that yields a pure real value of s. Describe the effect on s as R increases in value.

Hint:
$1\ \mu\text{F} = 1\text{e} - 6\text{F}$
$1\ \text{mH} = 1\text{e} - 3\text{H}$

2.10 The equation that identifies the response parameter, s, of the parallel circuit shown in Figure P2.10 is

$$s = -\frac{1}{2RC} \pm \sqrt{\left(\frac{1}{2RC}\right)^2 - \frac{1}{LC}}.$$

(a) Determine the values of s for a resistance of 200 Ω.
(b) Create a vector of values for R ranging from 100 Ω to 1000 Ω and evaluate s. Refine your values of R until you find the size of resistor that yields a pure real value of s. Describe the effect on s as R decreases.

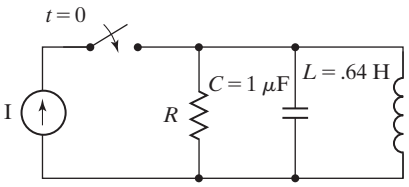


Figure P2.10
Parallel circuit.

2.11 Burning one gallon of gasoline in your car produces 19.4 pounds of CO_2. Calculate the amount of CO_2 emitted during a year for the following vehicles, assuming they all travel 12,000 miles per year. The reported fuel-efficiency numbers shown in Table P2.11 were extracted from the U.S. Department of Energy website, www.fueleconomy.gov, and reflect the combined city and highway estimates.

Table P2.11 Fuel Efficiency

2021	Tesla Model 3 EV	142 mpg
2021	Honda Civic	36 mpg
2021	Toyota Corolla Hybrid	52 mpg
2021	Toyota Highlander Hybrid	35 mpg
2021	Ford Escape FWD Hybrid	41 mpg
2021	BMW i3 EV	113 mpg

2.12 **(a)** Create an evenly spaced vector of values from 1 to 20 in increments of 1.
(b) Create a vector of values from 0 to 2π in increments of $\pi/10$.
(c) Create a vector containing 15 values, evenly spaced between 4 and 20. (*Hint:* Use the `linspace` command. If you can't remember the syntax, type `help linspace`.)
(d) Create a vector containing 10 values, spaced logarithmically between 10 and 1000. (*Hint:* Use the `logspace` command.)

2.13 **(a)** Create a table of conversions from feet to meters. Start the feet column at 0, increment it by 1, and end it at 10 feet. (Look up the conversion factor in a textbook or online.)

(b) Create a table of conversions from radians to degrees. Start the radians column at 0 and increment by 0.1π radian, up to π radians. (Look up the conversion factor in a textbook or online.)

(c) Create a table of conversions from mi/h to ft/s. Start the mi/h column at 0 and end it at 100 mi/h. Print 15 values in your table. (Look up the conversion factor in a textbook or online.)

(d) The acidity of solutions is generally measured in terms of pH. The pH of a solution is defined as $-\log_{10}$ of the concentration of hydronium ions. Create a table of conversions from concentration of hydronium ion to pH, spaced logarithmically from 0.001 mol/liter to 0.1 mol/liter with 10 values. Assuming that you have named the concentration of hydronium ions `H_conc`, the syntax for calculating the negative of the logarithm of the concentration (and thus the pH) is

```
pH = -log10(H_conc)
```

2.14 The general equation for the distance that a freely falling body has traveled (neglecting air friction) is

$$d = \frac{1}{2} gt^2$$

Assume that $g = 9.8$ m/s^2. Generate a table of time versus distance traveled for values of time from 0 second to 100 seconds. Choose a suitable increment for your time vector. (*Hint:* Be careful to use the correct operators; t^2 is an array operation!)

2.15 In direct current applications, electrical power is calculated using Joule's law as

$$P = VI$$

where

P is power in watts,
V is the potential difference, measured in volts,
I is the electrical current, measured in amperes, and
Joule's law can be combined with Ohm's law

$$V = IR$$

to give

$$P = I^2R$$

where R is resistance measured in ohms.

The resistance of a conductor of uniform cross section (a wire or rod for example) is

$$R = \rho\frac{l}{A}$$

where

ρ is the electrical resistivity measured in ohm-meters,
l is the length of the wire, and
A is the cross-sectional area of the wire.
This results in the equation for power

$$P = I^2\rho\frac{l}{A}$$

Electrical resistivity is a material property that has been tabulated for many materials. For example,

Material	Resistivity, ohm-meters (measured at 20°C)
Silver	1.59×10^{-8}
Copper	1.68×10^{-8}
Gold	2.44×10^{-8}
Aluminum	2.82×10^{-8}
Iron	1.0×10^{-7}

Calculate the power that is dissipated through a wire with the following dimensions for each of the materials listed.

diameter	0.001 m
length	2.00 m

Assume the wire carries a current of 120 amps.

2.16 Repeat the previous problem for 10 wire lengths, from 1 m to 1 km. Use logarithmic spacing.

2.17 Newton's law of universal gravitation tells us that the force exerted by one particle on another is

$$F = G\frac{m_1 m_2}{r^2}$$

where the universal gravitational constant G is found experimentally to be

$$G = 6.673 \times 10^{-11} \text{ N m}^2/\text{kg}^2$$

The mass of each particle is m_1 and m_2, respectively, and r is the distance between the two particles. Use Newton's law of universal gravitation to find the force exerted by the earth on the moon, assuming that

the mass of the earth is approximately 6×10^{24} kg,
the mass of the moon is approximately 7.4×10^{22} kg, and
the earth and the moon are an average of 3.9×10^{8} m apart.

2.18 We know that the earth and the moon are not always the same distance apart. Based on the equation in the previous problem, find the force the moon exerts on the earth for 10 distances between 3.8×10^{8} m and 4.0×10^{8} m. Be careful to use the correct operator when you do the division.

2.19 Recall from Problem 2.7 that the ideal gas law is

$$PV = nRT$$

and that the Van der Waals modification of the ideal gas law is

$$\left(P + \frac{n^2 a}{V^2}\right)(V - nb) = nRT.$$

Using the data from Problem 2.7, find the value of temperature (T), for
(a) Ten values of pressure from 0 bar to 400 bar for volume of 1 L.
(b) Ten values of volume from 0.1 L to 10 L for a pressure of 220 bar.

Number Display

2.20 Create an array `a` equal to [−1/3, 0, 1/3, 2/3], and use each of the built-in format options to display the results:

```
format short (which is the default)
format long
```

```
format bank
format short e
format long e
format short eng
format long eng
format short g
format long g
format +
format rat
```

Saving Your Work

2.21
- Create an array called `D_to_R` composed of two columns, one representing degrees, and the other representing the corresponding value in radians. Any value set will do for this exercise.
- Save the array to a file called degrees.dat.
- Once the file is saved, clear your workspace then load the data from the file back into MATLAB.

2.22 Create a script M-file and use it to do the homework problems you've been assigned from this chapter. Your file should include appropriate comments to identify each problem and to describe your calculation process. Don't forget to include your name, the date, and any other information your instructor requests. Divide the script into convenient sections, using section dividers.

2.23 Convert the script M-file from the previous problem into a live script and run the program. Experiment with displaying the results both in-line and in the right-hand column. Save your results as a PDF.

CHAPTER

3 Built-In MATLAB Functions

Objectives

After reading this chapter, you should be able to:

- Access and use MATLAB's Help Feature
- Use a variety of common mathematical functions built into MATLAB.
- Use trigonometric functions in MATLAB to solve problems.
- Compute and use statistical and data analysis functions.
- Generate uniform and Gaussian random-number arrays.
- Create and use complex numbers.
- Describe the computational limits of MATLAB.
- Recognize and use the special values and functions built into MATLAB.

INTRODUCTION

The vast majority of engineering computations require quite complicated mathematical **functions**, including logarithms, trigonometric functions, and statistical analysis functions. MATLAB has an extensive library of built-in functions to allow you to perform these calculations easily.

3.1 USING BUILT-IN FUNCTIONS

KEY IDEA
Most of the MATLAB function names are the same as those used in other computer programs.

Many of the names for MATLAB's built-in functions are the same as those defined not only in the C programming language, but in Fortran, Python, and Java as well. For example, to take the square root of the variable `x`, we type

```
b = sqrt(x)
```

A big advantage of MATLAB is that function arguments can generally be either scalars or arrays. In our example, if x is a scalar, a scalar result is returned. Thus, the statement

```
x = 9;
b = sqrt(x)
```

returns a scalar:

```
b =
        3
```

However, the square-root function, sqrt, can also accept arrays as input. In this case, the square root of each element is calculated, so

```
x = [4, 9, 16];
b = sqrt(x)
```

returns

```
b =
      2   3   4
```

All functions can be thought of as having three components: a name, an input, and an output. In the preceding example, the name of the function is sqrt, the required **function input** (also called the **argument**) goes inside the parentheses and can be a scalar or an array, and the output is a calculated value or values. In this example, the output was assigned the variable name b.

KEY IDEA

The input to a function is also called the argument or the input argument.

Some functions require multiple inputs. For example, the remainder function, rem, requires two inputs: a dividend and a divisor. We represent this as rem(x,y), so

```
rem(10,3)
```

calculates the remainder of 10 divided by 3:

```
ans =
       1
```

The size function is an example of a function that returns two outputs, which are stored in a single array. It determines the number of rows and columns in an array. Thus,

```
d = [1, 2, 3; 4, 5, 6];
f = size(d)
```

returns the 1×2 result array.

```
f =
    2      3
```

You can also assign variable names to each of the answers by representing the left-hand side of the assignment statement as an array. For example,

```
[rows,cols] = size(d)
```

gives

```
rows =
       2
cols =
       3
```

A useful feature of the more recent versions of MATLAB is the adaptive help capability. As you type a function name, a screen tip appears showing the correct function format. It also includes a link to the function's help page.

You can create more complicated expressions by **nesting** functions. For instance,

```
g = sqrt(sin(x))
```

KEY IDEA
Computer code may be easier to read if nested code is broken into two separate statements.

finds the square root of the sine of whatever values are stored in the array named `x`. If `x` is assigned a value of 2,

```
x = 2;
```

the result is

```
g =
    0.9536
```

Nesting functions can result in some complicated MATLAB code. Be sure to include the arguments for each function inside their own set of parentheses. Often, your code will be easier to read if you break nested expressions into two separate statements. Thus,

```
a = sin(x);
g = sqrt(a)
```

gives the same result as `g = sqrt(sin(x))` and is easier to follow.

HINT

You can probably *guess* the name and syntax for many MATLAB functions. However, check to make sure that the function of interest is working the way you assume it is, before you do any important calculations.

3.2 USING THE HELP FEATURE

KEY IDEA
Use the help function to help you use MATLAB's built-in functions.

MATLAB includes extensive help tools, which are especially useful in understanding how to use functions. There are two ways to get help from within MATLAB: a command-line help function `help`, which is fairly limited; and an HTML-based set of documentation accessed with the `doc` function. You can also open the HTML-based documentation by selecting *Help* from the toolstrip or by using the *F1* function key. You should use both help options, since they provide different information and insights into how to use a specific function.

To get help on a particular topic, type `help <topic>`. (Angle brackets, < and >, identify where you should type your input; they are not included in your actual MATLAB statement.)

For example, to get help on the `tangent` function, type

```
help tan
```

The following should be displayed:

```
tan           Tangent of argument in radians.
  tan(X) is the tangent of the elements of X.
See also atan, atan2, tand, atan2d.
```

KEY IDEA
Both the help and doc functions offer suggestions on how to use MATLAB's built-in functions.

To use the windowed help screen, select *Help* → *Documentation* from the toolstrip. A windowed version of the help list will appear, which includes access to documentation for all of the MATLAB products you have installed on your computer. You can then navigate to the appropriate topic. To access this version of the help utility directly from the command window, type `doc <topic>`. Thus, to access the windowed help for tangent, type

```
doc tan
```

The contents of the two methods for getting help on a function are different. If your question isn't immediately answered by whichever method you try first, it's often useful to try the other technique.

You will find browsing through the windowed help utility extremely useful. First, select MATLAB from the Help opening window, which opens the interactive MATLAB documentation. On the left is the table of contents, which can either be expanded or minimized as needed. The main screen includes links to a list of topics, organized by category. You can use these links to find out what MATLAB functions are available to solve many problems.

PRACTICE EXERCISE 3.1

1. Use the `help` command in the command window to find the appropriate syntax for the following functions:
   a. `cos`
   b. `sqrt`
   c. `exp`
2. Use the windowed help function from the menu bar to learn about the functions in Exercise 1.
3. Access the windowed help information for the functions in Exercise 1 using the `doc` function from the command window.

KEY IDEA
Most functions accept scalars, vectors, or two-dimensional arrays as input.

3.3 ELEMENTARY MATH FUNCTIONS

Elementary math functions include logarithms, exponentials, absolute value, rounding functions, and functions used in **discrete mathematics**.

3.3.1 Common Computations

The functions listed in Table 3.1 accept either a scalar or an array of `x` values.

HINT

As a rule, the function `log` in all computer languages means the `natural logarithm`. Although not the standard in mathematics textbooks, it *is* the standard in computer programming. Not knowing this distinction is a common source of errors, especially for new users. If you want logarithms to the base 10, you'll need to use the `log10` function. A `log2` function is also included in MATLAB, but logarithms to any other base will need to be computed; there is no general logarithm function that allows the user to input the base.

PRACTICE EXERCISE 3.2

Consult Table 3.1 for the appropriate function to solve the following problems. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Create a vector x from −2 to +2 with an increment of 1. Your vector should be

```
x = [-2, -1, 0, 1, 2]
```

   a. Find the absolute value of each member of the vector.
   b. Find the square root of each member of the vector.
2. Find the square root of both −3 and +3
   a. Use the `sqrt` function.
   b. Use the `nthroot` function. (You should get an error statement for −3)
   c. Raise −3 and +3 to the ½ power (Don't forget parentheses around −3). How do the results vary?
3. Create a vector x from −9 to 12 with an increment of 3.
   a. Find the result of x divided by 2.
   b. Find the remainder of x divided by 2.
4. Using the vector from Exercise 3, find $\mathbf{e}^{x}$.
5. Using the vector from Exercise 3:
   a. Find $\ln(x)$ (the natural logarithm of x).
   b. Find $\log_{10}(x)$ (the common logarithm of x). Explain your results.
6. Use the `sign` function to determine which of the elements in vector x are positive.
7. Change the `format` to `rat`, and display the value of the x vector divided by 2.

   (Don't forget to change the format back to `format short` or `format shortg` when you are done with this exercise set.)

HINT

The mathematical notation and MATLAB syntax for raising e to a power are not the same. To raise e to the third power, the mathematical notation would be e^3. However, the MATLAB syntax is `exp(3)`. Students also sometimes confuse the syntax for scientific notation with exponentials. The number `5e3` should be interpreted as 5×10^3.

Table 3.1 Common Math Functions

`abs(x)`	Finds the absolute value of *x*.	`abs(-3)` `ans = 3`
`sqrt(x)`	Finds the square root of *x*.	`sqrt(85)` `ans = 9.2195`
`nthroot(x,n)`	Finds the real *n*th root of *x*. This function will not return complex results. Thus, `(-2)^(1 / 3)`	`nthroot(-2,3)` `ans = -1.2599`

(*continued*)

Table 3.1 (continued)

	does not return the same result, yet both answers are legitimate third roots of −2.	`(-2)^(1/3)` `ans = 0.6300 - 1.0911 i`
`sign(x)`	Returns a value of −1 if *x* is less than zero, a value of 0 if *x* equals zero, and a value of +1 if *x* is greater than zero.	`sign(-8)` `ans = -1`
`rem(x,y)`	Computes the remainder of *x*/*y*.	`rem(25,4)` `ans = 1`
`exp(x)`	Computes the value of e^x, where e is the base for natural logarithms, approximately 2.7183.	`exp(10)` `ans = 2.2026e + 004`
`log(x)`	Computes ln(*x*), the natural logarithm of *x* (to the base e).	`log(10)` `ans = 2.3026`
`log10(x)`	Computes $\log_{10}(x)$ the common logarithm of *x* (to the base 10).	`log10(10)` `ans = 1`

EXAMPLE 3.1

USING THE CLAUSIUS–CLAPEYRON EQUATION

Meteorologists study the atmosphere in an attempt to understand and ultimately predict the weather (see Figure 3.1). Weather prediction is a complicated process, even with the best data. Meteorologists study chemistry, physics, thermodynamics, and geography, in addition to specialized courses about the atmosphere.

One equation used by meteorologists is the Clausius–Clapeyron equation, which is usually introduced in chemistry classes and examined in more detail in advanced thermodynamics classes. Rudolf Clausius and Emile Clapeyron were physicists responsible for the early development of thermodynamic principles during the mid-1800s (see Figure 3.2(a) and 3.2(b)).

In meteorology, the Clausius–Clapeyron equation is employed to determine the relationship between saturation water-vapor pressure and the atmospheric temperature. The saturation water-vapor pressure can be used to calculate relative humidity, an important component of weather prediction, when the actual partial pressure of water in the air is known.

The Clausius–Clapeyron equation is

$$\ln\left(\frac{P^0}{6.11}\right) = \left(\frac{\Delta H_v}{R_{\text{air}}}\right) \times \left(\frac{1}{273} - \frac{1}{T}\right)$$

where

P^0 = saturation water-vapor pressure, in mbar, at temperature T,
ΔH_v = latent heat of vaporization for water, 2.453×10^6 J/kg,
R_{air} = gas constant for moist air, 461 J/kg, and
T = temperature in kelvins (K).

(*continued*)

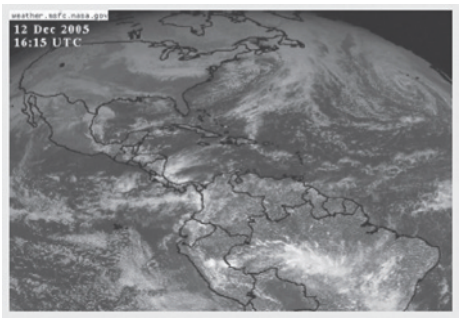


Figure 3.1
View of the earth's weather from space. (NASA Headquarters)

(a)



Figure 3.2(a)
Rudolf Clausius. (Bilwissedition Ltd. & Co. KG/Alamy Stock Photo)

(b)

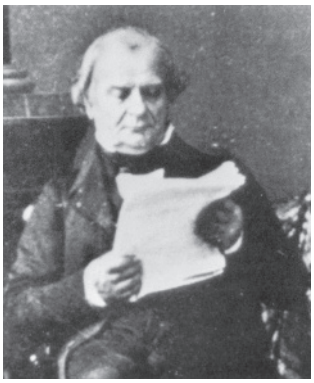


Figure 3.2(b)
Emile Clapeyron. (SPL/ Science Source)

It is rare that temperatures on the surface of the earth are lower than −60°F or higher than 120°F. Use the Clausius–Clapeyron equation to find the saturation vapor pressure for temperatures in this range. Present your results as an array of Fahrenheit temperatures and saturation vapor pressures.

1. State the problem
   Find the saturation vapor pressure at temperatures from −60°F to 120°F, using the Clausius–Clapeyron equation.

2. Describe the input and output

   Input

   $$\begin{aligned} \Delta H_v &= 2.453 \times 10^6 \text{ J/g} \\ R_{\text{air}} &= 461 \text{ J/kg} \\ T &= -60°\text{F to } 120°\text{F} \end{aligned}$$

   As the number of temperature values was not specified, we'll choose to recalculate every 10°F.

   Output

   Saturation vapor pressures.

3. Develop a hand example
   The Clausius–Clapeyron equation requires that all the variables have consistent units. This means that temperature (T) needs to be in kelvins. To change degrees Fahrenheit to kelvins, we use the conversion equation

   $$T_k = \frac{(T_f + 459.6)}{1.8}$$

(There are many places to find unit conversions. The Internet is one source, as are science and engineering textbooks.)
Now solve the Clausius–Clapeyron equation for the saturation vapor pressure P^0.

$$\ln\left(\frac{P^0}{6.11}\right) = \left(\frac{\Delta H_v}{R_{\text{air}}}\right) \times \left(\frac{1}{273} - \frac{1}{T}\right)$$

$$P^0 = 6.11 \times e\left(\left(\frac{\Delta H_v}{R_{\text{air}}}\right) \times \left(\frac{1}{273} - \frac{1}{T}\right)\right)$$

Next, solve for one temperature—for example, $T = 0°\text{F}$. As the equation requires temperature in kelvins, perform the unit conversion to obtain

$$T = \frac{(0 + 459.6)}{1.8} = 255.3333 \text{ K}$$

Finally, substitute values resulting in:

$$P^0 = 6.11 \times e\left(\left(\frac{2.453 \times 10^6}{461}\right) \times \left(\frac{1}{273} - \frac{1}{255.3333}\right)\right) = 1.5836 \text{ mbar}$$

4. Develop a MATLAB solution
Create a MATLAB program in either a script or a live script. Whichever you choose, the code is the same. In this case, we chose to use a script M-file.

```
%% Example 3.1
% Using the Clausius-Clapeyron Equation, find the
% saturation vapor pressure for water at different
% temperatures
%
clear, clc
TempF = [-60:10:120];          % Define temp array in F
TempK = (TempF + 459.6)/1.8;   % Convert temp to K
Delta_H = 2.45e6;              % Define latent heat of
                               % vaporization
R_air = 461;                   % Define ideal gas constant for air
%% Calculate the Vapor Pressures
Vapor_Pressure = 6.11*exp((Delta_H/R_air)*(1/273 - 1./TempK));
%% Display the results in an array
my_results = [TempF',Vapor_Pressure']
```

When you create a MATLAB program, it is a good idea to comment liberally (lines beginning with %). This makes your program easier for others to understand, and may make it easier for you to "debug." Notice that most of the lines of code end with a semicolon, which suppresses the output. Therefore, the only information that displays in the command window is the array `my_results`:

```
my_results =
   -60.0000    0.0698
   -50.0000    0.1252
   -40.0000    0.2184
   ...
   120.0000    118.1931
```

5. Test the solution
Compare the MATLAB solution when $T = 0°\text{F}$ with the hand solution:
Hand solution: $P^0 = 1.5888$ mbar

(*continued*)

MATLAB solution: $P^0 = 1.5888$ mbar

The Clausius–Clapeyron equation can be used for more than just humidity problems. By changing the values of ΔH and R, you could generalize the program to deal with any condensing vapor.

Instead of creating the program in a script M-file, we could have used a live script. Since the script already exists, it is easy to convert to a live script by right-clicking the file name of the script in the editing window and selecting "Open as a Live Script." The result is shown in Figure 3.3.

To run the MATLAB computer program as live script, select the run icon. The results will be displayed in the live script, either inline or in the right-hand column depending on the user selection of the output placement icons.

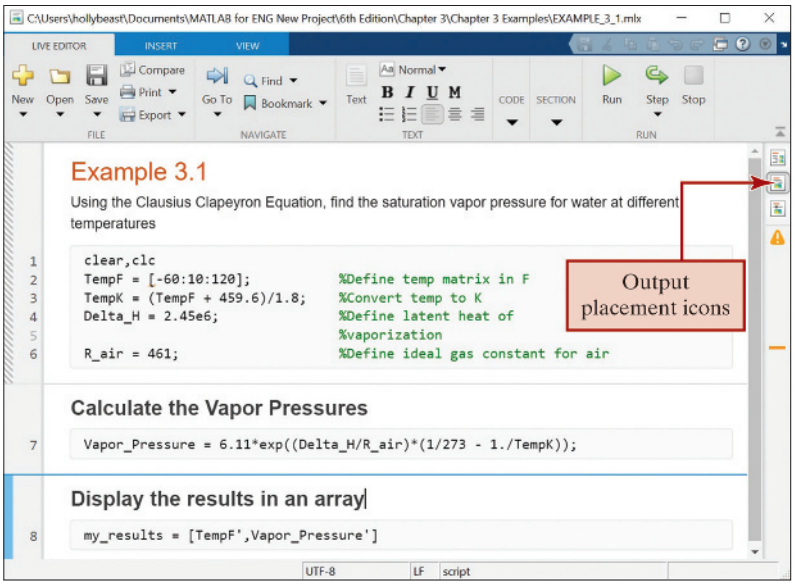


Figure 3.3
Live Scripts can be created from script M-files and contain both computer code and formatted text.

3.3.2 Rounding Functions

MATLAB contains functions for a number of different rounding techniques (see Table 3.2). You are probably most familiar with rounding to the closest integer; however, you may want to round either up or down, depending on the situation.

For example, suppose you want to buy apples at the grocery store. The apples cost \$0.52 apiece. You have \$5.00. How many apples can you buy? Mathematically,

$$\frac{\$5.00}{\$0.52/\text{apple}} = 9.6154 \text{ apples.}$$

But clearly, you can't buy part of an apple, and the grocery store won't let you round to the nearest number of apples. Instead, you need to round down. The MATLAB function to accomplish this is `fix`. Thus,

```
fix(5/0.52)
```

returns the maximum number of apples you can buy:

```
ans =
     9
```

Table 3.2 Rounding Functions

`round(x)`	Rounds `x` to the nearest integer.	`round(8.6)` `ans = 9`
`round(x,N)`	Rounds `x` to a specified decimal digit	`round(8.6436, 3)` `ans = 8.644`

`fix(x)`	Rounds (or truncates) `x` to the nearest integer toward zero. Notice that 8.6 truncates to 8, not 9, with this function.	`fix(8.6)`
		`ans = 8`
		`fix(-8.6)`
		`ans = -8`

`floor(x)`	Rounds `x` to the nearest integer toward negative infinity.	`floor(-8.6)` `ans = -9`
`ceil(x)`	Rounds `x` to the nearest integer toward positive infinity.	`ceil(-8.6)` `ans = -8`

3.3.3 Discrete Mathematics

Discrete mathematics is the mathematics of whole numbers. MATLAB includes functions to factor numbers, find common denominators and multiples, calculate factorials, and explore prime numbers (see Table 3.3). All of these functions require integer scalars as input. In addition, MATLAB includes the `rats` function, which expresses a floating-point number as a **rational number**—that is, a fraction.

A factorial is the product of all the positive integers from 1 to a given value. Thus 3 factorial (indicated in mathematics textbooks as 3!) is $3 \times 2 \times 1 = 6$. Many problems involving probability can be solved with factorials. For example, the number of ways that five cards can be arranged is $5 \times 4 \times 3 \times 2 \times 1 = 5! = 120$. When you select the first card, you have five choices; when you select the second card, you have only four choices remaining, then three, two, and one. This approach is called

Table 3.3 Functions Used in Discrete Mathematics

`factor(x)`	Finds the prime factors of `x`.	`factor(12)` `ans = 2 2 3`
`gcd(x,y)`	Finds the greatest common denominator of `x` and `y`.	`gcd(10,15)` `ans = 5`

`lcm(x,y)`	Finds the least common multiple of `x` and `y`.	`lcm(2,5)`
		`ans = 10`
		`lcm(2,10)`
		`ans = 10`

`rats(x)`	Represents `x` as a fraction.	`rats(1.5)` `ans = 3/2`
`factorial(x)`	Finds the value of `x` factorial (`x`!). A factorial is the product of all the integers less than `x`. For example, $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$	`factorial(6)` `ans = 720`
`nchoosek(n,k)`	Finds the number of possible combinations of *k* items from a group of *n* items. For example, use this function to determine the number of possible subgroups of 3 chosen from a group of 10.	`nchoosek(10,3)` `ans = 120`
`primes(x)`	Finds all the prime numbers less than `x`.	`primes(10)` `ans = 2 3 5 7`

`isprime(x)`	Checks to see if `x` is a prime number. If it is, the function returns 1; if not, it returns 0.	`isprime(7)`
		`ans = 1`
		`isprime(10)`
		`ans = 0`

combinatorial mathematics, or combinatorics. To calculate a factorial in MATLAB use the factorial function. Thus,

```
factorial(5)
ans =
      120
```

gives the same result as

```
5*4*3*2*1
ans =
      120
```

The value of a factorial quickly becomes very large. 10 factorial is 3,628,800. MATLAB can handle up to 170! Anything larger gives `Inf` for an answer, because the maximum value for a **real number** that can be expressed in MATLAB is exceeded.

```
factorial(170)
          ans =
          7.2574e+306
factorial(171)
          ans =
                Inf
```

Factorials are used to calculate the number of permutations and combinations of possible outcomes. A permutation is the number of subgroups that can be formed when sampling from a larger group, *when the order matters*. Consider the following problem. How many different teams of two people can you form from a group of four? Assume that the order matters, since for this problem the first person chosen is the group leader. If we represent each person as a letter, the possibilities are as follows:

AB	BA	CA	DA
AC	BC	CB	DB
AD	BD	CD	DC

For the first member of the team, there are four choices, and for the second, there are three choices, so the number of possible teams is $4 \times 3 = 12$. We could also express this as 4!/2!. More generally, if you have a large group to choose from, call the group size n, and the size of the subgroup (team) m. Then the possible number of permutations is

$$\frac{n!}{(n-m)!}$$

If there are 100 people to choose from, the number of teams of 2 (where order matters) is

$$\frac{100!}{(100-2)!} = 9900$$

But, what if the order doesn't matter? In this case, team AB is the same as team BA, and we refer to all the possibilities as combinations instead of permutations. The possible number of combinations is

$$\frac{n!}{(n-m)! \times m!}$$

Although you could use MATLAB's factorial function to calculate the number of combinations, the `nchoosek` function will do it for you, and it offers some advantages

when using larger numbers. If we want to know the number of possible teams of 2, chosen from a pool of 100 (100 choose 2),

```
nchoosek(100,2)
ans =
       4950
```

The `nchoosek` function allows us to calculate the number of combinations even if the pool size is greater than 170, which would not be possible using the factorial approach.

```
nchoosek(200,2)
ans =
    19900
         factorial(200)/(factorial(198)*factorial(2))
         ans =
              NaN
```

HINT

Although the exclamation point (!) is used in mathematics to indicate taking a factorial, it is not a MATLAB operator. Use the `factorial` function instead.

PRACTICE EXERCISE 3.3

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Factor the number 322.
2. Find the greatest common denominator of 322 and 6.
3. Is 322 a prime number?
4. How many primes occur between 0 and 322?
5. Approximate π as a rational number.
6. Find 10! (10 factorial).
7. Find the number of possible groups containing 3 people from a group of 20, when order does not matter (20 choose 3).

3.4 TRIGONOMETRIC FUNCTIONS

MATLAB includes a complete set of the standard trigonometric functions and the hyperbolic trigonometric functions. As is standard in most computer languages and in common mathematics usage, most of these functions assume that angles are expressed in radians. To convert radians to degrees or degrees to radians, we need to take advantage of the fact that π radians equals 180 degrees:

KEY IDEA
Most trigonometric functions require input in radians.

$$\text{degrees} = \text{radians}\left(\frac{180}{\pi}\right) \text{ and radians} = \text{degrees}\left(\frac{\pi}{180}\right)$$

The MATLAB code to perform these conversions is

```
degrees = radians * 180/pi;
radians = degrees * pi/180;
```

MATLAB also includes functions to convert from degrees to radians (`deg2rad`) or radians to degrees (`rad2deg`). For example, to convert pi radians to degrees, use

```
rad2deg(pi)
ans =
      180
```

Similarly to convert 90 degrees to radians use

```
deg2rad(90)
ans =
       1.5708
```

To carry out these calculations, we need the value of π so a constant, `pi`, is built into MATLAB. However, since π cannot be expressed exactly as a floating-point number, the constant `pi` in MATLAB is only an approximation of the mathematical quantity π Usually this is not important; however, you may notice some surprising results. For example, for

```
sin(pi)
ans =
       1.2246e-016
```

when you expect an answer of zero.

MATLAB also includes a set of trigonometric functions that accept the angle in degrees so that you need not do the conversion to radians. These include `sind`, `cosd`, and `tand`.

You may access the help function from the toolstrip for a complete list of trigonometric functions available in MATLAB. Table 3.4 shows some of the more common ones.

HINT

Math texts often use the notation $\sin^{-1}(x)$ to indicate an inverse sine function, also called an arcsine. Students are often confused by this notation and try to create parallel MATLAB code. Note, however, that

```
a = sin^-1(x)
```

is *not* a valid MATLAB statement but instead should be

```
a = asin(x)
```

Table 3.4 Some of the Available Trigonometric Functions

`deg2rad`	Converts degrees to radians.	`deg2rad(90)` `ans = 1.5708`
`rad2deg`	Converts radians to degrees.	`rad2deg(pi)` `ans = 180`
`sin(x)`	Finds the sine of `x` when `x` is expressed in radians.	`sin(0)` `ans = 0`
`cos(x)`	Finds the cosine of `x` when `x` is expressed in radians.	`cos(pi)` `ans = -1`
`tan(x)`	Finds the tangent of `x` when `x` is expressed in radians.	`tan(pi)` `ans = -1.2246e-016`

(*continued*)

Table 3.4 (continued)

`asin(x)`	Finds the arcsine, or inverse sine, of `x`, where `x` must be between −1 and 1. The function returns an angle in radians between $\pi/2$ and $-\pi/2$.	`asin(-1)` `ans = -1.5708`
`sinh(x)`	Finds the hyperbolic sine of `x` when `x` is expressed in radians.	`sinh(pi)` `ans = 11.5487`
`asinh(x)`	Finds the inverse hyperbolic sin of `x`.	`asinh(1)` `ans = 0.8814`
`sind(x)`	Finds the sin of `x` when `x` is expressed in degrees.	`sind(90)` `ans = 1`
`asind(x)`	Finds the inverse sin of `x` and reports the result in degrees.	`asind(1)` `ans = 90`

PRACTICE EXERCISE 3.4

Calculate the following (remember that mathematical notation is not necessarily the same as MATLAB notation). The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. $\sin(2\theta)$ for $\theta = 3\pi$.
2. $\cos(\theta)$ for $0 \leq \theta \leq 2\pi$; let θ change in steps of 0.2π.
3. $\sin^{-1}(1)$.
4. $\cos^{-1}(x)$ for $-1 \leq x \leq 1$; let x change in steps of 0.2.
5. Find the cosine of 45°.
   a. Convert the angle from degrees to radians, and then use the `cos` function.
   b. Use the `cosd` function.
6. Find the angle whose sine is 0.5. Is your answer in degrees or radians?
7. Find the cosecant of 60. You may have to use the help function to find the appropriate syntax.

EXAMPLE 3.2

USING TRIGONOMETRIC FUNCTIONS

A basic calculation in engineering is finding the resulting force on an object that is being pushed or pulled in multiple directions. Adding up forces is the primary calculation performed in both statics and dynamics classes. Consider a balloon that is acted upon by the forces shown in Figure 3.4.

To find the net force acting on the balloon, we need to add up the force due to gravity, the force due to buoyancy, and the force due to the wind. One approach is to find the force in the x direction and the force in the y direction for each individual force, and then to recombine them into a final result.

The forces in the x and y directions can be found by trigonometry:

F = total force

(*continued*)

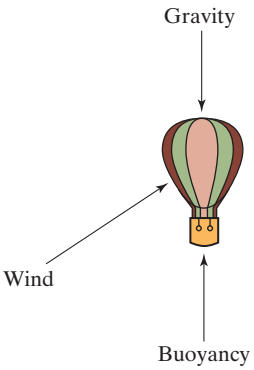


Figure 3.4
Force balance on a balloon.

F_x = force in the x direction
F_y = force in the y direction

We know from trigonometry that the sine is the opposite side over the hypotenuse, so

$$\sin(\theta) = \frac{F_y}{F}$$

and therefore,

$$F_y = F\sin(\theta).$$

Similarly, since the cosine is the adjacent side over the hypotenuse,

$$F_x = F\cos(\theta).$$

We can add up all the forces in the x direction and all the forces in the y direction, and use these totals to find the resulting force:

$$F_{x\ \text{total}} = \sum F_{xi} \qquad F_{y\ \text{total}} = \sum F_{yi}$$

To find the magnitude and angle for F_{total}, we use trigonometry again. The tangent is the opposite side over the adjacent side. Therefore,

$$\tan(\theta) = \frac{F_{y\ \text{total}}}{F_{x\ \text{total}}}$$

We use an inverse tangent to write

$$\theta = \tan^{-1}\left(\frac{F_{y\ \text{total}}}{F_{x\ \text{total}}}\right)$$

(The inverse tangent is also called the *arctangent,* which you may see on your scientific calculator as atan.)

Once we know θ, we can find F_{total}, using either the sine or the cosine. We have

$$F_{x\ \text{total}} = F_{\text{total}}\ \cos(\theta)$$

Rearranging terms gives

$$F_{\text{total}} = \frac{F_{x\ \text{total}}}{\cos(\theta)}$$

Now consider again the balloon shown in Figure 3.4. Assume that the force due to gravity on this particular balloon is 100 N, pointed downward. Assume further that the buoyant force is 200 N, pointed upward. Finally, assume that the wind is pushing on the balloon with a force of 50 N, at an angle of 30° from the horizontal.

Find the resulting force on the balloon.

1. State the problem
   Find the resulting force on a balloon. Consider the forces due to gravity, buoyancy, and the wind.

2. Describe the input and output
   Input

Force	Magnitude	Direction
Gravity	100 N	−90
Buoyancy	200 N	+90
Wind	50 N	+30

Output

We'll need to find both the magnitude and the direction of the resulting force.

3. Develop a hand example

First, find the x and y components of each force and sum the components:

Force	Horizontal Component	Vertical Component
Gravity	$F_x = F\cos(\theta)$	$F_y = F\sin(\theta)$
	$F_x = 100\cos(-90°) = 0$ N	$F_y = 100\sin(-90°) = -100$ N
Buoyancy	$F_x = F\cos(\theta)$	$F_y = F\sin(\theta)$
	$F_x = 200\cos(+90°) = 0$ N	$F_y = 200\sin(+90°) = +200$ N
Wind	$F_x = F\cos(\theta)$	$F_y = F\sin(\theta)$
	$F_x = 50\cos(+30°) = 43.301$ N	$F_y = 50\sin(+30°) = +25$ N
Sum	$F_{x\ \text{total}} = 0 + 0 + 43.301$ $= 43.301$ N	$F_{y\ \text{total}} = -100 + 200 + 25$ $= 125$ N

Find the resulting angle:

$$\theta = \tan^{-1}\left(\frac{F_{y\ \text{total}}}{F_{x\ \text{total}}}\right)$$

$$\theta = \tan^{-1}\frac{125}{43.301} = 70.89^{\circ}$$

Find the magnitude of the total force:

$$F_{\text{total}} = \frac{F_{x\ \text{total}}}{\cos(\theta)}$$

$$F_{\text{total}} = \frac{43.301}{\cos(70.89°)} = 132.29\ \text{N}$$

4. Develop a MATLAB solution program, either in a script or a live script.

One solution is

```
%% Example 3_2
clear, clc
% Define the input
Force = [100, 200, 50];
theta = [-90, +90, +30];
% convert angles to radians
theta = theta*pi/180;
% Find the x components
ForceX = Force.*cos(theta);
% Sum the x components
ForceX_total = sum(ForceX);
% Find and sum the y components in the same step
ForceY_total = sum(Force.*sin(theta));
% Find the resulting angle in radians
result_angle = atan(ForceY_total/ForceX_total);
% Find the resulting angle in degrees
result_degrees = result_angle*180/pi
% Find the magnitude of the resulting force
Force_total = ForceX_total/cos(result_angle)
```

(*continued*)

which returns

```
result_degrees = 70.8934
Force_total = 132.2876
```

Notice that the values for the force and the angle were entered into an array. This makes the solution more general. Notice also that the angles were converted to radians. In the program listing, the output from all but the final calculations was suppressed. However, while developing the program, we left off the semicolons so that we could observe the intermediate results.

5. Test the solution
Compare the MATLAB solution with the hand solution. Now that you know it works, you can use the program to find the resultant of multiple forces. Add the additional information to the definitions of the force vector `Force` and the angle vector `theta`. Note that we assumed a two-dimensional world in this example, but it would be easy to extend the solution to forces in all three dimensions.

3.5 DATA ANALYSIS FUNCTIONS

Analyzing data statistically in MATLAB is particularly easy, partly because whole data sets can be represented by a single array and partly because of the large number of built-in data analysis functions.

3.5.1 Maximum and Minimum

Table 3.5 lists functions that find the minimum and maximum in a data set, and the element at which those values occur.

Table 3.5 Maxima and Minima

`max(x)`	Finds the largest value in a vector `x`. For example, if $x = [1\ 5\ 3]$, the maximum value is 5.	`x = [1, 5, 3];`
		`max(x)`
		`ans = 5`

	Creates a row vector containing the maximum element from each column of an array `x`. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, then the maximum value in column 1 is 2, the maximum value in column 2 is 5, and the maximum value in column 3 is 6.	`x = [1, 5, 3; 2, 4, 6];`
		`max(x)`
		`ans = 2  5  6`

`[a,b] = max(x)`	Finds both the largest value in a vector `x` and its location in vector `x`. For $x = [1\ 5\ 3]$ the maximum value is named `a` and is equal to 5. The location of the maximum value is element 2 and is named `b`.	`x = [1, 5, 3];`
		`[a,b] = max(x)`
		`a = 5`
		`b = 2`

	Creates a row vector containing the maximum element from each column of an array `x` and returns a row vector with the location of the maximum in each column of array `x`. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, then the maximum value in column 1 is 2, the maximum value in column 2 is 5, and the maximum value in column 3 is 6. These maxima occur in row 2, row 1, and row 2, respectively.	`x = [1, 5, 3; 2, 4, 6];`
		`[a,b] = max(x)`
		`a = 2  5  6`
		`b = 2  1  2`

(*continued*)

Table 3.5 (continued)

max(x,y)	Creates an array the same size as x and y. (Both x and y must have the same number of rows and columns.) Each element in the resulting array contains the maximum value from the corresponding positions in x and y. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$ and $y = \begin{bmatrix} 10 & 2 & 4 \\ 1 & 8 & 7 \end{bmatrix}$, then the resulting array will be $x = \begin{bmatrix} 10 & 5 & 4 \\ 2 & 8 & 7 \end{bmatrix}$.	`x = [1, 5, 3; 2, 4, 6];`
		`y = [10,2,4; 1, 8, 7];`
		`max(x,y)`
		`ans = 10 5 4`
		`2 8 7`

min(x)	The syntax for the min function is the same as that for the max function, except that minimums are returned.	

All of the functions in this section work on the *columns* in two-dimensional arrays. MATLAB is column dominant—in other words, if there is a choice to make, MATLAB will choose columns first over rows. If your data analysis requires you to evaluate data in rows, the easiest approach is to transpose the data array. (In other words, the rows must become columns and the columns must become rows.) The transpose operator is a single quote ('). For example, if you want to find the maximum value in each *row* of the array

$$x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$$

use the command

```
max(x')
```

which returns

```
ans=
    5 6
```

Another approach to solving this same problem is to specify the dimension to be analyzed. Since MATLAB is column dominant, columns are specified as dimension 1 and rows are specified as dimension 2. The dimension must be entered in the third input field in the max function. Since most commonly the second field is not used, this requires a place-holder. Thus, to find the maximum in each row of the x array,

```
max(x,[],2)
```

returns

```
ans =
          5
          6
```

HINT

A common mistake when finding the maximum or minimum value in a data set is to name the result max or min. This overwrites the function, and the function is no longer available for calculations. For example,

```
max = max(x)
```

results in a variable named max for the answer. This is allowable MATLAB code, but is not wise. Trying to use the max function later in the program will result in an error. For example,

```
another_max = max(y)
```

will return

```
??? Index exceeds the number of array elements
```

PRACTICE EXERCISE 3.5

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore. Consider the following array:

$$x = \begin{bmatrix} 4 & 90 & 85 & 75 \\ 2 & 55 & 65 & 75 \\ 3 & 78 & 82 & 79 \\ 1 & 84 & 92 & 93 \end{bmatrix}$$

1. What is the maximum value in each column?
2. In which row does that maximum occur?
3. What is the maximum value in each row? (You'll have to transpose the array to answer this question or specify the dimension number.)
4. In which column does the maximum occur?
5. What is the maximum value in the entire array?

3.5.2 Averages

KEY IDEA
There are multiple ways to define the average value in a data set, including the mean, median and mode.

There are several ways to find the "**average**" value in a data set. In statistics, the **mean** of a group of values is probably what most of us would call the average. The mean is the sum of all the values, divided by the total number of values. Another kind of average is the **median**, or the middle value. There are an equal number of values both larger and smaller than the median. The mode is the value that appears most often in a data set. MATLAB provides functions for finding the mean, median, and the mode, as shown in Table 3.6. All of these functions are column dominant, and will return an answer for each column in a two-dimensional array. As is true with the `max` and `min` functions, you can specify the dimension of interest to determine these function values based on rows. The dimension is specified in the second field, unlike the `max` and `min` functions where it is specified in the third field.

Table 3.6 Averages

`mean(x)`	Computes the mean value (or average value) of a vector `x`. For example, if $x = [1\ 5\ 3]$, the mean value is 3.	`x = [1, 5, 3];`
		`mean(x)`
		`ans = 3.0000`

	Returns a row vector containing the mean value from each column of an array `x`. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, then the mean value of column 1 is 1.5, the mean value of column 2 is 4.5, and the mean value of column 3 is 4.5.	`x = [1, 5, 3; 2, 4, 6];`
		`mean(x)`
		`ans = 1.5  4.5  4.5`

`median(x)`	Finds the median of the elements of a vector `x`. For example, if $x = [1\ 5\ 3]$, the median value is 3.	`x = [1, 5, 3];`
		`median(x)`
		`ans = 3`

(*continued*)

Table 3.6 (continued)

	Returns a row vector containing the median value from each column of an array `x`. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \\ 3 & 8 & 4 \end{bmatrix}$, then the median value from column 1 is 2, the median value from column 2 is 5, and the median value from column 3 is 4.	`x = [1, 5, 3;`
		`2, 4, 6;`
		`3, 8, 4];`
		`median(x)`
		`ans = 2 5 4`

`mode(x)`	Finds the value that occurs most often in an array. Thus, for the array `x = [1, 2, 3, 3]` the mode is 3.	`x = [1, 2, 3, 3]`
		`mode(x)`
		`ans = 3`

PRACTICE EXERCISE 3.6

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore. Consider the following array:

$$x = \begin{bmatrix} 4 & 90 & 85 & 75 \\ 2 & 55 & 65 & 75 \\ 3 & 78 & 82 & 79 \\ 1 & 84 & 92 & 93 \end{bmatrix}$$

1. What is the mean value in each column?
2. What is the median for each column?
3. What is the mean value in each row?
4. What is the median for each row?
5. What is returned when you request the mode?
6. What is the mean for the entire array?

3.5.3 Sums and Products

Often it is useful to add up (sum) all of the elements in an array, or to multiply all of the elements together. MATLAB provides a number of functions to calculate both sums and products, as shown in Table 3.7.

In addition to simply adding up all the elements, which returns a single value for each column in the array, the `cumsum` function (cumulative sum) adds all of the previous elements in an array and creates a new array of these intermediate totals. This is useful when dealing with the sequences of numbers in a series. Consider the harmonic series

$$\sum_{k=1}^{n} \frac{1}{k}$$

which is equivalent to

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}$$

We could use MATLAB to create a sequence representing the first five values in the sequence as follows

```
k = 1:5;
sequence = 1./k
```

which gives us

```
sequence =
  1.0000    0.5000    0.3333    0.2500    0.2000
```

Table 3.7 Sums and Products

`sum(x)`	Sums the elements in vector `x`. For example, if $x = [1\ 5\ 3]$, the sum is 9.	`x = [1, 5, 3];`
		`sum(x)`
		`ans = 9`

	Computes a row vector containing the sum of the elements in each column of an array `x`. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, then the sum of column 1 is 3, the sum of column 2 is 9, and the sum of column 3 is 9.	`x = [1, 5, 3; 2, 4, 6];`
		`sum(x)`
		`ans = 3 9 9`

`prod(x)`	Computes the product of the elements of a vector `x`. For example, if $x = [1\ 5\ 3]$, the product is 15.	`x = [1, 5, 3];`
		`prod(x)`
		`ans = 15`

	Computes a row vector containing the product of the elements in each column of an array `x`. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, then the product of column 1 is 2, the product of column 2 is 20, and the product of column 3 is 18.	`x = [1, 5, 3; 2, 4, 6];`
		`prod(x)`
		`ans = 2 20 18`

`cumsum(x)`	Computes a vector of the same size as, and containing cumulative sums of the elements of, a vector `x`. For example, if $x = [1\ 5\ 3]$, the resulting vector is $x = [1\ 6\ 9]$.	`x = [1, 5, 3];`
		`cumsum(x)`
		`ans = 1 6 9`

	Computes an array containing the cumulative sum of the elements in each column of an array `x`. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, the resulting array is $x = \begin{bmatrix} 1 & 5 & 3 \\ 3 & 9 & 9 \end{bmatrix}$.	`x = [1, 5, 3; 2, 4, 6];`
		`cumsum(x)`
		`ans = 1 5 3`
		`3 9 9`

`cumprod(x)`	Computes a vector of the same size as, and containing cumulative products of the elements of, a vector `x`. For example, if $x = [1\ \ 5\ \ 3]$ the resulting vector is $x = [1\ \ 5\ \ 15]$.	`x = [1, 5, 3];`
		`cumprod(x)`
		`ans = 1 5 15`

	Computes an array containing the cumulative product of the elements in each column of an array `x`. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, the resulting array is $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 20 & 18 \end{bmatrix}$.	`x = [1, 5, 3; 2, 4, 6];`
		`cumprod(x)`
		`ans = 1 5 3`
		`2 20 18`

We could view the series as a sequence of fractions by changing the format to rational with the following code

```
format rat
sequence =
   1     1/2     1/3     1/4     1/5
```

Now we could use the `cumsum` function to find the value of the entire series for values of n from 1 to 5

```
format short
series = cumsum(sequence)
series =
   1.0000      1.5000      1.8333      2.0833      2.2833
```

Similarly, the `cumprod` function finds the cumulative product of a sequence of numbers stored in an array.

3.5.4 Sorting Values

Table 3.8 lists several commands to sort data in an array into ascending or descending order. For example, if we define an array ***x***

```
x = [1 6 3 9 4]
```

we can use the `sort` function to rearrange the values.

```
sort(x)
ans =
   1      3      4      6      9
```

Table 3.8 Sorting Functions

`sort(x)`	Sorts the elements of a vector `x` in ascending order. For example, if $x = [1\ 5\ 3]$, the resulting vector is $x = [1\ 3\ 5]$.	`x = [1,5,3];`
		`sort(x)`
		`ans = 1 3 5`

	Sorts the elements in each column of an array `x` in ascending order. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, the resulting array is $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$.	`x = [1,5,3; 2,4,6];`
		`sort(x)`
		`ans = 1 4 3`
		`2 5 6`

`sort(x,"descend")`	Sorts the elements in each column in descending order.	`x = [1,5,3; 2,4,6];`
		`sort(x,"descend")`
		`ans = 2 5 6`
		`1 4 3`

`sortrows(x)`	Sorts the rows in an array in ascending order on the basis of the values in the first column, and keeps each row intact. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \\ 3 & 8 & 4 \end{bmatrix}$, then using the `sortrows` command will move the middle row into the top position. The first column defaults to the basis for sorting.	`x = [3,1,3; 1,9,3;`
		`4, 3, 6]`
		`sortrows(x)`
		`ans = 1 9 3`
		`3 1 2`
		`4 3 6`

`sortrows(x,n)`	Sorts the rows in an array on the basis of the values in column n. If n is negative, the values are sorted in descending order. If n is not specified, the default column used as the basis for sorting is column 1.	`sortrows(x,2)`
		`ans = 3 1 2`
		`4 3 6`
		`1 9 3`

The default is ascending order, but adding the string 'descend' to the second field will force the function to list the values in descending order.

```
sort(x,"descend")
ans =
   9     6     4     3     1
```

You can also use the `sort` command to rearrange entire arrays. This function is consistent with other MATLAB functions, and sorts based on columns. Each column will be sorted independently. Thus,

```
x = [1 3; 10 2; 3 1; 82 4; 5 5]
```

gives the following:

```
x =
   1    3
   10   2
   3    1
   82   4
   5    5
```

When we sort the array

```
sort(x)
```

each column is sorted in ascending order.

```
ans =
      1   1
      3   2
      5   3
      10  4
      82  5
```

The `sortrows` function allows you to sort entire rows, based on the value in a specified column. Thus,

```
sortrows(x,1)
```

sorts based on the first column, but maintains the relationship between values in columns one and two.

```
ans =
     1    3
     3    1
     5    5
     10   2
     82   4
```

Similarly, you can sort based on values in the second column.

```
sortrows(x,2)
ans =
     3    1
     10   2
     1    3
     2    4
     5    5
```

Table 3.9 2021 Olympic Women's 200-meter Freestyle Swimming Final

Name	Country	Time	Lane
Federica Pellegrini	Italy	1:55.91	1
Barbora Seemanova	Czech Republic	1:55.45	2
Katie Ledecky	United States	1:55.21	3
Ariarne Titmus	Australia	1:53.50	4
Siobhan Haughey	Hong Kong	1:53.92	5
Yang Junxuan	China	1:55.01	6
Penny Oleksiak	Canada	1:54.72	7
Madison Wilson	Australia	1:56.39	8

These functions are particularly useful in analyzing data. Consider the results of the Women's 200-meter Freestyle Swimming event shown in Table 3.9. The swimmers are listed in the table based on the lane in which they swam, but once the race was complete it makes more sense to sort the table based on the times in the third column. These data were stored in MATLAB in a table array called `Swimming_Results_2021`. Tables can store multiple types of information, including text like that found in the first two columns, durations like the times in the third column, and numbers like those in the last column. Both the `sort` function and the `sortrows` function can operate on tables and can sort either numerically or alphabetically. The default is to sort in ascending order based on the first column. For this example, the result displayed in the Live Editor would be:

```
sortrows(Swimming_Results_2021)
ans = 8×4 table
```

	Name	Country	Time	Lane
1	"Ariarne Titmus"	"Australia"	01:53.50	4
2	"Barbora Seemanova"	"Czech Republic"	01:55.45	2
3	"Federica Pellegrini"	"Italy"	01:55.91	1
4	"Katie Ledecky"	"United States"	01:55.21	3
5	"Madison Wilson"	"Australia"	01:56.39	8
6	"Penny Oleksiak"	"Canada"	01:54.72	7
7	"Siobhan Haughey"	"Hong Kong"	01:53.92	5
8	"Yang Junxuan"	"China"	01:55.01	6

To sort by column the column number is added in the second input field, as in:

```
sortrows(Swimming_Results_2021,3)
ans = 8×4 table
```

	Name	Country	Time	Lane
1	"Ariarne Titmus"	"Australia"	01:53.50	4
2	"Siobhan Haughey"	"Hong Kong"	01:53.92	5
3	"Penny Oleksiak"	"Canada"	01:54.72	7
4	"Yang Junxuan"	"China"	01:55.01	6
5	"Katie Ledecky"	"United States"	01:55.21	3
6	"Barbora Seemanova"	"Czech Republic"	01:55.45	2
7	"Federica Pellegrini"	"Italy"	01:55.91	1
8	"Madison Wilson"	"Australia"	01:56.39	8

The winning time was posted by Ariarne Titmus of Australia, swimming in lane 4.

The `sortrows` function can also sort in descending order by specifying the order in the third field as "descend," or by placing a negative sign in front of the column number in the second field.

3.5.5 Determining Array Size

MATLAB offers several functions (see Table 3.10) that allow us to determine the size of an array. The most commonly used are: `size, length,` and `numel`. The `size` function returns the number of rows and columns in an array. The `length` function returns the larger of the array dimensions. The `numel` function returns the total number of elements in an array. For example, if

```
x = [1 2 3; 4 5 6];
size(x)
```

MATLAB returns the following result:

```
ans =
     2     3
```

This tells us that the `x` array has two rows and three columns. However, if we use the `length` function

```
length(x)
```

the result is

```
ans =
     3
```

Table 3.10 Array Size Functions

`size(x)`	Determines the number of rows and columns in array `x`. (If `x` is a multidimensional array, `size` determines how many dimensions exist and how big they are.)	`x =[1,5,3; 2,4,6];`
		`size (x)`
		`ans = 2  3`

`[a,b] = size(x)`	Determines the number of rows and columns in array `x` and assigns the number of rows to `a` and the number of columns to `b`.	`[a,b] = size (x)`
		`a = 2`
		`b = 3`

`height(x)`	Determines the number of rows in array `x`. It is equivalent to the value returned to `a` in `[a,b]= size(x)` or `size(x,1)`	`height(x)` `ans = 2`
`width(x)`	Determines the number of columns in array `x`. It is equivalent to the value returned to `b` in `[a,b]= size(x)` or `size(x,2)`	`width(x)` `ans = 3`

`length(x)`	Determines the largest dimension of an array `x`.	`x =[1,5,3; 2,4,6];`
		`length(x)`
		`ans = 3`

`numel(x)`	Determines the total number of elements in an array `x`.	`x =[1,5,3; 2,4,6];`
		`numel(x)`
		`ans = 6`

because the largest of the array dimensions is 3.

Finally, if we use the `numel` function

```
numel(x)
```

the result is

```
ans =
     6
```

The `length` function is particularly useful when used with a loop structure, since it can easily determine how many times to execute the loop—based on the dimensions of an array.

PRACTICE EXERCISE 3.7

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore. Consider the following array:

$$x = \begin{bmatrix} 4 & 90 & 85 & 75 \\ 2 & 55 & 65 & 75 \\ 3 & 78 & 82 & 79 \\ 1 & 84 & 92 & 93 \end{bmatrix}$$

1. Use the `size` function to determine the number of rows and columns in this array.
2. Use the `sort` function to sort each column in ascending order.
3. Use the `sort` function to sort each column in descending order.
4. Use the `sortrows` function to sort the array so that the first column is in ascending order, but each row still retains its original data. Your array should look like this:

$$x = \begin{bmatrix} 1 & 84 & 92 & 93 \\ 2 & 55 & 65 & 75 \\ 3 & 78 & 82 & 79 \\ 4 & 90 & 85 & 75 \end{bmatrix}$$

5. Use the `sortrows` function to sort the array from Exercise 4 in descending order, based on the third column.

EXAMPLE 3.3

WEATHER DATA

The United States National Weather Service collects massive amounts of weather data every day (see Figure 3.5). Those data are available to all of us on the agency's online service at http://www.ncdc.noaa.gov/. Analyzing large amounts of data can be confusing, so it's a good idea to start with a small data set, develop an approach that works, then apply it to the larger data set that we are interested in.

The file *Weather_Data.xlsx* contains precipitation data gathered by the United States National Weather Service for one location for all of 1999. (The .xlsx indicates

(*continued*)

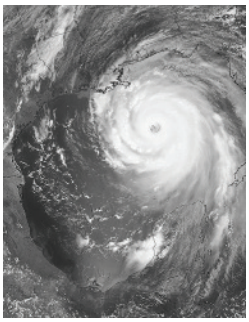


Figure 3.5
Satellite photo of a hurricane. (NASA Headquarters)

that the data are in an Excel spreadsheet.) Each row represents a month, so there are 12 rows, and each column represents the day of the month (1 to 31), so there are 31 columns. As not every month has the same number of days, data are missing for some locations in the last several columns. The precipitation information is presented in hundredths of an inch. For example, on February 1, there was 0.61 inch of precipitation, and on April 1, 2.60 inches. A sample of the data is displayed in Table 3.11, with labels added for clarity; however, *the data in the file contain only numbers.*

Use the data in the file to find the following:

a. The total precipitation in each month.
b. The total precipitation for the year.
c. The month and day on which the maximum precipitation during the year was recorded.

1. State the problem
Using the data in the file *Weather_Data.xlsx*, find the total monthly precipitation, the total precipitation for the year, and the day on which it rained the most.

2. Describe the input and output
Input The input for this example is included in a data file called *Weather_Data.xlsx* and consists of a two-dimensional array. Each row represents a month, and each column represents a day.
Output The output should be the total precipitation for each month, the total precipitation for the year, and the day on which the precipitation was a maximum. We have decided to present precipitation in inches, since no other units were specified in the statement of the problem.

3. Develop a hand example
For the hand example, deal only with a small subset of the data. The information included in Table 3.11 is enough. The total for January, days 1 to 4, is

$$\text{total_1} = (0 + 0 + 272 + 0)/100 = 2.72 \text{ inches}$$

The total for February, days 1 to 4, is

$$\text{total_2} = (61 + 103 + 0 + 2)/100 = 1.66 \text{ inches}$$

Table 3.11 Precipitation Data from Asheville, North Carolina

1999	Day1	Day2	Day3	Day4	...	Day28	Day29	Day30	Day31
January	0	0	272	0		0	0	33	33
February	61	103	0	2		62			
March	2	0	17	27		0	5	8	0
April	260	1	0	0		13	86	0	
May	47	0	0	0		0	0	0	0
June	0	0	30	42		14	14	8	
July	0	0	0	0		5	0	0	0
August	0	45	0	0		0	0	0	0
September	0	0	0	0		138	58	10	
October	0	0	0	14		0	0	0	1
November	1	163	5	0		0	0	0	
December	0	0	0	0		0	0	0	0

Now add the months together to get the combined total. If our sample "year" is just January and February, then

$$\text{total} = \text{total_1} + \text{total_2} = 2.72 + 1.66 = 4.38 \text{ inches}$$

To find the day on which the maximum precipitation occurred, first find the maximum in the table, and then determine in which row and column it occurred.

Working through a hand example allows you to formulate the steps required to solve the problem in MATLAB.

4. Develop a MATLAB solution

First, we'll need to save the data file into MATLAB as an array. Because the file is an Excel spreadsheet, the easiest approach is to import the data interactively. Double-click on the file in the current folder window to open the file in MATLAB. Select *Numeric Matrix* as the output type, and select Replace Blanks with NaN, then select *Import*.

The variable name `weatherdata` will appear in the workspace window (see Figure 3.6). Because not every month has 31 days, there were a number of blanks in the spreadsheet. The value NaN was inserted into those fields by MATLAB, based on our choices when we imported the file. You can double-click the variable name, `weatherdata`, in the workspace window, to view the resulting array (see Figure 3.7).

Now write the program to solve the problem:

```
%% Example 3.3 - Weather Data
% In this example we will find the total precipitation
% for each month, and for the entire year, using a data file
% We will also find the month and day on which the
% precipitation was the maximum
clc
% Find the sum of each row, which is the sum for each month
monthly_total=sum(weatherdata,2,'omitnan')/100;
monthly_total'
% Find the annual total
yearly_total = sum(monthly_total)
% Find the annual maximum, and the day on which it occurs
```

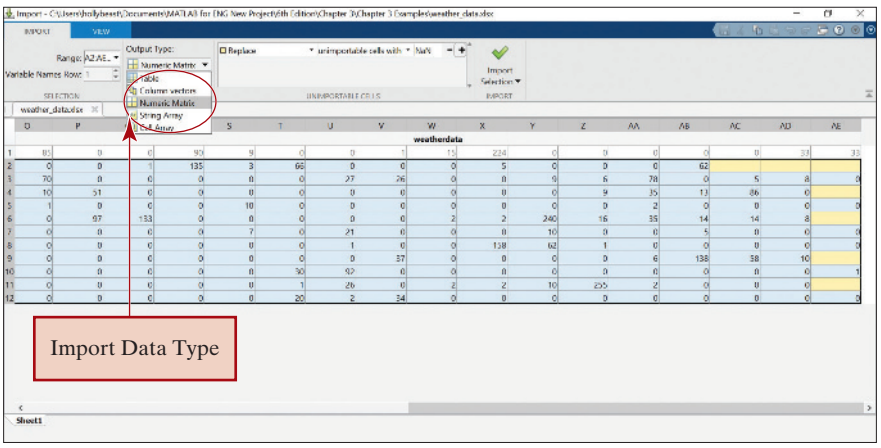


Figure 3.6
MATLAB allows the user to import data interactively.

(*continued*)

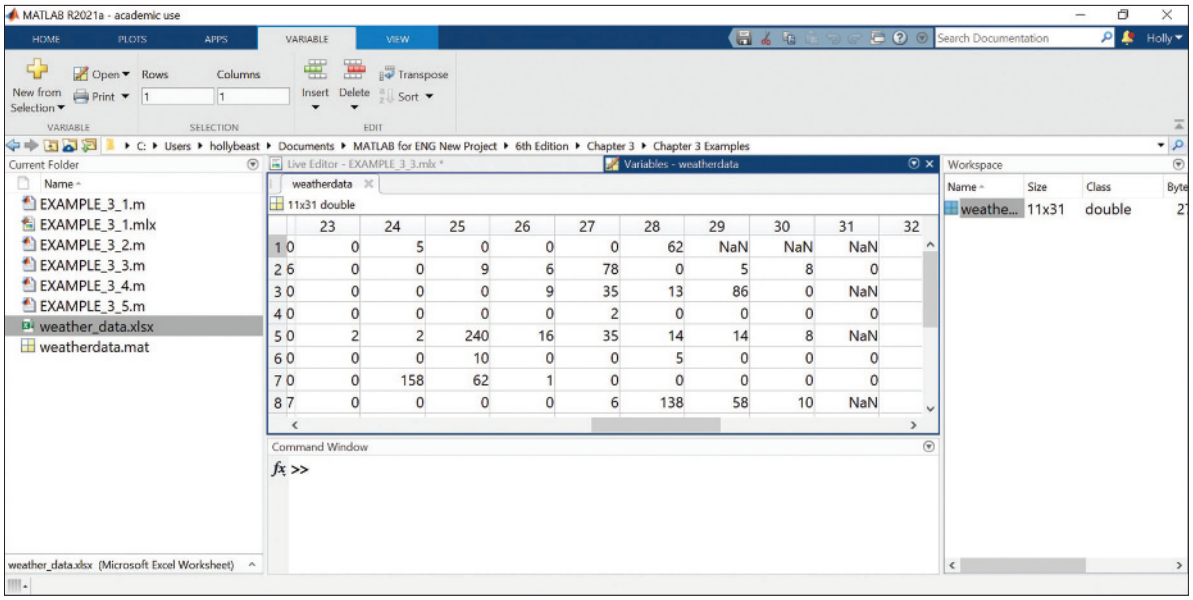


Figure 3.7
The MATLAB array editor. You can edit the array in this window and change any values if needed.

```
[max_precip,month]= max(max(weatherdata,[],2,'omitnan'))
% Find the annual maximum, and the month in which it occurs
[max_precip,day]= max(max(weatherdata,[],1,'omitnan'))
```

Notice that the code did not start with our usual `clear, clc` commands, because that would clear the workspace, effectively deleting the `weatherdata` variable.

To find the monthly total we needed to add up the values on each row. We could have transposed the array and just used sum directly,

$$\textbf{monthly_total} = \textbf{sum}(\textbf{weatherdata}', '\textbf{omitnan}')/\textbf{100}$$

but instead we chose to specify rows in the second field with the number 2. With either approach, we needed to tell MATLAB to ignore the NaN (not a number) entries.

Now we can add up all the monthly totals to get the total for the year. An alternative syntax is

$$\textbf{yearly_total} = \textbf{sum(sum(weatherdata,'omitnan'))/100}$$

Finding the maximum daily precipitation is easy; what makes this example hard is determining the day and month on which the maximum occurred. The command

$$[\textbf{max_precip, month}] = \textbf{max(max(weatherdata,[ ], 2,'omitnan'))}$$

is easier to understand if we break it up into two commands. First,

$$[\textbf{a,b}] = \textbf{max(weatherdata,[ ], 2,'omitnan')}$$

returns an array of maxima for each row, which in this case is the maximum for each month. This value is assigned to the variable name `a`. The variable `b`

becomes an array of index numbers that represent the row in each column at which the maximum occurred. The result, then, is

```
a =
    Columns 1 through 9
     272   135    78   260   115   240   157   158   138
    Columns 10 through 12
     156   255    97
b =
    Columns 1 through 9
       3    18    27     1     6    25    12    24    28
    Columns 10 through 12
       5    26    14
```

Now when we execute the `max` command the second time, we determine the maximum precipitation for the entire data set, which is the maximum value in array *a*. Also, from array *a*, we find the index number for that maximum:

```
[c,d] = max(a)
c =
     272
d =
     1
```

These results tell us that the maximum precipitation occurred in row 1 of the `a` array, which means that it occurred in the first month.

Similarly, specifying columns in the third field with the number one

$$[\mathbf{max_precip, day}] = \mathbf{max(weatherdata,[\ ],1,'omitnan')}$$

and finding the maximum twice allows us to find the day of the month on which the maximum occurred.

There are several things you should notice about the MATLAB screen shown in Figure 3.8. All of the variables created when the script was executed are now available to the command window. This makes it easy to perform additional calculations in the command window after the program has completed running.

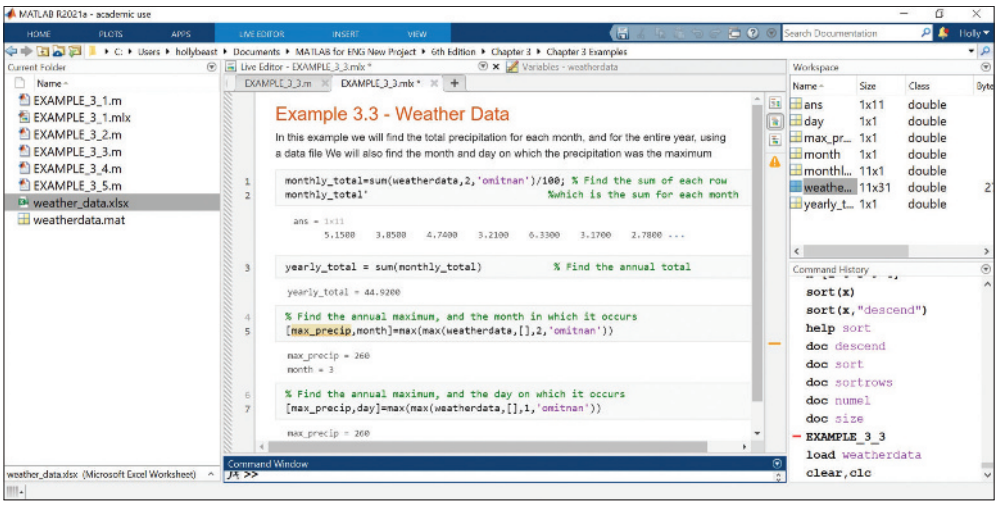


Figure 3.8
Results from the precipitation calculations.

(*continued*)

For example, notice that we forgot to change the `max_precip` value to inches from hundredths of an inch. Adding the command

$$\mathbf{max_precip = max_precip/100}$$

would correct that oversight. Notice also that the *Weather_Data.xlsx* file is still in the current folder. Finally, notice that the command history window reflects only commands issued from the command window; it does not show commands executed from a script.

5. Test the solution
Open the *Weather_Data.xlsx* file, and confirm that the maximum precipitation occurred on January 3. Once you've confirmed that your program works, you can use it to analyze other data. The United States National Weather Service maintains similar records for all of its recording stations. The World Meteorological Organization provides information for selected cities worldwide.

HINT

You can prompt a user to interactively import data from a MATLAB program, using the `uiimport` function. For example, in Example 3.3

```
uiimport("weatherdata.xlsx")
```

would have opened the data import window shown in Figure 3.6. You should also program in a message, so the user knows what is expected and a pause in the execution of your code.

```
disp("Import the data as a column vector")
disp("then select any key to continue")
pause
```

KEY IDEA
Standard deviation and variance are measures of the spread of values in a data set.

3.5.6 Variance and Standard Deviation

The **standard deviation** and **variance** are measures of how much elements in a data set vary with respect to each other. Every student knows that the average score on a test is important, but you also need to know the high and low scores to get an idea of how well you did. Test scores, like many kinds of data that are important in engineering, are often distributed in a bell-shaped curve. In a normal (Gaussian) distribution of a large amount of data, approximately 68% of the data falls within one standard deviation (sigma) of the mean ($\pm$ one sigma). If you extend the range to a two-sigma variation ($\pm$ two sigma), approximately 95% of the data should fall inside these bounds, and if you go out to three sigma, over 99% of the data should fall in this range (see Figure 3.9). Usually, measures such as the standard deviation and variance are meaningful only with large data sets.

Consider the data graphed in Figure 3.10. Both sets of data have the same average (mean) value of 50. However, it is easy to see that the first data set has more variation than the second.

The mathematical definition of variance is

$$\text{variance} = \sigma^2 = \frac{\sum_{k=1}^{N}(x_k - \mu)^2}{N-1}$$

In this equation, the symbol μ represents the mean of the values x_k in the data set. Thus, the term $x_k - \mu$ is simply the difference between the actual value and the average value. The terms are squared and added together:

$$\sum_{k=1}^{N}(x_k - \mu)^2$$

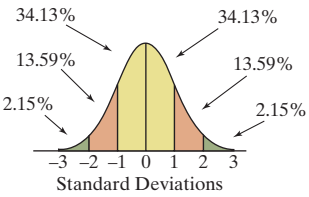


Figure 3.9
Normal distribution.

Finally, we divide the summation term by the number of values in the data set (N), minus 1.

The standard deviation which is used more often than the variance, is the square root of the variance.

The MATLAB function used to find the standard deviation is `std`. When we applied this function on the large data set shown in Figure 3.10, we obtained the following output:

```
std(scores1)
ans =
    20.3653
std(scores2)
ans =
    9.8753
```

In other words, approximately 68% of the data in the first data set fall between the average, 50, and ±20.3653. Similarly 68% of the data in the second data set fall between the same average, 50, and ±9.8753.

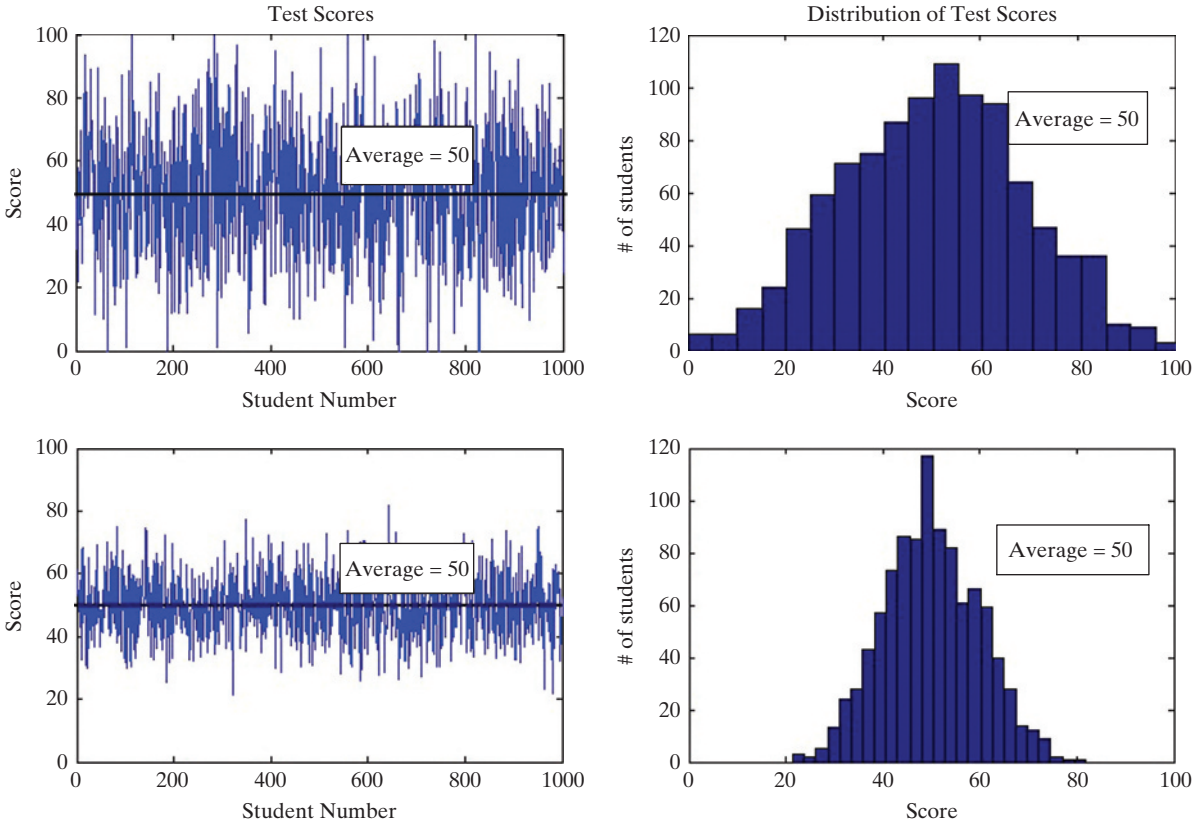


Figure 3.10
Test scores from two different tests.

The variance is found in a similar manner with the `var` function:

```
var(scores1)
ans =
  414.7454
var(scores2)
ans =
  97.5209
```

The syntax for calculating both standard deviation and variance is shown in Table 3.12.

Table 3.12 Statistical Functions

`std(x)`	Computes the standard deviation of the values in a vector `x`. For example, if $x = [1\ 5\ 3]$, the standard deviation is 2. However, standard deviations are not usually calculated for small samples of data.	`x = [1, 5, 3];`
		`std(x)`
		`ans = 2`

	Returns a row vector containing the standard deviation calculated for each column of an array `x`. For example, if $x = \begin{bmatrix} 1 & 5 & 3 \\ 2 & 4 & 6 \end{bmatrix}$, the standard deviation in column 1 is 0.7071, the standard deviation in column 2 is 0.7071, and the standard deviation in column 3 is 2.1213.	`x = [1, 5, 3;2, 4, 6];`
		`std (x)`
		`ans =`
		`0.7071 0.7071`
		`2.1213`

	Again, standard deviations are not usually calculated for small samples of data.	
`var (x)`	Calculates the variance of the data in `x`. For example, if $x = [1\ 5\ 3]$, the variance is 4. However, variance is not usually calculated for small samples of data. Notice that the standard deviation in this example is the square root of the variance.	`var(x)` `ans = 4`

PRACTICE EXERCISE 3.8

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore. Consider the following array:

$$x = \begin{bmatrix} 4 & 90 & 85 & 75 \\ 2 & 55 & 65 & 75 \\ 3 & 78 & 82 & 79 \\ 1 & 84 & 92 & 93 \end{bmatrix}$$

1. Find the standard deviation for each column.
2. Find the variance for each column.
3. Calculate the square root of the variance you found for each column.
4. How do the results from Exercise 3 compare against the standard deviation you found in Exercise 1?

EXAMPLE 3.4

CLIMATOLOGICAL DATA

Climatologists examine weather data over long periods of time, trying to find a pattern. Weather data have been kept reliably in the United States since the 1850s; however, most reporting stations have been in place only since the 1930s and 1940s (see Figure 3.11). Climatologists perform statistical calculations on the data they collect. Although the data in *Weather_Data.xlsx* represent just one location for one year, we can use them to practice statistical calculations. Find the mean daily precipitation for each month and the mean daily precipitation for the year, and then find the standard deviation for each month and for the year.

1. State the problem
   Find the mean daily precipitation for each month and for the year, on the basis of the data in *Weather_Data.xlsx*. Also, find the standard deviation of the data during each month and during the entire year.

2. Describe the input and output
   Input Use the *Weather_Data.xlsx* file as input to the problem.

   Output find:
   The mean daily precipitation for each month.
   The mean daily precipitation for the year.
   The standard deviation of the daily precipitation data for each month.
   The standard deviation of the daily precipitation data for the year.

3. Develop a hand example
   Use just the data for the first 4 days of the month:
   January average $= (0 + 0 + 272 + 0)/4 = 68$ hundredths of an inch of precipitation, or 0.68 inch.
   The standard deviation is found from the following equation:

$$\sigma = \sqrt{\frac{\sum_{k=1}^{N}(x_k - \mu)^2}{N - 1}}$$



Figure 3.11
The National Weather Service is an agency of the United States government, and is part of NOAA. (lafoto/Shutterstock)

(*continued*)

Using just the first 4 days of January, first calculate the sum of the squares of the difference between the mean and the actual value:

$$(0 - 68)^2 + (0 - 68)^2 + (272 - 68)^2 + (0 - 68)^2 = 55{,}488$$

Divide by the number of data points minus 1:

$$55{,}488/(4 - 1) = 18{,}496$$

Finally, take the square root, to give 136 hundredths of an inch of precipitation, or 1.36 inches.

4. Develop a MATLAB solution
First we need to load the *Weather_ Data.xlsx* file making sure the blank values are replaced with NaN (not a number). Although we could do that as described in Example 3.3, there is an easier way: The data from Example 3.3 could be saved to a file, so that they are available to use later. If we want to save the entire workspace, just type

```
save <filename>
```

where `filename` is a user-defined file name. If you just want to save one variable, type

```
save <filename> <variable_name>
```

which saves a single variable or a list of variables to a file. All we need to save is the variable `weatherdata`, so the following command is sufficient:

```
save weatherdata weatherdata
```

This command saves the array `weatherdata` into the *weatherdata.mat* file. Check the current folder window to make sure that *weatherdata.mat* has been stored (see Figure 3.12).

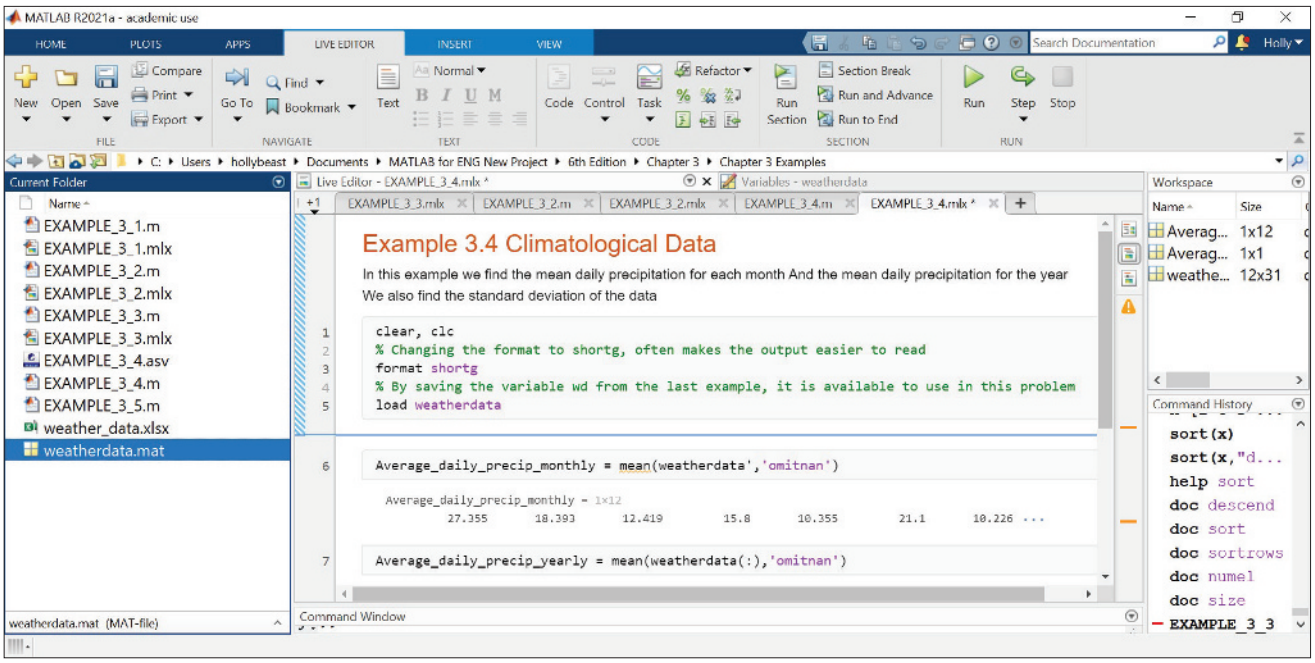


Figure 3.12
The current folder records the name of the saved file. The program code used to solve the problem is the same whether you use a script M-file or a live script MLX-file.

Now the program we create to solve this example can load the data automatically. This program can be executed as either a script M-file or a live script—the program code is the same.

```
%% Example 3.4 Climatological Data
% In this example we find the mean daily
% precipitation for each month
% And the mean daily precipitation for the year
% We also find the standard deviation of the data
clear, clc
% Changing the format to shortg, often makes the output
% easier to read
format shortg
% By saving the variable wd from the last example, it is
% available to use in this problem
load weatherdata
%%
Average_daily_precip_monthly = mean(weatherdata','omitnan')
Average_daily_precip_yearly = mean(weatherdata(:),'omitnan')
%% Now calculate the standard deviation
Monthly_Stdeviation = std(weatherdata','omitnan')
Yearly_Stdeviation = std(weatherdata(:),'omitnan')
```

Notice that we included the command `omitnan` to tell the functions not to include the array values listed as NaN (not a number) in the calculation. The weatherdata array was also transposed because we desired the standard deviation of the row, not the column.

Some new syntax was introduced in this example. The command

```
weatherdata(:)
```

converts the two-dimensional array `weatherdata` into a one-dimensional array, thus making it possible to find the mean and the standard deviation of the entire array.

5. Test the solution

First, check the results to make sure they make sense. Although calculating the mean daily rainfall for one month by hand would serve as an excellent check, it would be tedious. You can use MATLAB to help you by calculating the mean without using a predefined function. The command window is a convenient place to perform these calculations:

```
load weatherdata
sum(weatherdata(:,1))     % Find the sum of all the rows in
                          % column one of array weather_data
ans =
    848.00
ans/31
ans =
    27.35
```

Compare these results with those for January (month 1).

HINT

Use the colon operator to change a two-dimensional array into a single column:

```
A = X (:)
```

3.6 RANDOM NUMBERS

Random numbers are often used in engineering calculations to simulate measured data. Measured data rarely behave exactly as predicted by mathematical models, so we can add small values of random numbers to our predictions to make a model behave more like a real system. Random numbers are also used to model games of chance. Two different types of random numbers can be generated in MATLAB: **uniform random numbers** and **Gaussian random numbers** (often called a normal distribution or **normal random number**).

3.6.1 Uniform Random Numbers

Uniform random numbers (also called normal random numbers) are generated with the `rand` function. These numbers are evenly distributed between 0 and 1. (Consult the help function for more details.) Table 3.13 lists several MATLAB commands for generating random numbers.

We can create a set of random numbers over other ranges by modifying the numbers created by the `rand` function. For example, to create a set of 100 evenly distributed numbers between 0 and 5, first create a set over the default range with the command

```
r = rand(100,1);
```

Table 3.13 Random-Number Generators

`rand(n)`	Returns an $n \times n$ array. Each value in the array is a random number between 0 and 1.	`rand(2)`
		`ans =`
		`0.9501 0.6068`
		`0.2311 0.4860`

`rand(m,n)`	Returns an $m \times n$ array. Each value in the array is a random number between 0 and 1.	`rand(3,2)`
		`ans =`
		`0.8913 0.0185`
		`0.7621 0.8214`
		`0.4565 0.4447`

`randn(n)`	Returns an $n \times n$ array. Each value in the array is a Gaussian (or normal) random number with a mean of 0 and a variance of 1.	`randn(2)`
		`ans =`
		`-0.4326 -0.1253`
		`-1.6656 -0.2877`

`randn(m,n)`	Returns an $m \times n$ array. Each value in the array is a Gaussian (or normal) random number with a mean of 0 and a variance of 1.	`randn(3,2)`
		`ans =`
		`-1.1465 -0.0376`
		`1.1909 0.3273`
		`1.1892 0.1746`

`randi(imax,sz)`	Returns an array of random integers between 1 and imax, where sz is an array that defines the size of the resulting array. This function has multiple input options. For more details consult the help feature.	`sz = [2,3]`
		`randi(10,sz)`
		`ans =`
		`3 4 9`
		`6 3 7`

This results in a 100 × 1 array of values. Now multiply by 5 to expand the range to 0 to 5:

```
r = r * 5;
```

To change the range to 5 to 10, add 5 to every value in the array:

```
r = r + 5;
```

The result will be random numbers varying from 5 to 10. We can generalize these results with the equation

$$x = (\max - \min) \cdot \text{random_number_set} + \min$$

3.6.2 Gaussian Random Numbers

Gaussian random numbers have the normal distribution shown in Figure 3.9. There is no absolute upper or lower bound to a data set of this type; we are just less and less likely to find data, the farther away from the mean we get. Gaussian random-number sets are described by specifying their average (mean) and the standard deviation of the data set.

MATLAB generates Gaussian values with a mean of 0 and a variance of 1.0, using the `randn` function. For example,

```
randn(3)
```

returns a 3 × 3 array:

```
ans =
 -0.4326    0.2877    1.1892
 -1.6656   -1.1465   -0.0376
  0.1253    1.1909    0.3273
```

To create a data set with a different average or a different standard deviation, start with the default set of random numbers, and then modify it. Since the default standard deviation is 1, *multiply* by the required standard deviation for the new data set. Since the default mean is 0, that means you'll need to *add* the new mean:

$$x = \text{standard_deviation} \cdot \text{random_data_set} + \text{mean}$$

For example, to create a sequence of 500 Gaussian random variables with a standard deviation of 2.5 and a mean of 3, type

```
x = randn(1,500)*2.5 + 3;
```

Notice that both `rand` and `randn` can accept either one or two input values. If only one is specified, the result is a square array. If two values are specified, they represent the number of rows and the number of columns in the resulting array.

PRACTICE EXERCISE 3.9

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Create a 3 × 3 array of evenly distributed random numbers.
2. Create a 3 × 3 array of normally distributed random numbers.
3. Create a 100 × 5 array of evenly distributed random numbers. Be sure to suppress the output.

4. Find the maximum, the standard deviation, the variance, and the mean for each column in the array that you created in Exercise 3.
5. Create a 100 × 5 array of normally distributed random numbers. Be sure to suppress the output.
6. Find the maximum, the standard deviation, the variance, and the mean for each column in the array you created in Exercise 5.
7. Explain why your results for Exercises 4 and 6 are different.

EXAMPLE 3.5

NOISE

Random numbers can be used to simulate the noise we hear as static on the radio. By adding this noise to data files that store music, we can study the effect of static on recordings.

MATLAB has the ability to play music files by means of the `sound` function. To demonstrate this function, it also has a built-in music file with a short segment of Handel's *Messiah* (see Figure 3.13). In this example, we will use the `randn` function to create noise, then we'll add the noise to the music clip.

Music is stored in MATLAB as an array with values from −1 to 1. To convert this array into music, the `sound` function requires a sample frequency. The *handel.mat* file contains both an array representing the music and the value of the sample frequency. To hear the *Messiah,* you must first load the file, using the command

```
load handel
```

Notice that two new variables—`y` and `Fs`—were added to the workspace window when the `handel` file was loaded. To play the clip, type

```
sound(y, Fs)
```



Figure 3.13
Utah Symphony Orchestra. (Chris Fredriksson/Alamy Stock Photo)

Experiment with different values of `Fs` to hear the effect of different sample frequencies on the music. For example, multiply `Fs` by 1.5 and observe the result. (The sound must be engaged on your computer, or you won't be able to hear the playback.)

1. State the problem
   Add a noise component to the recording of Handel's *Messiah* included with MATLAB.

2. Describe the input and output

   Input MATLAB data file of Handel's *Messiah,* stored as the built-in file *handel.mat*

   Output An array representing the *Messiah,* with static added
   A graph of the first 200 elements of the data file

3. Develop a hand example
   As the data in the music file vary between −1 and +1, we should add noise values of a smaller order of magnitude. First, we'll try values centered on 0 and with a standard deviation of 0.1.

4. Develop a MATLAB solution

```
%% Example 3.5
% Noise
load handel          % Load the music data file
sound(y,Fs)          % Play the music data file
pause                % Pause to listen to the music
% Be sure to hit enter in the command window to continue
% after playing the music
% Add random noise
noise = randn(length(y),1)*0.1;
sound(y+noise,Fs)
```

   This program allows you to play the recording of the *Messiah,* both with and without the added noise. You can adjust the multiplier on the noise line to observe the effect of changing the magnitude of the added static. For example,

```
noise=randn(length(y),1)*0.20
```

5. Test the solution
   In addition to playing back the music both with and without the added noise, we could plot the results. Because the file is quite large (73,113 elements), we'll just plot the first 200 points:

```
% Plot the first 200 data points in each file
t=1:length(y);
noisy = y+noise;
plot(t(1,1:200),y(1:200,1),t(1,1:200),y(1:200,1) ...
    +noisy(1:200,1),':')
title("Handel's Messiah")
xlabel("Element Number in Music Array")
ylabel("Frequency")
```

   These commands tell MATLAB to plot the index number of the data on the *x*-axis, and the value stored in the music array on the *y*-axis. Plotting is introduced in more detail in a later chapter.

(*continued*)

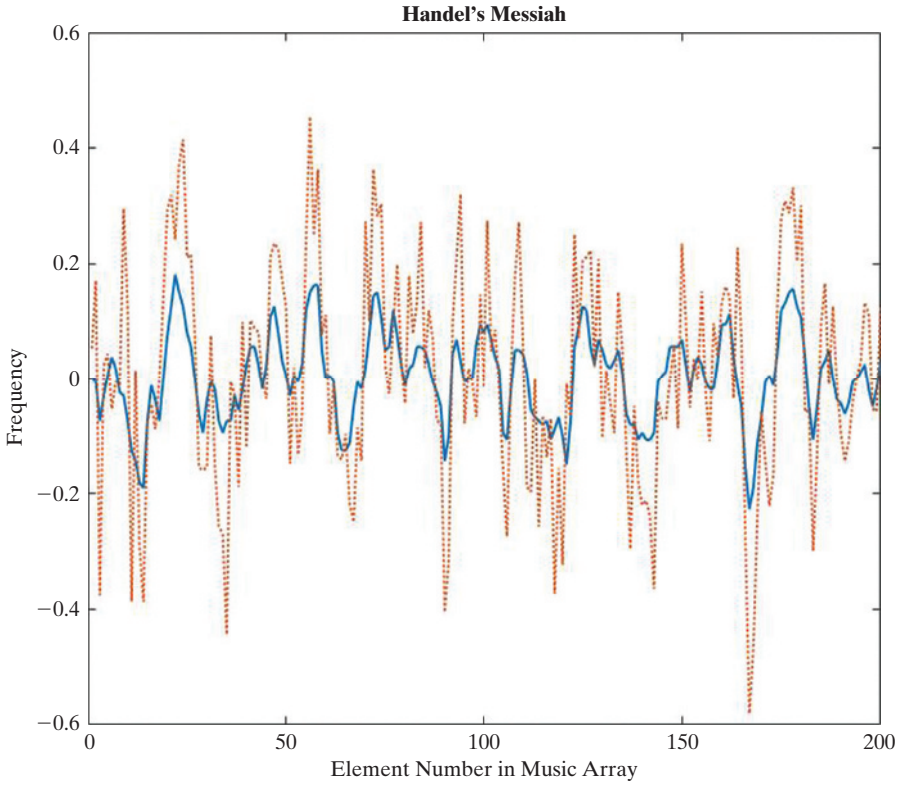


Figure 3.14
Handel's *Messiah*. The solid line represents the original data, and the dotted line is the data to which we've added noise.

In Figure 3.14, the solid line represents the original data, and the dotted line the data to which we've added noise. As expected, the noisy data have a bigger range and don't always follow the same pattern as the original.

3.7 COMPLEX NUMBERS

MATLAB includes several functions used primarily with **complex numbers**. Complex numbers consist of two parts: a real component and an imaginary component. For example,

$$5 + 3i$$

is a complex number. The real component is 5, and the imaginary component is 3. Complex numbers can be entered into MATLAB in two ways: as an addition problem, such as

`A = 5 + 3i` or `A = 5 + 3*i` or `A = 5 + 3*j`

or with the `complex` function, as in

```
A = complex (5,3)
```

which returns

```
A =
  5.0000 + 3.0000i
```

KEY IDEA
MATLAB supports the use of both i and j for imaginary numbers.

As is standard in MATLAB, the input to the `complex` function can be either two scalars or two arrays of values. Thus, if ***x*** and ***y*** are defined as

```
x = 1:3;
y = [-1,5,12];
```

then the `complex` function can be used to define an array of complex numbers as follows:

```
complex (x, y)
ans =
  1.0000 - 1.0000i   2.0000 + 5.0000i   3.0000 + 12.0000i
```

The `real` and `imag` functions can be used to separate the real and imaginary components of complex numbers. For example, for `A = 5 + 3*i`, we have

```
real(A)
ans =
      5
imag(A)
ans =
      3
```

The `isreal` function can be used to determine whether a variable is storing a complex number. It returns a 1 if the variable is real, and a 0 if it is complex. Since `A` is a complex number, we get

```
isreal(A)
ans =
      0
```

Thus, the `isreal` function is false and returns a value of 0.

The complex conjugate of a complex number consists of the same real component, but an imaginary component of the opposite sign. The `conj` function returns the complex conjugate:

```
conj(A)
ans =
     5.0000 - 3.0000i
```

The transpose operator also returns the complex conjugate of an array, in addition to converting rows to columns and columns to rows. Thus, we have

```
A'
ans =
     5.0000 - 3.0000i
```

Of course, in this example `A` is a scalar. We can create a complex array `B` by using `A` and performing both addition and multiplication operations:

```
B = [A, A+1, A*3]
B =
  5.0000 + 3.0000i   6.0000 + 3.0000i   15.0000 + 9.0000i
```

The transpose of `B` is

```
B'
ans =
   5.0000 - 3.0000i
   6.0000 - 3.0000i
  15.0000 - 9.0000i
```

Complex numbers are often thought of as describing a position on an *x*–*y* plane. The real part of the number corresponds to the *x*-value, and the imaginary component corresponds to the *y*-value, as shown in Figure 3.15(a). Another way to think about this point is to describe it with **polar coordinates**—that is, with a radius and an angle (see Figure 3.15(b)).

KEY IDEA
One way to think of complex numbers is as a point on an *x*-*y* plane, which can also be described using polar coordinates.

MATLAB includes functions to convert complex numbers from Cartesian to polar form.

When the absolute-value function is used with a complex number, it calculates the radius, using the Pythagorean theorem:

```
abs(A)
ans =
     5.8310
```

$$\text{radius} = \sqrt{(\text{real component})^2 + (\text{imaginary component})^2}$$

Since, in this example, the real component is 5, and the imaginary component is 3,

$$\text{radius} = \sqrt{5^2 + 3^2} = 5.8310$$

We could also calculate the radius in MATLAB, using the `real` and `imag` functions described earlier:

```
sqrt(real(A).^2 + imag(A).^2)
ans =
    5.8310
```

Similarly, the angle is found with the angle function:

```
angle(A)
ans =
    0.5404
```

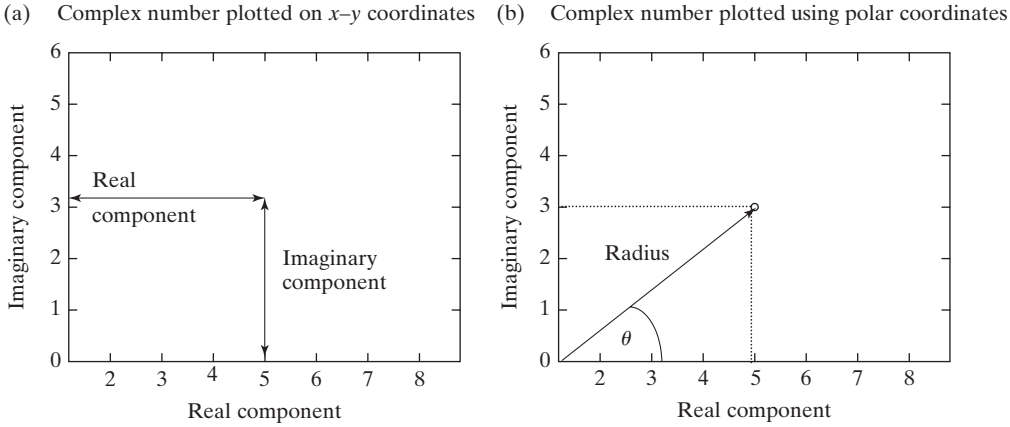


Figure 3.15
(a) Complex number represented in a Cartesian coordinate system. (b) A complex number can also be described with polar coordinates.

The result is expressed in radians. Both functions, `abs` and `angle`, will accept scalars or arrays as input. Recall that `B` is a 1×3 array of complex numbers:

```
B =
  5.0000 + 3.0000i   6.0000 + 3.0000i   15.0000 + 9.0000i
```

The `abs` function returns the radius if the number is represented in polar coordinates:

```
abs(B)
ans =
    5.8310    6.7082   17.4929
```

The angle from the horizontal can be found with the `angle` function:

```
angle(B)
ans =
    0.5404    0.4636    0.5404
```

The MATLAB functions commonly used with complex numbers are summarized in Table 3.14.

Table 3.14 Functions Used with Complex Numbers

`abs(x)`	Computes the absolute value of a complex number, using the Pythagorean theorem. This is equivalent to the radius if the complex number is represented in polar coordinates.	`x = 3 + 4i;`
		`abs(x)`
		`ans =`
		`5`

`angle(x)`	For example, if $x = 3 + 4i$, the absolute value is $\sqrt{3^2 + 4^2} = 5$. Computes the angle from the horizontal in radians when a complex number is represented in polar coordinates.	`x = 3 + 4i;`
		`angle(x)`
		`ans =`
		`0.9273`

`complex(x,y)`	Generates a complex number with a real component *x* and an imaginary component *y*.	`x = 3;`
		`y = 4;`
		`complex(x,y)`
		`ans =`
		`3.0000+`
		`4.0000i`

`real(x)`	Extracts the real component from a complex number.	`x = 3 + 4i;`
		`real(x)`
		`ans =`
		`3`

`imag(x)`	Extracts the imaginary component from a complex number.	`x = 3 + 4i;`
		`imag(x)`
		`ans =`
		`4`

`isreal(x)`	Determines whether the values in an array are real. If they are real, the function returns a 1; if they are complex, it returns a 0.	`x = 3 + 4i;`
		`isreal(x)`
		`ans =`
		`0`

`conj(x)`	Generates the complex conjugate of a complex number.	`x = 3 + 4i;`
		`conj(x)`
		`ans =`
		`3.0000-`
		`4.0000i`

PRACTICE EXERCISE 3.10

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Create the following complex numbers:
   a. $A = 1 + i$
   b. $B = 2 - 3i$
   c. $C = 8 + 2i$
2. Create a vector `D` of complex numbers whose real components are 2, 4, and 6 and whose imaginary components are −3, 8, and −16.
3. Find the magnitude (absolute value) of each of the vectors you created in Exercises 1 and 2.
4. Find the angle from the horizontal of each of the complex numbers you created in Exercises 1 and 2.
5. Find the complex conjugate of vector `D`.
6. Use the transpose operator to find the complex conjugate of vector `D`.
7. Multiply `A` by its complex conjugate, and then take the square root of your answer. How does this value compare against the magnitude (absolute value) of `A`?

KEY IDEA
There is a limit to how small or how large a number can be handled by computer programs.

3.8 COMPUTATIONAL LIMITATIONS

The variables stored in a computer can assume a wide range of values. On the majority of computers, the range extends from about 10^{-308} to 10^{308}, which should be enough to accommodate most computations. MATLAB includes functions to identify the largest real numbers and the largest integers the program can process (see Table 3.15).

The value of `realmax` is the largest possible floating-point number used in MATLAB and corresponds roughly to 2^{1024}, since computers actually perform their calculations in binary (base-2) arithmetic. Of course, it is possible to formulate a problem in which the result of an expression is larger or smaller than the permitted maximum. For example, suppose that we execute the following commands:

```
x = 2.5e200;
y = 1.0e200;
z = x*y
```

Table 3.15 Computational Limits

`realmax`	Returns the largest possible floating-point number used in MATLAB.	`realmax`
		`ans =`
		`1.7977e + 308`

`realmin`	Returns the smallest possible floating-point number used in MATLAB.	`realmin`
		`ans =`
		`2.2251e - 308`

`intmax`	Returns the largest possible integer number used in MATLAB.	`intmax`
		`ans =`
		`2147483647`

`intmin`	Returns the smallest possible integer number used in MATLAB.	`intmin`
		`ans =`
		`-2147483648`

MATLAB responds with

```
z =
   Inf
```

because the answer (2.5e400) is outside the allowable range. This error is called *exponent* **overflow**, because the exponent of the result of an arithmetic operation is too large to store in the computer's memory.

Exponent **underflow** is a similar error, caused by the exponent of the result of an arithmetic operation being too *small* to store in the computer's memory. The `realmin` function returns the smallest possible floating point number used in MATLAB. We can obtain an exponent underflow with the following commands:

```
x = 2.5e-200;
y = 1.0e200
z = x/y
```

Together, these commands return

```
z = 0
```

The result of an exponent underflow is zero.

We also know that division by zero is an invalid operation. If an expression results in a division by zero, the result of the division is infinity:

```
z = y/0
z =
        Inf
```

In performing calculations with very large or very small numbers, it may be possible to reorder the calculations to avoid an underflow or an overflow. Suppose, for example, that you would like to perform the following string of multiplications:

KEY IDEA
Careful planning can help you avoid calculational overflow or underflow.

$$(2.5 \times 10^{200}) \times (2 \times 10^{200}) \times (1 \times 10^{-100})$$

The answer is 5×10^{300}, within the bounds allowed by MATLAB. However, consider what happens when we enter the problem into MATLAB:

```
2.5e200*2e200*1e-100
ans =
          Inf
```

Because MATLAB executes the problem from left to right, the first multiplication yields a value outside the allowable range (5×10^{400}), resulting in an answer of infinity. However, by rearranging the problem to

```
2.5e200*1e-100*2e200
ans =
      5.0000e+300
```

we avoid the overflow and find the correct answer.

3.9 SPECIAL VALUES AND MISCELLANEOUS FUNCTIONS

Most, but not all, functions require an input argument. Although used as if they were scalar constants, the commands listed in Table 3.16 are functions that do *not* require any input.

Table 3.16 Special Functions

`pi`	Mathematical constant π.	`pi` `ans = 3.1416`
`i`	Imaginary number.	`i` `ans = 0 + 1.0000i`
`j`	Imaginary number.	`j` `ans = 0 + 1.0000i`

`Inf`	Infinity, which often occurs during a calculational overflow or when a number is divided by zero.	`5/0`
		`Warning: Divide by zero.`
		`ans = Inf`

`NaN`	Not a number.
	Occurs when a calculation is undefined.
		`0/0`
		`Warning: Divide by zero.`
		`ans = NaN`
		`inf/inf`
		`ans = NaN`

`clock`	Current time.
	Returns a six-member array [year month day hour minute second]. When the `clock` function was called on November 26, 2021, at 9:46 A.M and 4.6 seconds, MATLAB returned the output shown at the right. The format was set to shortg.
		`clock`
		`ans =`
		`2021 11 26 9`
		`46 4.655`

`date`	Current date.
	Similar to the `clock` function. However, it returns the date in a "string format."
		`date`
		`ans =`
		`'26-Nov-2021'`

`eps`	The distance between 1 and the next-larger double-precision floating-point number.	`eps` `ans = 2.2204e-016`

MATLAB allows you to redefine these functions as variable names; however, doing so can have unexpected consequences. For example, the following MATLAB code is allowed, even though it is not wise:

```
pi = 12.8;
```

From this point on, whenever the variable `pi` is called, the new value will be used. Similarly, you can redefine *any* function as a variable name, such as

```
sin = 10;
```

To restore `sin` to its job as a trigonometric function (or to restore the default value of `pi`), you must clear the workspace with

```
clear
```

or you may clear each independently with

```
clear sin
clear pi
```

Now check to see the result by issuing the command for π.

```
pi
```

This command returns

```
pi =
3.1416
```

HINT

The function *i* is the most common of these functions to be unintentionally renamed by MATLAB users.

The `NaN` function stands for "not a number." It is returned when a user attempts a calculation where the result is undefined—for example, 0/0. It can also be useful as a placeholder in an array.

PRACTICE EXERCISE 3.11

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Use the `clock` function to add the time and date to your work sheet.
2. Use the `date` function to add the date to your work sheet.
3. Convert the following calculations to MATLAB code and explain your results:
   a. 322! (Remember that, to a mathematician, the symbol ! means factorial.)
   b. $5 * 10^{500}$
   c. $1/5 * 10^{500}$
   d. 0/0

SUMMARY

In this chapter, we explored a number of predefined MATLAB functions, including the following:

- General mathematical functions, such as
  - exponential functions
  - logarithmic functions
  - roots
- Rounding functions
- Functions used in discrete mathematics, such as
  - factoring functions
  - prime-number functions
- Trigonometric functions, including
  - standard trigonometric functions
  - inverse trigonometric functions
  - hyperbolic trigonometric functions
  - trigonometric functions that use degrees instead of radians
- Data analysis functions, such as
  - maxima and minima
  - averages (mean and median)
  - sums and products
  - sorting
  - standard deviation and variance

- Random-number generation for both
  - uniform distributions
  - Gaussian (normal) distributions
- Functions used with complex numbers

We explored the computational limits inherent in MATLAB and introduced special values, such as `pi`, that are built into the program.

MATLAB SUMMARY

The following MATLAB summary lists and briefly describes the special characters, commands, and functions that were defined in this chapter:

Special Characters and Functions	
`eps`	smallest difference recognized
`i`	imaginary number
`clock`	returns the time
`date`	returns the date
`Inf`	infinity
`intmax`	returns the largest possible integer number used in MATLAB
`intmin`	returns the smallest possible integer number used in MATLAB
`j`	imaginary number
`NaN`	not a number
`pi`	mathematical constant π
`realmax`	returns the largest possible floating-point number used in MATLAB
`realmin`	returns the smallest possible floating-point number used in MATLAB

Commands and Functions	
`abs`	computes the absolute value of a real number or the magnitude of a complex number
`angle`	computes the angle when complex numbers are represented in polar coordinates
`asin`	computes the inverse sine (arcsine)
`asind`	computes the inverse sine and reports the result in degrees
`ceil`	rounds to the nearest integer toward positive infinity
`complex`	creates a complex number
`conj`	creates the complex conjugate of a complex number
`cos`	computes the cosine using radians as input
`cosd`	computes the cosine using degrees as input
`cumprod`	computes a cumulative product of the values in an array
`cumsum`	computes a cumulative sum of the values in an array
`deg2rad`	converts degrees to radians
`erf`	calculates the error function
`exp`	computes the value of e^x
`factor`	finds the prime factors
`factorial`	calculates the factorial
`fix`	rounds to the nearest integer toward zero
`floor`	rounds to the nearest integer toward minus infinity
`gcd`	finds the greatest common denominator
`help`	opens the help function
`helpwin`	opens the windowed help function
`imag`	extracts the imaginary component of a complex number
`isprime`	determines whether a value is prime
`isreal`	determines whether a value is real or complex

(*continued*)

Commands and Functions	
`lcm`	finds the least common multiple
`length`	determines the largest dimension of an array
`log`	computes the natural logarithm or the logarithm to the base *e* ($\log_e$)
`log10`	computes the common logarithm or the logarithm to the base 10 ($\log_{10}$)
`log2`	computes the logarithm to the base 2 ($\log_2$)
`max`	finds the maximum value in an array and determines which element stores the maximum value
`mean`	computes the average of the elements in an array
`median`	finds the median of the elements in an array
`min`	finds the minimum value in an array and determines which element stores the minimum value
`mode`	finds the most common number in an array
`nchoosek`	finds the number of possible combinations when a subgroup of k values is chosen from a group of n values
`nthroot`	find the real *n*th root of the input array
`numel`	determines the total number of elements in an array
`primes`	finds the prime numbers less than the input value
`prod`	multiplies the values in an array
`rad2deg`	converts radians to degrees
`rand`	calculates evenly distributed random numbers
`randn`	calculates normally distributed (Gaussian) random numbers
`randi`	calculates evenly distributed random integers
`rats`	converts the input to a rational representation (i.e., a fraction)
`real`	extracts the real component of a complex number
`rem`	calculates the remainder in a division problem
`round`	rounds to the nearest integer
`sign`	determines the sign (positive or negative)
`sin`	computes the sine, using radians as input
`sind`	computes the sine, using angles in degrees as input
`sinh`	computes the hyperbolic sine
`size`	determines the number of rows and columns in an array
`sort`	sorts the elements of a vector
`sortrows`	sorts the rows of a vector on the basis of the values in the first column
`sound`	plays back music files
`sqrt`	calculates the square root of a number
`std`	determines the standard deviation
`sum`	sums the values in an array
`tan`	computes the tangent, using radians as input
`var`	computes the variance

KEY TERMS

argument
average
complex number
discrete mathematics
function
function input
Gaussian random numbers
mean
median
nesting
normal random numbers
overflow
polar coordinates
rational numbers
real numbers
standard deviation
underflow
uniform random number
variance

PROBLEMS

Elementary Math Functions

3.1 Find the cube root of -5, both by using the `nthroot` function and by raising -5 to the $1/3$ power. Explain the difference in your answers. Prove that both results are indeed correct answers by cubing them and showing that they equal -5.

3.2 MATLAB contains functions to calculate the natural logarithm (`log`), the logarithm to the base 10 (`log10`), and the logarithm to the base 2 (`log2`). However, if you want to find a logarithm to another base—for example, base b—you'll have to do the math yourself with the formula

$$\log_b(x) = \frac{\log_e(x)}{\log_e(b)}$$

What is the $\log_b$ of 10 when b is defined from 1 to 10 in increments of 1?

3.3 Populations tend to expand exponentially, that is,

$$P = P_0 e^{rt}$$

where

P = current population,
P_0 = original population,
r = continuous growth rate, expressed as a fraction, and
t = time.

If you originally have 100 rabbits that breed at a continuous growth rate of 90% ($r = 0.9$) per year, find how many rabbits you will have at the end of 10 years.

3.4 Chemical reaction rates are proportional to a rate constant k that changes with temperature according to the Arrhenius equation

$$k = k_0 e^{-Q/RT}$$

For a certain reaction,

$$Q = 8000 \text{ cal/mol}$$
$$\mathrm{R} = 1.987 \text{ cal/mol K}$$
$$\mathrm{k}_0 = 1200 \text{ min}^{-1}$$

Find the values of k for temperatures from 100 K to 500 K, in increments of 50. Create a table of your results.

3.5 Consider the air-conditioning requirements of the large home shown in Figure P3.5.

The interior of the house is warmed by waste heat from lighting and electrical appliances, by heat leaking in from the outdoors, and by heat generated by the people in the home. An air-conditioner must be able to remove all this thermal energy in order to keep the inside temperature from rising. Suppose there are 20 light bulbs emitting 100 J/s of energy each, and four appliances emitting 500 J/s each. Suppose also that heat leaks in from the outside at a rate of 3000 J/s.

(a) How much heat must the air-conditioner be able to remove from the home per second?

(b) One particular air-conditioning unit can handle 2000 J/s. How many of these units are needed to keep the home at a constant temperature?

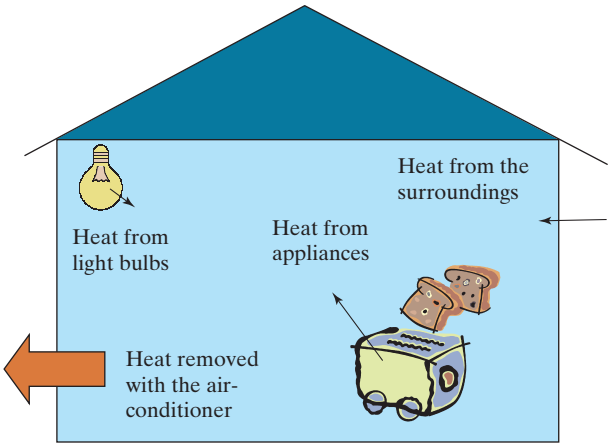


Figure P3.5
Air-conditioning must remove heat from a number of sources.

3.6 **(a)** If you have four people, how many different ways can you arrange them in a line?

(b) If you have 10 different tiles, how many different ways can you arrange them?

3.7 **(a)** If you have 12 people, how many different committees of two people each can you create? Remember that a committee of Maria and Alice is the same as a committee of Alice and Maria.

(b) How many different soccer teams of 11 players can you form from a class of 30 students? (Combinations—order does not matter.)

(c) Since each player on a soccer team is assigned a particular role, order *does* matter. Recalculate the possible number of different soccer teams that can be formed when order is taken into account.

3.8 There are 52 different cards in a deck. How many different hands of 5 cards each are possible? Remember, every hand can be arranged 120 (5!) different ways.

3.9 Very large prime numbers are used in cryptography. How many prime numbers are there between 10,000 and 20,000? (These aren't big enough primes to be useful in ciphers.) (*Hint:* Use the `primes` function and the `length` command.)

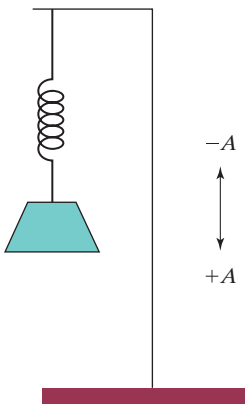


Figure P3.11
An oscillating spring.

Trigonometric Functions

3.10 Sometimes it is convenient to have a table of sine, cosine, and tangent values instead of using a calculator. Create an array of all three of these trigonometric functions for angles from 0 to 2π, with a spacing of 0.1 radian. Your array should contain a column for the angle and columns for the sine, cosine, and tangent.

3.11 The displacement of the oscillating spring shown in Figure P3.11 can be described by

$$x = A\cos(\omega t)$$

where

x = displacement at time t,
A = maximum displacement,
ω = angular frequency, which depends on the spring constant and the mass attached to the spring, and
t = time.

Find the displacement x for times from 0 to 10 seconds when the maximum displacement A is 4 cm, and the angular frequency is 0.6 radian/second. Present your results in a table of displacement and time values.

3.12 The acceleration of the spring described in the preceding exercise is

$$a = -A\omega^2 \cos(\omega t)$$

Find the acceleration for times from 0 to 10 seconds, using the constant values from the preceding problem. Create a table that includes the time, the displacement from corresponding values in the previous exercise, and the acceleration.

3.13 You can use trigonometry to find the height of a building, as shown in Figure P3.13. Suppose you measure the angle between the line of sight and the horizontal line connecting the measuring point and the building. You can calculate the height of the building with the following formulas:

$$\tan(\theta) = h/d$$
$$h = d \tan(\theta)$$

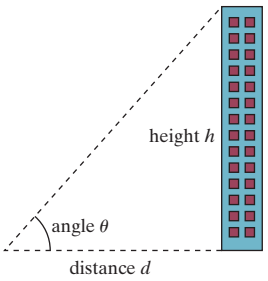


Figure P3.13
You can determine the height of a building with trigonometry.

Assume that the distance to the building along the ground is 120 m, and the angle measured along the line of sight is 30° ± 3°. Find the maximum and minimum heights the building can be.

3.14 Consider the building from the previous exercise.

(a) If it is 200 feet tall and you are 20 feet away, at what angle from the ground will you have to tilt your head to see the top of the building? (For simplicity assume that your head is even with the ground.)

(b) How far is it from your head to the top of the building?

(c) *Challenge:* Repeat parts (a) and (b) assuming your head is NOT even with the ground, but is 6 feet above ground level.

Data Analysis Functions

3.15 Consider the data in Table P3.15, which represents temperature readings in a reactor:

Table P3.15 Reactor Temperature Readings

Thermocouple 1	Thermocouple 2	Thermocouple 3
84.3	90.0	86.7
86.4	89.5	87.6
85.2	88.6	88.3
87.1	88.9	85.3
83.5	88.9	80.3
84.8	90.4	82.4
85.0	89.3	83.4
85.3	89.5	85.4
85.3	88.9	86.3
85.2	89.1	85.3
82.3	89.5	89.0
84.7	89.4	87.3
83.6	89.8	87.2

Your instructor may provide you with a file named *thermocouple.dat*, or you may need to enter the data yourself. Use MATLAB to find

(a) The maximum temperature measured by each thermocouple.

(b) The minimum temperature measured by each thermocouple.

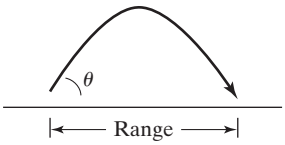


Figure P3.16
The range depends on the launch angle and the launch velocity.

3.16 The range of an object shot at an angle θ with respect to the x-axis and an initial velocity v_0 (see Figure P3.16) is given by

$$\text{Range} = \frac{v_0^2}{g}\sin(2\theta)$$

for $0 \leq \theta \leq \pi/2$ and neglecting air resistance. Use $g = 9.81\ \text{m/s}^2$ and an initial velocity v_0 of 100 m/s. Show that the maximum range is obtained at approximately $\theta = \pi/4$ by computing the range by changing theta in increments of $\pi/100$ between 0 and $\pi/2$. You won't be able to find the exact angle that results in the maximum range, because your calculations are at evenly spaced angles of $\pi/100$ radian.

3.17 The vector

$$G = [68, 83, 61, 70, 75, 82, 57, 5, 76, 85, 62, 71, 96, 78, 76, 68, 72, 75, 83, 93]$$

represents the distribution of final grades in a dynamics course. Compute the mean, median, mode, and standard deviation of *G*. Which better represents the "most typical grade," the mean, median, or mode? Why? Use MATLAB to determine the number of grades in the array (don't just count them) and to sort them into ascending order.

3.18 Generate 10,000 Gaussian random numbers with a mean of 80 and standard deviation of 23.5. (You'll want to suppress the output so that you don't overwhelm the command window with data.) Use the `mean` function to confirm that your array actually has a mean of 80. Use the `std` function to confirm that your standard deviation is actually 23.5.

3.19 Use the `date` function to add the current date to your homework.

Random Numbers

3.20 Many games require the player to roll two dice. The number on each die can vary from 1 to 6.

(a) Use the `randi` function to create a simulation of one roll of one die.

(b) Use your results from part (a) to create a simulation of the value rolled with a second die.

(c) Add your two results to create a value representing the total rolled during each turn.

(d) Use your program to determine the values rolled in a favorite board game.

3.21 Suppose you are designing a container to ship sensitive medical materials between hospitals. The container needs to keep the contents within a specified temperature range. You have created a model predicting how the container responds to the exterior temperature, and you now need to run a simulation.

(a) Create a normal distribution (Gaussian distribution) of temperatures with a mean of 70°F and a standard deviation of 2°, corresponding to a 2-hour duration. You'll need a temperature for each time value from 0 to 120 minutes, at 0.5 minute intervals. (That's 241 values.)

(b) Plot the data on an x–y plot. Don't worry about labels. Recall that the MATLAB function for plotting is `plot(x,y)`.

(c) Find the maximum temperature, the minimum temperature, and the times at which they occur.

Complex Numbers

3.22 Consider the circuit shown in Figure P3.22, which includes the following:

- A sinusoidally varying voltage source, *V*.
- An inductor, with an inductance, *L*.
- A capacitor, with a capacitance, *C*.
- A resistor, with a resistance, *R*.

We can find the current, *I*, in the circuit by using Ohm's law (generalized for alternating currents),

$$V = IZ_T$$

where Z_T is the total impedance in the circuit. (Impedance is the AC corollary to resistance.)

Assume that the impedance for each component is as follows:

$$\begin{aligned} Z_L &= 0 + 5j \text{ ohms} \\ Z_C &= 0 - 15j \text{ ohms} \\ R &= Z_R = 5 + 0j \text{ ohms} \\ Z_T &= Z_C + Z_L + R \end{aligned}$$

and that the applied voltage is

$$V = 10 + 0j \text{ volts}$$

(Electrical engineers usually use *j* instead of *i* for imaginary numbers.)

Find the current, *I*, in the circuit. You should expect a complex number as a result. Enter the complex values of impedance into your calculations using the `complex` function.

3.23 Impedance is related to the inductance, *L*, and the capacitance, *C* by the following equations:

$$\begin{aligned} Z_C &= \frac{1}{\omega C j} \\ Z_L &= \omega L j \end{aligned}$$

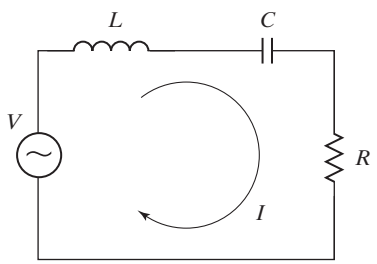


Figure P3.22
A simple circuit illustrating a sinusoidally varying voltage source, *V*.

For a circuit similar to the one shown in Figure P3.22, assume the following:

$$\begin{aligned} C &= 1\ \mu\text{F (microfarads)}. \\ L &= 200\text{ mH (millihenries)}. \\ R &= 5\text{ ohms}. \\ f &= 15\text{ kHz (kilohertz)}. \\ \omega &= 2\pi f. \\ V &= 10\text{ volts}. \end{aligned}$$

(a) Find the impedance for the capacitor (Z_C) and for the inductor (Z_L).

(b) Find the total impedance

$$Z_T = Z_C + Z_L + R$$

(c) Find the current by solving Ohm's law for *I*:

$$V = IZ_T$$

(d) Electrical engineers often describe complex parameters using polar coordinates, that is, the parameter has both an angle and a magnitude. (Imagine plotting a point on the complex plane, where the *x*-axis represents the real part of the number, and the *y*-axis represents the imaginary part of the number.) Use the `abs` function to find the magnitude of the current found in part (c), and use the `angle` function to find the corresponding angle.

3.24 Consider Table P3.24. The data was downloaded from the NOAA National Centers for Environmental Information website, and represents information collected at the Denver airport during 2015. It may be available to you as an Excel data file called *Denver_Climate_Data_2015.xlsx*. Import the data file to MATLAB and save each column as a vector.

EMXP: Extreme maximum daily precipitation
MMNT: Monthly mean minimum temperature
EMXT: Extreme maximum daily temperature
MXSD: Maximum snow depth
MMXT: Monthly mean maximum temperature
TPCP: Total precipitation
EMNT: Extreme minimum daily temperature
MNTM: Monthly mean temperature
TSNW: Total snow fall

(a) Find the total precipitation for 2015, using the TPCP column.
(b) Find the average monthly mean maximum temperature for 2015, using the MMXT column.
(c) Find the maximum snow depth for 2015 using the MXSD column.
(d) Find the maximum total snow fall, using the TSNW column.

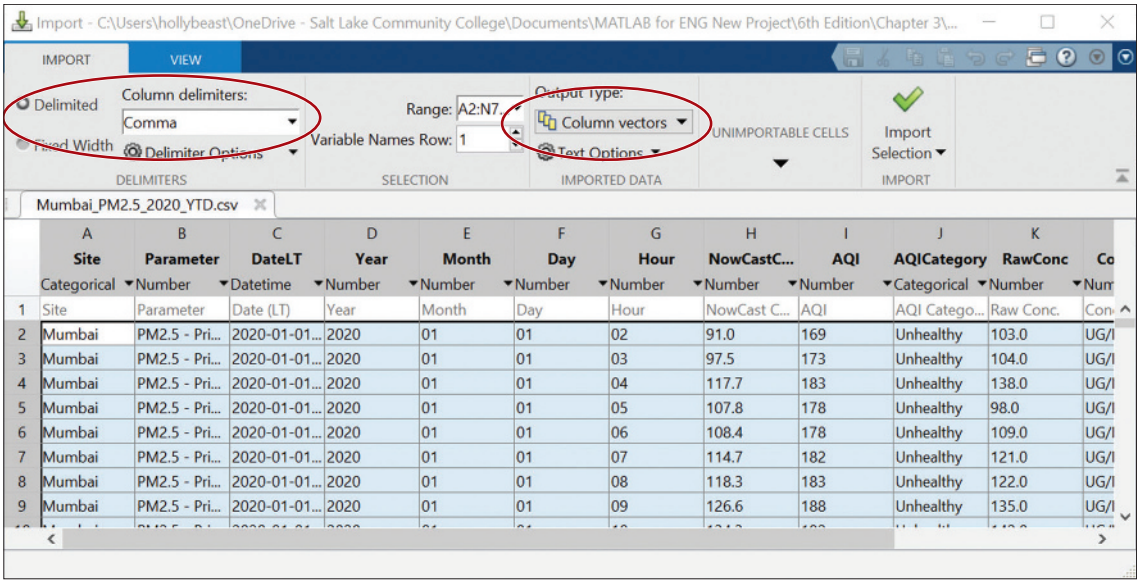
3.25 Consult the NOAA National Centers for Environmental Information website and download climate data for a location of your choice. (Airport names are usually good choices, for example, DEN for Denver, Colorado.) Specify temperature data in your request. Use the data file to determine the maximum and minimum temperatures during the previous year.

Table P3.24 Monthly Climate Data for Denver, Colorado, 2015

DATE	EMXP	MXSD	TPCP	TSNW	EMXT	EMNT	MMXT	MMNT	MNTM
20150101	0.12	0	0.37	7.2	75	−9.8	46.8	21.2	34
20150201	0.37	1	1.25	22.4	73.9	−5.8	46.6	21.2	34
20150301	0.34	0	0.79	2.9	81	2.1	58.8	31.3	45.1
20150401	1.22	0	2.65	5.3	79	27.1	62.1	35.4	48.7
20150501	1.08	0	3.76	4	84.9	27.1	63.3	42.8	53.1
20150601	0.99	0	2.54	0	93.9	51.1	82.9	56.1	69.4
20150701	0.33	0	1.07	0	97	52	87.6	57.9	72.7
20150801	0.68	0	1.18	0	98.1	43	89.8	58.3	74.1
20150901	0.07	0	0.11	0	91.9	44.1	85.3	53.6	69.4
20151001	0.79	0	1.76	0	87.1	28.2	70.2	42.8	56.5
20151101	0.5	0	2.13	11.2	75	3.2	51.3	25.7	38.5
20151201	0.32	1	0.71	11.3	69.1	0.1	41	18.1	29.5

3.26 The air quality index (AQI) for many locations around the world is reported online by AirNow. AirNow is a partnership among multiple United States federal, tribal, state, and local agencies that gather air quality data, including but not limited to the AQI. In addition to locations in the United States, the United States Department of State provides data from US Embassies and Consulates. Multiple locations in Canada and Mexico also participate in this data sharing partnership. You can download information from AirNow as a .csv file.

- Use a search engine to find the AirNow webpage and select a location that interests you.
- Select historical data.
- Download the AQI data for the most recent year. The file should look something like this: Mumbai_PM2.5_2020_YTD.csv
- Import the file into MATLAB using the `uiimport` function. Make sure to select an output type of 'Column vectors' and confirm that the 'Delimited' option is selected as shown in Figure P3.26.
- Use the `max` function to determine the maximum value of the AQI observed during the year.



	A	B	C	D	E	F	G	H	I	J	K	L
	Site	Parameter	DateLT	Year	Month	Day	Hour	NowCastC...	AQI	AQICategory	RawConc	Co
	Categorical	Number	Datetime	Number	Number	Number	Number	Number	Number	Categorical	Number	Num
1	Site	Parameter	Date (LT)	Year	Month	Day	Hour	NowCast C...	AQI	AQI Catego...	Raw Conc.	Con
2	Mumbai	PM2.5 - Pri...	2020-01-01...	2020	01	01	02	91.0	169	Unhealthy	103.0	UG/I
3	Mumbai	PM2.5 - Pri...	2020-01-01...	2020	01	01	03	97.5	173	Unhealthy	104.0	UG/I
4	Mumbai	PM2.5 - Pri...	2020-01-01...	2020	01	01	04	117.7	183	Unhealthy	138.0	UG/I
5	Mumbai	PM2.5 - Pri...	2020-01-01...	2020	01	01	05	107.8	178	Unhealthy	98.0	UG/I
6	Mumbai	PM2.5 - Pri...	2020-01-01...	2020	01	01	06	108.4	178	Unhealthy	109.0	UG/I
7	Mumbai	PM2.5 - Pri...	2020-01-01...	2020	01	01	07	114.7	182	Unhealthy	121.0	UG/I
8	Mumbai	PM2.5 - Pri...	2020-01-01...	2020	01	01	08	118.3	183	Unhealthy	122.0	UG/I
9	Mumbai	PM2.5 - Pri...	2020-01-01...	2020	01	01	09	126.6	188	Unhealthy	135.0	UG/I

Figure P3.26
Air Quality Data for Mumbai, India, for 2020. Retrieved from AirNow on July 30, 2021. (www.airnow.gov)

CHAPTER

4 Manipulating MATLAB Arrays

Objectives

After reading this chapter, you should be able to:

- Manipulate numeric arrays.
- Extract data from numeric arrays.
- Solve problems with two array variables of different sizes using meshgrid.
- Create and use special numeric arrays.
- Explain the difference between character and string arrays.
- Use techniques to manipulate numeric arrays to analyze and manipulate textual data.

KEY IDEA

In MATLAB the terms array and matrix are often used interchangeably. However, technically an array is only a matrix when it is used for matrix calculations in linear algebra.

INTRODUCTION

The basic MATLAB data type is the **array**, and all information in MATLAB is stored in arrays. In this chapter, we'll explore additional strategies for manipulating arrays, first in arrays designed for numbers and then in arrays designed to hold text.

4.1 MANIPULATING NUMERIC ARRAYS

As you solve more and more complicated problems with MATLAB, you'll find that you will need to combine small arrays into larger arrays, extract information from large arrays, create very large arrays, and use arrays with special properties.

4.1.1 Defining Numeric Arrays

To review, in MATLAB you can define a numeric array by typing in a list of numbers enclosed in square brackets. You can separate the numbers by spaces or by commas, at

KEY IDEA
There are multiple ways to define an array.

your discretion. (You can even combine the two techniques in the same array definition.) To indicate a new row, you can use a semicolon. For example,

```
A = [3.5];
B = [1.5, 3.1]; or B = [1.5 3.1];
C = [-1, 0, 0; 1, 1, 0; 0, 0, 2];
```

You can also define an array by listing each row on a separate line, as in the following set of MATLAB commands:

```
C =   [-1,  0, 0;
        1,  1, 0;
        1, -1, 0;
        0,  0, 2]
```

You don't even need to enter the semicolon to indicate a new row. MATLAB interprets

```
C =    [-1,  0, 0
         1,  1, 0
         1, -1, 0
         0,  0, 2]
```

as a 4×3 array. You could also enter a column in this manner:

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

If there are too many numbers in a row to fit on one line, you can continue the statement on the next line, but a comma and an **ellipsis** (...) are required at the end of the line, indicating that the row is to be continued. You can also use the ellipsis to continue other long assignment statements in MATLAB.

If we want to define array `F` with 10 values, we can use either of the following statements:

```
F = [1, 52, 64, 197, 42, -42, 55, 82, 22, 109]; or
F = [1, 52, 64, 197, 42, -42, ...
       55, 82, 22, 109];
```

MATLAB also allows you to define an array in terms of another array that has already been defined. For example, the statements

```
B = [1.5, 3.1];
S = [3.0, B]
```

return

KEY IDEA
You can create larger arrays by concatenating smaller arrays.

```
S =
    3.0   1.5   3.1
```

Similarly,

```
T = [ 1, 2, 3; S]
```

returns

```
T =
     1   2     3
     3   1.5   3.1
```

Combining arrays in this manner is called **concatenation**.
We can change values in an array, or include additional values, by using an **index** number to specify a particular **element**. This process is called **indexing into an array**. Thus, the command

```
S(2) = -1.0;
```

changes the second value in the array `S` from 1.5 to −1. If we type the array name

```
S
```

into the command window, then MATLAB returns

```
S =
      3.0   -1.0   3.1
```

Similarly, we can also extend an array by defining new elements. If we execute the command

```
S(4) = 5.5;
```

we extend `S` to four elements instead of three. If we define element

```
S(8) = 9.5;
```

`S` will have eight values, and the values of `S(5)`, `S(6)`, and `S(7)` will be set to 0. Thus,

```
S
```

returns

```
S =
      3.0   -1.0   3.1   5.5   0   0   0   9.5
```

KEY IDEA
Index numbers are used to identify the location of an element in an array, to extract the value of an element from an array, or to define new members of an array.

4.1.2 Using the Colon Operator

The colon operator is very powerful in defining new arrays and modifying the existing ones. First, we can use it to define an evenly spaced array. For example,

```
H = 1:8
```

returns

```
H =
      1   2   3   4   5   6   7   8
```

The default spacing is 1. However, when colons are used to separate three numbers, the middle value becomes the spacing. Thus,

```
time = 0.0:0.5:2.0
```

returns

```
time =
      0   0.5000   1.0000   1.5000   2.0000
```

The colon operator can also be used to extract data from arrays, a feature that is very useful in data analysis. When a colon is used in an array reference in place of a specific index number, the colon represents the entire row or column.

Suppose we define `M` as

```
M = [1 2 3 4 5;
     2 3 4 5 6;
     3 4 5 6 7];
```

We can extract column 1 from array M with the command

```
x = M(:, 1)
```

which returns

```
x =
     1
     2
     3
```

We read this syntax as "all the rows in column 1." We can extract any of the columns in a similar manner. For instance,

```
y = M(:, 4)
```

returns

```
y =
     4
     5
     6
```

KEY IDEA
The colon operator makes it easy to extract groups of elements from an array.

and can be interpreted as "all the rows in column 4." Similarly, to extract a row,

```
z = M(1,:)
```

returns

```
z =
     1   2   3   4   5
```

and is read as "row 1, all the columns."

We don't have to extract an entire row or an entire column. The colon operator can also be used to mean "from row to row" or "from column to column." To extract the two bottom rows of M, type

```
w = M(2:3,:)
```

which returns

```
w =
     2   3   4   5   6
     3   4   5   6   7
```

and reads "rows 2 to 3, all the columns." Similarly, to extract just the four numbers in the lower right-hand corner M,

```
w = M(2:3, 4:5)
```

returns

```
w =
     5   6
     6   7
```

and reads "rows 2 to 3 in columns 4 to 5."

In MATLAB, it is valid to have an array that is empty. For example, each of the following statements will generate an empty array:

```
a = [ ];
b = 4:-1:5;
```

Finally, using the array name with a single colon, such as

```
M(:)
```

transforms the array into one long column.

```
M  =
   1
   2
   3
   2
   3
   4
   3
   4
   5
   4
   5
   6
   5
   6
   7
```

This array was formed by first listing column 1, then adding column 2 onto the end, tacking on column 3, and so on. Actually, the computer does not store two-dimensional arrays in a two-dimensional pattern. Rather, it "thinks" of an array as one long list, just like the array `M` at the left. There are two ways you can extract a single value from an array: by using a single index number or by using the row, column notation sometimes called **subscripts**. To find the value in row 2, column 3, use the following commands:

```
M
M =
     1     2     3     4     5
     2     3     4     5     6
     3     4     5     6     7

M(2, 3)
ans =
     4
```

KEY IDEA

You can identify an element using either a single number, or indices representing the row and column (subscripting).

Alternatively, you can use a single index number. The value in row 2, column 3 of `M` is element number 8. (Count down column 1, then down column 2, and finally down column 3 to the correct element.) The associated MATLAB command is

```
M(8)
ans = 4
```

HINT

Use the word "end" to identify the final row or column in an array, even if you don't know how big it is. For example,

```
M(1,end)
```

returns

```
ans =
     5
```

and

```
M(end, end)
```

returns

```
ans =
     7
```

as does

```
M(end)
ans =
     7
```

PRACTICE EXERCISE 4.1

Create MATLAB variables to represent the following arrays, and use them in the exercises that follow. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

$$a = \begin{bmatrix} 12 & 17 & 3 & 6 \end{bmatrix} \quad b = \begin{bmatrix} 5 & 8 & 3 \\ 1 & 2 & 3 \\ 2 & 4 & 6 \end{bmatrix} \quad c = \begin{bmatrix} 22 \\ 17 \\ 4 \end{bmatrix}$$

1. Assign to the variable `x1` the value in the second column of array `a`. This is sometimes represented in mathematics textbooks as element $a_{1,2}$ and could be expressed as `x1 = a1,2`.
2. Assign to the variable `x2` the third column of array `b`.
3. Assign to the variable `x3` the third row of array `b`.
4. Assign to the variable `x4` the values in array `b` along the diagonal (i.e., elements $b_{1,1}$, $b_{2,2}$, and $b_{3,3}$).
5. Assign to the variable `x5` the first three values in array `a` as the first row, and all the values in array `b` as the second through the fourth row.
6. Assign to the variable `x6` the values in array `c` as the first column, the values in array `b` as columns 2, 3, and 4, and the values in array `a` as the last row.
7. Assign to the variable `x7` the value of element 8 in array `b`, using the single-index-number identification scheme.
8. Convert array `b` to a column vector named `x8`.

EXAMPLE 4.1

USING TEMPERATURE DATA

The data collected by the National Weather Service are extensive but are not always organized in exactly the way we would like (see Figure 4.1). Take, for example, the summary of the 1999 Asheville, North Carolina, Climatological Data. We'll use these data to practice manipulating arrays—both extracting elements and recombining elements to form new arrays.

The numeric information has been extracted from the table (Appendix D) and is in an Excel file called *Asheville_1999.xls*. Use MATLAB to confirm that the reported values on the annual row are correct for the mean maximum temperature and the mean minimum temperature, as well as for the annual high temperature and the annual low temperature. Combine these four columns of data into a new array called *temp_data*.

1. State the problem
   Calculate the annual mean maximum temperature, the annual mean minimum temperature, the highest temperature reached during the year, and the lowest temperature reached during the year for 1999 in Asheville, North Carolina.

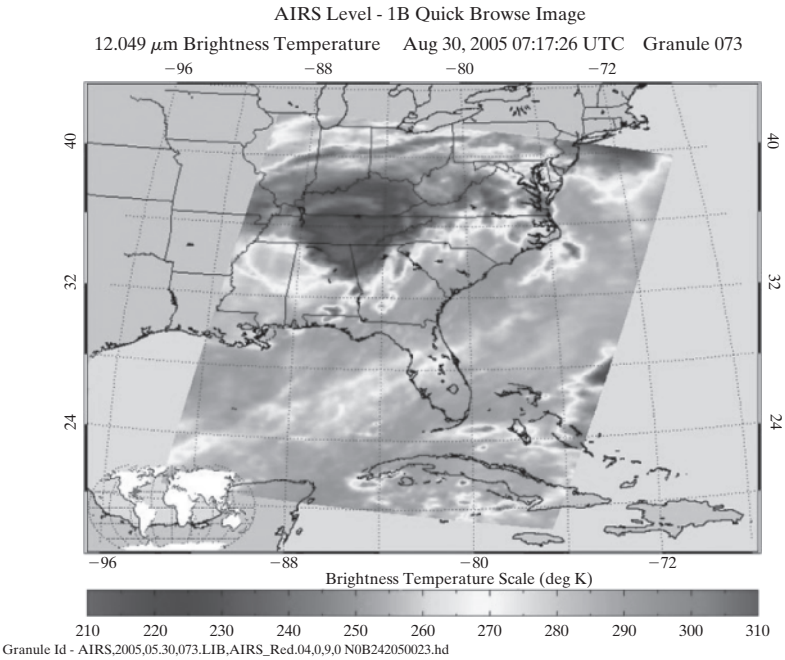


Figure 4.1
Temperature data collected from a weather satellite were used to create this composite false-color image. (NASA)

2. Describe the input and output

Input Import an array from the Excel file *Asheville_1999.xls*.

Output Find the following four values: annual mean maximum temperature
annual mean minimum temperature
highest temperature
lowest temperature

Create an array composed of the mean maximum temperature values, the mean minimum temperature values, the highest monthly temperatures, and the lowest monthly temperatures. Do not include the annual data.

3. Develop a hand example

Using a calculator, add the values in column 2 of the table and divide by 12.

4. Develop a MATLAB solution

First import the data from Excel, then save them in the current directory as *Asheville_1999*. Save the variable *Asheville_1999* as the file *Asheville_1999.mat*. This makes it available to be loaded into the workspace from the program:

```
%%  Example 4.1
% In this example we extract data from a large array, and
% use the data analysis functions to find the mean high and
% mean low temperatures for the year, and to find the high
% temperature and the low temperature for the year
%
clear, clc
%%  load the data array from a file
load asheville_1999
%%  extract the mean high temperatures from the large array
mean_max = asheville_1999(1:12,2);
```

(*continued*)

```
% extract the mean low temperatures from the large array
mean_min = asheville_1999(1:12,3);
% Calculate the annual means
annual_mean_max = mean(mean_max)
annual_mean_min = mean(mean_min)
% extract the high and low temperatures from the large array
high_temp = asheville_1999(1:12,8);
low_temp = asheville_1999(1:12,10);
% Find the max and min temperature for the year
max_high = max(high_temp)
min_low = min(low_temp)
% Create a new array with just the temperature information
new_table =[mean_max, mean_min, high_temp, low_temp]
```

The results are displayed in the command window or the live script editing window.

```
annual_mean_max =
    68.0500
annual_mean_min =
    46.3250
max_high =
    96
min_low =
    9
new_table =
    51.4000    31.5000    78.0000     9.0000
    52.6000    32.1000    66.0000    16.0000
    52.7000    32.5000    76.0000    22.0000
    ...
    67.6000    45.5000    78.0000    28.0000
    62.2000    40.7000    76.0000    26.0000
    53.6000    30.5000    69.0000    15.0000
```

5. Test the solution
   Compare the results against the bottom line of the table from the Asheville, North Carolina, Climatological Survey, found in Appendix C. It is important to confirm that the results are accurate before you start to use any computer program to process data.

4.2 PROBLEMS WITH TWO VARIABLES—USING MESHGRID

All of the calculations we have performed thus far have used only one variable. Of course, most physical phenomena can vary with many different factors. In this section, we consider how to perform the same calculations when the variables are represented by vectors.

Consider the following MATLAB statements:

```
x = 3;
y = 5;
A = x * y
```

Since x and y are scalars, it's an easy calculation: $x * y = 15$, or

```
A =
        15
```

Now, let's see what happens if x is an array and y is still a scalar:

```
x = 1:5;
```

returns five values of x. Because y is still a scalar with only one value (5),

```
A = x * y
```

returns

```
A =
     5    10    15    20    25
```

This is still a review. But what happens if y is now a vector? Then,

```
y = 1:3;
A = x * y
```

returns an error statement:

```
??? Error using *
Incorrect dimensions for matrix multiplication.
```

KEY IDEA
When formulating problems with two variables, the array dimensions must agree.

This error statement reminds us that the asterisk is the operator for **matrix** multiplication, which is not what we want. We want the dot-asterisk operator (`.*`), which will perform an element-by-element multiplication. However, the two vectors, x and y, will need to be the same length for this to work. Thus,

```
y = linspace(1,3,5)
```

creates a new vector y with five evenly spaced elements:

```
y =
    1.0000    1.5000    2.0000    2.5000    3.0000
A = x .* y
A =
     1     3     6    10    15
```

However, although this solution works, the result is probably not what you really want. You can think of the results as the diagonal on an array (see Table 4.1).

What if we want to know the result for element 3 of vector x and element 5 of vector y? This approach obviously doesn't give us all the possible answers. We want a two-dimensional array of answers that corresponds to all the combinations of x and y. In order for the answer A, to be a two-dimensional array, the input vectors must be two-dimensional arrays. MATLAB has a built-in function called `meshgrid` that will help us accomplish this—and x and y don't even have to be the same size.

Table 4.1 Results of an Element-by-Element Calculation

		x				
		1	**2**	**3**	**4**	**5**
y	**1.0**	1				
	1.5		3			
	2.0			6		
	2.5				10	
	3.0			?		15

First, change y back to a three-element vector:

```
y = 1:3;
```

Then, use meshgrid to create a new two-dimensional version of both x and y that we'll call new_x and new_y:

```
[new_x, new_y] = meshgrid(x,y)
```

KEY IDEA
Use the meshgrid function to map two one-dimensional variables into two-dimensional variables of equal size.

The meshgrid command takes the two input vectors and creates two two-dimensional arrays. Each of the resulting arrays has the same number of rows and columns. The number of columns is determined by the number of elements in the first input vector (x), and the number of rows is determined by the number of elements in the second input vector (y). This operation is called **mapping** *the vectors into a two-dimensional array*:

```
new_x =
     1     2     3     4     5
     1     2     3     4     5
     1     2     3     4     5
new_y =
     1     1     1     1     1
     2     2     2     2     2
     3     3     3     3     3
```

Notice that all the rows in new_x are the same, and all the columns in new_y are the same. Now, it's possible to multiply new_x by new_y and get the two-dimensional grid of results we really want:

```
A = new_x.*new_y
A =
     1     2     3     4     5
     2     4     6     8    10
     3     6     9    12    15
```

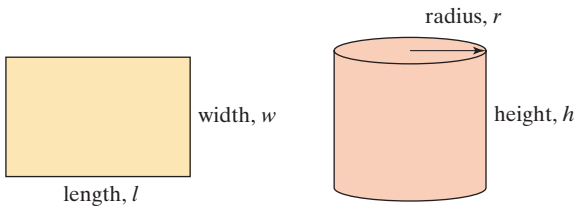
PRACTICE EXERCISE 4.2

USING MESHGRID

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. The area of a rectangle (see Figure 4.2) is length times width (area = length × width). Find the areas of rectangles with lengths of 1, 3, and 5 cm, and with widths of 2, 4, 6, and 8 cm. (You should have 12 answers.)
2. The volume of a circular cylinder is, volume = $\pi r^2 h$. Find the volume of cylindrical containers with radii from 0 to 12 m, and heights from 10 to 20 m. Increment the radius dimension by 3 m and the height by 2 m as you span the two ranges.

Figure 4.2
Dimensions of a rectangle and a circular cylinder.



EXAMPLE 4.2

DISTANCE TO THE HORIZON



Figure 4.3
Junko Tabei, the first woman to summit Mt. Everest. (Keystone Press/Alamy Stock Photo)

In 1975, Junko Tabei (Figure 4.3) became the first woman to summit Mt. Everest and was the first woman to ascend the "Seven Summits." A mountain range on Pluto, Tabei Montes is named in her honor. The mountains on Pluto are almost as high as those on earth, the tallest of which reaches 6200 m and is named after Tenzing Norgay, the famous Nepalese mountaineer who, with Edmund Hillary, was one of the first two individuals to summit Everest. Everest is only slightly taller than Tenzing Montes on Pluto, at 8850 m.

You may have experienced standing on top of a hill or a mountain and feeling like you can see forever. How far can you really see? It depends on the height of the mountain and the radius of the earth, as shown in Figure 4.4. The distance to the horizon is quite different on Pluto than on the earth, because the radius is different for each. Using the Pythagorean theorem, we see that

$$R^2 + d^2 = (R + h)^2$$

and solving for d yields $d = \sqrt{h^2 + 2Rh}$.

From this last expression, find the distance to the horizon on the earth and on Pluto, for mountains from 0 to 8000 m. The radius of the earth is 6378 km, and the radius of Pluto is 1185 km.

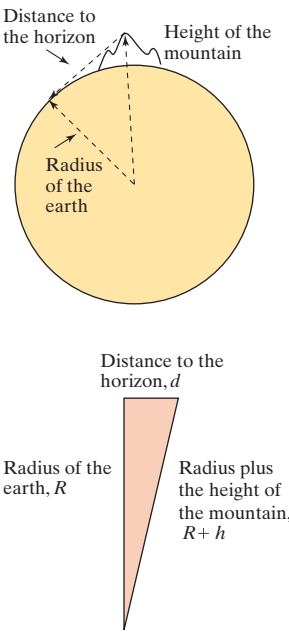


Figure 4.4
Distance to the horizon.

1. State the problem

Find the distance to the horizon from the top of a mountain on Pluto and on the earth.

2. Describe the input and output

Input

Radius of Pluto	1185 km
Radius of the earth	6378 km
Height of the mountains	0 to 8000 m

Output

Distance to the horizon, in kilometers.

3. Develop a hand example

$$d = \sqrt{h^2 + 2Rh}$$

Using the radius of the earth and an 8000-m mountain yields

$$d = \sqrt{(8\text{ km})^2 + 2 \times 6378\text{ km} \times 8\text{ km}} = 319\text{ km}$$

4. Develop a MATLAB solution

```
%% Example 4.2
% Find the distance to the horizon
% Define the height of the mountains
%  in meters
clear, clc
format bank
```

(*continued*)

```
% Define the height vector
height = 0:1000:8000;
% Convert meters to km
height = height/1000;
% Define the radii of Pluto and Earth
radius = [1185  6378];
% Map the radii and heights onto a 2-D grid
[Radius,Height] = meshgrid(radius,height);
% Calculate the distance to the horizon
distance = sqrt(Height.^2 + 2*Height.*Radius)
```

Executing the preceding program returns an array of the distances to the horizon in meters on both Pluto and the earth:

```
distance =
         0         0
   48.69   112.95
   68.88   159.74
   84.37   195.65
   97.45   225.92
  108.97   252.60
  119.40   276.72
  128.99   298.90
  137.93   319.55
```

5. Test the solution
   Compare the MATLAB solution with the hand solution. The distance to the horizon from near the top of Mount Everest (8000 m) is over 300 km and matches the value calculated in MATLAB.

EXAMPLE 4.3

FREE FALL

The general equation for the distance that a freely falling body has traveled (neglecting air friction) is

$$d = \frac{1}{2}gt^2$$

where

d = distance,
g = acceleration due to gravity, and
t = time.

When a satellite orbits a planet, it is in free fall. Many people believe that when the spacecraft enters orbit, it leaves gravity behind; gravity, though, is what keeps the spacecraft in orbit. The spacecraft (or any satellite) is actually falling toward the earth (see Figure 4.5). If it is going fast enough horizontally, it stays in orbit; if it's going too slowly, it hits the ground.

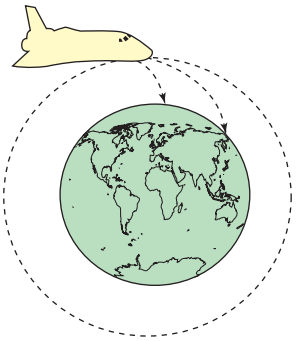




Figure 4.5
NASA astronauts Leland Melvin and Daniel Tani and ESA astronaut Leopold Eyharts onboard the Destiny laboratory of the International Space Station. The ISS is constantly falling toward the earth. (Courtesy of NASA)

The value of the constant g, the acceleration due to gravity, depends on the mass of the planet. On different planets, g has different values (see Table 4.2).

Find how far an object would fall at times from 0 to 100 seconds on each planet in our solar system, and on our moon.

1. State the problem
Find the distance traveled by a freely falling object on planets with different gravities.

2. Describe the input and output
Input Value of g, the acceleration due to gravity, on each of the planets and the moon.

$$\text{Time} = 0 \text{ to } 100 \text{ s}$$

Output Distances calculated for each planet and the moon.

3. Develop a hand example

$$d = 1/2\, gt^2, \text{ so on Mercury at 100 seconds:}$$
$$d = 1/2 \times 3.7 \text{ m/s}^2 \times 100^2 \text{s}^2$$
$$d = 18{,}500 \text{ m}$$

Table 4.2 Acceleration Due to Gravity in Our Solar System

Mercury	$g = 3.7$ m/s^2
Venus	$g = 8.87$ m/s^2
Earth	$g = 9.8$ m/s^2
Moon	$g = 1.6$ m/s^2
Mars	$g = 3.7$ m/s^2
Jupiter	$g = 23.12$ m/s^2
Saturn	$g = 8.96$ m/s^2
Uranus	$g = 8.69$m/s^2
Neptune	$g = 11.0$ m/s^2
Pluto	$g = 0.58$ m/s^2

4. Develop a MATLAB solution

```
%% Example 4.3
% Freefall
clear, clc
format shortg
%% Try the problem first with two planets, and a course
% grid
% Define constants for acceleration due to gravity on Mercury
% and Venus
acceleration_due_to_gravity = [3.7, 8.87];
time=0:10:100;
```

(*continued*)

```
% Define time vector
% Map G and T into 2D arrays
[g,t]=meshgrid(acceleration_due_to_gravity,time);
% Calculate the distances
distance = 1/2*g.*t.^2
```

Executing the preceding program returns the following values of distance traveled on Mercury and on Venus.

```
distance =
           0         0
         185     443.5
         740      1774
        1665    3991.5
        2960      7096
        4625     11087
        6660     15966
        9065     21731
       11840     28384
       14985     35923
       18500     44350
```

5. Test the solution
Compare the MATLAB solution with the hand solution. We can see that the distance traveled on Mercury at 100 seconds is 18,500 m, which corresponds to the hand calculation.

The program included the calculations for just the first two planets and was performed first to work out any programming difficulties. Once we've confirmed that the program works, it is easy to redo with the data for all the planets:

```
%% Redo the problem with all the data
clear, clc
format shortg
% Define constants
acceleration_due_to_gravity = [3.7,  8.87,  9.8,  1.6,...
3.7,  23.12,  8.96,  8.69,  11.0,  0.58];
time=0:10:100;
% Map G and T into 2D arrays
[g,t]=meshgrid(acceleration_due_to_gravity,time);
% Calculate the distances
distance = 1/2*g.*t.^2
```

There are several important things to notice about the results shown in Figure 4.6. First, look at the workspace window—`acceleration_due_to_gravity` is a 1×10 array (one value for each of the planets and the moon), and time is a 1×11 array (11 values of time). However, both g and t are 11×10 arrays—the result of the `meshgrid` operation. The results shown in the command window were formatted with the `format shortg` command to make the output easier to read; otherwise there would have been a common scale factor.

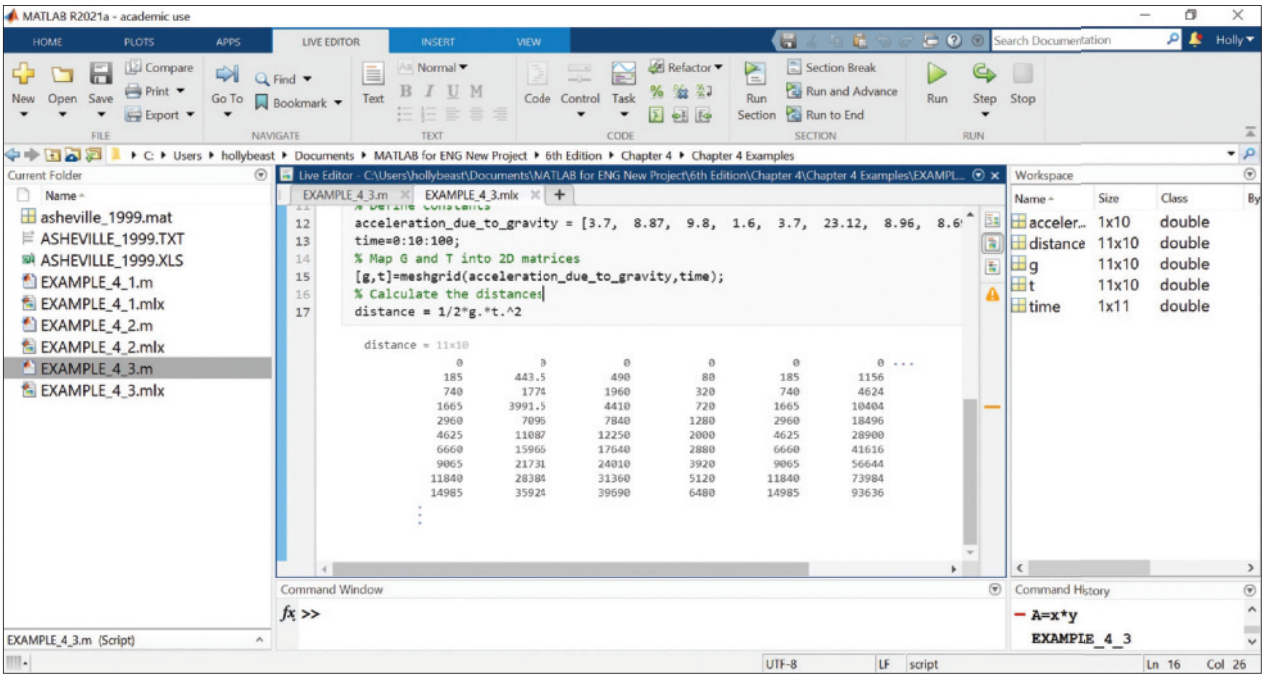


Figure 4.6
Results of the distance calculations for an object falling on each of the planets.

HINT

As you create a MATLAB program in the editing window, either as a script or a live script, you may want to comment out sections of the code which you know work, then uncomment them later. Although you can do this by adding one % at a time to each line, it's easier to highlight the code you'd like to comment out, then select "Comment" on the editing toolstrip. You can also access this menu by right-clicking in the edit window.

4.3 SPECIAL ARRAYS

MATLAB contains a group of functions that generate special arrays, shown in Table 4.3.

4.3.1 Array of Zeros

It is sometimes useful to create an array of all zeros. When the `zeros` function is used with a single scalar input argument, n, a square $n \times n$ array is generated:

```
A = zeros(3)
A =
     0     0     0
     0     0     0
     0     0     0
```

Table 4.3 Functions to Create and Manipulate Arrays

`zeros(m)`	Creates an $m \times m$ array of zeros.	`zeros(3)`
		`ans =`
		`0 0 0`
		`0 0 0`
		`0 0 0`

`zeros(m,n)`	Creates an $m \times n$ array of zeros.	`zeros(2,3)`
		`ans =`
		`0 0 0`
		`0 0 0`

`ones(m)`	Creates an $m \times m$ array of ones.	`ones(3)`
		`ans =`
		`1 1 1`
		`1 1 1`
		`1 1 1`

`ones(m,n)`	Creates an $m \times n$ array of ones.	`ones(2,3)`
		`ans =`
		`1 1 1`
		`1 1 1`

`diag(A)`	Extracts the diagonal of a two-dimensional array `A`.	`A = [123;345;123];`
		`diag(A)`
		`ans =`
		`1`
		`4`
		`3`

	For any vector `A`, creates a square array with `A` as the diagonal. Check the `help` function for other ways the `diag` function can be used.	`A = [123];`
		`diag(A)`
		`ans =`
		`1 0 0`
		`0 2 0`
		`0 0 3`

`fliplr`	Flips an array into its mirror image, from right to left.	`A = [100;020;003];`
		`fliplr(A)`
		`ans =`
		`0 0 1`
		`0 2 0`
		`3 0 0`

`flipud`	Flips an array vertically.	`flipud(A)`
		`ans =`
		`0 0 3`
		`0 2 0`
		`1 0 0`

`magic(m)`	Creates an $m \times n$ "magic" array.	`magic(3)`
		`ans =`
		`8 1 6`
		`3 5 7`
		`4 9 2`

If two scalar arguments are used, the first value specifies the number of rows and the second the number of columns:

```
B = zeros(3,2)
B =
        0    0
        0    0
        0    0
```

KEY IDEA
Use an array of zeros or ones as placeholders for future calculations.

4.3.2 Array of Ones

The `ones` function is similar to the `zeros` function, but creates an array of ones:

```
A = ones(3)
A =
        1    1    1
        1    1    1
        1    1    1
```

As with the `zeros` function, with two inputs we can control the number of rows and columns:

```
B = ones(3,2)
B =
        1    1
        1    1
        1    1
```

The `zeros` and `ones` functions are useful for creating arrays with "placeholder" values that will be filled in later. For example, to create a vector of five numbers, all of which are equal to π, first create a vector of ones:

```
a = ones(1,5)
```

This gives

```
a =
      1    1    1    1    1
```

Then, multiply by π.

```
b = a*pi
```

The result is

```
b =
      3.1416    3.1416    3.1416    3.1416    3.1416
```

The same result could be obtained by adding π to an array of zeros. For example,

```
a = zeros(1,5);
b = a+pi
```

gives

```
b =
      3.1416    3.1416    3.1416    3.1416    3.1416
```

A placeholder array is especially useful in MATLAB programs with a loop structure, because it can reduce the time required to execute the loop.

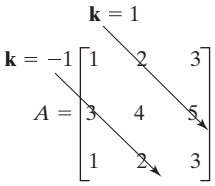


Figure 4.7
Each diagonal in an array can be described by means of the parameter **k**.

4.3.3 Diagonal Arrays

The `diag` function is used to extract the diagonal from an array. For example, for the square array

```
A = [1 2 3; 3 4 5; 1 2 3];
diag(A)
```

extracts the main diagonal and gives the following results:

```
ans =
    1.00
    4.00
    3.00
```

Other diagonals can be extracted by defining a second input, `k`, to `diag`. Positive values of `k` specify diagonals in the upper right-hand corner of the array and negative values specify diagonals in the lower left-hand corner (see Figure 4.7).

Thus, the command

```
diag(A,1)
```

returns

```
ans =
     2
     5
```

If, instead of using a two-dimensional array as input to the `diag` function, a vector is used such as

```
B = [1 2 3];
```

then, MATLAB uses the vector for the values along the diagonal of a new array and fills in the remaining elements with zeros:

```
diag(B)
ans =
     1   0   0
     0   2   0
     0   0   3
```

By specifying a second parameter, we can move the diagonal to any place in the array:

```
diag(B,1)
ans =
     0   1   0   0
     0   0   2   0
     0   0   0   3
     0   0   0   0
```

4.3.4 Magic Squares

MATLAB includes an array function called `magic` that generates an array with unusual properties. At the present time, there does not seem to be any practical use for **magic squares**—except that they are interesting. In a magic square, the sums of the columns are the same, as are the sums of the rows. An example is

```
A = magic(4)
A =
     16    2    3   13
      5   11   10    8
      9    7    6   12
      4   14   15    1
sum(A)
ans =
     34   34   34   34
```

To find the sums of the rows, we can either transpose the array:

```
sum(A')
ans =
  34   34   34   34
```

or request that the function operate on rows by specifying the dimension in the second field

```
sum(A,2)
```

Not only are the sums of all the columns and rows the same, but the sums of the diagonals are also the same. The diagonal from left to right is

```
diag(A)
ans =
  16
  11
   6
   1
```

The sum of the diagonal is the same number as the sums of the rows and columns:

```
sum(diag(A))
ans =
  34
```

Finally, to find the diagonal from lower left to upper right, we first have to "flip" the array, then find the sum of the diagonal:

```
fliplr(A)
ans =
  13    3    2   16
   8   10   11    5
  12    6    7    9
   1   15   14    4
diag(ans)
ans =
  13
  10
   7
   4
sum(ans)
ans =
  34
```



Figure 4.8
"Melancholia" by Albrecht Dürer, 1514. (INTERFOTO/Fine Arts/Alamy Stock Photo)

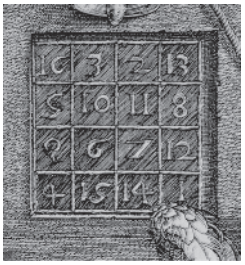


Figure 4.9
Albrecht Dürer included the date of the woodcut (1514) in the magic square. (INTERFOTO/ Fine Arts/Alamy Stock Photo)

Figure 4.8 shows one of the earliest documented examples of a magic square—Albrecht Dürer's woodcut "Melancholia," created in 1514. Scholars believe the square was a reference to alchemical concepts popular at the time. The date 1514 is included in the two middle squares of the bottom row (see Figure 4.9).

Magic squares have fascinated both professional and amateur mathematicians for centuries. For example, Benjamin Franklin experimented with magic squares. You can create magic squares of any size greater than 2×2 in MATLAB. MATLAB's solution is not the only one; other magic squares are possible.

PRACTICE EXERCISE 4.3

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Create a 3×3 array of zeros.
2. Create a 3×4 array of zeros.
3. Create a 3×3 array of ones.
4. Create a 5×3 array of ones.

5. Create a 4×6 array in which all the elements have a value of pi.
6. Use the `diag` function to create an array whose diagonal has values of 1, 2, 3.
7. Create a 10×10 magic square.
   a. Extract the diagonal from this array.
   b. Extract the diagonal that runs from lower left to upper right from this array.
   c. Confirm that the sums of the rows, columns, and diagonals are all the same.

4.4 INTRODUCTION TO CHARACTER AND STRING ARRAYS

KEY IDEA
Text can be stored as either a character or string array in MATLAB.

Although much of what an engineer or scientist will do with MATLAB will revolve around numbers, textual information still plays a big role in programming tasks. For example, text is required for labeling graphs, creating file names, and adding comments to data. Text is stored in arrays, just like numbers and the skills learned to manipulate numeric arrays apply to **character** arrays as well.

MATLAB includes both the character and **string** data type to store textual information. The character data type is best used when each character has a specific meaning. An example suggested by The MathWorks is DNA encoding. Each letter in a DNA sequence has a specific meaning, and the order is important. A *character* array of letters used to represent a strand (or portion of a strand) of DNA makes sense. The *string* data type is used to store blocks of text and is the correct choice for most applications. For example, a patient name or the title of a book should be stored as a string because the entire block of text is what has meaning. When MATLAB was originally created it only included the character data type, so the program was manipulated to make character arrays serve both the function of the character and string data type. Strings are a comparatively recent addition to MATLAB (they were added with release 2016b) and have supplanted character arrays in a wide range of applications as the preferred way of accomplishing a task. However, MATLAB functions have been designed so that they will accept either character or string arrays as input when textual information is required. This makes them "backward" compatible, meaning older code will still work but that you can feel free to update new programs to take advantage of the improvements offered by the implementation of string arrays in your code.

You may be wondering why mention character arrays at all if strings are more commonly used in new coding projects and if they offer advantages over character arrays. There is a rich library of **legacy code**, all of which uses the older character array approach to handling textual data, and it is important to understand how this code works. In addition, most of the vast documentation library, including a plethora of useful examples, uses character arrays to handle textual data, because it was created before string arrays were implemented in MATLAB.

The arrays used to store both character and string data are structurally very similar to numeric arrays, and most of the syntax used to describe numeric arrays also applies to these data types.

4.4.1 Character Arrays

Single quotes are used to define character arrays and to differentiate them from variable names. When we type:

```
H = 'Holly';
```

a 1×5 *character* array is created. Each letter is a separate element of the array. You can index into character arrays just like you index into numeric arrays. For example:

```
     H(5)
ans =
     'y'
```

This line of text

```
J = 'MATLAB is fun';
```

becomes a 1×13 character array. Notice that the spaces between the words are counted as characters. The variables `H` and `J` are shown in the workspace window (Figure 4.10). The name column displays a symbol containing the letters "c|h," which indicates that `H` and `J` are character arrays. Each character array requires 2 bytes of storage space.

Creating a character array with multiple rows is a bit more difficult. Just as with numeric arrays, we can create two-dimensional character arrays only if the number of elements in each row is the same. Thus, a list of names such as the following won't work because each name has a different number of characters:

```
M=['Steven';'Jasmin';'David';'Terrell';'Maria']
Error using vertcat
Dimensions of arrays being concatenated are not consistent.
```

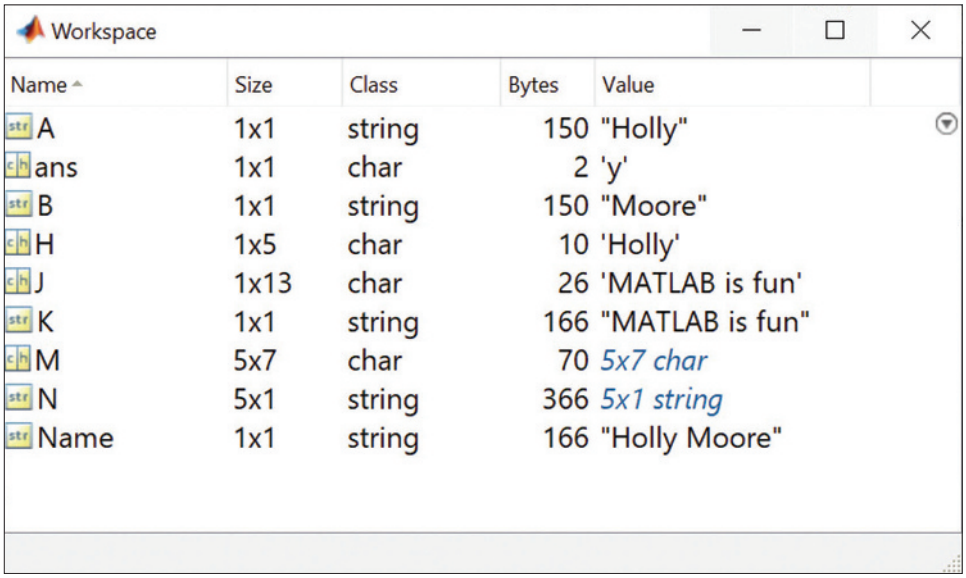


Figure 4.10
MATLAB includes both character and string arrays.

The `char` function "pads" a character array with spaces, so that every row has the same number of elements:

```
M=char('Steven','Jasmin','David','Terrell','Maria')
M =
  5×7 char array
    'Steven '
    'Jasmin '
    'David  '
    'Terrell'
    'Maria  '
```

`M` is a 5×7 character array, as shown in Figure 4.10. Notice that commas are used between each of the character vectors in the `char` function.

Indexing into character arrays works the same way as it does in numeric arrays. For example, to extract the third row from the `M` array type:

```
    M(3,:)
ans =
    'David  '
```

To determine the dimensions of a character array, use the size function . . . just as you did for numeric arrays:

```
size(M)
ans =
  5     7
```

Similarly, the `numel` and `length` functions also behave in the same manner for character arrays as they do for numeric arrays.

PRACTICE EXERCISE 4.4

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Create a character array corresponding to your name, and name it my_name.
2. Use the `char` function to create a character array containing a column of at least five names, and name it `some_students`. The result will be a two-dimensional character array.
3. Use the `size` function to determine the dimensions of the resulting `some_students` array.
4. Use the `numel` function to determine the number of elements in the `some_students` array.
5. Index into the some_students array and extract the third name in the array.

4.4.2 String Arrays

String arrays store entire groups of characters as a single element in an array. You can either convert an existing character array to a string array or create one from scratch. To reformat the M character array, use the string function:

```
N=string(M)
N =
 5×1 string array
  "Steven "
  "Jasmin "
  "David  "
  "Terrell"
  "Maria  "
```

KEY IDEA
String arrays can be concatenated (combined) with the + operator.

Notice in Figure 4.10 that N is a 5×1 array versus the 5×7 array used to store the character array M. String arrays are indicated with the str symbol in the Name column.

One problem with reformatting a character array to create a string array is that extra spaces created by the padding are retained. Strings can be created directly using double quotes, as in:

```
N=["Steven";"Jasmin";"David";"Terrell";"Maria"]
N =
 5×1 string array
  "Steven"
  "Jasmin"
  "David"
  "Terrell"
  "Maria"
```

In either case, the resulting variable is a 5×1 array. Each of the names is a single element in the array, so if we index into N as in:

```
N(2)
```

the result is the entire name:

```
ans =
  "Jasmin"
```

The same rules for combining and accessing numeric and character arrays apply to string arrays. However, strings (but not character arrays) can be concatenated (combined) with the plus sign. Here's a simple example.

```
A= "Holly";
B= "Moore";
Name = A + " " + B
```

which returns

```
Name =
     "Holly Moore"
```

The variable, Name, is 1×1 string array.

Conveniently, the plus operator will also convert other data types to the string data type so that they can be concatenated. This is useful for creating text that contains numbers. For example, this might be the title for a graph, where the variable `Test` is a double representing a test number:

```
Test = 11;
GraphName = "Trial Run " + Test

GraphName =
    "Trial Run 11"
```

The `GraphName` variable can now be used as input to the `title` function used to add titles to plots. Creating graphs is discussed in a later chapter.

Just as for character arrays, the dimensions of string arrays can be determined with the same functions used for numeric arrays. Thus:

```
length(N)
ans =
  5
```

There are five elements, each of which is a string, in the `N` array. If we want to know how many characters are in each string in the array, use `strlength`.

```
strlength(N)
ans =
  6
  6
  5
  7
  5
```

Strings will be used extensively in the upcoming chapters. As a rule, they are easier to work with than character arrays. Unless the MATLAB documentation specifically requires character arrays be used as a function input, strings are probably the better choice. MATLAB functions have been extensively updated to accept both character and string input to allow for backward compatibility and maximum programming flexibility.

PRACTICE EXERCISE 4.5

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Create a string array corresponding to your name, and name it `my_name`.
2. Compare the results when you use both `length` and `strlength` on your `my_name` array.
3. Create a string array containing a column of at least five first names, and name it `some_students`. The result will be a column vector of first names.
4. Use the `size` function to determine the dimensions of the resulting `some_students` array.
5. Create a string array called `last_names` containing a column of last names to match the first names in `some_students`.

6. Combine the first names in `some_students` with the last names in the `last_names` array using the plus (+) operator and call the result `names`. You should put a space between the arrays for readability.
7. Combine the first names in `some_students` with the last names in the `last_names` array using square brackets and call the result `names2`.
8. Use the `size` function to determine the size of the `names` and `names2` arrays. How are they different?
9. Extract the `third` row from the `names` array.
10. Extract the element in the fourth row, second column from the `names2` array.
11. Compare and explain the results when you use the `length` and the `strlength` functions on the `names2` array.

SUMMARY

This chapter concentrated on manipulating arrays, a capability that allows the user to create complicated arrays by combining smaller ones. It also lets you extract portions of an existing array. The colon operator is especially useful for these operations. The colon operator should be interpreted as "all of the rows" or "all of the columns" when used in place of a row or column designation. It should be interpreted as "from _ to _" when it is used between row or column numbers. For example,

```
A(:,2:3)
```

should be interpreted as "all the rows in array `A`, and all the columns from 2 to 3." When used alone as the sole index, as in `A(:)`, it creates an array that is a single column from a two-dimensional representation. The computer actually stores all array information as a list, making both single-index notation and row-column notation useful alternatives for specifying the location of a value in an array.

The `meshgrid` function is extremely useful, since it can be used to map vectors into two-dimensional arrays, making it possible to perform array calculations with vectors of unequal size.

MATLAB contains a number of functions that make it easy to create special arrays:

- `zeros`, which is used to create an array composed entirely of zeros.
- `ones`, which is used to create an array composed entirely of ones.
- `diag`, which can be used to extract the diagonal from an array or, if the input is a vector, to create a square array.
- `magic`, which can be used to create an array with the unusual property that all the rows and columns add up to the same value, as do the diagonals.

In addition, a number of functions were included that allow the user to "flip" the array either from left to right, or from top to bottom.

In addition to numeric arrays, MATLAB includes two data types to store textual information, character, and string arrays. Character arrays store individual characters and are best used when each character has specific meaning. Character arrays are

created with single quotes. String arrays store blocks of text into a single element in an array and are created with double quotes. Indexing into both character and string arrays is accomplished using the same strategies used for numeric arrays. The plus (+) operator can be used to concatenate arrays onto strings, including arrays of different data types. The `strlength` function can be used to find the length of individual string elements in a string array. Strings offer significant advantages over character arrays and should be used in most applications unless there is a specific need to use a character array. MATLAB functions that require textual input accept either character or string arrays.

MATLAB SUMMARY

The following MATLAB summary lists and briefly describes all of the special characters, commands, and functions that were defined in this chapter.

Special Characters	
`:`	colon operator
`...`	ellipsis, indicating continuation on the next line
`[ ]`	empty array
`+`	concatenation operator for strings
`' '`	character information
`" "`	string information
`[ ]`	array constructor

Commands and Functions	
`char`	creates a padded character array
`diag`	extracts the diagonal from an array
`fliplr`	flips an array into its mirror image, from left to right
`flipud`	flips an array vertically
`length`	determines the largest dimension in an array
`magic`	creates a "magic" square
`meshgrid`	maps vectors into a two-dimensional array
`numel`	determines the number of elements in an array
`ones`	creates an array of ones
`string`	creates a string array
`strlength`	determines number of characters in each element in a string array
`size`	determines the dimensions of an array
`zeros`	creates an array of zeros

KEY TERMS

array
character
concatenation
element
ellipsis
index
indexing
legacy code
magic squares
mapping
matrix
string
subscripts

PROBLEMS

Manipulating Arrays

4.1 Create the following arrays, and use them in the exercises that follow:

$$\mathbf{a} = \begin{bmatrix} 15 & 3 & 22 \\ 3 & 8 & 5 \\ 14 & 3 & 82 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 5 \\ 6 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 12 & 18 & 5 & 2 \end{bmatrix}$$

(a) Create an array called `d` from the third column of array `a`.
(b) Combine array `b` and array `d` to create array `e`, a two-dimensional array with three rows and two columns.
(c) Combine array `b` and array `d` to create array `f`, a one-dimensional array with six rows and one column.
(d) Create an array `g` from array `a` and the first three elements of array `c`, with four rows and three columns.
(e) Create an array `h` with the first element equal to $a_{1,3}$, the second element equal to $c_{1,2}$, and the third element equal to $b_{2,1}$.

4.2 Load the file *thermo_scores.dat* provided by your instructor, or enter the array shown in Table P4.2 and name it `thermo_scores`. (Enter only the numbers.)

Table P4.2 Thermodynamics Test Scores

Student No.	Test 1	Test 2	Test 3
1	68	45	92
2	83	54	93
3	61	67	91
4	70	66	92
5	75	68	96
6	82	67	90
7	57	65	89
8	5	69	89
9	76	62	97
10	85	52	94
11	62	34	87
12	71	45	85
13	96	56	45
14	78	65	87
15	76	43	97
16	68	76	95
17	72	65	89
18	75	67	88
19	83	68	91
20	93	90	92

(a) Extract the scores and student number for student 5 into a row vector named `student_5`.
(b) Extract the scores for Test 1 into a column vector named `test_1`.
(c) Find the standard deviation and variance for each test.
(d) Assuming that each test was worth 100 points, find each student's final total score and final percentage. (Be careful not to add in the student number.)
(e) Create a table that includes the final percentages and the scores from the original table.
(f) Sort the array on the basis of the final percentage, from high to low (in descending order), keeping the data in each row together. (You may need to consult the `help` function to determine the proper syntax.)

4.3 Consider the thermocouple data in Table P4.3:

Table P4.3 Thermocouple Data

Time (hr)	Thermocouple 1 °F	Thermocouple 2 °F	Thermocouple 3 °F
0	84.3	90.0	86.7
2	86.4	89.5	87.6
4	85.2	88.6	88.3
6	87.1	88.9	85.3
8	83.5	88.9	80.3
10	84.8	90.4	82.4
12	85.0	89.3	83.4
14	85.3	89.5	85.4
16	85.3	88.9	86.3
18	85.2	89.1	85.3
20	82.3	89.5	89.0
22	84.7	89.4	87.3
24	83.6	89.8	87.2

(a) Create a column vector named `times` going from 0 to 24 in 2-hour increments.
(b) Your instructor may provide you with the thermocouple temperatures in a file called *thermocouple.dat*, or you may need to create an array named `thermocouple` yourself by typing in the data.
(c) Combine the `times` vector you created in part (a) with the data from `thermocouple` to create an array corresponding to the table in this problem.
(d) Recall that both the `max` and `min` functions can return not only the maximum values in a column, but also the element number where those values occur. Use this capability to determine the values of `times` at which the maxima and minima occur in each column.

4.4 Suppose that a file named *sensor.dat* contains information collected from a set of sensors. Your instructor may provide you with this file, or you may need to enter it by hand from the data in table P4.4.

Each row contains a set of sensor readings, with the first row containing values collected at 0 seconds, the second row containing values collected at 0.5 seconds, and so on.

(a) Read the data file and print the number of sensors and the number of seconds of data contained in the file. Use the `size` function—don't just count the two numbers.

Table P4.4 Sensor Data

Time (s)	Sensor 1	Sensor 2	Sensor 3	Sensor 4	Sensor 5
0.0000	70.6432	68.3470	72.3469	67.6751	73.1764
0.5000	73.2823	65.7819	65.4822	71.8548	66.9929
1.0000	64.1609	72.4888	70.1794	73.6414	72.7559
1.5000	67.6970	77.4425	66.8623	80.5608	64.5008
2.0000	68.6878	67.2676	72.6770	63.2135	70.4300
2.5000	63.9342	65.7662	2.7644	64.8869	59.9772
3.0000	63.4028	68.7683	68.9815	75.1892	67.5346
3.5000	74.6561	73.3151	59.7284	68.0510	72.3102
4.0000	70.0562	65.7290	70.6628	63.0937	68.3950
4.5000	66.7743	63.9934	77.9647	71.5777	76.1828
5.0000	74.0286	69.4007	75.0921	77.7662	66.8436
5.5000	71.1581	69.6735	62.0980	73.5395	58.3739
6.0000	65.0512	72.4265	69.6067	79.7869	63.8418
6.5000	76.6979	67.0225	66.5917	72.5227	75.2782
7.0000	71.4475	69.2517	64.8772	79.3226	69.4339
7.5000	77.3946	67.8262	63.8282	68.3009	71.8961
8.0000	75.6901	69.6033	71.4440	64.3011	74.7210
8.5000	66.5793	77.6758	67.8535	68.9444	59.3979
9.0000	63.5403	66.9676	70.2790	75.9512	66.7766
9.5000	69.6354	63.2632	68.1606	64.4190	66.4785

(b) Find both the maximum value and the minimum value recorded on each sensor. Use MATLAB to determine at what times they occurred.

(c) Find the mean and standard deviation for each sensor and for all the data values collected. Remember, column 1 does not contain sensor data; it contains time data.

4.5 The American National Oceanic and Atmospheric Administration (NOAA) measures the intensity of a hurricane season with the accumulated cyclone energy (ACE) index. The ACE for a season is the sum of the ACE for each tropical storm with winds exceeding 35 knots (65 km/h). The maximum sustained winds (measured in knots) in the storm are measured or approximated every six hours. The values are squared and summed over the duration of the storm. The total is divided by 10,000, to make the parameter easier to use.

$$\text{ACE} = \frac{\sum v_{\max}^2}{10^4}$$

This parameter is related to the energy of the storm, since kinetic energy is proportional to velocity squared. However, it does not take into account the size of the storm, which would be necessary for a true total energy estimate. Reliable storm data have been collected in the Atlantic Ocean since 1950 and are shown in table P4.5. These data may also be available to you from

your instructor as an EXCEL worksheet, *ace.xlsx*, and was extracted from the *Accumulated Cyclone Energy* article in Wikipedia. (http://en.wikipedia.org/wiki/Accumulated_cyclone_energy). It was collected by the National Oceanic and Atmospheric Administration (http://www.aoml.noaa.gov/hrd/tcfaq/E11.html).

Table P4.5 Atlantic Basin Hurricane Seasons, 1950–2016

Year	ACE Index	# Tropical Storms	# Hurricanes Cat. 1–5	# Major Hurricanes Cat. 3–5
1950	211.2825	16	11	6
1951	126.325	12	8	3
1952	69.08	11	5	2
1953	98.5075	14	7	3
1954	110.88	16	7	3
1955	158.17	13	9	4
1956	56.6725	12	4	1
1957	78.6625	8	3	2
1958	109.6925	12	7	3
1959	77.1075	14	7	2
1960	72.9	8	4	2
1961	205.395	12	8	5
1962	35.5675	7	4	0
1963	117.9325	10	7	3
1964	169.7675	13	7	5
1965	84.33	10	4	1
1966	145.2175	11	7	3
1967	121.705	8	6	1
1968	45.0725	8	5	0
1969	165.7375	18	12	5
1970	40.18	11	5	2
1971	96.5275	13	6	1
1972	35.605	7	3	0
1973	47.85	8	4	1
1974	68.125	11	4	2
1975	76.0625	9	6	3
1976	84.1725	10	6	2
1977	25.3175	6	5	1
1978	63.2175	12	5	2
1979	92.9175	9	6	2
1980	148.9375	11	9	2
1981	100.3275	12	7	3
1982	31.5025	6	2	1
1983	17.4025	4	3	1
1984	84.295	13	5	1
1985	87.9825	11	7	3
1986	35.7925	6	4	0

(*continued*)

Table P4.5 (continued)

Year	ACE Index	# Tropical Storms	# Hurricanes Cat. 1–5	# Major Hurricanes Cat. 3–5
1987	34.36	7	3	1
1988	102.9925	12	5	3
1989	135.125	11	7	2
1990	96.8025	14	8	1
1991	35.5375	8	4	2
1992	76.2225	7	4	1
1993	38.665	8	4	1
1994	32.02	7	3	0
1995	227.1025	19	11	5
1996	166.1825	13	9	6
1997	40.9275	8	3	1
1998	181.7675	14	10	3
1999	176.5275	12	8	5
2000	119.1425	15	8	3
2001	110.32	15	9	4
2002	67.9925	12	4	2
2003	176.84	16	7	3
2004	226.88	15	9	6
2005	250.1275	28	15	7
2006	78.535	10	5	2
2007	73.885	15	6	2
2008	145.7175	16	8	5
2009	52.58	9	3	2
2010	165.4825	19	12	5
2011	126.3025	19	7	4
2012	132.6325	19	10	2
2013	36.12	14	2	0
2014	66.725	8	6	2
2015	62.685	11	4	2
2016	141.2525	15	7	4
2017	224.8775	17	10	6
2018	132.5825	15	8	2
2019	132.2025	18	6	3
2020	180.3725	30	14	7
2021*	142.365	21	7	4

*As of November 27, 2021

(a) Import the data into MATLAB, and name the array `ace_data`, using the `readTable` function.

(b) Extract the data from each column, into individual arrays. You should have arrays named

- `years`
- `ace`
- `tropical_storms`
- `hurricanes`
- `major_hurricanes`

(c) Use the `max` function to determine which year had the highest
- ACE value
- Number of tropical storms
- Number of hurricanes
- Number of major hurricanes

(d) Determine the mean and the median values for each column in the array, except for the year.

(e) Use the `sortrows` function to rearrange the `ace_data` array based on the ACE value, sorted from high to low.

The data presented in this problem are updated regularly.

4.6 ACE data similar to those presented in Problem 4.5 are collected by NOAA for the eastern and central Pacific oceans, and for the northern Indian ocean. The data can be found online from a number of sources including Wikipedia. Repeat 4.5 using the Pacific data.

4.7 The data used to calculate the ACE values mainly come from satellite imaging, and are verified by actual measurements taken on the ground and by instruments onboard hurricane reconnaissance planes. ACE values are regularly updated, based on improved analysis techniques. Historical information has allowed researchers to estimate ACE values for the Atlantic basin back to 1851. These data are available from the North Atlantic hurricane database (HURDAT). You can find it on Wikipedia, or directly from NOAA, or your instructor may have a copy of a spreadsheet, ACE_Data_expanded.xlsx containing the information. Repeat problem 4.5 using the expanded database (from 1851 to 2021).

Problems with Two Variables

4.8 The area of a triangle is, area $= \frac{1}{2}$base $\times$ height (see Figure P4.8). Find the area of a group of triangles whose base varies from 0 to 10 m, and whose height varies from 2 to 6 m. Choose an appropriate spacing for your calculational variables. Your answer should be a two-dimensional array.

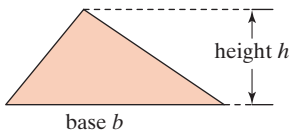


Figure P4.8
The area of a triangle.

4.9 A barometer (see Figure P4.9) is used to measure atmospheric pressure and is filled with a high-density fluid. In the past, mercury was used, but because of its toxic properties it has been replaced with a variety of other fluids. The pressure P measured by a barometer is the height of the fluid column, h, times the density of the liquid, ρ, times the acceleration due to gravity, g, or

$$P = h\rho g$$

This equation could be solved for the height:

$$h = \frac{P}{\rho g}$$

Find the height to which the liquid column will rise for pressures from 0 to 100 kPa for two different barometers. Assume that the first uses mercury, with a density of 13.56 g/cm^3 (13,560 kg/m^3) and the second uses water, with a density of 1.0 g/cm^3 (1000 kg/m^3). The acceleration due to gravity is 9.81 m/s^2. Before you start calculating, be sure to check the units in this calculation. The metric measurement of pressure is a pascal (Pa), equal to 1 kg/ms^2. A kPa is 1000 times as big as a Pa. Your answer should be a two-dimensional array.

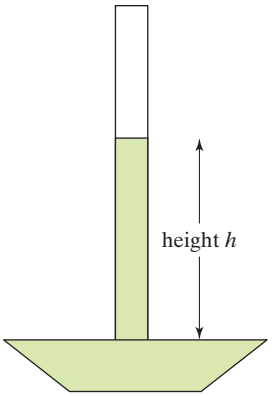


Figure P4.9
Barometric pressure is directly proportional to the height of the liquid column.

4.10 The ideal gas law, $Pv = RT$, describes the behavior of many gases. When solved for v (the specific volume, m^3/kg), the equation can be written

$$v = \frac{RT}{P}$$

Find the specific volume for air, for temperatures from 100 to 1000 K and for pressures from 100 kPa to 1000 kPa. The value of R for air is 0.2870 kJ/(kg K). In this formulation of the ideal gas law, R is different for every gas. There are other formulations in which R is a constant, and the molecular weight of the gas must be included in the calculation. You'll learn more about this equation in chemistry classes and thermodynamics classes. Your answer should be a two-dimensional array.

Special Arrays

4.11 Create an array of zeros the same size as each of the arrays `a`, `b`, and `c` from 4.1. (Use the `size` function to help you accomplish this task.)

4.12 Create a 6×6 magic square.
- **(a)** What is the sum of each of the rows?
- **(b)** What is the sum of each of the columns?
- **(c)** What is the sum of each of the diagonals?

4.13 Extract a 3×3 array from the upper left-hand corner of the magic square you created in 4.11. Is this also a magic square?

4.14 Create a 5×5 magic square named `a`.
- **(a)** Is **a** times a constant such as 2 also a magic square?
- **(b)** If you square each element of `a`, is the new array a magic square?
- **(c)** If you add a constant to each element, is the new array a magic square?
- **(d)** Create a 10×10 array out of the following components (see Figure P4.14):
  - The array `a`
  - 2 times the array `a`
  - An array formed by squaring each element of `a`
  - 2 plus the array `a`

  Is your result a magic square? Does the order in which you arrange the components affect your answer?

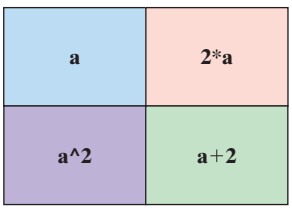


Figure P4.14
Create an array out of other arrays.

4.15 Albrecht Dürer's magic square (see Figure 4.9) is not exactly the same as the 4×4 magic square created with the command

```
magic(4)
```

- **(a)** Recreate Dürer's magic square in MATLAB by rearranging the columns.
- **(b)** Prove that the sum of all the rows, columns, and diagonals is the same.

4.16 One common use for strings is to create meaningful file names or variable names. Create a 3×1 string array called `Dates` containing the dates

June_10
July_12
Aug_15

Now use the + operator to create a new string array to use as file names for the data collected on those days:

```
"Data_Collected_" + Dates
```

4.17 Create a string array to use as a title for a graph. The title should have two lines of text and should therefore be a 2×1 array.

4.18 Use `strlength` to determine the number of characters in each line of the titles created in the previous problem.

4.19 MATLAB contains example data in a file called *patients.mat* with which to experiment. Use it or a similar file to complete the following tasks.

(a) Load the `LastName` variable stored in the file into your workspace with the following command

```
load patients LastName
```

(b) The `LastName` variable is a cell array that stores 100 character arrays. Convert it to a padded character array called `Name_char` with the `char` function.

(c) Convert the `LastName` variable called `Name_string` to a string array with the `string` function.

(d) Use the `size` function to determine the dimensions of each of the variables created in parts **b** and **c** and compare the results.

(e) Use the `sort` function to sort each of the arrays. Suppress the results so they don't print to your screen but examine them in the array editor. (Double-click on the variable name.)

(f) Use the `sortrows` function to sort each of the arrays. How are the results different from part **e**?

CHAPTER

5 Plotting

Objectives

After reading this chapter, you should be able to:

- Create and label two-dimensional plots.
- Adjust the appearance of your plots.
- Divide the plotting window into a tiled chart layout, where multiple plots can be displayed.
- Create three-dimensional plots.
- Use the interactive MATLAB plotting tools.

INTRODUCTION

Large tables of data are difficult to interpret. Engineers use graphing techniques to make the information easier to understand. With a graph, it is easy to identify trends, pick out highs and lows, and isolate data points that may be measurement or calculation errors. Graphs can also be used as a quick check to determine whether a computer solution is yielding expected results and to help interpret results of computer calculations.

5.1 TWO-DIMENSIONAL PLOTS

The most common plot used to display data is the ***x–y* plot**. A set of **ordered pairs** is used to identify points on a two-dimensional graph; the points are then connected by straight lines. The values of x and y may be measured or calculated. Generally, the **independent variable** is given the name x and is plotted on the x-axis, and the **dependent variable** is given the name y and is plotted on the y-axis.

5.1.1 Basic Plotting

Simple x–y Plots

Once vectors of x-values and y-values have been defined, MATLAB makes it easy to create plots. Suppose a set of time versus distance data were obtained through measurement.

We can store the time values in a vector called `x` (the user can define any convenient name) and the distance values in a vector called `y`:

```
x = [0:2:18];
y = [0 0.33 4.13 6.29 6.85 11.19 13.19 13.96 16.33 18.17];
```

To plot these points, use the `plot` command, with `x` and `y` as arguments:

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

Time (s)	Distance (ft)
0	0
2	0.33
4	4.13
6	6.29
8	6.85
10	11.19
12	13.19
14	13.96
16	16.33
18	18.17

A graphics window automatically opens, which MATLAB names Figure 1. The resulting plot is shown in Figure 5.1(a). Slight variations in scaling of the plot may occur, depending on the size of the graphics window. The area plot is quite similar, the only exception being that the area under the curve is filled in. The code

```
area(x,y)
```

results in the plot shown in Figure 5.1(b).

The graphs shown in this chapter include the entire figure window, so that you can see how they look as they are generated during a MATLAB session.

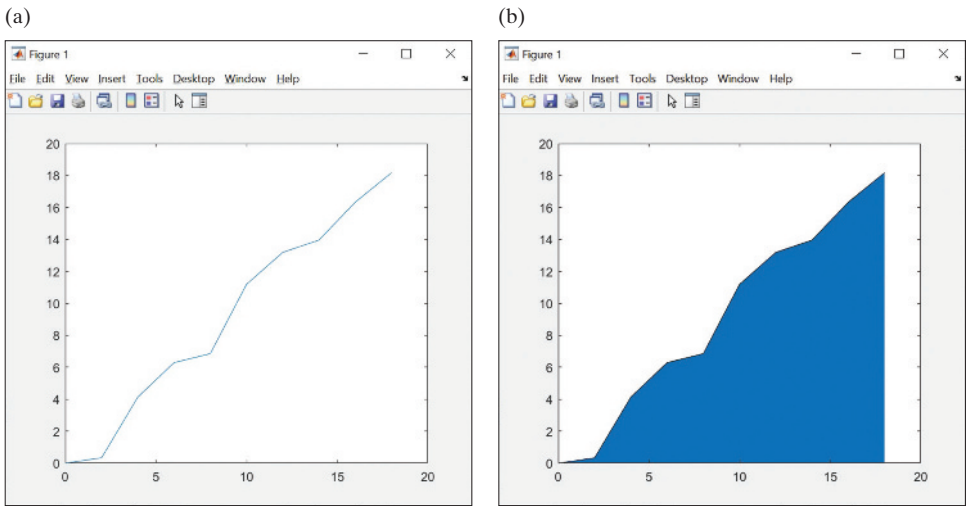


Figure 5.1
Simple plot of time versus distance created in MATLAB. (a) The data are represented as an *x*–*y* plot. (b) The data are represented as an area plot.

Later in the chapter, you will learn how to save the plot in a format more appropriate for inclusion in reports and other documents.

An area plot is used when the area under the curve has some physical meaning. For example, in a piston-cylinder device the work done by the moving piston is described as:

$$W = \int_1^2 P\,dV$$

If you plot V vs P (volume on the x-axis and pressure on the y-axis), the area under the curve from condition 1 to condition 2 is the work, W.

Titles, Labels, and Grids

Good engineering practice requires plots always include axis labels and a title which can be added with the following code:

```
plot(x,y)
title("Laboratory Experiment 1")
xlabel("Time, sec")
ylabel("Distance, ft")
grid on
```

These commands generate the plot in Figure 5.2. An optional grid was also included. As with any MATLAB commands, they could also be combined onto one or two lines to save space.

```
plot(x,y),title("Laboratory Experiment 1")
xlabel("Time, sec"),ylabel("Distance, ft"),grid on
```

As you type the preceding commands into MATLAB, notice that the text color changes to red when you enter a double quote ("). This alerts you that you are starting a string. The color changes to purple when you type the final double quote ("), indicating that you have completed the string. Paying attention to these visual clues will help you avoid coding mistakes.

KEY IDEA
Always include a meaningful title on a plot.

KEY IDEA
Always include units on axis labels.

HINT

Single versus Double Quotes

If you've used MATLAB before, you probably are familiar with using single instead of double quotes for inputs to functions like `title` and `xlabel`. In fact if you check the MATLAB documentation, you'll find examples of both approaches. Before release 2016b, the only option was to use single quotes, which denote a character array. Double quotes indicate a string, which more accurately describes the text input to these functions. Character arrays should really only be used when each character has a particular meaning. An example found in the MATLAB documentation is an array of characters describing a strand of DNA. A group of characters that has meaning when taken as the group should be stored as a string. Details about the differences between character arrays and strings are discussed in other chapters. Most MATLAB functions that accept text will accept it as a string array (with double quotes) or as a character array (with single quotes). For now, either approach will work, but there are real advantages to using a string (double quotes) in any new code you write.

If you are working in the command window, the graphics window will open on top of the other windows (see Figure 5.2). To continue working, either click in the command window, or minimize the graphics window. You can also resize the graphics window to whatever size is convenient for you, or add it to the MATLAB desktop by selecting the docking arrow underneath the exit icon in the upper right-hand corner of the figure window.

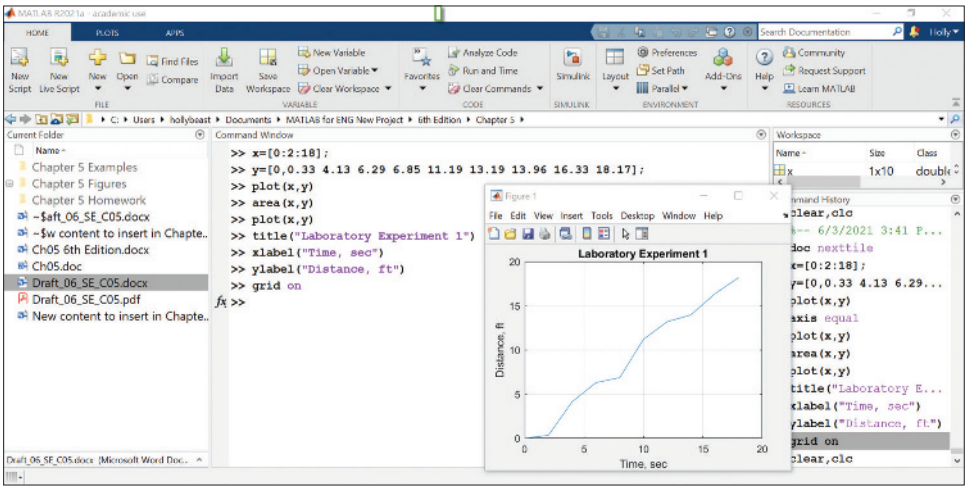


Figure 5.2
The graphics window opens on top of the command window. You can resize it to a convenient shape or dock it with the MATLAB desktop. Adding a grid, title, and axis labels makes a plot easier to interpret.

HINT

Once you click in the command window, the figure window is hidden behind the current window. To see the changes to your figure, select the figure from the Windows task bar at the bottom of the screen.

HINT

You must create a graph *before* you add the title and labels. If you specify the title and labels first, they are erased when the plot command executes.

HINT

If you use a single quote to identify the input to `xlabel`, `ylabel`, and `title` commands, MATLAB interprets an apostrophe (as in the word *it's*) as the end of the character array. Entering the single quote twice, as in `xlabel ('Holly"s Data')`, will allow you to use apostrophes in your text. (Don't use a double quote, which is a different character.) Using double quotes to identify the input string avoids this problem and is a better solution, as in `xlabel ("Holly's Data")`.

Creating Multiple Plots

One disadvantage to working in a script M-file is that when you request a plot, and then continue with more computations, MATLAB will generate and display the graphics window and then return immediately to execute the rest of the commands in the program. If you request a second plot, the graph you created will be overwritten.

This is exactly what happens when you work in the command window. There are three possible solutions to this problem:

- Use sections to execute your program interactively, one section at a time—making sure that each section generates no more than one figure. This allows you to view the plot before it is overwritten.
- Run your entire program using the `pause` command to temporarily halt the execution so that you can examine the figure. The `pause` command stops the program execution until any key is pressed. If you want to pause for a specified number of seconds, use the `pause(n)` command, which will cause execution to pause for `n` seconds before continuing.
- Create multiple figure windows, using the `figure` function.

Issuing the `figure` command opens a new figure window. The next time you request a plot, it will be displayed in this new window. For example,

```
figure(2)
```

opens a window named "Figure 2," which then becomes the window used for subsequent plotting. Executing `figure` without an input parameter causes a new window to open, numbered consecutively one higher from the current window. For example, if the current figure window is named "Figure 2," executing `figure` will cause "Figure 3" to open. The commands used to create a simple plot are summarized in Table 5.1.

Using a live script (an MLX-file) avoids many of these problems. Every time the `plot` function is executed in a live script, a new figure is created without the need to issue the `figure` command, as shown in the simple example illustrated in Figure 5.3. To open a stand-alone figure window, a user simply selects the figure and follows the interactive prompts.

Plots with More than One Line

Plots with more than one line can be created in several ways. By default, the execution of a second `plot` statement will replace the first plot. However, you can layer plots on

Table 5.1 Basic Plotting Functions

`plot`	Creates an *x*–*y* plot	`plot(x,y)`
`title`	Adds a title to a plot	`title("My Graph")`
`xlabel`	Adds a label to the *x*-axis	`xlabel("Independent Variable")`
`ylabel`	Adds a label to the *y*-axis	`ylabel("Dependent Variable")`
`grid`	Adds a grid to the graph	`grid`
	When issued alone grid toggles between on and off	`grid on` `grid off`
`grid minor`	Adds minor grid lines	`grid minor`
`pause`	Pauses the execution of the program, allowing the user to view the graph	`pause`
`figure`	Determines which figure will be used for the current plot. When issued without an input argument, it adds the next consecutive figure.	`figure` `figure(2)`
`hold`	Freezes the current plot, so that an additional plot can be overlaid	`hold on` `hold off`

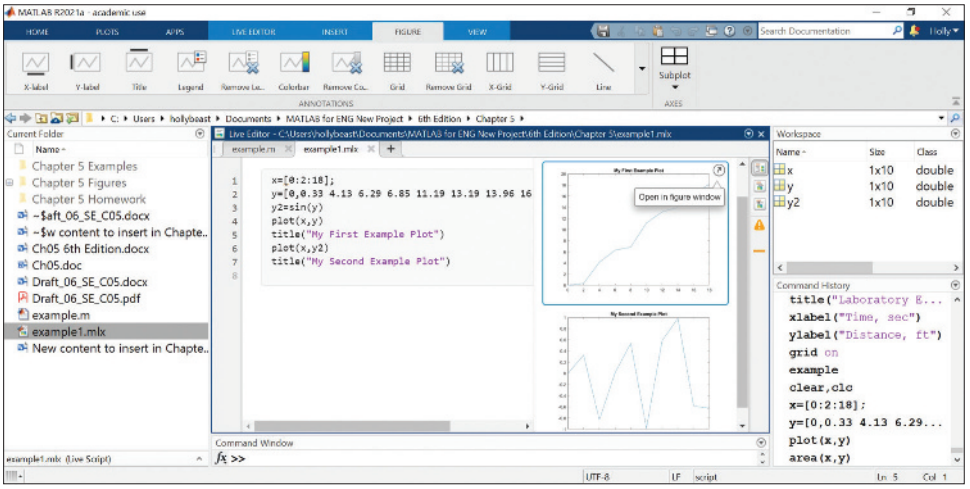


Figure 5.3
Plotting tasks are easier in live scripts. A new figure is created every time the `plot` function is executed unless the `hold on` command is issued.

top of one another by using the `hold on` command. Execute the following statements to create a plot with both functions plotted on the same graph, as shown in Figure 5.4:

```
x = 0:pi/100:2*pi;
y1 = cos(x*4);
plot(x,y1)
y2 = sin(x);
hold on;
plot(x,y2)
```

Semicolons are optional for both the `plot` statement and the `hold on` statement. MATLAB will continue to layer the plots until the `hold off` command is executed:

```
hold off
```

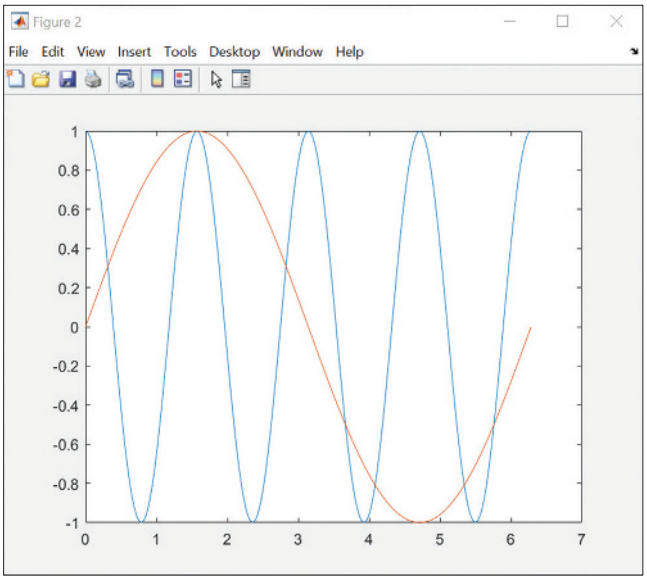


Figure 5.4
The `hold on` command can be used to layer plots onto the same figure.

KEY IDEA
The most common plot used in engineering is the *x*–*y* scatter plot.

Another way to create a graph with multiple lines is to request both lines in a single `plot` command. MATLAB interprets the input to `plot` as alternating **x** and **y** vectors, as in

```
plot(X1, Y1, X2, Y2)
```

where the variables `X1`, `Y1` form an ordered set of values to be plotted, and `X2`, `Y2` form a second ordered set of values. Using the data from the previous example,

```
plot(x, y1, x, y2)
```

produces the same graph as Figure 5.4.

If the `plot` function is called with a single array argument, MATLAB draws a separate line for each column of the array. The *x*-axis is labeled with the row index vector, 1:*k*, where *k* is the number of rows in the array. This produces an evenly spaced plot, called a **line plot**. If `plot` is called with two arguments, one a vector and the other an array, MATLAB successively plots a line for each row in the array. For example, we can combine `y1` and `y2` into a single array and plot against `x`:

```
Y = [y1; y2];
plot(x,Y)
```

This also creates the same plot as Figure 5.4.

Here's another more complicated example:

```
X = 0:pi/100:2*pi;
Y1 = cos(X)*2;
Y2 = cos(X)*3;
Y3 = cos(X)*4;
Y4 = cos(X)*5;
Z = [Y1; Y2; Y3; Y4];
plot(X, Y1, X, Y2, X, Y3, X, Y4)
```

This code produces the same result (see Figure 5.5) as

```
plot(X, Z)
```

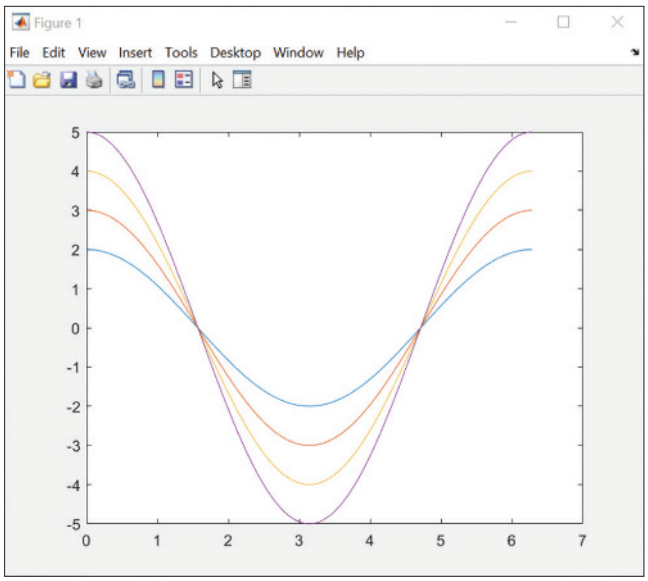


Figure 5.5
MATLAB allows the user to plot multiple lines on the same graph.

A function of two independent variables, the `peaks` function produces sample data that are useful for demonstrating certain graphing functions. (The data are created by scaling and translating Gaussian distributions.) Calling `peaks` with a single argument `n` will create an $n \times n$ array. We can use `peaks` to demonstrate the power of using an array argument in the `plot` function. The command

```
plot(peaks(100))
```

results in the impressive looking graph in Figure 5.6. The input to the `plot` function created by `peaks` is a 100×100 array. Notice that the x-axis goes from 1 to 100, the index numbers of the data. You undoubtedly can't tell, but there are 100 lines drawn to create this graph—one for each column.

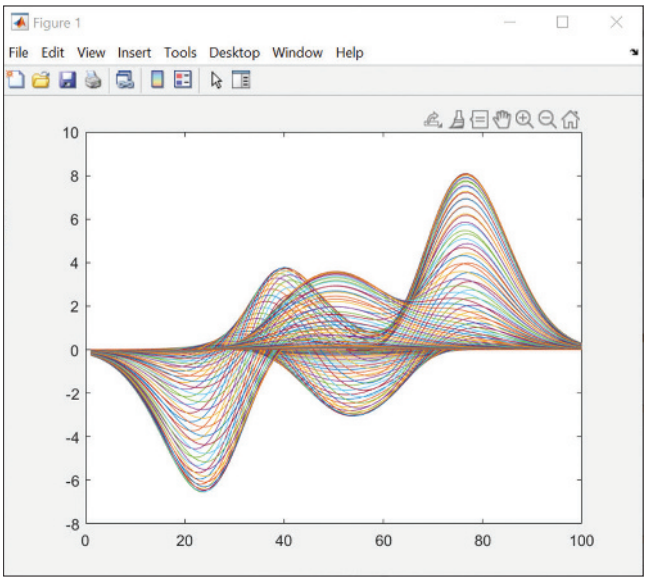


Figure 5.6
The `peaks` function, plotted with a single argument in the `plot` command.

5.1.2 Line, Color, and Mark Style

You can change the appearance of your plots by selecting user-defined line styles and line colors, and by choosing to show the data points on the graph with user-specified mark styles, specified in a string immediately after the independent and dependent variable in the plot function. The command

```
help plot
```

returns a list of the available options. You can select solid (the default), dashed, dotted, and dash-dot line styles, and you can choose to show the points. The choices among marks include plus signs, stars, circles, and x-marks, among others. There are eight different color choices. (See Table 5.2 for a complete list.)

The following commands illustrate the use of line, color, and mark styles:

```
x = [1:10];
y = [58.5 63.8 64.2 67.3 71.5 88.3 90.1 90.6 89.5 90.4];
plot(x,y,":ok")
```

You can either use the abbreviated color name, or the complete color name in the string. For example, the following plot statement is equivalent and gives the same result:

```
plot(x,y,":oblack")
```

Table 5.2 Line, Mark, and Color Options

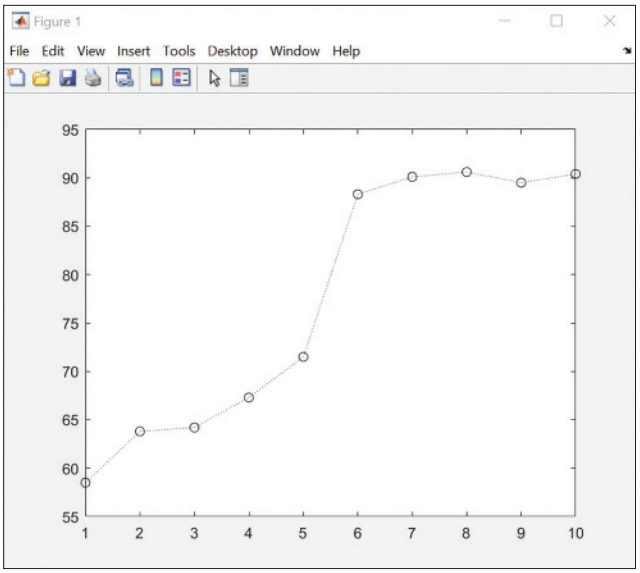
Line Type	Indicator	Point Type	Indicator	Color	Indicator
solid	-	point	.	blue	b
dotted	:	circle	o	green	g
dash-dot	-.	x-mark	x	red	r
dashed	--	plus	+	cyan	c
(none)	no line	star	*	magenta	m
		square	s	yellow	y
		diamond	d	black	k
		triangle down	v	white	w
		triangle up	^		
		triangle left	<		
		triangle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

The resulting plot (see Figure 5.7(a)) consists of a dashed line, with data points marked with circles. The line, the points, and the circles are drawn in black. The indicators were listed inside a string, denoted with double quotes. The order in which they are entered is arbitrary and does not affect the output.

To specify line, mark, and color styles for multiple lines, add a string containing the choices after each pair of data arrays. If the string is not included, the defaults are used. For example,

```
plot(x,y,":ok",x,y*2,"--xr",x,y/2,"-b")
```

(a)



(b)

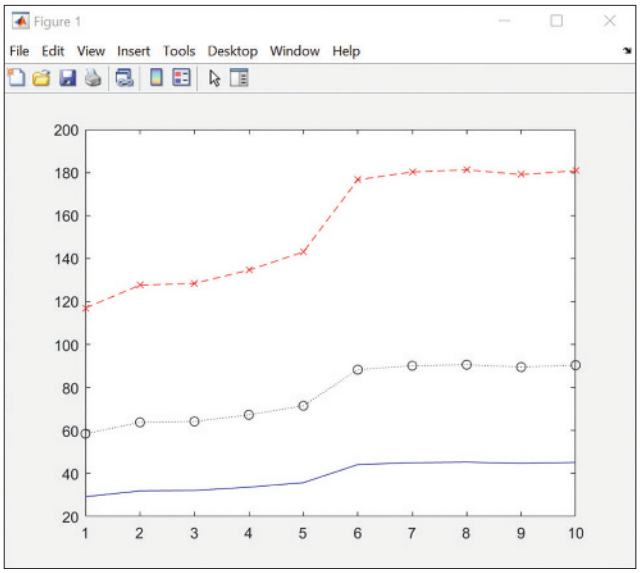


Figure 5.7
(a) Adjusting the line, mark, and color style. (b) Multiple plots with varying line styles and point styles.

KEY IDEA
You may use either single or double quotes to identify the input to plotting functions.

results in the graph shown in Figure 5.7(b). You should be aware that the MATLAB documentation currently refers to both single quote and double quote notation. You may use either approach successfully.

The `plot` command offers additional options to control the way the plot appears. To control these properties you need to specify the property name and value. For example, plots intended for overhead presentations may look better with thicker lines. First, define the data to be plotted, then add the two additional fields. In the first, specify the property as a string and in the second, specify the property value.

```
plot(x,y,"LineWidth",2)
```

creates a plot of a single line with a width of 2 (0.5 is the default) as shown in Figure 5.8(a), and

```
plot(x,y,"LineWidth",2,"Marker","o","MarkerSize",10)
```

creates a plot (see Figure 5.8(b)) with a line width of 2, circles for the markers, and a marker that is larger than the default (10 vs the default of 6). You can add as many property pairs as needed. Explore the possible combinations by searching the help documentation:

```
doc Chart Line Properties
```

Starting with Release 2021a, you can also define these properties using a Name=Value syntax. For the previous example, the equivalent code would be

```
plot(x,y,LineWidth = 2,Marker = "o",MarkerSize = 10)
```

KEY IDEA
Always pair the `hold on` command with the `hold off` command.

This is a more intuitive syntax, and is easier to use, but older (existing) programs will probably continue to use the comma separated pair strategy.

Unfortunately, if you want to fine tune line properties using this technique, it is necessary to define each line in a separate plot statement. To create a graph with multiple lines, use the `hold on` command, then a new `plot` statement for each line. Thus, to add more lines to the graph shown in Figure 5.8(b),

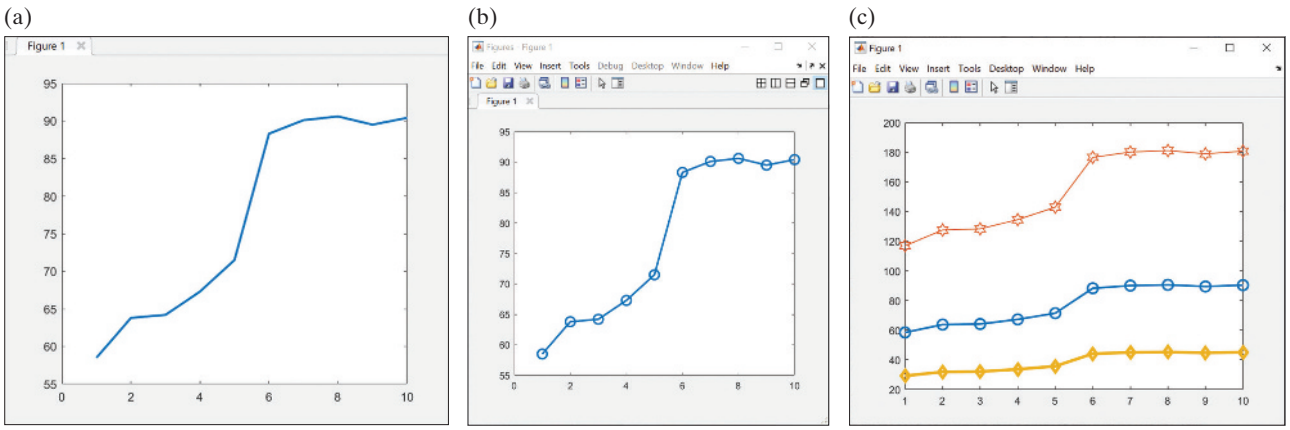


Figure 5.8
(a) More control can be exerted over line properties by adjusting them using line property identifiers and value definitions. (b) Multiple properties can be adjusted on a single line in the same plot statement. (c) Each line requires a separate plot statement, when using property identifiers and value definitions.

```
hold on
plot(x,y*2,LineWidth = 1,Marker = "h",MarkerSize = 12)
plot(x,y/2,LineWidth = 3,Marker = "d",MarkerSize = 8)
```

which results in the graph shown in Figure 5.8(c). The color and line style for each line is determined by the default configuration for the plot. Don't forget to issue the `hold off` command when you are done.

```
hold off
```

5.1.3 Axis Scaling and Annotating Plots

MATLAB automatically selects appropriate *x*-axis and *y*-axis scaling, but sometimes, it is useful for the user to be able to control the scaling. Control is accomplished with the `axis` function, shown in Table 5.3. Executing the `axis` function without any input

```
axis
```

freezes the scaling of the plot. If you use the `hold on` command to add a second line to your graph, the scaling cannot change. To return control of the scaling to MATLAB, simply re-execute the `axis` function.

The `axis` function also accepts input defining the *x*-axis and *y*-axis scaling. The argument is a single array, with four values representing:

- The minimum *x* value shown on the *x*-axis.
- The maximum *x* value shown on the *x*-axis.
- The minimum *y* value shown on the *y*-axis.
- The maximum *y* value shown on the *y*-axis.

Thus, the command

```
axis([-2, 3, 0, 10])
```

fixes the plot axes to *x* from -2 to $+3$ and *y* from 0 to 10.

It is often useful to create plots where the scaling is the same on the *x*- and *y*-axis. This is accomplished with the command

```
axis equal
```

MATLAB offers several additional functions, also listed in Table 5.3, that allow you to annotate your plots. The `legend` function requires the user to specify a

Table 5.3 Axis Scaling and Annotating Plots

`axis`	When the `axis` function is used without inputs, it freezes the axis at the current configuration. Executing the function a second time returns axis control to MATLAB.
`axis(v)`	The input to the `axis` command must be a four-element vector that specifies the minimum and maximum values for both the *x*- and *y*-axes—for example, [`xmin, xmax, ymin, ymax`].
`axis equal`	Forces the scaling on the *x*- and *y*-axis to be the same.
`legend("string1", "string 2", etc)`	Allows you to add a legend to your graph. The legend shows a sample of the line and lists the string you have specified.
`text(x_coordinate,y_coordinate, "string")`	Allows you to add a text box to the graph. The box is placed at the specified *x*- and *y*-coordinates and contains the string value specified.
`gtext("string")`	Similar to text. The box is placed at a location determined interactively by the user by clicking in the figure window.

legend in the form of a string for each line plotted, and defaults to a display in the upper right-hand corner of the plot. (Consult the help feature to determine how to move the legend box to different locations.) The `text` function allows you to add a text box to the plot, which is useful for describing features on the graph. It requires the user to specify the location of the lower left-hand corner of the box in the plot window as the first two input fields, with a string specifying the contents of the text box in the third input field. The use of both `legend` and `text` is demonstrated in the following code, which modifies the graph from Figure 5.7(b). Additionally, the clear figure command, `clf`, was used to make sure a 'fresh' figure window was used.

```
clf
plot(x,y,":ok",x,y*2,"--r",x,y/2,"-b")
legend("line 1","line 2","line 3")
text(1,100,"Label plots with the text command")
```

We added a title, x and y labels, and adjusted the axis with the following commands:

```
xlabel("My x-axis label")
ylabel("My y-axis label")
title("Example graph for Chapter 5")
axis([0,11,0,200])
```

The results are shown in Figure 5.9.

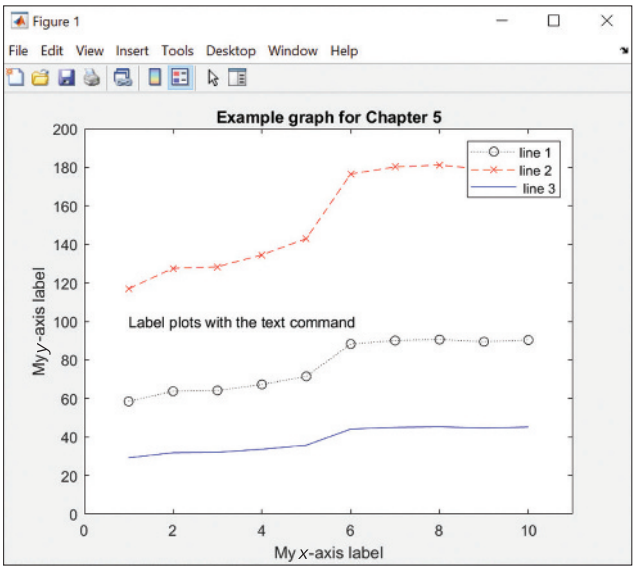


Figure 5.9
Final version of the sample graph, annotated with a legend, a text box, a title, x and y labels, and a modified axis.

HINT

To use Greek letters in your titles and labels enter a backslash (\) before the name of the letter. For example,

```
title("\alpha \beta \gamma")
```

creates the plot title

$$\alpha\ \beta\ \gamma$$

KEY IDEA
When in doubt, use double quotes, not single quotes, to store text in MATLAB.

To create a superscript, use a caret. Thus,

```
title("x^2")
```

gives

$$x^2$$

To create a subscript, use an underscore:

```
title("x_5")
```

gives

$$x_5$$

If your expression requires a group of characters as either a subscript or a superscript, enclose them in curly braces. For example,

KEY IDEA
Strings, designated with double quotes, are easier to work with in MATLAB than character arrays. Character arrays are designated with single quotes.

```
title("k^{-1}")
```

returns

k^{-1}

Finally, to create a title with more than one line of text, store the string input to title as an array with multiple rows. For example

```
title(["First Line of Text";"Second Line of Text"])
```

You can even store the input to title as a variable if it makes sense programmatically. For example:

```
a="First Line of Text";
b="Second Line of Text";
t=[a;b];
title(t)
```

This would be more difficult if the text were stored as character arrays, which is how the information is stored when single quotes are used.

MATLAB has the ability to create other more complicated mathematical expressions for use as titles, axis labels, and other text strings, using the TeX markup language. To learn more, consult the help feature. (Search on "text properties.")

```
doc text properties
```

PRACTICE EXERCISE 5.1

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Plot x versus y for $y = \sin(x)$. Let x vary from 0 to 2π in increments of 0.1π.
2. Add a title and labels to your plot.
3. Plot x versus y_1 and y_2 for $y_1 = \sin(x)$ and $y_2 = \cos(x)$. Let x vary from 0 to 2π in increments of 0.1π. Add a title and labels to your plot.

4. Re-create the plot from Exercise 3, but make the $\sin(x)$ line dashed and red. Make the $\cos(x)$ line green and dotted.
5. Add a legend to the graph in Exercise 4.
6. Adjust the axes so that the x-axis goes from -1 to $2\pi + 1$, and the y-axis from -1.5 to $+1.5$.
7. Plot x versus y for $y = \sin(2*x)$, with a line weight of 3, a line color of green, and with markers as triangles with a weight of 10. Use the Name = Value syntax.
8. Create a new vector, `a = cos(x)`. Let `x` vary from 0 to 2π in increments of 0.1π. Plot just `a` without specifying the `x` values `(plot(a))` and observe the result. Compare this result with the graph produced by plotting `x` versus `a`.

EXAMPLE 5.1

USING THE CLAUSIUS–CLAPEYRON EQUATION

The Clausius–Clapeyron equation can be used to find the saturation vapor pressure of water in the atmosphere, for different temperatures. The saturation water vapor pressure is useful to meteorologists because it can be used to calculate relative humidity, an important component of weather prediction, when the actual partial pressure of water in the air is known.

The Clausius–Clapeyron equation is

$$\ln\left(P^0/6.11\right) = \left(\frac{\Delta H_v}{R_{air}}\right) * \left(\frac{1}{273} - \frac{1}{T}\right)$$

where

P^0 = saturation vapor pressure for water, in mbar, at temperature T,
ΔH_v = latent heat of vaporization for water, 2.453×10^6 J/kg,
R_{air} = gas constant for moist air, 461 J/kg, and
T = temperature in kelvins.

1. State the problem
   Find the saturation vapor pressure at temperatures from -60°F to 120°F, using the Clausius–Clapeyron equation, and present the results graphically.

2. Describe the input and output

   Input

   $$\Delta H_v = 2.453 \times 10^6 \text{ J/kg}$$
   $$R_{air} = 461 \text{ J/kg}$$
   $$\text{T} = -60\text{°F to } 120\text{°F}$$

   Since the number of temperature values was not specified, we'll choose to recalculate every 10°F.

   Output

   Array of temperature and saturation vapor pressures
   Graph of temperature versus saturation vapor pressures

(*continued*)

3. Develop a hand example
Change the temperatures from degrees Fahrenheit to kelvins:

$$T_k = \frac{(T_F + 459.6)}{1.8}$$

Solve the Clausius–Clapeyron equation for the saturation vapor pressure (P^0):

$$\ln\left(\frac{P^0}{6.11}\right) = \left(\frac{\Delta H_v}{R_{air}}\right) \times \left(\frac{1}{273} - \frac{1}{T}\right)$$

$$P^0 = 6.11 * \exp\left(\left(\frac{\Delta H_v}{R_{air}}\right) \times \frac{1}{273} - \frac{1}{T}\right)$$

Notice that the expression for the saturation vapor pressure, P^0, is an exponential equation. We would thus expect the graph to have the shape shown in Figure 5.10.

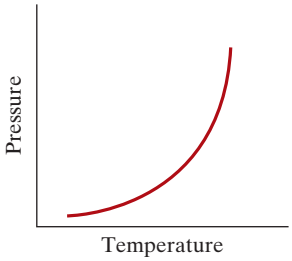


Figure 5.10
A sketch of the predicted equation behavior.

4. Develop a MATLAB solution

```
%% Example 5.1
% Using the Clausius-Clapeyron Equation, find the
% saturation vapor pressure for water at different
% temperatures
%
 TF = [-60:10:120];      % Define temp matrix in F
 TK = (TF + 459.6)/1.8; % Convert temp to K
 Delta_H=2.45e6;         % Define latent heat of vaporization
 R_air = 461;            % Define ideal gas constant for air

%
%% Calculate the Vapor Pressures
 Vapor_Pressure = 6.11*exp((Delta_H/R_air)*(1/273 - 1./TK));
 %% Display the results in an array
  my_results = [TF',Vapor_Pressure']
%
%% Create an x-y plot
 plot(TF,Vapor_Pressure)
 title("Clausius Clapeyron Behavior")
 xlabel("Temperature, ^oF")
 ylabel("Saturation Vapor Pressure, mbar")
```

The resulting array is

```
my_results =
       -60.0000    0.0698
       -50.0000    0.1252
       -40.0000    0.2184
       -30.0000    0.3714
       -20.0000    0.6163
       -10.0000    1.0000
              0    1.5888
        10.0000    2.4749
        20.0000    3.7847
        30.0000    5.6880
```

```
  40.0000      8.4102
  50.0000     12.2458
  60.0000     17.5747
  70.0000     24.8807
  80.0000     34.7729
  90.0000     48.0098
 100.0000     65.5257
 110.0000     88.4608
 120.0000    118.1931
```

A figure window opens to display the graphical results, shown in Figure 5.11.

5. Test the solution
The plot follows the expected trend. It is almost always easier to determine whether computational results make sense if a graph is produced. Tabular data are extremely difficult to interpret.

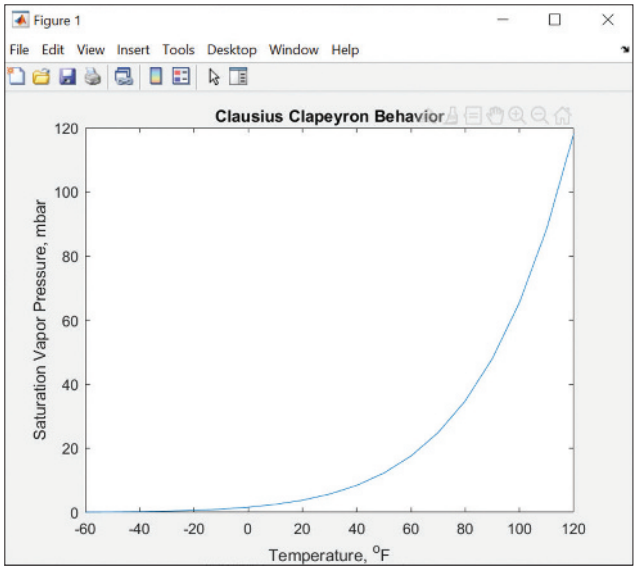


Figure 5.11
A plot of the Clausius–Clapeyron equation.

EXAMPLE 5.2

BALLISTICS

The range of an object (see Figure 5.12) shot at an angle θ with respect to the x-axis and an initial velocity v_0 is given by:

$$R(\theta) = \frac{v^2}{g}\sin(2\theta) \quad \text{for } 0 \leq \theta \leq \frac{\pi}{2} \text{(neglecting air resistance)}$$

Use $g = 9.9$ m/s^2 and an initial velocity of 100 m/s. Show that the maximum range is obtained at $\theta = \pi/4$ by computing and plotting the range for values of θ from

(*continued*)

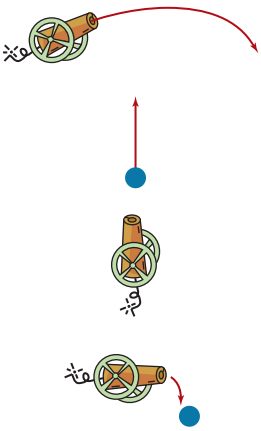


Figure 5.12
The range is zero, if the cannon is perfectly vertical or perfectly horizontal.

$$0 \leq \theta \leq \frac{\pi}{2}$$

in increments of 0.05.

Repeat your calculations with an initial velocity of 50 m/s, and plot both sets of results on a single graph.

1. State the problem
   Calculate the range as a function of the launch angle.

2. Describe the input and output

   Input

   $g = 9.9\ \text{m/s}^2$
   $\theta = 0$ to $\pi/2$, incremented by 0.05
   $v_0 = 50$ m/s and 100 m/s

   Output

   Range R
   Present the results as a plot

3. Develop a hand example
   If the cannon is pointed straight up, we know that the range is zero, and if the cannon is horizontal, the range is also zero (see Figure 5.11).

   This means that the range must increase with the cannon angle up to some maximum, and then decrease. A sample calculation at 45 degrees ($\pi/4$ radians) shows that

   $$R(\theta) = \frac{v^2}{g}\sin(2\theta)$$
   $$R\left(\frac{\pi}{4}\right) = \frac{100^2}{9.9}\sin\left(\frac{2\pi}{4}\right) = 1010 \text{ m when the initial velocity is 100 m/s.}$$

4. Develop a MATLAB solution

```
%% Example 5.2
% The program calculates the range of a ballistic projectile
%% Define the constants
  g = 9.9;  % m/s^2
  v1 = 50;  % m/s^2
  v2 = 100; % m/s^2
%% Define the angle vector
  angle = 0:0.05:pi/2;   %radians
%%  Calculate the range
  R1 = v1^2/g*sin(2*angle);
  R2 = v2^2/g*sin(2*angle);  % meters
%% Plot the results
  plot(angle,R1,angle,R2,"--")
  title("Cannon Range")
  xlabel("Cannon Angle")
  ylabel("Range, meters")
  legend("Initial Velocity=50 m/s","Initial Velocity=100 m/s")
```

Notice that in the `plot` command, we requested MATLAB to print the second set of data as a dashed line. A title, labels, and a legend were also added. The results are shown in Figure 5.13.

5. Test the solution
Compare the MATLAB results with those from the hand example. Both graphs start and end at zero. The maximum range for an initial velocity of 100 m/s is approximately 1000 m, which corresponds well to the calculated value of 1010 m. Notice that both solutions peak at the same angle, approximately 0.8 radian. The numerical value for $\pi/4$ is 0.785 radian, confirming the hypothesis presented in the problem statement that the maximum range is achieved by pointing the cannon at an angle of $\pi/4$ radians (45 degrees).

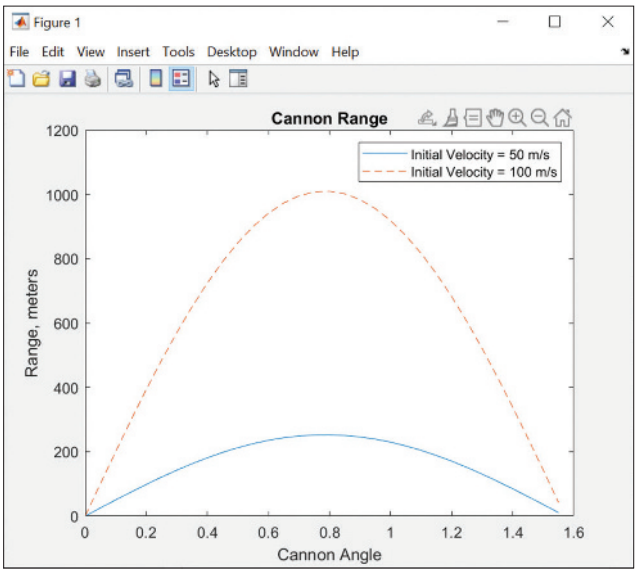


Figure 5.13
The predicted range of a projectile.

HINT

To clear a figure, use the `clf` command. To close the active figure window, use the `close` command, and to close all open figure windows use `close all`.

5.2 SUBPLOTS—TILED CHART LAYOUTS

The `tiledlayout` function allows you to subdivide the graphing window into a grid of m rows and n columns, identifying an area called a **tile** where a plot can be drawn. (Earlier versions of MATLAB used the **subplot** function.). The function

```
tiledlayout(m,n)
```

splits the figure into an $m \times n$ grid. Then the `nexttile` command is used to identify in which tile the plot is drawn, as shown in Figure 5.14.

```
nexttile(p)
```

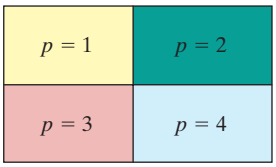


Figure 5.14
Tiles are used to subdivide the figure window into an $m \times n$ array.

The windows are numbered from left to right, top to bottom. If the `nexttile` function is executed without an input argument, plots are created consecutively in numerical order. For example, the following commands split the graphing window into a top plot and a bottom plot:

```
x = 0:pi/20:2*pi;
tiledlayout(2,1)
nexttile
plot(x,sin(x))
title("Sin(x)")
xlabel("x-axis"), ylabel("y-axis"),grid minor
nexttile
plot(x,sin(2*x))
title("Sin(2x)")
xlabel("x-axis"), ylabel("y-axis"), grid minor
```

The first graph is drawn in the top tile. Notice it is not necessary to specify the tile number, since the plots are drawn sequentially. Then the `nexttile` command is used again to draw the next plot in the bottom tile, as shown in Figure 5.15. Titles are added above each plot as the graphs are drawn, as are *x*- and *y*-axis labels and any annotation desired, for example a legend or grid lines.

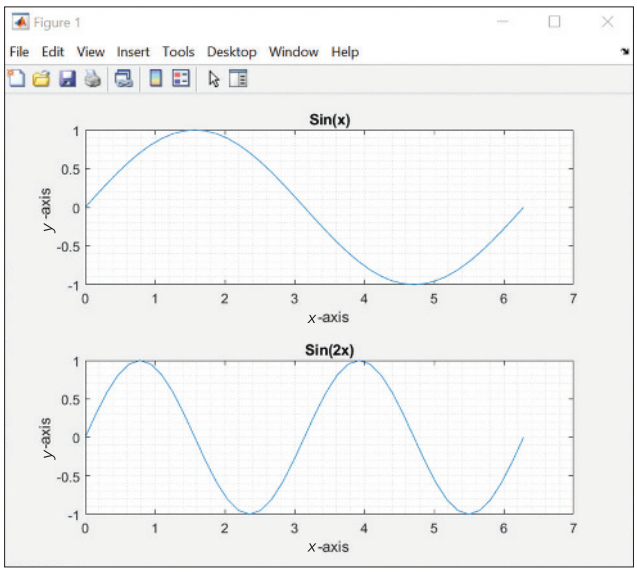


Figure 5.15
The `tiledlayout` and `nexttile` functions allow the user to create multiple graphs in the same figure window.

The `tiledlayout` function and associated `nexttile` function were introduced in release 2019b and replace the `subplot` function, which will eventually be retired. They have several advantages, which make creating multiple plots in the same figure window easier. For example, if you don't know how many tiles will be needed for a particular task, use

```
tiledlayout("flow")
```

KEY IDEA
Tiles are another name for subplots.

The first plot will be created in a figure with a single tile. As additional plots are created, the figure window will be "reflowed" with enough tiles to meet the demand. The program will create a grid using an aspect ratio of approximately 4:3 for each of the tiles.

You can also easily create a title for the entire plot window by specifying a name for the TiledChartLayout object, as in:

```
t = tiledlayout()
```

The object name can be used to specify other properties, such as the tile spacing or padding, or to create shared axis labels among the tiles in the same row or column. You should consult the MATLAB help feature for details of these features after you've mastered the basics of using tiles.

The following code was used to create Figure 5.16a, and illustrates how to create a tiled plot using the flow option, with a title added for the entire plot group.

```
x = 0:pi/20:2*pi;
tile_name = tiledlayout("flow");
% tile_name = tiledlayout(2,2); % use this code to create
% figure 5.16b
nexttile
plot(x,sin(x))
title("Sin(x)"),xlabel("x-axis"), ylabel("y-axis")
nexttile
plot(x,sin(x).^2+cos(x))
title("Trig Function"),xlabel("x-axis"), ylabel("y-axis")
nexttile
% nexttile(3,[1,2]) % use this code to create figure 5.16b
plot(x,2+3*x-8*x.^2+1.5*x.^3)
title("Polynomial"),xlabel("x-axis"), ylabel("y-axis")
title(tile_name,["Some Sample Plots";"Created Using the Flow
Option"])
```

Finally, the `nexttile` function allows the user to span multiple tiles, so that each plot need not be the same size. Reconsider the code used to create Figure 5.16a. If we had specified the layout as:

```
tile_name = tiledlayout(2,2);
```

as shown in the commented out section above, we would have achieved the same result. We need to modify the `nexttile` function for the final plot to "span" tiles 3 and 4, so that the last plot fills the bottom half of the window as shown in Figure 5.16b. The required input is the tile number for the upper left-hand corner of the expanded tile, and the size of the expanded tile expressed in rows and columns. In this example, to fill the bottom half of the window the upper left-hand corner is tile number 3 and the expanded tile is a 1 row by 2 column array.

```
nexttile(3,[1,2])
```

This is shown as commented out code in the section above, so that it is clear how to create the figure.

(a) (b)

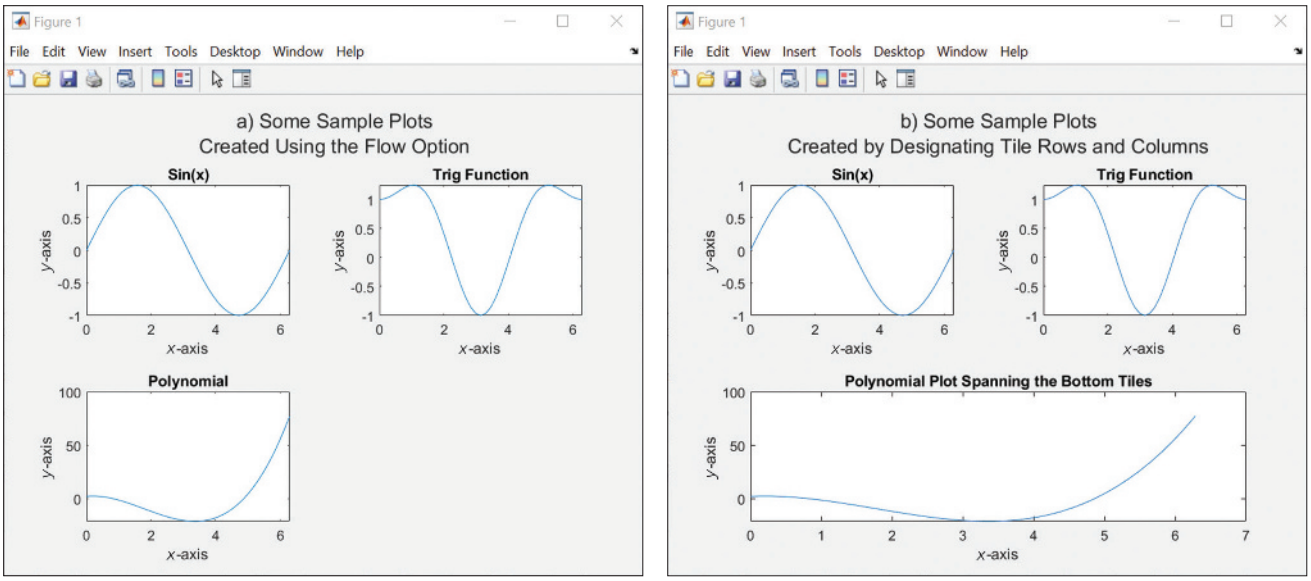


Figure 5.16
Tiled plots created using (a) the flow option and (b) the designated number of rows and columns in the chart. Plots can also be created that span multiple tiles.

PRACTICE EXERCISE 5.2

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Subdivide a figure window into two rows and one column.
2. In the top tile, plot $y = \tan(x)$ for $-1.5 \leq x \leq 1.5$. Use an increment of 0.1.
3. Add a title and axis labels to your graph.
4. In the bottom tile, plot $y = \sinh(x)$ for the same range.
5. Add a title and labels to your graph.
6. Try the preceding exercises again, but divide the figure window vertically instead of horizontally.
7. Add a title to your entire chart.

5.3 OTHER TYPES OF TWO-DIMENSIONAL PLOTS

Although simple *x*–*y* plots are the most common type of engineering plot, there are many other ways to represent data. Depending on the situation, these techniques may be more appropriate than an *x*–*y* plot.

5.3.1 Polar Plots

MATLAB provides plotting capability with polar coordinates.

```
polarplot(theta, r)
```

generates a polar plot of angle theta (in radians) and radial distance *r*. (Earlier versions of MATLAB used `polar` instead of `polarplot`.)

For example, the code

```
x = 0:pi/100:pi;
y = sin(x);
polarplot(x,y)
```

generates the plot in Figure 5.17. A title was added in the usual way:

```
title("The sine function in polar coordinates is a circle")
```

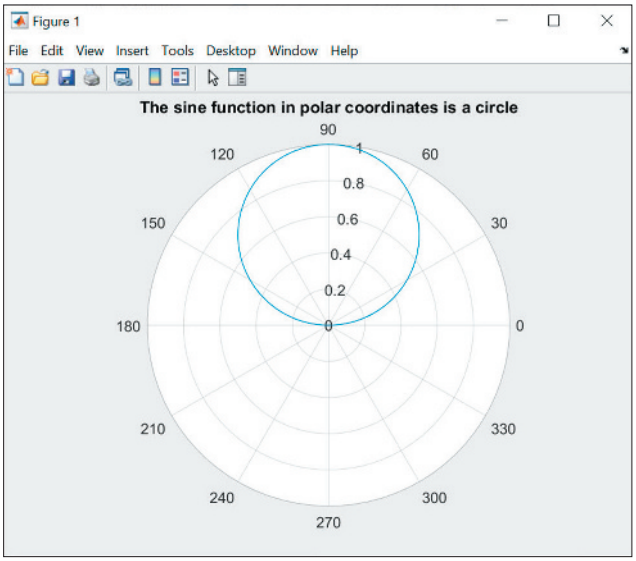
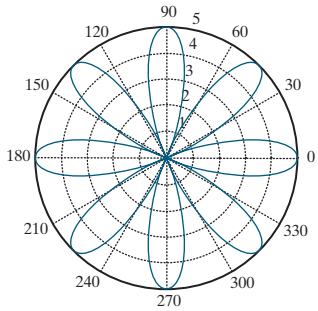


Figure 5.17
A polar plot of the sine function.

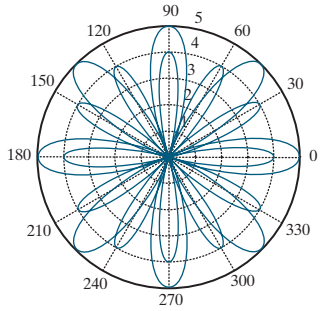
PRACTICE EXERCISE 5.3

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.
Use the `polarplot` function, not the older `polar` function.

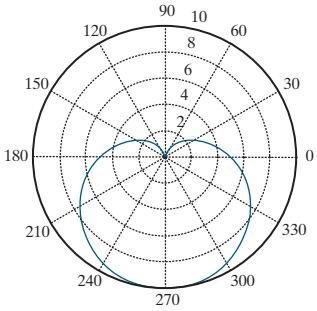
1. Define an array called `theta`, from 0 to 2π, in steps of 0.01π.
Define an array of distances `r = 5*cos(4*theta)`.
Make a polar plot of `theta` versus `r`.



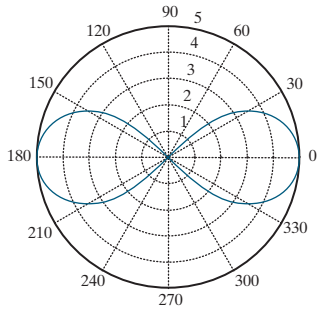
2. Use the `hold on` command to freeze the graph.
Assign `r = 4*cos(6*theta)` and plot.
Add a title.



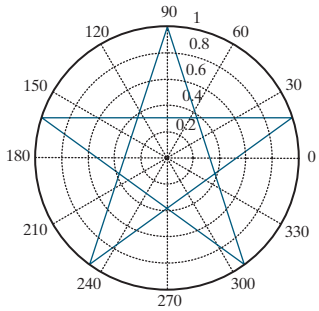
3. Create a new figure.
Use the `theta` array from the preceding exercises.
Assign `r = 5 - 5*sin(theta)` and create a new polar plot.



4. Create a new figure.
Use the `theta` array from the preceding exercises.
Assign `r = sqrt(5^2*cos(2*theta))` and create a new polar plot.



5. Create a new figure.
Define a `theta` array such that
`theta = pi/2:4/5*pi:4.8*pi`
Create a six-member array of ones called `r`.
Create a new polar plot of `theta` versus `r`.



KEY IDEA
Logarithmic plots are especially useful if the data vary exponentially.

5.3.2 Logarithmic Plots

For most plots that we generate, the *x*- and *y*-axes are divided into equally spaced intervals; these plots are called *linear* or *rectangular* plots. Occasionally, however, we may want to use a logarithmic scale on one or both of the axes. A logarithmic scale (base 10) is convenient when a variable changes over many orders of magnitude, because the wide range of values can be graphed without compressing the smaller values. Logarithmic plots are also useful for representing data that vary exponentially. Appendix B discusses in more detail when to use the various types of logarithmic scaling.

The MATLAB commands for generating linear and logarithmic plots of the vectors `x` and `y` are listed in Table 5.4.

Table 5.4 Rectangular and Logarithmic Plots

`plot(x,y)`	Generates a linear plot of the vectors `x` and `y`
`semilogx(x,y)`	Generates a plot of the values of `x` and `y`, using a logarithmic scale for `x` and a linear scale for `y`
`semilogy(x,y)`	Generates a plot of the values of `x` and `y`, using a linear scale for `x` and a logarithmic scale for `y`
`loglog(x,y)`	Generates a plot of the vectors `x` and `y`, using a logarithmic scale for both `x` and `y`

Remember that the logarithm of a negative number or of zero does not exist (although MATLAB represents logs of negative numbers as complex values). If your data include these values, MATLAB will issue a warning message and will not plot the points in question. However, it will generate a plot based on the remaining points.

Usually you will use logarithmic plots with two inputs, both the dependent and independent variables. However, each command for logarithmic plotting can be executed with a single argument, as we saw in `plot(y)` for a linear plot. In these cases, the plots are generated with the values of the indices of the vector `y` used as `x` values.

As an example, plots of $y = 5x^2$ were created using all four scaling approaches, as shown in Figure 5.18. The linear (rectangular) plot, semilog plot along the x-axis, semilog plot along the y-axis, and log–log plot are all shown on one figure, created with the `tiledlayout` and `nexttile` functions. Notice that the flow option was enabled, allowing the `tiledlayout` function to automatically determine the required number of tiles required.

```
t=tiledlayout("flow");
x = 0:0.5:50;
y = 5*x.^2;
nexttile
plot(x,y)
title("Polynomial - linear/linear")
ylabel("y-axis"), grid
nexttile
semilogx(x,y)
title("Polynomial - log/linear")
ylabel("y-axis"), grid
nexttile
semilogy(x,y)
title("Polynomial - linear/log")
xlabel("x-axis"), ylabel("y-axis"), grid
nexttile
loglog(x,y)
title("Polynomial - log/log")
xlabel("x-axis"), ylabel("y-axis"), grid
m = "A Polynomial Plotted Using Multiple Graphing Strategies";
title(t,m)
```

As a matter of style, notice that only the bottom two subplots have x-axis labels.

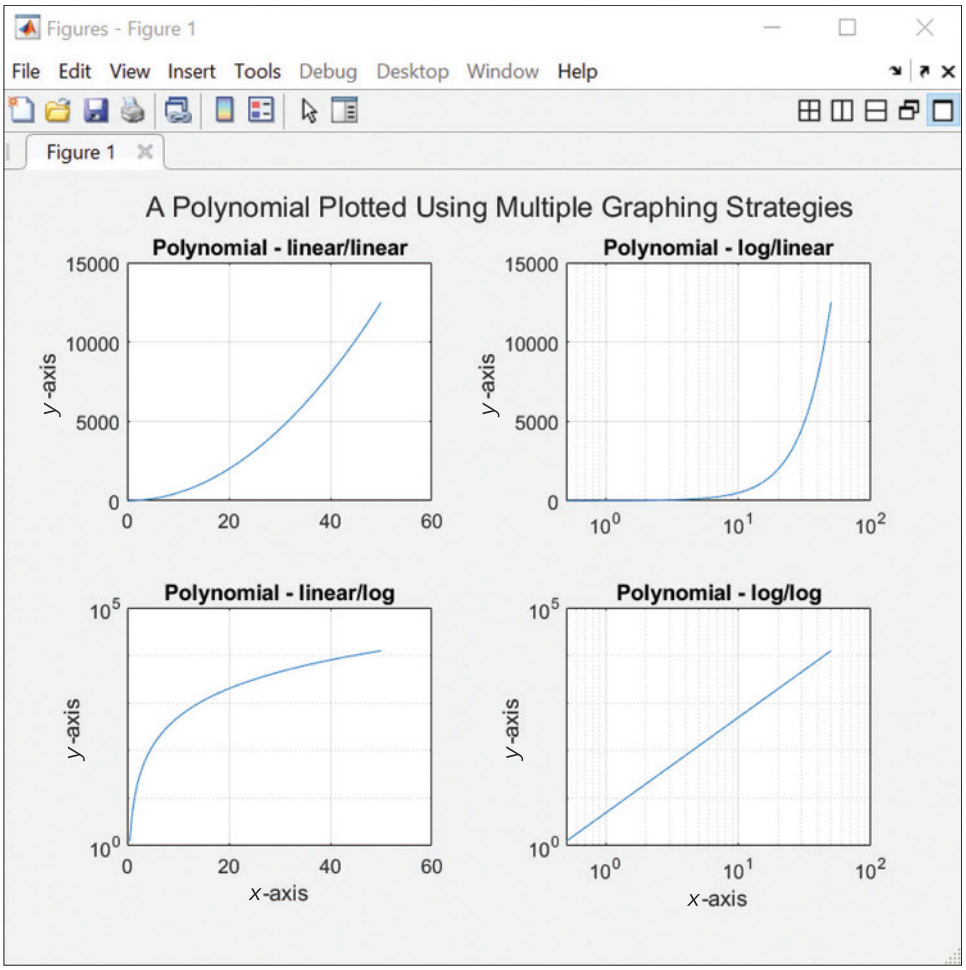


Figure 5.18
Linear and logarithmic plots, displayed using the `tiledlayout` and `nexttile` functions.

EXAMPLE 5.3

RATES OF DIFFUSION

Metals are often treated to make them stronger and therefore wear longer. One problem with making a strong piece of metal is that the metal becomes difficult to form into a desired shape. A strategy that gets around this problem is to form a soft metal into the shape you desire, then harden the surface. This makes the metal wear well without making it brittle.

A common hardening process is called *carburizing*. The metal part is exposed to carbon, which diffuses into the part, making it harder. This is a very slow process if performed at low temperatures, but it can be accelerated by heating the part. The diffusivity is a measure of how fast diffusion occurs, and can be modeled as

$$D = D_0 \exp\left(\frac{-Q}{RT}\right)$$

where

D = diffusivity, cm^2/s,
D_0 = diffusion coefficient, cm^2/s,
Q = activation energy, J/mol, 8.314 J/mol K,
R = ideal gas constant, J/mol K, and
T = temperature, K.

As iron is heated, it changes structure and its diffusion characteristics change. The values of D_0 and Q are shown in the following table for carbon diffusing through each of iron's structures:

Type of Metal	D_0 (cm^2/s)	Q (J/mol K)
alpha Fe (BCC)	0.0062	80,000
gamma Fe (FCC)	0.23	148,000

Create a plot of diffusivity versus inverse temperature $(1/T)$, using the data provided. Try the rectangular, semilog, and log–log plots to see which you think might represent the results best. Let the temperature vary from room temperature (25°C) to 1200°C.

1. State the problem
Calculate the diffusivity of carbon in iron.

2. Describe the input and output

Input

For C in alpha iron, $D_0 = 0.0062$ cm^2/s and $Q = 80{,}000$ J/mol K
For C in gamma iron, $D_0 = 0.23$ cm^2/s and $Q = 148{,}000$ J/mol K
$R = 8.314$ J/mol K
T varies from 25°C to 1200°C

(*continued*)

Output

Calculate the diffusivity and plot it.

3. Develop a hand example
The diffusivity is given by

$$D = D_0 \exp\left(\frac{-Q}{RT}\right).$$

At room temperature, the diffusivity for carbon in alpha iron is

$$D = 0.0062 \exp\left(\frac{-80{,}000}{8.314 \times (25 + 273)}\right)$$
$$D = 5.9 \times 10^{-17}.$$

(Notice that the temperature had to be changed from Celsius to Kelvin.)

4. Develop a MATLAB solution

```
%% Example 5.3
% Calculate the diffusivity of Carbon in Iron
clear, clc
%% Define the constants and variables
D0alpha = .0062;
D0gamma = 0.23;
Qalpha = 80000;
Qgamma = 148000;
R = 8.314;
T = 25:5:1200;
%% Change T from C to K
T = T+273;
%% Calculate the Diffusivity
Dalpha = D0alpha*exp(-Qalpha./(R*T));
Dgamma = D0gamma*exp(-Qgamma./(R*T));
%% Plot the results
t=tiledlayout("flow")
nexttile
plot(1./T,Dalpha, 1./T,Dgamma)
title("Diffusivity of C in Fe")
ylabel("Diffusivity, cm^2/s")
legend("\alpha-Ferrite","\gamma-Austenite")
grid on

nexttile
semilogx(1./T,Dalpha, 1./T,Dgamma)
title("Diffusivity of C in Fe")
grid on

nexttile
semilogy(1./T,Dalpha, 1./T,Dgamma)
title("Diffusivity of C in Fe")
xlabel("Inverse Temperature, K^{-1}")
ylabel("Diffusivity, cm^2/s")
grid on
```

```
nexttile
loglog(1./T,Dalpha, 1./T,Dgamma)
title("Diffusivity of C in Fe")
xlabel("Inverse Temperature, K^{-1}")
grid on
a="A Variety of Graphing Strategies";
b="Highlighting the Effect of Temperature on Diffusivity"
c="For Carbon in Iron"
title(t,[a;b;c])
```

Tiles were used to create Figure 5.19, so that all four variations of the plot are in the same figure. Notice that x-labels were added only to the bottom two graphs, to reduce clutter, and that a legend was added only to the first plot.

A title was added to the entire figure, in addition to titles for each of the individual graphs. Superscripts were used in the x-axis and y-axis labels.

The `semilogy` plot resulted in straight lines and allows a user to read values off the graph easily over a wide range of both temperatures and diffusivities.

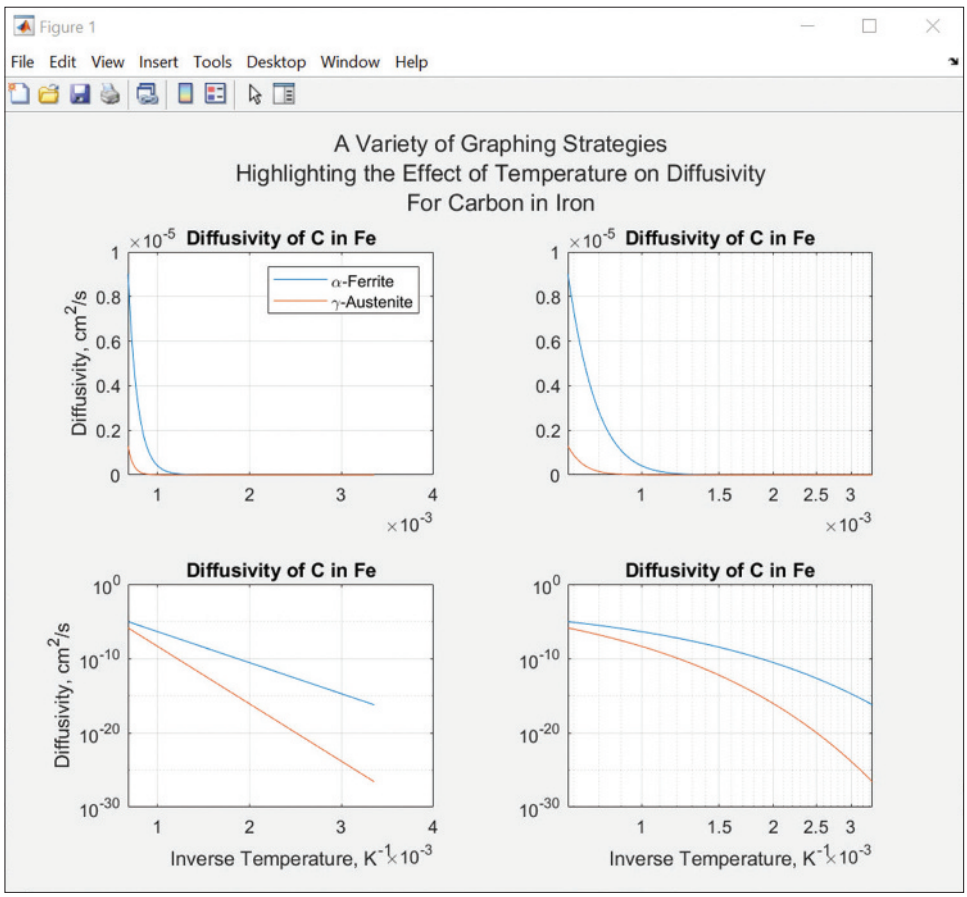


Figure 5.19
Diffusivity data plotted on different scales. The data follows a straight line when the $\log_{10}$ of the diffusivity is plotted on the y-axis versus the inverse temperature on the x-axis.

(*continued*)

This is the plotting scheme usually used in textbooks and handbooks to present diffusivity values.

5. Test the solution
Compare the MATLAB results with those from the hand example.
We calculated the diffusivity to be

$$5.9 \times 10^{-17} \text{ cm}^2/\text{s at } 25^\circ\text{C}$$

for carbon in alpha iron. To check our answer, we'll need to change 25°C to kelvins and take the inverse:

$$\frac{1}{(25 + 273)} = 3.36 \times 10^{-3}$$

From the semilogy graph (lower left-hand corner), we can see that the diffusivity for alpha iron is approximately 10^{-17} cm^2/s.

PRACTICE EXERCISE 5.4

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.
Create appropriate x and y arrays to use in plotting each of the expressions that follow. Use `tiles` to divide your figures into four sections, and create each of these four graphs for each expression:

- Rectangular
- Semilogx
- Semilogy
- Loglog

1. $y = 5x + 3$
2. $y = 3x^2$
3. $y = 12e^{(x+2)}$
4. $y = 1/x$

Physical data usually are plotted so that they fall on a straight line. Which of the preceding types of plot results in a straight line for each problem?

5.3.3 Bar Graphs and Pie Charts

Bar graphs, histograms, and pie charts are popular forms for reporting data. Some of the commonly used MATLAB functions for creating bar graphs and pie charts are listed in Table 5.5.

Examples of some of these graphs generated by the following code are shown in Figure 5.20. The graphs make use of the `tiledlayout` functions to allow four plots in the same figure window:

```
clear, clc, clf
x = [1,2,5,4,8];
y = [x;1:5];
t=tiledlayout("flow");
```

Table 5.5 Bar Graphs and Pie Charts

`bar(x)`	When `x` is a vector, `bar` generates a vertical bar graph. When `x` is a two-dimensional array, `bar` groups the data by row.
`barh(x)`	When `x` is a vector, `barh` generates a horizontal bar graph. When `x` is a two-dimensional array, `barh` groups the data by row.
`bar3(x)`	Generates a three-dimensional bar chart.
`bar3h(x)`	Generates a three-dimensional horizontal bar chart.
`pie(x)`	Generates a pie chart. Each element in the array is represented as a slice of the pie.
`pie3(x)`	Generates a three-dimensional pie chart. Each element in the array is represented as a slice of the pie.
`histogram(x)`	Generates a histogram.

```
% t=tiledlayout(2,2); % This would give the same result
nexttile
bar(x),title("A Bar Graph of Vector x")
nexttile
bar(y),title("A Bar Graph of Array y")
nexttile
bar3(y),title("A Three-dimensional Bar Graph")
nexttile
pie(x),title("A Pie Chart of x")
title(t,"Sample Bar Graphs and Pie Charts")
```

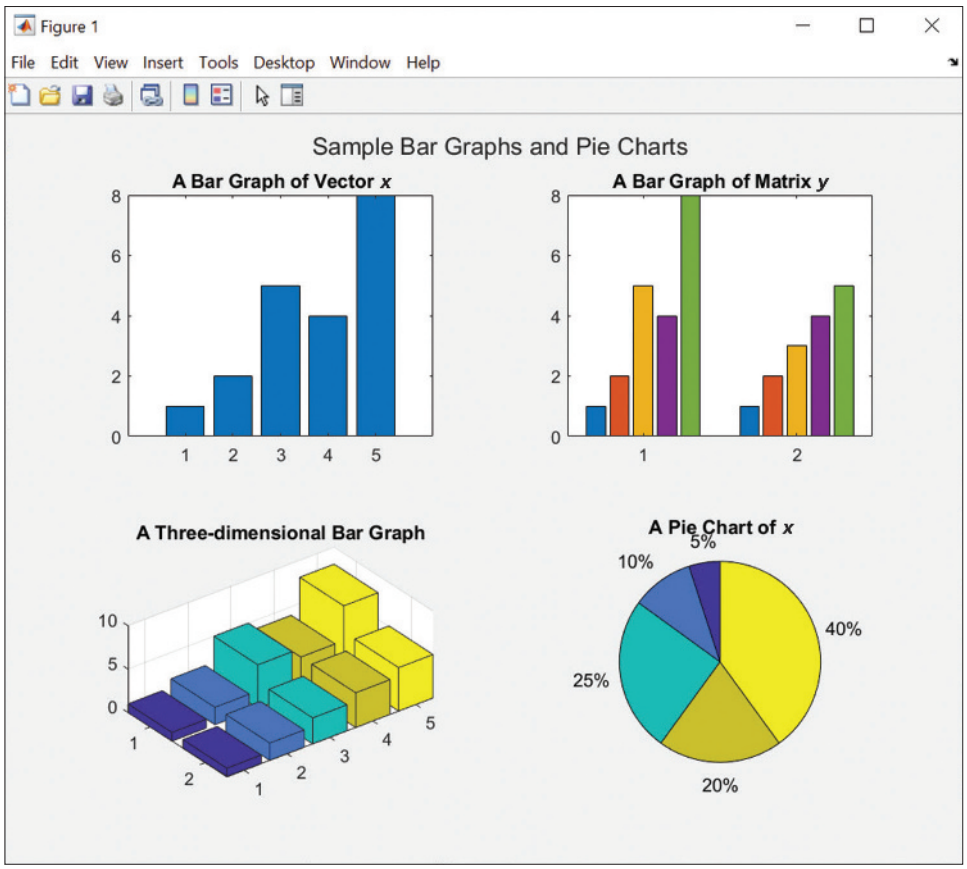


Figure 5.20
Sample bar graphs and pie charts. The functions, `tiledlayout` and `nexttile`, were used to divide the window into quadrants.

KEY IDEA
Histograms are useful in statistical analysis.

5.3.4 Histograms

A **histogram** is a plot showing the distribution of a set of values, especially useful in statistical analysis. In MATLAB, the histogram computes the number of values falling into a number of **bins** (categories) that are equally spaced between the minimum and maximum values. For example, if we define an array x as the set of grades from the Introduction to Engineering final, the scores could be represented in a histogram, shown in Figure 5.21(a) and generated with the following code:

```
x = [100,95,74,87,22,78,34,82,93,88,86,69,55,72];
histogram(x)
```

The `histogram` function determines an appropriate number of bins automatically, but also allows for the user to specify the number of bins. For example, to create a histogram with 5 bins, the command would be

```
histogram(x, 5)
```

The results are shown in Figure 5.21(b).

Although it usually makes sense to space the bins equally between the minimum and maximum data value, sometimes it is more meaningful to define the bins based

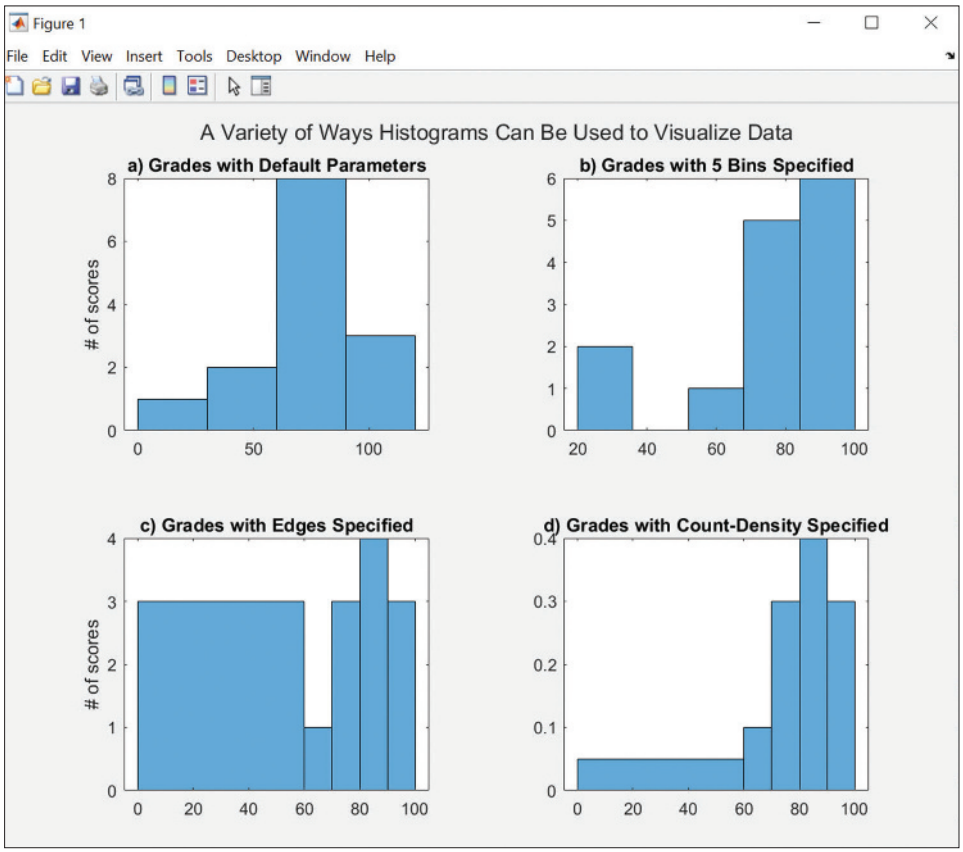


Figure 5.21
Histograms represent the distribution of data. (a) Grade data using the `histogram` function with default parameters, (b) the number of bins specified, (c) the bin edges specified, and (d) the bin edges specified but normalized so that the area of each block corresponds to the number of items in each bin.

on some other criteria. For example, consider the grade data. It would be preferable to define each bin so that they correspond to specific grades as follows:

A	90–100
B	80–90
C	70–80
D	60–70
E	0–60

First, define the bin edges as an array, then enter them in the second field in the histogram function.

```
edges=[0,60,70,80,90,100];
histogram(x,edges)
```

The results are shown in Figure 5.21(c). Notice that three of the scores fell between 0 and 60, yet at first glance the left-most bin looks much larger than the other results. An alternate way to represent the data is to make the *area* of the bin proportional to the number of values in the bin, using a normalization scheme.

```
histogram(x,edges,"normalization","countdensity")
```

The results are shown in Figure 5.21(d). Notice that the left-most bin has a width of 60 and a height of 0.05. The area of the bin is 3 (0.05*60), the number of scores that fell inside the bin corresponding to a grade of E.

Setting the results of the histogram function equal to a variable creates a **handle** to the plot. You can think of a handle as a nickname. Handles are covered in more detail in a latter chapter. If you would like to save the plotting data, such as the number of values in each bin, use the `histcounts` function,

```
a = histcounts(x)
```

which creates an array representing the number of values used to create Figure 5.21(a).

```
a =
    1 2 8 3
```

The `histogram` and `histcounts` functions replace the earlier `hist` and `histc` functions used before version 2014a. Older code should be updated to the newer functions.

EXAMPLE 5.4

WEIGHT DISTRIBUTIONS

The average 18-year-old American male weighs 152 pounds. A group of 100 young males were weighed and the data stored in a file called *weight.dat*. Create a graph to represent the data.

1. State the problem
   Use the data file to create a line graph and a histogram. Which is a better representation of the data?

2. Describe the input and output

Input	*weight.dat*, an ASCII data file that contains weight data
Output	A line plot of the data A histogram of the data

(*continued*)

3. Develop a hand example
   Since this is a sample of actual weights, we would expect the data to approximate a normal random distribution (a Gaussian distribution). The histogram should be bell-shaped.

4. Develop a MATLAB solution
   The following code generates the plots shown in Figure 5.22:

```
%% Example 5.4
% Using Weight Data
clear, clc, clf
load weight.dat
%% Create the line plot of weight data
tiledlayout(1,2)
nexttile
plot(weight)
title("Weight of Freshman Class Males")
xlabel("Student Number")
ylabel("Weight, lbs")
grid on
%% Create the histogram of the data
nexttile
histogram(weight)
xlabel("Weight, lbs")
ylabel("Number of students")
title("Weight of Freshman Class Males")
```

5. Test the solution
   The graphs match our expectations. The weight appears to average about 150 lb and varies in what looks like a normal distribution. We can use MATLAB

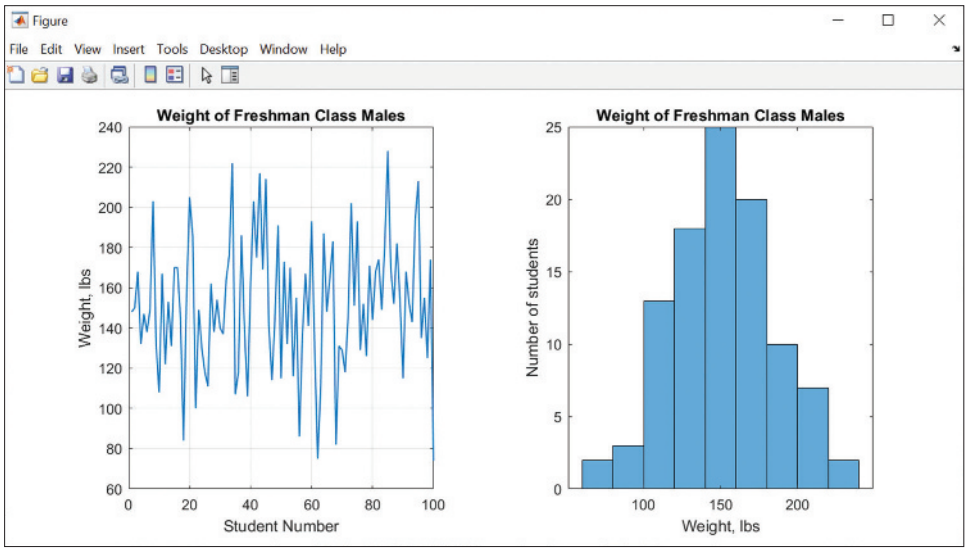


Figure 5.22
Histograms and line plots are two different ways to visualize numeric information.

to find the average and the standard deviation of the data, as well as the maximum and minimum weights in the data set. The MATLAB code

```
average_weight = mean(weight)
standard_deviation = std(weight)
maximum_weight = max(weight)
minimum_weight = min(weight)
```

returns

```
average_weight = 151.1500
standard_deviation = 32.9411
maximum_weight = 228
minimum_weight = 74
```

5.3.5 Graphs with Two *y*-Axes

Sometimes, it is useful to overlay two x–y plots onto the same figure. However, if the orders of magnitude of the y-values are quite different, it may be difficult to see how the data behave. Consider, for example, a graph of $\sin(x)$ and e^x drawn on the same figure. The results, obtained with the following code, are shown in Figure 5.23(a):

```
x = 0:pi/20:2*pi;
y1 = sin(x);
y2 = exp(x);
t=tiledlayout("flow");
nexttile
plot(x,y1,x,y2)
xlabel("x-axis"), ylabel("y-axis")
```

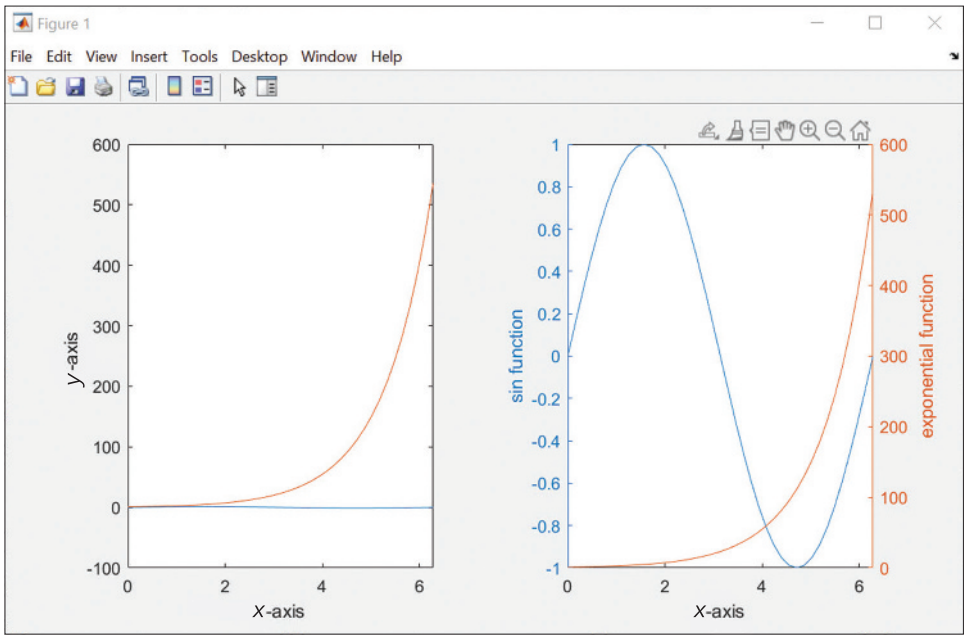


Figure 5.23
MATLAB allows the *y*-axis to be scaled differently on the left-hand and right-hand sides of the figure. (a) Both lines were drawn using the same scaling. (b) The sine curve was drawn using the scaling on the left axis, while the exponential curve was drawn using the scaling on the right axis.

The plot of $\sin(x)$ looks like it runs straight along the line $x = 0$, because of the scale. The `yyaxis` function allows us to create a graph with two y-axes, one on the left and one on the right, by specifying which axis to use.

```
tiledlayout(1,2)
nexttile
plot(x,y1,x,y2)
xlabel("x-axis"), ylabel("y-axis")
nexttile
yyaxis left % default - this line of code is not necessary
plot(x,y1)
xlabel("x-axis"),ylabel("sin function")
yyaxis right
plot(x,y2)
ylabel("exponential function")
```

EXAMPLE 5.5

PERIODIC PROPERTIES OF THE ELEMENTS

The properties of elements in the same row or column in the periodic table usually display a recognizable trend as we move across a row or down a column. For example, the melting point usually goes down as we move down a column, because the atoms are farther apart and the bonds between the atoms are therefore weaker. Similarly, the radius of the atoms goes up as we move down a column, because there are more electrons in each atom and correspondingly bigger orbitals. It is instructive to plot these trends against atomic number on the same graph.

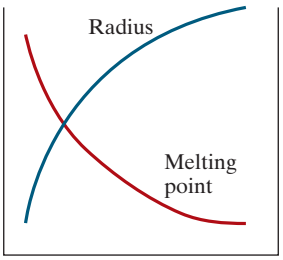


Figure 5.24
Sketch of the predicted data behavior.

1. State the problem
   Plot the melting point and the atomic radius of the Group I elements against the atomic number, and comment on the trends you observe.

2. Describe the input and output

   Input The atomic numbers, melting points, and atomic radii of the Group I elements are listed in Table 5.6.

   Output Plot with both melting point and atomic radius on the same graph.

3. Develop a hand example
   We would expect the graph to look something like the sketch shown in Figure 5.24.

Table 5.6 Group I Elements and Selected Physical Properties

Element	Atomic Number	Melting Point, °C	Atomic Radius, pm
Lithium	3	181	0.1520
Sodium	11	98	0.1860
Potassium	19	63	0.2270
Rubidium	37	34	0.2480
Cesium	55	28.4	0.2650

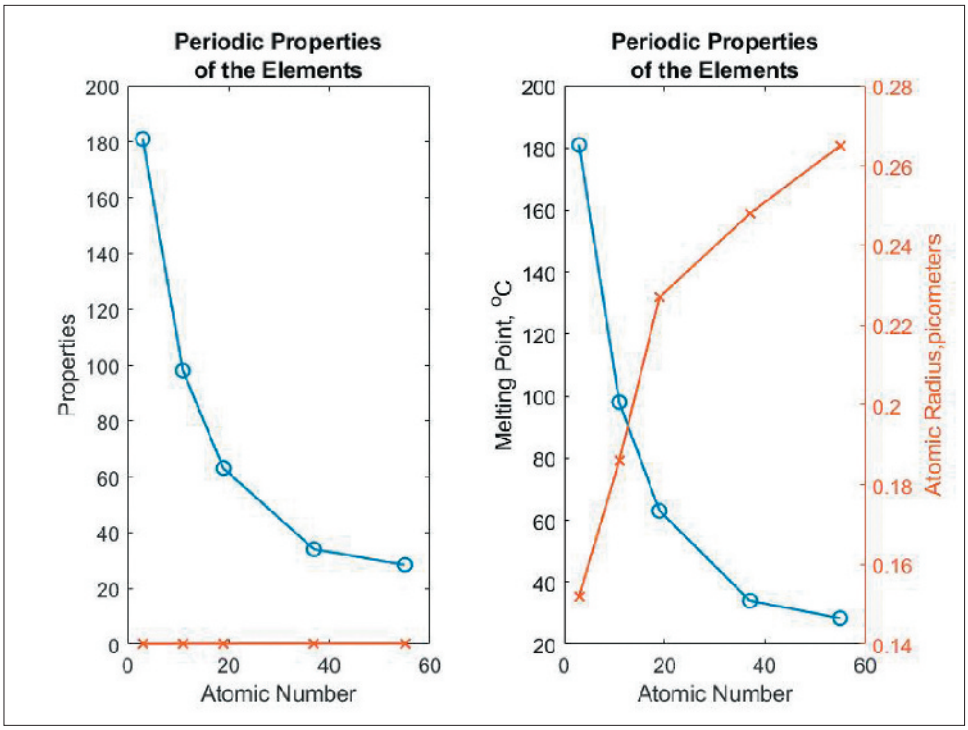


Figure 5.25
In the left-hand figure, the two sets of values were plotted using the same scale. Using two *y*-axes allows us to plot data with different units on the same graph, as shown in the right-hand figure.

4. Develop a MATLAB solution
The following code produces the plot shown in Figure 5.25:

```
%% Example 5.5
% Periodic Properties of the Elements
clear, clc, clf
%% Define the variables
at_num = [ 3, 11, 19, 37, 55]; % atomic number
mp = [181, 98, 63, 34, 28.4]; % melting point
at_r = [0.152, 0.186, 0.227, 0.2480, 0.2650]; % atomic radius
%% Create the plot with two lines on the same scale
tiledlayout(1,2)
nexttile
plot(at_num,mp,"-o",at_num,at_r,"-x")
title(["Periodic Properties";"of the Elements"])
xlabel("Atomic Number")
ylabel("Properties")

%% Create the second plot with two different y scales
nexttile
plot(at_num,mp,"-o")
title(["Periodic Properties";"of the Elements"])
xlabel("Atomic Number")
ylabel("Melting Point, °C")
yyaxis right
```

(*continued*)

```
plot(at_num,at_r,"-x")
ylabel("Atomic Radius,picometers")
```

On the second graph, which has two different *y* scales, we used `yyaxis` to define the right-hand axis. This forced the addition of a second scale, on the right-hand side of the plot. We needed it because atomic radius and melting point have different units, and the values for each have different magnitudes. Notice that in the first plot, it is almost impossible to see the atomic-radius line; it is on top of the *x*-axis because the numbers are so small.

5. Test the solution
   Compare the MATLAB results with those from the hand example. The trend matches our prediction. Clearly, the graph with two *y*-axes is the superior representation, because we can see the property trends.

5.3.6 Function Plots

The `fplot` function allows you to plot a function without defining arrays of corresponding *x*- and *y*-values. For example,

```
fplot(@(x)sin(x),[-2*pi,2*pi])
```

creates a plot (see Figure 5.26) of *x* versus sin (*x*) for *x*-values from -2π to 2π. MATLAB automatically calculates the spacing of *x*-values to create a smooth curve. The first argument in the `fplot` function starts with the `@` symbol to specify the independent variable and continues with the function to be plotted. The second argument is an array defining the plotting range. For more complicated functions that may be inconvenient to enter in this way, you may define an anonymous function and enter

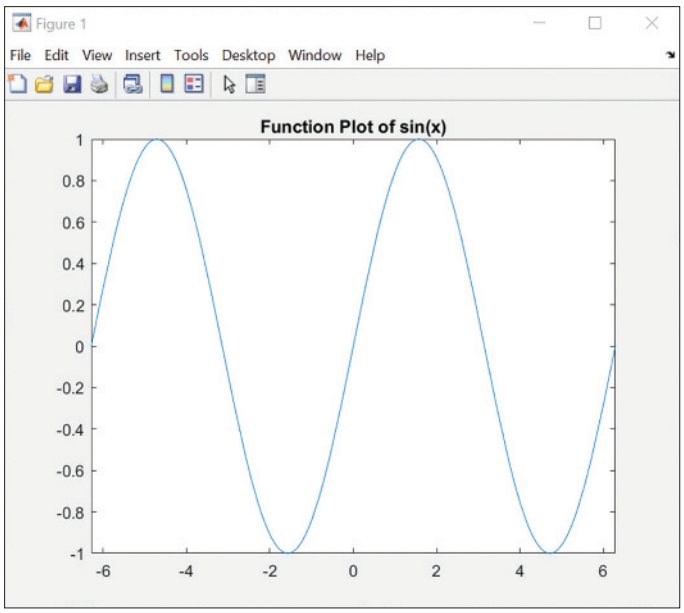


Figure 5.26
Function plots do not require the user to define arrays of ordered pairs.

the function handle. Anonymous functions and function handles are described in a later chapter devoted to functions. The correct syntax for this case would be

```
f = @(x) sin(x)
fplot(f, [-2*pi,2*pi])
```

PRACTICE EXERCISE 5.5

Create a plot of the functions that follow, using `fplot`. You'll need to select an appropriate range for each plot. Don't forget to title and label your graphs.

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. $f(t) = 5t^2$
2. $f(t) = 5\sin^2(t) + t\cos^2(t)$
3. $f(t) = te^t$
4. $f(t) = \ln(t) + \sin(t)$

HINT

The correct MATLAB syntax for the mathematical expression $\sin^2(t)$ is `sin(t).^2`

5.4 THREE-DIMENSIONAL PLOTTING

MATLAB offers a variety of three-dimensional plotting commands, several of which are listed in Table 5.7.

5.4.1 Three-Dimensional Line Plot

The `plot3` function is similar to the `plot` function, except that it accepts data in three dimensions. Instead of just providing x and y vectors, the user must also provide

Table 5.7 Three-Dimensional Plots

`plot3(x,y,z)`	Creates a three-dimensional line plot
`comet3(x,y,z)`	Generates an animated version of `plot3`
`mesh(z) or mesh(x,y,z)`	Creates a meshed surface plot
`surf(z) or surf(x,y,z)`	Creates a surface plot; similar to the `mesh` function
`shading interp`	Interpolates between the colors used to illustrate surface plots
`shading flat`	Colors each grid section with a solid color
`colormap(map_name)`	Allows the user to select the color pattern used on surface plots
`contour(z) or contour(x,y,z)`	Generates a contour plot
`contourf(z) or contour(x,y,z)`	Generates a filled-contour plot
`surfc(z) or surfc(x,y,z)`	Creates a combined surface plot and contour plot
`pcolor(z) or pcolor(x,y,z)`	Creates a pseudo color plot

a z vector. These ordered triples are then plotted in three-space and connected with straight lines. For example,

```
clear, clc
x = linspace(0,10*pi,1000);
y = cos(x);
z = sin(x);
plot3(x,y,z)
grid
xlabel("angle"), ylabel("cos(x)"), zlabel("sin(x)")
title("A Spring")
```

KEY IDEA
The axes used for three-dimensional plotting correspond to the right-hand rule.

The title, labels, and grid are added to the graph in Figure 5.27 in the usual way, with the addition of `zlabel` for the z-axis.

The coordinate system used with `plot3` is oriented using the right-handed coordinate system familiar to engineers.

HINT

Just for fun, recreate the plot shown in Figure 5.27, but this time with the `comet3` function:

```
comet3(x,y,z)
```

This plotting function "draws" the graph in an animation sequence. If your animation runs too quickly, add more data points. For two-dimensional line graphs, use the `comet` function.

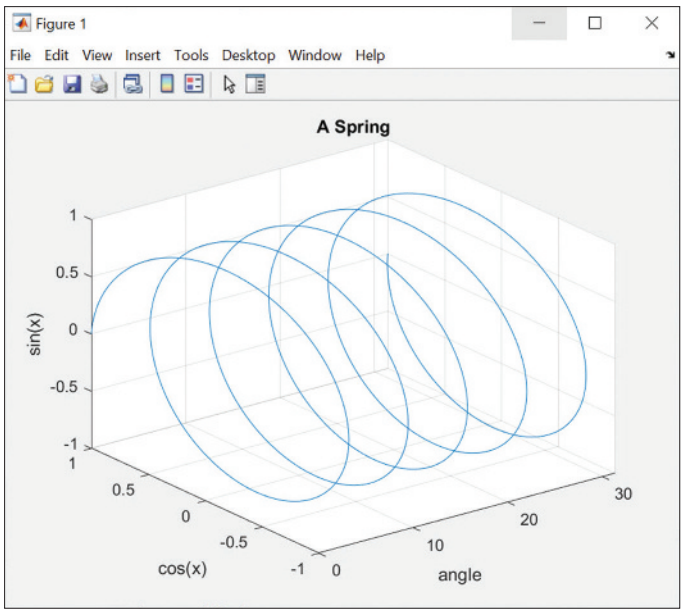


Figure 5.27
A three-dimensional plot of a spring. MATLAB uses a coordinate system consistent with the right-hand rule.

5.4.2 Surface Plots

Surface plots allow us to represent data as a surface. We will be experimenting with two types of surface plots: mesh plots and surf plots.

Mesh Plots

There are several ways to use mesh plots. They can be used to good effect with a single two-dimensional $m \times n$ array. In this application, the value in the array represents the z-value in the plot. The x- and y-values are based on the array dimensions. Take, for example, the following very simple array:

```
z = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10;
     2, 4, 6, 8, 10, 12, 14, 16, 18, 20;
     3, 4, 5, 6, 7, 8, 9, 10, 11, 12];
```

The code

```
mesh(z)
xlabel("x-axis")
ylabel("y-axis")
zlabel("z-axis")
```

generates the graph in Figure 5.28.

The graph is a "mesh" created by connecting the points defined in `z` into a rectilinear grid. Notice that the x-axis goes from 0 to 10 and y goes from 1 to 3. The array index numbers were used for the axis values. For example, note that $z_{1,5}$—the value of z in row 1, column 5—is equal to 5. This element is circled in Figure 5.28.

The `mesh` function can also be used with three arguments: `mesh(x,y,z)`. In this case, `x` is a list of x-coordinates, `y` is a list of y-coordinates, and `z` is a list of z-coordinates.

```
x = linspace(1,50,10)
y = linspace(500,1000,3)
z = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10;
     2, 4, 6, 8, 10, 12, 14, 16, 18, 20;
     3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

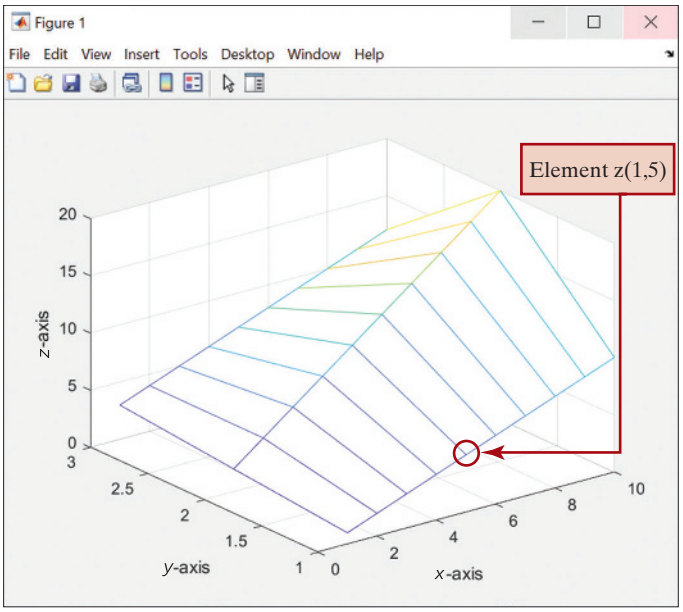


Figure 5.28
Simple mesh created with a single two-dimensional array.

The *x* vector must have the same number of elements as the number of columns in the *z* vector and the *y* vector must have the same number of elements as the number of rows in the *z* vector. The command

```
mesh(x,y,z)
```

creates the plot in Figure 5.29(a). Notice that the *x*-axis varies from 0 to 50, with data plotted from 1 to 50. Compare this scaling with that in Figure 5.28, which used the `z` array index numbers for the *x*- and *y*-axes.

Surf Plots

Surf plots are similar to mesh plots, but `surf` creates a three-dimensional colored surface instead of a mesh. The colors vary with the value of `z`.

The `surf` command takes the same input as `mesh`: either a single input—for example, `surf(z)`, in which case it uses the row and column indices as *x* and *y*-coordinates—or three arrays. Figure 5.29(b) was generated with the same commands as those used to generate Figure 5.29(a), except that `surf` replaced `mesh`.

The shading scheme for surface plots is controlled with the `shading` command. The default, shown in Figure 5.29(b), is "faceted." Interpolated shading can create interesting effects. The plot shown in Figure 5.29(c) was created by adding

```
shading interp
```

to the previous list of commands. Flat shading without the grid is generated when

```
shading flat
```

is used, as shown in Figure 5.29(d).

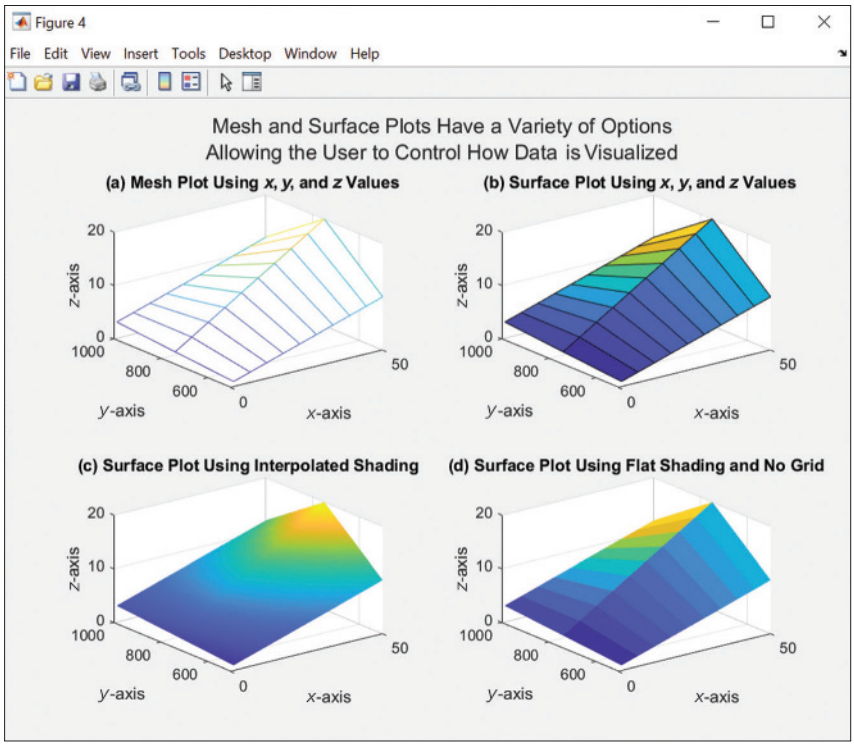


Figure 5.29
Mesh and surf plots are created with three input arguments. (Reprinted with permission from The Mathworks, Inc.)

KEY IDEA
The colormap function controls the colors used on surface plots.

The color scheme used in surface plots can be controlled with a **colormap** defined by the `colormap` function. For example,

```
colormap(gray)
```

forces a grayscale representation for surface plots. This may be appropriate if you'll be making black-and-white copies of your plots. Other available `colormaps` are

autumn	bone	hot
spring	colorcube	hsv
summer	cool	pink
winter	copper	prism
jet	flag	white
parula (default)	gray	lines

Use the `help` command to see a description of the various options:

```
help colormap
```

Another Example

A more complicated surface can be created by calculating the values of `Z`:

```
x = [-2:0.2:2];
y = [-2:0.2:2];
[X,Y] = meshgrid(x,y);
Z = X.*exp(-X.^2 - Y.^2);
```

In the preceding code, the `meshgrid` function is used to create the two-dimensional arrays `X` and `Y` from the one-dimensional vectors `x` and `y`. The values in `Z` are then calculated. The following code plots the calculated values:

```
t = tiledlayout(2,2);
nexttile
mesh(X,Y,Z)
title("(a) Mesh Plot"), xlabel("x-axis"), ylabel("y-axis")
zlabel("z-axis")
nexttile
surf(X,Y,Z)
title("(b) Surface Plot"), xlabel("x-axis"), ylabel("y-axis")
zlabel("z-axis")
```

Either the `x`, `y` vectors or the `X`, `Y` arrays can be used to define the *x*- and *y*-axes. Figure 5.30(a) is a mesh plot of the given function, and Figure 5.30(b) is a surf plot of the same function.

HINT

If a single vector is used in the `meshgrid` function, the program interprets it as

```
[X,Y] = meshgrid(x,x)
```

You could also use the vector definition as input to `meshgrid`:

```
[X,Y] = meshgrid(-2:0.2:2)
```

Both of these lines of code would produce the same result as the commands listed in the example.

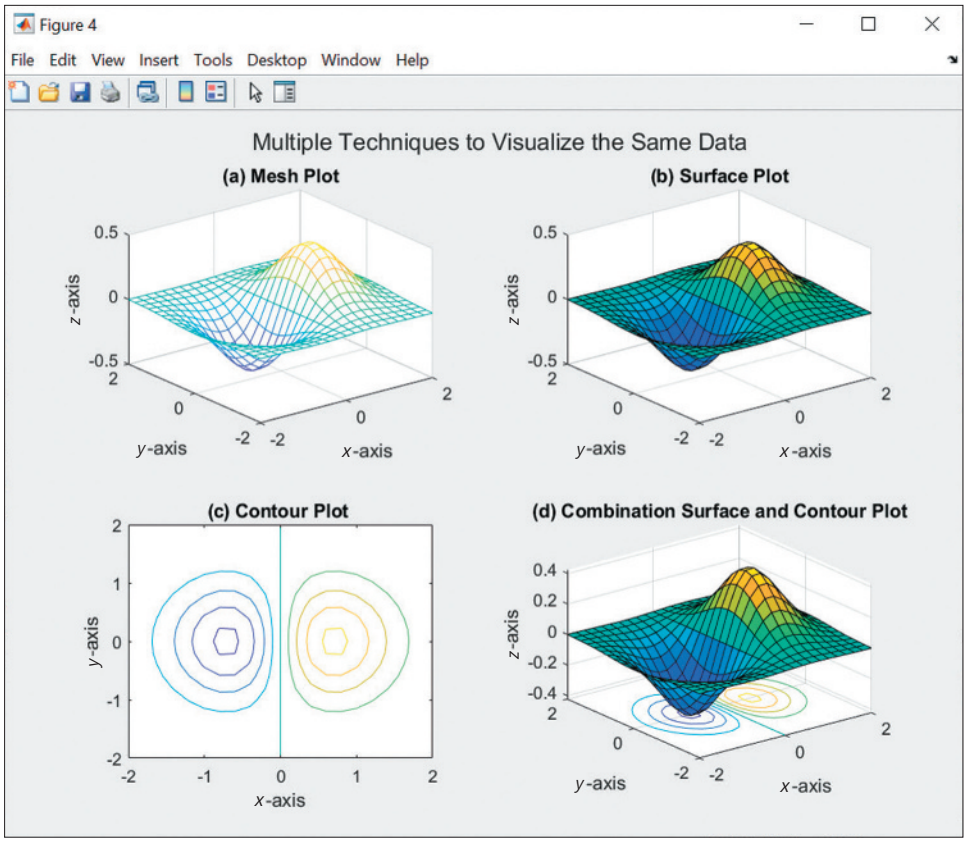


Figure 5.30
Surface and contour plots are different ways of visualizing the same data.

Contour Plots

Contour plots are two-dimensional representations of three-dimensional surfaces, much like the familiar contour maps used by many hikers. The `contour` command was used to create Figure 5.30(c), and the `surfc` command was used to create Figure 5.30(d):

```
nexttile
contour(X,Y,Z)
xlabel("x-axis"), ylabel("y-axis"), title("(c) Contour Plot")
nexttile
surfc(X,Y,Z)
xlabel("x-axis"), ylabel("y-axis"), zlabel("z-axis")
title("(d) Combination Surface and Contour Plot")
title(t,"Multiple Techniques to Visualize the Same Data")
```

Pseudo Color Plots

Pseudo color plots are similar to contour plots, except that instead of lines outlining a specific contour, a two-dimensional shaded map is generated over a grid. MATLAB includes a sample function called `peaks` that generates the `x`, `y`, and `z` arrays of an interesting surface that looks like a mountain range:

```
[x,y,z] = peaks;
```

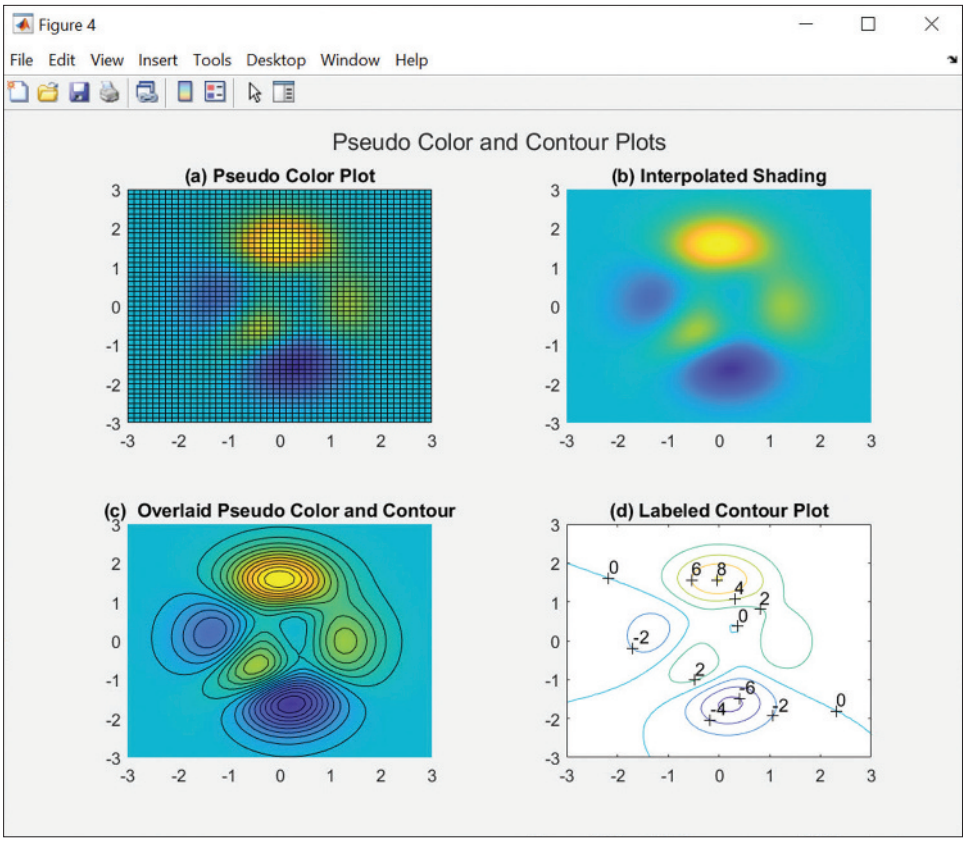


Figure 5.31
A variety of contour plots is available in MATLAB.

With the following code, we can use this surface to demonstrate the use of pseudo color plots, shown in Figure 5.31:

```
t=tiledlayout(2,2)
nexttile
pcolor(x,y,z), title("(a) Pseudo Color Plot")
```

The grid is deleted when interpolated shading is used:

```
nexttile
pcolor(x,y,z), title("(b) Interpolated Shading")
shading interp
```

You can add contours to the image by overlaying a contour plot:

```
nexttile
pcolor(x,y,z), title("(c) Overlaid Pseudo Color and Contour")
shading interp
hold on
contour(x,y,z,20,"k")
hold off
```

The same result can be obtained by using the filled contour function:

```
contourf(x,y,z,20,"k")
```

The number `20` specifies that 20 contour lines are drawn, and the `'k'` indicates that the lines should be black. If we hadn't specified black lines, they would have

been the same color as the pseudo color plot, and would have disappeared into the image. A simple contour plot was added to the figure for comparison:

```
nexttile
h = contour(x,y,z); clabel(h)
title("(d) Labeled Contour Plot")
```

Additional options for using all the three-dimensional plotting functions are included in the help window. One particularly useful option is to add labels to the contour lines, which is accomplished using graphics object names. The plot is given a name, which is used as input to the `clabel` function.

```
h = contour(x,y,z); clabel(h)
```

Finally, a title was added at the top of the chart.

```
title(t,"Pseudo Color and Contour Plots")
```

KEY IDEA
When you interactively edit a plot, your changes will be lost if you rerun the program.

5.5 EDITING PLOTS FROM THE MENU BAR

In addition to controlling the way your plots look by using MATLAB commands, you can edit a plot once you've created it. The plot in Figure 5.32 was created with the `sphere` command, which is one of several sample functions, like `peaks`, used to demonstrate plotting.

```
sphere
```

In the figure, the *Insert menu* has been selected. Notice that you can insert labels, titles, legends, text boxes, and so on, all by using this menu. The *Tools menu* allows you to change the way the plot looks, by zooming in or out, changing the aspect ratio,

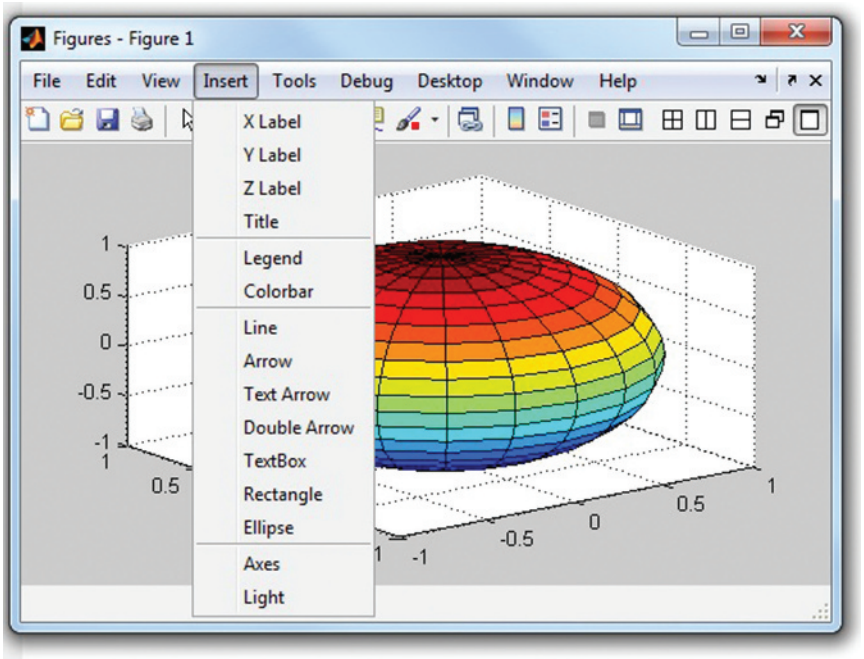


Figure 5.32
MATLAB offers interactive tools, such as the insert tool, that allow the user to adjust the appearance of graphs.

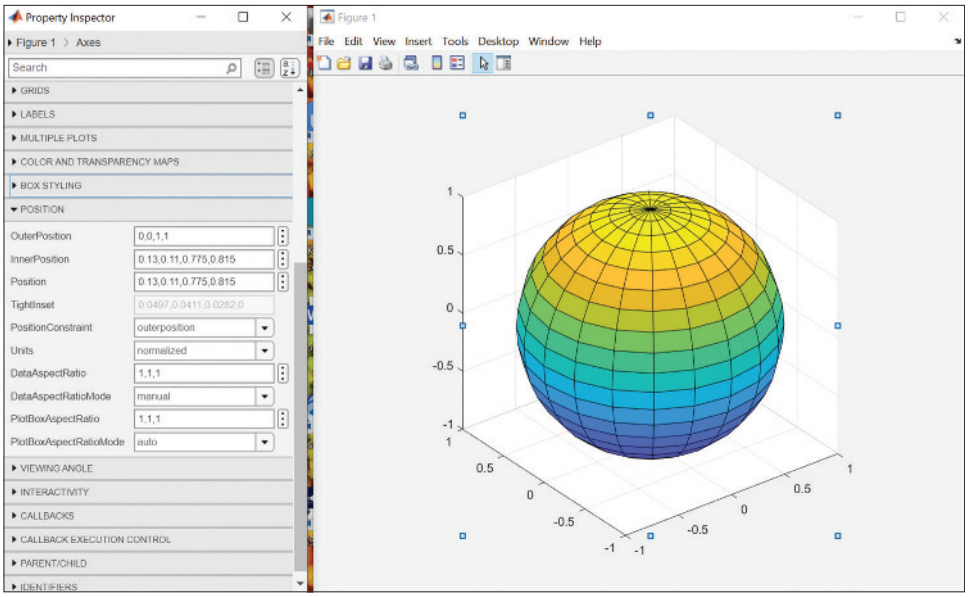


Figure 5.33
MATLAB allows you to edit plots by using commands from the toolbar.

and other features. The figure toolbar, underneath the menu toolbar, offers icons that allow you to do the same thing.

The plot in Figure 5.32 doesn't really look like a sphere; it's also missing labels and a title, and the meaning of the colors may not be clear. We edited this plot by first adjusting the shape:

- Select *Edit* → *Axes Properties* from the menu toolbar.
- From the *Property Inspector—Axes window*, select *Position* → *Data Aspect Ratio Mode*.
- Set the mode to manual (see Figure 5.33).

Similarly, labels, a title, and a color bar were added (see Figure 5.34) using the Property Editor. They could also have been added by using the *Insert menu* option on

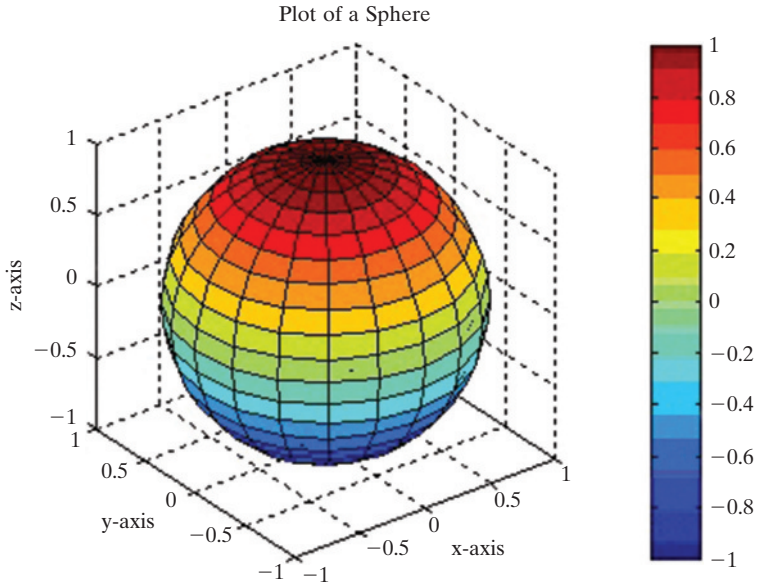


Figure 5.34
Edited plot of a sphere.

the menu bar. Editing your plot in this manner is more interactive and allows you to fine-tune the plot's appearance. The only problem with editing a figure interactively is that if you run your MATLAB program again, you will lose all of your improvements.

HINT

You can force a plot to space the data equally on all the axes by using the `axis equal` command. This approach has the advantage that you can program *axis equal* into an M-file and retain your improvements.

5.6 CREATING PLOTS FROM THE WORKSPACE WINDOW

A great feature of MATLAB is its ability to create plots interactively using the variables in the workspace window. Select the variables you would like to plot, then select the *PLOTS* tab from the MATLAB desktop (shown in Figure 5.35). MATLAB will list the plotting options it "thinks" are reasonable for the data stored in your variable(s). Simply select the appropriate option, and your plot is created in the current figure window. If you don't like any of the suggested types of plot, choose the drop-down menu, and a new window will open with the complete list of available plotting options for you to choose from. This is especially useful, because it may suggest options that had not occurred to you. For example, Figure 5.35 shows a three-dimensional **scatter plot** of the `x, y,` and `z` arrays highlighted in the figure. The arrays were created by loading the seamount data set, which is built into MATLAB.

If you want to plot more than one variable, highlight the first, then hold down the *Ctrl* key and select the additional variables. To annotate your plots, use the interactive editing process described in Section 5.5. The interactive environment is a rich resource. You'll get the most out of it by exploring and experimenting.

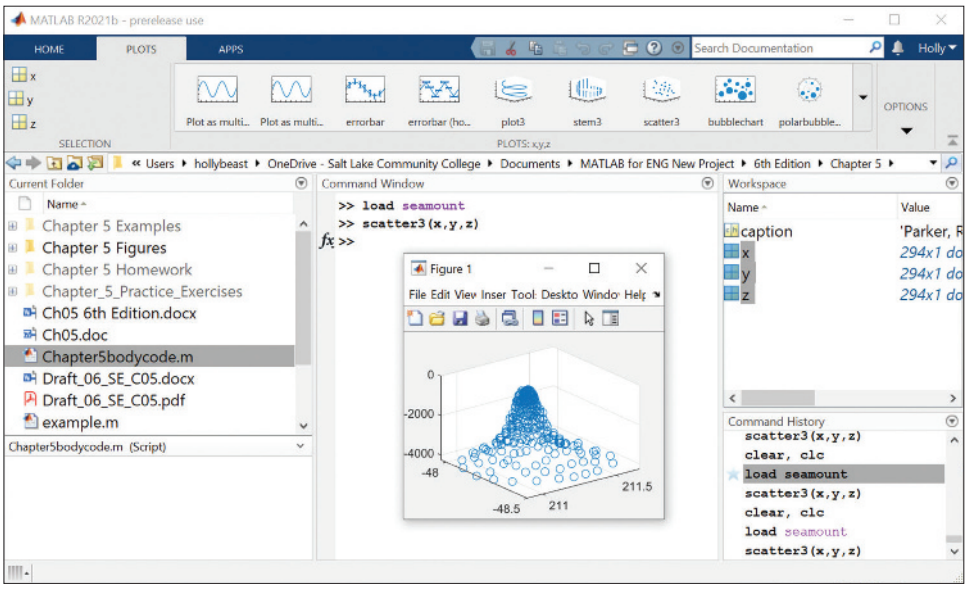


Figure 5.35
Plotting from the workspace window, using the interactive plotting feature.

5.7 SAVING YOUR PLOTS

There are several ways to save plots created in MATLAB:

- If you created the plot with programming code stored in a MATLAB program simply rerunning the code will recreate the figure.
- You can also save the figure from the file menu, using the *Save As …* option. You'll be presented with several choices:

1. You may save the figure as a *.fig* file, which is a MATLAB-specific file format. To retrieve the figure, just double-click on the file in the current folder.
2. You may save the figure in a number of different standard graphics formats, such as jpeg (*.jpg*) and enhanced metafile (*.emf*). These versions of the figure can be inserted into other documents, such as a Word document.
3. You can select Edit from the menu bar, then select *copy figure*, and paste the figure into another document.
4. You can use the file menu to create a script that will recreate the figure.

PRACTICE EXERCISE 5.6

Create a plot of $y = \cos(x)$. Practice saving the file and inserting it into a Word document. The correct answers can be found on the Pearson-website at https://www.pearsonhighered.com/moore.

5.8 OTHER PLOTTING OPTIONS

There are many additional types of plots and charts that can be created in MATLAB, and more are being added all the time. Figure 5.36 shows the variety of options available when this book was published, but to see the most recent list use the MATLAB help feature and search on 'types of MATLAB plots'. The table will link you to instructions on their use, and to a gallery where other users have posted examples.

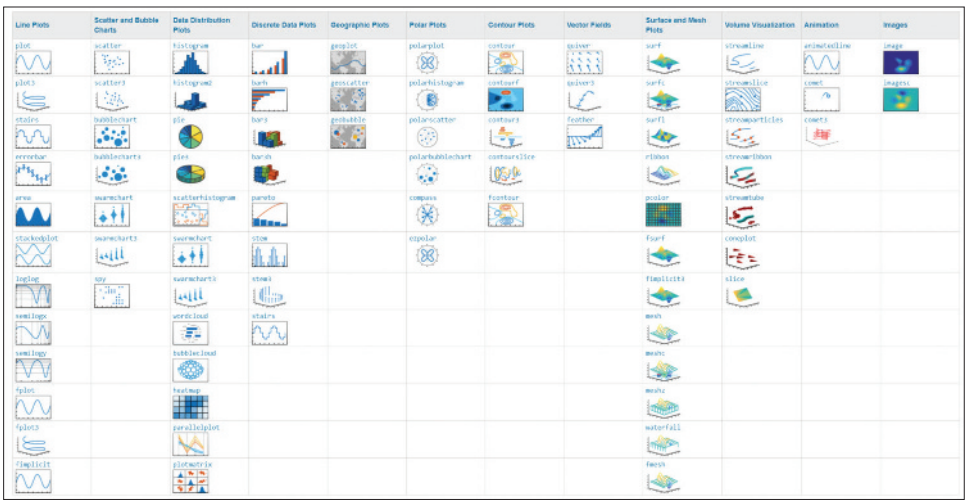


Figure 5.36
MATLAB includes a wide variety of plotting options, all of which are documented in the help feature.

SUMMARY

The most commonly used graph in engineering is the x–y plot. This two-dimensional plot can be used to graph data or to visualize mathematical functions. No matter what a graph represents, it should always include a title and axis labels. Axis labels should be descriptive and should include units, such as ft/s or kJ/kg.

MATLAB includes extensive options for controlling the appearance of your plots. The user can specify the color, line style, and marker style for each line on a graph. A grid can be added to the graph, and the axis range can be adjusted. Text boxes and a legend can be employed to describe the graph. The `tiledlayout` function is used to divide the plot window into an $m \times n$ grid. Inside each of these subwindows, any of the MATLAB plots can be created and modified.

In addition to x–y plots, MATLAB offers a variety of plotting options, including polar plots, pie charts, bar graphs, histograms, and x–y graphs with two y-axes. The scaling on x–y plots can be modified to produce logarithmic plots on either or both x- and y-axes. Engineers often use logarithmic scaling to represent data as a straight line.

The function `fplot` allows the user to plot a function without defining a vector of x- and y-values. MATLAB automatically chooses the appropriate number of points and spacing to produce a smooth graph. Additional function-plotting capability is available in the symbolic toolbox.

The three-dimensional plotting options in MATLAB include a line plot, a number of surface plots, and contour plots. Most of the options available in two-dimensional plotting also apply to these three-dimensional plots. The `meshgrid` function is especially useful in creating three-dimensional surface plots.

Interactive tools allow the user to modify existing plots. These tools are available from the figure menu bar. Plots can also be created with the interactive plotting option from the workspace window. The interactive environment is a rich resource. You'll get the most out of it by exploring and experimenting.

Figures created in MATLAB can be saved in a variety of ways, either to be edited later or to be inserted into other documents. MATLAB offers both proprietary file formats that minimize the storage space required to store figures and standard file formats suitable to import into other applications.

A wide variety of plotting options is available in MATLAB. Instructions and examples can be found using the help feature.

MATLAB SUMMARY

The following MATLAB summary lists the special characters, commands, and functions that were defined in this chapter:

Special Characters					
Line Type	**Indicator**	**Point Type**	**Indicator**	**Color**	**Indicator**
solid	-	point	.	blue	b
dotted	:	circle	o	green	g
dash-dot	-.	x-mark	x	red	r
dashed	- -	plus	+	cyan	c
		star	*	magenta	m
		square	s	yellow	y
		diamond	d	black	k
		triangle down	v	white	w
		triangle up	^		
		triangle left	<		
		triangle right	>		
		pentagram	p		
		hexagram	h		

Commands and Functions	
`autumn`	optional colormap used in surface plots
`axis`	freezes the current axis scaling for subsequent plots or specifies the axis dimensions
`axis equal`	forces the same scale spacing for each axis
`bar`	generates a bar graph
`bar3`	generates a three-dimensional bar graph
`barh`	generates a horizontal bar graph
`bar3h`	generates a horizontal three-dimensional bar graph
`bone`	optional colormap used in surface plots
`clabel`	add labels to a contour plot
`clf`	clear figure
`close`	close the current figure window
`close all`	close all open figure windows
`colorcube`	optional colormap used in surface plots
`colormap`	color scheme used in surface plots
`comet`	draws an *x*–*y* plot in a pseudo animation sequence
`comet3`	draws a three-dimensional line plot in a pseudo animation sequence
`contour`	generates a contour map of a three-dimensional surface
`contourf`	generates a filled contour map of a three-dimensional surface
`cool`	optional colormap used in surface plots
`copper`	optional colormap used in surface plots
`figure`	opens a new figure window
`flag`	optional colormap used in surface plots
`fplot`	creates an *x*–*y* plot based on a function
`gtext`	similar to text; the box is placed at a location determined interactively by the user by clicking in the figure window
`grid`	adds a grid to the current plot only
`grid off`	turns the grid off
`grid on`	adds a grid to the current and all subsequent graphs in the current figure
`histogram`	generates a histogram
`histcounts`	returns the number of data points in each bin

(*continued*)

Commands and Functions (continued)	
`hold off`	instructs MATLAB *to* erase figure contents before adding new information
`hold on`	instructs MATLAB *not to* erase figure contents before adding new information
`hot`	optional colormap used in surface plots
`hsv`	optional colormap used in surface plots
`jet`	default colormap used in surface plots
`legend`	adds a legend to a graph
`linspace`	creates a linearly spaced vector
`loglog`	generates an *x*–*y* plot with both axes scaled logarithmically
`mesh`	generates a mesh plot of a surface
`meshgrid`	places each of two vectors into separate two-dimensional arrays, the size of which is determined by the source vectors
`nexttile`	causes the program to move to the next tile available for plotting
`pause`	pauses the execution of a program until any key is hit
`pcolor`	creates a pseudo color plot similar to a contour map
`peaks`	creates a sample array used to demonstrate graphing functions
`pie`	generates a pie chart
`pie3`	generates a three-dimensional pie chart
`pink`	optional colormap used in surface plots
`plot`	creates an *x*–*y* plot
`plot3`	generates a three-dimensional line plot
`polarplot`	creates a polar plot
`prism`	optional colormap used in surface plots
`semilogx`	generates an *x*–*y* plot with the *x*-axis scaled logarithmically
`semilogy`	generates an *x*–*y* plot with the *y*-axis scaled logarithmically
`shading flat`	shades a surface plot with one color per grid section
`shading interp`	shades a surface plot by interpolation
`sphere`	sample function used to demonstrate graphing
`spring`	optional colormap used in surface plots
`summer`	optional colormap used in surface plots
`surf`	generates a surface plot
`surfc`	generates a combination surface and contour plot
`text`	adds a text box to a graph
`tiledlayout`	divides the graphics window into sections available for plotting
`title`	adds a title to a plot
`white`	optional colormap used in surface plots
`winter`	optional colormap used in surface plots
`xlabel`	adds a label to the *x*-axis
`ylabel`	adds a label to the *y*-axis
`yyaxis`	specifies which *y*-axis to use
`zlabel`	adds a label to the *z*-axis

KEY TERMS

bin
colormap
contour plot
dependent variable
handle
histogram
independent variable
line plot
ordered pairs
scatter plot
subplot
surface plot
tile
x-*y* plot

PROBLEMS

Two-Dimensional (x – y) Plots

5.1 Create plots of the following functions from $x = 0$ to 10.

(a) $y = e^x$
(b) $y = \sin(x)$
(c) $y = ax^2 + bx + c$, where $a = 5$, $b = 2$, and $c = 4$
(d) $y = \sqrt{x}$

Each of your plots should include a title, an x-axis label, a y-axis label, and a grid.

5.2 Plot the following set of data:

$$y = [12,\ 14,\ 12,\ 22,\ 8,\ 9]$$

Allow MATLAB to use the array index number as the parameter for the x-axis.

5.3 Plot the following functions on the same graph for x values from $-\pi$ to π, selecting spacing to create a smooth plot:

$$y_1 = \sin(x)$$
$$y_2 = \sin(2x)$$
$$y_3 = \sin(3x)$$

(Hint: Recall that the appropriate MATLAB syntax for $2x$ is **2* x**.)

5.4 Adjust the plot created in Problem 5.3 so that:

- Line 1 is red and dashed.
- Line 2 is blue and solid.
- Line 3 is green and dotted.
- Do not include markers on any of the graphs. In general, markers are included only on plots of measured data, not for calculated values.

5.5 Adjust the plot created in Problem 5.4 so that the x-axis goes from -6 to $+6$.

- Add a legend.
- Add a text box describing the plots.

x–y Plotting with Projectiles

Use the following information in Problems 5.6 through 5.10:
The distance a projectile travels when fired at an angle θ is a function of time, and can be divided into horizontal and vertical distances according to the formulas

$$\text{horizontal}(t) = tV_0 \cos(\theta)$$

and

$$\text{vertical}(t) = tV_0 \sin(\theta) - \tfrac{1}{2}gt^2$$

where

horizontal	=	distance traveled in the x direction,
vertical	=	distance traveled in the y direction,
V_0	=	initial velocity
g	=	acceleration due to gravity, 9.8 m/s^2, and
t	=	time, s.

5.6 Suppose the projectile just described is fired at an initial velocity of 100 m/s and a launch angle of $\pi/4(45°)$. Find the distance traveled both horizontally and vertically (in the *x* and *y* directions) for times from 0 to 20 s with a spacing of .01 s.

(a) Graph horizontal distance versus time.

(b) In a new figure window, plot vertical distance versus time (with time on the *x*-axis).

Don't forget a title and labels.

5.7 In a new figure window, plot horizontal distance on the *x*-axis and vertical distance on the *y*-axis.

5.8 Replot horizontal distance on the *x*-axis and vertical distance on the *y*-axis using the `comet` function. If the plot draws too quickly or too slowly on your computer, adjust the number of time values used in your calculations.

5.9 Calculate three new vectors for each of the vertical (v_1, v_2, v_3) and horizontal (h_1, h_2, h_3) distances traveled, assuming launch angles of $\pi/2$, $\pi/4$, and $\pi/6$.

- In a new figure window, graph horizontal distance on the *x*-axis and vertical distance on the *y*-axis, for all three cases. (You'll have three lines.)
- Make one line solid, one dashed, and one dotted. Add a legend to identify which line is which.

5.10 Recreate the plot from Problem 5.9. This time, create an array `theta` of the three angles, $\pi/2$, $\pi/4$, and $\pi/6$. Use the `meshgrid` function to create a mesh of `theta` and the time vector (`t`). Then use the two new meshed variables you create to recalculate vertical distance (v) and horizontal distance (h) traveled. Each of your results should be a 2001 × 3 array. Use the `plot` command to plot h on the *x*-axis and v on the *y*-axis.

5.11 A tensile testing machine such as the one shown in Figure P5.11(a) and (b) is used to determine the behavior of materials as they are deformed. In the typical test, a specimen is stretched at a steady rate. The force (load) required to deform the material is measured, as is the resulting deformation. An example set of data measured in one such test is shown in Table P5.11.



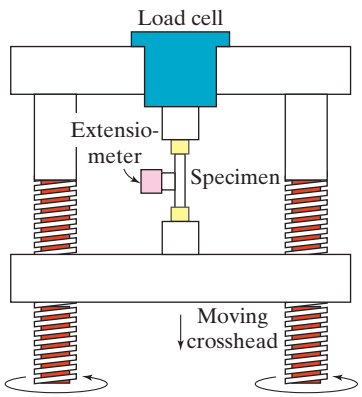


Figure P5.11
A tensile testing machine is used to measure stress and strain and to characterize the behavior of materials as they are deformed. (N_Sakarin/Shutterstock)

Table P5.11 Tensile Testing Data

Load (lbf)	Length (inches)
0	2.0000
2000	2.0024
4000	2.0047
6000	2.0070
7500	2.0094
8000	2.0128
8500	2.0183
9000	2.0308
9500	2.0500
10000	2.075

These data can be used to calculate the applied stress and the resulting strain with the following equations.

$$\sigma = \frac{F}{A} \text{ and } \varepsilon = \frac{l - l_0}{l_0}$$

where

σ = stress in (Psi) $\text{lb}_\text{f}/\text{in.}^2$,
F = applied force in lb_f,
A = sample cross-sectional area in in.^2,
ε = strain in in./in.,
l = sample length, and
l_0 = original sample length.

(a) Use the provided data to calculate the stress and the corresponding strain for each data pair. The tested sample was a rod of diameter 0.505 in., so you'll need to find the cross-sectional area to use in your calculations.
(b) Create an *x*–*y* plot with strain on the *x*-axis and stress on the *y*-axis. Connect the data points with a solid black line, and use circles to mark each data point.
(c) Add a title and appropriate axis labels.
(d) The point where the graph changes from a straight line with a steep slope to a flattened curve is called the *yield stress* or *yield point*. This corresponds to a significant change in the material behavior. Before the yield point the material is elastic, returning to its original shape if the load is removed—much like a rubber band. Once the material has been deformed past the yield point, the change in shape becomes permanent and is called plastic deformation. Use a text box to mark the yield point on your graph.

5.12 In the previous chapter, the accumulated cyclone energy index (ACE) was introduced (see Problem 4.5). Use that data to solve the following problems. It may also be available to you as an Excel spreadsheet, named *ace_data.xlsx*.

(a) Create an *x*–*y* plot of the year (on the *x*-axis) versus the ACE index values (on the *y*-axis).
(b) Calculate the mean ACE value, and use it to draw the mean value on your graph. (*Hint:* You just need two points, one at the first year and one at the final year).

(c) Use the `filter` function to find a running weighted average of the ACE data, over a 10-year period, using the following syntax, assuming you have named the data extracted from the ACE column, `ace`.

```
running_avg_ace = filter(ones(1,10)/10,1,ace);
```

Create a plot of the year (on the *x*-axis) versus the ACE value and the weighted average on the *y*-axis. (You will have two lines.) From your graph, do you think hurricane intensity is increasing? You can find out more about the `filter` function by searching the `help` documentation.

Using Tiled Chart Layouts

5.13 In Problem 5.1, you created four plots. Combine these into one figure with four subwindows, using the `tiledlayout` and `nexttile` functions.

5.14 In Problems 5.6, 5.7, and 5.9, you created a total of four plots. Combine these into one figure with four subwindows, using the `tiledlayout` and `nexttile` functions.

Polar Plots

5.15 Create a vector of angles from 0 to 2π. Use the `polarplot` plotting function to create graphs of the functions that follow. Remember, polar plots expect the angle and the radius as the two inputs to the `polarplot` function. Use the `tiledlayout` and `nexttile` functions to put all four of your graphs in the same figure.

(a) $r = \sin^2(\theta) + \cos^2(\theta)$
(b) $r = \sin(\theta)$
(c) $r = e^{\theta/5}$
(d) $r = \sinh(\theta)$

5.16 In Practice Exercise 5.3, you created a number of interesting shapes in polar coordinates. Use those exercises as a help in creating the following figures:

(a) Create a "flower" with three petals.
(b) Overlay your figure with eight additional petals, half the size of the three original ones.
(c) Create a heart shape.
(d) Create a six-pointed star.
(e) Create a hexagon.

Logarithmic Plots

5.17 When interest is compounded continuously, the following equation represents the growth of your savings:

$$P = P_0 e^{rt}$$

In this equation,

P = current balance,
P_0 = initial balance,
r = growth constant, expressed as a decimal fraction, and
t = time invested.

Determine the amount in your account at the end of each year if you invest \$1000 at 8% (0.08) for 30 years. (Make a chart.)

Create a figure with four tiles. Plot time on the x-axis, and current balance P on the y-axis.

(a) In the first quadrant, plot t versus P in a rectangular coordinate system.
(b) In the second quadrant, plot t versus P, scaling the x-axis logarithmically.
(c) In the third quadrant, plot t versus P, scaling the y-axis logarithmically.
(d) In the fourth quadrant, plot t versus P, scaling both axes logarithmically.

Which of the four plotting techniques do you think displays the data best?

5.18 According to Moore's law (an observation made in 1965 by Gordon Moore, a co-founder of Intel Corporation), the number of transistors that would fit per square inch on a semiconductor integrated circuit doubles approximately every two years. Although Moore's law is often reported as predicting doubling every 18 months, this is incorrect. A colleague of Moore took into account the fact that transistor performance is also improving, and when combined with the increased number of transistors results in doubling of *performance* every 18 months. The year 2005 was the 40th-anniversary of the law. Over the last 40 years, Moore's projection has been consistently met. In 1965, the then state-of-the-art technology allowed for 30 transistors per square inch. Moore's law says that transistor density can be predicted by $d(t) = 30(2^{t/2})$, where t is measured in years.

(a) Letting $t = 0$ represent the year 1965 and $t = 46$ represent 2011, use this model to calculate the predicted number of transistors per square inch for the 46 years from 1965 to 2011. Let t increase in increments of 2 years. Display the results in a table with two columns—one for the year, and one for the number of transistors.

(b) Using the `tiledlayout` and `nexttile` functions, plot the data in a linear x–y plot, a semilog x plot, a semilog y plot, and a log–log plot. Be sure to title the plots and label the axes.

5.19 The total number of transistor count on integrated circuits produced over the last 45 years is shown in Table P5.19. Create a semilog plot (with the y-axis scaled logarithmically) of the actual data, using circles only to indicate the data points (no lines). Include a second line representing the predicted values using Moore's law, based on the 1971 count as the starting point. Add a legend to your plot.

Table P5.19 Exponential Increase in Transistor Count on Integrated Circuits

Processor	Transistor Count	Date of Introduction	Manufacturer
Intel 4004	2300	1971	Intel
Intel 8008	2500	1972	Intel
Intel 8080	4500	1974	Intel
Intel 8088	29,000	1979	Intel
Intel 80286	134,000	1982	Intel
Intel 80386	275,000	1985	Intel
Intel 80486	1,200,000	1989	Intel
Pentium	3,100,000	1993	Intel
AMD K5	4,300,000	1996	AMD

(*continued*)

Table P5.19 (continued)

Processor	Transistor Count	Date of Introduction	Manufacturer
Pentium II	7,500,000	1997	Intel
AMD K6	8,800,000	1997	AMD
Pentium III	9,500,000	1999	Intel
AMD K6-III	21,300,000	1999	AMD
AMD K7	22,000,000	1999	AMD
Pentium 4	42,000,000	2000	Intel
Barton	54,300,000	2003	AMD
AMD K8	105,900,000	2003	AMD
Itanium 2	220,000,000	2003	Intel
Itanium 2 with 9MB cache	592,000,000	2004	Intel
Cell	241,000,000	2006	Sony/IBM/Toshiba
Core 2 Duo	291,000,000	2006	Intel
Core 2 Quad	582,000,000	2006	Intel
G80	681,000,000	2006	NVIDIA
POWER6	789,000,000	2007	IBM
Dual-Core Itanium 2	1,700,000,000	2006	Intel
Quad-Core Itanium Tukwila (processor)[1]	2,000,000,000	2008	Intel
8-Core Xeon Nehalem-EX	2,300,000,000	2010	Intel
10-Core Xeon Westmere-EX	2,600,000,000	2011	Intel
61-Core Xeon Phi	5,000,000,000	2012	Intel
18-Core Xeon Haswell-E5	5,560,000,000	2014	Intel
IBM z13 Storage Controller	7,100,000,000	2015	IBM
22-core Xeon Broadwell-E5	7,200,000,000	2016	Intel
SPARC M8	10,000,000,000	2017	Oracle
z15	9,200,000,000	2019	IBM
M1	16,000,000,000	2020	Apple

5.20 Many physical phenomena can be described by the Arrhenius equation. For example, reaction-rate constants for chemical reactions are modeled as

$$k = k_0 e^{(-Q/RT)}$$

where

k_0 = constant with units that depend upon the reaction,
Q = activation energy, kJ/kmol,
R = ideal gas constant, kJ/kmol K, and
T = temperature in K.

For a certain chemical reaction, the values of the constants are

$$\begin{aligned} Q &= 1000 \text{ J/mol} \\ k_0 &= 10 \text{ s}^{-1} \\ R &= 8.314 \text{ J/mol K} \end{aligned}$$

for T from 300 K to 1000 K. Find the values of k. Create the following two graphs of your data in a single figure window:

(a) Plot T on the x-axis and k on the y-axis.
(b) Plot your results as the $\log_{10}$ of k on the y-axis and $1/T$ on the x-axis.

Bar Graphs, Pie Charts, and Histograms

5.21 Use tiles to create the four plots in this problem. Let the vector

$$G = [68, 83, 61, 70, 75, 82, 57, 5, 76, 85, 62, 71, 96, 78, 76, 68, 72, 75, 83, 93]$$

represent the distribution of final grades in an engineering course.

(a) Use MATLAB to sort the data, and create a bar graph of the scores.
(b) Create a histogram of the scores, using `histogram`.
(c) Assume the following grading scheme is used:

$$\begin{aligned} &\text{A} > 90 - 100 \\ &\text{B} > 80 - 90 \\ &\text{C} > 70 - 80 \\ &\text{D} > 60 - 70 \\ &\text{E} > 0 - 60 \end{aligned}$$

Use the `histogram` function and an appropriate edges vector to create a histogram showing the grade distribution.

(d) Repeat part (c) but normalize the data using the `countdensity` option.

5.22 In the engineering class mentioned in Problem 5.21, there are

2 As
4 Bs
8 Cs
4 Ds
2 Es

(a) Use the `histcounts` function to create a vector called `grades` corresponding to this grade distribution, based on the grading scheme in Problem 5.21.

Create a pie chart of the `grades` vector. Add a legend listing the grade names (A, B, C, etc.).

(b) Use the interactive text option accessed from the figure window menu to add a text box to each slice of pie, with the grade shown in the text box. Save your modified graph as a *.fig* file.

(c) Create a three-dimensional pie chart of the same data, including a legend.

5.23 The National Oceanic and Atmospheric Administration (NOAA) provides climate information to the public through the National Centers for Environmental Information. You can probably find information there for your hometown or favorite vacation spot. (Perform an internet search to find the website.) Browse the available datasets and download one of the annual summaries that include monthly precipitation and temperature data. Your instructor may have the dataset for Orlando International Airport (MCO) available for you already from the Pearson website. Upload the data from the Excel spreadsheet, and create a histogram of the extreme maximum temperature data.

5.24 The inventory of a certain type of screw in a warehouse at the end of each month is listed in table P5.24:

Table P5.24 Screw Inventory

	Year 1	Year 2
January	2345	2343
February	4363	5766
March	3212	4534
April	4565	4719
May	8776	3422
June	7679	2200
July	6532	3454
August	2376	7865
September	2238	6543
October	4509	4508
November	5643	2312
December	1137	4566

Plot the data in a bar graph. Add an appropriate legend and title.

5.25 Use the `randn` function to create 1000 values in a normal (Gaussian) distribution of numbers with a mean of 70 and a standard deviation of 3.5. Create a histogram of the data set you calculated.

Graphs with Two *y*-Axes

5.26 In the introduction to Problems 5.6 through 5.10, we learned that the equations for the distance traveled by a projectile as a function of time are:

$$\text{horizontal}\ (t) = tV_0 \cos(\theta)$$
$$\text{vertical}\ (t) = tV_0 \sin(\theta) - \tfrac{1}{2}\, gt^2$$

For time from 0 to 20 s, plot both the horizontal distance versus time and the vertical distance versus time on the same graph, using separate *y*-axes for each line. Assume a launch angle of 45 degrees ($\pi/4$ radians) and an initial velocity of 100 m/s. Assume also that the acceleration due to gravity, *g*, is 9.8 m/s. Be sure to label both *y*-axes.

5.27 If the equation modeling the vertical distance traveled by a projectile as a function of time is

$$\text{vertical}\ (t) = tV_0 \sin(\theta) - 1/2\ gt^2$$

then, from calculus, the velocity in the vertical direction is

$$\text{velocity}\ (t) = V_0 \sin(\theta) - gt$$

Create a vector **t** from 0 to 20 s, and calculate both the vertical position and the velocity in the vertical direction, assuming a launch angle θ of $\pi/4$ radians and an initial velocity of 100 m/s. Plot both quantities on the same graph with separate *y*-axes. Be sure to label both *y*-axes.

The velocity should be zero at the point where the projectile is the highest in the vertical direction. Does your graph support this prediction?

5.28 For many metals, deformation changes their physical properties. In a process called *cold work*, metal is intentionally deformed to make it stronger. The data in table P5.28 lists tabulate both the strength and ductility of a metal that has been cold worked to different degrees:

Table P5.28 Effect of Cold Work on a Hypothetical Metal

Percent Cold Work	Yield Strength (MPa)	Ductility (%)
10	275	43
15	310	30
20	340	23
25	360	17
30	375	12
40	390	7
50	400	4
60	407	3
68	410	2

Plot these data on a single *x*–*y* plot with two *y*-axes. Be sure to label both *y*-axes.

Three-Dimensional Line Plots

5.29 Create a vector x of values from 0 to 20π, with a spacing of $\pi/100$. Define vectors y and z as

$$y = x \sin(x)$$

and

$$z = x \cos(x)$$

(a) Create an *x*–*y* plot of `x` and `y`.
(b) Create a polar plot of `x` and `y`.
(c) Create a three-dimensional line plot of `x`, `y`, and `z`. Don't forget a title and labels.

5.30 Figure out how to adjust your input to `plot3` in Problem 5.29 to create a graph that looks like a tornado (see Figure P5.30). Use `comet3` instead of `plot3` to create the graph.

Three-Dimensional Surface and Contour Plots

5.31 Create `x` and `y` vectors from -5 to $+5$ with a spacing of 0.5. Use the `meshgrid` function to map `x` and `y` onto two new two-dimensional arrays called `X` and `Y`. Use your new arrays to calculate array Z, with magnitude

$$Z = \sin\left(\sqrt{X^2 + Y^2}\right)$$

(a) Use the `mesh` plotting function to create a three-dimensional plot of Z.
(b) Use the `surf` plotting function to create a three-dimensional plot of Z. Compare the results you obtain with a single input (Z) with those obtained with inputs for all three dimensions (X, Y, Z).

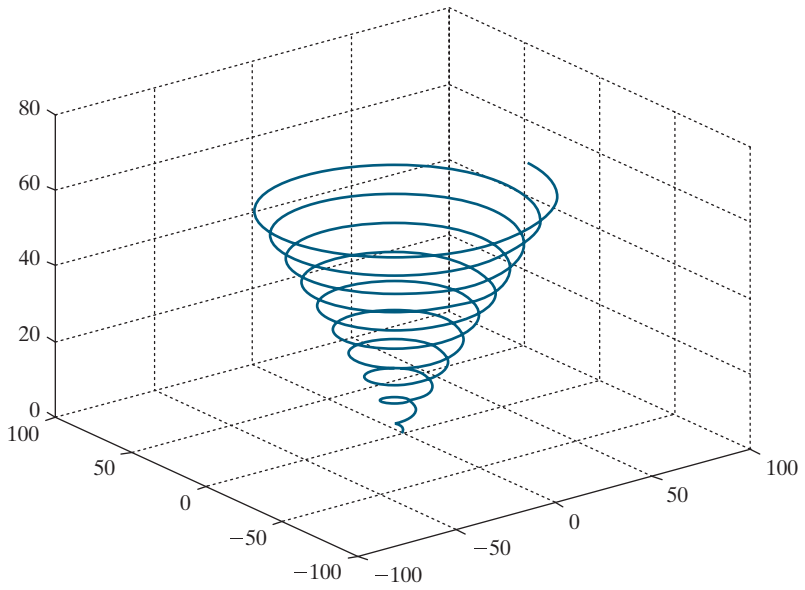


Figure P5.30
Tornado plot.

(c) Modify your surface plot with interpolated shading. Try using different `colormaps`.

(d) Generate a contour plot of *Z*. Use the `clabel` function to label the contour lines.

(e) Generate a combination surface and contour plot of *Z*.

Challenge Problems

5.32 In Problem 5.31, you generated five different plots. Use subplots to plot the first four into a rectangular grid, much like we did in earlier problems. Consult the MATLAB help feature to figure out how to add the fifth graph on top of the first four plots.

```
doc tiledlayout
```

The international sunspot number is a measure of sunspot activity first proposed by Rudolf Wolf in 1848. It is sometimes called the Wolf number or the Zurich number, since Wolf worked in Zurich, Switzerland. Based on recorded historical observations, the number has been calculated back to 1700 with annual values. There are daily values since 1818. There are other techniques for measuring sunspot activity, but this strategy has been around the longest.

Included with MATLAB is a sunspot dataset. To download it, type

```
load sunspot
```

The sunspot variable will be created in the workspace. Column 1 is the year, and column 2 is the international sunspot number. Plot the data, including *x*- and *y*- axis labels, a tile, and a grid.

The data end in 1987. Search the internet and either

(a) find the data to complete the series to the present day, merge it with your data set and replot, or

(b) find a new data set based on one of the new strategies for measuring sunspot activity and create a plot.

(c) Whatever you do, add annotations to document your plot appropriately.

(d) Cyclical data such as sunspots can be analyzed with Fourier transforms. Use the MATLAB help function to search for the sunspot example, and use the procedure suggested to utilize the `fft` function (fast Fourier transform) to determine the periodicity of the sunspot data.

CHAPTER

6 User-Defined Functions

Objectives

After reading this chapter, you should be able to:

- Create and use your own MATLAB functions with both single and multiple inputs and outputs.
- Store and access your own functions in toolboxes.
- Create and use anonymous functions.
- Create and use function handles.
- Create and use subfunctions and **nested** subfunctions.

INTRODUCTION

The MATLAB programming language is built around functions. A **function** is a piece of computer code that accepts an input **argument** from the user and provides output to the program. Functions allow us to program efficiently, enabling us to avoid rewriting the computer code for calculations that are performed frequently. For example, most computer programs contain a function that calculates the sine of a number. In MATLAB, `sin` is the **function name** used to call up a series of commands that perform the necessary calculations. The user needs to provide an angle, and MATLAB returns a result. It isn't necessary for the programmer to know how MATLAB calculates the value of $\sin(x)$.

6.1 CREATING FUNCTION FILES

We have already explored many of MATLAB's built-in functions, but you may wish to define your own functions—those that are used commonly in your programming. User-defined functions are stored either as scripts or live scripts (M-files or MLX files) and can be accessed by MATLAB if they are in the current folder, or if they are located on the MATLAB search **path**. The search path is defined by default settings but can be modified based on user settings. If user-defined functions will not be used outside of a single program file, they can be stored at the end of either a script or a live script.

In that case, they are known as local functions. Local functions are only visible to other functions or code in the same file.

6.1.1 Syntax

KEY IDEA
A local function is only visible to other functions or code in the same file.

Both built-in MATLAB functions and user-defined MATLAB functions have the same structure. Each consists of a name, user-provided input, and calculated output. For example, the function

```
cos(x)
```

- is named `cos`,
- takes the user input inside the parentheses (in this case, `x`), and
- calculates a result.

The user does not see the calculations performed, but just accepts the answer. User-defined functions work the same way. Imagine that you have created a function called `my_function`. Using

```
my_function(x)
```

in a program or from the command window will return a result, as long as `x` is defined and the logic in the function definition works.

KEY IDEA
Functions allow us to program more efficiently.

Each function must start with a function-definition line that contains the following:

- The word `function`
- A variable that defines the function output
- A function name
- A variable used for the input argument

For example,

```
function output = my_function(x)
```

is the first line of the user-defined function called `my_function`. It requires one input argument, which the program will call `x`, and will calculate one output argument, which the program will call `output`. The function name and the names of the input and output variables are arbitrary and are selected by the programmer. Here's an example of an appropriate first line for a function called `calculation`:

```
function result = calculation(a)
```

In this case, the function name is `calculation`, the input argument will be called `a` in any calculations performed in the function program, and the output will be called `result`. Although any valid MATLAB names can be used, it is good programming practice to use meaningful names for all variables, and for function names.

KEY IDEA
A function is a piece of computer code that accepts an input, performs a calculation, and provides an output.

HINT

Students are often confused about the use of the word *input* as it refers to a function. We use it here to describe the input argument—the value that goes inside the parentheses when we call a function. In MATLAB, input arguments are different from the input command, introduced in a later chapter.

Here's an example of a very simple MATLAB function that calculates the value of a particular polynomial:

```
function output = poly(x)
% This function calculates the value of a 3rd-order polynomial
output = 3*x.^3 + 5*x.^2 - 2*x +1;
end
```

KEY IDEA
It is good programming practice to always end a function with an end statement.

The function name is `poly`, the input argument is `x`, and the output variable is named `output`. The last line of your function should always be an `end` statement. This is not strictly required for functions that are stored as separate scripts or live scripts, but it is a requirement if the function is stored at the end of another program.

Functions Stored as Separate Files

Because there are multiple ways to store functions, we'll start with examples where the functions are stored separately as scripts or live scripts in the current **directory**, and then move on to other strategies. This approach allows groups of functions to be stored together and to be accessed from different programs as needed.

The **file name** *must be the same* as the function name in order for MATLAB to find it. All of the MATLAB naming conventions for naming variables apply to naming user-defined functions. In particular,

KEY IDEA
Name functions using the standard MATLAB naming conventions for variables.

- The function name must start with a letter.
- It can consist of letters, numbers, and the underscore.
- Reserved names cannot be used.
- Any length is allowed, although long names are not a good programming practice.

Once the function has been saved, it is available for use from the command window, from a script, live script, or from another function. You cannot execute a function directly from the function itself. This makes sense, since the input parameters have not been defined until you call the function. Consider the `poly` function just created. If, in the command window, we type

```
poly(4)
```

then MATLAB responds with

```
ans =
      265
```

If we set `a` equal to 4 and use `a` as the input argument, we get the same result:

```
a = 4;
poly(a)
ans =
      265
```

If we define a vector, we get a vector of answers. Thus,

```
y = 1:5;
poly(y)
```

gives

```
ans =
      7    41    121    265    491
```

If, however, you try to execute the function by selecting the *Save-and-run* icon from the function menu bar, the following error message is displayed:

```
Not enough input arguments.
Error in poly (line 3)
output = 3*x.^3 + 5*x.^2 - 2*x +1;
```

The value of the input argument must be passed to the function when it is used.

Functions Stored as Parts of Larger Programs

If a function will only be used by a single program, it makes more sense to store it inside that program rather than creating a separate file. This facilitates modular programming strategies, and it keeps all of the "pieces" of the code in one place. Functions must be stored at the end of the file, and must each be terminated with an end statement.

A good use for this strategy would be the first three practice exercises in Practice 6.1. You are unlikely to need to reuse the functions created. However, the next five exercises ask you to create functions for common conversions, which you might want to use again. It makes sense to store those in separate files where they can be used for other projects.

HINT

While you are creating a function, it may be useful to allow intermediate calculations to print to the command window. However, once you complete your "debugging," make sure that all your output is suppressed. If you don't, you'll see extraneous information in the command window.

KEY IDEA
Suppress the output from intermediate calculations performed in functions.

PRACTICE EXERCISE 6.1

Create MATLAB functions to evaluate the following mathematical functions (make sure you select meaningful function names) and test them. To test your functions, call them from the command window, or use them in a script or a live script. Each function requires its own file or it must be stored at the very end of a MATLAB program file—either a script or live script. Each function requires its own file. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. $y(x) = x^2$
2. $y(x) = e^{1/x}$
3. $y(x) = \sin(x^2)$

Create MATLAB functions for the following unit conversions (you may need to consult a textbook or the Internet for the appropriate conversion factors). Be sure to test your functions, either from the command window, or by using them in a MATLAB program.

1. Inches to feet
2. Calories to joules
3. Watts to BTU/hr
4. Meters to miles
5. Miles per hour (mph) to ft/s

EXAMPLE 6.1

CONVERTING BETWEEN DEGREES AND RADIANS

Engineers usually measure angles in degrees, yet most computer programs and many calculators require that the input to trigonometric functions be in radians. Write and test a function `DR` that changes degrees to radians, and another function `RD` that changes radians to degrees. Your functions should be able to accept both scalar and array input. Because you will be using these functions regularly store them as separate script M-files or live script MLX-files.

1. State the problem
   Create and test two functions, `DR` and `RD`, to change degrees to radians and radians to degrees respectively.

2. Describe the input and output

Input	A vector of degree values
	A vector of radian values
Output	A table converting degrees to radians
	A table converting radians to degrees

3. Develop a hand example

   $$\text{degrees} = \text{radians} \times 180/\pi$$
   $$\text{radians} = \text{degrees} \times \pi/180$$

Degrees to Radians	
Degrees	**Radians**
0	0
30	$30(\pi/180) = \pi/6 = 0.524$
60	$601\,\pi/1802 = \pi/3 = 1.047$
90	$901\,\pi/1802 = \pi/2 = 1.571$

4. Develop a MATLAB solution which could be stored as either a script or a live script. In this example we chose a script M-file. The MATLAB code is exactly the same in either case.

```
%% Example 6.1
clear, clc
% Define a vector of degree values
degrees = 0:15:180;
%% Call the DR function, and use it to find radians
radians = DR(degrees);
% Create an array to use in the output
degrees_radians = [degrees;radians]'
% Define a vector of radian values
radians = 0:pi/12:pi;
%% Call the RD function, and use it to find degrees
degrees = RD(radians);
radians_degees = [radians;degrees]'
```

The functions called by the program are

```
function output = DR(x)
```

```
% This function changes degrees to radians
output=x*pi/180;
end
```

and

```
function output=RD(x)
% This function changes radians to degrees
output = x*180/pi;
end
```

In order for the script to find the functions, they must be in the current **folder** and must be named *DR.m* and *RD.m* or *DR.mlx* and *RD.mlx* depending on whether the function was saved as a script or a live script. Either format can be called from any MATLAB program, independent of whether it is a script or a live script. The program generates the following results:

```
degrees_radians =
    0     0.000
    15    0.262
    30    0.524
    45    0.785
    60    1.047
    75    1.309
    90    1.571
    105   1.833
    120   2.094
    135   2.356
    150   2.618
    165   2.880
    180   3.142

radians_degrees =
    0.000     0.000
    0.262    15.000
    0.524    30.000
    0.785    45.000
    1.047    60.000
    1.309    75.000
    1.571    90.000
    1.833   105.000
    2.094   120.000
    2.356   135.000
    2.618   150.000
    2.880   165.000
    3.142   180.000
```

5. Test the solution
Compare the MATLAB solution with the hand solution. Since the output is an array, it is easy to see that the conversions generated by MATLAB correspond to those calculated by hand.

EXAMPLE 6.2

ASTM GRAIN SIZE

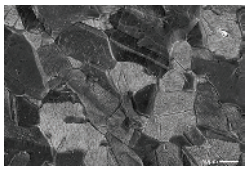


Figure 6.1
Microstructure of iron. (Sheri Neva/Image Source/Alamy Stock Photo)

You may not be used to thinking of metals as crystals, but they are. If you look at a polished piece of metal under a microscope, the structure becomes clear, as seen in Figure 6.1. Every crystal (called a grain in metallurgy) is a different size and shape. The size of the grains affects the metal's strength; the finer the grains, the stronger the metal.

Because it is difficult to determine an "average" grain size, a standard technique has been developed by ASTM (formerly known as the American Society for Testing and Materials, but now known just by its initials). A sample of metal is examined under a microscope at a magnification of 100, and the number of grains in 1 square inch is counted. The parameters are related by

$$N = 2^{n-1}$$

where n is the ASTM grain size, and N is the number of grains per square inch at 100×. The equation can be solved for n to give

$$n = \frac{(\log(N) + \log(2))}{\log(2)}$$

This equation is not hard to use, but it's awkward. Instead, create a MATLAB function called `grain_size`. This function will be used by multiple programs in the future, so we'll store it in a separate file.

1. State the problem
   Create and test a function called `grain_size` to determine the ASTM grain size of a piece of metal.

2. Describe the input and output
   To test the function, choose an arbitrary number of grains. For example:

   Input 16 grains per square inch at

   Output ASTM grain size

3. Develop a hand example

$$n = \frac{(\log(N) + \log(2))}{\log(2)}$$

$$n = \frac{(\log(16) + \log(2))}{\log(2)} = 5$$

4. Develop a MATLAB solution
   The function, created in a separate file, is

```
function output = grain_size(N)
% Calculates the ASTM grain size n
output = (log10(N) + log10(2))./log10(2);
end
```

   which was saved as *grain_size.m* in the current folder. To use this function, we can call it from the command window:

```
grain_size(16)
ans =
      5
```

5. Test the solution
The MATLAB solution is the same as the hand solution. It might be interesting to see how the ASTM grain size varies with the number of grains per square inch. We could use the function with an array of values and plot the results in Figure 6.2.

```
%% Example 6.2
% ASTM Grain Size
N = 1:100;
n = grain_size(N);
plot(N,n)
title("ASTM Grain Size")
xlabel("Number of grains per square inch at 100x")
ylabel("ASTM Grain Size")
grid
```

As expected, the grain size increases as the number of grains per square inch increases.

If we had chosen to store the function at the end of a program file as a local function, as shown in Figure 6.3, it would not be available except to function calls inside the program. This is often not a problem, if the function is only intended for use inside that program. It is important to remember that functions must be stored at the end of the program and must be terminated with an end statement.

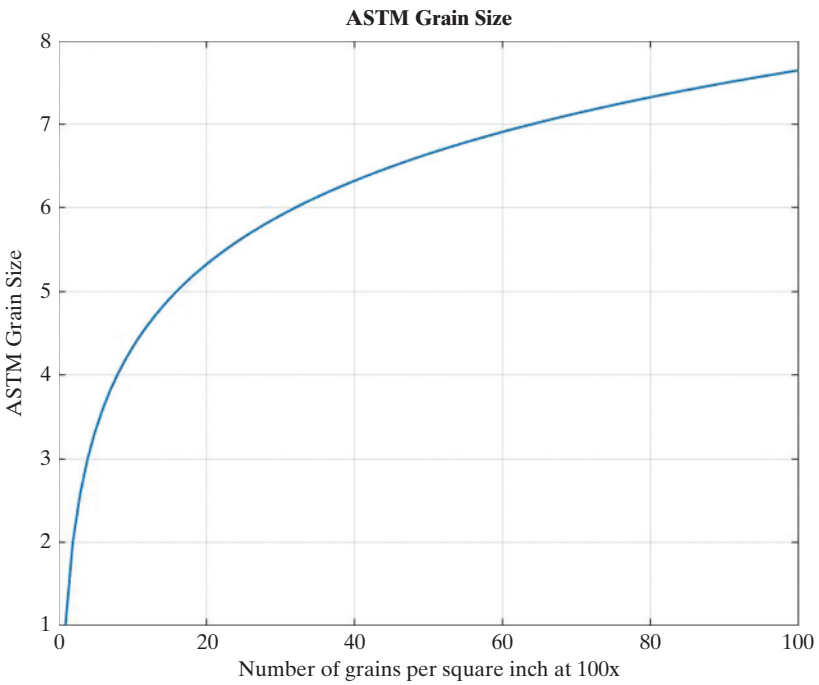


Figure 6.2
A plot of a function's behavior is a good way to help determine whether you have programmed it correctly.

(*continued*)

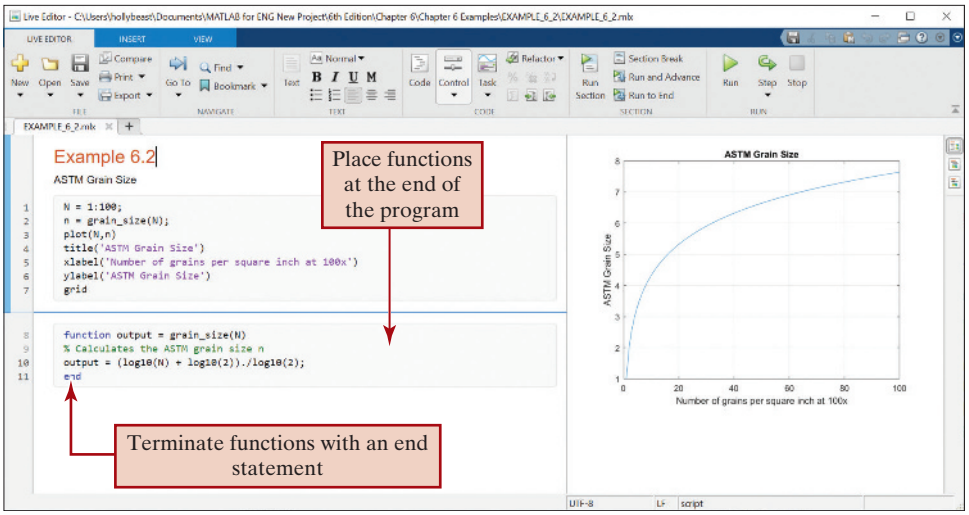


Figure 6.3
When functions are stored in a program file, they are only available for use inside that program.

6.1.2 Comments

KEY IDEA
Function comments are displayed when you use the help feature.

As with any computer program, you should **comment** your code liberally so that it is easy to follow. However, in a MATLAB function, the comments on the line immediately following the very first line serve a special role. These lines are returned when the `help` function is queried from the command window. Consider, for example, the following function:

```
function results = f(x)
% This function converts seconds to minutes
results = x./60;
end
```

Querying the `help` function from the command window

```
help f
```

returns

```
This function converts seconds to minutes
```

6.1.3 Functions with Multiple Inputs and Outputs

Just as the predefined MATLAB functions may require multiple inputs and may return multiple outputs, more complicated user-defined functions can be written. Recall, for example, the remainder function. This predefined function calculates the remainder in a division problem and requires the user to input the dividend and the divisor. For the problem $\frac{5}{3}$, the correct syntax is

```
rem(5,3)
```

which gives

```
ans =
     2
```

Similarly, a user-defined function could be written to multiply two vectors together:

```
function output = g(x,y)
% This function multiplies x and y together
% x and y must be the same size arrays
a = x.*y;
output = a;
end
```

When `x` and `y` are defined in the command window and the function `g` is called, a vector of output values is returned:

```
x = 1:5;
y = 5:9;
g(x,y)
ans =
     5    12    21    32    45
```

You can use the comment lines to let users know what kind of input is required and to describe the function. In this example, an intermediate calculation (`a`) was performed, but the only output from this function is the variable we've named `output`. This output can be an array containing a variety of numbers, but it's still only one variable.

You can also create functions that return more than one output variable. Many of the predefined MATLAB functions return more than one result. For example, `max` returns both the maximum value in an array and the index number at which the maximum occurs. To achieve the same result in a user-defined function, make the output an array of answers instead of a single variable, as in

```
function [dist, vel, accel] = motion(t)
% This function calculates the distance, velocity, and
% acceleration of a particular car for a given value of t
% assuming all 3 parameters are initially 0.
accel = 0.5.*t;
vel = t.^2/4;
dist = t.^3/12;
end
```

Once saved as **motion.m** in the current folder, you can use the function to find values of `distance`, `velocity`, and `acceleration` at specified times:

```
[distance, velocity, acceleration] = motion(10)
distance =
       83.33
velocity =
       25
acceleration =
       5
```

If you call the `motion` function without specifying all three outputs, only the first output will be returned:

```
motion(10)
ans =
        83.3333
```

Remember, all variables in MATLAB are arrays, so it's important in the preceding example to use the .* operator, which specifies element-by-element multiplication. For example, using a vector of time values from 0 to 30 in the motion function

```
time = 0:10:30;
[distance, velocity, acceleration] = motion(time)
```

returns three vectors of answers:

```
distance =
       0  83.333  666.67  2250
velocity =
       0  25  100  225
acceleration =
       0  5  10  15
```

It's easier to see the results if you group the vectors together, as in

```
results = [time',distance',velocity',acceleration']
```

which returns

```
results =
       0         0         0         0
      10    83.333        25         5
      20    666.67       100        10
      30    2250         225        15
```

Because `time`, `distance`, `velocity`, and `acceleration` were row vectors, the transpose operator was used to convert them into columns.

PRACTICE EXERCISE 6.2

Assuming that the array dimensions agree, create and test MATLAB functions to evaluate the following simple mathematical functions with multiple input vectors and a single output vector. Because these functions will not be used in any other program, store your functions at the end of your program file (either a script or a live script). In this case, they become local functions. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. $z(x, y) = x + y$
2. $z(a, b, c) = ab^c$
3. $z(w, x, y) = we^{(x/y)}$
4. $z(p, t) = p/\sin(t)$

Assuming that the array dimensions agree, create and test MATLAB functions to evaluate the following simple mathematical functions with a single input vector and multiple output vectors:

5. $f(x) = \cos(x)$
   $f(x) = \sin(x)$
6. $f(x) = 5x^2 + 2$
   $f(x) = \sqrt{5x^2 + 2}$
7. $f(x) = \exp(x)$
   $f(x) = \ln(x)$

Assuming that the array dimensions agree, create, and test MATLAB functions to evaluate the following simple mathematical functions with multiple input vectors and multiple output vectors:

8. $f(x, y) = x + y$
$f(x, y) = x - y$

9. $f(x, y) = ye^x$
$f(x, y) = xe^y$

EXAMPLE 6.3

HOW GRAIN SIZE AFFECTS METAL STRENGTH: A FUNCTION WITH THREE INPUTS

Metals composed of small crystals are stronger than metals composed of fewer large crystals. The metal yield strength (the amount of stress at which the metal starts to permanently deform) is related to the average grain diameter by the *Hall–Petch equation*:

$$\sigma = \sigma_0 + Kd^{-1/2}$$

where the symbols σ_0 and K represent constants that are different for every metal.

Create a function called `HallPetch` that requires three inputs—σ_0, K, and d—and calculates the value of yield strength. Call this function from a MATLAB program that supplies values of σ_0 and K, then plots the value of yield strength for values of d from 0.1 to 10 mm.

1. State the problem
Create a function called `HallPetch` that determines the yield strength of a piece of metal, using the Hall–Petch equation. Use the function to create a plot of yield strength versus grain diameter.

2. Describe the input and output

Input $K = 9600$ psi$\sqrt{\text{mm}}$
$\sigma = 1200$ psi
$d = 0.1$ to 10 mm

Output Plot of yield strength versus grain diameter

3. Develop a hand example

The Hall–Petch equation is

$$\sigma = \sigma_0 + Kd^{-1/2}.$$

Substituting values of 12,000 psi and 9600 psi$\sqrt{\text{mm}}$ for σ_0 and K, respectively, then

$$\sigma = 12{,}000 + 9600d^{-1/2}$$

For $d = 1$ mm,

$$\sigma = 12{,}000 + 9600 = 21{,}600 \text{ psi}$$

(*continued*)

4. Develop a MATLAB solution
In this example we chose to save the function as a live script, since the choice is simply a matter of style. The MATLAB code is exactly the same.

```
function output = HallPetch(sigma0,k,d)
% Hall-Petch equation to determine the yield
% strength of metals
output = sigma0 + K*d.^(-0.5);
end
```

The function was saved as *HallPetch.mlx* in the current folder. The decision was made to create a separate function file because this function will be used by multiple programs in the future.

Once the function was created a MATLAB program was used to call the function and create a plot.

```
%% Example 6.3
clear, clc
s0 = 12000; K = 9600 ;
% Define the values of grain diameter
diameter = 0.1:0.1:10;
yield = HallPetch(s0,K,diameter);
% Plot the results
figure(1)
plot(diameter,yield)
title("Yield Strengths Found with the Hall-Petch Equation")
xlabel("diameter, mm"),ylabel("yield strength, psi")
```

The graph shown in Figure 6.4 was generated by the program.

5. Test the solution
We can use the graph to compare the results to the hand solution.

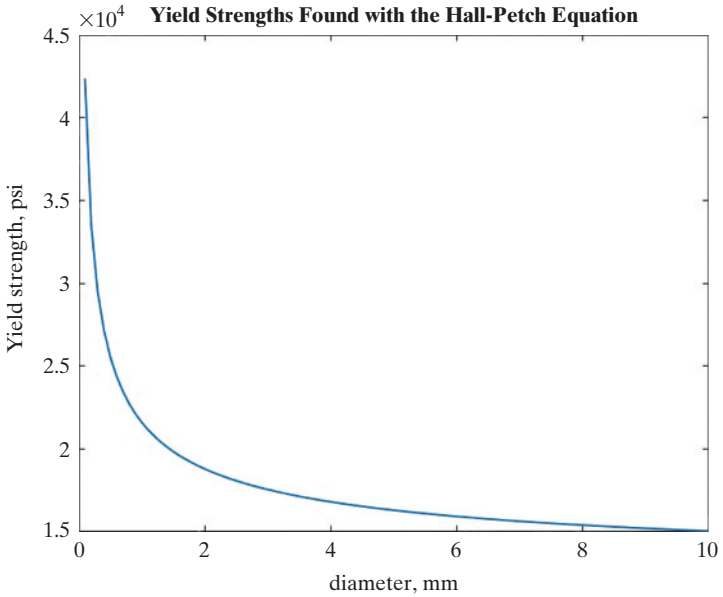


Figure 6.4
Yield strengths predicted with the Hall–Petch equation. Small grain diameters correspond to large values of the yield strength.

EXAMPLE 6.4

KINETIC ENERGY: A FUNCTION WITH TWO INPUTS

The kinetic energy of a moving object (see Figure 6.5) is

$$\text{KE} = 1/2\ mv^2.$$

Create and test a function called `KE` to find the kinetic energy of a moving car, if you know the mass m and the velocity v of the vehicle.



Figure 6.5
Race cars store a significant amount of kinetic energy. (Alex Zanardi in Target/Ganassi Car./Getty Images). (Alexey Kuznetsov/123RF)

1. State the problem
   Create a function called `KE` to find the kinetic energy of a car.

2. Describe the input and output

   Input Mass of the car, in kilograms
   Velocity of the car, in m/s

   Output Kinetic energy, in joules

3. Develop a hand example
   If the mass is 1000 kg, and the velocity is 25 m/s, then

$$\text{KE} = \tfrac{1}{2} \times 1000\ \text{kg} \times (25\ \text{m/s})^2 = 312{,}500\ \text{J} = 312.5\ \text{kJ}.$$

4. Develop a MATLAB solution
   This function probably won't be used in multiple programs, so it makes sense to simply position it at the end of the test program, without creating a separate program file.

```
function output = ke(mass,velocity)
output = 1/2*mass*velocity.^2;
end
```

5. Test the solution
   Since this is a simple function and is unlikely to be used by multiple programs, it makes sense to store it at the end of the script or live script where it is needed.

```
%% Example 6.4
```

   Define test input

```
v = 25; % m/s
m = 1000; % kg
kinetic_energy = ke(m,v)
```

   Function

```
function output = ke(mass,velocity)
output = 1/2*mass*velocity.^2;
end
```

   This result matches the hand example, confirming that the function works correctly.

6.1.4 Functions with No Input or No Output

Although most functions need at least one input and return at least one output value, in some situations no inputs or outputs are required. For example, consider this function, which draws a star in polar coordinates:

```
function [] = star( )
theta = pi/2:0.8*pi:4.8*pi;
r = ones(1,6);
polarplot(theta,r)
end
```

The square brackets on the first line indicate that the output of the function is an empty array (i.e., no value is returned). The empty parentheses tell us that no input is expected. If, from the command window, you type

```
star
```

then no values are returned, but a figure window opens showing a star drawn in polar coordinates (see Figure 6.6a). Adding

KEY IDEA
Not all functions require an input.

```
axis off
```

to the function definition removes the grid, as shown in Figure 6.6b.)

There are numerous built-in MATLAB functions that do not require any input. For example,

```
A = clock
```

returns the current time:

```
A =
  Columns 1 through 4
    2021     6     9    16
  Columns 5 through 6
    18       4.122
```

If your output doesn't look like the results shown above try changing your format to shortg.

```
format shortg
```

Also,

```
A = pi
```

returns the value of the mathematical constant π:

```
A =
   3.1416
```

HINT

You may ask yourself whether the `star` function is really an example of a function that does not return an output; after all, it does draw a star. But the output of a function is defined as a *value* that is returned when you call the function. If we ask MATLAB to perform the calculation

```
A = star
```

an error statement is generated, because the `star` function does not return anything! Thus, there is nothing to set `A` equal to.

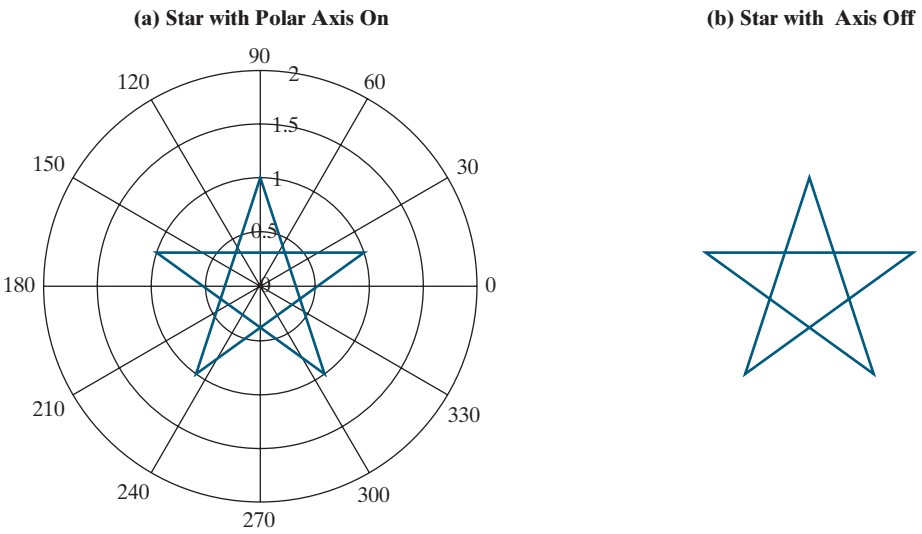


Figure 6.6
The user-defined function star requires no input and produces no output values, but it does draw a star in polar coordinates. (a) The graph plots with the polar axes visible. (b) Issuing the axis off command suppresses the axis and allows the star to render without the surrounding axes visible.

6.1.5 Determining the Number of Input and Output Arguments

There may be times when you want to know the number of input arguments or output values associated with a function. MATLAB provides two built-in functions for this purpose.

KEY IDEA
Using the `nargin` or `nargout` functions is useful in programming functions with variable inputs and outputs.

The `nargin` function determines the number of input arguments in either a user-defined function or a built-in function. The name of the function may be specified as a string, as, for example, in

```
nargin("sin")
ans =
    1
```

or as a **function handle**. Function handles will be introduced later in this chapter. The remainder function, `rem`, requires two inputs; thus,

```
nargin("rem")
ans =
    2
```

KEY IDEA
The input to most MATLAB functions that accept text can be either character arrays or string arrays—so you may use either single or double quotes.

When `nargin` is used inside a user-defined function, it determines how many input arguments were actually entered. This allows a function to have a variable number of inputs. Recall graphing functions such as `surf`. When `surf` has a single array input, a graph is created, using the array index numbers as the x- and y-coordinates. When there are three inputs, x, y, and z, the graph is based on the specified x- and y-values. The `nargin` function allows the programmer to determine how to create the plot, based on the number of inputs.

The `surf` function is an example of a function with a variable number of inputs. If we use `nargin` from the command window to determine the number of declared inputs, there isn't one correct answer. The `nargin` function returns a negative number to let us know that a variable number of inputs are possible:

```
nargin("surf")
ans =
   -1
```

The `nargout` function is similar to `nargin`, but it determines the number of outputs from a function:

```
nargout("sin")
ans =
    1
```

The number of outputs is determined by how many arrays are returned, not how many values are in the array. We know that `size` returns the number of rows and columns in an array, so we might expect `nargout` to return 2 when applied to size. However,

```
nargout("size")
ans =
    1
```

returns only one array, which has just two elements, as, in

```
x = 1:10;
size(x)
ans =
   1 10
```

An example of a function with multiple outputs is `max`:

```
nargout("max")
ans =
    2
```

When used inside a user-defined function, `nargout` determines how many outputs have been requested by the user. Consider this example, in which we have rewritten the function from Section 6.1.4 to create a star:

```
function A = star1( )
theta = pi/2:0.8*pi:4.8*pi;
r = ones(1,6);
polarplot(theta,r)
axis off
if nargout==1
   A = "Twinkle twinkle little star";
end
end
```

If we use `nargout` from the command window, MATLAB tells us that one output is specified.

```
nargout("star1")
ans =
    1
```

If we call the function simply as

```
star1
```

nothing is returned to the command window, although the plot is drawn. If we call the function by setting it equal to a variable, as in

```
x = star1
x =
  "Twinkle twinkle little star"
```

a value for x is returned, based on the if statement embedded in the function, which used nargout to determine the number of output values.

If statements will be introduced in Chapter 8.

6.1.6 Local Variables

The variables used in functions are known as **local variables**. The only way a function can communicate with the workspace is through input arguments and the output it returns. Any variables defined within the function exist only for the function to use. For example, consider the g function previously described:

```
function output = g(x,y)
% This function multiplies x and y together
% x and y must be the same size arrays
a = x.*y;
output = a;
end
```

KEY IDEA
A local variable only has meaning inside a program or function.

The variables a, x, y, and output are local variables. They can be used for additional calculations inside the g function, but they are not stored in the workspace. To confirm this, clear the workspace and the command window, then call the g function:

```
clear, clc
g(10,20)
```

The function returns

```
g(10,20)
ans =
    200
```

Notice that the only variable stored in the workspace window is ans, which is characterized as follows:

Name	Value	Size	Bytes	Class
ans	200	1×1	8	double array

Just as calculations performed in the command window or from a script cannot access variables defined in functions, functions cannot access the variables defined in the workspace. This means that functions must be completely self-contained. The only way they can get information from your program is through the input arguments, and the only way they can deliver information is through the function output.

Consider a function written to find the distance an object falls due to gravity:

```
function result = distance(t)
% This function calculates the distance a falling object
% travels due to gravity
g = 9.8 % meters per second squared
result = 1/2*g*t.^2;
end
```

The value of g must be included *inside* the function. It doesn't matter whether g has or has not been used in the main program. How g is defined is hidden to the distance function, unless g is specified inside the function.

Of course, you could also pass the value of g to the function as an input argument:

```
function result = distance(g,t)
% This function calculates the distance a falling object
% travels due to gravity
result = 1/2*g*t.^2;
end
```

HINT

The same array names can be used in both a function and the program that references it. However, they do not *have* to be the same. Since variable names are local to either the function or the program that calls the function, the variables are completely separate. As a beginning programmer, you would be wise to use different variable names in your functions and your programs—just so you don't confuse *yourself*.

KEY IDEA
It is usually a bad idea to define global variables.

6.1.7 Global Variables

Unlike local variables, **global variables** are available to all parts of a computer program. In general, *it is a bad idea* to define global variables. However, MATLAB protects users from unintentionally using a global variable by requiring that it be identified both in the command-window environment (or in a program file) and in the function that will use it.

Consider the distance function once again:

```
function result = distance(t)
% This function calculates the distance a falling object
% travels due to gravity
global G
result = 1/2*G*t.^2;
end
```

The `global` command alerts the function to look in the workspace for the value of `G`. `G` must also have been defined in the command window (or program file) as a global variable:

```
global G
G = 9.8;
```

This approach allows you to change the value of `G` without needing to redefine the distance function or providing the value of `G` as an input argument to the distance function.

HINT

As a matter of style, always make the names of global variables uppercase. MATLAB doesn't care, but it is easier to identify global variables if you use a consistent naming convention.

HINT

It may seem like a good idea to use global variables because they can simplify your programs. However, consider this example of using global variables in your everyday life: It would be easier to order a book from an online bookseller if you had posted your credit card information on a site where any retailer could just look it up. Then the bookseller wouldn't have to ask you to type in the number. However, this might produce some unintended consequences (like other people using your credit card without your permission or knowledge!). When you create a global variable, it becomes available to other functions and can be changed by those functions, sometimes leading to unintended consequences.

6.1.8 Accessing Function Code

The functions provided with MATLAB are of two types. One type is built in, and the code is not accessible for us to review. The other type consists of files, stored in **toolboxes** provided with the program. We can see these function files (or the function files written by users) with the `type` command. For example, the `sphere` function creates a three-dimensional representation of a sphere; thus,

```
type sphere
```

or

```
type("sphere")
```

returns the contents of the **sphere.m** file:

```
function [xx,yy,zz] = sphere(varargin)
% SPHERE Generate sphere.
%   [X,Y,Z] = SPHERE(N) generates three (N+1)-by-(N+1)
%   matrices so that SURF(X,Y,Z) produces a unit sphere.
%   [X,Y,Z] = SPHERE uses N = 20.
%   SPHERE(N) and just SPHERE graph the sphere as a SURFACE
%   and do not return anything.
%   SPHERE(AX, . . .) plots into AX instead of GCA.
%   See also ELLIPSOID, CYLINDER.
%   Clay M. Thompson 4-24-91, CBM 8-21-92.
%   Copyright 1984-2002 The MathWorks, Inc.
%   $Revision: 5.8.4.1 $ $Date: 2002/09/26 01:55:25 $
%   Parse possible Axes input
error(nargchk(0,2,nargin));
[cax,args,nargs] = axescheck(varargin{:});
n = 20;
if nargs > 0, n = args{1}; end
% -pi <= theta <= pi is a row vector.
% -pi/2 <= phi <= pi/2 is a column vector.
theta = (-n:2:n)/n*pi;
phi = (-n:2:n)'/n*pi/2;
cosphi = cos(phi); cosphi(1) = 0; cosphi(n+1) = 0;
sintheta = sin(theta); sintheta(1) = 0; sintheta(n+1) = 0;
x = cosphi*cos(theta);
y = cosphi*sintheta;
z = sin(phi)*ones(1,n+1);
```

```
if nargout == 0
   cax = newplot(cax);
   surf(x,y,z,'parent',cax)
else
   xx = x; yy = y; zz = z;
end
```

HINT

Notice that the `sphere` function uses `varargin` to indicate that it will accept a variable number of input arguments. The function also makes use of the `nargin` and `nargout` functions. Studying this function may give you ideas on how to program your own functions. Notice that the `sphere` function uses an if/else structure, which is introduced in a subsequent chapter of this text.

6.2 SUBFUNCTIONS

Subfunctions can be created by grouping functions together in a single file. There are two approaches: You can add subfunctions to a MATLAB program, which we have done already. The other approach is to use multiple subfunctions in a primary function. Subfunctions can be used to modularize your code and to make the primary code easier to read. They have the disadvantage that they are only available from the primary file, and cannot be used by other programs, which is why they are also known as local functions.

HINT

You should not attempt to create code using subfunctions until you have first mastered simple functions.

KEY IDEA
Local function and subfunction are two different names for the same thing.

6.2.1 Using Subfunctions in Other Functions

Each MATLAB function has *one* primary function. The name of the function file must be the same as the *primary* function name. Thus, the primary function stored as *my_function.m* must be `my_function`. Subfunctions are added after the primary function and can have any legitimate MATLAB variable name. Figure 6.7 shows a very simple example of a function that both adds and subtracts two vectors. The primary function is named `subfunction_demo`. The file includes two subfunctions: `add` and `subtract`.

Notice in the editing window that the contents of each function are identified with a gray bracket. Each code section can be either collapsed or expanded, to make the contents easier to read, by clicking on the + or − sign included with the bracket. MATLAB uses the term **folding** for this functionality.

This function can be called from the command window or from any program in the current directory. For example, first define an `x` and a `y` array in the command window.

```
x=[10, 15, 17];
y=[ 1,  2,  3];
```

and then call the function

```
[add_result,subtract_result]=subfunction_demo(x,y)
```

which returns two arrays

```
add_result =
    11    17    20
subtract_result =
     9    13    14
```

Example 6.5 demonstrates the use of a more complicated example.

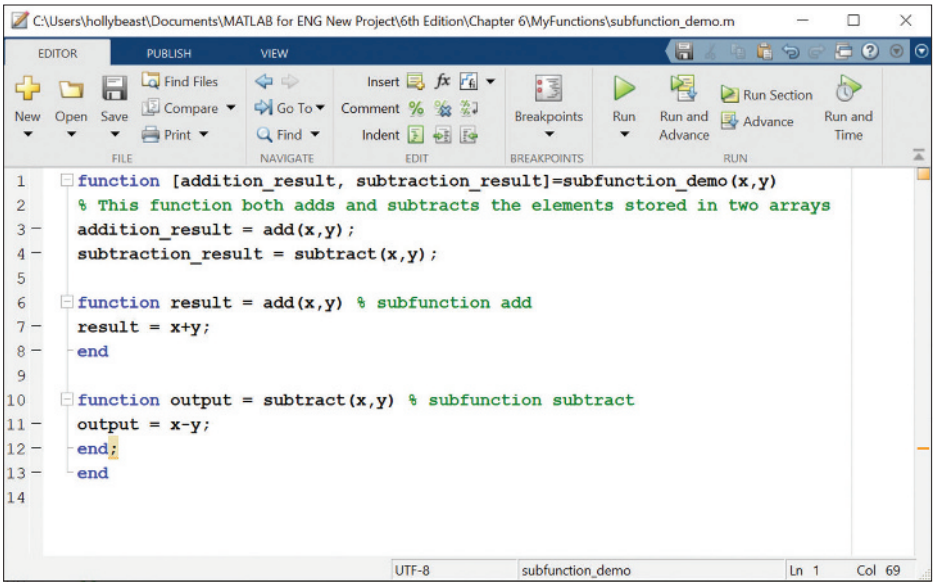


Figure 6.7
MATLAB allows the user to create subfunctions within a function. This file includes the primary function, `subfunction_demo`, and two subfunctions `add` and `subtract`.

EXAMPLE 6.5

INTEGRATION: USING SUBFUNCTIONS

In a course in numerical methods, you will be introduced to multiple techniques to perform integrations, which can be thought of as finding the area under a curve. The simplest is the trapezoid rule, followed by Simpson's 13 technique and Simpson's 38 technique. A common strategy is to use each in the following conditions.

If only two values of the independent variable are specified, defining a single interval, use:	Trapezoid rule
If an odd number of points are defined, which results in an even number of segments, use:	Simpson's 13 technique
If 4 points are defined, which results in 3 segments, use:	Simpson's 38 technique
Otherwise use:	A combination of Simpson's 13 technique and Simpson's 38 techniques

Create a function that combines these techniques into a single function called `integrate` and test it on the polynomial

$$f(x) = 10\sin(5x) + 5x^2$$

(*continued*)

with the limits of integration equal to

$$0 \leq x \leq 2.$$

This is equivalent to finding the area under the curve as shown in Figure 6.8a.

1. State the problem
Create a primary function called `integrate` to find the area under a curve, using subfunctions.

2. Describe the input and output

Input	The logic for each of the subfunctions is provided A sample function, $f(x) = 10*\sin(5*x) + 5*x^2$
Output	The value of the integral, for a variety of input parameters

3. Develop a hand example
Choose a simpler function, for example the equation of a straight line such as

$$f(x) = 5*x + 3$$

from $x = 0$ to $x = 5$. It is then easy to find the area under the curve, which for a straight line is the average value of *f(x)* times the difference in *x* values, as shown in Figure 6.8b.

$$area = \frac{(f(0) + f(5))}{2} * (5 - 0)$$

$$area = \frac{(3 + 28)}{2} * 5 = 77.5$$

All of the techniques specified in this example should give the same result.

4. Develop a MATLAB solution
This solution uses if/else structures which will not be introduced until a later chapter. However, the structure of the primary function and local functions is the most important part of this example.

```
function I = integrate(f,xL,xU,n) %Primary function
% f is a function handle
% xL is the lower limit of integration
```

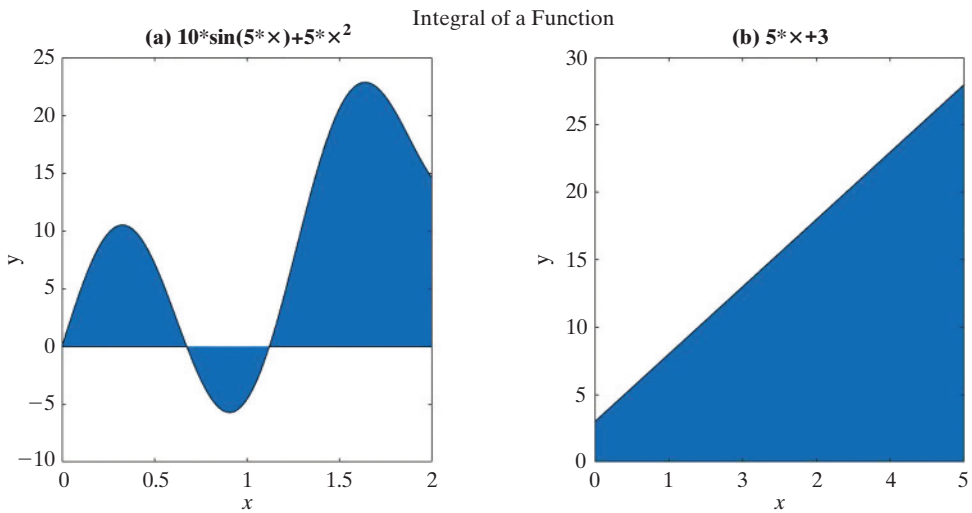


Figure 6.8a
(a) The area under the curve is equivalent to the integral of a function. (b) The area under a line is equal to the average value of *y* times the difference in *x* values.

```
% xU is the upper limit of integration
% n is the number of segments to be used in the integration
% n+1 is the number of data points
remainder = rem(n,2);
if  n==1
    I = trapf(f,xL,xU,n);
elseif remainder ==0
    I = simp13m(f,xL,xU,n);
elseif n == 3
    I = simp38(f,xL,xU);
else h=(xU-xL)/n;
    xM=xU-h*3;
    s13 = simp13m(f,xL,xM,n-3);
    s38 = simp38(f,xM,xU);
    I = s13 + s38;
end
end

function output = trapf(f,xL,xU,n) % The first subfunction
x=linspace(xL,xU,n+1);
y=f(x);
h = (xU-xL)/n;
total=2*sum(y)-y(1)-y(end);
output = h*total/2;
end

function output = simp38(f,xL,xU) % The second subfunction
h = (xU-xL)/3;
x = linspace(xL,xU,4);
y = f(x);
output = 3*h*(y(1) + 3*(y(2)+y(3)) + y(4))/8;
end

function output = simp13m(f,xL,xU,n) % The third subfunction
h = (xU-xL)/n;
x=linspace(xL,xU,n+1);
y = f(x);
even = 2:2:n+1;
odd = 3:2:n;
output = h*(y(1) + 4*(sum(y(even)))+2*sum(y(odd))+y(end))/3;
end
```

5. Test the solution

Test the function with the simple equation of a straight line for values of n from 1 to 5

```
f = @(x) 5*x + 3;
integrate(f,0,5,1)
integrate(f,0,5,2)
integrate(f,0,5,3)
integrate(f,0,5,4)
integrate(f,0,5,5)
```

(*continued*)

As expected, every case returned the same value, 77.5. This result matches the hand example, suggesting that the function works correctly and can now be used with the more complicated function.

```
f = @(x) 10*sin(5*x)+ 5*x.^2
integrate(f,0,2,1)
ans =
      14.56
integrate(f,0,2,2)
ans =
      -1.2657
integrate(f,0,2,3)
ans =
      13.35
integrate(f,0,2,4)
ans =
      19.473
integrate(f,0,2,5)
ans =
      18.417
integrate(f,0,2,21)
ans =
      17.013
```

6.3.2 Subfunctions in MATLAB Programs

When could you use subfunctions effectively in a script? Imagine that your instructor has assigned three homework problems, each requiring you to create and test a function.

- **Problem 1** Create and test a function called `square` to square values of x. Assume x varies between -3 and $+3$.
- **Problem 2** Create and test a function called `cold_work` to find the percent cold work experienced by a metallic rod, as it is drawn into a wire. Cold work is described by the following equation

$$\%\ \text{Cold Work} = \frac{r_i^2 - r_f^2}{r_i^2} \times 100$$

  where r_i is the initial radius of the rod, and r_f is the final radius of the rod. To test your function, let $r_i = 0.5$ cm and let $r_f = 0.25$ cm.
- **Problem 3** Create and test a function called `potential_energy` to determine the potential energy change of a given mass. The change in potential energy is given by

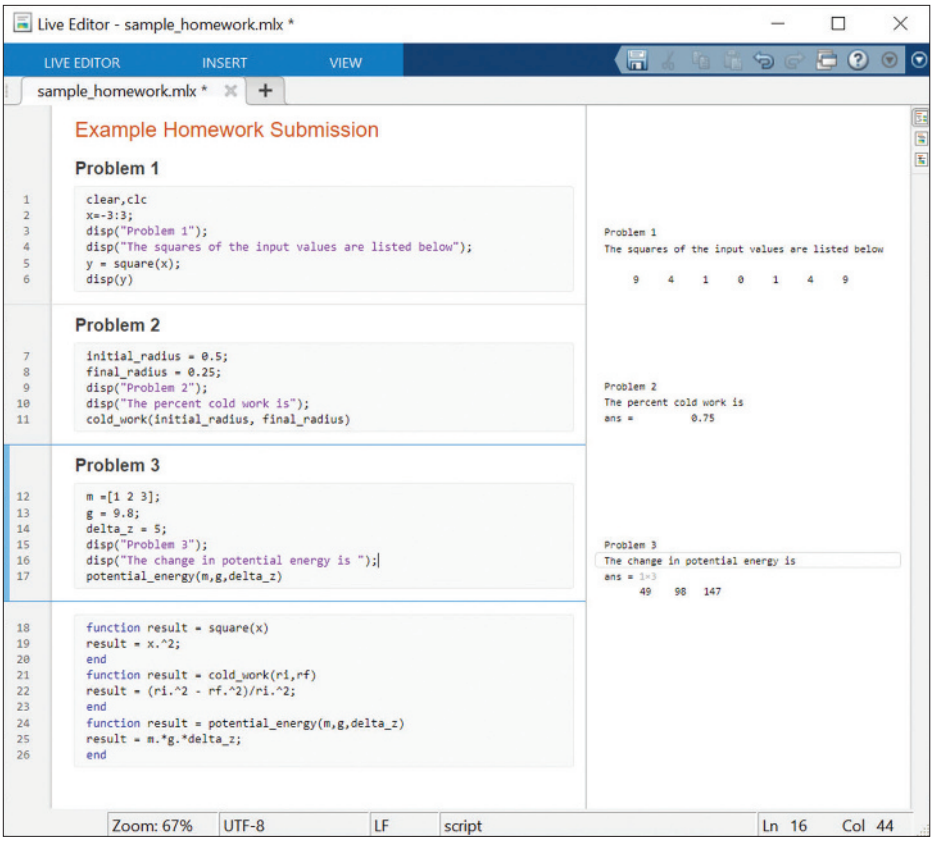
$$\Delta\text{PE} = m \times g \times \Delta z.$$

  Your function should have three inputs: m, g, and Δz. Use the following data to test your function:

$$m = [1\ 2\ 3]\text{kg}\ (\text{The array represents three different masses.})$$
$$g = 9.8\ \text{m/s}^2$$
$$\Delta z = 5\ \text{m}$$

To complete the assignment, you would need to create four files: one for each function, and one to call and test the functions. We can use subfunctions to reduce the number of files to one, as shown in Figure 6.9.

Figure 6.9
This Live Script is an example of a program with sequential subfunctions.



KEY IDEA
Group your functions together into toolboxes.

6.3 CREATING YOUR OWN TOOLBOX OF FUNCTIONS

When you call a function in MATLAB, the program first looks at the end of the current program file, then in the current folder to see if the function is defined. If it can't find the function listed there, it starts down a predefined search path, looking for a file with the function name. To view the path the program takes as it looks for files, select

```
Set Path
```

from the Home toolstrip or type

```
pathtool
```

in the command window (see Figure 6.10).

As you create more and more functions to use in your programming, you may wish to modify the path to look in a directory where you've stored your own personal tools. For example, suppose you have stored the `RD` and `DR` functions created in Example 6.1 in a directory called *MyFunctions*.

You can add this directory (folder) to the path by selecting *Add Folder* from the list of option buttons in the Set Path dialog window, as shown in Figure 6.10. You'll be prompted to either supply the folder location or browse to find it, as shown in Figure 6.11.

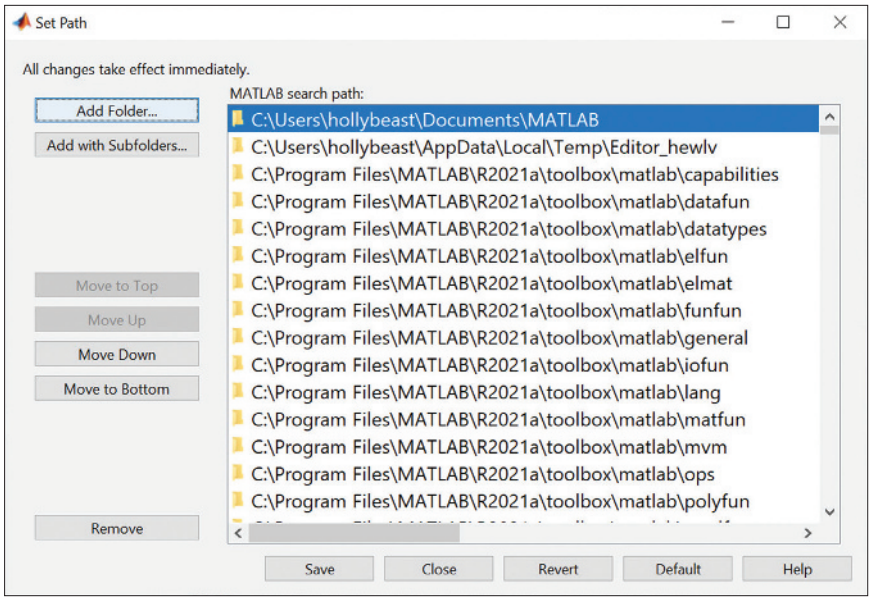


Figure 6.10
The path tool allows you to change where MATLAB looks for function definitions.

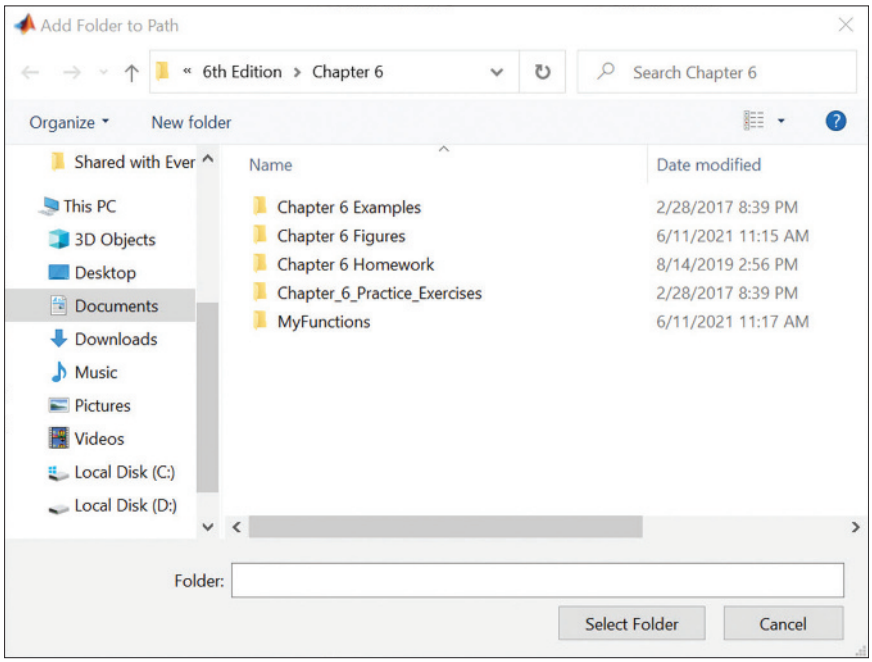


Figure 6.11
The Browse for Folder window.

MATLAB now first looks into the current folder for function definitions, then works down the modified search path, as shown in Figure 6.12.

Once you've added a folder to the path, the change applies only to the current MATLAB session, unless you save your changes permanently. Clearly, you should never make permanent changes to a public computer. However, if someone else has made changes you wish to reverse, you can select the default button as shown in Figure 6.12 to return the search path to its original settings.

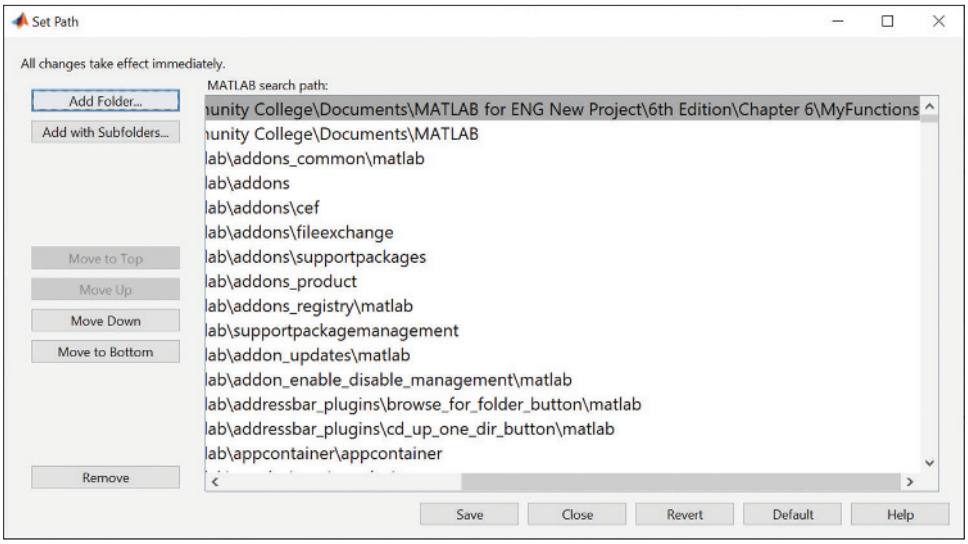


Figure 6.12
Modified MATLAB search path.

The path tool allows you to change the MATLAB search path interactively; however, the `addpath` function allows you to insert the logic to add a search path to any MATLAB program. Consult

```
help addpath or doc addpath
```

if you wish to modify the path in this way.

MATLAB provides access to numerous toolboxes developed at The MathWorks or by the user community. For more information, see the developer's website, www.mathworks.com.

KEY IDEA
Anonymous functions may be included in MATLAB programs with other commands or may be defined from the command window.

6.4 ANONYMOUS FUNCTIONS AND FUNCTION HANDLES

Normally, if you go to the trouble of creating a function, you'll want to store it for use in other programming projects. However, MATLAB includes a simpler kind of function, called an **anonymous function**. Anonymous functions are defined in the command window or in a script, and are available—much as are variable names—only until the workspace is cleared. To create an anonymous function, consider the following example:

```
ln = @(x)  log(x)
```

- The `@` symbol alerts MATLAB that `ln` is a function.
- Immediately following the `@` symbol, the input to the function is listed in parentheses.
- Finally, the function is defined.

The function name appears in the variable window, listed as a function_handle:

Name	Value	Size	Bytes	Class
ln	@(x) log(x)	1 × 1	16	function_handle

HINT

Think of a **function handle** as a nickname for the function.

Anonymous functions can be used like any other function—for example,

```
ln(10)
ans =
       2.3026
```

Once the workspace is cleared, the anonymous function no longer exists. Anonymous functions can be saved as *.mat* files, just like any variable, and can be restored with the `load` command. For example, to save the anonymous function `ln`, type:

```
save my_ln_function ln
```

A file named *my_ln_function.mat* is created, which contains the anonymous `ln` function. Once the workspace is cleared, the `ln` function no longer exists, but it can be reloaded from the *.mat* file

```
load my_ln_function
```

It is possible to assign a function handle to any function. Earlier in this chapter, we created a function file called *distance.m.*

```
function result = distance(t)
result = 1/2*9.8*t.^2;
end
```

The command

```
distance_handle = @(t) distance(t)
```

assigns the handle `distance_handle` to the distance function.

Anonymous functions and the related function handles are useful in functions that require other functions as input (function functions). For example, earlier in this chapter we introduced the `nargin` function, which accepts either a function name represented as a string, or a function handle. If we define `ln` as an anonymous function

```
ln = @(x) log(x)
```

then we can use the function handle, `ln`, as input to `nargin`

```
nargin(ln)
ans =
    1
```

KEY IDEA

Function functions require functions or function handles as input.

6.5 FUNCTION FUNCTIONS

MATLAB's **function functions** have an odd, but descriptive name. They are functions that require other functions as input. One example of a MATLAB built-in function function is the function plot, `fplot`. This function requires two inputs: a function or a function handle, and a range over which to plot. We can demonstrate the use of `fplot` with the function handle `ln`, defined as

```
ln = @(x) log(x)
```

The function handle can now be used as input to the `fplot` function:

```
fplot(ln,[0.1, 10])
```

The result is shown in Figure 6.12.

A wide variety of MATLAB functions accept function handles as input. For example, the `fzero` function finds the value of x where $f(x)$ is equal to 0. It accepts a function handle and a rough guess for x. From Figure 6.13, we see that the natural log has a zero between 0.5 and 1.5, so a rough guess for the zero point might be $x = 0.8$.

```
fzero(ln,0.8)
ans =
    1
```

We can check this result by substituting back into the `ln` function.

```
ln(ans)
ans =
    0
```

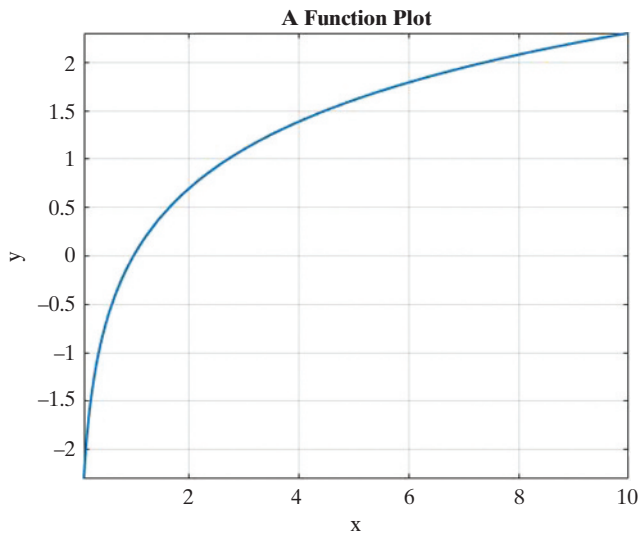


Figure 6.13
Function handles can be used as input to a function function, such as `fplot`.

SUMMARY

MATLAB contains a wide variety of built-in functions. However, you will often find it useful to create your own MATLAB functions. A common type of user-defined MATLAB function is the stand-alone function file, which must start with a function-definition line that contains the following:

- the word `function`,
- a variable that defines the function output,
- a function name, and
- a variable used for the input argument.

For example,

```
function output=my_function(x)
```

The function name must also be the name of the file in which the function is stored. Function names follow the standard MATLAB naming rules.

Like the built-in functions, user-defined functions can accept multiple inputs and can return multiple results.

Comments immediately following the function-definition line can be accessed from the command window with the `help` command.

Variables defined within a function are local to that function. They are not stored in the workspace and cannot be accessed from the command window. Global variables can be defined with the `global` command used in both the command window (or script file) and a MATLAB function. Good programming style suggests that you define global variables with capital letters. In general, however, it is not wise to use global variables.

Groups of user-defined functions, called "toolboxes," may be stored in a common directory and accessed by modifying the MATLAB search path. This is accomplished interactively with the path tool, either from the menu bar, as in

File → Set Path

or from the command line, with

```
pathtool
```

MATLAB provides access to numerous toolboxes developed at The MathWorks or by the user community.

Another type of function is the anonymous function, which is defined in a MATLAB session or in a script and exists only during that session. Anonymous functions are especially useful for very simple mathematical expressions, or as input to the more complicated function functions.

Multiple functions can be used either in a single script or in a single function, when they are listed as subfunctions. Subfunctions are also called local functions and are only available to code inside the primary file, unlike stand alone function files, which are available to any code that can access the directory where they are stored.

MATLAB SUMMARY

The following MATLAB summary lists and briefly describes the special characters, commands, and functions that were defined in this chapter:

Special Characters	
`@`	identifies a function handle, such as that used with anonymous functions
`%`	comment

Commands and Functions	
`addpath`	adds a directory to the MATLAB search path
`fminbnd`	a function function that accepts a function handle or function definition as input and finds the function minimum between two bounds

Commands and Functions	
`fplot`	a function function that accepts a function handle or function definition as input and creates the corresponding plot between two bounds
`fzero`	a function function that accepts a function handle or function definition as input and finds the function zero point nearest a specified value
`function`	identifies an M-file as a function
`global`	defines a variable that can be used in multiple sections of code
`local function`	function that is only visible to other functions or code in the same file
`meshgrid`	maps two input vectors onto two two-dimensional arrays
`nargin`	determines the number of input arguments in a function
`nargout`	determines the number of output arguments from a function
`pathtool`	opens the interactive path tool
`subfunction`	another name for a local function
`varargin`	indicates that a variable number of arguments may be input to a function

KEY TERMS

anonymous
argument
comments
directory
file name
folder
folding
function
function function
function handle
function name
global variable
local variable
nest
path
toolbox

PROBLEMS

Function Files

As you create functions in this section, be sure to comment them appropriately. Remember that, although many of these problems could be solved without a function, the objective of this chapter is to learn to write and use functions. Your instructor may ask you to submit your assignment using one of two approaches. In the first approach, each function should be created in its own file. You would then call the function from a main program file to complete the tasks required in each problem. In this approach you will be submitting multiple files to your instructor.

In the second approach, you can solve the homework problems in the same program file where the functions are stored. The functions must be stored at the end of the file, and must terminate with an end statement. In this case, you need only submit one file.

No matter which approach you use, you can still choose between scripts (M-files) and live scripts (MLX-files).

6.1 As described in Example 6.2, metals are actually crystalline materials. Metal crystals are called grains. When the average grain size is small, the metal is strong; when it is large, the metal is weaker. Since every crystal in a particular sample of metal is a different size, it isn't obvious how we should describe the average crystal size. The American Society for Testing and Materials (ASTM) has developed the following correlation for standardizing grain-size measurements:

$$N = 2^{n-1}$$

The ASTM grain size (n) is determined by looking at a sample of a metal under a microscope at a magnification of 100 × (100 power). The number of grains in a 1-square-inch area (actual dimensions of 0.01 in × 0.01 in) is estimated (N) and used in the preceding equation to find the ASTM grain size.

(a) Write a MATLAB function called `num_grains` to find the number of grains in a 1-square-inch area (N) at 100 × magnification when the ASTM grain size is known.
(b) Use your function to find the number of grains for ASTM grain sizes $n = 10$ to 100.
(c) Create a plot of your results.

6.2 Perhaps the most famous equation in physics is

$$E = mc^2$$

which relates energy E to mass m. The speed of light in a vacuum, c, is the property that links the two together. The speed of light in a vacuum is $2.9979 \times 10^8\ \mathrm{m/s}$.

(a) Create a function called `energy` to find the energy corresponding to a given mass in kilograms. Your result will be in joules, since $1\ \mathrm{kg\ m^2/s^2} = 1\ \mathrm{J}$.
(b) Use your function to find the energy corresponding to masses from 1 kg to 10^6 kg. Use the `logspace` function (consult `help logspace`) to create an appropriate mass vector.
(c) Create a plot of your results. Try using different logarithmic plotting approaches (e.g., `semilogy`, `semilogx`, and `loglog`) to determine the best way to graph your results.

6.3 The future-value-of-money formula relates how much a current investment will be worth in the future, assuming a constant interest rate:

$$\mathrm{FV} = \mathrm{PV} \times (1 + I)^n$$

where

FV is the future value,
PV is the present value or investment,
I is the interest rate expressed as a fractional amount per compounding period—i.e., 5% is expressed as .05, and
N is the number of compounding periods.

(a) Create a MATLAB function called `future_value` with three inputs: the investment (present value), the interest rate expressed as a fraction, and the number of compounding periods.
(b) Use your function to determine the value of a $1000 investment in 10 years, assuming the interest rate is 0.5% per month, and the interest is compounded monthly.

6.4 In freshman chemistry, the relationship between moles and mass is introduced:

$$n = \frac{m}{\text{MW}}$$

where

n = number of moles of a substance,
m = mass of the substance, and
MW = molecular weight (molar mass) of the substance.

(a) Create a function called `nmoles` that requires two vector inputs—the mass and molecular weight—and returns the corresponding number of moles. Because you are providing vector input, it will be necessary to use the `meshgrid` function in your calculations.
(b) Test your function for the compounds shown in Table P6.4, for masses from 1 to 10 g:

Table P6.4 Molar Mass of Model Compounds

Compound	Molecular Weight (Molar Mass)
Benzene	78.115 g/mol
Ethyl alcohol	46.07 g/mol
Refrigerant R134a	102.3 g/mol

Your result should be a 10×3 array.

6.5 By rearranging the preceding relationship between moles and mass, you can find the mass if you know the number of moles of a compound:

$$m = n \times \text{MW}$$

(a) Create a function called `mass` that requires two vector inputs—the number of moles and the molecular weight—and returns the corresponding mass. Because you are providing vector input, it will be necessary to use the `meshgrid` function in your calculations.
(b) Test your function with the compounds listed in the previous problem, for values of n from 1 to 10.

6.6 The distance to the horizon increases as you climb a mountain (or a hill). The expression

$$b = \sqrt{2rh + h^2}$$

where

d = distance to the horizon
r = radius of the earth
h = height of the hill

can be used to calculate that distance. The distance depends on how high the hill is, and on the radius of the earth (or another planetary body).

(a) Create a function called `distance` to find the distance to the horizon. Your function should accept two vector inputs—radius and height—and should return the distance to the horizon. Don't forget that you'll need to use `meshgrid` because your inputs are vectors.

(b) Create a MATLAB program that uses your `distance` function to find the distance in miles to the horizon, both on the Earth and on Mars, for hills from 0 to 10,000 feet. Remember to use consistent units in your calculations. Note that
- Earth's diameter = 7926 miles
- Mars' diameter = 4217 miles

Report your results in a table. Each column should represent a different planet, and each row a different hill height.

6.7 A rocket is launched vertically. At time $t = 0$, the rocket's engine shuts down. At that time, the rocket has reached an altitude of 500 m and is rising at a velocity of 125 m/s. Gravity then takes over. The height of the rocket as a function of time is

$$h(t) = -\frac{9.8}{2}t^2 + 125t + 500 \text{ for } t>0$$

(a) Create a function called `height` that accepts time as an input and returns the height of the rocket. Use your function in your solutions to parts b and c.

(b) Plot height vs. time for times from 0 to 30 seconds. Use an increment of 0.5 second in your time vector.

(c) Find the time when the rocket starts to fall back to the ground. (The `max` function will be helpful in this exercise.)

6.8 The distance a freely falling object travels is

$$x = \frac{1}{2}gt^2$$

where

g = acceleration due to gravity, 9.8 m/s^2,
t = time in seconds, and
x = distance traveled in meters.

If you have taken calculus, you know that we can find the velocity of the object by taking the derivative of the preceding equation. That is,

$$\frac{dx}{dt} = v = gt$$

We can find the acceleration by taking the derivative again:

$$\frac{\mathrm{d}v}{\mathrm{d}t} = a = g$$

(a) Create a function called `free_fall` with a single input vector t that returns values for distance `x`, velocity `v`, and acceleration `g`.
(b) Test your function with a time vector that ranges from 0 to 20 seconds.

6.9 Create a function called `polygon` that draws a polygon with any number of sides. Your function should require a single input: the number of sides desired. It should not return any value to the command window, but should draw the requested polygon in polar coordinates.

Creating Your Own Toolbox

6.10 This problem requires you to generate temperature-conversion tables. Use the following equations, which describe the relationships between temperatures in degrees Fahrenheit (T_F), degrees Celsius (T_C), kelvins (T_K), and degrees Rankine (T_R), respectively:

$$T_\mathrm{F} = T_\mathrm{R} - 459.67°\mathrm{R}$$
$$T_\mathrm{F} = \frac{9}{5}T_\mathrm{C} + 32°\mathrm{F}$$
$$T_\mathrm{R} = \frac{9}{5}T_\mathrm{K}$$

You will need to rearrange these expressions to solve some of the problems.

(a) Create a function called `F_to_K` that converts temperatures in Fahrenheit to Kelvin. Use your function to generate a conversion table for values from 0°F to 200°F.
(b) Create a function called `C_to_R` that converts temperatures in Celsius to Rankine. Use your function to generate a conversion table from 0°C to 100°C. Print 25 lines in the table. (Use the `linspace` function to create your input vector.)
(c) Create a function called `C_to_F` that converts temperatures in Celsius to Fahrenheit. Use your function to generate a conversion table from 0°C to 100°C. Choose an appropriate spacing.
(d) Group your functions into a folder (directory) called `my_temp_conversions`. Adjust the MATLAB search path so that it finds your folder. (Don't save any changes on a public computer!)

Anonymous Functions and Function Handles

6.11 Barometers have been used for almost 400 years to measure pressure changes in the atmosphere. The first known barometer was invented by Evangelista Torricelli (1608–1647), a student of Galileo during his final years in Florence, Italy. The height of a liquid in a barometer is directly proportional to the atmospheric pressure, or

$$P = \rho g h$$

where P is the pressure, ρ is the density of the barometer fluid, and h is the height of the liquid column. For mercury barometers, the density of the fluid is $13{,}560\ \text{kg/m}^3$. On the surface of the Earth, the acceleration due to gravity, g, is approximately $9.8\ \text{m/s}^2$. Thus, the only variable in the equation is the height of the fluid column, h, which should have the unit of meters.

(a) Create an anonymous function `P` that finds the pressure if the value of h is provided. The units of your answer will be

$$\frac{\text{kg}}{\text{m}^3}\frac{\text{m}}{\text{s}^2}\text{m} = \frac{\text{kg 1}}{\text{ms}^2} = \text{Pa}.$$

(b) Create another anonymous function to convert pressure in Pa (Pascals) to pressure in atmospheres (atm). Call the function `Pa_to_atm`. Note that

$$1\ \text{atm} = 101{,}325\ \text{Pa}.$$

(c) Use your anonymous functions to find the pressure for fluid heights from 0.5 m to 1.0 m of mercury.

(d) Save your anonymous functions as *.mat* files

6.12 The energy required to heat water at constant pressure is approximately equal to

$$E = mC_p\,\Delta T$$

where

m = mass of the water, in grams,
C_p = heat capacity of water, 1 cal/g K, and
ΔT = change in temperature, K.

(a) Create an anonymous function called `heat` to find the energy required to heat 1 gram of water if the change in temperature is provided as the input.

(b) Your result will be in calories:

$$g\ \frac{\text{cal}}{\text{g K}}\text{K} = \text{cal}$$

Joules are the unit of energy used most often in engineering. Create another anonymous function `cal_to_J` to convert your answer from part (a) into joules. (There are 4.2 J/cal.)

(c) Save your anonymous functions as *.mat* files.

6.13. **(a)** Create an anonymous function called `my_function`, equal to

$$-x^2 - 5x - 3 + e^x.$$

(b) Use the `fplot` function to create a plot from $x = -5$ to $x = +5$. Recall that the `fplot` function can accept a function handle as input.

(c) Use the `fminbnd` function to find the minimum function value in this range. The `fminbnd` function is an example of a function function, since it requires a function or function handle as input. The syntax is

```
fminbnd(function_handle, xmin, xmax)
```

Three inputs are required: the function handle, the minimum value of x, and the maximum value of x. The function searches between the minimum value of x and the maximum value of x for the point where the function value is a minimum.

6.14 In Problem 6.7, you created a function called `height` to evaluate the height of a rocket as a function of time. The relationship between time, t, and height, $h(t)$, is:

$$h(t) = -\frac{9.8}{2}t^2 + 125t + 500 \text{ for } t > 0$$

(a) Create a function handle to the `height` function called `height_handle`.
(b) Use `height_handle` as input to the `fplot` function, and create a graph from 0 to 60 seconds.
(c) Use the `fzero` function to find the time when the rocket hits the ground (i.e., when the function value is zero). The `fzero` function is an example of a function function, since it requires a function or function handle as input. The syntax is

```
fzero(function_handle, x_guess)
```

The `fzero` function requires two inputs—a function handle, and your guess as to the time value where the function is close to zero. You can select a reasonable `x_guess` value by inspecting the graph created in part (b).

Subfunctions

6.15 In Problem 6.10, you were asked to create and use three different temperature-conversion functions, based on the following conversion equations:

$$T_F = T_R - 459.67°\text{R}$$
$$T_F = \frac{9}{5}T_C + 32°\text{F}$$
$$T_R = \frac{9}{5}T_K$$

Recreate Problem 6.10 using nested subfunctions. The primary function should be called `temperature_conversions` and should include the following subfunctions

```
F_to_K
C_to_R
C_to_F
```

Within the primary function, use the subfunctions to:

(a) Generate a conversion chart in the form of an array for values from 0°F to 200°F. Include a column for temperature in Fahrenheit and Kelvin.

(b) Generate a conversion chart in the form of an array from 0°C to 100°C. Print 25 lines in the array. (Use the `linspace` function to create your input vector.) Your array should include a column for temperature in Celsius and Rankine.

(c) Generate a conversion chart in the form of an array from 0°C to 100°C. Choose an appropriate spacing. Include a column for temperature in Celsius and Fahrenheit.

CHAPTER

7 User-Controlled Input and Output

Objectives

After reading this chapter, you should be able to:

- Prompt the user for input to a program.
- Create output with the `disp` function.
- Create formatted output using `fprintf`.
- Create tables and use the `table` function to store information and display data.
- Create formatted output for use in other functions with the `sprintf` function.
- Use graphical techniques to provide program input.
- Use sections to modify and run MATLAB programs.

INTRODUCTION

So far, we have explored the use of MATLAB in two modes: in the command window as a scratch pad, and in the editing window to write simple programs. The programmer has been the user. Now we move on to more complicated programs, written in either scripts (M-files) or live scripts (MLX-files), where the programmer and the user may be different people. That will make it necessary to use input and output commands to communicate with the user, instead of rewriting the actual code to solve similar problems. MATLAB offers built-in functions to allow a user to communicate with a program as it executes. The `input` command pauses the program and prompts the user for input; the `disp`, `table` and `fprintf` commands provide output to the command window.

7.1 USER-DEFINED INPUT

Although we have written MATLAB programs, we have assumed that the programmer (you) and the user are the same person. To run the program with different input values, we actually changed some of the code. We can create more general programs by

allowing the user to input values of an array from the keyboard while the program is running, using the `input` function. The input function displays a text **string** in the command window, then waits for the user to provide the requested input. For example,

```
z=input("Enter a value")
```

displays

```
Enter a value
```

in the command window. If the user enters a value such as

```
5
```

the program assigns the value 5 to the variable `z`. If the `input` command does not end with a semicolon, the value entered is displayed on the screen:

```
z =
     5
```

The same approach can be used to enter a one- or two-dimensional array. The user must provide the appropriate brackets and delimiters (commas and semicolons). For example,

```
z = input("Enter values for an array in brackets")
```

KEY IDEA
The `input` function can be used to communicate with the program user.

requests the user to input an array such as

```
[1, 2, 3; 4, 5, 6]
```

and responds with

```
z =
       1 2 3
       4 5 6
```

This input value of `z` can then be used in subsequent calculations by the script.

Data entered with `input` does not need to be numeric information. Suppose we prompt the user with the command

```
x = input("Enter your name in double quotes")
```

and enter

```
"Holly"
```

when prompted. Because we haven't used a semicolon at the end of the `input` command, MATLAB will respond

```
x =
  "Holly"
```

Notice in the workspace window that `x` is listed as a 1×1 string.

Name	Value	Size	Bytes	Class
x	"Holly"	1×1	150	string

If you are entering a string you must enclose the characters in double quotes. An alternate form is to use a **character array** to represent text, which is created by

entering the characters in single quotes. This is the original strategy used by MATLAB and it still dominates most of the documentation. If we prompt the user with

```
x = input('Enter your name in single quotes')
```

and enter

```
'Holly'
```

when prompted MATLAB will respond

```
x =
  'Holly'
```

Notice that this time the workspace window lists x as a 1 × 5 character array:

Name	Value	Size	Bytes	Class
x	'Holly'	1 × 5	6	char

From a practical point of view, the decision on whether to use strings or character arrays for input statements is a matter of style. There are advantages to both decisions, which will become clear when data types are discussed in a later chapter. If you decide to use character arrays, you can take advantage of an alternate form of the input command, which alerts the function to expect character input without the single quotes by specifying character input in the second field:

```
x = input('Enter your name', 's')
```

Now you need only enter the characters, such as

```
Fernando
```

and the program responds with

```
x =
  'Fernando'
```

PRACTICE EXERCISES 7.1

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Calculate the area A of a triangle:

$$A = \frac{1}{2}\text{base} * \text{height}$$

   Prompt the user to enter the values for the base and for the height.
2. Find the volume V of a right circular cylinder:

$$V = \pi r^2 h$$

   Prompt the user to enter the values of r and h.
3. Create a vector from 0 to n, allowing the user to enter the value of n.
4. Create a vector that starts at a, ends at b, and has a spacing of c. Allow the user to input all of these parameters.

EXAMPLE 7.1

FREELY FALLING OBJECTS

In 1971, during the last Apollo mission to the moon, astronaut Dave Scott performed an experiment first proposed by Galileo in 1634. He dropped both a hammer and a feather and observed that they both hit the ground at the same time (Figure 7.1). You can see the video of the experiment by searching the Internet for hammer and feather drop on moon.

Mission Controller Joe Allen wrote in the Apollo 15 Preliminary Science Report: During the final minutes of the third extravehicular activity, a short demonstration experiment was conducted. A heavy object (a 1.32-kg aluminum geological hammer) and a light object (a 0.03-kg falcon feather) were released simultaneously from approximately the same height (approximately 1.6 m) and were allowed to fall to the surface. Within the accuracy of the simultaneous release, the objects were observed to undergo the same acceleration and strike the lunar surface simultaneously, which was a result predicted by well-established theory, but a result nonetheless reassuring considering both the number of viewers that witnessed the experiment and the fact that the homeward journey was based critically on the validity of the particular theory being tested.

Whether you are on the moon, the earth, or in interstellar space, the position of a freely falling object can be predicted by the same equation:

$$d = \frac{1}{2}gt^2$$

where

d = distance the object travels,
g = acceleration due to gravity, and
t = elapsed time.

Notice that mass is not part of this equation

In this example we shall allow the user to specify the value of g—the acceleration due to gravity—and a vector of time values.



Figure 7.1 During the Apollo 15 mission to the moon in 1971, Astronaut Dave Scott performed the "Hammer and Feather" experiment first proposed by Galileo in 1634. He successfully demonstrated that both objects fall at the same rate, independent of their mass. (Paul Fjeld/ NASA)

1. State the problem
Find the distance traveled by a freely falling object and plot the results.

2. Describe the input and output

Input	Value of g, the acceleration due to gravity, provided by the user
	Time, provided by the user
Output	Plot of distance versus time

3. Develop a hand example

$$d = \frac{1}{2} gt^2, \text{ so, on the moon at 100 seconds}$$

$$d = \frac{1}{2} \times 1.6 \text{ m/s}^2 \times 100^2 \text{s}^2$$

$$d = 8000 \text{ m}$$

4. Develop a MATLAB solution

```
%% Example 7.1
% Freefall
clear, clc
%% Request input from the user
g = input("What is the value of acceleration due to gravity?")
start = input("What starting time would you like?")
finish = input("What ending time would you like?")
incr = input("What time increments would you like calculated?")
time=start:incr:finish;
%% Calculate the distance
distance=1/2*g*time.^2;
% plot the results
loglog(time,distance)
title("Distance Traveled in Freefall")
xlabel("time,s"),ylabel("distance, m")
%% Find the maximum distance traveled
final_distance = max(distance)
```

The interaction in the command window is:

```
What is the value of acceleration due to gravity? 1.6
g =
  1.6000
What starting time would you like? 0
start =
  0
What ending time would you like? 100
finish =
  100
What time increments would you like calculated? 10
incr =
  10
final_distance =
  8000
```

The results are plotted in Figure 7.2.

(*continued*)

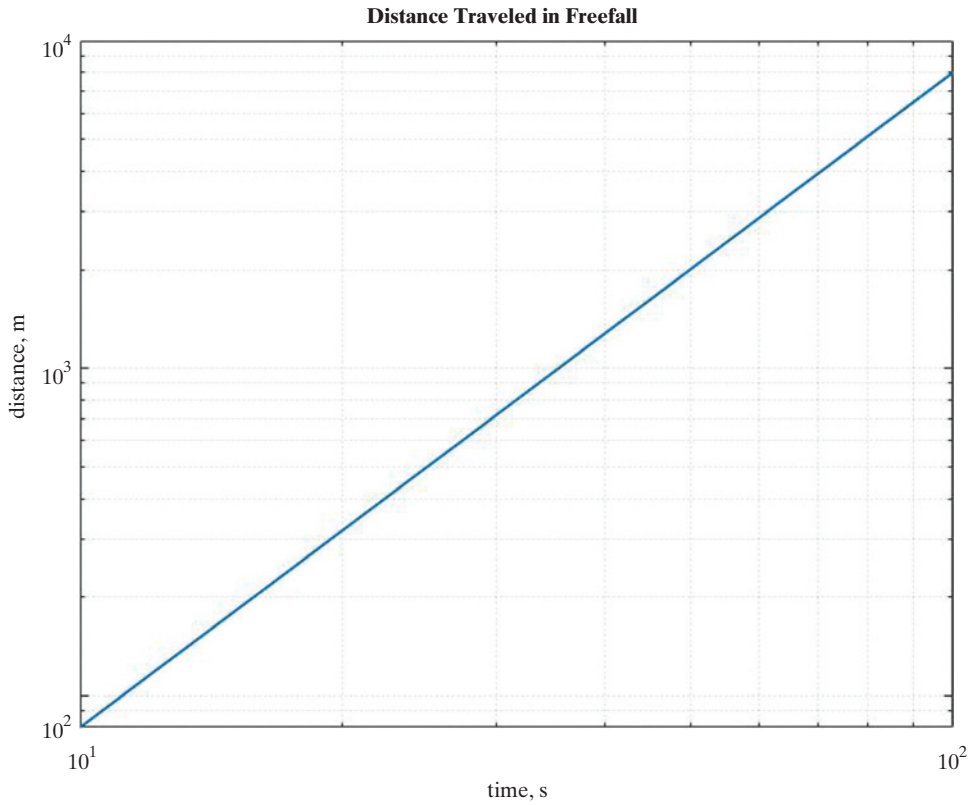


Figure 7.2
Distance traveled when the acceleration is $1.6 m/s^2$. Notice that the figure is a loglog plot.

5. Test the solution
   Compare the MATLAB solution with the hand solution. Since the user can control the input, the data used in the hand solution was entered. MATLAB tells us that the final distance traveled is 8000 m, which, since we entered 100 seconds as the final time, corresponds to the distance traveled after 100 seconds.

7.2 OUTPUT OPTIONS

There are several ways to display the contents of an array. The simplest is to enter the name of the array, without a semicolon. The name will be repeated, and the values in the array will be displayed, starting on the next line. For example, first define an array `x`:

```
x = 1:5;
```

Because there is a semicolon at the end of the assignment statement, the values in `x` are not repeated in the command window. However, if you want to display `x` later in your program, simply type in the variable name

```
x
```

which returns

```
x =
     1     2     3     4     5
```

KEY IDEA
The `disp` function can display either character arrays or numeric arrays.

MATLAB offers three other approaches to displaying results: the `disp` function, the `fprintf` function, and the table function.

7.2.1 Display Function

The display `(disp)` function can be used to display the contents of an array without printing the array name. It accepts a single array as the input argument. Thus,

```
disp(x)
```

returns

```
1  2  3  4  5
```

The display function can also be used to display a string (text enclosed in double quotation marks). For example,

```
disp("The values in the x array are:")
```

returns

```
The values in the x array are:
```

When you enter a string as input into the `disp` function, you are really entering an array. Try entering the following on the command line:

```
"The values in the x array are:"
```

MATLAB responds

```
ans =
 "The values in the x array are:"
```

The workspace window lists `ans` as a 1×1 string.

Name	Size	Bytes	Class
ans	1×1	198	string

KEY IDEA
Characters can be letters, numbers, or symbols.

When we execute the two display functions

```
disp("The values in the x array are:")
disp(x)
```

MATLAB responds

```
The values in the x array are:
1 2 3 4 5
```

Notice that the two `disp` functions are displayed on separate lines. You can display all of the information on the same line by creating a combined array of your two outputs. However, the first output was a string, and the second was numeric information, and you can't normally combine data of different types into a single array. Fortunately, MATLAB's built-in rules automatically convert numeric information to string information when they are added together. For a single numeric value, as in

```
y=5
disp("The value in the y array is : " + y)
```

MATLAB returns

```
The value in the y array is : 5
```

However, if you try this with a variable that stores an array of values such as the x value previously defined, the display function repeats once for each value of x,

```
x=1:5;
disp("The values in the x array are: " + x')
```

giving the result

```
"The values in the x array are: 1"
"The values in the x array are: 2"
"The values in the x array are: 3"
"The values in the x array are: 4"
"The values in the x array are: 5"
```

If you are looking for a result where all the x values print on the same line the x array must be converted to a text array using the `num2str` function. Despite its name, this function creates a character array, but the result is the same.

```
disp("The values in the x array are: " + num2str(x))
```

returns

```
The values in the x array are: 1  2  3  4  5
```

If you choose to use character arrays (single quotes) for input to the display function, the rules for **concatenation** (combining multiple components into a single array) are slightly different. You need to use square brackets to identify the combined character array in concert with the `num2str` function.

```
disp(['The values in the x array are: ' num2str(x)])
```

This gives the same result as the previous code.

HINT

If you want to include an apostrophe in a character array, you need to enter the apostrophe twice. If you don't do this, MATLAB will interpret the apostrophe as terminating the array. An example of the use of two apostrophes is

```
disp('The moon"s gravity is 1/6th that of the earth')
```

Similarly, if you want to include a quotation in a string, you need to enter the quote mark twice. For example

```
disp("Mark Twain once said ""Age is an issue of mind over
matter.""")
disp("""If you don't mind, it doesn't matter.""")
```

returns

```
Mark Twain once said "Age is an issue of mind over matter."
"If you don't mind, it doesn't matter."
```

You can use a combination of the `input` and `disp` functions to mimic a conversation. Try creating and running the following script:

```
clear, clc
disp("Hi There");
disp("I'm your MATLAB program");
name = input("Who are you? ",'s');
disp("Hi"+ name);
age = input("How old are you?");
disp(age + "that's not very old")
disp("The first computer was invented in 1936")
today=clock;
computer_age =today(1)-1936;
disp("I'm " + computer_age +" years old")
answer = input("Don't you just love computers? ",'s');
disp(answer + "?");
disp("Computers are very useful");
disp("You'll use them a lot in college!!");
disp("Good luck with your studies")
pause(2);
disp("Goodbye")
```

This interaction made use of the `pause` function. If you execute `pause` without any input, the program waits until the user hits the Enter key. If a value is used as input to the `pause` function, the program waits for the specified number of seconds, then continues.

7.2.2 Formatted Output—The `fprintf` Function

The `fprintf` function (formatted print function) gives you even more control over the output than you have with the `disp` function. In addition to displaying both text and array values, you can specify the format to be used in displaying the values, and you can specify when to skip to a new line. If you are a C programmer, you will be familiar with the syntax of this function. With few exceptions, the MATLAB `fprintf` function uses the same formatting specifications as the C `fprintf` function. This is hardly surprising, since MATLAB was written in C. (It was originally written in Fortran and then later rewritten in C.)

The general form of the `fprintf` command contains two arguments, one a string, and the other a list of arrays

```
fprintf(format-string, var,. . .)
```

Consider the following example:

```
cows = 5;
fprintf("There are %f cows in the pasture", cows)
```

The string, which is the first argument inside the `fprintf` function, contains a placeholder (`%`) where the value of the variable (in this case, `cows`) will be inserted. The placeholder also contains formatting information. In this example, the `%f` tells MATLAB to display the value of `cows` in a default fixed-point format. The default format displays six places after the decimal point:

```
There are 5.000000 cows in the pasture
```

Table 7.1 Type Field Format

Type Field	Result
`%f`	fixed-point notation
`%e`	exponential notation
`%d`	decimal notation—does not include trailing zeros if the value displayed is an integer. If the number includes a fractional component, it is displayed using exponential notation.
`%g`	whichever is shorter, `%f` or `%e`
`%c`	character information (displays one character at a time)
`%s`	string of characters (displays the entire string)

Besides defaulting to a fixed-point format, MATLAB allows you to specify an exponential format, `%e`, or lets you allow MATLAB to choose whichever is shorter, fixed point or exponential `(%g)`. It also lets you display character information `(%c)` or a string of characters `(%s)`. The decimal format `(%d)` is especially useful if the number you wish to display is an integer.

```
fprintf("There are %d cows in the pasture", cows)
There are 5 cows in the pasture
```

Table 7.1 illustrates the various formats supported by `fprintf`, and the related `sprintf` functions.

Additional type fields are described in the help feature.

KEY IDEA
The `fprintf` function allows you to control how numbers are displayed.

MATLAB does not automatically start a new line after an `fprintf` function is executed. If you tried out the preceding `fprintf` command example, you probably noticed that the command prompt is on the same line as the output:

```
There are 5.000000 cows in the pasture>>
```

If we execute another command, the results will appear on the same line instead of moving down. Thus, if we issue the new commands

```
cows = 6;
fprintf("There are %f cows in the pasture", cows)
```

from a script, MATLAB continues the command window display on the same line:

```
There are 5.000000 cows in the pasture There are 6.000000 cows
in the pasture
```

KEY IDEA
The `fprintf` function allows you to display both character and numeric information with a single command.

To cause MATLAB to start a new line, use `\n`, called a linefeed, at the end of the string. For example, the code

```
cows = 5;
fprintf("There are %f cows in the pasture \n", cows)
cows = 6;
fprintf("There are %f cows in the pasture \n", cows)
```

returns the following output:

```
There are 5.000000 cows in the pasture
There are 6.000000 cows in the pasture
```

HINT

The backslash (\) and forward slash (/) are different characters. It's a common mistake to confuse them—and then the linefeed command doesn't work! Instead, the output to the command window will be

```
There are 5.000000 cows in the pasture /n
```

Other special format commands are listed in Table 7.2. The tab (`\t`) is especially useful for creating output in which everything lines up neatly.

Table 7.2 Special Format Commands

Format Command	Resulting Action
`\n`	linefeed
`\r`	carriage return (similar to linefeed)
`\t`	tab
`\b`	backspace

You can further control how the variables are displayed using the optional **width field** and **precision field** with the format command. The `width field` controls the minimum total number of characters to be printed. It must be a positive decimal integer. The `precision field` is preceded by a period (.) and specifies the number of decimal places after the decimal point for exponential and fixed-point types. For example, `%8.2f` specifies that the minimum total width available to display your result is eight digits, two of which are after the decimal point. Thus, the code

```
voltage = 3.5;
fprintf("The voltage is %8.2f millivolts \n",voltage);
```

returns

```
The voltage is     3.50 millivolts
```

Notice the empty space before the number 3.50. This occurs because we reserved six spaces (eight total, two after the decimal) for the portion of the number to the left of the decimal point.

HINT

A common misconception is that the field identifiers refer to the numbers before and after the decimal point; that is that %8.2f means 8 spaces before the decimal point and 2 after. However, the first value refers to the **total** field width allocated for the number. A field identifier of %2.3f makes no sense, since it allocates 2 spaces for the number, three of which are after the decimal point.

Often when you use the `fprintf` function, your variable will be an array—for example,

```
x = 1:5;
```

MATLAB will repeat the string in the `fprintf` command until it uses all the values in the array. Thus,

```
fprintf("%8.2f \n",x);
```

returns

```
1.00
2.00
3.00
4.00
5.00
```

If the variable is a two-dimensional array, MATLAB uses the values one *column* at a time, going down the first column, then the second, and so on. Here's a more complicated example:

```
feet = 1:3;
inches = feet.*12;
```

Combine these two arrays:

```
conversions = [feet;inches]
```

MATLAB then returns

```
conversions =
        1  2  3
       12 24 36
```

The `conversions` array consists of two rows. The first row corresponds to feet, and the second row corresponds to inches. Now we can use the `fprintf` function to create output that is easier to interpret. For instance,

```
fprintf("%4.0f %7.2f \n",conversions)
```

returns the following output:

```
1   12.00
2   24.00
3   36.00
```

Why don't the two outputs look the same? The `fprintf` statement we created uses two values at a time. It goes through the `conversions` array one *column* at a time to find the numbers it needs. Thus, the first two numbers used in the `fprintf` output are from the first column of the `conversions` array.

The `fprintf` function can accept a variable number of arrays after the string. It uses all of the values in each of these arrays, in order, before moving on to the next array. As an example, suppose we wanted to use the feet and inches arrays without combining them into a single array. Then we could type

```
fprintf("%4.0f %7.2f \n", feet, inches)
```

which returns

```
1     2.00
3    12.00
24   36.00
```

The function works through the values of `feet` first, then uses the values in `inches`. It is unlikely that this is what you really want the function to do (in this example it wasn't), so the output values are almost always grouped into a single array to use in `fprintf`.

The `fprintf` command gives you considerably more control over the form of your output than MATLAB's simple format commands. It does, however, require some care and forethought to use.

In addition to creating **formatted output** for display in the command window, the `fprintf` function can be used to send formatted output to a file. First, create and open an output file and assign it a file identifier (nickname). You do this with the `fopen` function

```
file_id = fopen("my_output_file.txt", "wt");
```

The first field is the name of the file, and the second field makes it possible to write data to the file (hence the string `"wt"`). Once the file has been identified and opened for writing, use the `fprintf` function, adding the file identifier as the first field in the function input.

KEY IDEA
The `fprintf` function allows you to send formatted output to a file.

```
fprintf(file_id, 'Some example output is %4.2f \n', pi*1000)
```

This form of the function sends the result of the formatted string

```
Some example output is 3141.59
```

to *my_output_file.txt*. To the command window, the function sends a count of the number of bytes transferred to the file.

```
ans =
  32
```

You can examine the contents of *my_output_file.txt* by double-clicking on the file name in the current folder window. The file opens in the editing window, as shown in Figure 7.3.

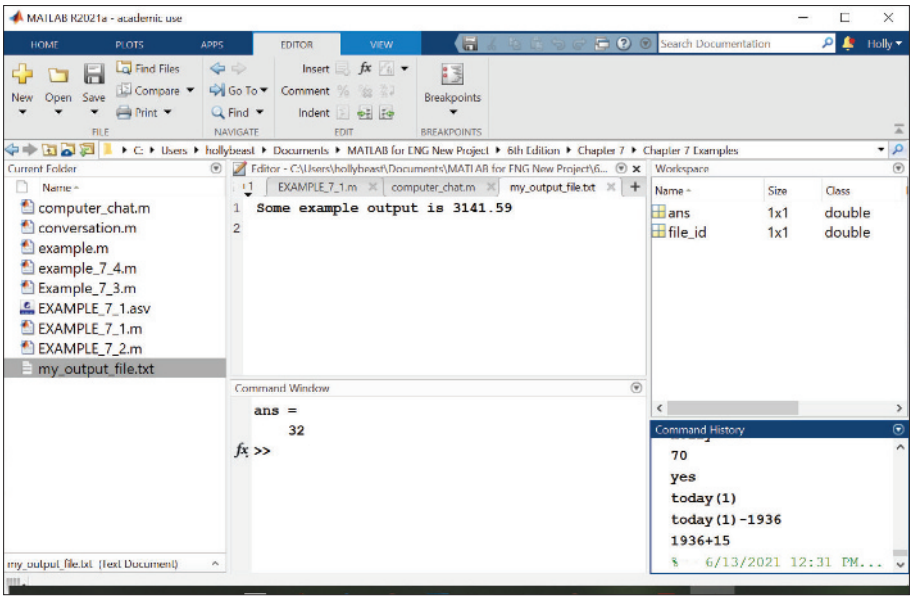


Figure 7.3
Data created using `fprintf` are often stored in text files. The contents of text files can be viewed in the editing window by double-clicking on the file name in the current directory.

HINT

A common mistake new programmers make when using `fprintf` is to forget to include the field type identifier, such as `f`, in the placeholder sequence. The `fprintf` function won't work, but no error message is returned either.

HINT

If you want to include a percentage sign in an `fprintf` statement, you need to enter the `%` twice. If you don't, MATLAB will interpret the `%` as a placeholder for data. For example,

```
fprintf('The interest rate is %5.2f %% \n', 5)
```

results in

```
The interest rate is 5.00 %.
```

EXAMPLE 7.2

FREE FALL: FORMATTED OUTPUT

Redo Example 7.1, but this time create an array of results instead of a plot. Use the `disp` and `fprintf` commands to control the appearance of the output.

1. State the problem
   Find the distance traveled by a freely falling object.
2. Describe the input and output

   Input Value of g, the acceleration due to gravity, provided by the user
   Time t, provided by the user

   Output Distances calculated for each planet and the moon
3. Develop a hand example

$$d = \frac{1}{2}gt^2, \text{ so on the moon at 100 seconds,}$$
$$d = \frac{1}{2} \times 1.6 \text{ m/s}^2 \times 100^2 \text{s}^2$$
$$d = 8000 \text{ m}$$

4. Develop a MATLAB solution
   This example was created in a live script (MLX-file). The program code is exactly the same in a script and in a live script.

```
%% Example 7.2
% Freefall
clear, clc
% Request input from the user
g = input("What is the value of acceleration due to gravity? ")
start=input("What starting time would you like? ")
finish = input("What ending time would you like? ")
incr = input("What time increments would you like calculated? ")
time=start:incr:finish;
% Calculate the distance
distance=1/2*g*time.^2;
% Create a matrix of the output data
results=[time;distance];
% Create the output
fprintf("For an acceleration due to gravity of   %5.1f "+...
    seconds \n the … following data were calculated \n", g)
disp("Distance Traveled in Freefall")
```

```
disp("time,sec      distance, m")
fprintf("%8.0f   %10.2f\n",results)
```

This program produces the following interaction in the command window:

```
What is the value of acceleration due to gravity? 1.6
g =
  1.6000
What starting time would you like? 0
start =
  0
What ending time would you like? 100
finish =
  100
What time increments would you like calculated? 10
incr =
  10
```

The input interactions are repeated in the live script. The results are sent to the live script instead of to the command window and are displayed either in the right-hand frame or inline depending on the user's selection, as shown in Figure 7.4. This makes documenting your work especially easy, since the file can be saved as a PDF directly from the save menu.

5. Test the solution
   Compare the MATLAB solution with the hand solution. Since the output is a table, it is easy to see that the distance traveled at 100 seconds is 8000 m. Try using other data as input, and compare your results with the graph produced in Example 7.1.

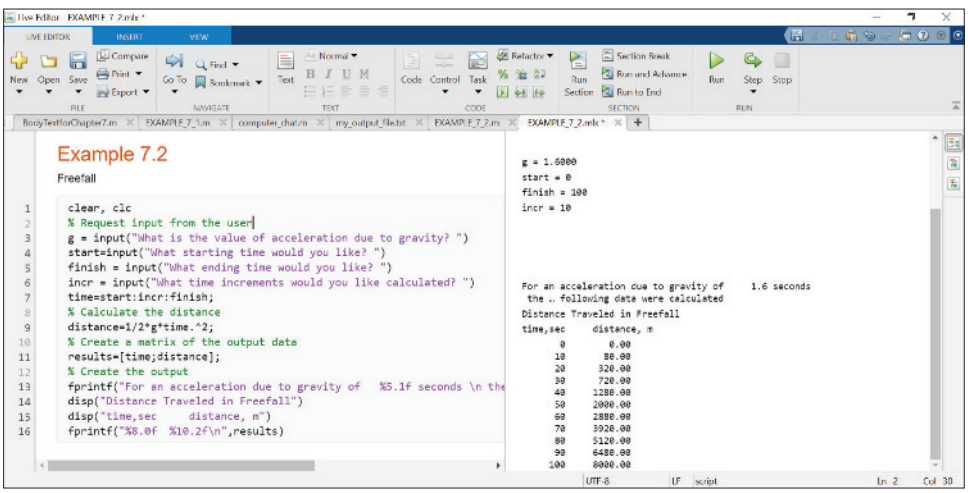


Figure 7.4
Live Scripts can store not only the computer code necessary to solve a problem, but the calculation results.

PRACTICE EXERCISES 7.2

These exercises should be completed in either a script (M-file) or a live script (MLX-file). The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Use the `disp` command to create a title for a chart that converts inches to feet.
2. Use the `disp` command to create column headings for your chart.

3. Create an `inches` vector from 0 to 120 with an increment of 10.
4. Calculate the corresponding values of `feet`.
5. Group the `inch` vector and the `feet` vector together into a `results` array.
6. Use the `fprintf` command to send your results to the command window.

KEY IDEA
The `sprintf` function is similar to `fprintf` and is useful for annotating plots.

7.2.3 Formatted Output—The `sprintf` Function

The `sprintf` function is similar to `fprintf`, but instead of just sending the result of the formatted string to the command window, `sprintf` assigns it a name and sends it to the command window.

```
a = sprintf("Some example output is %4.2f \n", pi*1000)
a =
     Some example output is 3141.59
```

When would this be useful? In Example 7.3, the `sprintf` function is used to specify the contents of a text box, which is then used as an annotation on a graph.

EXAMPLE 7.3

PROJECTILE MOTION: ANNOTATING A GRAPH

Recall from earlier examples that the equation describing the range of a projectile fired from a cannon is

$$R(\theta) = \frac{v^2}{g}\sin(2\theta)$$

where

$R(\theta)$ is the range in meters,

v is the initial projectile velocity in m/s,

θ is the launch angle, and

g is the acceleration due to gravity, 9.9 m/s^2.

Plot the angle on the x-axis versus the range on the y-axis, and add a text box indicating the value of the maximum range.

1. State the problem
   Find and plot the distance traveled by projectile, as a function of launch angle. Annotate a plot, indicating the maximum range.

2. Describe the input and output

Input	Acceleration due to gravity, $g = 9.9\ \text{m/s}^2$
	Launch angle
	Initial projectile velocity, 100 m/s
Output	An annotated graph indicating the maximum range.

3. Develop a hand example
We know from physics and from previous examples that the maximum range occurs at a launch angle of 45 degrees. Substituting into the provided equation,

$$R(45^\circ) = \frac{100^2\ \mathrm{m}^2/\mathrm{s}^2}{9.9\ \mathrm{m}/\mathrm{s}^2} \sin(2 * 45^\circ)$$

Since the angle is specified in degrees, you must either set your calculator to accept degrees into the sine function, or else convert 45 degrees to the corresponding number of radians $(\pi/4)$. After you have done so, the result is

$$R(45^\circ) = 1010\ \mathrm{m}.$$

4. Develop a MATLAB solution

```
%% Example 7.3
% Find the maximum projectile range
% for a given set of conditions
% Create an annotated graph of the results
clear, clc
% Define the input parameters
g = 9.9;   % Acceleration due to gravity
velocity = 110; % Initial velocity, m/s^2
theta = [0:5:90]  % Launch angle in degrees
%  Calculate the range
range = velocity^2/g*sind(2*theta);
%  Calculate the maximum range
maximum = max(range);
% Create the input for the textbox
t = "The maximum range was ";
text_input=sprintf("%s %4.0f meters \n", t, maximum);
% Plot the results
plot(theta,range), grid on
title("Range of a Projectile")
xlabel("Angle, degrees"), ylabel("Range, meters")
text(20,maximum,text_input)
```

There are several things to notice about this program. First, we took advantage of the `sind` function to calculate the value of sine, using degrees as input. Part of the input argument to `sprintf` was defined as a string, and was called using %s to keep the code more compact. Finally, the location of the text box will always start on the graph at 20 degrees (measured on the *x*-axis), but the *y* location depends on the maximum range. This program produces the graph shown in Figure 7.5a.

5. Test the solution
Compare the MATLAB solution with the hand solution. The text box used to annotate the graph lists the maximum range as 1010 m, the same value calculated by hand. We could also test the program with a different initial velocity, for example, 110 m/s. The result is shown in Figure 7.5b.

(*continued*)

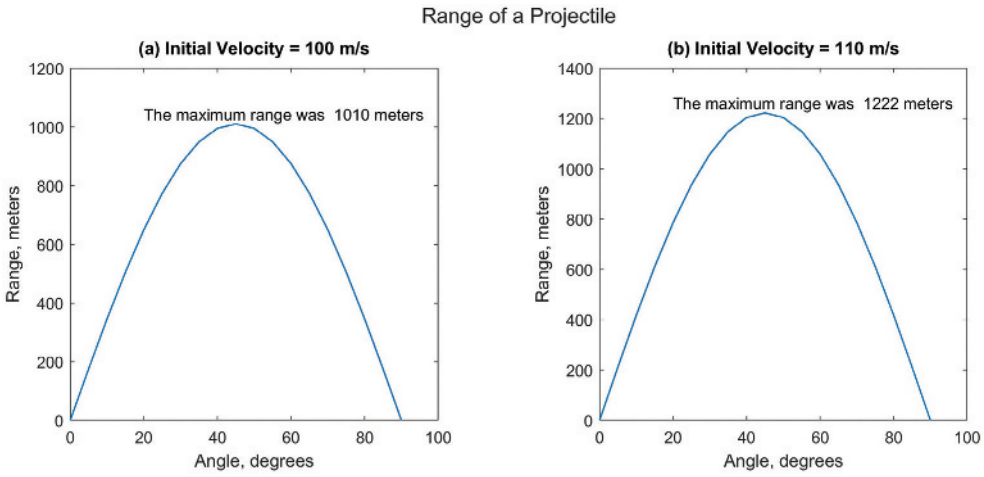


Figure 7.5
The contents of the text box change, depending on the input to the program, and are controlled by the `sprintf` function.

7.2.4 The `table` Function

Using `disp` and `fprintf` allows us to create formatted output in the command window, but it is awkward to create output where the columns are different data types; for example, characters or strings and numbers using these two functions. The `table` function uses a different data type called a *table* to accomplish this. We will focus on how to use the `table` function to create a table array and display it as output.

Consider a calculation where we want to know how far an object will fall, both on the earth and the moon, after 100 seconds. The acceleration due to gravity in each location can be represented by a column vector, `g`:

```
g = [9.8; 1.6]
```

The distance travelled is one half the acceleration due to gravity times the time squared, so

```
d = 0.5 * g * 100^2
```

which also returns a column vector.

We would like to have a table that lists the planet names in the first column, the value of `g` in the second column, and the distance travelled in the third. To do this, define a column vector of planet names using strings. Be sure to use strings, not character arrays which won't work in this case.

```
p=["Earth";"Moon"];
```

Once `g`, `d`, and `p` have been defined they can be combined to create the table using the `table` function (all of the input vectors need to be columns.)

```
table(p,g,d)
```

which returns

```
ans =

 2 × 3 table
       p         g        d
    ________    ____    _____
    "Earth"     9.8     49000
    "Moon"      1.6      8000
```

Notice that the column names default to the variable names. To specify more meaningful column names add two additional fields

```
table(p,g,d,'VariableNames',["Planet","g","Distance"])
ans =

 2 × 3 table
     Planet       g      Distance
    ________    ____    _________
    "Earth"     9.8       49000
    "Moon"      1.6        8000
```

Notice that 'VariableNames' is specified with single quotes. This is one of the few instances where you still need to stick with single quotes. If you forget, MATLAB will issue an error message with the appropriate reminder.

Finally, the output table would be cleaner without the `ans =` line. To accomplish this, nest the code inside the `disp` function

```
disp(table(p,g,d,'VariableNames',["Planet","g","Distance"])
```

which gives

```
     Planet       g       Distance
    ________    ____    _________
    "Earth"     9.8       49000
    "Moon"      1.6        8000
```

To save the table information, set it equal to a variable name.

```
my_earth_moon_data = table(p,g,d,'VariableNames',["Planet",...
"g","Distance"]);
```

You can see what is stored in the table variable by double-clicking on it in the Workspace window. The variable editor will open in the editing window, as shown in Figure 7.6. You can also interactively change the value of the variables stored in a table (or any other variable) in the variable editor, or even add new lines of information to the data set.

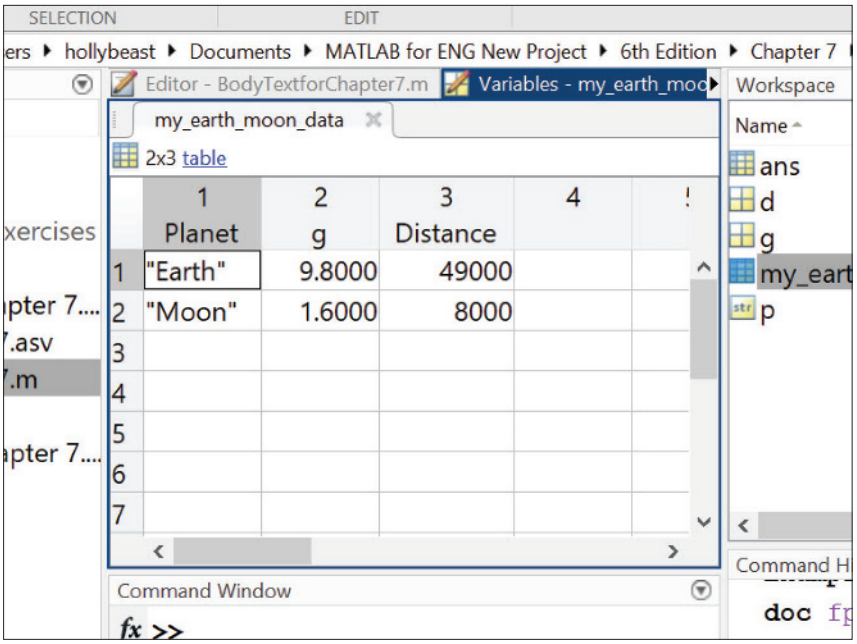


Figure 7.6
Table arrays can store different types of data, and can be inspected in the variable editor. Select the variable name in the Workspace window and double-click to open the variable editor window.

7.3 GRAPHICAL INPUT

MATLAB offers a technique for entering ordered pairs of *x*- and *y*-values graphically. The `ginput` command allows the user to select points from a figure window and converts the points into the appropriate *x*- and *y*-coordinates. In the statement

```
[x,y] = ginput(n)
```

MATLAB requests the user to select *n* points from the figure window. If the value of *n* is not included, as in

```
[x,y] = ginput
```

MATLAB accepts points until the return key is entered

This technique is useful for picking points off a graph. Consider the graph in Figure 7.7. The figure was created by defining *x* from 5 to 30 and calculating *y*:

```
x = 5:30;
y = x.^2 - 40.*x + 400;
plot(x, y)
axis([5,30, -50,250])
```

The axis values were defined so that the graph would be easier to trace.

Once the `ginput` function has been executed, as in

```
[a,b] = ginput
```

MATLAB adds a floating crosshair to the graph. After this crosshair is positioned to the user's satisfaction, right-clicking and selecting Return (Enter) sends the values of the *x*- and *y*-coordinates to the program:

```
a =
  24.4412
b =
  19.7368
```

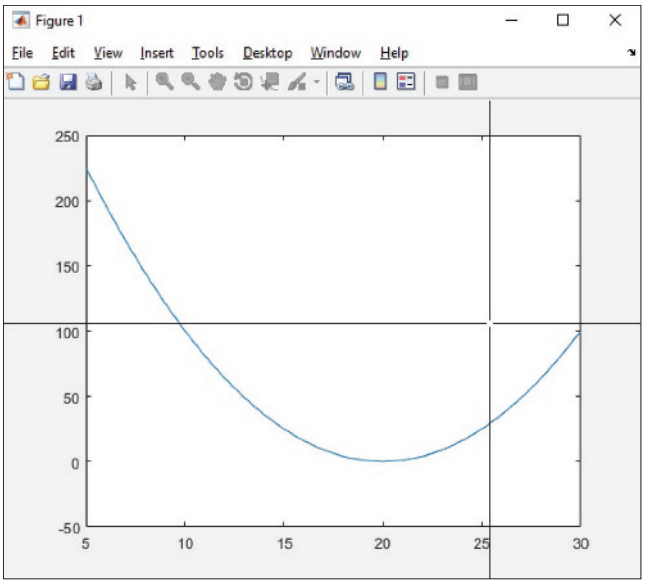


Figure 7.7
The `ginput` function allows the user to pick points off a graph.

7.4 READING AND WRITING DATA FROM FILES

Data are stored in many different formats, depending on the devices and programs that created the data and on the application. For example, sound might be stored in a *.wav* file, and an image might be stored in a *.jpg* file. Many applications store data in Excel spreadsheets (*.xls* or *xlsx* files). The most generic of these files is the ASCII file, usually stored as a *.dat* or a *.txt* file. You may want to import these data into MATLAB to analyze in a MATLAB program, or you might want to save your data in one of these formats to make the file easier to export to another application.

7.4.1 Importing Data

KEY IDEA
MATLAB can import data from files using a variety of formats.

Import Wizard

If you select a data file from the current folder and double-click on the file name, the Import Wizard launches. The Import Wizard determines what kind of data is in the file, and suggests ways to represent the data in MATLAB. Table 7.3 is a list of some of the data types MATLAB recognizes. Not every possible data format is supported by MATLAB. You can find a complete list by searching the help feature for supported file formats.

The Import Wizard can be used for simple ASCII files. Many of the other formats can also be imported with the Import Wizard, which can be launched programmatically or from the command line, using the `uiimport` function:

```
uiimport("filename.extension")
```

For example, it is easy to record sound files using a variety of software tools, or to find existing files on the Internet. To import a sound file, such as one called **dave.wav**, type

```
uiimport("dave.wav")
```

The Import Wizard then opens, as shown in Figure 7.8.

Either technique for launching the Import Wizard (double-clicking on the file name in the current folder window, or using the `uiimport` function in the

Table 7.3 Some of the Data File Types Supported by MATLAB

File type	Extension	Remark
Text	**.mat**	MATLAB workspace
	.dat	ASCII data
	.txt	ASCII data
	.csv	Comma-separated values
Other common scientific	**.cdf**	common data format
data formats	**.fits**	flexible image transport system data
	.hdf	hierarchical data format
Spreadsheet data	**.xls, xlsx**	Excel spreadsheet
Image data	**.tiff**	tagged image file format
	.bmp	bitmap
	.jpeg or jpg	joint photographics expert group
	.gif	graphics interchange format
Audio data	**.au, snd**	audio
	.wav	Microsoft wave file
	.mp4	MPEG-4
Movie	**.avi**	audio/video interleaved file
	.mpg	motion picture experts group

command window) requires an interaction with the user (through the Wizard). If you want to load a data file from a MATLAB program, you'll need a different approach.

Import Commands

You can bypass the Wizard interactions using one of the functions that are especially designed to read each of the supported file formats. For example, to read in a **.wav** file, use the `audioread` function:

```
[data,fs] = audioread("dave.wav")
```

Clearly, you need to understand what kind of data to expect, so that you can name the created variables appropriately. To use the files you have imported, you'll need to use a function appropriate to the data. In the case of a *.wav* file, the sound function is appropriate, so the code to play the data from the *dave.wav* file is

```
sound(data,fs)
```

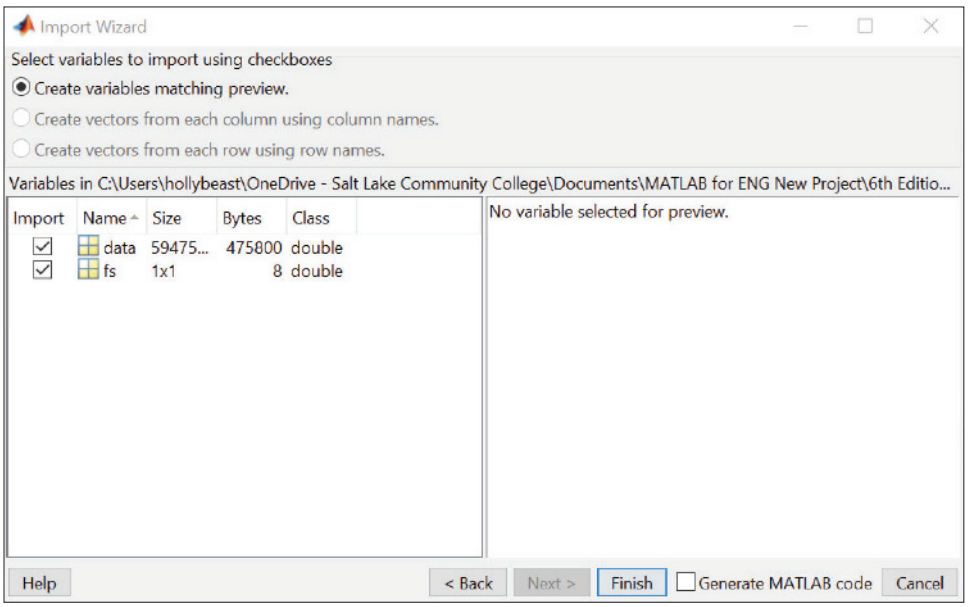


Figure 7.8
The Import Wizard launches when the `uiimport` command is executed.

Table 7.4 The readtable import function

Extensions	Data type
.txt, .dat, .csv	delimited text files
.xls, .xlsb, .xlsm, .xlsx, .xltm, .xltx, .ods	spreadsheet files
.xml	Extensible Markup Language file

One extremely useful import function is `readtable`, which imports data that is stored in tabular form and stores them in a table array. The data types are determined by the file extension. Supported data types are listed in Table 7.4.

The function automatically determines an appropriate data type for each column in the resulting table array and assigns variable names for the columns based on the data in the first row. Table arrays are discussed in more detail in a later chapter. To access data in a table you have imported using the `readtable` function, double-click on the variable in the workspace window and notice what names have been assigned to each column. MATLAB contains a built-in data file that allows for experimentation. To import the file type:

```
T = readtable("patients.dat");
```

The result is a table array, named T in the workspace. Double-clicking on the variable opens the variable editor, as shown in Figure 7.9. You can extract a single column by renaming it, using dot syntax as in:

```
Age = T.Age
```

If you don't know which function to use to automatically import a data file in a program, the Import Wizard includes the option to generate MATLAB code to

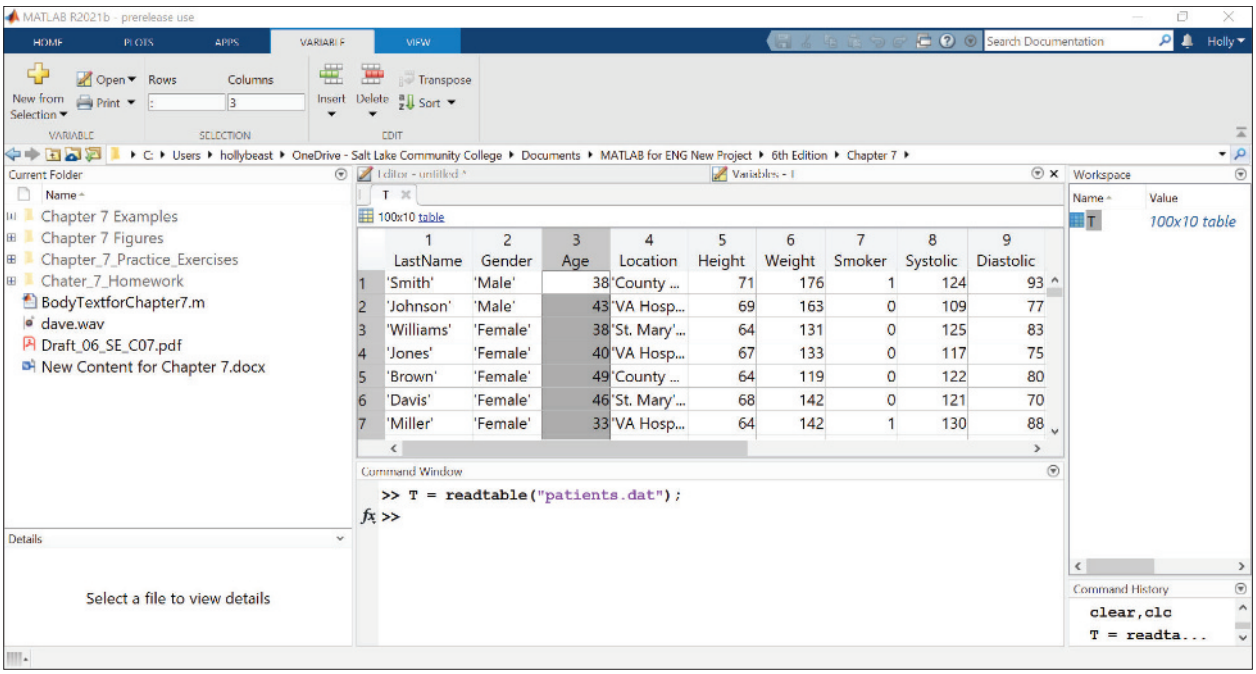


Figure 7.9
The variable editor displays the contents of a table array. Each column corresponds to a column vector stored in the table array. Access the contents using dot syntax, such as `T.age` to retrieve the contents of the third column of the table array shown in the figure.

complete the import task automatically (located along the bottom task bar). If we had checked the box in the previous example, the following function would have been generated.

```
function importfile(fileToRead1)
% IMPORTFILE(FILETOREAD1)
% Imports data from the specified file
% FILETOREAD1: file to read

% Auto-generated by MATLAB on 13-Jun-2021 15:06:01

% Import the file
newData1 = importdata(fileToRead1);
% Create new variables in the base workspace from those fields.
vars = fieldnames(newData1);
for i = 1:length(vars)
    assignin('base', vars{i}, newData1.(vars{i}));
end
```

To use this function save it to your current directory. To programmatically load the data in the `dave.wav` file the code is:

```
importfile("dave.wav")
```

You should be aware that data storage formats are constantly changing, which can affect MATLAB's ability to interpret them. For example, some but not all **.wav** files use a data compression algorithm not supported by MATLAB.

7.4.2 Exporting Data

The easiest way to find the appropriate function for writing a file is to use the help tutorial to find the correct function to read it, then to follow the links to the `write` function. For example, to read an Excel spreadsheet file (*.xls* or *.xlsx*), use `readtable`:

```
readtable("filename.xlsx")
```

At the end of the tutorial page, MATLAB refers you to the correct function for writing an Excel file, namely,

```
writetable(T,"filename")
```

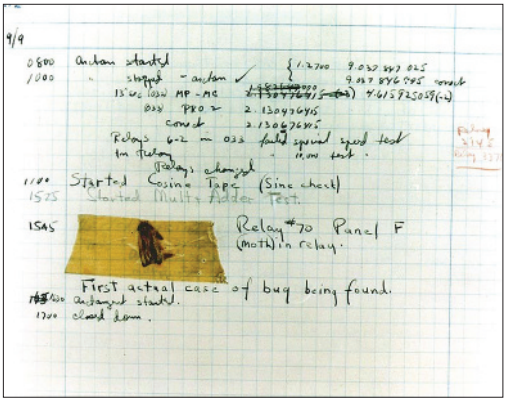
where `T` is the array you want to store in the Excel spreadsheet.

7.5 DEBUGGING YOUR CODE

A software bug is a problem that exists in the code you have written. It can be a mistake that results in the code not working at all (a coding error), or it can be a logic error that results in a wrong answer. The term "bug" has its genesis in electronics, where actual insects sometimes caused equipment failure. Perhaps the most famous example is the moth (see Figure 7.10a) found in the innards of one of the earliest computers, the Harvard Mark II Aiken Relay Calculator, in 1947, by a team of researchers led by the famous early computer scientist Grace Hopper (Figure 7.10b).

MATLAB includes a number of tools to help you debug your code, including the code analyzer (error bar) and more comprehensive tools that allow you to step through the code.

(a)



(b)



Figure 7.10
(a) The moth found trapped in a relay in Harvard's Mark II Aiken Relay Calculator. This is often erroneously reported to be the first use of the term "bug" as a synonym for a computer problem. This page from the computer log book is currently on exhibit in the Smithsonian Institute's National Museum of American History. (Image courtesy of the Naval Surface Warfare Center, Dalgren, VA, 1988.)
(b) Grace Hopper was an American computer scientist and U.S. Navy Rear Admiral. She was the team leader of the group that found the 'bug' in the Harvard Mark II Computer and was largely responsible for sharing the anecdote with the public. Admiral Hopper was a major contributor to the creation of COBOL, the first major computer language written with an English-like syntax. (Picture courtesy of Public.Resource.Org/Flickr)

7.5.1 Code Analyzer

Whenever you create either a script or a live script notice that along the right-hand side of the figure window, a vertical bar appears that marks locations where there are actual errors or where MATLAB has issued warnings. The portion of the code that concerns MATLAB is highlighted. For example, in Figure 7.11, which shows a program written in a script M-file, there are several places marked with a light orange highlight, which indicates a warning. If you run your cursor over the highlight (either in the code or along the bar), a message appears with a suggested fix for the problem. Not every warning corresponds to a real problem. For example, the warnings issued for the program in Figure 7.11 resulted from

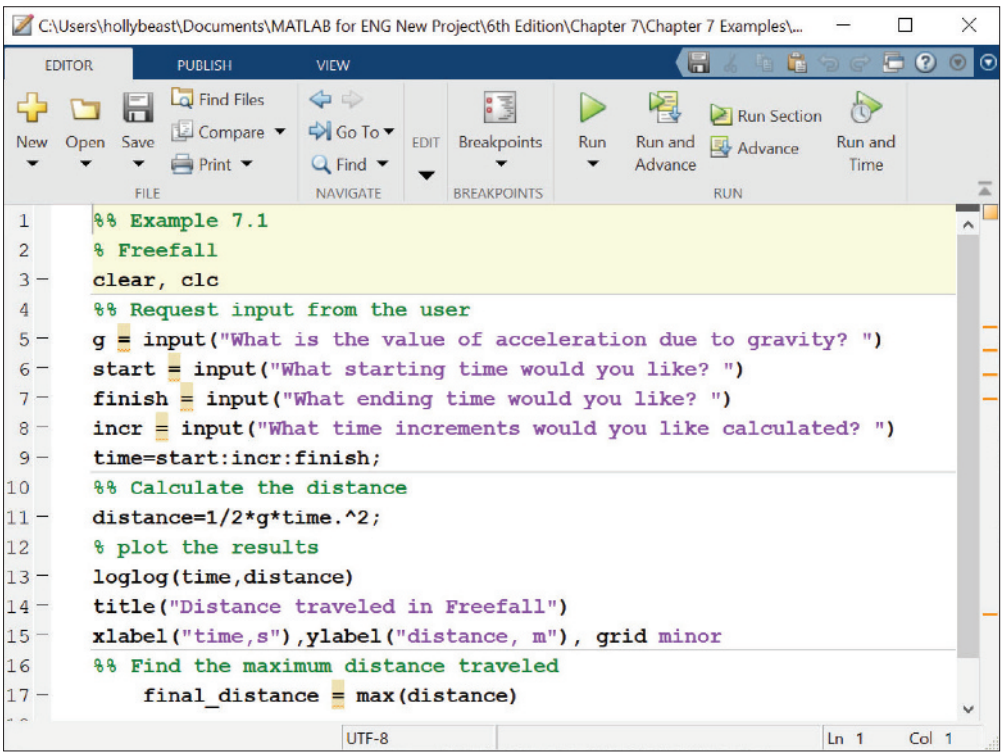
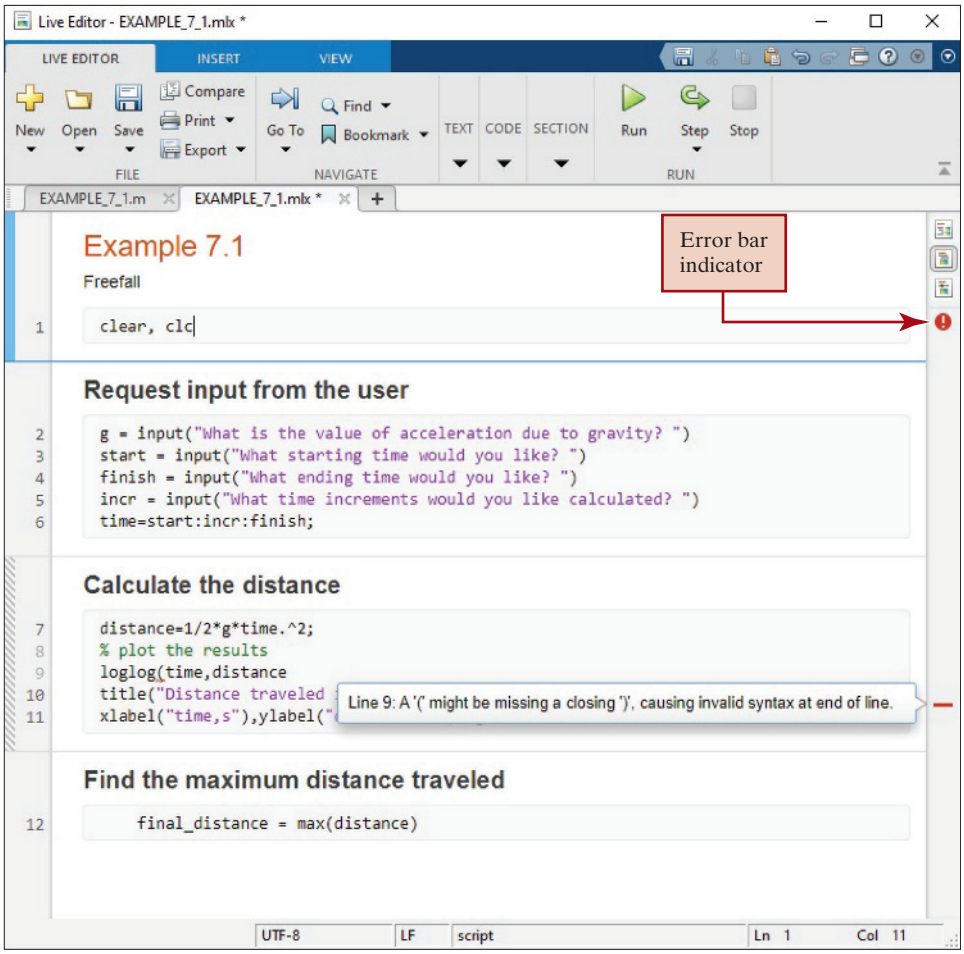


Figure 7.11
The code analyzer on the right-hand side of the screen identifies lines of code with potential errors. Locations in the code with potential errors are indicated with a light orange highlight. This version of the code is written in a script M-file, but the code analyzer bar appears the same in a live script MLX-file.

Figure 7.12
Script with an error on line 9.



lines of code without semicolons at the end of the line. In this particular case, we wanted the code to report answers to the command window; in other cases you might want to suppress the output. You can edit which error messages are shown by selecting

Preferences → Code Analyser

found on the Home tab.

If the errors shown on the code analyzer error bar are marked in red, they will cause the file to stop executing. In Figure 7.12, the code was adjusted to introduce such an error. In this example the code is shown in a live script MLX-file. The code analyzer bar appears almost exactly the same in each of the options for storing the MATLAB code. On line 9, the right-hand parentheses are missing, as indicated by the error message. You can walk through the warnings and actual error messages by clicking on the indicator at the top of the error bar which is a square in the script M-file and a red exclamation point in the live script MLX-file.

7.5.2 Debugging Toolbar

When trying to find logic errors in a piece of code, it is often useful to run sections of the program, then to stop, evaluate what has happened, and continue.

Using **sections** is one way to accomplish this, but a more comprehensive approach is offered by the debugging toolbar. It allows you to set breakpoints (places in the code where the execution stops while you evaluate results) and to step through the code one line at a time. Breakpoints can't be enabled until all of the syntax errors have been resolved.

To set a breakpoint, click next to the line number on the left-hand side of the editing window, or select the *Set/Clear Breakpoint* icon on the toolbar. A red indicator should appear, as shown in Figure 7.13. If the indicator is gray, syntax errors still exist in the program, or you have not saved the most recent version of the code.

When you run the program, the debugging toolbar is activated, the execution will pause at the breakpoint, and the location will be marked with a green arrow in script files or a green highlight in live script files as shown in Figure 7.14. To continue, select the continue icon from the breakpoint toolbar.

You can also choose to step through the code one line at a time, using the **Step** icon. If your code includes calls to user-defined functions, you can step into the function, then step through the function code one line at a time, using the **Step In** icon. To leave the user-defined function, select the **Step Out** icon. For example, Figure 7.15 shows a program that calls the user-defined functions, `DR` and `RD` to change degrees to radians and radians to degrees. These functions are stored as separate files because they have utility for multiple programs. In this case we've chosen to display both the main live script and the first of the function files, `DR`, in the editing window by selecting the VIEW tab, and choosing the left/right option. Notice that the line where we "stepped out" of the main program and into the function is highlighted. Once you've completed the calculations in the function, MATLAB returns to the main program and you can continue working through the remaining code, either one line at a time, or by selecting *Continue* which will move you to the next break point.

KEY IDEA
MATLAB includes debugging tools to help you find errors in your code.

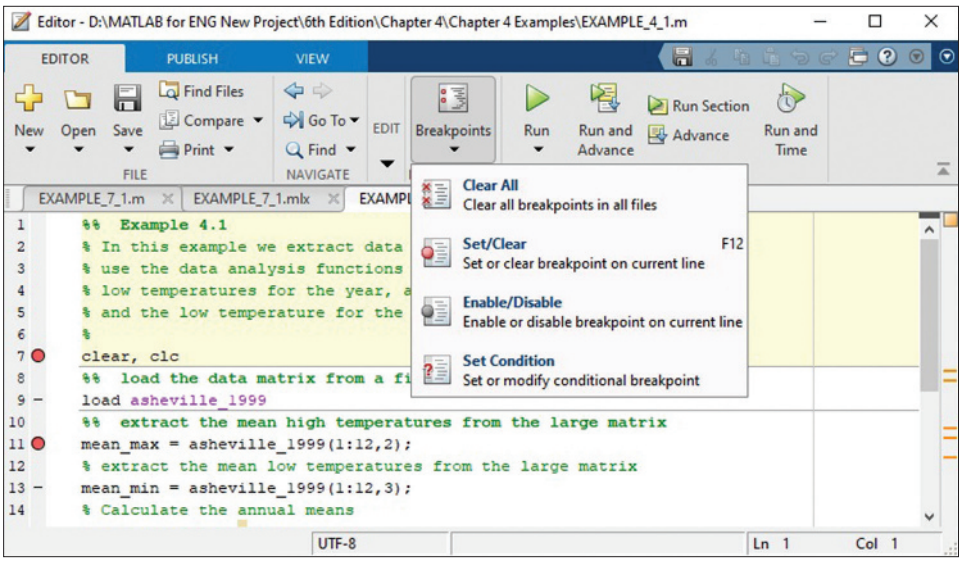


Figure 7.13
MATLAB includes code analysis tools that allow for easier debugging.

Figure 7.14
Breakpoints enable the user to move through the code in small pieces.

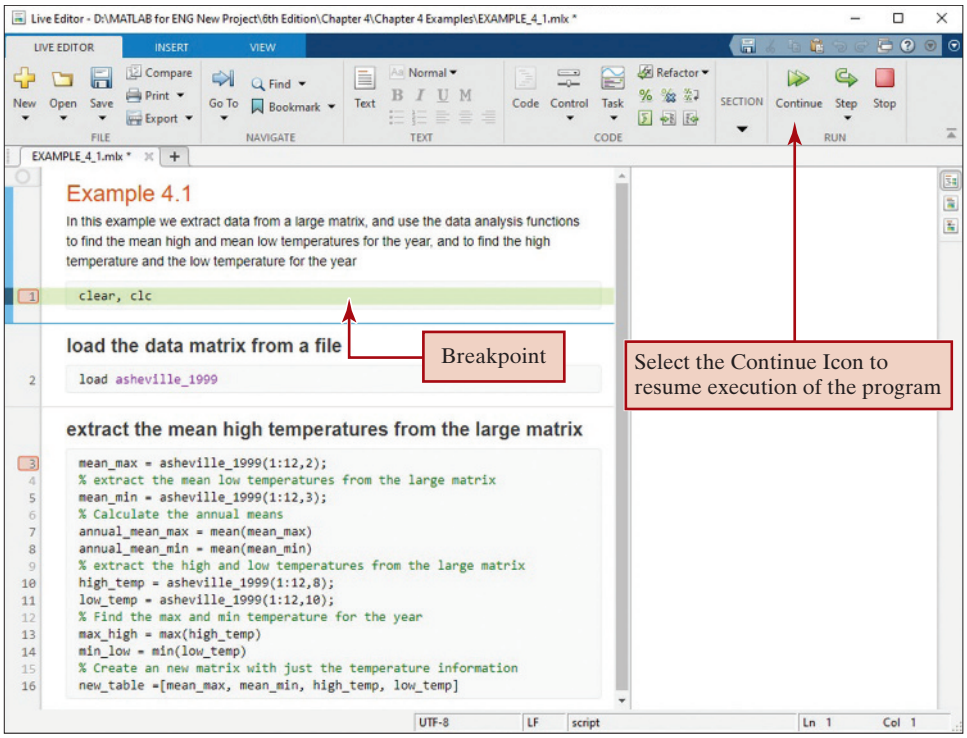
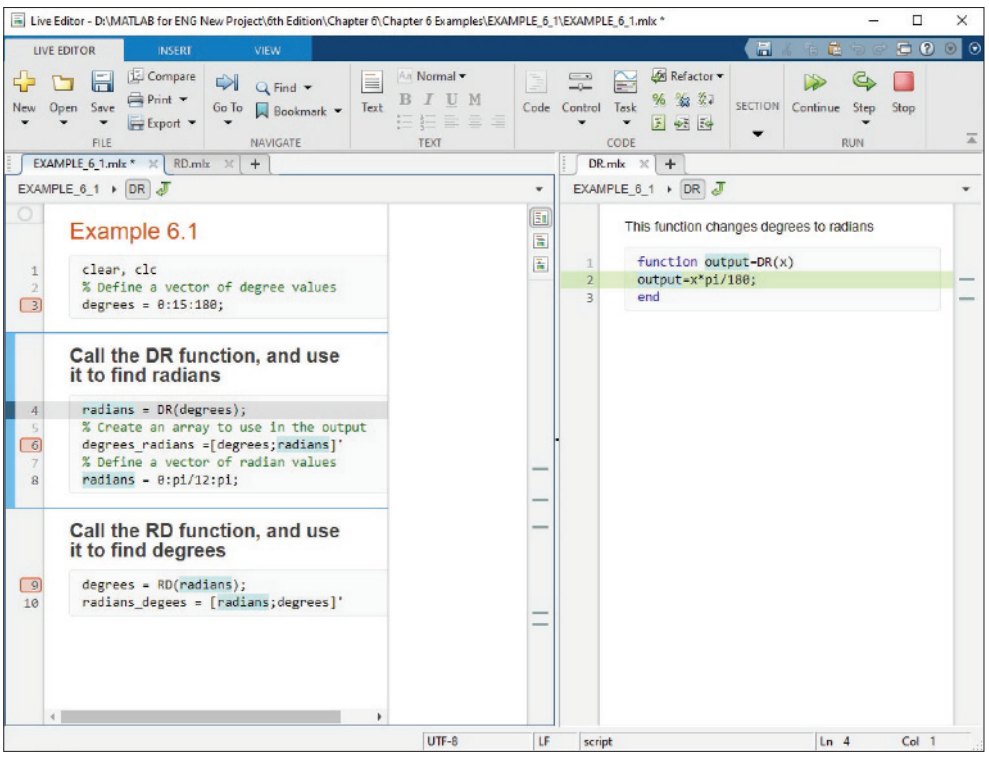


Figure 7.15
The step in option makes it possible to step through user-defined functions one line at a time, as they are called by the main program.



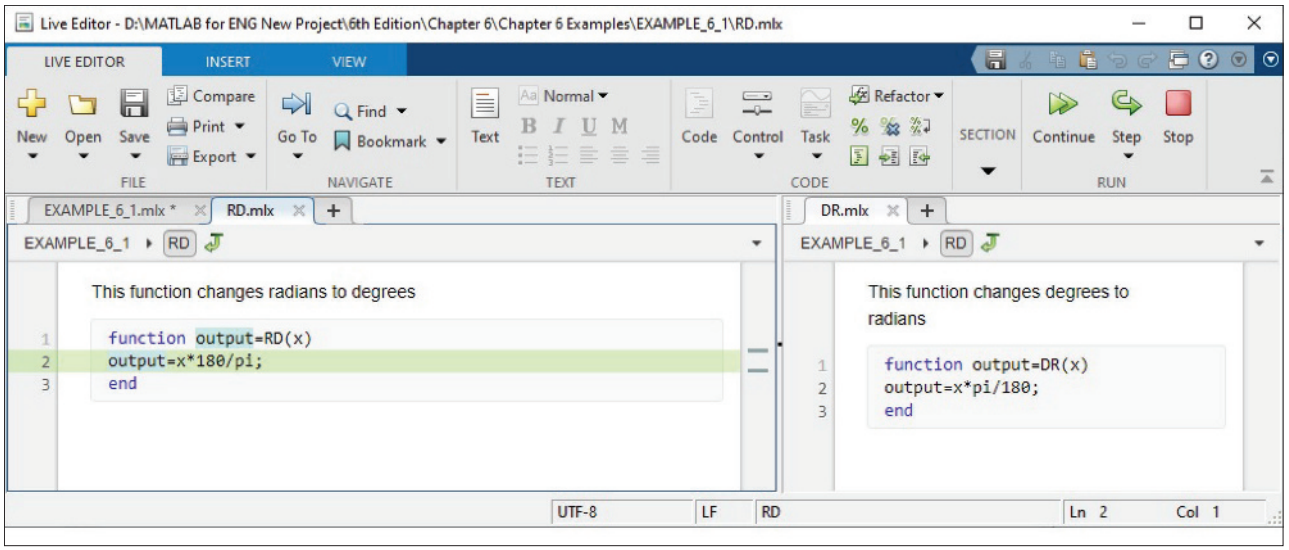


Figure 7.16
User-defined functions may be displayed side-by-side with the main program in the editing window, or can be displayed automatically when called by the MATLAB debugging features.

SUMMARY

MATLAB provides functions that allow the user to interact with a program and allow the programmer to control the output to the command window.

The `input` function pauses the program and sends a prompt determined by the programmer to the command window. Once the user has entered a value or values and hits the return key, program execution continues.

The display (`disp`) function allows the programmer to display the contents of a string or an array in the command window. Although the `disp` function is adequate for many display tasks, the `fprintf` function gives the programmer considerably more control over the way results are displayed. The programmer can combine text and calculated results on the same line, and specify the number format used. The `sprintf` function behaves exactly the same way as the `fprintf` function. However, the result of `sprintf` is assigned a variable name and can be used with other functions that require strings as input. For example, the functions used to annotate graphs such as `title, text,` and `xlabel` all accept strings as input, and therefore will accept the result of the `sprintf` function as input. The `table` function creates tables that combine both text (strings and character arrays) and numeric values, in addition to other variable types.

For applications in which graphical input is required, the `ginput` command allows the user to provide input to a program by selecting points from a graphics window.

MATLAB includes functions that allow the user to import and export data in a number of popular file formats. A complete list of these formats is available in the help tutorial on the File Formats page. The `uiimport` tool automatically determines data file types and allows users to interactively import data, and create corresponding functions to programmatically import data files. Commonly used file types can usually be imported as tables using the `readtable` function, and can be saved using the `writetable` function.

The code analyzer error bar, located on the right-hand side of the editing window, identifies lines of code with potential errors. Warnings are indicated in orange, and errors that will cause the execution of the code to terminate are shown in red. More extensive debugging tools are available from the debugging toolbar.

MATLAB SUMMARY

The following MATLAB summary lists special characters, commands, and functions that were defined in this chapter:

Special Characters	
"	begins and ends a string
`%`	placeholder used in the `fprintf` function
`%f`	fixed-point, or decimal, notation
`%d`	signed integer notation
`%e`	exponential notation
`%g`	either fixed point or exponential notation
`%s`	string notation
`%%`	section divider
`\n`	linefeed
`\r`	carriage return (similar to linefeed)
`\t`	tab
`\b`	backspace

Commands and Functions	
`disp`	displays a string or an array in the command window
`fprintf`	creates formatted output which can be sent to the command window or to a file
`ginput`	allows the user to pick values from a graph
`input`	allows the user to enter values
`num2str`	changes a number to a string
`pause`	pauses the program
`readtable`	imports tabular data into tables
`sound`	plays MATLAB data through the speakers
`sprintf`	similar to `fprintf` creates formatted output which is assigned to a variable name and stored as a character array
`table`	creates a table containing multiple data types
`uiimport`	launches the Import Wizard
`writetable`	exports data stored in MATLAB as a table into popular data formats

KEY TERMS

character array
concatenate
formatted output
precision field
section
step in, step out
string
width field

PROBLEMS

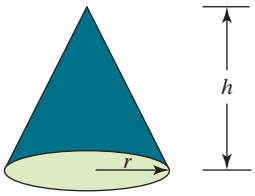


Figure P7.3
Volume of a cone.

Input Function

7.1 Prompt the user to enter a value of x and then calculate the value of $\sin(x)$.

7.2 Prompt the user to enter an array, then use the `max` function to determine the largest value entered. Use the following array to test your program:

```
[1, 5, 3, 8, 9, 22]
```

7.3 The volume of a cone is

$$V = \frac{1}{3} \times \text{area of the base} \times \text{height}.$$

Prompt the user to enter the area of the base and the height of the cone (see Figure P7.3). Calculate the volume of the cone.

`disp` Function

7.4 One of the first computer programs many students write is called "Hello, World." The only thing the program does is print this message to the computer screen. Write a "Hello, World" program using the `disp` function.

7.5 Use two separate `input` statements to prompt a user to enter their first and last names. Use the `disp` function to display those names on one line. (You'll need to combine the names and some spaces into an array.)

7.6 Prompt the user to enter their age. Then use the `disp` function to report the age back to the command window. If, for example, the user enters 5 when prompted for their age, your display should read

```
Your age is 5
```

This output requires combining both character data (a string) and numeric data in the `disp` function—which can be accomplished by concatenating the string and numeric value with the + operator.

7.7 Prompt the user to enter an array of numbers. Use the `length` function to determine how many values were entered, and use the `disp` function to report your results to the command window.

`fprintf`

7.8 Repeat Problem 7.7, and use `fprintf` to report your results.

7.9 Use `fprintf` to create the multiplication tables from 1 to 13 for the number 6. Your output should look like this:

```
1 times 6 is 6
2 times 6 is 12
3 times 6 is 18
      ⋮
```

7.10 Before calculators were readily available (about 1974), students used charts to determine the values of mathematical functions like sine, cosine, and log. These charts often took up entire books. Create such a chart for sine, using the following steps:

- Create a vector of angle values from 0 to 2π in increments of $\pi/10$.
- Calculate the sine of each of the angles, and group your results into an array that includes the angle and the sine.
- Use `disp` to create a title for the chart, and a second `disp` command to create column headings.

- Use the `fprintf` function to display the numbers. Display only two values past the decimal point.

7.11 Very small dimensions—those on the atomic scale—are often measured in angstroms. An angstrom is represented by the symbol Å and corresponds to a length of 10^{-10} m. Create an inches-to-angstroms conversion chart as follows for values of inches from 1 to 10:

- Use `disp` to create a title and column headings.
- Use `fprintf` to display the numerical information.
- Because the length represented in angstroms is so big, represent your result in scientific notation, showing two values after the decimal point. This corresponds to three significant figures (one before and two after the decimal point).

7.12 Use your favorite Internet search engine and browser to identify recent currency conversions for British pounds sterling, Japanese yen, and the European euro to US dollars. Use the conversion tables to create the following charts (use the `disp` and `fprintf` commands in your solution, which should include a title, column labels, and formatted output):

(a) Generate a chart of conversions from yen to US dollars. Start the yen column at 5 and increment by 5 yen. Print 25 lines in the chart.
(b) Generate a chart of conversions from euros to US dollars. Start the euro column at 1 euro and increment by 2 euros. Print 30 lines in the chart.
(c) Generate a chart with four columns. The first should contain US dollars, the second the equivalent number of euros, the third the equivalent number of pounds, and the fourth the equivalent number of yen. Let the dollar column vary from 1 to 10.

`table`

7.13 Consider the patient data contained in table P7.13.

Table P7.13 Patient Data

Patient Last Name	Patient First Name	Age	Height (inches)	Weight (lb)
Smith	Jessica	6	47	82
Toledo	Maria	22	66	140
Webb	Milton	92	62	110
Nguyen	Phan	45	72	190

(a) Create a column vector of last names called last, using a string array.
(b) Create a column vector of first names called first, using a string array.
(c) Create column vectors for age, height, and weight.
(d) Display the information using the `table` function.

Problems combining the `input`, `disp`, `fprintf`, and `table` functions

7.14 This problem requires you to generate temperature conversion charts. Use the following equations, which describe the relationships between temperatures in degrees Fahrenheit (T_F), degrees Celsius (T_C), kelvins (T_K), and degrees Rankine (T_R), respectively:

$$T_F = T_R - 459.67°R$$

$$T_F = \frac{9}{5}T_C + 32°F$$

$$T_R = \frac{9}{5}T_K$$

You will need to rearrange these expressions to solve some of the problems.

(a) Generate a chart of conversions from Fahrenheit to Kelvin for values from 0°F to 200°F. Allow the user to enter the increments in degrees F between lines. Use `disp` and `fprintf` to create a chart with a title, column headings, and appropriate spacing.

(b) Generate a chart of conversions from Celsius to Rankine. Allow the user to enter the starting temperature and the increment between lines. Print 25 lines in the chart. Use `disp` and `fprintf` to create a chart with a title, column headings, and appropriate spacing.

(c) Generate a chart of conversions from Celsius to Fahrenheit. Allow the user to enter the starting temperature, the increment between lines, and the number of lines for the table. Use `disp` and `fprintf` to create a chart with a title, column headings, and appropriate spacing.

7.15 Engineers use both English and SI (Système International d'Unités) units on a regular basis. Some fields use primarily one or the other, but many combine the two systems. For example, the rate of energy input to a steam power plant from burning fossil fuels is usually measured in Btu/hour. However, the electricity produced by the same plant is usually measured in joules/s (watts). Automobile engines, by contrast, are often rated in horsepower or in ft $1b_f$/s. Here are some conversion factors relating these different power measurements:

$$1 \text{ kW} = 3412.14 \text{ Btu/h} = 737.56 \text{ ft } 1b_f/\text{s}$$

$$1 \text{ hp} = 550 \text{ ft } 1b_f/\text{s} = 2544.5 \text{ Btu/h}$$

(a) Generate a table of conversions from kW to hp. The table should start at 0 kW and end at 15 kW. Use the `input` function to let the user define the increment between table entries. Use `disp` and `fprintf` to create a chart with a title, column headings, and appropriate spacing.

(b) Repeat part (a) using the `table` function.

(c) Generate a table of conversions from ft $1b_f$/s to Btu/h. The table should start at 0 ft $1b_f$/s but let the user define the increment between table entries and the final table value. Use `disp` and `fprintf` to create a table with a title, column headings, and appropriate spacing.

(d) Repeat part (c) using the `table` function.

(e) Generate a table that includes conversions from kW to Btu/h, hp, and ft $1b_f$/s. Let the user define the initial value of kW, the final value of kW, and the number of entries in the table. Use `disp` and `fprintf` to create output with a title, column headings, and appropriate spacing.

(f) Repeat part (d) using the `table` function.

`ginput`

7.16 At time $t = 0$, when a rocket's engine shuts down, the rocket has reached an altitude of 500 m and is rising at a velocity of 125 m/s. At this point, gravity takes over. The height of the rocket as a function of time is

$$h(t) = -\frac{9.8}{2}t^2 + 125t + 500 \text{ for } t > 0.$$

Plot the height of the rocket from 0 to 30 seconds, and

- Use the `ginput` function to estimate the maximum height the rocket reaches and the time when the rocket hits the ground.
- Use the `disp` command to report your results to the command window.

7.17 The `ginput` function is useful for picking distances off a graph. Demonstrate this feature by doing the following:

- Create a graph of a circle by defining an array of angles from 0 to 2 π, with a spacing of $\pi/100$.
- Use the `ginput` function to pick two points on the circumference of the circle.
- Use `hold on` to keep the figure from refreshing, and plot a line between the two points you picked.
- Use the data from the points to calculate the length of the line between them. (*Hint*: Use the Pythagorean theorem in your calculation.)

Table P7.18 A Comparison of Standard and Hybrid Vehicles

Year	Model	Base MSRP	Gas Efficiency, in-town/highway (mpg)
2021	Prius Prime	$28220	54/133
2021	Toyota Camry	$25045	28/39
2021	Toyota Camry Hybrid	$27270	51/53 (hybrids may actually get better mileage in town than on the road)
2021	Toyota Highlander	$35085	22/29
2021	Toyota Highlander Hybrid	$38735	36/35
2020	Ford Fusion Titanium	$34,450	21/31
2020	Ford Fusion Titanium Hybrid	$34,595	43/41
2020	Ford Fusion Titanium Plug-In Hybrid	$35,000	109/97
2021	Ford F-150	$29,290	19/22
2022	Ford F-150 Lightning	$39,974	85

7.18 In recent years, the price of gasoline has fluctuated dramatically. Automobile companies have responded with more fuel-efficient cars, in particular, hybrid and hybrid plug-in models. But will you save money by purchasing a hybrid such as the Toyota Camry rather than a Camry with a standard engine? The hybrid vehicles are considerably more expensive, but get better gas mileage. Consider the vehicle prices and gas efficiencies shown in Table P7.18.
One way to compare two vehicles is to find the "cost to own."

$$\text{Cost to own} = \text{Purchase cost} + \text{Upkeep} + \text{Gasoline cost}$$

Assume for this exercise that the upkeep costs are the same, so in our comparison we'll set them equal to zero.

(a) What do you think the cost of gasoline will be over the next several years? Prompt the user to enter an estimate of gasoline cost in dollars/gallon.
(b) Find the "cost to own" as a function of the number of miles driven for a pair of vehicles from the table, based on the fuel price estimate from part (a). Plot your results on an *x*–*y* graph. The point where the two lines cross is the break-even point.
(c) Use the `ginput` function to pick the break-even point off the graph.
(d) Use `sprintf` to create a string identifying the break-even point, and use the result to create a text-box annotation on your graph. Position the text box using the `gtext` function.

7.19 Save your homework assignment as a Live Script. Run the file, with the results displayed on the right-hand side of the page. You will achieve a better looking and more functional result if you start each problem as a new section.

CHAPTER

8 Logical Functions and Selection Structures

Objectives

After reading this chapter, you should be able to:

- Use relational and logical operators in comparison statements to make programming decisions.
- Solve problems using the `find` function to select appropriate programming paths.
- Use logical indexing to access elements of an array.
- Differentiate between the appropriate uses of the `if/else` family of commands.
- Use the `switch/case` structure with both numeric and textual input.
- Explain the advantages and disadvantages of the logical functions, indexing approaches and selection structures available in MATLAB.

INTRODUCTION

KEY IDEA
Control structures determine the order in which individual components of a computer program are executed.

One way to think of a computer program (not just MATLAB) is to consider how the statements that compose it are organized. Usually, sections of computer code can be categorized as **sequences**, **selection structures**, and **repetition structures** (see Figure 8.1). These are called **control structures**. So far, we have written code that contains sequences, but none of the other structures:

- A sequence is a list of commands that are executed one after another.
- A selection structure allows the programmer to execute one command (or set of commands) if some criterion is true and a second command (or set of commands) if the criterion is false. A selection statement provides the means of choosing between these paths, based on a **logical condition**. The conditions that are evaluated often contain both **relational** and **logical operators** or functions.
- A repetition structure, or **loop**, causes a group of statements to be executed multiple times. The number of times a loop is executed depends on either a counter or the evaluation of a logical condition.

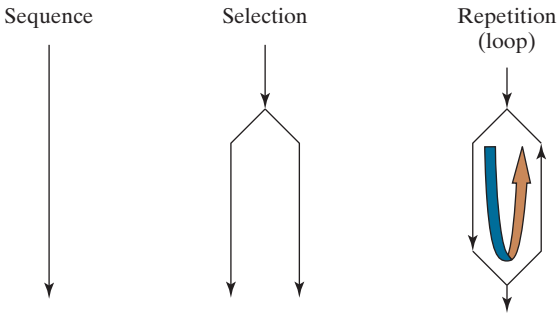


Figure 8.1
Programming structures used in MATLAB.

8.1 RELATIONAL AND LOGICAL OPERATORS

The selection and repetition structures used in MATLAB depend on relational and logical operators. MATLAB has six relational operators for comparing two arrays of equal size, as shown in Table 8.1.

Comparisons are either true or false, and most computer programs (including MATLAB) use the number 1 for true and 0 for false. (MATLAB actually takes any number that is not 0 to be true.) If we define two scalars

```
x = 5;
y = 1;
```

and use a relational operator such as <, the result of the comparison

```
x<y
```

is either true or false. In this case, x is not less than y, so MATLAB responds

```
ans =
     0
```

indicating that the comparison is false. MATLAB uses this answer in selection statements and in repetition structures to make decisions.

Of course, variables in MATLAB usually represent entire arrays. If we redefine x and y, we can see how MATLAB handles comparisons between arrays. For example,

```
x = 1:5;
y = x -4;
x<y
```

KEY IDEA
Relational operators compare values.

Table 8.1 Relational Operators

Relational Operator	Interpretation
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
~=	not equal to

returns

```
ans =
     0   0   0   0   0
```

MATLAB compares corresponding elements and creates an answer array of zeros and ones. In the preceding example, x was greater than y for every comparison of elements, so every comparison was false, and the answer was a string of zeros. If, instead, we have

```
x = [ 1, 2, 3, 4, 5];
y = [-2, 0, 2, 4, 6];
x<y
```

then

```
ans =
     0   0   0   0   1
```

The results tell us that the comparison was false for the first four elements, but true for the last. For a comparison to be true for an entire array, it must be true for *every* element in the array. In other words, all the results must be ones.

HINT

One common source of confusion is the difference between the equals operator (==) and the assignment operator (=). The equals operator (==) compares two values and returns a result of true if they are the same and false if they are different. The assignment operator (=) assigns the value on the right-hand side of the expression to a memory location. For example, the statement

```
x = 5
```

assigns the value 5 to a memory location named x. Once we've made that assignment the comparison

```
x == 3
```

returns a value of 0, since the comparison is false

```
ans =
     0
```

KEY IDEA
Logical operators are used to combine comparison statements.

MATLAB also allows us to combine comparisons with the logical operators *and*, *not*, and *or* (see Table 8.2).

Table 8.2 Logical Operators

Logical Operator	Interpretation
&	and
~	not
|	or
xor	exclusive or
&&	short circuit and (used with scalars)
||	short circuit or (used with scalars)

The code

```
x = [ 1, 2, 3, 4, 5];
y = [-2, 0, 2, 4, 6];
z = [ 8, 8, 8, 8, 8];
z>x & z>y
```

returns

```
ans =
     1  1  1  1  1
```

because z is greater than both x and y for every element. The statement

```
x>y | x>z
```

is read as "x is greater than y or x is greater than z" and returns

```
ans =
  1  1  1  0  0
```

This means that the condition is true for the first three elements, and false for the last two.

MATLAB supports the && (shortcut and) and || (shortcut or) operators; however, these operators apply only to scalar quantities. To save computational time, they evaluate only enough of the statement to determine whether it is true or false. For example, in the statement

```
5<3 && 10>2
```

the first comparison is false, so the result of the second comparison is irrelevant. Similarly, in the statement

```
12>3 || 2>5
```

the first comparison is true, making the second comparison irrelevant.

These relational and logical operators are used in both selection structures and loops to determine what commands should be executed.

8.2 FLOWCHARTS AND PSEUDOCODE

With the addition of selection and repetition structures to your group of programming tools, it becomes even more important to plan your program before you start coding. Two common approaches are to use **flowcharts** and **pseudocode**. A flowchart is a graphical approach to creating your coding plan. Pseudocode is a verbal description of your plan. You may want to use either or both for your programming projects.

KEY IDEA
Flowcharts and pseudocode are used to plan programming tasks.

For simple programs, pseudocode may be the best (or at least the simplest) planning approach:

- Outline a set of statements describing the steps you will take to solve a problem.
- Convert these steps into comments in a MATLAB program.
- Insert the appropriate MATLAB code into the file between the comment lines.

Here's a really simple example: Suppose you've been asked to create a program to convert mph to ft/s. The output should be a chart, complete with a title and column headings. Here's an outline of the steps you might follow:

- Define a vector of mph values.
- Convert mph to ft/s.
- Combine the mph and ft/s vectors into an array.
- Create a chart title.
- Create column headings.
- Display the chart.

KEY IDEA
Convert pseudocode into comments, then create code based on the comments.

Once you've identified the steps, put them into a MATLAB program as comments:

```
% Define a vector of mph values
% Convert mph to ft/s
% Combine the mph and ft/s vectors into an array
% Create a table title
% Create column headings
% Display the table
```

Now insert the appropriate MATLAB code into the program

```
%% Define a vector of mph values
  mph = 0:10:100;
% Convert mph to ft/s
  fps = mph*5280/3600;
% Combine the mph and ft/s vectors into an array
  chart = [mph;fps]
% Create a chart title
  disp("Velocity Conversion Table")
% Create column headings
  disp(" mph f/s")
% Display the chart
  fprintf("%8.0f %8.2f \n",chart)
```

If you put some time into your planning, you probably won't need to change the pseudocode much, once you start programming.

HINT

Students often think programming using the strategy described in the previous section (creating a comment describing every line of code) is 'overkill', and skip this step. However, when you are new to creating computer programs it is harder than it first appears to follow what is happening in each step. Taking the time to systematically plan even simple code will make the total time required to complete a project less, not greater. As you acquire expertise you can use shorter comments in your code, but the need for good documentation never goes away.

KEY IDEA
Flowcharts, which are pictorial representations of a computer program, are especially appropriate for complicated programming tasks.

Flowcharts alone, or flowcharts combined with pseudocode, are especially appropriate for more complicated programming tasks. You can create a "big picture" of your program graphically, then convert your project to pseudocode suitable to enter into the program as comments. Before you can start flowcharting, you'll need to be introduced to some standard flowcharting symbols (see Table 8.3).

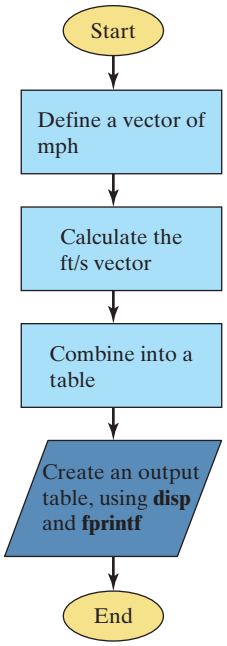


Figure 8.2
Flowcharts make it easy to visualize the structure of a program.

Table 8.3 Flowcharting for Designing Computer Programs

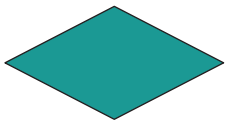
	An oval is used to indicate the beginning or the end of a section of code.
	A parallelogram is used to indicate input or output processes.
	A diamond indicates a decision point.
	Calculations are placed in rectangles.

Figure 8.2 is an example of a flowchart for the mph-to-ft/s problem. For a problem this simple, you would probably never actually create a flowchart. However, as problems become more complicated, flowcharts become an invaluable tool, allowing you to organize your thoughts.

Once you've created a flowchart, you should transfer the ideas into comment lines in a MATLAB program, then add the appropriate code between the comments.

Remember, both flowcharts and pseudocode are tools intended to help you create better computer programs. They can also be used effectively to illustrate the structure of a program to nonprogrammers, since they emphasize the logical progression of ideas over programming details.

8.3 LOGICAL FUNCTIONS

MATLAB offers both traditional selection structures, such as the family of `if` functions, and a series of logical functions that perform much the same task. The primary logical function is `find`, which can often be used in place of both traditional selection structures and loops. The `find` function, when combined with logical operators, is a powerful tool that allows users to analyze and manipulate data effectively.

HINT

Experienced programmers who are learning MATLAB often ignore logical functions in favor of `if` statements and loops, because they already understand how these structures work. However, logical functions such as `find` can make your code easier to understand and can make it execute faster. In particular, you may be able to avoid nested loops—loops, inside of loops, inside of loops, by strategically using a `find` function. Nested loops are notoriously difficult to understand and are often sources of logic errors.

8.3.1 Find

The `find` command searches an array and identifies which elements in that array meet a given criterion. For example, the U.S. Naval Academy requires applicants to be at least 5′6"(66") tall. Consider this list of applicant heights:

```
height = [63,67,65,72,69,78,75];
```

You can find the **index** numbers of the elements that meet the criterion by using the `find` command:

```
accept = find(height>=66)
```

This command returns

```
accept =
    2   4   5   6   7
```

KEY IDEA
Logical functions are often more efficient programming tools than traditional selection structures.

The `find` function returns the index numbers from the array that meet the criterion. If you want to know what the actual heights are, you can call each element, using the index number:

```
height(accept)
ans =
     67   72   69   78   75
```

An alternative approach would be to nest the commands:

```
height(find(height>=66))
```

You could also determine which applicants do *not* meet the criterion. Use

```
decline = find(height<66)
```

which gives

```
decline =
     1    3
```

To create a more readable report, use the `disp` and `fprintf` functions:

```
disp("The following candidates meet the height requirement");
fprintf("Candidate #%2.0f is%3.0f inches tall\n",[accept;
height(accept)])
```

These commands return the following output:

```
The following candidates meet the height requirement

Candidate # 2 is 67 inches tall
Candidate # 4 is 72 inches tall
Candidate # 5 is 69 inches tall
Candidate # 6 is 78 inches tall
Candidate # 7 is 75 inches tall
```

You could also create a chart of those who do not meet the requirement:

```
disp("These candidates do not meet the height requirement")
fprintf("Candidate #%2.0f is%3.0f inches tall\n",[decline;
height(decline)])
```

Similar to the previous code, the following output is returned.

```
These candidates do not meet the height requirement

Candidate # 1 is 63 inches tall
Candidate # 3 is 65 inches tall
```

You can create fairly complicated search criteria that use logical operators. For example, suppose the applicants must be at least 18 years old and less than 35 years old. Then, your data might look like this:

Height (inches)	Age (years)
63	18
67	19
65	18
72	20
69	36
78	34
75	12

There are many ways to solve this problem. One approach would be to create a single array, called applicants, that contains all the data.

```
applicants = [63, 18; 67, 19; 65, 18; 72, 20; 69, 36;
              78, 34; 75, 12]
```

We then can analyze the data with a `find` statement, first checking column one with a logical statement to determine which applicants are greater than or equal to 66 inches tall and then comparing that list to the applicants in column 2 who are greater than or equal to 18 years old and less than or equal to 35 years old.

```
pass = find(applicants(:,1)>=66 & applicants(:,2)>=18
            & applicants(:,2) <= 35)
```

MATLAB returns

```
pass =
   2
   4
   6
```

the list of applicants that meet all the criteria. We could use `fprintf` to create a nicer output. First, create an array of the data to be displayed:

```
results = [pass,applicants(pass,1),applicants(pass,2)]';
```

Then use `fprintf` to send the results to the command window:

```
fprintf("Applicant #%d is %2.0f inches tall and %2.0f years
old\n",results)
```

The resulting list is

```
Applicant #2 is 67 inches tall and 19 years old
Applicant #4 is 72 inches tall and 20 years old
Applicant #6 is 78 inches tall and 34 years old
```

So far, we've used `find` only to return a single index number. If we define two outputs from `find`, as in

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

it will return the appropriate row and column numbers (also called the row and column index numbers).

Here's another example. Imagine that you have an array of patient temperature values measured in a clinic. The column represents the ID number of the station where the temperature was taken. Thus, the command

```
temp = [95.3, 100.2, 98.6; 97.4, 99.2, 98.9; 100.1, 99.3, 97]
```

gives

```
temp =
   95.3000   100.2000   98.6000
   97.4000    99.2000   98.9000
  100.1000    99.3000   97.0000
```

and

```
index = find(temp>98.6)'
```

gives us the element number for the single-index representation:

```
index =
      3    4    5    6    8
```

Notice that the `element` array was transposed so that it would print as a row instead of a column to save space.

When the `find` command is used with a two-dimensional array, it uses an element-numbering scheme that works down each column one at a time. For example, consider our 3×3 array. The element index numbers are shown in Figure 8.3. The elements that contain values greater than 98.6 are shown in bold.

In order to determine the row and column numbers, we need the syntax

KEY IDEA
MATLAB is column dominant.

```
[row, col] = find(temp>98.6)
```

which gives us the following row and column numbers:

```
row =
     3
     1
     2
     3
     2

col =
     1
     2
     2
     2
     3
```

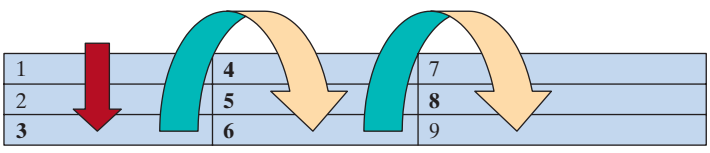


Figure 8.3
Element-numbering sequence for an array. The elements that meet the criterion are shown in bold.

1, 1	**1, 2**	1, 3
2, 1	**2, 2**	**2, 3**
3, 1	**3, 2**	3, 3

Figure 8.4
Row, element designation for a 3 × 3 array. The elements that meet the criterion are shown in bold.

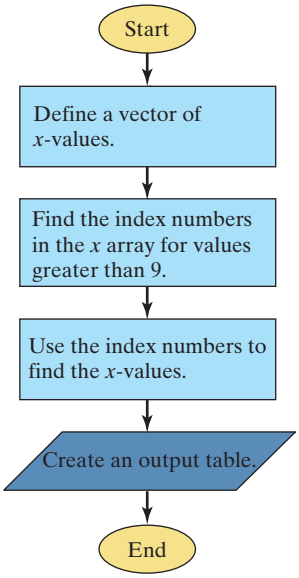


Figure 8.5
Flowchart illustrating the `find` command.

Together, these numbers identify the elements shown in Figure 8.4. Using `fprintf`, we can create a more readable report. For example,

```
fprintf("Patient%3.0f at station%3.0f had a temp of%6.1f\n",
[row,col,temp(index')]')
```

returns

```
Patient 3 at station 1 had a temp of 100.1
Patient 1 at station 2 had a temp of 100.2
Patient 2 at station 2 had a temp of 99.2
Patient 3 at station 2 had a temp of 99.3
Patient 2 at station 3 had a temp of 98.9
```

8.3.2 Flowcharting and Pseudocode for `find` Commands

The `find` command returns only one answer: a vector of the index numbers requested. For example, you might flowchart a sequence of commands, as shown in Figure 8.5. If you use `find` multiple times to separate an array into categories, you may choose to employ a diamond shape, indicating the use of `find` as a selection structure.

```
% Define a vector of x-values
 x = [1,2,3;10,5,1;12,3,2;8,3,1]
% Find the index numbers where x>9
 index = find(x>9)
% Use the index numbers to find the x-values
% greater than 9 by plugging them into x
 values = x(index)
% Create an output table
 disp('Elements greater than 9')
 disp(table(index,values,'VariableNames',["Index Number",
"X Value"]))
```

Notice in this example we chose to display the output with the `table` function instead of `fprintf`. This is a matter of style. Either approach could be used.

EXAMPLE 8.1

SIGNAL PROCESSING USING THE SINC FUNCTION

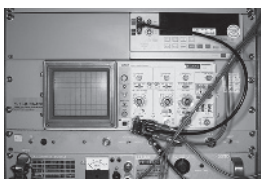


Figure 8.6
Oscilloscopes are widely used in signal-processing applications. (Erik Von Weber/Photodisc/Getty Images)

The `sinc` function is used in many engineering applications, but especially in signal processing (see Figure 8.6). Unfortunately, this function has two widely accepted definitions:

$$f_1(x) = \frac{\sin(\pi x)}{\pi x} \text{ and } f_2(x) = \frac{\sin(x)}{x}$$

Both of these functions have an indeterminate form of $0/0$ when $x = 0$. In this case, l'Hôpital's theorem from calculus can be used to prove that both functions are equal to 1 when $x = 0$. For values of x not equal to zero, the two functions have a similar form. The first function, $f_1(x)$, crosses the x-axis when x is an integer; the second function crosses the x-axis when x is a multiple of π.

Suppose you would like to define a function called `sinc_x` that uses the second definition. Test your function by calculating values of `sinc_x` for x from -5π to $+5\pi$ and plotting the results.

1. State the problem
   Create and test a function called `sinc_x`, using the second definition:

$$f_2(x) = \frac{\sin(x)}{x}$$

2. Describe the input and output

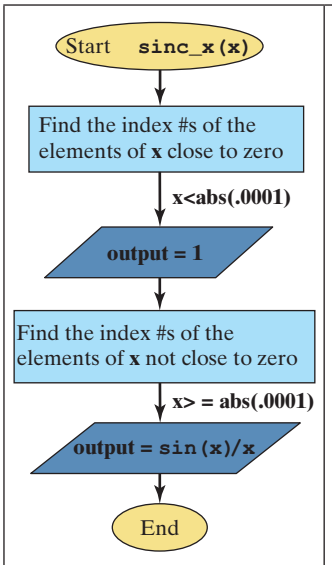
Input	Let x vary from -5π to $+5\pi$ to
Output	Create a plot of x versus `sinc_x`

3. Develop a hand example

4. Develop a MATLAB solution
   Outline your function in a flowchart, as shown in Figure 8.7. Then convert the flowchart to pseudocode comments, and insert the appropriate MATLAB code. Once we've created the function, we should test it in the command window:

```
sinc_x(0)
ans =
       1
sinc_x(pi/2)
ans =
       0.6366
sinc_x(pi)
ans =
       3.8982e-017
sinc_x(-pi/2)
ans =
       0.6366
```

Notice that `sinc_x(pi)` equals a very small number, but not zero. That is because MATLAB treats π as a floating-point number and uses an approximation of its real value (see Table 8.4).




```
function output = sinc_x(x)
% This function finds the value of sinc,
% using the second definition,
% sin(x)/x
% Determine the index #s of the
% elements in the x array that are close to 0
      set1 = find(abs(x)<0.0001);
% Set those elements in the output
% array equal to 1
      output(set1) = 1;
% Determine the index #s of the
% elements in the x array that are not
% close to 0
      set2 = find(abs(x)>=0.0001);
% Calculate sin(x)/x for the elements
% that are not close to 0,
% and assign the results to the corresponding
% output array elements
      output(set2) = sin(x(set2))./x(set2);
end
```

Figure 8.7
Flowchart of the logic to define the `sinc` function using `find`.

(*continued*)

Table 8.4 Calculating the Sinc Function

x	sin(x)	sinc_x(x) = sin(x)/x
0	0	$0/0 = 1$
$\pi/2$	1	$1/(\pi/2) = 0.637$
π	0	0
$-\pi/2$	-1	$-1/(\pi/2) = -0.637$

5. Test the solution
   When we compare the results with those of the hand example, we see that the answers match. Now we can use the function confidently in our problem.
   We have

```
%% Example 8.1
       clear, clc
% Define an array of angles
      x = -5*pi:pi/100:5*pi;
% Calculate sinc_x
      y = sinc_x(x);
% Create the plot
      plot(x,y)
      title("Sinc Function")
      xlabel("angle, radians")
      ylabel("sinc")
```

which generates the plot in Figure 8.8.

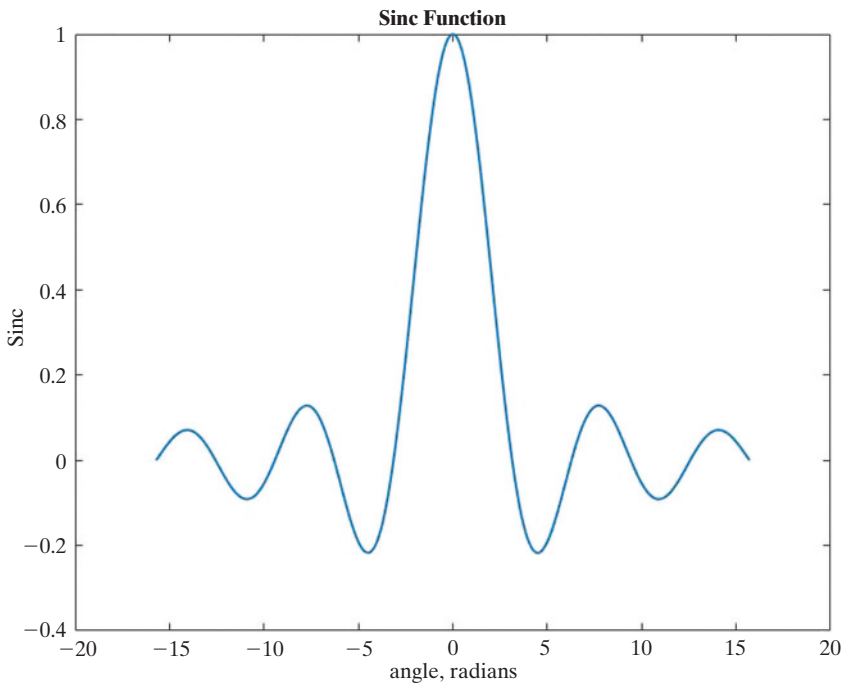


Figure 8.8
The `sinc` function.

The plot also supports our belief that the function is working properly. Testing `sinc_x` with one value at a time validated its answers for a scalar input; however, the program that generated the plot sent a vector argument to the function. The plot confirms that it also performs properly with vector input.

If you have trouble understanding how this function works, remove the semicolons that are suppressing the output, then run the program. Understanding the output from each line will help you understand the program logic better.

In addition to `find`, MATLAB offers two other logical functions: `all` and `any`. The `all` function checks to see if a logical condition is true for *every* member of an array, and the `any` function checks to see if a logical condition is true for *any* member of an array. Consult MATLAB's built-in `help` function for more information.

PRACTICE EXERCISE 8.1

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore. Consider the following arrays:

$$x = \begin{bmatrix} 1 & 10 & 42 & 6 \\ 5 & 8 & 78 & 23 \\ 56 & 45 & 9 & 13 \\ 23 & 22 & 8 & 9 \end{bmatrix}, \; y = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 10 & 12 \\ 7 & 21 & 27 \end{bmatrix}, \; z = \begin{bmatrix} 10 & 22 & 5 & 13 \end{bmatrix}$$

1. Using single-index notation, find the index numbers of the elements in each array that contain values greater than 10.
2. Find the row and column numbers (sometimes called **subscripts**) of the elements in each array that contain values greater than 10.
3. Find the values in each array that are greater than 10.
4. Using single-index notation, find the index numbers of the elements in each array that contain values greater than 10 and less than 40.
5. Find the row and column numbers for the elements in each array that contain values greater than 10 and less than 40.
6. Find the values in each array that are greater than 10 and less than 40.
7. Using single-index notation, find the index numbers of the elements in each array that contain values between 0 and 10 or between 70 and 80.
8. Use the `length` command together with results from the `find` command to determine how many values in each array are between 0 and 10 or between 70 and 80.

8.4 LOGICAL INDEXING

The `find` function returns the indices of elements that meet specified test criteria. **Logical indexing** is similar. Its purpose is to select the elements of an array based on specified test criteria. Imagine we have a string array of patient names, and a corresponding vector of temperature readings.

```
Patient_Names = ["Jason","Jose","Wesley","Rose"]
Temp = [98.2, 100.3, 97, 101]
```

We'd like to determine which patients have a fever. We could use the find function to accomplish the task, but we could also use logical indexing.

The first step is to create a **logical array** indicating whether the criteria we are interested in has been met. For example:

```
fever = Temp>98.6
```

returns

```
fever =
  1×4 logical array
   0   1   0   1
```

KEY IDEA
Use logical indexing to select elements of an array based on test criteria.

where 0 indicates the condition is false and 1 indicates it is true. Now if we use this array as input to the `Patient_Names` vector the result becomes

```
ans =
  1×2 string array
    "Jose"    "Rose"
```

The whole process could actually be completed in a single step with the following syntax

```
Patient_Names(Temp>98.6)
```

which returns exactly the same result as the two-step process detailed above. To determine those patients who do not meet the criteria, use the not operator ~.

```
Patient_Names(~(Temp>98.6))


ans =
  1×2 string array
    "Jason"    "Wesley"
```

Another use for logical indexing is to modify the values in an existing array, or to create a new array based on the values (like the temperature data) in an existing array. Perhaps we'd like to create a vector with test results for these patients, called results. The code

```
result(fever)="Sick"
```

and

```
result(Temp>98.6)="Sick"
```

gives the same result and returns a vector with some missing values:

```
result =
  1×4 string array
        <missing>    "Sick"    <missing>    "Sick"
```

The missing values are filled in when we determine the index values where the criteria are not met,

```
result(~fever)="Well"
```

```
or
    result(~(Temp>98.6))="Well"
```

giving us a completed vector (string array).

```
result =
  1×4 string array
    "Well"    "Sick"    "Well"    "Sick"
```

A convenient way to store all this information would be in a table array with some meaningful column headings.

```
T = ["Patients","Temperature","Diagnosis"];
   disp(table(Patients',Temp',result','VariableNames',T));
```

giving an easy to read chart.

Patients	Temperature	Diagnosis
"Jason"	98.2	"Well"
"Jose"	100.3	"Sick"
"Wesley"	97	"Well"
"Rose"	101	"Sick"

EXAMPLE 8.2

USING LOGICAL INDEXING TO CREATE THE `Sinc` FUNCTION

Recall from Example 8.1 that there are two definitions for the `sinc` function, which is widely used in electrical engineering applications:

$$f_1(x) = \frac{\sin(\pi x)}{\pi x} \text{ and } f_2(x) = \frac{\sin(x)}{x}$$

Both of these functions have an indeterminate form of $0/0$ when $x = 0$. However, L'Hopital's theorem can be used to show that both functions are equal to 1 when $x = 0$.

In Example 8.1, we used the `find` function to help us create a user-defined function called `sinc_x` to evaluate the second function for values of x stored in arrays. In this example, we'll recreate the function using logical indexing. As before we'll test it by calculating values of `sinc_x` for x from -5π to $+5\pi$ and plot the results.

1. State the problem
   Create and test a function called `sinc_x`, using the second definition:
   $$f_2(x) = \frac{\sin(x)}{x}$$
   and logical indexing.
2. Describe the input and output

   Input Let x vary from -5π to $+5\pi$

   Output Create a plot of `x` versus `sinc_x`
3. Develop a hand example

(*continued*)

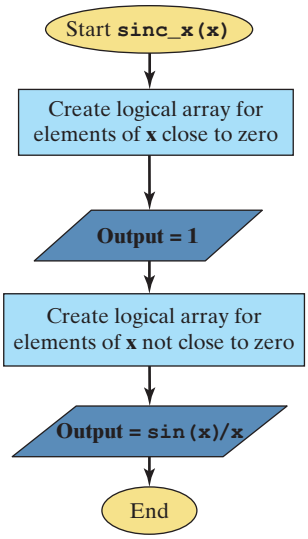


Figure 8.9
Flow chart of the logic to define the `sinc` function using logical indexing.

4. Develop a MATLAB solution
Outline the function in a flowchart, as shown in Figure 8.9. Then convert the flowchart to pseudocode comments and insert the appropriate MATLAB code.

```
function output = sinc_x(x)
% This function finds the value of sinc,
% using the second definition
% sin(x)/x
% Create logical vector to find values of x
% near 0
set1=abs(x)<0.0001;
% Use logical indexing to define the output
output(set1) = 1;
% Create logical vector to find values of x
% not near 0
set2=~(abs(x)<0.0001);
% Use logical indexing to define the output
output(set2) = sin(x(set2))./x(set2);
end
```

5. Test the Solution
Once the function has been created it should be tested. In this case, the function was included in a live script and the results plotted, as shown in Figure 8.10.

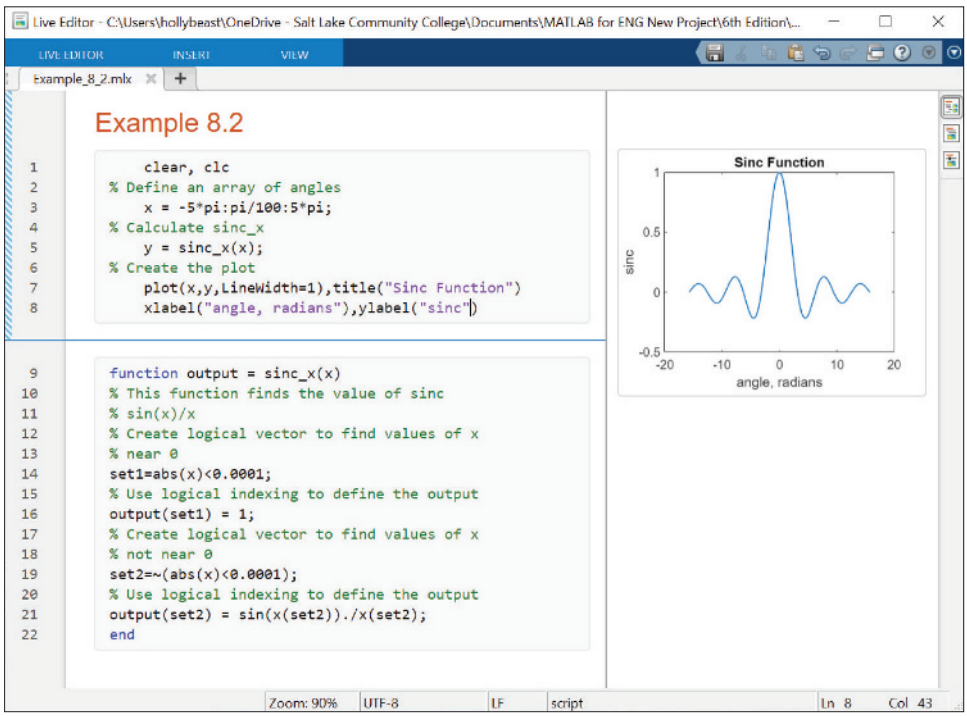


Figure 8.10
Logical Indexing provides an efficient technique for defining different results depending on predetermined criteria. (Reprinted with permission from The Mathworks, Inc.)

PRACTICE EXERCISE 8.2

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore. Consider the following arrays:

$$x = \begin{bmatrix} 1 & 10 & 42 & 6 \\ 5 & 8 & 78 & 23 \\ 56 & 45 & 9 & 13 \\ 23 & 22 & 8 & 9 \end{bmatrix}, y = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 10 & 12 \\ 7 & 21 & 27 \end{bmatrix}, z = \begin{bmatrix} 10 & 22 & 5 & 13 \end{bmatrix}$$

1. Create a logical index array called `k` that represents the elements in `x` where `x` is greater than 10. In other words, the array should have values of 1 (true) in locations where `x` is greater than 10, and values of 0 (false) where `x` is less than or equal to 10. Confirm your results by checking the workspace window and observing that the class is listed as logical, the array size is 4×4, and the icon representing the array is a check box. Repeat for the `y` and `z` arrays, naming your results `j` and `m` respectively.
2. Use the logical index arrays created in problem 1 to extract the values from each array that are greater than 10 and store them in new arrays called `ax`, `ay`, and `az` respectively.
3. In the previous two problems you used a two-step process to extract the values of interest. Repeat the process but do it in a single step.
4. Create logical index arrays called `k`, `j`, and `m` that represent the elements in each of the arrays where the element value is greater than 10 but less than 40.
5. Use the logical index arrays created in problem 4 to extract the values from each array that meet the criteria and store them in new arrays called `bx`, `by`, and `bz`, respectively.
6. Repeat the process used in the previous two problems, but do it in a single step.
7. Imagine these arrays represent some data that was collected. We know that the maximum possible value of the data should be 50. Use logical indexing to replace any values that exceed 50 with a value of 50. You need only use the `x` array for this problem.

8.5 SELECTION STRUCTURES

Most of the time, either the find command or logical indexing command can and should be used instead of an `if` statement. In some situations, however, the `if` statement is required. This section describes the syntax used in `if` statements.

8.5.1 The Simple `if`

A simple `if` statement has the following form:

```
if comparison
    statements
end
```

If the comparison (a logical expression) is true, the statements between the `if` statement and the `end` statement are executed. If the comparison is false, the program jumps immediately to the statement following `end`. It is good programming practice to indent the statements within an `if` structure for readability. However, recall that MATLAB ignores white space. Your programs will run regardless of whether you do or do not indent any of your lines of code.

Here's a simple example of an `if` statement:

```
if G<50
  disp("G is a small value equal to:")
  disp(G);
end
```

KEY IDEA
`if` statements usually work best with scalars.

This statement (from `if` to `end`) is easy to interpret if `G` is a scalar. If `G` is less than 50, then the statements between the `if` and the `end` lines are executed. For example, if `G` has a value of 25, then

```
G is a small value equal to:
    25
```

is displayed. However, if `G` is not a scalar, then the `if` statement considers the comparison true *only if it is true for every element!* in the array. Thus, if `G` is defined from 0 to 80,

```
G = 0:10:80;
```

the comparison is false, and the statements inside the `if` statement are not executed! In general, `if` statements work best when dealing with scalars.

8.5.2 The `if/else` Structure

KEY IDEA
There should never be a comparison on the else line of an if/else structure.

The simple `if` allows us to execute a series of statements if a condition is true, and to skip those steps if the condition is false. The `else` clause allows us to execute one set of statements if the comparison is true, and a different set if the comparison is false. Suppose you would like to take the logarithm of a variable *x*. You know from algebra classes that the input to the `log` function must be greater than 0. Here's a set of `if/else` statements that calculates the logarithm if the input is positive, and sends an error message if the input to the function is 0 or negative:

```
if x >0
   y = log(x)
else
   disp("The input to the log function must be positive")
end
```

Notice that there is no logical comparison on the `else` line. This section of code executes only when the if statement is false, so an additional comparison (for example, `x<=0`) is not needed.

When `x` is a scalar, this code segment is easy to interpret. However, when `x` is an array, the comparison is true only if it is true for every element in the array. So, if

```
x = 0:0.5:2;
```

then the elements in the array are not all greater than 0. Therefore, MATLAB skips to the `else` portion of the statement and displays the error message. The `if/else` statement is probably best confined to use with scalars, although you may find it to be of limited use with vectors.

HINT

MATLAB includes a function called `beep` that causes the computer to "beep" at the user. You can use this function to alert the user to an error. For example, in the `if/else` clause, you could add a beep to the portion of the code that includes an error statement:

```
x = input("Enter a value of x greater than 0:");
if x >0
   y = log(x)
else
   beep
   disp("The input to the log function must be positive")
end
```

This segment of code mimics the `error` function, the only difference being that the `error` function causes the program to abort—whereas the code segment above simply results in an error message going to the command window.

```
x = input("Enter a value of x greater than 0:");
if x >0
   y = log(x)
else
  error("The input to the log function must be positive")
end
```

Try both code segments in MATLAB to observe their behavior.

8.5.3 The `elseif` Structure

When we nest several levels of `if/else` statements, it may be difficult to determine which logical expressions must be true (or false) in order to execute each set of statements. The `elseif` function allows you to check multiple criteria while keeping the code easy to read. Consider the following lines of code that evaluate whether to issue a driver's license, based on the applicant's age:

```
if age<16
   disp("Sorry - You'll have to wait")
elseif age<18
   disp("You may have a youth license")
elseif age<70
   disp("You may have a standard license")
else
   disp("Drivers over 70 require a special license")
end
```

In this example, MATLAB first checks to see `if age<16`. If the comparison is true, the program executes the next line or set of lines, displays the message `Sorry—You'll have to wait`, and then exits the `if` structure. If the comparison is false, MATLAB moves on to the next `elseif` comparison, checking to see if `age < 18` this time. The program continues through the `if` structure until it finally finds a true comparison, or until it encounters the `else`. Notice that the `else` line

does not include a comparison. The only way for the executing code to reach the `else` line is if the `elseif` immediately before it is false.

The flowchart for this sequence of commands (see Figure 8.11) uses the diamond shape to indicate a selection structure.

This structure is easy to interpret if `age` is a scalar. If it is an array, the comparison must be true for every element in the array. Consider this age array

```
age = [15,17,25,55,75]
```

The first comparison, `if age<16`, is false, because it is not true for every element in the array. The second comparison, `elseif age<18`, is also false. The third comparison, `elseif age<70`, is false as well, since not all of the ages are below 70. The result is `Drivers over 70 require a special license`—a result that won't please the other drivers in the group.

HINT

One common mistake new programmers make when using `if` statements is to overspecify the criteria. In the preceding example, it is enough to state that `age < 18` in the `elseif` clause, because age cannot be less than 16 and still reach this statement. You don't need to specify `age < 18 & age >= 16`. If you overspecify the criteria, you risk defining a path for which there is no correct answer. For example, in the code

```
if age<16
   disp("Sorry - You'll have to wait")
```

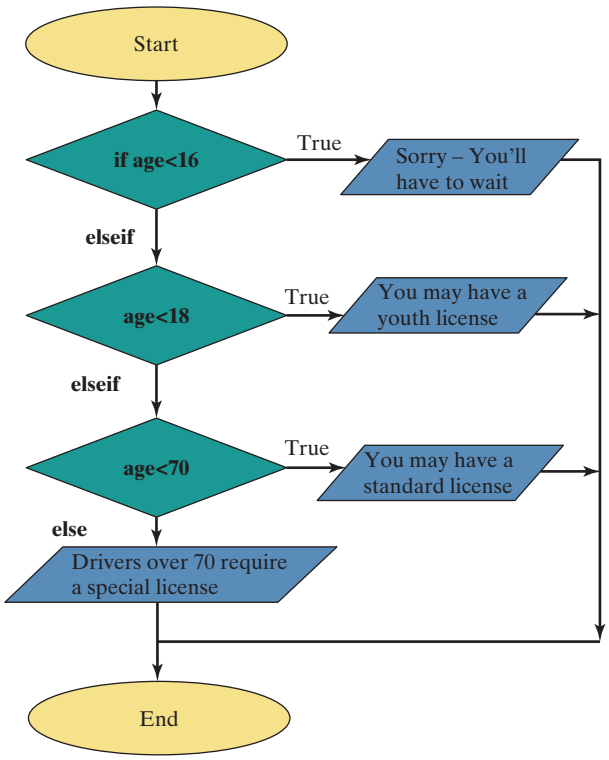


Figure 8.11
Flowchart using multiple `if` statements.

```
elseif age<18 & age>16
   disp("You may have a youth license")
elseif age<70 & age>18
   disp("You may have a standard license")
elseif age>70
   disp("Drivers over 70 require a special license")
end
```

there is no correct choice for age = 16, 18, or 70.

In general, `elseif` structures work well for scalars, but either logical indexing or `find` is a better choice for arrays. Here's an example that uses `find` with an array of ages and generates a table of results in each category:

```
age = [15,17,25,55,75];
set1 = find(age<16);
set2 = find(age>=16 & age<18);
set3 = find(age>=18 & age<70);
set4 = find(age>=70);
fprintf("Sorry - You'll have to wait - " + ...
"you're only %3.0f \n",age(set1))
fprintf("You may have a youth license " + ...
"because you're %3.0f\n",age(set2))
fprintf("You may have a standard license " + ...
"because you're %3.0f \n",age(set3))
fprintf("Drivers 70+ need a special license. " + ...
"You're %3.0f \n",age(set4))
```

These commands return

```
Sorry - You'll have to wait - you're only 15
You need a youth license because you're 17
You may have a license because you're 25
You may have a license because you're 55
Drivers 70+ need a special license. You're 75
```

Since every `find` in this sequence is evaluated, it is necessary to specify the range completely (for example, `age>=16 & age<18`).

EXAMPLE 8.3

ASSIGNING GRADES

The `if` family of statements is used most effectively when the input is a scalar. Create a function to determine test grades based on the score and assuming a single input into the function. The grades should be based on the following criteria:

Grade	Score
A	90 to 100
B	80 to 90
C	70 to 80
D	60 to 70
E	<60

(*continued*)

1. State the problem
   Determine the grade earned on a test.
2. Describe the input and output

   Input Single score, not an array

   Output Letter grade
3. Develop a hand example
   85 should be a B
   But should 90 be an A or a B? We need to create more explicit criteria.

Grade	Score
A	≥90 to 100
B	≥80 and <90
C	≥70 and <80
D	≥60 and <70
E	>60

4. Develop a MATLAB solution
   Outline the function, using the flowchart shown in Figure 8.12.

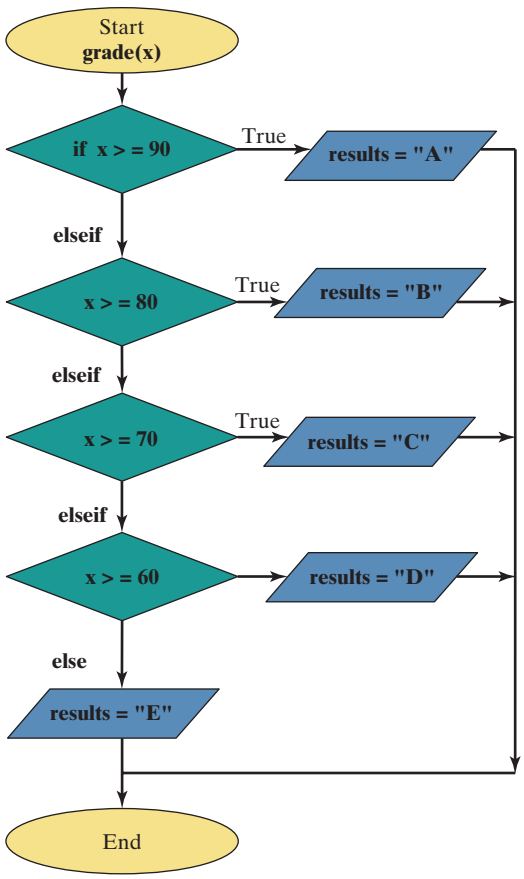


Figure 8.12
Flowchart for a grading scheme.

5. Test the solution
Now test the function in the command window:

```
grade(25)
ans =
E
grade(80)
ans =
B
grade(-52)
ans =
E
grade(108)
ans =
A
```

Notice that although the function seems to work properly, it returns grades for values over 100 and values less than 0. If you'd like, you can now go back and add the logic to exclude those values:

```
function results = grade(x)
% This function requires a scalar input
  if(x>=0 & x<=100)
    if(x>=90)
       results = "A";
    elseif(x>=80)
       results = "B";
    elseif(x>=70)
       results = "C";
    elseif(x>=60)
       results = "D";
    else
       results = "E";
    end
  else
    results = 'Illegal Input';
end
```

An alternative strategy would be to use the built-in `error` function.

```
function results = grade(x)
% This function requires a scalar input
  if(x<0 | x>100)
  error("Illegal Input")
  end
    if(x>=90)
      results = "A";
    elseif(x>=80)
      results = "B";
    elseif(x>=70)
      results = "C";
    elseif(x>=60)
      results = "D";
```

(*continued*)

```
        else
          results = "E";
    end
```

We can test the function again in the command window:

```
grade(-10)
ans =
Illegal Input
grade(108)
ans =
Illegal Input
```

This function works great for scalars, but if you send a vector to the function, you will get some unexpected results, such as

```
score = [95,42,83,77];
grade(score)
ans =
E
```

PRACTICE EXERCISE 8.3

The `if` family of functions is particularly useful in functions. Write and test a function for each of these problems, assuming that the input to the function is a scalar. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Suppose the legal drinking age for a state in the US is 21. Write and test a function to determine whether a person is old enough to drink. If the user enters a negative age, use the `error` function to exit the function and to return an error message.
2. Many rides at amusement parks require riders to be a certain minimum height. Assume that the minimum height is 48" for a certain ride. Write and test a function to determine whether the rider is tall enough to ride. If the user enters a negative height, use the `error` function to exit the function and to return an error message.
3. When a part is manufactured, the dimensions are usually specified with a tolerance. Assume that a certain part needs to be 5.4 cm long, plus or minus 0.1 cm (5.4 ± 0.1 cm). Write a function to determine whether a part is within these specifications. If the user enters a negative length, use the `error` function to exit the function and to return an error message.
4. The United States currently uses both metric and English units. Suppose the part in Exercise 3 was inspected by measuring the length in inches instead of centimeters. Write and test a function that determines whether the part is within specifications and that accepts input into the function in inches. If the user enters a negative length, use the `error` function to exit the function and to return an error message.
5. Many solid-fuel rocket engines consist of three stages. Once the first stage burns out, it separates from the missile and the second stage lights. Then the second stage burns out and separates, and the third stage lights. Finally,

once the third stage burns out, it also separates from the missile. Assume that the following data approximately represent the times during which each stage burns:

Stage 1	0–100 seconds
Stage 2	100–170 seconds
Stage 3	170–260 seconds

Write and test a function to determine whether the missile is in Stage 1 flight, Stage 2 flight, Stage 3 flight, or free flight (unpowered). If the user enters a negative time, use the `error` function to exit the function and to return an error message.

8.5.4 The switch/case Structure

The `switch/case` structure is often used when a series of programming path options exists for a given variable, depending on its value. The `switch/case` is similar to the `if/elseif/else`. As a matter of fact, anything you can do with `switch/case` could be done with `if/elseif/else`. However, the code is a bit easier to read with `switch/case`, a structure that allows you to choose between multiple outcomes, based on some criterion. The criterion can be either a scalar (a number) or a string. In practice, it is used more with strings than with numbers. The structure of `switch/case` is

KEY IDEA
The switch/case structure is similar to the if/elseif/else structure.

```
switch  variable
case  option1
  code to be executed if variable is equal to option 1
case option2
  code to be executed if variable is equal to option 2
case option_n
  code to be executed if variable is equal to option n
otherwise
  code to be executed if variable is not equal to any of
     the options
end
```

Here's an example: Suppose you want to create a function that tells the user what the airfare is to one of three different cities:

```
city = input("Enter the name of a city in single quotes:")
switch city
  case "Boston"
    disp("$345")
  case "Denver"
    disp("$150")
  case "Honolulu"
    disp("Stay home and study")
  otherwise
    disp("Not on file")
end
```

If, when you run this script, you reply `"Boston"` at the prompt, MATLAB responds

```
city =
   "Boston"
$345
```

You can tell the `input` command to expect a string by adding "s" in a second field. This relieves the user of the awkward requirement of adding double quotes around any string input. With the added "s," the preceding code now reads:

```
city = input("Enter the name of a city:","s")
switch city
 case 'Boston'
   disp('$345')
 case 'Denver'
   disp('$150')
 case 'Honolulu'
   disp('Stay home and study')
 otherwise
  disp('Not on file')
end
```

The `otherwise` portion of the `switch/case` structure is not required for the structure to work. However, you should include it if there is any way that the user could input a value not equal to one of the cases.

Switch/case structures are flowcharted exactly the same as `if/else` structures.

HINT

If you are a C programmer, you may have used `switch/case` in that language. One important difference in MATLAB is that once a "true" case has been found, the program does not check the other cases.

EXAMPLE 8.4

BUYING GASOLINE



Figure 8.13
Gasoline is sold in both liters and gallons.

Three countries in the world do not officially use the metric system: the United States, Liberia, and Myanmar. Even in the United States, the practice is that some industries are almost completely metric, and others still use the English system of units. For example, any shade-tree mechanic will tell you that although older cars have a mixture of components—some metric and others English—all but the oldest cars (any car built after 1989) are almost completely metric. Wine is packaged in liters, but milk is packaged in gallons. Americans measure distance in miles, but power in watts. Confusion between metric and English units is common. American travelers to Canada are regularly confused because gasoline is sold by the liter in Canada, but by the gallon in the United States.

Imagine that you want to buy gasoline (see Figure 8.13). Write a program that:

- Asks the user whether they want to request the gasoline in liters or in gallons.
- Prompts the user to enter how many units they want to buy.
- Calculates the total cost to the user, assuming that gasoline costs $2.89 per gallon.

Use a `switch/case` structure.

1. State the problem
   Calculate the cost of a gasoline purchase.

2. Describe the input and output

Input Specify gallons or liters
Number of gallons or liters

Output Cost in dollars, assuming $2.89 per gallon

3. Develop a hand example
If the volume is specified in gallons, the cost is

$$\text{volume} \times \$2.89$$

So, for 10 gallons,

$$\text{cost} = 10 \text{ gallons} \times \$2.89/\text{gallon} = \$28.90$$

If the volume is specified in liters, we need to convert liters to gallons and then calculate the cost:

$$\text{volume} = \text{liters} \times 0.264 \text{ gallon/liter}$$
$$\text{cost} = \text{volume} \times \$2.89$$

So, for 10 liters,

$$\text{volume} = 10 \text{ liters} \times 0.264 \text{ gallon/liter} = 2.64 \text{ gallons}$$
$$\text{cost} = 2.64 \text{ gallons} \times 2.89 = \$7.63$$

4. Develop a MATLAB solution
First create a flowchart (see Figure 8.14). Then convert the flowchart into pseudocode comments. Finally, add the MATLAB code:

```
%% Example 8.4
clear, clc
% Define the cost per gallon
 rate = 2.89;
% Ask the user to input gallons or liters
 unit = input("Enter gallons or liters\n ","s");
% Use a switch/case to determine the conversion factor
 switch unit
  case "gallons"
     factor = 1;
  case "liters"
    factor = 0.264;
  otherwise
    error("Not available")
end

% Ask the user how much gas they would like to buy
query_str ="Enter the volume you would like to buy in ";
volume = input([query_str + unit + ": \n"] );
% Calculate the cost of the gas
cost = volume * factor*rate;
% Report the results
result_str = "That will be $ %5.2f for %5.1f %s \n";
fprintf(result_str,cost,volume,unit)
```

(*continued*)

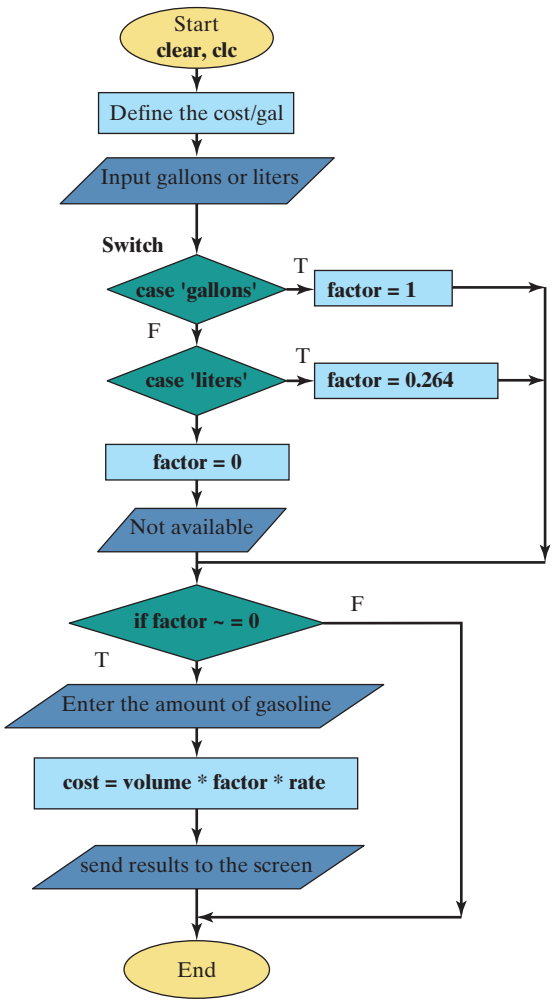


Figure 8.14
Flowchart to determine the cost of gasoline, using the `switch/case` structure.

There are several things to notice about this solution. First, the variable `unit` contains an array of character information even though it was entered as a string array. As of release 2021 the input function is still mixing the string and character array nomenclature in both the documentation and in the function behavior. There are good reasons for this, since existing code would stop working if the *input* function suddenly started returning a different data type. If you check the workspace window after you run this program, you'll notice that `unit` is either a 1×6 character array (if you entered liters) or a 1×7 character array (if you entered gallons).
On the line

```
unit = input('Enter gallons or liters ','s');
```

the second field, `'s'`, tells MATLAB to expect a character array as input. This allows the user to enter gallons or liters without the surrounding quotes. Using either single quotes, which indicate a character array, or double quotes which indicate a string, give the same result. The variable `unit` always is stored as a character array.

On the line

```
volume = input([query_str + unit + ": \n"]);
```

we created a string array out of three components using the concatentation operator, +.

- The string array `query_str`
- The character array `unit`
- The string array `': \n'`

By combining these three components, we were able to make the program prompt the user with either

```
Enter the volume you would like to buy in liters:
```

or

```
Enter the volume you would like to buy in gallons:
```

In the `fprintf` statement, we included a field for string input by using the placeholder `%s`:

```
result_str = "That will be $ %5.2f for %5.1f %s \n";
fprintf(result_str,cost,volume,unit)
```

In this case, by string MATLAB means either a character array or an actual string.

This allowed the program to tell the users that the gasoline was measured either in gallons or in liters.

Finally, an `if` statement with the `error` function was used, so that if the user entered something besides gallons or liters, no calculations were performed and the program terminated.

5. Test the solution

The solution was tested by running the program three separate times, once for gallons, once for liters, and once for some unit not supported. The interaction for gallons is

```
Enter gallons or liters
gallons
Enter the volume you would like to buy in gallons:
10
That will be $ 28.90 for 10.0 gallons
```

For liters, the interaction is

```
Enter gallons or liters
liters
Enter the volume you would like to buy in liters:
10
That will be $ 7.63 for 10.0 liters
```

Finally, if the user enters anything besides gallons or liters, the program responds with an error message:

```
Enter gallons or liters
quarts
Not available
```

Since the program results are the same as the hand calculation, it appears that the program works as planned.

HINT

The nomenclature surrounding the use of text in MATLAB is confusing because the original code only included one data type, the character array. It was regularly referred to as a container for strings. However, when the actual string data type was introduced in 2016 this meant the meaning of the word "string" had changed. This is undoubtedly an area in the MATLAB programming language that will continue to evolve in future revisions.

KEY IDEA
Graphical user interfaces like the menu box reduce the opportunity for user errors, such as spelling mistakes.

8.5.5 Menu

The `menu` function can be used in conjunction with a `switch/case` structure. This function causes a menu box to appear on the screen, with a series of buttons defined by the programmer. It can be used with either character array or string array input.

```
input = menu('Message to the user','text for button
   1',' text for button 2', etc.)
```

We can use the `menu` option in our previous airfare example to ensure that the user chooses only allowable cities. This also means that the `otherwise` syntax is no longer needed, since it is not possible to choose a city "not on file."

```
prompt = "Select a city from the menu:";
list =["Boston","Denver","Honolulu"];
city = menu(prompt,list)
switch city
    case 1
       disp("$345")
    case 2
       disp("$150")
    case 3
       disp("Stay home and study")
end
```

Notice that a case number has replaced the string on each `case` line. When the script is executed, the menu box shown in Figure 8.15 appears and waits for the user to select one of the buttons. If you choose Honolulu, MATLAB will respond

```
city =
     3
Stay home and study.
```

It is possible to suppress the output from the `menu` command, which was included here for clarity.

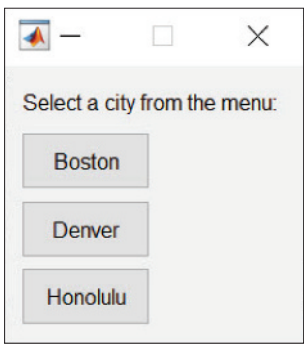


Figure 8.15
The pop-up menu window.

HINT

The MathWorks recommends you use the `listdlg` function instead of the `menu` function. The results look the same, but the function offers more options. However, the input is significantly more complicated. You can explore the `listdlg` function by searching the MATLAB documentation

```
doc listdlg
```

Both functions were introduced before 2006a. A more recent approach to creating graphical user interfaces, of which a menu is a very simple example, is to use the app creator.

EXAMPLE 8.5

BUYING GASOLINE: A MENU APPROACH

In Example 8.4, a `switch/case` approach was used to determine whether the customer wanted to buy gasoline measured in gallons or liters. One problem with our program is that if the user can't spell, the program won't work. For example, if, when prompted for gallons or liters, the user enters

```
litters
```

The program will respond

```
Not available
```

Using a menu gets around this problem; then the user need only press a button to make a choice. We'll still use the `switch/case` structure, but will combine it with the menu.

1. State the problem
   Calculate the cost of a gasoline purchase.
2. Describe the input and output

   Input Specify gallons or liters, using a menu
   Number of gallons or liters

   Output Cost in dollars, assuming $2.89 per gallon
3. Develop a hand example
   If the volume is specified in gallons, the cost is

   $$\text{volume} \times \$2.89$$

   So, for 10 gallons,

   $$\text{cost} = 10 \text{ gallons} \times \$2.89/\text{gallon} = \$28.90$$

   If the volume is specified in liters, we need to convert liters to gallons, then calculate the cost:

   $$\text{volume} = \text{liters} \times 0.264 \text{ gallon/liter}$$
   $$\text{cost} = \text{volume} \times \$2.89$$

   So, for 10 liters,

   $$\text{volume} = 10 \text{ liters} \times 0.264 \text{ gallon/liter} = 2.64 \text{ gallons}$$
   $$\text{cost} = 2.64 \text{ gallons} \times 2.89 = \$7.63$$

(*continued*)

4. Develop a MATLAB solution
First, create a flowchart (see Figure 8.16). Then, convert the flowchart into pseudocode comments. Finally, add the MATLAB code:

```
%% Example 8.5
clear, clc
% Define the cost per gallon
rate = 2.89;
% Ask the user to input gallons or liters, using a menu
disp("Use the menu box to make your selection")
prompt ="Measure the gasoline in liters or gallons?";
list =["gallons","liters"];
choice = menu(prompt,list);
%% Use a switch/case to determine the conversion factor
switch choice
    case 1
        factor = 1;
        unit = 'gallons'
    case 2
        factor = 0.264;
        unit = 'liters'
end
query_str = "Enter the volume you would like to buy in";
volume = input([query_str + unit + ":\n"] );
% Calculate the cost of the gas
cost = volume * factor*rate;
% Report the results
result_str = "That will be $ %5.2f for %5.1f %s \n";
fprintf(result_str,cost,volume,unit)
```

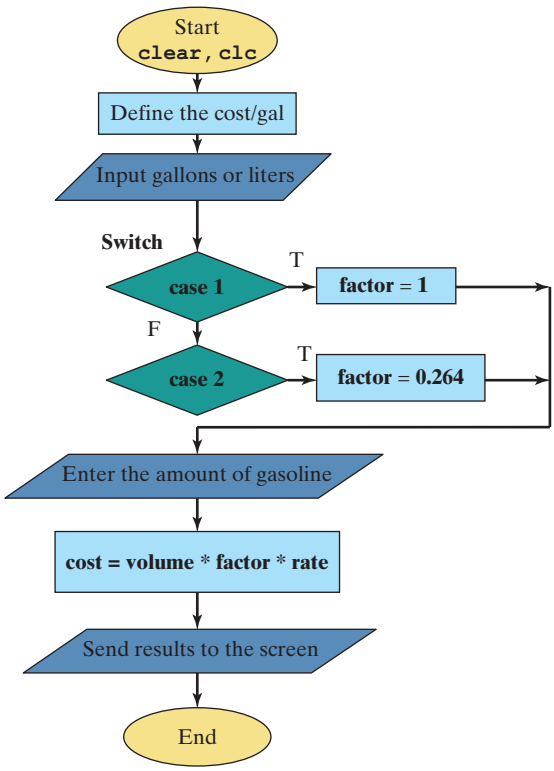


Figure 8.16
Flowchart to determine the cost of gasoline, using a menu.

This solution is simpler than the one in Example 8.4 because there is no chance for bad input. There are a few things to notice, however.

When we define `choice` by using the `menu` function, the result is a number, not a character array. You can check this by consulting the workspace window, in which `choice` is listed as a 1×1 double-precision number.

Because we did not use the `input` command to define the variable `unit`, which is a character array, we needed to specify the value of `unit` as part of the case calculations:

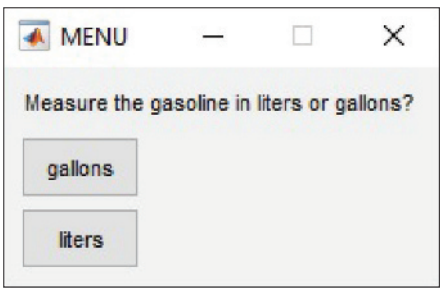
```
case 1
  factor = 1;
  unit = 'gallons'
case 2
  factor = 0.264;
  unit = 'liters'
```

Doing this allows us to use the value of `unit` in the output to the command window, both in the `disp` command and in `fprintf`.

5. Test the solution

As in Example 8.4, we can test the solution by running the program, but this time we need to try it only twice—once for gallons and once for liters. The interaction in the command window for gallons is

```
Use the menu box to make your selection
```




```
Enter the volume you would like to buy in gallons:
10
That will be $ 28.90 for 10.0 gallons
```

For liters, the interaction is

```
Use the menu box to make your selection
Enter the volume you would like to buy in liters:
10
That will be $ 7.63 for 10.0 liters
```

These values match those in the hand solution, and have the added advantage that you can't misspell any of the input.

PRACTICE EXERCISE 8.4

Use the `switch/case` structure to solve these problems. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Create a program that prompts the user to enter his or her year in school—freshman, sophomore, junior, or senior. The input will be a string. Use the `switch/case` structure to determine which day finals will be given for each group—Monday for freshmen, Tuesday for sophomores, Wednesday for juniors, and Thursday for seniors.
2. Repeat Exercise 1, but this time with a menu.
3. Create a program to prompt the user to enter the number of candy bars they would like to buy. The input will be a number. Use the `switch/case` structure to determine the bill, where

$$\begin{aligned} 1 \text{ bar} &= \$0.75 \\ 2 \text{ bars} &= \$1.25 \\ 3 \text{ bars} &= \$1.65 \end{aligned}$$

$$\text{more than 3 bars} = \$1.65 + \$0.30 \times (\text{number ordered} - 3).$$

8.6 DEBUGGING

As the code you are writing becomes more complicated, the debugging tools available in MATLAB become more valuable. Consider the simple program shown in Figure 8.17 that demonstrates the use of the `if/else` structure. A breakpoint has been added on line three. When the code is executed by selecting the save and run icon, it will first pause on line two waiting for the user to enter a number. Once the

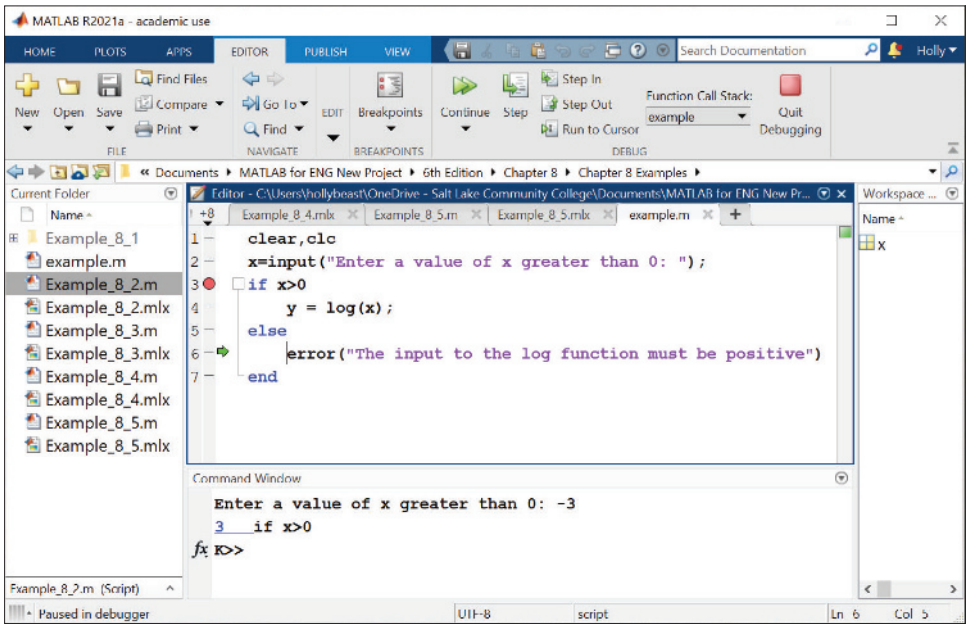


Figure 8.17
Using debugging tools is an effective way to evaluate how MATLAB moves through the code as it executes.

number has been entered, the program moves to line three and stops because the breakpoint has been encountered. Selecting the *Step* icon will progress the execution through the code one line at a time, allowing the programmer to observe the effect of each line of code. The current line is indicated with the green arrow.

Also notice that the **folding** capability available in MATLAB has been activated for `if/else` structures. This was accomplished by selecting the Home tab then

Preferences -> Editor/Debugger -> Code Folding

and activating code folding for `if/else` structures. A visual cue is created, making it easier to keep track of which lines of code are included in the structure.

SUMMARY

Sections of computer code can be categorized as sequences, selection structures, and repetition structures. Sequences are lists of instructions that are executed in order. Selection structures allow the programmer to define criteria (conditional statements) that the program uses to choose execution paths. Repetition structures define loops in which a sequence of instructions is repeated until some criterion is met (also defined by conditional statements).

MATLAB uses the standard mathematical relational operators, such as greater than (>) and less than (<). The not-equal-to (`~=`) operator's form is not usually seen in mathematics texts. MATLAB also includes logical operators such as *and* (`&`) and *or* (`|`). These operators are used in conditional statements, allowing MATLAB to make decisions regarding which portions of the code to execute.

The `find` function is unique to MATLAB. It allows the user to specify a condition by using both logical and relational operators. The function is then used to identify elements of an array that meet the condition. In addition to the `find` function MATLAB includes a variety of logical functions such as `all` and `any` to determine if a criterion is met by members of an array.

Logical indexing is another approach to using criteria to either extract data from arrays or to modify data in existing arrays. Logical indexing is usually more computationally efficient than using a logical function to accomplish the same task.

Although the `if`, `else`, and `elseif` commands can be used for both scalars and array variables, they are useful primarily for scalars. These commands allow the programmer to identify alternative computing paths on the basis of the results of conditional statements.

The switch/case structure is an alternative to if/else structures. It is particularly useful when there are multiple logical options and for programming tasks involving text. When paired with graphical user interfaces such as the `menu` or `listdlg` functions it can be used to create programs that reduce the opportunity for user error.

MATLAB SUMMARY

The following MATLAB summary lists and briefly describes the special characters, commands, and functions that were defined in this chapter:

Special Characters	
<	less than
<=	less than or equal to
>	greater than

Special Characters	
>=	greater than or equal to
==	equal to
~=	not equal to
&	and
|	or
~	not
&&	short circuit and (used with scalars)
||	short circuit or (used with scalars)

Commands and Functions	
`all`	checks to see if a criterion is met by all the elements in an array
`any`	checks to see if a criterion is met by any of the elements in an array
`case`	selection structure
`else`	defines the path if the result of an if statement is false
`elseif`	defines the path if the result of an if statement is false and specifies a new logical test
`end`	identifies the end of a control structure
`error`	sends a message to the command window and aborts the program
`find`	determines which elements in an array meet the input criterion
`if`	checks a condition, resulting in either true or false
`listdlg`	creates a menu to use as an input vehicle
`menu`	creates a menu to use as an input vehicle
`otherwise`	part of the case selection structure
`switch`	part of the case selection structure

KEY TERMS

code folding
control structure
flowchart
index
logical array
logical condition
logical indexing
logical operator
loop
pseudocode
relational operator
repetition
selection
sequence
subscript

PROBLEMS

Logical Functions: `find`

8.1 A sensor that monitors the temperature of a backyard hot-tub records the data shown in Table P8.1. The temperature data is stored in a data file called *hot_tub.dat*, which is available on the Pearson website or which may be available to you from your instructor. Load the data into MATLAB and create a companion time vector called `time`.

Table P8.1 Hot-Tub Temperature Data

Time of Day	Temperature (°F)
00:00	100
01:00	101
02:00	102
03:00	103
04:00	103
05:00	104
06:00	104
07:00	105
08:00	106
09:00	106
10:00	106
11:00	105
12:00	104
13:00	103
14:00	101
15:00	100
16:00	99
17:00	100
18:00	102
19:00	104
20:00	106
21:00	107
22:00	105
23:00	104
24:00	104

(a) The temperature should never exceed 105°F. Use the `find` function to find the index numbers of the temperatures stored in the `hot_tub` vector that exceed the maximum allowable temperature.

(b) Use the `length` function with the results from part (a) to determine how many times the maximum allowable temperature was exceeded.

(c) Determine at what times the temperature exceeded the maximum allowable temperature, using the index numbers found in part (a).

(d) The temperature should never be lower than 102°F. Use the `find` function together with the `length` function to determine how many times the temperature was less than the minimum allowable temperature.

(e) Determine at what times the temperature was less than the minimum allowable temperature.

(f) Determine at what times the temperature was within the allowable limits (i.e., between 102°F and 105°F, inclusive).

(g) Use the `max` function to determine the maximum temperature reached and the time at which it occurred.

8.2 The height of a rocket (in meters) can be represented by the following equation:

$$\text{height} = 2.13t^2 - 0.0013t^4 + 0.000034t^{4.751}$$

Create a vector of time (t) values from 0 to 100 at 2-second intervals.

(a) Use the `find` function to determine when the rocket hits the ground to within 2 seconds. (*Hint*: The value of `height` will be positive for all values until the rocket hits the ground.)
(b) Use the `max` function to determine the maximum height of the rocket and the corresponding time.
(c) Create a plot with `t` on the horizontal axis and `height` on the vertical axis for times until the rocket hits the ground. Be sure to add a title and axis labels.*

8.3 Reconsider the data from Table P8.1 that represents the temperature of a backyard hot-tub. The temperature data is stored in a file called *hot_tub.dat* and can be loaded into MATLAB. The time information must be created by the statement:

```
time = 0:24;
```

(a) The temperature in the hot-tub should never exceed 105°F. Create a logical index array that represents the elements in the `hot_tub` array where the temperature exceeds 105°F.
(b) Use the logical index array from part (a) to extract the temperature values that exceed 105°F.
(c) Instead of extracting the temperatures in a two-step process, extract the temperatures using a single step process.
(d) Use the `length` function with the results from part (c) to determine how many times the maximum allowable temperature was exceeded.
(e) Use the logical index array from part (a) to determine at what times the temperature was exceeded.
(f) Using the logical index array from part (a) makes determining the times a two-step process. Repeat part (e) in a single step process.
(g) The temperature should never be lower than 102°F. Determine at what times the temperature was within the allowable limits (i.e., between 102°F and 105°F, inclusive)
(h) Create a table, using the `table` function, of the times when the temperature was outside of the allowable limits and the corresponding temperatures.

8.4 As we saw in Problem 8.2, the height of a rocket in meters can be represented by the following equation:

$$\text{height} = 2.13t^2 - 0.0013t^4 + 0.000034t^{4.751}$$

Create a vector of time values, `t`, from 0 to 100 at 2-second intervals and the corresponding `height` vector to use in the following problems.

(a) Create a logical index array based on whether the value stored in the `height` vector is positive or negative.
(b) Use the logical index array from part (a) to extract the positive values from the `height` vector into a new vector called `h`.
(c) Use the logical index array from part (a) to extract the time vector corresponding to positive height values into a new vector called `t2`.
(d) Repeat parts (b) and (c) in a single step process.

---

*From Etter, Kuncicky, and Moore, *Introduction to Matlab* 7 (Upper Saddle River, NJ: Pearson/Prentice Hall, 2005).

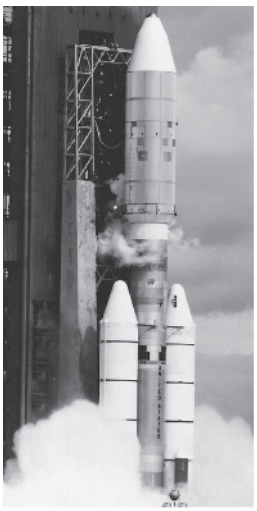


Figure P8.5
Solid-fuel rocket booster to a Titan missile. (NASA Headquarters)

(e) Determine when the rocket hits the ground within 2 seconds. (*Hint*: The value of `height` will be positive until the rocket hits the ground.)
(f) Create a plot with time on the horizontal axis and height on the vertical axis for times until the rocket hits the ground. Be sure to add a title and axis labels.
(g) Suggest and implement a strategy to estimate the time the rocket hits the ground with greater accuracy.

8.5 Solid-fuel rocket engines are used as boosters, in satellite launch vehicles, and in weapons systems (see Figure P8.5). The propellant is a solid combination of fuel and oxidizer, about the consistency of an eraser. For the space shuttle, the fuel component is aluminum and the oxidizer is ammonium perchlorate, held together with an epoxy resin "glue." The propellant mixture is poured into a motor case, and the resin is allowed to cure under controlled conditions. Because the engines are extremely large, they are cast in segments, each requiring several "batches" of propellant to fill. (Each engine contains over 1.1 million pounds of propellant!) This casting–curing process is sensitive to temperature, humidity, and pressure. If the conditions aren't just right, the fuel could ignite or the properties of the propellant grain (which means its shape; the term *grain* is borrowed from artillery) might be degraded. Solid-fuel rocket engines are extremely expensive as well as dangerous and clearly must work right every time, or the results will be disastrous. Failures can cause loss of human life and irreplaceable scientific data and equipment. Highly public failures can destroy a company. Actual processes are tightly monitored and controlled. However, for our purposes, consider these general criteria:

The temperature should remain between 115°F and 125°F.
The humidity should remain between 40% and 60%.
The pressure should remain between 100 and 200 Torr.

Imagine that the data in Table P8.5 were collected during a casting–curing process.

Table P8.5 Casting–Curing Data

Batch Number	Temperature (°F)	Humidity (%)	Pressure (Torr)
1530	116	45	110
1240	114	42	115
2380	118	41	120
1470	124	38	95
3590	126	61	118

Source: Pearson Education

(a) Use logical indexing to determine which batches did and did not meet the criterion for temperature.
(b) Use logical indexing to determine which batches did and did not meet the criterion for humidity.
(c) Use logical indexing to determine which batches did and did not meet the criterion for pressure.
(d) Use logical indexing to determine which batches failed for any reason and which passed.

(e) Use your results from the previous questions, along with the `length` function, to determine what percentage of engines passed or failed on the basis of each criterion and to determine the total passing rate.

8.6 Two gymnasts are competing with each other. Their scores are shown in Table P8.6.

Table P8.6 Gymnastics Scores

Event	Gymnast 1	Gymnast 2
Pommel horse	9.821	9.700
Vault	9.923	9.925
Floor	9.624	9.83
Rings	9.432	9.987
High bar	9.534	9.354
Parallel bars	9.203	9.879

(a) Write a program that uses logical indexing to determine how many events each gymnast won.

(b) Use the `mean` function to determine each gymnast's average score.

8.7 Use logical indexing to create a function called `f` that satisfies the following criteria:

$$\text{For values of } x > 2,\ f(x) = x^2$$

$$\text{For values of } x \leq 2,\ f(x) = 2x$$

Plot your results for values of x from -3 to 5. Choose your spacing to create a smooth curve. You should notice a break in the curve at $x = 2$. What happens if you try to do this problem using an `if/else` structure?

8.8 Create a function called `g` that satisfies the following criteria:

$$\begin{aligned} &\text{For } x < -\pi, && g(x) = -1 \\ &\text{For } x \geq -\pi \text{ and } x \leq \pi, && g(x) = \cos(x) \\ &\text{For } x > \pi, && g(x) = -1 \end{aligned}$$

Plot your results for values of x from -2π to $+2\pi$. Choose your spacing to create a smooth curve. (You may be tempted to use an `if/else` structure to solve this problem. Don't do it — use logical indexing instead.)

8.9 A file named *temp.dat* contains information collected from a set of thermocouples. The data in the file are shown in Table P8.9. The first column consists of time measurements (one for each hour of the day), and the remaining columns correspond to temperature measurements at different points in a process.

Table P8.9 Temperature Data

Hour	Temp1	Temp2	Temp3
1	68.70	58.11	87.81
2	65.00	58.52	85.69
3	70.38	52.62	71.78
4	70.86	58.83	77.34
5	66.56	60.59	68.12
6	73.57	61.57	57.98
7	73.57	67.22	89.86
8	69.89	58.25	74.81
9	70.98	63.12	83.27
10	70.52	64.00	82.34

(*continued*)

Table P8.9 (continued)

Hour	Temp1	Temp2	Temp3
11	69.44	64.70	80.21
12	72.18	55.04	69.96
13	68.24	61.06	70.53
14	76.55	61.19	76.26
15	69.59	54.96	68.14
16	70.34	56.29	69.44
17	73.20	65.41	94.72
18	70.18	59.34	80.56
19	69.71	61.95	67.83
20	67.50	60.44	79.59
21	70.88	56.82	68.72
22	65.99	57.20	66.51
23	72.14	62.22	77.39
24	74.87	55.25	89.53

(a) Write a program that prints the index numbers (rows and columns) of temperature data values greater than 85.0. (Decide for yourself whether this problem is best solved using logical indexing or the logical `find` function.)

(b) Find the maximum temperature in the file, the corresponding hour value, and thermocouple number.

8.10 The amount of sea ice surrounding the Antarctic continent varies dramatically during the year, and from year to year (see Figure P8.10).

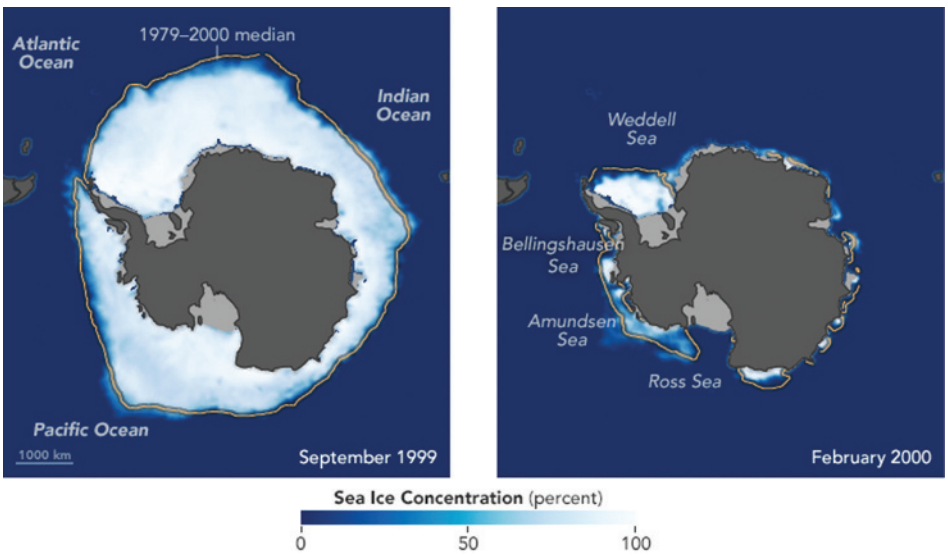


Figure P8.10
Antarctic sea ice extent changes dramatically through the year. (National Snow and Ice Data Center)

The following data, shown in Table P8.10, were obtained from the National Snow and Ice Data Center (NSIDC) website and represent the extent of the sea ice during September when the maximum usually occurs, and February when the minimum usually occurs.

(a) Plot both the September values and February values versus the year, and add a line representing the average ice extent for 1979–2010 of

Table P8.10 Antarctic Sea Ice Extent (Data from National Snow and Ice Data Center, https://nsidc.org/data/seaice_index/archives)

Year	September Average Extent (millions of square kilometers)	February Average Extent (millions of square kilometers)
1979	18.2	3.14
1980	18.82	2.82
1981	18.6	2.87
1982	18.35	3.13
1983	18.59	3.07
1984	18.17	2.68
1985	18.7	2.84
1986	17.68	3.19
1987	18.25	3.25
1988	18.31	2.89
1989	18.12	2.98
1990	18.19	3.04
1991	18.31	3.16
1992	18.15	2.84
1993	18.54	2.48
1994	18.61	3.2
1995	18.37	3.56
1996	18.63	2.98
1997	18.6	2.49
1998	18.9	2.99
1999	18.7	2.97
2000	18.89	2.91
2001	18.17	3.65
2002	17.89	2.96
2003	18.3	3.79
2004	18.92	3.58
2005	18.8	2.97
2006	19.09	2.65
2007	18.86	2.9
2008	18.15	3.89
2009	18.96	2.99
2010	18.8	3.11
2011	18.74	2.52
2012	19.21	3.55
2013	19.39	3.84
2014	19.76	3.84
2015	18.44	3.8
2016	18.15	2.79
2017	17.91	2.29
2018	17.96	2.33
2019	18.33	2.65
2020	18.84	2.92
2021		2.83

3.04 million square kilometers for February and 18.79 million square kilometers for September.

(b) Use logical indexing to determine which years exceeded the average September values for 1979–2010.

(c) Use logical indexing to determine which years were lower than the average February values for 1979–2010.

The average from 1979 to 2010 was selected as a metric to which to compare. Should it be changed every year?

8.11 Sea ice in the Arctic Ocean increases and decreases annually. Declines have even made it possible for shipping to occur through the "Northwest Passage" in recent summer seasons (see Figure P8.11).

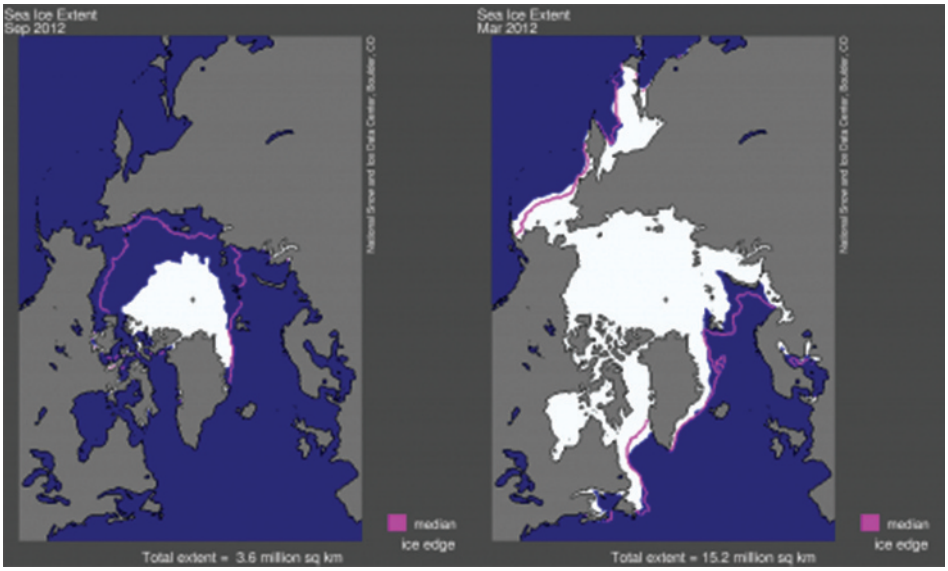


Figure P8.11
The extent of arctic sea ice varies dramatically during the year. Recent decreases in the summer extent have opened up the "Northwest Passage" from the Atlantic to the Pacific oceans. (NOAA)

The data shown in Table P8.11 were obtained from the National Snow and Ice Data Center (NSIDS) website and represent the extent of the sea ice during September when the minimum usually occurs, and March when the maximum usually occurs.

Table P8.11 Arctic Sea Ice Extent (Data from National Snow & Ice data center, https://nsidc.org/data/seaice_index/archives)

Year	September Average Extent (millions of square kilometers)	March Average Extent (millions of square kilometers)
1979	7.05	16.34
1980	7.67	16.04
1981	7.14	15.63
1982	7.3	16.04
1983	7.39	16.09
1984	6.81	15.58
1985	6.7	15.89
1986	7.41	15.91
1987	7.28	15.82
1988	7.37	15.96

(*continued*)

Table P8.11 (continued)

Year	September Average Extent (millions of square kilometers)	March Average Extent (millions of square kilometers)
1989	7.01	15.42
1990	6.14	15.87
1991	6.47	15.42
1992	7.47	15.48
1993	6.4	15.81
1994	7.14	15.55
1995	6.08	15.26
1996	7.58	15.12
1997	6.69	15.47
1998	6.54	15.6
1999	6.12	15.4
2000	6.25	15.22
2001	6.73	15.52
2002	5.83	15.35
2003	6.12	15.48
2004	5.98	14.99
2005	5.5	14.69
2006	5.86	14.42
2007	4.27	14.54
2008	4.69	15.18
2009	5.26	14.98
2010	4.87	15.14
2011	4.56	14.55
2012	3.57	15.2
2013	5.21	15.03
2014	5.22	14.76
2015	4.62	14.37
2016	4.53	14.4
2017	4.82	14.29
2018	4.79	14.3
2019	4.36	14.57
2020	4	14.73
2021		14.64

(a) Plot both the September values and March values on the *y*-axis, versus the year on the *x*-axis, and add a line representing the average ice extent for 1979–2010 of 15.52 million square kilometers for March and 6.51 million square kilometers for September.

(b) Use logical indexing to determine which years exceeded the average September values for 1979–2010.

(c) Use logical indexing to determine which years were lower than the average March values for 1979–2010.

8.12 The Colorado River Drainage Basin covers parts of seven western American states. A series of dams has been constructed on the Colorado River and its tributaries to store runoff water and to generate low-cost hydroelectric

power (see Figure P8.12). The ability to regulate the flow of water has made the growth of agriculture and population in these arid desert regions possible. Even during periods of extended drought, a steady, reliable source of water and electricity has been available. Lake Powell located on the border of Utah and Arizona is one of these reservoirs. The file *lake_powell.dat* contains data on the water level in the reservoir for the 8 years from 2013 to 2020. Additional historical data are available from the US Bureau of Reclamation online. These data are shown in Table P8.12. Use the information in the file to answer the following questions:

(a) Determine the average elevation of the water level for each year, and for the period over which the data were collected.
(b) Determine how many months each year exceed the overall average for the period.
(c) Create a report that lists the month and the year for each of the months that exceed the overall average.
(d) Determine the average elevation of the water for each month for the period.

Note: This problem should be solved using either logical indexing or the `find` function, the `mean` function, and the `length` function. Programmers with previous experience may be tempted to use a loop structure, which is not required.



Figure P8.12
Glen Canyon Dam at Lake Powell. (David Toussaint/ Moment Open/Getty Images)

Table P8.12 Water-Level Data for Lake Powell, Measured in Feet above Sea Level (Data from Bureau of reclamation, https://www.usbr.gov/rsvrWater/HistoricalApp.html on June 29, 2021)

	2013	2014	2015	2016	2017	2018	2019	2020
January	3604.42	3578.69	3593.57	3596.58	3595.86	3619.14	3576.34	3605.48
February	3601.47	3575.55	3592.23	3594.41	3594.33	3616.02	3571.89	3602.72
March	3598.96	3574.76	3591.02	3592.18	3595.91	3612.23	3569.28	3600.71
April	3596.53	3577.56	3590.18	3592.12	3604.14	3609.39	3571.12	3599.32
May	3599.44	3589.38	3597.27	3603.87	3619.09	3611.54	3584.65	3605.05
June	3600.07	3609.19	3613.54	3620.01	3634.89	3609.98	3611.82	3610.62
July	3594.17	3608.05	3612.62	3618.22	3634.69	3603.8	3621.6	3606.25
August	3589.64	3605.82	3609.07	3613.55	3630.88	3597.12	3618.55	3599.72
September	3591.25	3605.53	3606.01	3610.93	3628.31	3592.28	3615.36	3595.98
October	3590.88	3605.57	3606.44	3609.48	3627.09	3590.46	3612.99	3591.72
November	3587.9	3601.87	3605.47	3605.81	3625.29	3586.5	3611.23	3587.72
December	3584.43	3597.75	3600.8	3600.49	3622.85	3581.85	3608.74	3582.21

IF STRUCTURES

8.13 Create a program that prompts the user to enter a scalar value of temperature. If the temperature is greater than 98.6°F, send a message to the command window telling the user that they have a fever.

8.14 Create a program that first prompts the user to enter a value for `x` and then prompts the user to enter a value for `y`. If the value of `x` is greater than the value of `y`, send a message to the command window telling the user that $x > y$. If `x` is less than or equal to `y`, send a message to the command window telling the user that $y >= x$.

8.15 The inverse sine (`asin`) and inverse cosine (`acos`) functions are valid only for inputs between -1 and $+1$, because both the sine and the cosine have values only between -1 and $+1$ (see Figure P8.15). MATLAB interprets the result of `asin` or `acos` for a value outside this range as a complex number. For example, we might have

```
acos(-2)
ans =
  3.1416 - 1.3170i
```

which is a questionable mathematical result. Create a function called `my_asin` that accepts a single value of `x` and checks to see if it is between -1 and $+1$ ($-1 <= x <= 1$). If `x` is outside the range, send an error message to the screen. If it is inside the allowable range, return the value of `asin`.

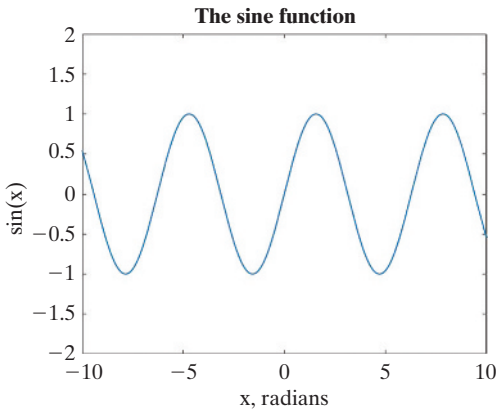


Figure P8.15
The sine function varies between -1 and $+1$. Thus, the inverse sine (`asin`) is not defined for values greater than 1 and values less than -1.

8.16 Create a program that prompts the user to enter a scalar value for the outside air temperature. If the temperature is equal to or above 80°F, return a message telling the user to wear shorts. If the temperature is between 60°F and 80°F, return a message telling the user that it is a beautiful day. If the temperature is equal to or below 60°F, return a message telling the user to wear a jacket or coat.

8.17 Suppose the following array represents the number of saws ordered from your company each month over the last year:

```
saws = [1,4,5,3,7,5,3,10,12,8,7,4]
```

All the numbers should be zero or positive.

(a) Use an `if` statement to check whether any of the values in the array are invalid. (Evaluate the whole array at once in a single `if` statement.) Send the message "All valid" or else "Invalid number found" to the screen, depending on the results of your analysis.

(b) Change the `saws` array to include at least one negative number, and check your program to make sure that it works for both cases.

8.18 Most large companies encourage employees to save by matching their contributions to a 401(k) plan. The government limits how much you can save in these plans, because they shelter income from taxes until the money is withdrawn during your retirement. The amount you can save is tied to your income, as is the amount your employer can contribute. The government will allow you to save additional amounts without the tax benefit. These plans change from year to year, so this example is just a made-up "what if."

Suppose the Quality Widget Company has the savings plan described in Table P8.18. Create a function that finds the total yearly contribution to your savings plan, based on your salary and the percentage you contribute. Remember, the total contribution consists of the employee contribution and the company contribution.

Table P8.18 Quality Widget Company Savings Plan

Income	Maximum You Can Save Tax Free	Maximum the Company Will Match
Up to $30,000	10%	10%
Between $30,000 and $60,000	10%	10% of the first $30,000 and 5% of the amount above $30,000
Between $60,000 and $100,000	10% of the first $60,000 and 8% of the amount above $60,000	10% of the first $30,000 and 5% of the amount between $30,000 and $60,000; nothing for the remainder above $60,000
Above $100,000	10% of the first $60,000 and 8% of the amount between $60,000 and $100,000; nothing on the amount above $100,000	Nothing—highly compensated employees are exempt from this plan and participate in stock options instead

SWITCH/CASE

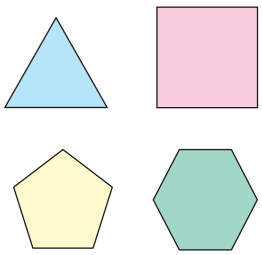


Figure P8.19
Regular polygons.

8.19 In order to have a closed geometric figure composed of straight lines (see Figure P8.19), the angles in the figure must add to

$$(n - 2)(180 \text{ degrees})$$

where n is the number of sides.

(a) Prove this statement to yourself by creating a vector called `n` from 3 to 6 and calculating the angle sum from the formula. Compare what you know about geometry with your answer.

(b) Write a program that prompts the user to enter one of the following:

triangle
square
pentagon
hexagon

Use the input to define the value of `n` via a `switch/case` structure; then use `n` to calculate the sum of the interior angles in the figure.

(c) Reformulate your program from part (b) so that it uses a menu.

8.20 At a local university, each engineering major requires a different number of credits for graduation. For example, recently the requirements were as follows:

Civil Engineering	130
Chemical Engineering	130
Computer Engineering	122
Electrical Engineering	126.5
Mechanical Engineering	129

Prompt the user to select an engineering program from a menu. Use a `switch/case` structure to send the minimum number of credits required for graduation back to the command window.

8.21 The easiest way to draw a star in MATLAB is to use polar coordinates. You simply need to identify points on the circumference of a circle, and draw lines between those points. For example, to draw a five-pointed star, start at the top of the circle ($\theta = \pi/2,\ r = 1$) and work counterclockwise (see Figure P8.21).

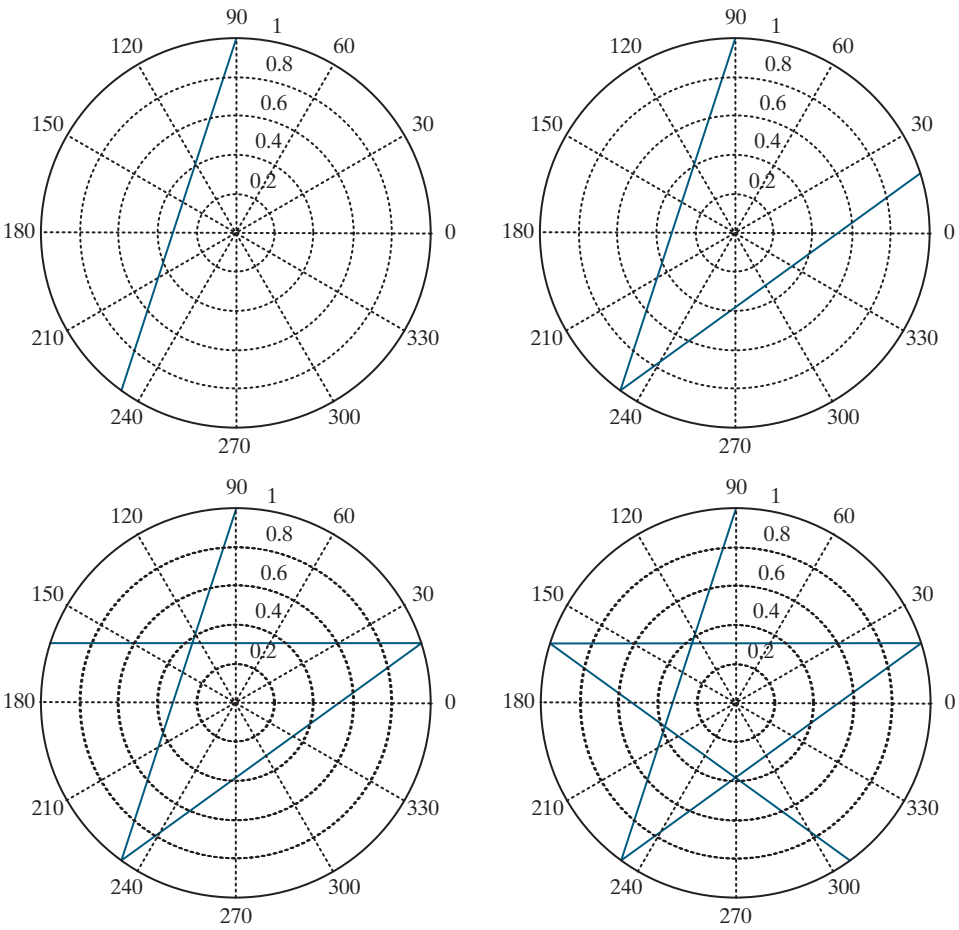
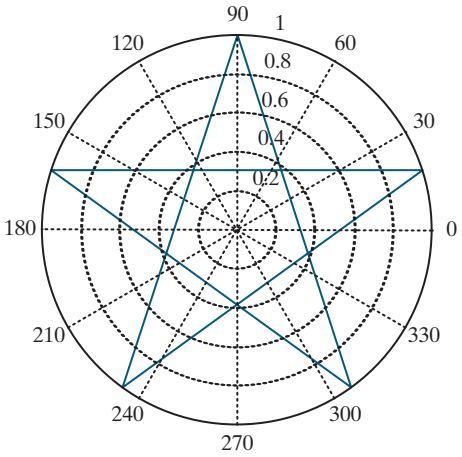


Figure P8.21
Steps required to draw a five-pointed star in polar coordinates.



Prompt the user to specify either a five-pointed or a six-pointed star, using a menu. Then create the star in a MATLAB figure window. Note that a six-pointed star is made of three triangles and requires a strategy different from that used to create a five-pointed star.

CHALLENGE PROBLEMS

8.22 Most major airports have separate lots for long-term and short-term parking. The cost to park depends on the lot you select, and how long you stay. Consider this rate structure.

- Long-Term (Economy) Parking
  - The first hour is $2.00, and each additional hour or fraction thereof is $1.00
  - Daily maximum $10
  - Weekly maximum $60.00
- Short-Term Parking
  - The first 30 minutes are $2.00 and each additional 20 minutes or fraction thereof is $1.00
  - Daily maximum $32.00

Write a program that asks the user the following:

- Which lot are you using?
- How many weeks, hours, days, and minutes did you park? Your program should then calculate the parking bill.

8.23 Lake Mead, like Lake Powell, is part of the Colorado River drainage system. Lake Mead sits on the border of Nevada and Arizona. It is the largest reservoir in the United States and is a critical water source for the people of Arizona, Nevada, California, and even some areas of Mexico. Due to a prolonged drought and increasing demand for water, the water level in the lake has been dropping since 1983 and in 2021 reached the lowest level since it was filled in the 1930s (Figure P8.23). The United States Bureau of Reclamation maintains data on water levels online, which is available to you. Search for information on the lower Colorado River if you have trouble finding it.

(a) Replace any missing data in the data set with NaN, using logical indexing.
(b) Create a plot of the data from 1935 to the present day, with appropriate annotations.
(c) Determine the average elevation of the water level for each year.
(d) At full capacity the water elevation is 1221.4 feet above sea level. Use logical indexing to determine how many months out of the lifetime of the reservoir it has NOT been at full capacity.



Figure P8.23
Lake Mead as seen from the Hoover Dam with the white band clearly showing the high water level on December 22, 2012. (Forbis/123RF)

CHAPTER

9 Repetition Structures

Objectives

After reading this chapter, you should be able to:

- Write and use `for` loops.
- Write and use `while` loops.
- Create midpoint break structures.
- Measure the time required to execute program components.
- Improve program execution times by using effective programming techniques.
- Explain the advantages of vectorizing code.

INTRODUCTION

A way to think of a computer program (not just MATLAB) is to consider how the statements that compose it are organized. Usually, sections of computer code can be categorized as **sequences**, **selection structures**, and **repetition structures**. The previous chapter described selection structures; in this chapter, we will focus on repetition structures. As a rule of thumb, if a section of code is repeated more than three times, it is a good candidate for a repetition structure.

KEY IDEA
If you find yourself repeating a section of code more than three times, it should probably be in a loop.

Repetition structures are also called **loops**. All loops consist of five basic parts:

- A parameter to be used in determining whether or not to end the loop
- Initialization of this parameter
- A way to change the parameter each time through the loop. (If you don't change it, the loop will never stop executing.)
- A comparison, using the parameter, to a criterion used to decide when to end the loop
- Calculations to do inside the loop

KEY IDEA
Loops allow you to repeat sequences of commands until some criterion is met.

MATLAB supports two different types of loops: the **`for` loop**, and the **`while` loop**. Two additional commands, `break` and `continue`, can be used to create a third type of loop, called a **midpoint break loop**. The `for` loop is the easiest choice when you know how many times you need to repeat the loop. `While` loops are the easiest choice when you need to keep repeating the instructions until a criterion is met. Midpoint break loops are useful for situations where the commands in the loop must be executed at least once, but where the decision to exit the loop is based on some criterion. If you have previous programming experience, you may be tempted to use loops extensively. However, in many cases you can compose MATLAB programs that avoid loops, either by using the `find` command, using logical indexing, or by vectorizing the code. (In **vectorization**, we operate on entire vectors at a time instead of one element at a time.) It's a good idea to avoid loops if a vectorized solution is possible, because vectorized programs run faster and usually require fewer programming steps.

9.1 `for` LOOPS

The structure of the `for` loop is simple. The first line identifies the loop and defines an index, which is a number that changes on each pass through the loop, and is used to determine when to end the repetitions. After the identification line comes the group of commands to execute. Finally, the end of the loop is identified by the command `end`. Thus, the structure of a `for` loop can be summarized as

KEY IDEA
Use `for` loops when you know how many times you need to repeat a sequence of commands.

```
for index = array
      commands to be executed
end
```

The loop is executed once for each element of the index array identified in the first line. Here's a really simple example:

```
for k = [1,3,7]
  k
end
```

During the first pass through the loop, `k` is assigned a value of 1, the first value in the `k` array. During the next pass the value of `k` is modified to 3, the second value in the `k` array. Each time through the loop, `k` is modified and assigned to subsequent values from the index array. This example code returns

```
k =
  1
k =
  3
k =
  7
```

Since there were 3 values of `k` in the array the loop executed 3 times, once for each value of `k`.

The index in this example is `k`. Programmers often use `k` as an index variable, as a matter of style. The index array can also be defined with the colon operator or, in a number of other ways as well. Here's an example of code that finds the value of 5 raised to powers between 1 and 3:

```
for
k = 1:3
  a = 5^k
end
```

On the first line, the index, k, is defined as the array [1, 2, 3]. The first time through the loop, k is assigned a value of 1, and 5^1 is calculated. Then the loop repeats, but now k is equal to 2, and 5^2 is calculated. The last time through the loop, k is equal to 3, and 5^3 is calculated. Because the statements in the loop are repeated three times, the value of a is displayed three times in the command window:

```
a =
  5
a =
  25
a =
  125
```

Although we defined k as an array in the first line of the for loop, because k is an index number when it is used in the loop, it can equal only one value at a time. After we finish executing the loop, if we call for k, it has only one value: the value of the index the final time through the loop. For the preceding example,

```
k
```

returns

```
k =
  3
```

Notice that k is listed as a 1×1 array in the workspace window.

A common way to use a for loop is in defining a new array. Consider, for example, the following code:

```
for k = 1:5
  a(k) = k^2
end
```

This loop defines a new array a, one element at a time. Since the program repeats its set of instructions five times, a new element is added to the a array each time through the loop, with the following output in the command window:

```
a =
    1
a  =
    1    4
a  =
    1    4    9
a  =
    1    4    9   16
a  =
    1    4    9   16   25
```

KEY IDEA
Avoid loops in favor of vectorizing code. Vectorized programs run faster and usually require fewer programming steps.

HINT

Most computer programs do not have MATLAB's ability to handle arrays so easily; therefore, they rely on loops similar to the one just presented to define arrays. It would be easier to create the vector `a` in MATLAB with the code

```
k = 1:5
a = k.^2
```

which returns

```
k =
   1    2    3    4    5
a =
   1    4    9   16   25
```

This is an example of *vectorizing* the code, and is much more efficient.

Another common use for a `for` loop is to combine it with an `if` statement and determine how many times something is true. For example, in the list of test scores shown in the first line, how many are above 90?

```
scores [76,45,98,97];
count = 0;
for k=1:length(scores)
  if scores(k)>90
     count = count + 1;
  end
end
disp (count)
```

The variable `count` is initialized as zero, then each time through the loop, if the score is greater than 90, the count is incremented by 1. Notice that the `length` function was used to determine how many times the `for` loop should repeat. In this case

```
length (scores)
```

is equal to four, the number of values in the `scores` array.

Most of the time, `for` loops are created with an index array that is a single row. However, if a two-dimensional array is defined in the index specification, MATLAB uses an entire column as the index each time through the loop. For example, suppose we define the index array as

$$\mathtt{k} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 4 & 9 \\ 1 & 8 & 27 \end{bmatrix}$$

Then

```
for k = [1, 2, 3; 1, 4, 9; 1, 8, 27]
  a  =  k'
end
```

returns

```
a =
   1     1     1
a =
   2     4     8
a =
   3     9     27
```

Notice that k was transposed when it was set equal to a, so our results are rows instead of columns. We did this to make the output easier to read.

We can summarize the use of for loops with the following rules:

- The loop starts with a for statement and ends with the word end.
- The first line in the loop defines the number of times the loop will repeat, using an index array.
- The index of a for loop must be a variable. (The index is the number that changes each and every time through the loop.) Although k is often used as the symbol for the index, any variable name may be employed. The use of k is a matter of style.
- Any of the techniques learned to define an array can be used to define the index array. One common approach is to use the colon operator, as in

```
for index = start:inc:final
```

- If the expression is a row vector, the elements are used one at a time—once for each time through the loop.
- If the expression is a two-dimensional array (this alternative is not common), each time through the loop, the index will contain the next *column* in the array. This means that the index will be a column vector.
- Once you've completed a for loop, the index variable retains the last value used.
- for loops can often be avoided by vectorizing the code.

The basic flowchart for a for loop includes a diamond, which reflects the fact that a for loop starts each pass with a check to see if there is a new value in the index array (see Figure 9.1). If there isn't, the loop is terminated, and the program continues with the statements after the loop.

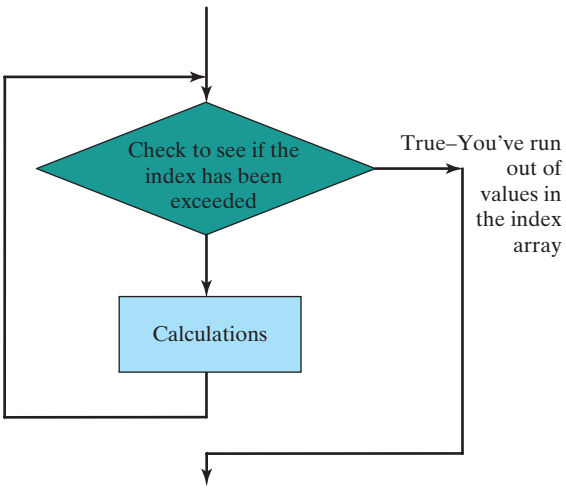


Figure 9.1
Flowchart for a for loop.

EXAMPLE 9.1

CREATING A DEGREES-TO-RADIANS TABLE

Although it would be much easier, and better coding practice to use MATLAB's vector capability to create a degrees-to-radians table, we can demonstrate the use of `for` loops with this example.

1. State the problem
   Create a table that converts angle values from degrees to radians, from 0 to 360 degrees, in increments of 10 degrees.

2. Describe the input and output

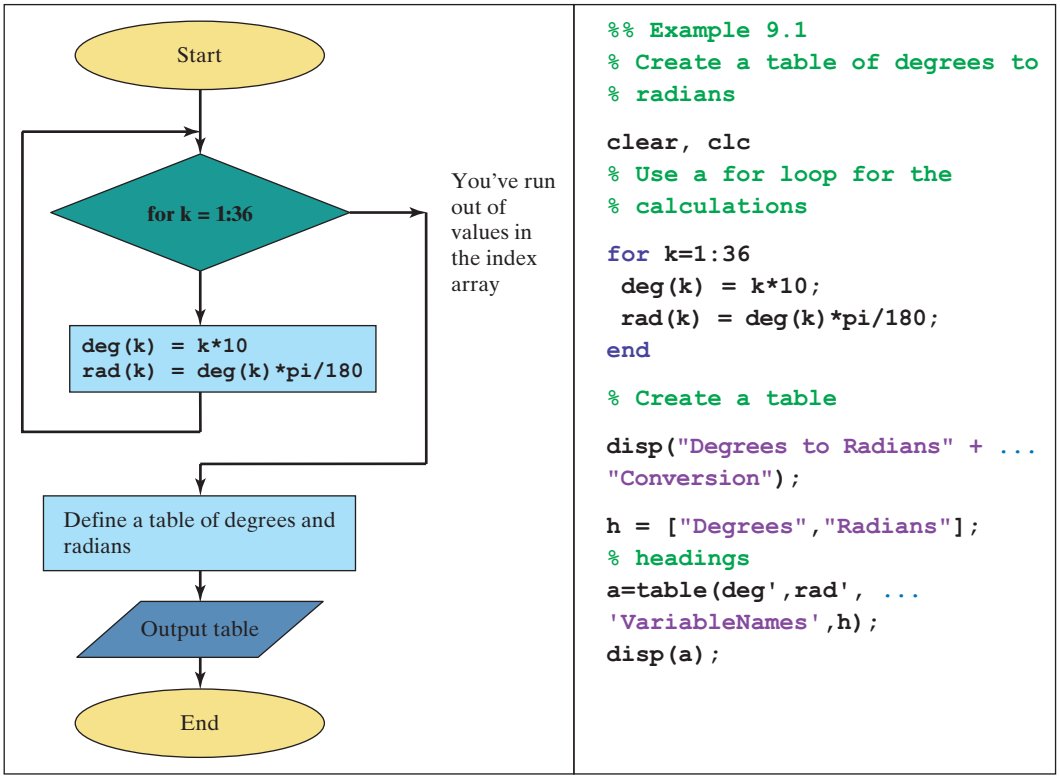
   Input An array of angle values in degrees

   Output A table of angle values in both degrees and radians

3. Develop a hand example
   For 10 degrees,

$$\text{Radians} = (10)\frac{\pi}{180} = 0.1745$$

4. Develop a MATLAB solution
   First develop a flowchart (see Figure 9.2) to help you plan your code, then convert the flow chart to a MATLAB program.




```
%% Example 9.1
% Create a table of degrees to
% radians

clear, clc
% Use a for loop for the
% calculations

for k=1:36
 deg(k) = k*10;
 rad(k) = deg(k)*pi/180;
end

% Create a table

disp("Degrees to Radians" + ...
"Conversion");

h = ["Degrees","Radians"];
% headings
a=table(deg',rad', ...
'VariableNames',h);
disp(a);
```

Figure 9.2
Flowchart for changing degrees to radians.

which gives the following results:

```
Degree to Radians Conversion

Degrees     Radians

   10       0.17453
   20       0.34907
   30       0.5236      etc.
```

5. Test the solution
   The value for 10 degrees calculated by MATLAB is the same as the hand calculation.

   It is much easier to use MATLAB's vector capabilities for this calculation. You get exactly the same answer, and it takes significantly less computing time. This approach is called vectorization of your code, and is one of the strengths of MATLAB. The vectorized code is

```
degrees = 0:10:360;
radians = degrees*pi/180;
```

EXAMPLE 9.2

CALCULATING FACTORIALS WITH A `for` LOOP

A factorial is the product of all the integers from 1 to *N*. For example, 5 factorial is

$$1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$$

In mathematics texts, factorial is usually indicated with an exclamation point:

$$5! \text{ is five factorial}$$

MATLAB contains a built-in function for calculating factorials, called `factorial`. However, suppose you would like to program your own factorial function called `fact`.

1. State the problem
   Create a function called `fact` to calculate the factorial of any number. Assume scalar input.

2. Describe the input and output

   Input A scalar value *N*

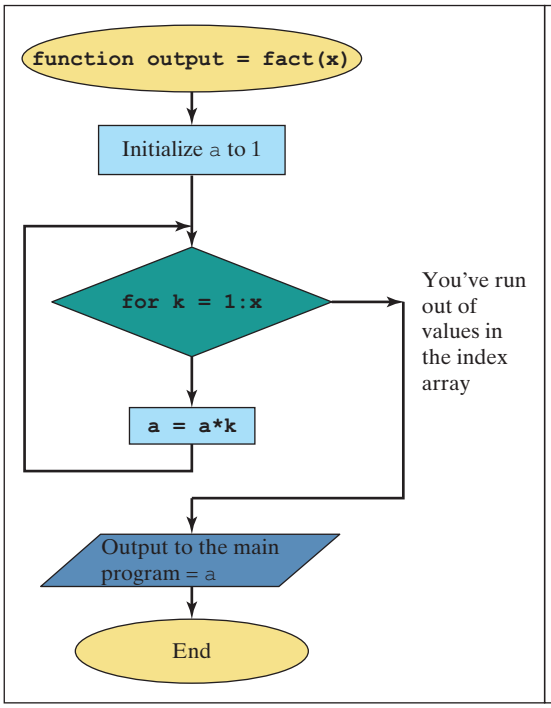
   Output The value of *N*!

3. Develop a hand example

$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$$

(*continued*)

4. Develop a MATLAB solution
First develop a flowchart (see Figure 9.3) to help you plan your code. Then convert the flowchart to a MATLAB program.




```
function output = fact(x)
% This function accepts a
% scalar
% input and calculates its
% factorial

a = 1;  % initialize a
% Use a loop to calculate the
% factorial
for k=1:x
   a=a*k;
end

output = a;
end
```

Figure 9.3
Flowchart for finding a factorial, using a `for` loop.

5. Test the solution
Test the function in the command window:

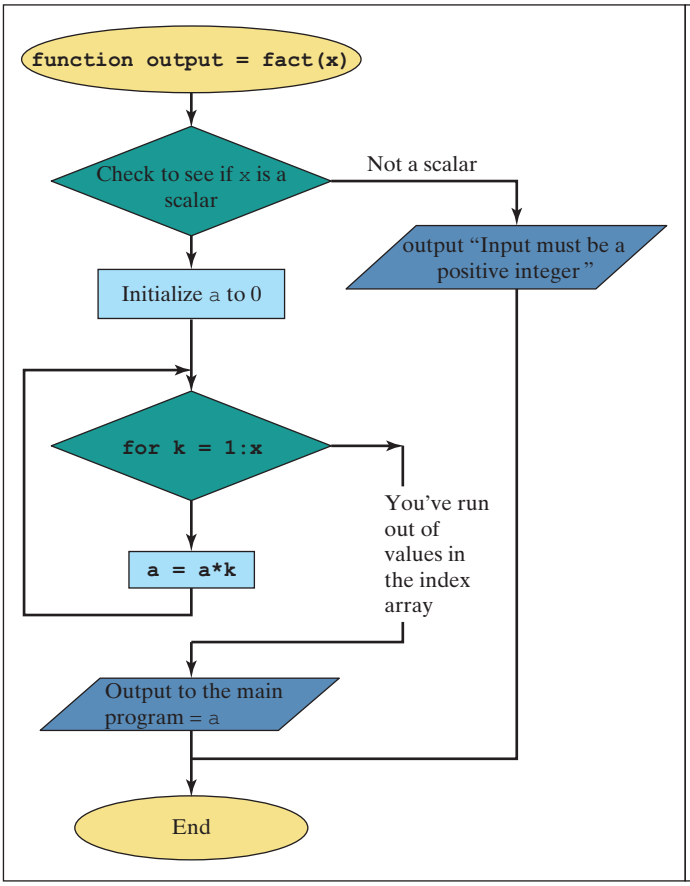
```
fact(5)
ans =
  120
```

This function works only if the input is a scalar. If an array is entered, the `for` loop does not execute, and the function returns a value of 1:

```
x=1:10;
fact(x)
ans =
  1
```

You can add an `if` statement to confirm that the input is a positive integer and not an array, as shown in the flowchart in Figure 9.4 and the accompanying code. Check the new function in the command window:

```
fact(-4)
Error using fact (line 8)
Input must be a positive integer
fact(x)
Error using fact (line 8)
Input must be a positive integer
```




```
function output = fact(x)
% This function accepts a scalar
% input and calculates its factorial
% Check to confirm x is a single
% value array
if(length(x)>1 | x<0)
   error(["Input must be a "+...
    "positive integer"]);
else

   a = 1;  % initialize a
   % Use a loop to calculate the
   % factorial
   for k = 1:x
     a = a*k;
   end
   output = a;
end
end
```

Figure 9.4
Flowchart for finding a factorial, including error checking.

PRACTICE EXERCISE 9.1

Use a `for` loop to solve the following problems. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Create a table that converts inches to feet.
2. Consider the following array of values:

$$x = [45,\ 23,\ 17,\ 34,\ 85,\ 33]$$

   How many values are greater than 30? (Use a counter.)
3. Repeat Exercise 2, this time using the logical indexing command.
4. Use a `for` loop to sum the elements of the array in Problem 2. Check your results with the `sum` function. (Use the help feature if you don't know or remember how to use `sum`.)

5. Use a `for` loop to create a vector containing the first 10 elements in the harmonic **series**, that is,

$$1/1 \quad 1/2 \quad 1/3 \quad 1/4 \quad 1/5 \ldots 1/10$$

6. Use a `for` loop to create a vector containing the first 10 elements in the alternating harmonic series, that is,

$$1/1 \quad -1/2 \quad 1/3 \quad -1/4 \quad 1/5 \ldots -1/10$$

9.2 `while` LOOPS

`While` loops are similar to `for` loops. The big difference is the way MATLAB decides how many times to repeat the loop. `While` loops continue until some criterion is met. The format for a `while` loop is:

KEY IDEA
Use `while` loops when you don't know how many times a sequence of commands will need to be repeated.

```
while  criterion
  commands to be executed
end
```

Here's an example:

```
k = 0;
while k<3
 k = k+1
end
```

In this case, a counter, `k`, was initialized before the loop. (Initialized just means it was set equal to a starting value.) Then the loop repeated as long as `k` was less than 3. The variable `k` was incremented by 1 every time through the loop, so that the loop repeated three times, giving

```
k =
  1
k =
  2
k =
  3
```

Notice that when `k=3`, the criterion in the `while` statement

```
k<3
```

is false. Thus, when MATLAB checks to see if it should make another pass through the loop, the program makes the decision (based on the criterion) to skip to the end of the structure.

KEY IDEA
Any problem that can be solved using a `for` loop can also be solved using a `while` loop.

The `k` variable can be used as an index number to define an array or just as a counter. Any problem that can be formulated as a `for` loop can also be formulated as a `while` loop. Recall the `for` loop in the previous section used to calculate the first three powers of 5. The following `while` loop accomplishes the same task:

```
k = 0;
while k<3
  k = k+1;
  a(k) = 5^k
end
```

The code returns

```
a =
  5
a =
  5   25
a =
  5   25   125
```

Each time through the loop, another element is added to the array a. As another example, first **initialize** a:

```
a = 0;
```

Then find the first multiple of 3 that is greater than 10:

```
while(a<10)
  a = a + 3
end;
```

The first time through the loop, a is equal to 0, so the comparison is true. The next statement (a = a + 3) is executed, and the loop is repeated. This time a is equal to 3 and the condition is still true, so execution continues. In succession, we have

```
a =
  3
a =
  6
a =
  9
a =
  12
```

The last time through the loop, a starts out as 9, then becomes 12 when 3 is added to 9. The comparison is made one final time, but since a is now equal to 12—which is greater than 10—the program skips to the end of the while loop and no longer repeats.

While loops can also be used to count how many times a condition is true by incorporating an if statement. Recall the test scores we counted in a for loop earlier. We can also count them with a while loop:

```
scores = [76,45,98,97];
count = 0;
k = 0;
while k<length(scores)
  k = k+1;
   if scores(k)>90
      count = count + 1;
  end
end
disp(count)
```

The variable count is used to count how many values are greater than 90. The variable k is used to count how many times the loop is executed.

The basic flow chart for a while loop (see Figure 9.5) is the same as that for a for loop (see Figure 9.1).

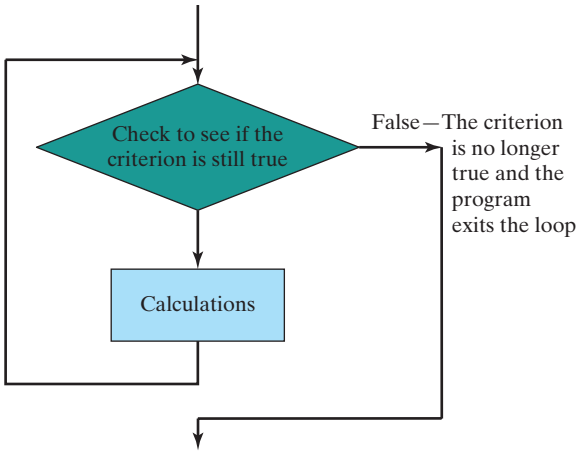


Figure 9.5
Flowchart for a `while` loop.

One common use for a `while` loop is error checking of user input. Consider a program where we prompt the user to input a positive number, and then we calculate the log base 10 of that value. We can use a `while` loop to confirm that the number is positive, and if it is not, to prompt the user to enter an allowed value. The program keeps on prompting for a positive value until the user finally enters a valid number.

```
x = input("Enter a positive value of x:")
while (x<=0)
  disp("log(x) is not defined for negative numbers")
  x = input("Enter a positive value of x:")
end
  y = log10(x);
fprintf("The log base 10 of %4.2f is %5.2f \n",x,y)
```

KEY IDEA
It is easy to create an `infinite loop` with a `while` structure.

If, when the code is executed, a positive value of `x` is entered, the `while` loop does not execute (since `x` is not less than 0). If, instead, a zero or negative value is entered, the `while` loop is executed, an error message is sent to the command window, and the user is prompted to reenter the value of `x`. The `while` loop continues to execute until a positive value of `x` is finally entered.

HINT

The variable used to control the `while` loop must be updated every time through the loop. If not, you'll generate an **infinite loop**. When a calculation is taking a long time to complete, you can confirm that the computer is really working on it by checking the lower left-hand corner for the "busy" indicator. If you want to exit the calculation manually, type `Ctrl c`. (Depress the `Ctrl` and `c` key at the same time.) Make sure that the command window is the active window when you execute this command.

HINT

Many computer texts and manuals indicate the control key with the ^symbol. This is confusing at best. The command `^C` usually means to strike the `Ctrl` key and the `c` key at the same time.

EXAMPLE 9.3

CREATING A TABLE FOR CONVERTING DEGREES TO RADIANS WITH A while LOOP

Just as we used a `for` loop to create a table for converting degrees to radians in Example 9.2, we can use a `while` loop for the same purpose.

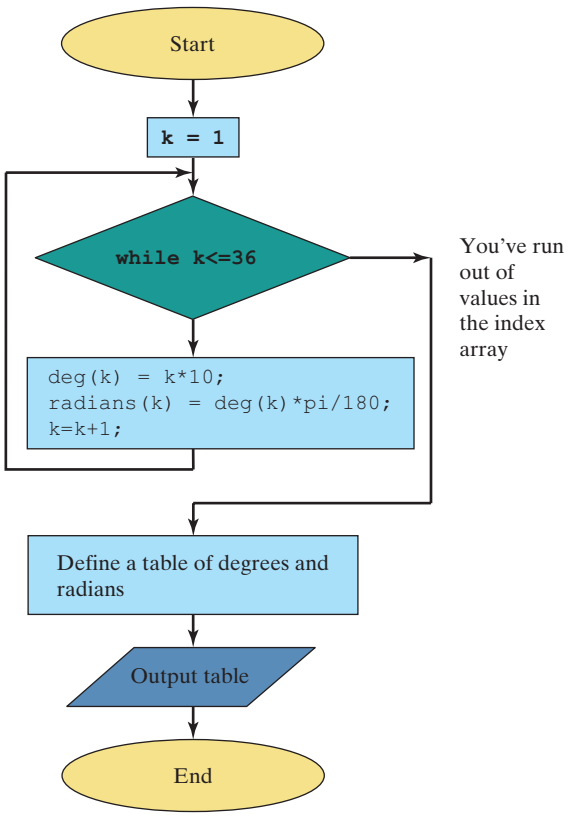
1. State the problem
   Create a table that converts degrees to radians, from 0 to 360 degrees, in increments of 10 degrees.
2. Describe the input and output

   Input An array of angle values in degrees

   Output A table of angle values in both degrees and radians
3. Develop a hand example
   For 10 degrees,

$$\text{radians} = (10)\frac{\pi}{180} = 0.1745$$

4. Develop a MATLAB solution
   First develop a flowchart (see Figure 9.6) to help you plan your code, then convert the flowchart to a MATLAB program.




```
%% Example 9.3
% Create a table of degrees to radians
clear, clc
% Use a while loop for the
% calculations
k=1;
while k<=36
    deg(k) = k*10;
    rad(k) = deg(k)*pi/180;
    k = k+1;
end
% Create a table
disp("Degrees to Radians Conversion");
h=["Degrees","Radians"]; % headings
a=table(deg',rad','VariableNames',h);
disp(a);
```

Figure 9.6
Flowchart for converting degrees to radians with a `while` loop.

(*continued*)

which gives the following results:

```
Degrees to Radians Conversion

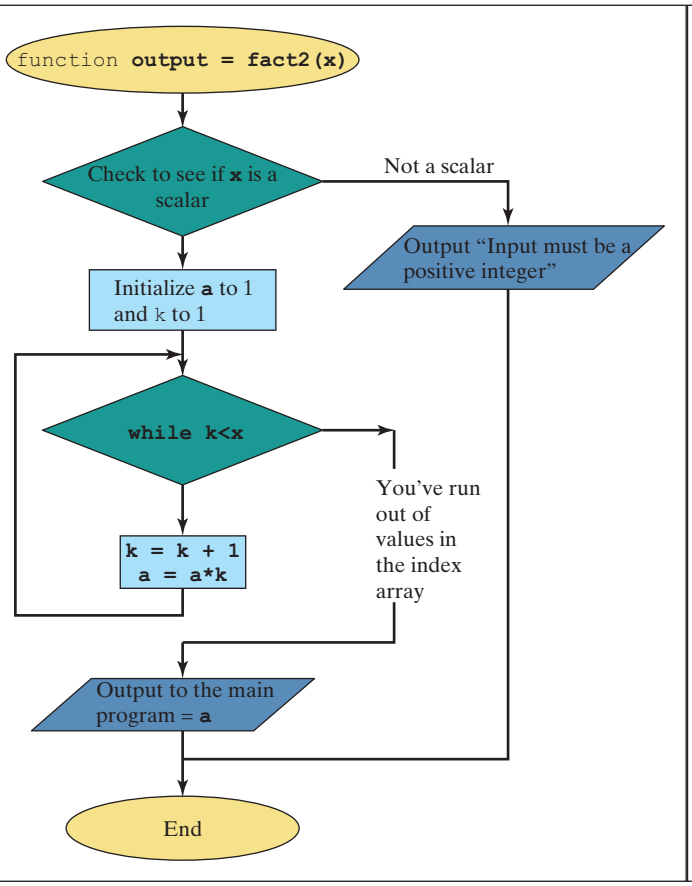
Degrees     Radians
  10        0.17453
  20        0.34907
  30        0.5236        etc.
```

5. Test the solution
   The value for 10 degrees calculated by MATLAB is the same as the hand calculation.

EXAMPLE 9.4

CALCULATING FACTORIALS WITH A `while` LOOP

Create a new function called `fact2` that uses a `while` loop to find $N!$. Include an `if` statement to check for negative numbers and to confirm that the input is a scalar.




```
function output = fact2(x)
% This function uses a while loop to
% find x!
% The input must be a positive
% integer
if(length(x)>1 | x<0)
    error(["Input must be a "+...
    "positive integer"]);
else
    a = 1; % Initialize the running
% product
    k = 1; % Initialize the counter
while k<x
    k = k + 1; % Increment the counter
    a = a*k; % Calculate the running
% product
end
output = a;
end
end
```

Figure 9.7
Flowchart for finding a factorial with a `while` loop.

1. State the problem
   Create a function called `fact2` to calculate the factorial of any number.
2. Describe the input and output

   Input A scalar value N

   Output The value of $N!$
3. Develop a hand example

$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$$

4. Develop a MATLAB solution
   First develop a flowchart (see Figure 9.7) to help you plan your code, then convert the flowchart to a MATLAB program.
5. Test the solution
6. Test the function in the command window:

```
fact2(5)
ans =
  120
fact2(-10)
Error using fact (line 8)
Input must be a positive integer
fact2([1:10])
Error using fact (line 8)
Input must be a positive integer
```

EXAMPLE 9.5

THE ALTERNATING HARMONIC SERIES

The alternating harmonic series **converges** to the natural log of 2:

$$\sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \ldots = \ln(2) = 0.6931471806$$

Because of this, we can use the alternating harmonic series to approximate the $\ln(2)$. But how far out do you have to take the series to get a good approximation of the final answer? We can use a `while` loop to solve this problem.

1. State the problem
   Use a `while` loop to calculate the members of the alternating harmonic sequence and the value of the series until it converges to values that vary by less than .001. Compare the result to the natural log of 2.
2. Describe the input and output

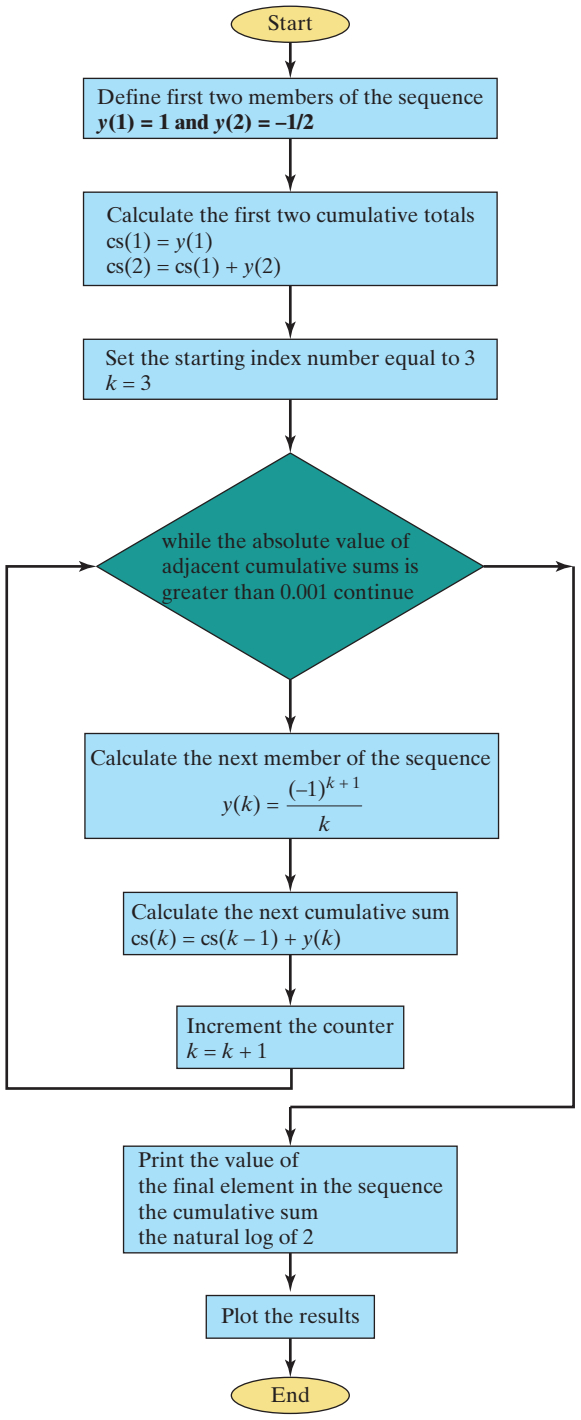
   Input The description of the alternating harmonic series

$$\sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \ldots \frac{1}{\infty}$$

   Output The value of the truncated series, once the convergence criterion is met.

   Plot the cumulative sum of the series elements, up to the point where the convergence criterion is met.

(*continued*)




```
%% Example 9.5
% Calculating the Alternating Harmonic
% Series
clear, clc
% Define the first two terms in the
% series
y(1) = 1;
y(2) = -1/2;
% Find the first two cumulative sums
cs(1) = y(1);
cs(2) = cs(1) + y(2);
k=3;
while (abs(cs(k-1) - cs(k-2))> .001)
     y(k) = (-1)^(k+1)/k;
    cs(k) = cs(k-1) + y(k);
    k=k+1;
end

disp("At convergence")
fprintf("y(n) is %8.6f \n", y(k-1));
fprintf("cs(n) is %5.4f \n",cs(k-1))
fprintf("k is %3.0f \n",k)
fprintf("ln(2) is,%5.4f \n", log(2));

%% Plot the results
semilogx(cs)
title("Value of the Alternating
Harmonic Series")
xlabel("Number of Terms")
ylabel("Sum of the terms")
```

Figure 9.8
Flowchart to evaluate the alternating harmonic series until it converges.

3. Develop a hand example
Let's calculate the value of the alternating harmonic series for 1 to 5 terms. First, find the value for each of the first five terms in the sequence

$$1.0000 \quad -0.5000 \quad 0.3333 \quad -0.2500 \quad 0.2000$$

Now, calculate the sum of the series assuming 1 to 5 terms

$$1.0000 \quad 0.5000 \quad 0.8333 \quad 0.5833 \quad 0.7833$$

The calculated sums are getting closer together, as we can see if we find the difference between adjacent pairs

$$-0.5000 \quad 0.3333 \quad -0.2500 \quad 0.2000$$

4. Develop a MATLAB solution
First, develop a flowchart (see Figure 9.8) to help you plan your code, then convert it to a MATLAB program. When we run the program, the following results are displayed in the command window.

```
The sequence converges when the final element is equal to 0.001
At which point the value of the series is 0.6936
This compares to the value of the ln(2), 0.6931
The sequence took 1002 terms to converge
```

The series is quite close to the ln(2), but perhaps we could get closer with more terms. If we change the convergence criterion to 0.0001 and run the program, we get the following results:

```
The sequence converges when the final element is equal to
 -0.000
At which point the value of the series is 0.6931
This compares to the value of the ln(2), 0.6931
The sequence took 10001 terms to converge
```

5. Test the solution
Compare the result of the hand solution to the MATLAB solution, by examining the graph (see Figure 9.9). The first five values for the series match those displayed in the graph. We can also see that the series seems to be converging to approximately 0.69, which is approximately the natural log of 2.

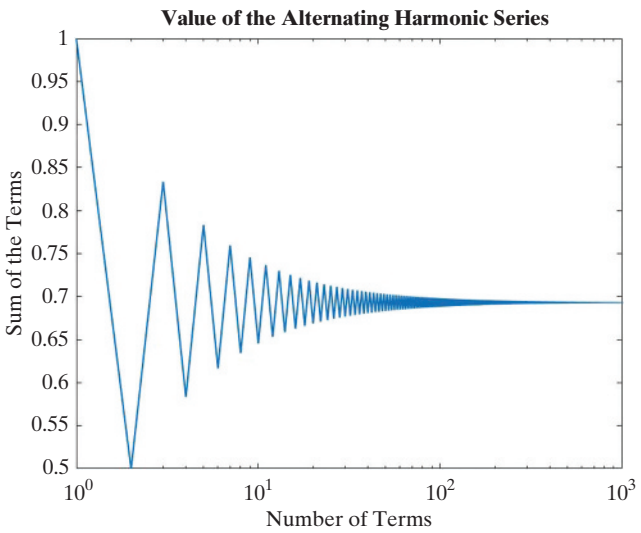


Figure 9.9
The alternating harmonic series converges slowly to the ln(2).

PRACTICE EXERCISE 9.2

Use a `while` loop to solve the following problems. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Create a conversion table of inches to feet. Display your results in a table array.
2. Consider the following array of values:

   $$x = [45,\ 23,\ 17,\ 34,\ 85,\ 33]$$

   How many values are greater than 30? (Use a counter.)
3. Compare your solution to Exercise 2 to the solutions you created in Practice Exercise 9.1, where you used both a `for` loop and logical indexing to solve the same problem.
4. Use a `while` loop to sum the elements of the array in Exercise 2. Check your results with the `sum` function. (Use the help feature if you don't know or remember how to use `sum`.)
5. Use a `while` loop to create a vector containing the first 10 elements in the harmonic series, that is,

   $$1/1\ \ 1/2\ \ 1/3\ \ 1/4\ \ 1/5 ... 1/10$$

6. Use a `while` loop to create a vector containing the first 10 elements in the alternating harmonic series, that is,

   $$1/1\ \ -1/2\ \ 1/3\ \ -1/4\ \ 1/5 ... -1/10$$

HINT

To make the signs in a sequence of numbers alternate, raise -1 to a power. For example, to generate the values in a vector

```
-1 2 -3 4 -5
```

use the following loop

```
for k = 1:5
  a(k) = (-1)^k*k
end
```

Note: Order of operation is important. The result of

```
(-1)^k
```

alternates the sign depending on the value of `k`. Without the parentheses, the result of

```
-1^k
```

is always -1.

9.3 break AND continue

The `break` command can be used to terminate a loop prematurely (while the comparison in the first line is still true). A `break` statement will cause termination of the smallest enclosing `while` or `for` loop. Here's an example:

```
n = 0;
while(n<10)
  n = n+1;
  a = input('Enter a value greater than 0:');
  if(a<=0)
    disp('You must enter a positive number')
    disp('This program will terminate')
    break
  end
  disp('The natural log of that number is')
  disp(log(a))
end
```

In this program, the value of `n` is initialized outside the loop. Each time through, the input command is used to ask for a positive number. The number is checked, and if it is zero or negative, an error message is sent to the command window and the program jumps out of the loop. If the value of `a` is positive, the program continues and another pass through the loop occurs, until `n` is finally greater than 10.

The `continue` command is similar to `break`; however, instead of terminating the loop, the program just skips to the next pass:

```
n=0;
while(n<10)
  n=n+1;
  a=input("Enter a value greater than 0:");
  if(a<=0)
    disp("You must enter a positive number")
    disp("Try again")
    continue
  end
  disp("The natural log of that number is")
  disp(log(a))
end
```

In this example, if you enter a negative number, the program lets you try again—until the value of `n` is finally greater than 10.

9.4 MIDPOINT BREAK LOOPS

The loops described in the previous sections are examples of midpoint break loops. In these constructs, the loop is entered, calculations are processed, and a decision is made at some arbitrary point in the loop whether or not to exit. Then, additional calculations are processed, and the loop repeats. This strategy can be used either with a `for` loop or a `while` loop.

In a `while` structure, the loop continues to repeat until the criterion specified in the first line of the loop is false. For example

KEY IDEA
Midpoint break loops require the logic in the loop to execute at least once.

```
while (x>.001)
  . . . do some calculations that result in updating x
end
```

When the comparison between `x` and 0.001 is evaluated, either a 1 (for true) or a 0 (for false) is returned. If the result is 0, the loop is terminated. One potential

problem with this structure is that if the original value of x is very small, for example, in this case 0.0005, the loop will never execute. A way around this is to force the result to true, and add an `if` statement and corresponding `break` structure

```
while(1)
   . . . do some calculations
   if (x<=.001)
      break
   end
   . . . do any additional calculations or information
      processing
end
```

Recall that if a comparison is true, MATLAB returns the number 1, so saying `while(1)` is equivalent to always making the comparison that starts a pass through the `while` loop true. The `while(1)` implementation allows the loop to continue executing for an infinite number of iterations. The decision to exit the loop is then controlled by the `if/break` structure. When would this be useful? One example is error checking, similar to the example in the previous problem. Consider another MATLAB program that prompts the user to enter the number of candy bars purchased, and then finds the cost to the user. If the user enters a negative number, the program should prompt the user to try again. If a positive number is entered, the program completes the calculations and exits the loop.

```
while(1)
candy_bars = input("Enter the number of candy bars");
  if candy_bars<0
    disp("Must be a positive number")
  else
    total = candy_bars *.75;
    fprintf("The total cost is %5.2f dollars \n",total)
    break
  end
end
```

Here's the interaction when the program is executed.

```
Enter the number of candy bars -3
Must be a positive number
Enter the number of candy bars 5
The total cost is 3.75 dollars
```

One issue with this strategy is that the loop need never end. In this program, if the user keeps replying with a negative number, the program will continue to prompt for a positive value. One way to get around this is to use a `for` loop, which has a preset number of iterations. In this example, we've defined 3 as the maximum number of passes through the loop.

```
for k=1:3
    candy_bars = input("Enter the number of candy bars:");
    if candy_bars<0
        disp("Must be a positive number")
    else
```

```
            total = candy_bars *.75;
            fprintf("The total cost is %5.2f dollars \n",total)
            break
        end
    end
```

Here's the command window interaction.

```
Enter the number of candy bars -3
Must be a positive number
Enter the number of candy bars -2
Must be a positive number
Enter the number of candy bars -5
Must be a positive number
```

After three iterations, the loop ends.

These may seem like trivial examples. A more complicated case is described in Example 9.6.

EXAMPLE 9.6

Calculating the value of the alternating harmonic series in order to approximate the value of ln(2) (as illustrated in Example 9.5) is an example of a numerical method. Many functions that you use routinely, such as `sine` and `cosine`, are approximated using similar series, called Taylor series or Maclaurin series. The alternating harmonic series is an example of a series that converges, but not every series does. For example, simply changing the alternating negative signs in the alternating harmonic series to positive numbers (the harmonic series vs. the alternating harmonic series) results in a series that **diverges**—it just keeps getting bigger and bigger with every new term. In cases such as these, we would want to specify a maximum number of iterations in our problem before giving up and exiting the loop.

A less obvious example of a series that diverges is

$$1 - 2 + 3 - 4 + 5 - 6 \ldots + (n \ -1) - n$$

which can be expressed mathematically as

$$\sum_{k=1}^{n} (-1)\text{^}(k + 1) * k$$

Write a program that calculates the value of the summation. Assume that we don't know that it diverges, and specify an exit from the loop if two adjacent values of the cumulative sum are less than 0.01. Also, specify a maximum of 50 iterations.

1. State the problem
   Calculate the sum of the alternating series, assuming it converges, to within 0.001

2. Describe the input and output

   Input $\sum_{k=1}^{n} (-1)\text{^}(k + 1) * k$

   Output Find the cumulative sum of the series for each iteration

   Create a plot of the cumulative sums versus the number of terms

3. Develop a hand example
   The first six terms in the series are

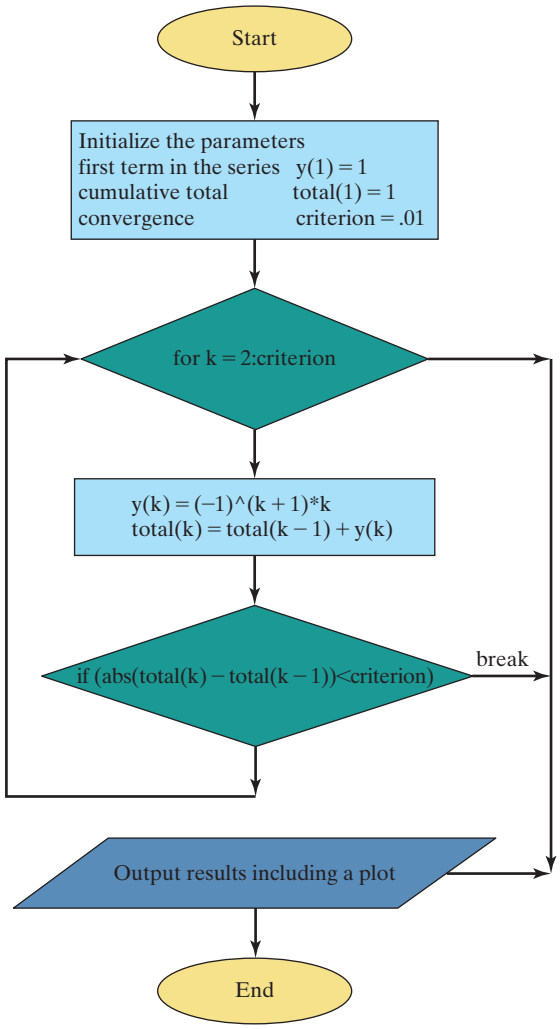
   $$1 - 2 + 3 - 4 + 5 - 6$$

(*continued*)

Thus, the first six cumulative sums are

$n = 1$	total $= 1$
$n = 2$	total $= 1 - 2 = -1$
$n = 3$	total $= 1 - 2 + 3 = 2$
$n = 4$	total $= 1 - 2 + 3 - 4 = -2$
$n = 5$	total $= 1 - 2 + 3 - 4 + 5 = 3$
$n = 6$	total $= 1 - 2 + 3 - 4 + 5 - 6 = -3$

4. Develop a MATLAB solution
Outline your program in a flowchart, as shown in Figure 9.10. Next, convert the flowchart to pseudocode comments, and insert the appropriate MATLAB code.




```
%% Example 9.6
% Calculating the Alternating Harmonic Series
clear, clc
% Define the starting parameters
y(1) = 1;
total(1) = y(1);
criterion = 0.01;
max_iterations = 50;
%% Execute the loop
for k=2:max_iterations
    y(k) = (-1)^(k+1)*k;
    total(k) = total(k-1) + y(k);
    if (abs(total(k) - total(k-1))<criterion)
        break
    end
end
%% Specify the output
if k==max_iterations
    disp("Did not converge")
    fprintf("Iterations = %5.0f \n",k)
    fprintf("Total Value = %5.4f
\n",total(k))
else
    disp("Converged")
    fprintf("Iterations = %5.0f \n",k)
    fprintf("y(k) equals %8.3f \n", y(k));
    fprintf("Total Value = %5.4f
\n",total(k))
end
%% Plot the results
plot(total)
title("Summation of Alternating Terms")
xlabel("Number of iterations")
ylabel("Cumulative Sum")
```

Figure 9.10
Flowchart for calculating the cumulative sums of the alternating numeric series.

When this program is executed, the result in the command window is:

```
Did not converge
Iterations =   50
Total Value = -25.0000
```

The resulting plot is shown in Figure 9.11.

5. Test the solution
The MATLAB solution matches the hand calculations for the first six terms of the series. If we had not programmed in a maximum number of iterations, in the form of a `for` loop structure, the program would have entered an infinite loop. An alternate approach in a `while` loop would be to use an if structure in conjunction with a break statement to determine if a maximum number of iterations had been reached.

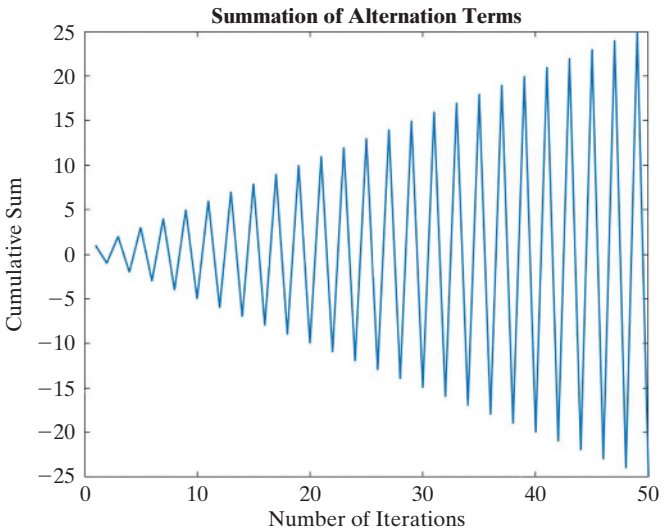


Figure 9.11
The cumulative sum of the alternating numeric series does not converge, but rather oscillates around zero.

9.5 NESTED LOOPS

It is often useful to **nest** loops inside other loops. This is actually how many of the MATLAB built-in functions operate. For example, consider the `max` function. This function looks for the maximum value for each column in an array. We can develop a program to find the maximum, using a simple 4×4 array, `x`.

```
x = [1    2    6    3;
     4    8    2    1;
     12   18   3    5;
     6    4    2    13]
```

If we use the `max` function

```
max(x)
```

MATLAB returns the maximum value in each column

```
ans =
  12 18 6 13
```

KEY IDEA
Nested loops can be used to evaluate multidimensional data.

We can achieve the same result with nested `for` loops. First, determine the dimensions of the `x` array, using the `size` function.

```
[rows,cols]=size(x);
```

Using the information from the `size` function create the external `for` loop, which then is programmed to execute once for each column in the array. Next, define a provisional value for the maximum, based on the first value in each column. Finally, use an internal `for` loop, which will execute once for each row in the array.

```
for k=1:cols ..... External for loop
  maximum(k)=x(1,k); ..... Internal for loop
    for j=1:rows ..... If structure
      if x(j,k)>maximum(k)
      maximum(k)=x(j,k);
    end .....
  end .....
end .....
maximum % returns the results
```

9.6 IMPROVING THE EFFICIENCY OF LOOPS

KEY IDEA
Loops are generally less efficient than vectorized calculations.

In general, using a `for` loop (or a `while` loop) is less efficient in MATLAB than using array operations. We can test this assertion by timing the multiplication of the elements in a very large array, using the `clock` function and the function `etime`, which measures elapsed time. If you have a fast computer, you may need to use a larger array than the one in this example. The structure of the clocking code is

```
t0 = clock;
. . . code to be timed
etime (clock, t0)
```

The `clock` function polls the computer clock for the current time. The `etime` function compares the current time with the initial time, and subtracts the two values to give the elapsed time.

Here's an example.

```
clear, clc
A = ones(1000); % Creates a 1000 x 1000 array of ones
t0 = clock;
B = A*pi; % Multiplies 1*pi
time = etime(clock, t0)
```

which gives a result of

```
time =
     0.003
```

The array calculation took 0.003 seconds. Every time you run these lines of code, you should get a different answer. The `clock` and `etime` functions used here measure how long the CPU worked between receiving the original and final timing requests. However, the CPU is doing other things besides this problem: At a minimum, it is performing system tasks, and it may be running other programs in the background.

To measure the time required to perform the same calculation with a loop, clear the memory and re-create the array of ones:

```
clear
A = ones(1000);
```

This ensures that we are comparing calculations from the same starting point.

```
t0 = clock;
for k = 1:length(A(:))
   B(k) = A(k)*pi; % Multiplies 1*pi
end
time = etime(clock, t0)
```

which gives the result

```
time =
     0.185
```

KEY IDEA
Preallocate memory to improve run times.

It took almost 0.2 seconds to perform the same calculation! (This was on an older computer—your result will depend on the machine you use.) The number of iterations through the `for` loop was determined by finding how many elements are in `A`. This was accomplished with the `length` command. Recall that `length` returns the largest array dimension, which is 1000 for our array and isn't what we want. To find the total number of elements, we used the colon operator (:) to represent `A` as a single list, 1 million elements long, and then used `length`, which returned 1,000,000. Each time through the `for` loop, a new element was added to the `B` array. This is the step that took all the time, since the computer must allocate additional memory 1,000,000 times. We can reduce the time required for this calculation by creating the `B` array first (so that the memory allocation process takes place only once) and then replacing the values one at a time. The code is

```
clear
A = ones(1000);
t0 = clock;
B = A; % Create a B array of ones
  for k = 1:length(A(:))
    B(k) = A(k)*pi;
  end
time = etime(clock, t0)
```

which gives the result

```
time =
     0.031
```

This is a significant improvement. You could see an even bigger difference between the first example, a simple multiplication of the elements of an array, and the last example if you created a larger array. By contrast, the intermediate example, in which we did not initialize `B`, would take a prohibitive amount of time to execute.

MATLAB also includes a set of commands called `tic` and `toc` that can be used in a manner similar to the `clock` and `etime` functions to time a piece of code.

```
clear
A = ones(1000);
tic
  B = A;
  for k = 1:length(A(:))
    B(k) = A(k)*pi;
  end
toc
```

returns

```
Elapsed time is 0.031090 seconds.
```

KEY IDEA
For small programs, background tasks may take as much or more time as the actual code to execute.

The difference in execution time is expected, since the computer is busy doing different background tasks each time the program is executed. These background tasks are often called **housekeeping** functions. As with `clock/etime`, the `tic/toc` commands measure elapsed time, not the time devoted to just this program's execution.

The `timeit` function is a little more complicated to use, but it measures the amount of computer time devoted to executing the code . . . not the elapsed time. The code to be timed must be stored as a function. For example, this function is named `f`, and has no input and no output but stores all the logic from the previous example.

```
function []= f()
A = ones(1000);
B = A;
for k = 1:length(A(:))
  B(k) = A(k)*pi;
end
end
```

To time the function call the `timeit` function, using a function handle to `f` as the input. You can store the function at the end of the program (either a script or live script.)

```
t=timeit(@f)
```

The result for this example is 0.01685 seconds, indicating that most of the time was spent on computer tasks other than the MATLAB calculations.

The functions described to this point are programmatic—they must be embedded in a MATLAB program. Run times can also be determined using the *Run and Time* icon found on the Editor toolstrip. (See Figures 9.12 and 9.13.)

HINT

Suppress intermediate calculations when you use a loop. Printing values to the screen will greatly increase the amount of execution time. If you are brave, repeat the preceding example, but delete the semicolons inside the loop just to check out this claim. Don't forget that you can stop the execution of the program with `Ctrl c`. Be sure the command window is the active window when you execute `Ctrl c`.

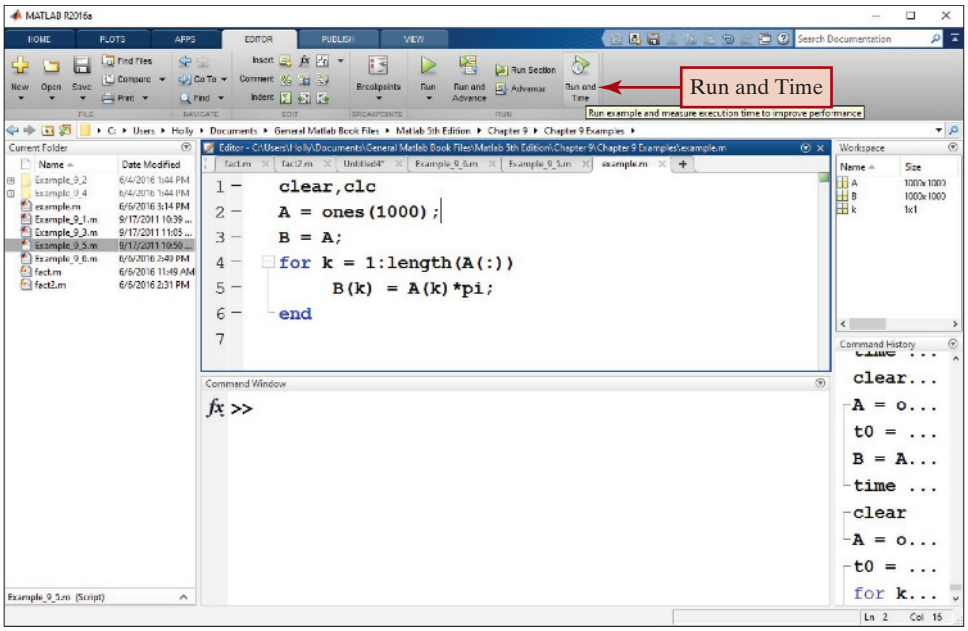


Figure 9.12
MATLAB allows you to measure execution times using the *Run and Time* icon from the Editor toolstrip.

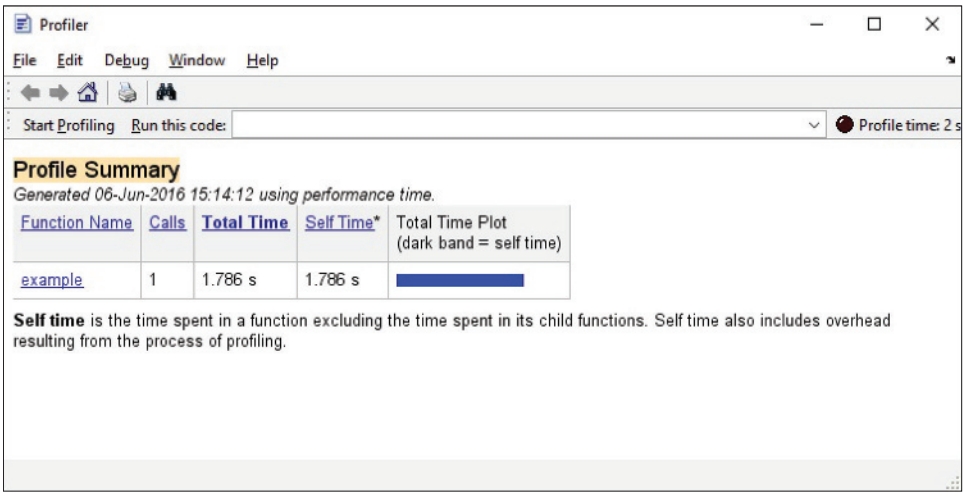


Figure 9.13
The Profile Summary is generated when a script is executed using the *Run and Time* icon from the Editor toolstrip.

SUMMARY

Repetition structures (loops) are used when a section of code needs to be repeated several times. As a rule of thumb, if you find yourself repeating a section of code more than three times, it probably should be in a repetition structure. MATLAB supports two types of repetition structures: the `for` loop and the `while` loop. In addition, the `break` and `continue` commands can be used to modify either type of loop to create a midpoint break loop.

`For` loops are used mainly when the programmer knows how many times a sequence of commands should be executed. `While` loops are used when the commands should be executed until a condition is met. Most problems can be structured so that either `for` or `while` loops are appropriate.

The `break` and `continue` statements are used to exit a loop prematurely. They are usually used in conjunction with `if` statements. The `break` command causes a jump out of a loop and execution of the remainder of the program. The `continue` command skips execution of the current pass through a loop, but allows the loop to continue until the completion criterion is met. This type of structure, called a midpoint break loop, is commonly used in applications such as numerical analysis.

Vectorization of MATLAB code allows it to execute much more efficiently and, therefore, more quickly. Loops should be avoided in MATLAB if the code can be formulated into a vectorized format. When loops are unavoidable, they can be improved by defining "dummy" variables with placeholder values, such as ones or zeros to preallocate memory. These placeholders can then be replaced in the loop. Doing this will result in significant improvements in execution time, a fact that can be confirmed with timing experiments.

The `clock` and `etime` functions are used to poll the computer clock and determine the time required to execute pieces of code. The time calculated is the "elapsed" time. During this time, the computer not only has been running MATLAB code, but also has been executing background jobs and housekeeping functions. (Housekeeping functions are automated computer activities such as cleaning up computer disk storage or data compression tasks.) The `tic` and `toc` functions perform a similar task. Either the `tic/toc` or `clock/etime` functions can be used to compare execution time for different code options. The `timeit` function is a more accurate method of determining the amount of time required to execute computer code; however, it requires the code to be stored as a function.

MATLAB SUMMARY

Commands and Functions	
`break`	causes the execution of a loop to be terminated
`clock`	determines the current time on the CPU clock
`continue`	terminates the current pass through a loop, but proceeds to the next pass
`end`	identifies the end of a control structure
`error`	terminates the program and sends a message to the screen
`etime`	finds elapsed time
`for`	generates a loop structure
`ones`	creates an array of ones
`tic`	starts a timing sequence
`timeit`	finds time required to execute a function
`toc`	stops a timing sequence
`while`	generates a loop structure

KEY TERMS

converge
diverge
`for` loop
housekeeping
infinite loop
initialize
loop
repetition
selection structures
sequences
midpoint break loop
nested loop
`while` loop
series
vectorization

PROBLEMS

9.1 Use a `for` loop to sum the elements in the following vector:

$$x = [1,\ 23,\ 43,\ 72,\ 87,\ 56,\ 98,\ 33]$$

Check your answer with the `sum` function.

9.2 Repeat the previous problem, this time using a `while` loop.

9.3 Use a `for` loop to create a vector of the squares of the numbers 1 through 5.

9.4 Use a `while` loop to create a vector of the squares of the numbers 1 through 5.

9.5 Use the `primes` function to create a list of all the primes below 100. Now use a `for` loop to multiply adjacent values together. For example, the first four prime numbers are

$$2 \quad 3 \quad 5 \quad 7$$

Your calculation would be

$$2 * 3 \quad 3 * 5 \quad 5 * 7$$

which gives

$$6 \quad 15 \quad 35$$



Figure P9.6
Chambered nautilus. (Joao Virissimo/Shutterstock)

9.6 A Fibonacci sequence is composed of elements created by adding the two previous elements. The simplest Fibonacci sequence starts with 1, 1, and proceeds as follows:

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ \ldots$$

However, a Fibonacci sequence can be created with any two starting numbers. Fibonacci sequences appear regularly in nature. For example, the shell of the chambered nautilus (see Figure P9.6) grows in accordance with a Fibonacci sequence.

Prompt the user to enter the first two numbers in a Fibonacci sequence, and the total number of elements requested for the sequence. Find the sequence and store it in an array by using a `for` loop. Now plot your results on a `polarplot` graph. Use the element number for the angle, and the value of the element in the sequence for the radius.

9.7 Repeat the preceding problem, this time using a `while` loop.

9.8 One interesting property of a Fibonacci sequence is that the ratio of the values of adjacent members of the sequence approaches a number called "the golden ratio" or Φ (phi). Create a program that accepts the first two numbers of a Fibonacci sequence as user input, then calculates additional values in the sequence until the ratio of adjacent values converges to within a difference of 0.001. You can do this in a `while` loop by comparing the ratio of element `k` to element `k - 1`, and the ratio of element `k - 1` to element `k - 2`. If you call your sequence `x`, then the code for the `while` statement is

```
while abs(x(k)/x(k-1) - x(k-1)/x(k-2)) > 0.001
```

9.9 Recall from trigonometry that the tangent of both $\pi/2$ and $-\pi/2$ is infinity. This may be seen from the fact that

$$\tan(\theta) = \sin(\theta)/\cos(\theta)$$

and since

$$\sin(\pi/2) = 1$$

and

$$\cos(\pi/2) = 0$$

it follows that

$$\tan(\pi/2) = \text{infinity}$$

Because MATLAB uses a floating-point approximation of π, it calculates the tangent of $\pi/2$ as a very large number, but not infinity.

Prompt the user to enter an angle θ between $\pi/2$ and $-\pi/2$, inclusive. If it is between $\pi/2$ and $-\pi/2$, but not equal to either of those values, calculate $\tan(\theta)$ and display the result in the command window. If it is equal to $\pi/2$ or $-\pi/2$, set the result equal to `Inf` and display the result in the command window. If it is outside the specified range, send the user an error message in the command window and prompt the user to enter another value. Continue prompting the user for a new value of theta until they enter a valid number.

9.10 Imagine that you are a proud new parent. You decide to start a college savings plan now for your child, hoping to have enough in 18 years to pay the sharply rising cost of education. Suppose a generous relative gives you \$1000 to get started and that each month you can contribute \$100. Suppose also that the interest rate is 6% per year compounded monthly, which is equivalent to 0.5% each month.

Because of interest payments and your contribution, each month your balance will increase in accordance with the formula

$$\text{new balance} = \text{old balance} + \text{interest} + \text{your contribution}$$

Use a `for` loop to find the amount in the savings account each month for the next 18 years. (Create a vector of values.) Plot the amount in the account as a function of time. (Plot time on the horizontal axis, and dollars on the vertical axis.)

9.11 Imagine that you have a crystal ball and can predict the percentage increases in tuition for the next 22 years. The following vector `increase` shows your predictions, in percent, for each year:

```
increase = [10, 8, 10, 16, 15, 4, 6, 7, 8, 10, 8, 12,
   14, 15, 8, 7, 6, 5, 7, 8, 9, 8]
```

Use a `for` loop to determine the cost of a four-year education, assuming that the current cost for one year at a local school is \$9000.

9.12 Use an `if` statement to compare your results from the previous two problems. Are you saving enough? Send an appropriate message to the command window.

9.13 Edmond Halley (the astronomer famous for discovering Halley's comet) invented a fast algorithm for computing the square root of a number, A. Halley's algorithm approximates $\sqrt{A}$ as follows:

Start with an initial guess x_1. The new approximation is then given by

$$Y_n = \frac{1}{A} x_n{}^2$$

$$x_{n+1} = \frac{x_n}{8}(15 - y_n(10 - 3y_n))$$

These two calculations are repeated until some convergence criterion, ε, is met.

$$|x_{n+1} - x_n| \leq \varepsilon$$

Write a MATLAB function called `my_sqrt` that approximates the square root of a number. It should have two inputs, the initial guess and the convergence criterion.

Test your function by approximating the square root of 5 and comparing it to the value calculated with the built-in MATLAB function, `sqrt`.

9.14 The value of $\cos(x)$ can be approximated using a Maclaurin series

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

which can be expressed more compactly as

$$\sum_{k=1}^{\infty} (-1)^{k-1} \frac{x^{(k-1)*2}}{((k-1)*2)!}$$

(recall that the symbol ! stands for factorial).

Use a midpoint break loop to determine how many terms must be included in the summation, in order to find the correct value of $\cos(2)$ within an error of 0.001. Limit the number of iterations to a maximum of 10.

9.15 The value of $\sin(x)$ can be approximated as

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Create a function called `my_sin`, using a midpoint break loop to approximate the value of $\sin(x)$. Determine convergence by comparing successive values of the summation as you add additional terms. These successive sums should be within an absolute value of 0.001 of each other. Test your function by evaluating the `my_sin(2)` and comparing it to the built-in MATLAB sine function.

9.16 The value of π can be approximated with the Gregory–Leibniz series summation

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots$$

or

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \ldots$$

The series converges very slowly. Calculate π, using a midpoint break loop. Determine convergence by comparing successive values of the summation as you add additional terms until the difference between successive sums is less than 0.001. Set the maximum number of iterations to 3000.

9.17 A store owner asks you to write a program for use in the checkout process. The program should:

- Prompt the user to enter the cost of the first item.
- Continue to prompt for additional items, until the user enters 0.
- Display the total.
- Prompt for the amount the customer submits as payment.
- Display the change due.

Nested Loops

9.18 In the previous chapter, the Lake Powell water elevation data were evaluated using the `find` function and logical indexing. Repeat the calculations, using

a nested loop structure. The file *lake_powell.dat* contains the data listed in Table 8.12.

Faster Loops

9.19 Whenever possible, it is better to avoid using loops, because they are slow to execute.

(a) Generate a 100,000-item vector of random digits called `x`; square each element in this vector and name the result `y`; use the commands `tic` and `toc` to time the operation.

(b) Next, perform the same operation element by element in a `for` loop. Before you start, clear the values in your variables with

```
clear x y
```

Use `tic` and `toc` to time the operation.

Depending on how fast your computer runs, you may need to stop the calculations by issuing the `Ctrl c` command in the command window.

(c) Now convince yourself that suppressing the printing of intermediate answers will speed up execution of the code by allowing these same operations to run and print the answers as they are calculated. You will almost undoubtedly need to cancel the execution of this loop because of the large amount of time it takes. Recall that `Ctrl c` terminates the program.

(d) If you are going to use a constant value several times in a `for` loop, calculate it once and store it, rather than calculating it each time through the loop. Demonstrate the increase in speed of this process by adding `(sin(0.3) + cos(pi/3))*5!` to every value in the long vector in a `for` loop. (Recall that ! means factorial, which can be calculated with the MATLAB function `factorial`.)

(e) As discussed in this chapter, if MATLAB must increase the size of a vector every time through a loop, the process will take more time than if the vector were already the appropriate size. Demonstrate this fact by repeating part (b) of this problem. Create the following vector of *y*-values, in which every element is equal to zero before you enter the `for` loop:

```
y = zeros(1,100000);
```

You will be replacing the zeros one at a time as you repeat the calculations in the loop.

(f) Using `tic/toc` or `clock/etime` is an easy way to compare coding strategies; however, the times reported include the 'housekeeping' background tasks performed by the computer. The `timeit` function returns just the time required to execute the actual code but requires that the code be packaged as a function. Create an anonymous function with no input to calculate the square of `x`, and use it as the input to the `timeit` function.

```
f=@() x.^2
timeit(f)
```

CHALLENGE PROBLEMS

9.20 **(a)** Create a function called `polygon` that draws a polygon in a polar plot. Your function should have a single input parameter—the number of sides.

(b) Use a `for` loop to create a figure with four subplots, (tiles), showing a triangle in the first tile, a square in the second tile, a pentagon in the third tile, and a hexagon in the fourth tile. You should use the function you created in part (a) to draw each polygon. Use the index parameter from the `for` loop to specify the tile in which each polygon is drawn, and in an expression to determine the number of sides used as input to the `polygon` function.

9.21 Consider the following method to approximate the mathematical constant, *e*. Start by generating *K* uniform random integers between 1 and *K*. Compute *J*, the number of integers between 1 and *K*, which were never generated. We then approximate *e* by the ratio

$$\frac{K}{J}$$

Consider the following example for $K = 5$. Assume that the following five integers are randomly generated between 1 and 5.

1 1 2 3 2

The number of times the integers are generated is given by

Integers	1	2	3	4	5
Number of instances	2	2	1	0	0

In this example, there are two integers, namely 4 and 5, which were never generated. This means that $J = 2$. Consequently, *e* is approximated by

$$\frac{5}{2} = 2.5$$

Write a function called `eapprox` that takes the value of *K* as input, and which then approximates *e* using the method described above. Test your function several times with different values of *K*, and compare the result to the value of *e* calculated using the built-in MATLAB function.

exp(1)

Use the randi function to create an array of random integers.

9.22 Vectorize (replace loops with a single statement) the calculations in the function created in the previous problem, by using the built-in MATLAB functions `histcounts` and `sum`.

9.23 Revisit Example 9.2, which describes a function to calculate factorials. The function directs the user to enter an integer but doesn't check to confirm that the input is a whole number. The answer is the same for `fact(4)` and `fact(4.3)`. Modify the code using a midpoint break loop and a message that prompts the user to enter an integer. Give the user 3 tries, and if they still enter an illegal value exit the function with an error message produced by the `error` function.

CHAPTER

10 Matrix Algebra

Objectives

After reading this chapter, you should be able to:

- Perform the basic operations of matrix algebra.
- Explain the difference between array operations and matrix operations.
- Use dot products in MATLAB to solve engineering problems.
- Use matrix multiplication in MATLAB to solve engineering problems.
- Solve systems of linear equations using multiple built-in MATLAB techniques.
- Use some of MATLAB's special matrices to demonstrate matrix algebra concepts.

INTRODUCTION

KEY IDEA
All matrices are arrays, but not all arrays are matrices.

The terms **array** and **matrix** are often used interchangeably in engineering. However, technically, an array is an orderly grouping of information, whereas a matrix is a two-dimensional numeric array used in linear algebra. Arrays can contain numeric information, but they can also contain character data, symbolic data, and other types of information. Thus, not all arrays are matrices. Only those upon which you intend to perform linear transformations meet the strict definition of a matrix. In this chapter, we'll assume that the numeric arrays we define will be used for linear algebra calculations, so they can be called matrices.

Matrix algebra is used extensively in engineering applications. The mathematics of matrix algebra is first introduced in college algebra courses, and is extended in linear algebra courses and courses in differential equations. Engineering students start using matrix algebra regularly in statics and dynamics classes.

10.1 MATRIX OPERATIONS AND FUNCTIONS

In this chapter, MATLAB functions and operators are introduced that are intended specifically for use in matrix algebra. These functions and operators are contrasted with

MATLAB's array functions and operators, from which they differ significantly. Much of this material may be a review, but is included for completeness.

10.1.1 Transpose

The `transpose` operator changes the rows of a matrix into columns and the columns into rows. In mathematics texts, you will often see the **transpose** indicated with superscript T (as in A^T). Don't confuse this notation with MATLAB syntax. In MATLAB, the transpose operator is a single quote (`'`), so that the transpose of matrix `A` is `A'`.

Consider the following matrix and its transpose:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ \boxed{7} & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}, \quad A^T = \begin{bmatrix} 1 & 4 & \boxed{7} & 10 \\ 2 & 3 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix}$$

The rows and columns have been switched. Notice that the value in position (3, 1) of A has now moved to position (1, 3) of A^T, and the value in position (4, 2) of A has now moved to position (2, 4) of A^T. In general, the row and column subscripts (index numbers) are interchanged to form the transpose.

In MATLAB, one of the most common uses of the transpose operation is to change row vectors into column vectors. For example:

```
A = [1 2 3];
A'
```

returns

```
A = 1
    2
    3
```

KEY IDEA
The terms array and matrix are often used interchangeably, although they are technically different.

When used with complex numbers, the transpose operation returns the complex conjugate. For example, we may define a vector of negative numbers, take the square root, and then transpose the resulting matrix of complex numbers. Thus, the code

```
x = [-1:-1:-3]
```

returns

```
x =
  -1  -2  -3
```

Then, taking the square root

```
y = sqrt(x)
y =
  0 + 1.0000i   0 + 1.4142i   0 + 1.7321i
```

and finally transposing `y`

```
y'
```

gives

```
ans =
  0 - 1.0000i
  0 - 1.4142i
  0 - 1.7321i
```

Notice that the results (`y'`) are the complex conjugates of the elements in `y`.

10.1.2 Dot Product

The **dot product** (sometimes called the scalar product) is the sum of the results obtained when you multiply two vectors together, element by element. Consider the following two vectors:

```
A = [ 1 2 3];
B = [ 4 5 6];
```

The result of the array multiplication of these two vectors is

```
y = A.*B
y =
    4   10   18
```

If you add the elements up, you get the dot product:

```
sum(y)
ans =
      32
```

A mathematics text would represent the dot product as

$$\sum_{i=1}^{n} A_i \cdot B_i$$

which we could write in MATLAB as

```
sum(A.*B)
```

MATLAB includes a function called `dot` to compute the dot product:

```
dot(A,B)
ans =
     32
```

It doesn't matter whether `A` and `B` are row or column vectors, just as long as they have the same number of elements.

The dot product finds wide use in engineering applications, such as in calculating the center of gravity (see Example 10.1) and in carrying out vector algebra calculations (see Example 10.2).

HINT

With dot products, it doesn't matter if both the vectors are rows, both are columns, or one is a row and the other a column. It also doesn't matter what order you use to perform the process: The result of `dot(A,B)` is the same as that of `dot(B,A)`. This isn't true for most matrix operations.

EXAMPLE 10.1

CALCULATING THE CENTER OF MASS

The mass of a space vehicle is an extremely important quantity. Whole groups of people in the design process keep track of the location and mass of every nut and bolt. Not only is the total mass of the vehicle important, but information about mass is also used to determine the center of mass of the vehicle. One reason the center of mass is important

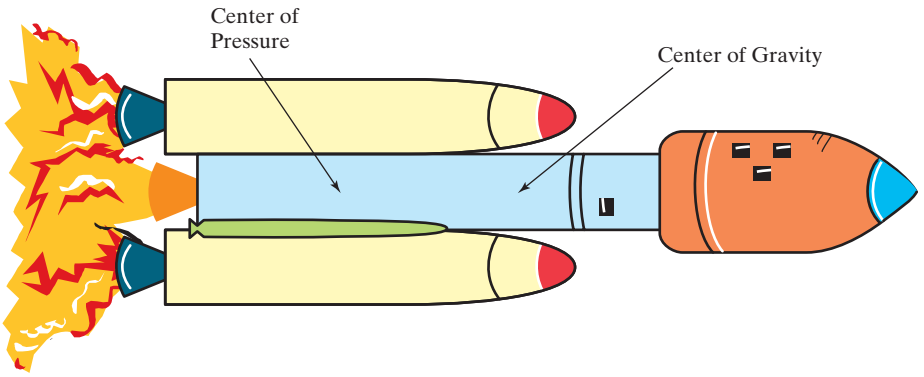


Figure 10.1
The center of pressure needs to be behind the center of gravity for stable flight. For many cases, the center of mass and the center of gravity are the same.

is that rockets tumble if the center of pressure is forward of the center of gravity, which is directly related to the center of mass. (see Figure 10.1). You can demonstrate the importance of the center of gravity to flight characteristics with a paper airplane. Put a paper clip on the nose of the paper airplane and observe how the flight pattern changes.

Although finding the center of mass is a fairly straightforward calculation, it becomes more complicated when you realize that both the mass of the vehicle *and* the distribution of mass change as the fuel is burned.

The location of the center of mass can be calculated by dividing the vehicle into small components. In a rectangular coordinate system,

$$\begin{aligned} \bar{x}W &= x_1W_1 + x_2W_2 + x_3W_3 \\ \bar{y}W &= y_1W_1 + y_2W_2 + y_3W_3 \\ \bar{z}W &= z_1W_1 + z_2W_2 + z_3W_3 \end{aligned}$$

where

$\bar{x}_1$, $\bar{x}_2$, and $\bar{z}$ are the coordinates of the center of mass,
W is the total mass of the system,
x_1, x_2, x_3, . . . are the x-coordinates of system components 1, 2, 3,…, respectively,
y_1, y_2, y_3, . . . are the y-coordinates of system components 1, 2, 3,…, respectively,
z_1, z_2, z_3, . . . are the z-coordinates of system components 1, 2, 3,…, respectively, and
W_1, W_2, W_3, . . . are the masses of system components 1, 2, 3,…, respectively.

In this example, we will find the center of mass of a small collection of the components used in a complicated space vehicle (see Table 10.1). We can formulate this problem in terms of the dot product.

Table 10.1 Vehicle Component Locations and Mass

Item	x (m)	y (m)	z (m)	Mass (g)
Bolt	0.1	2.0	3.0	3.50
Screw	1.0	1.0	1.0	1.50
Nut	1.5	0.2	0.5	0.79
Bracket	2.0	2.0	4.0	1.75

(*continued*)

1. State the problem
   Find the center of mass of the space vehicle.
2. Describe the input and output

   Input Location of each component in an $x - y - z$ coordinate system
   Mass of each component

   Output Location of the center of mass of the vehicle
3. Develop a hand example
   The x-coordinate of the center of mass is equal to

$$\overline{x} = \frac{\sum_{i=1}^{3} x_i m_i}{m_{\text{Total}}} = \frac{\sum_{i=1}^{3} x_i m_i}{\sum_{i=1}^{3} m_i}$$

   so, from Table 10.2,

$$\overline{x} = \frac{6.535}{7.54} = 0.8667 \text{ m}$$

   Notice that the summation of the products of the x-coordinates and the corresponding masses could be expressed as a dot product.
4. Develop a MATLAB solution
   The MATLAB code

```
%% Example 10.1
% Calculating the Center of Mass
mass = [3.5, 1.5, 0.79, 1.75];
x = [0.1, 1, 1.5, 2];
x_bar = dot(x,mass)/sum(mass)
y = [2, 1, 0.2, 2];
y_bar = dot(y,mass)/sum(mass)
z = [3, 1, 0.5, 4];
z_bar = dot(z,mass)/sum(mass)
```

returns the following result:

```
x_bar =
  0.8667
y_bar =
  1.6125
z_bar =
  2.5723
```

Table 10.2 Finding the *x*-Coordinate of the Center of Mass

Item	*x* (m)		Mass (g)	*x* × *m* (gm)
Bolt	0.1	×	3.50	= 0.35
Screw	1.0	×	1.50	= 1.50
Nut	1.5	×	0.79	= 1.1850
Bracket	2.0	×	1.75	= 3.50
Sum			7.54	6.535

5. Test the solution
Compare the MATLAB solution with the hand solution. The *x*-coordinate appears to be correct, so the *y*- and *z*-coordinates are probably correct, too. Plotting the results would also help us evaluate them:

```
%% Plot the results
plot3 (x,y,z,"o",x_bar,y_bar,z_bar,"s")
grid on
xlabel("x-axis"),ylabel("y-axis"),zlabel("z-axis")
title("Center of Mass")
axis([0,2,0,2,0,4])
```

The resulting plot is shown in Figure 10.2.
Now that we know the program works, we can use it for any number of items. The program will be the same for three components as for 3000.

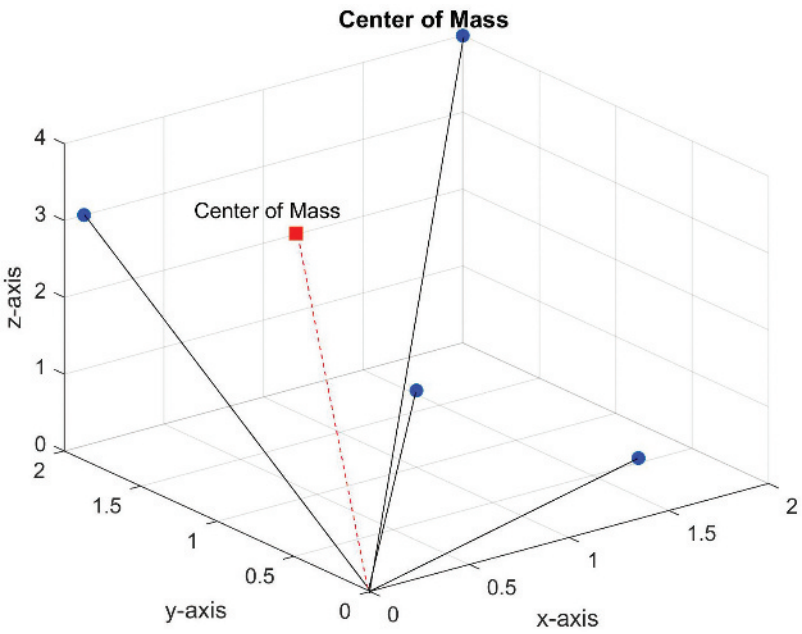


Figure 10.2
Center of mass of some sample data. This plot was enhanced with the use of MATLAB's interactive plotting tools.

EXAMPLE 10.2

FORCE VECTORS

Statics is the study of forces in systems that don't move (and hence are *static*). These forces are usually described as vectors. If you add up the vectors, you can determine the total force on an object. Consider the two force vectors *A* and *B* shown in Figure 10.3.

(*continued*)

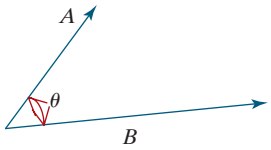


Figure 10.3
Force vectors are used in the study of both statics and dynamics.

Each has a magnitude and a direction. One typical notation would show these vectors as $\vec{A}$ and $\vec{B}$ but would represent the magnitude of each (their physical length) as A and B. The vectors could also be represented in terms of their magnitudes along the x-, y-, and z-axes, multiplied by a **unit vector** $(\vec{i}, \vec{j}, \vec{k})$ Then

$$\vec{A} = A_x\vec{i} + A_y\vec{j} + A_z\vec{k}$$

and

$$\vec{B} = B_x\vec{i} + B_y\vec{j} + B_z\vec{k}$$

The dot product of $\vec{A}$ and $\vec{B}$ is equal to the magnitude of $\vec{A}$ times the magnitude of $\vec{B}$ times the cosine of the angle between them:

$$\vec{A} \cdot \vec{B} = AB\cos(\theta)$$

Finding the magnitude of a vector involves using the Pythagorean theorem. In the case of three dimensions,

$$A = \sqrt{A_x^2 + A_y^2 + A_z^2}$$

We can use MATLAB to solve problems like this if we define the vector A as

```
A = [Ax Ay Az]
```

where `Ax`, `Ay`, and `Az` are the component magnitudes in the x-, y-, and z-directions, respectively. For example, use the dot product to find the angle between the following two force vectors:

$$\vec{A} = 5\vec{i} + 6\vec{j} + 3\vec{k}$$
$$\vec{B} = 1\vec{i} + 3\vec{j} + 2\vec{k}$$

1. State the problem
   Find the angle between two force vectors.

2. Describe the input and output

   Input $\vec{A} = 5\vec{i} + 6\vec{j} + 3\vec{k}$
   $\vec{B} = 1\vec{i} + 3\vec{j} + 2\vec{k}$

   Output θ, the angle between the two vectors

3. Develop a hand example

$$\vec{A} \cdot \vec{B} = 5 \cdot 1 + 6 \cdot 3 + 3 \cdot 2 = 29$$
$$A = \sqrt{5^2 + 6^2 + 3^2} = 8.37$$
$$B = \sqrt{1^2 + 3^2 + 2^2} = 3.74$$
$$\cos(\theta) = \vec{A} \cdot \vec{B} \,/\, AB = 0.9264$$
$$\cos^{-1}(0.9264) = 0.386$$

   Thus, the angle between the two vectors is 0.386 radians, or 22.12 degrees.

4. Develop a MATLAB solution
   The MATLAB code

```
%% Example 10.2
% Find the angle between two force vectors
% Define the vectors
```

```
A = [5 6 3];
B = [1 3 2];
%% Calculate the magnitude of each vector
mag_A = sqrt(sum(A.^2));
mag_B = sqrt(sum(B.^2));
%% Calculate the cosine of the angle
cos_theta = dot(A,B)/(mag_A*mag_B);
%% Calculate the angle
theta = acos(cos_theta);
%% Report the results
fprintf("The angle between the vectors is \n")
fprintf("%4.3f radians \n",theta)
fprintf("or %6.2f degrees \n",theta*180/pi)
```

generates the following interaction.

```
The angle between the vectors is
  0.386 radians or 22.12 degrees
```

5. Test the solution

In this case, we just reproduced the hand solution in MATLAB. However, doing so gives us confidence in our solution process. We could expand our problem to allow the user to enter any pair of vectors. Consider this example:

```
%% Example 10.2 - expanded
% Finding the angle between two force vectors
% Define the vectors
disp("Component magnitudes should be entered")
disp("Using matrix notation, i.e.")
disp("[ A B C]")
A = input("Enter the x y z component magnitudes of vector A:   ")
B = input("Enter the x y z component magnitudes of vector B:   ")
%% Calculate the magnitude of each vector
mag_A = sqrt(sum(A.^2));
mag_B = sqrt(sum(B.^2));
% Calculate the cosine of the angle
cos_theta = dot(A,B)/(mag_A*mag_B);
% Calculate the angle
theta = acos(cos_theta);
%% Report the results
fprintf("The angle between the vectors is \n")
fprintf("%4.3f radians \n",theta)
fprintf("or %6.2f degrees \n",theta*180/pi)
```

gives the following interaction.

```
Component magnitudes should be entered
Using matrix notation, i.e.
 [ A B C]
```

(*continued*)

```
Enter the x y z component magnitudes of vector A: [1 2 3]
A =
    1   2   3
Enter the x y z component magnitudes of vector B: [4 5 6]
B =
    4   5   6
The angle between the vectors is
0.226 radians
or 12.93 degrees
```

PRACTICE EXERCISE 10.1

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Use the `dot` function to find the dot product of the following vectors:

$$\vec{A} = [1\ 2\ 3\ 4];$$

$$\vec{B} = [12\ 20\ 15\ 7]$$

2. Find the dot product of $\vec{A}$ and $\vec{B}$ by summing the array products of $\vec{A}$ and $\vec{B}$ (`sum(A.*B)`)
3. A group of friends went to a local fast-food establishment. They ordered four hamburgers at $3.99 each, three soft drinks at $1.49 each, one milk shake at $2.50, two orders of fries at $0.99 each, and two orders of onion rings at $1.29. Use the dot product to determine the bill.

10.1.3 Matrix Multiplication

Matrix multiplication is similar to the dot product. If you define

```
A = [1 2 3]
B = [ 3;
      4;
      5]
```

then

```
A*B
ans =
 26
```

KEY IDEA
Matrix multiplication results in an array in which each element is a dot product.

gives the same result as

```
dot(A,B)
ans  =
 26
```

Matrix multiplication results in an array in which each element is a dot product. The preceding example is just the simplest case. In general, the results are found by

taking the dot product of each row in matrix A with each column in matrix B. For example, if

```
A = [ 1 2 3;
      4 5 6 ]
```

and

```
B = [ 10 20 30;
      40 50 60;
      70 80 90 ]
```

then the first element of the resulting matrix is the dot product of row 1 in matrix A and column 1 in matrix B, the second element is the dot product of row 1 in matrix A and column 2 in matrix B, and so on. Once the dot product is found for the first row in matrix A with all the columns in matrix B, we start over again with row 2 in matrix A. Thus,

```
C = A*B
```

returns

```
C =
  300 360 420
  660 810 960
```

Consider the result in row 2, column 2, of the matrix C. We can call this result C(2,2). It is the dot product of row 2 of matrix A and column 2 of matrix B:

```
dot(A(2,:), B(:,2))
ans =
    810
```

KEY IDEA

Matrix multiplication is not commutative. The order of operation is very important.

We could express this relationship in mathematical notation (instead of MATLAB syntax) as

$$C_{i,j} = \sum_{k=1}^{N} A_{i,k} B_{k,j}$$

Because matrix multiplication is a series of dot products, the number of columns in matrix A must equal the number of rows in matrix B. If matrix A is an $m \times n$ matrix, matrix B must be $n \times p$, and the results will be an $m \times p$ matrix. In this example, A is a 2×3 matrix and B is a 3×3 matrix. The result is a 2×3 matrix.

One way to visualize this set of rules is to write the sizes of the two matrices next to each other, in the order of their operation. In this example, we have

$$2 \times \underbrace{3 \quad 3} \times 3$$

The two inner numbers must match, and the two outer numbers determine the size of the resulting matrix.

Matrix multiplication is not in general **commutative**, which means that

$$\mathrm{A} \times \mathrm{B} \neq \mathrm{B} \times \mathrm{A}$$

We can see this in our example. When we reverse the order of the matrices, we have

$$3 \times \underbrace{3 \quad 2} \times 3$$

and it is no longer possible to take the dot product of the columns in the first matrix and the rows in the second matrix. If both matrices are square, we can indeed

calculate an answer for `A * B` and an answer for `B * A`, but the answers are not the same. Consider this example:

```
A = [1 2 3
     4 5 6
     7 8 9];
B = [2 3 4
     5 6 7
     8 9 10];
A*B
ans  =
  36   42   48
  81   96   111
 126   150  174
B*A
ans  =
  42   51   60
  78   96   114
 114   141  168
```

EXAMPLE 10.3

USING MATRIX MULTIPLICATION TO FIND THE CENTER OF MASS

In Example 10.1, we used the dot product to find the center of mass of a space vehicle. We could also use matrix multiplication to do the calculation in one step, instead of calculating each coordinate separately. Table 10.1 is repeated in this example for clarity.

1. State the problem
   Find the center of mass of the space vehicle.
2. Describe the input and output

   Input Location of each component in an $x - y - z$ coordinate system
   Mass of each component

   Output Location of the center of gravity of the vehicle
3. Develop a hand example
   We can create a two-dimensional matrix containing all the information about the coordinates and a corresponding one-dimensional matrix containing information about the mass. If there are n components, the coordinate information should be in a $3 \times n$ matrix, and the masses should be in an $n \times 1$ matrix. The

Table 10.1 Vehicle Component Locations and Mass

Item	x (m)	y (m)	z (m)	Mass (g)
Bolt	0.1	2.0	3.0	3.50
Screw	1.0	1.0	1.0	1.50
Nut	1.5	0.2	0.5	0.79
Bracket	2.0	2.0	4.0	1.75

result would then be a 3×1 matrix representing the $x - y - z$ coordinates of the center of gravity times the total mass.

4. Develop a MATLAB solution
The MATLAB code

```
%% Example 10.3
coord = [0.1    2      3
         1      1      1
         1.5    0.2    0.5
         2      2      4 ]';
mass = [3.5, 1.5, 0.79, 1.75]';
location = coord*mass/sum(mass)
```

reports the following results:

```
location =
  0.8667
  1.6125
  2.5723
```

5. Test the solution
The results are the same as those in Example 10.1.

PRACTICE EXERCISE 10.2

Which of the following sets of matrices can be multiplied together? The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. $A = \begin{bmatrix} 2 & 5 \\ 2 & 9 \\ 6 & 5 \end{bmatrix}, B = \begin{bmatrix} 2 & 5 \\ 2 & 9 \\ 6 & 5 \end{bmatrix}$

2. $A = \begin{bmatrix} 2 & 5 \\ 2 & 9 \\ 6 & 5 \end{bmatrix}, B = \begin{bmatrix} 1 & 3 & 12 \\ 5 & 2 & 9 \end{bmatrix}$

3. $A = \begin{bmatrix} 5 & 1 & 9 \\ 7 & 2 & 2 \end{bmatrix}, B = \begin{bmatrix} 8 & 5 \\ 4 & 2 \\ 8 & 9 \end{bmatrix}$

4. $A = \begin{bmatrix} 1 & 9 & 8 \\ 8 & 4 & 7 \\ 2 & 5 & 3 \end{bmatrix}, B = \begin{bmatrix} 7 \\ 1 \\ 5 \end{bmatrix}$

Show that, for each case, $A \cdot B \neq B \cdot A$.

KEY IDEA
A matrix must be square to be raised to a power.

10.1.4 Matrix Powers

Raising a matrix to a power is equivalent to multiplying the matrix by itself the requisite number of times. For example, A^2 is the same as $A \cdot A$, A^3 is the same as $A \cdot A \cdot A$. Recalling that the number of columns in the first matrix of a multiplication must

be equal to the number of rows in the second matrix, we see that in order to raise a matrix to a power, the matrix must be square (have the same number of rows and columns). Consider the matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

If we tried to square this matrix, we would get an error statement because the rows and columns do not match:

KEY IDEA
Array multiplication and matrix multiplication are different operations and yield different results.

$$\underset{\text{rows and columns must match}}{2 \times 3 \quad 2 \times 3}$$

Consider another example. The statement

```
A = randn(3)
```

creates a 3×3 matrix of random numbers, such as

```
A =
  -1.3362  -0.6918  -1.5937
   0.7143   0.8580  -1.4410
   1.6236   1.2540   0.5711
```

HINT

Remember that `randn` produces random numbers, so your computer may produce numbers different from those listed.

If we square this matrix, the result is also a 3×3 matrix:

```
A^2
ans =
  -1.2963  -1.6677   2.2161
  -2.6811  -1.5650  -3.1978
  -0.3463   0.6690  -4.0683
```

Raising a matrix to a noninteger power gives a complex result:

```
A^1.5
ans =
  -1.8446 - 0.0247i   -1.5333 + 0.0153i  -0.3150 - 0.0255i
  -0.7552 + 0.0283i    0.0668 - 0.0176i  -3.0472 + 0.0292i
   1.3359 + 0.0067i    1.5292 - 0.0042i  -1.5313 + 0.0069i
```

Note that raising `A` to the *matrix power* of two is different from raising `A` to the *array power* of two:

```
C = A.^2;
```

Raising `A` to the array power of two produces the following results:

```
C =
  1.7854  0.4786  2.5399
  0.5102  0.7362  2.0765
  2.6361  1.5725  0.3262
```

and is equivalent to squaring each term.

10.1.5 Matrix Inverse

In mathematics, what do we mean when we say "Take the inverse"? For a function, the inverse "undoes" the function, or gets us back to where we started. For example, $\sin^{-1}(x)$ is the inverse function of sin(x). We can demonstrate the relationship in MATLAB:

```
asin(sin(1.5))
```

(Recall that the MATLAB syntax for the inverse sine is `asin`.)

```
ans =
  1.5
```

HINT

Remember that $\sin^{-1}(x)$ does not mean the same thing as $1/\sin(x)$. Most current mathematics texts use the $\sin^{-1}(x)$ notation, but on your calculator and in computer programs, $\sin^{-1}(x)$ is represented as asin(x).

Another example of functions that are inverses is $\ln(x)$ and e^x:

```
log(exp(3))
```

(Recall that the MATLAB syntax for the natural logarithm is log, not ln.)

```
ans =
     3
```

But what does taking the inverse of a number mean? One way to think about it is that if you operated on the number 1 by multiplying it by a number, what could you do to undo this operation and get the number 1 back? You'd need to divide by your number or multiply by 1 over the number. This leads us to the conclusion that $1/x$ and x are inverses, since

KEY IDEA
A function times its inverse is equal to one.

$$\frac{1}{x}x = 1$$

These are *multiplicative* inverses, as opposed to the function inverse we first discussed. (There are also additive inverses, such as $-a$ and a.) Finally, what is the **inverse** of a matrix? It's the matrix you need to multiply by using matrix algebra to get the **identity matrix**. The identity matrix consists of ones down the main diagonal, and zeros in all the other locations:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

KEY IDEA
A matrix times its inverse is equal to the identity matrix.

The inverse operation is one of the few matrix multiplications that is commutative; that is,

$$A^{-1}A = AA^{-1} = \mathrm{I}$$

In order for the preceding statement to be true, matrix A must be square, which leads us to the conclusion that, in order for a matrix to have an inverse, it must be square.

We can demonstrate these concepts in MATLAB by first defining a matrix and then experimenting with its behavior. The "magic matrix," in which the sum of the rows equals the sum of the columns, as well as the sum of each diagonal, is easy to create, so we'll choose it for our experiment:

```
A = magic(3)
A =
 8  1  6
 3  5  7
 4  9  2
```

MATLAB offers two approaches for finding the inverse of a matrix. We could raise `A` to the −1 power with the code

```
A^-1
ans  =
     0.1472 -0.1444  0.0639
    -0.0611  0.0222  0.1056
    -0.0194  0.1889 -0.1028
```

or we could use the built-in function `inv`:

```
inv(A)
ans  =
     0.1472 -0.1444  0.0639
    -0.0611  0.0222  0.1056
    -0.0194  0.1889 -0.1028
```

Using either approach, we can show that multiplying the inverse of `A` by `A` gives the identity matrix:

```
inv(A)*A
ans  =
     1.0000   0       -0.0000
     0        1.0000   0
     0        0.0000   1.0000
```

and

```
A*inv(A)
ans  =
    1.0000  0       -0.0000
   -0.0000  1.0000   0
    0.0000  0        1.0000
```

Notice that some of the values in the result are negative zero. This occurs due to round-off error.

Determining the inverse of a matrix by hand can be difficult, so we'll leave that exercise to a course in matrix mathematics. There are matrices for which an inverse does not exist; these are called **singular matrices** or **ill-conditioned matrices**. When you attempt to compute the inverse of an ill-conditioned matrix in MATLAB, an error message is sent to the command window.

The matrix inverse is widely used in matrix algebra, although from a computational point of view it is rarely the most efficient way to solve a problem. This subject is discussed at length in linear algebra courses.

10.1.6 Determinants

KEY IDEA
If the determinant is zero, the matrix does not have an inverse.

Determinants are used in linear algebra and are related to the matrix inverse. If the determinant of a matrix is 0, the matrix does not have an inverse, and we say that it is singular. Determinants are calculated by multiplying together the elements along

the matrix's left-to-right diagonals and subtracting the product of the right-to-left diagonals. For example, for a 2×2 matrix

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

the determinant is

$$|A| = A_{11}A_{22} - A_{12}A_{21}$$

Thus, for

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$|A| = (1)(4) - (2)(3) = -2$$

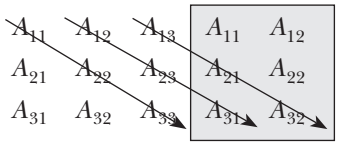
MATLAB has a built-in determinant function, `det`, that will find the determinant for you:

```
A = [1 2;3 4];
det(A)
ans =
    -2
```

Figuring out the diagonals for a 3×3 matrix

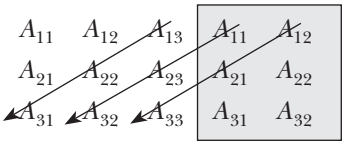
$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

is a bit harder. If you copy the first two columns of the matrix into columns 4 and 5, it becomes easier to see. Multiply each left-to-right diagonal and add them up:



$$(A_{11}A_{22}A_{33}) + (A_{12}A_{23}A_{31}) + (A_{13}A_{21}A_{32})$$

Then multiply each right-to-left diagonal and add them up:



$$(A_{13}A_{22}A_{31}) + (A_{11}A_{23}A_{32}) + (A_{12}A_{21}A_{33})$$

Finally, subtract the second calculation from the first. For example, we might have

$$|A| = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = (1 \times 5 \times 9) + (2 \times 6 \times 7) + (3 \times 4 \times 8)$$

$$-(3 \times 5 \times 7) - (1 \times 6 \times 8) - (2 \times 4 \times 9) = 225 - 225 = 0$$

Using MATLAB for the same calculation

```
A = [1 2 3;4 5 6;7 8 9];
det(A)
ans =
     0
```

Since we know that matrices with a determinant of zero do not have inverses, let's see what happens when we ask MATLAB to find the inverse of A:

```
inv(A)
Warning: Matrix is close to singular or badly scaled.
   Results may be inaccurate. RCOND = 1.541976e-018.
ans =
  1.0e+016 *
  -0.4504   0.9007  -0.4504
   0.9007  -1.8014   0.9007
  -0.4504   0.9007  -0.4504
```

PRACTICE EXERCISE 10.3

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Find the inverse of the following magic matrices, both by using the `inv` function and by raising the matrix to the -1 power:
   - **a.** magic(3)
   - **b.** magic(4)
   - **c.** magic(5)
4. Find the determinant of each of the matrices in Exercise 1.
5. Consider the following matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$

Would you expect it to be singular or not? (Recall that singular matrices have a determinant of 0 and do not have an inverse.)

10.1.7 Cross Products

KEY IDEA
The result of a cross product is a vector.

Cross products are sometimes called vector products, because, unlike dot products, which return a scalar, the result of a cross product is a vector. The resulting vector is always at right angles **(normal)** to the plane defined by the two input vectors—a property called **orthogonality**.

Consider two vectors in three-space that represent both a direction and a magnitude. (Force is often represented this way.) Mathematically,

$$\vec{A} = A_x\vec{i} + A_y\vec{j} + A_z\vec{k}$$
$$\vec{B} = B_x\vec{i} + B_y\vec{j} + B_z\vec{k}$$

The values A_x, A_y, A_z and B_x, B_y, B_z represent the magnitude of the vector in the x, y, and z directions, respectively. The $\vec{i}, \vec{j}, \vec{k}$ symbols represent unit vectors in the x, y, and z directions. The cross product of $\vec{A}$ and $\vec{B}$, $\vec{A} \times \vec{B}$ is defined as

$$\vec{A} \times \vec{B} = \left(A_y B_z - A_z b_y\right)\vec{i} + \left(A_z B_x - A_x B_z\right)\vec{j} + \left(A_x B_y - A_y B_x\right)\vec{k}$$

You can visualize this operation by creating a table

i	j	k
A_x	A_y	A_z
B_x	B_y	B_z

and then repeating the first two columns at the end of the table:

i	j	k	i	j
A_x	A_y	A_z	A_x	A_y
B_x	B_y	B_z	B_x	B_y

The component of the cross product in the i direction is found by obtaining the product $A_y B_z$ and subtracting the product $A_z B_y$ from it:

$\textcircled{i}$	j	k	i	j
A_x	A_y	A_z	A_x	A_y
B_x	B_y	B_z	B_x	B_y

Moving across the diagram, the component of the cross product in the j direction is found by obtaining the product $A_z B_x$ and subtracting the product $A_x B_z$ from it:

i	$\textcircled{j}$	k	i	j
A_x	A_y	A_z	A_x	A_y
B_x	B_y	B_z	B_x	B_y

Finally, the component of the cross product in the k direction is found by obtaining the product $A_x B_y$ and subtracting the product $A_y B_x$ from it:

i	j	$\textcircled{k}$	i	j
A_x	A_y	A_z	A_x	A_y
B_x	B_y	B_z	B_x	B_y

HINT

You may have noticed that the cross product is just a special case of a determinant whose first row is composed of unit vectors.

In MATLAB, the cross product is found using the function `cross`, which requires two inputs: the vectors `A` and `B`. Each of these MATLAB vectors must have three elements, since they represent the vector components in three-space. For example, we might have

```
A = [1 2 3];
```
(which represents $\vec{A} = 1\vec{i} + 2\vec{j} + 3\vec{k}$)

```
B = [4 5 6];
```
(which represents $\vec{B} = 4\vec{i} + 5\vec{j} + 6\vec{k}$)

```
cross(A,B)
ans =
  -3 6 -3
```
(which represents $\vec{C} = -3\vec{i} + 6\vec{j} + 3\vec{k}$)

Consider two vectors in the $x - y$ plane (with no z component):

```
A = [1 2 0]
B = [3 4 0]
```

The magnitude of these vectors in the z direction needs to be specified as zero in MATLAB.

The result of the cross product must be at right angles to the plane that contains the vectors `A` and `B`, which tells us that in this case it must be straight out of the $x - y$ plane, with only a z component.

```
cross(A,B)
ans  =
     0     0    -2
```

Cross products find wide use in statics, dynamics, fluid mechanics, and electrical engineering problems.

EXAMPLE 10.4

MOMENT OF A FORCE ABOUT A POINT

The moment of a force about a point is found by computing the cross product of a vector that defines the position of the force with respect to a point, with the force vector:

$$M_0 = r \times F$$

Consider the force applied at the end of a lever, as shown in Figure 10.4. If you apply a force to the lever close to the pivot point, the effect is different than if you apply a force further out on the lever. That effect is called the moment.

Calculate the moment about the pivot point on a lever for a force described as the vector

$$\vec{F} = -100\vec{i} + 20\vec{j} + 0\vec{k}$$

Assume that the lever is 12 inches long at an angle of 45 degrees from the horizontal. This means that the position vector can be represented as

$$\vec{r} = \frac{12}{\sqrt{2}}\vec{i} + \frac{12}{\sqrt{2}}\vec{j} + 0\vec{k}$$

1. State the problem
   Find the moment of a force vector about the pivot point of a lever.

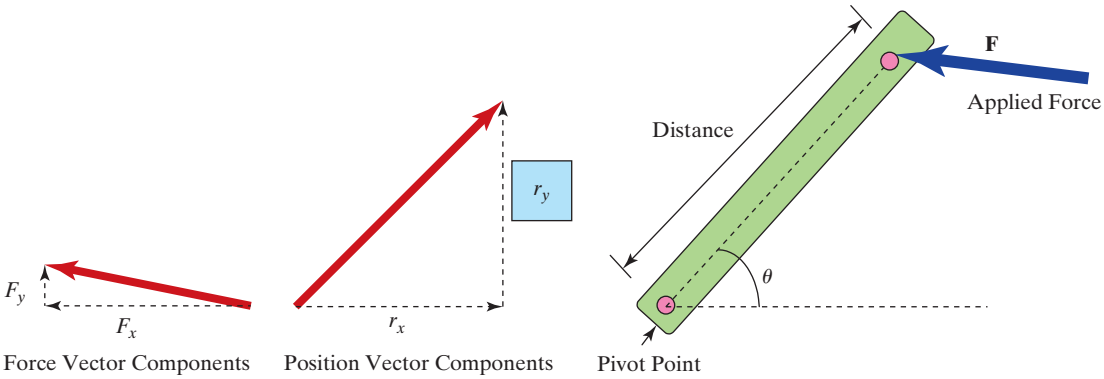


Figure 10.4
The force applied to a lever creates a moment about the pivot point.

2. Describe the input and output

Input position vector $\vec{r} = \frac{12}{\sqrt{2}}\vec{i} + \frac{12}{\sqrt{2}}\vec{j} + 0\vec{k}$

force vector $\vec{F} = -100\vec{i} + 20\vec{j} + 0\vec{k}$

Output Moment about the pivot point of the lever

3. Develop a hand example

Visualize the problem as the determinant of a 3×3 array:

$$M_0 = \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \\ \frac{12}{\sqrt{2}} & \frac{12}{\sqrt{2}} & 0 \\ -100 & 20 & 0 \end{bmatrix}$$

Since there are no $\vec{k}$ components in the input, there can be no $\vec{i}$ or $\vec{j}$ component in the answer. The moment must be

$$M_0 = \left(\frac{12}{\sqrt{2}} \times 20 - \frac{12}{\sqrt{2}} \times (-100)\right) \times \vec{k} = 1018.23\vec{k}$$

4. Develop a MATLAB solution

The MATLAB code

```
%% Example 10.4
% Moment about a pivot point
% Define the position vector
r = [12/sqrt(2), 12/sqrt(2), 0];
%% Define the force vector
F = [-100, 20, 0];
%% Calculate the moment
moment=cross(r,F)
```

returns the following result:

```
moment  =
  0     0    1018.23
```

This corresponds to a moment vector

$$M_0 = 0\vec{j} + 0\vec{j} + 1018.23\vec{k}$$

Notice that the moment is at right angles to the plane defined by the position and force vectors.

5. Test the solution

The hand and MATLAB solutions match, which means that we can now expand the program to a more general solution. For example, the following program prompts the user for the *x*, *y*, and *z* components of the position and force vectors, then calculates the moment:

```
% Example 10.4
% Moment about a pivot point
```

(*continued*)

```
% Define the position vector
  clear, clc
rx=input("Enter the x component of the position vector: ");
ry=input("Enter the y component of the position vector: ");
rz=input("Enter the z component of the position vector: ");
r = [rx, ry, rz];
  disp("The position vector is")
 fprintf("%8.2f i + %8.2f j  + %8.2f k  ft\n",r)
% Define the force vector
Fx=input("Enter the x component of the force vector: ");
Fy=input("Enter the y component of the force vector: ");
Fz=input("Enter the z component of the force vector: ");
F = [Fx, Fy, Fz];
  disp("The force vector is")
  fprintf("%8.2f i + %8.2f j  + %8.2f k lbf\n",F)
  moment=cross(r,F); % Calculate the moment
  fprintf("The moment vector about the pivot point is \n")
  fprintf("%8.2f i + %8.2f j  + %8.2f k ft-lbf\n",moment)
```

A sample interaction in the command window is

```
Enter the x component of the position vector: 2
Enter the y component of the position vector: 3
Enter the z component of the position vector: 4
The position vector is
  2.00 i +  3.00 j +  4.00 k ft
Enter the x component of the force vector: 20
Enter the y component of the force vector: 10
Enter the z component of the force vector: 30
The force vector is
  20.00 i +  10.00 j +  30.00 k lbf
The moment vector about the pivot point is
  50.00 i +  20.00 j +  -40.00 k ft-lbf
```

10.1.8 Identity Matrix

An identity matrix is a matrix with ones on the main diagonal and zeros everywhere else. For example, here is an identity matrix with four rows and four columns:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note that the main diagonal contains elements in which the row number is the same as the column number. The indices for elements on the main diagonal are (1, 1), (2, 2), (3, 3), etc.

In MATLAB, identity matrices can be generated with the `eye` function. The arguments of the `eye` function are similar to those of the `zeros` and the `ones` functions. If the argument of the function is a scalar, as in `eye(6)`, the function will generate a square matrix, using the argument as both the number of rows and the number of columns.

```
A = eye(3)
A =
 1 0 0
 0 1 0
 0 0 1
```

Recall that `A * inv(A)` equals the identity, as illustrated with the following statements:

HINT

Do not name an identity matrix *i*, because *i* is used to represent imaginary numbers.

```
A = [1,0,2; -1, 4, -2; 5,2,1]
A =
   1   0   2
  -1   4  -2
   5   2   1
inv(A)
ans =
   -0.2222  -0.1111   0.2222
    0.2500   0.2500   0.0000
    0.6111   0.0556  -0.1111
A*inv(A)
ans =
    1.0000   0         0.0000
   -0.0000   1.0000    0.0000
   -0.0000  -0.0000    1.0000
```

As discussed earlier, matrix multiplication is not in general commutative—that is,

$$AB \neq BA$$

However, for identity matrices,

$$AI = IA$$

as shown with the following MATLAB code:

```
I = eye(3)
I =
   1   0   0
   0   1   0
   0   0   1
A*I
ans =
    1   0   2
   -1   4  -2
    5   2   1
I*A
ans =
    1   0   2
   -1   4  -2
    5   2   1
```

10.2 SOLUTIONS OF SYSTEMS OF LINEAR EQUATIONS

Consider the following system of three equations with three unknowns:

$$\begin{aligned} 3x + 2y - z &= 10 \\ -x + 3y + 2z &= 5 \\ x - y - z &= -1 \end{aligned}$$

We can rewrite this system of equations by using the following matrices:

$$A = \begin{bmatrix} 3 & 2 & -1 \\ -1 & 3 & 2 \\ 1 & -1 & -1 \end{bmatrix}, \; X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \; B = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix}$$

Using matrix multiplication, we can then write the **system of equations** as

$$AX = B$$

10.2.1 Solution Using the Matrix Inverse

Probably the most straightforward way of solving this system of equations is to use the matrix inverse. Since we know that

$$\boldsymbol{A}^{-1}\boldsymbol{A} = \mathrm{I}$$

we can multiply both sides of the matrix equation $AX = B$ by $\boldsymbol{A}^{-1}$ to get

$$A^{-1}AX = A^{-1}B$$

giving

$$X = A^{-1}B$$

As in all matrix mathematics, the order of multiplication is important. Since A is a 3×3 matrix, its inverse $\boldsymbol{A}^{-1}$ is also a 3×3 matrix. The multiplication $A^{-1}B$

$$3 \times \underbrace{3 \qquad 3} \times 1$$

works because the dimensions match up. The result is the 3×1 matrix X. If we change the order to BA^{-1} the dimensions would no longer match, and the operation would be impossible.

KEY IDEA

The matrix inverse technique for solving systems of linear equations is straight forward in concept, but is inefficient and is prone to round-off error.

Since, in MATLAB, the matrix inverse is computed with the `inv` function, we can use the following set of commands to solve this problem:

```
A = [3 2 -1; -1 3 2; 1 -1 -1];
B = [10; 5; -1];
X = inv(A)*B
```

This code returns

```
X =
  -2.0000
   5.0000
  -6.0000
```

Alternatively, you could represent the matrix inverse as `A^-1`, so that

```
X = A^-1*B
```

which gives the same result.

```
X =
   -2.0000
    5.0000
   -6.0000
```

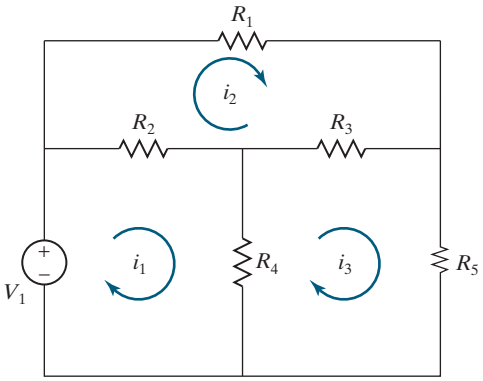
Although this technique corresponds well with the approach taught in college algebra classes when matrices are introduced, it is not very efficient and can result in excessive round-off errors. In general, using the matrix inverse to solve linear systems of equations should be avoided.

EXAMPLE 10.5

SOLVING SIMULTANEOUS EQUATIONS: AN ELECTRICAL CIRCUIT*

In solving an electrical circuit problem, one quickly finds oneself mired in a large number of simultaneous equations. For example, consider the electrical circuit shown in Figure 10.5.

Figure 10.5
An electrical circuit.



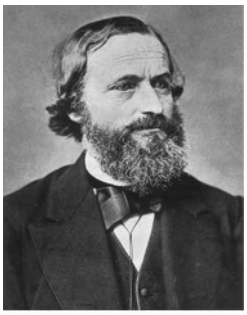


Figure 10.6
Gustav Kirchhoff was a German physicist, who formulated many of the basic laws of circuit theory. (Bilwissedition Ltd. & Co. KG /Alamy Stock Photo)

It contains a single voltage source and five resistors. You can analyze this circuit by dividing it up into smaller pieces and using two basic facts about electricity:

$\sum$*voltage* around a circuit must be zero (Kirchhoff's second law—see Figure 10.6)

$$\text{voltage} = \text{current} \times \text{resistance} (V = iR)$$

Following the lower left-hand loop results in our first equation:

$$-V_1 + R_2(i_1 - i_2) + R_4(i_1 - i_3) = 0$$

Following the upper loop results in our second equation:

$$R_1 i_2 + R_3(i_2 - i_3) + R_2(i_2 - i_1) = 0$$

Finally, following the lower right-hand loop results in the last equation:

$$R_3(i_3 - i_2) + R_5 i_3 + R_4(i_3 - i_1) = 0$$

Since all the resistances (the R values) and the voltage are known, we have three equations and three unknowns. Now we need to rearrange the equations so that

*From Etter, Kuncicky, and Moore *Introduction to MATLAB 7* (Upper Saddle River, NJ: Pearson Prentice Hall, 2005).

(*continued*)

they are in a form to which we can apply a matrix solution. In other words, we need to isolate the i variables:

$$\begin{aligned}(R_2 + R_4)i_1 + \quad & (-R_2)i_2 + \quad & (-R_4)i_3 &= V_1\\ (-R_2)i_1 + (R_1 + R_2 + R_3)i_2 + \quad & & (-R_3)i_3 &= 0\\ (-R_4)i_1 + \quad & (-R_3)i_2 + (R_3 + R_4 + R_5)i_3 & &= 0\end{aligned}$$

Create a MATLAB program to solve these equations, using the matrix inverse method. Allow the user to enter the five values of R and the voltage from the keyboard.

1. State the problem
   Find the three currents for the circuit shown.

2. Describe the input and output

   Input Five resistances R_1, R_2, R_3, R_4, R_5, and the voltage V, provided from the keyboard

   Output Three current values i_1, i_2, i_3

3. Develop a hand example
   If there is no applied voltage in a circuit, there can be no current, so if we enter any value for the resistances and enter zero for the voltage, the answer should be zero.

4. Develop a MATLAB solution
   The MATLAB code

```
%% Example 10.5
% Finding Currents
clear, clc
R1 = input("Input the value of R1: ");
R2 = input("Input the value of R2: ");
R3 = input("Input the value of R3: ");
R4 = input("Input the value of R4: ");
R5 = input("Input the value of R5: ");
V = input("Input the value of voltage, V: ");
coef = [(R2 + R4), -R2, -R4;
     -R2, (R1 + R2 +R3), (-R3);
     -R4, -R3, (R3 + R4 + R5)];
 result = [V; 0; 0];
I = inv(coef)*result
```

   generates the following interaction in the command window:

```
Input the value of R1: 5
Input the value of R2: 5
Input the value of R3: 5
Input the value of R4: 5
Input the value of R5: 5
Input the value of voltage, V: 0
I =
       0
       0
       0
```

5. Test the solution
We purposely chose to enter a voltage of zero in order to check our solution. Circuits without a driving force (voltage) cannot have a current flowing through them. Now, try the program with other values:

```
Input the value of R1: 2
Input the value of R2: 4
Input the value of R3: 6
Input the value of R4: 8
Input the value of R5: 10
Input the value of voltage, V: 10
```

Together, these values give

```
I =
    1.6935
    0.96774
    0.80645
```



Figure 10.7
Carl Friedrich Gauss was a remarkable mathematician and contributed to many other fields as well, including physics, astronomy, and electricity. (Georgios Kollidas/Alamy Stock Photo)

10.2.2 Solution Using Matrix Left Division

A better technique for solving a system of linear equations is **Gaussian elimination**. This is probably the way you learned to solve systems of equations in college algebra. Gaussian elimination was developed by Carl Friedrich Gauss, a German mathematician and scientist (see Figure 10.7).

Consider the problem of three equations in x, y, and z:

$$\begin{aligned} 3x &+2y &-z &= 10 \\ -x &+3y &+2z &= 5 \\ x &-y &-z &= -1 \end{aligned}$$

To solve this problem by hand, we would first consider the first two equations in the set and eliminate one of the variables—for example, x. To do this, multiply the second equation by 3, then add the resulting equation to the first one:

$$\begin{array}{rrrcr} 3x & +2y & -z & = & 10 \\ -3x & +9y & +6z & = & 15 \\ \hline 0 & +11y & +5z & = & 25 \end{array}$$

Now, repeat the process for the second and third equations:

$$\begin{array}{rrrcr} -x & +3y & +2z & = & 5 \\ x & -y & -z & = & -1 \\ \hline 0 & +2y & +z & = & 4 \end{array}$$

At this point, we've eliminated one variable and reduced the problem to two equations and two unknowns:

$$\begin{aligned} 11y &+5z &= 25 \\ 2y &+z &= 4 \end{aligned}$$

Now, we can repeat the elimination process by multiplying row 3 by $-11/2$:

$$\begin{array}{rrcr} 11y & +5z & = & 25 \\ -\frac{11}{2}*2y & \frac{11}{2}z & = & -\frac{11}{2}*4 \\ \hline 0 & -\frac{1}{2}z & = & 3 \end{array}$$

Finally, we can solve for z:

$$z = -6$$

Once we know the value of z, we can substitute back into either of the two equations in just z and y—namely,

$$\begin{array}{rr} 11y & +5z = 25 \\ 2y & +z = 4 \end{array}$$

to find that

$$y = 5$$

The last step is to substitute back into one of the original equations,

$$\begin{array}{rrrcr} 3x & +2y & -z & = & 10 \\ -x & +3y & +2z & = & 5 \\ x & -y & -z & = & -1 \end{array}$$

to find that

$$x = -2$$

KEY IDEA
Gaussian elimination is more efficient and less susceptible to round-off error than the matrix inverse method.

The technique of Gaussian elimination is an organized approach to eliminating variables until only one unknown exists and then substituting back until all the unknowns are determined. In MATLAB, we can use left division to solve the problem by Gaussian elimination. Thus,

```
X = A\B
```

returns

```
X =
  -2.0000
   5.0000
  -6.0000
```

This is the same result we obtained with the hand solution and the matrix inverse approach.

Over-defined and Under-defined Systems

MATLAB is also capable of solving problems which are either over-defined or under-defined using left division. Consider, for example, the following problem:

$$\begin{array}{rrrcr} 3*x & +2*y & +5*z & = & 22 \\ 4*x & +5*y & -2*z & = & 8 \\ x & +y & +z & = & 6 \end{array}$$

This problem is appropriately defined with three equations and three unknowns. When formulated as

```
A = [3   2   5
     4   5   -2
     1   1   1]
```

and

```
B = [22; 8; 6],
```

the left division operator can be used to solve for *x*, *y*, and *z*

```
X = A\B
```

which results in the solution

```
X =
  1
  2
  3
```

Suppose, however, that we knew four equations relating *x*, *y*, and *z*, such as

```
3*x +2*y +5*z =  22
4*x +5*y -2*z =   8
  x   +y   +z =   6
2*x -4*y -7*z = -27
```

Now, we have four equations and three unknowns, and the problem is over-defined. We can still solve it using the left division operator. The coefficient matrix is defined as

```
A = [3  2  5
     4  5 -2
     1  1  1
     2 -4 -7]
```

and the result matrix as

```
B = [22; 8; 6; -27]
```

When we execute the statement

```
X = A\B
```

we get the same result, because the equations were consistent.

```
X =
  1
  2
  3
```

However, it is possible when gathering data that there might be small errors that result in different numbers in the result matrix. Assume that instead the fourth equation tells us that the result is −28, instead of −27. This means that we'll need to adjust the `B` vector

```
B = [22; 8; 6; -28]
```

Now, when we execute

```
X = A\B
```

the result is

```
X =
  0.8618
  2.1234
  3.0328
```

MATLAB uses a least squared approach to find the set of X values (which correspond to x, y, z in our equations), which is the best match to the equations. If we use these values to find B

```
A*X
```

KEY IDEA
The left division operator uses a least squared approach to solve over-defined systems of linear equations.

The result is

```
ans =
   21.9962
    7.9982
    6.0180
  -27.9997
```

The least squared approach minimizes the absolute value of the difference between the calculated B values and the actual B values. This approach is described in a later chapter on numerical methods.

What if your system of equations is underdefined? For example, what if we only had two equations for three unknowns?

```
3*x +2*y +5*z = 22
4*x +5*y -2*z = 8
```

In this case, we'd define the coefficient matrix as

```
A = [3  2  5
     4  5 -2]
```

and the result matrix as

```
B = [22; 8]
```

MATLAB solves the problem by setting the first variable equal to 0, which effectively reduces the problem to two equations and two unknowns.

```
X = A\B
```

which results in

```
X =
  0
  2.8966
  3.2414
```

This is only one of an infinite number of possible solutions, but it does give the correct answer if we substitute back into our equation

```
A*X
ans =
    22.0000
    8.0000
```

10.2.3 Solution Using the Reverse Row Echelon Function

In a manner similar to left division, we could solve the system of linear equations

$$\begin{aligned} 3x &+2y &-z &= 10 \\ -x &+3y &+2z &= 5 \\ x &-y &-z &= -1 \end{aligned}$$

using the reduced row echelon function, `rref`. This function uses the Gauss–Jordan elimination technique.

Recall that we can rewrite this system of equations by using the following matrices:

$$A = \begin{bmatrix} 3 & 2 & -1 \\ -1 & 3 & 2 \\ 1 & -1 & -1 \end{bmatrix}, \; X = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \; B = \begin{bmatrix} 10 \\ 5 \\ -1 \end{bmatrix}$$

The `rref` function requires an expanded matrix as input, representing the coefficients and results. For our example system of equations the input would be

```
C = [A,B]
C =
   3   2  -1  10
  -1   3   2   5
   1  -1  -1  -1
rref(C)
ans =
   1   0   0  -2
   0   1   0   5
   0   0   1  -6
```

The solution to our problem is represented by the last column in the output array, and corresponds to the results achieved with the other methods.

In a simple problem like this, no matter which technique we use, round-off error and execution time are not big factors. However, some numerical techniques require the solution of matrices with thousands or even millions of elements. Execution times are measured in hours or days for these problems, and round-off error and execution time become critical considerations. For such problems, the matrix inverse technique is not appropriate.

EXAMPLE 10.6

MATERIAL BALANCES ON A DESALINATION UNIT: SOLVING SIMULTANEOUS EQUATIONS

Freshwater is a scarce resource in many parts of the world. For example, Israel supports a modern industrial society in the middle of a desert. To supplement local water sources, Israel depends on water desalination plants along the Mediterranean coast. Modern desalination plants use reverse osmosis, the process used in kidney dialysis! Chemical engineers make wide use of material-balance calculations to design and analyze plants such as the water desalination plants in Israel.

Consider the hypothetical desalination unit shown in Figure 10.8. The salty water flowing into the unit contains 4 wt% salt and 96 wt% water. Inside the unit, the water is separated into two streams by a series of reverse-osmosis operations. The stream flowing out the top is almost pure water. The remaining concentrated solution of salty water is 10 wt% salt and 90 wt% water. Calculate the mass flow rates coming out of the top and bottom of the desalination unit.

This problem requires us to perform a material balance on the reactor for both the salt and the water. The amount of any component flowing into the reactor must be the same as the amount of that component flowing out in the two exit streams. That is,

$$m_{\text{inA}} = m_{\text{topsA}} + m_{\text{bottomsA}}$$

(*continued*)

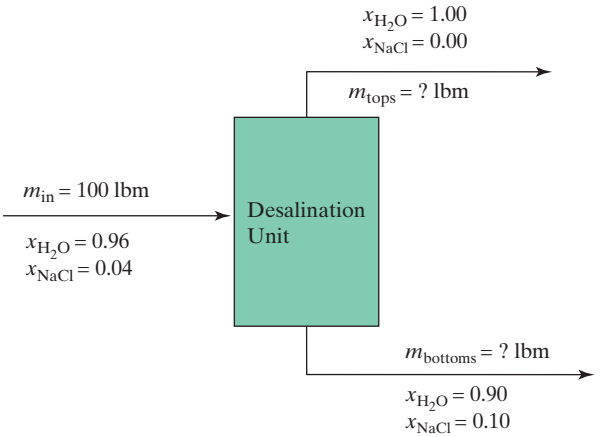


Figure 10.8
Water desalination is an important source of freshwater for desert nations such as Israel.

which could be rewritten as

$$x_A m_{\text{in total}} = x_{\text{Atops}} m_{\text{tops}} + x_{\text{Abottoms}} m_{\text{bottoms}}$$

Thus, we can formulate this problem as a system of two equations in two unknowns:

$$0.96 \times 100 = 1.00 m_{\text{tops}} + 0.90 m_{\text{bottoms}} \quad \text{(for water)}$$
$$0.04 \times 100 = 0.00 m_{\text{tops}} + 0.10 m_{\text{bottoms}} \quad \text{(for salt)}$$

1. State the problem
   Find the mass of freshwater produced and the mass of brine rejected from the desalination unit.

2. Describe the input and output

   Input Mass of 100 lb into the system
   Concentrations (mass fractions) of the input stream:

   $$x_{\text{H2O}} = 0.96$$
   $$x_{\text{Nacl}} = 0.04$$

   Concentrations (mass fractions) in the output streams:
   water-rich stream (tops)

   $$x_{\text{H2O}} = 1.00$$

   brine (bottoms)

   $$x_{\text{H2O}} = 0.90$$
   $$x_{\text{Nacl}} = 0.10$$

   Output Mass out of the water-rich stream (tops)
   Mass out of the brine (bottoms)

3. Develop a hand example
   Since salt (NaCl) is present only in one of the outlet streams, it is easy to solve the following system of equations:

   $$(0.96)(100) = 1.00 m_{\text{tops}} + 0.90 m_{\text{bottoms}} \quad \text{(for water)}$$
   $$(0.04)(100) = 0.00 m_{\text{tops}} + 0.10 m_{\text{bottoms}} \quad \text{(for salt)}$$

Starting with the salt material balance, we find that

$$4 = 0.1m_{\text{bottoms}}$$
$$m_{\text{bottoms}} = 40 \text{ lbm}$$

Once we know the value of m_{bottoms} we can substitute back into the water balance:

$$96 = 1m_{\text{tops}} + (0.90)(40)$$
$$m_{\text{tops}} = 60 \text{ lb}$$

4. Develop a MATLAB solution
We can use matrix mathematics to solve this problem, once we realize it is of the form

$$AX = B$$

where A is the coefficient matrix and thus the mass fractions of the water and salt. The result matrix, B, consists of the mass flow rate into the system of water and salt:

$$A = \begin{bmatrix} 1 & 0.9 \\ 0 & 0.1 \end{bmatrix}, \; B = \begin{bmatrix} 96 \\ 4 \end{bmatrix}$$

The matrix of unknowns, X, consists of the total mass flow rates out of the top and bottom of the desalination unit. Using MATLAB to solve this system of equations requires only three lines of code:

```
A = [1, 0.9; 0, 0.1];
B = [96; 4];
X = A\B
```

This code returns

```
X =
  60
  40
```

5. Test the solution
Notice that in this example we chose to use matrix left division. Using the matrix inverse approach gives the same result:

```
X = inv(A)*B
X =
  60
  40
```

The results from both approaches match that from the hand example, but one additional check can be made to verify the results. We performed material balances based on water and on salt, but an additional balance can be performed on the *total* mass in and out of the system:

$$m_{in} = m_{\text{tops}} + m_{\text{bottoms}}$$
$$m_{in} = 40 + 60 = 100$$

(*continued*)

Verifying that 100 lbm actually exits the system serves as one more confirmation that we performed the calculations correctly.

Although it was easy to solve the system of equations in this problem by hand, most real material-balance calculations include more process streams and more components. Matrix solutions such as the one we created are an important tool for chemical-process engineers.

EXAMPLE 10.7

A FORCE BALANCE ON A STATICALLY DETERMINATE TRUSS

A statically determinate truss is one of the early problems addressed in sophomore Statics classes. A typical problem is shown in Figure 10.9.

At the hinge (point 2), the truss cannot move in either the *x*- or the *y*-direction. At the roller (point 3), movement is allowed in the *x*-direction, but not in the *y*-direction. This results in reactive forces at point 2 in both the *x*- and the *y*-directions, and at point 3 in just the *y*-direction. If we also separate the applied force (at point 1) into *x*- and *y*-components, the freebody diagram can be draw as shown in Figure 10.10.

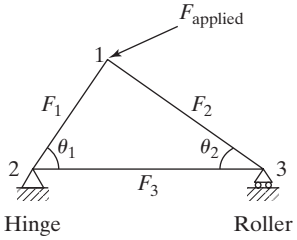


Figure 10.9
A simple statically determinate truss.

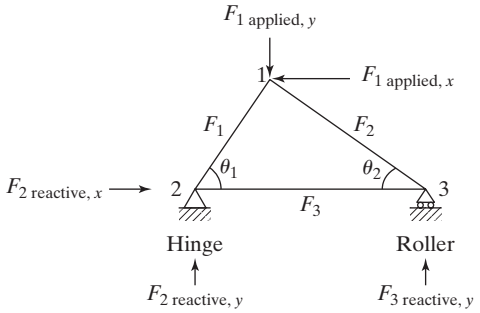


Figure 10.10
Freebody diagram for a statically determinate truss.

Because we assume that the truss is not moving, the sum of the forces at each of the nodes (1, 2, and 3) must be zero in both the *x*- and the *y*-directions. This gives us a total of six equations.

$$\sum F_{\text{at node1, } x \text{ direction}} = 0 = -F_1\cos(\theta_1) + F_2\cos(\theta_2) + F_{1\ \text{applied, } x}$$

$$\sum F_{\text{at node1, } y \text{ direction}} = 0 = -F_1\sin(\theta_1) - F_2\sin(\theta_2) + F_{1\ \text{applied, } y}$$

$$\sum F_{\text{at node2, } x \text{ direction}} = 0 = +F_{2\ \text{reactive,}x} + F_1\cos(\theta_1) + F_3$$

$$\sum F_{\text{at node2, } y \text{ direction}} = 0 = +F_{2\ \text{reactive,}y} + F_1\sin(\theta)$$

$$\sum F_{\text{at node3, } x \text{ direction}} = 0 = -F_2\cos(\theta_2) + F_3$$

$$\sum F_{\text{at node3, } y \text{ direction}} = 0 = -F_2\sin(\theta_2) - F_{3\ \text{reactive, } y}$$

If the applied force is known, as well as the angles, this results in six equations and six unknowns (F_1, F_2, F_3, $F_{2\ \text{reactive, } x}$, $F_{2\ \text{reactive, } y}$, and $F_{3\ \text{reactive, } y}$). It turns out that with a little rearranging, we can see that this is a linear system of equations.

$$
\begin{aligned}
-\cos(\theta_1) * F_1 + \cos(\theta_2) * F_2 + 0 * F_3 + 0 * F_{2\text{ reactive, }x} + 0 * F_{2\text{ reactive, }y} + 0 * F_{3\text{ reactive, }y} &= -F_{1\text{ applied, }x}\\
-\sin(\theta_1) * F_1 - \sin(\theta_2) * F_2 + 0 * F_3 + 0 * F_{2\text{ reactive, }x} + 0 * F_{2\text{ reactive, }y} + 0 * F_{3\text{ reactive, }y} &= -F_{1\text{ applied, }y}\\
\cos(\theta_1) * F_1 + 0 * F_2 + 1 * F_3 + 1 * F_{2\text{ reactive, }x} + 0 * F_{2\text{ reactive, }y} + 0 * F_{3\text{ reactive, }y} &= 0\\
\sin(\theta_1) * F_1 + 0 * F_2 + 0 * F_3 + 0 * F_{2\text{ reactive, }x} + 1 * F_{2\text{ reactive, }y} + 0 * F_{3\text{ reactive, }y} &= 0\\
+ 0 * F_1 - \cos(\theta_2) * F_2 - 1 * F_3 + 0 * F_{2\text{ reactive, }x} + 0 * F_{2\text{ reactive, }y} + 1 * F_{3\text{ reactive, }y} &= 0\\
+ 0 * F_1 + \sin(\theta_2) * F_2 + 0 * F_3 + 0 * F_{2\text{ reactive, }x} + 0 * F_{2\text{ reactive, }y} + 1 * F_{3\text{ reactive, }y} &= 0
\end{aligned}
$$

This system can be expressed, using matrix notation as

$$
\begin{vmatrix}
-\cos(\theta_1) & \cos(\theta_2) & 0 & 0 & 0 & 0\\
-\sin(\theta_1) & -\sin(\theta_2) & 0 & 0 & 0 & 0\\
\cos(\theta_1) & 0 & 1 & 1 & 0 & 0\\
\sin(\theta_1) & 0 & 0 & 0 & 1 & 0\\
0 & -\cos(\theta_2) & -1 & 0 & 0 & 0\\
0 & \sin(\theta_2) & 0 & 0 & 0 & 1
\end{vmatrix}
*
\begin{vmatrix}
F_1\\ F_2\\ F_3\\ F_{2\text{ reactive, }x}\\ F_{2\text{ reactive, }y}\\ F_{3\text{ reactive, }y}
\end{vmatrix}
=
\begin{vmatrix}
-F_{1\text{ applied, }x}\\ -F_{1\text{ applied, }y}\\ 0\\ 0\\ 0\\ 0
\end{vmatrix}
$$

Now that we've derived the appropriate equations, solve this system for the case where

$$
\begin{aligned}
\theta_1 &= 45^\circ,\\
\theta_2 &= 45^\circ
\end{aligned}
$$

and the applied load at node 1 is 1000 lbf in the negative vertical direction.

1. State the problem
   Find the loads experienced on the truss, shown in Figure 10.10.

2. Describe the input and output

   Input Negative vertical load at node 1 of 1000 lbf

   $$
   \begin{aligned}
   \theta_1 &= 45^\circ\\
   \theta_2 &= 45^\circ
   \end{aligned}
   $$

   Output Force experienced in each beam of the truss, F_1, F_2, and F_3, the reactive forces at the hinge, $F_{2\text{ reactive, }x}$ and $F_{2\text{ reactive,}y}$, and the reactive force at the roller, $F_{3\text{ reactive,}y}$.

3. Develop a hand example
   Substituting into the matrix previously derived gives

$$
\begin{vmatrix}
-0.7071 & +0.7071 & 0 & 0 & 0 & 0\\
-0.7071 & -0.7071 & 0 & 0 & 0 & 0\\
+0.7071 & 0 & 1 & 1 & 0 & 0\\
+0.7071 & 0 & 0 & 0 & 1 & 0\\
0 & -0.7071 & -1 & 0 & 0 & 0\\
0 & +0.7071 & 0 & 0 & 0 & 1
\end{vmatrix}
*
\begin{vmatrix}
F_1\\ F_2\\ F_3\\ F_{2\text{ reactive, }x}\\ F_{2\text{ reactive, }y}\\ F_{3\text{ reactive, }y}
\end{vmatrix}
=
\begin{vmatrix}
0\\ 1000\\ 0\\ 0\\ 0\\ 0
\end{vmatrix}
$$

(*continued*)

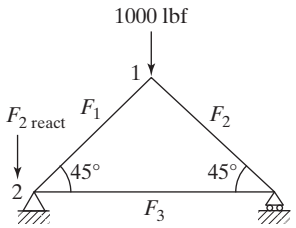


Figure 10.11
Freebody diagram for a balanced truss.

We could solve this equation using matrix algebra; however, an examination of the truss in Figure 10.11 leads to a simpler solution. Notice that there is no horizontal applied force. The reactive force resulting at node 2 must therefore be zero. Because the geometry of the truss is symmetrical that also leads to the conclusion that nodes 2 and 3 must also experience the same load—hence, in order for the net vertical force to equal zero $F_{2\ \text{reactive},y}$ and $F_{3\ \text{reactive},y}$, must both be 500 lbf. We've now determined three of the unknowns,

$$\begin{aligned} F_{2\ \text{reactive},x} &= 0 \\ F_{2\ \text{reactive},y} &= 500\ \text{lbf} \\ F_{3\ \text{reactive},y} &= 500\ \text{lbf} \end{aligned}$$

Examining the set of equations, we notice that the force balance in the vertical direction at node 2 can now be solved:

$$\sum F_{\text{atnode2, } y \text{ direction}} = 0 = +F_{2\text{reactive, } y} + F_1 \sin(\theta_1)$$

$$\sum F_{\text{atnode2, } y \text{ direction}} = 0 = +500 + F1 \sin(45°)$$

$$F_1 = \frac{-500}{\sin(45°)} = -707.1\ \text{lbf similarly…}$$

$$F_2 = -707.1\ \text{lbf}$$

Finally, we can use the balance at node 3 in the horizontal direction to give

$$\sum F_{\text{atnode3, } x \text{ direction}} = 0 = -F_2 \cos(\theta_2) - F_3$$

$$F_3 = -F_2 \cos(\theta_2) = 707.1 * \cos(45°) = 500\ \text{lbf}$$

4. Develop a MATLAB Solution
We can develop a general solution to this problem, and use the given data to check it.

```
theta1=45  % angle in degrees
theta2=45  % angle in degrees
F1x=0   % horizontal load
F1y=-1000 % vertical load
A= [-cosd(theta1),cosd(theta2),0,0,0,0
    -sind(theta1),-sind(theta2),0,0,0,0
    cosd(theta1),0,1,1,0,0
    sind(theta1),0,0,0,1,0
    0,-cosd(theta2),-1,0,0,0
    0,sind(theta2),0,0,0,1]
B=[F1x,-F1y,0,0,0,0]'
x=(A\B)' % use left division
```

This code returns the result

```
x =
  -707.11  -707.11  500.00  0  500.00  500.00
```

which corresponds to the hand solution.

5. Test the solution
Notice that in this example we chose to use matrix left division. Using the matrix inverse approach gives the same result:

```
x = ( inv(A)*B)'
```

returns the following to the command window

```
x =
  -707.11  -707.11  500.00  0  500.00  500.00
```

The results from both approaches match that from the hand example, which did not depend on matrix algebra. Now, we can use the same program to analyze the truss at different conditions. For example, assume the following:

$$\theta_1 = 30^\circ$$
$$\theta_2 = 60^\circ$$

and an applied load of 1000 lbf in the horizontal direction at node 1. The MATLAB code would be modified to read:

```
theta1=30 % angle in degrees
theta2=60 % angle in degrees
F1x=1000 % horizontal load
F1y=0 % vertical load
A= [-cosd(theta1),cosd(theta2),0,0,0,0
-sind(theta1),-sind(theta2),0,0,0,0
cosd(theta1),0,1,1,0,0
sind(theta1),0,0,0,1,0
0,-cosd(theta2),-1,0,0,0
0,sind(theta2),0,0,0,1]
B= [F1x,-F1y,0,0,0,0]'
x=inv(A)*B
x=A\B
```

giving a result of

```
x =
 -866.03  500.00  -250.00  1000.00  433.01  -433.01
```

Notice that the fourth value in the array, which corresponds to the reactive force in the *x*-direction at node 2 is 1000, just what we would expect.

10.3 SPECIAL MATRICES

MATLAB includes a number of matrices that are useful for testing numerical techniques, that serve in computational algorithms, or that are just interesting.

`pascal` Creates a Pascal matrix, using Pascal's triangle.

```
pascal(4)
ans =
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
```

`magic` Creates a Magic Matrix, in which all the rows, all the columns, and all the diagonals add up to the same value.

```
magic(3)
ans =
     8     1     6
     3     5     7
     4     9     2
```

`rosser`	The Rosser Matrix is used as an eigenvalue test matrix. It requires no input.	**rosser** **ans** = [**8** × **8**]
`gallery`	The gallery contains over 50 different test matrices.	The syntax for the gallery functions is different for each function. Use `help` to determine which is right for your needs.

SUMMARY

One of the most common matrix operations is the transpose, which changes rows into columns and columns into rows. In mathematics texts, the transpose is indicated with a superscript T, as in A^T. In MATLAB, the single quote is used as the transpose operator. Thus,

```
A'
```

is the transpose of `A`.

Another common matrix operation is the dot product, which is the sum of the array multiplications of two equal-size vectors:

$$C = \sum_{i=1}^{N} A_i \ * \ B_i$$

The MATLAB function for dot products is

```
dot(A,B)
```

Similar to the dot product is matrix multiplication. Each element in the result of a matrix multiplication is a dot product:

$$C_{i,j} = \sum_{k=1}^{N} A_{i,k} B_{k,j}$$

Matrix multiplication uses the asterisk operator in MATLAB, so that

```
C = A*B
```

indicates that matrix `A` is multiplied by matrix `B` in accordance with the rules of matrix algebra. Matrix multiplication is not commutative—that is,

$$AB \neq BA$$

Raising a matrix to a power is equivalent to multiple multiplication steps:

$$A^3 = AAA$$

Since a matrix must be square in order to be multiplied by itself, only square matrices can be raised to a power. When matrices are raised to noninteger powers in MATLAB, the result is a matrix of complex numbers.

A matrix times its inverse is the identity matrix:

$$AA^{-1} = I$$

MATLAB provides two techniques for determining a matrix inverse: the `inv` function,

```
inv_of_A = inv(A)
```

and raising the matrix to the -1 power, given by

```
inv_of_A = A^-1
```

If the determinant of a matrix is zero, the matrix is singular and does not have an inverse. The MATLAB function used to find the determinant is

```
det(A)
```

In addition to computing dot products, MATLAB contains a function that calculates the cross product of two vectors in three-space. The cross product is often called the vector product because it returns a vector:

$$C = A \times B$$

The cross product produces a vector that is at right angles (normal) to the two input vectors, a property called orthogonality. Cross products can be thought of as the determinant of a matrix composed of the unit vectors in the x, y, and z directions and the two input vectors:

$$C = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ A_x & A_y & A_z \\ B_x & B_y & B_z \end{vmatrix}$$

The MATLAB syntax for calculating a cross product uses the `cross` function:

```
C = cross(A,B)
```

One common use of the matrix inverse is to solve systems of linear equations. For example, the system

$$\begin{aligned} 3x + 2y - z &= 10 \\ -x + 3y + 2z &= 5 \\ x - y - z &= -1 \end{aligned}$$

can be expressed with matrices as

$$AX = B$$

To solve this system of equations with MATLAB, multiply `B` by the inverse of `A`:

```
X = inv(A)*B
```

This technique is less efficient than Gaussian elimination, which is accomplished in MATLAB using left division:

```
X = A\B
```

The left division technique can also be used to solve both over-defined and under-defined systems of equations. When the system is over-defined, a least squared approach is used to find the best fit result. When the system is under-defined, one or more of the variables is set equal to 0, and the remaining variables calculated.

MATLAB includes a number of special matrices that can be used to make calculations easier or to test numerical techniques. The `pascal` and `magic` functions are used to create Pascal matrices and magic matrices, respectively, which have no particular computational use but are interesting mathematically. The gallery function contains over 50 matrices especially formulated to test numerical techniques.

MATLAB SUMMARY

The following MATLAB summary lists and briefly describes special characters, commands, and functions that are defined in this chapter:

Special Characters	
'	indicates a matrix transpose
*	matrix multiplication
\	matrix left division
^	matrix exponentiation

Commands and Functions	
`cross`	computes the cross product
`det`	computes the determinant of a matrix
`dot`	computes the dot product
`eye`	generates an identity matrix
`gallery`	contains sample matrices
`inv`	computes the inverse of a matrix
`magic`	creates a "magic" matrix
`pascal`	creates a pascal matrix
`rref`	uses the reduced row echelon format scheme for solving a series of linear equations
`size`	determines the number of rows and columns in a matrix

KEY TERMS

array
commutative
cross product
determinant
dot product
Gaussian elimination
identity matrix
ill-conditioned matrix
inverse
matrix
matrix multiplication
normal
orthogonal
singular matrix
system of equations
transpose
unit vector

PROBLEMS

Dot Products

10.1 Compute the dot product of the following pairs of vectors, and then show that

$$A \cdot B = B \cdot A$$

(a) $A = [1, 3, 5], B = [-3, -2, 4]$
(b) $A = [0, -1, -4, -8], B = [4, -2, -3, 24]$

10.2 Compute the total mass of the components shown in Table P10.2, using a dot product.

Table P10.2 Component Mass Properties

Component	Density (g/m^3)	Volume (m^3)
Propellant	1.2	700
Steel	7.8	200
Aluminum	2.7	300

10.3 Use a dot product and the shopping list in Table P10.3 to determine your total bill at the grocery store.

Table P10.3 Shopping List

Item	Number Needed	Cost
Milk	2 gallons	$3.50 per gallon
Eggs	1 dozen	$1.25 per dozen
Cereal	2 boxes	$4.25 per box
Soup	5 cans	$1.55 per can
Cookies	1 package	$3.15 per package

10.4 Bomb calorimeters are used to determine the energy released during chemical reactions. The total heat capacity of a bomb calorimeter is defined as the sum of the products of the mass of each component and the specific heat capacity of each component, or

$$\text{CP} = \sum_{i=1}^{n} m_i C_i$$

where

m_i = mass of component i, g,
C_i = heat capacity of component, i, J/g K, and
CP = total heat capacity, J/K.

Find the total heat capacity of a bomb calorimeter, using the thermal data in Table P10.4.

Table P10.4 Thermal Data

Component	Mass (g)	Heat Capacity (J/gK)
Steel	250	0.45
Water	100	4.2
Aluminum	10	0.90

10.5 Organic compounds are composed primarily of carbon, hydrogen, and oxygen, and for that reason are often called hydrocarbons. The molecular weight (MW) of any compound is the sum of the products of the number of atoms of each element (Z) and the atomic weight (AW) of each element present in the compound.

$$\text{MW} = \sum_{i=1}^{n} \text{AW}_i \cdot Z_i$$

The atomic weights of carbon, hydrogen, and oxygen are approximately 12, 1, and 16, respectively. Use a dot product to determine the molecular weight of ethanol (C_2H_5OH), which has two carbon, one oxygen, and six hydrogen atoms.

10.6 It is often useful to think of air as a single substance with a molecular weight (molar mass) determined by a weighted average of the molecular weights of the different gases present. With little error, we can estimate the molecular weight of air using only nitrogen, oxygen, and argon. Use a dot product and Table P10.6 to approximate the molecular weight of air.

Table P10.6 Composition of Air

Compound	Fraction in Air	Molecular Weight (g/mol)
Nitrogen, N_2	0.78	28
Oxygen, O_2	0.21	32
Argon, Ar	0.01	40

Matrix Multiplication

10.7 Compute the matrix product *A*B* of the following pairs of matrices:

(a) $A = \begin{bmatrix} 12 & 4 \\ 3 & -5 \end{bmatrix} \quad B = \begin{bmatrix} 2 & 12 \\ 0 & 0 \end{bmatrix}$

(b) $A = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \quad B = \begin{bmatrix} -2 & 4 \\ 3 & 8 \\ 12 & -2 \end{bmatrix}$

Show that *A*B* is not the same as *B*A*.

10.8 You and a friend are both going to a grocery store. Your lists are shown in Table P10.8.

Table P10.8 Alfonzo and Mattie's Shopping List

Item	Number Needed by Alfonzo	Number Needed by Mattie
Milk	2 gallons	3 gallons
Eggs	1 dozen	2 dozen
Cereal	2 boxes	1 box
Soup	5 cans	4 cans
Cookies	1 package	3 packages

The items cost as follows:

Item	Cost
Milk	$3.50 per gallon
Eggs	$1.25 per dozen
Cereal	$4.25 per box
Soup	$1.55 per can
Cookies	$3.15 per package

Find the total bill for each shopper.

10.9 A series of experiments was performed with a bomb calorimeter. In each experiment, a different amount of water was used. Calculate the total heat capacity for the calorimeter for each of the experiments, using matrix multiplication, the data in Table P10.9, and the information on heat capacity that follows the table.

Table P10.9 Thermal Properties of a Bomb Calorimeter

Experiment No.	Mass of Water (g)	Mass of Steel (g)	Mass of Aluminum (g)
1	110	250	10
2	100	250	10
3	101	250	10
4	98.6	250	10
5	99.4	250	10

Component	Heat Capacity (J/gK)
Steel	0.45
Water	4.2
Aluminum	0.90

10.10. The molecular weight (MW) of any compound is the sum of the products of the number of atoms of each element (Z) and the atomic weight (AW) of each element present in the compound, or

$$\mathrm{MW} = \sum_{i=1}^{n} \mathrm{AW}_i \cdot Z_i$$

The compositions of the first five straight-chain alcohols are listed in Table P10.10. Use the atomic weights of carbon, hydrogen, and oxygen (12, 1, and 16, respectively) and matrix multiplication to determine the molecular weight (more correctly called the molar mass) of each alcohol.

Table P10.10 Composition of Alcohols

Name	Carbon	Hydrogen	Oxygen
Methanol	1	4	1
Ethanol	2	6	1
Propanol	3	8	1
Butanol	4	10	1
Pentanol	5	12	1

Matrix Exponentiation

10.11 Given the array

$$A = \begin{bmatrix} -1 & 3 \\ 4 & 2 \end{bmatrix}$$

(a) Raise A to the second power by array exponentiation. (Consult `help` if necessary.)
(b) Raise A to the second power by matrix exponentiation.
(c) Explain why the answers are different.

10.12 Create a 3×3 array called A by using the `pascal` function:

```
pascal (3)
```

(a) Raise A to the third power by array exponentiation.
(b) Raise A to the third power by matrix exponentiation.
(c) Explain why the answers are different.

Determinants and Inverses

10.13 Given the array $A = [-1, 3; 4\ 2]$, compute the determinant of A both by hand and by using MATLAB.

10.14 Recall that not all matrices have an inverse. A matrix is singular (i.e., it doesn't have an inverse) if its determinant equals 0 (i.e., $|A| = 0$). Use the determinant function to test whether each of the following matrices has an inverse:

$$A = \begin{bmatrix} 2 & -1 \\ 2 & 5 \end{bmatrix}, \quad B = \begin{bmatrix} 4 & 2 \\ 2 & 1 \end{bmatrix}, \quad C = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 2 & 2 \\ 5 & -4 & 0 \end{bmatrix}$$

If an inverse exists, compute it.

Cross Products

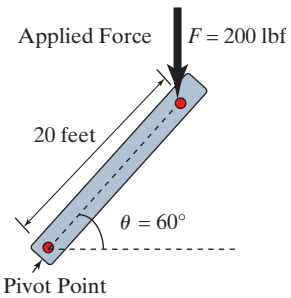


Figure P10.15
Moment of force acting on a lever about the origin.

10.15 Compute the moment of force around the pivot point for the lever shown in Figure P10.15. You'll need to use trigonometry to determine the x and y components of both the position vector and the force vector. Recall that the moment of force can be calculated as the cross product

$$M_0 = r \times F$$

A force of 200 lbf is applied vertically at a position 20 feet along the lever. The lever is positioned at an angle of 60° from the horizontal.

10.16 Determine the moment of force about the point where a bracket is attached to a wall. The bracket is shown in Figure P10.16. It extends 10 inches out from the wall and 5 inches up. A force of 35 lbf is applied to the bracket at an angle of 55° from the vertical. Your answer should be in ft-lbf, so you'll need to do some conversions of units.

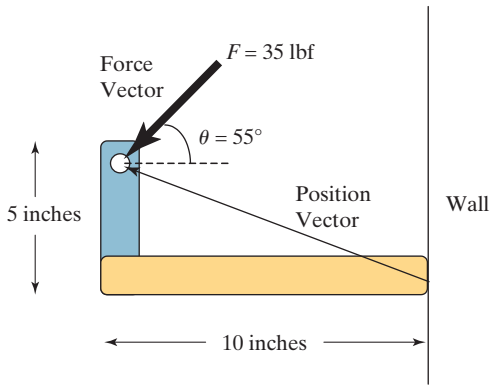


Figure P10.16
A bracket attached to a wall.

10.17 A rectangular shelf is attached to a wall by two brackets 12 inches apart at points *A* and *B*, as shown in Figure P10.17. A wire with a 10-lbf weight attached to it is hanging from the edge of the shelf at point *C*. Determine the moment of force about point *A* and about point *B* caused by the weight at point *C*.

You can formulate this problem by solving it twice, once for each bracket, or by creating a 2×3 matrix for the position vector and another 2×3 matrix for the force vector. Each row should correspond to a different bracket. The `cross` function will return a 2×3 result, each row corresponding to the moment about a separate bracket.

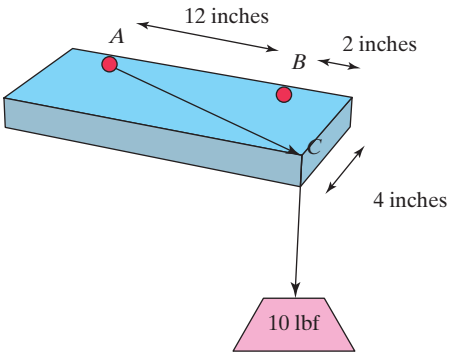


Figure P10.17
Calculation of moment of force in three dimensions.

Solving Linear Systems of Equations

10.18 Solve the following systems of equations, using both matrix left division and the inverse matrix method:

(a)
$$\begin{aligned} -2x + y &= 3 \\ x + y &= 10 \end{aligned}$$

(b)
$$\begin{aligned} 5x + 3y - z &= 10 \\ 3x + 2y + z &= 4 \\ 4x - y + 3z &= 12 \end{aligned}$$

(c)
$$\begin{aligned} 3x + y + z + w &= 24 \\ x - 3y + 7z + w &= 12 \\ 2x + 2y - 3z + 4w &= 17 \\ x + y + z + w &= 0 \end{aligned}$$

10.19 In general, matrix left division is faster and more accurate than the matrix inverse. Using both techniques, solve the following system of equations and time the execution with the `tic` and `toc` functions:

$$\begin{aligned} 3x_1 + 4x_2 + 2x_3 - x_4 + x_5 + 7x_6 + x_7 &= 42 \\ 2x_1 - 2x_2 + 3x_3 - 4x_4 + 5x_5 + 2x_6 + 8x_7 &= 32 \\ x_1 + 2x_2 + 3x_3 + x_4 + 2x_5 + 4x_6 + 6x_7 &= 12 \\ 5x_1 + 10x_2 + 4x_3 + 3x_4 + 9x_5 - 2x_6 + x_7 &= -5 \\ 3x_1 + 2x_2 - 2x_3 - 4x_4 - 5x_5 - 6x_6 + 7x_7 &= 10 \\ -2x_1 + 9x_2 + x_3 + 3x_4 - 3x_5 + 5x_6 + x_7 &= 18 \\ x_1 - 2x_2 - 8x_3 + 4x_4 + 2x_5 + 4x_6 + 5x_7 &= 17 \end{aligned}$$

If you have a new computer, you may find that this problem executes so quickly that you won't be able to detect a difference between the two techniques. If so, see if you can formulate a larger problem to solve.

10.20 In Example 10.5, we demonstrated that the circuit shown in Figure 10.5 could be described by the following set of linear equations:

$$\begin{aligned}(R_2 + R_4)\, i_1 + \quad & (-R_2)\, i_2 + \quad & (-R_4)\, i_3 &= V_1 \\ (-R_2)\, i_1 + (R_1 + R_2 + R_3)\, i_2 + \quad & & (-R_3)\, i_3 &= 0 \\ (-R_4)\, i_1 + \quad & (-R_3)\, i_2 + (R_3 + R_4 + R_5)\, i_3 &= 0\end{aligned}$$

We solved this set of equations by the matrix inverse approach. Redo the problem, but this time use the left-division approach.

10.21 Consider a separation process in which a stream of water, ethanol, and methanol enters a process unit. Two streams leave the unit, each with varying amounts of the three components (see Figure P10.21).

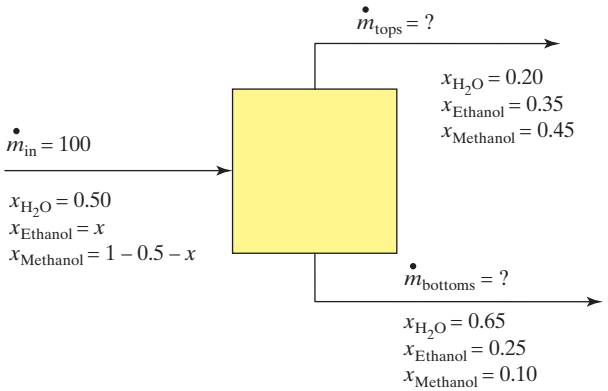


Figure P10.21
Separation process with three components.

Determine the mass flow rates into the system and out of the top and bottom of the separation unit.

(a) First, set up material-balance equations for each of the three components:

Water

$$(0.5)\,(100) = 0.2\dot{m}_{tops} + 0.65\dot{m}_{bottoms}$$

$$50 = 0.2\dot{m}_{tops} + 0.65\dot{m}_{bottoms}$$

Ethanol

$$100x = 0.35\dot{m}_{tops} + 0.25\dot{m}_{bottoms}$$

$$0 = -100x + 0.35\dot{m}_{tops} + 0.25\dot{m}_{bottoms}$$

Methanol

$$100(1 - 0.5 - x) = 0.45\dot{m}_{tops} + 0.1\dot{m}_{bottoms}$$

$$50 = 100x + 0.45\dot{m}_{tops} + 0.1\dot{m}_{bottoms}$$

(b) Arrange the equations you found in part (a) into a matrix representation:

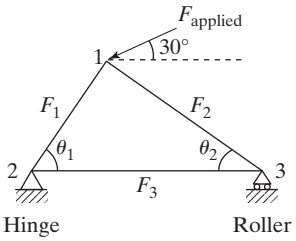


Figure P10.22
A statically determinate truss.

$$A = \begin{bmatrix} 0 & 0.2 & 0.65 \\ -100 & 0.35 & 0.25 \\ 100 & 0.45 & 0.1 \end{bmatrix}, \; B = \begin{bmatrix} 50 \\ 0 \\ 50 \end{bmatrix}$$

(c) Use MATLAB to solve the linear system of three equations.

10.22 Consider the statically determinate truss shown in Figure P10.22. The applied force has a magnitude of 1000 lbf at an angle of 30° from the horizontal, as shown in the figure. The inner angles, θ_1 and θ_2 are 45° and 65° respectively. Determine the values of the forces in each member of the truss, and the reactive forces experienced at the hinge and the roller (nodes 2 and 3).

CHALLENGE PROBLEM

10.23 Create a MATLAB function called *my_matrix_solver* to solve a system of linear equations, using nested `for` loops instead of MATLAB's built-in operators or functions. Your function should accept a coefficient matrix and a result matrix, and should return the values of the variables. For example, if you wish to solve the following matrix equation for X

$$AX = B$$

your function should accept A and B as input, and return X as the result. Test your function with the system of equations from the previous problem.

CHAPTER

11 Other Kinds of Arrays

Objectives

After reading this chapter, you should be able to:

- Identify the different kinds of data used in MATLAB.
- Create and use a variety of different types of numeric arrays.
- Create and use character and string arrays.
- Create and use specialty arrays including logical, datetime, and categorical arrays.
- Create multidimensional arrays, and access data in those arrays.
- Create and use cell, structure, table, and timetable arrays.

INTRODUCTION

The primary **data type** (also called a **class**) in MATLAB is the array. Within the array, MATLAB supports a number of different secondary data types. Because MATLAB was written in C, many of those data types parallel the data types supported in C. In general, all the data within an array must be the same type. However, MATLAB also includes functions to convert between data types, and array types to store different kinds of data in the same array. In this chapter we'll explore the different MATLAB data types and the arrays used to store them.

The types of data that can be stored in MATLAB are listed in Figure 11.1. They include numeric data, textual data, **logical data**, categorical data, date and time data, **symbolic** data, and function handles. Each can be stored either in arrays specifically designed for that data type, or in arrays that can store a variety of data. **Table**, **cell**, map, and **structure** arrays fall into the latter category (see Figure 11.2). A specialized table array specifically designed for tables with timestamped rows was recently added as a separate data type, to make handling large data sets easier.

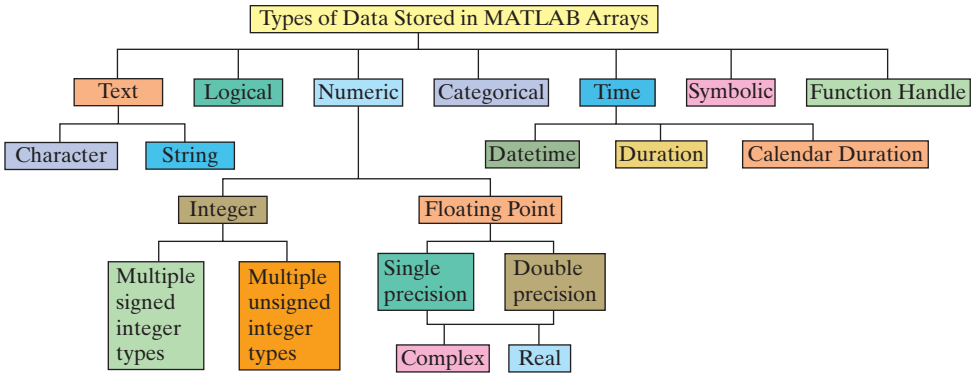


Figure 11.1
Many different kinds of data can be stored in MATLAB. For each of these data types, there is a corresponding array type used to store the data.

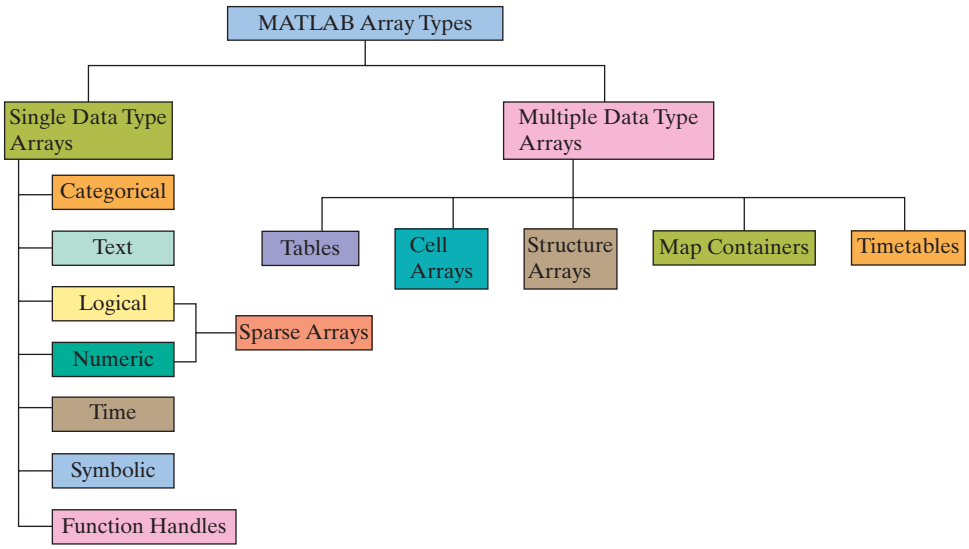


Figure 11.2
In MATLAB, data is stored in arrays. Most arrays store only a single kind of data; however, any data type can be stored in the multiple data type arrays. For example, both numbers and text can be stored in a single structure array. Either numeric or logical values can be stored in sparse arrays.

11.1 NUMERIC DATA TYPES

MATLAB includes arrays for storing multiple types of numeric data. They include **double-precision floating-point numbers**, **single-precision** floating-point numbers, and a variety of **integer** types. Although double-precision numeric numbers are used most commonly by engineers and scientists, each of these data types have different uses and are valuable tools for solving problems.

11.1.1 Double-Precision Floating-Point Numbers

The default numeric data type in MATLAB is the double-precision floating-point number, as defined by IEEE Standard 754. **IEEE**, the Institute of Electrical and Electronics Engineers, is the professional organization for electrical engineers. Recall that when we create a variable such as

```
A = 1
```

the variable is listed in the workspace window, and the class is "double," as shown in Figure 11.3. Notice that the array requires 8 bytes of storage space. Each byte is

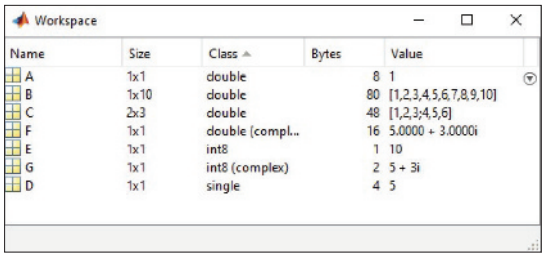


Figure 11.3
MATLAB supports multiple numeric data types, which is listed under the class heading in the workspace window.

equal to 8 bits, so the number 1 requires 64 bits of storage space. Also in Figure 11.3, notice how much storage space is required for variables `B` and `C`:

```
B = 1:10;              C = [1,2,3; 4,5,6];
```

The variable `B` requires 80 bytes, 8 for each of the 10 values stored, and `C` requires 48 bytes, again 8 for each of the 6 values stored.

KEY IDEA
MATLAB supports multiple data types.

You can use the `realmax` and `realmin` functions to determine the maximum possible value of a double-precision floating-point number:

```
realmax
ans =
  1.7977e+308
realmin
ans =
  2.2251e-308
```

If you try to enter a value whose absolute value is greater than `realmax`, or if you compute a number that is outside this range, MATLAB will assign a value of ±infinity:

```
x = 5e400
x =
   Inf
```

Similarly, if you try to enter a value whose absolute value is less than `realmin`, MATLAB will assign a value of zero:

```
x = 1e-400
x =
   0
```

11.1.2 Single-Precision Floating-Point Numbers

Single-precision floating-point numbers use only half the storage space of a double-precision number, and thus store only half the information. Each value requires only 4 bytes, or $4 \times 8 = 32$ bits, of storage space, as shown in the workspace window in Figure 11.3 when we define `D` as a single-precision number:

```
D = single(5)
D =
   5
```

KEY IDEA
Single-precision numbers require half the storage space of double-precision numbers.

We need to use the `single` function to change the value 5 (which is double precision by default) to a single-precision number. Similarly, the `double` function will convert a variable to a double, as in

```
double(D)
```

which changes the variable `D` back into a double.

Since single-precision numbers are allocated only half as much storage space, they cannot cover as large a range of values as double-precision numbers. We can use the `realmax` and `realmin` functions to show this:

```
realmax("single")
ans =
  3.4028e+038
realmin("single")
ans =
  1.1755e-038
```

KEY IDEA
Double-precision numbers are appropriate for most engineering applications.

Engineers will rarely need to convert to single-precision numbers, because today's computers have plenty of storage space for most applications and will execute most of the problems we pose in extremely short amounts of time. However, in some numerical analysis applications, you may be able to improve the run time of a long problem by changing from double- to single-precision. Note, though, that this has the disadvantage of making round-off errors more of a problem.

We can demonstrate the effect of round-off error in single-precision versus double-precision problems with an example. Consider the series

$$\sum\left(\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\cdots+\frac{1}{n}+\cdots\right)$$

A series is the sum of a sequence of numbers, and this particular series is called the *harmonic series*, represented with the following shorthand notation:

$$\sum_{n=1}^{\infty}\frac{1}{n}$$

The harmonic series diverges; that is, it just keeps getting bigger as you add more terms together. You can represent the first 10 terms of the harmonic sequence with the following commands:

```
n = 1:10;
harmonic = 1./n
```

You can view the results as fractions **(rational numbers)** if you change the format to rational:

```
format rat
harmonic =
  1    1/2    1/3    1/4    1/5    1/6    1/7    1/8    1/9    1/10
```

Or you can use the short format, which shows decimal representations of the numbers:

```
format short
harmonic =
  1.0000   0.5000   0.3333   0.2500   0.2000   0.1667   0.1429
           0.1250   0.1111   0.1000
```

No matter how the values are displayed on the screen, they are stored as double-precision floating-point numbers inside the computer. By calculating the partial

sums (also called cumulative sums), we can see how the value of the sum of these numbers changes as more terms are added:

```
partial_sum = cumsum(harmonic)
partial_sum =
 Columns 1 through 6
  1.0000    1.5000    1.8333    2.0833    2.2833    2.4500
 Columns 7 through 10
  2.5929    2.7179    2.8290    2.9290
```

The cumulative sum (`cumsum`) function calculates the sum of the values in the array up to the element number displayed. Thus, in the preceding calculation, the value in column 3 is the partial sum of the values in columns 1 through 3 of the input array (in this case, the array named `harmonic`). No matter how big the harmonic array becomes, the partial sums continue to increase.

The only problem with this process is that the values in `harmonic` keep getting smaller and smaller. Eventually, when n is big enough, `1./n` is so small that the computer can't distinguish it from zero. This happens much more quickly with single-precision than with double-precision representations of numbers. We can demonstrate this property with a large array of n values:

```
n = 1:1e7;
harmonic = 1./n;
partial_sum = cumsum(harmonic);
```

(This may take your computer a while to calculate.) All these calculations are performed with double-precision numbers, because double-precision is the default data type in MATLAB. Plotting the data allows us to see how the results behave.

```
plot(partial_sum)
```

Now we can repeat the calculations, but change to single-precision values. You may need to clear your computer memory before this step, depending on how much memory is available on your system.

```
n = single(1:1e7);
harmonic = 1./n;
partial_sum = cumsum(harmonic);
hold on
plot(partial_sum,'--')
```

KEY IDEA

Round-off error is a bigger problem in single-precision than in double-precision calculations.

The results are presented in Figure 11.4. The solid line represents the partial sums calculated with double-precision. The dashed line represents the partial sums calculated with single-precision. The single-precision calculation levels off, because the point where each successive term is so small that the computer sets it equal to zero is reached at about 2 million iterations. The double-precision values are still large enough for the computer to sum, even after 10 million iterations.

11.1.3 Integers

Although double-precision floating point numbers are the default in MATLAB, integer data types are also available. One use for integers is as counting numbers. For example, there can't be 2.5 people in a room, and you can't specify element number 1.5 in an array. Eight different types of integers are supported by MATLAB.

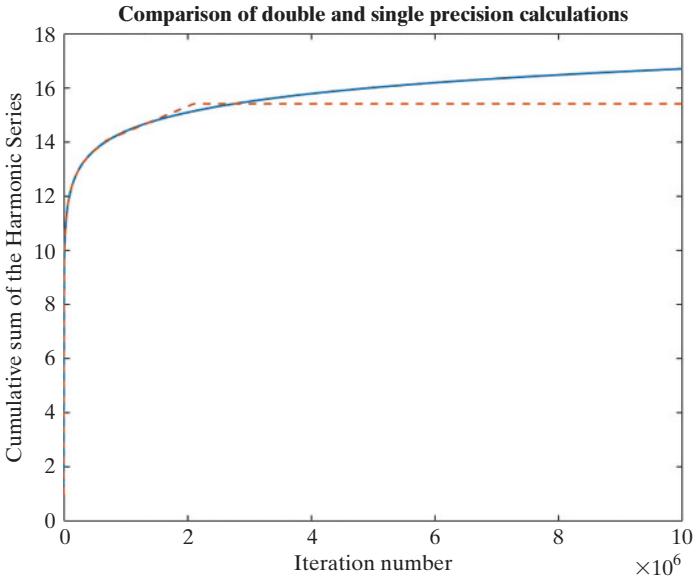


Figure 11.4
Round-off error degrades the harmonic series calculation for single-precision faster than for double-precision numbers.

They differ in how much storage space is allocated for the type, and in whether the values are signed or unsigned. The more storage space, the larger the value of an integer number you can use. The eight types are shown in Table 11.1.

Since 8 bits is 1 byte, when we assign `E` as an `int8` with the code

```
E = int8(10)
E =
  10
```

it requires only 1 byte of storage, as shown in Figure 11.3.

You can determine the maximum value of any of the integer types by using the `intmax` function. For example, the code

```
intmax("int8")
ans =
  127
```

indicates that the maximum value of an 8-bit signed integer is 127.

The four *signed*-integer types allocate storage space to specify whether the number is plus or minus. The four *unsigned*-integer types assume that the number is positive, and thus do not need to store that information, leaving more room to store numerical values.

The code

KEY IDEA
Integer data are often used to store image data.

```
intmax("uint8")
ans =
  255
```

Table 11.1 MATLAB Integer Types

8-bit signed integer	`int8`	8-bit unsigned integer	`uint8`
16-bit signed integer	`int16`	16-bit unsigned integer	`uint16`
32-bit signed integer	`int32`	32-bit unsigned integer	`uint32`
64-bit signed integer	`int64`	64-bit unsigned integer	`uint64`

reveals that the maximum value of an 8-bit unsigned integer is 255.

One place where integer arrays find use is to store image information. These arrays are often very large, but a limited number of colors are used to create the picture. Storing the information as unsigned-integer arrays reduces the storage requirement dramatically.

11.1.4 Complex Numbers

The default storage type for **complex numbers** is double; however, twice as much storage room is needed, because both the real and imaginary components must be stored:

```
F = 5+3i;
```

Thus, 16 bytes (= 128 bits) are required to store a double complex number. Complex numbers can also be stored as singles or integers (see Figure 11.3), as the following code illustrates:

```
G = int8(5+3i);
```

PRACTICE EXERCISE 11.1

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Enter the following list of numbers into arrays of each of the numeric data types [1, 4, 6; 3, 15, 24; 2, 3, 4]:
   (a) Double-precision floating point—name this array `A`
   (b) Single-precision floating point—name this array `B`
   (c) Signed-integer (pick a type)—name this array `C`
   (d) Unsigned-integer (pick a type)—name this array `D`
2. Create a new array `E` by adding `A` to `B`:

   $$\mathbf{E} = \boldsymbol{A} + \boldsymbol{B}$$

   What data type is the result?
3. Define `x` as an integer data type equal to 1 and `y` as an integer data type equal to 3.
   (a) What is the result of the calculation `x/y`?
   (b) What is the data type of the result?
   (c) What happens when you perform the division when `x` is defined as the integer 2 and `y` as the integer 3?
4. Use `intmax` to determine the largest number you can define for each of the numeric data types. (Be sure to include all eight integer data types.)
5. Use MATLAB to determine the smallest number you can define for each of the numeric data types. (Be sure to include all eight integer data types.)
6. Integers are often used to store image information because they require less storage space than double-precision numbers. Prove this to yourself by creating a 1000 x 1000 array of random integers ranging from 1 to 255 using the `randi` function. The array you create by default will be stored as a `double`. Convert it to a `uint8` and compare the amount of storage required for the new array to the original array.

11.2 CHARACTER AND STRING DATA

KEY IDEA
Each character, including spaces, is a separate element in a character array.

MATLAB includes both the **character** and **string** data type to store text information. The string data type was added with release R2016b and offers numerous advantages over character arrays. A significant amount of **legacy code** exists using character arrays, but you should choose strings for most new programs you create. MATLAB functions will accept input as either a character array or a string array and change it to the appropriate data type for use in the function.

11.2.1 Creating Character and String Arrays

Single quotes are used to create a character array and double quotes are used to create a string array. Character arrays store individual characters, whereas string arrays store entire blocks of text as array elements. Thus

```
H='Emanuel';
```

creates a 1×7 character array, but

```
J = "Sarah";
```

creates a 1×1 string array, as shown in Figure 11.5.

To convert a character array to a string array, use the string function, as in:

```
K=string(H);
```

and to convert a string array to a character array, use the char function.

```
L=char(J);
```

Because character arrays store individual characters, and because arrays need to be rectangular, storing character data in arrays is awkward. Two strategies are employed: padding empty spaces in the array with blanks or creating cell arrays of character arrays. Cell arrays are discussed in more detail later in this chapter.

To create a padded character array, use the `char` function, as in:

```
M=char('Steven','Jasmin','David','Terrell','Maria')
M = 5×7 char array
    'Steven '
    'Jasmin '
    'David  '
    'Terrell'
    'Maria  '
```

Notice the empty spaces inside the single quotes in the output. A comma is required between input parameters in the `char` function.

To create a cell array of character arrays, use the cell array constructors, also called 'curly brackets' or braces. Semicolons are required between the cell arrays.

```
N = {'Steven','Jasmin','David'; 'Terrell','Maria'}
N = 5×1 cell
'Steven'
'Jasmin'
'David'
'Terrell'
'Maria'
```

A more efficient approach to storing text is to use a string array, as in:

```
P=["Steven";"Jasmin";"David";"Terrell";"Maria"]
P = 5×1 string
"Steven"
"Jasmin"
"David"
"Terrell"
"Maria"
```

Notice in Figure 11.5 that strings require more memory than equivalent character arrays; however, *arrays* of strings require less memory than character arrays stored in *cell arrays* that contain the same information. The cell array `N` contains the same information as the string array `P` (the list of names), yet it takes 578 bytes of memory versus 366 bytes for the string array. String arrays are preferred for text because they are easier to manipulate, and because they are more memory efficient than cell arrays.

The same techniques used to index into numeric arrays are used to index into both character and string arrays. Manipulating character and string arrays was discussed in an earlier chapter.

11.2.2 Character Encoding Strategies

Understanding how computers store information is important to our understanding of characters. All information in computers is stored as a series of zeros and ones. There are two major coding schemes to do this: **ASCII** and **EBCDIC**. Most small computers use the ASCII coding scheme, whereas many mainframes (especially IBM) and supercomputers use EBCDIC. You can think of the series of zeros and ones as a **binary**, or **base-2**, number. In this sense, all computer information is stored numerically. Every base-2 number has a decimal equivalent. The first several numbers in each base are shown in Table 11.2.

KEY IDEA
Computers use binary coding schemes such as ASCII or EBCDIC to store data.

Every ASCII (or EBCDIC) character stored has both a binary representation, and a decimal equivalent. When MATLAB is asked to change a character to a double, the number returned is the decimal equivalent in the ASCII coding system.

```
double('a')
ans =
  97
```

Conversely, when the `char` function is used on a double, the character represented by that decimal number in ASCII is returned—for example,

```
char (98)
ans =
  b
```

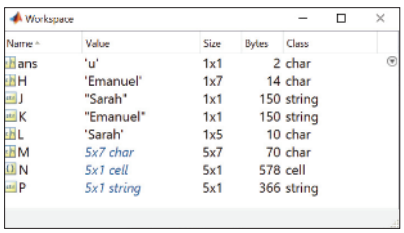


Figure 11.5
MATLAB includes arrays to store text in both character and string arrays.

Table 11.2 Binary-to-Decimal Conversions

Base 2 (binary)	Base 10 (decimal)
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8

To create an array containing both numeric and character information, MATLAB converts all the data to character information:

```
['a',98]
ans =
  ab
```

(The character *b* is equivalent to the number 98.) Not all numbers have a character equivalent. If this is the case, they are represented as a blank or a box in the resulting character array

```
['a',3]
ans =
  a
```

Although this result looks like it has only one character in the array, check the workspace window. The size is a 1×2 character array.

If we try to perform mathematical calculations with both numeric and character information, MATLAB converts the character to its decimal equivalent:

```
'a' + 3
ans =
  100
```

Since the decimal equivalent of *a* is 97, the problem is converted to

$$97 + 3 = 100$$

HINT

The + operator performs very different functions when used with characters and strings. When used with a character, as in

```
'a' + 3
```

the result is a double, since the character 'a' is converted to its numeric equivalent. The result is

```
97 + 3 =100
```

However, when used with a string, the + operator concatenates the 3 into the string array

```
"a" + 3
```

resulting in a new string

```
"a3"
```

PRACTICE EXERCISE 11.2

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Create a character array consisting of the letters in your name.
2. What is the decimal equivalent of the letter *g*?
3. Uppercase and lowercase letters are 32 apart in decimal equivalent. (Uppercase comes first.) Using nested functions, convert the character array 'matlab' to the uppercase equivalent, 'MATLAB'.

EXAMPLE 11.1

CREATING A SIMPLE SECRET CODING SCHEME

Keeping information private in an electronic age is becoming more and more difficult. One approach is to encode information, so that even if an unauthorized person sees the information, they won't be able to understand it. Perhaps one of the most famous examples of encoding information is the Enigma machine (Figure 11.6a). It was used extensively by the Nazis during World War II to encrypt military information.

Alan Turing (Figure 11.6b) was an English computer scientist who was a major contributor to the theoretical development of the field. He's considered by many to be the father of artificial intelligence. He worked for Britain's Government Code



Figure 11.6(a)
Military Enigma machine, model "Enigma I," used during the late 1930s and during World War II; displayed at Museo Nazionale Scienza e Tecnologia Leonardo da Vinci, Milan, Italy. (Chris Howes/Wild Places Photography/Alamy Stock Photo)

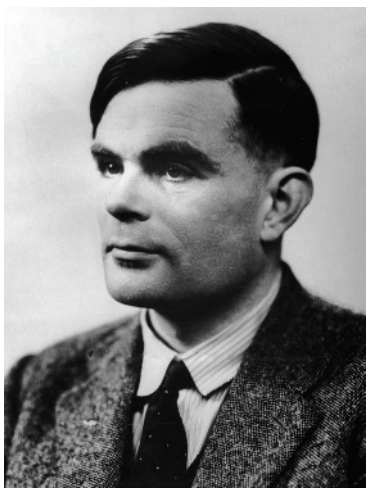


Figure 11.6(b)
Alan Turing was an English computer scientist and cryptanalyst. (Photo Researchers/Science History Images/Alamy Stock Photo)

and Cypher School at Bletchley Park during World War II and was influential in the work to decode messages that had been encrypted by machines such as Enigma. His life, especially his work at Bletchley Park, was chronicled in the movie *The Imitation Game*. He may be the best-known LGBTQ scientist of the 20th century.

Modern coding techniques are extremely complicated, and the strategy used in this example to create a code wouldn't have puzzled anyone during World War II for more than a minute or two. The example does illustrate how character information can be manipulated and combined with integer information, which is the first step in creating something more complicated.

Our simple code will add a constant integer value to character information, transforming it into something that is difficult to interpret.

1. State the problem
   Encode and decode a string of character information.

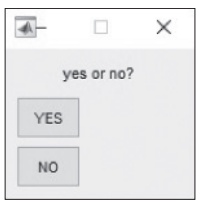
2. Describe the input and output

Input	Character information entered from the command window
Output	Encoded information

3. Develop a hand example
   The lowercase letter *a* is equivalent to the decimal number 97. If we add 5 to *a* and convert it back to a character, it becomes the letter *f*.

4. Develop a MATLAB solution

```
%% Example 11.1
% Prompt the user to enter some text.
clear, clc
A=input('Enter some text to be encoded: \n','s');
encoded = char(A+5);
disp("Your input has been transformed!");
disp(encoded);
disp("Would you like to decode this message?");
response = menu("yes or no?","YES","NO");
switch response
    case 1
        disp(char(encoded-5));
    case 2
       disp("OK - Goodbye");
end
```



5. Test the solution
   Run the program and observe what happens. The program prompts you in the command window for input:

```
Enter some text to be encoded:
'I love rock and roll'
```

   Once you hit the return key, the program responds

```
Your input has been transformed!
N%qt{j%wthp%fsi%wtqq
Would you like to decode this message?
```

(*continued*)

Because we chose to use a menu option for the response, the menu window pops up. When we choose YES, the program responds with

```
I love rock and roll
```

If we choose NO, it responds with

```
OK - Goodbye
```

11.3 SYMBOLIC DATA

Both function handles and the symbolic variables created by the symbolic toolbox are stored in symbolic arrays. Function handles were described in an earlier chapter, and the symbolic toolbox is described in a subsequent chapter.

The symbolic toolbox uses symbolic data to perform symbolic algebraic calculations. To create the symbolic variable `x`, use the `syms` command:

```
syms x
```

Symbolic variables can be combined to create symbolic equations, such as

```
S = x^2-2
S =
x^2-2
```

The storage requirements of a symbolic object depend on how large the object is. Notice, however, in Figure 11.7, that S is a 1×1 array. Subsequent symbolic objects could be grouped together into an array of mathematical expressions. The symbolic-variable icon shown in the left-hand column of Figure 11.7 is a cube.

11.4 LOGICAL DATA

KEY IDEA
Computer programs use the number 0 to mean false, and the number 1 to mean true.

Logical arrays may look like arrays of ones and zeros because MATLAB (as well as other computer languages) uses these numbers to denote true and false:

```
T = [true,false,true]
T =
  1    0    1
```

A double array that consists of 0's and 1's can be converted to a logical array, an array containing logical data, with the logical function. For example if `T` were defined as:

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

it would be stored as a double, which can be confirmed in the workspace window. To convert it to a logical array type

```
T = logical(T)
```

We don't often create logical arrays this way. Usually, they are the result of logical operations. For example,

```
x = 1:5;
y = [2,0,1,9,4];
z = x>y
```

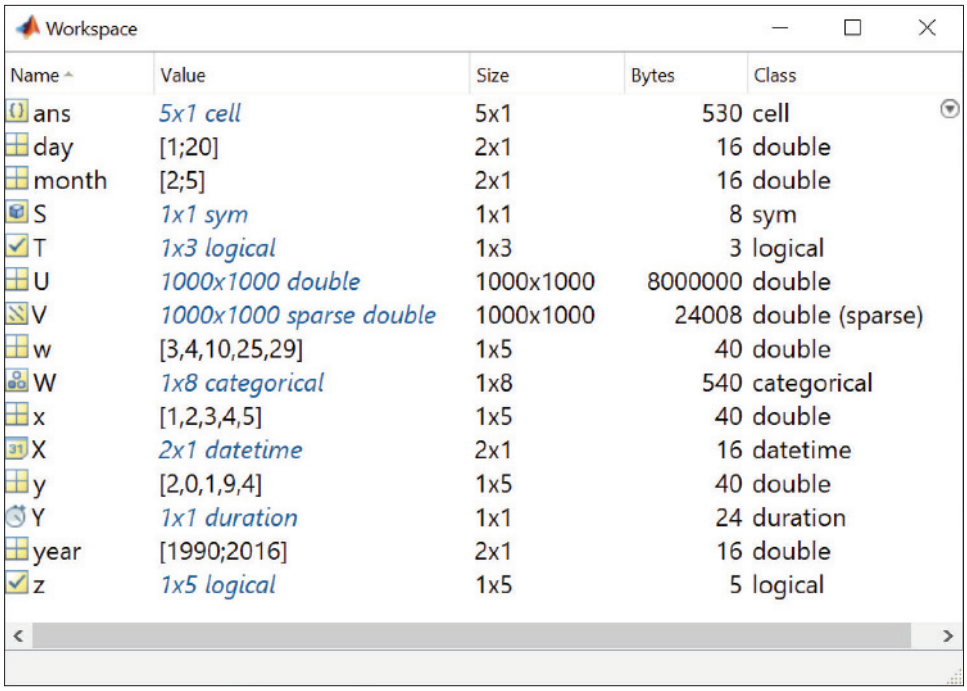


Figure 11.7
A wide variety of data types is available in MATLAB. The data type is listed in the 'Class' column of the workspace window, along with a representative icon next to the variable name.

returns

```
z =
  0   1   1   0   1
```

The variable `z` is a logical array. We can interpret this to mean that `x > y` is false for elements 1 and 3, and true for elements 2, 3, and 5. These arrays are used in logical functions where they are not even seen by the user. For example,

```
find(x>y)
ans =
  2   3   5
```

tells us that elements 2, 3, and 5 of the `x` array are greater than the corresponding elements of the `y` array. Thus, we don't have to analyze the results of the logical operation ourselves. Logical indexing is an example where logical arrays are created and used directly to make decisions. The `z` array defined in the example can be used to extract elements of an existing vector where the condition we analyzed is true. So, for example, if we define an arrray, `w` as:

```
w = x.^2 + y
```

The values of `w` where `x>y` are

```
w(z)
ans =
    4    10    29
```

The icon representing logical arrays is a check mark (see Figure 11.7).

11.5 SPARSE ARRAYS

Both double-precision and logical arrays can be stored either in full arrays or as sparse arrays. Sparse arrays are "sparsely populated," which means that many or most of the values in the array are zero. (Identity matrices are examples of sparse arrays.) If double-precision sparse arrays are stored in the full-array format, every data value takes 8 bytes of storage. The sparse-array format stores only the nonzero values and remembers where they are—a strategy that saves a lot of computer memory.

For example, define a 1000×1000 identity matrix, which is a one-million-element array:

```
U = eye(1000);
```

At 8 bytes per element, storing this array takes 8 MB. By converting it to a sparse array, we can save some space.

```
V = sparse(U);
```

Notice in the workspace window that array `V` requires only 24008 bytes! Sparse arrays can be used in calculations just like full arrays. This is important in some numerical methods applications where memory becomes the limiting factor. The icon representing a sparse array is a group of diagonal lines (see Figure 11.7).

11.6 CATEGORICAL ARRAYS

A categorical array stores information from a finite list of categories. In this way, it is similar to a logical array. The elements in a logical array can only be either "true" or "false"; the elements in a categorical array must have one of a user-defined list of values. Categorical arrays are often used as part of a table of related information. You can create a categorical array using the categorical function and either a string array or a cell array of character arrays. For example, to create an array of the grades earned on a given test

```
W = categorical(["A","C","B","D","E","D","B","A"])
 or
W = categorical({'A','C','B','D','E','D','B','A'})
both of which return
W =
     A C B D E D B A
```

Notice that `W` is listed in the workspace window as a categorical array (see Figure 11.7). The corresponding icon is a square with three circles, reminiscent of a Venn diagram. You can ask for a list of the categories with the `categories` function

```
categories(W)
ans =
  5×1 cell array
    'A'
    'B'
    'C'
    'D'
    'E'
```

The results are returned in alphabetical order. You can use the categories in logical operations, such as

```
find(W=='A')
ans =
     1     8
```

This tells us that elements one and eight in the `W` array are equal to category `A`. Categorical arrays are especially useful as part of a table array, where they can be used to sort the data based on category.

11.7 TIME ARRAYS

Date and time data can be stored in a datetime array. These arrays have the advantage of allowing information to be stored using standard date and time nomenclature. You can even find the differences between given days or times; these results are stored in duration arrays. There are multiple ways to create a datetime array, but here's one example. First create an array of years, then months, and then days. (In this example, we've just used two days to keep things simple.)

```
year = [1990;2016];
month = [2;5];
day = [1;20];
```

Use the `datetime` function to combine the vectors into a datetime array

```
X = datetime(year,month,day)
X =
   01-Feb-1990
   20-May-2016
```

Notice that the `X` array is listed in the workspace window as a datetime array, with an icon that looks like a calendar page. To find the difference between these two dates, use the `diff` function, or subtract `X(1)` from `X(2)`.

```
Y = diff(X); or
Y = X(2) - X(1)
Y =
   230520:00:00
```

The result is the difference in hours, minutes, and seconds. Notice that `Y` is listed as a duration array in Figure 11.7. The icon looks like a clock face.

Datetime arrays are particularly useful when analyzing and plotting data. Consider the monthly summary data in Table 11.3, which was collected at the Denver International Airport by the National Weather Service during 2015. It was stored in a spreadsheet, along with other climate data, called *Denver_Climate_Data_2015.xlsx,* and is available to you from the Pearson website.

Similar information is available for every location where the US National Weather Service gathers information.

To import the data, double-click on the Excel file while in MATLAB to open the import wizard. Select the two columns of data to import, DATE and EMXT. The data type for each will be Number (double). Then select the 'Import Selection' Button at the top of the page.

Table 11.3 Denver Temperature Data for 2015

Month	Extreme High for the Month °F
20150101	75
20150201	73.9
20150301	81
20150401	79
20150501	84.9
20150601	93.9
20150701	97
20150801	98.1
20150901	91.9
20151001	87.1
20151101	75
20151201	69.1

The first column represents the month during which the data were collected. The first four numbers are the year, the second two the month, and the last two the day. The data were stored in a vector named `DATE`. MATLAB automatically selected `double` as the data type when the file was imported. The second column represents the extreme maximum high during the month, and was stored in a column called EMXT, which once downloaded into MATLAB was renamed `HIGH`. A bar graph of the data is shown in Figure 11.8(a). The annotations were added in the usual way.

```
bar(DATE,HIGH)
```

The date information on the *x*-axis is confusing and difficult to read. It would be even worse if the data had been collected every day, since the values would skip, for example, from 20150131 to 20150201. That is precisely what occurred in the Orlando precipitation data plotted in Figure 11.8c. We can use the `datetime` function to convert the `DATE` data into a `datetime` array and then create the plot shown in Figure 11.8(b).

```
date = datetime(DATE,'ConvertFrom','YYYYMMDD')
```

Notice that the format needed to be converted using the ConvertFrom option. An even easier approach would be to import the `DATE` data from the beginning as a `datetime` array, which is one of the options in the import wizard.

Once the `DATE` array has been converted to the new format it's easy to recreate the plot.

```
bar(DATE,HIGH)
```

The resulting graph uses month names along the *x*-axis and adjusts the orientation to make the labels legible.

The graph in Figure 11.8d was modified in exactly the same way. The date data was converted to a `datetime` array and replotted, resulting in a more acceptable representation of the information.

Example 11.2 illustrates how to use a spreadsheet with a large amount of data and `datetime` arrays to create plots that are easy to read and use.

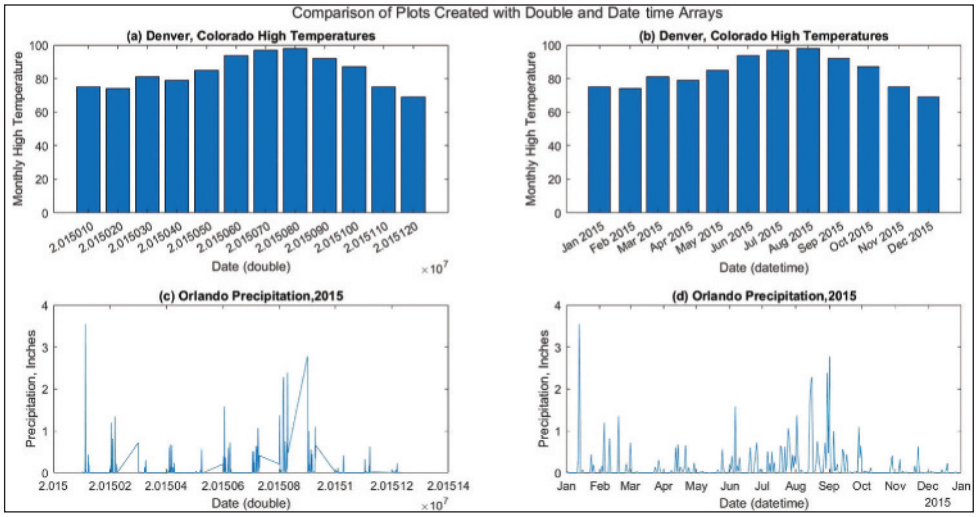


Figure 11.8
Extreme monthly high temperatures for 2015 in Denver, Colorado. (a) This bar graph was created using the date information stored in a double-precision numeric array. (b) In this bar graph, the date information was stored in a datetime array.

EXAMPLE 11.2

VISUALIZING CHANGES IN ARCTIC SEA ICE USING DATETIME ARRAYS

Recently, there has been considerable interest in changes in both Arctic and Antarctic sea ice. Use the daily data collected by the NOAA National Centers for Environmental Information to create a plot of sea ice extent. The data are available from NOAA online, but for this example were stored in an EXCEL spreadsheet named *NH_seaice_extent_final.xlsx*.

1. State the problem
   Create a plot of northern hemisphere (arctic) sea ice extent, using the data in *NH_seaice_extent_final.xlsx*. The first few lines of the spreadsheet are shown in Figure 11.9.

2. Describe the input and output

   Input Data from the EXCEL spreadsheet
   Output An *x*–*y* graph of the extent data

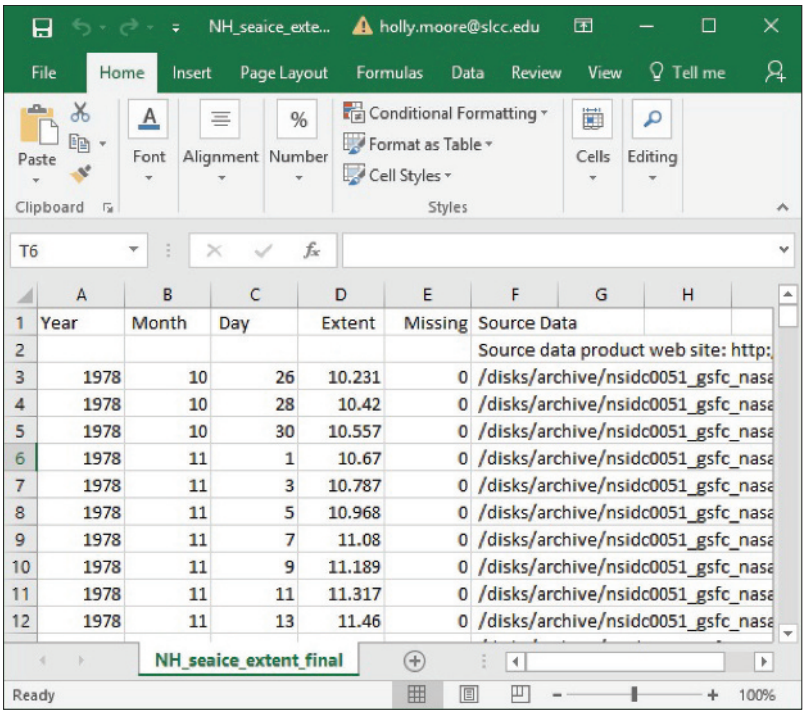
3. Develop a hand example
   The expectation is that the sea ice extent will increase in the winter and decrease in the summer.

4. Develop a MATLAB solution
   Import the data interactively by double-clicking on the spreadsheet file in the MATLAB current folder window, or by using the `uiimport` function

   ```
   uiimport("NH_seaice_extent_final.xlsx")
   ```

   The resulting import screen is shown in Figure 11.10. Select column vectors for the imported data type and then select the import selection icon. This creates vectors for each of the columns in the data file.

(*continued*)



	A	B	C	D	E	F	G	H
1	Year	Month	Day	Extent	Missing	Source Data		
2						Source data product web site: http:		
3	1978	10	26	10.231	0	/disks/archive/nsidc0051_gsfc_nasa		
4	1978	10	28	10.42	0	/disks/archive/nsidc0051_gsfc_nasa		
5	1978	10	30	10.557	0	/disks/archive/nsidc0051_gsfc_nasa		
6	1978	11	1	10.67	0	/disks/archive/nsidc0051_gsfc_nasa		
7	1978	11	3	10.787	0	/disks/archive/nsidc0051_gsfc_nasa		
8	1978	11	5	10.968	0	/disks/archive/nsidc0051_gsfc_nasa		
9	1978	11	7	11.08	0	/disks/archive/nsidc0051_gsfc_nasa		
10	1978	11	9	11.189	0	/disks/archive/nsidc0051_gsfc_nasa		
11	1978	11	11	11.317	0	/disks/archive/nsidc0051_gsfc_nasa		
12	1978	11	13	11.46	0	/disks/archive/nsidc0051_gsfc_nasa		

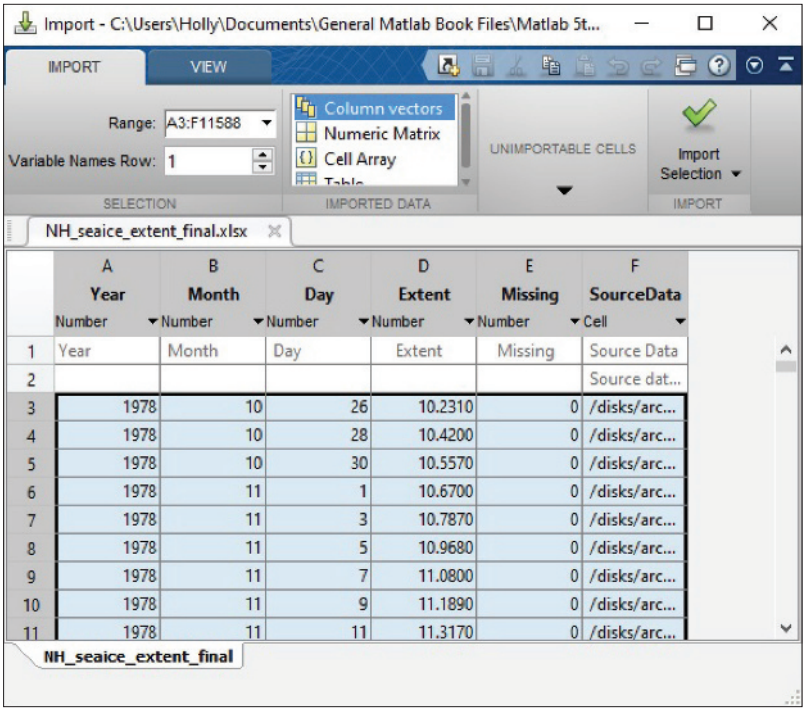
Figure 11.9
Northern Hemisphere sea ice extent. Much of the data stored by NOAA can be downloaded as an EXCEL spread sheet, or as a comma separated value (.csv) file and converted to a spread sheet format.

In this particular data file, the year, month, and day are stored separately, so the `datetime` function can be used to create a single variable representing time. Once a time vector is created, it can be used with the `Extent` vector to create the plot.

```
%% Example 11.2
t = datetime(Year,Month,Day);
plot(t,Extent)
xlabel("date")
ylabel("Arctic Sea Ice Extent, millions of square miles")
title("Variation of Arctic Sea Ice with the Seasons")
grid on
```

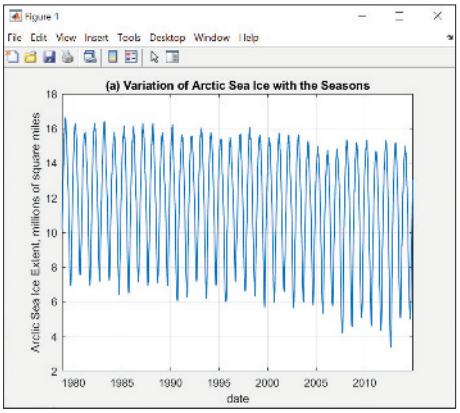
The resulting plot is shown in Figure 11.11(a). You can use the zoom feature, located in the Tools menu on the toolstrip, to better see the behavior of a shorter time span, as shown in Figure 11.11(b). Notice that the *x*-axis labels and the grid automatically rescale to an appropriate spacing.

5. Test the solution
The behavior depicted in the graph shows the sea ice extent increasing and decreasing on a yearly cycle as expected. The maximum values occur in the winter, and the minimum values occur in the summer, also as expected.

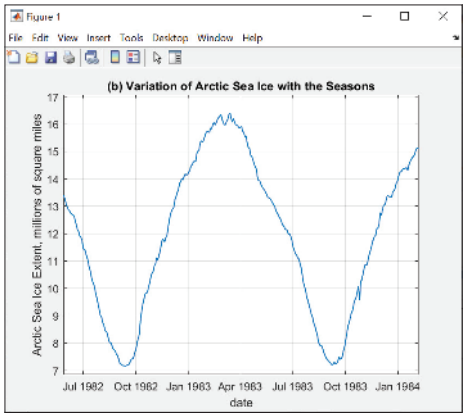


	A Year	B Month	C Day	D Extent	E Missing	F SourceData
	Number	Number	Number	Number	Number	Cell
1	Year	Month	Day	Extent	Missing	Source Data
2						Source dat...
3	1978	10	26	10.2310	0	/disks/arc...
4	1978	10	28	10.4200	0	/disks/arc...
5	1978	10	30	10.5570	0	/disks/arc...
6	1978	11	1	10.6700	0	/disks/arc...
7	1978	11	3	10.7870	0	/disks/arc...
8	1978	11	5	10.9680	0	/disks/arc...
9	1978	11	7	11.0800	0	/disks/arc...
10	1978	11	9	11.1890	0	/disks/arc...
11	1978	11	11	11.3170	0	/disks/arc...

Figure 11.10
The import screen allows the user to select the format used to import data from the spreadsheet.



(a)



(b)

Figure 11.11
A `datetime` array was used to represent the data displayed on the *x*-axis. (a) An appropriate label spacing was automatically chosen and (b), updated when the zoom feature was used to display selected portions of the graph.

11.8 MULTIDIMENSIONAL ARRAYS

When the need arises to store data in multidimensional (more than two-dimensional) arrays, MATLAB represents the data with additional **pages**. Suppose you would like to combine the following four two-dimensional arrays into a three-dimensional array:

```
x = [1,2,3;4,5,6];
y = 10*x;
z = 10*y;
w = 10*z;
```

KEY IDEA
MATLAB supports arrays in more than two dimensions.

Each page must be defined separately:

```
my_3D_array(:,:,1) = x;
my_3D_array(:,:,2) = y;
my_3D_array(:,:,3) = z;
my_3D_array(:,:,4) = w;
```

Read each of the previous statements as all the rows, all the columns, page 1, and so on.

Calling `my_3D_array`, as in:

```
my_3D_array
```

produces the following result.

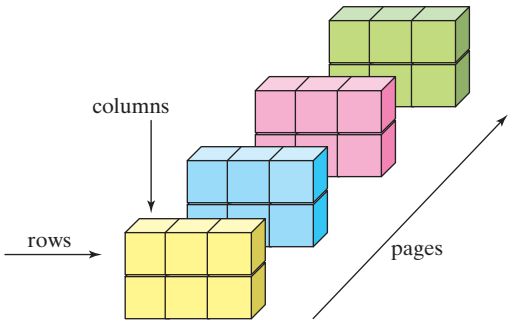
```
my_3D_array
my_3D_array(:,:,1) =
  1    2    3
  4    5    6
my_3D_array(:,:,2) =
  10    20    30
  40    50    60
my_3D_array(:,:,3) =
  100    200    300
  400    500    600
my_3D_array(:,:,4) =
  1000    2000    3000
  4000    5000    6000
```

An alternative approach is to use the `cat` function. When you concatenate a list, you group the members together in order, which is what the `cat` function does. The first field in the function specifies which dimension to use to concatenate the arrays, which follow in order. For example, to create the array we used in the previous example, the syntax is

```
cat(3,x,y,z,w)
```

A multidimensional array can be visualized as shown in Figure 11.12. Even higher-dimensional arrays can be created in a similar fashion. For example, pages can be grouped into **drawers** to create a 4th dimension.

Figure 11.12
Multidimensional arrays are grouped into pages.



HINT

The `squeeze` function can be used to eliminate singleton dimensions in multidimensional arrays. For example, consider the three-dimensional array with the following dimensions:

$$3 \times 1 \times 4$$

This represents an array with three rows, one column, and four pages. It could be stored more efficiently as a two-dimensional array by *squeezing* out the singleton column dimension

```
b = squeeze(a)
```

to give a new array with the dimensions

$$3 \times 4$$

PRACTICE EXERCISE 11.3

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Create a three-dimensional array consisting of a 3×3 magic square, a 3×3 array of zeros, and a 3×3 array of ones.
2. Use triple indexing such as `A(m,n,p)` to determine what number is in row 3, column 2, page 1 of the array you created in Exercise 1.
3. Find all the values in row 2, column 3 (on all the pages) of the array.
4. Find all the values in all the rows and pages of column 3 of the array.

11.9 CELL ARRAYS

KEY IDEA
Cell arrays can store information using various data types.

Unlike the arrays discussed thus far, the cell array can store different types of data inside the same array. Each element in the array is also an array. For example, consider these three different arrays:

```
A = 1:3;
B = 'abcdefg'
C = single([1,2,3;4,5,6]);
```

Each of these arrays is a different data type and size. `A` is a double, `B` is a char, and `C` is a single. They can be combined into one cell array by using curly brackets as the cell-array constructor (the standard array constructors are square brackets):

```
my_cellarray = {A,B,C}
```

returns

```
my_cellarray =
[1x3 double] 'abcdefg' [2x3 single]
```

Notice that the array is listed in the workspace window, as shown in Figure 11.13. An icon with curly brackets is used to represent cell arrays. To save space, large arrays are listed with just size information.

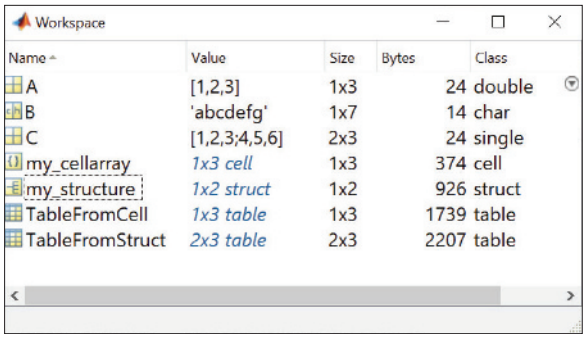


Figure 11.13
Cell, structure, and table arrays can contain many different types of data.

You can show the entire array by using the `celldisp` function:

```
celldisp(my_cellarray)
my_cellarray{1} =
  1    2    3
my_cellarray{2} =
abcdefg
my_cellarray{3} =
  1    2    3
```

The indexing system used for cell arrays is the same as that used in other arrays. You may use either a single index or a row-and-column indexing scheme. There are two approaches to retrieving information from cell arrays: You can use parentheses, as in

```
my_cellarray(1)
ans =
  [1x3 double]
```

which returns a result as a new cell array. An alternative is to use curly brackets, as in

```
my_cellarray{1}
ans =
  1    2    3
```

In this case, the answer is a double. To access a particular element inside an array stored in a cell array, you must use a combination of curly brackets and parentheses:

```
my_cellarray{3}(1,2)
ans =
  2
```

Cell arrays can become quite complicated. The `cellplot` function is a useful way to view the structure of the array graphically, as shown in Figure 11.14.

```
cellplot(my_cellarray)
```

Cell arrays are useful for complicated programming projects or for database applications. A use in engineering applications would be to store various kinds of data from a project in one variable name that can be disassembled and used later. However, a more common use is to use cell arrays to group information together as input to other functions. For example, to create a title on a graph that contains two lines, create a two-element cell array, arranged as a column, for use as input to the `title` function.

```
title({'My Graph '; 'Contains a lot of data '})
```

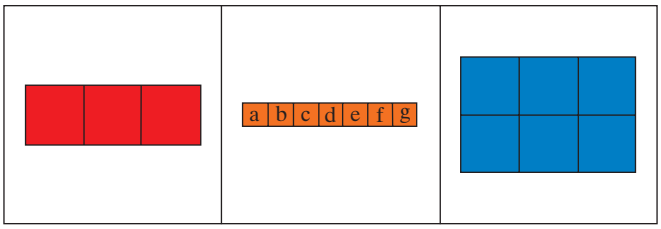


Figure 11.14
The `cellplot` function provides a graphical representation of the structure of a cell array.

Cell arrays of characters were commonly used in the past; however, it is much more efficient to use strings, and cell arrays of characters should be avoided. They are still found in legacy code, so it is important to understand how they function. Cell arrays of characters are still used in tables for labeling as of R2021b.

Table arrays have largely replaced cell arrays for most applications. It is easy to convert a cell array to a table array with the `cell2table` function. For example, to convert the `my_cellarray` variable type:

```
TableFromCell = cell2table(my_cellarray)
TableFromCell = 1×3 table
```

	my_cellarray1			my_cellarray2	my_cellarray3
1	1	2	3	'abcdefg'	[1,2,3;4,5,6]

The result is a 1x3 table array (Figure 11.13). Similarly, if you really need a cell array, use the `table2cell` function.

KEY IDEA
Structure arrays can store information using various data types.

11.10 STRUCTURE ARRAYS

Structure arrays are similar to cell arrays. Multiple arrays of differing data types can be stored in structure arrays, just as they can in cell arrays. Instead of using content indexing, however, each array stored in a structure array is assigned a location called a *field*. For example, define the three arrays from the previous section on cell arrays,

```
A = 1:3;
B = 'abcdefg';
C = single([1,2,3;4,5,6]);
```

To create a simple structure array called `my_structure` use dot notation, as in:

```
my_structure.some_numbers = A
```

which returns

```
my_structure =
  some_numbers: [1 2 3]
```

The name of the structure array is `my_structure`. It has one field, called `some_numbers`. To add the content in the character array `B` to a second field called `some_letters` use dot notation again to expand the structure:

```
my_structure.some_letters = B
my_structure =
  some_numbers: [1 2 3]
  some_letters: 'abcdefg'
```

Finally, the single-precision numbers in array `C` are used to create a third field called `some_more_numbers`:

```
my_structure.some_more_numbers = C
my_structure =
  some_numbers: [1 2 3]
  some_letters: 'abcdefg'
  some_more_numbers: [2x3 single]
```

Notice in the workspace window (see Figure 11.13) that the structure array (called a `struct`) is a 1×1 array that contains all the information from all three dissimilar arrays. The structure has three fields, each containing a different data type:

some_numbers	double-precision numeric data
some_letters	character data
some_more_numbers	single-precision numeric data

We can add more content to the structure, and expand its size, by adding more arrays to the fields we've defined:

```
my_structure(2).some_numbers = [2 4 6 8]
my_structure =
1x2 struct array with fields:
  some_numbers
  some_letters
  some_more_numbers
```

You can access the information in structure arrays by using the array name, field name, and index numbers. The syntax is similar to that used for other types of arrays. An example is

```
my_structure(2)
ans =
              some_numbers: [2 4 6 8]
              some_letters: []
         some_more_numbers: []
```

Notice that `some_letters` and `some_more_numbers` are empty arrays because information was never added to those fields.

To access just a single field, add the field name:

```
my_structure(2).some_numbers
ans =
    2    4    6    8
```

Finally, to determine the content of a particular element in a field, specify the element index number after the field name:

```
my_structure(2).some_numbers(2)
ans =
    4
```

The `disp` function displays the contents of structure arrays. For example,

```
disp(my_structure(2).some_numbers(2))
```

returns

```
4
```

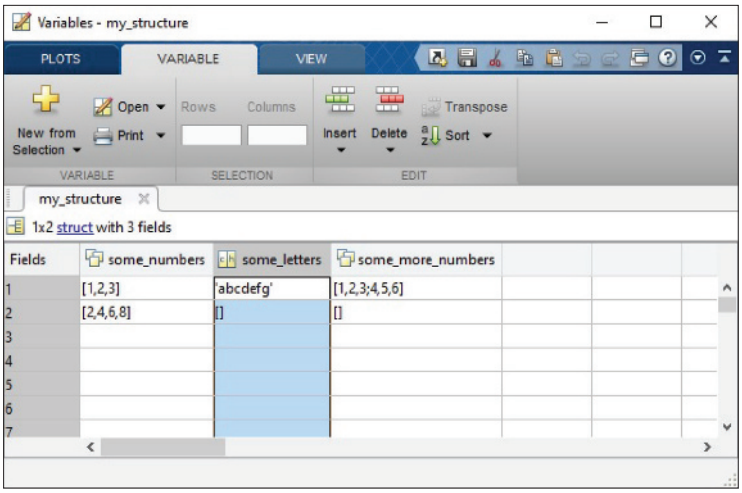


Figure 11.15
The variable editor displays the content of a structure array. Values stored in the array can also be modified or new values added from this screen.

You can use the variable editor to access the content of a structure array (and any other array, for that matter). When you double-click the structure array in the workspace window, the variable editor opens (see Figure 11.15), showing the fields and their content.

Structure arrays are widely used in large computer programs to pass information between functions. They are also extremely useful in applications such as database management. Since large amounts of engineering data are often stored in a database, the structure array is extremely useful for data analysis. The examples that follow will give you a better idea of how to manipulate and use structure arrays.

Table arrays have largely supplanted structures as a programing strategy since they were introduced with MATLAB R2016b. However, the decision on whether to use structure arrays or tables is largely a matter of style. There is a significant library of legacy code that utilizes structure arrays, so it is important to understand how they work so that you can interact with existing functions and programs. *Rectangular* structure arrays can be converted to table arrays with the `struct2table` function. To convert the `my_structure` array into a table type:

```
TableFromStruct = struct2table(my_structure)
```

This will return an error message unless you have added data to the empty fields, because the structure array was not rectangular. If desired, the information can be added in the variable editor. If you don't know what to add, NaN (not a number) will solve the problem. Similarly, if you really need a structure array, use the `table2struct` function to convert a `table` to a structure array.

EXAMPLE 11.3

STORING PLANETARY DATA WITH STRUCTURE ARRAYS

Structure arrays can be used much like a database. You can store numeric information, as well as character data or any of the other data types supported by MATLAB. Create a structure array to store information about the planets. Prompt the user to enter the data.

1. State the problem
Create a structure array to store planetary data and input the information from Table 11.4. Although Pluto is no longer considered a planet, we've included the relevant data because it is interesting.

(*continued*)

Table 11.4 Planetary Data

Planet Name	Mass (in Earth multiples)	Length of Year (in Earth years)	Mean Orbital Velocity (km/s)
Mercury	0.055	0.24	47.89
Venus	0.815	0.62	35.03
Earth	1	1	29.79
Mars	0.107	1.88	24.13
Jupiter	318	11.86	13.06
Saturn	95	29.46	9.64
Uranus	15	84.01	6.81
Neptune	17	164.8	5.43
Pluto	0.002	247.7	4.74

2. Describe the input and output

   Input Data from Table 11.4
   Output A structure array storing the data

3. Develop a hand example
   Developing a hand example for this problem would be difficult. Instead, a flowchart would be useful.

4. Develop a MATLAB solution

```
%% Example 11.3
% Create a structure with 4 fields
clear, clc
k=1;
response = menu("Would you like to enter planetary
data?","yes","no");
while response==1
    planetary(k).name=input("Enter a planet : ",'s');
    planetary(k).mass=input("Enter the mass in multiples of
earth's mass: ");
    planetary(k).year=input("Enter the length of the
planetary year in Earth years: ");
    planetary(k).velocity=input("Enter the mean orbital
velocity in km/sec: ");
    % Review the input
    planetary(k)
    increment = menu("Was the data correct?","Yes","No");
    switch increment
        case 1
            increment =1;
        case 2
            increment = 0;
    end
    k=k+increment;
    response = menu("Would you like to enter more planetary
data?","yes","no");
end
%%
planetary      %output the information stored in planetary
```

Here's a sample interaction in the command window when we run the program and start to enter data:

```
Enter a planet name: Mercury
Enter the planetary mass in multiples of Earth's mass: 0.055
Enter the length of the planetary year in Earth years: 0.24
Enter the mean orbital velocity in km/sec: 47.89
ans =
    name: 'Mercury'
    mass: 0.0550
    year: 0.2400
    velocity: 47.8900
```

5. Test the solution

Enter the data, and compare your array with the input table. As part of the program, the input values were reported back to the screen so that the user could check for accuracy. If the user responds that the data are not correct, the information is overwritten the next time through the loop. Menus were also used instead of free responses to some questions, so that there would be no ambiguity regarding the answers. Notice that the structure array we built, called `planetary`, is listed in the workspace window. If you double-click on `planetary`, the variable editor pops up and allows you to view any of the data in the array (see Figure 11.16). You can also update any of the values in the variable editor.

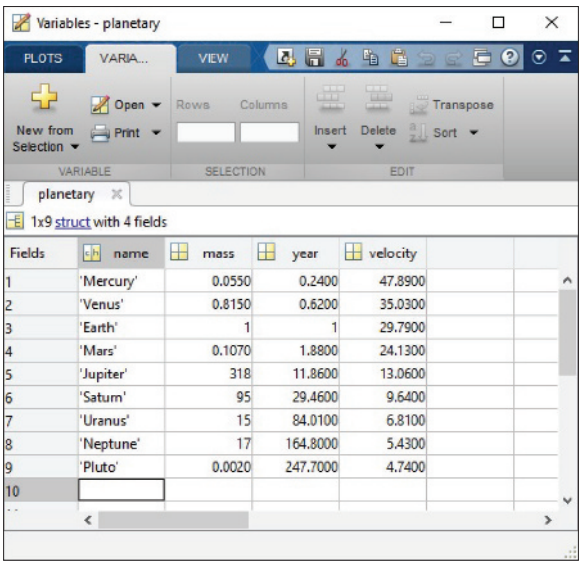


Figure 11.16
The variable editor allows you to view (and change) data in the structure array.

This structure array will be used in Example 11.4 to perform some additional calculations. Save your results as

```
save    planetary_information    planetary
```

This command sequence saves the structure array `planetary` into the file *planetary_information.mat*.

EXAMPLE 11.4

EXTRACTING AND USING DATA FROM STRUCTURE ARRAYS

Structure arrays have some advantages for storing information. First, they use field names to identify array components. Second, information can be added to the array easily and is always associated with a group. Finally, it's hard to accidentally scramble information in structure arrays. To demonstrate these advantages, use the data stored in the *planetary_information.mat* file to complete the following tasks:

- Identify the field names in the array, and list them.
- Create a list of the planet names.
- Create a report representing the data in the structure array. Include the field names as column headings in the report.
- Calculate and report the average of the mean orbital velocity values.
- Find the biggest planet and report its size and name.
- Find and report the orbital period of Jupiter.

1. State the problem
   Create a program to perform the tasks listed.
2. Describe the input and output

   Input *planetary_information.mat*, stored in the current folder
   Output Create a report in the command window
3. Develop a hand example
   You can complete most of the designated tasks by accessing the information in the planetary structural array through the array editor
4. Develop a MATLAB solution

```
%% Example 11.3
  clear, clc
  load planetary_information
%% Identify the field names in the structure array planetary
% recalls the contents of the structure array named planetary
  planetary
%% Create a list of planets in the file
  planetary.name
  fprintf('\n')   % Creates an empty line in the output
%% Create formated output with fprintf and a loop
disp("     Planet       mass       year   velocity");
for k=1:length(planetary)
fprintf(" %10s \t %8.4f  %8.4f  %8.4f \n ",planetary(k).
name,planetary(k).mass,planetary(k).year, planetary(k).
velocity)
end
%% Find the average planet mean orbital velocity
MOV = mean([planetary.velocity]);
fprintf("The mean orbital velocity is %8.2f km/sec\n",MOV)
%% Find the planet with the maximum mass
max_mass = max([planetary.mass]);
fprintf('The maximum mass is %8.2f times the earth''s
\n',max_mass)
```

```
%% Jupiter is planet #5
% Find the orbital period of Jupiter
planet_name = planetary(5).name;
planet_year = planetary(5).year;
fprintf(' %s has a year %6.2f times the earth''s \n',planet_
name,planet_year)
```

Most of this program consists of formatting commands. Before you try to analyze the code, run the program in MATLAB and observe the results.

5. Test the solution
Compare the information extracted from the array with that available from the array editor. Using the array editor becomes unwieldy as the data stored in `planetary` increases. It is easy to add new fields and new information as they become available. For example, these lines of code add the number of moons surrounding each planet to the existing structure:

```
%% Add a new field for the number of moons
planetary(1).moons = 0;
planetary(2).moons = 0;
planetary(3).moons = 1;
planetary(4).moons = 2;
planetary(5).moons = 60;
planetary(6).moons = 31;
planetary(7).moons = 27;
planetary(8).moons = 13;
planetary(9).moons = 1;
```

We can report the number of moons for each planet with the command

```
disp("The number of moons orbiting each planet")
disp([planetary.moons]')
```

11.11 TABLE ARRAYS

Table arrays were introduced in MATLAB R2013b and have been expanded and improved regularly since then. In an earlier chapter we explored how tables can be used to display information, but they are much more than that. They can store information grouped into any of the other data types. They are especially useful for tabular data such as that stored in spreadsheets.

11.11.1 Creating Tables from Workspace Variables

MATLAB includes sample data that can be used to demonstrate how to create tables, either from existing arrays, or from a data file. First clear the workspace and then create some variables in the workspace by loading the data from the *patients.mat* file.

```
clear, clc
load patients
```

As seen in Figure 11.17 a variety of variables are created, all of which are column vectors with 100 rows. Notice that when these arrays were created the choice was made to store text in cell arrays of character arrays. For most applications, string arrays are a better choice. For example, the `LastName` array requires 11616 bytes to store as a cell array. By converting it to a string array, as in

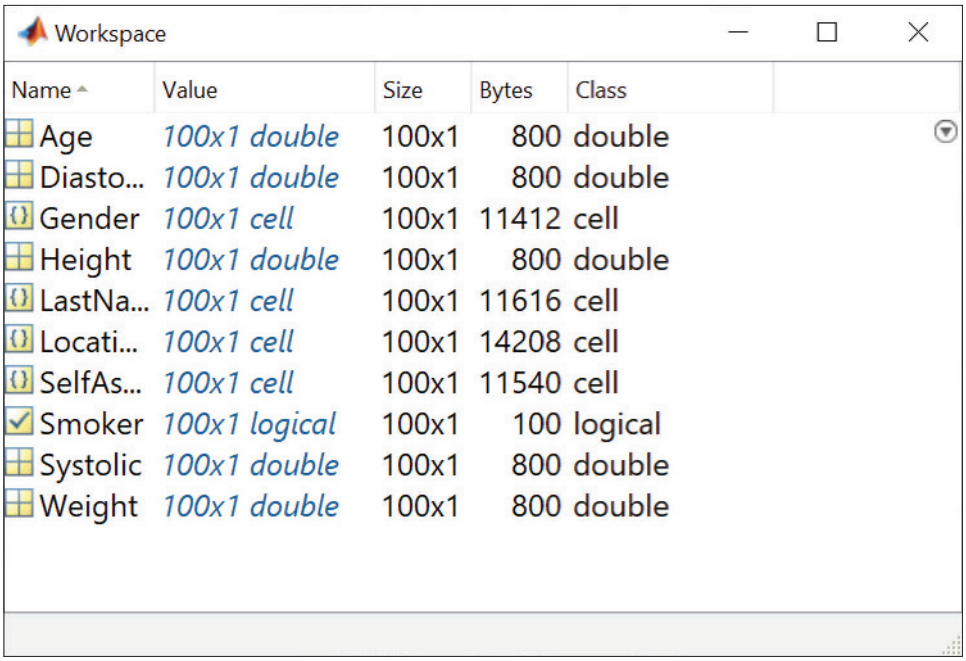


Figure 11.17
Table arrays can be created from workspace variables such as the ones downloaded from the sample data in the **patients.mat** file. The file comes with the standard MATLAB installation.

```
LastName = string(LastName);
```

the required storage drops to 5784 bytes.

The `Location` array is a list of the places where the data was gathered. It would be better stored as a categorical array, rather than a cell array.

```
Location=categorical(Location);
```

To create a table array from the stored variables, use the `table` function. For example, to create a table called `T` with just the `LastName, Location, Smoker` and `Gender` information type:

```
T=table(LastName,Location,Smoker,Gender);
```

To index into a table, use the same strategies used in numeric arrays; specify the row and column number. To determine if the patient on the second row is a smoker, this corresponds to row 2, column 3:

```
T(2,3)
```

which returns

```
ans = 1×1 table
```

	Smoker
1	0

The `Smoker` variable is a logical array, so a result of 0 corresponds to 'false'.

Tables are displayed a bit differently in the live editor, which is the format shown here, than in the command window. The information is the same no matter the format.

To see everything on a specific row or column use the colon operator. For example, to see all the information about the patient on the second row type:

```
T(2,:)
```

which gives

```
ans = 1×4 table
```

	LastName	Location	Smoker	Gender
1	"Johnson"	VA Hospital	0	'Male'

The columns are automatically named using the variable names from the workspace, but different names can be defined using the `VariableNames` property in the `table` function. For example, to rename the variable name in the second column `Hospital`, use the following code.

```
names=["LastName","Hospital","Smoker","Gender"];
T=table(LastName,Location,Smoker,Gender,'VariableNames',names);
```

The `names` array used for input to the `table` function can be either a cell array of character arrays, or a string array. However, the `VariableNames` property must be in single quotes; in other words it must be a character array. This is one of the few instances where a function will not accept either a string or a character array. It must be a character array.

The rows in a table array can also be assigned names. If the variable exists in the workspace it can be used directly, as in:

```
T.Properties.RowNames=LastName;
```

To refer to a variable from the table use dot notation, as in:

```
T.Properties.RowNames=T.LastName;
```

Once row names have been assigned, rows can be extracted using the row name(s) and the colon operator,

```
T("Johnson",:)
ans = 1×4 table
```

	LastName	Location	Smoker	Gender
1	"Johnson"	VA Hospital	0	'Male'

or

```
T(["Martinez","Bryant"],:)
ans = 2×4 table
```

	LastName	Hospital	Smoker	Gender
1 Martinez	"Martinez"	Count General Hospital	0	'Male'
2 Bryant	"Bryant"	County General Hospital	0	'Female'

Similarly, portions of the table based on the row and column names can also be extracted. For example, to determine at which hospitals patients Johnson and Garcia were seen, type:

```
T(["Johnson","Garcia"],["Hospital"])
ans = 2×1 table
```

	Hospital
1 Johnson	VA Hospital
2 Garcia	VA Hospital

11.11.2 Creating Table Arrays from Data Files

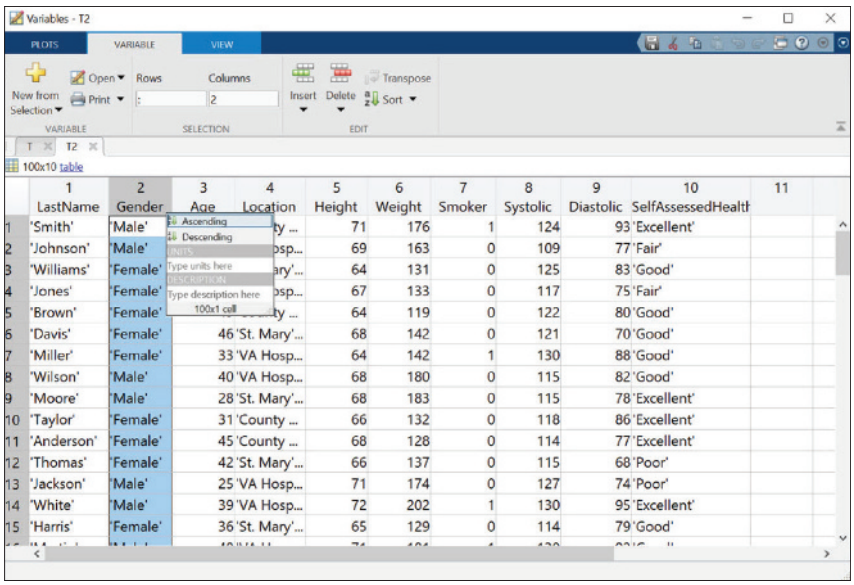
MATLAB offers functions to import tabular data into table arrays from multiple types of data files, including spreadsheets like Excel, text files such as .csv files, .txt files, or .dat files. The main requirement is that the data needs to be rectangular. These functions can ignore the information in headers and can evaluate the type of information stored in each column to determine the appropriate data type. For example, numbers will be stored as doubles and text will be stored as strings or cell arrays of character arrays. The information in the data file *patients.dat* can be loaded into MATLAB either interactively using the `uiimport` function or automatically using the `readtable` function. Both approaches offer advantages and disadvantages.

Using readtable

First import the data automatically using the `readtable` function. Clear the workspace and use a new name for the table array, `T2`.

```
clear, clc
T2=readtable("patients.dat");
```

The workspace reports an abbreviated description of what's in the table array, but the `summary` function gives more detail. It includes the names of each column, the data types for each of the columns in the array, and some statistical information. In fact, it's the same information that was presented in the workspace window when the individual arrays were downloaded. You can also view the information in the array in the variable-editing window (Figure 11.18) by double-clicking on the variable name in the workspace window. The variable editor will open automatically.



	1 LastName	2 Gender	3 Age	4 Location	5 Height	6 Weight	7 Smoker	8 Systolic	9 Diastolic	10 SelfAssessedHealth	11
1	'Smith'	'Male'		ty ...	71	176	1	124	93	'Excellent'	
2	'Johnson'	'Male'		osp...	69	163	0	109	77	'Fair'	
3	'Williams'	'Female'		ary'...	64	131	0	125	83	'Good'	
4	'Jones'	'Female'		osp...	67	133	0	117	75	'Fair'	
5	'Brown'	'Female'		ty ...	64	119	0	122	80	'Good'	
6	'Davis'	'Female'	46	'St. Mary'...	68	142	0	121	70	'Good'	
7	'Miller'	'Female'	33	'VA Hosp...	64	142	1	130	88	'Good'	
8	'Wilson'	'Male'	40	'VA Hosp...	68	180	0	115	82	'Good'	
9	'Moore'	'Male'	28	'St. Mary'...	68	183	0	115	78	'Excellent'	
10	'Taylor'	'Female'	31	'County ...	66	132	0	118	86	'Excellent'	
11	'Anderson'	'Female'	45	'County ...	68	128	0	114	77	'Excellent'	
12	'Thomas'	'Female'	42	'St. Mary'...	66	137	0	115	68	'Poor'	
13	'Jackson'	'Male'	25	'VA Hosp...	71	174	0	127	74	'Poor'	
14	'White'	'Male'	39	'VA Hosp...	72	202	1	130	95	'Excellent'	
15	'Harris'	'Female'	36	'St. Mary'...	65	129	0	114	79	'Good'	

Figure 11.18
The variable editor displays the contents of table arrays. A drop-down arrow in the column headers gives the user access to information on the column data types and allows the data to be sorted.

```
summary(T2)
Variables:
    LastName: 100×1 cell array of character vectors
    Gender: 100×1 cell array of character vectors
    Age: 100×1 double
        Values:
            Min          25
            Median       39
            Max          50
    Location: 100×1 cell array of character vectors
    Height: 100×1 double
        Values:
            Min          60
            Median       67
            Max          72
    Weight: 100×1 double
        Values:
            Min          111
            Median     142.5
            Max          202
    Smoker: 100×1 double
        Values:
            Min          0
            Median       0
            Max          1
    Systolic: 100×1 double
        Values:
            Min         109
            Median      122
            Max         138
    Diastolic: 100×1 double
        Values:
            Min           68
            Median      81.5
            Max           99
    SelfAssessedHealthStatus: 100×1 cell array of character
vectors
```

As noted before, some of this information is stored in arrays of a data type we might want to change. Since it is already in the `T2` array it can be changed by accessing the columns using dot syntax.

```
T2.LastName=string(T2.LastName);
T2.Gender=categorical(T2.Gender);
T2.Location=categorical(T2.Location);
T2.Smoker=logical(T2.Smoker);
T2.SelfAssessedHealthStatus=categorical(T2.SelfAssessedHealthStatus);
```

Using UIImport

The `uiimport` function offers more flexibility for importing data, since you get to make the decisions yourself about how to assign data types before the table array is created. Imagine that you are involved in a research project where you will be

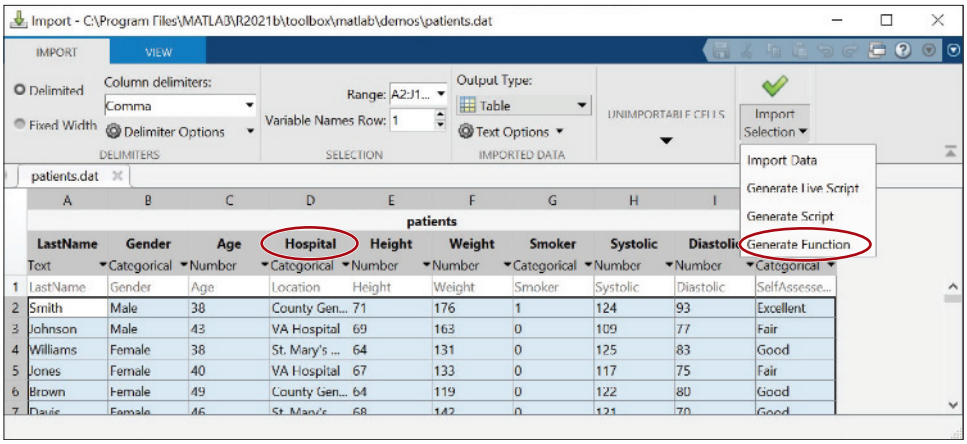


Figure 11.19
The import wizard allows the user to control how files are imported into tables. By selecting Import Selection, a function can be generated to import future selections.

generating multiple data sets to import. You can use the `uiimport` function to make the decisions on how the data should be stored, and then generate a function to recreate those decisions for future files.

Open the *patients.dat* file using the `uiimport` function

```
clear, clc
T3=uiimport("patients.dat")
```

The import wizard opens as shown in Figure 11.19.

As you examine the import wizard screen notice the drop-down menus on the column headers. These allow the user to select different data types for each column. In this case the wizard has already selected strings for the text columns instead of cell arrays of characters. It has also selected categorical arrays for Location, Gender, and SelfAssessedHealthStatus. Interestingly, logical arrays are not an option, so we can't change the `Smoker` array data type to logical, but we can change it to categorical. Change the name of the `Location` array to `Hospital` and the `SelfAssessedHeathStatus` to `SAHS`. After you've made all your changes, select the data without the column headings (rows 2 and below) and then select the Import Selection check box to import the data, which should appear in the workspace window as an array named `patients` because the file was named *patients.dat*.

Once you've checked the array that you've created to make sure it meets your needs you can go back to the import wizard, which should still be open, and select the drop-down menu on the Import Selection button. The generate function option will create a function with the import selections you've chosen. Here's the function created with the choices from this example. As with all functions it needs to be saved to a file with the same name as the function name.

```
function patients = importfile(filename, dataLines)
% IMPORTFILE1 Import data from a text file
% PATIENTS = IMPORTFILE1(FILENAME) reads data from text file
% FILENAME
% for the default selection. Returns the data as a table.
%% PATIENTS = IMPORTFILE1(FILE, DATALINES) reads data for the
% specified
% row interval(s) of text file FILENAME. Specify DATALINES as a
% positive scalar integer or an N-by-2 array of positive scalar
% integers
% for dis-contiguous row intervals.
```

```
%% Example:
% patients = importfile("C:\Program Files\MATLAB\R2021b\
% toolbox\matlab\demos\patients.dat", [2, Inf]);
%% See also READTABLE.
%% Auto-generated by MATLAB on 23-Jul-2021 10:52:27
%% Input handling
% If dataLines is not specified, define defaults
if nargin < 2
    dataLines = [2, Inf];
end
%% Set up the Import Options and import the data
opts = delimitedTextImportOptions("NumVariables", 10);
% Specify range and delimiter
opts.DataLines = dataLines;
opts.Delimiter = ",";
% Specify column names and types
opts.VariableNames = ["LastName", "Gender", "Age", "Hospital", ...
"Height", "Weight", "Smoker", "Systolic", "Diastolic", "SAHS"];
opts.VariableTypes = ["string", "categorical", "double", ...
"categorical", "double", "double", "categorical", "double", ...
"double", "categorical"];
% Specify file level properties
opts.ExtraColumnsRule = "ignore";
opts.EmptyLineRule = "read";
% Specify variable properties
opts = setvaropts(opts, "LastName", "WhitespaceRule", ...
"preserve");
opts = setvaropts(opts, ["LastName", "Gender", "Hospital", ...
"Smoker", "SAHS"], "EmptyFieldRule", "auto");
% Import the data
patients = readtable(filename, opts);
end
```

Notice that there is an example in the function showing how to use it. You can either specify the complete path to the file location, or if the file is in the current directory or on the MATLAB search path you need only specify the file name. In this case the patients file is on the MATLAB search path, so it can be loaded using our function with the following code.

```
T3=importfile("patients.dat");
```

Any new data file with similar information could be uploaded and analyzed without needing to be modified every time. You don't need to accept the function file name the function generator suggests, you can make it more meaningful for your project.

11.11.3 Working with Table Information

Tables are a great way to store and organize information, especially when the different data types available are utilized. For example, categorical information can be used to sort the table interactively in the variable editor, or you can use logical indexing to extract specific information. To extract all the female patients who said their health was "fair" and who were seen at the VA hospital, first create a logical index array based on the criteria. This generates an array of 1's and 0's ("true" and "false" determinations) which are then used to index into the table.

```
k=(T3.SAHS=="Fair"&
T3.Gender=="Female"&T3.Hospital=="VA Hospital");
T3(k,:)
ans = 5×10 table
```

	LastName	Gender	Age	Hospital	Height	Weight	Smoker	Systolic	Diastolic	SAHS
1	"Jones"	Female	40	VA Hospital	67	133	0	117	75	Fair
2	"Garcia"	Female	27	VA Hospital	69	131	1	123	79	Fair
3	"Rodriguez"	Female	39	VA Hospital	64	117	0	123	79	Fair
4	"Lewis"	Female	41	VA Hospital	62	137	0	114	88	Fair
5	"Watson"	Female	40	VA Hospital	64	127	1	122	91	Fair

You can perform calculations with data in tables and add it to the array. For example, the pulse pressure is the difference between the systolic and diastolic blood pressure measurements. It represents the force that the heart generates each time it contracts. To calculate the new variable `PulsePressure`, use the `Systolic` and `Diastolic` variables in the data set.

```
T3.PulsePressure = T3.Systolic - T3.Diastolic;
```

Checking the workspace reveals that the table has now grown to a 100 x 11 array.

Because it is easy to extract data from tables you can perform typical MATLAB operations on table data. For example, we could plot the `Height` versus `Weight` data.

```
plot(T3.Weight,T3.Height,'o')
title("Relationship Between Height and Weight")
xlabel("Height,inches"), ylabel("Weight, lbm")
```

which results in the plot in Figure 11.20a. The obvious grouping of the data into two clusters might lead you to think there is a relationship based on gender. Using categorical data and plotting results in the plot in Figure 11.20b. (The annotations were added in the usual way.)

```
k=T3.Gender=="Male";
j=T3.Gender~="Male";
```

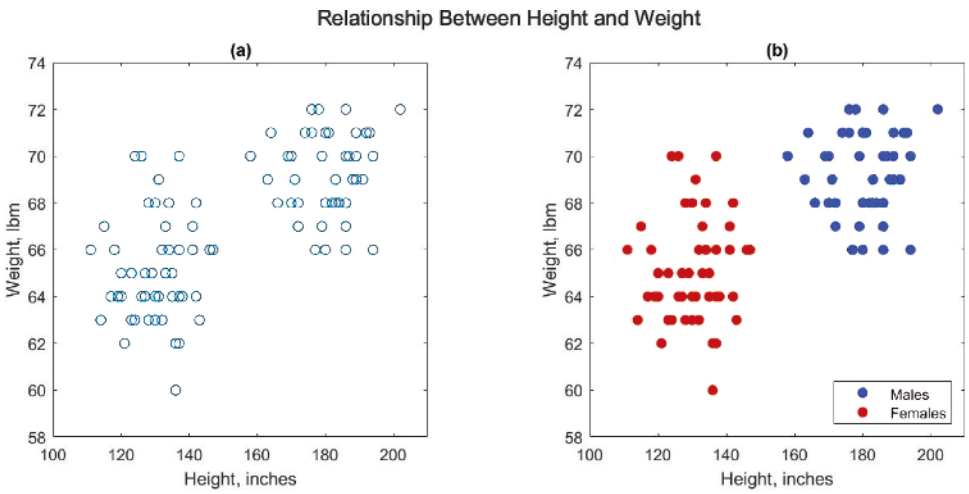


Figure 11.20
Data stored in tables can be easily used to create graphs used to evaluate data.

```
plot(T3.Weight,T3.Height,'o')
plot(T3.Weight(k),T3.Height(k),'o',MarkerFaceColor='blue')
hold on
plot(T3.Weight(j),T3.Height(j),'o',MarkerFaceColor='red')
hold off
```

The addition of color to the plot and separating out the two groups of data make the relationships clear. . . in this case so much so that we can surmise that the data were probably manufactured for illustrative purposes.

Another advantage of tables is that units can be added to the column headers. These units can be used in calculations by the symbolic algebra toolbox, described in the next chapter. You can add the units interactively by selecting the drop-down menu in the variable editor as shown in Figure 11.18, or programmatically. You can also add a description of the variable in the variable editor for documentation purposes, for example the equation used to calculate the `PulsePressure` or the definition used for a blood pressure reading.

Units can be added programmatically for all of the variables at the same time, or for each column individually using `Properties`. The properties for each table are stored in a MATLAB tabular symbolic object which can be seen by typing `<array name>.Properties`, as in

```
T3.Properties
ans =
  TableProperties with properties:
            Description: ''
               UserData: []
         DimensionNames: {'Row'  'Variables'}
          VariableNames: {'LastName'  'Gender'  'Age'
'Hospital'  'Height'  'Weight'  'Smoker'  'Systolic'
'Diastolic'  'SAHS'  'PulsePressure'}
    VariableDescriptions: {}
           VariableUnits: {}
      VariableContinuity: []
                RowNames: {}
        CustomProperties: No custom properties are set.
      Use addprop and rmprop to modify CustomProperties.
```

To determine just the dimension names dot notation is used:

```
T3.Properties.DimensionNames
ans = 1×2 cell
'Row' 'Variables'
```

The values in any of the Properties can be modified with an assignment statement. Either a string array or a cell array of character will be accepted as input.

```
T3.Properties.VariableUnits=["","","yrs","","in","lb","","mm
Hg","mm Hg","","mm Hg"];
```

To change an individual property you must first check to see how the property names are stored. From the list we retrieved notice that the property names are stored as cell arrays of characters, so the syntax would be:

```
T3.Properties.VariableUnits{'Height'} = 'inch'
```

If you check the results after changing the units you'll notice that no matter how you entered the units, either as a cell array of characters or as a string array, the results are stored as a cell array. (This was true as of 2021 but may change in the future.) Other properties can be changed in the same manner.

These units could now be used to confirm dimensional conformity in calculations, as input to labels for graphs or charts, or in actual symbolic calculations. Perhaps their most important use is for simple documentation.

There is a wealth of information about working with tables in the MATLAB documentation. Once you start creating and using data files you will want to manipulate the information, and tables are a great choice.

11.12 TIMETABLE ARRAYS

MATLAB includes a special type of array called a `timetable` designed for working with timestamped data. All of the functionality of regular tables works with **timetables**, but in addition there are functions that allow the user to manipulate data.

Consider two data sets that are included with MATLAB called `indoors` and `outdoors`. The `indoors` data is stored as a .csv file, so to load it into the workspace as a `timetable` type

```
indoors = readtimetable('indoors.csv');
```

The `outdoors` data is stored in a .mat file, so to load it type

```
load outdoors;
```

Both commands result in a `timetable` array being created in the workspace window. To examine the resulting arrays in the variable editor, use the `summary` function to see a summary of the contents, as in:

```
summary(indoors)
RowTimes:
    Time: 60×1 datetime
        Values:
            Min         2015-11-15 00:00:24
            Median      2015-11-16 11:59:45
            Max         2015-11-17 23:59:07
Variables:
    Humidity: 60×1 double
        Values:
            Min          35
            Median       37
            Max          37
    AirQuality: 60×1 double
        Values:
            Min          77
            Median       80
            Max          85
```

and

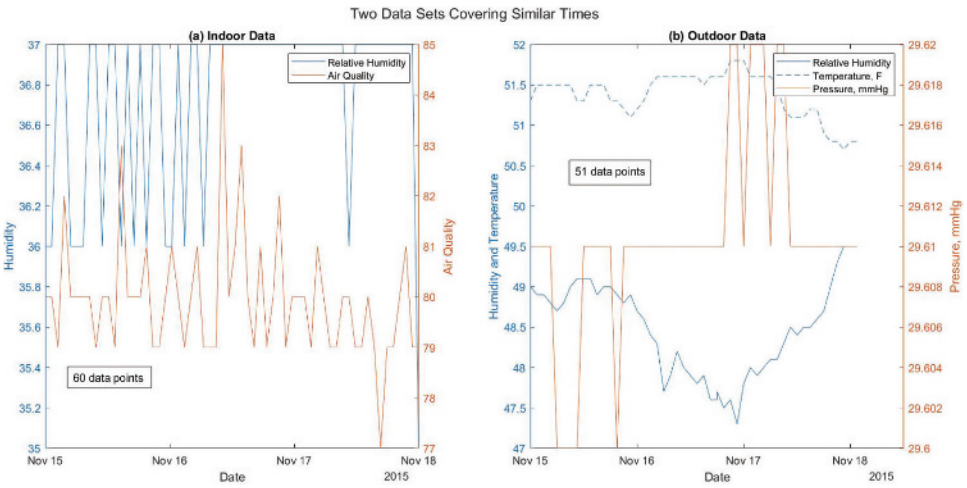


Figure 11.21
Two `datetime` arrays may contain information that covers the same time period without containing the same number of data points. MATLAB includes functions to merge the `datetime` arrays and combine the information.

```
summary(outdoors)
RowTimes:
    Time: 51×1 datetime
        Values:
            Min          2015-11-15 00:00:24
            Median       2015-11-16 13:30:24
            Max          2015-11-18 01:30:24
Variables:
    Humidity: 51×1 double
        Values:
            Min          47.3
            Median       48.5
            Max          49.5
    TemperatureF: 51×1 double
        Values:
            Min          50.7
            Median       51.5
            Max          51.8
    PressureHg: 51×1 double
        Values:
            Min          29.6
            Median      29.61
            Max         29.62
```

Although the two arrays contain data that cover approximately the same period of time, they do not have the same number of rows. In addition, the data was not collected at exactly the same time. Plots of each data set could be created separately, as shown in Figure 11.21, or the data could be layered creating a single plot, but it would be beneficial if the two data sets could be merged and values of the data estimated at equivalent times.

11.12.1 Synchronizing Timetables

To combine the two data sets into a single file, use the `synchronize` function.

```
combined_data=synchronize(indoors,outdoors)
```

The result is a 110 x 5 `timetable` since none of the times were the same in the separate arrays. Values of NaN, not a number, were added for the missing values as shown in the partial results below.

```
combined_data = 110×5 timetable
```

	Time	Humidity_indoors	AirQuality	Humidity_outdoors	TemperatureF	PressureHg
1	2015-11-15 00:00:24	36	80	49	51.3	29.61
2	2015-11-15 01:13:35	36	80	NaN	NaN	NaN
3	2015-11-15 01:30:24	NaN	NaN	48.9	51.5	29.61

The `size` function can be used to determine the dimensions of the resulting array and the `summary` function can be used to get an idea of how the two arrays were combined. Notice for example that there are 50 missing values in the column named `Humidity` _indoors. When the merged arrays both have columns with the same names MATLAB renames them so that both sets of data are preserved.

```
size(combined_data)
ans = 1×2
   110     5
summary(combined_data)
RowTimes:
    Time: 110×1 datetime
        Values:
            Min          2015-11-15 00:00:24
            Median       2015-11-16 13:03:22
            Max          2015-11-18 01:30:24
Variables:
    Humidity_indoors: 110×1 double
        Values:
            Min              35
            Median           37
            Max              37
            NumMissing       50
    AirQuality: 110×1 double
        Values:
            Min              77
            Median           80
            Max              85
            NumMissing       50
    Humidity_outdoors: 110×1 double
        Values:
            Min             47.3
            Median          48.5
            Max             49.5
            NumMissing      59
```

```
    TemperatureF: 110×1 double
        Values:
            Min             50.7
            Median          51.5
            Max             51.8
            NumMissing      59
PressureHg: 110×1 double
    Values:
            Min             29.6
            Median          29.61
            Max             29.62
            NumMissing      59
```

This version of a combined table probably isn't what is needed. . . instead it would be advantageous to combine the data and fill in the missing information with linear interpolation.

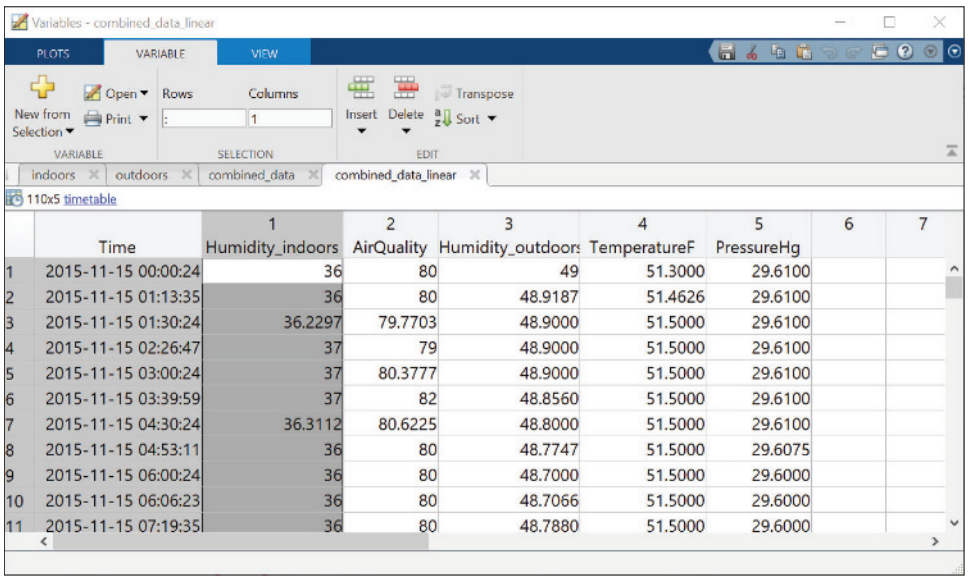
```
combined_data_linear =
synchronize(indoors,outdoors,'union','linear');
```

This also results in a 110 x 5 array. Examining the results in the variable editor (Figure 11.22) reveals that the missing data points have been replaced as expected. However, the spacing is very irregular. Some of the times are just minutes apart, and others are as much as an hour and a half apart. To create an even spacing of the data you can either define a `datetime` array with the desired time values, or you can choose from standard choices. For both options use the `retime` function. For example, to create values on the hour calculated using a cubic spline estimation technique type:

```
regular_spacing=retime(combined_data_linear,'hourly','spline');
```

The resulting array is reduced to just 75 evenly spaced values, as shown with the `size` function.

```
size(regular_spacing)
ans = 1×2
    75     5
```



	Time	1 Humidity_indoors	2 AirQuality	3 Humidity_outdoors	4 TemperatureF	5 PressureHg	6	7
1	2015-11-15 00:00:24	36	80	49	51.3000	29.6100		
2	2015-11-15 01:13:35	36	80	48.9187	51.4626	29.6100		
3	2015-11-15 01:30:24	36.2297	79.7703	48.9000	51.5000	29.6100		
4	2015-11-15 02:26:47	37	79	48.9000	51.5000	29.6100		
5	2015-11-15 03:00:24	37	80.3777	48.9000	51.5000	29.6100		
6	2015-11-15 03:39:59	37	82	48.8560	51.5000	29.6100		
7	2015-11-15 04:30:24	36.3112	80.6225	48.8000	51.5000	29.6100		
8	2015-11-15 04:53:11	36	80	48.7747	51.5000	29.6075		
9	2015-11-15 06:00:24	36	80	48.7000	51.5000	29.6000		
10	2015-11-15 06:06:23	36	80	48.7066	51.5000	29.6000		
11	2015-11-15 07:19:35	36	80	48.7880	51.5000	29.6000		

Figure 11.22
The variable editor is one option for viewing the contents of either a `table` or `timetable` array. The contents of this `timetable` have been modified using linear interpolation to estimate missing values.

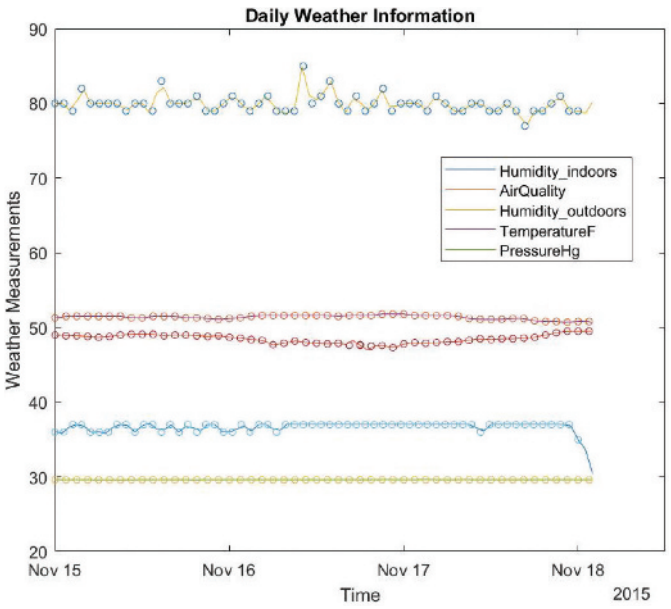


Figure 11.23
The weather information in the `indoors` and `outdoors` timetable arrays was merged, missing values were estimated and the resulting array was retimed so that the values were evenly spaced and smoothed based on a cubic spline regression. The results were plotted with the original data shown as circles for comparison.

Consult the MATLAB documentation for more details on `retime` for other options. Figure 11.23 shows a plot of the combined smoothed data with the original data points superimposed for comparison.

SUMMARY

MATLAB's primary data structure is the array. Within the array, MATLAB allows the user to store a number of different types of data. The default numeric data type is the double-precision floating-point number, usually referred to as a double. MATLAB also supports single-precision floating-point numbers, as well as eight different types of integers. Text information is also stored in arrays, either as single characters in a character array, or as strings of characters in a string array. In addition to numeric and character data, MATLAB includes a variety of other data types, including symbolic, datetime, logical, and categorical arrays.

All these kinds of data can be stored as two-dimensional arrays. Scalar and vector data are actually stored as two-dimensional arrays—they just have a single row or column. MATLAB also allows the user to store data in multidimensional arrays. Each two-dimensional slice of a three-dimensional or higher array is called a page.

In general, all data stored in a MATLAB array must be of the same type. If character and numeric data are mixed, the numeric data are changed to character data on the basis of their ASCII-equivalent decimal values. When calculations are attempted on combined character and numeric data, the character data are converted to their ASCII equivalents.

MATLAB offers three array types that can store multiple types of data at the same time: table, cell, and structure arrays. Cell arrays use curly brackets, `{ }` as array constructors. Structure arrays depend on named fields. Table arrays are especially useful for tabular data, such as that stored in spreadsheets. Timetables are a special type of table used to work with arrays containing timestamped rows. Table arrays are becoming the dominant array type for storing multiple types of data in MATLAB.

MATLAB SUMMARY

The following MATLAB summary lists and briefly describes all the special characters, commands, and functions that are defined in this chapter:

Special Characters

`{ }`	cell-array constructor
`' '`	character information
`" "`	string information
	character array
	string array
	numeric array
	symbolic array
	logical array
	sparse array
	duration array
	datetime array
	categorical array
	cell array
	structure array
	table array

Commands and Functions

`cell2table`	converts a cell array to a table array
`celldisp`	displays the contents of a cell array
`char`	creates a padded character array
`cumsum`	finds the cumulative sum of the members of an array
`categories`	determines the categories available in categorical arrays
`categorical`	creates a categorical array
`datetime`	creates a datetime array
`double`	changes an array to a double-precision array
`duration`	creates a duration array
`eye`	creates an identity matrix
`format rat`	converts the display format to rational numbers (fractions)
`int16`	16-bit signed integer
`int32`	32-bit signed integer
`int64`	64-bit signed integer
`int8`	8-bit signed integer
`intmax`	determines the largest integer that can be stored in MATLAB
`intmin`	determines the smallest integer that can be stored in MATLAB
`num2str`	converts a numeric array to a character array
`Properties`	when appended to a structure variable name it lists the available properties, as in name Properties.

(*continued*)

(Continued)

Commands and Functions	
`realmax`	determines the largest real number that can be expressed in MATLAB
`realmin`	determines the smallest real number that can be expressed in MATLAB
`readtable`	imports the contents of a rectangular data file from a variety of formats
`readtimetable`	imports the contents of a data file as a timetable
`retime`	distributes the values in a timetable evenly based on a specified strategy
`single`	changes an array to a single-precision array
`summary`	lists the variables in a table array and a summary of their contents
`sparse`	converts a full-format array to a sparse-format array
`squeeze`	removes singleton dimensions from multidimensional arrays
`str2num`	converts a character array to a numeric array
`string`	creates a string array
`struct2table`	converts a structure array to a table array
`synchronize`	combines timetables into a single array
`table`	creates a table array
`table2cell`	converts a table array to a cell array
`table2struct`	converts a table array to a structure array
`uiimport`	opens the file import wizard
`uint16`	16-bit unsigned integer
`uint32`	32-bit unsigned integer
`uint64`	64-bit unsigned integer
`uint8`	8-bit unsigned integer

KEY TERMS

ASCII
base 2
binary
cell
character
class
complex numbers
data type
double-precision
drawers
EBCDIC
floating-point numbers
IEEE
integer
legacy code
logical data
pages
rational numbers
single-precision
string
structure
symbolic
table
timetable

PROBLEMS

Numeric Data Types

11.1 Calculate the sum (not the partial sums) of the first 10 million terms in the harmonic series

$$\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\cdots+\frac{1}{n}+\cdots$$

using both double- and single-precision numbers. Compare the results. Explain why they are different.

11.2 Define an array of the first 10 integers, using the int8 type designation. Use these integers to calculate the first 10 terms in the harmonic series. Explain your results.

11.3 Explain why it is better to allow MATLAB to default to double-precision floating-point number representations for most engineering calculations than to specify single and integer types.

11.4 Complex numbers are automatically created in MATLAB as a result of calculations. They can also be entered directly, as the addition of a real and an imaginary number, and can be stored as any of the numeric data types. Define two variables: a single- and a double-precision complex number, as

$$\textbf{doublea} = 5 + 3\textbf{i}$$

$$\textbf{singlea} = \textbf{single}(5 + 3i)$$

Raise each of these numbers to the 100th power. Explain the difference in your answers.

Character and String Data

11.5 Use an Internet search engine to find a list showing the binary equivalents of characters in both ASCII and EBCDIC. Briefly outline the differences in the two coding schemes.

11.6 Sometimes, it is confusing to realize that numbers can be represented as both numeric data and character data. Use MATLAB to express the number 85 as a character array.

(a) How many elements are in this array?
(b) What is the numeric equivalent of the character 8?
(c) What is the numeric equivalent of the character 5?

11.7 **(a)** Create a padded character array with five different names.
(b) Create a two-dimensional numeric array called `birthdays` to represent the birthday of each person. For example, your array might look something like this:

```
birthdays=
     6    11    1983
     3    11    1985
     6    29    1986
    12    12    1984
    12    11    1987
```

(c) Use the `num2str` function to convert `birthdays` to a character array.
(d) Use the `disp` function to display a table of names and birthdays.
(e) Create a datetime array to represent birthdays.

11.8 Repeat the previous problem using a string array instead of a padded character array

(a) Create a string array with five different names.
(b) Create a two-dimensional numeric array called `birthdays` to represent the birthday of each person. For example, your array might look something like this:

```
birthdays=
     6    11    1983
     3    11    1985
     6    29    1986
    12    12    1984
    12    11    1987
```

(c) Use `fprintf` to display a table of names and birthdays. (There is no need to combine the name and birthday array.)

11.9 Imagine that you have the following character array, which represents the dimensions of some shipping boxes:

```
box_dimensions =
box1   1   3   5
box2   2   4   6
box3   6   7   3
box4   1   4   3
```

You need to find the volumes of the boxes to use in a calculation to determine how many packing "peanuts" to order for your shipping department. Since the array is a 4×12 character array, the character representation of the numeric information is stored in columns 6 to 12. Use the `str2num` function to convert the information into a numeric array, and use the data to calculate the volume of each box. (You'll need to enter the `box_dimensions` array as text data, using the `char` function.)

11.10 Consider the file called *thermocouple.dat* as shown in Table P11.10:

(a) Create a program that:

- Loads *thermocouple.dat* into MATLAB.
- Determines the size (number of rows and columns) of the file.
- Extracts each set of thermocouple data and stores it into a separate file. Name the various files *thermocouple1.mat, thermocouple2.mat,* etc.

(b) Your program should be able to accept a two-dimensional file of any size. Do not assume that there are only three columns; let the program determine the array size and assign appropriate file names.

Table P11.10 Thermocouple data

Thermocouple 1	Thermocouple 2	Thermocouple 3
84.3	90.0	86.7
86.4	89.5	87.6
85.2	88.6	88.3
87.1	88.9	85.3
83.5	88.9	80.3
84.8	90.4	82.4
85.0	89.3	83.4
85.3	89.5	85.4
85.3	88.9	86.3
85.2	89.1	85.3
82.3	89.5	89.0
84.7	89.4	87.3
83.6	89.8	87.2

11.11 Create a program that encodes text entered by the user and saves it into a file. Your code should add 10 to the decimal equivalent value of each character entered.

11.12 Create a program to decode a message stored in a data file by subtracting 10 from the decimal equivalent value of each character.

Table P11.13 Climate Date from the Denver International Airport for 2015

DATE	EMXP	TPCP	TSNW	EMXT	EMNT	MMXT	MMNT	MNTM
20150101	0.12	0.37	7.2	75	−9.8	46.8	21.2	34
20150201	0.37	1.25	22.4	73.9	−5.8	46.6	21.2	34
20150301	0.34	0.79	2.9	81	2.1	58.8	31.3	45.1
20150401	1.22	2.65	5.3	79	27.1	62.1	35.4	48.7
20150501	1.08	3.76	4	84.9	27.1	63.3	42.8	53.1
20150601	0.99	2.54	0	93.9	51.1	82.9	56.1	69.4
20150701	0.33	1.07	0	97	52	87.6	57.9	72.7
20150801	0.68	1.18	0	98.1	43	89.8	58.3	74.1
20150901	0.07	0.11	0	91.9	44.1	85.3	53.6	69.4
20151001	0.79	1.76	0	87.1	28.2	70.2	42.8	56.5
20151101	0.5	2.13	11.2	75	3.2	51.3	25.7	38.5
20151201	0.32	0.71	11.3	69.1	0.1	41	18.1	29.5

Datetime Arrays

11.13 Consider the data shown in Table P11.13 representing climate information gathered at the Denver International Airport during 2015 by NOAA, and available online from the NOAA National Centers for Environmental Information. The data may be available to you in an Excel spreadsheet called *Denver_Climate_Data_2015*. If so, download the data into column vectors.

(a) Create a datetime array from the `DATE` column of the table, and a numeric array from the monthly total precipitation column, `TPCP`. You will need to convert the `DATE` information using the `ConvertFrom` property in the `datetime` function.

(b) Plot `TPCP` versus `DATE` as a bar graph, with appropriate annotations such as a title and axis labels.

(c) On a separate *x*–*y* graph, plot the extreme maximum temperature (`EMXT`), extreme minimum temperature (`EMNT`), and the mean temperature (`MNTM`) versus the date. As always add appropriate annotations, such as the title, axis labels, and a legend.

EMXP	Extreme maximum daily precipitation, inches
TPCP	Total precipitation for the month, inches
TSNW	Total snowfall for the month, inches
EMXT	Extreme maximum temperature, °F
EMNT	Extreme minimum temperature, °F
MMXT	Monthly mean maximum temperature, °F
MMNT	Monthly mean minimum temperature, °F
MNTM	Monthly mean temperature, °F

11.14 Climate data such as that reported for Denver in the previous problem is available for most major U.S. cities. Download the data for the most recent year.

(a) Create a `datetime` array from the `DATE` column of the table, and a numeric array from the monthly total precipitation column, `TPCP`. You will need to convert the `DATE` information using the `ConvertFrom` property in the `datetime` function.

(b) Plot the data as a bar graph, with appropriate annotations such as a title and axis labels.

(c) On a separate *x*–*y* graph, plot the extreme maximum temperature (`EMXT`), extreme minimum temperature (`EMNT`), and the mean temperature (`MNTM`) versus the date. As always add appropriate annotations, such as the title, axis labels, and a legend.

Multidimensional Arrays

11.15 Create each of the following arrays:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix}, C = \begin{bmatrix} 3 & 16 \\ 9 & 12 \end{bmatrix}$$

(a) Combine them into one large $2 \times 2 \times 3$ multidimensional array called `ABC`.

(b) Extract each column 1 into a 2×3 array called `Column_A1B1C1`.

(c) Extract each row 2 into a 3×2 array called `Row_A2B2C2`.

(d) Extract the value in row 1, column 2, page 3.

11.16 A college professor would like to compare how students perform on a test given every year. Each year, the data is stored in a two-dimensional array. The first and second year's data are as follows:

Year 1	Question 1	Question 2	Question 3	Question 4
Student 1	3	6	4	10
Student 2	5	8	6	10
Student 3	4	9	5	10
Student 4	6	4	7	9
Student 5	3	5	8	10

Year 2	Question 1	Question 2	Question 3	Question 4
Student 1	2	7	3	10
Student 2	3	7	5	10
Student 3	4	5	5	10
Student 4	3	3	8	10
Student 5	3	5	2	10

(a) Create a two-dimensional array called `year1` for the first year's data, and another called `year2` for the second year's data.

(b) Combine the two arrays into a three-dimensional array with two pages, called `testdata`.

(c) Use your three-dimensional array to perform the following calculations:

- Calculate the average score for each question, for each year, and store the results in a two-dimensional array. (Your answer should be either a 2×4 array or a 4×2 array.)
- Calculate the average score for each question, using all the data.

- Extract the data for Question 3 for each year, and create an array with the following format:

	Question 3, Year 1	Question 3, Year 2
Student 1		
Student 2		
and so on		

11.17 If the teacher described in the preceding problem wants to include the results from a second and third test in the array, a four-dimensional array would need to be created. (The fourth dimension is sometimes called a *drawer*.) All the data are included in a file called *test_results.mat* consisting of six two-dimensional arrays similar to those described in Problem 11.8. The array names are

```
test1year1
test2year1
test3year1
test1year2
test2year2
test3year2
```

Organize these data into a four-dimensional array that looks like the following:

dimension 1	(row)	student
dimension 2	(column)	question
dimension 3	(page)	year
dimension 4	(drawer)	test

(a) Extract the score for Student 1, on Question 2, from the first year, on Test 3.
(b) Create a one-dimensional array representing the scores from the first student, on Question 1, on the second test, for all the years.
(c) Create a one-dimensional array representing the scores from the second student, on all the questions, on the first test, for Year 2.
(d) Create a two-dimensional array representing the scores from all the students, on Question 3, from the second test, for all the years.

Cell Arrays

11.18 Create a cell array called `sample_cell` to store the following individual arrays:

$$A = \begin{bmatrix} 1 & 3 & 5 \\ 3 & 9 & 2 \\ 11 & 8 & 2 \end{bmatrix} \text{ (a double} - \text{precision floating-point array)}$$

$$B = \begin{bmatrix} fred & ralph \\ ken & susan \end{bmatrix} \text{ (a padded character array)}$$

$$C = \begin{bmatrix} 4 \\ 6 \\ 3 \\ 1 \end{bmatrix} \text{ (an } \texttt{int8} \text{ integer array)}$$

(a) Extract array `A` from `sample_cell`.

(b) Extract the information in array `C`, row 3, from `sample_cell`.
(c) Extract the name "fred" from `sample_cell`. Remember that the name fred is a 1×4 array, not a single entity.

11.19 Cell arrays can be used to store character information without padding the character arrays. Create a separate character array for each of the strings

aluminum
copper
iron
molybdenum
cobalt

and store them in a cell array.

11.20 Consider the following information about metals:

Metal	Symbol	Atomic Number	Atomic Weight	Density $\left(g / cm^3\right)$	Crystal Structure
Aluminum	Al	13	26.98	2.71	FCC
Copper	Cu	29	63.55	8.94	FCC
Iron	Fe	26	55.85	7.87	BCC
Molybdenum	Mo	42	95.94	10.22	BCC
Cobalt	Co	27	58.93	8.9	HCP
Vanadium	V	23	50.94	6.0	BCC

(a) Create the following arrays:
- Store the name of each metal into an individual character array, and store all these character arrays into a cell array.
- Store the symbol for all these metals into a single padded character array.
- Store the atomic number into an `int8` integer array.
- Store the atomic weight into a double-precision numeric array.
- Store the density into a single-precision numeric array.
- Store the crystal structure into a categorical array.

(b) Group the arrays you created in part (a) into a single cell array.
(c) Extract the following information from your cell array:
- Find the name, atomic weight, and structure of the fourth element in the list.
- Find the names of all the elements stored in the array.
- Find the average atomic weight of the elements in the table. (Remember, you need to extract the information to use in your calculation from the cell array.)

Structure Arrays

11.21 Store the information presented in the previous problem in a structure array. Use your structure array to determine the element with the maximum density.

11.22 Create a program that allows the user to enter additional information into the structure array you created in the previous problem. Use your program to add the following data to the array:

Metal	Symbol	Atomic Number	Atomic Weight (g/mol)	Density $\left(g/cm^3\right)$	Crystal Structure
Lithium	Li	3	6.94	0.534	BCC
Germanium	Ge	32	72.59	5.32	Diamond cubic
Gold	Au	79	196.97	19.32	FCC

11.23 Use the structure array you created in the previous problem to find the element with the maximum atomic weight.

Table Arrays

11.24 Store the combined information from the previous problems as shown in Table P11.24 in a table array.

- The metal column should be stored as a string
- The symbol column should be stored as a character array
- The atomic number column should be stored as an integer array
- The atomic weight column should be stored as a double array
- The density column should be stored as a double array
- The crystal structure should be stored as a categorical array.

(a) Use the find function to determine which elements are BCC. Create a report using `fprintf`.

(b) Create a plot of atomic number on the *x*-axis versus atomic weight on the *y*-axis.

11.25 Import the Denver climate data described in Problem 11.13 into a table array. Assign appropriate data types to each column. Note: The `DATE` column will need to be converted into a `datetime` array.

(a) Plot the total monthly snowfall versus date as a bar graph, with appropriate annotations such as a title and axis labels.

(b) On a separate *x*–*y* graph, plot the mean monthly maximum temperature (`MMXT`), mean monthly minimum temperature (`MMNT`), and the mean temperature (`MNTM`) versus the date. As always add appropriate annotations, such as the title, axis labels and a legend.

(c) Use a string array to create an appropriate title for your graphs that includes two lines of text.

Table P11.24 Element Data

Metal	Symbol	Atomic Number	Atomic Weight, (g/mol)	Density $\left(g/cm^3\right)$	Crystal Structure
Aluminum	Al	13	26.98	2.71	FCC
Copper	Cu	29	63.55	8.94	FCC
Iron	Fe	26	55.85	7.87	BCC
Molybdenum	Mo	42	95.94	10.22	BCC
Cobalt	Co	27	58.93	8.9	HCP
Vanadium	V	23	50.94	6.0	BCC
Lithium	Li	3	6.94	0.534	BCC
Germanium	Ge	32	72.59	5.32	Diamond cubic
Gold	Au	79	196.97	19.32	FCC

CHAPTER

12 Symbolic Mathematics

Objectives

After reading this chapter, you should be able to:

- Create and manipulate symbolic variables.
- Factor and simplify mathematical expressions.
- Solve symbolic expressions.
- Solve systems of equations.
- Create and use symbolic functions.
- Determine the symbolic derivative of an expression.
- Integrate an expression symbolically.
- Solve differential equations symbolically.

INTRODUCTION

MATLAB's symbolic toolbox allows us to create **symbolic** arrays, which are used to manipulate symbolic expressions, to simplify them, to solve them symbolically, and to evaluate them numerically. It also allows us to take **derivatives**, to **integrate**, and to perform **linear** algebraic manipulations. More advanced features include LaPlace transforms, Fourier transforms, and variable-precision arithmetic. Although the Symbolic Toolbox is optional with the professional version of MATLAB, it comes standard with the student version of the software.

12.1 SYMBOLIC ALGEBRA

KEY IDEA
The symbolic toolbox is an optional component of the professional version, but is standard with the student version.

Symbolic mathematics is used regularly in math, engineering, and science classes. It is often preferable to manipulate equations symbolically before you substitute values for **variables**. For example, consider the equation

$$y = \frac{2(x + 3)^2}{x^2 + 6x + 9}$$

At first glance, y appears to be a fairly complicated function of x. However, if you expand the quantity $(x + 3)^2$, it becomes apparent that you can simplify the equation to

$$y = \frac{2*(x + 3)^2}{x^2 + 6x + 9} = \frac{2*(x^2 + 6x + 9)}{(x^2 + 6x + 9)} = 2$$

You may or may not want to perform this simplification, because, in doing so, you lose some information. For example, for values of x equal to -3, y is undefined, since $x + 3$ becomes 0, as does $x^2 + 6x + 9$. Thus,

$$y = \frac{2(-3 + 3)^2}{9 - 18 + 9} = 2\frac{0}{0} = \text{undefined}$$

MATLAB's symbolic algebra capabilities allow you to perform this simplification, or to manipulate the numerator and denominator separately.

Relationships are not always constituted in forms that are so easy to solve. For instance, consider the equation

$$k = k_0 e^{-Q/(RT)}$$

KEY IDEA
MATLAB makes it easy to solve equations symbolically.

If you know the values of k_0, Q, R, and T, it's easy to solve for k. It's not so easy to find T, when the values of k, k_0, R, and Q are known. Manipulating the relationship to get T on the left-hand side of the equation requires a little algebra

$$\ln(k) = \ln(k_0) - \frac{Q}{RT}$$

$$T = \frac{Q}{R\ln(k_0/k)}$$

Although solving for T was manually awkward, it's easy with MATLAB's symbolic capabilities. In this chapter, we'll explore the variety of ways to use the Symbolic Toolbox to perform symbolic mathematical tasks, and how to convert the results to **numeric** values.

12.1.1 Creating Symbolic Objects

Before solving any equations, it's necessary to create some symbolic **objects**. There are several different kinds of symbolic objects available to use in MATLAB, including simple variables, **expressions**, **functions** (which are a kind of expression), and **equations** among others. The most basic are simple variables. Simple symbolic variables can be created in two ways. For example, to create the symbolic variable `x`, type either

```
x = sym('x')
```

or

```
syms x
```

Both techniques set the character `'x'` equal to the symbolic variable `x`. The first strategy assigns the character 'x' the symbolic data type with the `sym` command, and then assigns it (with the assignment **operator** =) to a computer memory location called `x`. The second strategy completes both tasks in a single statement. For most programmers the second approach is the more straight forward. Symbolic objects representing entire expressions can be created by combining existing symbolic variables, as in

```
y = 2*(x + 3)^2/(x^2 + 6*x + 9)
```

Symbolic functions are created using a similar approach. First define a function name and its input using `syms`:

```
syms fun(x,z)
```

KEY IDEA
There are multiple kinds of symbolic objects, including variables, expressions, functions, and equations.

Not only is the symbolic function created, but the symbolic variables specified as the function input are created at the same time. The second step is to define the function itself. For example:

```
fun(x,z)=2*x^2+4*z
```

Thus

```
fun(3,4)
```

returns

```
ans =
34
```

This can also be accomplished using numeric variables that exist in the workspace. If the numeric variables `a` and `b` are specified:

```
a = 2; b=2;
```

they can be used in the function definition:

```
fun(x,z)=a*x^2+4*z
```

Substituting in the same values yields the same results.

```
fun(3,4)
```

returns

```
ans =
34
```

You can change the value of `a` and `b` in the future without affecting the function definition.

HINT

If the symbolic variables used in a function have already been defined you can skip the `syms` definition step and just define the function. For example, if `x` is already a symbolic variable:

```
f(x) = x^2 + 2
```

creates the symbolic function `f` in a single step.

KEY IDEA
Expressions are different from equations.

Notice in the workspace window (see Figure 12.1) that `x`, `y`, and `z` are listed as symbolic variables, `fun` is listed as a symbolic function, and that the array size for each is 1×1. They are each *symbolic* objects. Notice also that `a` and `b` are listed as numeric arrays (doubles).

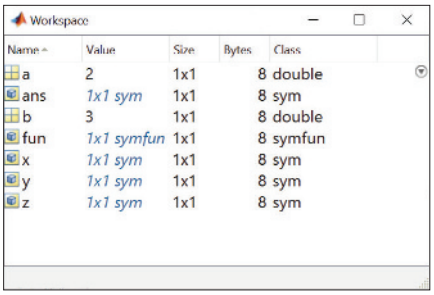


Figure 12.1
Symbolic objects are identified in the workspace window with a "box" icon.

The `syms` command can also be used to create multiple symbolic variables at the same time:

```
syms Q R T k0
```

These variables could be combined mathematically to create another symbolic variable, `k`:

```
k = k0*exp(-Q/(R*T))
```

Notice that the standard algebraic operators were used, not the array operators, such as **.*** or `.^`. This makes sense since array operators specify that corresponding elements in arrays are used in the associated calculations—a situation that does not apply here.

In this example, the *expression* `k0 * exp(-Q/(R * T))` is assigned to the variable name `k`. An entire *equation* can also be assigned a name. For example, we can define the ideal gas law by first defining symbolic variables, then combining them into an equation

KEY IDEA
The symbolic toolbox uses standard algebraic operators.

```
syms P V n R Temp
ideal_gas_law = P*V == R*Temp*n
```

Notice that the `==` operator is used in the equation definition, not the assignment operator, `=`.

Three approaches to creating symbolic objects that store entire expressions were discussed in this section. In the first the expression was given a name, for example:

```
k = k0*exp(-Q/(R*T))
```

In the second an entire equation was defined and given a name, for example:

```
ideal_gas_law = P*V == R*Temp*n
```

Each side of the equation is a separate expression, connected with the equality operator, `==`.

The third approach to symbolic object definition discussed in this section was to create a *function*, which allows different values to be substituted for the independent variables.

```
fun(x,z)=2*x^2+4*z
```

The function is a type of expression. All three approaches for symbolic object definitions have their uses. In general, if you are interested in algebraic manipulations, simple expressions and equations work best. If you are interested in substituting

in values or in performing calculations over specified ranges of the independent variables, functions work best.

HINT

One idiosyncracy of the Symbolic Toolbox is that a number of commonly used variables are reserved. They can be overwritten, however, if you try to use them inside expressions or equations you may run into problems. Try to avoid these names:

```
D, E, I, O, beta, zeta, theta, psi, gamma, Ci, Si, Ei
```

HINT

Notice that when you use symbolic variables, MATLAB does not indent the result, unlike the format used for numeric results. This can help you keep track of variable types without referring to the workspace window.

PRACTICE EXERCISE 12.1

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Create the following symbolic variables, using the `syms` command:

```
x, a, b, c, d
```

2. Verify that the variables you created in Exercise 1 are listed in the workspace window as symbolic variables. Use them to create the following symbolic expressions:

```
ex1 = x^2-1
ex2 = (x+1)^2
ex3 = a*x^2-1
ex4 = a*x^2 + b*x + c
ex5 = a*x^3 + b*x^2 + c*x + d
ex6 = sin(x)
```

3. Create the following symbolic functions. Because you have already created the symbolic variables there is no need to define the function names before you create the function itself.

```
fex1(x)  = x^2-1
fex2(x)  = (x+1)^2
fex3(x)  = a*x^2-1
fex4(x)  = a*x^2 + b*x + c
fex5(x)  = a*x^3 + b*x^2 + c*x + d
fex6(x)  = sin(x)
```

4. Create the following symbolic equations:

```
eq1 = x^2==1
eq2 = (x+1)^2==0
eq3 = a*x^2==1
```

```
eq4 = a*x^2 + b*x + c==0
eq5 = a*x^3 + b*x^2 + c*x + d==0
eq6 = sin(x)==0
```

Save the variables, expressions, functions, and equations you created in this practice to use in practice exercises later in the chapter.

12.1.2 Manipulating Symbolic Expressions and Symbolic Equations

First, recall how expressions and equations differ. Equations are set equal to something; expressions (including functions) are not. The variable `ideal_gas_law` has been set equal to an equation. If you type in

```
ideal_gas_law
```

MATLAB will respond

```
ideal_gas_law =
P*V == R*Temp*n
```

However, if you type in

```
k
```

MATLAB responds

```
k =
k0*exp(-Q/(R*T))
```

or if you type in

```
y
```

MATLAB responds

```
y =
2*(x+3)^2/(x^2+6*x+9)
```

The variables `k` and `y` are *expressions*, but the variable `ideal_gas_law` is an *equation*. Most of the time you will be working with symbolic expressions. As discussed previously, functions are a type of expression, because they are not set equal to anything. If you type in

```
fun
```

MATLAB responds

```
fun(x, z) =
2*x^2 + 4*z.
```

Most MATLAB functions that accept expressions as input will take them in either the form of a simple expression or as a function.

MATLAB has a number of functions designed to manipulate symbolic variables, including functions to expand or factor expressions, and to simplify expressions.

KEY IDEA
There is no need to redefine `x` as a symbolic variable every time you use it.

Simplification Functions

What constitutes a "simple" equation is not always obvious. The `simplify` function simplifies each part of an expression or equation, using MATLAB's built-in simplification rules. For example, assume that `z` has been defined as

```
syms a
z = a+2*a+(a-3)^2-(a^2+2*a+1)
```

MATLAB automatically does some combining of terms and returns

```
z =
a+(a-3)^2-a^2-1
```

The command

```
simplify (z)
```

returns

```
ans =
8-5*a
```

Similarly, if the function, `f2` has been defined as:

```
syms f2(x)
f2(x) = (x-2)*(x+2)
```

then

```
simplify(f2(x))
```

returns

```
ans(x) =
x^2-4
```

The `simplify` function works with equations too. If the equation `w` has been defined as

KEY IDEA
MATLAB defines the simplest representation of an expression as the shortest version of the expression.

```
syms x
w = x^3-1 == (x-3)*(x+3)
```

then

```
simplify(w)
```

returns

KEY IDEA
Many, but not all, symbolic functions work for both expressions, functions, and equations.

```
ans =
x^3+8==x^2
```

The `simplify` function is the most general simplification strategy available; however, other strategies and their corresponding functions are listed in Table 12.1.

Table 12.1 Functions Used to Manipulate Expressions and Equations

`expand(S)`	Multiplies out all the portions of the expression or equation	`syms x`
		`expand ((x-5) * (x+5))`
		`ans =`
		`x^2-25`

`factor(S)`	Factors the expression or equation. The result is an array of the factors.	`syms x`
		`factor (x^3-1)`
		`ans =`
		`x-1,x^2+x+1`

`collect(S)`	Collects like terms	`syms x`
		`s = 2*(x+3)^2+x^2+6*x+9`
		`collect(S)`
		`ans =`
		`3*x^2+18*x+27`

`simplify(S)`	Simplifies in accordance with MATLAB's simplification rules	`syms x`
		`simplify(exp(log(x)))`
		`ans =`
		`a`

`numden(S)`	Finds the numerator of an expression; this function is not valid for equations	`syms x`
		`numden((x-5/(x+5))`
		`ans =`
		`x-5`

`[num,den]=numden(s)`	Finds both the numerator and the denominator of an expression; this function is not valid for equations	`syms x`
		`[num,den] = numden((x-5)/(x+5))`
		`num =`
		`x-5`
		`den =`
		`x+5`

HINT

A shortcut to create a symbolic polynomial is the `poly2sym` function. This function requires a vector, `a`, as input and creates a polynomial of the form $a_1x^{n-1} + a_2x^{n-2} + \ldots + a_n$, using the vector for the coefficients of each term of the polynomial. For example,

```
a = [1,3,2];
b = poly2sym(a)
b =
x^2+3*x+2
```

Similarly, the `sym2poly` function converts a polynomial into a vector of coefficient values:

```
c = sym2poly(b)
c =
   1    3    2
```

PRACTICE EXERCISE 12.2

Use the variables defined in Practice Exercise 12.1 in these exercises. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Multiply `ex1` by `ex2`, and name the result `y1`.
2. Divide `ex1` by `ex2`, and name the result `y2`.
3. Use the `numden` function to extract the numerator and denominator from `y1` and `y2`.
4. Use the `factor`, `expand`, `collect`, and `simplify` functions on `y1`, `y2`.

(*continued*)

5. Repeat problems 1 through 4 using the symbolic functions created in Practice Exercise 12.1 (`fex1` and `fex2`).

12.2 SOLVING EXPRESSIONS AND EQUATIONS

A highly useful function in the symbolic toolbox is `solve`. It can be used to determine the **roots** of expressions, to find numerical answers when there is a single variable, and to solve for an unknown symbolically. The `solve` function can also solve systems of equations, both linear and **nonlinear**. When paired with the substitution function (`subs`), the `solve` function allows the user to find analytical solutions to a variety of problems.

12.2.1 The `solve` Function

When used with an expression, the `solve` function sets the expression equal to zero and solves for the roots. For example (assuming that `x` has already been defined as a symbolic variable), if

```
E1 = x-3
```

then

```
solve(E1)
```

returns

```
ans =
3
```

The `solve` function can be used either with an expression name or by creating a symbolic expression directly in the `solve` function. Thus,

```
solve(x^2-9)
```

returns

```
ans =
-3
3
```

Notice that `ans` is a 2×1 symbolic array. You can readily solve symbolic expressions with more than one variable. For example, for the quadratic expression $ax^2 + bx + c$,

```
syms x a b c
solve(a*x^2+b*x+c)
```

returns

```
ans =
-(b+(b^2-4*a*c)^(1/2))/(2*a)
-(b-(b^2-4*a*c)^(1/2))/(2*a)
```

KEY IDEA
MATLAB solves preferentially for *x*.

MATLAB preferentially solves for `x`. If there is no `x` in the expression, MATLAB finds the variable alphabetically closest to `x`. To specify a different solution variable,

include it in the second field. For instance, to solve the quadratic expression for `a`, the command

```
solve(a*x^2+b*x+c, a)
```

returns

```
ans =
-(c+b*x)/x^2
```

KEY IDEA
Even when the result of the `solve` function is a number, it is still stored as a symbolic variable.

To solve an expression set equal to something besides zero, there are two approaches. If the equation is simple, transform it into an expression by subtracting the right-hand side from the left-hand side. For example,

$$5x^2 + 6x + 3 = 10$$

could be reformulated as

$$5x^2 + 6x - 7 = 0$$

```
solve(5*x^2+6*x-7)
ans =
-(2*11^(1/2))/5-3/5
(2*11^(1/2))/5-3/5
```

Alternatively, the equation can be entered directly into `solve`, or can be assigned a name to be used as the input. It is important to use the comparison operator, == in the equation. For example:

```
e2 = 5*x^2+6*x+3 == 10
solve(e2)
```

returns

```
ans =
-(2*11^(1/2))/5-3/5
(2*11^(1/2))/5-3/5
```

Notice that in both cases, the results are expressed as simply as possible, using fractions (i.e., rational numbers). In the workspace, `ans` is listed as a 2×1 symbolic array. To convert the result to a double-precision floating point number, use the `double` function:

```
double(ans)
ans =
0.7266
-1.9266
```

The `solve` function is particularly useful with symbolic expressions having multiple variables:

```
syms P P0 r t
e3 = P==P0*exp(r*t)
solve(e3,t)
```

```
ans =
log(P/P0)/r
```

(Recall that the `log` function is a natural log.)

It is often useful to redefine a variable, such as `t`, in terms of the other variables:

```
t = solve(e3,t)
t =
log(P/P0)/r
```

PRACTICE EXERCISE 12.3

Use the variables and expressions defined in Practice Exercise 12.1 to complete these exercises. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Use the `solve` function to find the roots for both `ex1` and `fex1` and to solve `eq1` for `x`.
2. Use the `solve` function to find the roots for both `ex2` and `fex2` and to solve `eq2` for `x`.
3. Use the `solve` function to solve `ex3`, `fex3`, and `eq3` for both `x` and `a`.
4. Use the `solve` function to solve `ex4`, `fex4`, and `eq4` for both `x` and `a`.
5. Both `ex4` and `eq4` represent the quadratic equation—the general form of a second-order polynomial. Expression/equation 5 in these exercises is the general form of a third-order polynomial, and the solution is significantly more complicated. When you use the `solve` function the result is not informative (try it). You will need to specify the polynomial order to force a solution `solve(ex5,'MaxDegree',3)`
6. Use the `solve` function to solve `ex6`, `fex6`, and `eq6`. On the basis of your knowledge of trigonometry, comment on this solution.

EXAMPLE 12.1

USING SYMBOLIC MATH

MATLAB's symbolic capability allows us to let the computer do the math. Consider the equation for reaction rate constants:

$$k = k_0 \exp\left(\frac{-Q}{RT}\right)$$

Solve this equation for Q, using MATLAB.

1. State the problem

   Find the equation for Q.

2. Describe the input and output

   Input Equation for the reaction rate constant, k
   Output Equation for Q

3. Develop a hand example

$$\begin{aligned} k &= k_0 \exp\left(\frac{-Q}{RT}\right) \\ \frac{k}{k_0} &= \exp\left(\frac{-Q}{RT}\right) \\ \ln\left(\frac{k}{k_0}\right) &= \frac{-Q}{RT} \\ \mathrm{Q} &= RT \ln\left(\frac{k_0}{k}\right) \end{aligned}$$

Notice that the minus sign caused the values inside the natural logarithm to be inverted.

4. Develop a MATLAB solution
First, define a symbolic equation and give it a name:

```
syms k k0 Q R T
X = k == k0*exp(-Q/(R*T))
X =
k == k0/exp(Q/(R*T))
```

Now, ask MATLAB to solve the equation. It is necessary to specify that MATLAB is to solve for Q.

```
solve(X,Q)
ans =
-R*T*log(k/k0)
```

Rather than allowing the result to be called `ans` (the default variable name), it is often preferrable to give the result a more meaningful name.

```
Q = solve(X,Q)
Q =
-R*T*log(k/k0)
```

5. Test the solution
Compare the MATLAB solution with the hand solution. The only difference is that we pulled the minus sign outside the logarithm instead of inverting the ratio of k/k_0. Notice that MATLAB (as well as most computer programs) represents ln as `log` ($\log_{10}$ is represented as `log10`).

HINT

The `symvar` command is useful in determining which variables exist in a symbolic expression or equation. In the previous example, the variable `X` was defined as

```
X = k == k0*exp(-Q/(R*T))
```

The `symvar` function identifies all the variables and groups them into a symbolic array.

```
symvar(X)
ans =
[ Q, R, T, k, k0]
```

KEY IDEA
The `solve` function can solve both linear and nonlinear systems of equations.

12.2.2 Solving Systems of Equations

Not only can the `solve` function solve single equations or expressions for any of the included variables, but it can also solve systems of equations. Take, for example, these three symbolic equations:

```
syms x y z
one = 3*x + 2*y - z == 10;
two = -x + 3*y + 2*z == 5;
three = x - y - z == -1;
```

To solve for the three embedded variables `x`, `y`, and `z`, list all three equations in the `solve` function:

```
answer = solve(one,two,three)
answer =
struct with fields:
 x: -2
 y: 5
 z: -6
```

KEY IDEA
The results of the symbolic `solve` function are listed alphabetically.

These results are puzzling. Each answer is listed as a 1×1 symbolic variable, but they are listed as symbolics, not numerics in the workspace window. In addition, `answer` is listed in the workspace window as a 1×1 structure array. Structure arrays were introduced in a previous chapter. To access the values, use the structure array syntax:

```
answer.x
ans =
-2
answer.y
ans =
5
answer.z
ans =
-6
```

To force the results to be displayed without using a structure array assign names to the individual variables.

```
[x,y,z] = solve(one,two,three)
x =
-2
y =
5
z =
-6
```

The results are assigned alphabetically. For instance, if the variables used in the symbolic expressions are `q`, `x`, and `p`, the results will be returned in the order `p`, `q`, and `x`, independently of the names you have assigned for the results.

Notice in the example that `x`, `y`, and `z` are still listed as symbolic variables, even though the results are numbers. The result of the `solve` function is a symbolic variable, either `ans` or a user-defined name. To use that result in a MATLAB expression requiring a double-precision floating-point input, change the variable type with the `double` function. For example,

```
double(x)
```

changes `x` from a symbolic variable to a corresponding numeric variable.

HINT

Using the `solve` function for multiple equations has both advantages and disadvantages over using linear algebra techniques. In general, if a problem can be solved by means of matrices, the matrix solution will take less computer time. However, linear algebra is limited to first-order polynomials. The `solve` function may take longer, but it can solve nonlinear problems and problems with symbolic variables. Table 12.2 lists some uses of the `solve` function.

Table 12.2 Using the Solve Function

`solve(S)`	Solves an expression with a single variable	`syms x`
		`solve(x-5)`
		`ans=`
		`5`

`solve(S)`	Solves an equation with a single variable	`syms x`
		`solve(x^2= =-5)`
		`ans=`
		`7^(1/2)`
		`-7^(1/2)`

`solve(S)`	Solves an equation whose solutions are complex numbers	`syms x`
		`solve(x^2= =-5)`
		`ans =`
		`i*5^(1/2)`
		`-1*5^(1/2)`

`solve(S)`	Solves an equation with more than one variable for x or the closest variable to x	`syms x y`
		`solve(y= =x^2+2)`
		`ans =`
		`(y-2)^(1/2)`
		`-(y-2)^(1/2)`

`solve(S,y)`	Solves an equation with more than one variable for a specified variable	`syms x y`
		`solve(y+6*x,x)`
		`ans =`
		`-y/6`

`solve(S1,S2,S3)`	Solves a system of equations and presents the solutions as a structure array	`syms x y z`
		`one=3*x+2*y-z= =10;`
		`two=-x+3*y+2*z= =5;`
		`three=x-y-z= =-1;`
		`solve(one,two,three)`
		`ans =`
		`x: [1x1 sym]`
		`y: [1x1 sym]`
		`z: [1x1 sym]`

`[A,B,C]= solve (S1,S2,S3)`	Solves a system of equations and assigns the solutions to user-defined variable names; displays the results alphabetically	`one=3*x+2*y-z= =10;`
		`two= -x+3*y+2*z= =5;`
		`three= x-y-z= =-1;`
		`[x,y,z]=solve(one,two,three)`
		`x= -2`
		`y= 5`
		`z= -6`

PRACTICE EXERCISE 12.4

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

Use following system of linear equations in Exercises 12.1 through 12.5:

$$\begin{aligned} 5x + 6y - 3z &= 10 \\ 3x - 3y + 2z &= 14 \\ 2x - 4y - 12z &= 24 \end{aligned}$$

1. Solve this system of equations by means of the linear algebra techniques discussed in Chapter 10.
2. Define a symbolic equation representing each equation in the given system of equations. Use the `solve` function to solve for `x`, `y`, and `z`.
3. Display the results from Exercise 2 by using the structure array syntax.
4. Display the results from Exercise 2 by specifying the output names.
5. Consider the following nonlinear system of equations:

$$\begin{aligned} x^2 + 5y - 3z^3 &= 15 \\ 4x + y^2 - z &= 10 \\ x + y + z &= 15 \end{aligned}$$

Solve the nonlinear system with the `solve` function. Use the `double` function on your results to simplify the answer.

KEY IDEA

Once a variable is listed as a symbolic variable in the workspace window, it need not be redefined for every symbolic calculation.

12.2.3 Substitution and Symbolic Functions

Particularly for engineers or scientists, once a symbolic expression is defined, we often want to substitute values into it. Consider the quadratic equation:

```
syms x a b c
e4 = a*x^2+b*x+c
```

There are a number of possible substitutions. For example, to change the variable `x` into the variable `y`, the `subs` function requires three inputs: the expression to be modified, the variable to be modified, and the new variable to be inserted. To substitute `y` for all the `x`'s we would use the command

```
syms y
subs(e4,x,y)
```

which returns

```
ans =
a*y^2 + b*y + c
```

The variable `e4` has not been changed; rather, the new information is stored in `ans`, or it could be given a new name, such as `e5`:

```
e5 = subs(e4,x,y)
e5 =
a*y^2 + b*y + c
```

Recalling `e4`, notice that it remains unchanged:

```
e4
e4 =
a*x^2+b*x+c
```

Substituting numbers uses the same procedure:

```
subs(e4,x,3)
ans =
9*a + 3*b + c
```

To make multiple substitutions list the variables as arrays:

```
subs(e4,[a,b,c,x],[1,2,3,4])
ans =
      27
```

Numeric arrays can even be substituted in for variables. For example, create a new expression containing only `x`:

```
e6 = subs(e4,[a,b,c],[1,2,3])
```

This gives

```
e6 =
x^2+2*x+3
```

Now define an array of numbers and substitute them into `E6`:

```
numbers = 1:5;
subs(e6,x,numbers)
ans =
      [6    11    18    27    38]
```

This couldn't be done in a single step, because every element of the array must be the same size for the `subs` function to work.

An alternative approach is to use a symbolic function. Since `x` is already defined as symbolic the definition of the function would be

```
f(x) = x^2+2*x+3
```

However, the `symfun` function can convert an existing expression stored as a variable into a symbolic function without retyping any of the code. The `symfun` function requires two inputs; a symbolic expression, and an array of variable names.

```
f = symfun(e6,x)
```

and returns the symbolic function

```
f(x)
x^2+2*x+3
```

Here's an example of a symbolic expression with two symbolic variables, used as input to `symfun`:

```
g = symfun(x^2+y^2,[x,y])
```

which returns

```
g(x, y) =
x^2 + y^2
```

This new function can now be evaluated for specified values of `x` and `y`.

```
g(1,3)
ans =
10
```

PRACTICE EXERCISE 12.5

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Using the `subs` function, substitute `4` into each expression/equation defined in Practice Exercise 12.1 for `x`. Comment on your results.
2. Create a symbolic function called `g` using the `symfun` function for the following:

   ```
   x^2+sin(x)*x
   ```

   and use it to evaluate `g(a)`, `g(3)`, and `g([1:5])`.

EXAMPLE 12.2

USING SYMBOLIC MATH TO SOLVE A BALLISTICS PROBLEM

We can use the symbolic math capabilities of MATLAB to explore the equations representing the trajectory of an unpowered projectile, such as the cannonball shown in Figure 12.2.

From elementary physics the distance a projectile travels horizontally is

$$d_x = v_0 t \cos(\theta)$$

and the distance traveled vertically is

$$d_y = v_0 t \sin(\theta) - \frac{1}{2} g t^2$$

where

v_0 = velocity at launch,
t = time,
θ = launch angle, and
g = acceleration due to gravity.

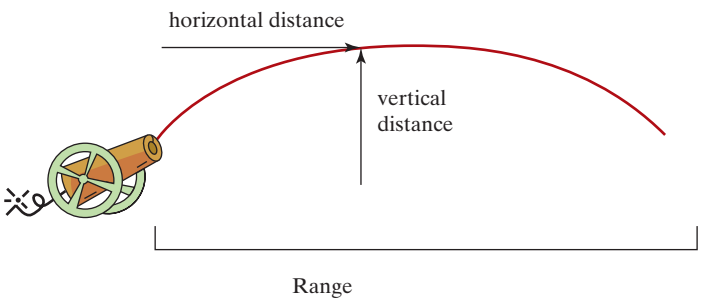


Figure 12.2 The range of a projectile depends on the initial velocity and the launch angle.

Use these equations and MATLAB's symbolic capability to derive an equation for the distance the projectile has traveled horizontally when it hits the ground (the range).

1. State the problem
   Find the range equation.
2. Describe the input and output
   Input Equations for horizontal and vertical distances
   Output Equation for range
3. Develop a hand example

$$d_y = v_0 t \sin(\theta) - \frac{1}{2} g t^2 = 0$$

   Rearrange to give

$$v_0 t \sin(\theta) = \frac{1}{2} g t^2$$

   Divide by t and solve:

$$t = \frac{2 v_0 \sin(\theta)}{g}$$

   Now substitute this expression for t into the horizontal-distance formula to obtain

$$\begin{aligned} d_x &= v_0 t \cos(\theta) \\ \text{range} &= v_0 \left(\frac{2 v_0 \sin(\theta)}{g} \right) \cos(\theta) \end{aligned}$$

   From trigonometry $2 \sin\theta \cos\theta$ is the same as $\sin(2\theta)$, which allows a further simplification if desired.
4. Develop a MATLAB solution
   First, define the symbolic variables:

```
syms v0 t theta g
```

   Next, define the symbolic expression for the vertical distance traveled:

```
Distancey = v0 * t *sin(theta) - 1/2*g*t^2;
```

   Now, define the symbolic expression for the horizontal distance traveled:

```
Distancex = v0 * t *cos(theta);
```

   Solve the vertical-distance expression for the time of impact, since the vertical distance = 0 at impact:

```
impact_time = solve(Distancey,t)
```

   This returns two answers:

```
impact_time =
[             0]
[2*v0*sin(theta)/g]
```

   This result makes sense, since the vertical distance is zero at launch and again at impact. Substitute the impact time into the horizontal-distance expression. Since only the second is meaningful use `impact_time(2)`:

```
impact_distance = subs(Distancex,t,impact_time(2))
```

(*continued*)

The substitution results in an equation for the distance the projectile has traveled when it hits the ground:

```
impact_distance =
(2*v0^2*cos(theta)*sin(theta))/g
```

5. Test the solution
   Compare the MATLAB solution with the hand solution. Both approaches give the same result.
   MATLAB can simplify the result, although it is already pretty simple. Using the simplify command

```
simplify(impact_distance)
```

gives the following results.

```
ans =
 (v0^2*sin(2*theta))/g
```

12.3 SYMBOLIC PLOTTING

The symbolic toolbox includes a group of functions to plot symbolic expressions and functions. The most basic is `fplot`, which was introduced in an earlier chapter. This is an example of an **overloaded function**, which means the same function name is used for more than one application. The MATLAB program decides which application is appropriate based on the input. When symbolic information is supplied, MATLAB uses the symbolic plotting tools.

12.3.1 The `fplot` Function

Consider a simple function of `x`, such as

```
syms x
y = x^2-2
```

The symbolic variable `y`, need not be stored as an actual symbolic function, such as

```
f(x) = x^2-2
```

to be used as input to `fplot`. As with all of the symbolic functions explored thus far, both symbolic variables and symbolic functions behave in the same manner.

To plot this function, use

```
fplot(y)
```

The resulting graph is shown in Figure 12.3. The `fplot` function defaults to an `x` range from -5 to $+5$. MATLAB created this plot by choosing values of `x` and calculating corresponding values of `y`, so that a smooth curve is produced.

The user who does not want to accept the default values can specify the minimum and maximum values of `x` in the second field of the `fplot` function:

```
fplot(y,[-10,10])
```

The values are enclosed in square brackets, indicating that they are elements in the array that defines the plot extremes. Titles, axis labels, and annotations are

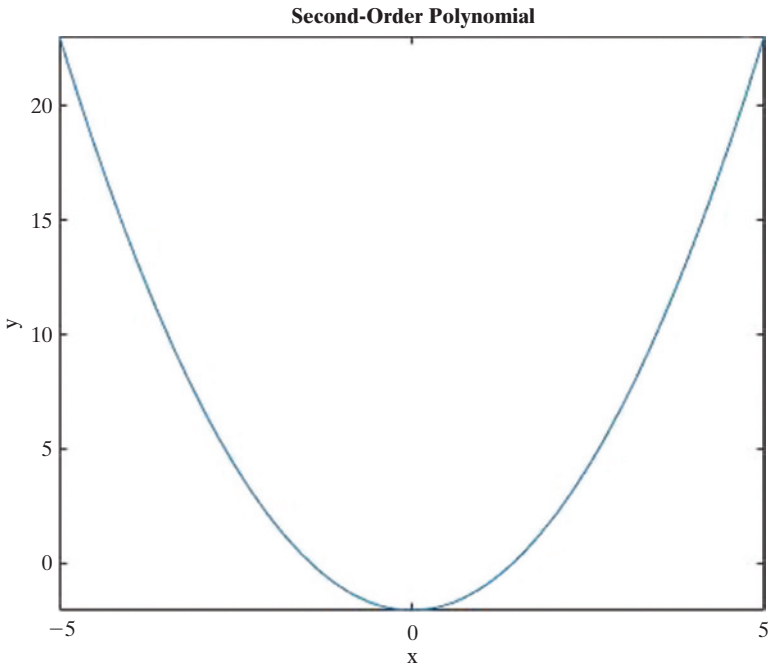


Figure 12.3
Symbolic expressions can be plotted with `fplot`. Titles, labels, and other annotations are added to `fplot` with the use of standard MATLAB annotation functions.

specified in the same manner as other MATLAB plots. For example, to add a title and labels to the plot, use

```
title("Second-Order Polynomial")
xlabel("x"), ylabel("y")
```

The `fimplicit` function allows you to plot implicit functions of x and y. For instance, consider the implicit equation

$$x^2 + y^2 = 1$$

KEY IDEA
It is often easier to define physical systems as parametric equations, where each dependent variable such as position is defined in terms of an independent variable such as time.

which you may recognize as the equation for a circle of radius 1. You could solve for y, but it's not necessary with `fimplicit`. Any of the commands

```
syms x y
fimplicit(x^2 + y^2 == 1,[-1.5,1.5])
fimplicit(x^2 + y^2 -1,[-1.5,1.5])
```

and

```
z = x^2 + y^2 -1
fimplicit(z,[-1.5,1.5])
```

can be used to create the graph of the circle shown on the left-hand side in Figure 12.4.

Another way to define similar equations is **parametrically**; that is, define separate equations for x and for y in terms of a third variable. A circle can be defined parametrically as

$$x = \sin(t)$$
$$y = \cos(t)$$

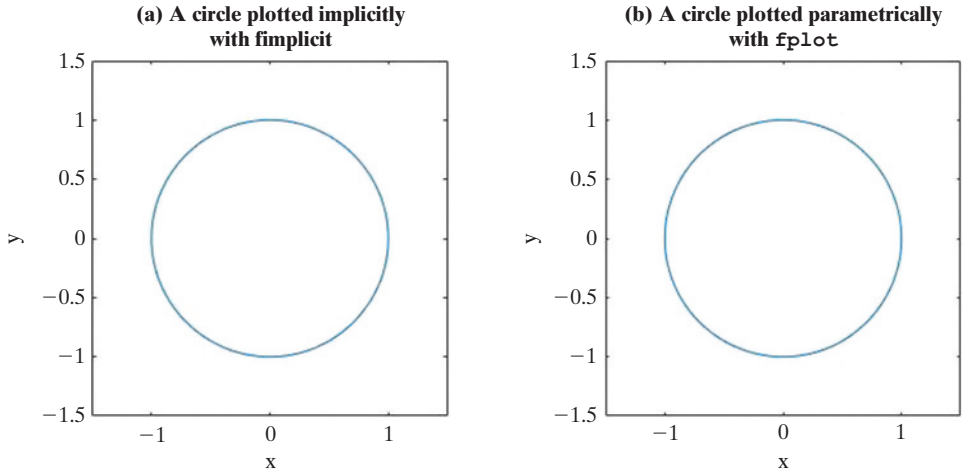


Figure 12.4
In addition to functions of a single variable, (a) The `fimplicit` function can be used to graph implicit functions. (b) Parametric functions are plotted using `fplot`.

To plot the circle parametrically with `fplot`, first define the symbolic variable `t`.

```
syms t
fplot(sin(t),cos(t),[0,2*pi])
axis([-1.5,1.5,-1.5,1.5])
axis square
```

The results are shown on the right-hand side of Figure 12.4.

Although annotation is done the same way for symbolic plots as for standard numeric plots, in order to plot multiple lines on the same graph, it is necessary to use the `hold on` command. To adjust colors, line styles, and marker styles, use the same strategies described for numeric plots. For example, to plot $sin(x)$, $sin(2x)$, and $sin(3x)$ on the same graph, first define some symbolic expressions:

KEY IDEA
Never issue the `hold on` command without also issuing the `hold off` command.

```
syms x
y1 = sin(x)
y2 = sin(2*x)
y3 = sin(3*x)
```

Then plot each expression, defining the line style and color as desired

```
fplot(y1,"-r")
hold on
fplot(y2,"-g")
fplot(y3,":b")
title("Multiple Symbolic Plots")
xlabel("x"), ylabel("y")
```

The results are shown in Figure 12.5. When you're done plotting, it is important to issue the

```
hold off
```

command.

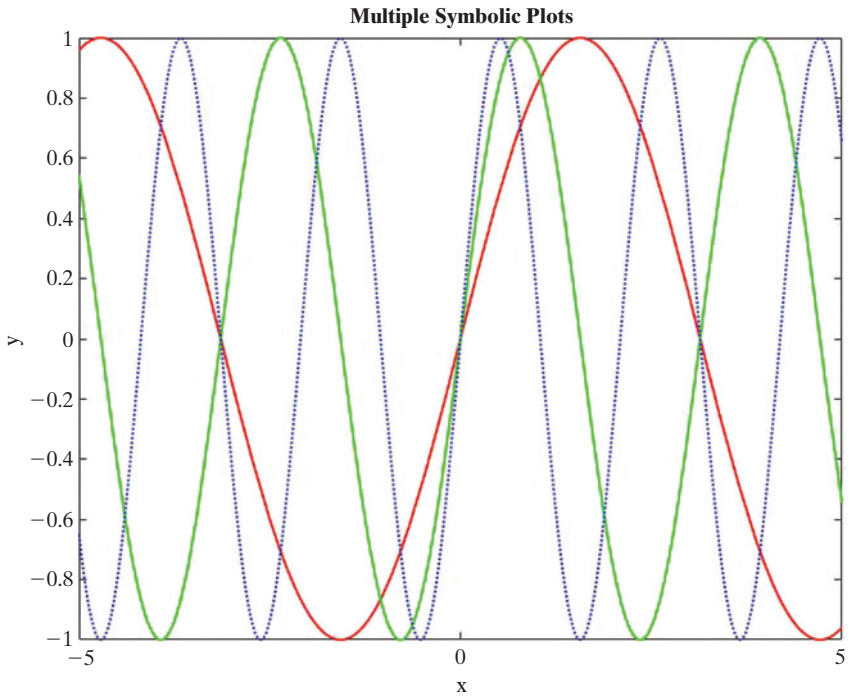


Figure 12.5
Use `fplot` to create multiple symbolic plots. Adjust line style, color, and markers in the usual way.

HINT

Most symbolic functions will allow you to enter either a symbolic variable that represents a function, the function itself, or a function handle such as f when a symbolic function such as f(x) is defined. For example,

```
syms x
y = x^2-1
fplot(y)
```

is equivalent to

```
fplot(x^2-1)
```

PRACTICE EXERCISE 12.6

Be sure to add titles and axis labels to all your plots. The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Use `fplot` to plot `ex1` from -2π to $+2\pi$.
2. Use `fplot` to plot `ex2` from -10 to $+10$.
3. Use `fplot` to plot `ex6` from -2π to $+2\pi$.
4. Repeat problems 1 through 3 using the symbolic function form of the expressions, `fex1`, `fex2`, and `fex6`.
5. Use `fplot` to plot $\cos(x)$ from -2π to $+2\pi$. Don't define an expression for $\cos(x)$; just enter it directly into `fplot`.

```
fplot(@(x) cos(x), [-2*pi, 2*pi])
```

(*continued*)

6. Use `fimplicit` to create an implicit plot of `x^2 - y^4 = 5`.
7. Use `fplot` to plot $\sin(x)$ and $\cos(x)$ on the same graph. Use the annotation tools to change the color of the sine graph from the default.
8. Use `fplot` to create a parametric plot of $x = \sin(t)$ and $y = 3\cos(t)$.

12.3.2 Additional Symbolic Plots

Additional symbolic plotting functions that mirror the functions used in numeric MATLAB plotting options are listed in Table 12.3.

To demonstrate how the three-dimensional surface plotting functions (`fmesh` and `fsurf`) work, first define a symbolic version of the `peaks` function:

```
syms x y
z1 = 3*(1-x)^2*exp(-(x^2)-(y+1)^2)
z2 = -10*(x/5-x^3-y^5)*exp(-x^2-y^2)
z3 = -1/3*exp(-(x+1)^2-y^2)
z = z1+z2+z3
```

This function was separated into three parts to make it easier to enter into the computer. Notice that no "dot" operators are used in these expressions, since they are all symbolic. These plotting functions work similarly to their numeric counterparts. They will accept either simple expressions or symbolic functions as input.

KEY IDEA
Most of the MATLAB plotting functions for arrays have corresponding functions for symbolic applications.

```
t=tiledlayout(1,2);
nexttile
fmesh(z)
title("(a) fmesh")
nexttile
fsurf(z)
title("(b) fsurf")
```

The plots resulting from these commands are shown in Figure 12.6. When we created the same plots via a standard MATLAB approach, it was necessary to define an array of both *x* and *y* values, mesh them together, and calculate the values of *z* on the basis of the two-dimensional arrays.
The symbolic plotting capability contained in the symbolic toolbox makes creating these graphs much easier.
They can be annotated by using the standard MATLAB functions, such as `title, xlabel, text`, etc.

```
title(t,["The Peaks Function";"Rendered with Symbolic Plotting Tools"])
```

Table 12.3 Symbolic Plotting Functions

fplot	Function plotter	If *z* is a function of *x*: `fplot(z)`
fplot3	Three-dimensional parametric curve plotter	If *x* is a function of *t*, if *y* is a function of *t*, and if *z* is a function of *t*: `fplot3(x,y,z)`
fmesh	Mesh plotter	If *z* is a function of *x* and *y*: `fmesh(z)`
fsurf	Surface plotter	If *z* is a function of *x* and *y*: `fsurf(z)`
fimplicit	Implicit function plotter	If z is a function of x and y: `fimplicit(z)`
fimplicit3	Three-dimensional implicit function plotter	If *w* is a function of x, y and `z: fimplicit3(w)`
fcontour	Contour plotter	If *z* is a function of *x* and *y*: `fcontour(z)`
ezpolar	polar plotter	If z is a function of x: `ezpolar(z)`

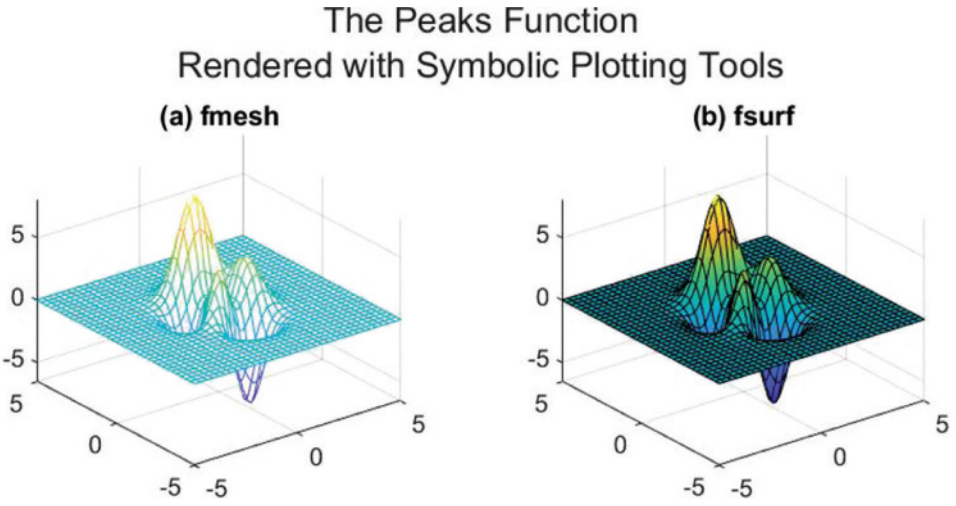


Figure 12.6
Examples of three-dimensional symbolic surface plots. (a) The peaks function rendered with `fmesh`. (b) The peaks function rendered with `fsurf`.

The two-dimensional plots and contour plots are also similar to their numeric counterparts. For example, this contour plot is a two-dimensional representation of the three-dimensional `peaks` function, and is shown in Figure 12.7(a).

```
t=tiledlayout(1,2);
nexttile
fcontour(z)
title ("(a) fcontour")
```

Any of these functions (`fmesh`, `fsurf`, and `fcontour`) can also handle parameterized functions (one function for *x*, one for *y*, and one for *z*). For example, the following code produces the torus shown in Figure 12.7(b).

```
nexttile
syms u v
x=4+(3+cos(v))*sin(u)
y=4 + (3 + cos(v))*cos(u)
```

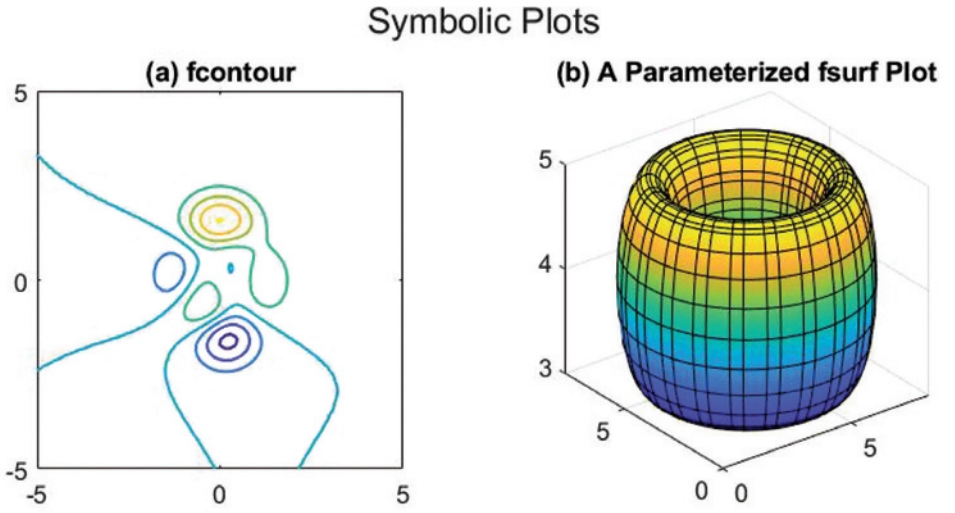


Figure 12.7
Symbolic plots: (a) The peaks function rendered as a contour plot. (b) A torus created from a parameterized function and rendered using `fsurf`.

```
z=4+sin(v)
fsurf(x,y,z)
title("(b) A Parameterized fsurf Plot")
title(t,["Symbolic Plots"])
```

PRACTICE EXERCISE 12.7

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

Create a symbolic expression for $Z = \sin\left(\sqrt{X^2 + Y^2}\right)$.

1. Use `fmesh` to create a mesh plot of *Z*. Be sure to add a title and axis labels.
2. Use `fsurf` to create a surface plot of *Z*. Be sure to add a title and axis labels.
3. Use `fcontour` to create a contour plot of *Z*. Be sure to add a title and axis labels.
4. The `fplot3` function requires us to define three variables as a function of a fourth. To do this, first define `t` as a symbolic variable, then let

$$\begin{aligned} x &= t \\ y &= \sin(t) \\ z &= \cos(t) \end{aligned}$$

   Use `fplot3` to plot this parametric function from 0 to 30.
5. A sphere can be defined parametrically as

$$\begin{aligned} x &= \cos(phi) \times \cos(theta) \\ y &= \cos(phi) \times \sin(theta) \\ z &= \sin(phi) \end{aligned}$$

   for values of both *phi* and *theta* from $-\pi$ to $+\pi$. Use `fsurf` to create the corresponding plot.

EXAMPLE 12.3

USING SYMBOLIC PLOTTING TO ILLUSTRATE A BALLISTICS PROBLEM

In Example 12.2, MATLAB's symbolic capabilities were used to derive an equation for the distance a projectile travels before it hits the ground. The horizontal-distance formula

$$d_x = v_0 t \cos(\theta)$$

and the vertical-distance formula

$$d_y = v_0 t \sin(\theta) - \frac{1}{2} g t^2$$

where

v_0 = the velocity at launch,
t = time,
θ = launch angle, and
g = acceleration due to gravity,

were combined to give

$$\text{range} = v_0\left(\frac{2v_0\sin(\theta)}{g}\right)\cos(\theta).$$

Using MATLAB's symbolic plotting capability, create a plot showing the range traveled for angles from 0 to $\pi/2$. Assume an initial velocity of 100 m/s and an acceleration due to gravity of 9.8m/s^2.

1. State the problem
   Plot the range as a function of launch angle.

2. Describe the input and output
   Input Symbolic equation for range

   $$v_0 = 100\,\text{m/s}$$
   $$\text{g} = 9.8\ \text{m/s}^2$$

   Output Plot of range versus angle

3. Develop a hand example

   $$\text{range} = v_0\left(\frac{2v_0\sin(\theta)}{g}\right)\cos(\theta)$$

   From trigonometry $2\ \sin\theta\ \cos\theta$ equals $\sin(2\theta)$. Simplifying the result gives

   $$\text{range} = \frac{v_0^2}{g}\sin(2\theta)$$

   With this equation, a few data points can be calculated:

Angle	Range, m
0	0
$\pi/6$	884
$\pi/4$	1020
$\pi/3$	884
$\pi/2$	0

   The range appears to increase with increasing angle, then decrease back to zero when the cannon is pointed straight up.

4. Develop a MATLAB solution
   First, modify the equation from Example 12.2 to include the launch velocity and the acceleration due to gravity. Recall that

```
impact_distance =
2*v0^2*sin(theta)/g*cos(theta)
```

(*continued*)

Use the `subs` function to substitute the numerical values into the equation:

```
impact_100 = subs(impact_distance,[v0,g],[100, 9.8])
```

This returns

```
impact_100 =
100000/49*sin(theta)*cos(theta)
```

Finally, plot the results, and add a title and labels:

```
fplot(impact_100,[0, pi/2])
title("Maximum Projectile Distance Traveled")
xlabel("angle, radians")
ylabel("range, m")
```

This generates Figure 12.8.

5. Test the solution
The MATLAB solution agrees with the hand solution. The range is zero when the cannon is pointed straight up and zero when it is pointed horizontally. The range appears to peak at an angle of about 0.8 radian, which corresponds roughly to 45°.

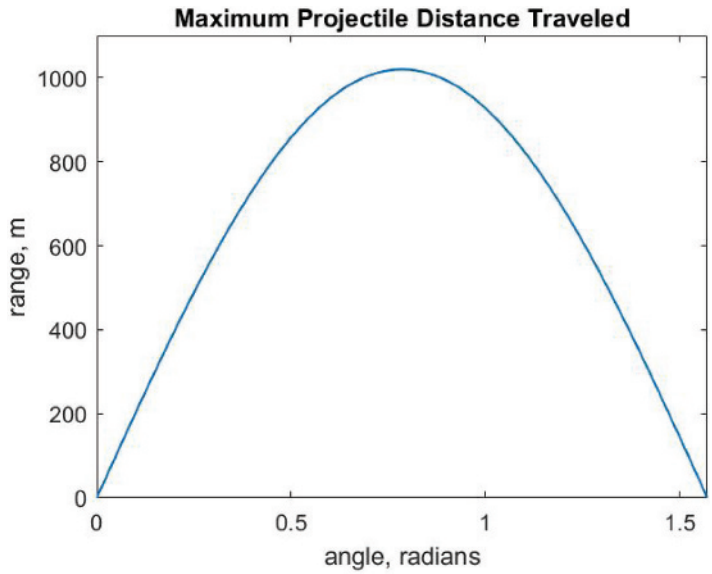


Figure 12.8
Projectile range.

12.4 UNITS OF MEASUREMENT

One of the most important things an engineer does when performing calculations is to keep track of units of measurement. Unfortunately, it's easy to lose track of units in computer calculations. The symbolic toolbox offers a way to attach units to variables. These units behave like other algebraic parameters, allowing the user to check for consistency, do unit conversions, and even include important physical constants in calculations without entering them directly.

The first step in adding units to calculations is to add `symunit` to the workspace.

```
u=symunit
```

which returns

```
u =
  symbolicUnitsCollection with units:

          ampere: [A]
          kelvin: [K]
       kilogram: [kg]
           meter: [m]
            mole: [mol]
          second: [s]
         candela: [cd]
 Show all units.
```

The seven categories of units shown represent the 7 base units used to define a system of measurement, in this case the SI system (System Internationale.) Just about any unit of measurement you can think of is built into the MATLAB `symbolicUnitsCollection`. You should take a minute to browse through the list by selecting Show `all units` in the Command Window.

Specify a unit by using the dot nomenclature used in structure arrays. To add a unit of meters and a value of 5 to a variable named `distance1`, if you've assigned the variable `u` as your symbolic unit name, type:

```
distance1 = 5*u.m
```

which returns

```
distance1 =
5*[m]
```

If you aren't sure of the correct unit name, use tab expansion to review the list. For example, if you type

```
distance2= 10*u.feet
```

an error message is generated. Try again, but this time stop and select Tab after `'u.'` as shown in Figure 12.9. The list that appears is extensive and is organized

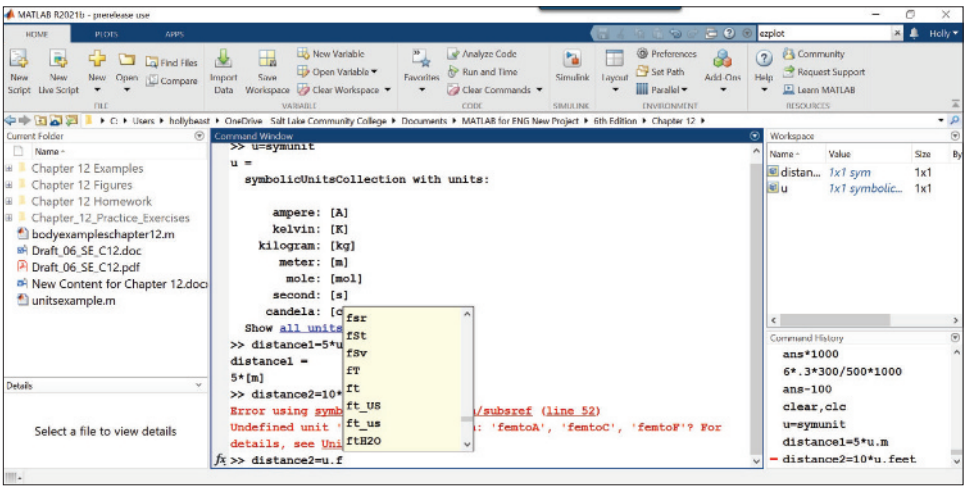


Figure 12.9
The MATLAB Symbolic Toolbox allows the user to add units to numbers, and to search the `symbolicUnits Collection` for unit names with the Tab expansion.

alphabetically. Once it opens you can start typing a guess. There are two logical choices that pop-up, either `foot` or `ft`. MATLAB will accept either choice:

```
distance2=10*u.ft
```

Units are not automatically simplified when mathematical operations are performed. For example, the following division problem results in mixed units:

```
ratio = distance1/distance2
ratio =
(1/2)*([m]/[ft])
```

Using the `simplify` function returns:

```
ratio = simplify(ratio)
ratio =
625/381
```

The result is still a symbolic number, which can be converted to a `double` if desired.

Here's a slightly more complicated example. First add a few more variables with attached units:

```
distance3=15*u.m;
velocity = 10*u.m/u.sec;
time = 15*u.sec;
```

and then combine them:

```
total = velocity*time + distance1 + distance3 + distance2
```

which returns

```
total =
10*[ft] + 170*[m]
```

Notice that the units were combined and canceled automatically. Using the `simplify` function results in:

```
total = simplify(total)
total =
(216310/381)*[ft]
```

The `simplify` function chooses the unit to which it simplifies. Use the `unitConvert/` function to change the results to the unit of your choice.

```
total = unitConvert(total,u.m)
total =
(21631/125)*[m]
```

You can separate the units from the calculated numerical values with the `separateUnits` function, and then convert the variable to a `double` for use in other calculations.

```
[total_value,total_unit]= separateUnits(total)
total_value =
```

```
21631/125
total_unit =
[m]
total_value = double(total_value)
total_value =
       173.05
```

KEY IDEA
The symbolic algebra toolbox allows you to add units to your calculations.

The `unitConvert` function is extremely useful, but some care needs to be exercised when using it with temperatures. The assumption is made that any temperature values entered are temperature differences, not actual temperature values. This can be demonstrated with the following calculation:

```
   delta_T_C = 5*u.Celsius;
delta_T_K = unitConvert(delta_T_C,u.K)
```

which returns

```
   delta_T_K =
5*[K]
```

If the desired calculation is to change an actual measured temperature from one scale to another you need to specify the values as absolute (versus the difference form) in the function input.

```
   T_C=5*u.Celsius;
T_K=unitConvert(T_C,u.K,"Temperature","Absolute")
```

This time the result is:

```
T_K =
(5563/20)*[K]
```

EXAMPLE 12.4

USING SYMBOLIC UNITS IN AN IDEAL GAS LAW PROBLEM

The ideal gas law is introduced in most freshman chemistry classes. The concepts behind the law and the equation itself are relatively easy to understand. However, many calculational errors result from problems with unit conversions.

The ideal gas law, as used in mechanical engineering is:

$$PV = mRT$$

where

P = absolute gas pressure

V = container volume

m = mass

T = absolute temperature

R = constant, defined for each gas as $\frac{R_u}{M}$, the universal gas constant divided by the molar mass. For air, R has a value of 0.2870 $kJ/kg{\cdot}K$

(*continued*)

Consider the following problem. A 500-L rigid tank contains 6 kg of air at 26°C. What is the gage pressure in the tank if the atmospheric pressure is 98 kPa? Solve this problem using MATLAB's ability to add units to variables.

1. State the problem
   Find the gage pressure in the tank.
   The gage pressure is the difference between the absolute pressure and the atmospheric pressure
   $$P_{gage} = P_{absolute} - P_{atm}$$

2. Describe the input and output
   Input $V = 500$ L
   $m = 6$ kg
   $T = 26\ ^{o}$C
   $R = 0.2870\ kJ/kg{\cdot}K$
   Output P_{gage}

3. Develop a hand example
   $$P_{absolute} = \frac{mRT}{V}$$
   $$P_{gage} = P_{absolute} - P_{atm}$$
   Estimate the answer with round numbers.
   $$P_{absolute} = \frac{6kg * 0.3kJ/kgK * 300K}{0.5m^3} = 1080kPa$$
   $$P_{gage} = 1080kPa - 100 = 980kPa$$

4. Develop a MATLAB solution
   First add `symunit` and then define each of the variables with units attached.

```
u=symunit;
V=500*u.liter;
m=6*u.kg;
T=26*u.Celsius;
Patm=98*u.kPa;
Rair=0.2870*u.kJ/u.kg/u.K
```

   The temperature variable, T, must then be converted to absolute.

```
T= unitConvert(T,u.K,"Temperature","absolute")
```

   The absolute pressure can then be calculated:

```
P=m*Rair*T/V
```

   which returns

```
P =
(5151363/5000000)*([kJ]/[l])
```

   The result is in units that are not very useful, kJ/liter. Therefore, the `unitConvert` function can be used to generate results in units of our choice, in this case kPa.

```
P = unitConvert(P,u.kPa)
P =(5151363/5000)*[kPa]
```

The gage pressure can now be calculated, units separated from the result, and the result converted to a double.

```
Pgage = P-Patm;
[P_value,P_units]=separateUnits(Pgage);
P_value=double(P_value)
```

which gives a final answer of

```
P_value =
932.27
```

5. Test the solution
   The MATLAB solution agrees well with the estimate numerically. In addition, the units resulting from the algebraic manipulation were pressure units (kPa) as expected.

12.5 CALCULUS

MATLAB's symbolic toolbox allows the user to differentiate symbolically, and to perform integrations. This makes it possible to find analytical solutions, instead of numeric approximations, for many problems.

12.5.1 Differentiation

Differential calculus is studied extensively in first-semester calculus. The derivative can be thought of as the slope of a function, or as the rate of change of the function. For example, consider a race car. The velocity of the car can be approximated by the change in distance divided by the change in time. Suppose that, during a race, the car starts slowly and reaches its fastest speed at the finish line. To avoid running into the stands, the car must then slow down until it finally stops. We might model the position of the car with a sine wave, as shown in Figure 12.10. The relevant equation is

$$d = 20 + 20 \sin\left(\frac{\pi(t-10)}{20}\right)$$

The graphs in Figure 12.10 were created with `fplot` and symbolic mathematics. First, create the symbolic variable `t`, then define a symbolic expression for distance:

```
syms t
dist = 20+20*sin(pi*(t-10)/20)
```

Once the symbolic expression has been created it can be substituted into the `fplot` function to create the graph.

```
p=tiledlayout(1,3);
nexttile
fplot(dist,[0,20])
title("(a) Car Position")
xlabel("Time"), ylabel("Distance")
text(10,20,"Finish Line")
```

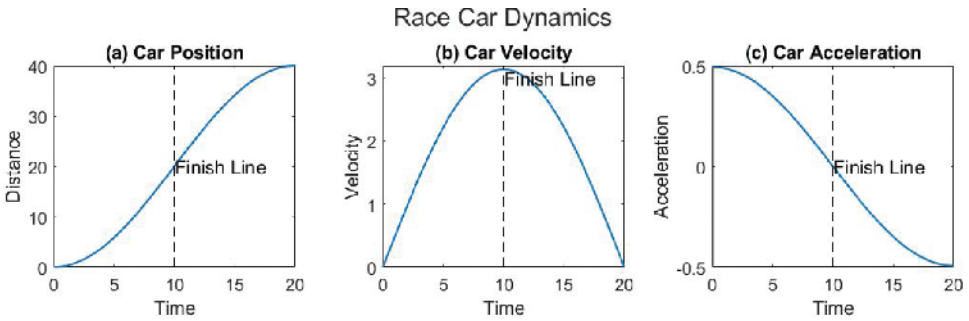


Figure 12.10
Optimally a race car speeds up until it reaches the finish line. Then it slows to a stop. (a) Car position continues to increase even after it passes the finish line. (b) Car velocity is a maximum at the finish line. (c) Car acceleration is positive before reaching the finish line, is zero at the finish line, and is negative after the finish line.

Velocity is the derivative of position; so to find the equation of the velocity of the car, use the `diff` function:

```
velocity = diff(dist)
```

The result is:

```
velocity =
pi*cos((pi*(t-10))/20)
```

Use the `fplot` function to add a velocity plot to the figure.

```
nexttile
fplot(velocity,[0,20])
title("(b) Car Velocity")
xlabel('Time'), ylabel('Velocity')
text(10,3,"Finish Line")
```

The results are shown in Figure 12.10(b).

The acceleration of the race car is the change in the velocity, divided by the change in time, so the acceleration is the derivative of the velocity function:

```
acceleration = diff(velocity)
```

which returns

```
acceleration =
-(pi^2*sin((pi*(t-10))/20))/20
```

The plot of acceleration (see Figure 12.10(c)) was also created with the use of the symbolic plotting function. Tiling (subplots) were used to put all 3 graphs in the same figure, annotations were added, and a title was added to the top of *the* entire figure.

```
nexttile
fplot(acceleration,[0,20])
title("(c) Car Acceleration")
xlabel('Time'), ylabel('Acceleration')
text(10,0,'Finish Line')
hold on
plot([10,10],[-0.5,+0.5],"--k")
hold off
title(p,"Race Car Dynamics")
```

Table 12.4 Symbolic Differentiation

`diff(f)`	Returns the derivative of the expression `f` with respect to the default independent variable, `x`	`syms x z`
		`y=x^3+z^2`
		`diff(y)`
		`ans =`
		`3*x^2`

`diff(f,t)`	Returns the derivative of the expression `f` with respect to the variable `t`	`syms x z`
		`y=x^3+z^2`
		`diff(y, z)`
		`ans =`
		`2*z`

`diff(f,n)`	Returns the nth derivative of the expression `f` with respect to the default independent variable	`syms x z`
		`y=x^3+z^2`
		`diff(y,2)`
		`ans =`
		`6*x`

`diff(f,t,n)`	Returns the nth derivative of the expression `f` with respect to the variable `t`	`syms x z`
		`y=x^3+z^2`
		`diff(y, z, 2)`
		`ans =`
		`2`

KEY IDEA
The symbolic algebra toolbox makes it easy to perform basic calculus operations such as differentiation and integration.

The acceleration is the first derivative of the velocity and the second derivative of the position. MATLAB offers several slightly different techniques to find both first derivatives and *n*th derivatives (see Table 12.4).

For more complicated equations with multiple variables, such as

```
syms x z t
y = x^2+t-3*z^3
```

MATLAB calculates the derivative with respect to `x`, the default variable:

```
diff(y)
ans =
2*x
```

The result is the rate of change of `y` as `x` changes (keeping all the other variables constant). This is usually depicted as $\partial y/\partial x$ and is called a partial derivative. To see how `y` changes with respect to another variable, such as `t`, specify it in the `diff` function (remember that all the variables must have been previously defined as symbolic.)

```
diff(y,t)
ans =
1
```

Similarly, to see how `y` changes with `z` when everything else is kept constant, use

```
diff(y,z)
ans =
-9*z^2
```

To find higher-order derivatives, either nest the `diff` function, or specify the order of the derivative in the `diff` function. Either of the statements

```
diff(y,2)
```

or

```
diff(diff(y))
```

returns the same result:

```
ans =
2
```

Notice that although the result appears to be a number, it is a symbolic variable. In order to use it in a numeric MATLAB calculation, convert it to a double-precision floating-point number using the `double` function.
To take a higher derivative of `y` with respect to a variable that is not the default, specify both the degree of the derivative and the variable. For example, to find the second derivative of `y` with respect to `z`, type

```
diff(y,z,2)
ans =
-18*z
```

PRACTICE EXERCISE 12.8

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Find the first derivative with respect to x of the following expressions:
   $x^2 + x + 1$
   $\sin(x)$
   $\tan(x)$
   $\ln(x)$

2. Find the first partial derivative with respect to x of the following expressions:
   $ax^2 + bx + c$
   $x^{0.5} - 3y$
   $\tan(x + y)$
   $3x + 4y - 3xy$

3. Find the second derivative with respect to x for each of the expressions in Exercise 12.8.1 and 12.8.2.

4. Find the first derivative with respect to y for the following expressions:
   $y^2 - 1$
   $2y + 3x^2$
   $ay + bx + cz$

5. Find the second derivative with respect to y for each of the expressions in Exercise 12.8.4.

EXAMPLE 12.5

USING SYMBOLIC MATH TO FIND THE OPTIMUM LAUNCH ANGLE

In Example 12.3, the symbolic plotting capability of MATLAB was used to create a graph of range versus launch angle, based on the range formula derived in Example 12.2, namely

$$\text{range} = v_0\left(\frac{2v_0\sin(\theta)}{g}\right)\cos(\theta)$$

where

v_0 = velocity at launch, which we chose to be 100 m/s,
θ = launch angle, and
g = accelaration due to gravity, which we chose to be $9.8\,\text{m/s}^2$.

Use MATLAB's symbolic capability to find the angle at which the maximum range occurs and to find the maximum range.

1. State the problem
   Find the angle at which the maximum range occurs.
   Find the maximum range.

2. Describe the input and output
   Input Symbolic equation for range

   $$v_0 = 100\ \text{m/s}$$
   $$\text{g} = 9.8\ \text{m/s}^2$$

   Output The angle at which the maximum range occurs
   The maximum range

3. Develop a hand example
   From the graph in Figure 12.11, the maximum range appears to occur at a launch angle of approximately 0.7 or 0.8 radian, and the maximum height appears to be approximately 1000 m.

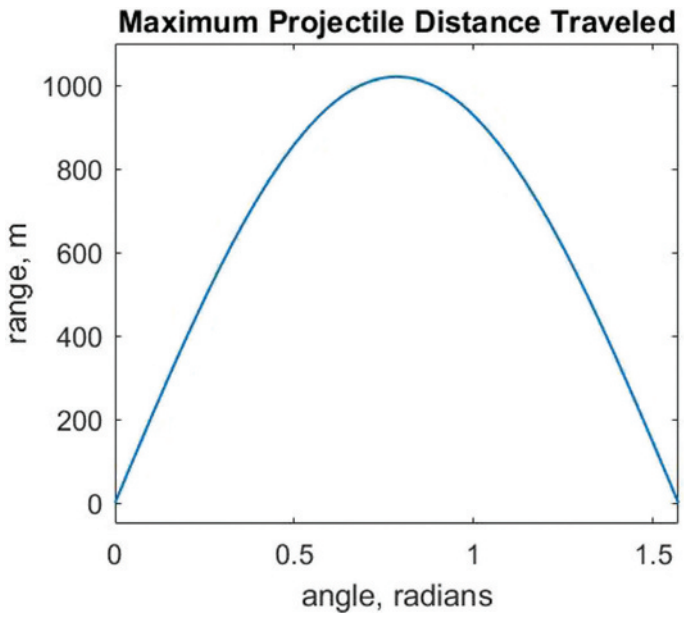


Figure 12.11
The projectile range as a function of launch angle.

(*continued*)

4. Develop a MATLAB solution
Recall that the symbolic expression for the impact distance with v_0 and g defined as 100 m/s and 9.8 m/s^2, respectively, is

```
impact_100 =
100000/49*sin(theta)*cos(theta)
```

From the graph, we can see that the maximum distance occurs when the slope is equal to zero. The slope is the derivative of `impact_100`, so the appropriate strategy is to set the derivative equal to zero and solve. Since MATLAB automatically assumes that an expression is equal to zero, the solution becomes

```
max_angle = solve(diff(impact_100))
```

which returns the angle at which the maximum height occurs:

```
max_angle =
pi/4
```

Now the result can be substituted into the expression for the range:

```
max_distance = subs(impact_100,theta,max_angle)
```

Finally, the result should be changed to a double precision number

```
double(max_distance)
ans =
1.0204e+003
```

12.5.2 Integration

KEY IDEA
Integration is the opposite of taking the derivative.

Integration can be thought of as the opposite of differentiation (finding a derivative) and is even sometimes called the **antiderivative**. It is commonly visualized as the area under a curve. For example, work done by a piston–cylinder device (Figure 12.12a) as it moves up or down can be calculated by taking the integral of $P\mathrm{d}V$—that is,

$$W = \int_1^2 P\mathrm{d}V$$

In order to do the calculation, it is necessary to know how P changes with V. If, for example, P is a constant, the resulting graph is a horizontal line as shown in Figure 12.12(b).

The work consumed or produced as the piston moves is the area under the curve from the initial volume to the final volume. For example, if the piston moves from 1 cm^3 to 4 cm^3, the work would correspond to the area shown in Figure 12.12b.

As you may know from introductory calculus, the integration is:

$$W = \int_1^2 P\mathrm{d}V = P\int_1^2 \mathrm{d}V = PV|_1^2 = PV_2 - PV_1 = P\,\Delta V$$

If

$$P = 100 \text{ psia, and } \Delta V = 3 \text{ cm}^3$$

then

$$W = 3 \text{ cm}^3 \times 100 \text{ psia}$$

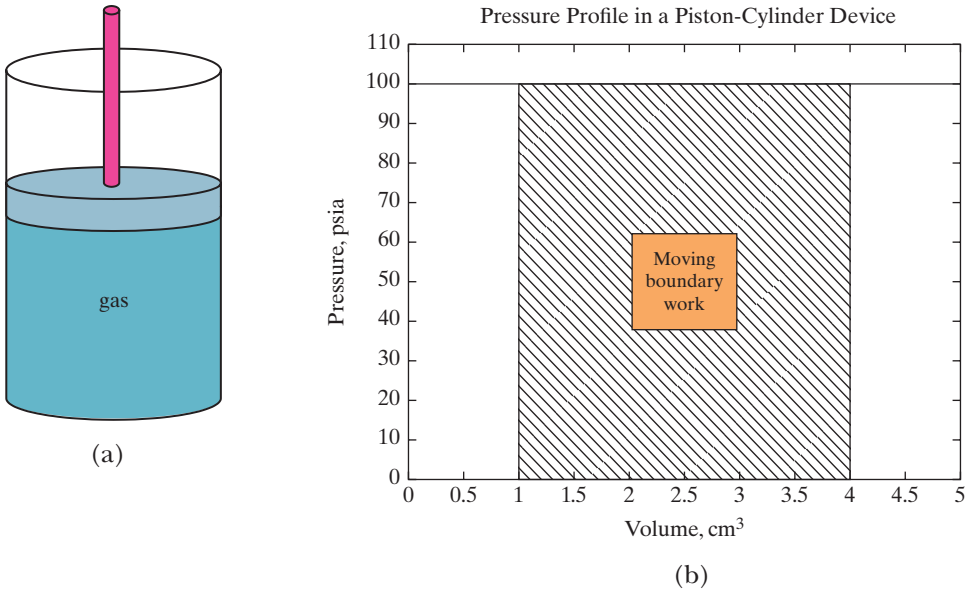


Figure 12.12
(a) Piston-cylinder device. (b) The work produced in a piston-cylinder device is the area under the Pressure-Volume curve. In this example the pressure is constant.

The symbolic toolbox allows us to easily take integrals of some very complicated functions. For example, to find an indefinite integral (an integral for which the boundary values of the variable are not specified), use the `int` function. First, specify a function:

```
syms x
y = x^3 + sin(x)
```

Then use the `int` to find the integral:

```
int(y)
ans =
x^4/4-cos(x)
```

The `int` function uses `x` as the default variable. For example, for a function with two variables, the `int` function will find the integral with respect to `x` or the variable closest alphabetically to `x`:

```
syms x t
y = x^3 +sin(t)
int(y)
ans =
x^4/4+sin(t)*x
```

To take the integral with respect to a user-defined variable, that variable needs to be specified in the second field of the `int` function:

```
int(y,t)
ans =
t*x^3-cos(t)
```

To find the definite integral, specify the range of interest. Consider this expression:

```
y = x^2
```

If the range of interest isn't specified, we get

```
int(y)
ans =
x^3/3
```

To evaluate this expression from 2 to 3 use the `subs` function:

```
yy = int(y)
yy =
x^3/3
subs(yy,3) - subs(yy,2)
ans =
  19/3
```

Notice that the result of the `subs` function is a symbolic; to change it to a floating-point representation, use the `double` function.

```
double(ans)
ans =
  6.3333
```

An alternate approach would be to use function syntax and define `y` as a symbolic function:

```
y(x) = x^2
```

The `int` function handles symbolic functions the same way it handles simple symbolic expressions:

```
yy = int(y)
```

As expected, the result is a function:

```
yy(x) =
x^3/3
```

To evaluate `yy` between the limits of integration (2 and 3) substitute in for `x`

```
yy(3)-yy(2)
```

which returns

```
ans =
19/3
```

A simpler approach to evaluating an integral between two points is to specify the bounds in the `int` function:

```
int(y,2,3)
ans =
19/3
```

To specify both the variable and the bounds, list them all as function input parameters:

```
syms x z
y = sin(x)+cos(z)
int(y,z,2,3)
ans =
sin(3)-sin(2)+sin(x)
```

The bounds can be numeric, or they can be symbolic variables:

```
syms b c
int(y,z,b,c)
ans =
sin(c) - sin(b) - sin(x)*(b-c)
```

Table 12.5 lists several of the ways to use the MATLAB symbolic integration function. The input expression, `y`, can be either a simple symbolic expression or a symbolic function.

Table 12.5 Symbolic Integration

`int(f)`	Returns the integral of the expression `f` with respect to the default independent variable	`syms x z`
		`y = x^3+z^2`
		`int(y)`
		`ans =`
		`1/4*x^4+z^2*x`

`int(f,t)`	Returns the integral of the expression `f` with respect to the variable `t`	`syms x z`
		`y = x^3+z^2`
		`int(y,z)`
		`ans =`
		`x^3*z+1/3*z^3`

`int(f,a,b)`	Returns the integral, with respect to the default variable, of the expression `f` between the numeric bounds `a` and `b`	`syms x z`
		`y = x^3+z^2`
		`int(y,2,3)`
		`ans =`
		`65/4+z^2`

`int(f,t,a,b)`	Returns the integral, with respect to the variable `t`, of the expression `f` between the numeric bounds `a` and `b`	`syms x z`
		`y = x^3+z^2`
		`int(y,z,2,3)`
		`ans =`
		`x^3+19/3`

`int(f,t,a,b)`	Returns the integral, with respect to the variable `t`, of the expression `f` between the symbolic bounds `a` and `b`	`syms x z a b`
		`y = x^3+z^2`
		`int(y,z,a,b)`
		`ans =`
		`-a^3/3-a*x^3+b^3/3+b*x^3`

PRACTICE EXERCISE 12.9

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Integrate the following expressions with respect to x:
   $x^2 + x + 1$
   $\sin(x)$
   $\tan(x)$
   $\ln(x)$
2. Integrate the following expressions with respect to x:
   $ax^2 + bx + c$
   $x^{0.5} - 3y$
   $\tan(x + y)$
   $3x + 4y - 3xy$
3. Perform a double integration with respect to x for each of the expressions in Exercise 1 and 2.
4. Integrate the following expressions with respect to y:
   $y^2 - 1$
   $2y + 3x^2$
   $ay + bx + cz$
5. Perform a double integration with respect to y for each of the expressions in Exercise 12.4.
6. Integrate each of the expressions in Exercise 1 with respect to x from 0 to 5.

EXAMPLE 12.6

USING SYMBOLIC MATH TO FIND WORK PRODUCED IN A PISTON–CYLINDER DEVICE

Piston–cylinder devices are used in a wide range of scientific instrumentation and engineering devices. One of the most pervasive is the internal combustion engine (see Figure 12.13), which typically uses four to eight cylinders.



Figure 12.13
Internal combustion engine. (Photostar72/Shutterstock)

The work produced by a piston–cylinder device depends on the pressure inside the cylinder and the amount the piston moves, resulting in a change in volume inside the cylinder. Mathematically,

$$W = \int P \mathrm{d}V$$

In order to integrate this equation, it is necessary to understand how the pressure changes with the volume. A common assumption is to model combustion gases as air and assume that they follow the ideal gas law

$$PV = nRT$$

where

P = pressure, kPa,
V = volume, m^3,
n = number of moles, kmol,
R = universal gas constant, 8.314 kPa m^3/kmol K, and
T = temperature, K.

Assuming that there is 1 mole of gas at 300 K, and that the temperature stays constant during the process, use these equations to calculate the work either done on the gas or produced by the gas as it expands or contracts between two known volumes.

1. State the problem
Calculate the work done per mole in an isothermal (constant-temperature) piston–cylinder device as the gas expands or contracts between two known volumes.

2. Describe the input and output
Input
Temperature = 300 K
Universal gas constant = 8.314 kPa m^3/kmol K = 8.314 kJ/kmol K
Arbitrary values of initial and final volume; for this example, we'll use

$$\text{initial volume} = 1\,\mathrm{m}^3$$
$$\text{final volume} = 5\,\mathrm{m}^3$$

Output
Work produced by the piston–cylinder device, in kJ.

3. Develop a hand example
First, solve the ideal gas law for P:

$$PV = nRT$$
$$P = nRT/V$$

Since n, R, and T are constant during the process, it is possible to perform the integration:

$$W = \int \frac{nRT}{V}\,\mathrm{d}V = nRT\int \frac{\mathrm{d}V}{V} = nRT\,\ln\left(\frac{V_2}{V_1}\right)$$

Substituting the known values gives:

$$W = 1\,\mathrm{kmol} \times 8.314\ \mathrm{kJ/kmol\ K} \times 300\ \mathrm{K} \times \ln\left(\frac{V_2}{V_1}\right)$$

For the arbitrary values $V_1 = 1\ \mathrm{m}^3$ and $V_2 = 5\ \mathrm{m}^3$, the work becomes

$$W = 4014\ \mathrm{kJ}$$

(*continued*)

Because the work is positive, it is produced *by* (not *on*) the system.

4. Develop a MATLAB solution
First, solve the ideal gas law for pressure. The code

```
syms P V n R T V1 V2                % Define variables
ideal_gas_law = P*V == n*R*T        % Define ideal gas law
P = solve(ideal_gas_law,P)          % Solve for P
```

returns

```
P =
n*R*T/V
```

Use the `int` function to perform the integration between V1 and V2:

```
W = int(P,V,V1,V2)          % Integrate P with respect
                            % to V from V1 to V2
```

returns

```
W =
piecewise([V1 <= 0 and 0 <= V2, R*T*n*int(1/V, V == V1..,
V2)],
[0 < V1 or V2 < 0, -R*T*n*(log(V1) - log(V2))])
```

This odd answer occurs because the log of zero or a negative number is undefined.
Finally, substitute numeric values into the equation

```
work=subs(W,[n,R,V1,V2,T],[1,8.314,1,5,300.0])
```

giving

```
work =(12471*log(5))/5
```

which can be simplified with the `double` function

```
work = double(work)
work =
4.0143e+03
```

5. Test the solution
The most obvious test is to compare the hand and computer solutions. However, the same answer with both techniques just means that we did the calculations the same way. One way to check reasonability would be to create a *PV* plot and estimate the area under the curve.
To create the plot, return to the equation for P and substitute values for n, R, and T:

```
p=subs(P,[n,R,T],[1,8.314,300])
```

This returns the following equation for P:

```
p =
12471/5/V
```

Now, use `fplot` to create a graph of P versus V (see Figure 12.14(a)):

```
fplot(p,[1,5])  % Plot the pressure vs. V
title("(a) Pressure change with Volume for an Isothermal
System")
```

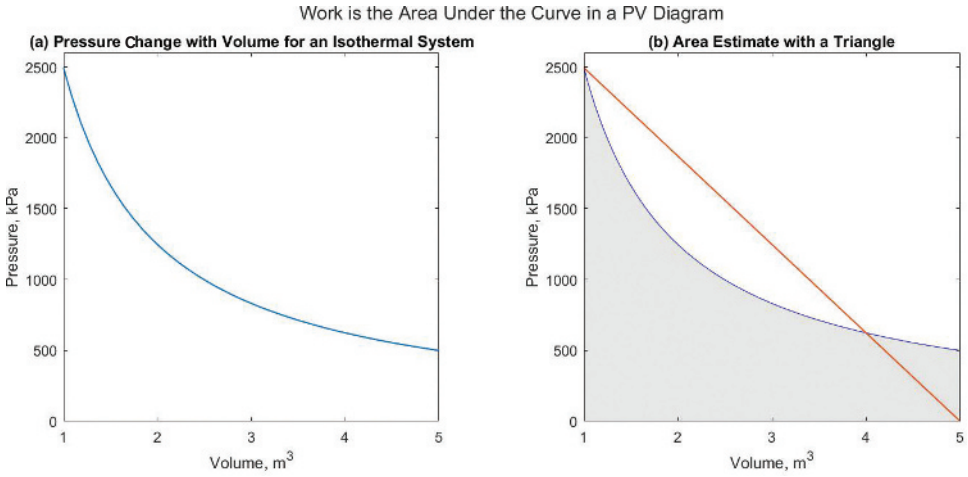


Figure 12.14
(a) For an isothermal system, as the volume increases the pressure decreases.
(b) Use a simple shape (a triangle) to estimate the area under the curve.

```
xlabel("Volume, m^3"), ylabel("Pressure, kPa")
axis([1,5,0,2600])
```

To estimate the work, find the area of a triangle that approximates the shape shown in Figure 12.14(b).

$$\text{area} = \frac{1}{2}\,\text{base} * \text{height}$$
$$\text{area} = 0.5 * (5 - 1) * 2400 = 4800$$

which corresponds to 4800 kJ. This matches nicely with the calculated value of 4014 kJ.

Now that the program calculations have been verified, it can be modified to prompt the user to enter values for any change in volume:

```
clear, clc
syms P V n R T V1 V2                % Define variables
ideal_gas_law = P*V == n*R*T        % Define ideal gas law
P = solve(ideal_gas_law,P)          % Solve for P
W = int(P,V,V1,V2)                  % Integrate to find work
% Now let the user input the data
temp=input("Enter a temperature: ")
v1=input("Enter the intial volume: ")
v2=input("Enter the final volume: ")
work=subs(W,[n,R,V1,V2,T],[1,8.314,v1,v2,temp])
double(work)
```

This program generates the following user interaction:

```
Enter a temperature: 300
temp =
   300
Enter the initial volume: 1
v1 =
   1
Enter the final volume: 5
```

(*continued*)

```
v2 =
   5
work =
   4.0143e+003
```

KEY IDEA
The default independent variable for differential equations in MATLAB is *t*.

12.6 DIFFERENTIAL EQUATIONS

Differential equations contain both a dependent variable and its derivative with respect to an independent variable. For example,

$$\frac{dy}{dt} = y$$

is a differential equation, where y is the dependent variable and t is the independent variable.

Although any symbol can be used for either the independent or the dependent variable, the default independent variable in MATLAB for ordinary differential equations is t (and it is the usual choice for most ordinary differential equation formulations). Consider this simple equation:

$$y = e^t$$

The derivative of y with respect to t is

$$\frac{dy}{dt} = e^t$$

This equation could also be expressed as a differential equation, since $y = e^t$:

$$\frac{dy}{dt} = y$$

The solution to a differential equation is an expression for y in terms of t. Differential equations typically have more than one solution. The following family of functions of t could be expressed by the same differential equation ($\mathrm{d}y/\mathrm{d}t = y$):

$$y = C_1 e^t$$

A particular equation of interest is defined by specifying an initial condition. For example, if

$$y(0) = 1,$$

then

$$C_1 = 1$$

A slightly more complicated function of y might be

$$y = t^2$$

The derivative of y with respect to t is

$$\frac{\mathrm{d}y}{\mathrm{d}t} = 2t$$

This equation could be rewritten as:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \frac{2t^2}{t} = \frac{2y}{t}$$

Since this equation contains both a derivative (dy/dt) and the dependent variable (y) it is a differential equation. Solving a differential equation means solving for the dependent variable (y) in terms of the independent variable (t).

The symbolic toolbox function `dsolve` solves differential equations, that is, it solves for y in terms of t.

12.6.1 First-Order Differential Equations

The `dsolve` function requires a differential equation, formulated with symbolic functions, as input. If we name the dependent variable y and the independent variable t, the result (the solution) will be a function where `y` is a function of `t`. To create the input for `dsolve` for a function of this form, the first step is to define a symbolic variable for the `y` function:

```
syms y(t)
```

Next the differential equation is defined. If we use

$$\frac{dy}{dt} = \frac{2y}{t}$$

as an example problem, we can define a variable called `eqn` and use the `diff` function to find the derivative of $y(t)$.

```
eqn = diff(y(t),t)==y(t)/t
```

An alternative syntax is:

```
eqn = diff(y,t)==y/t
```

As long as `y` has been defined as a symbolic function you may either use `y(t)` or just the function handle, `y`, in the equation definition. The result is the same, a symbolic function:

```
eqn(t) =
diff(y(t), t) == (2* y(t))/t
```

The `dsolve` function accepts the `eqn` variable as input and returns the solution to the differential equation.

```
dsolve(eqn)
 ans =
C1*t^2
```

You could also enter the function directly into `dsolve` and get the same result:

```
dsolve(diff(y,t)==2*y/t)
 ans =
C1*t^2
```

Differential equations have more than one solution, unless a boundary condition is specified. The constant, `C1`, has an infinite number of values. The `dsolve` function accepts boundary conditions as an equation in the second input field. You may either define a separate equation and give it a name or define it directly in the

dsolve function. For example, to set a boundary condition of y equal to 1 when t is equal to –1, enter y(-1)==1 in the second input field.

```
dsolve(eqn,y(-1)==1)
ans =
t^2
```

Thus, for these boundary conditions $C1=1$. Different boundary conditions would yield different values of C, and not all boundary conditions yield solutions.

12.6.2 Higher-Order Differential Equations

Higher-order differential equations are solved in a similar manner to first-order equations. Take for example the differential equation:

$$100\frac{d^2x}{dt^2} + 4\frac{dx}{dt} + 20x = 0$$

Assign boundary conditions $x(0)=1$ and $dx(0)= 1$. In addition to defining a symbolic function for x, a symbolic function for dx, which we'll call Dx is also needed for the boundary condition definition. It is created automatically when the diff function operates on x.

```
syms x(t)
Dx=diff(x)
```

The equation definition is longer but follows the same rules as the previous example. Two boundary conditions are required in dsolve because the equation is a second-order differential equation. First define the differential equation:

```
eqn=100*diff(x,t,2)+4*diff(x,t)+20*x==0
```

which returns

```
eqn(t) =20*x(t) + 200*diff(x(t),t) + 20*diff(x(t),t,t) == 0
```

Then solve the problem using dsolve:

```
dsolve(eqn,x(0)==1,Dx(0)==1)

ans =(6^(1/2)*exp(t*(2*6^(1/2) - 5))*(6^(1/2) + 3))/12 ...
 - exp(-t*(2*6^(1/2) + 5))*(6^(1/2)/4 - 1/2)
```

12.6.3 Systems of Differential Equations

To solve systems of differential equations such as

$$\frac{dx}{dt} = y$$
$$\frac{dy}{dt} = x$$

first, define symbolic functions

```
syms x(t) y(t)
```

then solve by listing the equations in dsolve.

```
a=dsolve(diff(x)==y,diff(y)==x)
a =
  struct with fields:
    y: [1×1 sym]
    x: [1×1 sym]
```

The results are a structure array. Boundary conditions may be added in the usual manner. One boundary condition is needed for each variable in a linear system of differential equations. Additional boundary conditions must be added if higher-order derivatives are included. The individual results can be retrieved using the typical structure array syntax.

```
a.y
ans =
C1*exp(t) + C2*exp(-t)
a.x
ans =
C1*exp(t) - C2*exp(-t)
```

KEY IDEA
Not every differential equation can be solved analytically.

MATLAB cannot solve every differential equation symbolically. Many differential equations can't be solved analytically at all, no matter how sophisticated the tool is. For those equations, numerical techniques are required.

12.7 CONVERTING SYMBOLIC EXPRESSIONS TO ANONYMOUS FUNCTIONS

It is often useful to evaluate mathematical expressions symbolically before using the results in more traditional MATLAB functions. To accomplish this, the `matlabFunction` function converts a symbolic expression into an anonymous function. Here's a very simple example.

```
syms x
y=cos(x)
dy=diff(y)
```

which returns the derivative of $\cos(x)$

```
dy =
-sin(x)
```

To convert this symbolic variable, `dy`, into an anonymous function use the following approach:

```
f=matlabFunction(dy)
```

which returns

```
f =
       @x -sin(x)
```

Now `f` can be used to evaluate $-\sin(x)$. For example to evaluate $-\sin(x)$ at $x = 2$

```
f(2)
ans =
     -0.9093
```

Here's a more complicated example, which also involves symbolically finding a derivative. This time we'll use function notation, which is an equally valid approach.

```
syms x
y(x)=(exp(-x)-1)/x
dy=diff(y)
g=matlabFunction(dy)
```

which results in a new anonymous function called `g`:

```
g=
  @(x)-exp(-x)./x-1.0./x.^2.*(exp(-x)-1.0)
```

Anonymous functions can be used like any other MATLAB function.

SUMMARY

The symbolic toolbox is an optional component of the professional version of MATLAB, but is included with the Student Version.

Symbolic variables are created in MATLAB with the `syms` command:

```
syms x
```

Multiple symbolic variables can be created in one statement:

```
syms a b c
```

Once symbolic variables have been defined, they can be used to create more complicated expressions. Since `x`, `a`, `b`, and `c` were defined as symbolic variables, they can be combined to create the quadratic equation:

```
EX = a*x^2 + b*x + c
```

MATLAB allows users to manipulate symbolic expressions, symbolic functions (a type of expression), and symbolic equations. Equations are set equal to something; expressions are not. All the statements in this summary so far have created expressions. By contrast, the statement

```
EQ = n == m/MW
```

defines a symbolic equation.

Both symbolic expressions and equations can be manipulated by using built-in MATLAB functions from the symbolic toolbox. The `numden` function extracts the numerator and denominator from an expression but is not valid for equations. The `expand`, `factor`, and `collect` functions can be used to modify either an expression or an equation. The `simplify` function simplifies an expression or an equation on the basis of built-in rules.

A highly useful symbolic function is `solve`, which allows the user to solve equations symbolically. If the input to the function is an expression, MATLAB sets the expression equal to zero. The `solve` function can solve not only a single equation for the specified variable, but also systems of equations. Unlike the techniques used in matrix algebra to solve systems of equations, the input to `solve` need not be linear.

The substitution function, `subs`, allows the user to replace variables with either numeric values or new variables. When `y` is defined as

```
syms m n p
y = m +2*n + p
```

the command

```
subs(y,[m,n,p], [1,2,3])
```

returns

```
ans =
    8
```

The `subs` command can be used to substitute both numeric values and symbolic variables.

An alternative strategy is to define expressions using function notation. This allows for easy substitution of values into expressions. Functions can be created directly using `syms`, as in:

```
syms f(x,y)
```

or existing expressions can be converted to functions using the `symfun` function.

Units are added to the MATLAB workspace using `symunit`. The application allows you to add units to symbolic variables, perform calculations with units, and convert between units.

MATLAB's symbolic plotting capability roughly mirrors the standard plotting options. The most useful of these plots for engineers and scientists is probably the $x-y$ plot, `fplot`. This function accepts a symbolic expression and plots it for values of the independent variable from -2π to $+2\pi$. The user can also assign the minimum and maximum values of the independent variable. Symbolic plots are annotated with the use of the same syntax as standard MATLAB plots.

The symbolic toolbox includes a number of calculus functions, the most basic being `diff` (differentiation) and `int` (integration). The `diff` function allows the user to take the derivative with respect to a default variable (`x` or whatever is closest alphabetically to `x` in the expression) or to specify the differentiation variable. Higher-order derivatives can also be specified. The `int` function also allows the user to integrate with respect to the default variable (`x`) or to specify the integration variable. Both definite and indefinite integrals can be evaluated. Additional calculus functions not discussed in this chapter are available. Use the `help` function for more information.

Differential equations can be solved with the `dsolve` function.

When solving a problem it is often useful to manipulate expressions symbolically before creating MATLAB functions. The `matlabFunction` function allows you to do this easily.

MATLAB SUMMARY

The following MATLAB summary lists special characters, commands, and functions that are defined in this chapter:

Commands and Functions	
`collect`	collects like terms
`diff`	finds the symbolic derivative of a symbolic expression
`double`	converts a symbolic variable into a double-precision floating point number
`dsolve`	differential equation solver
`expand`	expands an expression or equation
`ezpolar`	creates a polar plot
`fcontour`	creates a contour plot
`fimplicit`	creates a 2-D plot from an implicit equation
`fimplicit3`	creates a 3-D plot from an implicit equation
`fmesh`	creates a mesh plot from a symbolic expression
`fplot`	plots a symbolic expression (creates an $x - y$ plot)
`fplot3`	creates a three-dimensional line plot
`fsurf`	creates a surface plot from a symbolic expression
`factor`	factors an expression or equation
`int`	finds the symbolic integral of a symbolic expression
`matlabFunction`	converts a symbolic expression into an anonymous MATLAB function
`numden`	extracts the numerator and denominator from an expression
`poly2sym`	converts an array representing a polynomial into a symbolic
`separateUnits`	separates a variable into units and value components
`simplify`	simplifies using built-in simplification rules
`solve`	solves a symbolic expression or equation
`subs`	substitutes into a symbolic expression or equation
`sym`	creates a symbolic variable, expression, or equation
`sym2poly`	converts a symbolic polynomial into an array
`symfun`	creates a symbolic function
`syms`	creates symbolic variables
`symunit`	adds physical units of measurement to the workspace
`symvar`	determines the symbolic variables in a function and returns them in the alphabetical order closest to x
`unitConvert`	converts units

KEY TERMS

antiderivative
derivative
equation
expression
function
integration
linear
nonlinear
numeric
object
operator
overloaded function
parametric equations
root
symbolic
variable

PROBLEMS

Symbolic Algebra

12.1 Create the symbolic variables

```
a b c d x
```

and use them to create the following symbolic expressions:

```
(a) se1 = x^3 - 3*x^2 + x
(b) se2 = sin(x) + tan(x)
(c) se3 = (2*x^2 - 3*x - 2)/(x^2 - 5*x)
(d) se4 = (x^2 -9)/(x+3)
```

12.2 Divide `se1` by `se2`.

(a) Multiply `se3` by `se4`.
(b) Divide `se1` by `x`.
(c) Add `se1` to `se3`.

12.3 Create the symbolic variables

```
a b c d x
```

and use them to create the following symbolic functions:

```
(a) fse1(x) = x^3 - 3*x^2 + x
(b) fse2(x) = sin(x) + tan(x)
(c) fse3(x) = (2*x^2 - 3*x - 2)/(x^2 - 5*x)
(d) fse4(x) = (x^2 -9)/(x+3)
```

12.4 Divide `fse1` by `fse2`.

(a) Multiply `fse3` by `fse4`.
(b) Divide `fse1` by `x`.
(c) Add `fse1` to `fse3`.

12.5 Create the following symbolic equations:

```
(a) sq1 = x^2 + y^2 == 4
(b) sq2 = 5*x^5 - 4*x^4 + 3*x^3 + 2*x^2 - x == 24
(c) sq3 = sin(a) + cos(b)-x*c == d
(d) sq4 = (x^3 -3*x)/(x-3) == 14
```

12.6 Try to use the `numden` function to extract numerator and denominator from `se4, fse4,` and `sq4`. Does this function work for both expressions and equations? Describe how your results vary. Explain the differences.

12.7 Use the `expand`, `factor`, `collect`, and `simplify` functions on `se1` to `se4, fse1 to fse4,` and on `sq1` to `sq4`. In your own words, describe how these functions work for the various types of equations and expressions. (Recall that functions are a type of expression.)

Solving Symbolically and Using the Subs Command

12.8 Solve each of the expressions created in Problem 12.1 for `x`.

12.9 Solve each of the functions created in Problem 12.3 for `x`.

12.10 Solve each of the equations created in problem 12.5 for `x`.

12.11 Solve equation `sq3`, created in Problem 12.5 for `a`.

12.12 A pendulum is a rigid object suspended from a frictionless pivot point (see Figure P12.12). If the pendulum is allowed to swing back and forth with a given inertia, we can find the frequency of oscillation with the equation

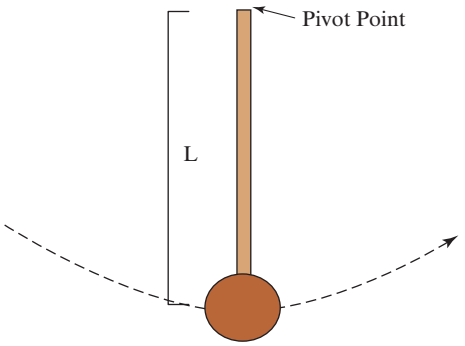


Figure P12.12
Pendulum described in Problem 12.12.

$$2\pi f = \sqrt{\frac{mgL}{I}}$$

where

f = frequency,
m = mass of the pendulum,
g = acceleration due to gravity,
L = distance from the pivot point to the center of gravity of the pendulum, and
I = inertia.

Use MATLAB's symbolic capability to solve for the length L.

12.13 Let the mass, inertia, and frequency of the pendulum in the previous problem be, respectively,

$$m = 10\ \text{kg}$$
$$f = 0.2\ \text{s}^{-1}$$
$$I = 60\ \text{kg m/s}.$$

If the pendulum is on the earth ($g = 9.8\ \text{m/s}^2$) what is the length from the pivot point to the center of gravity? (Use the `subs` function to solve this problem.)

12.14 Kinetic energy is defined as

$$\text{KE} = \frac{1}{2}mV^2$$

where

KE = kinetic energy, measured in J
m = mass, measured in kg
V = velocity, measured in m/s.

Create a symbolic equation for kinetic energy, and solve it for velocity.

12.15 Find the kinetic energy of a car that weighs 2000 lbm and is traveling at 60 mph (see Figure P12.15). Your units will be lbm mile^2/h^2. Once you've calculated this result, change it to Btu by using the following conversion factors:

$$\begin{aligned} 1 \text{ lbf} &= 32.174 \text{ lbm} \cdot \text{ft/s}^2 \\ 1 \text{ h} &= 3600 \text{ s} \\ 1 \text{ mile} &= 5280 \text{ ft} \\ 1 \text{ Btu} &= 778.169 \text{ ft} \cdot \text{lbf} \end{aligned}$$

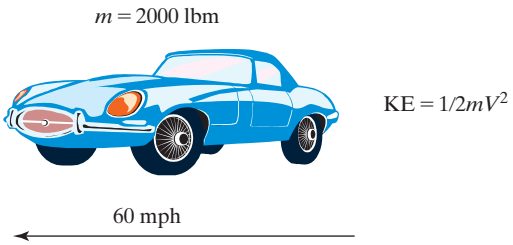


Figure P12.15 Car described in Problem 12.15.

12.16 Repeat the previous problem using symunit to attach units to your calculations. Convert to Btu using the built-in conversion function, `unitConvert`.

12.17 The heat capacity of a gas can be modeled with the following equation, composed of the empirical constants a, b, c, and d and the temperature T in kelvins:

$$C_P = a + bT + cT^2 + dT^3$$

Empirical constants do not have a physical meaning, but are used to make the equation fit the data. Create a symbolic equation for heat capacity and solve it for T.

12.18 Substitute the following values for a, b, c, and d into the heat-capacity equation from the previous problem and give your result a new name [these values model the heat capacity of nitrogen gas in kJ/(kmol K) as it changes temperature between approximately 273 and 1800 K]:

$$\begin{aligned} a &= 28.90 \\ b &= -0.1571 \times 10^{-2} \\ c &= 0.8081 \times 10^{-5} \\ d &= -2.873 \times 10^{-9} \end{aligned}$$

Solve your new equation for T if the heat capacity (C_p) is equal to 29.15 kJ/(kmol K).

12.19 The Antoine equation uses empirical constants to model the vapor pressure of a gas as a function of temperature. The model equation is

$$\log_{10}(P) = A - \frac{B}{C + T}$$

where

P = pressure, in mmHg,
A = empirical constant,
B = empirical constant,
C = empirical constant, and
T = temperature in °C.

The normal boiling point of a liquid is the temperature at which the vapor pressure (P) of the gas is equal to atmospheric pressure, 760 mmHg. Use MATLAB's symbolic capability to find the normal boiling point of benzene if the empirical constants are

$$\begin{aligned} A &= 6.89272 \\ B &= 1203.531 \\ C &= 219.888 \end{aligned}$$

12.20 Two hungry college students go to the cafeteria and buy lunch. The next day they spend twice as much. The third day they spend \$1 less than they did the second day. At the end of 3 days they have spent \$70. How much did they spend each day? Use MATLAB's symbolic capability to help you solve this problem.

Solving Systems of Equations

12.21 Consider the following set of seven equations:

$$\begin{aligned}
3x_1 + 4x_2 + 2x_3 - x_4 + x_5 + 7x_6 + x_7 &= 42 \\
2x_1 - 2x_2 + 3x_3 - 4x_4 + 5x_5 + 2x_6 + 8x_7 &= 32 \\
x_1 + 2x_2 + 3x_3 + x_4 + 2x_5 + 4x_6 + 6x_7 &= 12 \\
5x_1 + 10x_2 + 4x_3 + 3x_4 + 9x_5 - 2x_6 + x_7 &= -15 \\
3x_1 + 2x_2 - 2x_3 - 4x_4 - 5x_5 - 6x_6 + 7x_7 &= 10 \\
-2x_1 + 9x_2 + x_3 + 3x_4 - 3x_5 + 5x_6 + x_7 &= 18 \\
x_1 - 2x_2 - 8x_3 + 4x_4 + 2x_5 + 4x_6 + 5x_7 &= 17
\end{aligned}$$

Define a symbolic equation for each of the seven equations, then use MATLAB's symbolic capability to solve for each unknown.

12.22 Compare the amount of time it takes to solve the preceding problem by using left division, and by using symbolic math with the `tic` and `toc` functions, whose syntax is

```
tic
⋮
code to be timed
⋮
toc
```

12.23 Use MATLAB's symbolic capabilities to solve the following problem.
Consider a separation process in which streams of water, ethanol, and methanol enter a process unit. Two streams leave the unit, each with varying amounts of the three components (see Figure P12.23).
Determine the mass flow rates into the system and out of the top and bottom of the separation unit.

(a) First, set up the following material-balance equations for each of the three components:

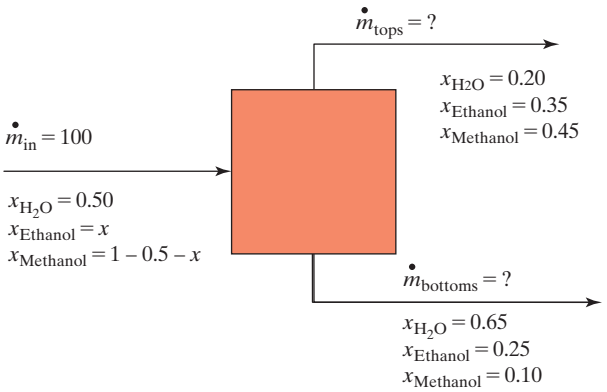


Figure P12.23
Separation process with three components: water, ethanol, and methanol.

Water

$$0.5(100) = 0.2\dot{m}_{tops} + 0.65\dot{m}_{bottoms}$$
$$50 = 0.2\dot{m}_{tops} + 0.65\dot{m}_{bottoms}$$

Ethanol

$$100x = 0.35\dot{m}_{tops} + 0.25\dot{m}_{bottoms}$$
$$0 = -100x + 0.35\dot{m}_{tops} + 0.25\dot{m}_{bottoms}$$

Methanol

$$100(1 - 0.5 - x) = 0.45\dot{m}_{tops} + 0.1\dot{m}_{bottoms}$$
$$50 = 100x + 0.45\dot{m}_{tops} + 0.1\dot{m}_{bottoms}$$

(b) Create symbolic equations to represent each material balance.
(c) Use the `solve` function to solve the system of three equations and three unknowns.

12.24 Consider the following two equations:

$$x^2 + y^2 = 42$$
$$x + 3y + 2y^2 = 6$$

Define a symbolic equation for each, and solve it by using MATLAB's symbolic capability. (You will need to use the `double` function on the answers to view the results numerically.) Could you solve these equations by using matrices?

Symbolic Plotting

12.25 Create plots of the following expressions from $x = 0$ to 10:
(a) $y = e^x$
(b) $y = \sin(x)$
(c) $y = ax^2 + bx + c$, where $a = 5$, $b = 2$, and $c = 4$
(d) $y = \sqrt{x}$

Each of your plots should include a title, an x-axis label, a y-axis label, and a grid.

12.26 Use `fplot` to graph the following expressions on the same figure for x-values from -2π to 2π (you'll need to use the `hold on` command):

$$y_1 = \sin(x)$$
$$y_2 = \sin(2x)$$
$$y_3 = \sin(3x)$$

Assign each line a different color and line style.

12.27 Use `fimplicit` to graph the following implicit equations:

(a) $x^2 + y^3 = 0$
(b) $x + x^2 - y = 0$
(c) $x^2 + 3y^2 = 3$
(d) $x \cdot y = 4$

12.28 Use `fplot` to graph the following parametric functions:

(a) $f_1(t) = x = \sin(t)$
$f_2(t) = y = \cos(t)$

(b) $f_1(t) = x = \sin(t)$
$f_2(t) = y = 3\cos(t)$

(c) $f_1(t) = x = \sin(t)$
$f_2(t) = y = \cos(3t)$

(d) $f_1(t) = x = 10\sin(t)$ from $t = 0$ to 30
$f_2(t) = y = t\cos(t)$

(e) $f_1(t) = x = t\ \sin(t)$ from $t = 0$ to 30
$f_2(t) = y = t\ \cos(t)$

12.29 The distance a projectile travels when fired at an angle θ is a function of time and can be divided into horizontal and vertical distances (see Figure P12.25), given respectively by

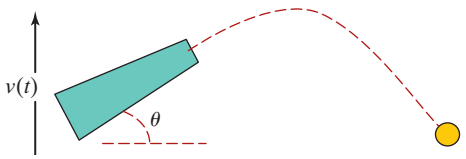


Figure P12.29
Trajectory of a projectile.

$$\text{horizontal}\ (t) = tV_0\ \cos(\theta)$$

and

$$\text{vertical}\ (t) = tV_0\ \sin(\theta) - \frac{1}{2}\ gt^2$$

where

horizontal = *distance* traveled in the *x* direction,
vertical = *distance* traveled in the *y* direction,
V_0 = initial velocity of the projectile,
g = acceleration due to gravity, 9.8 m/s^2, and
t = time, s.

Suppose a projectile is fired at an initial velocity of 100 m/s, and a launch angle of $\pi/4$ radians (45°). Use `fplot` to graph horizontal distance on the *x*-axis and vertical distance on the *y*-axis for times from 0 to 20 seconds.

12.30 For each of the following expressions, use the `ezpolar` plot function to create a graph of the expression, and use the `tiledlayout` function to put all four of your graphs in the same figure. Consult the help feature to determine the appropriate syntax:

(a) $\sin^2(\theta) + \cos^2(\theta)$
(b) $\sin(\theta)$
(c) $e^{\theta/5}$ for θ from 0 to 20
(d) $\sinh(\theta)$ for θ from 0 to 20

12.31 Use `fplot3` to create a three-dimensional line plot of the following functions:

$$\begin{aligned} f_1(t) &= x = t\,\sin(t) \\ f_2(t) &= y = t\,\cos(t) \\ f_3(t) &= z = t \end{aligned}$$

12.32 Use the following equation to create a symbolic function Z:

$$Z = \frac{\sin\left(\sqrt{X^2 + Y^2}\right)}{\sqrt{X^2 + Y^2}}$$

(a) Use the `fmesh` plotting function to create a three-dimensional plot of Z.
(b) Use the `fsurf` plotting function to create a three-dimensional plot of Z.
(c) Use `fcontour` to create a contour map of Z.

Use tiles (subplots) to put all the graphs you create into the same figure. Add a title for the entire figure.

Calculus

12.33 Determine the first and second derivatives of the following functions, using MATLAB's symbolic functions:

(a) $f_1(x) = y = x^3 - 4x^2 + 3x + 8$
(b) $f_2(x) = y = (x^2 - 2x + 1)(x - 1)$
(c) $f_3(x) = y = \cos(2x)\,\sin(x)$
(d) $f_4(x) = y = 3xe^{4x^2}$

12.34 Use MATLAB's symbolic functions to perform the following integrations:

(a) $\int (x^2 + x)\,dx$
(b) $\int_{0.3}^{1.3} (x^2 + x)\,dx$
(c) $\int (x^2 + y^2)\,dx$
(d) $\int_{3.5}^{24} (ax^2 + bx + c)\,dx$

12.35 Let the following polynomial represent the altitude in meters during the first 48 hours following the launch of a weather balloon:

$$h(t) = -0.12t^4 + 12t^3 - 380t^2 + 4100t + 220$$

Assume that the unit of t is hours.
(a) Use MATLAB together with the fact that the velocity is the first derivative of the altitude to determine the equation for the velocity of the balloon.
(b) Use MATLAB together with the fact that acceleration is the derivative of velocity, or the second derivative of the altitude, to determine the equation for the acceleration of the balloon.
(c) Use MATLAB to determine when the balloon hits the ground. Because $h(t)$ is a fourth-order polynomial, there will be four answers. However, only one answer will be physically meaningful.

(d) Use MATLAB's symbolic plotting capability to create plots of altitude, velocity, and acceleration from time 0 until the balloon hits the ground [which was determined in part (c)]. You'll need three separate plots, since altitude, velocity, and acceleration have different units.

(e) Determine the maximum height reached by the balloon.

Use the fact that the velocity of the balloon is zero at the maximum height.

12.36 Suppose that water is being pumped into an initially empty tank (see Figure P12.32). It is known that the rate of flow of water into the tank at time t (in seconds) is $50 - t$ l/s. The amount of water Q that flows into the tank during the first x seconds can be shown to be equal to the integral of the expression $(50 - t)$ evaluated from 0 to x seconds.*

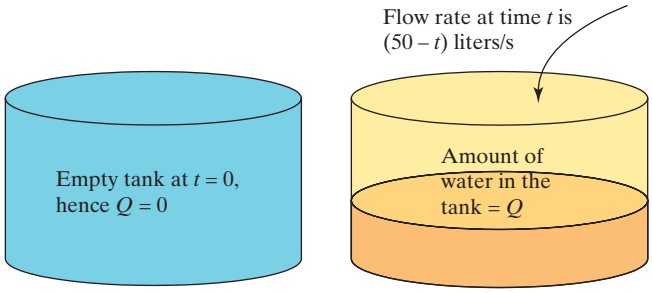


Figure P12.36
Tank-filling problem.

(a) Determine a symbolic equation that represents the amount of water in the tank after x seconds.

(b) Determine the amount of water in the tank after 30 seconds.

(c) Determine the amount of water that flowed into the tank between 10 and 15 seconds after the flow was initiated.

12.37 Consider a spring with the left end held fixed and the right end free to move along the x-axis (see Figure P12.37). We assume that the right end of the spring is at the origin $x = 0$ when the spring is at rest. When the spring is stretched, the right end of the spring is at some new value of x greater than zero. When the spring is compressed, the right end of the spring is at some value less than zero. Suppose that the spring has a natural length of 1 ft, and that a force of 10 lbf is required to compress it to a length of 0.5 ft. Then, it

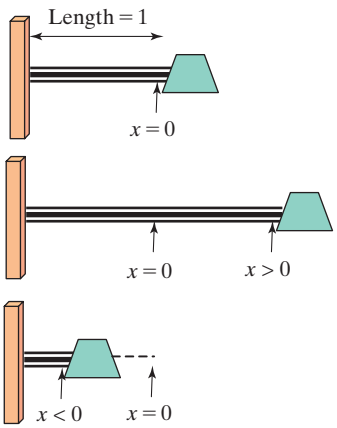


Figure P12.37
Spring problem described in Problem 12.37.

* From Etter, Kuncicky, and Moore, *Introduction to MATLAB 7* (Upper Saddle River, NJ: Pearson/Prentice Hall, 2005).

can be shown that the work, in ft lbf performed to stretch the spring from its natural length to a total of n ft, is equal to the integral of $20x$ over the interval from 0 to $n-1$.

(a) Use MATLAB to determine a symbolic expression that represents the amount of work necessary to stretch the spring to a total length of n ft.
(b) What is the amount of work done to stretch the spring to a total of 2 ft?
(c) If the amount of work exerted is 25 ft lbf, what is the length of the stretched spring?

12.38 The constant-pressure heat capacity C_p of a gas can be modeled with the empirical equation

$$C_p = a + bT + cT^2 + dT^3$$

where a, b, c, and d are empirical constants and T is the temperature in Kelvin. The change in enthalpy (a measure of energy) as a gas is heated from T_1 to T_2 is the integral of this equation with respect to T:

$$\Delta h = \int_{T_1}^{T_2} C_p \, dT$$

Find the change in enthalpy of oxygen gas as it is heated from 300 to 1000 K. The values of a, b, c, and d for oxygen are

$$\begin{aligned} a &= 25.48 \\ b &= 1.520 \times 10^{-2} \\ c &= -0.7155 \times 10^{-5} \\ d &= 1.312 \times 10^{-9} \end{aligned}$$

Creating Anonymous Functions from Symbolic Expressions

12.39 A first-order polynomial is often represented as

$$ax + b = 0$$

(a) Use the symbolic algebra capability in MATLAB to solve this equation for x.
(b) Use the `matlabFunction` function to convert your result from part (a) into an anonymous function.
(c) Evaluate your function with the following input:

$$\begin{aligned} a &= 4 \\ b &= 3 \end{aligned}$$

12.40 Consider the simple trigonometric function $\tan(x)$.

(a) Use the symbolic algebra capability in MATLAB to integrate this function.
(b) Use the `matlabFunction` function to convert your result from part a into an anonymous function.
(c) Use `fplot` to plot your function from -5 to $+5$.

CHAPTER

13 Numerical Techniques

Objectives

After reading this chapter, you should be able to:

- Interpolate between data points, using either linear or cubic spline models.
- Model a set of data points as a polynomial.
- Use the basic fitting tool to analyze data.
- Model data using the curve-fitting toolbox.
- Perform numerical differentiations.
- Perform numerical integrations.
- Solve differential equations numerically.

13.1 INTERPOLATION

Especially when measuring things, data isn't often gathered at every possible point. Consider a set of *x*–*y* data collected during an experiment. By using an **interpolation** technique, the value of *y* at values of *x* where measurements weren't taken can be estimated. (see Figure 13.1a). The two most common interpolation techniques are **linear interpolation** and **cubic spline** interpolation, both of which are supported by MATLAB.

KEY IDEA
The most common way to estimate a data point between two known points is linear interpolation.

13.1.1 Linear Interpolation

The most common way to estimate a data point between two known points is *linear interpolation*. In this technique, the function between the points is assumed to be a straight line, as shown in Figure 13.1a. Once the equation of a straight line defined by the two known points has been determined, we can find *y* for any value of *x*. The closer together the points are, the more accurate our **approximation** is likely to be.

Linear interpolation in MATLAB is performed with the `interp1` function. The data used to create the graph shown in Figure 13.1b are

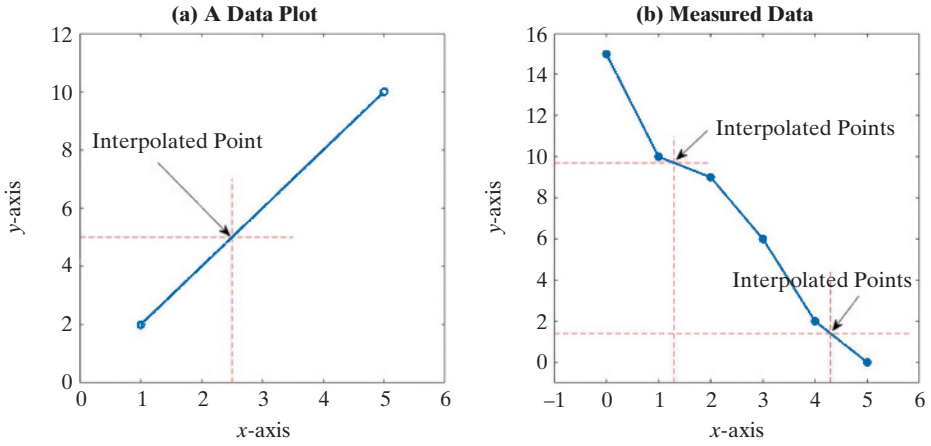


Figure 13.1
(a) In linear interpolation an intermediate value is estimated by assuming the dependent variable (*y*) changes linearly with the independent variable (*x*). This is equivalent to drawing a straight line between two points. (b) If the relationship between the dependent and independent variable is not actually linear, better approximations can be determined by gathering data points that are closer together.

HINT

Although possible, it is rarely wise to *extrapolate* past the region where you've collected data. It may be tempting to assume that data continue to follow the same pattern, but this assumption can lead to large errors.

HINT

The last character in the function name `interp1` is the number 1. Depending on the font, it may look like the lowercase letter "ell" (l).

```
x = 0:5;
y = [15, 10, 9, 6, 2, 0];
```

To perform a single interpolation, the input to `interp1` is the `x` data, the `y` data, and the new `x` value for which an estimate of `y` is requested. For example, to estimate the value of `y` when `x` is equal to 3.5, type

```
interp1(x,y,3.5)
ans =
   4
```

To perform multiple interpolations at the same time use a vector of *x*-values in the third field of the `interp1` function. For example, to estimate *y*-values for new *x*'s spaced evenly from 0 to 5 by 0.2, type

```
new_x = 0:0.2:5;
new_y = interp1(x,y,new_x)
```

which returns

```
new_y =
  Columns 1 through 5
  15.0000 14.0000 13.0000 12.0000 11.0000
...
  Column 26
       0
```

KEY IDEA
Most MATLAB functions accept text input as either character arrays or string arrays—so you can use either single or double quotes.

The results of the linear interpolations are plotted on the same graph with the original data in Figure 13.2a:

```
plot(x,y,new_x,new_y,'o')
```

(For simplicity, the commands used to add titles and axis labels to plots in this chapter have not been included.)

The `interp1` function defaults to linear interpolation to make its estimates. However, as we will see in the next section, other approaches are possible. To explicitly define the approach used in `interp1` as linear interpolation, specify it in the fourth field:

```
interp1(x,y,3.5,"linear")
ans =
  4
```

13.1.2 Cubic Spline Interpolation

Connecting data points with straight lines probably isn't the best way to estimate intermediate values, although it is surely the simplest. A smoother curve can be created by using the cubic spline interpolation technique, included in the `interp1` function. This approach uses a third-order polynomial and therefore requires at least 4 data points to model the behavior of the data. Cubic spline interpolation is specified in the fourth input field. For example to estimate the value of y when x equals 3.5, using cubic spline interpolation the code is:

```
interp1(x,y,3.5,"spline")
```

This command returns an improved estimate of

```
ans =
  3.9417
```

The cubic spline technique can also be used to create an array of new estimates for y for every member of an array of x-values:

```
new_y_spline =
interp1(x,y,new_x,"spline");
```

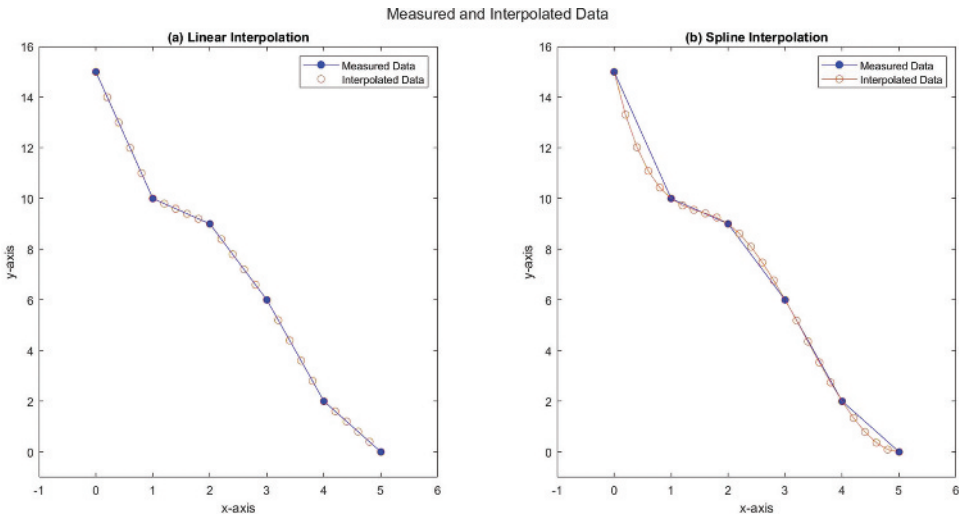


Figure 13.2
Both measured data points and interpolated data were plotted on the same graph. (a) In the linear interpolation case notice how all the points fall on the line between the measured points. (b) In the spline interpolation case the calculated points fall on the curve. Every measured point also falls on the curved line.

A plot of these data on the same graph as the measured data (see Figure 13.2b) using the command

```
plot(x,y,new_x,new_y_spline,'-o')
```

results in two different lines.

The curved line in Figure 13.2b was drawn with the use of the interpolated data points. The line composed of straight-line segments was drawn through just the original data.

Although the most common ways to interpolate between data points are linear and spline approaches, MATLAB does offer some other choices, as listed in Table 13.1.

Table 13.1 Interpolation Options in the `interp1` Function

`linear`	linear interpolation, which is the default	`interp1(x,y,3.5,"linear")`
		`ans =`
		`4`

`nearest`	nearest-neighbor interpolation, which returns the value of the nearest provided data point	`interp1(x,y,3.5,"nearest")`
		`ans =`
		`2`

`spline`	piecewise cubic spline interpolation	`interp1(x,y,3.5,"spline")`
		`ans =`
		`3.9417`

`pchip`	shape-preserving piecewise cubic interpolation	`interp1(x,y,3.5,"pchip")`
		`ans =`
		`3.9048`

`cubic`	same as `'pchip'`	`interp1(x,y,3.5,"cubic")`
		`ans =`
		`3.9048`

`v5cubic`	the cubic interpolation from MATLAB 5, which does not extrapolate and uses `'spline'` if `x` is not equally spaced	`interp1(x,y,3.5,"v5cubic")`
		`ans =`
		`3.9375`

EXAMPLE 13.1

THERMODYNAMIC PROPERTIES: USING THE STEAM TABLES

The subject of thermodynamics makes extensive use of tables. Although many thermodynamic properties can be described by fairly simple equations, others are either poorly understood, or the equations describing their behavior are very complicated. It is much easier to tabulate the values. For example, consider the values in Table 13.2 for steam at 0.1 MPa (approximately 1 atm) (see Figure 13.3).

Table 13.2 Internal Energy of Superheated Steam at 0.1 MPa, as a Function of Temperature

Temperature (°C)	Internal Energy, u (kJ/kg)
100	2506.7
150	2582.8
200	2658.1
250	2733.7
300	2810.4

(*continued*)

Table 13.2 (*Continued*)

Temperature (°C)	Internal Energy, u (kJ/kg)
400	2967.9
500	3131.6

Source: Data from Joseph H. Keenan, Frederick G. Keyes, Philip G. Hill, and Joan G. Moore, *Steam Tables, SI units* (New York: John Wiley & Sons, 1978).



Figure 13.3
Geysers spray high-temperature and high-pressure water and steam. (Johnmoorefour/Shutterstock)

Use linear interpolation to determine the internal energy at 215°C. Use linear interpolation to determine the temperature if the internal energy is 2600 kJ/kg.

1. State the problem
   Find the internal energy of steam, using linear interpolation.
   Find the temperature of the steam, using linear interpolation.
2. Describe the input and output

   Input Table of temperature and internal energy
   u unknown
   T unknown

   Output Internal energy
   Temperature
3. Develop a hand example
   In the first part of the problem, we need to find the internal energy at 215°C. The table includes values at 200°C and 250°C. First, determine the fraction of the distance between 200 and 250 at which the value 215 falls:

$$\frac{215 - 200}{250 - 200} = 0.30$$

   If we model the relationship between temperature and internal energy as linear, the internal energy should also be 30% of the distance between the tabulated values:

$$0.30 = \frac{u - 2658.1}{2733.7 - 2658.1}$$

   Solving for u gives

$$u = 2680.78 \text{ kJ/kg}$$

4. Develop a MATLAB solution
   Create the MATLAB solution in a script, then run it in the command environment:

```
%% Example 13.1
% Thermodynamics
```

```
clear, clc, format shortg
T = [100, 150, 200, 250, 300, 400, 500];
u = [2506.7, 2582.8, 2658.1, 2733.7, 2810.4, 2967.9, 3131.6];
newu = interp1(T,u,215,"linear")
newu = interp1(T,u,215,"spline")
newT = interp1(u,T,2600,"linear")
newT = interp1(u,T,2600,"spline")
```

The code returns

```
newu =
   2680.78
newT =
   161.42
```

5. Test the solution
   The MATLAB result matches the hand result. This approach could be used for any of the properties tabulated in the steam tables. The JANAF tables published by the National Institute of Standards and Technology are a similar source of thermodynamic properties.

EXAMPLE 13.2

THERMODYNAMIC PROPERTIES: EXPANDING THE STEAM TABLES

As seen in Example 13.1, thermodynamics makes extensive use of tables. Commonly, many experiments are performed at atmospheric pressure, so Table 13.3, which is just a portion of the steam tables is extremely useful (see Figure 13.4).

Notice that the table is spaced first at 50°C intervals, then at 100°C intervals. Suppose you have a project that requires you to use this table, and you prefer not to perform a linear interpolation every time you use it. Use MATLAB to create a table, employing linear interpolation, with a temperature spacing of 25°C.

1. State the problem
   Find the specific volume, internal energy, and enthalpy every 25°C.

Table 13.3 Properties of Superheated Steam at 0.1 MPa (Approximately 1 atm)

Temperature (°C)	Specific Volume, v (m^3/ kg)	Internal Energy, u (kJ/kg)	Enthalpy, h (kJ/kg)
100	1.6958	2506.7	2676.2
150	1.9364	2582.8	2776.4
200	2.172	2658.1	2875.3
250	2.406	2733.7	2974.3
300	2.639	2810.4	3074.3
400	3.103	2967.9	3278.2
500	3.565	3131.6	3488.1

Source: Data from Joseph H. Keenan, Frederick G. Keyes, Philip G. Hill, and Joan G. Moore, *Steam Tables, SI units* (New York: John Wiley & Sons, 1978).

(*continued*)

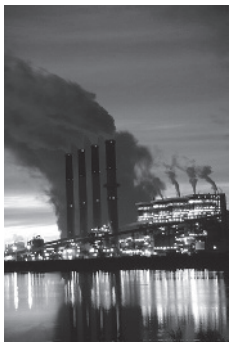


Figure 13.4
Power plants use steam as a "working fluid." (Lee Kline/Stock Connection Blue/Alamy Stock Photo)

2. Describe the input and output

Input	Table of temperature and internal energy New table interval of 25°C
Output	Table

3. Develop a hand example
In Example 13.1, we found the internal energy at 215°C. Since 215 is not on our output table, we'll redo the calculations at 225°C:

$$\frac{225 - 200}{250 - 200} = 0.50$$

and

$$0.50 = \frac{u - 2658.1}{2733.7 - 2658.1}$$

Solving for u gives

$$u = 2695.9 \text{ kJ/kg}$$

We can use this same calculation to confirm those in the table we create.

4. Develop a MATLAB solution
Create the MATLAB solution in a script, then run it in the command environment:

```
%% Example 13.2
% Thermodynamics
clear, clc
T = [100, 150, 200, 250, 300, 400, 500]';
v = [1.6958, 1.9364, 2.172, 2.406, 2.639, 3.103, 3.565]';
u = [2506.7, 2582.8, 2658.1, 2733.7, 2810.4, 2967.9, 3131.6]';
h = [2676.2, 2776.4, 2875.3, 2974.3, 3074.3, 3278.2, 3488.1]';
props = [v,u,h];
newT = [100:25:500]';
newprop = interp1(T,props,newT);
disp("Steam Properties at 0.1 MPa")
disp("Temp  Specific Volume Internal Energy Enthalpy")
disp("C  m^3/kg  kJ/kg  kJ/kg")
fprintf("%6.0f %10.4f  %8.1f  %8.1f \n",[newT,newprop]')
```

The program returns the following result:

```
Steam Properties at 0.1 MPa
Temp  Specific Volume  Internal Energy  Enthalpy
C          m^3/kg            kJ/kg         kJ/kg
100        1.6958            2506.7        2676.2
125        1.8161            2544.8        2726.3
...
500        3.5650            3131.6        3488.1
```

5. Test the solution
   The MATLAB result matches the hand result. Now that we know the program works, we can create more extensive tables by changing the definition of `newT` from

   ```
   newT = [100:25:500]';
   ```

   to a vector with a smaller temperature increment—for example,

   ```
   newT = [100:1:500]';
   ```

PRACTICE EXERCISE 13.1

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

Create x and y vectors to represent the following data:

x	y
10	23
20	45
30	60
40	82
50	111
60	140
70	167
80	198
90	200
100	220

1. Plot the data on an x–y plot.
2. Use linear interpolation to approximate the value of y when $x = 15$.
3. Use cubic spline interpolation to approximate the value of y when $x = 15$.
4. Use linear interpolation to approximate the value of x when $y = 80$.
5. Use cubic spline interpolation to approximate the value of x when $y = 80$.
6. Use cubic spline interpolation to approximate y-values for x-values evenly spaced between 10 and 100 at intervals of 2.
7. Plot the original data on an x–y plot as data points not connected by a line. Also, plot the values calculated in Exercise 6.

13.1.3 Extrapolation

Although it is rarely a good idea, the `interp1` function can be used to **extrapolate** data by adding two new fields. Consider the sample values of x and y used earlier

```
x = 0:5;
y = [15, 10, 9, 6, 2, 0];
```

KEY IDEA
It is rarely a good idea to extrapolate data outside the measured range.

To extrapolate y past a value of $x = 5$, for example at 6, you need to specify the modeling type, such as linear or spline, and tell MATLAB to extrapolate.

```
y_new = interp1(x,y, 6, 'linear','extrap')
ans =
                    -2
```

Choosing a different modeling strategy can give very different results.

```
y_new=interp1(x,y,6,'spline','extrap')
ans =
                     3.0667
```

13.1.4 Multidimensional Interpolation

Imagine you have a set of data z that depends on two variables, x and y. For example, consider this table:

	$x = 1$	$x = 2$	$x = 3$	$x = 4$
$y = 2$	7	15	22	30
$y = 4$	54	109	164	218
$y = 6$	403	807	1,210	1,614

If you wanted to determine the value of z at $y = 3$ and $x = 1.5$, you would have to perform two interpolations. One approach would be to find the values of z at $y = 3$ and all the given x-values by using `interp1`, then perform a second interpolation in your new chart. First, define x, y, and z in MATLAB:

```
y = 2:2:6;
x = 1:4;
z = [ 7   15    22    30
     54   109   164   218
    403   807   1210  1614];
```

Now use `interp1` to find the values of z at $y = 3$ for all the x-values:

`new_z = interp1(y,z,3)` returns

```
new_z =
   30.50    62.00    93.00   124.00
```

Finally, since we have z-values at $y = 3$, `interp1` can be used to find z at $y = 3$ and $x = 1.5$:

```
new_z2 = interp1(x,new_z,1.5)
new_z2 =
    46.25
```

Although this approach works, performing the calculations in two steps is awkward. MATLAB includes a two-dimensional linear interpolation function, `interp2`, that can solve the problem in a single step:

```
interp2(x,y,z,1.5,3)
ans =
  46.2500
```

The first field in the `interp2` function must be a vector defining the value associated with each column (in this case, `x`), and the second field must be a vector defining the values associated with each row (in this case, `y`). The array `z` must have the same number of columns as the number of elements in `x`, and must have the same number of rows as the number of elements in `y`. The fourth and fifth fields correspond to the values of `x` and of `y` for which you would like to determine new `z`-values.

MATLAB also includes a function, `interp3`, for three-dimensional interpolation. Consult the help feature for the details on how to use this function and `interpn`, which allows you to perform *n*-dimensional interpolation. All these functions default to the linear interpolation technique, but will accept any of the other techniques listed in Table 13.1.

PRACTICE EXERCISE 13.2

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

Create y and z vectors to represent the following data, one for each value of x:

	$x = 15$	$x = 30$
$y = 10$	$z = 23$	33
20	45	55
30	60	70
40	82	92
50	111	121
60	140	150
70	167	177
80	198	208
90	200	210
100	220	230

1. Plot both sets of y–z data on the same plot. Add a legend identifying which value of x applies to each data set.
2. Use two-dimensional linear interpolation to approximate the value of z when $y = 15$ and $x = 20$.
3. Use two-dimensional cubic spline interpolation to approximate the value of z when $y = 15$ and $x = 20$.
4. Use linear interpolation to create a new subtable for $x = 20$ and $x = 25$ for all the y-values.

KEY IDEA
Curve fitting is a technique for modeling data with an equation.

13.2 CURVE FITTING

Although interpolation techniques could be used to find values of y between measured x-values, it would be more convenient to model experimental data as $y = f(x)$. Then it would be possible to calculate any value of y required. If the underlying relationship between x and y is understood, we may be able to determine an equation on the basis of those principles. For example, the ideal gas law is based on two underlying assumptions:

- All the molecules in a gas collide elastically.
- The molecules don't take up any room in their container.

Neither assumption is entirely accurate, so the ideal gas law works only when they are a good approximation of reality, but that is true for many situations, and the ideal gas law is extremely valuable. However, when real gases deviate from this simple relationship, there are two choices for how to model their behavior. Either try to understand the physics of the situation and adjust the equation accordingly, or take the data and model them empirically. Empirical equations are not related to any theory of why a behavior occurs; they just do a good job of predicting how a parameter changes in relationship to another parameter.

MATLAB has built-in curve-fitting functions that allow us to model data empirically. It's important to remind ourselves that these models are good only in the region where data exist. If we don't understand why a parameter such as y changes as it does with x, we can't predict whether a data-fitting equation will still work outside the range where we've collected data.

13.2.1 Linear Regression

The simplest way to model a set of data is as a straight line. Let's revisit the data from Section 13.1.1:

```
x = 0:5;
y = [15, 10, 9, 6, 2, 0];
```

If we plot the data in Figure 13.5, we can try to draw a straight line through the data points to get a rough model of the data's behavior. This process is sometimes called "eyeballing it"—meaning that no calculations were done, but it looks like a good fit.

Looking at the plot, several of the points appear to fall exactly on the line, but others are off by varying amounts. In order to compare the quality of the fit of this line to other possible estimates, we find the difference between the actual y-value and the value calculated from the estimate. This difference, the error between the actual and predicted, is called the *residual*.

KEY IDEA
Linear regression is a technique for obtaining the best fit straight line model.

The equation of the line in Figure 13.5 is determined by noticing that at $x = 0$, $y = 15$ and at $x = 5$, $y = 0$. Thus, the slope of the line is

$$\frac{\text{rise}}{\text{run}} = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{0 - 15}{5 - 0} = -3$$

The line crosses the y-axis at 15, so the equation of the line is

$$y = -3x + 15$$

The differences between the actual values and the calculated values are listed in Table 13.4.

The **linear regression** technique uses an approach called **least squares** fit to compare how well different equations model the behavior of the data. In this technique,

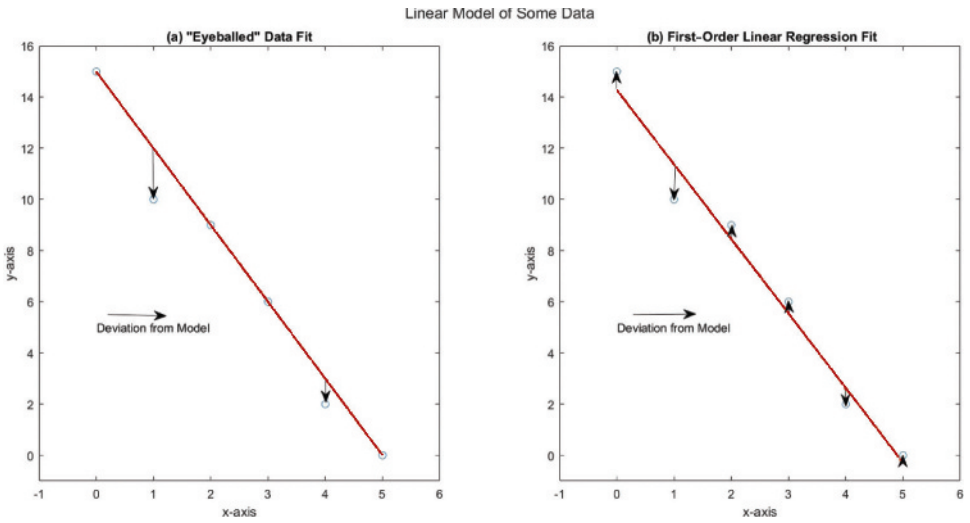


Figure 13.5
The data points were modeled as a straight line. (a) The easiest approach to linear modeling is to just "eyeball" a straight line through the data. Notice that in this approximation all the deviations from the model are in the same direction. (b) The linear regression approach to modeling a straight line minimizes the sum of the least squares to find the best fit. Notice that some of the deviations are positive and some are negative.

Table 13.4 Difference between Actual and Calculated Values

x	*y* (actual)	*y_calc* (calculated)	difference = *y* − *y_calc*
0	15	15	0
1	10	12	−2
2	9	9	0
3	6	6	0
4	2	3	−1
5	0	0	0

the differences between the actual and calculated values are squared and added together. This has the advantage that positive and negative deviations don't cancel each other out. MATLAB calculates this parameter for our data using the sum function. The calculated values of *y* are:

```
y_calc = -3*x +15;
```

and

```
sum_of_the_squares = sum((y-y_calc).^2)
```

which gives us

```
sum_of_the_squares =
  5
```

It's beyond the scope of this chapter to explain how the linear regression technique works, except to say that it compares different models and chooses the one in which the sum of the squares of the residuals is the smallest. Linear regression is accomplished in MATLAB with the `polyfit` function. Three fields are required by `polyfit`: a vector of *x*-values, a vector of *y*-values, and an integer indicating what order polynomial should be used to fit the data. Since a straight line is a first-order polynomial, the number 1 is entered into the `polyfit` function:

```
polyfit(x,y,1)
ans =
  -2.9143 14.2857
```

The results are the coefficients corresponding to the best-fit first-order polynomial equation:

$$y = -2.9143x + 14.2857$$

Is this really a better fit than the "eyeballed" model? We can calculate the sum of the squares to find out:

```
best_y = -2.9143*x+14.2857;
new_sum = sum((y-best_y).^2)
new_sum =
  3.3714
```

Since the result of the sum-of-the-squares calculation is indeed less than the value found for the "eyeballed" line, we can conclude that MATLAB found a better fit to the data. The data and the best-fit line determined by linear regression are plotted on the same graph (see Figure 13.5) to try to get a visual sense of whether the line fits the data well:

```
plot(x,y,'o',x,best_y)
```

13.2.2 Linear Regression Through Zero

Sometimes, based on the physics of a system, we know that a model should pass through zero. For example, consider the force applied to a kite by blowing wind. If the wind velocity is zero, the applied force should be zero. The `polyfit` function does not have the option of forcing the fit through the zero point, so we need an alternative strategy.

The left division operator, \, was introduced in a previous chapter. It employs Gaussian elimination to solve systems of linear equations. However, if a system of equations is overspecified, that is, if there are more equations than variables, the left-division operator uses a least-squares fit strategy similar to that used by `polyfit`. Consider the following arbitrary data. In this case, the wind velocity is the independent variable, x, and the resulting force is the dependent variable, y.

Wind Velocity, x	Resulting Force, y
0	0
10	24
20	38
30	64
40	82
50	90

Using `polyfit`, the result is a straight line that does not pass through zero.

```
a = polyfit(x,y,1)
a =
   1.8571   3.2381
```

This corresponds to the equation

$$y = a(1)*x + a(2)$$
$$y = 1.8571*x + 3.2381$$

KEY IDEA
Use left division to model straight lines with an intercept at the origin.

We can force the model through zero with left-division

```
a = x'\y'
a =
    1.9455
```

which corresponds to the equation

$$y = a(1)*x \quad \text{or}$$
$$y = 1.9455*x$$

The results are plotted in Figure 13.6. (Note that this technique requires the x and y vectors be expressed as columns.) Although this model is not "as good" as the linear model created with `polyfit` based on the sum-of-the-squares, it is a better model based on what we know about the physical system.

The left-division operator is a powerful tool and can be used to model many different linear systems. Consult the MATLAB help feature for more details.

13.2.3 Polynomial Regression

Of course, straight lines are not the only equations that could be analyzed with the regression technique. Taylor's theorem tells us that any smooth function can be approximated as a polynomial, so a common approach is to fit data with a higher-order polynomial of the form

$$y = a_1x^n + a_2x^{n-1} + a_3x^{n-2} + \cdots + a_nx + a_{n+1}$$

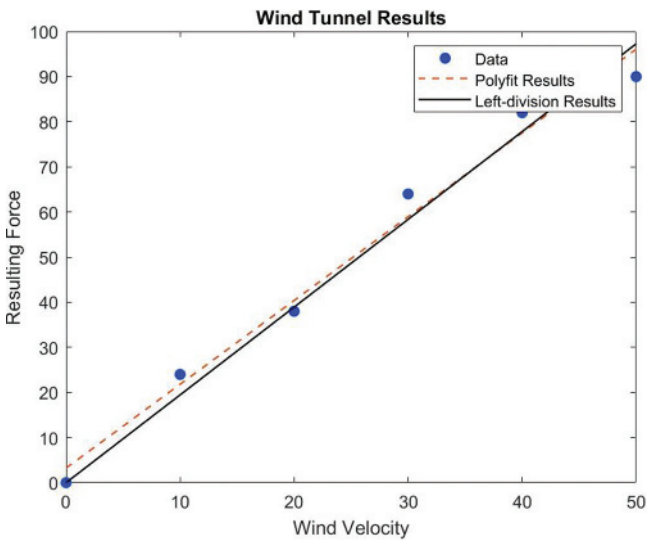
Polynomial regression is used to get the best fit by minimizing the sum of the squares of the deviations of the calculated values from the data. The `polyfit` function allows us to do this easily in MATLAB. Returning to our sample data

```
x = 0:5;
y = [15, 10, 9, 6, 2, 0];
```

we can fit second- and third-order equations with the commands

```
a = polyfit(x,y,2)
a =
  0.0536   -3.1821   14.4643
```

Figure 13.6
Left-division can be used to force a first-order data fit through zero.



and

```
a = polyfit(x,y,3)
a =
  -0.0648    0.5397    -4.0701    14.6587
```

which correspond to the following equations

$$y_2 = 0.0536x^2 - 3.1821x + 14.4643$$
$$y_3 = -0.0648x^3 + 0.5397x^2 - 4.0701x + 14.6587$$

We can find the sum of the squares to determine whether these models fit the data better:

```
y2 = 0.0536*x.^2-3.182*x + 14.4643;
sum((y2-y).^2)
ans =
  3.2643
y3 = -0.0648*x.^3+0.5398*x.^2-4.0701*x + 14.6587;
sum((y3-y).^2)
ans =
 2.9921
```

The more terms we add to our equation, the "better" the fit, at least in the sense that the distance between the measured and predicted data points decreases.

In order to plot the curves defined by these new equations, we'll need more than the six data points used in the linear model. MATLAB creates plots by connecting calculated points with straight lines, so if we want a smooth curve, we'll need more points.

```
smooth_x = 0:0.2:5;
smooth_y2 = 0.0536*smooth_x.^2-3.182*smooth_x + 14.4643;
t = tiledlayout(1,2);
nexttile
plot(x,y,'o',smooth_x,smooth_y2)
xlabel("x-axis"),ylabel("y-axis")
title("(a) Second-Order Model")
smooth_y3 = -0.0648*smooth_x.^3+0.5398*smooth_x.^2-...
4.0701*smooth_x + 14.6587;
nexttile
plot(x,y,'o',smooth_x,smooth_y3)
xlabel("x-axis"),ylabel("y-axis")
title("(b) Third-Order Model")
title(t,"Higher Order Polynomial Fits")
```

The results are shown in Figure 13.7. Notice the slight curvature in each model. Although mathematically these models fit the data better, they may not be as good a representation of reality as the straight line. As an engineer or scientist, you'll need to evaluate any modeling you do. You'll need to consider what you know about the physics of the process you're modeling, and the accuracy and reproducibility of your measurements.

KEY IDEA
Modeling of data should be based not only on the data collected but also on a physical understanding of the process.

13.2.4 The `Polyval` Function

The `polyfit` function returns the coefficients of a polynomial that best fits the data, at least on the basis of a regression criterion. In the previous section, those

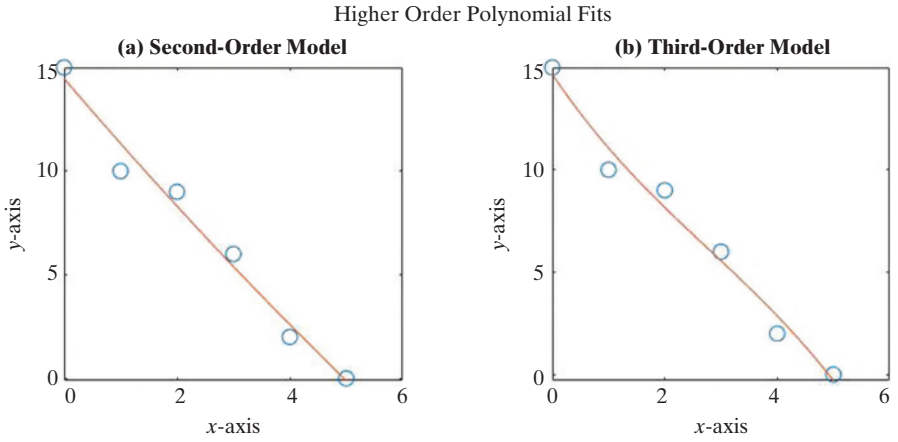


Figure 13.7
Second- and third-order polynomial fits.

coefficients were entered into a MATLAB expression for the corresponding polynomial and used to calculate new values of *y*. The `polyval` function performs the same job without the need to reenter the coefficients.

The `polyval` function requires two inputs. The first is a coefficient array, such as that created by `polyfit`. The second is an array of *x*-values for which new *y*-values are desired. For example,

```
coef = polyfit(x,y,1)
y_first_order_fit = polyval(coef,x)
```

These two lines of code could be shortened to one line by nesting functions:

```
y_first_order_fit = polyval(polyfit(x,y,1),x)
```

We can use our new understanding of the `polyfit` and `polyval` functions to write a program to calculate and plot the fourth- and fifth-order fits for the data from Section 13.1.1:

```
y4 = polyval(polyfit(x,y,4),smooth_x);
y5 = polyval(polyfit(x,y,5),smooth_x);
nexttile
plot(x,y,'o',smooth_x,y4)
axis([0,6,-5,15])
xlabel("x-axis"),ylabel("y-axis")
title("(a) Fourth-Order Model")
nexttile
plot(x,y,'o',smooth_x,y5)
axis([0,6,-5,15])
xlabel("x-axis"),ylabel("y-axis")
title("(b) Fifth-Order Model")
title(t,"Higher Order Polynomial Fits")
```

Figure 13.8 gives the results of our plot.

As expected, the higher-order fits match the data better and better. The fifth-order model matches exactly because there were only six data points.

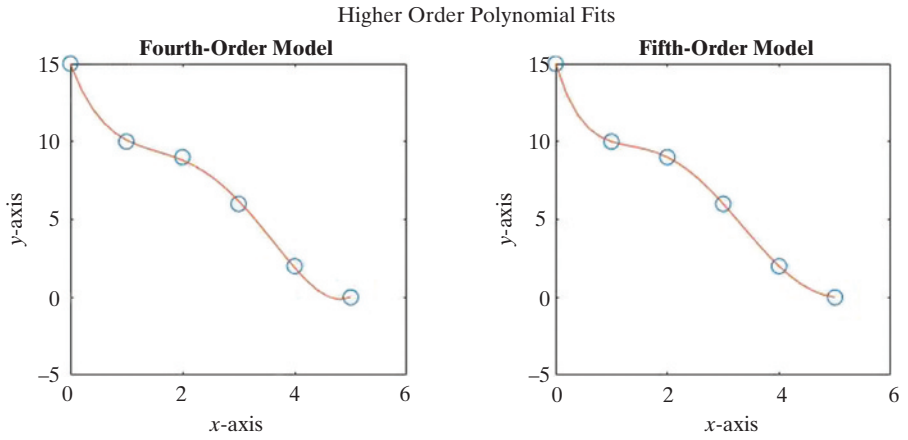


Figure 13.8
Fourth- and fifth-order model of six data points. The fifth-order model matches exactly because there are only 6 data points.

HINT

You could create all four of the graphs shown in Figures 13.7 and 13.8 by using a `for` loop that makes use of `tiles` (formerly called subplots) and the `sprintf` function.

```
x = 0:5;
y = [15, 10, 9, 6, 2, 0];
smooth_x = 0:0.2:5;
t=tiledlayout(2,2);
for k = 1:4
nexttile
plot(x,y,'o',smooth_x,polyval(polyfit(x,y,k+1),smooth_x))
axis([0,6,-5,15])
a = sprintf('Polynomial plot of order %1.0f \n',k+1);
title(a)
xlabel("x-axis"),ylabel("y-axis")
end
title(t,"Higher Order Polynomial Fit")
```

PRACTICE EXERCISE 13.3

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

Create x and y vectors to represent the following data:

$z = 15$		$z = 30$	
x	y	x	y
10	23	10	33
20	45	20	55
30	60	30	70
40	82	40	92
50	111	50	121
60	140	60	150
70	167	70	177

$z = 15$		$z = 30$	
80	198	80	208
90	200	90	210
100	220	100	230

1. Use the `polyfit` function to fit the data for $z = 15$ to a first-order polynomial.
2. Create a vector of new x values from 10 to 100 in intervals of 2. Use your new vector in the `polyval` function together with the coefficient values found in Exercise 1 to create a new y vector.
3. Plot the original data as circles without a connecting line and the calculated data as a solid line on the same graph. How well do you think your model fits the data?
4. Repeat Exercises 1–3 for the x and y data corresponding to $z = 30$.

EXAMPLE 13.3

WATER IN A CULVERT

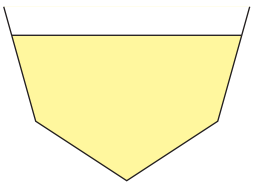


Figure 13.9
Culverts do not necessarily have a uniform cross section.

Determining how much water will flow through a culvert is not as easy as it might first seem. The channel could have a nonuniform shape (see Figure 13.9), obstructions might influence the flow, friction is important, and so on. A numerical approach allows us to fold all those concerns into a model of how the water actually behaves. Consider the following data collected from an actual culvert.

Height (ft)	Flow (ft^3/s)
0	0
1.7	2.6
1.95	3.6
2.60	4.03
2.92	6.45
4.04	11.22
5.24	30.61

Compute a best-fit linear, **quadratic**, and **cubic equation** for the data, and plot them on the same graph. Which model best represents the data? (Linear is first order, quadratic is second order, and cubic is third order.) Also include a linear model forced through 0.

1. State the problem
   Perform a polynomial regression on the data, plot the results, and determine which order best represents the data.

(*continued*)

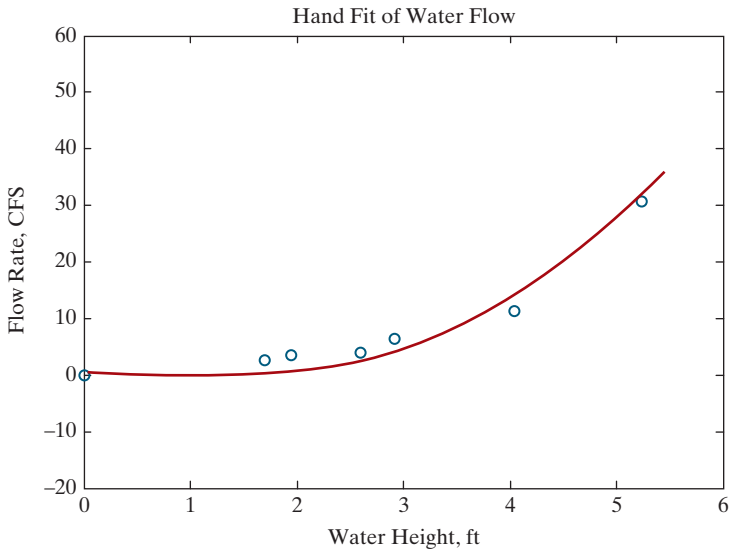


Figure 13.10
Hand fit of water flow data.

2. Describe the input and output

Input Height and flow data
Output Plot of the results

3. Develop a hand example

Draw an approximation of the curve by hand. Be sure to start at zero, since, if the height of water in the culvert is zero, no water should be flowing (see Figure 13.10).

4. Develop a MATLAB solution

Create a MATLAB program, then run it in the command environment:

```
%% 13.3 Example - Water in a Culvert
height = [0, 1.7, 1.95, 2.6, 2.92, 4.04, 5.24];
flow = [0, 2.6, 3.6, 4.03, 6.45, 11.22, 30.61];
new_height=0:0.5:6;
newf0=height'\flow'*new_height;
newf1=polyval(polyfit(height,flow,1),new_height);
newf2=polyval(polyfit(height,flow,2),new_height);
newf3=polyval(polyfit(height,flow,3),new_height);
plot(height,flow,'o',new_height,newf0,new_height,newf1, ...
new_height,newf2,new_height,newf3)
title("Fit of Water Flow")
xlabel("Water Height, ft")
ylabel("Flow Rate, CFS")
legend("Data","Linear Fit Through 0","Linear Fit", ...
"Quadratic Fit","Cubic Fit")
```

The MATLAB code generates the plot shown in Figure 13.11.

5. Test the solution

The question of which line best represents the data is difficult to answer. The higher-order polynomial approximation will follow the data points better, but it doesn't necessarily represent reality better.

The linear fit predicts that the water flow rate will be approximately −5 CFS at a height of zero, which doesn't match reality. Even the linear fit that

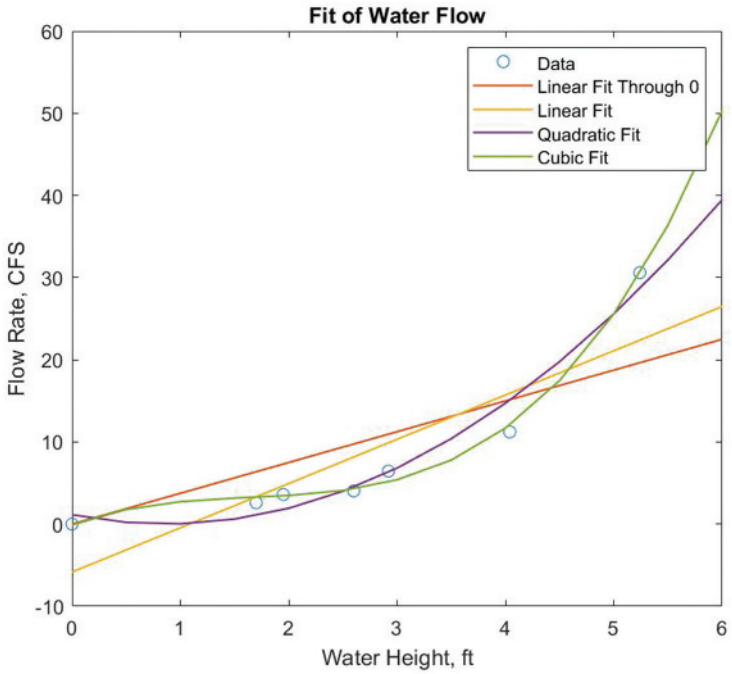


Figure 13.11
Different curve-fitting approaches.

passes through 0 is inadequate, because it doesn't capture the shape of the data. The quadratic fit goes back up after a minimum at a height of approximately 1.5 m—again a result inconsistent with reality. The cubic (third-order) fit follows the points the best and is probably the best polynomial fit. We should also compare the MATLAB solution with the hand solution. The third-order (cubic) polynomial fit approximately matches the hand solution.

EXAMPLE 13.4

HEAT CAPACITY OF A GAS

The amount of energy necessary to warm a gas 1°K (called the *heat capacity* of the gas) depends not only on the gas, but on its temperature as well. This relationship is commonly modeled with polynomials. For example, consider the data for carbon dioxide in Table 13.5.

Use MATLAB to model these data as a polynomial. Compare the results with those obtained from the model published in B. G. Kyle, *Chemical and Process Thermodynamics* (Upper Saddle River, NJ: Prentice Hall PTR, 1999), namely

$$C_p = 1.698 \times 10^{-10} T^3 - 7.957 \times 10^{-7} T^2 + 1.359 \times 10^{-3} T + 5.059 \times 10^{-1}$$

1. State the problem
   Create an empirical mathematical model that describes heat capacity as a function of temperature. Compare the results with those obtained from published models.
2. Describe the input and output

   Input Use the table of temperature and heat-capacity data provided.

   Output Find the coefficients of a polynomial that describes the data. Plot the results.

(*continued*)

Table 13.5 Heat Capacity of Carbon Dioxide

Temperature, T (K)	Heat Capacity, C_p (kJ/(kg K)
250	0.791
300	0.846
350	0.895
400	0.939
450	0.978
500	1.014
550	1.046
600	1.075
650	1.102
700	1.126
750	1.148
800	1.169
900	1.204
1,000	1.234
1,500	1.328

Source: *Tables of Thermal Properties of Gases*, NBS Circular 564, 1955.

3. Develop a hand example
   By plotting the data (see Figure 13.12) we can see that a straight-line fit (first-order polynomial) is not a good approximation of the data. We'll need to evaluate several different models—for example, from first to fourth order.

4. Develop a MATLAB solution

```
%% Example 13.4 Heat Capacity of a Gas
clear, clc
% Define the measured data
T=[250:50:800,900,1000,1500];
Cp=[0.791, 0.846, 0.895, 0.939, 0.978, 1.014, 1.046, 1.075,
    1.102,...
1.126, 1.148, 1.169, 1.204, 1.234, 1.328];
```

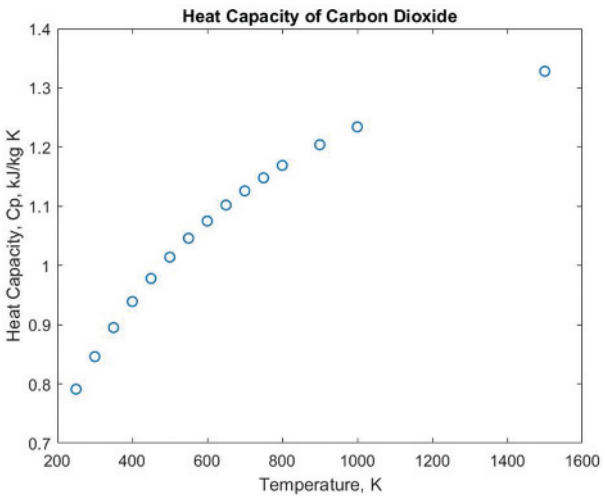


Figure 13.12
Heat capacity of carbon dioxide as a function of temperature.

```
% Define a finer array of temperatures
new_T=250:10:1500;

% Calculate new heat capacity values, using four different
% polynomial models
Cp1=polyval(polyfit(T,Cp,1),new_T);
Cp2=polyval(polyfit(T,Cp,2),new_T);
Cp3=polyval(polyfit(T,Cp,3),new_T);
Cp4=polyval(polyfit(T,Cp,4),new_T);

% Plot the results
t=tiledlayout(2,2);
nexttile
plot(T,Cp,'o',new_T,Cp1)
ylabel("Heat Capacity, kJ/kg K")
axis([0,1700,0.6,1.6])
title("First-Order Model")
nexttile
plot(T,Cp,'o',new_T,Cp2)
axis([0,1700,0.6,1.6])
title("Second-Order Model")
nexttile
plot(T,Cp,'o',new_T,Cp3)
xlabel("Temperature, K"), ylabel("Heat Capacity, kJ/kg K")
axis([0,1700,0.6,1.6])
title("Third-Order Model")
nexttile
plot(T,Cp,'o',new_T,Cp4)
xlabel("Temperature, K")
axis([0,1700,0.6,1.6])
title("Fourth-Order Model")
title(t,"Heat Capacity of Carbon Dioxide")
```

By looking at the graphs shown in Figure 13.13, we can see that a second- or third-order model adequately describes the behavior in this temperature region. If we decide to use a third-order polynomial model, the coefficients determined with `polyfit` are:

```
polyfit(T,Cp,3)
ans =
2.7372e-010 -1.0631e-006 1.5521e-003 4.6837e-001
```

The results correspond to the equation

$$C_p = 2.7372 \times 10^{-10}T^3 - 1.0631 \times 10^{-6}T^2 + 1.5521 \times 10^{-3}T + 4.6837 \times 10^{-1}$$

5. Test the solution

Comparing our result with that reported, we see that they are close, but not exact:

$$C_p = 2.737 \times 10^{-10}T^3 - 10.63 \times 10^{-7}T^2 + 1.552 \times 10^{-3}T + 4.683 \times 10^{-1}$$

(our fit)

$$C_p = 1.698 \times 10^{-10}T^3 - 7.957 \times 10^{-7}T^2 + 1.359 \times 10^{-3}T + 5.059 \times 10^{-1}$$

(literature)

(*continued*)

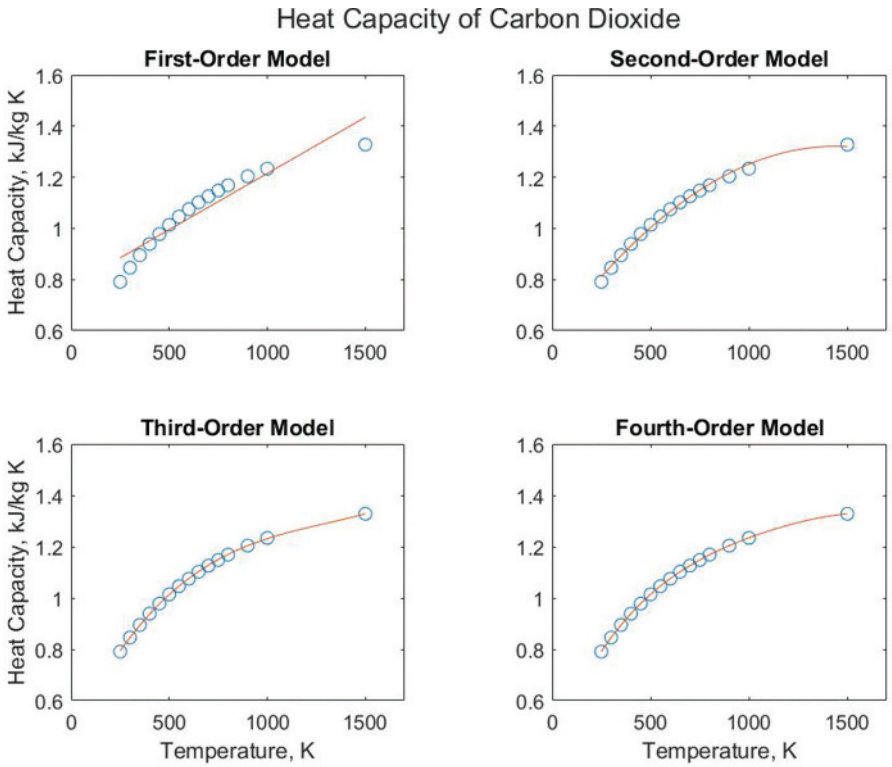


Figure 13.13
A comparison of different polynomials used to model the heat-capacity data of carbon dioxide.

This is not too surprising, since we modeled a limited number of data points. The models reported in the literature use more data, and are therefore probably more accurate.

13.3 USING THE INTERACTIVE FITTING TOOLS

MATLAB includes interactive plotting tools that allow annotation of plots without using the command window. Also included are basic curve fitting, more complicated curve fitting, and statistical tools.

13.3.1 Basic Fitting Tools

To access the basic fitting tools, first create a figure:

```
x = 0:5;
y = [0,20,60,68,77,110]
plot(x,y,'o')
axis([-1,7,-20,120])
```

These commands produce a graph (see Figure 13.14) with some sample data.

To activate the curve-fitting tools, select *Tools* → *Basic Fitting* from the menu bar in the figure. The basic fitting window opens in a new window. By checking *linear*, *cubic*, and *show equations* for each of the options (see Figure 13.16), the plot shown in Figure 13.15a was generated.

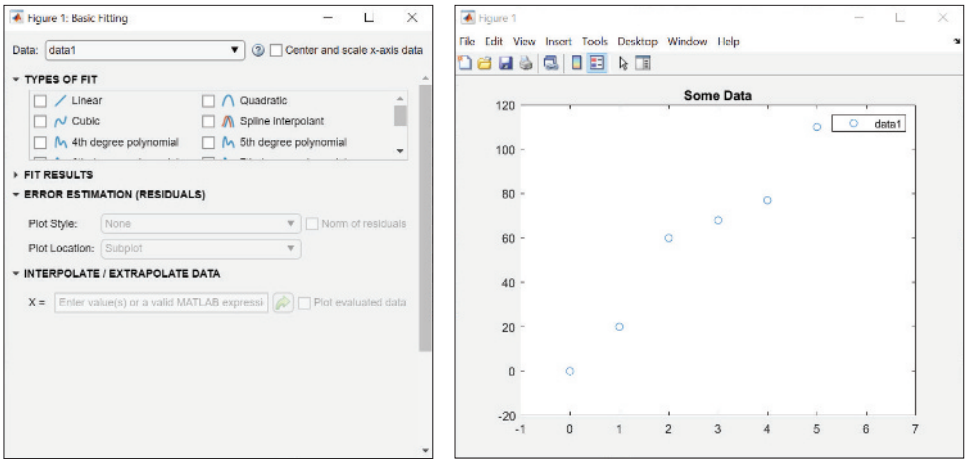


Figure 13.14
Interactive figure controls (a) The basic fitting window opens when Tools is selected from the figure window toolstrip. (b) The figure window includes tools to modify the appearance of the graph and to perform statistical calculations on the data used to create the figure, including curve fitting.

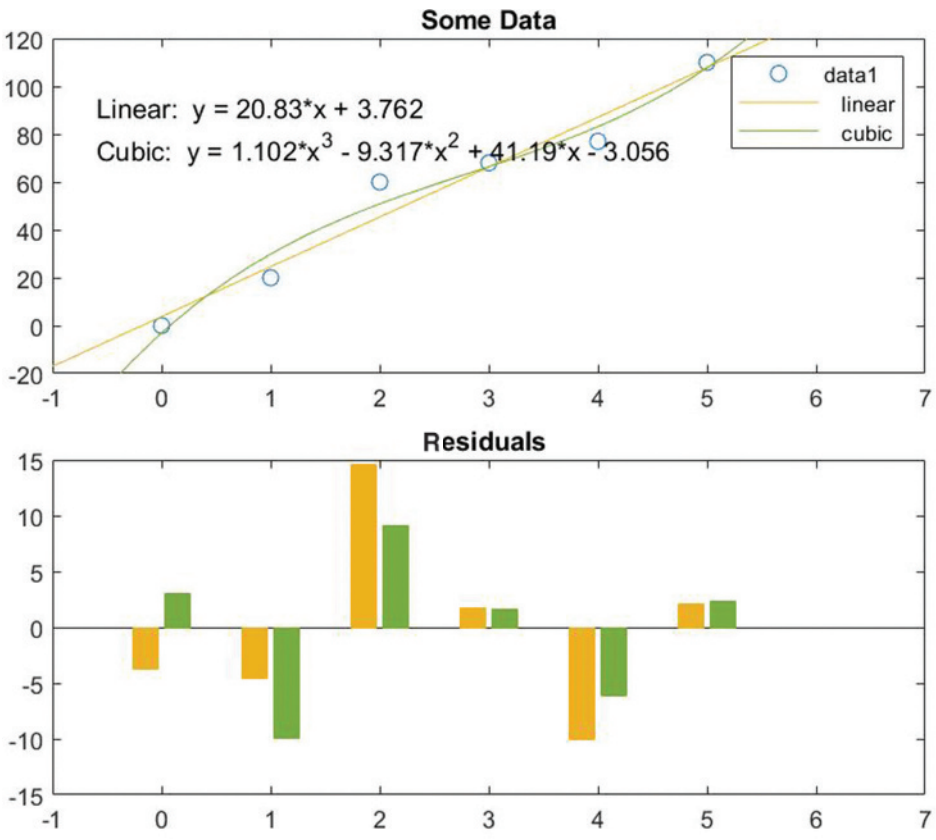


Figure 13.15
The fitting tool generates multiple types of information. (a) The upper figure includes the formula of the fitted curve. (b) The lower figure shows the residuals, which are the difference between the actual and calculated data points.

KEY IDEA
Residuals are used in regression strategies to evaluate curve fitting techniques.

Selecting the *Error Estimation* ***(Residuals)*** plot style of *bar* and a plot location of *Subplot* generates a second plot, showing how far each data point is from the calculated line, as shown in Figure 13.15b.

In the lower right-hand corner of the basic fitting window is an *arrow button*. Selecting that button offers the option of saving results to the workspace.

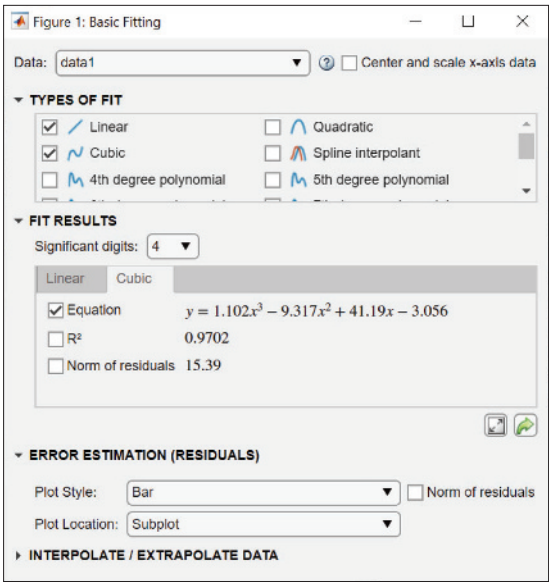


Figure 13.16
The basic fitting tool allows the user to select multiple fitting techniques, to display the results on the figure, to interpolate and extrapolate data, and to send information to the workspace.

Finally, you can select x-values and calculate y-values based on the equation displayed in the center panel using the *Interpolate/Extrapolate Data* option in the fitting window.

In addition to the basic fitting window, you can access the data statistics window (see Figure 13.17) from the figure menu bar. Select *Tools* → *Data Statistics* from the figure window. The data statistics window allows you to calculate statistical functions such as the mean and standard deviation interactively, based on the data in the figure, and allows you to save the results to the workspace.

13.3.2 Curve-Fitting Toolbox

In addition to the basic fitting utility, MATLAB contains toolboxes to help you perform specialized statistical and data-fitting operations. In particular, the curve-fitting toolbox provides an app that allows you to fit curves with more than just polynomials. Although the curve-fitting toolbox is sold separately it comes standard with the student edition of the software, so it is widely available.

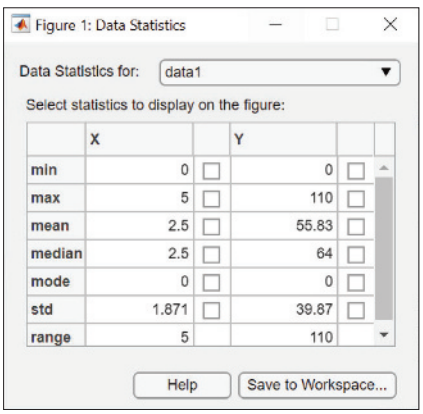


Figure 13.17
Data statistics window.

13.4 DIFFERENCES AND NUMERICAL DIFFERENTIATION

13.4.1 The `diff` Function

The **derivative** of the function $y = f(x)$ is a measure of how y changes with x. If you can define an equation that relates x and y, you can use the functions contained in the symbolic toolbox to find an equation for the derivative. However, if all you have are data, you can approximate the derivative by dividing the change in y by the change in x:

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x} \approx \frac{y_2 - y_1}{x_2 - x_1}$$

KEY IDEA
The `diff` function is used both with symbolic expressions, where it finds the derivative, and with numeric arrays.

If we plot the data from Section 13.1 that we've used throughout the chapter,

```
x = 0:5;
y = [15, 10, 9, 6, 2, 0];
```

this approximation of the derivative corresponds to the slope of each of the line segments used to connect the data, as shown in Figure 13.18.

KEY IDEA
The `diff` function is an example of an overloaded function. It has two separate definitions. The MATLAB software decides which version of the function to use, based on the input arguments.

If, for example, these data describe the measured temperature of a reaction chamber at different points in time, the slopes denote the cooling rate during each time segment. MATLAB has a built-in function called `diff` that will find the difference between element values in a vector and that can be used to calculate the slope of ordered pairs of data. (The `diff` function is an example of an "overloaded" function. MATLAB contains a version of `diff` used for symbolic algebra calculations, and a version that uses discrete data points. The software decides which version is appropriate based on the input you provide.)

For example, to find the change in x-values, type

```
delta_x = diff(x)
```

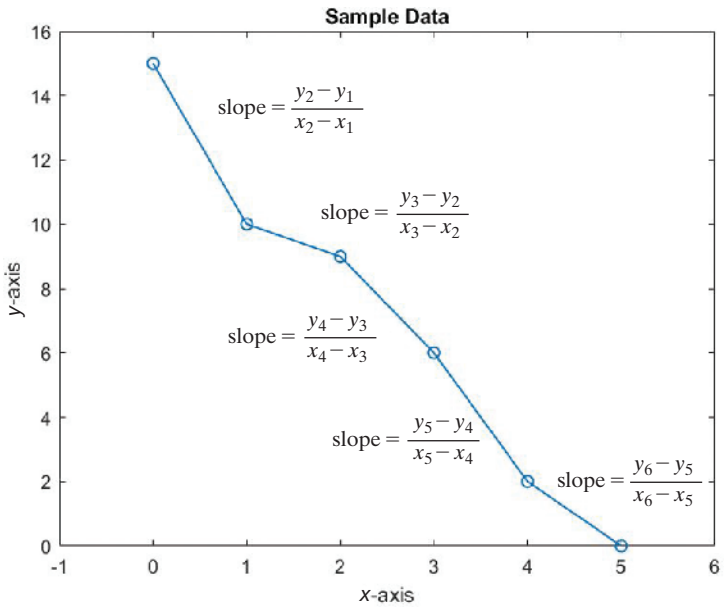


Figure 13.18
The derivative of a data set can be approximated by finding the slope of a straight line connecting each data point.

which, because the *x*-values are evenly spaced, returns

```
delta_x =
    1    1    1    1    1
```

Similarly, the difference in y-values is

```
delta_y = diff(y)
delta_y =
    -5 -1 -3 -4 -2
```

To find the slope, divide `delta_y` by `delta_x`:

```
slope = delta_y./delta_x
slope =
    -5 -1 -3 -4 -2
```

or

```
slope = diff(y)./diff(x)
slope =
    -5 -1 -3 -4 -2
```

Notice that the vector returned when you use the `diff` function is one element shorter than the input vector, because you are calculating differences. When you use the `diff` function to help calculate slopes, you are calculating the slope between values of *x*, not at a particular value. If you want to plot these slopes against *x*, probably the best approach is to create a bar graph, since the rates of change are not continuous. The *x*-values were adjusted to the average for each line segment:

```
x = x(:,1:5)+diff(x)/2;
bar(x,slope)
```

The resulting bar graph is shown in Figure 13.19.

The `diff` function can also be used to approximate a derivative numerically if you know the relationship between *x* and *y*. For example, if

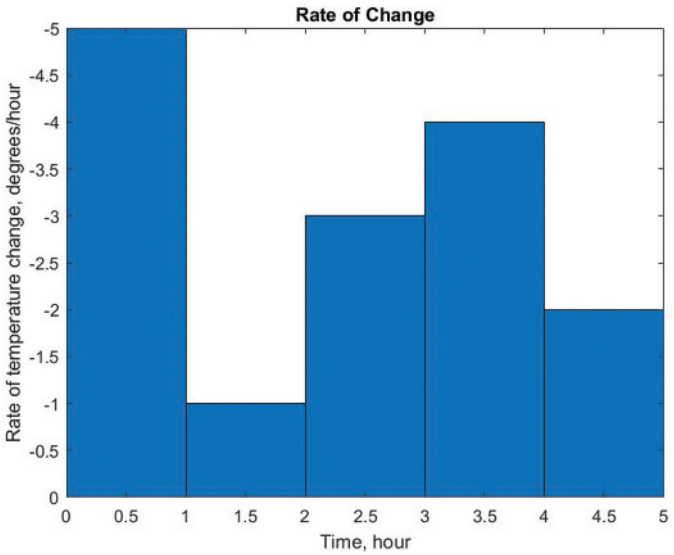


Figure 13.19
The calculated slopes are discontinuous if they are based on data. The appearance of this graph was adjusted with the interactive plotting tools.

$$y = x^2$$

a set of ordered pairs could be created for any number of x-values. The more values of x and y, the smoother the plot will be. Here are two sets of x and y vectors that were used to create the graph in Figure 13.20(a):

```
x = -2:2
y = x.^2;
big_x = -2:0.1:2;
big_y = big_x.^2;
plot(big_x,big_y,x,y,'-o')
```

Both lines in the graph are created by connecting the specified points with straight lines; however, the `big_x` and `big_y` values are so close that the graph looks like a continuous curve. The slope of the x–y plot was calculated with the `diff` function and plotted in Figure 13.20(b):

```
slope5 = diff(y)./diff(x);
x5 = x(:,1:4)+diff(x)./2;
% These values were based on a 5-point model
bar(x5,slope5)
```

The bar graph was modified slightly with the use of the interactive plotting tools to give the representation shown in Figure 13.20(b). A smoother representation, though still discontinuous, is accomplished by using more points:

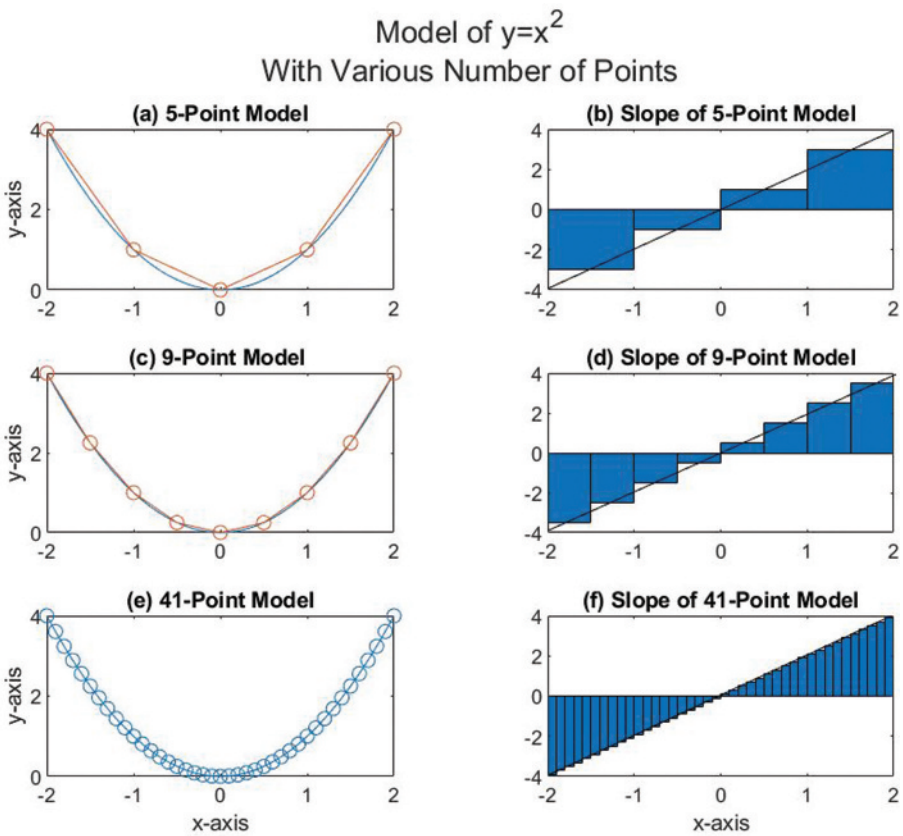


Figure 13.20
The slope of a function is approximated more accurately when more points are used to model the function.

```
x = -2:0.5:2;
y = x.^2;
plot(big_x,big_y,x,y,'-o')
slope9 = diff(y)./diff(x);
x9 = x(:,1:8)+diff(x)./2;
% These values were based on a 9-point model
bar(x9,slope9)
```

These results are shown in Figures 13.20(c,d). Even more points results in an almost smooth representation of the slope as a funciton of x, as calculated here:

```
plot(big_x,big_y,'-o')
slope41 = diff(big_y)./diff(big_x);
x41 = big_x(:,1:40)+diff(big_x)./2;    % 41-point model
bar(x41,slope41)
```

The results are shown in Figures 13.20(e,f).

13.4.2 Forward, Backward, and Central Difference Techniques

What if you want to approximate the derivative at a point, instead of over a range, as discussed earlier? One approach is to use the slope between adjacent points as the approximation of the derivative at a single value of x.

$$\left(\frac{dy}{dx}\right)_i = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

We can accomplish this by using the difference function

```
fd = diff(y)./diff(x)
```

and assigning the result as the derivative at the first point in the range. This is called a **forward difference**, since we are approximating the derivative by looking forward in the array to the next set of x and y values.

Take for example the sine function, whose analytical derivative is cosine. We can compare the forward difference derivative approximation to the analytical solution with the following code. First, create an array of values for the independent variable, x, and for the dependent variable, y.

```
x = linspace(0,pi/2,10)
y = sin(x)
```

We know from calculus that the derivative of $\sin(x)$ is $\cos(x)$, which is expressed as

$$\frac{dy}{dx} = \cos(x)$$

Thus, to find the derivative analytically in MATLAB we use the code

```
dydx=cos(x)
```

To approximate the derivative for the first nine values in the x array (which has a total of 10 values) using the forward difference approach

```
fd=diff(y)./diff(x)
```

It isn't possible to find an approximation for the derivative at the last point in the x array using this technique, so we use `NaN` (not a number) as a place holder. Notice that

in order to make the code more general, we've defined the last element number using the `length` function, which in this case returns a value of 10.

```
fd(length(x))=NaN;
```

To find the percentage error between this approximation and the analytical value, we'll use the following equation:

$$\%\ \text{error} = \frac{\text{actual_value} - \text{approximation}}{\text{actual_value}} \times 100$$

which corresponds to the following code:

```
error_percentage=(dydx-fd)./dydx*100
```

Finally, to create an output display so we can evaluate the results, the following code can be used:

```
a=["X","cos(x)","Forward Approximation","% Error"];
table(x',dydx',fd',error_percentage','VariableNames',a)
```

The resulting table is informative. There are significant errors in the approximation as the analytical result approaches 0, but the absolute error is fairly small.

Forward Difference Approximation of the Derivative of *Sin(x)*

x	dy/dx	dy/dx	%error
	cos(x)	forward approximation	(actual – est)/ actual *100
0.0000	1.0000	0.9949	0.5069
0.1745	0.9848	0.9647	2.0418
0.3491	0.9397	0.9052	3.6751
0.5236	0.8660	0.8181	5.5325
0.6981	0.7660	0.7062	7.8109
0.8727	0.6428	0.5728	10.8806
1.0472	0.5000	0.4221	15.5836
1.2217	0.3420	0.2585	24.4224
1.3963	0.1736	0.0870	49.8727
1.5708	0.0000	NaN	NaN

Notice that there is no approximation of the derivative for the last value of x, so (in the code) a value of `NaN` (not a number) was added. We repeated the calculations with 20 values, and plotted the results for both 10 values and 20 values in Figure 13.21.

We can do a better job of approximating the derivative by specifying more values of x (effectively making the points closer together).

The **backward difference** is very similar. Instead of assigning the approximation of the derivative to the first value in a range, it is assigned to the last value.

$$\left(\frac{dy}{dx}\right)_i = \frac{y_i - y_{i-1}}{x_i - x_{i-1}}$$

To solve this problem in MATLAB, we can use the `diff` function again. Similarly to the first example, a value of `NaN` was added to the `bd` array (backward difference), but this time it is the first value, not the last.

```
%% Backward difference
x=linspace(0,pi/2,10);
```

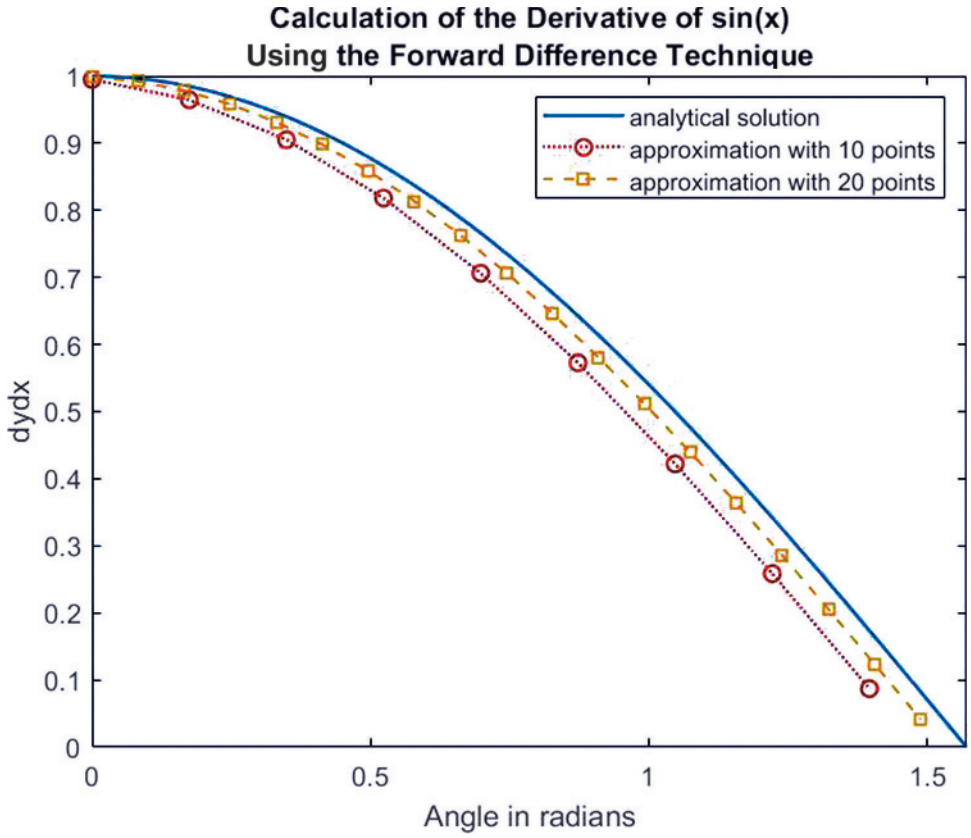


Figure 13.21
A comparison of the derivative approximation of *sin(x)*, based on the number of points used.

```
y=sin(x);
dydx=cos(x);
bd=diff(y)./diff(x);
bd=[NaN,bd];
error_percentageb = (dydx-bd)./dydx*100;
b=["X","cos(x)","Backward Approximation","% Error"];
table(x',dydx',bd',error_percentageb','VariableNames',a)
```

The resulting table is

Backward Difference Approximation of the Derivative of *Sin(x)*

x	dy/dx	dy/dx	%error
	cos(x)	backward approximation	(actual – est)/actual *100
0.0000	1.0000	NaN	NaN
0.1745	0.9848	0.9949	−1.0279
0.3491	0.9397	0.9647	−2.6613
0.5236	0.8660	0.9052	−4.5186
0.6981	0.7660	0.8181	−6.7970
0.8727	0.6428	0.7062	−9.8667
1.0472	0.5000	0.5728	−14.5697
1.2217	0.3420	0.4221	−23.4085

x	dy/dx	dy/dx	%error
1.3963	0.1736	0.2585	−48.8588
1.5708	0.0000	0.0870	−142155539756746180.0000

The absolute value of the error resulting from a forward difference technique versus a backward difference technique is very similar. (The large error for the final table entry in the backward difference table is due to the division by 0.) We can get closer by using a **central difference** technique that looks both forward and backward, and therefore is centered on the actual point of interest. The approximation is

$$\left(\frac{dy}{dx}\right)_i = \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}}$$

One downside of using this technique is that it won't work for either the first or last value in the array.

MATLAB includes a function, `gradient`, which approximates the derivative using a forward difference technique for the first point in an array, the backward difference for the last point in an array, and a centered difference for the remainder of the points. It requires two inputs, the *y* and *x* array

```
g = gradient(y,x)
```

and returns the derivative approximation. If you don't enter an *x* array, the program assumes the points are evenly spaced with a step size of 1. The results for all three approaches are shown in Figure 13.22.

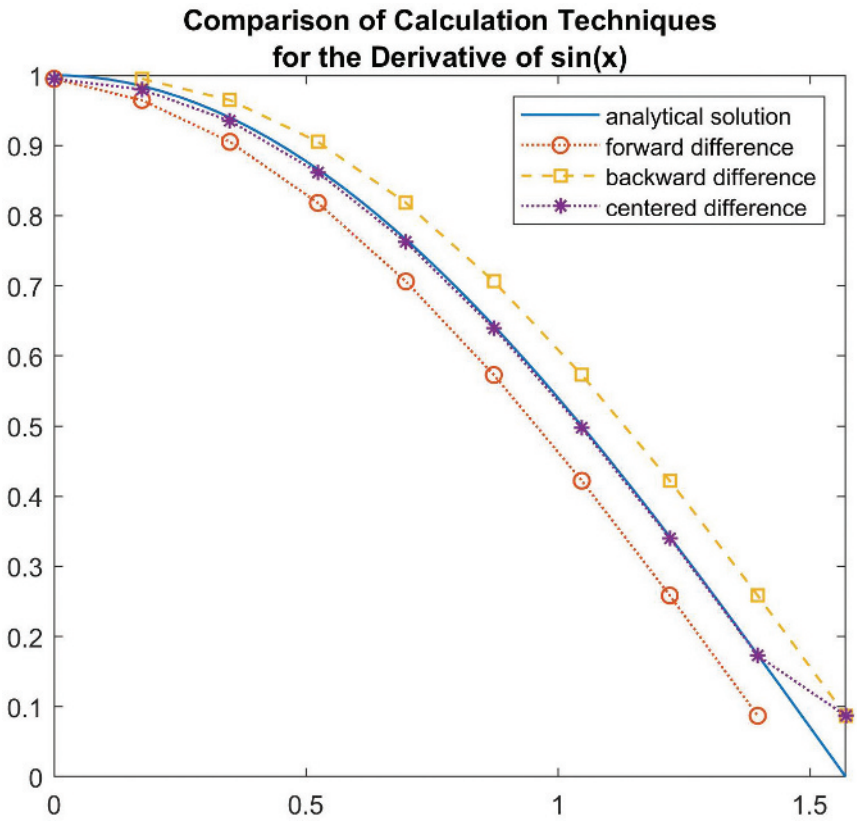


Figure 13.22
A superior approximation of the derivative is obtained using the centered difference approach, implemented in the `gradient` function.

The `gradient` function can also be used to approximate partial derivatives when used with two-dimensional arrays. To demonstrate the use of `gradient`, we'll need to create a two-dimensional array.

```
X = -2:.1:2;
Y = -2:.1:2;
[x,y]=meshgrid(X,Y);
z = 3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
- 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
- 1/3*exp(-(x+1).^2 - y.^2);
```

This is the equation for the `peaks` function, which is built into MATLAB for demonstration purposes. The behavior of this function can be visualized by creating a surface plot (see Figure 13.23(a)).

```
surf(x,y,z)
```

The partial derivative of z with respect to x, $\partial z/\partial x$, is the slope of this figure in the x direction. Similarly, the partial derivative of z with respect to y, $\partial z/\partial y$, is the slope of the figure in the y direction. We can calculate the value of these derivatives with the gradient function.

```
[dzdx, dzdy] = gradient(z,X,Y)
```

In this case, the `gradient` function requires three inputs; the two-dimensional array, a one-dimensional array in the x direction, and a one-dimensional array in the y direction. An alternative syntax simplifies the input by just specifying the spacing in each direction, if the x and y vectors are evenly spaced.

```
[dzdx, dzdy] = gradient(z,0.1,0.1)
```

A combination of the `contour` and `quiver` plot functions allows us to visualize these results (see Figure 13.23(b)).

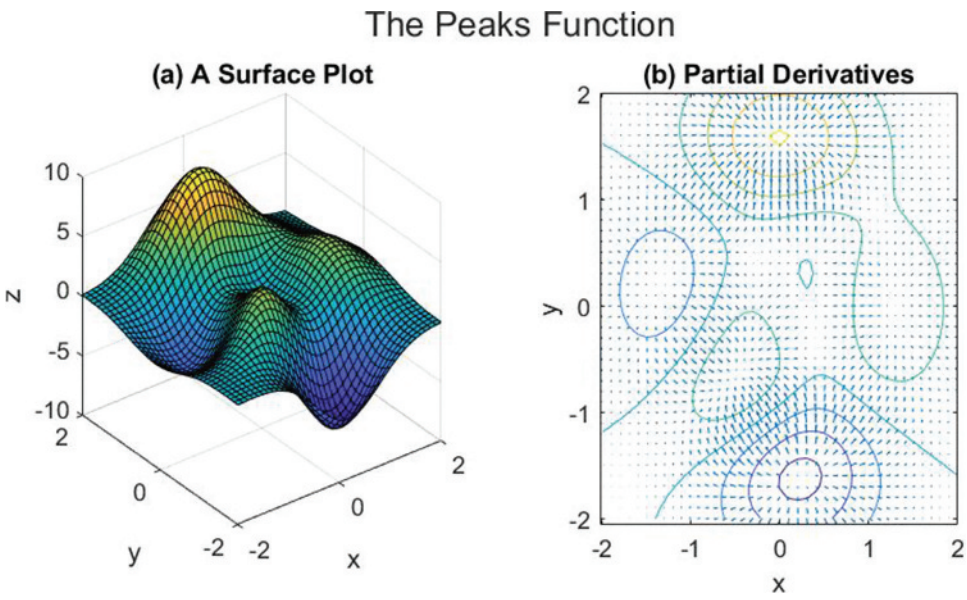


Figure 13.23
(a) Three-dimensional data, such as peaks, are conveniently visualized with a surface plot. (b) A combination contour and quiver plot illustrates the magnitude of the partial derivatives.

```
contour(x,y,z)
hold on
quiver(x,y,dzdx,dzdy)
hold off
```

PRACTICE EXERCISE 13.4

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Consider the following equation:

$$y = x^3 + 2x^2 - x + 3$$

Define an x vector from -5 to $+5$, and use it together with the `diff` function to approximate the derivative of y with respect to x, using the forward difference approach. Found analytically, the derivative is

$$\frac{dy}{dx} = y' = 3x^2 + 4x - 1$$

Evaluate this function, using your previously defined x vector. How do your results differ?

Function	Derivative
$y = \sin(x)$	$\frac{dy}{dx} = \cos(x)$
$y = x^5 - 1$	$\frac{dy}{dx} = 5x^4$
$y = 5xe^x$	$\frac{dy}{dx} = 5e^x + 5xe^x$

2. Repeat Exercise 1 for the following functions and their derivatives:
3. Use the `gradient` function to find the value of the derivatives in the previous problems.
4. Plot your results and compare the two approaches. Recall that the forward difference approach will provide one fewer values than the length of the `x` array. Be sure to pad the result array with a final value of `NaN` to make plotting easier.

13.5 NUMERICAL INTEGRATION

Integration is often thought of as finding the area under a curve, a technique sometimes called **quadrature**. Consider again the sample data, plotted in Figure 13.24. The area under the curve can be found by dividing the area into rectangles, then summing the contributions from all the rectangles:

$$A = \sum_{i=1}^{i=n-1} (x_{i+1} - x_i)(y_{i+1} + y_i)/2$$

The MATLAB commands to calculate this area are

```
avg_y = y(1:5)+diff(y)/2;
sum(diff(x).*avg_y)
```

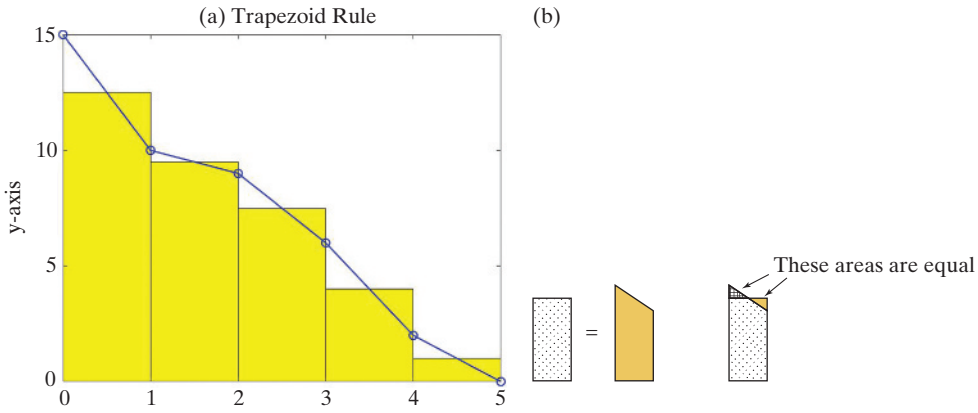


Figure 13.24
(a) The area under a curve can be approximated with the trapezoid rule. (b) The area of a trapezoid can be modeled with a rectangle.

This is called the **trapezoid rule**, since the rectangles have the same area as a trapezoid drawn between adjacent elements, Figure 13.24(b). MATLAB includes a built-in function, `trapz`, which gives the same result, and uses the syntax

```
trapz(x,y)
```

The area under a curve can be approximated by a function instead of data by creating a set of ordered *x*–*y* pairs. Better approximations are found as the number of elements in the *x* and *y* vectors are increased. For example, to find the area under the curve defined by the function

$$y = f(x) = x^2$$

from 0 to 1, a vector of 11 *x*-values would be defined and the corresponding *y*-values calculated:

```
x = 0:0.1:1;
y = x.^2;
```

The calculated values are plotted in Figure 13.25 and are used to find the area under the curve:

```
trapz(x,y)
```

This result gives us an approximation of the area under the curve defined by the function:

```
ans =
  0.3350
```

The preceding answer corresponds to an approximation of the integral from $x = 0$ to $x = 1$, or

$$\int_0^1 x^2 \, dx$$

KEY IDEA
Use `trapz` for ordered pairs of data. Use `integral` for functions.

MATLAB includes a built-in function, `integral`, which calculates the integral of a function without requiring the user to specify how the rectangles shown in Figure 13.25 are defined. It requires the user to enter a function handle in the first field; the last two fields in the function define the limits of integration, in this case from 0 to 1.

For example, consider an anonymous function for a third-order polynomial.

```
fun_handle = @(x) -x.^3 + 20*x.^2 -5
```

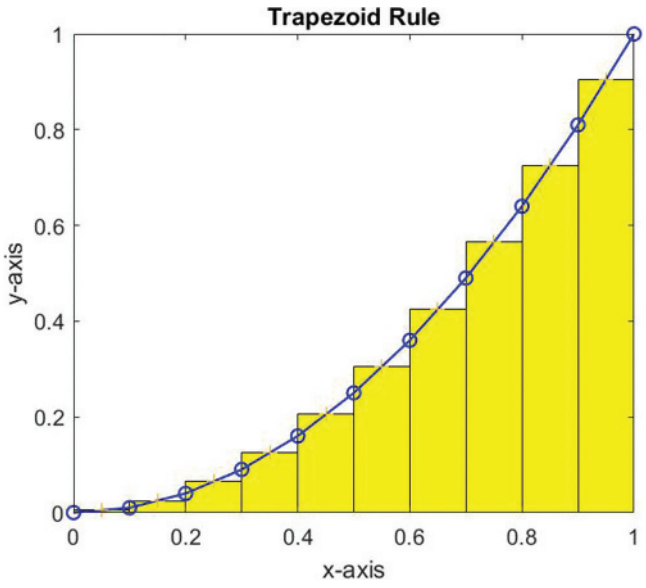


Figure 13.25
The integral of a function can be estimated with the trapezoid rule.

Now plot the function, to see how it behaves. The easiest approach is to use `fplot`, since it also accepts a function handle:

```
fplot(fun_handle,[-5,25])
```

The resulting plot is shown in Figure 13.26(a). The integral of this function

$$\int_{-5}^{25} - x^3 + 20x^2 - 5$$

is the shaded area under the curve, as shown in Figure 13.26(b).

Finally, to evaluate the integral use the `integral` function, with the function handle as input:

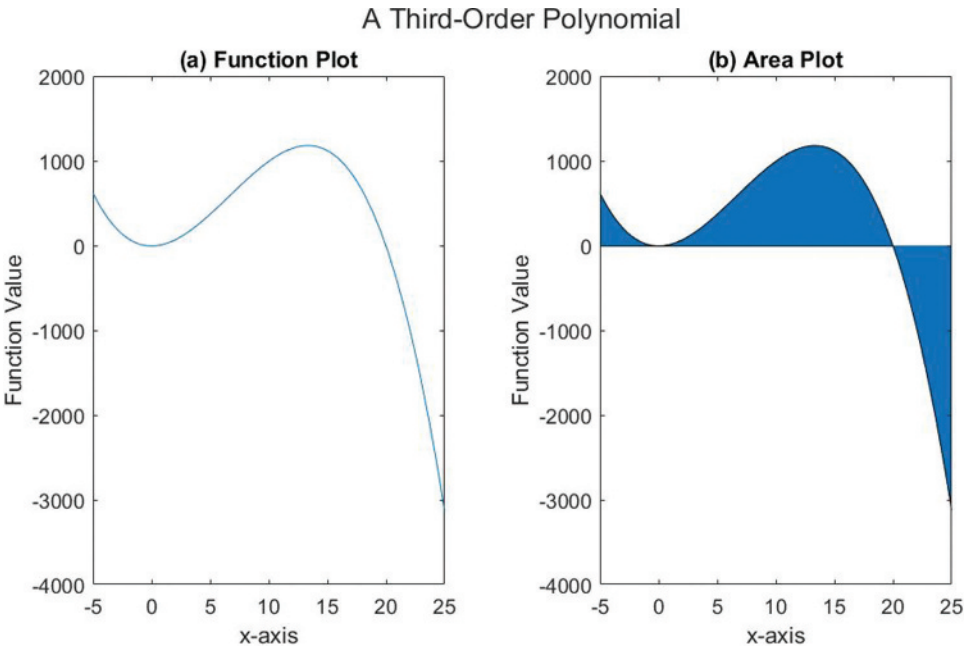


Figure 13.26
The integral of a function between two points can be thought of as the area under the curve. (a) This graph was created using `fplot` with a function handle representing the third-order polynomial. (b) This version of the graph was created with the `area` plotting function. The shaded portion of the figure represents the integral of the function between the plotted bounds of –5 to +25.

KEY IDEA
Numerical integration techniques all use various forms of quadrature - finding the area of geometric forms.

```
integral(fun_handle,0,25)
ans =
  6.3854e+003
```

Previous versions of MATLAB used the very similar `quad` and `quadl` functions instead of the newer `integrate` function to implement numerical integration. These functions may still appear in older code, but will eventually be eliminated.

The `integral2` and `integral3` functions are very similar, and are used to solve double and triple integrals respectively. You can find out more about how these techniques work by consulting a numerical methods textbook, such as John H. Mathews and Kurtis D. Fink, *Numerical Methods Using MATLAB*, 4th ed. (Upper Saddle River, NJ: Pearson, 2004).

PRACTICE EXERCISE 13.5

The correct answers can be found on the Pearson website at https://www.pearsonhighered.com/moore.

1. Consider the following equation:

$$y = x^3 + 2x^2 - x + 3$$

(a) Use the `trapz` function to estimate the integral of y with respect to x, evaluated from -1 to 1. Use 11 values of x, and calculate the corresponding values of y as input to the `trapz` function.

(b) Use the `integral` function to find the integral of y with respect to x, evaluated from -1 to 1.

(c) Compare your results with the values found by using the symbolic toolbox function `int` and the following analytical solution (remember that the `integral` function takes input expressed with array operators such as **.*** or **.^**, but that the `int` function takes a symbolic representation that does not use these operators):

$$\int_a^b (x^3 + 2x^2 - x + 3)\ dx =$$

$$\left(\frac{x^4}{4} + \frac{2x^3}{3} - \frac{x^2}{2} + 3x\right)\Big|_a^b =$$

$$\frac{1}{4}(b^4 - a^4) + \frac{2}{3}(b^3 - a^3) - \frac{1}{2}(b^2 - a^2) + 3(b - a)$$

2. Repeat Exercise 1 for the following functions:

Function	Integral
$y = \sin(x)$	$\int_a^b \sin(x)\,dx = \cos(x)\|_a^b = \cos(b) - \cos(a)$
$y = x^5 - 1$	$\int_a^b (x^5 - 1)\,dx = \left(\frac{x^6}{6} - x\right)\Big\|_a^b = \left(\frac{b^6 - a^6}{6} - (b - a)\right)$
$y = 5x^*e^x$	$\int_a^b (5e^x)\,dx = (-5e^x + 5xe^x)\|_a^b =$ $(-5(e^b - e^a) + 5(be^b - ae^a))$

EXAMPLE 13.5

CALCULATING MOVING BOUNDARY WORK

In this example, we'll use MATLAB's numeric integration technique, `integral`, to find the work produced in a piston–cylinder device by solving the equation

$$W = \int PdV$$

based on the assumption that

$$PV = nRT$$

where

P = pressure, kPa,
V = volume, m^3,
n = number of moles, kmol,
R = universal gas constant, 8.314 kPa m^3/kmol K, and
T = temperature, K.

We also assume that (1) the piston contains 1 mol of gas at 300 K, and (2) the temperature stays constant during the process.

1. State the problem
Find the work produced by the piston–cylinder device shown in Figure 13.27.

2. Describe the input and iutput
Input

$T = 300$ K
$n = 1$kmol
$V_1 = 1\ m^3$, $V_2 = 5\ m^3$ } limits of integration

Output Work done by the piston–cylinder device

3. Develop a hand example
Solving the ideal gas law

$$PV = nRT$$

or

$$P = nRT/V$$

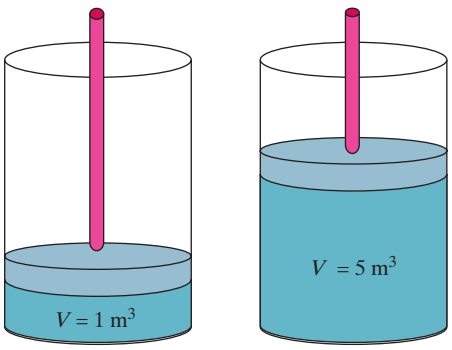


Figure 13.27
A piston–cylinder device.

(*continued*)

for P and performing the integration gives

$$W = \int \frac{nRT}{V}\, dV = nRT \int \frac{dV}{V} = nRT \ln\left(\frac{V_2}{V_1}\right)$$

Substituting in the values, we find that

$$W = 1 \text{ kmol} \times 8.314 \text{ kJ/kmol K} \times 300 \text{ K} \times \ln\left(\frac{V_2}{V_1}\right)$$

Since the integration limits are $V_2 = 5 \text{ m}^3$ and $V_1 = 1 \text{ m}^3$, the work becomes

$$W = 4014 \text{ kJ}$$

Because the work is positive, it is produced by (and not on) the system.

4. Develop a MATLAB solution

```
%% Example 13.5
% Calculating boundary work, using MATLAB's
% Quadrature functions
clear, clc
% Define constants
n=1;
R=8.314;
T=300
% Define an anonymous function for P
P=@(V) n*R*T./V;
%% Use integral to evaluate the integral
integral(P,1,5)
```

which returns the following results in the command window

```
ans =
  4.0143e+003
```

Notice that in this solution we defined an anonymous function for `P`, and used the function handle as input to integral.

5. Test the solution
We compare the results with our hand solution. The results are the same. It also helps to obtain a solution from the symbolic toolbox. Why do we need both kinds of MATLAB solution? Because some problems cannot be solved with MATLAB's symbolic tools, and others (those with singularities) are ill-suited to a numerical approach.

13.6 SOLVING DIFFERENTIAL EQUATIONS NUMERICALLY

MATLAB includes a number of functions that solve ordinary differential equations of the form

$$\frac{dy}{dt} = f(t, y)$$

numerically. In order to solve higher-order differential equations (and systems of differential equations), they must be reformulated into a system of first-order expressions.

This section outlines the major features of the ordinary differential equation solver functions. For more information, consult the help feature.

Not every differential equation can be solved by the same technique, so MATLAB includes a wide variety of differential equation solvers (see Table 13.6). However, all of these solvers have the same format. This makes it easy to try different techniques by just changing the function name.

KEY IDEA
MATLAB includes a large family of differential equation solvers.

Each solver requires the following three inputs as a minimum:

- A function handle to a function that describes the first-order differential equation or system of differential equations in terms of t and y, `fun`
- The time span of interest, `tspan = [t0,tf]`.
- An initial condition for each equation in the system `i_cond=[y1(1), y2(1),y3(1)...]`

The solvers all return an array of t- and y-values:

```
[t,y]=odesolver(fun,tspan,icond)
```

Table 13.6 MATLAB's Differential Equation Solvers

Ordinary Differential Equation Solver Function	Type of Problems Likely to Be Solved with This Technique	Numerical Solution Method	Comments
ode45	Nonstiff differential equations	Runge–Kutta	Best choice for a first-guess technique if you do not know much about the function. Uses an explicit Runge–Kutta (4, 5) formula called the Dormand–Prince pair.
ode23	Nonstiff differential equations	Runge–Kutta	This technique uses an explicit Runge–Kutta (2, 3) pair of Bogacki and Shampine. If the function is "mildly stiff," this may be a better approach than `ode45`.
ode78	Nonstiff differential equations	Seventh- and eighth-order Runge-Kutta	Can be more efficient than ode45 on nonstiff problems that are smooth
ode89	Nonstiff differential equations	Eighth- and ninth-order Runge-Kutta	Can be more efficient than ode78 on very smooth problems
ode113	Nonstiff differential equations	Adams	Unlike `ode45` and `ode23`, which are single-step solvers, this technique is a multistep solver.
ode15s	Stiff differential equation and differential algebraic equations	NDFs (BDFs)	Uses numerical differentiation formulas (NDFs) or backward differentiation formulas (BDFs). It is difficult to predict which technique will work best on a stiff differential equation.
ode23s	Stiff differential equations	Rosenbrock	Modified second-order Rosenbock formulation.
ode23t	Moderately stiff differential equations and differential algebraic equations	Trapezoid rule	Useful if you need a solution without numerical damping.
ode23tb	Stiff differential equations	TR–BDF2	This solver uses an implicit Runge–Kutta formula with the trapezoid rule (TR) and a second-order backward differentiation formula (BDF2).
ode15i	Fully implicit differential equations	BDF	This solver uses a backward difference formula (BDF) to solve implicit differential equations of the form $f(y, y', t) = 0$.

If you don't specify the resulting arrays `[t,y]`, the functions create a plot of the results.

13.6.1 Function Handle Input

As we've discussed before, a function handle is a "nickname" for a function. It can refer to either a standard MATLAB function, stored as either an M-file or an MLX-file, or an anonymous MATLAB function. Recall that the differential equations we're discussing are of the form

$$\frac{dy}{dt} = f(t, y)$$

so the function handle is equivalent to dy/dt.

Here's an example of an anonymous function for a single simple differential equation:

```
dydt = @(t,y) 2*t
```

corresponds to $\frac{dy}{dt} = 2t$

Although this particular function doesn't use a value of y in the result $(2t)$, y still needs to be part of the input in order to be acceptable input to the ode solvers.

If you want to specify a system of equations, it is probably easier to define a function M-file or MLX-file, or to define the function as a local function at the end of the program file. The output of the function must be a column vector of first-derivative values, as in

```
function dy=twofuns(t,y)
dy(1) = y(2);
dy(2) = -y(1);
dy=[dy(1); dy(2)];
end;
```

This function represents the system

$$\frac{dy}{dt} = x$$
$$\frac{dx}{dt} = -y$$

which could also be expressed in a more compact notation as

$$y_1' = y_2$$
$$y_2' = -y_1$$

where the prime indicates the derivative with respect to time, and the functions with respect to time are y_1, y_2, and so on. In this notation, the second derivative is equal to y'', and the third derivative is y''':

$$y' = \frac{dy}{dt}, \quad y'' = \frac{d^2y}{dt^2}, \quad y''' = \frac{d^3y}{dt^3}$$

13.6.2 Solving the Problem

Both the time span of interest, and the initial conditions for each equation, are entered as vectors into the solver equations, along with the function handle. To demonstrate, let's solve the equation

$$\frac{dy}{dt} = 2t$$

We created an anonymous function for this ordinary differential equation in the previous section, and called it `dydt`. We'll evaluate y from -1 to 1, and specify the initial condition as

$$y(-1) = 1$$

If you don't know how your equation or system of equations behaves, your first try should be `ode45`:

```
[t,y] = ode45(dydt,[-1,1],1)
```

This command returns an array of t-values and a corresponding array of y-values. You can either plot these yourself or allow the solver function to plot them if you don't specify the output array:

```
ode45(dydt,[-1,1],1)
```

The results are shown in Figure 13.28 and are consistent with the analytical solution, which is

$$y = t^2$$

Note that the first derivative of this function is $2t$ and that $y = 1$ when $t = -1$.

When the input function for a system of functions is stored in a separate M-file or MLX-file, the syntax is slightly different. The handle for an existing file is defined as `@file_name`. To solve the system of equations described in `twofuns` (from the previous section), we use the command

```
ode45(@twofuns,[-1,1],[1,1])
```

We could also assign a function handle to the function file such as

```
some_fun = @twofuns
```

and use it as input to the differential equation solver

```
ode45(some_fun,[-1,1],[1,1])
```

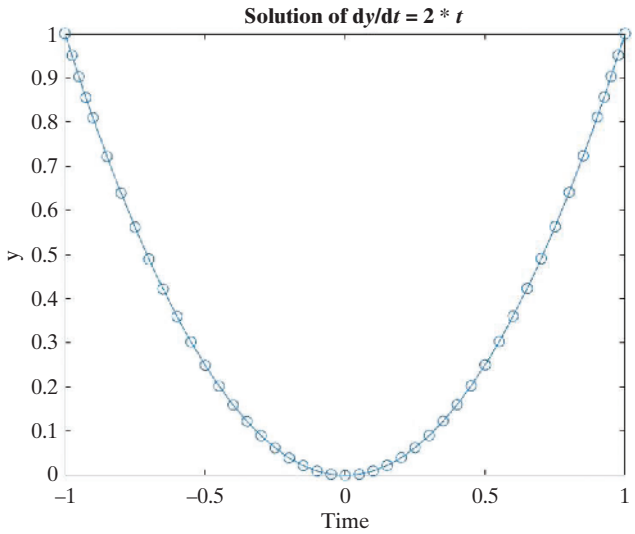


Figure 13.28
This figure was generated automatically by the `ode45` function. The title and labels were added in the usual way.

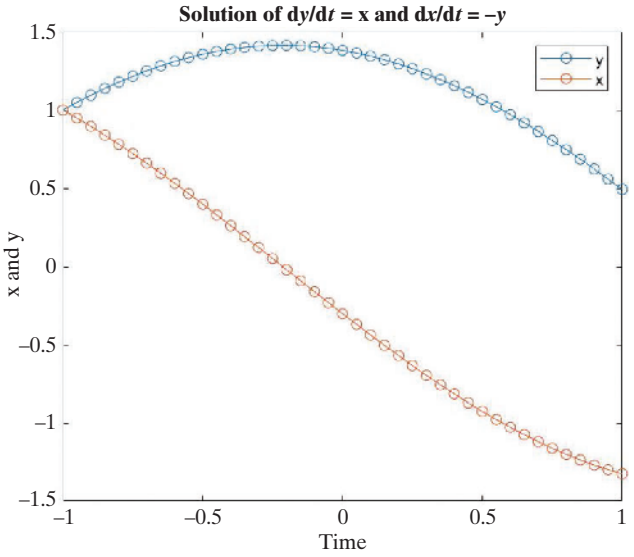


Figure 13.29
This system of equations was solved with `ode45`. The title, labels, and legend were added in the usual way.

The same syntax is used if the function is stored as a local function at the end of the program file.

The time span of interest is from −1 to 1, and the initial conditions are both 1. Notice that there is one initial condition for each equation in the system. The results are shown in Figure 13.29.

13.6.3 Solving Higher-Order Differential Equations

The ode series of functions (such as `ode45` or `ode23`) is used to solve either a single first-order differential equation, or a system of first-order differential equations. But what if you need to solve a higher-order problem? Fortunately, a higher-order differential equation can be expressed as a series of equations by making some simple substitutions. Consider the following equation:

$$\frac{\mathrm{d}^2y}{\mathrm{d}t^2} + \frac{\mathrm{d}y}{\mathrm{d}t} = y + t$$

We can reformulate it into a system of equations by introducing a new variable, z. Let

$$z = \frac{\mathrm{d}y}{\mathrm{d}t}$$

It's then easy to see that

$$\frac{\mathrm{d}z}{\mathrm{d}t} = \frac{\mathrm{d}^2y}{\mathrm{dt}^2}$$

Substituting into the original equation, we get

$$\frac{\mathrm{d}z}{\mathrm{d}t} + \frac{\mathrm{d}y}{\mathrm{d}t} = y + t,$$

which is a first-order differential equation. Effectively we've replaced

$$\frac{\mathrm{d}^2y}{\mathrm{d}t^2} + \frac{\mathrm{d}y}{\mathrm{d}t} = y + t$$

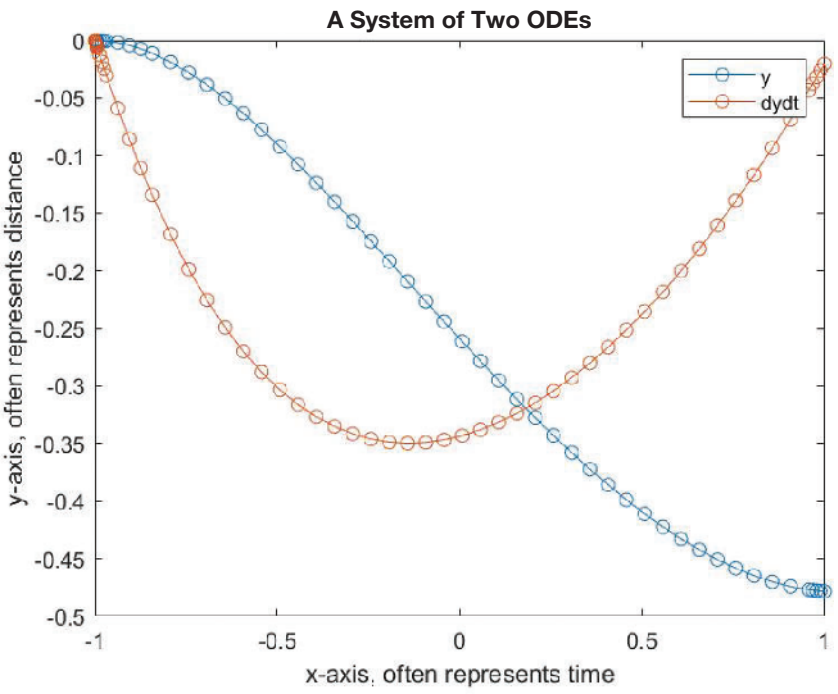


Figure 13.30
A higher-order differential equation is solved by creating a system of equations that represents the same information. A second-order ODE requires two equations, resulting in two lines represented in the graphical output, one for y, and one for dy/dt.

with the following two equations, which have been rearranged to solve for the first derivative of our two dependent variables, y and z

$$\frac{dy}{dt} = z$$

and

$$\frac{dz}{dt} = y + t - \frac{dy}{dt}$$

Now all we need to do is create a function file to use in one of the ode solvers. The function should have two inputs, which are typically called t and y. The variable t is the independent variable, and the variable y is an array of *dependent variables*. In this example, $y(1)$ corresponds to the y used in the hand formulation, and $y(2)$ corresponds to z. The function containing the system of equations should look like this:

```
function dydt = twoeq(t,y)
dydt(1) = y(2);
dydt(2) = y(1) + t - dydt(1);
dydt = dydt';
end;
```

Notice that the function output has been formulated as a column vector, as required by the ode solvers. Also recall that the function name is arbitrary. We could have called it anything, but `twoeq` is descriptive.

Once the system of equations is defined in a function it is available to use as input to an ode solver. For example, if the range of time is defined as -1 to $+1$, and the initial conditions are defined as $y = 0$ and $z = 0$ (which is the same as $y = 0$ and $dy/dt = 0$), then the command becomes

```
ode45(@twoeq,[-1,1],[0,0])
```

which gives the results shown in Figure 13.30. A problem where the starting values are known is called an **initial value** problem.

In this example the system of differential equations, `twoeq`, can either be stored as a separate file or as a local function at the end of the main progam. The syntax is the same in either case.

13.6.4 Boundary Value Problems

Initial value problems are a special case of the general category of **boundary value** problems. In initial value problems, the starting value of the dependent variable(s) is known. In the more general boundary value problem, the value of the dependent variable is known at some point not necessarily the initial point. Reconsider the function from the previous section, which describes a system of two ordinary differential equations. What would happen if we didn't know the initial value of y and dy/dt, but instead knew the value of y at both $t = -1$ and $t = 1$? This problem cannot be solved with the ode solver family of functions, but it can be solved using the `bvp4c` function. The `bvp4c` function requires three inputs:

- A function handle to the system of ode's to be solved.
- A function handle to another function that solves for the residual values (the errors) of the function based on the known boundary conditions and the predicted values at those points.
- A set of guesses for the initial conditions.

The first function handle is exactly the same as we used for the ode solver set of functions. It should contain the equations for the derivatives of interest, and the results must be a column vector.

To solve the problem, a guess is made for the initial value of all the dependent variables, then during the solution process the program checks to see how it did by comparing the calculated boundary values with the actual values. For example, if:

$$\text{at } t = -1, \quad y = 0 \qquad \text{and}$$
$$\text{at } t = 1, \quad y = 3$$

the program would solve the system of equations based upon an initial guess of both y and dy/dt, and would then check to see how close the result is at $t = -1$ and $t = +1$ (i.e., it would check to see if $y = 0$ at $t = -1$, and if $y = 3$ at $t = +1$). This is accomplished using a boundary condition function where the equations are arranged so that if the correct boundary condition is calculated, the function values are zero. In the case of our example,

```
function residual=bc(y_initial, y_final)
residual(1) = y_initial(1) + 0;
residual(2) = y_final(1) - 3;
residual = residual';
end;
```

The input to `bc` is two arrays, in this example called `y_initial` and `y_final`. Each array is composed of the dependant variables used in the problem—usually `y`, `dydt`, etc. Thus `y_initial(1)` is the initial value of `y`, but `y_initial(2)` is the initial value of `dydt`. If this function is executed for the values of `y_initial(1) = 0` and `y_final(1) = 3`, the result will be a column of zeros. Any other result means that the program has calculated the wrong values for `y_initial` and `y_final`, and the guesses for the initial conditions must be updated according to the function's algorithm, which is a finite difference strategy.

The last input to the `bvp4c` function is a mesh of guesses for the problem solution, which are used as the starting point in the calculation. MATLAB provides a helper function, `bvpinit`, to help create this mesh, which is stored as a structure array. It requires two inputs; an array of values corresponding to the independent variable (in this case t), and initial guesses for each of the variables defined in the ode system of equations. In our case, there are two equations, so we'll need a guess for y and dy/dt. The mesh need not be particularly fine, and the initial guesses need not be very good. For example:

KEY IDEA

You can store the functions used as input to the odesolvers and `bvp4c` either in separate files or as local functions at the end of the main program file or as anonymous functions.

```
initial_guess = bvpinit(-1:.5:1, [0, -1])
```

specifies five t values from -1 to 1 (-1, -0.5, 0, 0.5, 1) and initial guesses of $y = 0$ and $dy/dt = -1$ at all values of t.

Once the function describing the system of ode's, the function defining the residuals, and the initial guesses created with `bvpinit` have been created, the `bvp4c` function can be executed.

```
bvp4c(@twoeq, @bc, initial_guess)
```

which returns

```
ans =
       x: [1x9 double]
       y: [2x9 double]
      yp: [2x9 double]
  solver: 'bvp4c'
```

The result is a structure array, where `x` is the value of the independent variable (denoted as t in this problem) and where an array of `y` values corresponds to the solutions to the system of ode's. In this case, y and dy/dt. The `bvp4c` function assumes the name of your independent variable is x as a default—but works just as well if you happen to call your independent variable t. The default is x because boundary value problems are often based on changes in location, not in time.

To access the array of `x` values simply use the structure syntax, `ans.x`. If we had chosen to assign a name such as `solution` for the result instead of defaulting to `ans`, the structure would be called `solution`, and the `x` values would be stored in `solution.x`. The values of most interest are the `y` values, which can also be accessed using structure syntax, such as `solution.y`. To plot the results in a manner similar to that displayed by the odesolvers, use the code

```
plot(ans.x,ans.y, '-o')
```

or, if the results were named `solution`

```
plot(solution.x, solution.y, '-o'),
```

which gives the results shown in Figure 13.31. The annotations (titles, legends, etc.) were added in the usual way.

13.6.5 Partial Differential Equations

MATLAB also includes a limited partial differential equation solver, `pdepe`. For more information, consult the MATLAB help feature.

```
doc pdepe
```

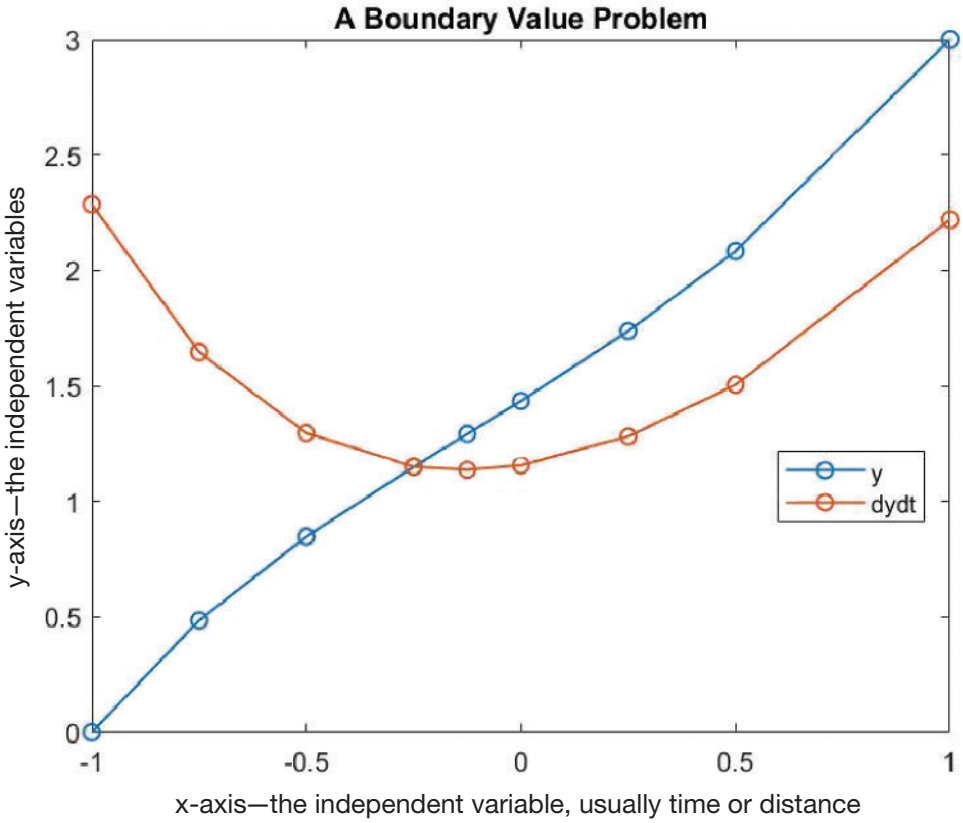


Figure 13.31
A boundary value problem solved using `bvp4c`.

SUMMARY

Tables of data are useful for summarizing technical information. However, if you need a value that is not included in the table, you must approximate that value by using some sort of interpolation technique. MATLAB includes such a technique, called `interp1`. This function requires three inputs: a set of *x*-values, a corresponding set of *y*-values, and a set of *x*-values for which you would like to estimate *y*-values. The function defaults to a linear interpolation technique, which assumes that you can approximate these intermediate *y*-values as a linear function of *x* that is,

$$y = f(x) = ax + b$$

A different linear function is found for each set of two data points, ensuring that the line approximating the data always passes through the tabulated points.

The `interp1` function can also model the data by using higher-order approximations, the most common of which is the cubic spline. The approximation technique is specified as a character string in a fourth optional field of the `interp1` function. If it's not specified, the function defaults to linear interpolation. An example of the syntax is

```
new_y = interp1(tabulated_x, tabulated_y, new_x, 'spline')
```

In addition to the `interp1` function, MATLAB includes a two-dimensional interpolation function called `interp2`, a three-dimensional interpolation function called `interp3`, and a multidimensional interpolation function called `interpn`.

Curve-fitting routines are similar to interpolation techniques. However, instead of connecting data points, they look for an equation that models the data as accurately as possible. Once you have an equation, you can calculate the corresponding values of *y*. The curve that is modeled does not necessarily pass through the measured data points. MATLAB's curve-fitting function is called `polyfit` and models the data as a polynomial by means of a least-squares regression technique. The function returns the coefficients of the polynomial equation of the form

$$y = a_1x^n + a_2x^{n-1} + a_3x^{n-2} + \cdots + a_nx + a_{n+1}$$

These coefficients can be used to create the appropriate expression in MATLAB, or they can be used as the input to the `polyval` function to calculate values of *y* at any value of *x*. For example, the following statements find the coefficients of a second-order polynomial to fit the input *x*–*y* data and then calculate new values of *y*, using the polynomial determined in the first statement:

```
coef = polyfit(x,y,2)
y_first_order_fit = polyval(coef,x)
```

These two lines of code could be shortened to one line by nesting functions:

```
y_first_order_fit = polyval(polyfit(x,y,1),x)
```

To fit data to a straight line that goes through the zero point, use left division to calculate the coefficient of the model, $y = a*x$.

```
a = x'\y'
```

The *x* and *y* arrays used as input must be column arrays.

MATLAB also includes an interactive curve-fitting capability that allows the user to model data not only with polynomials, but with more complicated mathematical functions. The basic curve-fitting tools can be accessed from the *Tools* menu in the figure window. More extensive tools are available in the curve-fitting toolbox.

Numerical techniques are used widely in engineering to approximate both derivatives and integrals. Functions to find derivatives and integrals can also be found in the symbolic toolbox.

The MATLAB `diff` function finds the difference between values in adjacent elements of a vector. By using the `diff` function with vectors of *x*- and *y*-values, the derivative can be approximated with the command

```
slope = diff(y)./diff(x)
```

The more closely spaced the *x* and *y* data are, the closer will be the approximation of the derivative.

The `gradient` function uses a forward difference approach to approximate the derivative at the first point in an array. It uses a backward difference approach for the final value in the array, and a central difference approach for the remainder of the points. In general, the central difference approach gives a more accurate approximation of the derivative than either of the other two techniques.

Integration of ordered pairs of data is accomplished using the trapezoid rule, with the `trapz` function. This approach can also be used with functions, by creating a set of ordered pairs based on a set of *x* values and the corresponding *y* values.

Integration of functions is accomplished more directly with the `integral` function. This function requires the user to input both a function and its limits of integration. The function can either be represented as an anonymous function, or as a function stored in a separate file. The `integral` function attempts to return an answer accurate to within 1×10^{-6}.

MATLAB includes a series of solver functions for first-order ordinary differential equations and systems of equations. All of the solver functions use the common format

```
[t,y] = ode solver(function_handle,[initial_time,
        final_time], [initial_cond_array])
```

A good first try is usually the `ode45` solver function, which uses a Runge–Kutta technique. Other solver functions have been formulated for stiff differential equations and implicit formulations.

The ode solver functions require that the user know the initial conditions for the problem. If, instead, boundary conditions are known at other than the starting conditions, the `bvp4c` function should be used.

MATLAB SUMMARY

The following MATLAB summary lists and briefly describes the commands and functions that were defined in this chapter:

Commands and Functions	
`bvp4c`	boundary value problem solver for ordinary differential equations
`diff`	computes the differences between adjacent values in an array if the input is an array; finds the symbolic derivative if the input is a symbolic expression
`gradient`	finds the derivative numerically using a combination of forward, backward, and central difference techniques
`int`	finds the symbolic integral
`integral`	computes the integral of a function
`interp1`	approximates intermediate data, using either the default linear interpolation technique or a specified higher-order approach
`interp2`	two-dimensional interpolation function
`interp3`	three-dimensional interpolation function
`interpn`	multidimensional interpolation function
`ode45`	ordinary differential equation solver
`ode23`	ordinary differential equation solver
`ode78`	ordinary differential equation solver
`ode89`	ordinary differential equation solver
`ode113`	ordinary differential equation solver
`ode15s`	ordinary differential equation solver
`ode23s`	ordinary differential equation solver
`ode23t`	ordinary differential equation solver
`ode23tb`	ordinary differential equation solver
`ode15i`	ordinary differential equation solver
`polyfit`	computes the coefficients of a least-squares polynomial
`polyval`	evaluates a polynomial at a specified value of x
`trapz`	approximates the integral based on ordered pairs of data

KEY TERMS

approximation
backward difference
boundary value problem
central difference
cubic equation
cubic spline
derivative
extrapolation
forward difference
interpolation
initial value problems
least squares
linear interpolation
linear regression
quadratic equation
quadrature
residual
trapezoid rule

PROBLEMS

Interpolation

13.1 Consider a gas in a piston–cylinder device in which the temperature is held constant. As the volume of the device was changed, the pressure was measured. The volume and pressure values are reported in the following table:

Volume (m^3)	Pressure (kPa), when T = 300 K
1	2494
2	1247
3	831
4	623
5	499
6	416

(a) Use linear interpolation to estimate the pressure when the volume is 3.8 m^3.
(b) Use cubic spline interpolation to estimate the pressure when the volume is 3.8 m^3.
(c) Use linear interpolation to estimate the volume if the pressure is measured to be 1000 kPa.
(d) Use cubic spline interpolation to estimate the volume if the pressure is measured to be 1000 kPa.

13.2 Use the data from Problem 13.1 and linear interpolation to create an expanded volume–pressure table with volume measurements every 0.2 m^3. Plot the calculated values on the same graph with the measured data. Show the measured data with circles and no line and the calculated values with a solid line.

13.3 Repeat Problem 13.2, using cubic spline interpolation.

13.4 The experiment described in Problem 13.1 was repeated at a higher temperature and the data recorded in the following table:

Volume (m^3)	Pressure (kPa), at 300 K	Pressure (kPa), at 500 K
1	2494	4157
2	1247	2078
3	831	1386
4	623	1039
5	499	831
6	416	693

Use these data to answer the following questions:

(a) Approximate the pressure when the volume is 5.2 m^3 for both temperatures (300 K and 500 K). (*Hint*: Make a pressure array that contains both sets of data; your volume array will need to be 6×1, and your pressure array will need to be 6×2.) Use linear interpolation for your calculations.

(b) Repeat your calculations, using cubic spline interpolation.

13.5 Use the data in Problem 13.4 to solve the following problems:

(a) Create a new column of pressure values at $T = 400$ K, using linear interpolation.

(b) Create an expanded volume–pressure table with volume measurements every 0.2 m^3, with columns corresponding to $T = 300$ K, $T = 400$ K, and $T = 500$ K.

13.6 Use the `interp2` function and the data from Problem 13.4 to approximate a pressure value when the volume is 5.2 m^3 and the temperature is 425 K.

Curve Fitting

13.7 Fit the data from Problem 13.1 with first-, second-, third-, and fourth-order polynomials, using the `polyfit` function:

- Plot your results on the same graph.
- Plot the actual data as a circle with no line.
- Calculate the values to plot from your polynomial regression results at intervals of 0.2 m^3.
- Do not show the calculated values on the plot, but connect the points with solid lines.
- Which model seems to do the best job?

13.8 The relationship between pressure and volume is not usually modeled by a polynomial. Rather, they are inversely related to each other by the ideal gas law,

$$P = \frac{nRT}{V}$$

We can plot this relationship as a straight line if we plot P on the y-axis, and $1/V$ on the x-axis. The slope then becomes the value of nRT. We can use the `polyfit` function to find this slope if we input P and $1/V$ to the function:

polyfit(1./V, P,1)

(a) Assuming that the value of n is 1 mol, and the value of R is 8.314 kPa/kmol K, show that the temperature used in the experiment is indeed 300 K.

(b) Create a plot with $1./V$ on the x-axis and P on the y-axis.

(c) Repeat using linear regression through 0.

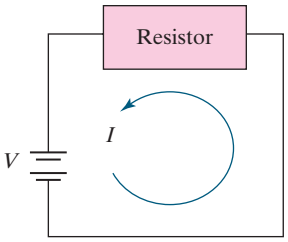


Figure P13.9
An electrical circuit.

13.9 Resistance and current are inversely proportional to each other in electrical circuits:

$$I = \frac{V}{R}$$

Consider the following data collected from an electrical circuit to which an unknown constant voltage has been applied (see Figure P13.9):

Resistance (ohms)	Measured Current (amps)
10	11.11
15	8.04
25	6.03
40	2.77
65	1.97
100	1.51

(a) Plot resistance (R) on the x-axis and measured current (I) on the y-axis.
(b) Create another plot with $1/R$ on the x-axis and I on the y-axis.
(c) Use `polyfit` to calculate the coefficients of the straight line shown in your plot in part (b). The slope of your line corresponds to the applied voltage.
(d) Use `polyval` to find calculated values of current (I) based on the resistors used. Plot your results in a new figure, along with the measured data.
(e) Find the applied voltage from part (c) using left division and compare your result to that found using polyfit. Which approach is superior?

13.10 Many physical processes can be modeled by an exponential equation. For example, chemical reaction rates depend on a reaction-rate constant that is a function of temperature and activation energy:

$$k = k_0 e^{-Q/RT}$$

In this equation,

R = universal gas constant, 8.314 kJ/kmol K,
Q = activation energy, in kJ/kmol,
T = temperature, in K, and
k_0 = constant whose units depend on characteristics of the reaction. One possibility is s^{-1}.

One approach to finding the values of k_0 and Q from experimental data is to plot the natural logarithm of k on the y-axis and $1/T$ on the x-axis. This should result in a straight line with slope $-Q/R$ and intercept $\ln(k_0)$—that is,

$$\ln(k) = \ln(k_0) - \frac{Q}{R}\left(\frac{1}{T}\right)$$

since the equation now has the form

$$y = ax + b$$

with $y = \ln(k)$, $x = 1/T$, $a = -Q/R$ and $b = \ln(k)$.
Now consider the following data:

T (K)	k (s^{-1})
200	1.46×10^{-7}
400	0.0012
600	0.0244
800	0.1099
1000	0.2710

(a) Plot the data with $1/T$ on the x-axis, and $\ln(k)$ on the y-axis.
(b) Use the `polyfit` function to find the slope of your graph, $-Q/R$, and the intercept, $\ln(k_0)$.
(c) Calculate the value of Q.
(d) Calculate the value of k_0.

13.11 Electrical power is often modeled as

$$P = I^2R$$

where

P = power, in watts,
I = current, in amperes, and
R = resistance, in ohms.

(a) Consider the following data and find the value of the resistor in the circuit by modeling the data as a second-order polynomial with the `polyfit` function:

Power (W)	Current (A)
50,000	100
200,000	200
450,000	300
800,000	400
1,250,000	500

(b) Plot the data and use the curve-fitting tools found in the figure window to determine the value of R by modeling the data as a second-order polynomial.

13.12 Using a polynomial to model a function can be very useful, but it is always dangerous to extrapolate beyond your data. We can demonstrate this pitfall by modeling a sine wave as a third-order polynomial.

(a) Define `x = -1:0.1:1`
(b) Calculate `y = sin(x)`
(c) Use the `polyfit` function to determine the coefficients of a third-order polynomial to model these data.
(d) Use the `polyval` function to calculate new values of `y` (`modeled_y`) based on your polynomial, for your `x` vector from −1 to 1.
(e) Plot both sets of values on the same graph. How good is the fit?
(f) Create a new `x` vector, `new_x = -4:0.1:4`.
(g) Calculate `new_y` values by finding `sin(new_x)`.
(h) Extrapolate `new_modeled_y` values by using `polyfit`, the coefficient vector you found in part (c) to model `x` and `y` between −1 and 1, and the `new_y` values.
(i) Plot both new sets of values on the same graph. How good is the fit outside of the region from −1 to 1?

Approximating Derivatives

13.13 Consider the following equation:

$$y = 12x^3 - 5x^2 + 3$$

(a) Define an `x` vector from -5 to $+5$, and use it together with the `diff` function to approximate the derivative of y with respect to x.

(b) Found analytically, the derivative of y with respect to x is

$$\frac{dy}{dx} = y' = 36x^2 - 10x$$

Evaluate this function, using your previously defined `x` vector. How do your results differ?

13.14 One very common use of derivatives is to determine velocities. Consider the following data, taken during a car trip in the United Satates from Salt Lake City, Utah, to Denver, Colorado:

Time (hours)	Distance (miles)
0	0
1	60
2	110
3	170
4	220
5	270
6	330
7	390
8	460

(a) Find the average velocity in mph during each hour of the trip.

(b) Plot these velocities on a bar graph. Edit the graph so that each bar covers 100% of the distance between entries.

13.15 Consider the following data, taken during a car trip from Salt Lake City to Los Angeles:

Time (hours)	Distance (miles)
0	0
1.0	75
2.2	145
2.9	225
4.0	300
5.2	380
6.0	430
6.9	510
8.0	580
8.7	635
9.7	700
10	720

(a) Find the average velocity in mph during each segment of the trip.
(b) Plot these velocities against the start time for each segment.
(c) Use the `find` command to determine whether any of the average velocities exceeded the speed limit of 75 mph.
(d) Is the overall average above the speed limit?

13.16 Consider the following data from a three-stage model rocket launch:

Time (seconds)	Altitude (meters)
0	0
1.00	107.37
2.00	210.00
3.00	307.63
4.00	400.00
5.00	484.60
6.00	550.00
7.00	583.97
8.00	580.00
9.00	549.53
10.00	570.00
11.00	699.18
12.00	850.00
13.00	927.51
14.00	950.00
15.00	954.51
16.00	940.00
17.00	910.68
18.00	930.00
19.00	1041.52
20.00	1150.00
21.00	1158.24
22.00	1100.00
23.00	1041.76
24.00	1050.00

(a) Create a plot with time on the x-axis and altitude on the y-axis.
(b) Use the `diff` function to determine the velocity during each time interval, and plot the velocity against the starting time for each interval.
(c) Use the `diff` function again to determine the acceleration for each time interval, and plot the acceleration against the starting time for each interval.
(d) Estimate the staging times (the time when a burnt-out stage is discarded and the next stage ignites) by examining the plots you've created.

Numerical Integration

13.17 Consider the following equation:

$$y = 5x^3 - 2x^2 + 3$$

Use the `integral` function to find the integral with respect to x, evaluated from -1 to 1. Compare your results with the values found with the

use of the symbolic toolbox function, `int`, and the following analytical solution

$$\int_a^b (5x^3 - 2x^2 + 3)\,dx =$$
$$\left(\frac{5x^4}{4} - \frac{2x^3}{3} + 3x\right)\Big|_a^b =$$
$$\frac{5}{4}(b^4 - a^4) + \frac{2}{3}(b^3 - a^3) + 3(b - a)$$

13.18 The equation

$$C_P = a + bT + cT^2 + dT^3$$

is an empirical polynomial that describes the behavior of the heat capacity C_P as a function of temperature in kelvins. The change in enthalpy (a measure of energy) as a gas is heated from T_1 to T_2 is the integral of this equation with respect to T:

$$\Delta h = \int_{T_1}^{T_2} C_P\,dT$$

Find the change in enthalpy of oxygen gas as it is heated from 300 to 1000 K, using the MATLAB quadrature function integral. The values of a, b, c, and d for oxygen are as follows:

$$a = 25.48$$
$$b = 1.520 \times 10^{-2}$$
$$c = -0.7155 \times 10^{-5}$$
$$d = 1.312 \times 10^{-9}$$

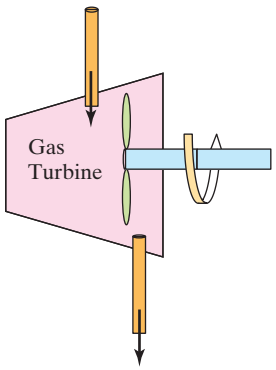


Figure P13.19
A gas turbine used to produce power.

13.19 In some sample problems in this chapter, we explored the equations that describe moving boundary work produced by a piston–cylinder device. A similar equation describes the work produced as a gas or a liquid flows through a pump, turbine, or compressor (see Figure P13.19).
In this case, there is no moving boundary, but there is shaft work, given by

$$\dot{W}_{\text{produced}} = -\int_{\text{inlet}}^{\text{outlet}} \dot{V}\,dP$$

This equation can be integrated if we can find a relationship between $\dot{V}$ and P. For ideal gases, that relationship is

$$\dot{V} = \frac{\dot{n}RT}{P}$$

If the process is isothermal, the equation for work becomes

$$\dot{W} = -\dot{n}RT\int_{\text{inlet}}^{\text{outlet}} \frac{dP}{P}$$

where

$\dot{n}$ = molar flow rate, in kmol/s,
R = universal gas constant, 8.314 kJ/kmol K,
T = temperature, in K,
P = pressure, in kPa, and
$\dot{W}$ = power, in kW.

Find the power produced in an isothermal gas turbine if

$$\begin{aligned} \dot{n} &= 0.1 \text{ kmol/s}, \\ R &= \text{universal gas constant, } 8.314 \text{ kJ/kmol K}, \\ T &= 400 \text{ K}, \\ P_{\text{inlet}} &= 500 \text{ kPa, and} \\ P_{\text{outlet}} &= 100 \text{ kPa} \end{aligned}$$

Differential Equations

13.20 Solve the following differential equation for values of t between 0 and 4, with the initial condition of $y = 1$ when $t = 0$,

$$\frac{dy}{dt} + \sin(t) = 1$$

(a) Analytically or using MATLAB's symbolic capabilities.
(b) Using the `ode45` function.
(c) Plot your results for both approaches.

13.21 Solve the following differential equation for values of t between 0 and 1, with the initial condition of $y = 0$ when $t = 0$.

$$\frac{dy}{dt} = t^2 + y$$

13.22 Blasius showed in 1908 that the solution to the incompressible flow field in a laminar boundary layer on a flat plate is given by the solution of the following third-order ordinary nonlinear differential equation

$$2\frac{d^3 f}{d\eta^3} + f\frac{d^2 f}{d\eta^2} = 0$$

Rewrite this equation into a system of three first-order equations, using the following substitutions:

$$\begin{aligned} h_1(\eta) &= f \\ h_2(\eta) &= \frac{df}{d\eta} \\ h_3(\eta) &= \frac{d^2 f}{d\eta^2} \end{aligned}$$

Solve using the `ode45` function with the following initial conditions:

$$\begin{aligned} h_1(0) &= 0 \\ h_2(0) &= 0 \\ h_3(0) &= 0.332 \end{aligned}$$

CHAPTER

14 Advanced Graphics

Objectives

After reading this chapter, you should be able to:

- Identify the three different types of image files.
- Assign an object name to plots and adjust plot properties.
- Create an animation using either of the two MATLAB techniques.
- Adjust lighting parameters, camera locations, and transparency values.
- Use visualization techniques for both scalar and vector information in three dimensions.

INTRODUCTION

Some of the basic graphs commonly used in engineering are the workhorse *x*–*y* plot, polar plots, and surface plots, as well as some graphing techniques more commonly used in business applications, such as pie charts, bar graphs, and histograms. MATLAB gives us significant control over the appearance of these plots, lets us manipulate images (such as digital photographs), and create three-dimensional representations (besides surface plots) of both data and models of physical processes.

14.1 IMAGES

Let us start our exploration of some of MATLAB's more advanced graphics capabilities by examining how images are handled with the `image` and `imagesc` functions. Because MATLAB is already an array-manipulation program, it makes sense that images are stored as arrays.

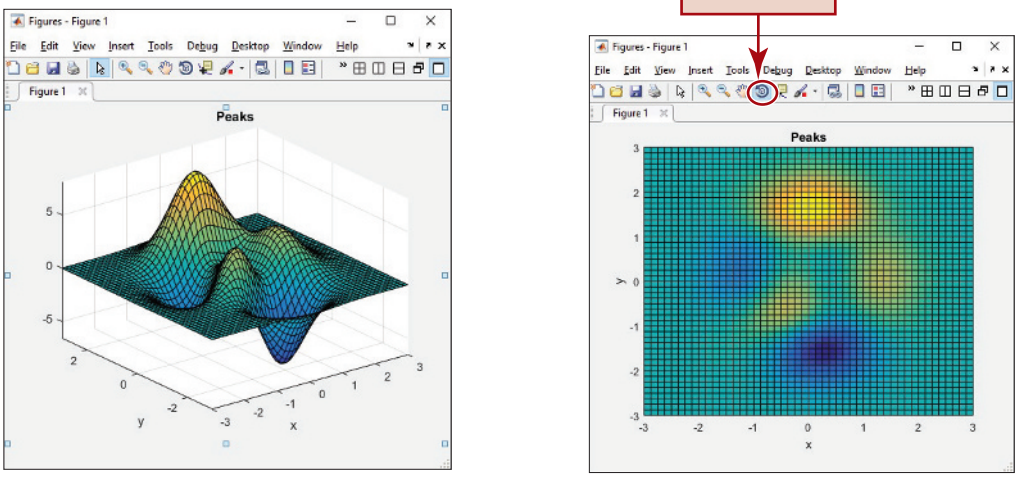


Figure 14.1
(a) The `peaks` function is built into MATLAB for use in demonstrating graphics capabilities. (Reprinted with permission from The Mathworks, Inc.) (b) A view of the surface plot of the `peaks` function looking down the *z*-axis.

We can create a three-dimensional surface plot of the `peaks` function by typing

```
surf(peaks)
```

We can manipulate the figure we have created (see Figure 14.1(a)) by using the interactive figure-manipulation tools, so that we are looking down from the top (see Figure 14.1(b)).

An easier way to accomplish the same thing is to use the pseudo color plot

```
pcolor(peaks)
```

We can remove the grid lines, which are plotted automatically, by specifying the shading option

```
shading flat
```

The colors in Figures 14.1–14.3 correspond to the values of *z*. The large positive values of *z* are yellow and the large negative values are blue. The value of *z* found in

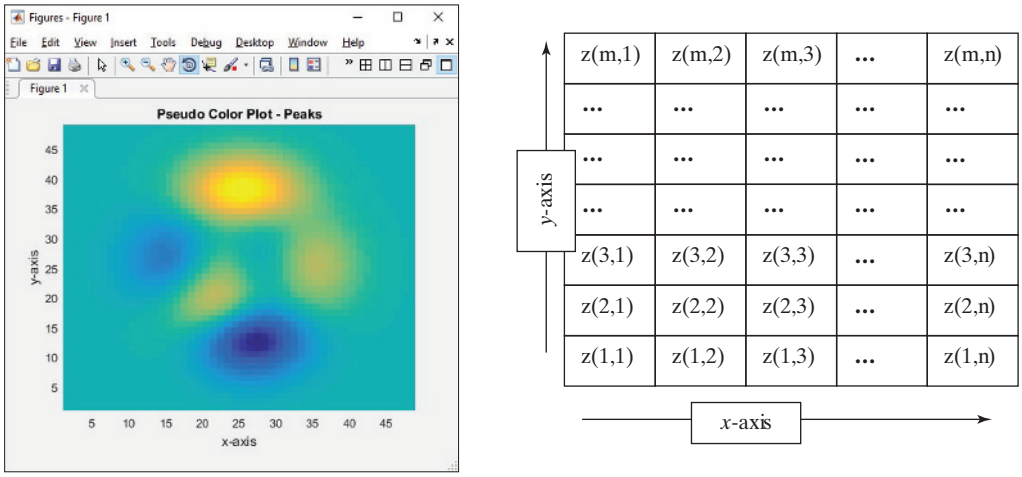


Figure 14.2
(a) A pseudo color plot (left) is the same thing as the view looking straight down at a surface plot. (b) Pseudo color plots organize the data on the basis of the right-hand rule, starting at the (0, 0) position on the graph (right).

the first *z* array element, $z(1, 1)$, is represented in the lower left-hand corner of the graph (see Figure 14.3(b)).

Although this strategy for representing data makes sense because of the coordinate system we typically use in graphing, it does not make sense for representing images such as photographs. When images are stored in arrays, we usually represent the data starting in the *upper* left-hand corner of the image and work across and down (see Figure 14.3(a)). In MATLAB, two functions used to display images—`image` and `imagesc`—use this format. The scaled image function (`imagesc`) uses the entire colormap to represent the data, just like the pseudo color plot function (`pcolor`). The results, obtained with

```
imagesc(peaks)
```

are shown in Figure 14.3(b).

Notice that the image is flipped in comparison to the pseudo color plot. Of course, in many graphics applications, it doesn't matter how the data are represented, as long as we understand the convention used. However, a photograph would be upside down in a vertical mirror image—clearly not an acceptable representation.

14.1.1 Image Types

MATLAB recognizes three different techniques for storing and representing images:

KEY IDEA
Two functions are used to display images, `imagesc` and `image`.

Intensity (or grayscale) **images**

Indexed images

RGB (or true color) images, also called image plots

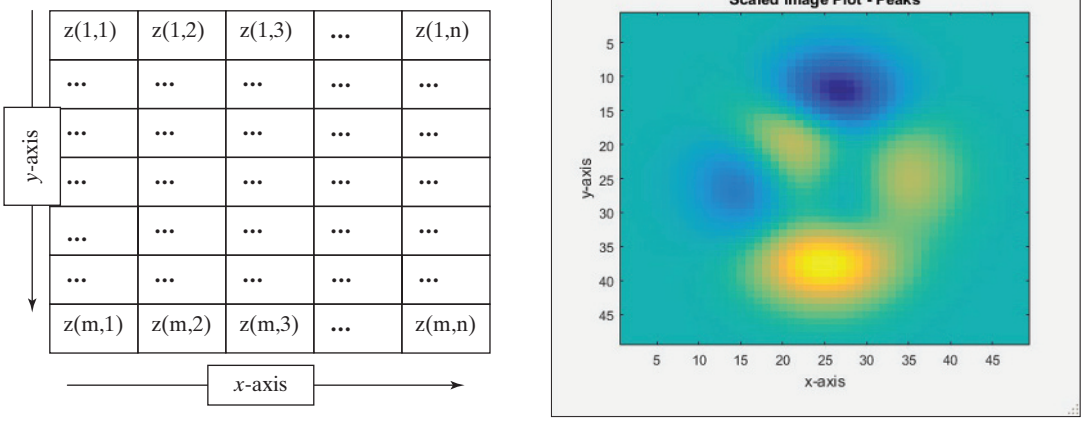


Figure 14.3
The `peaks` function rendered with the `imagesc` function. (a) Images are usually represented starting in the upper left-hand corner and working across and down, the way we read a book. (b) The `pcolor` plot and the `imagesc` plot are vertical mirror images of each other.

Intensity Images

We used an intensity image to create the representation of the `peaks` function (see Figure 14.3) with the scaled image function (`imagesc`). In this approach, the colors in the image are determined by a colormap. The values stored in the image array are scaled, and the values are correlated with a known map. (The `parula colormap` is the default.) This approach works well when the parameter being displayed does not correlate with an actual color. For example, the `peaks` function is often compared to a mountain and valley range—but what elevation is the color yellow? It's an arbitrary choice based partially on esthetics, but color maps can also be used to enhance features of interest in the image.

Consider this example: X-ray images traditionally were produced by exposing photographic film to X-ray radiation. Today, many X-rays are processed as digital images and stored in a data file—no film is involved. We can manipulate that file however we want, because the intensity of X-ray radiation does not correspond to a particular color.

MATLAB includes a sample file that is a digital X-ray photograph of a spine, suitable for display with the use of the scaled image function. First, you'll need to load the file:

```
load spine
```

The loaded file includes a number of arrays (see the workspace window); the intensity array is named `X`. Thus,

```
imagesc(X)
```

produces an image whose colors are determined by the current `colormap`, which defaults to `parula`. A representation that looks more like a traditional X-ray is returned if we use the `bone` colormap:

```
colormap(bone)
```

This image is shown in Figure 14.4.

The `spine` file also includes a custom colormap, which happens to correspond to the `bone` colormap. This array is called `map`. Custom colormaps are not necessary to display intensity images, and

```
colormap(map)
```

results in the same image we created earlier.

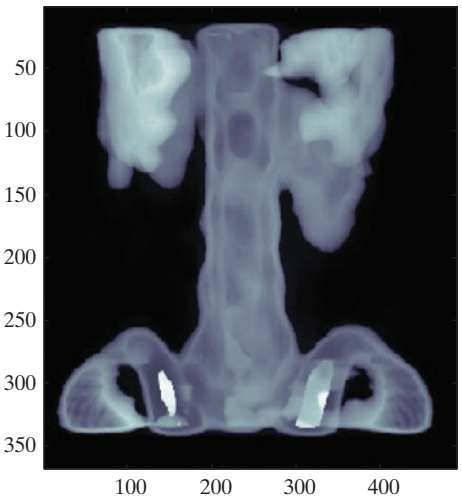


Figure 14.4
Digital X-ray displayed with the use of the `imagesc` function and the `bone` colormap. (Reprinted with permission from The Mathworks, Inc.)

Although it is convenient to think of image data as an array, such data are not necessarily stored that way in the standard graphics formats. MATLAB includes a function, `imfinfo`, that will read standard graphics files and determine what type of data are contained in the file. Consider the file *mimas.jpg*, which was downloaded off the Internet from a NASA website (http://saturn.jpl.nasa.gov). The command

```
imfinfo("mimas.jpg")
```

KEY IDEA
The color scheme for an image is controlled by the colormap.

returns the following information (be sure to list the file name in either single or double quotes—that is, as a string or as a character array; also, notice that the image is `'grayscale'`—another term for an intensity image):

```
ans =
  struct with fields:
              Filename: 'C:\Users\Holly\Documents\Gen…'
          FileModDate: '25-May-2013 11:42:23'
              FileSize: 5971
                Format: 'jpg'
         FormatVersion: ''
                 Width: 200
                Height: 210
              BitDepth: 8
             ColorType: 'grayscale'
       FormatSignature: ''
       NumberOfSamples: 1
          CodingMethod: 'Huffman'
        CodingProcess: 'Sequential'
               Comment: {'Created with The GIMP'}
```

In order to create a MATLAB array from this file, use the image read function `imread` and assign the results to a variable name, such as `X`:

```
X = imread("mimas.jpg");
```

We can then plot the image with the `imagesc` function and `gray` colormap:

```
imagesc(X)
colormap(gray)
```

The results are shown in Figure 14.5(a).

Indexed Image Function

When color is important, one technique for creating an image is called an indexed image. Instead of being a list of intensity values, the array is a list of colors. The image is created much like a paint-by-number painting. Each element contains a number that corresponds to a color. The colors are listed in a separate array called a *colormap*, which is an $n \times 3$ array that defines n different colors by identifying the red, green, and blue components of each color. A custom colormap can be created for each image, or a built-in colormap could be used.

Consider the built-in sample image of a mandrill, obtained with

```
load mandrill
```

The file includes an indexed array named `X` and a colormap named `map`. (Check the workspace window to confirm that these files have been loaded; the names are

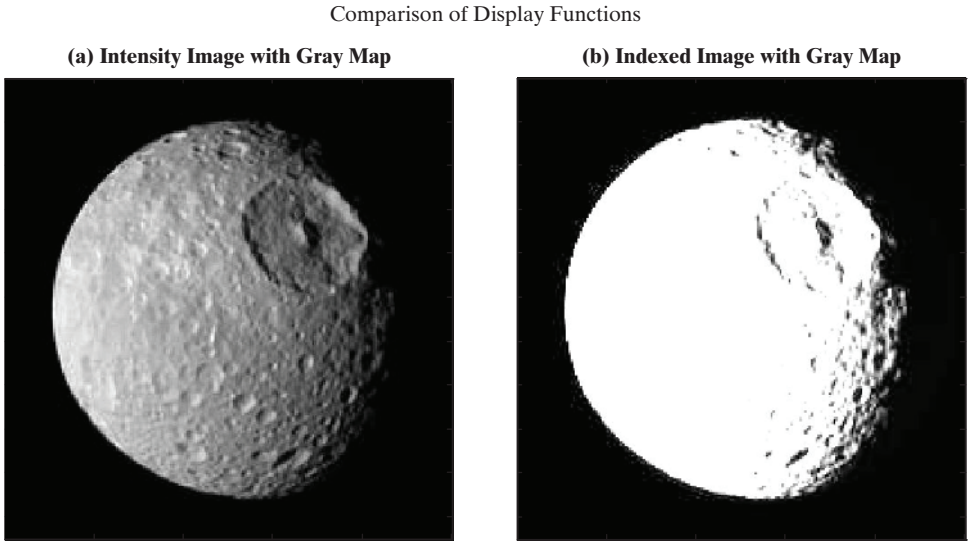


Figure 14.5
(a) Image of Mimas, a moon of Saturn, displayed by means of the scaled image function, `imagesc`, and a `gray` colormap. (b) Image displayed with the indexed image function, `image`, and a `gray` colormap. (NASA)

commonly used for images saved from a MATLAB program.) The `image` function is used to display indexed images:

```
image(X)
colormap(map)
```

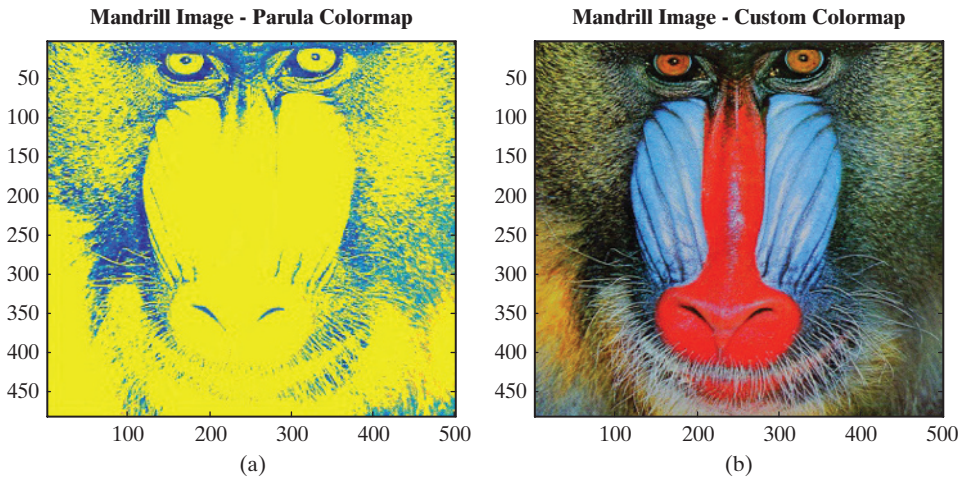
MATLAB images adjust to fill the figure window, so the image may appear warped. We can force the correct aspect to be displayed by using the `axis` command:

```
axis image
```

The results are shown in Figure 14.6.

The `image` and `imagesc` functions are similar, yet they can give very different results. The image of Mimas in Figure 14.5(b) was produced by the `image`

Figure 14.6
(a) Mandrill image before the custom colormap is applied. (b) Mandrill image with the custom color map. (Toni Angermayer/Science Source)



function instead of the more appropriate `imagesc` function. The `gray` colormap does not correspond to the colors stored in the intensity image; the result is the washed-out image and a lack of contrast. It is important to recognize what kind of file you are displaying, so that you can make the optimum choice of how to represent the image.

Files stored in the GIF graphics format are often stored as indexed images. This may not be apparent when you use the `imfinfo` function to determine the file parameters. For example, the image in Figure 14.7 is a gif image found on the Internet. The image was copied into the current folder, and `imfinfo` was used to determine the file type:

```
imfinfo("sunflower.gif")
ans =
  struct with fields:
              Filename: 'C:\Users\Holly\Documents\General
            FileModDate: '24-Aug-2016 12:33:17'
              FileSize: 342287
                Format: 'GIF'
         FormatVersion: '89a'
                  Left: 1
                   Top: 1
                 Width: 1024
                Height: 768
              BitDepth: 8
             ColorType: 'indexed'
       FormatSignature: 'GIF89a'
       BackgroundColor: 1
           AspectRatio: 0
            ColorTable: [256×3 double]
            Interlaced: 'no'
             DelayTime: 0
       TransparentColor: 253
         DisposalMethod:                                'DoNotspecify'
```



Figure 14.7
An image stored in the GIF file format. (Sawang Petvises/EyeEm/Getty Images)

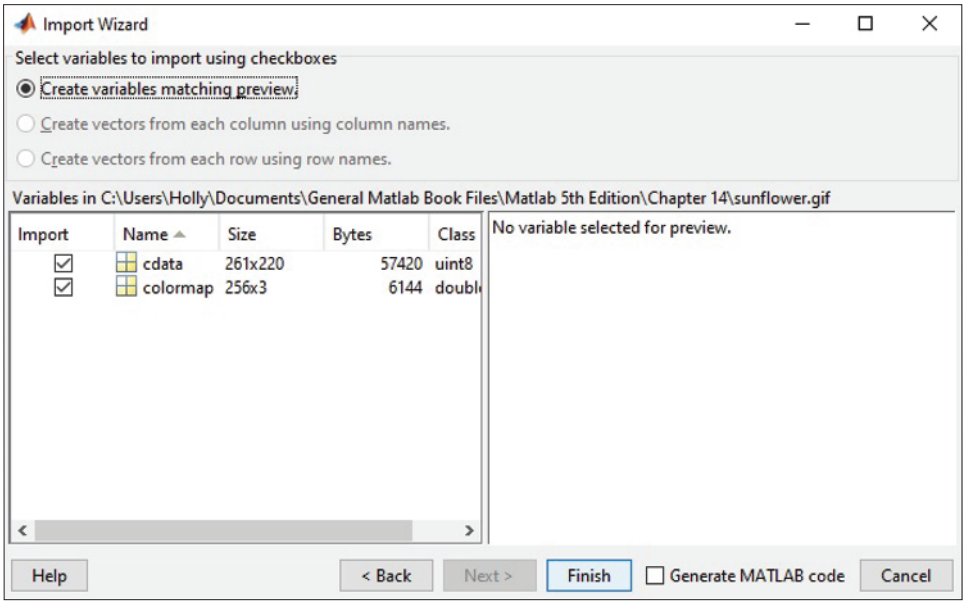


Figure 14.8
The Import Wizard is used to create an indexed image array and color map from a gif file.

The results don't tell us much, but if you double-click on the file name in the current folder, the Import Wizard (see Figure 14.8) launches and suggests that we create two arrays: `cdata` and `colormap`. The `cdata` array is an indexed image array and `colormap` is the corresponding `colormap`. Actually, the suggested name `colormap` is rather strange, because if we use it, it will supersede the `colormap` function. You'll need to rename this array to something different, such as `map`, by clicking on the variable name in the Import Wizard before you actually complete the import process. After importing, you can view the image with the following commands:

```
axis image
axis off
```

HINT

A number of sample images are built into MATLAB and stored as indexed images. You can access these files by typing

```
load <imagename>
```

Some of the available images are

```
flujet
durer
detail
mandrill
clown
spine
cape
earth
gatlin
```

Each of these image files creates an array of index values called `X` and a color map called `map`. For example, to see the image of the earth, type

```
load earth
image(X)
colormap(map)
```

You'll also need to adjust the aspect ratio of the display and remove the axis with the commands

```
axis image
axis off
```

True Color (RGB) Images

KEY IDEA
The primary colors of light are red, green, and blue.

The third technique for storing image data is in a three-dimensional array, $m \times n \times 3$. Recall that a three-dimensional array consists of rows, columns, and pages. **True color** image files consist of three pages, one for each color intensity, red, green, or blue, (RGB) as shown in Figure 14.9. The result is often called an **image plot**.

Consider a file called *airplanes.jpg*. You can copy this or a similar file (a colored .jpg image) into your current folder to experiment with true color images. We can use the `imfinfo` function to determine how the airplanes file stores the image:

```
imfinfo("airplanes.jpg")
ans =
  struct with fields:
Filename: 'airplanes.jpg'
FileModDate: '12-Sep-2005 17:51:48'
FileSize: 206397
Format: 'jpg'
FormatVersion: ''
Width: 1800
Height: 1200
BitDepth: 24
ColorType: 'truecolor'
FormatSignature: ''
NumberOfSamples: 3
CodingMethod: 'Huffman'
CodingProcess: 'Sequential'
Comment: {}
```

Notice that the color type is `'truecolor'` and that the number of samples is 3, indicating a page for each color intensity.

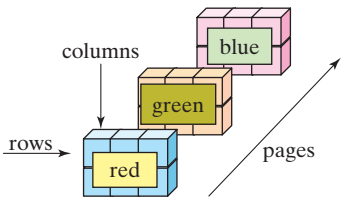


Figure 14.9
True color images use a multidimensional array to represent the color of each element.



Figure 14.10
True color image of airplanes. All of the color information is stored in a three-dimensional array. (Picture used with permission of Dr. G. Jimmy Chen, Salt Lake Community College, Department of Computer Science.)

We can load the image with the `imread` function and display it with the `image` function:

```
X = imread("airplanes.jpg");
image(X)
axis image
axis off
```

Notice in the workspace window that `X` is a $1200 \times 1800 \times 3$ array, one page for each color. We don't need to load a colormap, because the color-intensity information is included in the array (see Figure 14.10).

EXAMPLE 14.1

MANDELBROT AND JULIA SETS

Benoit Mandelbrot is largely responsible for the current interest in fractal geometry. His work built upon concepts developed by the French mathematician Gaston Julia in his 1919 paper *Mémoire sur l'iteration des fonctions rationelles.* Advances in Julia's work had to wait for the development of the computer and computer graphics in particular. In the 1970s, Mandelbrot, then at IBM, revisited and expanded upon Julia's work, and actually developed some of the first computer graphics programs to display the complicated and beautiful fractal patterns that today bear his name. Mandelbrot's work was recently described in a song by Jonathan Coulton. You can listen to it at http://www.jonathancoulton.com/store/downloads/, or search YouTube for a music video.

The Mandelbrot image is created by considering each point in the complex plane, $x + yi$. We set $z(0) = x + yi$, then iterate according to the following strategy:

$$\begin{aligned} z(0) &= x + yi \\ z(1) &= z(0)^2 + z(0) \\ z(2) &= z(1)^2 + z(0) \\ z(3) &= z(2)^2 + z(0) \\ z(n) &= z(n-1)^2 + z(0) \end{aligned}$$

The series seems either to converge or to head off toward infinity. The Mandelbrot set is composed of the points that converge. The beautiful pictures you have probably seen were created by counting how many iterations were necessary for the z-value at a particular point to exceed some threshold value, often the square root of 5. We assume, though we can't prove, that if that threshold is reached, the series will continue to diverge and eventually approach infinity.

1. State the problem
   Write a MATLAB program to display the Mandelbrot set.
2. Describe the input and output

   Input We know the Mandelbrot set lies somewhere in the complex plane, and that

   $$-1.5 \leq x \leq 1.0$$
   $$-1.5 \leq y \leq 1.5$$

   We also know that we can describe each point in the complex plane as

   $$z = x + yi$$

3. Develop a hand example
   Let's work the first few iterations for a point we hope converges, such as $(x = -0.5,\ y = 0)$

   $$z(0) = -0.5 + 0i$$
   $$z(1) = z(0)^2 + z(0) = (-0.5)^2 - 0.5 = 0.25 - 0.5 = -0.25$$
   $$z(2) = z(1)^2 + z(0) = (-0.25)^2 - 0.5 = 0.0625 - 0.5 = -0.4375$$
   $$z(3) = z(2)^2 + z(0) = (-0.4375)^2 - 0.5 = 0.1914 - 0.5 = -0.3086$$
   $$z(4) = z(3)^2 + z(0) = (-0.3086)^2 - 0.5 = 0.0952 - 0.5 = -0.4048$$

   It appears that this sequence is converging to a value around -0.4 (As an exercise, you could create a MATLAB program to calculate the first 20 terms of the series and plot them.)
4. Develop a MATLAB solution

```
%% Example 14.1 Mandelbrot Image
clear, clc, clf
iterations = 80;
grid_size = 500;
x=linspace(-1.5,1.0,grid_size);
y=linspace(-1.5,1.5,grid_size);
[x,y]=meshgrid(x,y);
c = x+i*y;
z = zeros(size(x));           % set the initial array to 0
map = zeros(size(x));         % create a map of all grid
                              % points equal to 0
%% For Loop
for k = 1:iterations
z = z.^2 +c;
a = find(abs(z)>sqrt(5));     % Determine which elements have
                              % exceeded sqrt(5)
```

(*continued*)

```
map(a) = k;
end
figure(1)
image(map)                          % Create an image
colormap(jet)
```

The image produced is shown in Figure 14.11.

5. Test the solution
We know that all the elements in the dark blue region of the image will be below the square root of 5. An alternative way to examine the results is to create an image based on those values instead of the number of iterations needed to exceed the threshold. We'll need to multiply each value by a common multiple in order to achieve any color variation. (Otherwise, the values are too close to each other.) The MATLAB code is as follows:

```
figure(2)
multiplier = 100;
map = abs(z)*multiplier;
image(map)
colormap(jet)
```

The results are shown in Figure 14.12.

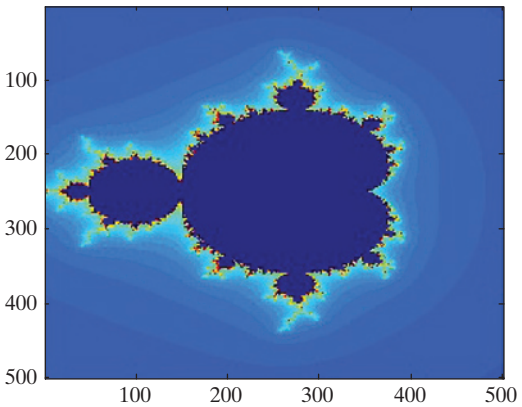


Figure 14.11
Mandelbrot image. The figure was created by determining how many iterations were required for the calculated element values to exceed the square root of 5.

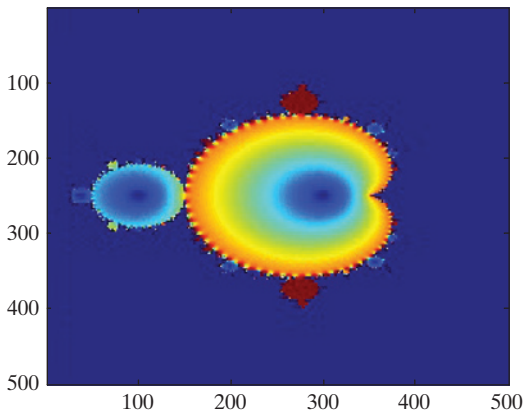


Figure 14.12
An image based on the Mandelbrot set, showing how the members of the set vary. The really interesting structure is at the boundary of the set.

Now that we've created an image of the entire Mandelbrot set, it would be interesting to look more closely at some of the structures at the boundary. By adding the following lines of code to the program, we can repeatedly zoom in on any point in the image:

```
while(cont==1)
figure(1)
disp('Now lets zoom in')
disp('Move the cursor to the upper left-hand corner of the area
you want to expand')
[y1,x1]=ginput(1);
disp('Move to the lower right-hand corner of the area you want
to expand')
[y1, x1] = ginput(1);
disp('Move to the lower right-hand corner of the area you
   want to expand')
[y2, x2] = ginput(1);
xx1 = x(round(x1),round(y1));
yy1 = y(round(x1),round(y1));
xx2 = x(round(x2),round(y2));
yy2 = y(round(x2),round(y2));
%%
[x, y] = meshgrid(linspace(xx1,xx2,grid_size),...
    linspace(yy1, yy2,grid_size));
c = x+i*y;
z = zeros(size(x));
map = zeros(size(x));
for k=1:iterations
    z=z.^2 +c;
    a=find(abs(z)>sqrt(5)& map==0);
    map(a)=k;
end
image(map)
colormap(jet)
again = menu('Do you want to zoom in again? ','Yes','No');
switch again
   case 1
     cont=1;
   case 2
     cont=0;
end
end
```

Figure 14.13 shows some of the images created by recalculating with smaller and smaller areas.

You can experiment with using both the `image` function and the `imagesc` function and observe how the pictures differ. Try some different colormaps as well.

(*continued*)

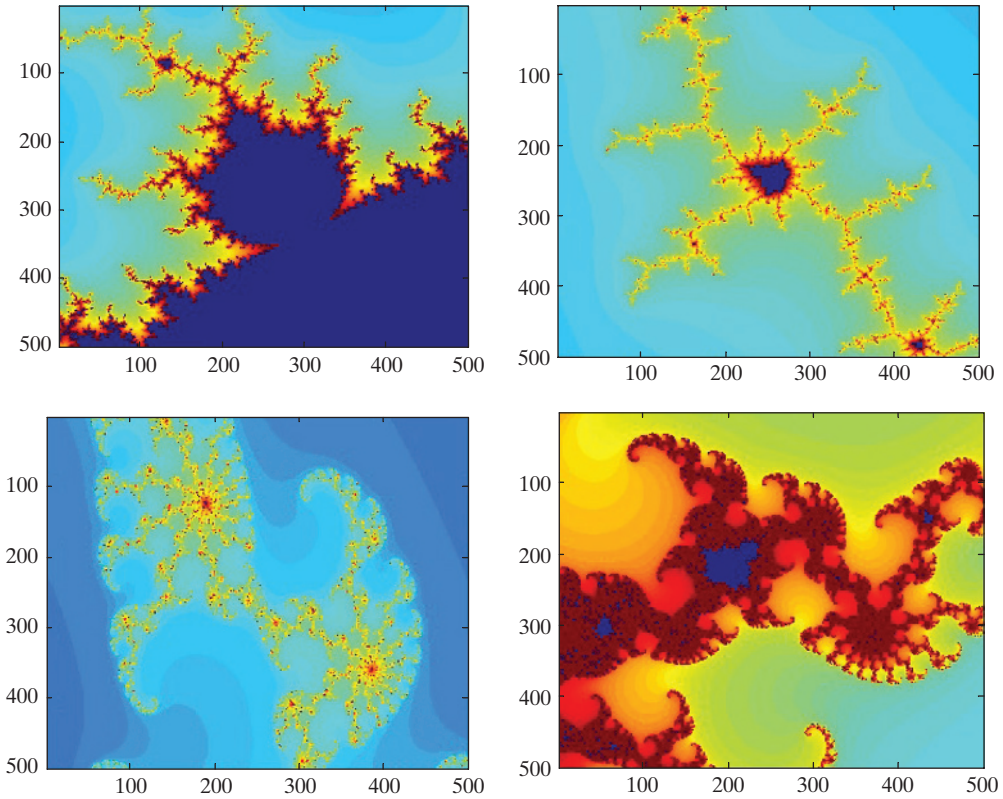


Figure 14.13
Images created by zooming in on the Mandelbrot set from a MATLAB program.

14.1.2 Reading and Writing Image Files

We introduced functions for reading image files as we explored the three techniques for storing image information. MATLAB also includes functions to write user-created images in any of a variety of formats. In this section, we'll explore these reading and writing functions in more detail.

Reading Image Information

Probably the easiest way to read image information into MATLAB is to take advantage of the interactive Import Wizard. In the current folder window, simply double-click the file name of the image to be imported. MATLAB will suggest appropriate variable names and will make the arrays available to preview in the edit window (see Figure 14.8).

The problem with interactively importing any data is that you can't include the instructions in a MATLAB program—for that, we need to use one of the import functions. For most of the standard image formats, such as .jpg or .tif, the `imread` function described in the preceding section is the appropriate technique. On the

other hand, if the file is a `.mat` or a `.dat` file, the easiest way to import the data is to use the `load` function:

```
load <filename>
```

For `.mat` files, you don't even need to include the `.mat` extension. However, you will need to include the extension for a `.dat` file:

```
load <filename.dat>
```

This is the technique we used to load the built-in image files described earlier. For example,

```
load cape
```

imports the image array and colormap into the current folder, and the commands

```
image(X)
colormap(map)
axis image
axis off
```

can then be used to create the picture, shown in Figure 14.14.

Storing Image Information

You can save an image you've created in MATLAB the same way you save any figure. First, make sure the figure window is active, and select

```
File → Save As ...
```

then choose the file type and the location where you'd like to save the image. For example, to save the image of the Mandelbrot set created in Example 14.1 and shown in Figure 14.11, you might want to specify a .jpg format as shown in Figure 14.15.

You could also save the file by using the `imwrite` function. This function accepts a number of different inputs, depending on the type of data you would like to store.

For example, if you have an intensity array (grayscale) or a true-color array (RGB), the `imwrite` function expects input of the form

```
imwrite(arrayname,"filename.format")
```

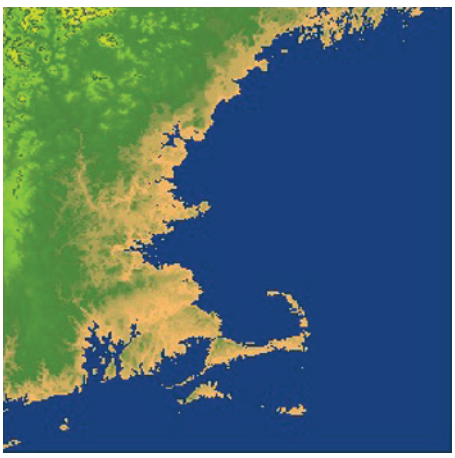


Figure 14.14
Image created by loading a built-in file. (NASA)

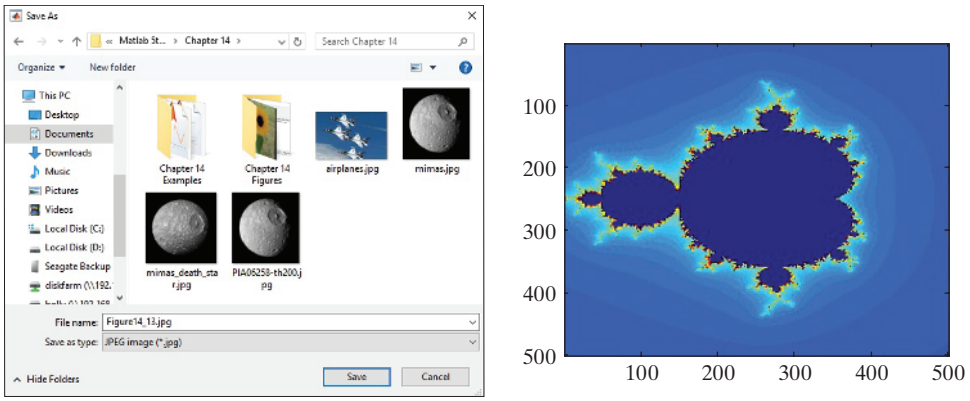


Figure 14.15
This image of a Mandelbrot set is being saved as a .jpg file.

where

`arrayname` is the name of the MATLAB array in which the data are stored, `filename` is the name you want to use to store the data, and `format` is the file extension, such as jpg or tif.

Thus, to store an RGB image in a .jpg file named flowers, the command would be

```
imwrite(X,"flowers.jpg")
```

(Consult the help files for a list of graphics formats supported by MATLAB.)

If you have an indexed image (an image with a custom colormap), you'll need to store both the array and the colormap:

```
imwrite(arrayname, colormap_name,"filename.format")
```

In the case of the Mandelbrot set, we would need to save the array and the colormap used to select the colors in the image:

```
imwrite(map,jet,"my_mandelbrot.jpg")
```

14.2 GRAPHICS OBJECTS

KEY IDEA
A handle is a nickname given to an object in MATLAB.

A **handle** is a nickname given to an **object** in MATLAB. A complete description of the graphics system used in MATLAB is complicated and beyond the scope of this text. (For more details, refer to the MATLAB help tutorial.) However, we'll give a brief introduction to assigning handles to graphics objects and then illustrate some of the uses of this strategy.

MATLAB uses a hierarchical system for creating graphs (see Figure 14.16). The basic plotting object is the figure. The figure can contain a number of different objects, including a set of axes. Think of the axes as being layered on top of the figure window. The axes can also contain a number of different objects, including a plot such as the one shown in Figure 14.17. Again, think of the plot being layered on top of the axes.

When you use a `plot` function, either from the command window or from a program, MATLAB automatically creates a figure and appropriate axes, then draws the graph on the axes. MATLAB uses default values for many of the plotted object's properties. For example, the first line drawn is always blue, unless the user specifically changes it.

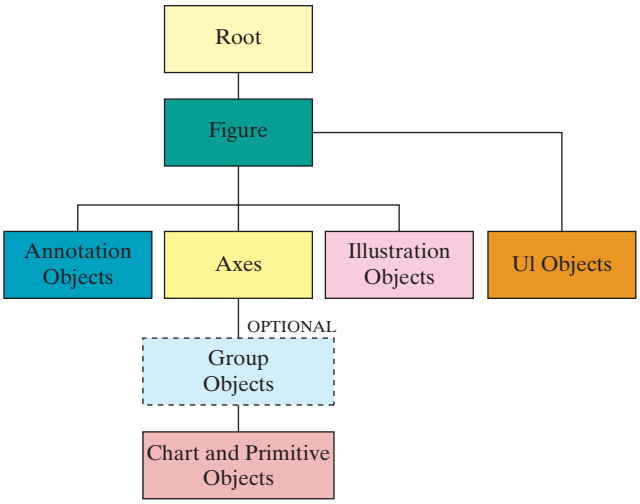


Figure 14.16
MATLAB uses a hierarchical system for organizing plotting information, as shown in this representation from MATLAB's help feature.

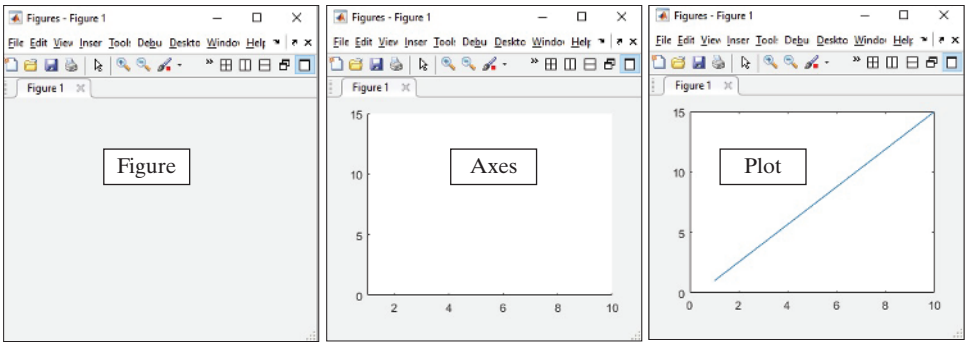


Figure 14.17
Anatomy of a graph. (a) Figure windows are used for many things, including graphical user interfaces and plots. In order to create a plot you need a figure window. (b) Before you can draw a graph in this figure window, you need a set of axes to draw on. (c) Once you know where the axes are and what the axis properties are (such as the spacing), you can draw the graph.

14.2.1 Plot Handles

Assigning a plot a name (called a *handle*) allows us to easily ask MATLAB to list the plotted object's properties. For example, let's create the simple plot shown in Figure 14.18(a) and assign a handle to it:

KEY IDEA
MATLAB functions such as title and `xlabel` will accept either character arrays or string arrays as input.

```
clear, clc, clf
x=linspace(-10,10);
y=cos(x);
h=plot(x,y)
```

The title and axis labels were added in the usual way.

```
title("Trigonometric Function Plot")
xlabel("angle, radians"), ylabel("cos(x)")
```

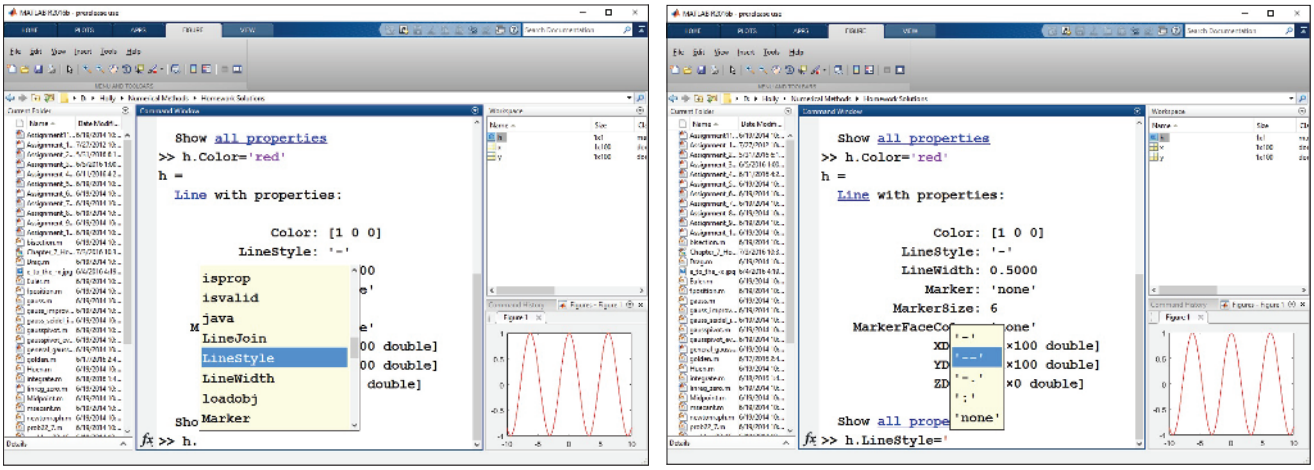
The variable `h` is the handle for the plot. (We could have chosen any variable name.) It is listed in the workspace window as a `matlab.graphics.chart.primitive.Line`, and stores all of the properties of the line. Since we didn't suppress the output with a semicolon, when the code is executed it returns an abbreviated list

of the plot properties with a link to the remaining properties. Older MATLAB code (before release 2014b) required the user to use the `get` function to ask MATLAB for the plot properties:

```
get(h)
```

The result is a more complete list of properties representing the line that was drawn in the axes.

```
(h)
    AlignVertexCenters: off
            Annotation: [1×1 matlab.graphics.eventdata.
Annotation]
          BeingDeleted: off
            BusyAction: 'queue'
         ButtonDownFcn: ''
              Children: [0×0 GraphicsPlaceholder]
              Clipping: on
                 Color: [0 0.447 0.741]
             ColorMode: 'auto'
           ContextMenu: [0×0 GraphicsPlaceholder]
             CreateFcn: ''
       DataTipTemplate: [1×1 matlab.graphics.datatip.
DataTipTemplate]
             DeleteFcn: ''
           DisplayName: ''
      HandleVisibility: 'on'
               HitTest: on
         Interruptible: on
              LineJoin: 'round'
             LineStyle: '-'
         LineStyleMode: 'auto'
             LineWidth: 0.5
                Marker: 'none'
       MarkerEdgeColor: 'auto'
       MarkerFaceColor: 'none'
         MarkerIndices: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 … ]
            MarkerMode: 'auto'
            MarkerSize: 6
                Parent: [1×1 Axes]
         PickableParts: 'visible'
              Selected: off
    SelectionHighlight: on
           SeriesIndex: 1
                   Tag: ''
                  Type: 'line'
              UserData: []
               Visible: on
                 XData: [-10 -9.798 -9.596 -9.3939 -9.1919
-8.9899 -8.7879 -8.5859 ... ]
            XDataMode: 'manual'
           XDataSource: ''
                 YData: [-0.83907 -0.93116 -0.98538 -0.99952
```



(a) (b)

Figure 14.18
Adding a handle to graphics objects can be used to adjust the plot, figure, and axis properties. Using dot notation a list of completion choices is accessed with the tab key. Completions exist for both (a) the property name and for (b) acceptable property values.

```
-0.97301 -0.90692 … ]
            YDataSource: ''
                  ZData: [1×0 double]
            ZDataSource: ''
```

The plot handle, h, refers to the line drawn on the axis, which is different from the axes or from the figure window. You can change the properties of the graphics object with either dot notation, or for versions of MATLAB before release 2014b, with the `set` function. For a line with the handle `h`, you can change the line color to red by typing either

```
set(h, 'Color', 'red'),
set(h,Color='red')  or
h.Color='red'
```

One advantage of the dot notation is the availability of the completion tab. For example, once you've created the plot handle `h`, if you start typing

```
h.
```

a list of possible completions appears, as shown in Figure 14.18(a). Double-click on your choice of properties to add it to the line of code. You can also use completions to see a list of possible property values as shown in Figure 14.18(b).

HINT

You can narrow down the list of possible property values by typing in the first letter of the property of interest. For example, if `h` is the plot handle enter

```
h.l
```

followed by the tab key to narrow the choices down to properties that start with the letter 'l'.

The completion function works slightly differently in scripts and live scripts. Experiment with both to see which you prefer.

14.2.2 Figure Handles

We can also specify an object name (handle) for the figure window. Since we drew this graph in the figure window named Figure 1, the command would be

```
f_handle = figure(1)
```

The handle is listed in the workspace window as a `matlab.ui.Figure`. Using the `get` command returns similar results. Recall that using `get` is not necessary unless you are working with older code or an older version of MATLAB.

```
get(f_handle)
f_handle =
  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [1 1 390 402]
       Units: 'pixels'
```

Notice that the properties are different for a figure window compared with the plotted line. For example, notice the color (which is the window background color) is [0.9400, 0.9400, 0.9400], which specifies equal intensities of red, green, and blue—which results in a light gray background. You can change the background color using either dot notation or the set function. For example, either

```
f_handle.Color=[0.4,0.4,0.4] or
set(f_handle,'Color',[0.4,0.4,0.4]) or
set(f_handle,Color=[0.4,0.4,0.4])
```

results in a darker gray background.

Similarly, you could change the name of the figure window.

```
f_handle.Name = "My Graph"
```

or

```
set(f_handle,Name="My Graph")
```

If we haven't specified a handle name, we can ask MATLAB to determine the current figure with the `gcf` (get current figure) command,

```
get(gcf)
```

which returns the figure properties. Thus, using `gcf` and the `set` command, we could change the background color back to default with the following command:

```
set(gcf,'Color',[0.94,0.94,0.94])
```

14.2.3 Axes Handles

Just as we can assign an object name to the figure window and the plot itself, we can assign an object name (handle) to the axes by means of the `gca` (get current axes) function:

```
h_axes = gca
```

This command returns the axes properties, but you can also use the handle (in this case `h_axes`) with the `get` command to view the properties:

```
get(h_axes)
                      ALim: [0 1]
                  ALimMode: 'auto'
      ActivePositionProperty: 'outerposition'
           AmbientLightColor: [1 1 1]
                BeingDeleted: 'off'
                         Box: 'on'
                   BoxStyle: 'back'
                 BusyAction: 'queue'
               ButtonDownFcn: ''
                        CLim: [0 1]
                   CLimMode: 'auto'
              CameraPosition: [5 7.5000 17.3205]
          CameraPositionMode: 'auto'
                      CameraTarget: [5 7.5000 0]
```

Just as with the plot handle and the figure handle, you can use the axis handle to change the properties. This is especially useful for creating custom axis labels. Consider the plot shown in Figure 14.18. The *x*-axis represents an angle in radians. It is often preferable to represent the angles as multiples of pi. We can accomplish this by assigning and using an axis handle with the `XTick` and `YTick` properties to position the tick marks, and the `XTick` property to label them appropriately. Notice that a backslash `(\)` is used to convert the word pi to the symbol π.

```
h_axes.XTick = [-3*pi, -2*pi, -pi, 0, pi, 2*pi, 3*pi]
h_axes.YTick = [-1, -0.5, 0, 0.5, 1]
h_axes.XTickLabel = ["-3\pi","-2\pi","-\pi","0","\pi",...
    "2\pi","3\pi"];
h_axes.YTickLabel = ["min = -1","-0.5","0","0.5","max = 1"];
```

Notice that a string array was used for the axis labels, where in the past a cell array of character arrays would have been suggested.

Finally, the labels would look better if rotated

```
h_axes.XTickLabelRotation = 45;
h_axes.YTickLabelRotation = 45;
```

Notice that just as with the plot handle and the figure handle, dot notation is available to change the axis properties; however, the `set` function could also be used.

The results are shown in Figure 14.19(a). (The title and axis labels were added in the usual way). This example is based on one located in the MATLAB help documentation. There are multiple examples of techniques there that you can use to improve your graphs.

Here's another example. Consider test scores for a group of students. The results could be shown as a bar graph, but it would be nice if the *x*-axis were labeled with names instead of numbers. To create the graph, first create an array of names and an array of scores.

```
names = ["Thao","Phan","Fred","Olive","Rose","John"];
scores =[100;98;85;79;82;62];
```

Use the `bar` function to create the graph. Create a handle to the axes with the get current axes function.

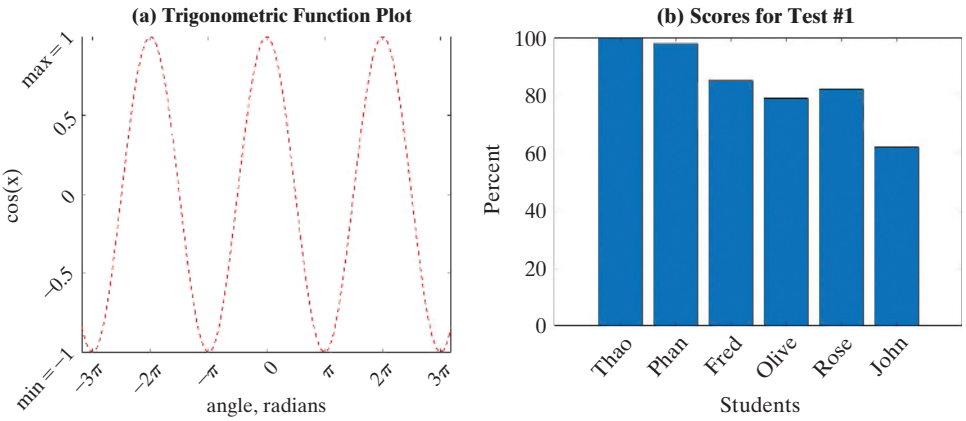


Figure 14.19
(a) Object names (handles) can be used to adjust the appearance of any graph created in MATLAB.
(b) Names are added to a bar graph using the axis object name and dot notation.

```
bar(scores)
ax = gca;
```

Now use the axes handle to add the names to the *x*-axis and to rotate the names so they can be read easily.

```
ax.XTickLabel=names;
ax.XTickLabelRotation=45;
```

The font size and weight can be adjusted similarly.

```
ax.FontSize=14;
ax.FontWeight="bold";
```

Finally, the title and axis labels were added in the usual way.

```
title("(b) Scores for Test #1")
xlabel("Students"), ylabel("Percent")
```

The results are shown in Figure 14.19(b).

14.2.4 Annotation Axes

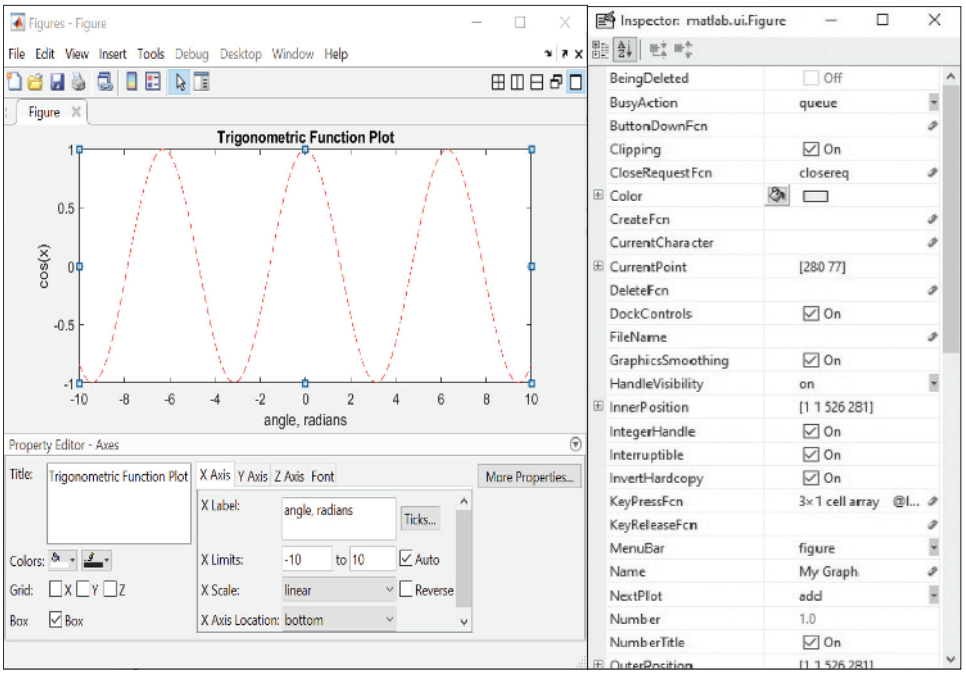
Besides the three components described in earlier sections, another transparent layer is added to the plot. This layer is used to insert annotation objects, such as lines, legends, and text boxes into the figure. The properties of these annotation objects can be controlled by assigning a handle, in a manner similar to the one used for figures, plots, and axes.

14.2.5 Property Editor

Properties can be adjusted interactively by selecting `View` from the figure menu bar, and choosing the property editor (see Figure 14.20(a)):

```
View → Property Editor
```

You can access all the properties if you choose property inspector from the property editor pop-up window (see Figure 14.20(b)). Exploring the property inspector window is a great way to find out which properties are available for each graphics object.



(a) (b)

Figure 14.20
Interactive property editing. (a) Edit a MATLAB figure by selecting **View → Property Editor**. (b) More options are available from the More Properties menu which opens the property inspector.

14.3 ANIMATION

There are two techniques for creating an animation in MATLAB:

- Redrawing and erasing
- Creating a movie

We create object names (handles) in each case to help create the animation.

14.3.1 Redrawing and Erasing

To create an animation by redrawing and erasing, you should first create a plot and then adjust the properties of the graph each time through a loop. Consider the following example: We can define a set of parabolas with the following equation:

$$y = kx^2 - 2$$

Each value of k defines a different parabola. We could represent the data with a three-dimensional plot; however, another approach would be to create an animation in which we draw a series of graphs, each with a different value of k. The code to create that animation is

```
clear, clc, clf
x = -10:0.01:10;      % Define the x-values
k = -1;               % Set an initial value of k
y = k*x.^2-2;         % Calculate the first set of y-values
h = plot(x,y);        % Create the figure and assign
                      % a handle to the graph
grid on
axis([-10,10,-100,100])          % Specify the axes
```

```
while k<1                          % Start a loop
    k = k + 0.01;                  % Increment k
    y = k*x.^2-2;                  % Recalculate y
    h.YData = y;                   % Reassign the y
                                       % values used in the graph
    drawnow                    % Redraw the graph now - don't wait
                               % until the program finishes running
end
```

In this example, we redrew just the graph each time through the loop, instead of creating a new figure window each time. We used the `YData` object from the plot, which assigns the *y* data points to be plotted (The *x* values stay the same). Using the `set` function allows us to specify new *y*-values and to create a different graph every time the `drawnow` function is called. A selection of the frames created by the program and used in the animation is shown in Figure 14.21.

A more complicated kind of animation you may want to use involves moving a single object, but keeping the remainder of the figure static. Consider modeling a ball rolling up and over a hill. The hill can be modeled as a portion of the sine function. First, we'll need to create a plot of the sine function, which will remain static during the animation

```
clear, clc
theta = linspace(-pi/2,pi/2,1000);  % Define 'hill' parameters
x=linspace(0,20,1000);
y=20*cos(theta);
plot(x,y)                          % Create the plot of the sine wave
axis([-1,21,-1,21])  % Specify the axis
title("A Ball on a Hill")
hold  on
```

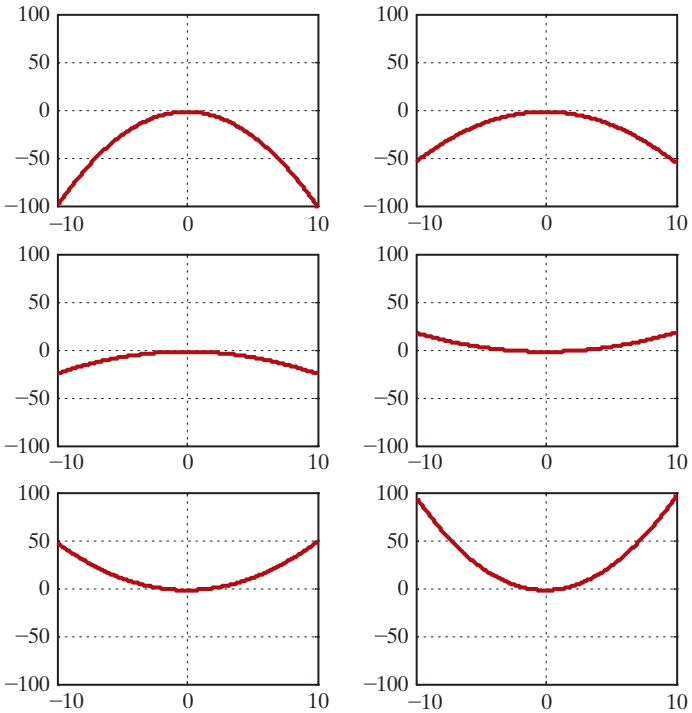


Figure 14.21
Animation works by redrawing the graph multiple times.

The next step is to create a circular object, which will move during the animation. The circle is created using the patch function. Notice in the workspace window that `t` is listed as a `matlab.graphics.primitive.Patch`.

```
theta2 = linspace(-pi,pi);  % Define parameters used to draw the
% circle
xc = cos(theta2);
yc = -sin(theta2);
t=patch(xc,yc,'red');    % Create a circular object
hold  off
```

Finally create a loop. Each time through the loop the circle will be redrawn

```
%%
for j = 1:length(theta)
    xnew = xc + x(j);  % define new circle position
    ynew = yc + y(j);
    t.XData = xnew; % update data properties
    t.YData = ynew;
    drawnow  % display updates
end
```

When you run this code, the red circle moves up and over the 'hill' represented by the sine function, as illustrated in Figure 14.22. If you are working in a live script the animation renders with play controls that allow the user to replay the result at different speeds and to export the animation in a variety of common video formats, including mp4 and avi.

14.3.2 Movies

Animating the motion of a line is not computationally intensive, and it's easy to get nice, smooth movement. Consider this code that produces a more complicated surface plot animation:

```
clear, clc
x = 0:pi/100:4*pi;
y = x;
[X, Y] = meshgrid(x,y);
```

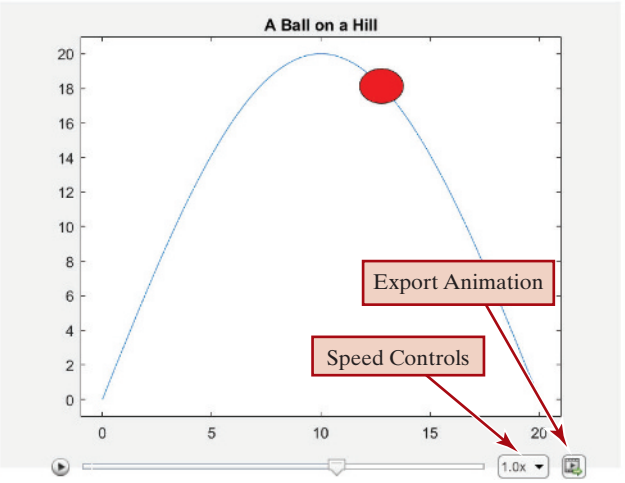


Figure 14.22
MATLAB allows the animation of a graphics object over a static background. Creating an animation in a Live Script generates playback controls and the ability to export the animation in a variety of common video formats.

```
z = 3*sin(X)+ cos(Y);
h = surf(x,y,z);
title("Waves in a Pond")
axis([0,4*pi,0,4*pi,-3,3])
shading interp
colormap(jet)
for k = 0:pi/100:2*pi
  z = (sin(X) + cos(Y)).*sin(k);
  h.Zdata = z;
  drawnow
end
```

A sample frame from this animation is shown in Figure 14.23.

If you have a fast computer, the animation may still be smooth. However, on a slower computer, you may see jerky motion and pauses while the program creates each new plot. To avoid this problem, you can create a program that captures each "frame" then, once all the calculations are done, plays the frames as a movie.

```
clear, clc
x = 0:pi/100:4*pi;
y = x;
[X, Y] = meshgrid(x,y);
z = 3*sin(X)+ cos(Y);
h = surf(x,y,z);
axis([0,4*pi,0,4*pi,-3,3])
shading interp
colormap(jet)
m = 1;
for k = 0:pi/100:2*pi
z = (sin(X) + cos(Y)).*sin(k);
```

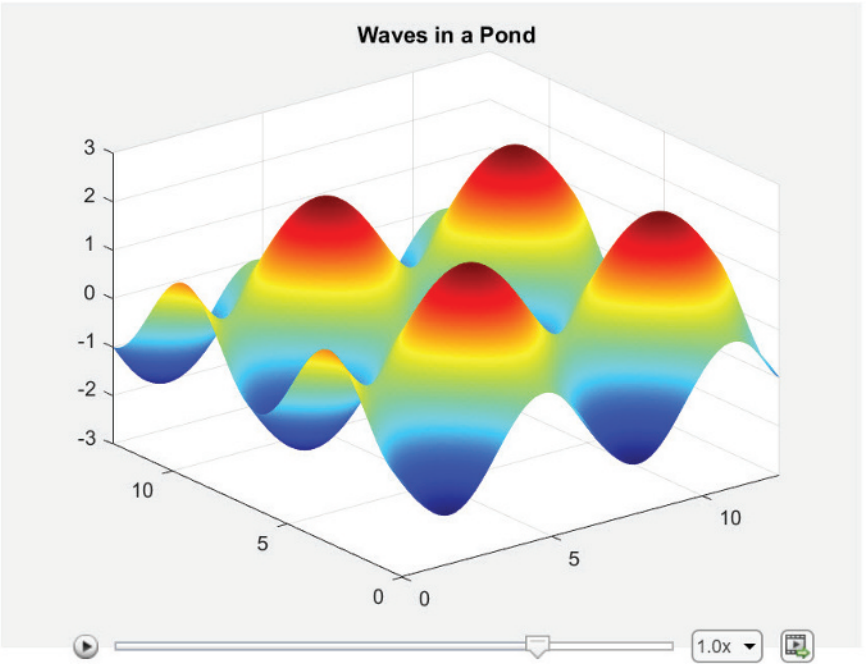
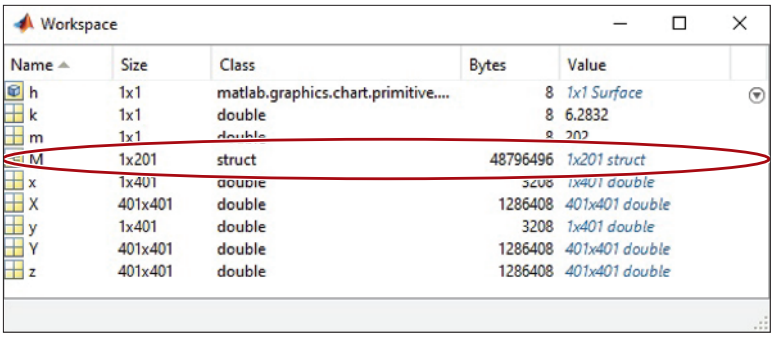


Figure 14.23
The animation of this figure moves up and down, and looks like waves in a pond.

Figure 14.24
Movies are saved in a structure array, such as the `M` array shown in this figure.




```
h.ZData = z;
M(m) = getframe; % Creates and saves each frame
% of the movie
m = m+1;
end
movie(M,2) % Plays the movie twice
```

KEY IDEA
Movies record an animation for later playback.

When you run this program, you will actually see the movie three times: once as it is created, and the two times specified in the movie function. (In earlier versions of MATLAB, the movie would have played one additional time as the animation was loaded.) One advantage of this approach is that you can play the movie again without redoing the calculations, since the information is stored (in our example) in the array named `M`. Notice in the workspace window (see Figure 14.24) that `M` is a moderately large structure array (~49 MB).

EXAMPLE 14.2

A MANDELBROT MOVIE

The calculations required to create a Mandelbrot image require significant computer resources and can take several minutes. If we want to zoom in on a point in a Mandelbrot image, a logical choice is to do the calculations and create a movie, which we can view later. In this example, we start with the MATLAB program first described in Example 14.1 and create a 100-frame movie.

1. State the problem
   Create a movie by zooming in on a Mandelbrot set.

2. Describe the input and output

   Input The complete Mandelbrot image described in Example 14.1

   Output A 100-frame movie

3. Develop a hand example
   A hand example doesn't make sense for this problem, but what we can do is create a program with a small number of iterations and elements to test our solution, then use it to create a more detailed sequence that is more computationally intensive. Here is the first program:

```
%% Example 14.2 Mandelbrot Image
% The first part of this program is the same as Example 14.1
clear, clc, clf
```

(*continued*)

```
iterations=20; % Limit the number of iterations in this first
% pass
grid_size = 50; % Use a small grid to make the program run
% faster
X=linspace(-1.5,1.0,grid_size);
Y=linspace(-1.5,1.5,grid_size);
[x,y]=meshgrid(X,Y);
c = x+i*y;
z=zeros(size(x));
map=zeros(size(x));
%%
for k=1:iterations
    z=z.^2 +c;
    a=find(abs(z)>sqrt(5)& map==0);
    map(a)=k;
end
%%
figure(1)
h=imagesc(map)

%% New code section

N(1)=getframe; % Get the first frame of the movie
disp("Now lets zoom in")
disp("Move the cursor to a point where you'd like to zoom")
[y1,x1]=ginput(1) % Select the point to zoom in on

xx1=x(round(x1),round(y1))
yy1=y(round(fx1),round(y1))
%%
for k=2:60 % Calculate and display the new images
   k % Send the iteration number to the command window
[x,y]=meshgrid(linspace(xx1-1/1.1^k,xx1+1/1.1^k,...
    grid_size),linspace(yy1-1/1.1^k,yy1+1/1.1^k,grid_size));
c = x+i*y;
z=zeros(size(x));
map=zeros(size(x));
for j=1:iterations
    z=z.^2 +c;
    a=find(abs(z)>sqrt(5) & map==0);
    map(a)=j;
end
%%
h.CData = map;% Retrieve the image data from the variable map
colormap(jet)
N(k)=getframe; % Capture the current frame
end
movie(N,2) % Play the movie twice
```

This version of the program runs quickly and returns low-resolution images (see Figure 14.25), which demonstrate that the program works.

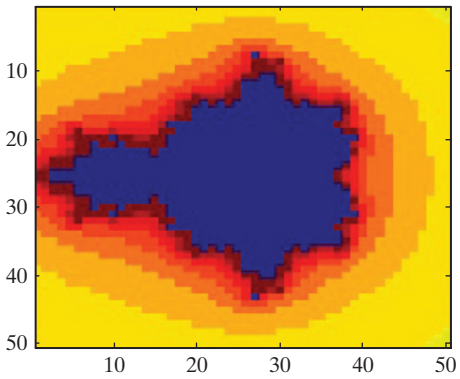


Figure 14.25
Low resolution Mandelbrot image used to test the animation program.

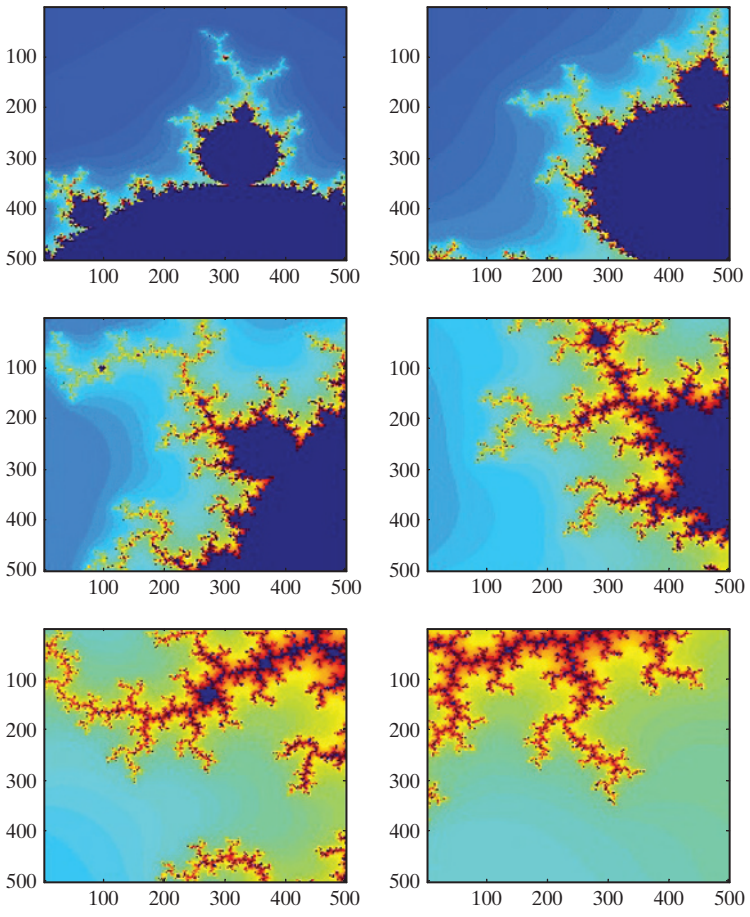


Figure 14.26
This series of Mandelbrot images is a selection of the frames captured to create a movie with the program in this example. Each movie will be different, since it zooms in on a different point of the image.

4. Develop a MATLAB solution

The final version of the program is created by changing just two lines of code:

```
iterations = 80;     % Increase the number of iterations
grid_size = 500;     % Use a large grid to see more detail
```

This "full-up" version of the program took approximately 40 seconds to run on a 3.0-GHz AMD dual-core processor with 2.0 GB of RAM. Selected frames are shown in Figure 14.26. Of course, the time it takes on your computer will

(*continued*)

be more or less, depending on your system resources. One cycle of the movie created by the program plays in about 10 seconds.

5. Test the solution
   Try the program several times, and observe the images created when you zoom in to different portions of the Mandelbrot set. You can experiment with increasing the number of iterations used to create the image and with the colormap.

14.4 OTHER VISUALIZATION TECHNIQUES

14.4.1 Transparency

When we render surfaces in MATLAB, we use an opaque coloring scheme. This approach is great for many surfaces, but can obscure details in others. Take, for example, this series of commands that creates two spheres, one inside the other:

```
clear, clc, clf % Clear the command window and current
% figure window
n = 20; % Define the surface of a sphere,
% using spherical coordinates
Theta = linspace(-pi,pi,n);
Phi = linspace(-pi/2,pi/2,n);
[theta,phi] = meshgrid(Theta,Phi);
X = cos(phi).*cos(theta); % Translate into the xyz
% coordinate system
Y = cos(phi).*sin(theta);
Z = sin(phi);
surf(X,Y,Z) % Create a surface plot of a sphere of radius 1
axis square
axis([-2,2,-2,2,-2,2]) % Specify the axis size
hold on
pause (2) % Pause the program
surf(2*X,2*Y,2*Z) % Add a second sphere of radius 2
pause (2) % Pause the program
alpha(0.5) % Set the transparency level
```

The interior sphere is hidden by the outer sphere until we issue the transparency command,

```
alpha(0.5)
```

which sets the transparency level. A value of 1 corresponds to opaque, and 0 to completely transparent. The results are shown in Figure 14.27. Transparency can be added to surfaces, images, and patch objects.

The command `alpha(0.5)` sets the transparency for all objects plotted on the axis. We can use handle graphics to specify the transparency for specific graphical objects. For example, first clear the figure window, but NOT the workspace window.

```
clf
```

Assign a handle to each of the surface plots

```
h1 = surf(X,Y,Z);
hold on
h2=surf(2*X, 2*Y,2*Z);
```

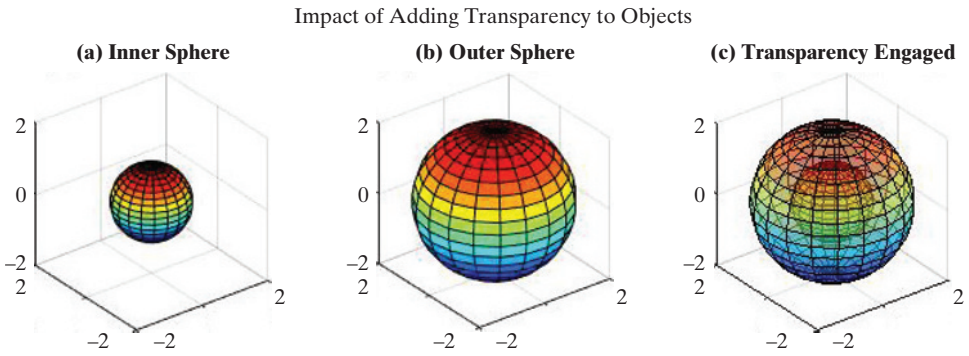


Figure 14.27
Adding transparency to a surface plot makes it possible to see hidden details.

To change the transparency of the outer sphere

```
h2.FaceAlpha=0.3
```

14.4.2 Hidden Lines

When mesh plots are created, any part of the surface that is obscured is not drawn. Usually, this makes the plot easier to interpret. The two spheres shown in Figure 14.28 were created with the use of the *X*-, *Y*-, and *Z*-coordinates calculated in the preceding section. Here are the MATLAB commands:

```
figure(3)
tiledlayout(1,2)
nexttile
mesh(X,Y,Z)
title("(a) Mesh Lines Hidden")
axis square
nexttile
mesh(X,Y,Z)
axis square
hidden off
title("(b) Mesh Lines Revealed")
```

The default value for the `hidden` command is `on`, which results in mesh plots in which the obscured lines are automatically hidden, as shown in Figure 14.28(a). Issuing the `hidden off` command gives the results as shown in Figure 14.28(b).

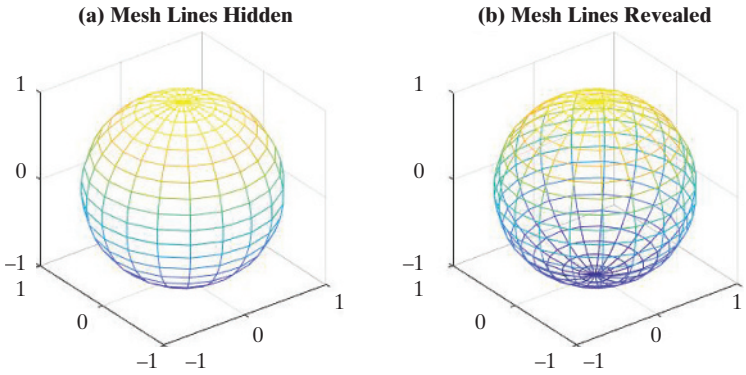


Figure 14.28
(a) Mesh plots do not show mesh lines that would be obscured by a solid figure. (b) The `hidden off` command forces the program to draw the hidden lines.

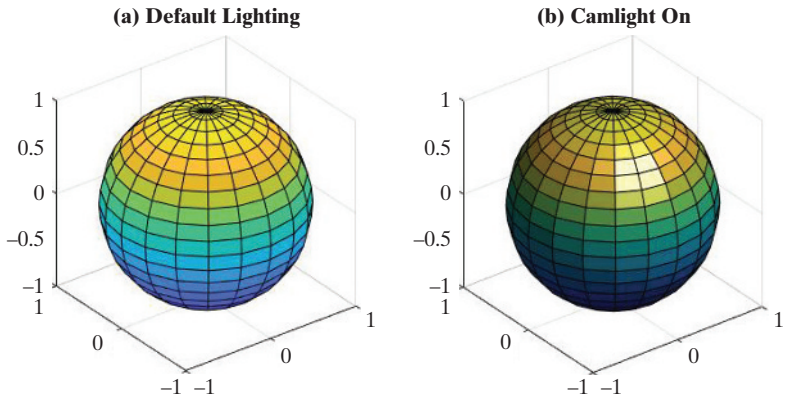


Figure 14.29
(a) The default lighting is diffuse. (b) When the `camlight` command is issued, a spotlight is modeled, located at the camera position.

KEY IDEA
The camlight allows you to adjust the figure lighting.

14.4.3 Lighting

MATLAB includes extensive techniques for manipulating the lighting used to represent **surface plots**. The position of the virtual light can be changed and even be manipulated during animations. The figure toolbar includes icons that allow you to adjust the lighting interactively, so that you can get just the effect you want. However, most graphs really need the lighting only turned on or off, which is accomplished with the `camlight` function. (The default is off.) Figure 14.29 shows the results achieved when the `camlight` is turned onto a simple sphere. The code to use is

```
sphere
camlight
```

The default position for the camlight is up and to the right of the "camera." The choices include the following:

`camlight right`	up and to the right of the camera (the default)
`camlight left`	up and to the left of the camera
`camlight headlight`	positioned on the camera
`camlight(azimuth,elevation)`	lets you determine the position of the light
`camlight('infinite')`	models a light source located at infinity (such as the sun)

14.5 INTRODUCTION TO VOLUME VISUALIZATION

MATLAB includes a number of visualization techniques that allow us to analyze data collected in three dimensions, such as wind speeds measured at a number of locations and elevations. It also lets us visualize the results of calculations performed with three variables, such as $y = f(x, y, z)$. These visualization techniques fall into two categories:

- **Volume visualization** of **scalar data** (where the data collected or calculated is a single value at each point such as temperature).
- Volume visualization of **vector data** (where the data collected or calculated is a vector, such as **velocity**).

14.5.1 Volume Visualization of Scalar Data

In order to work with scalar data in three dimensions, we need four three-dimensional arrays:

- X data, a three-dimensional array containing the *x*-coordinate of each grid point
- Y data, a three-dimensional array containing the *y*-coordinate of each grid point
- Z data, a three-dimensional array containing the *z*-coordinate of each grid point
- Scalar values associated with each grid point—for example, a temperature or pressure

The *x*, *y*, and *z* arrays are often created with the `meshgrid` function. For example, we might have

```
x = 1:3;
y = [2,4,6,8];
z = [10, 20];
[X,Y,Z] = meshgrid(x,y,z);
```

The calculations produce three arrays that are $4 \times 3 \times 2$ and define the location of every grid point. The fourth array required is the same size and contains the measured data or the calculated values. For example, the fourth array might contain temperature measurements at each location in the field, or scalar values of speed.

Isosurfaces

MATLAB contains a program file called `flow` which contains information in this format. You can call it to experiment with the data with the following code:

```
[x,y,z,v]=flow;
```

Notice in the workspace window that four new $25 \times 50 \times 25$ arrays are created. The *x*, *y*, and *z* arrays are the location information, and the *v* array represents the speed profile of a submerged jet in an infinite tank. You can examine the array values in the array editor, but this doesn't really tell you very much. The `max` and `min` functions give us more information.

```
max(v(:))
ans =        2.4832
min(v(:))
ans =       -11.542
```

This tells us the range of values of the speed, but a plot would be more instructive.

One way to examine the speed graphically is with an **isosurface**, which is similar in concept to a contour plot. Every data point on the surface has the same value. There are multiple options for using the `isosurface` function, but the simplest is to enter the *x*, *y*, and *z* arrays and the array representing the scalar quantity of interest, plus a single value for which you'd like to create a surface. For example, to create a surface where all the speeds are equal to -3 type:

```
isosurface(x,y,z,v,-3);
```

The result, shown in Figure 14.30(a), could be improved by adding some lighting and changing the viewing angle.

```
view(30,-15)
camlight left
```

The results are shown in Figure 14.30(b).

The MATLAB documentation contains multiple examples of how to use isosurfaces to visualize scalar data.

Contour Slices

An alternative to isosurfaces is the contour slice. Contour slices involve slicing through a three-dimensional field of data and creating a contour plot. Think of the CT scans you may be familiar with such as the one shown in Figure 14.31. Slicing the array reduces the problem to two dimensions and then allows us to use familiar plotting techniques.

Consider again the flow data from the submerged jet and draw a contour plot in the plane where $z = 0$.

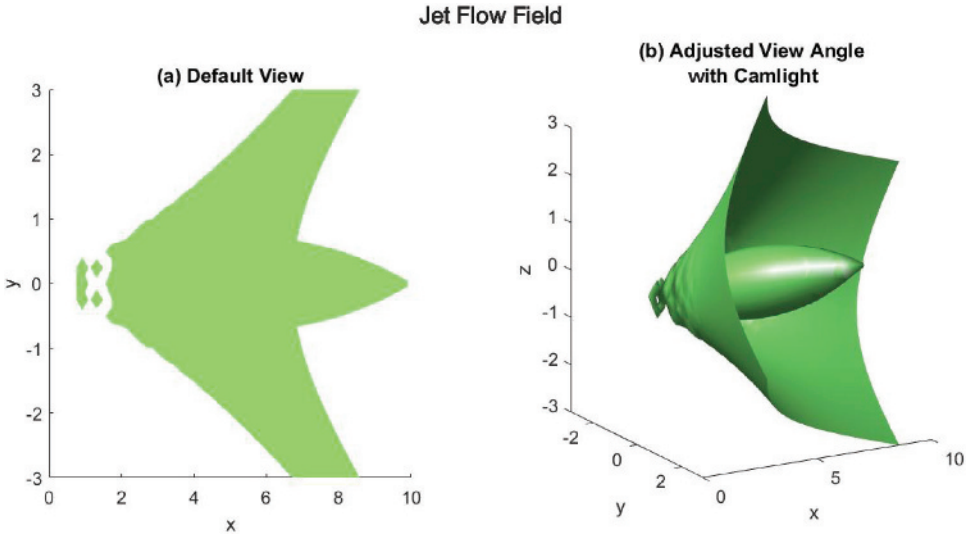


Figure 14.30
The images show a surface representing a constant value of speed for a submerged jet in an infinite tank. The data is available from a sample MATLAB program called `flow` stored on the standard MATLAB search path. (a) An isosurface created for speed values of –3. (b) The same isosurface with the `camlight` function engaged and the view angle adjusted.

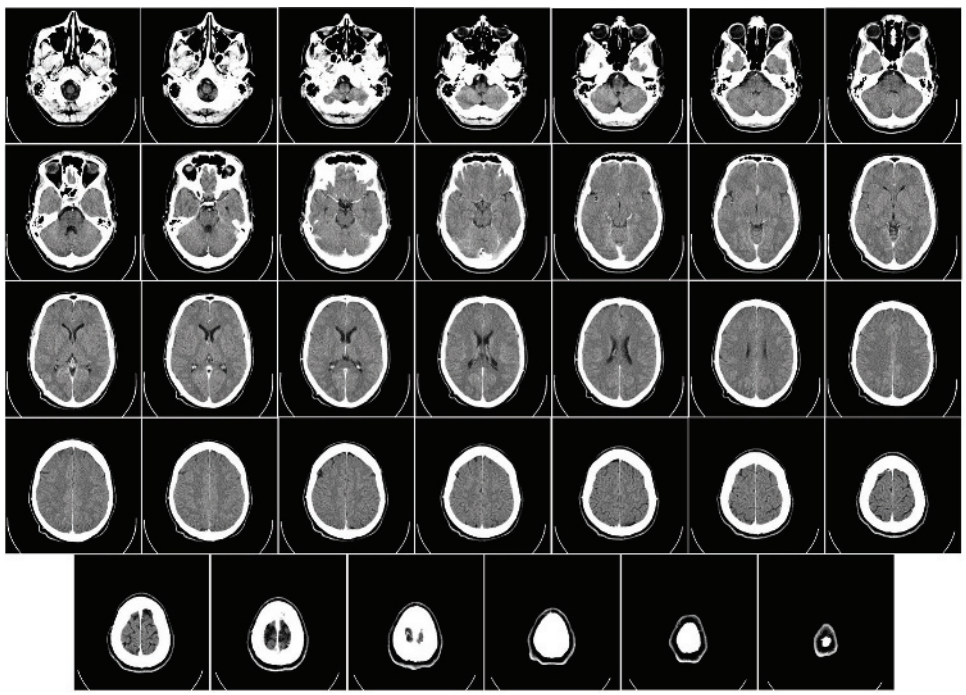


Figure 14.31
Computed tomography of a human brain, from base of the skull to top. Taken with intravenous contrast medium. Department of Radiology, Uppsala University Hospital. Uploaded by Mikael Häggström. - Radiology, Uppsala University Hospital. Uploaded by Mikael Häggström.

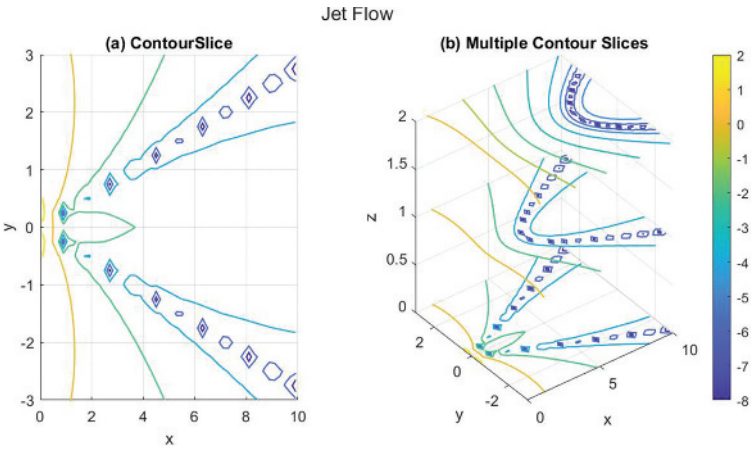


Figure 14.32
(a) The default flow field from a submerged jet used to illustrate visualization techniques is orthogonal (at right angles) to the *z*-axis. (b) Multiple contour slices create a flow field visualization that is easier to interpret.

```
[x,y,z,v]=flow;
zslice = 0;
contourslice(x,y,z,v,[],[],zslice)
```

The `zslice` variable specifies the plane where the slice is drawn. Because no values of *x* and *y* are specified the plane is orthogonal (at right angles) to the *z* axis. The results are shown in Figure 14.32a. If multiple slices that are parallel to each other are desired, you need only specify multiple values in the input vector, such as:

```
zslice = [0,1,2];
contourslice(x,y,z,v, [],[],zslice);
view(3)
```

The `view(3)` instruction forces a three-dimensional view of the plot shown in Figure 14.32(b) so that it is easier to interpret. A `colorbar` was also added for similar reasons. There are multiple examples in the MATLAB help function when you are ready to use contour slicing to visualize your data.

14.5.2 Volume Visualization of Vector Data

In order to display vector data, you need six three-dimensional arrays:

- Three arrays to define the *x*, *y*, and *z* locations of each grid point
- Three arrays to define the vector data *u*, *v*, and *w*

An example of vector data would be velocities, where instead of just speed we know the actual direction air is moving in a flow field.

A sample set of vector volume data, called `wind`, is included in MATLAB as a data file. The command

```
load wind
```

sends six three-dimensional arrays to the workspace. Visualizing this type of data can be accomplished with a number of different techniques, such as

- cone plots
- streamlines
- curl plots

The vector data can be processed into scalar data, and the techniques used in the previous section can be used. However, velocities are not just speeds; they are speeds plus directional information. Thus, velocities are vector data, with components

(called *u*, *v*, and *w*, respectively) in the *x*, *y*, and *z* directions. We could convert velocities to speed by using the formula

```
speed = sqrt(u.^2 + v.^2 + w.^2)
```

The speed data could be represented as one or more contour slices or as isosurfaces (among other techniques). Figure 14.33(a) shows the contourslice plot of the speed at the eighth-elevation (*z*) data set, produced by

```
contourslice(x,y,z,speed,[ ],[ ], 8)
```

and Figure 14.33(b) shows a set of contour slices. The graph was interactively adjusted so that you could see all four slices.

```
contourslice(x,y,z,speed,[ ],[ ],[1, 5, 10, 15])
```

A cone plot of the same data is probably more revealing. Follow the example used in the `coneplot` function description in the help tutorial to create the cone plot shown in Figure 14.34.

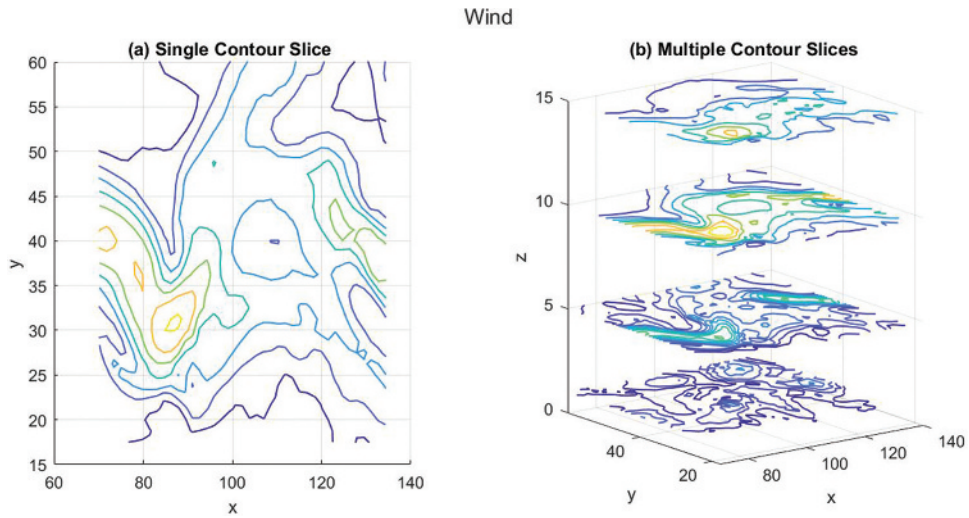


Figure 14.33
Contour slices of the wind-speed data included with the MATLAB program.

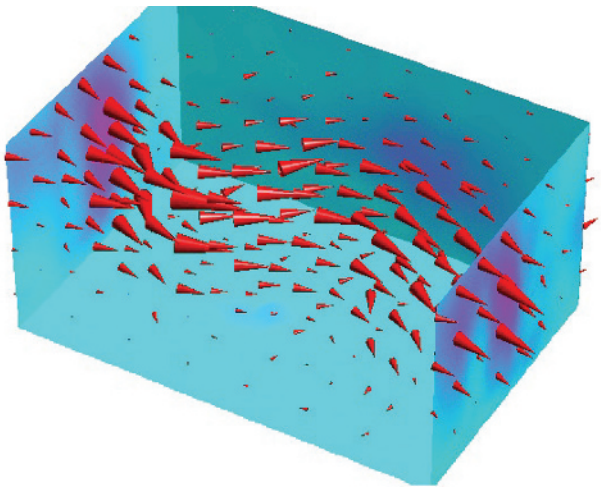


Figure 14.34
Cone plot of the wind-velocity data included with the MATLAB program.

SUMMARY

MATLAB recognizes three different techniques for storing and representing images:

Intensity (or grayscale) images
Indexed images
RGB (or true color) images

The `imagesc` function is used to display intensity images that are sometimes called grayscale images. Indexed images are displayed with the `image` function and require a colormap to determine the appropriate coloring of the image. A custom colormap can be created for each image, or a built-in colormap can be used. *RGB (true color)* images are also displayed with the `image` function but do not require a colormap, since the color information is included in the image file.

If you don't know what kind of image data you are dealing with, the `imfinfo` function can be used to analyze the file. Once you know what kind of file you have, the `imread` function can be used to load an image file into MATLAB, or you can use the software's interactive data controls. The `load` command can load a .dat or a .mat file. To save an image in one of the standard image formats, use the `imwrite` function or the interactive data controls. You can also save the image data as .dat or .mat files, using the `save` command.

A handle is a nickname given to an object in MATLAB. The graphics displayed by MATLAB are composed of several different objects, all of which can be given a handle. The fundamental graphics object is the figure. Layered on the top of the figure is the axis object, and layered on top of that is the actual plot object. Each of these objects includes properties than can be determined with their object name and property name using dot notation, or using the alternate `get` and `set` functions. If you don't know the appropriate object name (also called a handle), the function `gcf` (get current figure) returns the current figure name and `gca` (get current axis) returns the current axis name. For example, to change the color of a plot (the line you drew) named `h`, use

```
set(h,Color="red")
```

or

```
h.Color = 'red'
```

Animation in MATLAB is handled with one of two techniques: redrawing and erasing, or creating a movie. Usually, redrawing and erasing is easier for animations, which represent data that can be quickly computed and are not visually complicated. For tasks that take significant computing power, it is generally easier to capture individual frames and then combine them into a movie to be viewed at a later time.

Complex surfaces are often difficult to visualize, especially since there may be surfaces underneath other surfaces. It is possible to render these hidden surfaces with a specified transparency, which allows us to see the obscured details. This is accomplished with the `alpha` function. The input to this function can vary between 0 and 1, ranging from completely transparent to opaque.

To make surfaces easier to interpret, by default hidden lines are not drawn. The `hidden off` command forces the program to draw these lines.

Although MATLAB includes an extensive lighting-manipulation capability, it is usually sufficient to turn the direct-lighting function on or off. By default, the lighting is diffuse, but it can be changed to direct with the `camlight` function.

Volume-visualization techniques allow us to display three-dimensional data in a number of different ways. Volume data fall into two categories: scalar and vector data. Scalar data involve properties such as temperature or pressure, and vector data

include properties such as velocities or forces. The MATLAB `help` function contains numerous examples of visualization techniques.

MATLAB SUMMARY

The following MATLAB summary lists and briefly describes the special characters, commands, and functions that were defined in this chapter.

Commands and Functions	
`alpha`	sets the transparency of the current plot object
`axis`	controls the properties of the figure axis
`bone`	colormap that makes an image look like an X-ray
`cape`	sample MATLAB image file of a cape
`camlight`	turns the camera light on
`clown`	sample MATLAB image file of a clown
`colormap`	defines which colormap should be used by graphing functions
`camlight`	camera light for graphics
`coneplot`	creates a plot with markers indicating the direction of input vectors
`contourslice`	creates a contour plot from a slice of data
`detail`	sample MATLAB image file of a section of a Dürer wood carving
`drawnow`	forces MATLAB to draw a plot immediately
`durer`	sample MATLAB image file of a Dürer wood carving
`earth`	sample MATLAB image file of the earth
`flow`	built in program containing data to evaluate visualization techniques
`flujet`	sample MATLAB image file showing fluid behavior
`gatlin`	sample MATLAB image file of a photograph
`gca`	get current axis handle
`gcf`	get current figure handle
`get`	returns the properties of a specified object
`getframe`	gets the current figure and saves it as a movie frame in a structure array
`gray`	colormap used for grayscale images
`hidden off`	forces MATLAB to display obscured grid lines
`image`	creates a two-dimensional image
`imagesc`	creates a two-dimensional image by scaling the data
`imfinfo`	reads a standard graphics file and determines what type of data it contains
`imread`	reads a graphics file
`imwrite`	writes a graphics file
`isosurface`	creates surface-connecting volume data, all of the same magnitude
`mandrill`	sample MATLAB image file of a mandrill
`movie`	plays a movie stored as a MATLAB structure array
`patch`	creates a closed shape
`pcolor`	pseudo color plot (similar to a contour plot)
`peaks`	creates a sample plot
`set`	establishes the properties assigned to a specified object
`shading`	determines the shading technique used in surface plots and pseudo color plots
`spine`	sample MATLAB image file of a spine X-ray
`wind`	sample MATLAB data file of wind-velocity information

KEY TERMS

handle
image plot
indexed image
intensity image
isosurface
object
RGB (true color)
scalar data
surface plot
vector data
velocity
volume visualization

PROBLEMS

14.1 On the Internet, find an example of an intensity image, an indexed image, and an RGB image. Import these images into MATLAB and display them as MATLAB figures.

14.2 A quadratic Julia set has the form:

$$z(n+1) = z(n)^2 + c$$

The special case where $c = -0.123 + 0.745i$ is called Douday's rabbit fractal. Follow Example 14.1, and create an image using this value of c. For the Mandelbrot image, we started with all z-values equal to 0. You'll need to start with $z = x + yi$. Let both x and y vary from -1.5 to 1.5.

14.3 A quadratic Julia set has the form

$$z(n+1) = z(n)^2 + c$$

The special case where $c = -0.391 - 0.587i$ is called the Siegel disk fractal. Follow Example 14.1 and create an image using this value of c. For the Mandelbrot image, we started with all z-values equal to 0. You'll need to start with $z = x + yi$. Let both x and y vary from -1.5 to 1.5.

14.4 A quadratic Julia set has the form

$$z(n+1) = z(n)^2 + c$$

The special case where $c = -0.75$ is called the San Marco fractal. Follow Example 14.1 and create an image using this value of c. For the Mandelbrot image, we started with all z-values equal to 0. You'll need to start with $z = x + yi$. Let both x and y vary from -1.5 to 1.5.

14.5 Create a plot of the function

$$y = \sin(x) \qquad \text{for x from } -2\pi \text{ to } +2\pi$$

Assign the plot an object name (handle), and use the handle with appropriate dot nomenclature to change the following properties (if you aren't sure what the object name is for a given property, use the `get` function to see a list of available property names):

(a) Line color from blue to green
(b) Line style to dashed
(c) Line width to 2

14.6 Assign a handle to the figure created in Problem 14.5, and use the dot nomenclature to change the following properties (if you aren't sure what

the object name is for a given property, use the `get` function to see a list of available property names):

(a) Figure background color to red
(b) Figure name to "A Sine Function"

14.7 Assign a handle to the axes created in Problem 14.5, and use dot nomenclature to change the following properties (if you aren't sure what the object name is for a given property, use the `get` function to see a list of available property names):

(a) Background color to blue
(b) *x*-axis scale to log

14.8 Repeat the three previous problems, changing the properties by means of the interactive property inspector. Experiment with other properties and observe the results on your graphs.

14.9 Use the inventory data in the following table to create a line plot with two lines, one for each year. Create a handle to the axes and use it to specify tick marks on the *x*-axis for each month, label each data point with the name of the corresponding month, and turn the labels 45 degrees to make reading the results easier. Add a legend and an appropriate title.

	Year 1	Year 2
January	2345	2343
February	4363	5766
March	3212	4534
April	4565	4719
May	8776	3422
June	7679	2200
July	6532	3454
August	2376	7865
September	2238	6543
October	4509	4508
November	5643	2312
December	1137	4566

14.10 Repeat the previous problem, but create an appropriate bar graph instead.

14.11 Create an animation of the function

$$x \text{ ranging from } -2\pi \text{ to } +2\pi$$

$$y = \sin(x - a) \text{ for}$$

$$a \text{ ranging from } 0 \text{ to } 8\pi$$

- Use a step size for x that results in a smooth graph.
- Let a be the animation variable. (Draw a new picture for each value of a.)
- Use a step size for a that creates a smooth animation. A smaller step size will make the animation seem to move more slowly.

14.12 Create a movie of the function described in the preceding problem.

14.13 Create an animation of the following:

Let x vary from -2π to $+2\pi$
Let $y = \sin(x)$

Let $z = \sin(x - a)\cos(y - a)$
Let a be the animation variable.

Remember that you'll need to mesh x and y to create two-dimensional arrays use the resulting arrays to find z.

14.14 Create a movie of the function described in the preceding problem.

14.15 Create a program that allows you to zoom in on the "rabbit fractal" described in Problem 14.2, and create a movie of the results (see Example 14.2).

14.16 Use a surface plot to plot the `peaks` function. Issue the `hold on` command and plot a sphere that encases the entire plot. Adjust the transparency so that you can see the detail in the interior of the sphere.

14.17 Plot the `peaks` function then issue the `camlight` command. Experiment with placing the `camlight` in different locations, and observe the effect on your plot.

14.18 Load the `wind` data set into the MATLAB workspace using the load command

```
load wind
```

and calculate the scalar wind speed, based on the values of *u*, *v*, and *w*.

```
speed = sqrt(u.^2 + v.^2 + w.^2);
```

Use the `isosurface` function to plot isosurfaces for wind speeds of 10, 20, 30, 40, 50, 60, and 70 mph. The best approach would be to use

```
tiledlayout("flow")
```

and a `for` loop.

Based on your results, do you think an isosurface is a good way to represent the information in this data set?

14.19 Using the `wind` data set from the previous problem, create contour slices orthogonal to the *x*-axis for four evenly spaced values of *x*, based on the calculated speeds.

14.20 Load the *fluidtemp.mat* file into the MATLAB workspace using the `load` command. It includes position and temperature data. Use the `min` and `max` functions to determine reasonable temperatures for which to create isosurfaces, then use a `tiledlayout` to create at least 5 different isosurfaces at different temperatures. Is this a good strategy for visualizing the behavior of the fluid?

14.21 Use the data in the `fluidtemp` data set and create multiple contour slices orthogonal to the *x*-axis.

CHAPTER

15 Simulink®—A Brief Introduction

Objectives

After reading this chapter you should be able to:

- Use Simulink blocks to represent common mathematical processes.
- Create and run a simple Simulink Model.
- Solve differential equations using Simulink models.
- Import Simulink results into MATLAB.

INTRODUCTION

Simulink is an interactive, graphics-based program that allows you to solve problems by creating **models** using a set of built-in "**blocks**." It is part of the MATLAB software suite and requires MATLAB to run. Simulink is included with the student edition of the software, but is not part of the standard installation of the professional edition; this means that it may or may not be included on your version of MATLAB.

15.1 APPLICATIONS

Simulink is designed to provide a convenient method for analyzing **dynamic systems**, that is, systems that change with time. In particular, it found early acceptance in the signal processing community, and is reminiscent of the approach used to program *analog computers*. In fact, one way to think of Simulink is as a virtual **analog computer**. Analog computers required the user to make actual physical connections between electrical components that acted as adders, multipliers, integrators, and so on.

Output from the computer was viewed on an oscilloscope. This is reflected in both the names of the blocks used in Simulink, and in the icons used to represent various operations.

KEY IDEA
Simulink's strength is in modeling dynamic systems.

One shouldn't jump to the conclusion that Simulink is only useful for analyzing electrical systems. Similar mathematical equations describe the behavior of dynamic mechanical systems, reactive chemical systems, and dynamic fluid systems. In fact, it is common to introduce students to the behavior of electricity through analogy with pipe flow problems.

Simulink's strength is its ability to model dynamic systems—which are modeled mathematically as **differential equations**. Usually these systems change with time, but the independent variable could also be location. Differential equations can be solved numerically in MATLAB by making use of functions such as `ode45`, which utilizes Runge–Kutta techniques. They can also be solved analytically using symbolic algebra. Simulink uses similar methods, but they are transparent to the user. Instead of programming equations directly, a visual model is created by collecting appropriate Simulink blocks, then connecting them together using a graphical user interface.

15.2 GETTING STARTED

To start Simulink, open MATLAB and type

```
Simulink
```

into the command window. (Or select the *Start Simulink* icon from the toolstrip as shown in Figure 15.1.)

The Simulink Start Page opens (see Figure 15.2), providing the user access to example models, templates, and blank files. To create a new model select Blank Model (see Figure 15.3). The model window is the workspace where Simulink models are created and executed.

In order to create a model, you'll need to open the Library Browser by selecting the icon from the model window. The browser contains the available libraries of blocks used to create a Simulink model (see Figure 15.4). It's the location where you'll select blocks and drag them into the model workspace. Spend a few minutes exploring the browser. To view the blocks available in each library, either select the library from the left-hand pane or double-click on the icons in the right-hand pane.

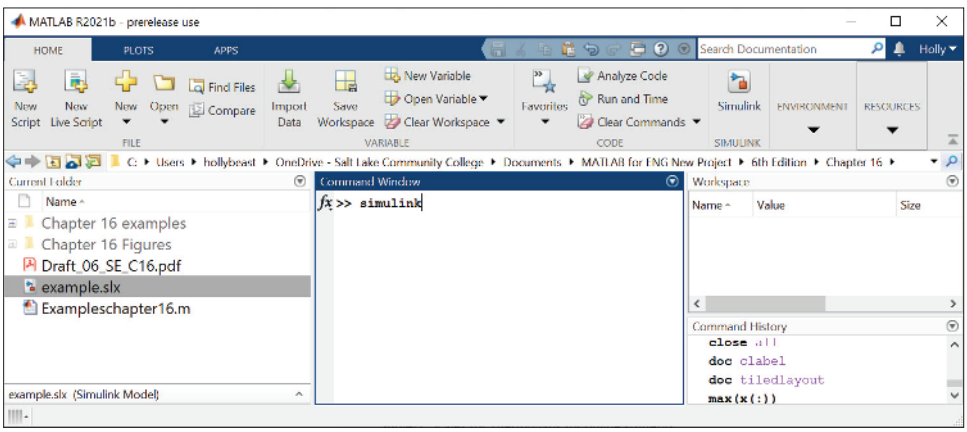


Figure 15.1
Access Simulink either from the command window, or by selecting the **Start Simulink** icon from the tool strip.

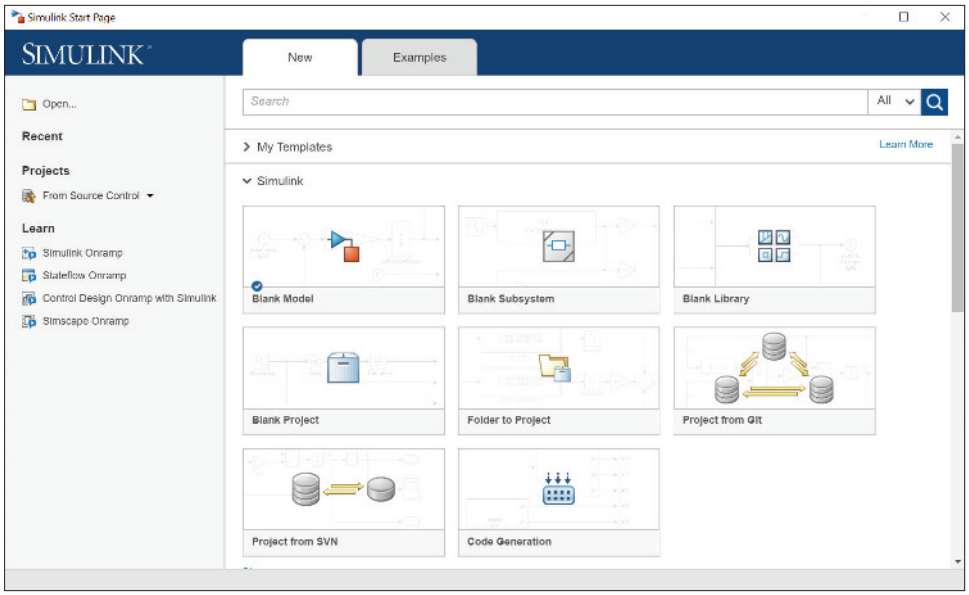


Figure 15.2
The Simulink Start Page provides access to examples, templates, and blank files used to create Simulink models.

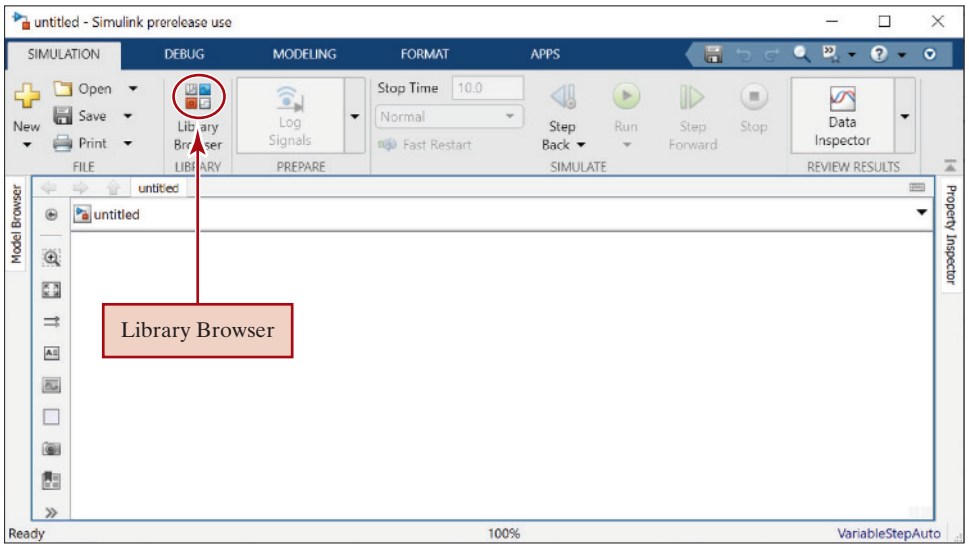


Figure 15.3
The model window is the workspace where Simulink models are created and executed. Access the library of available blocks used to create models with the **Library Browser** icon.

KEY IDEA
Anything you can model in Simulink you can model in MATLAB.

In particular, take a look at the Commonly Used Blocks library—the Source and Sink libraries, and the Math Operations library.

Simulink's strength is in solving complex dynamic systems, but before we try to work on a complex system, it would be better to build some very simple static models to demonstrate the problem-solving process. For convenience, resize the library browser window and the model window so that you can see both on the computer screen. You'll also want to keep the MATLAB desktop open, but resize it so that it fits on your computer screen without overlapping the other windows. See, for example, Figure 15.5.

Our first model will simply add two numbers. From either the Source library or the Commonly Used Blocks library, click and drag the constant block into the model window.

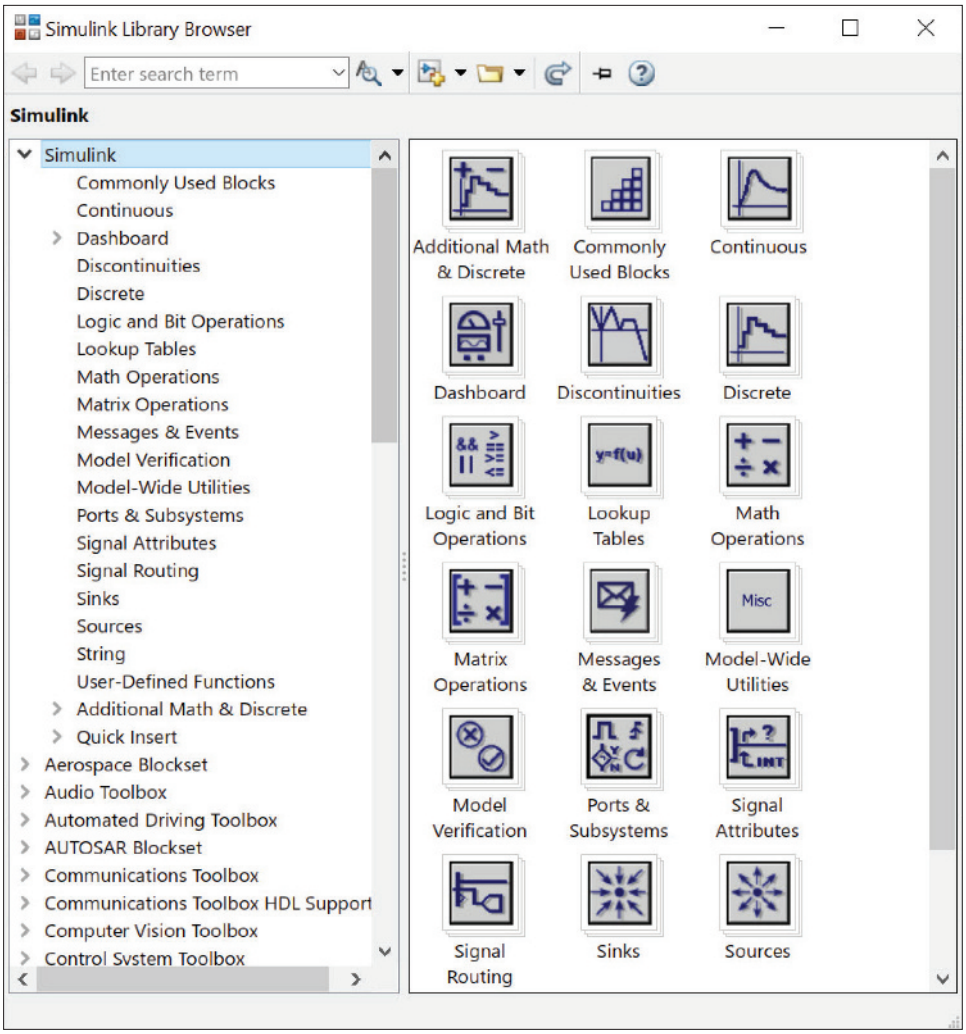


Figure 15.4
The Simulink Library Browser contains numerous blocks that are used to create a Simulink model.

Repeat the process, so that you have two copies of the constant block in the model, as shown in Figure 15.6(a).

Now drag the sum block into the model. It is found both in the Commonly Used Blocks library and the Math Operations library. Notice that the sum block has two "ports." You can draw connections between the constants and the sum block by clicking and dragging between the ports, as shown in Figure 15.6(b). You should notice that the cursor changes to a cross-hair as you connect the ports. The model we've created thus far just adds 1 + 1, and doesn't display the answer. We'll need to modify the constant blocks to specify a value different from the default, which in this case is 1. Double-click on each constant block, and change the *constant value* field, for example, to 5 in the top block and 6 in the bottom block.

To add a display option, look in the Sink library. For this case, the display block is all we need, so drag it to the model and connect it to the output port of the sum block. The last thing we need to do before running the model is to adjust the **simulation** time, from the box on the menu bar (see Figure 15.7). Since nothing in this calculation will change with time, we can change the value to zero. Run the

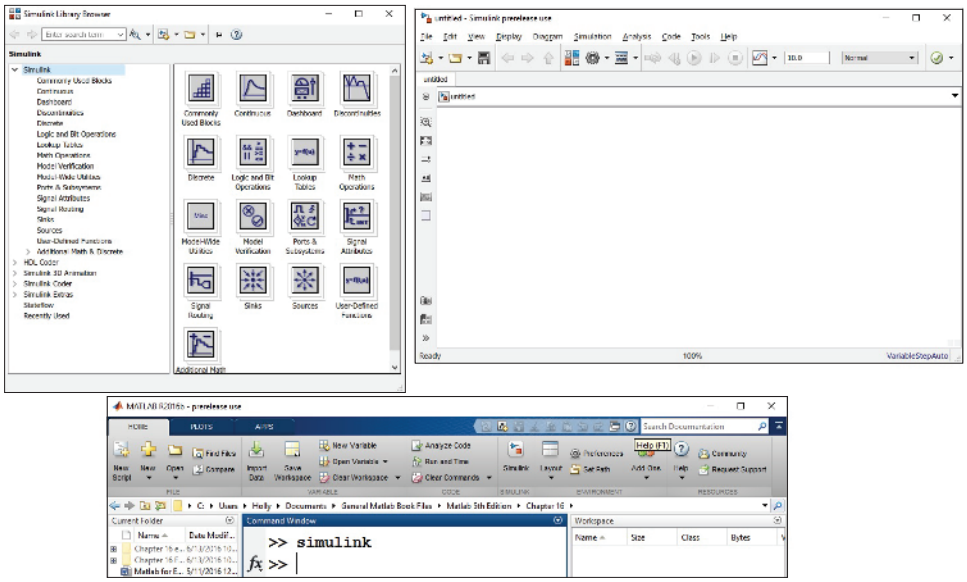
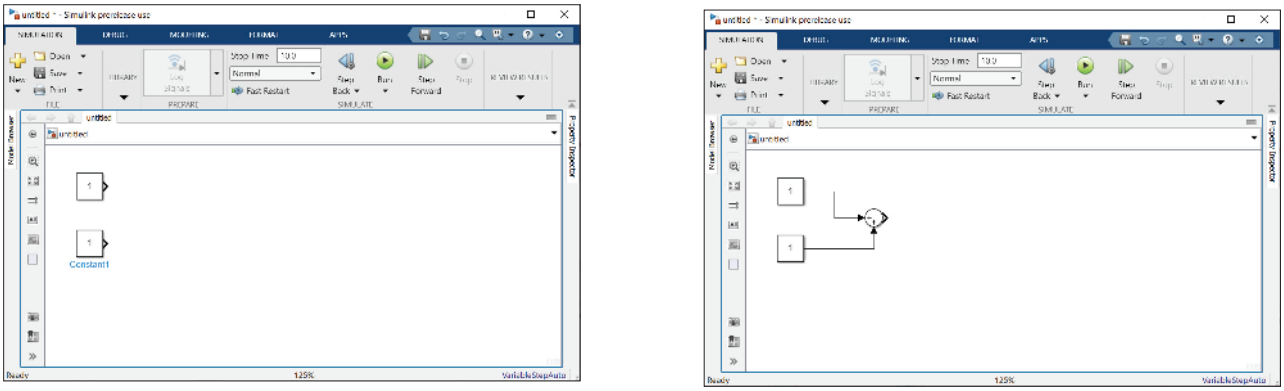


Figure 15.5
Simulink uses multiple windows. Arrange them on your computer desktop so that you can easily drag blocks from the Simulink Library Browser to the model window.



(a) (b)

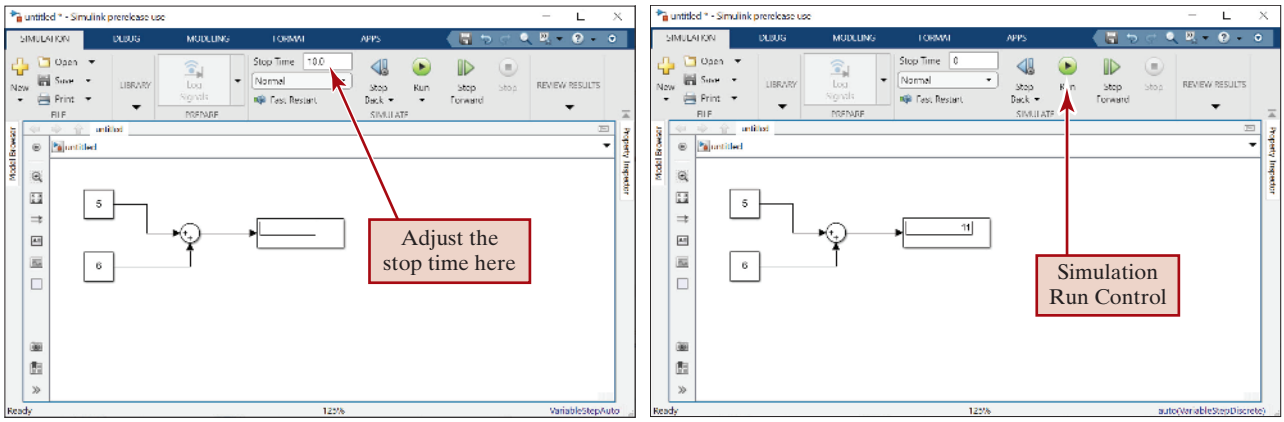
Figure 15.6
(a) Two copies of the constant block were added to the model. (b) The constants are connected to the sum block. Change the values in the constant blocks by double-clicking and modifying the "constant value" field.

simulation by selecting the run button on the toolbar (the black triangle) or by selecting *Simulation* → *Run* from the menu bar.

Save this model in the usual way, by selecting *File* → *Save* and adding an appropriate name. The files are stored with the extension .slx. (In earlier versions of Simulink, the file extension was .mdl.)

As the sum block serves both the addition and subtraction functions, you could use this same model to perform subtraction operations. Double-click the sum block in the model and the block parameter window opens, as shown in Figure 15.8.

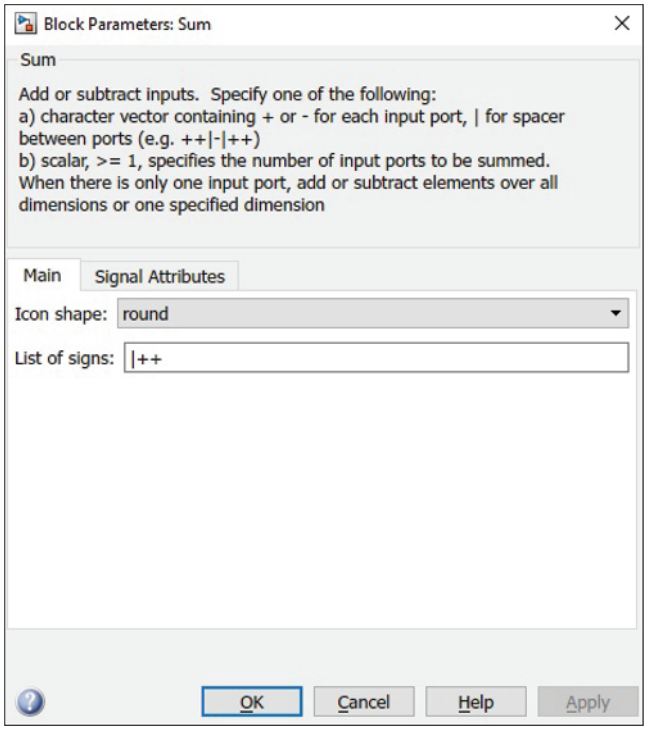
The block description is located near the top of the window, and it provides information on how to use the block—in this case the sum block. This description includes instructions to change the block into a "subtraction block" by changing



(a) (b)

Figure 15.7
(a) Change the stop time variable to zero before executing the completed model, since the solution is not time dependent. (b) The results are shown in the display block after executing the file by selecting the *Run* icon.

Figure 15.8
The sum block can be used to perform subtraction operations, as well as for adding more than two input values.



the input from $|++$ to $|+-$. We could also adjust the block to add three inputs by changing the list of signs field to include three plus signs. Adjust your model and run it several more times as you explore the possibilities for the sum block.

HINT

Simulink includes a "subtraction" block, but if you open its block parameter window you'll notice the block title is "sum."

The previous example was trivial. A slightly more complex model, with results that change with time, is described in Example 15.1.

EXAMPLE 15.1

RANDOM NUMBERS

As we saw in Example 3.5, random numbers can be used to simulate the noise we hear on the radio as static. Although we could solve a similar problem in MATLAB, let's use Simulink. In this case, instead of a music file, use a sine wave as the input to which we want to add the noise, using the following equation:

$$y = 5*\sin(2t) + noise$$

The noise should be the result of a uniform random number generator, with a range of 0 to 1.

1. State the problem
   Create a Simulink model of the equation

   $$y = 5*\sin(2t) + noise$$

   where the noise is based on a random number.

2. Describe the input and output

   Input Use Simulink's built-in sine wave generator to provide the sine wave. Use Simulink's built-in random number generator to simulate the noise.

   Output View the results using the Simulink Scope block.

3. Develop a hand example
   In this case, since we are well versed in MATLAB, a MATLAB solution will substitute for a hand example.

   ```
   t = 0:0.1:10;
   noise = rand(size(t));
   y = 5*sin(2*t)+noise;
   plot(t,y)
   title("A sine wave with noise added")
   xlabel("time,s"), ylabel ("function value")
   ```

   which results in the plot shown in Figure 15.9.

4. Develop a Simulink solution
   Simulink includes blocks for creating both sine waves, and for uniform random number generators. (Be sure to choose *uniform* random numbers; there is also a choice that generates Gaussian random numbers.) You can find both in the Source Library. You'll also need to include an add block found either in the Math library or the Commonly Used Blocks. Finally, add a scope (the name comes from the word "oscilloscope") to view the plotted result. Your model should resemble the one shown in Figure 15.10. To generate the sine wave you can either use the sine wave generator block from the source library (see Figure 15.10(a)) or specify a time source (shown as a clock symbol) and a block to operate on the signal. (see Figure 15.10(b)). The trigonometric function block is found in the Math Operations library. Notice that the time field in the upper right corner of the model is set to 10 seconds, and that two additional

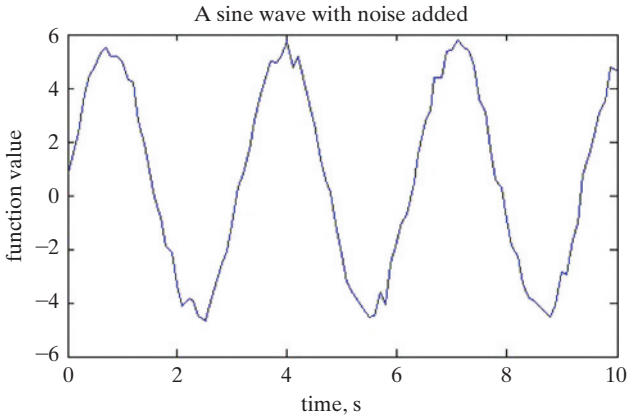
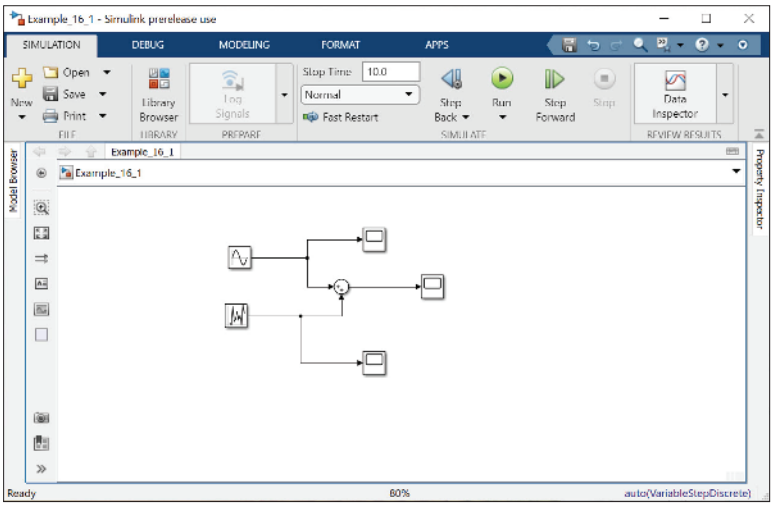
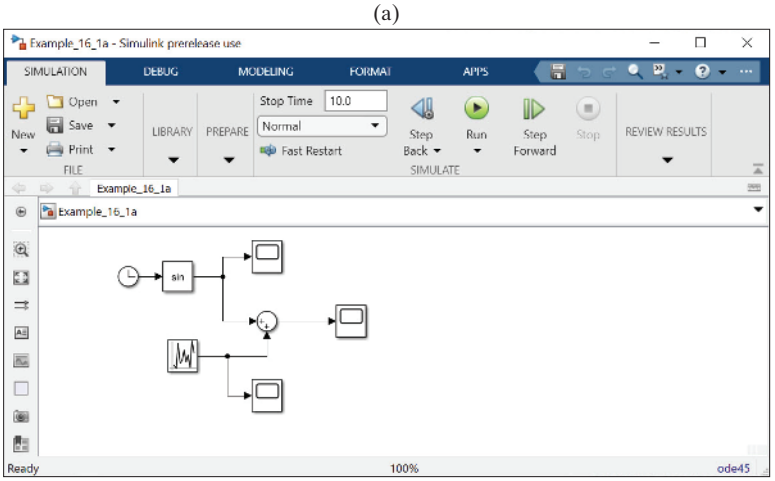


Figure 15.9
A sine wave with noise added. Adding noise to a sine wave can be accomplished using MATLAB, as well as Simulink.



(a)



(b)

Figure 15.10
Simulink model to add noise to a sine wave. a) A sine wave can be generated directly with the Sine Wave source in the Source library or b) A sine wave can be generated with a time source and the trigonometric function block found in the Math Operations library.

scopes were added so that we can observe the behavior of the sine wave generator, the random number generator, and the combined output.

The model specifies only a sine wave, not the entire sine portion of the expression, 5*sin(2t). Open the Sine Wave block by double-clicking on the icon

(*continued*)

inside the model. The Source Block Parameters window opens (as shown in Figure 15.11), allowing us to specify the amplitude, the frequency, and additional parameters as needed. By changing the amplitude to 5 and the frequency to 2, the block now represents the first term in our equation.

Similarly, the random number generator parameter window can be modified to specify a minimum value of 0 and a maximum value of 1. Run the model by selecting the black start simulation triangle, or by selecting *Simulation* → *Start*. To view the output, double-click on each of the scopes. You may need to scale the images by selecting the scaling icon as shown in Figure 15.12, which shows the results of the combined inputs.

5. Test the solution
Compare the results to those found with the MATLAB solution. The model could also be revised so that the results are sent to MATLAB by replacing the scope for the combined output with the simout block, as shown in Figure 15.13. The simout block is found in the Sink library. The simout block has a number of options you can modify. However, if you just accept the defaults a symbolic object called `out` is created in the MATLAB workspace (Figure 15.13) when you run the Simulink model, which can then be used for plotting.

The result of the `plot` command,

```
plot(out)
```

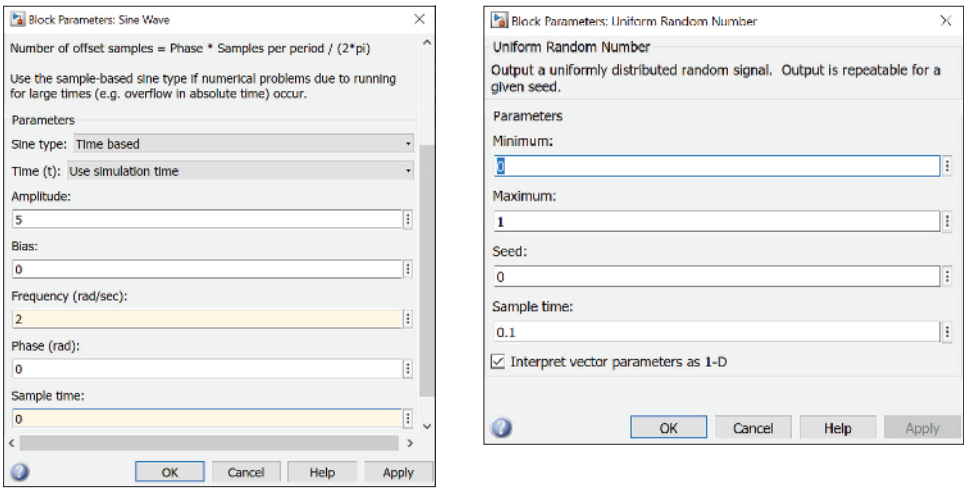


Figure 15.11
(a) The Sine Wave parameter window. (b) The Uniform Random Number parameter window. The Source Block Parameter window for each Simulink block allows the user to modify the default values of the input parameters. Access the parameter window by double-clicking on the block in the model window.

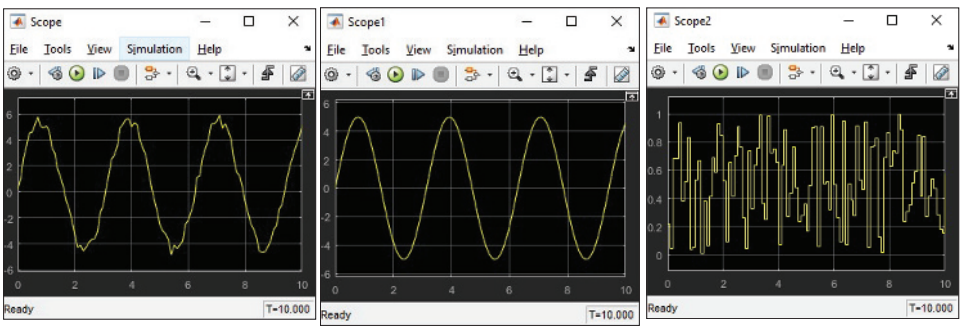


Figure 15.12
The scope output from the three oscilloscopes specified in the Simulink model.

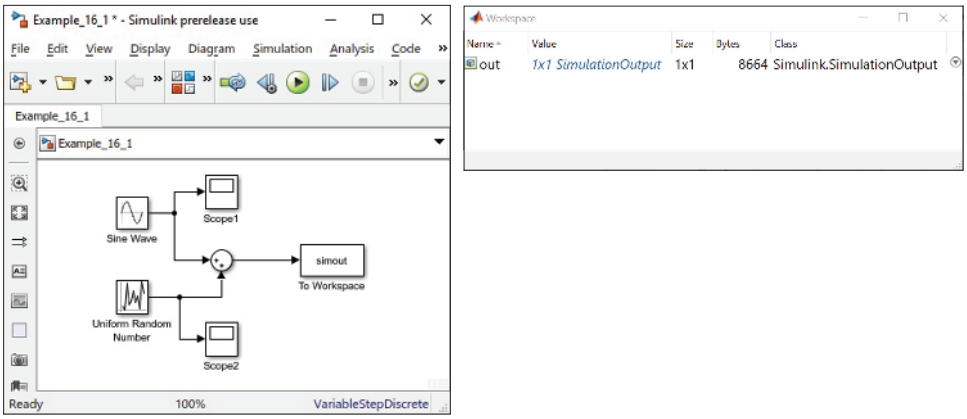


Figure 15.13
The simout block sends simulation results to the MATLAB workspace, where they can be used in other calculations as needed.

is shown in Figure 15.14. Notice that the Simulation Data Inspector is also activated and shows that additional variables are stored inside the `out` variable. They can be extracted using the variable editor, or programatically using dot syntax.

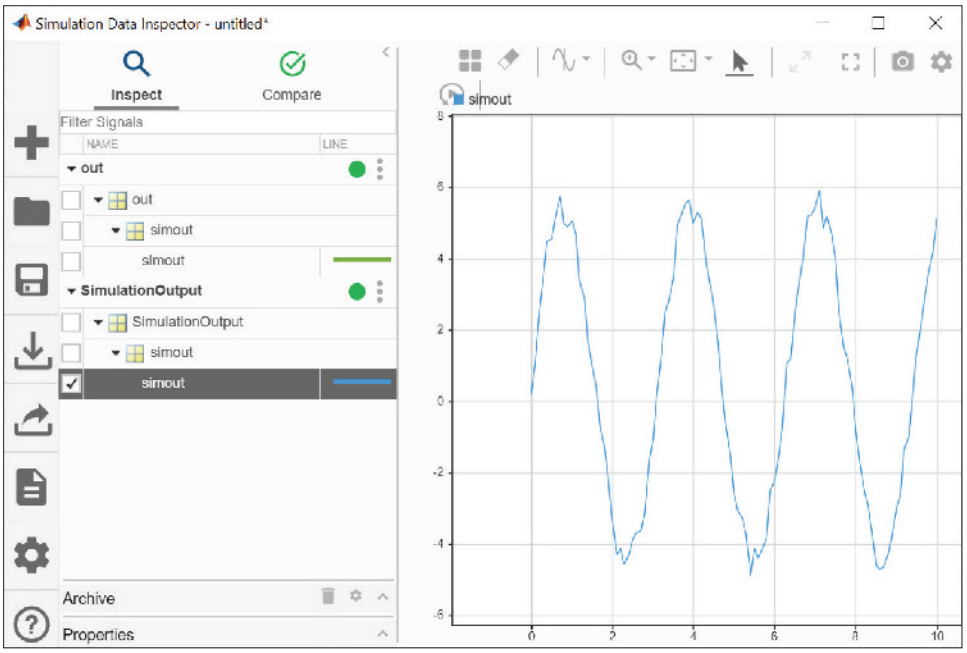


Figure 15.14
The Simulation Data Inspector is activated when the simout block is utilized. It allows the user to view and analyze the output data in the MATLAB workspace.

15.3 SOLVING DIFFERENTIAL EQUATIONS WITH SIMULINK

Thus far the problems described by creating models in Simulink could have been solved more readily in MATLAB. Where Simulink really excels is in solving differential equations. In general, a differential equation includes a dependent variable,

an independent variable, and the derivative of the dependent variable with respect to the independent variable. For example,

$$\frac{dy}{dt} = t^2 + y$$

is a differential equation. In this case y is the dependent variable, t is the independent variable, and dy/dt is the derivative with respect to t. In function notation,

$$\frac{dy}{dt} = f(t, y)$$

KEY IDEA
Simulink makes it easy to visualize systems of differential equations.

To find y, we could integrate

$$y = \int \frac{dy}{dt} dt = \int f(t, y) dt$$

This equation has an infinite number of solutions, unless the initial value of y is defined. For this problem we'll set $y(0) = 0$.

To solve this problem in Simulink, create a model by dragging the appropriate blocks onto the model window, and connecting them as shown in Figure 15.15.

The blocks include the following:

- A clock, to generate times (Source library)
- A math function block, modified in the parameter window to square the block input (Math Operations library)
- A sum block (Commonly Used Blocks library)
- An integrator block (Continuous library)
- A scope block (Sink library).

Adjust the integrator block in the parameter window so that the initial condition is 0. The scope output, after running the model, is shown in Figure 15.16. (You may need to resize to see the entire plot in the scope screen.)

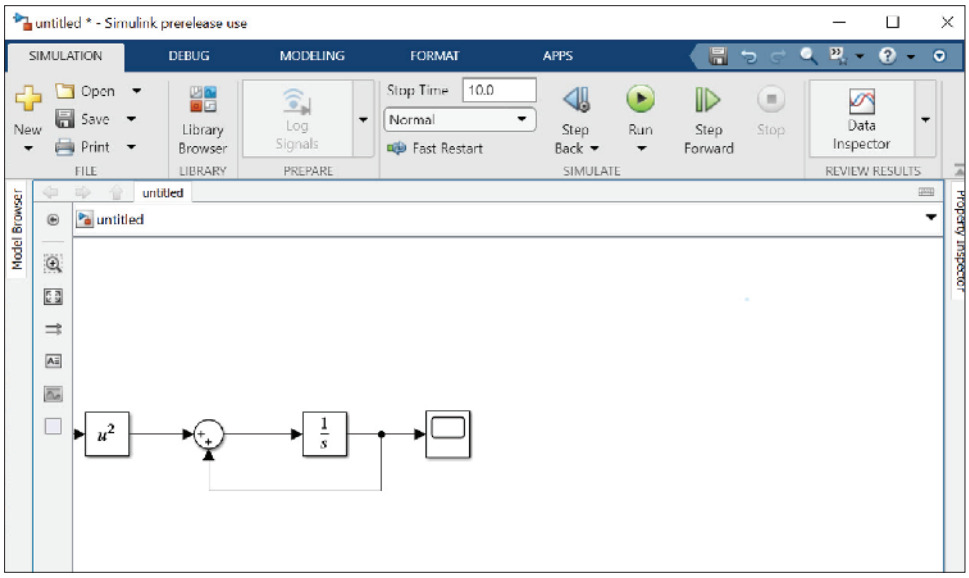


Figure 15.15
Simulink model to solve the differential equation $\frac{dy}{dt} = t^2 + y$.

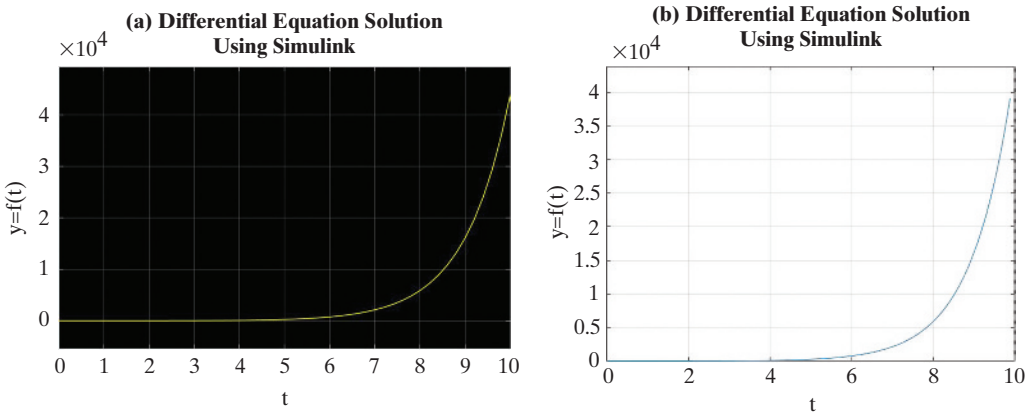


Figure 15.16
A plot of the solution to the ordinary differential equation, $dy/dt = t^2 + y$, with $y(0) = 0$. (a) Plot created with Simulink. (b) Plot created in MATLAB using symbolic algebra.

An alternative approach to this problem might be to use MATLAB's symbolic algebra capability to solve the same problem, as discussed in an earlier chapter. Because this is a simple differential equation, the `dsolve` function can be used:

```
syms y(t)
y=dsolve(diff(y,t)==t^2+y,y(0)==0)
```

which returns the analytical solution of:

```
y =
2*exp(t) - 2*t - t^2 - 2
```

The `fplot` function can be used to plot the result of symbolic calculations. The results are shown in Figure 15.16(b).

EXAMPLE 15.2

VELOCITY OF A FALLING OBJECT

Consider an object falling toward the ground. A widely reported equation describing the resulting velocity is the differential equation:

$$\frac{dv}{dt} = g - \frac{c}{m} v^2$$

where

g is the acceleration due to gravity,
v is the velocity,
m is the mass, and
c is the second-order drag coefficient.

Solve this equation by finding velocity as a function of time, for the first 15 seconds.

1. State the problem
   Use Simulink to find the time versus velocity behavior of a falling object.

(*continued*)

2. Describe the input and output

Input $g = 9.81\ \text{m/s}^2$
$m = 70\ \text{kg}$
$c = 0.3\ \text{kg/m}$
$v(0) = 0\ \text{m/s}$

Output Plot of velocity versus time from 0 to 15 seconds.

3. Develop a hand example
Given that the initial velocity is 0, we would expect that the velocity would rapidly increase—but would eventually level off and reach a terminal value due to drag forces from the air. We would expect a plot much like the sketch shown in Figure 15.17.

4. Develop a Simulink Solution
The Simulink model is shown in Figure 15.18 along with the resulting plot displayed on the scope. It is composed of:

- Three constant blocks,
- Both a divide and product block,
- An add block,
- An integrator block, and
- Math function block, set to square the output of the integrator block.

As you build the model, you will notice that some of the blocks are reversed from their standard orientation. You can accomplish this by placing the block into the model, right-clicking the icon, and selecting Format from the drop-down menu. There are a number of choices that allow the user to select a convenient block orientation. In particular, notice that the math function block has been flipped

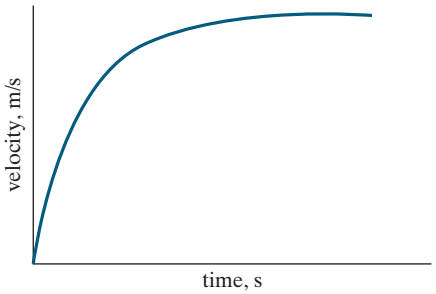


Figure 15.17
Projected behavior of the velocity versus time curve, for a falling object.

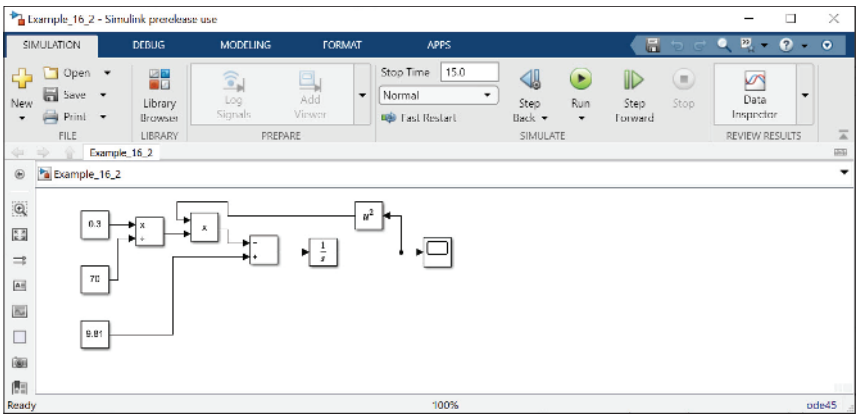


Figure 15.18
Simulink model to solve the falling object problem.

to accommodate the data flow leaving the integrator block. Also notice that the time block has been set to 15 seconds.

If a simout block is used to replace the scope block, the output data is sent to MATLAB, where it could be used in other programs, or plotted in the usual manner.

5. Test the solution
Using the tools found in the symbolic algebra toolbox.

```
clear, clc
syms v(t)
g=9.81; c=0.3; m=70;
y=dsolve(diff(v,t)==g-c/m*v^2,v(0)==0)
fplot(y,[0,15])
title("A falling object")
grid on
xlabel("time,s"), ylabel("velocity, m/s")
```

The resulting plot is shown in Figure 15.19(b), and corresponds well to the scope output from the Simulink model.

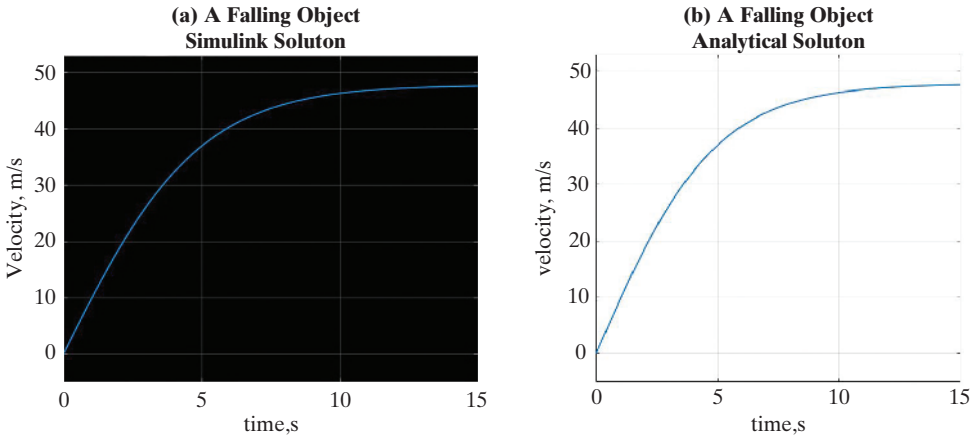


Figure 15.19
The velocity plot for a falling object. (a) Plot created using Simulink. (b) Plot created using MATLAB's symbolic algebra tools. Notice that in both cases the velocity levels are off around 47 m/s.

EXAMPLE 15.3

POSITION OF A FALLING OBJECT

In the previous example, the following differential equation for velocity was solved as a function of time:

$$\frac{dv}{dt} = g - \frac{c}{m} v^2$$

However, velocity can also be described as a derivative; it is the rate of change of position with time

$$v = \frac{dx}{dt}$$

(*continued*)

The velocity equation could be reformulated in terms of position as

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = g - \frac{c}{m}\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)^2.$$

Solve this second-order differential equation using Simulink to create a plot showing how far the object has fallen, as a function of time.

1. State the problem
   Solve the second-order differential equation

   $$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = g - \frac{c}{m}\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)^2$$

   for x as a function of t.

2. Describe the input and output

   Input

   $$\begin{aligned} g &= 9.81\ \mathrm{m/s^2} \\ m &= 70\ \mathrm{kg} \\ c &= 0.3\ \mathrm{kg/m} \\ v(0) &= 0 \\ x(0) &= 0 \\ t &= 0\text{–}15\ \mathrm{seconds} \end{aligned}$$

   Output Create a plot, showing how x changes with time, using the Simulink Scope. Also send the results to MATLAB.

3. Develop a hand example
   As the object falls, it eventually reaches a terminal velocity, as shown in the previous example problem. At that point, x should be increasing at a steady rate. A sketch of the expected behavior is shown in Figure 15.20.

4. Develop a Simulink solution
   The model created in the previous example can be expanded by adding an integration block, and by splitting the output into feeds leading to both the scope and the simout block (see Figure 15.21). Adjust the simout block to report the data as an array.
   The plot created in the scope is shown in Figure 15.22(a).

5. Test the solution
   Once again, MATLAB could be used to solve this second-order differential equation using the symbolic algebra toolbox.

```
syms x(t)
syms g c m
Dx(t)=diff(x,t)
x=dsolve(diff(x,t,2)==g-c/m*Dx^2,x(0)==0, Dx(0)==0)
```

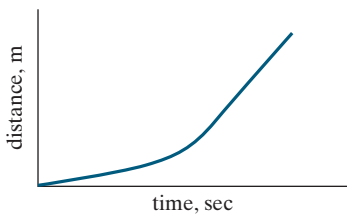


Figure 15.20
Expected position of an object reaching terminal velocity.

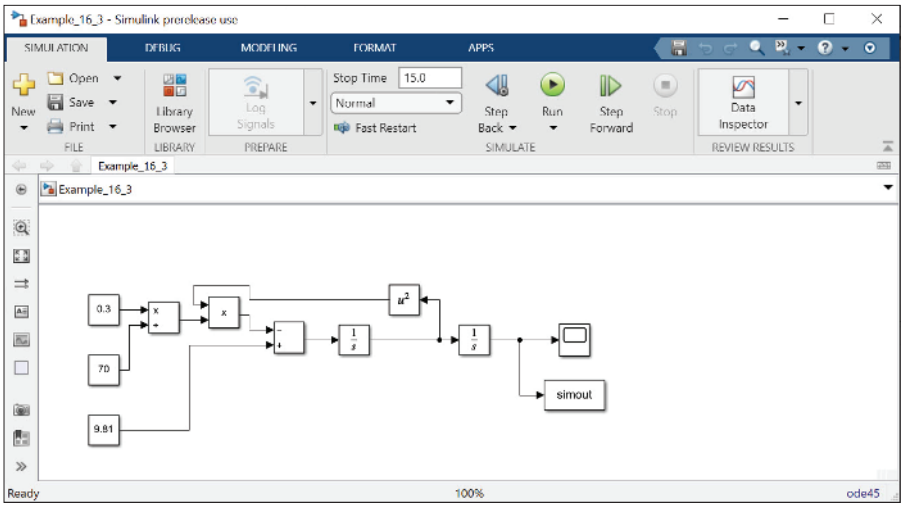


Figure 15.21
Simulink model to solve the second-order differential equation, $\frac{d^2x}{dt^2} = g - \frac{c}{m}\left(\frac{dx}{dt}\right)^2$.

KEY IDEA
It is important to have multiple tools to solve problems. Simulink is better at some tasks than MATLAB and MATLAB is better than Simulink at others.

```
x = subs(x,[g,c,m],[9.81,0.30,70])
fplot(x(1),[0,15])
title(["(b) A Falling Object";"Analytical Solution"])
xlabel("time,s"), ylabel("distance, m")
```

The resulting plot (see Figure 15.22(b)) matches the scope output, and thus verifies our calculations.

Using both Simulink and a symbolic algebra approach, we can develop confidence in the Simulink solutions. Not all problems can be solved symbolically, so having both approaches available is important. In particular, most of the really interesting differential equations do not have an analytical solution.

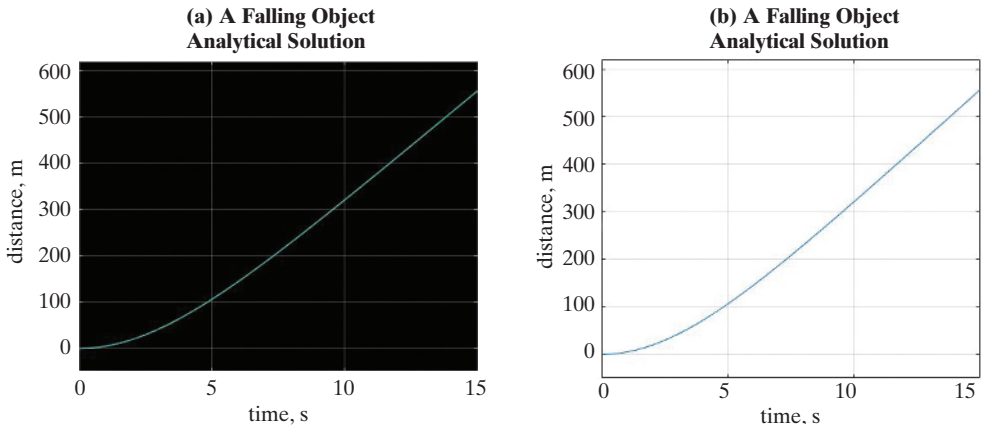


Figure 15.22
The position of a falling object. (a) Results from the Simulink Scope. (b) Results from MATLAB using a symbolic algebra solution.

SUMMARY

Simulink is part of the MATLAB family of programs. It uses a graphical user interface to facilitate the development of models that represent real systems. Simulink is especially useful for modeling dynamic systems—those that can be mathematically described as differential equations.

Simulink relies on a large library of blocks, which can be combined to solve a wide variety of problems. Its visual approach offers an alternative to building MATLAB programs using the numerical techniques described in earlier chapters. However, these same techniques (for example, `ode45`) are used by Simulink when its models are executed.

The MATLAB help function includes an extensive tutorial on using Simulink, including many examples.

Command and Function	
`Simulink`	opens the Simulink library browser

KEY TERMS

analog computers
block
differential equations
dynamic systems
model
simulation

PROBLEMS

15.1 The `sinc` function is often used in electrical engineering applications. It is defined as

$$\text{sinc}(x) = \frac{\sin(x)}{x}$$

Use Simulink to model the behavior of the `sinc` function, from −20 to 20 seconds. Display your results using Simulink's scope block. To adjust the simulation time, in the model window menu bar select *Simulation* → *Configuration Parameters*.

15.2 The equation of a circle can be represented parametrically as

$$x = \sin(t)$$
$$y = \cos(t)$$

where t varies from 0 to 2*pi. Create a Simulink model to parametrically graph a circle using the *xy* graph block found in the Sink library. To model cosine, you will need to modify the Sine block.

15.3 The multiplexer block (Mux) accepts multiple inputs that can then be sent to a scope block to create a graph with multiple signal plots. Use two sine blocks to create a signal representing the $\sin(t)$ and the $\cos(t)$. Combine the signals with the Mux block (found in the Commonly Used Blocks library), and plot the results from 0 to 20 seconds, using a Scope block.

15.4 The derivative block finds the derivative (rate of change) of the incoming signal. Create a Simulink model that finds the derivative of

$$y = \frac{1}{t}$$

and which plots both y and dy/dt in the scope window, for times from 0 to 10 seconds. You'll need a Clock (time) block, the Math Function block, the Derivative block, and a Mux block, in addition to the Scope block.

Applications

15.5 The change in internal energy (kJ/kmol) of an ideal gas over a given temperature range can be represented by the equation:

$$\Delta u = \int_{T_1}^{T_2} (a - R + bT + cT^2 + dT^3)\, dT$$

where T is the temperature in kelvin.

For nitrogen, the constant values are:

$a = 28.90,$
$b = -0.1571 \times 10^{-2},$
$c = 0.8081 \times 10^{-5},$
$d = -2.873 \times 10^{-9},$ and
$R = 8.31447$ kJ/kmol K.

Use Simulink to plot the value of the change in internal energy (Δu) between 0 K and a temperature of 1000 K. (Use the time block to simulate the values of T.) Data Source: B.G. Kyle, *Chemical and Process Thermodynamics.* Englewood Cliffs, NJ: Prentice Hall, 1984.

15.6 Newton's law of cooling tells us that the rate at which an object cools is proportional to the difference in temperature between the object and the surroundings (see Figure P15.6). In other words,

$$\frac{dT}{dt} = k\left(T - T_{\text{surroundings}}\right)$$

where k is a proportionality constant. If, for a cup of hot coffee, the surroundings temperature is 70°F, the constant is 0.5 min^{-1}, and the initial temperature is 110°F, plot the temperature of the object as a function of time for 10 min.



Figure P15.6
A cup of hot coffee cools according to Newton's law of cooling.

15.7 The rate of a chemical reaction is related to the concentration of the reactants. For example, a first-order reaction would have the following relationship between the rate of change of the reactant and the concentration of said reactant:

$$\frac{d[A]}{dt} = -k^*[A]$$

A slightly more complicated reaction might be dependent upon the square of the reactant concentration:

$$\frac{d[A]}{dt} = -k^*[A]^2$$

Model the change in concentration, $[A]$, with time using Simulink for both the first- and second-order reaction problems. Assume $k = 0.1\ \text{min}^{-1}$ for the first-order reaction and $k = 0.1$ l/mol min for the second-order

reaction. The initial concentration, $[A]$, is 5 mol/l. Display the results using a Simulink Scope block. (Choose an appropriate length of time for the simulation, based upon your intermediate results.)

15.8 Blasius showed in 1908 that the solution to the incompressible flow field in a laminar boundary layer on a flat plate is given by the solution of the following third-order ordinary nonlinear differential equation:

$$2\frac{d^3f}{d\eta^3} + f\frac{d^2f}{d\eta^2} = 0$$

To solve this system for f, first solve for the highest-order derivative

$$\frac{d^3f}{d\eta^3} = -0.5f\,\frac{d^2f}{d\eta^2}$$

Now use Simulink to create a model. You'll need three integration blocks plus a multiplier and a gain block (the gain block multiplies by a constant), in addition to a scope block to view the output. The initial conditions are:

$$\frac{d^2f(0)}{dt^2} = 0.332$$
$$\frac{df(0)}{dt} = 0$$
$$f(0) = 0$$

15.9 If a projectile such as a bullet or a rocket is fired vertically, the only force acting on it is the force due to gravity. A force balance yields the equation:

$$\frac{d^2x}{dt^2} = -g\left(\frac{R^2}{(R+x)^2}\right)$$

where

x is the vertical distance measured from the surface of the earth in meters,
R is the radius of the Earth, 6.4×10^6 m, and
g is the acceleration due to gravity, 9.81 m/s^2.

Model this equation using Simulink. Display a graph of the projectile height, x, as a function of time. Assume that the initial height is 0, and the initial velocity is 100 m/s ($dx/dt = 100$ at time $= 0$).

15.10 The motion of a pendulum (see Figure P15.10) can be modeled with an ordinary second-order differential equation as:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{L}\sin(\theta)$$

where

θ is the vertical angle,
g is the acceleration due to gravity, 9.81 m/s^2, and
L is the length of the pendulum, 2 m.

Model the behavior of the pendulum (i.e., the angle as a function of time) with Simulink. Assume the initial angle, θ, is 30° ($\pi/6$ radian) and the initial angular velocity is 0 0 ($d\theta/dt = 0$).

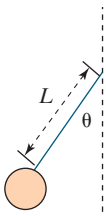


Figure P15.10
The motion of a pendulum is described by a second-order differential equation.

15.11 Consider the simple RC series circuit shown in Figure P15.11. At time zero, the switch is turned on allowing the current to flow. Assuming that constant

voltage is applied, the response of the circuit can be described by the differential equation:

$$R\frac{\mathrm{d}i}{\mathrm{d}t} + \frac{i}{C} = 0,$$

which can be rearranged to

$$\frac{\mathrm{d}i}{\mathrm{d}t} = -\frac{1}{R * C} * i.$$

Use Simulink to model the system response, assuming that $R = 100{,}000\ \Omega$ and $C = 1 \times 10^{-6}$ F. Calculate the initial current value from Ohm's law

$$V = iR$$

with a constant voltage value of 5 V applied to the system.

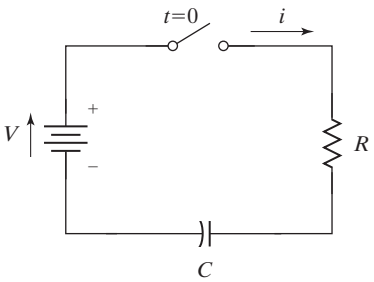


Figure P15.11
A simple RC series circuit.

15.12 The current, i, flowing through the circuit shown in Figure P15.12, can be described by a second-order differential equation:

$$L\frac{\mathrm{d}^2 i}{\mathrm{d}t^2} + R\frac{\mathrm{d}i}{\mathrm{d}t} + \frac{1}{C}i = 0$$

which can be rearranged to give

$$\frac{\mathrm{d}^2 i}{\mathrm{d}t^2} = -\frac{R}{L}\frac{\mathrm{d}i}{\mathrm{d}t} - \frac{1}{L * C}\ i$$

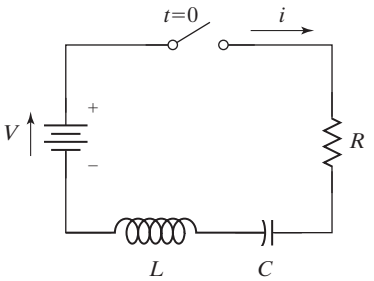


Figure P15.12
A simple RCL circuit can be described by a second-order differential equation.

The behavior of this system depends upon the relative values of L, C, and R (the inductance, capacitance, and the resistance). When

$R^2 > \frac{4L}{C}$ the system is "over-damped," when

$R^2 < \frac{4L}{C}$ the system is "under-damped," and when

$R^2 = \frac{4L}{C}$ the system is "critically damped."

Use Simulink to model the system response, assuming that $R = 100{,}000\ \Omega$ and $C = 1 \times 10^{-6}$ F. Select values of L to meet each of the damping conditions described above. Calculate the initial current value from Ohm's law

$$V = iR$$

with a constant voltage value of 5 V applied to the system.

APPENDIX

A Special Characters, Commands, and Functions

The tables presented in this appendix are grouped according to category, which roughly parallels the chapter organization.

Special Characters	Matrix Definition	Chapter
[ ]	forms arrays	Chapter 2
( )	used in statements to group operations; used with an array name to identify specific elements	Chapter 2
,	separates array elements	Chapter 2
;	separates rows in an array definition; suppresses output when used in commands	Chapter 2
:	used to generate arrays; indicates all rows or all columns	Chapter 2
{ }	used to create cell arrays	Chapter 11

Special Characters	Operators Used in MATLAB Calculations (Scalar and Array)	Chapter
=	assignment operator assigns a value to a memory location; not the same as an equality	Chapter 2
%	indicates a comment in a program.	Chapter 2
%%	creates a section, used to organize code	Chapter 2
+	scalar and array addition	Chapter 2
−	scalar and array subtraction	Chapter 2
*	scalar multiplication and multiplication in matrix algebra	Chapter 2
.*	array multiplication (dot multiply)	Chapter 2
/	scalar division and division in matrix algebra	Chapter 2
./	array division (dot divide)	Chapter 2
^	scalar exponentiation and matrix exponentiation in matrix algebra	Chapter 2
.^	array exponentiation (dot power or dot caret)	Chapter 2
...	ellipsis, continued on the next line	Chapter 4
[]	empty array	Chapter 4
\	left division, used to implement Gaussian elimination with matrices	Chapter 10

Commands	Formatting	Chapter
`format +`	sets format to plus and minus signs only	Chapter 2
`format compact`	sets format to compact form	Chapter 2
`format long`	sets format to 14 decimal places	Chapter 2
`format long e`	sets format to 14 exponential places	Chapter 2
`format long eng`	sets format to engineering notation with 14 decimal places	Chapter 2
`format long g`	allows MATLAB to select the best format (either fixed point or floating point), using 14 decimal digits	Chapter 2
`format loose`	sets format back to default, noncompact form	Chapter 2
`format short`	sets format back to default, 4 decimal places	Chapter 2
`format short e`	sets format to scientific notation with 4 decimal places	Chapter 2
`format short eng`	sets format to engineering notation with 4 decimal places	Chapter 2
`format short g`	allows MATLAB to select the best format (either fixed point or floating point), using 4 decimal digits	Chapter 2
`format rat`	sets format to rational (fractional) display	Chapter 2

Commands	Basic Workspace Commands	Chapter
`ans`	default variable name for results of MATLAB calculations	Chapter 2
`clc`	clears command screen	Chapter 2
`clear`	clears workspace	Chapter 2
`exit`	terminates MATLAB	Chapter 2
`help`	invokes help utility	Chapter 2
`load`	loads arrays from a file	Chapter 2
`quit`	terminates MATLAB	Chapter 2
`save`	saves variables in a file	Chapter 2
`who`	lists variables in memory	Chapter 2
`whos`	lists variables and their sizes	Chapter 2
`help`	opens the help function	Chapter 3
`helpwin`	opens the windowed help function	Chapter 3
`clock`	returns the time	Chapter 3
`date`	returns the date	Chapter 3
`intmax`	returns the largest possible integer number used in MATLAB	Chapter 3
`intmin`	returns the smallest possible integer number used in MATLAB	Chapter 3
`realmax`	returns the largest possible floating-point number used in MATLAB	Chapter 3
`realmin`	returns the smallest possible floating-point number used in MATLAB	Chapter 3
`ascii`	indicates that data should be saved in a standard ASCII format	Chapter 2
`pause`	pauses the execution of a program until any key is hit	Chapter 5

Special Functions	Functions with Special Meaning That Do Not Require an Input	Chapter
`pi`	numeric approximation of the value of π	Chapter 2
`eps`	smallest difference recognized	Chapter 3
`i`	imaginary number	Chapter 3
`Inf`	infinity	Chapter 3
`j`	imaginary number	Chapter 3
`NaN`	not a number	Chapter 3
`clock`	returns the time	Chapter 3
`date`	returns the date	Chapter 3

Functions	Elementary Math	Chapter
`abs`	computes the absolute value of a real number or the magnitude of a complex number	Chapter 3
`erf`	calculates the error function	Chapter 3
`exp`	computes the value of e^x	Chapter 3
`factor`	finds the prime factors	Chapter 3
`factorial`	calculates the factorial	Chapter 3
`gcd`	finds the greatest common denominator	Chapter 3
`isprime`	determines whether a value is prime	Chapter 3
`isreal`	determines whether a value is real or complex	Chapter 3
`lcn`	finds the least common denominator	Chapter 3
`log`	computes the natural logarithm, or log base e ($\log_e$)	Chapter 3
`log10`	computes the common logarithm, or log base 10 ($\log_{10}$)	Chapter 3
`log2`	computes the log base 2 ($\log_2$)	Chapter 3
`nthroot`	finds the real *n*th root of the input array	Chapter 3
`primes`	finds the prime numbers less than the input value	Chapter 3
`prod`	multiplies the values in an array	Chapter 3
`rats`	converts the input to a rational representation (i.e., a fraction)	Chapter 3
`rem`	calculates the remainder in a division problem	Chapter 3
`sign`	determines the sign (positive or negative)	Chapter 3
`sqrt`	calculates the square root of a number	Chapter 3
`sum`	sums the values in an array	Chapter 3

Functions	Trigonometry	Chapter
`asin`	computes the inverse sine (arcsine)	Chapter 3
`asind`	computes the inverse sine and reports the result in degrees	Chapter 3
`cos`	computes the cosine	Chapter 3
`sin`	computes the sine, using radians as input	Chapter 3
`sind`	computes the sine, using angles in degrees as input	Chapter 3
`sinh`	computes the hyperbolic sine	Chapter 3
`tan`	computes the tangent, using radians as input	Chapter 3
`deg2rad`	changes degrees to radians	Chapter 3
`rad2deg`	changes radians to degrees	Chapter 3

MATLAB includes all of the trigonometric functions; only those specifically discussed in the text are included here.

Functions	Complex Numbers	Chapter
`abs`	computes the absolute value of a real number or the magnitude of a complex number	Chapter 3
`angle`	computes the angle when complex numbers are represented with polar coordinates	Chapter 3
`complex`	creates a complex number	Chapter 3
`conj`	creates the complex conjugate of a complex number	Chapter 3
`imag`	extracts the imaginary component of a complex number	Chapter 3
`isreal`	determines whether a value is real or complex	Chapter 3
`real`	extracts the real component of a complex number	Chapter 3

Functions	Rounding	Chapter
`ceil`	rounds to the nearest integer toward positive infinity	Chapter 3
`fix`	rounds to the nearest integer toward zero	Chapter 3
`floor`	rounds to the nearest integer toward minus infinity	Chapter 3
`round`	rounds to the nearest integer	Chapter 3

Functions	Data Analysis	Chapter
`cumprod`	computes the cumulative product of the values in an array	Chapter 3
`cumsum`	computes the cumulative sum of the values in an array	Chapter 3
`length`	determines the largest dimension of an array	Chapter 3
`max`	finds the maximum value in an array and determines which element stores the maximum value	Chapter 3
`mean`	computes the average of the elements in an array	Chapter 3
`median`	finds the median of the elements in an array	Chapter 3
`min`	finds the minimum value in an array and determines which element stores the minimum value	Chapter 3
`mode`	finds the most common number in an array	Chapter 3
`nchoosek`	finds the number of possible combinations when a subgroup of *k* values is chosen from a group of *n* values	Chapter 3
`numel`	determines the total number of elements in an array	Chapter 3
`size`	determines the number of rows and columns in an array	Chapter 3
`sort`	sorts the elements of a vector	Chapter 3
`sortrows`	sorts the rows of a vector on the basis of the values in the first column	Chapter 3
`prod`	multiplies the values in an array	Chapter 3
`sum`	sums the values in an array	Chapter 3
`std`	determines the standard deviation	Chapter 3
`var`	computes the variance	Chapter 3

Functions	Random Numbers	Chapter
`rand`	calculates evenly distributed random numbers	Chapter 3
`randn`	calculates normally distributed (Gaussian) random numbers	Chapter 3
`randi`	calculates random integers	Chapter 3

Functions	Array Formulation, Manipulation, and Analysis	Chapter
meshgrid	maps vectors into a two-dimensional array	Chapters 4 and 5
diag	extracts the diagonal from an array	Chapter 4
fliplr	flips an array into its mirror image from left to right	Chapter 4
flipud	flips an array vertically	Chapter 4
linspace	linearly spaced vector function	Chapter 2
logspace	logarithmically spaced vector function	Chapter 2
cross	computes the cross product	Chapter 10
det	computes the determinant of a matrix	Chapter 10
dot	computes the dot product	Chapter 10
inv	computes the inverse of a matrix	Chapter 10
rref	uses the reduced row echelon format scheme for solving a series of linear equations	Chapter 10

Functions	Two-Dimensional Plots	Chapter
bar	generates a bar graph	Chapter 5
barh	generates a horizontal bar graph	Chapter 5
contour	generates a contour map of a three-dimensional surface	Chapter 5
contourf	generates a filled contour map	Chapter 5
comet	draws an *x*–*y* plot in a pseudo animation sequence	Chapter 5
fplot	creates an *x*–*y* plot on the basis of a function	Chapter 5
histogram	generates a histogram	Chapter 5
histcounts	returns the number of items in each bin	Chapter 5
loglog	generates an *x*–*y* plot with both axes scaled logarithmically	Chapter 5
pcolor	creates a pseudo color plot similar to a contour map	Chapter 5
pie	generates a pie chart	Chapter 5
plot	creates an *x*–*y* plot	Chapter 5
polarplot	creates a polar plot	Chapter 5
semilogx	generates an *x*–*y* plot with the *x*-axis scaled logarithmically	Chapter 5
semilogy	generates an *x*–*y* plot with the *y*-axis scaled logarithmically	Chapter 5

Functions	Three-Dimensional Plots	Chapter
bar3	generates a three-dimensional bar graph	Chapter 5
bar3h	generates a horizontal three-dimensional bar graph	Chapter 5
comet3	draws a three-dimensional line plot in a pseudo animation sequence	Chapter 5
mesh	generates a mesh plot of a surface	Chapter 5
peaks	creates a sample three-dimensional array used to demonstrate graphing functions	Chapter 5
pie3	generates a three-dimensional pie chart	Chapter 5
plot3	generates a three-dimensional line plot	Chapter 5
sphere	sample function used to demonstrate graphing	Chapter 5
surf	generates a surface plot	Chapter 5
surfc	generates a combination surface and contour plot	Chapter 5

Special Characters	Control of Plot Appearance	Chapter
Indicator	**Line Type**	
-	solid	Chapter 5
:	dotted	Chapter 5
-.	dash-dot	Chapter 5
--	dashed	Chapter 5
Indicator	**Point Type**	
.	point	Chapter 5
o	circle	Chapter 5
x	x-mark	Chapter 5
+	plus	Chapter 5
*	star	Chapter 5
s	square	Chapter 5
d	diamond	Chapter 5
v	triangle down	Chapter 5
^	triangle up	Chapter 5
<	triangle left	Chapter 5
>	triangle right	Chapter 5
p	pentagram	Chapter 5
h	hexagram	Chapter 5
Indicator	**Color**	
b	blue	Chapter 5
g	green	Chapter 5
r	red	Chapter 5
c	cyan	Chapter 5
m	magenta	Chapter 5
y	yellow	Chapter 5
k	black	Chapter 5
w	white	Chapter 5

Functions	Figure Control and Annotation	Chapter
`axis`	freezes the current axis scaling for subsequent plots or specifies the axis dimensions	Chapter 5
`axis equal`	forces the same scale spacing for each axis	Chapter 5
`colormap`	color scheme used in surface plots	Chapter 5
`figure`	opens a new figure window	Chapter 5
`gtext`	similar to text; box is placed at a location determined interactively by the user clicking in the figure window	Chapter 5
`grid`	adds a grid to the current plot only	Chapter 5
`grid off`	turns the grid off	Chapter 5
`grid on`	adds a grid to the current and all subsequent graphs in the current figure	Chapter 5
`hold off`	instructs MATLAB to erase figure contents before adding new information	Chapter 5
`hold on`	instructs MATLAB not to erase figure contents before adding new information	Chapter 5
`legend`	adds a legend to a graph	Chapter 5
`nexttile`	causes the program to move to the next tile available for plotting	Chapter 5
`shading flat`	shades a surface plot with one color per grid section	Chapter 5
`shading interp`	shades a surface plot by interpolation	Chapter 5
`tiledlayout`	divides the graphics window into sections available for plotting	Chapter 5
`text`	adds a text box to a graph	Chapter 5
`title`	adds a title to a plot	Chapter 5
`xlabel`	adds a label to the *x*-axis	Chapter 5
`ylabel`	adds a label to the *y*-axis	Chapter 5
`zlabel`	adds a label to the *z*-axis	Chapter 5

Functions	Figure Color Schemes	Chapter
`autumn`	optional colormap used in surface plots	Chapter 5
`bone`	optional colormap used in surface plots	Chapter 5
`colorcube`	optional colormap used in surface plots	Chapter 5
`cool`	optional colormap used in surface plots	Chapter 5
`copper`	optional colormap used in surface plots	Chapter 5
`flag`	optional colormap used in surface plots	Chapter 5
`hot`	optional colormap used in surface plots	Chapter 5
`hsv`	optional colormap used in surface plots	Chapter 5
`jet`	optional colormap used in surface plots	Chapter 5
`parula`	default colormap used in surface plots	Chapter 5
`pink`	optional colormap used in surface plots	Chapter 5
`prism`	optional colormap used in surface plots	Chapter 5
`spring`	optional colormap used in surface plots	Chapter 5
`summer`	optional colormap used in surface plots	Chapter 5
`white`	optional colormap used in surface plots	Chapter 5
`winter`	optional colormap used in surface plots	Chapter 5

Functions and Special Characters	Function Creation and Use	Chapter
`addpath`	adds a directory to the MATLAB search path	Chapter 6
`function`	identifies an M-file as a function	Chapter 6
`nargin`	determines the number of input arguments in a function	Chapter 6
`nargout`	determines the number of output arguments from a function	Chapter 6
`pathtool`	opens the interactive path tool	Chapter 6
`varargin`	indicates that a variable number of arguments may be input to a function	Chapter 6
`@`	identifies a function handle, such as any of those used with anonymous functions	Chapter 6
`%`	comment	Chapter 6
`matlabFunction`	converts a symbolic expression into a MATLAB function	Chapter 13

Special Characters	Format Control	Chapter
'	begins and ends a string	Chapter 7
%	placeholder used in the fprintf command	Chapter 7
%f	fixed-point, or decimal, notation	Chapter 7
%d	decimal notation	Chapter 7
%e	exponential notation	Chapter 7
%g	either fixed-point or exponential notation	Chapter 7
%s	string notation	Chapter 7
%%	cell divider	Chapter 7
\n	linefeed	Chapter 7
\r	carriage return (similar to linefeed)	Chapter 7
\t	tab	Chapter 7
\b	backspace	Chapter 7

Functions	Input/Output (I/O) Control	Chapter
disp	displays a string or a matrix in the command window	Chapter 7
fprintf	creates formatted output which can be sent to the command window or to a file	Chapter 7
ginput	allows the user to pick values from a graph	Chapter 7
input	allows the user to enter values	Chapter 7
pause	pauses the program	Chapter 7
sprintf	similar to fprintf creates formatted output which is assigned to a variable name and stored as a character array	Chapter 7
uiimport	launches the import screen	Chapter 7
xlsimport	imports Excel data files	Chapter 7
xlswrite	exports data as an Excel file	Chapter 7
load	loads arrays from a file	Chapter 2
save	saves variables in a file	Chapter 2
celldisp	displays the contents of a cell array	Chapter 11
imfinfo	reads a standard graphics file and determines what type of data it contains	Chapter 14
imread	reads a graphics file	Chapter 14
imwrite	writes a graphics file	Chapter 14
readtable	imports tabular data	Chapter 7
writetable	exports data stored in MATLAB as a table into popular data formats	Chapter 7

Functions	Comparison Operators	Chapter
<	less than	Chapter 8
<=	less than or equal to	Chapter 8
>	greater than	Chapter 8
>=	greater than or equal to	Chapter 8
==	equal to	Chapter 8
~=	not equal to	Chapter 8

Special Characters	Logical Operators	Chapter
`&`	and	Chapter 8
`|`	or	Chapter 8
`~`	not	Chapter 8
`xor`	exclusive or	Chapter 8

Functions	Control Structures	Chapter
`break`	causes the execution of a loop to be terminated	Chapter 9
`case`	sorts responses	Chapter 8
`continue`	terminates the current pass through a loop, but proceeds to the next pass	Chapter 9
`else`	defines the path if the result of an `if` statement is false	Chapter 8
`elseif`	defines the path if the result of an `if` statement is false, and specifies a new logical test	Chapter 8
`end`	identifies the end of a control structure	Chapter 8
`for`	generates a loop structure	Chapter 9
`if`	checks a condition resulting in either true or false	Chapter 8
`menu`	creates a menu to use as an input vehicle	Chapter 8
`listdlg`	creates a menu to use as an input vehicle	Chapter 8
`otherwise`	part of the case selection structure	Chapter 8
`switch`	part of the case selection structure	Chapter 8
`while`	generates a loop structure	Chapter 9

Functions	Logical Functions	Chapter
`all`	checks to see if a criterion is met by all the elements in an array	Chapter 8
`any`	checks to see if a criterion is met by any of the elements in an array	Chapter 8
`find`	determines which elements in a matrix meet the input criterion	Chapter 8
`isprime`	determines whether a value is prime	Chapter 3
`isreal`	determines whether a value is real or complex	Chapter 3

Functions	Timing	Chapter
`clock`	determines the current time on the CPU clock	Chapter 9
`etime`	finds elapsed time	Chapter 9
`tic`	starts a timing sequence	Chapter 9
`toc`	stops a timing sequence	Chapter 9
`date`	returns the date	Chapter 3
`timeit`	finds time required to execute a function	Chapter 9

Functions	Special Arrays	Chapter
`eye`	generates an identity matrix	Chapter 10
`magic`	creates a "magic" square	Chapter 10
`ones`	creates an array containing all ones	Chapter 10
`pascal`	creates a Pascal matrix	Chapter 10
`zeros`	creates an array containing all zeros	Chapter 10
`gallery`	contains example arrays	Chapter 10

Special Characters	Data Types	Chapter
{ }	cell array constructor	Chapters 11 and 12
"	string data (character information)	Chapters 11 and 12
	character array	Chapter 11
	numeric array	Chapter 11
	symbolic array	Chapter 11
	logical array	Chapter 11
	sparse array	Chapter 11
	cell array	Chapter 11
	structure array	Chapter 11

Functions	Data Type Manipulation	Chapter
`celldisp`	displays the contents of a cell array	Chapter 11
`cell2table`	converts a cell array to a table array	Chapter 11
`char`	creates a padded character array	Chapter 11
`double`	changes an array to a double-precision array	Chapter 11
`int16`	16-bit signed integer	Chapter 11
`int32`	32-bit signed integer	Chapter 11
`int64`	64-bit signed integer	Chapter 11
`int8`	8-bit signed integer	Chapter 11
`num2str`	converts a numeric array to a character array	Chapter 11
`single`	changes an array to a single-precision array	Chapter 11
`sparse`	converts a full-format matrix to a sparse-format matrix	Chapter 11
`string`	creates a string array	Chapter 11
`str2num`	converts a character array to a numeric array	Chapter 11
`sym2poly`	converts a symbolic polynomial into an array	Chapter 11
`table`	creates a table array	Chapters 4 and 11
`uint16`	16-bit unsigned integer	Chapter 11
`uint32`	32-bit unsigned integer	Chapter 11
`uint64`	64-bit unsigned integer	Chapter 11
`uint8`	8-bit unsigned integer	Chapter 11
`Properties`	when appended to a structure variable name it lists the available properties	Chapter 11

Functions	Manipulation of Symbolic Expressions	Chapter
`collect`	collects like terms	Chapter 12
`diff`	finds the symbolic derivative of a symbolic expression	Chapter 12
`dsolve`	differential equation solver	Chapter 12
`expand`	expands an expression or equation	Chapter 12
`factor`	factors an expression or equation	Chapter 12
`int`	finds the symbolic integral of a symbolic expression	Chapter 12
`matlabFunction`	converts a symbolic expression into an anonymous MATLAB function	Chapter 12
`numden`	extracts the numerator and denominator from an expression or an equation	Chapter 12
`simplify`	simplifies using MATLAB's built-in simplification rules	Chapter 12
`solve`	solves a symbolic expression or equation	Chapter 12
`subs`	substitutes into a symbolic expression or equation	Chapter 12
`sym`	creates a symbolic variable, expression, or equation	Chapter 12
`symfun`	creates a symbolic function	Chapter 12
`syms`	creates symbolic variables	Chapter 12
`symvar`	determines the symbolic variables in a function and returns them in the alphabetical order closest to *x*	Chapter 12
`symunit`	adds physical units of measurement to the workspace	Chapter 12
`unitConvert`	converts units	Chapter 12
`separateUnits`	separates a variable into units and value components	Chapter 12

Functions	Symbolic Plotting	Chapter
`fcontour`	creates a contour plot	Chapter 12
`fcontourf`	creates a filled contour plot	Chapter 12
`fimplicit`	creates a 2-D plot from an implicit equation	Chapter 12
`implicit3`	creates a 3-D plot from an implicit equation	Chapter 12
`fmesh`	creates a mesh plot from a symbolic expression	Chapter 12
`fplot`	creates an *x*–*y* plot of a symbolic expression	Chapter 12
`fplot3`	creates a three-dimensional line plot	Chapter 12
`ezpolar`	creates a plot in polar coordinates	Chapter 12
`fsurf`	creates a surface plot from a symbolic expression	Chapter 12

Functions	Numerical Techniques	Chapter
`bvp4c`	boundary value problem solver for ordinary differential equations	Chapter 13
`cftool`	opens the curve-fitting graphical user interface	Chapter 13
`diff`	computes the differences between adjacent values in an array if the input is an array; finds the symbolic derivative if the input is a symbolic expression	Chapter 13
`fminbnd`	a function that accepts a function handle or function definition as input and numerically finds the function minimum between two bounds—known as a "function-function"	Chapter 6
`fzero`	a function that accepts a function handle or function definition as input and finds the zero point nearest a specified value—known as a "function-function"	Chapter 6
`gradient`	finds the derivative numerically using a combination of forward, backward, and central difference techniques	Chapter 13
`interp1`	approximates intermediate data, using either the default linear interpolation technique or a specified higher order approach	Chapter 13
`interp2`	two-dimensional interpolation function	Chapter 13
`interp3`	three-dimensional interpolation function	Chapter 13
`interpn`	multidimensional interpolation function	Chapter 13
`ode45`	ordinary differential equation solver	Chapter 13
`ode23`	ordinary differential equation solver	Chapter 13
`ode78`	ordinary differential equation solver	Chapter 13
`ode89`	ordinary differential equation solver	Chapter 13
`ode113`	ordinary differential equation solver	Chapter 13
`ode15s`	ordinary differential equation solver	Chapter 13
`ode23s`	ordinary differential equation solver	Chapter 13
`ode23t`	ordinary differential equation solver	Chapter 13
`ode23tb`	ordinary differential equation solver	Chapter 13
`ode15i`	ordinary differential equation solver	Chapter 13
`polyfit`	computes the coefficients of a least-squares polynomial	Chapter 13
`polyval`	evaluates a polynomial at a specified value of *x*	Chapter 13
`integral`	computes the integral under a curve	Chapter 13

Functions	Sample Data Sets and Images	Chapter
`cape`	sample MATLAB image file of a cape	Chapter 14
`clown`	sample MATLAB image file of a clown	Chapter 14
`detail`	sample MATLAB image file of a section of a Dürer wood carving	Chapter 14
`durer`	sample MATLAB image file of a Dürer wood carving	Chapter 14
`earth`	sample MATLAB image file of the earth	Chapter 14
`flujet`	sample MATLAB image file showing fluid behavior	Chapter 14
`gatlin`	sample MATLAB image file of a photograph	Chapter 14
`mandrill`	sample MATLAB image file of a mandrill	Chapter 14
`mri`	sample MRI data set	Chapter 14
`peaks`	creates a sample plot	Chapter 14
`seamount`	sample MATLAB data file of a seamount	Chapter 5
`spine`	sample MATLAB image file of a spine X-ray	Chapter 14
`wind`	sample MATLAB data file of wind velocity information	Chapter 14
`sphere`	sample function used to demonstrate graphing	Chapter 5
`census`	a built-in data set used to demonstrate numerical techniques	Chapter 13
`handel`	a built-in data set used to demonstrate the sound function	Chapter 3

Functions	Advanced Visualization	Chapter
`alpha`	sets the transparency of the current plot object	Chapter 14
`camlight`	turns the camera light on	Chapter 14
`coneplot`	creates a plot with markers indicating the direction of input vectors	Chapter 14
`contourslice`	creates a contour plot from a slice of data	Chapter 14
`drawnow`	forces MATLAB to draw a plot immediately	Chapter 14
`gca`	gets current axis handle	Chapter 14
`gcf`	gets current figure handle	Chapter 14
`get`	returns the properties of a specified object	Chapter 14
`getframe`	gets the current figure and saves it as a movie frame in a structure array	Chapter 14
`image`	creates a two-dimensional image	Chapter 14
`imagesc`	creates a two-dimensional image by scaling the data	Chapter 14
`imfinfo`	reads a standard graphics file and determines what type of data it contains	Chapter 14
`imread`	reads a graphics file	Chapter 14
`imwrite`	writes a graphics file	Chapter 14
`isosurface`	creates surface-connecting volume data of the same magnitude	Chapter 14
`movie`	plays a movie stored as a MATLAB structure array	Chapter 14
`set`	establishes the properties assigned to a specified object	Chapter 14
`shading`	determines the shading technique used in surface plots and pseudo color plots	Chapter 14

APPENDIX

B Scaling Techniques

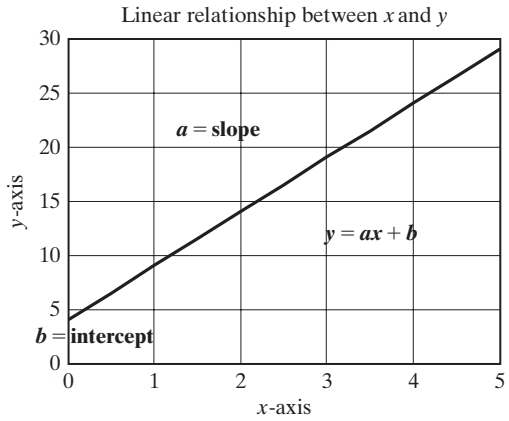
Plotting data using different scaling techniques is a useful way to try to determine how *y*-values change with *x*. This approach is illustrated in the following sections.

LINEAR RELATIONSHIPS

If *x* and *y* are related by a linear relationship, a standard *x*–*y* plot will be a straight line. Thus, for

$$y = ax + b$$

an *x*–*y* plot is a straight line with slope *a* and *y*-intercept *b*.

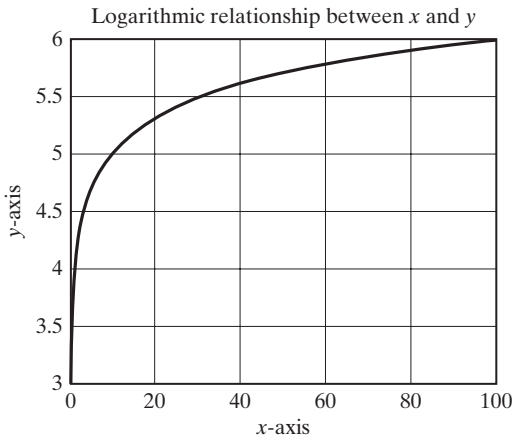


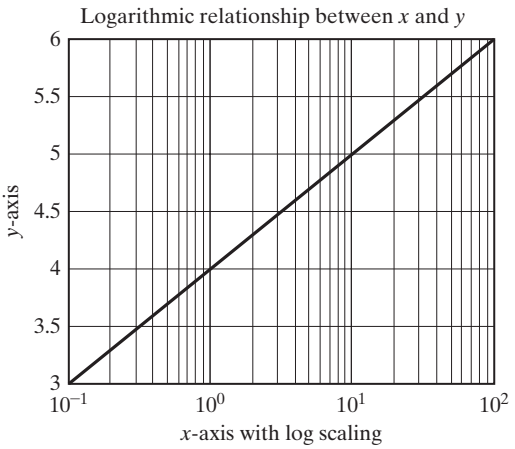
LOGARITHMIC RELATIONSHIP

If x and y are related logarithmically

$$y = a \log_{10}(x) + b$$

a standard plot on an evenly spaced grid is curved. However, a plot scaled evenly on the y-axis, but logarithmically on the x-axis, is a straight line of slope a. The y-intercept doesn't exist, since $\log_{10}(0)$ is undefined. However, when $x = 1$, the value of $\log_{10}(1)$ is zero and y is equal to b.





EXPONENTIAL RELATIONSHIP

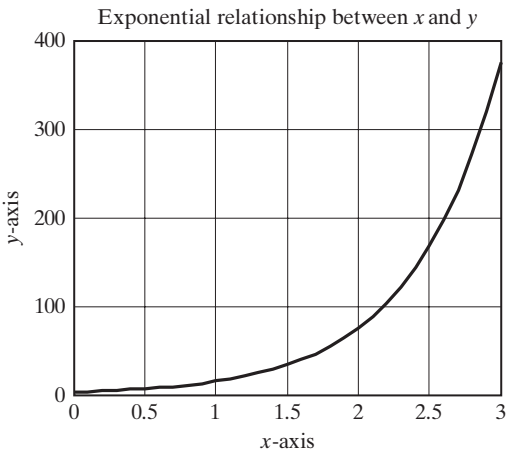
When x and y are related by an exponential relationship such as

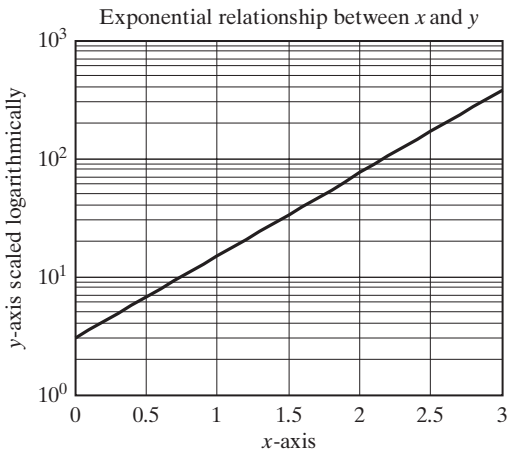
$$y = b * a^x$$

a plot of $\log_{10}(y)$ versus x gives a straight line because

$$\log_{10}(y) = \log_{10}(a) * x + \log_{10}(b)$$

In this case, the slope of the plot is $\log_{10}(a)$.





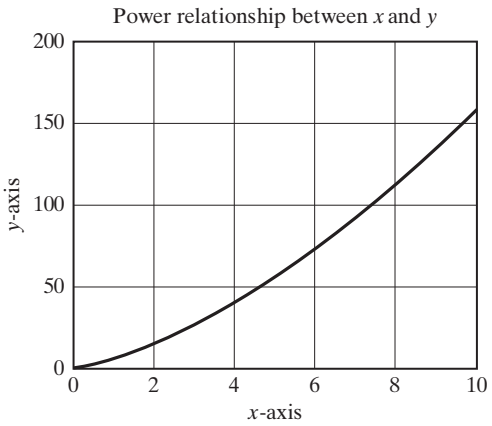
POWER RELATIONSHIP

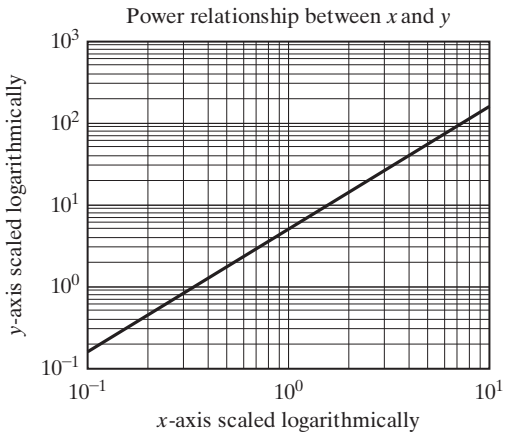
Finally, if x and y are related by a power relationship such as

$$y = bx^a$$

a plot scaled logarithmically on both axes produces a straight line with a slope of a. When x is equal to 1, the $\log_{10}(1)$ is zero, and the value of $\log_{10}(y)$ is $\log_{10}(b)$.

$$\log_{10}(y) = a * \log_{10}(x) + \log_{10}(b)$$





APPENDIX

C Annual Climatological Summary

Table C.1 Annual Climatological Summary, Station: 310301/13872, Asheville, North Carolina, 1999 (Elev. 2240 ft. above sea level; Lat. 35°36′N, Lon. 82°32′W)

Date	Temperature (°F)										Number of Days				Precipitation (inches)									
Elem->	MMXT	MMNT	MNTM	DPNT	HTDD	CLDD	EMXT		EMNP		DT90	DX32	DT32	DT00	TPCP	DPNP	EMXP		TSNW	MXSD		DP01	DP05	DP10
1999 Month	Mean Max.	Mean Min.	Mean	Depart. from Normal	Heating Degree Days	Cooling Degree Days	Highest	High Date	Lowest	Low Date	Max >=90°	Max <=32°	Min <=32°	Min <=0°	Total	Depart. from Normal	Greatest Observed Day	Greatest Observed Date	Snow, Sleet Total Fall	Snow, Sleet Max Depth	Snow, Sleet Max Date	Number of Days >=.10	Number of Days >=.50	Number of Days >=1.0
1	51.4	31.5	41.5	5.8	725	0	78	27	9	5	0	2	16	0	4.56	2.09	1.61	2	2.7	1	31	9	2	2
2	52.6	32.1	42.4	3.5	628	0	66	8	16	14	0	2	16	0	3.07	–0.18	0.79	17	1.2	0T	1	6	3	0
3	52.7	32.5	42.6	-4.8	687	0	76	17	22	8	0	0	19	0	2.47	–1.41	0.62	3	5.3	1	26	8	1	0
4	70.1	48.2	59.2	3.6	197	30	83	10	34	19	0	0	0	0	2.10	–1.02	0.48	27	0.0T	0T	2	6	0	0
5	75.0	51.5	63.3	-0.1	69	25	83	29	40	2	0	0	0	0	2.49	–1.12	0.93	7	0.0	0		5	2	0
6	80.2	60.9	70.6	0.3	4	181	90	8	50	18	1	0	0	0	2.59	–0.68	0.69	29	0.0	0		6	2	0
7	85.7	64.9	75.3	1.6	7	336	96	31	56	13	8	0	0	0	3.87	0.94	0.80	11	0.0	0		10	4	0
8	86.4	63.0	74.7	1.9	0	311	94	13	54	31	7	0	0	0	0.90	–2.86	0.29	8	0.0	0		4	0	0
9	79.1	54.6	66.9	0.2	43	106	91	2	39	23	3	0	0	0	1.72	–1.48	0.75	28	0.0	0		4	1	0
10	67.6	45.5	56.6	0.4	255	1	78	15	28	25	0	0	2	0	1.53	–1.24	0.59	4	0.0	0		3	2	0
11	62.2	40.7	51.5	4.0	397	0	76	9	26	30	0	0	8	0	3.48	0.56	1.71	25	0.3	0		5	3	1
12	53.6	30.5	42.1	2.7	706	0	69	4	15	25	0	0	20	0	1.07	–1.72	0.65	13	0.0T	0T	17	3	1	0
Annual	68.0	46.3	57.2	1.6	3718	990	96	Jul	9	Jan	19	4	81	0	29.85	–8.12	1.71	Nov	9.5	1	Mar	69	21	3

Notes

(blank) Not reported.

– Occurred on one or more previous dates during the month. The date in the Date field is the last day of occurrence. Used through December 1983 only.

A Accumulated amount. This value is a total that may include data from a previous month or months or year (for annual value).

B Adjusted total. Monthly value totals based on proportional available data across the entire month.

E An estimated monthly or annual total.

X Monthly means or totals based on incomplete time series. 1 to 9 days are missing. Annual means or totals include one or more months which had 1 to 9 days that were missing.

M Used to indicate data element missing.

T Trace of precipitation, snowfall, or snowdepth. The precipitation data value will = zero.

S Precipitation amount is continuing to be accumulated. Total will be included in a subsequent monthly or yearly value. *Example*: Days 1–20 had 1.35 inches of precipitation, then a period of accumulation began. The element TPCP would then be 00135S and the total accumulated amount value appears in a subsequent monthly value. If TPCP = "M", there was no precipitation measured during the month. Flag is set to "S" and the total accumulated amount appears in a subsequent monthly value.

National Oceanic and Atmospheric Administration (NOAA)

Index

", 662
%, 652, 659
%%, 652, 660
&, 293, 661
&&, 293
~, 293, 661
',', 651
'...', 652
'.', 657
'-.', 657
( ), 651
{ }, 662
*, 30, 652, 657
+, 24, 164, 652, 657
–, 24, 164, 652
-, 657
.*, 129, 228, 652
.^, 30, 652
/, 24, 30, 652
=, 21, 293, 652
= =, 293, 660
@, 659
[ ], 122, 232, 651
^, 164, 652, 657
|, 661
||, 293
Ð, 164, 657
';', 14, 15, 28
':', 99, 123–125, 262, 651, 657
<, 292, 657, 660
<=, 292, 660
>, 292, 657, 660
>=, 292
'~=', 660

A

`[a,b] =max(x)` function, 80
`abs` function, 68, 107, 112, 654
`[a,b] =size(x)` function, 88
absolute value, 422
`Add` Folder, 243
`addpath` function, 245
Albrecht Dürer's woodcut "Melancholia," 140
algebraic rules, 10
`all` function, 303, 661
`alpha` function, 665
alternating harmonic series, 355–357
`angle` function, 106, 112, 654
animation, 611–615
  movies, 613–615
  redrawing and erasing, 611–613
annotation axes, 610
anonymous functions, 245–246
  converting symbolic expression into, 519–520
  for a single simple differential equation, 572
  third-order polynomial, 566–567
`ans` (variable), 13
antiderivative, 508
`any` function, 303, 661
approximation, 532, 542, 557, 560–564
arcsine, 76
arctangent, 78
`argument` function, 65–66, 109
arithmetic operations, 20–21
array editor, 15–16
array multiplication, 30
array operations, 28–30
arrays
  categorical, 434–435
  cell, 441–443
  logical, 432–433
  multidimensional, 439–441
  sparse, 434
  structure, 443–445, 484
  table, 449–458
  time, 435–437
  timetable, 458–462
arrays in MATLAB
  character, 142–143
  defining, 121–123
  diagonal, 138
  of ones, 137
  problems with two variables, 128–130
  strings, 144–145
  of zeros, 135–137
ASCII coding scheme, 428
`-ascii` command, 43–44, 653
ASCII format, 43, 428
`asind` function, 77, 112, 654
`asin` function, 77, 112, 654
`asinh` function, 77
assignment operator, 21
assignment statements, 122
`audioread` function, 278
Avogadro's constant, 38
axes handles, 608–610

`axis` command, 594
`axis equal` command, 166, 658
`axis` function, 166, 658

B
b, 164, 660
backward difference derivative approximation, 561–563
ballistics, 171–173
  optimum launch angle, 507–508
  using symbolic math to solve, 488–490, 507–508
  using symbolic plotting to solve, 496–498
`bar` function, 185, 656
`bar3` function, 185, 656
bar graphs, 184–185
`barh` function, 185, 656
`bar3h` function, 185, 656
8-bit signed integer, 425
16-bit signed integer, 425
32-bit signed integer, 425
64-bit signed integer, 425
8-bit unsigned integer, 425
16-bit unsigned integer, 425
32-bit unsigned integer, 425
64-bit unsigned integer, 425
boundary value problems, 576–577
`break` statement, 358–359, 661
browsing, 15
built-in functions
  complex numbers, 104–108
  computational limitations, 108–109
  data analysis functions, 80–99
  help feature, 66–67
  math functions, 67–75
  random numbers, 100–104
  special values and miscellaneous functions, 109–111
  trigonometric functions, 75–80
  using, 64–66
buoyancy force, calculating, 77–80
`bvp4c` function, 664

C
c, 164, 657
%c, 266
C++, 1
`calculation` function, 219
calculus, 503–506
`camlight` function, 665
`cape` function, 665
carburizing, 181
cartesian coordinate system, 106
`case` command, 661
`cat` function, 440
`ceil` function, 73, 112, 655
cell arrays, 428, 441–443
`celldisp` function, 660, 662
`cellplot` function, 442
`cell2table` function, 662
`census` data file, 665
center of mass, calculating, 376–379
  using multiplication of matrix, 384–385
central difference technique, 563
`cftool` function, 664
character data, 427–432
  character encoding strategies, 428–429
  creating, 427–428
  using character, 427–432
character encoding strategies, 428–429
`char` function, 428, 662
clausius–clapeyron equation, 69–72, 169–171
`clc` command, 12, 14
`clear` command, 15
climatologic data, analysis of, 97–99
`clock` function, 110, 364–366, 653, 661
`clown` image file, 665
code folding, 324–325
coding schemes, 428
coefficient matrix, 401–402
`collect` function, 663
`collect(S)` function, 479
`colormap` function, 197, 596
`colormap(map_name)` function, 193, 658
color schemes, 659
`comet` function, 656
`comet3` function, 193, 656
command history window, 11–12
command prompt, 3
command window, 10–12, 42, 220
  writing comments, 226
`complex` function, 107, 654
complex numbers, 104–108, 426
computational limitations, 108–109
`coneplot` function, 665
`conj` function, 107, 112, 654
`continue` statement, 358–359, 661
`contour` function, 193, 656
`contourf` function, 656
contour plots, 198
`contourslice` function, 622–623, 665
control structures, 291
converge, 355–357, 361–363
conversions
  to degrees and radians, 222–223, 346–347, 353–354
  to values in newtons (N), 32–34
`cos` function, 76, 112, 654
`cosd` function, 112
`cross` function, 656
cross products, 390–392
cubic spline interpolation, 534–535
`cumprod` function, 84, 112, 655
`cumsum` function, 83–85, 112, 424, 655

current folder window, 11, 15
curve-fitting functions
  linear regression, 542–545
  polynomial regression, 545–546
  polyval function, 546–548
  polyfit function, 543, 580
curve-fitting toolbox, 556

D
d, 164, 657
%d, 266, 660
data analysis functions, 80–99
  averages, 82–83
  determining array size, 88–89
  maximum function, 80–82
  minimum function, 80–82
  sorting values, 85–88
  sums and products, 83–85
  variance and standard deviation, 94–96
database management, 445
data_2.dat, data file, 44
data types, 20, 278
  complex numbers, 426
  integers, 424–426
  logical, 432–433
  numeric, 421–426
  symbolic, 432
date function, 110, 436, 653, 661
debugging code, 324–325
  code analyzer, 281–282
  debugging toolbar, 282–285
deg2rad function, 654
dependent variables, 575
detail, image file, 665
determinants, 388–390
det function, 389, 656
diag function, 136, 138, 656
diagonal arrays, 138
differential calculus, 503–506
  plot of acceleration, 504
differential equations, 516–519
  first-order, 517–518
  higher-order, 518
  Simulink, solving with, 639–641
  solvers, 571
  solving numerically, 570–578
  systems of, 518–519
  for velocity of a falling object, 641–645
diff function, 517, 518, 557–560, 663, 664
diffusion, rates of, 181–184
discrete mathematics, 73–75
display (disp) function, 263–265, 297, 444, 660
diverge, 361, 423
document windows, 11, 15–16
dot function, 376, 656
dot multiplication, 30
dot product, 376
double function, 423, 484, 662
double-precision floating-point number, 421–422, 481, 506
double-precision sparse arrays, 434
drag equation, 34–38
drawnow function, 665
dsolve function, 641, 663
durer image file, 665
dydt function, 572

E
%e, 266, 660
earth, image file, 665
EBCDIC coding system, 428
edit command, 17
editing windows, 11, 17
electrical circuit problem, 397–399
elementary math functions, 67–75, 654
ellipsis, 122
elseif statement, 309–310, 661
else statement, 661
end statement, 308, 661
eps function, 110, 653
erf function, 112, 654
error statement, 129, 386
etime function, 364–366, 661
expand function, 478, 663
exp function, 68, 112, 654
explicit list, 28
exponential relationship, 667
exponentials, 67, 68
exponent overflow, 109
exponent underflow, 109
exporting data, 280
eye function, 394, 662
ezpolar function, 494, 663

F
%f, 266, 660
factor function, 73, 112, 478, 654, 663
factorial function, 73, 112, 654
factorials, calculating, 73
  for loop, 347–349
  while loop, 354–355
fcontourf function, 663
fcontour function, 494, 663
field, 443
figure function, 160, 658
figure handles, 608
fimplicit function, 494, 663
fimplicit3 function, 494, 663
find command, 297–303, 661
fix function, 73, 112, 655

fliplr function, 136, 656
flipud function, 136, 656
floating-point numbers, 38
floor function, 73, 112, 655
flowcharts, 294–296
  for find command, 297–303
flujet data file, 665
fmesh function, 494, 663
fminbnd function, 664
force vectors, calculating, 379–382
for loop, 342–345, 661
  conversions to degrees and radians using, 346–347
format bank command, 39
format + command, 41
format long command, 39–40
format long e command, 40, 42
format long eng command, 40, 42
format long g command, 41, 42
format rat command, 41, 42
format short command, 40
format short e command, 42
format short eng command, 42
format short g command, 41
FORTRAN, 1
forward difference derivative approximation, 560–565
fplot function, 192, 246–247, 494, 567, 641, 656, 663
fplot3 function, 494, 663
fprintf function (formatted print function), 41, 265–269, 297, 660
freely falling body, equation for, 132–135, 260–262
  formatted output, 270–271
fsurf function, 494, 663
function, defined, 219
function functions, 246–247
function input, 65–66
function plots, 192–193
fzero function, 247, 664

G

g, 164, 657
%g, 266, 660
gallery matrix, 410, 662
gasoline purchase, determining, 316–319
gatlin image file, 665
gaussian elimination, 399–402
gaussian random numbers, 101
gca function, 665
gcd function, 73, 112, 654
gcf function, 665
getframe function, 665
get function, 665
ginput, command, 660
global command, 236
global variables, 236–237
grades, assigning, 311–314
gradient function, 563–564, 664
grain size
  "average" (ASTM), determining, 224–226
  metal strength and, 229–230
grain_size function, 224, 658
graphics objects
  annotation axes, 610
  axes handles, 608–610
  figure handles, 608
  plot handles, 605–607
  property editor, 610–611
graphics windows, 11, 16–17
grid command, 158–160, 658
grid off command, 658
grid on command, 658
gtext('string') function, 166, 658

H

h, 164, 657
Hall–Petch equation, 229–230
handel data file, 665
harmonic array, 424
harmonic series, 423
heat capacity of a gas, 551–554
help function, 112, 226
helpwin function, 112
higher-order derivatives, 506
higher-order differential equations, 574–576
histcounts function, 656
hist(x) function, 184–187
histogram function, 656
histograms, 186–187
hold function, 160
hold off function, 658
hold on function, 658
horizon on the earth, calculating distance to, 131–132

I

i (imaginary number) function, 110, 653
ideal gas law, 24–27, 475, 513, 569
identity matrix, 387–388, 394–395
if/else statement, 308–309, 324
if statement, 234–235, 308, 661
ill-conditioned matrices, 388
image(s), 589–604
  animation, 611–615
  cdata function, 596
  colormap function, 596
  GIF graphics format, 595
  graphics objects, 604–611
  hidden lines, 619–620
  image and imagesc functions, 589, 594
  indexed image function, 593–597
  lighting, 620
  peaks function, 564, 590

reading and writing image files, 602–604
reading information, 602–603
storing information, 603–604
transparency, 618–619
true color (RGB), 597–598
visualization techniques, 620–624
image function, 654, 665
image plots, 591, 597
image processing toolbox, 3
imagesc function, 665
imag function, 107, 112
imfinfo function, 660, 665
importing data
import commands, 278–280
import wizard, 277–278
imread function, 660, 665
imwrite function, 603, 660, 665
indexing into an array, 123
index numbers, 297, 299, 444
Inf function, 110, 653
input command, 219, 316, 660
input options
graphical, 276–277
user-defined, 257–259
integer number, 424–426
integral function, 664
integration, 508–511
numerical, 565–568
intensity image, 592–593
interactive figure-manipulation tools, 590
interactive fitting tools
basic fitting tools, 554–556
curve-fitting toolbox, 556
interp1 function, 533–535, 664
interp2 function, 664
interp3 function, 664
interpn function, 664
interpolation techniques
cubic spline, 534–535
extrapolation, 540
linear, 532–534
multidimensional, 540–541
int function, 509–510, 663
int8 function, 662
int16 function, 662
int32 function, 662
int64 function, 662
intmax function, 108, 425
intmin function, 108
inverse of a matrix, 387–388
solution of systems of linear equations using, 397
inv function, 656
iskeyword function, 18
isosurface function, 621–622, 624, 626, 665
isprime function, 73, 112, 654, 661
isreal function, 107, 113, 654, 661
isvarname function, 18

J
JAVA, 1
j (imaginary number) function, 110, 653

K
k, 164, 657
kinetic energy of a moving object, 231
kmoles of gas, 25

L
labels, 158–159
lcm(x,y) function, 73, 113, 654
least squares, 542–544
legend function, 166–167, 658
length function, 88–89, 113, 655
l'Hôpital's theorem, 300
linear equations, solutions of systems of, 396–409
using matrix inverse, 396–397
using matrix left division, 399–402
using reduced row echelon function, 402–403
linear interpolation, 532–534
linear plots, 178
linear regression, 542–544
differences between actual and calculated values, 542
through zero, 544–545
linear relationship, 666
line plots, three-dimensional, 193–194
linspace command, 29
linspace function, 656
listdlg function, 661
load function, 603, 660
local variables, 235–236
logarithmic plots, 178–180
logarithmic relationship, 667
log function, 69, 113, 308, 654
log2 function, 67, 113, 654
log10 function, 67, 69, 113, 654
logical condition, 291, 303
logical data, 432–433
logical functions, 296–303
logical indexing, 303–306
logical operators, 292–294
loglog function, 179, 656
logspace function, 29, 656

M
m, 163, 657
Maclaurin series, 361
magic function, 136, 662
magic matrices, 409
magic squares, 138–140
mandelbrot, benoit, 598–602

mandelbrot image, 615–618
`mandrill` image file, 665
mapping of two-dimensional array, 130
mars surveyor program, 32
.mat files, 43
mathematical constant, 23
math functions
  common computations, 67–68
  discrete mathematics, 73–75
  rounding functions, 72–73
MathWorks packages, 2
MATLAB
  approach to solving technical problems, 5–6
  in biomedical engineering, 3–4
  command prompt, 3
  data types, 20
  in electrical engineering, 3
  exiting, 10
  in fluid dynamics, 4–5
  multiparadigm programming language, 1
  overview, 1–2
  plotting options, 203
  professional version, 2
  program, 45–47
  release R2021b, 2
  saving data, 41–53
  screen, 10
  starting, 9–10
  student versions, 2–3
  toolboxes, 3
  use in industry, 3–5
`matlabFunction` function, 519, 659, 663
matlab.mat, 42
matrices in MATLAB
  array operations, 28–30
  cross products, 390–392
  determinants, 388–390
  dot product, 376
  identity matrix, 394–395
  ill-conditioned, 388
  inverse, 387–388
  multiplication, 382–385
  operations and functions, 374–392
  order of operations, 22–24
  powers, 385–386
  scalar operations, 20–21
  singular, 388
  transpose operator, 375
`max` function, 80–82, 113
`mean` function, 82–83, 113, 655
`median` function, 82–83, 113, 655
`menu` function, 320–321, 661
`mesh` function, 193, 656
`meshgrid` function, 130, 134, 656
mesh plots, 195–196
M-files, 12, 42, 45, 46, 50, 51, 281
  accessing code, 237–238
  to calculate the acceleration of a spacecraft, 47–50
  comments, 226
  creating, 218–238
  functions with multiple inputs and outputs, 226–228
  functions with no inputs and outputs, 232–233
  global variables, 236
  input arguments or output arguments, 233–235
  live scripts and functions, 50
  local variables, 235–236
  syntax, 219–220
midpoint break loops, 359–361
`min` function, 80–82, 113, 655
`minor` function, 160
`mode` function, 82–83, 113, 655
moment of a force about a point, calculating, 392–394
`motion` function, 227
`movie` function, 665
moving boundary work, 569–570
`mri` data file, 665
multidimensional arrays, 439–441
multidimensional interpolation, 540–541
multiplication of matrix, 382–384
multiplicative inverses, 387
`my_3D_array`, 439–440
`my_function`, 219, 238
my_ln_function.mat, 246
`myscript` function, 46

N

n, 164, 660
`namelengthmax` function, 17
`NaN` function, 110, 653
`nargin` function, 233, 659
`nargout` function, 234–235, 659
`nchoosek` function, 73, 113, 655
nested loops, 363–364
nesting functions, 66
`nexttile` function, 173–175, 179, 180, 185, 658
noise, analysis of, 102–104
`nthroot` function, 68, 113, 654
number display format, 38–41
`numden` function, 479, 663
`[num,den]=numden(S)` function, 479
`numel` function, 88–89, 113, 655
numerical integration, 565–568
numerical techniques
  curve fitting, 542–549
  differential equations, 516–519
  integration, 508–512
  interpolation techniques, 532–541
  linear regression, 542–544
  numerical integration, 565–568

polynomial regression, 545–546
polyfit function, 543, 580
polyval function, 546–548
numeric data type, 421–426
num2str function, 264, 662

O
ode23 function, 580
ode45 function, 580
ode113 function, 580
ode15i function, 580
ode15s function, 580
ode23s function, 580
ode23tb function, 580
ode23t function, 580
ode78 function, 580, 664
ode89 function, 580, 664
o, 657
ones function, 137, 364, 662
order of operations, 22–24
orthogonality, 390
otherwise function, 661
output options
display (disp) function, 263–265, 297
fprintf function (formatted print function), 265–269, 297
sprintf function (formatted output), 272
table function, 274–276

P
p, 164, 657
partial derivative, 505
partial differential equations, 577–578
pascal matrix, 409, 662
pathtool, 659
pause function, 160, 660
pcolor function, 193, 656
peaks function, 163, 494, 564, 590, 656, 665
periodic table, 190–192
pie charts, 184–185
pie function, 184, 656
pie3 function, 185, 656
pi function, 110, 653
piston–cylinder devices, work produced in, 512–516, 569–570
placeholder matrix, 137
plot function, 160, 178, 604, 656
plot3 function, 193, 656
plot handles, 605–607
plotting
axis scaling and annotating, 166–168
bar graphs, 184–185
contour plots, 198
creating from workspace window, 202
creating multiple, 159–160
editing of, 200–202
function plots, 192–193
histograms, 186–189
line colors for, 163–166
logarithmic plots, 178–180
mark styles for, 163–166
mesh plots, 195–196
with more than one line, 160–163
pie charts, 184–185
polar plots, 176
pseudo color, 198–200
rectangular plots, 178
saving of plots, 203
subplots, 173–176
surface, 195–200
surf plots, 196–198
symbolic, 490–503
three-dimensional, 193–200
two-dimensional plots, 156–168
x–*y* plots, 189
polar coordinates, 106
polarplot function, 656
polar plots, 176–177
polyfit function, 543, 580
polynomial, 23
polynomial regression, 545–546
poly2sym function, 479
polyval function, 546–548, 580
power of matrix, 385–386
power relationship, 668
primary function, 238
primes function, 73, 113, 654
prod function, 84, 113, 654
projectile motion, 272–274
properties function, 662
property inspector, 610
pseudocode, 294–296
for find command, 297–300
pseudo color plots, 198–200
pythagorean theorem, 106, 131

Q
quad function, 568
quadrature, 565–568

R
r, 164, 657, 660
rad2deg function, 654
radians command, 31
radius, 22
rand function, 100, 113, 655
randi function, 113, 655
randn function, 100, 113, 665
random-number generators, 100
Simulink, using, 636–639

random numbers, 100–104
  Gaussian, 101
  uniform, 100–101
rational number, 73
`rats` function, 73, 113, 654
`readtable` function, 660
`real` function, 108, 113, 654
`realmax` function, 108, 422
`realmin` function, 108, 422
reduced row echelon, 402–403
relational operators, 292–294
`rem` function, 69, 113, 654
repetition structures, 291–292, 294
  basic parts, 341
  `break` statement, 358–359
  `continue` statement, 358–359
  improving efficiency of loops, 364–366
  for loop, 342–345
  midpoint break loops, 359–361
  nested loops, 363–364
  while loop, 350–352
residual, 542
rosser matrix, 410
`round` function, 73, 113, 655
rounding functions, 72–73, 655
round-off error, 423
row-and-column indexing scheme, 442
`rref` function, 402, 656
`run and advance` command, 53
`run` command, 53

S

s, 164, 657
%s, 266, 660
`save` function, 660
scalar operations, 20–21
scientific notation, 38
script, 159
`seamount` data file, 665
secret coding scheme, 430–432
section mode features, 50–53
selection structures, 291–292, 294
  menu function, 320–321
  simple `elseif` statement, 309–311
  simple `if/else` statement, 308–309
  simple `if` statement, 307–308
  simple `switch/case` structure, 315–316
semicolon, 651
`semilogx` function, 179, 656
`semilogy` function, 179, 656
semilogy graph, 184
`separateUnits` function, 663
sequence, 291–292
series, 361
`set` function, 609, 665
`shading flat` function, 193, 658
`shading` function, 665
`shading interp` function, 193, 658
`sign` function, 69, 113, 654
`simple` function, 478–479
`simplify` function, 479, 663
Simulink, 2
  applications, 630–631
  getting started, 631–636
  random numbers to simulate noise, 636–639
  solving differential equations with, 639–641
`sinc` function, 300–303, 305–306
`sinc_x` function, 300–303
`sind` function, 76, 113, 654
`sin` function, 19, 76, 113, 654
`single` function, 422, 662
single-precision floating-point numbers, 422–424, 444
singular matrices, 388
`sinh` function, 77, 113, 654
`size` function, 88, 113, 460, 655
`solve` function, 480–485, 663
solving problems with MATLAB
  data types, 20
  numeric arrays, 20–24
  number display, 38–41
  using variables, 17–19
`sort` function, 113, 655
sorting values, 85–88
`sortrows` function, 88, 113, 655
`sound` function, 102–103, 113
sparse arrays, 434
sparse matrices, 662
`sphere` function, 237, 656, 665
`spine` data file, 665
`sprintf` function (formatted output), 272, 660
`sqrt` function, 68, 113, 654
`squeeze` function, 441
standard deviation, 94–96
`star` function, 232–233
statistical functions, 96
`std` function, 96, 113, 655
storage space, 421
string, 158
string data, 427–430
  character encoding strategies, 428–429
  creating, 427–428
`string` function, 662
`str2num` function, 464, 662
structure arrays, 443–445, 484
  extracting and using data from, 448–449
  storing planetary data with, 445–447
subfunctions, 238–243
subplot, 173–176
`subs` function, 663
substitutions, 486–488

sum function, 83, 113, 654
summary function, 458, 460
surface area, 22–23
surface plot, 589, 590, 626
surfc function, 656
surf function, 193, 233, 656
switch/case structure, 315–316, 661
symbolic capability of MATLAB
  algebra, 472–480
  calculus, 503–511
  converting symbolic expression into anonymous functions, 519
  differential calculus, 503–508
  differential equations, 516–519
  integration, 508–511
  manipulating symbolic expressions and symbolic equations, 477–479
  piston–cylinder devices, work produced in, 512–516
  plotting, 490–498
  solving expressions and equations, 480–490
  substitution and symbolic functions, 486–488
  symbolic math, 482–483
  symbolic variables, 472–480
  units of measurement, 498–503
symbolic data, 432
symbolic toolbox, 472
symbolic variable, 480
sym function, 432, 663
symfun function, 487–488, 663
sym2poly function, 662
syms function, 474, 663
symvar command, 483
symunit function, 474, 663
syntax errors, 283
system of equations, 396–409, 484–486, 574

T
t, 660
table arrays, 449–458
  creating table arrays from data files, 452–455
  creating tables from workspace variables, 449–452
  working with table information, 455–458
table function, 274–275, 662
tan function, 76, 113, 654
tangent function, 66
Taylor series, 361
temperature data, manipulation using matrices, 126–128
text function, 166, 658
thermodynamics, 535–539
third-order polynomial, 566–567
three-dimensional plots, 193–200
tic function, 365, 661
tiledlayout function, 173–175, 179, 180, 185, 658
time arrays, 435–439
timeit function, 366, 661
timetable arrays, 458–462
  synchronizing, 459–462
title command, 158–160, 658
toc function, 365, 661
transpose operator, 228, 375
trapezoid rule, 566
trigonometric functions, 75–80
true color (RGB), 597–598
truss, statistical determination of, 406–409
two-dimensional arrays, 129–130, 439
two-dimensional plots, 156–168
.txt file, 44, 277
type command, 237

U
uiimport function, 277–278, 453–454, 660
uint8 function, 425, 662
uint16 function, 425, 662
uint32 function, 425, 662
uint64 function, 425, 662
uncomment option, 47, 135
unitConvert function, 663
unit vector, 391
UNIX environment
  MATLAB, starting, 10
user-defined functions
  anonymous functions and function handles, 245–246
  function functions, 246–247
  M-files, 218–243
  subfunctions, 238–243
  toolbox, 243–245

V
v, 164
varargin function, 238, 659
var function, 96, 113, 655
variable(s), 13
  dependent, 575
  global, 236
  local, 235–236
  saving, 42–45
  using, 17–19
variable editor, 15, 445
variance, 94–96
vector, 20
vectorizing the code, 344
visualization techniques, 620–624
volume visualization
  of scalar data, 621–623
  of vector data, 623–624

W
w, 164
water desalination plants, solving simultaneous equations, 403–406
water flow through a culvert, calculating, 549–551
weather data, analysis of, 89–94
weight distributions, 187–189
`which` function, 19
`while loop`, 350–352, 661
  calculating factorials, 354–355
  conversions to degrees and radians using, 353–354
white space, 24
`wind` data file, 665
windows
  command history window, 11–12
  command window, 10–12
  current folder window, 11, 15
  document windows, 11, 15–16
  editing windows, 11, 17
  graphics windows, 11, 17
  workspace windows, 11, 13–15
wind tunnels, calculating volume of air in, 24–27
workspace commands, 653
workspace windows, 11, 13–15, 43, 444
`writetable` function, 660
`write` function, 280

X
`xlabel` function, 160, 658
`xlsimport` function, 660
`xlswrite` function, 660
xor, 293, 661
x–*y* plots, 189
  with two *y*-axes, 189

Y
y, 164, 657
yellow grid-like symbol, 13
`ylabel` function, 160, 658

Z
`zeros` function, 135–137, 662
`zlabel` function, 194, 658
`zslice` function, 623

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank