Second Edition

A Guide to MATLAB®

for Beginners and Experienced Users

Brian R. Hunt
Ronald L. Lipsman
Jonathan M. Rosenberg

With Kevin R. Coombes,
John E. Osborn, Garrett J. Stuck

CAMBRIDGE

CAMBRIDGE www.cambridge.org/9780521850681

This page intentionally left blank

A Guide to MATLAB®

for Beginners and Experienced Users

Second Edition

Brian R. Hunt
Ronald L. Lipsman
Jonathan M. Rosenberg

with

Kevin R. Coombes
John E. Osborn
Garrett J. Stuck

A Guide to MATLAB®

for Beginners and Experienced Users

Second Edition

Updated for MATLAB® 7 and Simulink® 6

Brian R. Hunt
Ronald L. Lipsman
Jonathan M. Rosenberg
All of the University of Maryland, College Park

with

Kevin R. Coombes
John E. Osborn
Garrett J. Stuck

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 2RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521850681

© B.Hunt, R.Lipsman, J.Rosenberg, K.Coombes, J. Osborn, and G.Stuck 2001, 2006

This publication is in copyright. Subject to statutory exception and to the provision of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published in print format 2006

ISBN-13 978-0-511-21985-6 eBook (EBL)
ISBN-10 0-511-21985-7 eBook (EBL)

ISBN-13 978-0-521-85068-1 hardback
ISBN-10 0-521-85068-1 hardback

ISBN-13 978-0-521-61565-5 paperback
ISBN-10 0-521-61565-8 paperback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

Contents

The symbol ✰ denotes a more advanced chapter or section.

Preface *page* **xi**

Why We Wrote This Book xi
Who Should Read This Book xii
How This Book Is Organized xii
Conventions Used in This Book xiv
About the Authors xv

1 Getting Started **1**

Platforms and Versions 1
Installation 2
Starting MATLAB 2
Typing in the Command Window 3
Online Help 3
MATLAB Windows 6
Ending a Session 6

2 MATLAB Basics **7**

Input and Output 7
Arithmetic 8
Recovering from Problems 9
Errors in Input 9
Aborting Calculations 9
Algebraic or Symbolic Computation 10
Substituting in Symbolic Expressions 10
Symbolic Expressions, Variable Precision, and Exact Arithmetic 11
Vectors and Matrices 12
Vectors 13
Matrices 14
Suppressing Output 15
Functions 15
Built-in Functions 15
User-Defined Functions 16
Managing Variables 17

Variables and Assignments 18
Solving Equations 20
Graphics 22
Graphing with **ezplot** 22
Modifying Graphs 23
Graphing with **plot** 24
Plotting Multiple Curves 25

3 Interacting with MATLAB 27
The MATLAB Interface 27
The Desktop 27
The Workspace 28
The Current Directory and Search Path 28
The Command History Window 30
M-Files 31
Script M-Files 31
Function M-Files 33
Loops 35
Presenting Your Results 35
Publishing an M-File 36
Diary Files 37
Interactive M-Files 37
Wrapping Long Input and Output Lines 38
Printing and Saving Graphics 38
M-Books 39
Fine-Tuning Your M-Files 41

Practice Set A: Algebra and Arithmetic 43

4 Beyond the Basics 45
Suppressing Output 45
Data Classes 46
String Manipulation 47
Symbolic and Floating-Point Numbers 48
Functions and Expressions 48
Substitution 50
More about M-Files 50
Variables in Script M-Files 50
Variables in Function M-Files 50
Structure of Function M-Files 51
Complex Arithmetic 51
More on Matrices 52
Solving Linear Systems 53
Calculating Eigenvalues and Eigenvectors 53
Doing Calculus with MATLAB 54

Differentiation 54
Integration 54
Limits 55
Sums and Products 56
Taylor Series 57
Default Variables 57

5 MATLAB Graphics 59
Two-Dimensional Plots 59
Parametric Plots 59
Contour Plots and Implicit Plots 60
Field Plots 62
Three-Dimensional Plots 62
Curves in Three-Dimensional Space 62
Surfaces in Three-Dimensional Space 64
Figure Windows 65
Multiple Figure Windows 66
The Figure Toolbar 66
Combining Plots in One Window 67
☆ Customizing Graphics 67
Annotation 69
Change of Plot Style 70
Full-Fledged Customization 71
☆ Images, Animations, and Sound 73
Images 74
Animations 76
Sound 77

Practice Set B: Calculus, Graphics, and Linear Algebra 79

6 MATLAB Programming 85
Branching 85
Branching with `if` 85
Logical Expressions 87
Branching with `switch` 91
More about Loops 92
Open-Ended Loops 92
Breaking from a Loop 93
Other Programming Commands 94
Subfunctions 94
Cell and Structure Arrays 94
Commands for Parsing Input and Output 95
Evaluation and Function Handles 97
User Input and Screen Output 98
Debugging 100

☆ Interacting with the Operating System 100
Calling External Programs 101
File Input and Output 102

7 Publishing and M-Books 103
Fine Points of Publishing 103
More on M-Books 106
The Notebook Menu Items 107
M-Book Graphics 108
More Hints for Effective Use of M-Books 108

8 Simulink 111
A Simple Differential Equation 111
☆ An Engineering Example 115
☆ Communication with the Workspace 120

9 ☆ GUIs 123
GUI Layout and GUIDE 123
Saving and Running a GUI 126
GUI Callback Functions 127

10 Applications 131
Illuminating a Room 132
Mortgage Payments 138
Monte Carlo Simulation 144
Population Dynamics 150
Linear Economic Models 160
Linear Programming 165
The 360° Pendulum 172
☆ Numerical Solution of the Heat Equation 175
☆ A Model of Traffic Flow 185

Practice Set C: Developing Your MATLAB Skills 195

11 Troubleshooting 205
Common Problems 205
Wrong or Unexpected Output 205
Syntax Error 207
Spelling Error 209
Error or Warning Messages When Plotting 209
A Previously Saved M-File Evaluates Differently 210
Computer Won't Respond 211
The Most Common Mistakes 211
Debugging Techniques 212

Solutions to the Practice Sets **219**
Solutions to Practice Set A: Algebra and Arithmetic 219
Solutions to Practice Set B: Calculus, Graphics, and Linear Algebra 231
Solutions to Practice Set C: Developing Your MATLAB Skills 255

Glossary **287**

Index **302**

Preface

> MATLAB is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numerical computation. Using MATLAB, you can solve technical computing problems faster than with traditional programming languages, such as C, C++, and Fortran. – The MathWorks, Inc.

That statement encapsulates the view of *The MathWorks, Inc.*, the developer of MATLAB®. MATLAB 7 is an ambitious program. It contains hundreds of commands to do mathematics. You can use it to graph functions, solve equations, perform statistical tests, and much more. It is a high-level programming language that can communicate with its cousins, e.g., Fortran and C. You can produce sound and animate graphics. You can do simulations and modeling (especially if you have access not just to basic MATLAB but also to its accessory Simulink®). You can prepare materials for export to the World Wide Web. In addition, you can use MATLAB to combine mathematical computations with text and graphics in order to produce a polished, integrated, interactive document.

A program this sophisticated contains many features and options. There are literally hundreds of useful commands at your disposal. The MATLAB help documentation contains thousands of entries. The standard references, whether the MathWorks User's Guide for the product, or any of our competitors, contain a myriad of tables describing an endless stream of commands, options, and features that the user might be expected to learn or access.

MATLAB is more than a fancy calculator; it is an extremely useful and versatile tool. Even if you know only a little about MATLAB, you can use it to accomplish wonderful things. The hard part, however, is figuring out which of the hundreds of commands, scores of help pages, and thousands of items of documentation you need to look at to start using it quickly and effectively.

That's where we come in.

Why We Wrote This Book

The goal of this book is to get you started using MATLAB successfully and quickly. We point out the parts of MATLAB you need to know without overwhelming you with details. We help you avoid the rough spots. We give you examples of real uses of MATLAB that you can refer to when you're doing your own work. We provide a handy reference to the most useful features of MATLAB. When you've finished

reading this book, you will be able to use MATLAB effectively. You'll also be ready to explore more of MATLAB on your own.

You might not be a MATLAB expert when you finish this book, but you will be prepared to become one – if that's what you want. We figure you're probably more interested in being an expert at your own specialty, whether that's finance or physics, psychology or engineering. You want to use MATLAB the way we do, as a tool. This book is designed to help you become a proficient MATLAB user as quickly as possible, so you can get on with the business at hand.

Who Should Read This Book

This book will be useful to complete novices, occasional users who want to sharpen their skills, intermediate or experienced users who want to learn about the new features of MATLAB 7, or who want to learn how to use Simulink, and even experts who want to find out whether we know anything they don't.

You can read through this guide to learn MATLAB on your own. If your employer (or your professor) has plopped you in front of a computer with MATLAB and told you to learn how to use it, then you'll find the book particularly useful. If you are teaching or taking a course in which you want to use MATLAB as a tool to explore another subject – whether in mathematics, science, engineering, business, or statistics – this book will make a perfect supplement.

As mentioned, we wrote this guide for use with MATLAB 7. If you plan to continue using MATLAB 5 or MATLAB 6, however, you can still profit from this book. Virtually all of the material on MATLAB commands in this book applies to all these versions. The primary features of MATLAB 7 not found in earlier versions are anonymous functions, discussed in Chapter 2, and publishing, discussed in Chapters 3 and 7. Beyond that, only a small amount of material on the MATLAB interface, found mainly in Chapters 1, 3, and 9, does not apply to MATLAB 5.

How This Book Is Organized

In writing, we used our experience to focus on providing important information as quickly as possible. The book contains a short, focused introduction to MATLAB. It contains practice problems (with complete solutions) so you can test your knowledge. There are several illuminating sample projects that show you how MATLAB can be used in real-world applications and an entire chapter on troubleshooting.

The core of this book consists of about 70 pages: Chapters 1–4, and the beginning of Chapter 5. Read that much and you'll have a good grasp of the fundamentals of MATLAB. Read the rest – the remainder of the Graphics chapter as well as the chapters on Programming, Publishing, Simulink, GUIs, Applications, Troubleshooting, and the Glossary – and you'll know enough to do a great deal with MATLAB.

Here is a detailed summary of the contents of the book.

Chapter 1, *Getting Started*, describes how to start MATLAB on various platforms. It tells you how to enter commands, how to access online help, how to recognize the

various MATLAB windows you will encounter, and how to exit the application.

Chapter 2, *MATLAB Basics*, shows you how to do elementary mathematics using MATLAB. This chapter contains the most essential MATLAB commands.

Chapter 3, *Interacting with MATLAB*, contains an introduction to the MATLAB Desktop interface. This chapter will introduce you to the basic window features of the application, to the small program files (M-files) that you will use to make the most effective use of the software, and to a few methods for presenting the results of your MATLAB sessions. After completing this chapter, you'll have a better appreciation of the breadth described in the quote that opens this Preface.

Practice Set A, *Algebra and Arithmetic*, contains some simple problems for practicing your newly acquired MATLAB skills. Solutions are presented at the end of the book.

Chapter 4, *Beyond the Basics*, contains an explanation of some of the finer points that are essential for using MATLAB effectively.

Chapter 5, *MATLAB Graphics*, contains a more detailed look at many of the MATLAB commands for producing graphics.

Practice Set B, *Calculus, Graphics, and Linear Algebra*, gives you another chance to practice what you've just learned. As before, solutions are provided at the end of the book.

Chapter 6, *Programming*, introduces you to the programming features of MATLAB. This chapter is designed to be useful both to the novice programmer and to the experienced Fortran or C programmer.

Chapter 7, *Publishing and M-Books*, contains an introduction to the word-processing and desktop-publishing features available in MATLAB 7, either using M-files and the **`publish`** command or else combining MATLAB with Microsoft Word.

Chapter 8, *Simulink*, describes the MATLAB companion software Simulink, a graphically oriented package for modeling, simulating, and analyzing dynamical systems. Many of the calculations that can be done with MATLAB can be done equally well with Simulink. If you don't have access to Simulink, you may skip Chapter 8.

Chapter 9, *GUIs*, contains an introduction to the construction and deployment of Graphical User Interfaces, that is *GUI*s, using MATLAB. This chapter is a little more advanced than most of the others.

Chapter 10, *Applications*, contains examples, from many different fields, of solutions of real-world problems using MATLAB and/or Simulink.

Practice Set C, *Developing Your MATLAB Skills*, contains practice problems whose solutions use the methods and techniques you learned in Chapters 6–10.

Chapter 11, *Troubleshooting*, is the place to turn when anything goes wrong. Many common problems can be resolved by reading (and rereading) the advice in this chapter.

Next, we have *Solutions to the Practice Sets*, which contains solutions to all the problems from the three Practice Sets. The *Glossary* contains short descriptions (with examples) of many MATLAB commands and objects. Though it is not a complete reference, the Glossary is a handy guide to the most important features of MATLAB. Finally, there is a comprehensive *Index*.

Conventions Used in This Book

We use distinct fonts to distinguish various entities. When new terms are first introduced, they are typeset in an *italic* font. Output from MATLAB is typeset in a `monospaced typewriter` font; commands that you type for interpretation by MATLAB are indicated by a **`boldface`** version of that font. These commands and responses are often displayed on separate lines as they would be in a MATLAB session, as in the following example:

```
>> x = sqrt(2*pi + 1)

x =
   2.697
```

Selectable menu items (from the menu bars in the MATLAB Desktop, figure windows, etc.) are typeset in a **boldface** font. Submenu items are separated from menu items by a colon, as in **File**:**Open...**. Labels such as the names of windows and buttons are quoted, in a "regular" font. File and folder names, as well as web addresses, are printed in a `typewriter` font. Finally, names of keys on your computer keyboard are set in a SMALL CAPS font.

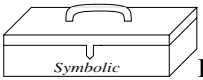
We use six special symbols throughout the book. Here they are, together with their meanings.

☞ *Paragraphs like this one contain cross-references to other parts of the book, or suggestions of where you can skip ahead to another chapter.*

⇨ **Paragraphs like this one contain important notes. Our favorite is "Save your work frequently." Pay careful attention to these paragraphs.**

✓ Paragraphs like this one contain useful tips or point out features of interest in the surrounding landscape. You might not need to think carefully about them on a first reading, but they may draw your attention to some of the finer points of MATLAB if you go back to them later.

☆ Chapters, sections, or problems starting with this symbol are a little more advanced than the rest of the book, and can be skipped on first reading if you wish.



Paragraphs like this discuss features of MATLAB's Symbolic Math Toolbox, used for *symbolic* (as opposed to *numerical*) calculations. If you are not using the Symbolic Math Toolbox, you can skip these sections.

Paragraphs like this discuss features of Simulink. If you are not using Simulink, you can skip these sections.

Incidentally, if you are a student and you have purchased the MATLAB Student Version, then the Symbolic Math Toolbox and Simulink are automatically included with your software, along with basic MATLAB.

About the Authors

We are mathematics professors at the University of Maryland, College Park. We have used MATLAB in our research, in our mathematics courses, for presentations and demonstrations, for production of graphics for books and for the Web, and even to help our kids do their homework. We hope you'll find MATLAB as useful as we do, and that this book will help you learn to use it quickly and effectively.

Acknowledgment and Disclaimer. We are pleased to acknowledge support of our research by the National Science Foundation, which contributed over many years to the writing of this book. Our work on the second edition was partially supported by NSF Grants DMS-0103647, DMS-0104087, ATM-0434225, and DMS-0504212. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Brian R. Hunt
Ronald L. Lipsman
Jonathan M. Rosenberg

College Park, Maryland
January, 2006

Chapter 1

Getting Started

In this chapter, we will introduce you to the tools you need in order to begin using MATLAB effectively. These include the following: some relevant information on computer platforms and software; installation protocols; how to launch MATLAB, enter commands and use online help; a roster of MATLAB's various windows; and finally, how to exit the program. We know you are anxious to get started using MATLAB, so we will keep this chapter brief. After you complete it, you can go immediately to Chapter 2 to find concrete and simple instructions for using MATLAB to do mathematics. We describe the MATLAB interface more elaborately in Chapter 3.

Platforms and Versions

It is likely that you will use MATLAB on a computer running Microsoft Windows or on some form of a UNIX operating system (such as Linux). Some previous versions of MATLAB (Releases 11 and 12) did not support Macintosh, but the most current versions (Releases 13 and 14) do. If you are using a Macintosh, you should find that our instructions for Windows will suffice for most of your needs. Like MATLAB 6 (Releases 12 and 13), and unlike earlier versions, MATLAB 7 (Release 14) looks virtually identical on these different platforms. For definitiveness, we shall assume that the reader is using a Windows computer. In those very few instances where our instructions must be tailored differently for Linux, UNIX, or Macintosh users, we shall point it out clearly.

✓ We use the word Windows to refer to all flavors of the Windows operating system. MATLAB 7 (R14) *will* run on Windows 2000, Windows NT (4.0 and higher), and on Windows XP. MATLAB 7 will *not* run on Windows 95, Windows 98, or on Windows ME. However, MATLAB 6.5 (R13) *does* run on Windows 98 or Windows ME.

This book is written to be compatible with the current version of MATLAB, namely MATLAB 7 (R14). The vast majority of the MATLAB commands we describe, as well as many features of the MATLAB interface (e.g., the M-file Editor/Debugger and M-books) are valid for version 6.5 (R13), and earlier versions in some cases. When important differences between different versions arise, we will point them out. We also note that the differences between the MATLAB Professional Version and the MATLAB Student Version are rather minor, and virtually unnoticeable to a beginner, or even a mid-level user. Again, in the few instances where we

describe a MATLAB feature that is available only in the Professional Version, we highlight that fact clearly.

Installation

If you intend to run MATLAB, especially the Student Version, on your own computer, it is quite possible that you will have to install it yourself. You can easily accomplish this using the product CDs. Follow the installation instructions as you would with installation of any new software. At some point in the installation you may be asked which *toolboxes* you wish to install. Unless you have severe space limitations, we suggest that you install any that seem of interest to you or that you think you might use at some point in the future. However, for the purposes of this book, you should be sure to include the *Symbolic Math Toolbox*. We also strongly encourage you to install Simulink, which is described in Chapter 8.

Starting MATLAB

You start MATLAB as you would any other software application. In Windows, you access it via the **Start** menu, under a name like **MATLAB 7.0** or **Student MATLAB**. Alternatively, you may have a desktop icon that enables you to start MATLAB with a simple double-click. In Linux or UNIX, generally you need only type **`matlab`** in a terminal window, though you may first have to find the `bin` subdirectory of the MATLAB installation directory and add it to your path. Or you may have an icon on your desktop that achieves the task.

However you start MATLAB, you will briefly see a window that displays the MATLAB logo as well as some product information, and then a *MATLAB Desktop* window will launch. That window will contain a *title bar*, a *menu bar*, a *tool bar* and four embedded windows, one of which is hidden. The largest and most important window is the *Command Window* on the right. We will go into more detail in Chapter 3 on the use and manipulation of the other three windows: the *Command History Window*, the *Current Directory Browser* and the *Workspace Browser*. For now we concentrate on the Command Window in order to get you started issuing MATLAB commands as quickly as possible. At the top of the Command Window, you may see some general information about MATLAB, perhaps some special instructions for getting started or accessing help, but most important of all, you will see a *command prompt* (**`>>`** or **`EDU>>`**). If the Command Window is "active," its title bar will be dark, and the prompt will be followed by a cursor (a blinking vertical line). That is the place where you will enter your MATLAB commands (see Chapter 2). If the Command Window is not active, just click in it anywhere. Figure 1.1 contains an example of a newly launched MATLAB Desktop.

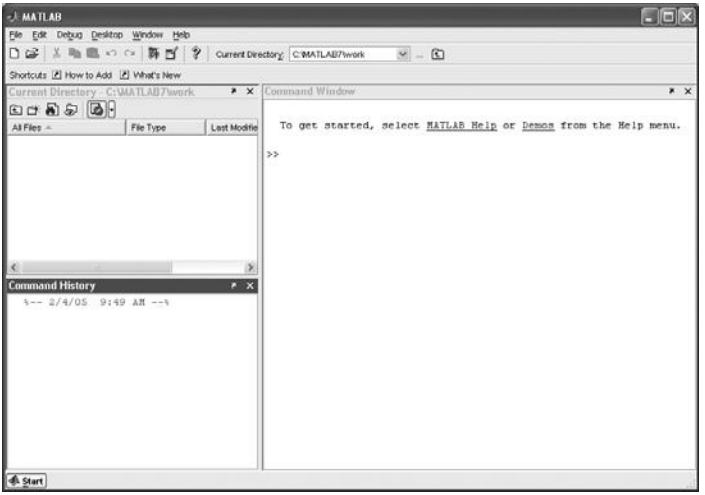


Figure 1.1. A MATLAB Desktop.

✓ MATLAB 6 has a Desktop, but in older versions of MATLAB, for example 5.3, there was no integrated Desktop. Only the Command Window appeared when you launched the application. (On UNIX systems, the terminal window from which you invoked MATLAB 5 became the Command Window.) Commands that we instruct you to enter in the Command Window inside the Desktop for MATLAB can be entered directly into the Command Window in any older version.

Typing in the Command Window

Click in the Command Window to make it active. When a window becomes active, its title bar darkens and a blinking cursor appear after the prompt. Now you can begin entering commands. Try typing **`2+2`**, then press the ENTER or RETURN key. Next try **`factor(123456789)`**, and finally **`sin(100)`**. Your MATLAB Desktop should look like Figure 1.2.

Online Help

MATLAB has extensive online help. In fact, using only this book and the online help, you should be able to become quite proficient with MATLAB.

You can access the online help in one of several ways. Typing **`help`** at the command prompt will reveal a long list of topics for which help is available. Just to illustrate, try typing **`help general`**. Now you see a long list of "general purpose" MATLAB commands. Next, try **`help factor`** to learn about the **`factor`** command. In every instance above, more information than your screen can hold will

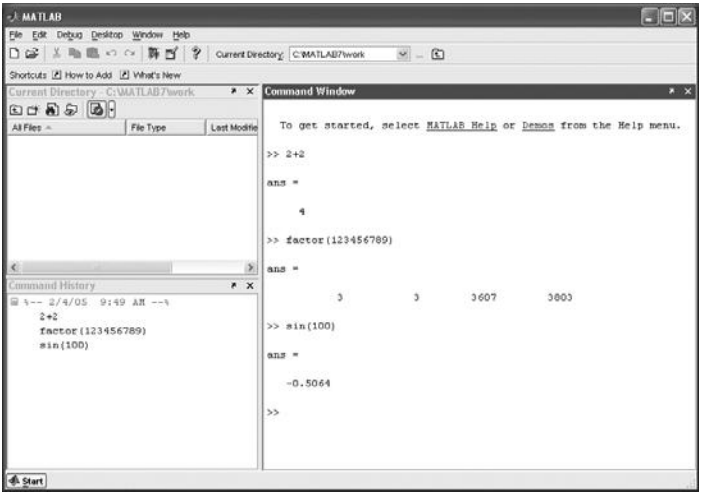


Figure 1.2. The MATLAB Desktop with Several Commands Evaluated.

scroll by. You can use the scroll bar on the right of the window to scroll back up. Alternatively, you can force MATLAB to display information one screenful at a time by typing **`more on`**. You press the space bar to display the next screenful; type **`help more`** for details. Typing **`more on`** affects all subsequent commands, until you type **`more off`**.

The **`lookfor`** command searches the first line of every MATLAB help file for a specified string (use **`lookfor -all`** to search all lines). For example, if you wanted to see a list of all MATLAB commands that contain the word "factor" as part of the command name or brief description, then you would type **`lookfor factor`**. If the command you are looking for appears in the list, then you can use **`help`** on that command to learn more about it.

While **`help`** in the Command Window is useful for getting quick information on a particular command, more extensive documentation is available via the MATLAB *Help Browser*. You can activate it in several ways, for example, by typing **`doc`** at the command prompt. Alternatively, it is available through the menu bar under **Help**. Finally, the question-mark button on the tool bar will also invoke the Help Browser. Upon its launch you will see two windows or *panes*. The first is called the *Help Navigator*; it is used to find documentation. The second, called the *display pane*, is used for viewing documentation. The display pane works much like a normal web browser. It has an address window, buttons for moving forward and back (among the windows you have visited), hyperlinks for moving around in the documentation, the capability of storing favorite pages, and other useful tools.

A particularly useful way to invoke the Help Browser is to type, for example, **`doc sin`**. This launches the Help Browser and displays the reference page for **`sin`**. The reference page for a command is generally similar to the text available (through **`help`**), but sometimes has more information.

You can also use the Help Navigator to locate the documentation that you will explore in the display pane. The Help Navigator has four tabs that allow you to arrange your search for documentation in different ways. The first is the *Contents* tab that displays a tree view of all the documentation topics available. The extent of that tree will be determined by how much you (or your system administrator) included in the original MATLAB installation (how many toolboxes, etc.). The second tab is an *Index* that displays all available documentation in index format. It responds to your key entry of likely items you want to investigate in the usual alphabetic reaction mode. The third tab provides the *Search* mechanism. You type in what you seek, either a function or some other descriptive term, and the search engine locates documentation pertaining to your entry. Finally, the fourth tab is a small collection of *Demos* that you can run to learn more about MATLAB. Clicking on an item that appears in any of these tabs brings up the corresponding documentation in the display pane.

The Help Browser has an excellent tutorial describing its own operation. To view it, open the browser and select **Help**:**Using the Help Browser** from its menu bar. The Help Browser is a powerful and easy-to-use aid in finding the information you need on various features of MATLAB. Like any such tool, the more you use it, the more adept you become at its use. Figure 1.3 depicts the *MATLAB Help Browser*.

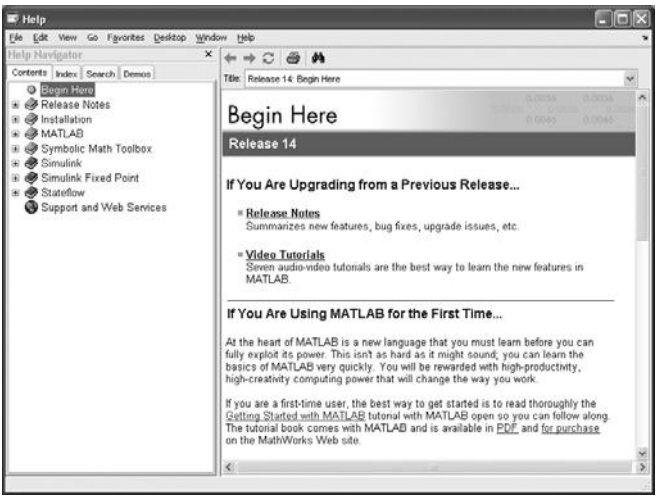


Figure 1.3. The MATLAB Help Browser.

✓ If you are working with MATLAB version 5.3 or earlier, then typing **`help`**, **`help general`** or **`help factor`** at the command prompt will work as indicated above. The Help Browser is not available, but the commands **`helpwin`** and **`helpdesk`** call up more primitive, although still quite useful, forms of help windows.

If you are patient, and not overly anxious to get to Chapter 2, you can type **`demo`** to try some of MATLAB's online demonstrations.

MATLAB Windows

We have already described the MATLAB Command Window and the Help Browser, and have mentioned in passing the Command History Window, the Current Directory Browser, and the Workspace Browser. These, and several other windows you will encounter as you work with MATLAB, will allow you to do the following: control files and folders that you and MATLAB will need to access; write and edit the small MATLAB programs (M-files) that you will use to run MATLAB effectively; keep track of the variables and functions that you define as you use MATLAB; display and manipulate graphics; and design models to solve problems and simulate processes. Some of these windows launch separately, and some are embedded in the Desktop. You can dock into the Desktop those that launch separately through their **Desktop** menus, or by clicking on the downward curved arrow in their tool bars. You can separate windows inside your MATLAB Desktop out to your computer desktop by clicking on the upward curved arrow in the upper right-hand corner of the window's tool bar.

These features are described more thoroughly in later chapters. For now, we want to call your attention to the other main type of window you will encounter, namely graphics windows. Many of the commands you issue will generate graphics or pictures. These will appear in a separate window, called a *figure window*. In Chapter 5, we will teach you how to generate and manipulate MATLAB figure windows most effectively.

See Figure 2.1 in Chapter 2 for a simple example of a figure window.

✓ In MATLAB 6 or earlier versions, you cannot dock figure windows. Nor can you in version 7 if you are using a Macintosh.

Ending a Session

The simplest way to conclude a MATLAB session is to type **`quit`** at the prompt. You can also click on the button that generally closes your windows (usually an $\times$ in the upper right-hand corner). Still another way to exit is to use the **Exit MATLAB** item from the **File** menu of the Desktop. *Before* you exit MATLAB, you should be sure to save your work, print any graphics or other files you need, and in general clean up after yourself. Some strategies for doing so are discussed in Chapter 3.

Chapter 2

MATLAB Basics

In this chapter, you will start learning how to use MATLAB to do mathematics. We recommend that you read this chapter while running MATLAB. Try the commands as you go along. Feel free to experiment with variants of the examples we present; the best way to find out how MATLAB responds to a command is to try it.

☞ *For further practice, you can work the problems in* Practice Set A. *You may also consult the* Glossary *for a synopsis of many MATLAB operators, constants, functions, commands, and programming instructions.*

Input and Output

You input commands to MATLAB in the Command Window. MATLAB returns output in two ways: typically, text or numerical output is returned in the same Command Window, but graphical output appears in a separate figure window. A sample screen, with both a MATLAB Desktop and a figure window, labeled "Figure 1", is shown in Figure 2.1. To generate this screen on your computer, first type **`1/2 + 1/3`**. Then type **`ezplot('x^3 - x')`**.

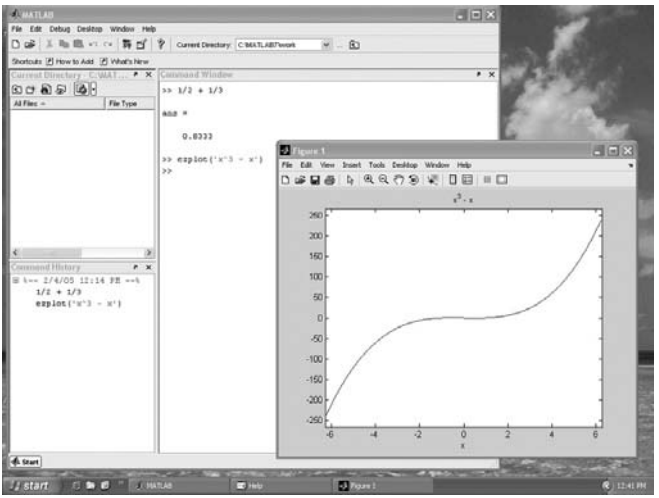


Figure 2.1. MATLAB Windows.

✓ While MATLAB is working, it may display a "wait" symbol – for example, an hourglass appears on many operating systems. Or it may give no visual evidence until it is finished with its calculation.

Arithmetic

As we have just seen, you can use MATLAB to do arithmetic as you would a calculator. You can add with **+**, subtract with **-**, multiply with *****, divide with **/**, and exponentiate with **^**. For example:

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

ans =
   22.5000
```

MATLAB prints the answer and assigns the value to a *variable* called **ans**. If you want to perform further calculations with the answer, you can use the variable **ans** rather than retype the answer. For example, you can compute the sum of the square and the square root of the previous answer as follows:

```
>> ans^2 + sqrt(ans)

ans =
   510.9934
```

Observe that MATLAB assigns a new value to **ans** with each calculation. To do more complex calculations, you can assign computed values to variables of your choosing. For example:

```
>> u = cos(10)

u =
   -0.8391

>> v = sin(10)

v =
   -0.5440

>> u^2 + v^2

ans =
   1
```

✓ Notice that trigonometric functions in MATLAB use radians, not degrees.

MATLAB uses double-precision floating-point arithmetic, which is accurate to approximately 15 digits; however, MATLAB displays only 5 digits by default. To display more digits, type **format long**. Then all subsequent numerical output will have 15 digits displayed. Type **format short** to return to 5-digit display.

MATLAB differs from a calculator in that it can do *exact* arithmetic. For example, it can add the fractions $1/2$ and $1/3$ symbolically to obtain the correct fraction $5/6$. We discuss how to do this in the section *Symbolic Expressions, Variable Precision, and Exact Arithmetic* later in this chapter.

Recovering from Problems

Inevitably, when using any mathematical software system, you are bound to encounter minor glitches. Even while entering simple arithmetic commands, you may accidentally mistype an entry or inadvertently violate a MATLAB rule. In this brief section, we mention three methods for coping with these kinds of problems.

Errors in Input

If you make an error in an input line, MATLAB will normally print an error message. For example, here's what happens when you try to evaluate **`3u^2`**:

```
>> 3u^2

??? 3u^2
     |
Error: Missing MATLAB operator.
```

The error is a missing multiplication operator **`*`**. The correct input would be **`3*u^2`**. Note that MATLAB places a marker (a vertical line segment) at the place where it thinks the error might be; however, the actual error may have occurred earlier or later in the expression.

✓ The error message generated in MATLAB 7.0.4 (the latest upgrade as this is written) is somewhat different. You will find that error and warning messages you encounter may vary with versions of the program.

⇨ **Missing multiplication operators and parentheses are among the most common input errors made by beginning users.**

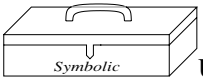
☞ *Recall our discussion of online help in Chapter 1. If you can't decipher an error message caused by a MATLAB command, refer to the online help for that command, or use the search feature of the Help Browser.*

You can edit an input line by using the UP-ARROW key to redisplay the previous command, editing the command using the LEFT- and RIGHT-ARROW keys, and then pressing RETURN or ENTER. The UP- and DOWN-ARROW keys allow you to scroll back and forth through all the commands you've typed in a MATLAB session, and are very useful when you want to correct, modify, or reenter a previous command.

Aborting Calculations

If MATLAB gets hung up in a calculation, or seems to be taking too long to perform an operation, you can usually abort it by typing CTRL+C; that is, hold down the key labeled CTRL, or CONTROL, and press C. While not foolproof, this is the method of choice when MATLAB is not responding.

Algebraic or Symbolic Computation



Using MATLAB's Symbolic Math Toolbox, you can carry out algebraic or symbolic calculations such as factoring polynomials or solving algebraic equations. Type **`help symbolic`** to make sure that the Symbolic Math Toolbox is installed on your system.

To perform symbolic computations, you must use **`syms`** to declare the variables you plan to use to be symbolic variables. Consider the following series of commands:

```
>> syms x y
>> (x - y)*(x - y)^2

ans =
(x-y)^3

>> expand(ans)

ans =
x^3-3*x^2*y+3*x*y^2-y^3

>> factor(ans)

ans =
(x-y)^3
```

✓ Notice that symbolic output is left-justified, while numerical output is indented. This feature is often useful in distinguishing symbolic output from numerical output.

Although MATLAB makes minor simplifications to the expressions you type, it does not make major changes unless you tell it to. The command **`expand`** told MATLAB to multiply out the expression, and **`factor`** forced MATLAB to restore it to factored form.

MATLAB has a command called **`simplify`**, which you can sometimes use to express a formula as simply as possible. For example,

```
>> simplify((x^3 - y^3)/(x - y))

ans =
x^2+x*y+y^2
```

✓ MATLAB has a more robust command, called **`simple`**, that sometimes does a better job than **`simplify`**. Try both commands on the trigonometric expression **`sin(x)*cos(y) + cos(x)*sin(y)`** to compare – you'll have to read the online help for **`simple`** to completely understand the answer.

Substituting in Symbolic Expressions

When you work with symbolic expressions you often need to substitute a numerical value, or even another symbolic expression, for one (or more) of the original variables

in the expression. This is done by using the **subs** command. For example, presuming that the symbol expression **w** is defined and it involves the symbolic variable **u**, then the command **subs(w, u, 2)** will substitute the value 2 for the variable **u** in the expression **w**. More examples:

```
>> d = 1, syms u v

d =
   1

>> w = u^2 - v^2

w =
u^2-v^2

>> subs(w, u, 2)

ans =
4-v^2

>> subs(w, v, d)

ans =
u^2-1

>> subs(w, v, u + v)

ans =
u^2-(u+v)^2

>> simplify(ans)

ans =
-2*u*v-v^2
```

✓ When you enter multiple commands on a single line separated by commas, MATLAB evaluates each command and displays the output on separate lines. In the first input line above, the second command produced no output, so we only saw output from the first command.

Symbolic Expressions, Variable Precision, and Exact Arithmetic

As we have noted, MATLAB uses floating-point arithmetic for its calculations. Using the Symbolic Math Toolbox, you can also do exact arithmetic with symbolic expressions. Consider the following example:

```
>> cos(pi/2)

ans =
   6.1232e-17
```

The answer is written in floating-point format and means 6.1232×10^{-17}. However, we know that $\cos(\pi/2)$ is really equal to 0. The inaccuracy is due to the fact that typing **pi** in MATLAB gives an approximation to π accurate to about 15 digits, not its exact value. To compute an exact answer, instead of an approximate answer, we

must create an exact *symbolic* representation of $\pi/2$ by typing **sym('pi/2')**. Now let's take the cosine of the symbolic representation of $\pi/2$:

```
>> cos(sym('pi/2'))
ans =
0
```

This is the expected answer.

The quotes around **pi/2** in **sym('pi/2')** create a *string* consisting of the characters **pi/2** and prevent MATLAB from evaluating **pi/2** as a floating-point number. The command **sym** converts the string to a symbolic expression.

The commands **sym** and **syms** are closely related. In fact, **syms x** is equivalent to **x = sym('x')**. The command **syms** has a lasting effect on its argument. In fact, even if **x** was previously defined, **syms x** clears that definition and renders **x** a symbolic variable – which it remains until (if ever) it is redefined. On the other hand, **sym** has only a temporary effect unless you assign the output to a variable, as in **x = sym('x')**.

Here is how to add $1/2$ and $1/3$ symbolically:

```
>> sym('1/2') + sym('1/3')
ans =
5/6
```

Finally, you can also do *variable-precision arithmetic* with **vpa**. For example, to print 50 digits of $\sqrt{2}$, type:

```
>>  vpa('sqrt(2)', 50)
ans =
1.4142135623730950488016887242096980785696718753769
```

If you don't specify the number of digits, the default setting is 32. You can change the default with the command **digits**.

⇨ **You should be wary of using sym or vpa on an expression that MATLAB must evaluate before applying variable-precision arithmetic. To illustrate, evaluate the expressions 3^45, vpa(3^45), and vpa('3^45'). The first gives a floating-point approximation to the answer, the second – because MATLAB carries only 16-digit precision in its floating-point evaluation of the exponentiation – gives an answer that is correct only in its first 16 digits, and the third gives the exact answer.**

☞ *See the section* Symbolic and Floating-Point Numbers *in Chapter 4 for details about how MATLAB converts between symbolic and floating-point numbers.*

Vectors and Matrices

MATLAB was written originally to allow mathematicians, scientists, and engineers to handle the tools of linear algebra – that is, vectors and matrices – as effortlessly as possible. In this section we introduce these concepts.

Vectors

A *vector* is an ordered list of numbers. You can enter a vector of any length in MATLAB by typing a list of numbers, separated by commas and/or spaces, inside square brackets. For example:

```
>> Z = [2,4,6,8]

Z =
   2     4     6     8

>> Y = [4 -3 5 -2 8 1]

Y =
   4    -3     5    -2     8     1
```

Suppose that you want to create a vector of values running from 1 to 9. Here's how to do it without typing each number:

```
>> X = 1:9

X =
   1     2     3     4     5     6     7     8     9
```

The notation **1:9** is used to represent a vector of numbers running from 1 to 9 in increments of 1. The increment can be specified as the middle of three arguments:

```
>> X = 0:2:10

X =
   0     2     4     6     8    10
```

Increments can be fractional or negative, for example, **0:0.1:1** or **100:-1:0**.

The elements of the vector **X** can be extracted as **X(1)**, **X(2)**, etc. For example:

```
>> X(3)

ans =
   4
```

To change the vector **X** from a row vector to a column vector, put a prime (**'**) after **X**:

```
>> X'

ans =
    0
    2
    4
    6
    8
   10
```

You can perform mathematical operations on vectors. For example, to square the elements of the vector **X**, type

```
>> X.^2

ans =
   0     4    16    36    64   100
```

The period in this expression is very important; it says that the numbers in **X** should be squared individually, or *element-by-element*. Typing **X^2** would tell MATLAB to use matrix multiplication to multiply **X** by itself and would produce an error message in this case. (We discuss matrices below and in Chapter 4.) Similarly, you must type **.*** or **./** if you want to multiply or divide vectors element-by-element. For example, to multiply the elements of the vector **X** by the corresponding elements of the vector **Y**, type

```
>> X.*Y

ans =
   0     -6     20     -12     64     10
```

Most MATLAB operations are, by default, performed element-by-element. For example, you do not type a period before the addition and subtraction operators, and you can type **exp(X)** to get the exponential of each number in **X** (the matrix exponential function is **expm**). One of the strengths of MATLAB is its ability to efficiently perform operations on vectors.

Matrices

A *matrix* is a rectangular array of numbers. Row and column vectors, which we discussed above, are examples of matrices. Consider the 3×4 matrix

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}.$$

It can be entered in MATLAB with the command

```
>> A = [1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12]

A =
   1     2     3     4
   5     6     7     8
   9    10    11    12
```

Note that the matrix *elements* in any row are separated by commas, and the rows are separated by semicolons. The elements in a row can also be separated by spaces.

If two matrices **A** and **B** are the same size, their (element-by-element) sum is obtained by typing **A + B**. You can also add a scalar (a single number) to a matrix; **A + c** adds **c** to each element in **A**. Likewise, **A - B** represents the difference of **A** and **B**, and **A - c** subtracts the number **c** from each element of **A**. If **A** and **B** are multiplicatively compatible, i.e., if **A** is $n \times m$ and **B** is $m \times \ell$, then their product **A*B** is $n \times \ell$. Recall that the element of **A*B** in the ith row and jth column is the sum of the products of the elements from the ith row of **A** times the elements from the jth column of **B**, i.e.,

$$(\mathbf{A*B})_{ij} = \sum_{k=1}^{m} \mathbf{A}_{ik}\mathbf{B}_{kj},\ 1 \le i \le n,\ 1 \le j \le \ell.$$

The product of a number **c** and the matrix **A** is given by **c*A**, and **A'** represents the conjugate transpose of **A**. (For more information, see the online help for **ctranspose** and **transpose**.)

A simple illustration is given by the matrix product of the 3×4 matrix **A** above by the 4×1 column vector **Z'**:

```
>> A*Z'

ans =
    60
   140
   220
```

The result is a 3×1 matrix, in other words, a column vector.

☞ *MATLAB has many commands for manipulating matrices. You can read about them in the section* More about Matrices *in Chapter 4 and in the online help; some of them are illustrated in the section* Linear Economic Models *in Chapter 10.*

Suppressing Output

Typing a semicolon at the end of an input line suppresses printing of the output of the MATLAB command. The semicolon should generally be used when defining large vectors or matrices (such as **X = -1:0.1:2;**). It can also be used in any other situation where the MATLAB output need not be displayed.

Functions

In MATLAB you will use both built-in functions and functions that you create yourself.

Built-in Functions

MATLAB has many built-in functions. These include **sqrt**, **cos**, **sin**, **tan**, **log**, **exp**, and **atan** (for $\arctan$) as well as more specialized mathematical functions like **gamma**, **erf**, and **besselj**. MATLAB also has several built-in constants, including **pi** (the number π), **i** (the complex number $i = \sqrt{-1}$), and **Inf** (∞). Some examples:

```
>> log(exp(3))

ans =
   3
```

The function **log** is the natural logarithm, called "$\ln$" in many texts.

```
>> sin(2*pi/3)

ans =
   0.8660
```

To get an exact answer, you need to use a symbolic argument.

```
>> sin(sym('2*pi/3'))

ans =
1/2*3^(1/2)
```

User-Defined Functions

In this section we will examine two methods to define your own functions in MATLAB. The first uses the command **inline**, and the second uses the operator **@** to create what is called an "anonymous function." The latter method is new in MATLAB 7, and is now the preferred method. We shall mention **inline** periodically for the sake of users of earlier versions, but we strongly recommend that users of MATLAB 7, and users of older versions when they upgrade, adopt **@** as the usual method for defining their functions. Functions can also be defined in separate files called M-files – see Chapter 3.

Here's how to define the function $f(x) = x^2$ using these commands.

```
>> f = @(x) x^2

f =
   @(x) x^2
```

Alternatively,

```
>> f1 = inline('x^2', 'x')

f1 =
   Inline function:
   f1(x) = x^2
```

Once the function is defined – by either method – you can evaluate it:

```
>> f(4)

ans =
   16

>> f1(4)

ans =
   16
```

As we observed earlier, most MATLAB functions can operate on vectors as well as scalars. To insure that your user-defined function can act on vectors, insert dots before the mathematical operators *****, **/**, and **^**. Thus to obtain a vectorized version of $f(x) = x^2$, type either

```
>> f = @(x) x.^2
```

or else

```
>> f1 = inline('x.^2', 'x')
```

Now we can evaluate either function on a vector, for example

```
>> f(1:5)
```

```
ans =
   1     4     9     16     25
```

You can plot **f** and **f1**, using MATLAB graphics, in several ways that we will explore in the section *Graphics* later in this chapter. We conclude this section by remarking that one can also define functions of two or more variables. For example, either of the following

```
>> g = @(x, y) x^2 + y^2; g(1, 2)
>> g1 = inline('x^2 + y^2', 'x', 'y'); g1(1, 2)
```

results in the answer 5. If instead you define

```
>> g = @(x, y) x.^2 + y.^2;
```

then you can evaluate on vectors; thus

```
>> g([1 2], [3 4])
ans =
   10     20
```

gives the values of the function at the points $(1, 3)$ and $(2, 4)$.

Managing Variables

We have now encountered four different types of MATLAB data: floating-point numbers, strings, symbolic expressions, and functions. In a long MATLAB session it may be hard to remember the names and types of all the variables you have defined. You can type **whos** to see a summary of the names and types, or *classes*, of your currently defined variables. But before you do that, first assign **a = pi**, **b = 'pi'**, and **c = sym('pi')**; now type **whos**. Here's the output for the MATLAB session displayed in this chapter.

```
>> whos
  Name      Size        Bytes  Class

  A         3x4            96  double array
  X         1x31          248  double array
  Y         1x6            48  double array
  Z         1x4            32  double array
  a         1x1             8  double array
  ans       1x2            16  double array
  b         1x2             4  char array
  c         1x1           128  sym object
  d         1x1             8  double array
  f         1x1            16  function_handle array
  f1        1x1           824  inline object
  g         1x1            16  function_handle array
  g1        1x1           882  inline object
  u         1x1           126  sym object
  v         1x1           126  sym object
  w         1x1           138  sym object
```

```
  x         1x1          126  sym object
  y         1x1          126  sym object

Grand total is 183 elements using 2968 bytes
```

The variables **A**, **X**, **Y**, **Z**, **a**, and **d** were assigned numerical data and are reported as "double array". That means that they are arrays of double-precision numbers; in this case the arrays **a** and **d** are of size 1×1, i.e., scalars. The "Bytes" column shows how much computer memory is allocated to each variable. Also, **ans** is numerical since the last unassigned output was a 1×2 vector. The variable **b** is a string, reported as "char array", while the variables **c**, **u**, **v**, **w**, **x** and **y** are symbolic. Finally we see two function handles and two inline objects, corresponding to the pairs of anonymous functions and inline functions.

The command **whos** shows information about all defined variables, but it does not show the values of the variables. To see the value of a variable, simply type the name of the variable and press ENTER or RETURN.

MATLAB commands expect particular classes of data as input, and it is important to know what class of data is expected by a given command; the help text for a command usually indicates the class or classes of input it expects. The wrong class of input usually produces an error message or unexpected output. For example, type **sin('pi')** to see how unexpected output can result from supplying a string to a function that isn't designed to accept strings.

To clear all defined variables, type **clear** or **clear all**. You can also type, for example, **clear x y** to clear only **x** and **y**.

You should generally clear variables before starting a new calculation. Otherwise values from a previous calculation can creep into the new calculation by accident. Finally, we observe that the *Workspace Browser* presents a graphical alternative to **whos**. You can activate it by clicking on its tab under the Current Directory Browser, or by typing **workspace** at the command prompt. Figure 2.2 depicts a Desktop in which the Command Window and the Workspace Browser contain the same information as displayed above.

Variables and Assignments

In MATLAB, you use the *equal sign* to assign values to a variable. For instance,

```
>> x = 7

x =
   7
```

will give the variable **x** the value 7 from now on. Henceforth, whenever MATLAB sees the letter **x**, it will substitute the value 7. For example, if **y** has been defined as a symbolic variable:

```
>> x^2 - 2*x*y + y

ans =
49-13*y
```

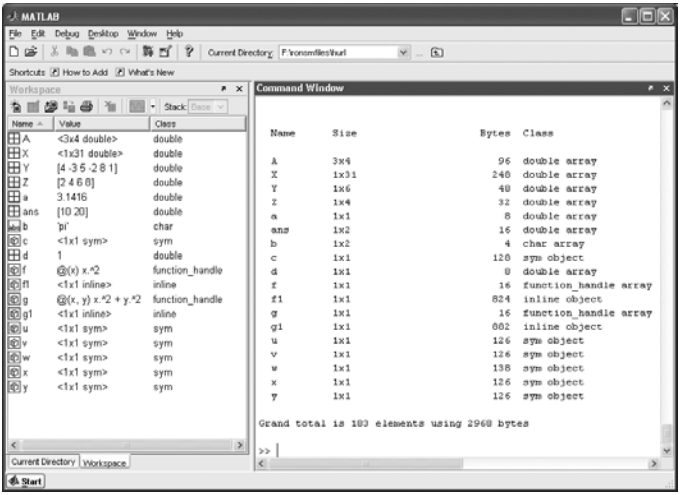


Figure 2.2. Desktop with the Workspace Browser.

You can make very general assignments for symbolic variables and then manipulate them:

```
>> syms x y
>> z = x^2 - 2*x*y + y

z =
x^2-2*x*y+y

>> 5*y*z

ans =
5*y*(x^2-2*x*y+y)
```

A variable name or function name can be any string of letters, digits, and underscores, provided that it begins with a letter (punctuation marks are not allowed). MATLAB distinguishes between uppercase and lowercase letters. You should choose distinctive names that are easy for you to remember, generally using lowercase letters. For example, you might use **`cubicsol`** as the name of the solution of a cubic equation.

⇨ **A common source of puzzling errors is the inadvertent reuse of previously defined variables.**

MATLAB never forgets your definitions unless instructed to do so. You can check on the current value of a variable by simply typing its name.

Solving Equations

You can solve equations involving variables with **solve** or **fzero**. For example, to find the solutions of the quadratic equation $x^2 - 2x - 4 = 0$, type

```
>> solve('x^2 - 2*x - 4 = 0')
```

```
ans =
 5^(1/2)+1
 1-5^(1/2)
```

Note that the equation to be solved is specified as a string; i.e., it is surrounded by single quotes. The answer consists of the exact (symbolic) solutions $1 \pm \sqrt{5}$. To get numerical solutions, type **double (ans)**, or **vpa(ans)** to display more digits. The input to **solve** can also be a symbolic expression, but then MATLAB requires that the right-hand side of the equation be 0, and in fact the syntax for solving $x^2 - 3x = -7$ is:

```
>> syms x
>> solve(x^2 - 3*x + 7)
```

```
ans =
 3/2+1/2*i*19^(1/2)
 3/2-1/2*i*19^(1/2)
```

The answer consists of the exact (symbolic) solutions $(3 \pm \sqrt{19}i)/2$ (complex numbers, where the letter **i** in the answer stands for the imaginary unit $\sqrt{-1}$). To get numerical solutions, type **double(ans)**, or **vpa(ans)** to display more digits.

The command **solve** can solve higher-degree polynomial equations, as well as many other types of equations. It can also solve equations involving more than one variable. If there are fewer equations than variables, you should specify (as strings) which variable(s) to solve for. For example, type **solve('2*x - log(y) = 1', 'y')** to solve $2x - \log y = 1$ for y in terms of x. You can specify more than one equation as well. For example:

```
>> [x, y] = solve('x^2 - y = 2', 'y - 2*x = 5')
```

```
x =
 1+2*2^(1/2)
 1-2*2^(1/2)
y =
 7+4*2^(1/2)
 7-4*2^(1/2)
```

This system of equations has two solutions. MATLAB reports the solution by giving the two x-values and the two y-values for those solutions. Thus the first solution consists of the first value of x together with the first value of y. You can extract these values by typing **x(1)** and **y(1)**:

```
>> x(1)
```

```
ans =
1+2*2^(1/2)
```

```
>> y(1)
```

```
ans =
7+4*2^(1/2)
```

The second solution can be extracted with **`x(2)`** and **`y(2)`**.

Note that, in the preceding **`solve`** command, we assigned the output to the *vector* **`[x, y]`**. If you use **`solve`** on a system of equations without assigning the output to a vector, then MATLAB does not automatically display the values of the solution:

```
>> sol = solve('x^2 - y = 2', 'y - 2*x = 5')

sol =
 x: [2x1 sym]
 y: [2x1 sym]
```

To see the vectors of x- and y-values of the solution, type **`sol.x`** and **`sol.y`**. To see the individual values, type **`sol.x(1)`**, **`sol.y(1)`**, etc.

☞ *In this example, the output of* **`solve`** *is a* structure array. *See the section* Cell and Structure Arrays *in Chapter 6 for more about this data class.*

Some equations cannot be solved symbolically, and in these cases **`solve`** tries to find a numerical answer. For example:

```
>> solve('sin(x) = 2 - x')

ans =
1.1060601577062719106167372970301
```

Sometimes there is more than one solution, and you might not get what you expected. For example:

```
>> solve('exp(-x) = sin(x)')

ans =
-2.0127756629315111633360706990971+2.7030745115909622139316148044265*i
```

The answer is a complex number. Though it is a valid solution of the equation, there are also real-number solutions. In fact, the graphs of $\exp(-x)$ and $\sin(x)$ are shown in Figure 2.3; each intersection of the two curves represents a solution of the equation $e^{-x} = \sin(x)$.

You can numerically find the (approximate) solutions shown on the graph with the **`fzero`** command, which looks for a zero of a given function near a specified value of x. A solution of the equation $e^{-x} = \sin(x)$ is a zero of the function $e^{-x} - \sin(x)$, so to find an approximate solution near $x = 0.5$, type

```
>> h = @(x) exp(-x) - sin(x);
>> fzero(h, 0.5)

ans =
   0.5885
```

Replace **`0.5`** with **`3`** to find the next solution, and so forth.

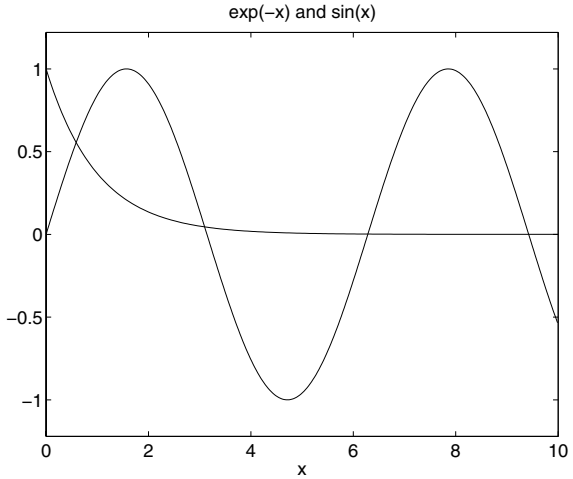


Figure 2.3. Two Intersecting Curves.

Graphics

In this section, we introduce MATLAB's two basic plotting commands and show how to use them.

Graphing with `ezplot`

The simplest way to graph a function of one variable is with **`ezplot`**, which expects a string, a symbolic expression, or an anonymous function, representing the function to be plotted. For example, to graph $x^2 + x + 1$ on the interval -2 to 2 (using the string form of **`ezplot`**), type

```
>> ezplot('x^2 + x + 1', [-2 2])
```

The plot will appear on the screen in a new window labeled "Figure 1".
Using a symbolic expression, you can produce the plot in Figure 2.4 with the following input:

```
>> syms x, ezplot(x^2 + x + 1, [-2 2])
```

Finally, you can use an anonymous function as the argument to **`ezplot`**, as in:

```
>> ezplot(@(x) x.^2 + x + 1, [-2 2])
```

✓ Graphs can be misleading if you do not pay attention to the axes. For example, the input **`ezplot(x^2 + x + 3, [-2 2])`** produces a graph that looks identical to the previous one, except that the vertical axis has different tick marks (and MATLAB assigns the graph a different title).

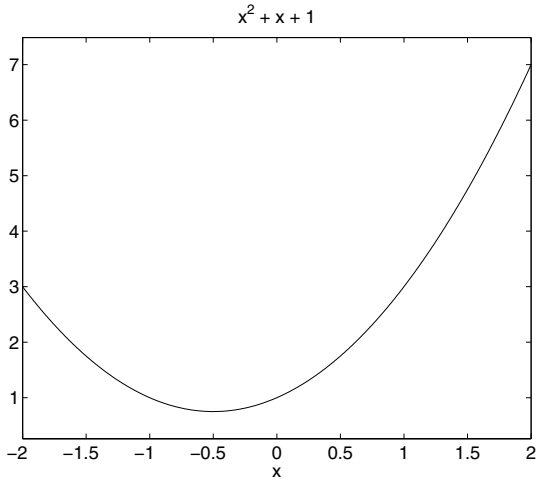


Figure 2.4. The Parabola $y = x^2 + x + 1$ on the Interval $[-2, 2]$.

Modifying Graphs

You can modify a graph in a number of ways. You can change the title above the graph in Figure 2.4 by typing (in the Command Window, not the figure window)

```
>> title 'A Parabola'
```

The same change can be made directly in the figure window by selecting **Axes Properties...** from the **Edit** menu at the top of the figure window. (Just type the new title in the box marked "Title.") You can add a label on the vertical axis with **`ylabel`** or change the label on the horizontal axis with **`xlabel`**. Also, you can change the horizontal and vertical ranges of the graph with the **`axis`** command. For example, to confine the vertical range to the interval from 0 to 3, type

```
>> axis([-1 2 0 3])
```

The first two numbers are the range of the horizontal axis; both ranges must be included, even if only one is changed.

To make the shape of the graph square, type **`axis square`**; this also makes the scale the same on both axes if the x and y ranges have equal length. For ranges of any lengths, you can force the same scale on both axes without changing the shape by typing **`axis equal`**. Generally, this command will expand one of the ranges as needed. However, if you unintentionally cut off part of a graph, the missing part is not forgotten by MATLAB. You can readjust the ranges with an **`axis`** command like the one above, or type **`axis tight`** to automatically set the ranges to include the entire graph. Type **`help axis`** for more possibilities. (Remember to type **`more on`** first if you want to read a screenful at a time.)

You can make some of these changes directly in the figure window as you will see if you explore the pull-down menus on the figure window's tool bar. Our experience

is that doing so via MATLAB commands in the Command Window provides more robustness, especially if you want to save your commands in an M-file (see Chapter 3) in order to reproduce the same graph later on. To close the figure, type **`close`** or **`close all`**, or simply click on the × in the upper right-hand corner of the window.

☞ *See Chapter 5 for more ways to manipulate graphs.*

Graphing with `plot`

The **`plot`** command works on vectors of numerical data. The syntax is **`plot(X, Y)`**, where **`X`** and **`Y`** are vectors of the same length. For example:

```
>> X = [1 2 3]; Y = [4 6 5]; plot(X, Y)
```

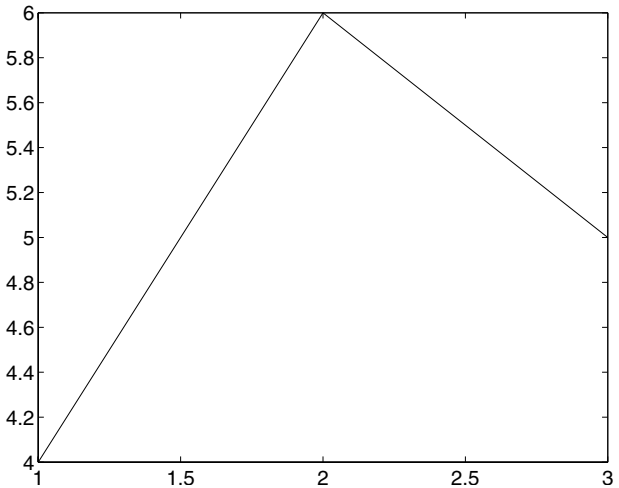


Figure 2.5. Plotting Line Segments.

✓ Here we separated multiple commands on one line with semicolons instead of commas. Notice that the output of the commands preceding the semicolons is suppressed.

The **`plot`** command considers the vectors **`X`** and **`Y`** to be lists of the x and y coordinates of successive points on a graph, and joins the points with line segments. So, in Figure 2.5, MATLAB connects $(1, 4)$ to $(2, 6)$ to $(3, 5)$.

To plot x^2 on the interval from -1 to 2 we first make a list **`X`** of x values, and then type **`plot(X, X.^2)`**. (The **`.`** here is essential since **`X.^2`** represents the element-by-element square of the vector **`X`**, not the matrix square.) We need to use enough x values to ensure that the resulting graph drawn by "connecting the dots" looks smooth. We'll use an increment of 0.01. Thus a recipe for graphing the parabola is:

```
>> X = -1:0.01:2; plot(X, X.^2)
```

The result appears in Figure 2.6. Note that we used a semicolon to suppress printing of the 301-element vector **X**.

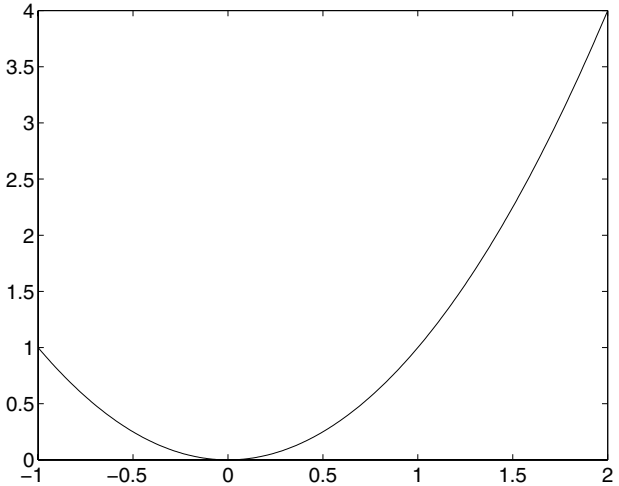


Figure 2.6. Plot of a Parabola.

☞ *We describe more of MATLAB's graphics commands in Chapter 5.*

For now, we content ourselves with demonstrating how to plot a pair of expressions on the same graph.

Plotting Multiple Curves

Each time you execute a plotting command, MATLAB erases the old plot and draws a new one. If you want to overlay two or more plots, type **hold on**. This command instructs MATLAB to retain the old graphics and draw any new graphics on top of the old. It remains in effect until you type **hold off**. Here's an example using **ezplot**:

```
>> ezplot('exp(-x)', [0 10])
>> hold on
>> ezplot('sin(x)', [0 10])
>> hold off
>> title 'exp(-x) and sin(x)'
```

The result is shown in Figure 2.3 earlier in this chapter. The commands **hold on** and **hold off** work with all graphics commands.

With **plot**, you can plot multiple curves directly. For example:

```
>> X = 0:0.01:10; plot(X, exp(-X), X, sin(X))
```

Note that the vector of x-coordinates must be specified once for each function being plotted.

Chapter 3

Interacting with MATLAB

In this chapter we describe an effective procedure for working with MATLAB, and for preparing and presenting the results of a MATLAB session. In particular we discuss some features of the MATLAB interface and the use of *M-files*. We introduce a new command in MATLAB 7, **`publish`**, which produces formatted output. We also give some simple hints for debugging your M-files.

The MATLAB Interface

Starting with version 6, MATLAB has an interface called the MATLAB *Desktop*. Embedded inside it is the Command Window that we described in Chapter 2.

The Desktop

By default, the MATLAB Desktop (Figure 1.1 in Chapter 1) contains four windows inside it, the Command Window on the right, the Current Directory Browser and the Workspace Browser in the upper left, and the Command History Window in the lower left. Notice that there are tabs for alternating between the Current Directory and Workspace Browsers. You can change which windows are currently visible with the **Desktop** menu (in MATLAB 6, the **View** menu) at the top of the Desktop, and you can adjust the sizes of the windows by dragging their edges with the mouse. The Command Window is where you type the commands and instructions that cause MATLAB to evaluate, compute, draw, and perform all the other wonderful magic that we describe in this book. We will discuss the other windows in separate sections below.

The MATLAB Desktop includes a *menu bar* and a *tool bar*; the tool bar contains icons that give quick access to some of the items you can select through the menu bar. Many menu items also have keyboard shortcuts, listed to their right when you select the menu. Some of these shortcuts depend on your operating system, and we generally will not mention them. Nonetheless, you may find it useful to note and use the keyboard shortcuts for menu items you frequently use.

Each of the windows in the Desktop contains two small buttons in the upper right-hand corner. The one labeled $\times$ allows you to close the window, while the curved arrow will "undock" the window from the Desktop (you can return it to the Desktop by selecting **Dock** from the **Desktop** menu of the undocked window, or by clicking on the curved arrow on its menu bar).

✓ While the Desktop provides some new features and a common interface for the *Windows* and *UNIX* versions of MATLAB, it may also run more slowly than the basic Command Window interface, especially on older computers. You can run MATLAB with the old interface by starting the program with the command **`matlab -nodesktop`**.

The Workspace

In Chapter 2, we introduced the commands **`clear`** and **`whos`**, which can be used to keep track of the variables you have defined in your MATLAB session. The complete collection of defined variables is referred to as the Workspace, which you can view using the Workspace Browser. You can make this browser appear by typing **`workspace`** or, in the default layout of the MATLAB Desktop, by clicking on the "Workspace" tab below the Current Directory Browser. The Workspace Browser contains a list of the current variables and their sizes (but not their values). If you double-click on a variable, its contents will appear in a new window called the *Array Editor*, which you can use to edit individual entries in a vector or matrix. (The command **`openvar`** also will open the Array Editor.) You can remove a variable from the Workspace by selecting it in the Workspace Browser and choosing **Edit**:**Delete**.

If you need to interrupt a session and don't want to be forced to recompute everything later, then you can save the current Workspace with **`save`**. For example, typing **`save myfile`** saves the values of all currently defined variables in a file called `myfile.mat`. To save only the values of the variables **`X`** and **`Y`**, type

```
>> save myfile X Y
```

When you start a new session and want to recover the values of those variables, use **`load`**. For example, typing **`load myfile`** restores the values of all the variables stored in the file `myfile.mat`.

✓ By default, variables are stored in a binary format that is specific to MATLAB, but you can also **`save`** or **`load`** data as ASCII text. For details, see the online help for these commands. This feature is useful for exchanging data with other programs.

The Current Directory and Search Path

New files that you create from within MATLAB will be stored in your *current directory*. The name of this directory is displayed in the MATLAB Desktop tool bar, and the files and subdirectories it contains are listed in the Current Directory Browser. You can also display the name of the current directory by typing **`pwd`** ("print working directory") in the Command Window, and can get a list of the directory's contents by typing **`dir`** or **`ls`**.

✓ The term *folder* is now more common than *directory*; for a computer file system, they mean the same thing. We will use "directory" because MATLAB uses this term in its documentation. However, its interface sometimes uses "folder", for example in the "File Type" column in the Current Directory Browser.

You may want to change the current directory from its default location, or you may want to maintain different directories for different projects. You can change the current directory in MATLAB by using the command **`cd`**, the Current Directory Browser, or the "Current Directory" box on the Desktop tool bar. You can type the directory name into this box and type ENTER, select a directory you have used before by clicking on the arrow at the right of the box, or browse for a directory by clicking on the "Browse for folder" icon to the right of the box.

For example, on a *Windows* computer, the default current directory is a subdirectory called `work` of the directory in which MATLAB is installed; for example, `C:\MATLAB7\work`. You can create a new directory, say `ProjectA`, within it by typing **`mkdir ProjectA`**. You can also right-click in the Current Directory Browser and select **New**:**Folder**, or click on the "New folder" icon in the browser's tool bar. Then type **`cd ProjectA`** or double-click on it in the Current Directory Browser to make it your current directory. You will then be able to read and write files in this directory in your current MATLAB session.

If you need only to be able to read files from a certain directory, an alternative to making it your current directory is to add it to the *path* of directories that MATLAB searches to find files. The current directory and the directories in your path are the only places MATLAB searches for files, unless you explicitly type the directory name as part of the file name. To see the current path, type **`path`**. To add the directory `C:\MATLAB7\work\ProjectA` to your path, type

```
>> addpath C:\MATLAB7\work\ProjectA
```

When you add a directory to the path, the files it contains remain available for the rest of your session regardless of whether you subsequently add another directory to the path or change the current directory. The potential disadvantage of this approach is that you must be careful when naming files. When MATLAB searches for files, it uses the first file with the correct name that it finds in the path list, starting with the current directory. If you use the same name for different files in different directories in your path, you can run into problems.

You can also control the MATLAB search path from the Set Path tool. To open this tool, type **`editpath`** or **`pathtool`**, or select **File**:**Set Path...**. The "Set Path" tool consists of a panel, with a list of directories in the current path, and several buttons that allow you to add, remove, and re-order the directories in your path.

✓ If you have many toolboxes installed, path searches can be slow, especially with **`lookfor`**. Removing the toolboxes you are not currently using from the MATLAB path is one way to speed up execution.

☞ *Changes you make to the current directory and path are not saved from one MATLAB session to the next. At the end of the* Script M-files *section below, we describe how to change these and other items automatically each time you start MATLAB.*

The Command History Window

The Command History Window contains a running history of the commands that you type into the Command Window. It is useful in two ways. First, it lets you see at a quick glance a record of the commands that you have entered previously. Second, it can save you some typing time. If you double-click on an entry in the Command History Window, then it will be executed immediately in the Command Window. However, often you will want to edit a previous command before executing it. If you right-click (that is, click with the right mouse button) on an entry in the Command History Window, it becomes highlighted and a menu of options appears. You can select **Copy**, then right-click in the Command Window and select **Paste**, whereupon the command you selected will appear at the command prompt and be ready for editing. As we described in *Recovering from Problems* in the previous chapter, you can also type the UP- and DOWN-ARROW keys in the Command Window to scroll through the commands that you have used recently. Then when you locate the correct command line, you can use the LEFT- and RIGHT-ARROW keys to move around in the command line, deleting and inserting changes as necessary, and then press ENTER to tell MATLAB to evaluate the modified command.

For example, you might want to calculate to 15 digits the values of $\sin(0.1)/0.1$, $\sin(0.01)/0.01$, and $\sin(0.001)/0.001$. Here is a first try at a solution, together with the response that MATLAB displays in the Command Window:

```
>> x = [0.1, 0.01, 0.001];
>> y = sin(x)/x

y =
   0.9984
```

This is not the intended result; only one number is displayed instead of three. Remember that to divide two vectors element-by-element, you must type **./** rather than **/**. (Typing only **/** "solves" the equation **y*x = sin(x)** for **y** in the "least-square" sense; type **help slash** for more information.) Another problem is that only 5 digits are displayed, not 15. To correct these problems, first type **format long**. Then type UP-ARROW twice to redisplay the command defining **y**, and type RIGHT-ARROW twice to move the cursor between the **)** and **/**. Finally, type **.** and ENTER:

```
>> y = sin(x)./x

y =
   0.99833416646828    0.99998333341667    0.99999983333334
```

M-Files

M-files allow you to save multiple MATLAB commands in a file and then run them with a single command or mouse click. While you may have solved the simple problem above correctly on the first try, more complicated problems generally require some trial and error – running, editing, and re-running a series of commands several times. While the Command History Window can be useful during the first stages of this process, eventually you will find it more efficient to use M-files. M-files also allow you to share your solution to a problem with other MATLAB users, and to format your results for others to read. There are two different kinds of M-files: script M-files and function M-files. We shall illustrate the use of both types of M-files as we present different solutions to the problem described above.

M-files are ordinary text files containing MATLAB commands. You can create and modify them using any text editor or word processor that is capable of saving files as plain ASCII text. (Such text editors include *Notepad* and *WordPad* in *Windows*, and **emacs** and **vi** in *UNIX*.) More conveniently, you can use the built-in *Editor/Debugger*, which you can start by typing **edit**, either by itself (to edit a new file) or followed by the name of an existing M-file in the current directory. You can also use the **File** menu or the two leftmost icons on the tool bar to start the Editor/Debugger, either to create a new M-file or to open an existing M-file. Double-clicking on an M-file in the Current Directory Browser will also open it in the Editor/Debugger.

Script M-Files

A script M-file contains a sequence of MATLAB commands to be run in order. We now show how to construct a script M-file to solve the mathematical problem described earlier. Create a file containing the following lines:

```
format long
x = [0.1, 0.01, 0.001];
y = sin(x)./x
```

We will assume that you have saved this file with the name `task1.m` in your current directory, or in some directory in your path. You can name the file any way you like (subject to the usual naming restrictions on your operating system), but the "`.m`" suffix is mandatory.

You can tell MATLAB to run (or *execute*) this script by typing **task1** in the Command Window. (You must *not* type the "`.m`" extension here; MATLAB automatically adds it when searching for M-files.) The output – but not the commands that produce them – will be displayed in the Command Window. Now the sequence of commands can easily be changed by modifying the M-file `task1.m`. For example, if you also wish to calculate $\sin(0.0001)/0.0001$, you can modify the M-file to read

```
format long
x = [0.1, 0.01, 0.001, 0.0001];
y = sin(x)./x
```

and then run the modified script by again typing **task1**. Be sure to save your changes

to `task1.m` first; otherwise, MATLAB will not recognize them.

✓ Any variables that are set by running a script M-file will persist exactly as if you had typed them into the Command Window directly. For example, the program above will cause all future numerical output to be displayed with 15 digits. In order to revert to 5-digit format, you would have to type **`format short`**.

Adding Comments It is worthwhile to include comments in M-files. These comments might explain what is being done in the calculation, or might interpret the results of the calculation. In MATLAB, the percent sign (`%`) begins a comment; the rest of the line is not executed by MATLAB. (The Editor/Debugger colors comments green to help distinguish them from commands, which appear in black.) Here is our new version of `task1.m` with a few comments added:

```
format long   % turn on 15 digit display
x = [0.1, 0.01, 0.001];
y = sin(x)./x
% These values illustrate the fact that the limit of
% sin(x)/x as x approaches 0 is 1.
```

Notice that a multi-line comment requires a percent sign at the beginning of each line.

Cell Mode A new feature in MATLAB 7 allows one to divide a script M-file into subunits called *cells*. This is especially useful if your M-file is long or if you are going to *publish* it, as explained below in *Publishing an M-file*. To start a new cell, insert a comment line (which will serve as the "title" of the cell that follows) starting with a double percent sign **`%%`** followed by a space. If you open the M-file in the Editor/Debugger and click on **Enable Cell Mode** in the **Cell** menu, then a second tool bar will appear below the first one. Then when you click somewhere in the M-file, the cell that contains that location will be highlighted in pale yellow. You can evaluate that cell by then selecting **Cell**:**Evaluate Current Cell** or pressing the "Evaluate cell" icon. This can be a big help if you've made a change in just one cell and do not want to run the whole script all over again. There are also a menu item and icon to "Evaluate cell and advance". Once you have enabled cell mode, you can also create more cells by selecting **Cell**:**Insert Cell Divider** or by clicking on the corresponding icon.

Initializing Script M-files In order for the results of a script M-file to be reproducible, the script should be self-contained, unaffected by other variables that you might have defined elsewhere in the MATLAB session, and uncorrupted by leftover graphics. For example, if you define a variable named **`sin`** in the Command Window and then run the script `task1.m` above, you will get an error message because **`sin`** now represents a variable rather than the usual built-in function. With this in mind, you can type the line **`clear all`** at the beginning of the script to ensure that previous definitions of variables do not affect the results. You can also type **`close all`**

at the beginning of a script M-file that creates graphics, to close all figure windows and start with a clean slate.

As mentioned above, the commands in a script M-file will not automatically be displayed in the Command Window. If you want the commands to be displayed along with the results, add the command **`echo on`** to the beginning of script (it is then a good idea to add **`echo off`** to the end of the script). Any comments in the M-file will then be echoed too. When running a long script M-file, echoing is useful to keep track of which outputs go with which inputs.

Here is a more carefully commented version of `task1.m` that will display input as well as output:

```
clear all   % remove old variable definitions
echo on   % display the input in the command window
format long   % turn on 15 digit display

x = [0.1, 0.01, 0.001];   % define the x values
y = sin(x)./x   % compute the desired quotients

% These values illustrate the fact that the limit of
% sin(x)/x as x approaches 0 is equal to 1.

echo off
```

Startup M-File When MATLAB starts, it searches the default path (which includes the default current directory) for a script M-file called `startup.m`. If you create such a file, the commands it contains will be run each time MATLAB starts. You can use this file to save customizations that do not normally persist from one session to the next, such as changes to the current directory or path. (In addition to the commands **`cd`** and **`addpath`** described above, you can remove directories from the path with **`rmpath`**.)

Function M-Files

Function M-files, unlike script M-files, allow you to specify input values when you run them from the MATLAB command line or from another M-file. As we described in the previous chapter, you can also use the anonymous-function (**`@`**) syntax – not available in MATLAB 6 or earlier versions – or **`inline`** to define your own functions on the command line. However, these methods allow only a one-line function definition, so M-files are necessary to define more complicated functions. Like a script M-file, a function M-file is a plain text file that should reside in your current directory or elsewhere in your MATLAB path.

Let us return to the problem described above, where we computed some values of $\sin(x)/x$ with $x = 10^{-b}$ for several values of b. Suppose, in addition, that you want to find the smallest value of b for which $\sin(10^{-b})/(10^{-b})$ and 1 agree to 15 digits. Here is a function M-file called `sinelimit.m` designed to explore this question:

```
function y = sinelimit(c)
% SINELIMIT computes sin(x)/x for x = 10^(-b),
% where b = 1, ..., c.
format long
b = 1:c;
x = 10.^(-b);
y = (sin(x)./x)';
```

The first line of the file starts with **function**, which identifies the file as a function M-file. (The Editor/Debugger colors this special word blue.) The first line of the M-file specifies the name of the function and describes both its input arguments (or parameters) and its output values. In this example, the function is called **sinelimit**. The file name (without the `.m` extension) and the function name should match. When you create this new function M-file in an untitled editor window and select **Save**, the Editor/Debugger knows to call the file `sinelimit.m`. The function in our example takes one input, which is called **c** inside the M-file. It also returns one output, which is the value of **y** that results when the function finishes executing.

It is good practice to follow the first line of a function M-file with one or more comment lines explaining what the M-file does. If you do, **help** will automatically retrieve this information. For example:

```
>> help sinelimit

  SINELIMIT computes sin(x)/x for x = 10^(-b),
  where b = 1, ..., c.
```

The remaining lines of the M-file define the function. In this example, **b** is defined to be a row vector consisting of the integers from **1** to **c**, then **x** is computed from **b**, and finally **y** is determined from **x**.

☞ *The variables used in a function M-file, like* **b**, **x**, *and* **y** *in* `sinelimit.m`, *are* local variables. *This means that, unlike the variables that are defined in a script M-file, these variables are completely unrelated to any variables with the same names that you may have used in the Command Window, and MATLAB does not remember their values after the function M-file has been executed. For further information, see* Variables in Function M-Files *in Chapter 4.*

Notice that the output of the lines defining **b**, **x**, and **y** is suppressed with a semicolon. Since these variables are only used internally by the M-file, it would be misleading to see output about them in the Command Window. While displaying the output of intermediate calculations can be useful for debugging, in general you should suppress all output in a function M-file.

Here is an example that shows how to use the function **sinelimit**:

```
>> sinelimit(5)

ans =
   0.99833416646828
   0.99998333341667
   0.99999983333334
```

```
0.99999999833333
0.99999999998333
```

None of the values of b from 1 to 5 yields the desired answer, 1, to 15 digits. Judging from the output, you can expect to find the answer to the question we posed above by typing **`sinelimit(10)`**. Try it!

Loops

A *loop* specifies that a command or group of commands should be repeated several times. The easiest way to create a loop is to use a **`for`** statement. Here is a simple example that computes and displays $10! = 10 \cdot 9 \cdot 8 \cdots 2 \cdot 1$.

```
f = 1;
for n = 2:10
   f = f*n;
end
f
```

The loop begins with the **`for`** statement and ends with the **`end`** statement. The command between those statements is executed a total of nine times, once for each value of **`n`** from 2 to 10. We used a semicolon to suppress intermediate output within the loop. In order to see the final output, we then needed to type **`f`** after the end of the loop. Without the semicolon, MATLAB would display each of the intermediate values 2!, 3!, ...

The Editor/Debugger automatically colors the commands **`for`** and **`end`** blue. It is not necessary, but improves readability, if you indent the commands in between (as we did above); the Editor/Debugger does this automatically. If you type **`for`** in the Command Window, MATLAB does not give you a new prompt (**`>>`**) until you enter an **`end`** command, at which time MATLAB will evaluate the entire loop and display a new prompt.

✓ If you use a loop in a script M-file with **`echo on`** in effect, the commands will be echoed every time through the loop. You can avoid this by inserting the command **`echo off`** just before the **`end`** statement and inserting **`echo on`** just afterward; then each command in the loop (except **`end`**) will be echoed once.

Presenting Your Results

Sometimes you may want to show other people the results of a script M-file that you have created. For a polished presentation, you can import your results into another program, such as a word processor, use **`publish`** to convert your results to a format such as HTML, or (on a *Windows* computer) use an M-book. To share your results more informally, you can give someone else your M-file, assuming that person has a copy of MATLAB on which to run it, or you can provide the output you obtained in the form of a *diary* file. We now discuss these different approaches.

✓ You can greatly enhance the readability of your M-file by including frequent comments. Your comments should explain what is being calculated, so the reader can understand your procedures and strategies. Once you've done the calculations, you can also add comments that interpret the results.

Publishing an M-File

MATLAB 7 comes with a very convenient command called **`publish`** to convert a script M-file to a readable document. You should use this feature in combination with cells; see *Cell Mode* above. The default output format, which we have found works best, is HTML (i.e., a web page), but you can also publish to a *Word* document or *PowerPoint* presentation (on a *Windows* computer) or to a *LaTeX* document. Once you have enabled cell mode in the Editor/Debugger, you can also use the "Publish to HTML" icon on the Cell tool bar (which appears under the regular Editor tool bar).

Publishing an M-file reproduces the M-file along with its text and graphics output. Each cell of the M-file appears as a separate section, under the heading taken from the appropriate line starting with a double percent sign **`%%`** followed by a space. Any comment lines that immediately follow this line appear as formatted text in the output, provided that they start with a percent sign and a space, and that there are no intervening blank lines. (You can select several formats, such as bulleted lists, using the **Insert Text Markup** submenu of the **Cell** menu in the Editor/Debugger.) Next, all of the MATLAB input lines and any remaining comments in the cell are reproduced without any formatting, followed by any text output (in gray), followed by graphics. (It is not necessary to use **`echo`** when publishing an M-file; the input and comments always appear in the published document.)

✓ A line that starts with a double percent sign but has no following text starts a new cell but does not produce a section heading in the published output. Starting a new cell in this way allows you to insert formatted text that discusses the output from the previous cell, without starting a new section. Perhaps more importantly, it allows you to keep input and output close to each other in a section with many input lines. If you don't break up such a section into smaller cells, all of its input will appear before all of its output, making it hard to match output to input.

Here is another version of the script M-file `task1.m` discussed above that is suitable for publishing.

```
%% Sample Script M-file
% This script computes sin(x)/x for x = 0.1, 0.01, 0.001.

clear all   % remove old variable definitions
format long   % turn on 15 digit display

x = [0.1, 0.01, 0.001];   % define the x values
y = sin(x)./x   % compute the desired quotients
```

```
%%
% These values illustrate the fact that the limit of
% sin(x)/x as x approaches 0 is equal to 1.
```

If your M-file puts two successive figures in the same figure window, then only the last figure will be written to the published document, unless the figures appear in different cells. So you should either start a new cell each time you create a new figure, or else use **figure** to open separate windows for each figure produced by a given cell.

☞ *See Chapter 7 for a more thorough discussion of publishing. See Chapter 5 for more about figure windows.*

Diary Files

While **publish** is available only starting with MATLAB 7, all versions of MATLAB have the more primitive command **diary**, which saves all of the text output (and, with **echo on**, the input) from a session to a file. Here is a way to use **diary** to automatically save the text input and output from a script M-file to a file each time you run it. At the beginning of the M-file, say `task1.m`, you can include the commands

```
delete task1.txt
diary task1.txt
echo on
```

The script M-file should then end with the commands

```
echo off
diary off
```

The first **diary** command causes all subsequent input to and output from the Command Window to be copied into the specified file – in this case, `task1.txt`. The diary file `task1.txt` is a plain text file that is suitable for printing or importing into another program.

By using **delete** at the beginning of the M-file, you ensure that the file only contains the output from the most recent run of the script. If you omit the **delete** command, then **diary** will add any new output to the end of an existing file, and the file `task1.txt` can end up containing the results of several runs of the M-file. (Putting the **delete** command in the script will lead to a harmless warning message about a nonexistent file the first time you run the script.) You can also get extraneous output in a diary file if you type CTRL+C to halt a script containing a **diary** command. If you do so, you should then type **diary off** in the Command Window before proceeding.

Interactive M-Files

If you want someone else to run your script M-file and see the results, it is helpful to use the command **pause** to stop execution at various points: after each graph is produced, after important comments, and after critical places where your script generates numerical output. Each time MATLAB reaches a **pause** command, it

waits until the user presses a key before proceeding. Of course, if the recipient of your M-file is not familiar with **pause**, then you should include appropriate guidance at such points; for example, with **echo on** in effect you could type

```
pause   % Press any key to continue...
```

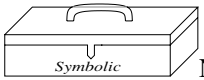
If you are not using **echo on**, then you can use the command **disp** to display an appropriate message before pausing. Another command that may be useful is **input**, which both displays a message and waits for the user to enter a number or other MATLAB expression; see the online help.

✓ These interactive commands can be an annoyance if, for example, you want to publish the M-file. If your recipient is familiar with cell mode in the Editor/Debugger, then an alternative to using **pause** is to separate the M-file into cells, and let your recipient run the file one cell at a time.

☞ *MATLAB also has commands and tools for developing a graphical user interface (GUI); see Chapter 9. This approach allows your recipient to interact with your M-file in a separate window that you design, rather than in the Command Window.*

Wrapping Long Input and Output Lines

Sometimes you may need to type, either in the Command Window or in an M-file, a command that is too long to fit on one line. If so, when you get near the end of a line you can type **...** (that is, three successive periods) followed by ENTER, and continue the command on the next line. If you do this in the Command Window, you will not see a command prompt on the new line.



Numerical output is formatted by MATLAB to fit your screen, but symbolic expressions generally appear on one line no matter how long they are. If **s** is a symbolic expression, then typing **pretty(s)** displays **s** in *pretty print* format, which uses multiple lines on your screen to imitate written mathematics. The result is often easier to read than the default format. An important feature of **pretty** is that it *wraps* long expressions to fit within the margins (80 characters wide) of a standard-sized window. If your symbolic output is long enough to extend past the right-hand edge of your window, it may be truncated when you print your output, so you should use **pretty** to make the entire expression visible in your printed output.

Printing and Saving Graphics

As indicated in Chapters 1 and 2, graphics appear in a separate window. You can save the current graph by selecting **File**:**Save As...** in the figure window. The default file format (`.fig`) is specific to MATLAB and is not understood by most other programs; however, several other formats are available. You can also print the current figure by selecting **File**:**Print...**. Alternatively, typing **print** causes the figure in the current figure window to be printed on your default printer. Since you probably don't want

to print the figure every time you run a script, you should not include a bare **print** command in an M-file. Instead, you should use a form of **print** that sends the output to a file. It is also helpful to give informative titles to your figures. For example:

```
close all   % remove old figure windows

% Graph sin(x) from 0 to 2*pi:
x = 2*pi*(0:0.01:1);
plot(x, sin(x))

title('Figure A: Sine Curve')   % title the figure
print -deps figureA   % store the graph in figureA.eps
```

The form of **print** used in this script causes the current figure to be written in *Encapsulated PostScript*® format to a file in the current directory, called `figureA.eps`. This file can be printed later (on a PostScript printer), or it can be imported into another program that can read PostScript format. If you intend to paste MATLAB graphics into a web page, it is better to save the figure as a `png` file with **print -dpng**. (You can also select **Save As...** from the **File** menu of the figure window and under "Save as type:" select "Portable Network Graphics file (*.png)".) See the online help for **print** for other formats available.

As a final example involving graphics, let's consider the problem of graphing the functions $\sin(x)$, $\sin(2x)$, and $\sin(3x)$ on the same set of axes. This is a typical example; we often want to plot several similar curves whose equations depend on a parameter. Here is a script M-file solution to the problem:

```
x = 2*pi*(0:0.01:1);   % define the x values
figure, hold on   % open a new figure window with overlay

% Run a loop to plot three sine curves:
for c = 1:3
   plot(x, sin(c*x))
end

hold off, axis([0, 2*pi, -1, 1])   % adjust the x axis
title('Several Sine Curves')   % title the figure
```

The result is shown in Figure 3.1.

Let's analyze this solution. We start by defining a grid of 101 evenly spaced x values from 0 to 2π. Then **figure** opens a new figure window, and **hold on** lets MATLAB know that we want to draw several curves on the same set of axes. Rather than type three **plot** commands, we use a **for** loop, as described above. Then we use **hold off** to release the figure window, make the graph look a little nicer by changing the horizontal axis from the range MATLAB chooses to the actual range of x, and **title** the figure.

M-Books

Another sophisticated way of presenting MATLAB output, without having to produce an M-file first, is in a Microsoft *Word* document, incorporating text, MATLAB com-

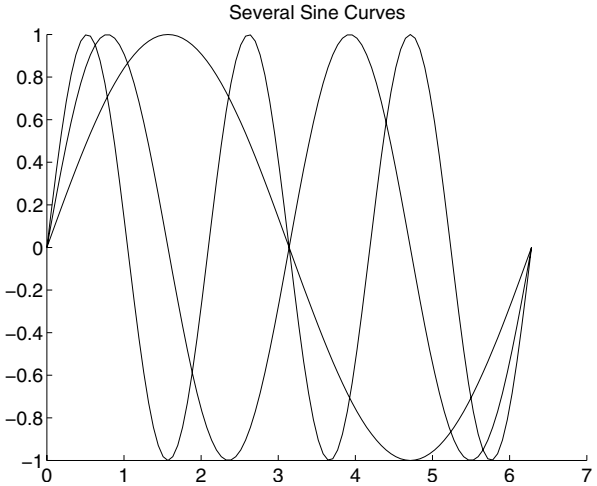


Figure 3.1. Sine Curves.

mands, and graphics. This allows you more control over formatting than **`publish`**. A simple first approach is to prepare a *Word* document with explanatory comments, and to paste in your MATLAB commands (you can do this in another font) or include your M-files using **File...** from *Word*'s **Insert** menu. Finally, you can paste in the graphics by selecting **Insert**:**Picture**:**From File...** in *Word*. You should have first saved the graphics in a common format such as `png`, `tiff`, or `eps`.

A more direct approach is to enable M-books on your computer. An M-book is a *Word* document with embedded executable MATLAB code (that runs as a *Word* macro, via the intermediary of *Visual Basic*). You can launch an M-book by typing **`notebook`** in the Command Window, or by starting *Word*, choosing **New...** from its **File** menu, and selecting `m-book` as the Document template. If the file `m-book.dot` does not already exist on your computer, you need to *enable M-books* first. This is done by typing

```
>> notebook -setup
```

in the MATLAB Command Window and following the instructions. You will be prompted for the version of *Word* that you are using, and possibly for the location of some associated files.

✓ In Windows XP, 2000, or NT, you will get an error message if you run **`notebook -setup`** from an account without administrator privileges. However, the setup procedure should still copy `m-book.dot` into your Word templates directory. You then should be able to open an M-book from Word, though running **`notebook`** in MATLAB will still fail. In order to fully enable M-books, you must get an administrator to run **`notebook -setup`**, and possibly to override the default template location in order to designate a directory that you can access.

⇨ **For M-books to run properly, you must enable execution of the necessary macros in *Word*. The safest way to do this is to see whether you get a security warning dialog box when you open an M-book and, if so, select "Always trust macros from this source". This will maintain your security level at "high" for other macros. Alternatively, you can set your security level to "medium" for all macros in the dialog box you get by selecting Tools:Macros:Security... in *Word*.**

Once you have successfully launched an M-book, it will behave just like any other *Word* document, except for the **Notebook** menu at the top. If you type a MATLAB command and hit CTRL+ENTER, or else highlight the command with the mouse and select **Evaluate Cell** in the **Notebook** menu, MATLAB will evaluate the command and send the output back to *Word*. For ease in reading, *Word* typesets "input cells" (MATLAB input) in green Courier bold and "output cells" (MATLAB output) in blue Courier type. You have an option (which you can adjust with the **Notebook Options...** item in the **Notebook** menu) of having figure windows appear separately, having them appear in the M-book, or both.

In one respect, M-books behave like M-files; you can modify them and run them again and again. If you find you mistyped a command or want to change a command, you can simply go back to the appropriate input cell, change it, and then re-evaluate it. The new output will replace the old. Keep in mind, though, that the output will reflect the order in which you evaluate the cells, not the order in which they appear in the M-book. When preparing your final document, you should re-run the entire M-book by selecting **Notebook**:**Evaluate M-book** and make sure that the commands produce the output you desire when they are run in order.

Fine-Tuning Your M-Files

Script M-files allow you to refine your strategy for solving a problem without starting from scratch each time. Nonetheless, debugging a lengthly M-file can be time-consuming. Here are some general tips for diagnosing errors (and avoiding them in the first place).

☞ *We will discuss features of the Editor/Debugger and more advanced MATLAB debugging commands in the section* Debugging *in Chapter 6 and in the section* Debugging Techniques *in Chapter 11*

- Use **`publish`** with frequent cell dividers, or use **`echo on`** near the beginning of an M-file, so that you can see "cause" as well as "effect."

- Use cell mode in the Editor/Debugger, or make liberal use of **`pause`**, so that you can see the results of one part of an M-file at a time.

- Use the semicolon to suppress unneeded output, especially large vectors and arrays that you define.

- If you are producing graphics with **`hold on`**, don't forget to type **`hold off`** before starting a new figure, so that the next figure is not mixed with the old one.
- Also when starting a new figure, insert a cell divider or **`pause`** command, or use **`figure`** to open a new window, so that the next graphics command will not obliterate the current one before you have a chance to see it.
- Do not include bare **`print`** commands in your M-files. Instead, print to a file.
- The command **`keyboard`** is like **`pause`**, but is more useful for debugging. If you have the line **`keyboard`** in your M-file, then when MATLAB reaches it, execution of your script is interrupted, and a new prompt appears with the letter **`K`** before it. At this point you can type any normal MATLAB command. This is useful if you want to examine or reset some variables in the middle of running a script. To resume execution, type the command **`return`**.
- Finally, remember that you can stop a running M-file by typing CTRL+C. This is useful if a calculation is taking too long, if MATLAB is spewing out undesired output, or if, at a **`pause`** command, you realize that you want to stop execution completely.

Practice Set A

Algebra and Arithmetic

Symbolic Problems 3–8 require the Symbolic Math Toolbox. The others do not.

1. Compute the following quantities.

   (a) $1111 - 345$.

   (b) e^{14} and 382801π to 15 digits each. Which is bigger?

   (c) the fractions $2709/1024$, $10583/4000$, and $2024/765$. Which of these is the best approximation to $\sqrt{7}$?

2. Compute to 15 digits the following quantities.

   (a) $\cosh(0.1)$.

   (b) $\ln(2)$. (*Hint*: the natural logarithm in MATLAB is called **`log`**, not **`ln`**.)

   (c) $\arctan(1/2)$. (*Hint*: the inverse tangent function in MATLAB is called **`atan`**, not **`arctan`**.)

3. Solve (symbolically) the system of linear equations

$$\begin{aligned} 3x + 4y + 5z &= 2 \\ 2x - 3y + 7z &= -1 \\ x - 6y + z &= 3. \end{aligned}$$

   Check your answer using matrix multiplication.

4. Try to solve (symbolically) the system of linear equations

$$\begin{aligned} 3x - 9y + 8z &= 2 \\ 2x - 3y + 7z &= -1 \\ x - 6y + z &= 3. \end{aligned}$$

   What happens? Can you see why? Again check your answer using matrix multiplication. Is the answer "correct"?

5. Factor the polynomial $x^4 - y^4$.

6. Use **`simplify`** or **`simple`** to simplify the following expressions:

(a) $1/\left(1+1/\left(1+\frac{1}{x}\right)\right)$,

(b) $\cos^2 x - \sin^2 x$.

7. Compute 3^{301}, both as an approximate floating-point number and as an exact integer (written in usual decimal notation).

8. Use either **`solve`** or **`fzero`**, as appropriate, to solve the following equations.

   (a) $67x + 32 = 0$ (exact solution).

   (b) $67x + 32 = 0$ (numerical solution to 15 places).

   (c) $x^3 + px + q = 0$. (Solve for x in terms of p and q.)

   (d) $e^x = 8x - 4$ (*all* real solutions). It helps to draw a picture first.

9. Use **`plot`** and/or **`ezplot`**, as appropriate, to graph the following functions.

   (a) $y = x^3 - x$ for $-4 \le x \le 4$.

   (b) $y = \sin(1/x^2)$ for $-2 \le x \le 2$. Try this with both **`plot`** and **`ezplot`**. Are both results "correct"? If you use **`plot`**, be sure to plot enough points.

   (c) $y = \tan(x/2)$ for $-\pi \le x \le \pi, -10 \le y \le 10$. (*Hint*: first draw the plot; then use **`axis`**.)

   (d) $y = e^{-x^2/2}$ and $y = x^4 - x^2$ for $-1.5 \le x \le 1.5$ (on the same set of axes).

10. Plot the functions x^4 and 2^x on the same graph and determine how many times their graphs intersect. (*Hint*: you will probably have to make several plots, using intervals of various sizes, in order to find all the intersection points.) Now find the approximate values of the points of intersection using **`fzero`**.

Chapter 4

Beyond the Basics

In this chapter, we describe some of the finer points of MATLAB and review in more detail some of the concepts introduced in Chapter 2. We explore enough of MATLAB's internal structure to improve your ability to work with complicated functions, expressions, and commands. At the end of this chapter, we introduce some of the MATLAB commands for doing calculus.

Suppressing Output

Some MATLAB commands produce output that is superfluous. For example, when you assign a value to a variable, MATLAB echoes the value. But, as we mentioned briefly in Chapter 2, you can suppress the output of a command by putting a *semicolon* after the command. Here is an example.

```
>> syms x
>> y = x + 7
```

```
y=
x+7
```

```
>> z = x + 7;
>> z
```

```
z=
x+7
```

The semicolon does not affect the way MATLAB processes the command internally, as you can see from its response to the command **z**.

You can also use semicolons to separate a series of commands when you are interested only in the output of the final command (several examples appear later in the chapter). As we noted in Chapter 2, commas can also be used to separate commands – without suppressing output. If you use a semicolon after a graphics command, it will not suppress the graphic.

⇨ **The most common use of the semicolon is to suppress the printing of a long vector or big matrix.**

Another object that you may want to suppress is MATLAB's label for the output of a command. The command **disp** is designed to achieve that; typing **disp(x)** will print the value of the variable **x** without printing the label and the equal sign. So

```
>> x = 7; disp(x)
```

```
     7
```

or

```
>> disp(solve('x + tan(y) = 5', 'y'))
-atan(x-5)
```

Data Classes

Every variable you define in MATLAB, as well as every input to, and output from, a command, is an *array* of data belonging to a particular *class*. In this book we use primarily the following types of data: floating-point numbers, symbolic expressions, character strings, function handles, and inline functions. We introduced each of these types in Chapter 2. In Table 4.1, we list for each type of data its class (as given by **whos**) and how you can create it.

Table 4.1. MATLAB Data Classes.

Type of data	Class	Created by
Floating-point	`double`	Typing a number
Symbolic	`sym`	Using **sym** or **syms**
Character string	`char`	Typing a string inside single quotes
Function handle	`function_handle`	Using **@**
Inline function	`inline`	Using **inline**

You can think of an array as a two-dimensional grid of data. A single number (or symbolic expression) is regarded by MATLAB as a 1×1 array, sometimes called a *scalar*. A $1 \times n$ array is called a *row vector*, and an $m \times 1$ array is called a *column vector*. (A string is actually a row vector of characters.) An $m \times n$ array of numbers is called a *matrix*; see *More on Matrices* below. You can see the class and array size of every variable you have defined by looking in the Workspace Browser or typing **whos** (see *Managing Variables* in Chapter 2). The set of variable definitions shown by **whos** is called your *Workspace*.

In order to use MATLAB commands effectively, you must pay close attention to the class of data each command accepts as input and returns as output. The input to a command consists of one or more arguments separated by commas; some arguments are optional. Some commands, like **whos**, do not require any input. The help text (see *Online Help* in Chapter 2) for each command usually tells what classes of inputs the command expects as well as what class of output it returns.

✓ When you type a pair of words, like **hold on**, MATLAB interprets the second word as a string argument to the command given by the first word; thus, **hold on** is equivalent to **hold('on')**.

Many commands allow more than one class of input, though sometimes only one data class is mentioned in the online help. This flexibility can be a convenience in some cases and a pitfall in others. For example, the integration command, **int**, accepts strings as well as symbolic input, though its help text mentions only symbolic input. On the other hand, suppose that you have already defined **a = 10**, **b = 5**, and now you attempt to factor the expression $a^2 - b^2$, forgetting your previous definitions and that you have to declare the variables symbolic:

```
>> factor(a^2 - b^2)

ans =
   3     5     5
```

The reason you don't get an error message is that **factor** is the name of a command that factors integers into prime numbers as well as factoring expressions. Since $a^2 - b^2 = 75 = 3 \cdot 5^2$, the numerical version of **factor** is applied. The output is clearly not what you intended, but, in the course of a complicated series of commands, be careful not to be fooled by such unintended output.

✓ Note that typing **help factor** shows you only the help text for the numerical version of the command, but gives a cross-reference to the symbolic version at the bottom. If you want to see the help text for the symbolic version instead, type **help sym/factor**. Functions such as **factor** with more than one version are called *overloaded*.

Sometimes you need to convert one data class into another in order to prepare the output of one command to serve as the input for another. For example, we have used **double** to convert symbolic expressions to floating-point numbers and **sym** to convert numbers or strings to symbolic expressions. The commands **num2str** and **str2num** convert between numbers and strings, while **char** converts a symbolic expression to a string. You can also use **vectorize** to convert a symbolic expression to a vectorized string; it adds a **.** before every *****, **/**, and **^** in the expression.

String Manipulation

Often it is useful to concatenate two or more strings together. The simplest way to do this is to use MATLAB's vector notation, keeping in mind that a string is a "row vector" of characters. For example, typing **[string1, string2]** combines **string1** and **string2** into one string.

Here is a useful application of string concatenation. You may need to define a string variable containing an expression that takes more than one line to type. (In most circumstances you can continue your MATLAB input onto the next line by typing **...** followed by ENTER or RETURN, but this is not allowed in the middle of a string.) The solution is to break the expression into smaller parts and concatenate them, as in:

```
>> eqn = ['left hand side of equation = ', ...
'right hand side of equation']
```

```
eqn =
left hand side of equation = right hand side of equation
```

✓ Notice that string output, like symbolic output, is not indented.

Symbolic and Floating-Point Numbers

We mentioned above that you can convert between symbolic numbers and floating-point numbers with **double** and **sym**. Numbers that you type are, by default, floating-point. However, if you mix symbolic and floating-point numbers in an arithmetic expression, the floating-point numbers are automatically converted into symbolic numbers. This explains why you can type **syms x** and then **x^2** without having to convert **2** into a symbolic number. Here is another example:

```
>> a = 1
a =
   1
>> b = a/sym(2)
b =
1/2
```

MATLAB was designed so that some floating-point numbers are restored to their exact values when converted into symbolic numbers. Integers, rational numbers with small numerators and denominators, square roots of small integers, the number π, and certain combinations of these numbers are so restored. For example:

```
>> c = sqrt(3)
c =
   1.7321
>> sym(c)
ans =
sqrt(3)
```

Since it is difficult to predict when MATLAB will preserve exact values, it is best to suppress the floating-point evaluation of a numerical argument to **sym** by enclosing it in single quotes to make it a string, e.g., **sym('1 + sqrt(3)')**. We will see below another way in which single quotes suppress evaluation.

Functions and Expressions

We have used the terms *expression* and *function* without carefully making a distinction between the two. Strictly speaking, if we define $f(x) = x^3 - 1$, then f (written without any particular input) is a function while $f(x)$ and $x^3 - 1$ are expressions involving the variable x. In mathematical discourse we often blur this distinction by calling $f(x)$ or $x^3 - 1$ a function, but in MATLAB the difference between functions and expressions is important.

In MATLAB, an expression can belong to either the string or the symbolic class of data. Consider the following example.

```
>> f = 'x^3 - 1';
>> f(7)

ans =
1
```

This result may be puzzling if you are expecting **f** to act like a function. Since **f** is a string, **f(7)** denotes the seventh character in **f**, which is **1** (the spaces count). Recall that, like symbolic output, string output is not indented from the left-hand margin. This is a clue that the answer above is a string (consisting of one character), not a floating-point number. Typing **f(5)** would yield **-** and **f(-1)** would produce an error message.

You have learned three ways to define your own functions, using **@** to create an anonymous function, using **inline** (see Chapter 2 for these), or using an M-file (see Chapter 3). Anonymous or inline functions are most useful for defining simple functions that can be expressed in one line and for turning the output of a symbolic command into a function. Function M-files are useful for defining functions that require several intermediate commands to compute the output. Most MATLAB commands are actually M-files, and you can peruse them for ideas to use in your own M-files – to see the M-file for, say, the command **mean** you can enter **type mean**. See also *More about M-files* below.

Some commands, such as **ode45** (a numerical ordinary differential equation (ODE) solver, which we use later in Chapter 10), require their first argument to be a function – to be precise, either an inline function (as in **ode45(f, [0 2], 1)**) or a *function handle*, that is, the name of a built-in function or a function M-file preceded by the special symbol **@** (as in **ode45(@func, [0 2], 1)**). The **@** syntax was introduced in MATLAB 6; in earlier versions of MATLAB, the substitute was to enclose the name of the function in single quotes to make it a string. But with or without quotes, typing a symbolic expression as the first input to **ode45** gives an error message. On the other hand, most symbolic commands require their first argument to be either a string or a symbolic expression, not a function.

An important difference between strings and symbolic expressions is that MATLAB automatically substitutes user-defined functions and variables into symbolic expressions, but not into strings. (This is another sense in which the single quotes you type around a string suppress evaluation.) For example, if you type

```
>> h = @(t) t^3; int('h(t)', 't')

Warning: Explicit integral could not be found.
> In sym.int at 58
  In char.int at 9

ans =
int(h(t),t)
```

then the integral cannot be evaluated because within a string **h** is regarded as an unknown function. But if you type

```
>> syms t, int(h(t), t)
ans =
1/4*t^4
```

then the previous definition of **h** is substituted into the symbolic expression **h(t)** before the integration is performed.

Substitution

In Chapter 2 we described how to create an anonymous or inline function from an expression. You can then plug numbers into that function, to make a graph or table of values for instance. But you can also substitute numerical values directly into an expression with **subs**. See the subsection *Substituting in Symbolic Expressions* in Chapter 2 for instructions.

More about M-Files

Files containing MATLAB statements are called M-files. There are two kinds of M-files: *function M-files*, which accept arguments and produce output, and *script M-files*, which execute a series of MATLAB statements. In Chapter 3 we created and used both types. In this section we present additional information on M-files.

Variables in Script M-Files

When you execute a script M-file, the variables you use and define belong to your Workspace; i.e., they take on any values you assigned earlier in your MATLAB session, and they persist after the script has finished executing. Consider the following one-line script M-file, called `scriptex1.m`:

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

Typing **scriptex1** assigns the given vector to **u** but displays no output. Now consider another script M-file, called `scriptex2.m`:

```
n = length(u)
```

If you have not previously defined **u**, then typing **scriptex2** will produce an error message. However, if you type **scriptex2** after running **scriptex1**, then the definition of **u** from the first script will be used in the second script and the proper output `n = 4` will be displayed.

If you don't want the output of a script M-file to depend on any earlier computations in your MATLAB session, put the line **clear all** near the beginning of the M-file, as we suggested in *Structuring Script M-files* in Chapter 3.

Variables in Function M-Files

The variables used in a function M-file are *local*, meaning that they are unaffected by, and have no effect on, the variables in your Workspace. Consider the following

function M-file, called `sq.m`:

```
function z = sq(x)
% sq(x) returns the square of x.
z = x.^2;
```

Typing **`sq(3)`** produces the answer **`9`**, whether or not **`x`** or **`z`** is already defined in your Workspace. Running the M-file neither defines them, nor changes their definitions if they have been previously defined.

Structure of Function M-Files

The first line in a function M-file is called the *function definition line*; it specifies the function name, as well as the number and order of input and output arguments. Following the function definition line, there can be several comment lines that begin with a percent sign **`%`**. These lines are called *help text* and are displayed in response to the command **`help`**. In the M-file `sq.m` above, there is only one line of help text; it is displayed when you type **`help sq`**. The remaining lines constitute the *function body*; they contain the MATLAB statements that calculate the function values. In addition, there can be comment lines (lines beginning with **`%`**) anywhere in an M-file. All statements in a function M-file that normally produce output should end with a semicolon to suppress the output.

Function M-files can have multiple input and output arguments. Here is an example, called `polarcoordinates.m`, with two input and two output arguments.

```
function [r, theta] = polarcoordinates(x, y)
% polarcoordinates(x, y) returns the polar coordinates
% of the point with rectangular coordinates (x, y).
r = sqrt(x^2 + y^2);
theta = atan2(y,x);
```

If you type **`polarcoordinates(3,4)`**, only the first output argument is returned and stored in **`ans`**; in this case, the answer is **`5`**. To see both outputs, you must assign them to variables enclosed in square brackets:

```
>> [r, theta] = polarcoordinates(3,4)

r =
   5
theta =
   0.9273
```

By typing **`r = polarcoordinates(3,4)`** you can assign the first output argument to the variable **`r`**, but you cannot get only the second output argument; in fact, typing **`theta = polarcoordinates(3,4)`** will still assign the first output, **`5`**, to **`theta`**.

Complex Arithmetic

MATLAB does most of its computations using *complex numbers*; i.e., numbers of the form $a + bi$, where $i = \sqrt{-1}$, and a and b are real numbers. The complex

number i is represented as **i** in MATLAB. Although you may never have occasion to enter a complex number in a MATLAB session, MATLAB often produces an answer involving complex numbers. For example, many polynomials with real coefficients have complex roots.

```
>> solve('x^2 + 2*x + 2 = 0')

ans =
 -1+i
 -1-i
```

Both roots of this quadratic equation are complex numbers, expressed in terms of the number i. Some common functions also return complex values for certain values of the argument.

```
>> log(-1)

ans =
   0 + 3.1416i
```

You can use MATLAB to do computations involving complex numbers by entering numbers in the form **a + b*i**.

```
>> (2 + 3*i)*(4 - i)

ans =
   11.0000 + 10.0000i
```

Complex arithmetic is a powerful and valuable feature. Even if you don't intend to use complex numbers, you should be alert to the possibility of complex-valued answers when evaluating MATLAB expressions.

More on Matrices

In addition to the usual algebraic methods of combining matrices (e.g., matrix multiplication), we can also combine them element-wise. Specifically, if **A** and **B** are the same size, then **A.*B** is the *element-by-element* product of **A** and **B**, i.e., the matrix whose i, j element is the product of the i, j elements of **A** and **B**. Likewise, **A./B** is the element-by-element quotient of **A** and **B**, and **A.^c** is the matrix formed by raising each of the elements of **A** to the power **c**. More generally, if **f** is one of the built-in mathematical functions in MATLAB, or is a user-defined vectorized function, then **f(A)** is the matrix obtained by applying **f** element-by-element to **A**. See what happens when you type **sqrt(A)**, where **A** is the matrix defined at the beginning of the *Matrices* section of Chapter 2.

Recall that **x(3)** is the third element of a vector **x**. Likewise, **A(2,3)** represents the $2, 3$ element of **A**, i.e., the element in the second row and third column. You can specify submatrices in a similar way. Typing **A(2,[2 4])** yields the second and fourth elements of the second row of **A**. To select the second, third, and fourth elements of this row, type **A(2,2:4)**. The submatrix consisting of the elements in rows 2 and 3 and in columns 2, 3, and 4 is generated by **A(2:3,2:4)**. A colon

by itself denotes an entire row or column. For example, **A(:,2)** denotes the second column of **A**, and **A(3,:)** yields the third row of **A**.

MATLAB has several commands that generate special matrices. The commands **zeros(n,m)** and **ones(n,m)** produce $n \times m$ matrices of zeros and ones, respectively. Also, **eye(n)** represents the $n \times n$ identity matrix.

Solving Linear Systems

Suppose that **A** is a non-singular $n \times n$ matrix and **b** is a column vector of length n. Then typing **x = A\b** numerically computes the unique solution to **A*x = b**. Type **help mldivide** for more information.

If either **A** or **b** is symbolic rather than numerical, then **x = A\b** computes the solution to **A*x = b** symbolically. To calculate a symbolic solution when both inputs are numerical, type **x = sym(A)\b**.

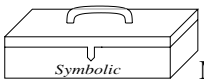
Calculating Eigenvalues and Eigenvectors

The eigenvalues of a square matrix **A** are calculated with **eig(A)**. The command **[U, R] = eig(A)** calculates both the eigenvalues and eigenvectors. The eigenvalues are the diagonal elements of the diagonal matrix **R**, and the columns of **U** are the eigenvectors. Here is an example illustrating the use of **eig**.

```
>> A = [3 -2 0; 2 -2 0; 0 1 1];
>> eig(A)

ans =
    1
   -1
    2

>> [U, R] = eig(A)

U =
        0  -0.4082    -0.8165
        0  -0.8165    -0.4082
   1.0000   0.4082    -0.4082


R =
   1     0     0
   0    -1     0
   0     0     2
```

The eigenvector in the first column of **U** corresponds to the eigenvalue in the first column of **R**, and so on. These are numerical values for the eigenpairs. To get symbolically calculated eigenpairs, type **[U, R] = eig(sym(A))**.

Doing Calculus with MATLAB



MATLAB has in its Symbolic Math Toolbox built-in commands for most of the computations of basic calculus.

Differentiation

You can use **diff** to differentiate symbolic expressions, and also to approximate the derivative of a function given numerically (say by an M-file).

```
>> syms x, diff(x^3)
```

```
ans =
3*x^2
```

Here MATLAB has figured out that the variable is **x**. (See *Default Variables* at the end of the chapter.) Alternatively,

```
>> f = @(x) x^3; diff(f(x))
```

```
ans =
3*x^2
```

The syntax for second derivatives is **diff(f(x), 2)**, and for nth derivatives, **diff(f(x), n)**. The command **diff** can also compute partial derivatives of expressions involving several variables, as in **diff(x^2*y, y)**, but to do multiple partials with respect to mixed variables you must use **diff** repeatedly, as in **diff(diff(sin(x*y/z), x), y))**. (Remember to declare **y** and **z** to be symbolic.)

There is one instance where differentiation must be represented by the letter **D**, namely when you need to specify a differential equation as input to a command. For example, to use the symbolic ODE solver on the differential equation $xy' + 1 = y$, you enter

```
>> dsolve('x*Dy + 1 = y', 'x')
```

```
ans =
1+x*C1
```

Integration

MATLAB can compute definite and indefinite integrals. Here is an indefinite integral:

```
>> int('x^2', 'x')
```

```
ans =
1/3*x^3
```

As with **diff**, you can declare **x** to be symbolic and dispense with the character string quotes. Note that MATLAB does not include a constant of integration; the output is a single antiderivative of the integrand. Now here is a definite integral:

```
>> syms x, int(asin(x), 0, 1)
ans =
1/2*pi-1
```

You are undoubtedly aware that not every function that appears in calculus can be symbolically integrated, and so numerical integration is sometimes necessary. MATLAB has two commands for numerical integration of a function $f(x)$: **quad** and **quadl**. We recommend **quadl**.

```
>> hardintegral = int(log(1 + x^2)*exp(-x^2), x, 0, 1)
Warning: Explicit integral could not be found.
> In sym.int at 58

hardintegral =
int(log(1+x^2)*exp(-x^2)),x = 0 .. 1)
>> quadl(@(x) log(1 + x.^2).*exp(-x.^2), 0, 1)
ans =
   0.1539
```

⇨ **The commands `quad` and `quadl` will not accept `Inf` or `-Inf` as a limit of integration (though `int` will). The best way to handle a numerical improper integral over an infinite interval is to evaluate it over intervals of increasing length until the result stabilizes.**

✓ You have another option. If you type **`double(int(hardintegral))`**, MATLAB uses the *Symbolic Math Toolbox* to evaluate the integral – even over an infinite range.

MATLAB can also do multiple integrals. The following command computes the double integral

$$\int_0^{\pi} \int_0^{\sin x} (x^2 + y^2)\, dy\, dx.$$

```
>> syms x y; int(int(x^2 + y^2, y, 0, sin(x)), 0, pi)
ans =
pi^2-32/9
```

Note that MATLAB presumes that the variable of integration in **`int`** is **`x`** unless you prescribe otherwise. Note also that the order of integration is as in calculus, from the "inside out." Finally, we observe that there is a numerical double-integral command **`dblquad`**, whose properties and use we will allow you to discover from the online help.

Limits

You can use **`limit`** to compute right- and left-handed limits and limits at infinity. For example, here is $\lim_{x\to 0} \sin(x)/x$.

```
>> syms x; limit(sin(x)/x, x, 0)
ans =
1
```

To compute one-sided limits, use the **'right'** and **'left'** options. For example:

```
>> limit(abs(x)/x, x, 0, 'left')
ans =
-1
```

Limits at infinity can be computed using the symbol **Inf**.

```
>> limit((x^4 + x^2 - 3)/(3*x^4 - log(x)), x, Inf)
ans =
1/3
```

Sums and Products

Finite numerical sums and products can be computed easily using the vector capabilities of MATLAB and the commands **sum** and **prod**. For example,

```
>> X = 1:7;
>> sum(X)
ans =
   28
>> prod(X)
ans =
   5040
```

You can do finite and infinite symbolic sums using the command **symsum**. To illustrate, here is the telescoping sum

$$\sum_{k=1}^{n} \left(\frac{1}{k} - \frac{1}{1+k} \right).$$

```
>> syms k n; symsum(1/k - 1/(k + 1)), 1, n)
ans =
-1/(n+1)+1
```

Here is the well-known infinite sum

$$\sum_{n=1}^{\infty} \frac{1}{n^2}.$$

```
>> symsum(1/n^2, 1, Inf)
ans =
1/6*pi^2
```

Another familiar example is the sum of the infinite geometric series:

```
>> syms a k; symsum(a^k, 0, Inf)
```

```
ans =
-1/(a-1)
```

Note, however, that the answer is valid only for $-1 < a < 1$.

☞ *The command* **symsum** *uses the variable* **k** *as its default variable. See the section below,* Default Variables*, for more information.*

Taylor Series

You can use **taylor** to generate Taylor polynomial expansions of a specified degree at a specified point. For example, to generate the Taylor polynomial up to degree 9 at $x = 0$ of the function $\sin x$, we enter

```
>> syms x; taylor(sin(x), x, 10)
```

```
ans =
x-1/6*x^3+1/120*x^5-1/5040*x^7+1/362880*x^9
```

Note that the command **taylor(sin(x), x, 11)** would give the same result, since there is no term of degree 10 in the Taylor expansion of $\sin x$. You can also compute a Taylor polynomial at a point other than the origin. For example:

```
>> taylor(exp(x), 4, 2)
```

```
ans =
exp(2)+exp(2)*(x-2)+1/2*exp(2)*(x-2)^2+1/6*exp(2)*(x-2)^3
```

computes a Taylor polynomial of e^x centered at the point $x = 2$.

The command **taylor** can also compute Taylor expansions at infinity.

```
>> taylor(exp(1/x^2), 6, Inf)
```

```
ans =
1+1/x^2+1/2/x^4
```

Default Variables

You can use any letters to denote variables in functions – either MATLAB's or the ones you define. For example, there is nothing special about the use of **t** in the following, any letter will do as well:

```
>> syms t; diff(sin(t^2))
```

```
ans =
2*cos(t^2)*t
```

However, if there are multiple variables in an expression and you employ a MATLAB command that does not make explicit reference to one of them, then either you must make the reference explicit or MATLAB will use a built-in hierarchy to decide which variable is the "one in play." For example: **solve('x + y = 3')** solves for **x**,

not **y**. If you want to solve for **y** in this example, you need to enter **solve('x + y = 3', 'y')**. MATLAB's default variable for **solve** is **x**. If there is no **x** in the equation(s) to solve, MATLAB looks for the nearest letter to **x** (where **y** takes precedence over **w**, but **w** takes precedence over **z**, etc.). Similarly for **diff**, **int**, and many other symbolic commands. Thus **syms w z; diff w*z** yields **z** as an answer. On occasion MATLAB assigns a different primary default variable – for example, the default independent variable for MATLAB's symbolic ODE solver **dsolve** is **t**, and, as we have noted earlier, the default variable for **symsum** is **k**. This is mentioned clearly in the online help for these commands. If you have doubt about the default variables for any MATLAB command, you should check the online help.

✓ Calculations with the Symbolic Math Toolbox are in fact sent by MATLAB to another program called Maple for processing. The Maple kernel, as it is called, performs the symbolic calculation and sends the result back to MATLAB. On rare occasions, you may want to access the Maple kernel directly. In the Professional Version of MATLAB, this can be done with the **maple** and **mhelp** commands. On similarly infrequent occasions, the output of a symbolic computation may involve functions that either do not exist in MATLAB, or are not properly converted into MATLAB functions. This may cause problems when you attempt to use the output in other MATLAB commands. For help in specific situations consult MATLAB help in the Help Browser and/or Maple help via **mhelp**.

Chapter 5

MATLAB Graphics

In this chapter we describe more of MATLAB's graphics commands and the most common ways of manipulating and customizing graphics. For an overview of commands, type **`help graphics`** (for general graphics commands), **`help graph2d`** (for two-dimensional graphics commands), **`help graph3d`** (for three-dimensional graphics commands), and **`help specgraph`** (for specialized graphing commands).

We have already discussed the commands **`plot`** and **`ezplot`** in Chapter 2. We will begin this chapter by discussing more uses of these commands, as well as some of the other most commonly used plotting commands. Then we will discuss methods for customizing and manipulating graphics. Finally, we will introduce some commands and techniques for creating and modifying images and sounds.

✓ For most types of graphs we describe below, there is a command like **`plot`** that draws the graph from numerical data, and a command like **`ezplot`** that graphs functions specified by string or symbolic input. The latter commands may be easier to use at first, but are more limited in their capabilities and less amenable to customization. Thus, we emphasize the commands that plot data, which are likely to be more useful to you in the long run.

Two-Dimensional Plots

Often one wants to draw a curve in the x-y plane, but with y not given explicitly as a function of x. There are two main techniques for plotting such curves: parametric plotting and contour or implicit plotting.

Parametric Plots

Sometimes x and y are both given as functions of some parameter. For example, the circle of radius 1 centered at $(0, 0)$ can be expressed in *parametric* form as $x = \cos(2\pi t)$, $y = \sin(2\pi t)$, where t runs from 0 to 1. Though y is not expressed as a function of x, you can easily graph this curve with **`plot`**, as follows (see Figure 5.1):

```
>> T = 0:0.01:1;
>> plot(cos(2*pi*T), sin(2*pi*T))
>> axis square
```

If we had used an increment of only 0.1 in the **`T`** vector, the result would have been a polygon with clearly visible corners. When your graph has corners that shouldn't be there, you should repeat the process with a smaller increment until you get a graph

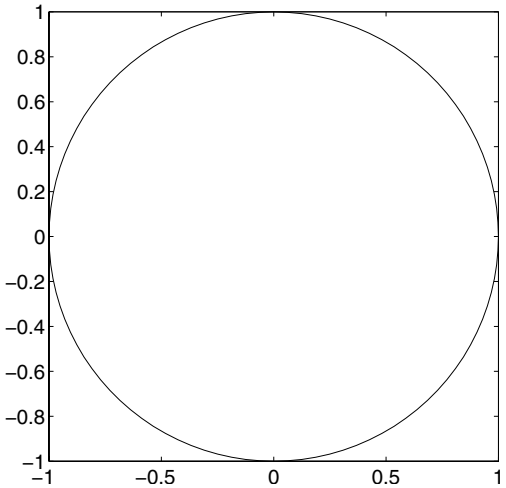


Figure 5.1. The Unit Circle $x^2 + y^2 = 1$.

that looks smooth. Here also **`axis square`** forces the same scale on both axes; without it the circle would look like an ellipse.

Parametric plotting is also possible with **`ezplot`**. You can obtain almost the same picture as Figure 5.1 with the command:

```
>> ezplot('cos(t)', 'sin(t)', [0 2*pi]); axis square
```

Notice that we used a semicolon after the **`ezplot`** command, but it did not prevent the graph from appearing. In general, the semicolon suppresses only text output.

Contour Plots and Implicit Plots

A *contour plot* of a function of two variables is a plot of the *level curves* of the function, i.e., sets of points in the x-y plane where the function assumes a constant value. For example, the level curves of $x^2 + y^2$ are circles centered at the origin, and the *levels* are the squares of the radii of the circles. Contour plots are produced in MATLAB with **`meshgrid`** and **`contour`**. The command **`meshgrid`** produces a grid of points in a rectangular region, with a specified spacing. This grid is used by **`contour`** to produce a contour plot in the specified region.

You can make a contour plot of $x^2 + y^2$ as follows (see Figure 5.2):

```
>> [X Y] = meshgrid(-3:0.1:3, -3:0.1:3);
>> contour(X, Y, X.^2 + Y.^2); axis square
```

You can specify particular level sets by including an additional vector argument to **`contour`**. For example, to plot the circles of radius 1, $\sqrt{2}$, and $\sqrt{3}$, type

```
>> contour(X, Y, X.^2 + Y.^2, [1 2 3])
```

The vector argument must contain at least two elements, so, if you want to plot a

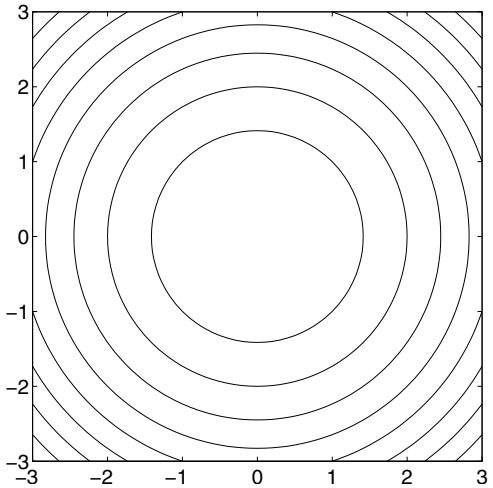


Figure 5.2. Contour Plot of $x^2 + y^2$.

single level set, you must specify the same level twice. This is quite useful for implicit plotting of a curve given by an equation in x and y. For example, to plot the circle of radius 1 about the origin, type

```
>> contour(X, Y, X.^2 + Y.^2, [1 1])
```

Or to plot the lemniscate $x^2 - y^2 = (x^2 + y^2)^2$, rewrite the equation as

$$(x^2 + y^2)^2 - x^2 + y^2 = 0$$

and type (see Figure 5.3)

```
>> [X Y] = meshgrid(-1.1:0.01:1.1, -1.1:0.01:1.1);
>> contour(X, Y, (X.^2 + Y.^2).^2 - X.^2 + Y.^2, [0 0])
>> axis square
>> title('The lemniscate x^2-y^2=(x^2+y^2)^2')
```

☞ *In this case, we used* ^ *to produce exponents in the title. You can also use* _ *for subscripts, and to produce a Greek letter, precede its name with a backslash – for example,* **\theta**. *Type* **doc title** *and look under "Examples" for other tricks with titles; these features also apply to labeling commands like* **xlabel** *and* **ylabel**. *For more on annotating graphs, see the section* Customizing Graphics *later in this chapter.*

You can also do contour plotting with the command **ezcontour**, and implicit plotting of a curve $f(x, y) = 0$ with **ezplot**. In particular, you can obtain almost the same picture as Figure 5.2 with the command:

```
>> ezcontour('x^2 + y^2', [-3, 3], [-3, 3]); axis square
```

and almost the same picture as Figure 5.3 with the command:

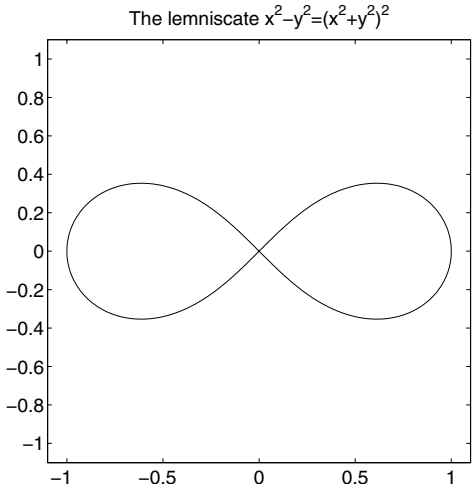


Figure 5.3. A Lemniscate.

```
>> ezplot('(x^2 + y^2)^2 - x^2 + y^2', ...
[-1.1, 1.1], [-1.1, 1.1]); axis square
```

Field Plots

The MATLAB routine **quiver** is used to plot vector fields or arrays of arrows. The arrows can either be located at equally spaced points in the plane (if x- and y-coordinates are not given explicitly), or they can be placed at specified locations. Sometimes some fiddling is required to scale the arrows so that they don't come out looking too big or too small. For this purpose, **quiver** takes an optional scale-factor argument. The following code, for example, plots a vector field with a "saddle point," corresponding to a combination of an attractive force pointing toward the x-axis and a repulsive force pointing away from the y-axis. The output is shown in Figure 5.4.

```
>> [x, y] = meshgrid(-1.1:0.2:1.1, -1.1:0.2:1.1);
>> quiver(x, -y); axis equal; axis off
```

Three-Dimensional Plots

MATLAB has several routines for producing three-dimensional plots.

Curves in Three-Dimensional Space

For plotting curves in 3-space, the basic command is **plot3**. It works like **plot**, except that it takes three vectors instead of two, one for the x-coordinates, one for the

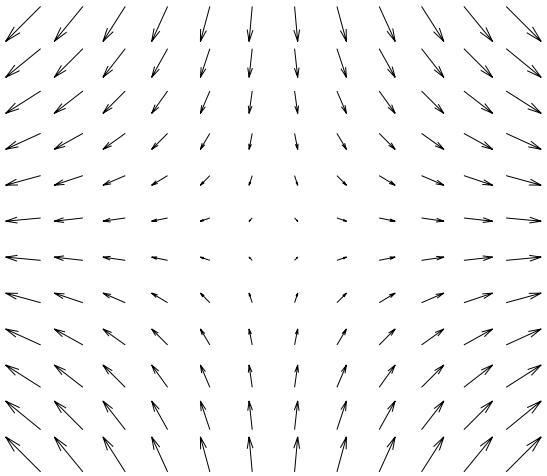


Figure 5.4. A Vector-Field Plot of $(x, -y)$.

y-coordinates, and one for the z-coordinates. For example, we can plot a helix with

```
>> T = -2:0.01:2;
>> plot3(cos(2*pi*T), sin(2*pi*T), T)
```

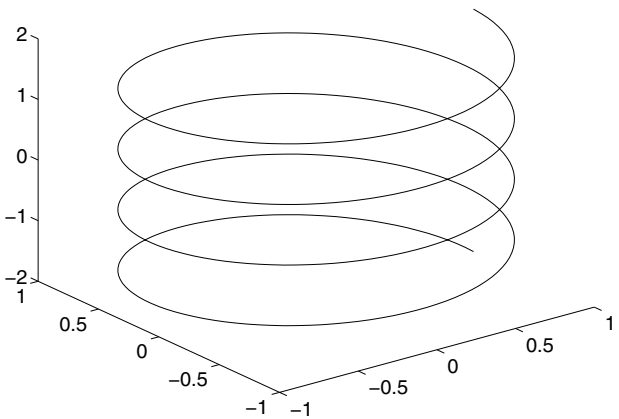


Figure 5.5. The Helix $x = \cos(2\pi z)$, $y = \sin(2\pi z)$.

There is also a three-dimensional analog to **`ezplot`** called **`ezplot3`**; you can instead plot the helix in Figure 5.5 with

```
>> ezplot3('cos(2*pi*t)', 'sin(2*pi*t)', 't', [-2, 2])
```

Surfaces in Three-Dimensional Space

There are two basic commands for plotting surfaces in 3-space: **mesh** and **surf**. The former produces a transparent "mesh" surface, the latter an opaque shaded one. There are two different ways of using each command, one for plotting surfaces in which the z-coordinate is given as a function of x and y, and one for *parametric surfaces* in which x, y, and z are all given as functions of two other parameters. Let us illustrate the former with **mesh** and the latter with **surf**.

To plot $z = f(x, y)$, one begins with a **meshgrid** command as in the case of **contour**. For example, the "saddle surface" $z = x^2 - y^2$ can be plotted with

```
>> [X,Y] = meshgrid(-2:0.1:2, -2:0.1:2);
>> Z = X.^2 - Y.^2; mesh(X, Y, Z)
```

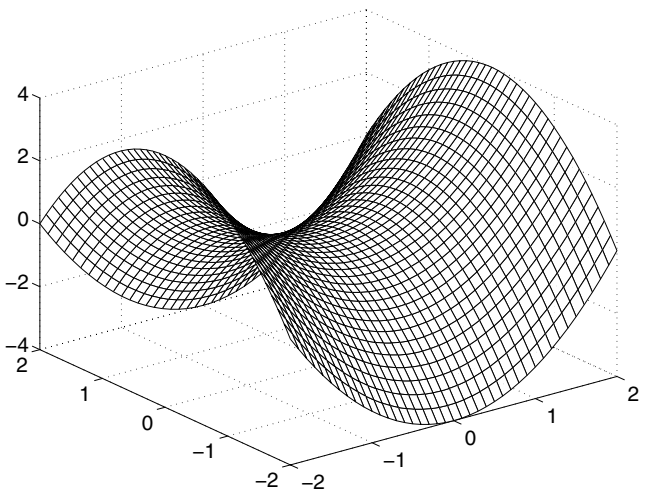


Figure 5.6. The Surface $z = x^2 - y^2$.

The resulting graph looks better on the computer screen since MATLAB shades the surface with a color scheme depending on the z-coordinate. We could have produced an opaque surface instead by replacing **mesh** with **surf**.

There are also shortcut commands **ezmesh** and **ezsurf**; you can obtain a result very similar to Figure 5.6 with

```
>> ezmesh('x^2 - y^2', [-2, 2, -2, 2])
```

If one wants to plot a surface that cannot be represented by an equation of the form $z = f(x, y)$, for example the sphere $x^2 + y^2 + z^2 = 1$, then it is better to parameterize the surface using a suitable coordinate system, in this case cylindrical or spherical coordinates. For example, we can take as parameters the vertical coordinate z and the polar coordinate θ in the x-y plane. If r denotes the distance to the z-axis, then the equation of the sphere becomes $r^2 + z^2 = 1$, or $r = \sqrt{1 - z^2}$, and so

$x = \sqrt{1-z^2}\cos\theta$, $y = \sqrt{1-z^2}\sin\theta$. Thus we can produce our plot with

```
>> [Z, Theta] = meshgrid(-1:0.1:1, (0:0.1:2)*pi);
>> X = sqrt(1 - Z.^2).*cos(Theta);
>> Y = sqrt(1 - Z.^2).*sin(Theta);
>> surf(X, Y, Z); axis square
```

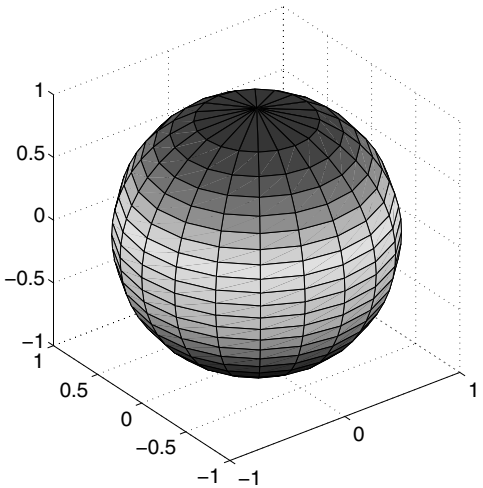


Figure 5.7. The Unit Sphere $x^2 + y^2 + z^2 = 1$.

You can also plot a surface parametrically with **ezmesh** or **ezsurf**; you can obtain a result very similar to Figure 5.7 with

```
>> ezsurf('sqrt(1-z^2)*cos(t)', ...
'sqrt(1-z^2)*sin(t)', 'z', [0, 2*pi, -1, 1]); axis equal
```

Notice that we had to specify the **t** range first because it is alphabetically before **z**, even though **z** occurs before **t** in the strings we entered. Rather than remember such a rule, in MATLAB 7 you can enter functions of more than one variable as anonymous functions, for example **@(z,t) sqrt(1-z.^2).*cos(t)**; then, since you used **z** as the first input variable, you would specify its range first.

Figure Windows

So far we have discussed only graphics commands that produce or modify a single plot. But MATLAB is also capable of opening multiple figure windows or combining several plots in one figure window. Here we discuss some basic methods for managing and manipulating figure windows.

Multiple Figure Windows

When you execute the first plotting command in a given MATLAB session, the graph appears in a new window labeled "Figure 1". Subsequent graphics commands either modify or replace the graph in this window. You have seen that **`hold on`** directs that new plotting commands should add to, rather than replace, the current graph. If instead you want to create a new graph in a separate window while keeping the old graph on your screen, type **`figure`** to open a new window, labeled "Figure 2". Alternatively, you can select a New Figure from the **File** menu in either your main MATLAB window or the first figure window. Subsequent graphics commands will affect only this window, until you change your *current figure* again with another **`figure`** command (e.g., type **`figure(1)`** to switch the current figure back to "Figure 1"), or by bringing another figure to the foreground with the mouse. When multiple figure windows are open, you can find out which is the current one by typing **`gcf`**, short for "get current figure." Finally, you can delete figure windows in the usual manner with the mouse, or with the command **`close`**; see the online help for details.

The Figure Toolbar

Each figure window has a tool bar underneath its menu bar with shortcuts for several menu items, including on the left icons for opening, saving, and printing figures. Near the middle, there are several icons that correspond to items in the **Tools** menu.

The two icons with plus and minus signs control zooming in and out. Click on the icon with a plus sign, and then click on a point in your graph to zoom in near that point. You can click and zoom multiple times; each zoom changes the scale on both axes by a factor of roughly 2. However, don't click too fast, because double-clicking resets the graph to its original state. Clicking on the icon with the minus sign allows you to zoom out gradually. MATLAB 7 also has an icon in the shape of a hand that allows you to click and drag the graph to pan both horizontally and vertically within the current axes. More zooming options are available by right-clicking in the figure window while zooming, by using the **Options** submenu of the **Tools** menu, or by using the command **`zoom`** (see its online help).

Clicking the next icon to the right with the circular arrow allows you to rotate three-dimensional (3D) graphics. For more 3D effects, select the Camera Toolbar from the **View** menu (in MATLAB 6 and higher). You can also change the viewpoint with the command **`view`**. In particular, the command **`view(2)`** projects a figure into the x-y plane (by looking down on it from the positive z-axis), and the command **`view(3)`** views it from the default direction in 3-space, which is in the direction looking toward the origin from a point far out on the ray $x = -0.5272t$, $y = -0.6871t$, $z = 0.5t$, $t > 0$.

✓ In MATLAB, any two-dimensional plot can be "viewed in 3D," and any three-dimensional plot can be projected into the plane. Thus Figure 5.5 above (the helix), if followed by the command **`view(2)`**, produces a circle.

In MATLAB 7, clicking the next icon to the right of the rotate icon enables the Data Cursor, which allows you to display the coordinates of a point on a curve by

clicking on or near the curve. Right-clicking in the figure window gives several options; changing Selection Style to Mouse Position will allow you to click on an arbitrary point on the curve rather than just a data point. (Remember that the curves plotted by MATLAB are piecewise-linear curves connecting a finite number of data points.) This can be useful especially after zooming, because data points may then be spaced far apart. Another way to get coordinates of a point (in earlier versions as well as MATLAB 7) is by typing **`ginput(1)`** in the Command Window; this allows you to click on any point in the figure window, not just on a curve. While this gives you more flexibility, if you want the precise coordinates of a point on a curve, it is best to use the Data Cursor, because it will always highlight a point on the curve even if you don't click exactly on the curve.

Also in MATLAB 7 only, the rightmost icon on the Figure Toolbar opens three windows surrounding the figure; these are collectively known as Plot Tools, and you can also open them with the command **`plottools`**. You can also control the display of these three windows – the Figure Palette, the Plot Browser, and the Property Editor – individually from the **View** menu. These windows enable various means of editing figures. Many of these capabilities are also available in the **Insert** and **Tools** menus, and from the Plot Edit Toolbar in the **View** menu. We will briefly discuss some editing options in the *Customizing Graphics* section below, but there are many more possibilities and we encourage you to experiment with these tools.

✓ In earlier versions of MATLAB, as well as MATLAB 7, more limited editing capabilities are available by clicking the arrow icon to the right of the print icon on the Figure Toolbar and then right-clicking in the figure.

Combining Plots in One Window

The command **`subplot`** divides the figure window into an array of smaller plots. The first two arguments give the dimensions of the array of subplots, and the last argument gives the number of the subplot (counting from left to right across the first row, then from left to right across the next row, and so on) in which to put the output of the next graphing command. The following example, whose output appears in Figure 5.8, produces a 2×2 array of plots of the first four Bessel functions J_n, $0 \leq n \leq 3$.

```
>> X = 0:0.05:40;
>> for n = 1:4
    subplot(2,2,n)
    plot(X, besselj(n - 1, X))
end
```

✓ In MATLAB you can also create subplots using the Figure Palette, which you can enable from the **View** menu or as part of the Plot Tools described above.

☆ Customizing Graphics

☞ *This is a more advanced topic; if you wish you can skip it on a first reading.*

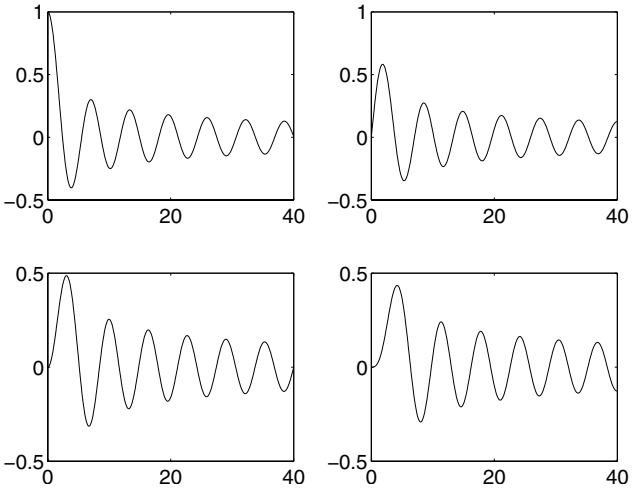


Figure 5.8. Bessel Functions $J_0(x)$ (upper left), $J_1(x)$ (upper right), $J_2(x)$ (lower left), and $J_3(x)$ (lower right).

So far in this chapter, we have discussed some commonly used MATLAB routines for generating and manipulating plots. But often, to get a more precise result, you need to customize or edit the graphics these commands produce. In order to do this, you must understand a few basic principles concerning the way MATLAB stores and displays graphics. For most purposes, the discussion here will be sufficient. But if you need more information, you may want to consult one of the books devoted exclusively to MATLAB graphics, such as *Using MATLAB Graphics*, which comes free (in PDF format) with the software and can be accessed in the "Printable Documentation" section in the Help Browser (or under "Full Documentation Set" from the **`helpdesk`** in MATLAB 5.3), or P. Marchand & O. Holland, *Graphics and GUIs with MATLAB*, 3rd ed., Chapman & Hall/CRC, London, 2002.

Once you have created a figure, there are two basic ways to manipulate it. The current figure can be modified by typing MATLAB commands in the Command Window, such as the commands **`title`** and **`axis square`** that we have already encountered. Or you can modify the figure with the mouse, using the menus and icons in the figure window itself. Almost all of the text commands have counterparts that can be executed directly in the figure window. So why bother learning both techniques? The reason is that editing in the figure window is often more convenient, especially when one wishes to "experiment" with multiple changes, while editing a figure with MATLAB commands in an M-file makes your customizations reproducible. So the true MATLAB expert uses both techniques. While the text commands generally remain the same from one version of MATLAB to the next, the figure-window menus and tools are significantly different in MATLAB 5.3, 6, and 7. All of these versions have a Property Editor, but it is accessed in different ways. In MATLAB 7, you can open it with Plot Tools as described above, or from the **View** menu. In MATLAB 6, select

Edit:**Current Object Properties...**. In MATLAB 5.3, select **File**:**Property Editor...**.

To modify objects in the figure window with the mouse, editing must be enabled in that window. In MATLAB 6 and later, you can enable or disable editing by selecting **Tools**:**Edit Plot** or by clicking the arrow icon to the right of the print icon. When editing is enabled, this arrow icon is highlighted, and there is a check mark next to **Edit Plot** in the **Tools** menu. In several places below we will tell you to click on an object in the figure window in order to edit it. When you click on the object, it should be highlighted with small black squares. If this doesn't happen, then you need to enable editing.

Annotation

In order to insert labels or text into a plot, you can use the commands **`text`**, **`xlabel`**, **`ylabel`**, **`zlabel`**, and **`legend`**, in addition to **`title`**. As the names suggest, **`xlabel`**, **`ylabel`**, and **`zlabel`** add text next to the coordinate axes, **`legend`** puts a "legend" on the plot, and **`text`** adds text at a specific point. These commands take various optional arguments that can be used to change the font family and font size of the text. As an example, let's illustrate how to modify our plot of the lemniscate (Figure 5.3) by adding and modifying text:

```
>> title('The lemniscate x^2-y^2=(x^2+y^2)^2', 'FontSize', ...
20, 'FontName', 'Helvetica', 'FontWeight', 'bold')
>> text(0, 0, ' \leftarrow a node, also an inflection', ...
'FontSize', 12)
>> text(0.2, -0.1, 'point for each branch', 'FontSize', 12)
>> xlabel x, ylabel y
```

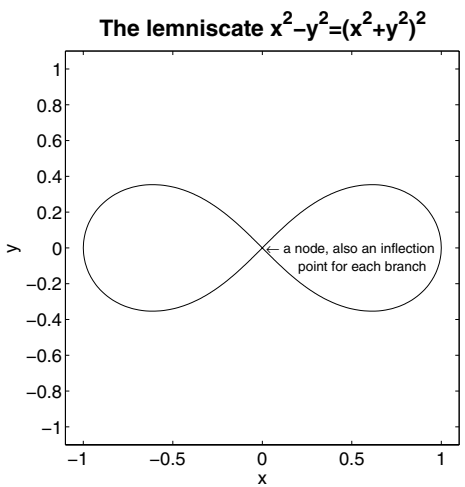


Figure 5.9. The Lemniscate from Figure 5.3 with Annotation and a Larger Title.

Notice that many symbols (such as the arrow pointing to the left in Figure 5.9) can be inserted into a text string by calling them with names starting with \. (If you've used the scientific typesetting program TEX, you'll recognize the convention here.) In most cases the names are self-explanatory. For example, you get a Greek π by typing **\pi**, a summation sign $\sum$ by typing either **\Sigma** (for a capital 'sigma') or **\sum**, and arrows pointing in various directions with **\leftarrow**, **\uparrow**, and so on. For more details and a complete list of available symbols, see the listing for "Text Properties" in the Help Browser (you can find this listing from the "Search" tab in the Help Browser, or type **doc text** and click on "Text Properties" at the bottom of its page).

✓ In MATLAB 6 and later, you can insert the same types of annotation using the **Insert** menu in the Figure Window. In MATLAB 7, many more annotations are available by enabling the Plot Edit Toolbar, the Figure Palette, and/or the Property Editor. You can also use the Property Editor to change the font of a text label; click on the text you want to change, then go to the Property Editor.

Change of Plot Style

Another important way to change the style of graphics is to modify the color or line style in a plot or to change the tick marks and labeling on the axes. Within a **plot** command, you can change the color of a graph, or plot with a dashed or dotted line, or mark the plotted points with special symbols, simply by adding a string third argument for every x-y pair. Symbols for colors are **'y'** for yellow, **'m'** for magenta, **'c'** for cyan, **'r'** for red, **'g'** for green, **'b'** for blue, **'w'** for white, and **'k'** for black. Symbols for point markers include **'o'** for a circle, **'x'** for a cross, **'+'** for a plus sign, and **'*'** for a star. Symbols for line styles include **'-'** for a solid line, **':'** for a dotted line, **'--'** for a dashed line. If a point style is given but no line style, then the points are plotted but no curve is drawn connecting them. (The same methods work with **plot3** in place of **plot**.) For example, you can produce a solid red sine curve together with a dotted blue cosine curve, marking all the local maximum points on each curve with a distinctive symbol of the same color as the plot, as follows:

```
>> X = (-2:0.02:2)*pi; Y1 = sin(X); Y2 = cos(X);
>> plot(X, Y1, 'r-', X, Y2, 'b:'); hold on
>> X1 = [-3*pi/2 pi/2]; Y3 = [1 1]; plot(X1, Y3, 'r*')
>> X2 = [-2*pi 0 2*pi]; Y4 = [1 1 1]; plot(X2, Y4, 'b+')
>> axis([-7 7 -1.1 1.1])
```

Here you may want the tick marks on the x-axis located at multiples of π. This can be done with the command **set**, which is used to change various properties of graphics. To apply it to the axes, it has to be combined with the command **gca**, which stands for "get current axes." The code

```
>> set(gca, 'XTick', (-2:2)*pi, 'XTickLabel', ...
'-2pi|-pi|0|pi|2pi', 'FontSize', 16)
```

in combination with the code above gets the current axes, sets the ticks on the x-axis to go from -2π to 2π in multiples of π, and then labels these ticks symbolically

(rather than in decimal notation, which is ugly here). It also increases the size of the labels to a 16-point font. The result is shown in Figure 5.10.

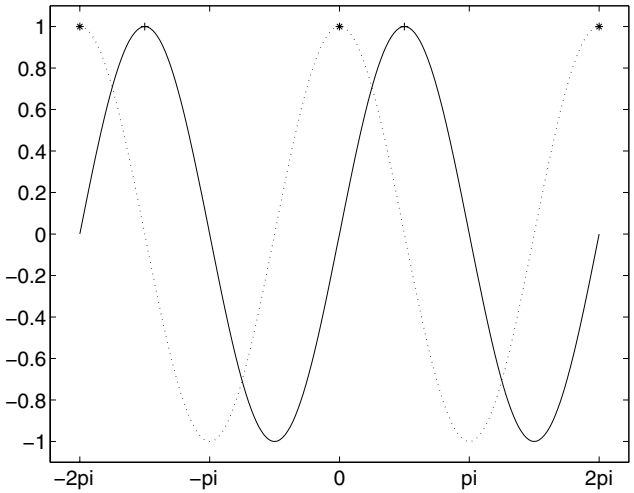


Figure 5.10. Two Periods of $\sin x$ and $\cos x$.

Incidentally, you might wonder how to label the ticks as -2π, $-\pi$, etc., instead of **`-2pi`**, **`-pi`**, and so on. This is trickier but you can do it by typing

```
>> set(gca, 'FontName', 'Symbol')
>> set(gca, 'XTickLabel', '-2p|-p|0|p|2p')
```

since in the Symbol font, π occupies the slot held by **`p`** in text fonts.

✓ In MATLAB 7, you can again use the Property Editor to make the same types of stylistic changes. Click on a curve and go to the Property Editor to change its style, color, width, etc. In a 3D plot, you can click on a surface and see options for changing its coloring and other properties. To change the tick marks, labeling, etc., click on the axes or a blank area inside them to focus the Property Editor on the axes. In MATLAB 6, click on a curve and select **Edit**:**Current Object Properties...** to modify its properties, or select **Edit**:**Axes Properties...** to change the font for the tick labels.

Full-Fledged Customization

What about changes to other aspects of a plot? The commands **`get`** and **`set`** can be used to obtain a complete list of the properties of the objects in a figure window and then to modify them. These objects and properties are arranged in a hierarchical structure, with each object identified by a floating-point number called a *handle*. If you type **`get(gcf)`**, you will "get" a (rather long) list of properties of the current figure (whose handle is returned by the function **`gcf`**). Some of these might read

```
Color = [0.8 0.8 0.8]
CurrentAxes = [151.001]
Children = [151.001]
```

Here **Color** gives the background color of the plot in red-green-blue (RGB) coordinates, where [0 0 0] is black and [1 1 1] is white; [0.8 0.8 0.8] is light gray. Notice that **CurrentAxes** and **Children** in this example have the same value, the one-element vector containing the funny-looking number 151.001. This number is the handle of the current axes, which would also be returned by the command **gca** ("get current axes"). The fact that this handle also shows up under **Children** indicates that the axes are "children" of the figure, i.e., they lie one level down in the hierarchical structure. Typing **get(gca)** would then give you a list of axis properties, including the handles of further **Children** such as **Line** objects, within which you would find the **XData** and **YData** encoding the actual plot.

⇨ **In the example above,** 151.001 **is not the exact value of the axes handle, just its first few decimal places. So, typing** **get(151.001)** **would yield an error message. To retrieve the exact value of** **Children** **in the example above, type** **get(gcf, 'Children')**. **In many cases, a figure will have multiple children, in which case this command will return a vector of handles.**

Once you have located the properties you're interested in, they can be changed with **set**. For example,

```
>> set(gcf, 'Color', [1 0 0])
```

changes the background color of the border of the figure window to red, and

```
>> set(gca, 'Color', [1 1 0])
```

changes the background color of the plot itself (a child of the figure window) to yellow (which in the RGB scheme is half red, half green).

This "one at a time" method for locating and modifying figure properties can be speeded up using the command **findobj** to locate the handles of all the descendents (the main figure window, its children, children of children, etc.) of the current figure. One can also limit the search to handles containing elements of a specific type. For example, **findobj('type', 'line')** hunts for all handles of objects containing a Line element. Once you have located these, you can use **set** to change the **LineStyle** from solid to dashed, etc. In addition, the low-level graphics commands **line**, **rectangle**, **fill**, **surface**, and **image** can be used to create new graphics elements within a figure window.

✓ In MATLAB 7, you can also see and modify a full list of properties for the figure, axes, or other object using the Property Editor. Click on the object and then click on the "Inspector..." button in the Property Editor. To select the figure itself, click on the border of the figure, outside the axes.

As an example of these techniques, the following code creates a chessboard on a white background, as shown in Figure 5.11.

```
>> white = [1 1 1]; gray = 0.7*white;
>> a = [0 1 1 0]; b = [0 0 1 1]; c = [1 1 1 1];
>> figure; hold on
>> for k = 0:1, for j = 0:2:6
    fill(a'*c + c'*(0:2:6) + k, b'*c + j + k, gray)
end, end
>> plot(8*a', 8*b', 'k')
>> set(gca, 'XTickLabel', [], 'YTickLabel', [])
>> set(gcf, 'Color', white); axis square
```

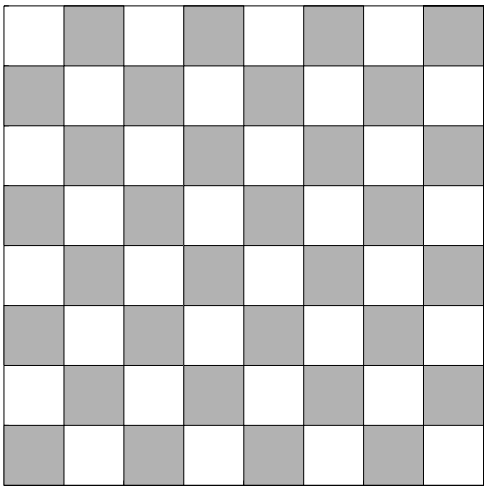


Figure 5.11. A Chessboard.

Here **`white`** and **`gray`** are the RGB codings for white and gray. The double **`for`** loop draws the 32 dark squares on the chessboard, using **`fill`**, with **`j`** indexing the dark squares in a single vertical column, with **`k = 0`** giving the odd-numbered rows, and with **`k = 1`** giving the even-numbered rows. Notice that **`fill`** here takes three arguments: a matrix, each of whose columns gives the x-coordinates of the vertices of a polygon to be filled (in this case a square), a second matrix whose corresponding columns give the y-coordinates of the vertices, and a color. We've constructed the matrices with four columns, one for each of the solid squares in a single horizontal row. The **`plot`** command draws the solid black line around the outside of the board. Finally, the first **`set`** command removes the printed labels on the axes, and the second **`set`** command resets the background color to white.

✰ Images, Animations, and Sound

MATLAB is also able to create and manipulate full-color images, animations, and sound files. In addition to the command-line methods described below, you can open

a media file in a format that MATLAB supports by double-clicking on it in the Current Directory Browser or by selecting **File**:**Import Data...**.

Images

MATLAB can read, write, and edit images, such as those created by a digital camera, found on the World Wide Web, or created from within MATLAB. An image is simply a two-dimensional array of tiny colored squares called "pixels." The image may be stored in a file in a variety of formats, including `png`, `jpeg`, and `gif`. In MATLAB, a color image with height h pixels and width w pixels is generally stored in one of two ways: as an RGB image or an indexed image. An RGB image is represented as an $h \times w \times 3$ array, so that the color of each pixel is specified by three values: a red intensity, a green intensity, and a blue intensity. (A similar type of image is grayscale, which is represented as an $h \times w$ array of pixel intensities.) An indexed image consists of an $h \times w$ array together with an auxiliary $c \times 3$ array called a "colormap"; each element in the first array represents the index of a row in the colormap, and this row gives the RGB values for the corresponding pixel. (What MATLAB calls RGB format is often called "true color" in graphics programming; an indexed image is often called "pseudocolor.")

The command **`imread`** will read an image from any of a large variety of image-file formats; see the online help for supported formats. Some formats, such as `png`, can store RGB or indexed images. Other formats can store only one type or the other; `gif` images are always indexed, whereas `jpeg` images are never indexed. Most images you find on the World Wide Web will be stored in RGB format unless they are `gif` files. To read an RGB image from the file `picture.png` and store it in the array **`rgbpic`**, type

```
>> rgbpic = imread('picture.png');
```

You can read an indexed image with **`imread`** by assigning its output to two variables, one for the image array and one for the colormap. You can then convert these arrays to a single RGB array with **`ind2rgb`**. For example:

```
>> [indpic, map] = imread('picture.gif');
>> rgbpic = ind2rgb(indpic, map);
```

✓ Converting from RGB to indexed format is harder because generally the number of different colors in the image must be reduced; colormaps often have 256 or fewer colors. MATLAB's Image Processing Toolbox has a command **`rgb2ind`** that offers several algorithms for making this conversion.

The command **`image`** displays an image in a figure window. For an RGB image **`rgbpic`**, simply type

```
>> image(rgbpic)
>> axis equal tight
```

The second command is not necessary, but ensures that the image is displayed in its intended aspect ratio. For an indexed image **`indpic`** with colormap **`map`**, type

```
>> image(indpic)
>> colormap(map)
```

The command **colormap** changes the colormap of the current axes or, with no input arguments, outputs the current colormap.

You can edit an image by changing the values in the image array. Notice that when you display an image, the axes are labeled with the indices of the image array. You can use the zoom feature of the figure window (see the *Figure Windows* section earlier in this chapter) to locate more precisely the indices of a particular feature of the image, or even an individual pixel. Theoretically at least, you can then edit the image in any way you want by changing the numbers in the array. Here we describe how to carry out several common manipulations in a practical manner.

⇨ **We will describe mainly how to edit RGB images, since this format allows you more freedom and you can always convert from indexed format to RGB as described above. However, RGB images are stored in three-dimensional arrays, which may take some time to get used to. Previously in this book we have discussed only two-dimensional arrays, and some MATLAB commands that manipulate two-dimensional arrays do not work for three-dimensional arrays.**

To reverse an image up-to-down or left-to-right, you simply need to reverse the array with **flipdim**. For example,

```
>> image(flipdim(rgbpic, 2))
```

will display a left-to-right mirror image, while **flipdim(rgbpic, 1)** reverses the array up-to-down. (For indexed images and other two-dimensional arrays, you can use the more mnemonic commands **fliplr** and **flipud** instead.)

To crop an image, select the appropriate subarray. For example, to remove 50 pixels from the top and bottom of the image and 100 pixels from the left and right, type

```
>> newpic = rgbpic(51:end-50, 101:end-100, :);
```

You can then display the cropped image **newpic** as described above, or save it as described below.

To examine the color of an individual pixel, you can display its RGB values in the Command Window and/or display it in a figure window. For example, to examine the pixel of **rgbpic** in the lower left-hand corner, type:

```
>> rgbpic(end, 1, :)

ans(:,:,1) =
   240
ans(:,:,2) =
   114
ans(:,:,3) =
   14
```

The output above gives (hypothetical) red, green, and blue values for the pixel. To see this color in the current figure window, type **image(rgbpic(end, 1, :))**. You

can then adjust the color as desired; for instance, typing **rgbpic(end, 1, 2) = 180** will increase the green intensity, making the color lighter and more yellow.

Of course, changing a single pixel will not change the appearance of the figure much, but you can also change the color of a whole block of pixels or the entire image in a similar manner. For example, to black out a rectangle within the picture, set all of the values in the corresponding subarray to 0. Thus, **rgbpic(40:60, 90:110, :) = 0** will change to black all the pixels in a 21-by-21 square centered 50 pixels from the top and 100 pixels from the left of the image.

✓ Changing the numbers in the array as we have just described will not change the image displayed in the figure window until you issue a new **image** command.

In addition to manipulating images that you read into MATLAB, you can create your own images to visualize numerical data. Suppose, for example, that you have an array **temp** that contains temperatures for some geographical region. You can display the temperatures as an indexed image by typing

```
>> imagesc(temp)
```

The command **imagesc** works like **image**, except that it rescales the values in a two-dimensional array so that the highest number corresponds to the highest numbered color in the current colormap and the lowest number corresponds to the lowest numbered color. With the default colormap in effect, hot regions will be colored red and cold regions will be colored blue, with other colors representing intermediate temperatures.

To display the colormap next to the image, type **colorbar**. Notice that the numbers on this bar correspond to the numbers in your original array, and are independent of the rescaling done by **imagesc**. If you want a different colormap, you can create your own or use one of the other colormaps built into MATLAB; type **doc colormap** for a selection. For example, to transform the temperature map described above into a black-and-white image where white represents hot, black represents cold, and intermediate temperatures are in shades of gray, type **colormap(gray)**.

Finally, you can save an image in one of the standard formats like `png` with the command **imwrite**. For example, to save **newpic** to the file `newpict.png`, type

```
>> imwrite(newpic, 'newpict.png')
```

⇨ **In Chapter 3, we discussed how to save a figure in a format such as `png` using either `print` or File:Save As... from the figure window menu. This will save the entire figure, including the border and axis labels, into the image file. If you only want to save the image inside the axes, use `imwrite`.**

Animations

The simplest way to produce an animated picture is with **comet**, which produces a parametric plot of a curve (the way **plot** does), except that you can see the curve being traced out in time. For example,

```
>> T = (0:0.01:2)*pi;
>> figure, axis equal, axis([-1 1 -1 1]), hold on
>> comet(cos(T), sin(T))
```

displays uniform circular motion.

✓ We used **hold on** here not to save a previous graph, but to preserve the axis properties we had just set. Without **hold on**, MATLAB would revert to its default axes, and the curve would look elliptical rather than circular as it is being traced.

For more complicated animations, you can use **getframe** and **movieview**. The command **getframe** captures the active figure window for one frame of the movie, and **movieview** (available in MATLAB 6 and later) then plays back the result in a separate window. For example, the following commands produce a movie of a vibrating string.

```
>> X = 0:0.01:1;
>> for n = 0:50
    plot(x, sin(n*pi/5)*sin(pi*X)), axis([0, 1, -2, 2])
    M(n+1) = getframe;
end
>> movieview(M)
```

The **axis** command here is important, to ensure that each frame of the movie is drawn with the same coordinate axes. (Otherwise, the scale of the axes will be different in each frame, and the resulting movie will be totally misleading.) The semicolon after **getframe** is also important, in order to avoid the spewing forth of a lot of numerical data with each frame of the movie.

⇨ **Make sure that while MATLAB executes the loop that generates the frames, you do not cover the active figure window with another window (such as the Command Window). If you do, the contents of the other window will be stored in the frames of the movie.**

✓ You can also use **movie** (which, unlike **movieview**, is available in MATLAB 5.3) to play back the movie in the current figure window. This command allows additional options, such as varying the frame rate; see the online help for details.

Once you have created a movie, you can use **movie2avi** (in MATLAB 6 and later) to save it as an AVI file, which is a standard format that can be used in other movie-viewing programs, such as Windows Media Player and QuickTime. For example, to save the movie created above to the file `string.avi`, type **movie2avi(M, 'string.avi')**.

Sound

You can use **sound** to generate sound on your computer (provided that your computer is suitably equipped). This command takes a vector, views it as the waveform of a

sound, and "plays" it. A "sinusoidal" vector corresponds to a pure tone, and the frequency of the sinusoidal signal determines the pitch. Thus the following example plays the motto from Beethoven's Fifth Symphony:

```
>> x = (0:0.1:250)*pi; y = zeros(1,200); z = (0:0.1:1000)*pi;
>> sound([sin(x), y, sin(x), y, sin(x), y, sin(z*4/5), y, ...
sin(8/9*x), y, sin(8/9*x), y, sin(8/9*x), y, sin(z*3/4)]);
```

Notice that the zero vector **y** in this example creates a very short pause between successive notes.

For **sound**, the values in the input vector should be between -1 and 1. Within that range, the amplitude of the vector determines the volume of the sound; to play the sound above at half volume, you can multiply the input vector by 0.5. For a vector with an amplitude greater than 1, you can use **soundsc** to rescale the vector to the range -1 to 1 before playing it.

By default, the sound is played at 8192 samples per second, so the length of the vector, divided by 8192, is the length of the sound in seconds. You can change the sample rate with an optional second argument to **sound**; this will change both the pitch and the duration of the sound you hear.

Finally, you can read and write sound files in MATLAB, but only in two formats: `wav` and `au`. More popular formats such as `mp3` are not available, but you may have software that converts other formats to and from `wav`. The commands **wavread** and **wavwrite** read and write this format; see the online help for details.

Practice Set B

Calculus, Graphics, and Linear Algebra

Symbolic Problems 2, 3, 5–7, and parts of 10–12 require the Symbolic Math Toolbox. The others do not.

1. Use **`contour`** to do the following.

   (a) Plot the level curves of the function $f(x, y) = 3y + y^3 - x^3$ in the region where x and y are between -1 and 1 (to get an idea of what the curves look like near the origin), and in some larger regions (to get the big picture).

   (b) Plot the curve $3y + y^3 - x^3 = 5$.

   (c) Plot the level curve of the function $f(x, y) = y \ln x + x \ln y$ that contains the point $(1, 1)$.

2. Find the derivatives of the following functions, and if possible, simplify each answer:

   (a) $f(x) = 6x^3 - 5x^2 + 2x - 3$

   (b) $f(x) = \dfrac{2x - 1}{x^2 + 1}$

   (c) $f(x) = \sin(3x^2 + 2)$

   (d) $f(x) = \arcsin(2x + 3)$

   (e) $f(x) = \sqrt{1 + x^4}$

   (f) $f(x) = x^r$

   (g) $f(x) = \arctan(x^2 + 1)$.

3. See whether MATLAB can do the following integrals symbolically, and for the indefinite integrals, check the results by differentiating:

   (a) $\int_0^{\pi/2} \cos x \, dx$

   (b) $\int x \sin(x^2) \, dx$

   (c) $\int \sin(3x) \sqrt{1 - \cos(3x)} \, dx$

   (d) $\int x^2 \sqrt{x + 4} \, dx$

(e) $\int_{-\infty}^{\infty} e^{-x^2}\, dx$.

4. Compute the following integrals numerically using **`quadl`**.

   (a) $\int_0^{\pi} e^{\sin x} dx$.

   (b) $\int_0^1 \sqrt{x^3+1}\, dx$.

   (c) $\int_{-\infty}^{\infty} e^{-x^2} dx$. In this case, also find the error in the numerical answer, by comparing it with the exact answer found in Problem 3.

5. Evaluate the following limits:

   (a) $\lim_{x\to 0} \dfrac{\sin x}{x}$

   (b) $\lim_{x\to -\pi} \dfrac{1+\cos x}{x+\pi}$

   (c) $\lim_{x\to\infty} x^2\, e^{-x}$

   (d) $\lim_{x\to 1^-} \dfrac{1}{x-1}$

   (e) $\lim_{x\to 0^+} \sin\left(\dfrac{1}{x}\right)$.

6. Compute the following sums:

   (a) $\sum_{k=1}^{n} k^2$

   (b) $\sum_{k=0}^{n} r^k$

   (c) $\sum_{k=0}^{\infty} \dfrac{x^k}{k!}$; the MATLAB function **`factorial`** does not work with symbolic input, but you can use either **`sym('k!')`** or the gamma function $\Gamma(x) = \int_0^{\infty} e^{-t} t^{x-1}\, dt$, called **`gamma`** in MATLAB, which satisfies $\Gamma(k+1) = k!$

   (d) $\sum_{k=-\infty}^{\infty} \dfrac{1}{(z-k)^2}$.

7. Find the Taylor polynomial of the indicated degree n at the indicated point c for the following functions:

   (a) $f(x) = e^x$, $n = 6$, $c = 0$

   (b) $f(x) = \sin x$, $n = 4$ and 6, $c = 0$

   (c) $f(x) = \sin x$, $n = 5$, $c = 2$

   (d) $f(x) = \tan x$, $n = 6$, $c = 0$

   (e) $f(x) = \ln x$, $n = 4$, $c = 1$

(f) $f(x) = \mathrm{erf}(x)$, $n = 8$, $c = 0$.

8. Plot the following surfaces:

   (a) $z = \sin x \sin y$ for $-3\pi \le x \le 3\pi$ and $-3\pi \le y \le 3\pi$,

   (b) $z = (x^2 + y^2)\cos(x^2 + y^2)$ for $-1 \le x \le 1$ and $-1 \le y \le 1$.

9. Create a 17-frame movie, whose frames show filled red circles of radius $1/2$ centered at the points $\big(4\cos(j\pi/8), 4\sin(j\pi/8)\big)$, $j = 0, 1, \ldots, 16$. Make sure that all the circles are drawn on the same set of axes, and that they look like circles, not ellipses.

10. In this problem we use the *backslash* operator, or "left-matrix-divide" operator introduced in the section *Solving Linear Systems* of Chapter 4.

    (a) Use the backslash operator to solve the system of linear equations in Problem 3 of Practice Set A.

    (b) Now try the same method on Problem 4 of Practice Set A. MATLAB finds one, but not all, answer(s). Can you explain why? If not, see Problem 11 below, as well as part (d) of this problem

    (c) Next try the method on this problem:

$$\begin{aligned} w + 3x - 2y + 4z &= 1 \\ -2w + 3x + 4y - z &= 1 \\ -4w - 3x + y + 2z &= 1 \\ 2w + 3x - 4y + z &= 1. \end{aligned}$$

    Check your answer by matrix multiplication.

    (d) Finally, try the matrix division method on

$$\begin{aligned} ax + by &= u \\ cx + dy &= v. \end{aligned}$$

    Don't forget to declare the variables to be symbolic. Your answer should involve a fraction, and so will be valid only when its denominator is nonzero. Evaluate **`det`** on the coefficient matrix of the system. Compare this with the denominator.

11. We deal in this problem with 3×3 matrices, although the concepts are valid in any dimension.

    (a) Consider the rows of a square matrix A. They are vectors in 3-space and so span a subspace of dimension 3, 2, 1, or possibly 0 (if all the entries of A are zero). That number is called the *rank* of A. The MATLAB command **`rank`** computes the rank of a matrix. Try it on the four coefficient matrices in each of the parts of Problem 10. Comment on MATLAB's answer for the fourth one.

(b) An $n \times n$ matrix is *non-singular* if its rank is n. Which of the four you computed in part (a) are non-singular?

(c) Another measure of non-singularity is given by the *determinant* – a fundamental result in linear algebra is that a matrix is non-singular precisely when its determinant is non-zero. In that case a unique matrix B exists that satisfies $AB = BA =$ the identity matrix. We denote this inverse matrix by A^{-1}. MATLAB can compute inverses with **`inv`**. Compute **`det(A)`** for the four coefficient matrices, and for the non-singular ones, find their inverses. Note: the matrix equation $Ax = b$ has a unique solution, namely $x = A^{-1}b =$ **`A\b`**, when A is non-singular.

12. As explained in Chapter 4, when you compute **`[U, R] = eig(A)`**, each column of U is an eigenvector of A associated with the eigenvalue that appears in the corresponding column of the diagonal matrix R. This says exactly that $AU = UR$.

(a) Verify the equality $AU = UR$ for each of the coefficient matrices in Problem 10.

(b) In fact, rank$(A) =$ rank(U), so, when A is non-singular,

$$U^{-1}AU = R.$$

Thus, if two non-singular matrices A and B have the same set of eigenvectors, then the fact that diagonal matrices commute implies the same for A and B. Verify these facts for the two matrices

$$A = \begin{pmatrix} 1 & 0 & 2 \\ -1 & 0 & 4 \\ -1 & -1 & 5 \end{pmatrix}, \quad B = \begin{pmatrix} 5 & 2 & -8 \\ 3 & 6 & -10 \\ 3 & 3 & -7 \end{pmatrix};$$

that is, show that the matrices of eigenvectors are the "same" – that is, the columns are the same up to a scalar multiple – and verify that $AB = BA$.

13. This problem, having to do with genetic inheritance, is based on Chapter 12 in C. Rorres and H. Anton, *Applications of Linear Algebra*, 3rd ed., John Wiley & Sons, New York, 1984. In a typical inheritance model, a trait in the offspring is determined by the passing of a genotype from the parents, where there are two independent possibilities from each parent, say A and a, and each is equally likely. (A is the dominant gene, and a is recessive.) Then we have the following table of probabilities of the possible genotypes for the offspring for all possible combinations of the genotypes of the parents:

Genotype of Offspring			Parental Genotype			
	AA-AA	AA-Aa	AA-aa	Aa-Aa	Aa-aa	aa-aa
AA	1	1/2	0	1/4	0	0
Aa	0	1/2	1	1/2	1/2	0
aa	0	0	0	1/4	1/2	1

Now suppose that one has a population in which mating occurs only with one's identical genotype. (That's not far-fetched if we are considering a controlled plant or vegetable population.) Next suppose that x_0, y_0, and z_0 denote the percentages of the population with genotype AA, Aa, and aa, respectively, at the outset of observation. We then denote by x_n, y_n, and z_n the percentages in the nth generation. We are interested in knowing these numbers for large n, and how they depend on the initial population. Clearly

$$x_n + y_n + z_n = 1, \quad n \geq 0.$$

Now we can use the table to express a relationship between the nth and $(n+1)$st generations. Because of our presumption on mating, only the first, fourth and sixth columns are relevant. Indeed a moment's reflection reveals that we have

$$\begin{aligned} x_{n+1} &= x_n + \frac{1}{4}y_n \\ y_{n+1} &= \frac{1}{2}y_n \\ z_{n+1} &= z_n + \frac{1}{4}y_n. \end{aligned}$$

(a) Write these equations as a single matrix equation $X_{n+1} = MX_n$, $n \geq 0$. Explain carefully what the entries of the column matrix X_n are, and what the coefficients of the square matrix M are.

(b) Apply the matrix equation recursively to express X_n in terms of X_0 and powers of M.

(c) Next use MATLAB to compute the eigenvalues and eigenvectors of M.

(d) From Problem 12 you know that $MU = UR$, where R is the diagonal matrix of eigenvalues of M. Solve that equation for M. Can you see what $R_\infty = \lim_{n\to\infty} R^n$ is? Use that and your above expression of M in terms of R to compute $M_\infty = \lim_{n\to\infty} M^n$.

(e) Describe the eventual population distribution by computing $M_\infty X_0$.

(f) Check your answer by directly computing M^n for large specific values of M. (*Hint*: MATLAB can compute the powers of a matrix **`M`** by entering **`M^10`**, for example.)

(g) You can alter the fundamental presumption in this problem by assuming, alternatively, that all members of the nth generation must mate only with a parent whose genotype is purely dominant. Compute the eventual population distribution of that model. Chapters 12–14 in Rorres and Anton have other interesting models.

14. ☆ The French flag is 1.5 times as wide as it is high, and is divided into three vertical stripes, colored (in order) blue, white, and red. The Italian flag is the same, except that blue is replaced by green. Create a $200 \times 300 \times 3$ array representing the French flag, view it in a figure window, and convert it to a

`jpeg` file `tricolore.jpg`. Do the same with the Italian flag, this time converting to a file `italia.jpg`. Finally, create a movie showing the French flag "transform" into the Italian one.

Chapter 6

MATLAB Programming

Every time you create an M-file, you are writing a computer program using the MATLAB programming language. You can do quite a lot in MATLAB using no more than the most basic programming techniques that we have already introduced. In particular, we discussed simple loops (using **for**) and a rudimentary approach to debugging in Chapter 3. In this chapter, we will cover some further programming commands and techniques that are useful for attacking more complicated problems with MATLAB. If you are already familiar with another programming language, much of this material will be quite easy for you to pick up!

✓ Many MATLAB commands are themselves M-files, which you can examine using **type** or **edit**, e.g., enter **type isprime** to see the M-file for the command **isprime**. You can learn a lot about MATLAB programming techniques by inspecting the built-in M-files.

Branching

For many user-defined functions, you can use a function M-file that executes the same sequence of commands for each input. However, one often wants a function to perform a different sequence of commands in different cases, depending on the input. You can accomplish this with a branching command, and, as in many other programming languages, branching in MATLAB is usually done with the command **if**, which we will discuss now. Later we will describe the other main branching command, **switch**.

Branching with `if`

For a simple illustration of branching with **if**, consider the following function M-file `absval.m`, which computes the absolute value of a real number.

```
function y = absval(x)
if x >= 0
    y = x;
else
    y = -x;
end
```

The first line of this M-file states that the function has a single input **x** and a single output **y**. If the input **x** is non-negative, the **if** statement is determined by MATLAB to be true. Then the command between the **if** and the **else** statements is executed

to set **y** equal to **x**, while MATLAB skips the command between the **else** and **end** statements. On the other hand, if **x** is negative, then MATLAB skips to the **else** statement and executes the succeeding command, setting **y** equal to **-x**. As with a **for** loop, the indentation of commands above is optional; it is helpful to the human reader and is done automatically by MATLAB's built-in Editor/Debugger.

✓ Most of the examples in this chapter will give peculiar results if their input is of a different type than intended. The M-file `absval.m` is designed only for scalar real inputs **x**, not for complex numbers or vectors. If **x** is complex for instance, then **x >= 0** checks only whether the real part of **x** is non-negative, and the output **y** will be complex in either case. MATLAB has a built-in function **abs** that works correctly for vectors of complex numbers.

In general, **if** must be followed on the same line by an expression that MATLAB will test to be true or false; see the section below on *Logical Expressions* for a discussion of available expressions and how they are evaluated. After some intervening commands, there must be (as with **for**) an **end** statement. In between, there may be one or more **elseif** statements (see below) and/or an **else** statement (as above). If the result of the test is true, MATLAB executes all commands between the **if** statement and the first **elseif**, **else**, or **end** statement, then skips all other commands until after the **end** statement. If the result of the test is false, MATLAB skips to the first **elseif**, **else**, or **end** statement and proceeds from there, carrying out a new test in the case of an **elseif** statement. In the example below, we reformulate `absval.m` so that no commands are necessary if the result of the test is false, eliminating the need for an **else** statement.

```
function y = absval(x)
y = x;
if y < 0
    y = -y;
end
```

The **elseif** statement is useful if there are more than two alternatives and they can be distinguished by a sequence of true/false tests. It is essentially equivalent to an **else** statement followed immediately by a nested **if** statement. In the example below, we use **elseif** in an M-file `signum.m`, which evaluates the function

$$\operatorname{sgn}(x) = \begin{cases} 1 & x > 0, \\ 0 & x = 0, \\ -1 & x < 0. \end{cases}$$

(Again, MATLAB has a built-in function **sign** that implements this function for more general inputs than we consider here.)

```
function y = signum(x)
if x > 0
    y = 1;
elseif x == 0
    y = 0;
```

```
else
    y = -1;
end
```

Here if the input **x** is positive, then the output **y** is set to **1** and all commands from the **elseif** statement to the **end** statement are skipped. (In particular, the test in the **elseif** statement is not performed.) If **x** is not positive, then MATLAB skips to the **elseif** statement and tests to see whether **x** equals **0**. If so, **y** is set to **0**, otherwise **y** is set to **-1**. Notice that MATLAB requires a double equals sign **==** to test for equality; a single equals sign is reserved for the assignment of values to variables.

✓ Like **for** and the other programming commands you will encounter, **if** and its associated commands can be used in the Command Window. Doing so can be useful for practice with these commands, but they are intended mainly for use in M-files. In our discussion of branching, we consider primarily the case of function M-files; branching is less often used in script M-files.

Logical Expressions

In the examples above, we used *relational operators* such as **>=**, **>**, and **==** to form a logical expression, and instructed MATLAB to choose between different commands according to whether the expression is true or false. Type **help relop** to see all of the available relational operators. Some of these operators, like **&** (AND) and **|** (OR) can be used to form logical expressions that are more complex than those that simply compare two numbers. For example, the expression **(x > 0) | (y > 0)** will be true if **x** or **y** (or both) is positive, and false if neither is positive. In this particular example, the parentheses are not necessary, but generally compound logical expressions like this are both easier to read and less prone to errors if parentheses are used to avoid ambiguities.

So far in our discussion of branching, we have considered only expressions that can be evaluated as true or false. While such expressions are sufficient for many purposes, you can also follow **if** or **elseif** with any expression that MATLAB can evaluate numerically. In fact, MATLAB makes almost no distinction between logical expressions and ordinary numerical expressions. Consider what happens if you type a logical expression by itself in the Command Window:

```
>> 2 > 3

ans =
   0
```

When evaluating a logical expression, MATLAB assigns it a value of **0** (for FALSE) or **1** (for TRUE). Thus if you type **2 < 3**, the answer is **1**. The relational operators are treated by MATLAB like arithmetic operators, inasmuch as their output is numerical.

✓ MATLAB makes a subtle distinction between the output of relational operators and ordinary numbers. For example, if you type **whos** after the command above, you will see that **ans** is a *logical array*. We will give an example of how this feature can be used shortly. Type **help logical** for more information.

Here is another example:

```
>> 2 | 3

ans =
   1
```

The OR operator | gives the answer **0** if both operands are zero and **1** otherwise. Thus, while the output of relational operators is always **0** or **1**, any non-zero input to operators such as **&** (AND), | (OR), and **~** (NOT) is regarded by MATLAB to be true, while only **0** is regarded to be false.

If the inputs to a relational operator are vectors or matrices rather than scalars, then, as for arithmetic operations such as **+** and **.***, the operation is done term-by-term and the output is an array of zeros and ones. Here are some examples:

```
>> [2 3] < [3 2]

ans =
   1     0

>> x = -2:2; x >= 0

ans =
   0     0     1     1     1
```

In the second case, **x** is compared term-by-term with the scalar **0**. Type **help relop** for more information.

You can use the fact that the output of a relational operator is a logical array to select the elements of an array that meet a certain condition. For example, the expression **x(x >= 0)** yields a vector consisting of only the non-negative elements of **x** (or more precisely, those with non-zero real part). So, if **x = -2:2** as above,

```
>> x(x >= 0)

ans =
   0     1     2
```

If a logical array is used to choose elements from another array, the two arrays must have the same size. The elements corresponding to each **1** in the logical array are selected while the elements corresponding to each **0** are not. In the example above, the result is the same as if we had typed **x(3:5)**, but in this case **3:5** is an ordinary numerical array specifying the numerical indices of the elements to choose.

Next, we discuss how **if** and **elseif** decide whether an expression is true or false. For an expression that evaluates to a scalar real number, the criterion is the same as described above – namely, a non-zero number is treated as true while **0** is treated as false. However, for complex numbers only the real part is considered – thus, in an **if** or **elseif** statement, any number with non-zero real part is treated as true, while numbers with zero real part are treated as false. Furthermore, if the expression

evaluates to a vector or matrix, an **if** or **elseif** statement must still result in a single true-or-false decision. The convention MATLAB uses is that all elements must be true – that is, all elements must have non-zero real part – for an expression to be treated as true. If any element has zero real part, then the expression is treated as false.

You can manipulate the way branching is done with vector input by inverting tests with **~** and using the commands **any** and **all**. For example, the statements **if x == 0; ...; end** will execute a block of commands (represented here by **...**) when all the elements of **x** are zero; if you would like to execute a block of commands when *any* of the elements of **x** is zero you could use the form **if x ~= 0; else; ...; end**. Here **~=** is the relational operator for "does not equal," so the test fails when any element of **x** is zero, and execution skips past the **else** statement. You can achieve the same effect in a more straightforward manner using **any**, which outputs true when any element of an array is non-zero: **if any(x == 0); ...; end** (remember that if any element of **x** is zero, the corresponding element of **x == 0** is non-zero). Likewise **all** outputs true when all elements of an array are non-zero.

Here is a series of examples to illustrate some of the features of logical expressions and branching that we have just described. Suppose that you want to create a function M-file that computes the following function:

$$f(x) = \begin{cases} \sin(x)/x & x \neq 0, \\ 1 & x = 0. \end{cases}$$

You could construct the M-file as follows.

```
function y = f(x)
if x == 0
    y = 1;
else
    y = sin(x)/x;
end
```

This will work fine if the input **x** is a scalar, but not if **x** is a vector or matrix. Of course you could change **/** to **./** in the second definition of **y**, and change the first definition to make **y** the same size as **x**. But, if **x** has both zero and non-zero elements, then MATLAB will declare the **if** statement to be false, and use the second definition. Then some of the entries in the output array **y** will be **NaN**, "not a number," because **0/0** is an indeterminate form.

One way to make this M-file work for vectors and matrices is to use a loop to evaluate the function element-by-element, with an **if** statement inside the loop.

```
function y = f(x)
y = ones(size(x));
for n = 1:prod(size(x))
    if x(n) ~= 0
        y(n) = sin(x(n))/x(n);
    end
end
```

In the M-file above, we first create the eventual output **y** as an array of ones with the same size as the input **x**. Here we use **size(x)** to determine the number of rows

and columns of **x**; recall that MATLAB treats a scalar or a vector as an array with one row and/or one column. Then **prod(size(x))** yields the number of elements in **x**. So in the **for** statement **n** varies from **1** to this number. For each element **x(n)**, we check to see whether it is non-zero, and if so we redefine the corresponding element **y(n)** accordingly. (If **x(n)** equals **0**, there is no need to redefine **y(n)** since we defined it initially to be **1**.)

✓ We just used an important but subtle feature of MATLAB, namely that each element of a matrix can be referred to with a single index; for example if **x** is a 3-by-2 array then its elements can be enumerated as **x(1)**, **x(2)**, ..., **x(6)**. In this way, we avoided using a loop within a loop. Similarly, we could use **length(x(:))** in place of **prod(size(x))** to count the total number of entries in **x**. However, one has to be careful. If we had not predefined **y** to have the same size as **x**, but rather used an **else** statement inside the loop to let **y(n)** be **1** when **x(n)** is **0**, then **y** would have ended up a 1-by-6 array rather than a 3-by-2 array. We then could have used the command **y = reshape(y, size(x))** at the end of the M-file to make **y** have the same shape as **x**. However, even if the shape of the output array is not important, it is generally best to predefine an array of the appropriate size before computing it element-by-element in a loop, because the loop will then run faster.

Next, consider the following modification of the M-file above.

```
function y = f(x)
if x ~= 0
    y = sin(x)./x;
    return
end
y = ones(size(x));
for n = 1:prod(size(x))
    if x(n) ~= 0
        y(n) = sin(x(n))/x(n);
    end
end
```

Above the loop we added a block of four lines whose purpose is to make the M-file run faster if all the elements of the input **x** are non-zero. Because MATLAB processes vectors more efficiently than loops, the new M-file runs several times faster if **x** has a large number of elements (all non-zero). Here is how the new block of four lines works. The first **if** statement will be true provided that all the elements of **x** are non-zero. In this case, we define the output **y** using MATLAB's vector operations, which are generally much more efficient than running a loop. Then we use the command **return** to stop execution of the M-file without running any further commands. (The use of **return** here is a matter of style; we could instead have indented all of the remaining commands and put them between **else** and **end** statements.) If, on the other hand, **x** has some zero elements, then the **if** statement is false and the M-file skips ahead to the commands after the next **end** statement.

Often you can avoid the use of loops and branching commands entirely by using logical arrays. Here is another function M-file that performs the same task as in the previous examples. It has the advantage of being more concise and more efficient to run than the previous M-files, since it avoids a loop in all cases.

```
function y = f(x)
y = ones(size(x));
n = (x ~= 0);
y(n) = sin(x(n))./x(n);
```

Here **n** is a logical array of the same size as **x** with a **1** in each place where **x** has a non-zero element and zeros elsewhere. Thus the line that defines **y(n)** only redefines the elements of **y** corresponding to non-zero values of **x**, and leaves the other elements equal to **1**.

Branching with switch

The other main branching command is **switch**. It allows you to branch among several cases just as easily as among two cases, though the cases must be described through equalities rather than inequalities. Here is a simple example, which distinguishes between three cases for the input.

```
function y = count(x)
switch x
case 1
    y = 'one';
case 2
    y = 'two';
otherwise
    y = 'many';
end
```

Here the **switch** statement evaluates the input **x** and then execution of the M-file skips to whichever **case** statement has the same value. Thus, if the input **x** equals **1**, then the output **y** is set to be the string **'one'**, whereas if **x** is **2**, then **y** is set to **'two'**. In each case, once MATLAB encounters another **case** statement or an **otherwise** statement, it skips to the **end** statement, so that at most one case is executed. If no match is found among the **case** statements, then MATLAB skips to the (optional) **otherwise** statement, or else to the **end** statement. In the example above, an **otherwise** statement is present, so the output is **'many'** if the input is not **1** or **2**.

Unlike **if**, the command **switch** does not allow vector expressions, but it does allow strings. Type **help switch** to see an example using strings. This feature can be useful if you want to design a function M-file that uses a string input argument to select among several different variants of a program you write.

✓ Though strings cannot be compared with relational operators such as **==** (unless they happen to have the same length), you can compare strings in an **if** or **elseif** statement by using either **strcmp** or **isequal**. The latter command works more generally when comparing arrays that may have different sizes or even different data types.

More about Loops

In Chapter 3 we introduced the command **for**, which begins a loop – a sequence of commands to be executed multiple times. When you use **for**, you effectively specify the number of times to run the loop in advance (though this number may depend for instance on the input to a function M-file). Sometimes you may want to keep running the commands in a loop until a certain condition is met, without deciding in advance on the number of iterations. In MATLAB, the command that allows you to do so is **while**.

⇨ **Using while, one can easily end up accidentally creating an "infinite loop," one that will keep running indefinitely because the condition you set is never met. Remember that you can generally interrupt the execution of such a loop by typing CTRL+C; otherwise, you may have to shut down MATLAB.**

Open-Ended Loops

Here is a simple example of a script M-file that uses **while** to numerically sum the infinite series $1/1^4 + 1/2^4 + 1/3^4 + \cdots$, stopping only when the terms become so small (compared with the machine precision) that the numerical sum stops changing.

```
n = 1;
oldsum = -1;
newsum = 0;
while newsum > oldsum
    oldsum = newsum;
    newsum = newsum + n^(-4);
    n = n + 1;
end
newsum
```

Here we initialize **newsum** to **0** and **n** to **1**, and in the loop we successively add **n^(-4)** to **newsum**, add **1** to **n**, and repeat. The purpose of the variable **oldsum** is to keep track of how much **newsum** changes from one iteration to the next. Each time MATLAB reaches the end of the loop, it starts over again at the **while** statement. If **newsum** exceeds **oldsum**, the expression in the **while** statement is true, and the loop is executed again. But the first time the expression is false, which will happen when **newsum** and **oldsum** are equal, MATLAB skips to the **end** statement and executes the next line, which displays the final value of **newsum** (the result is **1.0823** to 5 significant digits). The initial value of **-1** that we gave to **oldsum** is somewhat

arbitrary, but it must be negative so that the first time the **`while`** statement is executed, the expression therein is true; if we set **`oldsum`** to **`0`** initially, then MATLAB would skip to the **`end`** statement without ever running the commands in the loop.

✓ Even though you can construct an M-file like the one above without deciding exactly how many times to run the loop, it may be useful to consider roughly how many times it will need to run. Since the floating-point computations on most computers are accurate to about 16 decimal digits, the loop above should run until **`n^(-4)`** is about **`10^(-16)`**; that is, until **`n`** is about **`10^4`**. Thus the computation will take very little time on most computers. However, if the exponent were **`2`**, not **`4`**, the computation would take about **`10^8`** operations, which would take a long time on most (current) computers – long enough to make it wiser for you to find a more efficient way to sum the series, for example using **`symsum`** if you have the Symbolic Math Toolbox!

☞ *Though we have classified it here as a looping command,* **`while`** *also has features of a branching command. Indeed, the types of expressions allowed and the method of evaluation for a* **`while`** *statement are exactly the same as for an* **`if`** *statement. See the section* Logical Expressions *above for a discussion of the possible expressions you can put in a* **`while`** *statement.*

Breaking from a Loop

Sometimes you may want MATLAB to jump out of a **`for`** loop prematurely, for example if a certain condition is met. Or, in a **`while`** loop, there may be an auxiliary condition that you want to check in addition to the main condition in the **`while`** statement. Inside either type of loop, you can use the command **`break`** to tell MATLAB to stop running the loop and skip to the next line after the end of the loop. The command **`break`** is generally used in conjunction with an **`if`** statement. The following script M-file computes the same sum as in the previous example, except that it places an explicit upper limit on the number of iterations.

```
newsum = 0;
for n = 1:100000
    oldsum = newsum;
    newsum = newsum + n^(-4);
    if newsum == oldsum
        break
    end
end
newsum
```

In this example, the loop stops after **`n`** reaches **`100000`** or when the variable **`newsum`** stops changing, whichever comes first. Notice that **`break`** ignores the **`end`** statement associated with **`if`** and skips ahead past the nearest **`end`** statement associated with a loop command, in this case **`for`**.

Other Programming Commands

Here we describe a few more advanced programming commands and techniques.

Subfunctions

In addition to appearing on the first line of a function M-file, the command **function** can be used later in the M-file to define an auxiliary function, or *subfunction*, which can be used anywhere within the M-file but will not be accessible directly from the command line. For example, the following M-file sums the cube roots of a vector **x** of real numbers:

```
function y = sumcuberoots(x)
y = sum(cuberoot(x));

% ---- Subfunction starts here.
function z = cuberoot(x)
z = sign(x).*abs(x).^(1/3);
```

Here the subfunction **cuberoot** takes the cube root of **x** element-by-element. You can use subfunctions only in a function M-file, not in a script M-file. For examples of the use of subfunctions, you can examine many of MATLAB's built-in function M-files. For example, typing **type ezplot** will display three different subfunctions.

Cell and Structure Arrays

In Chapter 4, we discussed several data types, and earlier in this chapter we introduced another type, namely logical arrays. Two other data types that are useful in MATLAB programming are *cell arrays* and *structure arrays*. Cell arrays are essentially "arrays of arrays"; the elements of a cell array can have different data types and different sizes. The command **cell** creates an empty cell array, but the more common way to create a cell array is with curly braces:

```
>> ca = {1, [2 3], 'four'}

ca =
   [1]    [1x2 double]    'four'
```

Use curly braces also to access a particular element of a cell array; for example, type **ca{2}** to display the second element of **ca**.

Structure arrays are similar to structures in programming languages like C; they allow you to name the elements of the array rather than number them. Like cell arrays, the elements can have different types and sizes. One way to create a structure array is with the command **struct**:

```
>> sa = struct('data', [1 4 9 16 25], 'description', ...
'perfect squares')

sa =
          data: [1 4 9 16 25]
   description: 'perfect squares'
```

To access a particular element, or "field," type the name of the array and the name of the field with a period in between; for example, **sa.data**. You can also use the same syntax to create and add fields to a structure array. Another way to define **sa** is with the following commands:

```
>> sa = struct;
>> sa.data = [1 4 9 16 25];
>> sa.description = 'perfect squares'
```

The first command defines an empty structure, and the subsequent commands add fields to it.

✓ MATLAB also has signed and unsigned integer data types; see the online help for **int32** and **uint32**, for example.

Commands for Parsing Input and Output

You may have noticed that many MATLAB functions allow you to vary the type and/or the number of arguments you give as input to the function. You can use the commands **nargin**, **nargout**, **varargin**, and **varargout** in your own M-files to handle variable numbers of input and/or output arguments, whereas to treat different types of input arguments differently you can use commands such as **isnumeric**, **ischar**, etc.

When a function M-file is executed, the functions **nargin** and **nargout** report, respectively, the number of input and output arguments that were specified on the command line. To illustrate the use of **nargin**, consider the following M-file `add.m` that adds either two or three inputs.

```
function s = add(x, y, z)
if nargin < 2
    error('At least two input arguments are required.')
end
if nargin == 2
    s = x + y;
else
    s = x + y + z;
end
```

First the M-file checks to see whether fewer than two input arguments were given, and if so it prints an error message and quits. (See the next section for more about **error** and related commands.) MATLAB automatically checks to see whether there are more arguments than specified on the first line of the M-file, so there is no need to do so within the M-file. If the M-file reaches the second **if** statement in the M-file above, we know that there are either two or three input arguments; the **if** statement selects the proper course of action in either case. If you type, for instance, **add(4,5)** at the command line, then, within the M-file, **x** is set to **4**, **y** is set to **5**, and **z** is left undefined; thus it is important to use **nargin** to avoid referring to **z** in cases where it is undefined.

To allow a greater number of possible inputs to `add.m`, we could add additional arguments on the first line of the M-file and add more cases for **nargin**. A better way to do this is to use the specially named input argument **varargin**.

```
function s = add(varargin)
s = sum([varargin{:}]);
```

In this example, all of the input arguments are assigned to the *cell array* **varargin**. The expression **varargin{:}** forms a comma-separated list of the input arguments. In the example above, we convert this list to a vector by enclosing it in square brackets, forming suitable input for **sum**.

The sample M-files above assume that their input arguments are numerical, and will attempt to add them even if they are not. This may be desirable in some cases; for instance, both M-files above will correctly add a mixture of numerical and symbolic inputs. However, if some of the input arguments are strings, the result will be either an essentially meaningless numerical answer or an error message that may be difficult to decipher. MATLAB has a number of test functions that you can use to make an M-file treat different types of input arguments differently – either to perform different calculations, or to produce a helpful error message if an input is of an unexpected type. For a list of some of these test functions, look up the commands beginning with **is** in the *Programming Commands* section of the *Glossary*.

As an example, here we use **isnumeric** in the M-file `add.m` to print an error message if any of the inputs are not numerical.

```
function s = add(varargin)
if ~isnumeric([varargin{:}])
    error('Inputs must be floating point numbers.')
end
s = sum([varargin{:}]);
```

When a function M-file allows multiple output arguments, then, if fewer output arguments are specified when the function is called, the remaining outputs are simply not assigned. Recall that if no output arguments are explicitly specified on the command line, then a single output is returned and assigned to the variable **ans**. For example, consider the following M-file `rectangular.m` that changes coordinates from polar to rectangular.

```
function [x, y] = rectangular(r, theta)
x = r.*cos(theta);
y = r.*sin(theta);
```

Typing **[x, y] = rectangular(2, 1)** at the command line then stores the rectangular coordinates of the point with polar coordinates $(2, 1)$ in the variables **x** and **y**. But if you type only **rectangular(2, 1)**, then the answer will be just the x-coordinate. The following modification to **rectangular.m** adjusts the output in this case to be a complex number $x + iy$ containing both coordinates.

```
function [x, y] = rectangular(r, theta)
x = r.*cos(theta);
y = r.*sin(theta);
if nargout < 2
```

```
    x = x + i*y;
end
```

See the online help for **varargout** and the functions described above for additional information and examples.

Evaluation and Function Handles

The command **eval** allows you to run a command that is stored in a string as if you had typed the string on the command line. For example, typing **eval('cos(1)')** will produce the same result as typing **cos(1)**. If the entire command you want to run is contained in a string **str**, then you can execute it with **eval(str)**. Generally **eval** is used in an M-file to define a variable or run a command whose name depends on an input or loop variable; type **help eval** for examples.

Another useful feature of **eval** is that, with two input strings, it will try the first command and, if that produces an error, try the second command instead. This form of **eval** allows some flexibility in the number of inputs to an anonymous function. The command

```
>> add = @(x,y,z) eval('x + y + z', 'x + y')
```

creates an anonymous function that adds either two or three input arguments, like `add.m` above.

✓ You can also create anonymous functions with multiple outputs using the command **deal**. For example, typing

```
>> rectangular = @(r,theta) deal(r.*cos(theta), ...
r.*sin(theta))
```

yields a function that converts a pair of polar coordinates into a pair of rectangular coordinates, like `rectangular.m` above. The only difference is that this function will yield an error message if you specify only one output argument. To allow for either one or two output arguments, you can use the two-input form of **eval** as in the example above; we leave the details to you as an exercise.

In Chapter 4 we mentioned function handles, which were introduced in MATLAB 6. Function handles are now the preferred way to specify a function as input to another function (like **fzero** or **quadl**, for example), though for backward compatibility many such functions also allow you to input a function name as a string. Starting in MATLAB 7, function handles also make it simple for you to write function M-files that input another function. For example, the following M-file `iterate.m` inputs a function handle (or an inline function) and an initial value and iterates the function a specified number of times.

```
function final = iterate(func, init, num)
final = init;
for k = 1:num
    final = func(final);
end
```

Recall that putting the "at sign" **@** before the name of a built-in function or function M-file makes a handle for that function. Thus, typing **iterate(@cos, 1, 2)** yields the numerical value of $\cos(\cos(1))$, while **iterate(@cos, 1, 100)** yields an approximation to the real number x for which $\cos(x) = x$. (Think about it!) Remember also that an anonymous function is a function handle. So, typing **iterate(@(x) 3.9*x*(1 - x), 0.5, 100)** iterates the "logistic map" $f(x) = 3.9x(1 - x)$ one hundred times starting at $x = 0.5$.

☞ *See* Population Dynamics *in the* Applications *chapter for more about the logistic map.*

✓ In MATLAB 6 and earlier versions, one must use the command **feval** in function M-files like `iterate.m`. Specifically, replace the command **final = func(final)** in the M-file with **final = feval(func, final)**. When using **feval**, the input function can be specified as a string, as well as a function handle or inline function. For example, typing either **feval('atan2', 1, 0)** or **feval(@atan2, 1, 0)** is equivalent to typing **atan2(1, 0)**.

Another feature of anonymous functions that is useful in programming is that their definition can include a parameter stored in a variable. (This is not the case for inline functions.) Consider the following commands:

```
>> c = 3; f = @(x) c*x; f(2)

ans =
   6
```

The value of **c** at the time **f** was defined is incorporated into **f**, and any subsequent changes to **c** do not affect **f**:

```
>> c = 5; f(2)

ans =
   6
```

This feature can be used in an M-file to define an anonymous function that depends on an input parameter. Another use is to vary a parameter in a loop. For example the loop

```
>> for c = 1:3
    ezplot(@(x) sin(c*x), [0 2*pi])
end
```

will reproduce the curves in Figure 3.1.

User Input and Screen Output

In the previous section we used **error** to print a message to the screen and then terminate execution of an M-file. You can also print messages to the screen without stopping execution of an M-file by using **disp** or **warning**. Not surprisingly, **warning** is intended to be used for warning messages, when the M-file detects a

problem that might affect the validity of its result but is not necessarily serious. You can suppress warning messages, either from the command prompt or within an M-file, with the command **warning off**. There are several other options for handling warning messages; type **help warning** for details.

In Chapter 4 we used **disp** to display the output of a command without printing the "ans =" line. You can also use **disp** to display informational messages on the screen while an M-file is running, or to combine numerical output with a message on the same line. For example, the commands

```
x = 2 + 2; disp(['The answer is ' num2str(x) '.'])
```

will set **x** equal to **4** and then print The answer is 4.

MATLAB also has several commands that solicit input from the user running an M-file. At the end of Chapter 3 we discussed three of them: **pause**, **keyboard**, and **input**. Briefly, **pause** simply pauses execution of an M-file until the user hits a key, while **keyboard** both pauses and gives the user a prompt that can be used like the regular command-line interface before typing **return** to continue executing the M-file. Lastly, **input** displays a message and allows the user to enter input for the program on a single line. For example, in a program that makes successive approximations to an answer until some accuracy goal is met, you could add the following lines to be executed after a large number of steps have been taken.

```
answer = input(['Algorithm is converging slowly; ', ...
    'continue (yes/no)? '], 's');
if ~isequal(answer, 'yes')
    return
end
```

Here the second argument **'s'** to **input** directs MATLAB not to evaluate the answer typed by the user, just to assign it as a character string to the variable **answer**. We use **isequal** to compare the answer to the string **'yes'** because **==** can be used only to compare arrays (in this case strings) of the same length. In this case we decided that if the user types anything but the full word **yes**, the M-file should terminate. Other approaches would be to compare only the first letter **answer(1)** with **'y'**, to stop only if the answer is **'no'**, etc.

If a figure window is open, you can use **ginput** to get the coordinates of a point that the user selects with the mouse. As an example, the following M-file prints an "X" where the user clicks.

```
function xmarksthespot
if isempty(get(0, 'CurrentFigure'))
    error('No current figure.')
end
flag = ~ishold;
if flag
    hold on
end
disp('Click on the point where you want to plot an X.')
[x, y] = ginput(1);
plot(x, y, 'xk')
```

```
if flag
    hold off
end
```

First the M-file checks to see whether there is a current figure window. If so, it proceeds to set the variable **flag** to **1** if **hold off** is in effect and **0** if **hold on** is in effect. The reason for this is that we need to have **hold on** in effect to plot an "X" without erasing the figure, but afterward we want to restore the figure window to whichever state it was in before the M-file was executed. The M-file then displays a message telling the user what to do, gets the coordinates of the point selected with **ginput(1)**, and plots a black "X" at those coordinates. The argument **1** to **ginput** means to get the coordinates of a single point; using **ginput** with no input argument would collect coordinates of several points, stopping only when the user presses the ENTER key.

☞ *In Chapter 9 we describe how to create a Graphical User Interface (GUI) within MATLAB to allow more sophisticated user interaction.*

Debugging

In Chapter 3 we discussed some rudimentary debugging procedures. One suggestion was to insert the command **keyboard** into an M-file, for instance right before the line where an error occurs, so that you can examine the Workspace of the M-file at that point in its execution. A more effective and flexible way to do this kind of debugging is to use **dbstop** and related commands. With **dbstop** you can set a *breakpoint* in an M-file in a number of ways; for example, at a specific line number, or whenever an error occurs. Type **help dbstop** for a list of available options.

When a breakpoint is reached, a prompt beginning with the letter **K** will appear in the Command Window, just as if **keyboard** were inserted into the M-file at the breakpoint. In addition, the location of the breakpoint is highlighted with an arrow in the Editor/Debugger (which is opened automatically if you were not already editing the M-file). At this point you can examine in the Command Window the variables used in the M-file, set another breakpoint with **dbstop**, clear breakpoints with **dbclear**, etc. If you are ready to continue running the M-file, type **dbcont** to continue or **dbstep** to step through the file line-by-line. You can also stop execution of the M-file and return immediately to the usual command prompt with **dbquit**.

☞ *You can also perform all the command-line functions that we described in this section with the mouse and/or keyboard shortcuts in the Editor/Debugger. See the section* Debugging Techniques *in Chapter 11 for more about debugging commands and features of the Editor/Debugger.*

☆ Interacting with the Operating System

☞ *This section is somewhat advanced. On a first reading, you might want to skip ahead to the next chapter.*

Calling External Programs

MATLAB allows you to run other programs on your computer from its command line. For example, you can conveniently enter UNIX or DOS file-manipulation commands directly in the Command Window rather than opening a separate window. Or, you may want to use MATLAB to graph the output of a program written in a language like Fortran or C. Finally, for large-scale computations, you can combine routines written in another programming language with routines you write in MATLAB.

The simplest way to run an external program is to type an exclamation point at the beginning of a line, followed by the operating-system command you want to run. For example, typing **`!dir`** on a Windows system or **`!ls -l`** on a UNIX system will generate a more detailed listing of the files in the current working directory than the MATLAB command **`dir`**. In Chapter 3 we described **`dir`** and other MATLAB commands like **`cd`**, **`delete`**, **`pwd`**, and **`type`** that mimic similar commands from the UNIX or DOS prompt. However, for certain operations (such as renaming a file) you may need to run an appropriate command from the operating system.

✓ If you use the operating-system interface in an M-file that you want to run on either a Windows or UNIX system, you should use the test functions **`ispc`** and/or **`isunix`** to set off the appropriate commands for each type of system, e.g., **`if isunix; ...; else; ...; end`**. If you need to distinguish between different versions of UNIX (Linux, Solaris, etc.), you can use **`computer`** instead of **`isunix`**.

The output from an operating-system command preceded by **`!`** can only be displayed to the screen. To assign the output of an operating-system command to a variable, you must use **`dos`** or **`unix`**. Though each is documented to work only for its respective operating system, in current versions of MATLAB they work interchangeably. For example, if you type **`[stat, data] = dos('myprog 0.5 1000')`**, the program **`myprog`** will be run with command-line arguments **`0.5`** and **`1000`** and its "standard output" (which would normally appear on the screen) will be saved as a string in the variable **`data`**. (The variable **`stat`** will contain the exit status of the program you run, normally **`0`** if the program runs without error.) If the output of your program consists only of numbers, then **`str2num(data)`** will yield a row vector containing those numbers. You can also use **`sscanf`** to extract numbers from the string **`data`**; type **`help sscanf`** for details.

⇨ **A program you run with `!`, `dos`, or `unix` must be in the current directory or elsewhere in the path your system searches for executable files; the MATLAB path will not be searched.**

✓ MATLAB also allows for more sophisticated interfaces with C, Fortran, and Java programs, but these are beyond the scope of this book. See the section on External Interfaces in the Help Browser for details.

File Input and Output

In Chapter 3 we discussed how to use **`save`** and **`load`** to transfer variables between the Workspace and a disk file. By default the variables are written and read in MATLAB's own binary format, which is signified by the file extension `.mat`.

⇨ **MATLAB 7 uses a new binary format that cannot be read by earlier versions of MATLAB. To save in a backward-compatible format, type `-v6` after the file name in the `save` command.**

You can also read and write text files, which can be useful for sharing data with other programs. With **`save`**, type **`-ascii`** after the file name to save numbers as text rounded to 8 digits, or **`-ascii -double`** for 16-digit accuracy. With **`load`** the data is assumed to be in text format if the file name does not end in `.mat`. This provides an alternative to importing data with **`dos`** or **`unix`** in case you have previously run an external program and saved the results in a file.

✓ Beginning with version 6, MATLAB 6 also offers an interactive tool called the *Import Wizard* to read data from files in different formats; to start it type **`uiimport`** (optionally followed by a file name), select **File**:**Import Data...**, or right-click on the file name in the Current Directory Browser and select **Import Data...**. Using **`uiimport`** also allows you to import data that you have cut or copied to the system clipboard.

For more control over file input and output – for example to annotate numerical output with text – you can use **`fopen`**, **`fprintf`**, and related commands. Type **`help iofun`** for an overview of input and output functions.

Chapter 7

Publishing and M-Books

MATLAB is exceptionally strong in linear algebra, numerical methods, and graphical interpretation of data. It is easily programmed and relatively easy to learn to use. Hence, it has proven invaluable to engineers and scientists who rely on the scientific techniques and methods at which MATLAB excels. Very often the individuals and groups that so employ MATLAB are primarily interested in the numbers and graphs that emerge from MATLAB commands, processes and programs. Therefore, it is enough for them to work in a MATLAB Command Window, from which they can easily print or export their desired output.

However, other practitioners of mathematical software find themselves with two additional requirements. First, they need a mathematical software package embedded in an interactive environment, in which it is easy to make changes and regenerate results. Second, they need a higher-level presentation mode, which integrates computation and graphics with text, uses different formats for input and output, and communicates effortlessly with other software applications. These additional requirements can be accomplished using either cells and the **`publish`** command, or else the *M-book* interface, both of which were briefly described in Chapter 3. The present chapter goes into more detail and discusses some of the fine points of these methods.

Fine Points of Publishing

As we mentioned Chapter 3, the simplest way to produce a finished presentation with MATLAB is to prepare your work in a script M-file and then **`publish`** the result. To do this effectively, you should first make sure that you have "enabled cell mode" in the Editor/Debugger, so that the cell structure of your M-file is readily visible. If you are going to post your results on a web page, or even just print them out for viewing, publishing to `HTML` (which stands for "hypertext markup language," the formatting language for web pages) works best. MATLAB has a built-in web browser (which can be accessed directly with the command **`web`**) that will display your published M-file as soon as MATLAB has finishing compiling it. You may then either print the result directly from the browser window or else copy the published M-file to your web server. Note that MATLAB stores the `html` code of the published M-file, along with all the graphics produced by the M-file (usually in `png` format), in a folder named `html`, so, if you copy the published file, be sure to copy all the graphics files that accompany it. If you are using MATLAB to produce slides for a presentation, it is better to **`publish`** to a PowerPoint file (with the file extension `.ppt`) instead.

To produce a nicely formatted published M-file, recall from the section *Publish-*

ing an M-file in Chapter 3 that all the output from a single cell will appear together, following the MATLAB commands from that cell. Thus, if you want to intersperse comments and calculations, you need to break your work up into multiple cells. Comments will be formatted only if they appear at the *beginning* of a cell, so you need to arrange your work in the following order: new cell (i.e., **%%** at the beginning of a line, whether or not followed by space and a cell title), then comments, then MATLAB code. As a simple example, the code that produced the published solution to Problem 5(e) in Practice Set B reads as follows:

```
%% (e)
limit(sin(1/x), x, 0, 'right')

%%
% This means that every real number in the interval
% between -1 and +1 is a "limit point" of sin(1/x)
% as x tends to zero.  You can see why if you plot
% sin(1/x) on the interval (0, 1].
ezplot(sin(1/x), [0 1])
```

Sometimes you may need special formatting beyond what appears in the above example. For example, you might want your published M-file to include an equation involving mathematical symbols that are not easy to imitate in regular text. In this case, you can typeset the equation in TEX and insert it using **Cell**:**Insert Text Markup**:**TeX Equation** from the menu items at the top of the Editor/Debugger. If you are not familiar with TEX, a few things to keep in mind are that you create Greek letters by typing the name of the letter preceded by a backslash (\), and create subscripts and superscripts with underscores (_) and carets (^). Grouping is done with braces (**{ }**). Mathematical symbols have names (usually self-explanatory) beginning with a backslash (\), such as **\times** for a multiplication sign and **\sum** for a summation sign. Here is a simple example of an M-file `aboutbessel.m`, containing formatted text, suitable for publishing:

```
%% Sample M-File About Bessel Functions
% The *Bessel function* J_n(x) can be defined in many ways:
%
% * For example, for |n| an integer, J_n(x) can be defined
% by the formal series expansion
%
% $$ e^{z(t-1/t)/2} = \sum_{n=-\infty}^\infty t^n J_n(z). $$
%
% * Secondly, J_n(x) can be defined as the solution of
%
% $$ x^2y''(x) + xy'(x) + (x^2 - n^2)y(x)=0 $$
%
% that is non-singular at x = 0 and satisfies a
% normalization condition.
%%
% Here is a graph of some of the Bessel functions:
x = 0:0.05:10;
for n = 0:5
```

```
    plot(x, besselj(n, x)), hold on
end, hold off
```

In this example, the lines beginning with `% *` produce a bulleted list, the vertical bars around the letter `n` in the third line are an instruction to MATLAB to set this word in monospaced, rather than Roman, type, and the asterisks around the phrase "Bessel function" are an instruction to set these words in boldface. If you publish this example to LaTeX, the TeX code for the equations will be incorporated verbatim. If you publish to `html`, MATLAB typesets each equation in TeX, stores the result as a graphics file, and then pastes the graphics file into the web page.

As we have hinted before, the command **`publish`** works only with script M-files, not with function M-files. If you are doing a calculation that requires a function M-file and you want to publish it, you can call the function M-file from a publishable script M-file, and include in the latter the line

```
type <function M-file>
```

where **`<function M-file>`** should be replaced with the name of the M-file. This will ensure that the text of the function M-file appears in your published output. You can see examples of this trick in the published M-file on *Numerical Solution of the Heat Equation* in Chapter 10.

One problem you may encounter occurs when you publish an M-file that has a mistake in it, or at least that contains code that produces a MATLAB error message. (For an example, see the solution in *Solutions to the Practice Sets* to Problem 6(c) in Practice Set B.) You will discover that, ordinarily, **`publish`** terminates as soon as it encounters an error, and does not evaluate the rest of the M-file. A way around this is to **`publish`** with "options," by creating a *structure* with the options you require. (See *Cell and Structure Arrays* in Chapter 6.) For example, our published solution to the M-file for Practice Set B was produced with the following MATLAB code:

```
>> options = struct;
>> options.format = 'latex';
>> options.stopOnError = logical(false);
>> publish('ExBsols', options)
```

Another fine point is that you may find that the default size for graphics or text in a published M-file is too small or too large for your purposes. You can change the default picture size by going (in the Desktop toolbar) to **File**:**Preferences...**:**Editor/Debugger**:**Publishing Images** and can change the default style sheet by going to **File**:**Preferences...**:**Editor/Debugger**:**Publishing**.

We conclude this section with one more fine point about what happens when you publish to LaTeX. If you do this, the output LaTeX code and graphics files (this time in `eps` format) are stored in your `html` folder. You can immediately run `latex` on the `tex` file, or else you can paste it into another LaTeX file. If you do the latter, you have to remember to remove the lines

```
\documentclass{article}
\begin{document}
```

at the beginning and

```
\end{document}
```

at the end, and you should move the lines

```
\usepackage{graphicx}
\usepackage{color}
```

to the preamble of the document into which you are pasting the code (unless they are already present there, in which case you should delete the extra copies).

More on M-Books

The M-book interface allows a Windows user to operate MATLAB from a special Microsoft Word document instead of from a MATLAB Command Window. In this mode, the user should think of Word as running in the foreground and MATLAB as running in the background as an *Automation Server*. Lines that you enter into your Word document are passed to the MATLAB server in the background and executed there, whereupon the output is returned to Word, and then both input and output are automatically formatted. One obtains a living document in the sense that one can edit the document as one normally edits a word-processing document. So one can revisit input lines that need adjustment, change them, and re-execute on the spot – after which the old outdated output is automatically overwritten with new output. The graphical output that results from MATLAB graphics commands appears in the Word document, immediately after the commands that generated it. Erroneous input and output are easily expunged, enhanced formatting can be done in a way that is no more complicated than what one does in a word processor, and in the end the result of your MATLAB session can be an attractive, easily readable, and highly informative document. Of course, one can "cheat" by editing one's output – we shall discuss that and other pitfalls and strengths of M-books in what follows.

☞ *We are assuming here that M-books have been enabled on your computer. If not, or if you get an error message about a missing file* `m-book.dot`*, go back to the section on* M-Books *in Chapter 3 for instructions on how to fix this problem.*

The most common way to start up the M-book interface is to type **`notebook`** at the Command Window prompt, or, if you want to open an existing M-book, to type

```
>> notebook <M-book>
```

with **`<M-book>`** replaced by the name of the M-book, ordinarily ending in `.doc`.

✓ An alternate, and on some systems (especially networked systems) a preferable, launch method is first to open a previously saved M-book – either directly through **File**:**Open...** in Word, or by double-clicking on the file name in Windows Explorer. Word recognizes that the document is an M-book and it automatically launches MATLAB if it is not already running. A word of caution: if you have more than one version of MATLAB installed, Word will launch the version you installed last. To override this, you can open the MATLAB version you want *before* you open the M-book.

You can now type into the M-book in the usual way. The MATLAB automation server is activated only if you do one of two things: either access the items in the **Notebook** menu, or press the key combination CTRL+ENTER. You may note that your commands take a little longer to evaluate than they would inside a normal MATLAB Command Window. This is not surprising considering the amount of information that is passing back and forth between MATLAB and Word.

✓ If you want to start a fresh M-book, click on **File**:**New M-book** in the Menu Bar, or **File**:**New...**, then click on **m-book.dot**.

✓ Note that the **Help** item on the menu bar is Word help, not MATLAB help. If you want to invoke MATLAB help, then either type **`help`** or **`doc`** (with CTRL+ENTER of course), or bring the MATLAB Command Window to the foreground (see below) and use MATLAB help in the usual fashion.

The Notebook Menu Items

Next let's examine the items in the **Notebook** menu. First comes **Define Input Cell**. If you put your cursor on any line and select **Define Input Cell**, then that line will become an input line. To evaluate it, you must press CTRL+ENTER or choose **Evaluate Cell** from the **Notebook** menu. The advantage to this method is that you can create an input cell containing more than one line. For example, type

```
syms x y
factor(x^2 - y^2)
```

then select both lines (by clicking and dragging over them) and choose **Define Input Cell**. Pressing CTRL+ENTER will then cause *both* lines to be evaluated. You can recognize that both lines are incorporated into one input cell by looking at the brackets, or *Cell Markers*. The menu item **Hide Cell Markers** will cause the Cell Markers to disappear; in fact that menu item is a toggle switch that turns the Markers on and off. If you have several input cells, you can convert them into one input cell by selecting them and choosing **Group Cells**. You can break them apart by choosing **Ungroup Cells**. If you click in an input cell and choose **Undefine Cells**, that cell ceases to be an input cell; its formatting reverts to the default Word format, as does the corresponding output cell. If you "undefine" an output cell, it loses its format, but the corresponding input cell remains unchanged.

If you select some portion of your M-book (e.g., the entire M-book by using **Edit**:**Select All**) and then choose **Purge Selected Output Cells**, all output cells in the selection will be deleted. This is particularly useful if you wish to change some data on which the output in your selection depends, and then re-evaluate the entire selection by choosing **Evaluate Cell**. You can re-evaluate the entire M-book at any time by choosing **Evaluate M-book**. If your M-book contains a loop, you can evaluate it by selecting it and choosing **Evaluate Loop**, or for that matter **Evaluate Cell**, provided that the entire loop is inside a single input cell.

It is often handy to purge all output from an M-book before saving, to economize on storage space or on time upon reopening, especially if there are complicated graphs

in the document. If there are any input cells that you want to automatically evaluate upon opening of the M-book, select them and click on **Define AutoInit Cell**. The color of the text in those cells will change from green to navy blue. If you want to separate out a series of commands, say for repeated evaluation, then select the cells and click on **Define Calc Zone**. The commands selected will be encased in a Word section (with section breaks before and after it). If you click in the section and select **Evaluate Calc Zone** from the **Notebook** menu, the commands in only that zone will be (re)evaluated.

The last two buttons are also useful. The button **Bring MATLAB to Front** does exactly that; it reveals the MATLAB Command Window that has been hiding behind the M-book. You may want to enter some commands directly into the Command Window and not have them in your M-book – for example, a `help` entry. Finally, the last button, **Notebook Options...** brings up a panel in which you can do some customization of your M-book: set the numerical format, establish the size of graphics figures, etc.

M-Book Graphics

All MATLAB commands that generate graphics work in M-books. The figure produced by a graphics command appears immediately below that command. However, if you refine your graphics in an M-book, you will find that it is more desirable to modify the input cells that generated them, rather than producing more pictures by repeating the command with new options. So when adding things like `xlabel`, `ylabel`, `legend`, `title`, etc., it is usually best just to add them to the graphics input cell and re-evaluate.

In most cases, it is desirable to have figures appear directly in the M-book, using the default option **Notebook**:**Notebook Options...**:**Embed Figures in M-Book**, and you might not care whether they do or do not appear in a separate figure window. But sometimes you might want to use a figure window along with an M-book, for example to rotate a plot with the mouse. If you type `figure` from the Command Window to open a figure window, then subsequent graphics from the M-book will appear *simultaneously* in the figure window and in the M-book itself.

Finally, we note the button **Toggle Graph Output for Cell**, the only button on the **Notebook** menu not previously described. If you select a cell containing a graphics command and click on this button, no graphical output will result from the evaluation of this command. This can be useful when used in conjunction with `hold on` if you want to produce a single graphic using multiple command lines.

More Hints for Effective Use of M-Books

You may want to run script or function M-files in an M-book. You must still make sure that these M-files are located in your MATLAB path, just as you would in a Command Window. But, assuming that this is the case, M-files are executed in an M-book exactly as in a Command Window. You invoke them simply by typing their names and pressing CTRL+ENTER. The outputs they generate, both intermediate

and final, are determined as before. In particular, semicolons at ends of lines are important. One thing that does not work so well is the command **`more`**. A command as simple as

```
more on; help print
```

may confuse Word and cause it to hang. Thus you may want to bring MATLAB to the foreground and enter your **`help`** requests in the Command Window.

Another standard MATLAB feature that does not work so well in M-books is the **`...`** construct for continuing a long command entry on a second line. Word automatically converts three dots into a single special *ellipsis* character and so confuses MATLAB. There are two ways around this difficulty. Either do not use ellipses; rather simply continue typing and allow Word to wrap as usual – the command will be interpreted properly when passed to MATLAB. Or, turn off the "Auto Correct" feature of Word that converts the three dots into an ellipsis. This is most easily done by typing CTRL+Z after the three dots. Alternatively, open **Tools**:**Auto Correct...** and change the settings that appear there.

The ellipsis difficulty described in the last section is just one of many of its type. The various kinds of automatic formatting that Word carries out can truly confuse MATLAB. Several such instances that we find particularly annoying are as follows: fractions (1/2 is converted to a single character $^1/_2$ representing one-half); the character combination ":)", a construct often used when specifying the rows of a matrix, which Word converts to a "smiley face" ☺; and various dashes that wreak havoc with MATLAB's attempts to interpret an ordinary hyphen as a minus sign. Examine these in **Tools**:**Auto Correct...** and, if you use M-books regularly, consider turning them off.

A more insidious problem is the following. If you cut and paste character strings into an input cell, the characters in the original font may be converted into something you don't anticipate in the Courier input cell. Mysterious and unfathomable error messages upon execution are a tip-off to this problem. In general, you should not copy cells for evaluation unless it is from a cell that has already evaluated successfully – it is safer to type in the line anew.

Finally, we have seen instances in which a cell, for no discernible reason, fails to evaluate. If this happens, try pressing CTRL+ENTER again. If all else fails, your only choice may be to delete and retype the cell.

Chapter 8

Simulink

In this chapter we describe Simulink, a MATLAB accessory for simulating dynamical processes and engineering systems. This brief introduction, together with the online documentation, should be enough to get you started using Simulink.

A Simple Differential Equation

If you want to learn about Simulink in depth, you can read the massive PDF document available at

```
www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf
```

Here we give a brief introduction for the user who wants to get going with Simulink quickly. You start Simulink by double-clicking on Simulink in the Launch Pad, by clicking on the Simulink button on the MATLAB Desktop tool bar, or simply by typing **`simulink`** in the Command Window. This opens the Simulink library window, which is shown for UNIX systems in Figure 8.1. On Windows systems, you see instead the Simulink Library Browser, shown in Figure 8.2.

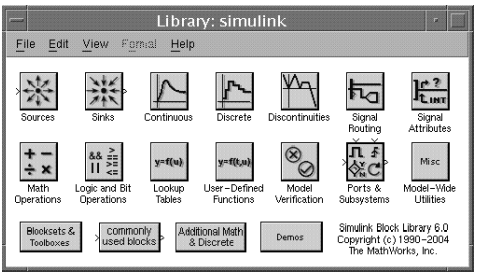


Figure 8.1. The Simulink Library.

To begin to use Simulink, click **New**:**Model** from the **File** menu. This opens a blank model window. You create a Simulink model by copying units, called *blocks*, from the various Simulink libraries into the model window. We will explain how to use this procedure to model the homogeneous linear ordinary differential equation (ODE) $u'' + 2u' + 5u = 0$, which represents a damped harmonic oscillator.

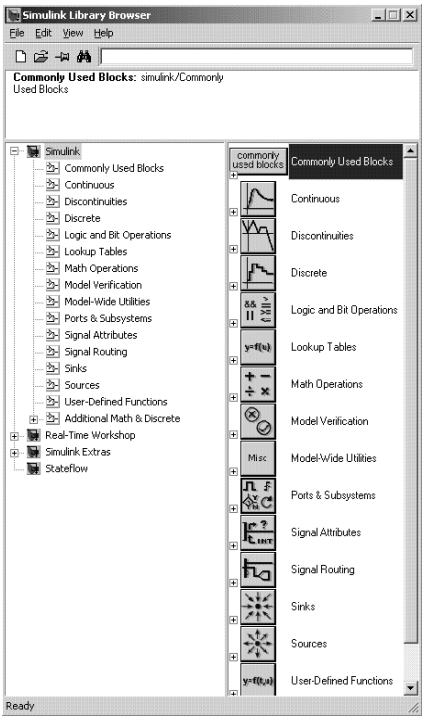


Figure 8.2. The Simulink Library Browser on a PC.

First we have to figure out how to represent the equation in a way that Simulink can understand. One way to do this is as follows. Since the time variable is *continuous*, we start by opening the "Continuous" library, in UNIX by double-clicking on the third icon from the left of the top row in Figure 8.1, or in Windows either by clicking on the small symbol ⊞ to the left of the "Continuous" icon at the top right of Figure 8.2, or else by clicking on the small icon to the left of the word "Continuous" in the left-hand panel of the Simulink Library Browser. When opened (on a Windows system), the "Continuous" library looks like Figure 8.3.

Notice that u and u' are obtained from u' and u'' (respectively) by integrating. Therefore, drag two copies of the Integrator block into the model window, and line them up with the mouse. Simulink will automatically label the second one "Integrator1", to distinguish it from the first block called "Integrator". Note that each Integrator block has an input port and an output port. Align the output port of the Integrator with the input port of the Integrator1 and join them with an arrow, using the left button on the mouse. The arrow joining two blocks is called a *signal*. Double-click on this arrow and a little box will appear, in which you can type the label u'. (Alternatively, you can right-click on the arrow, select **Signal Properties...**, and type your label in the **Signal Name** box.) As before, add arrows leading into the Integrator (representing u'') and leading out of the Integrator1 (representing u); these will not

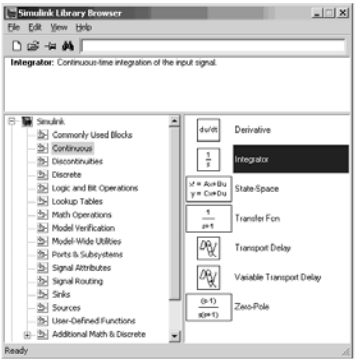


Figure 8.3. The Continuous Library.

yet connect to another block, so Simulink marks them with dotted lines to remind you that they are not fully operative. The idea so far is that u is obtained by integration from u', and u' is obtained by integration from u''. Your model window should now look like Figure 8.4.

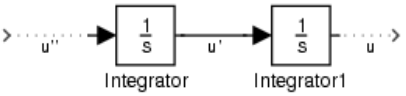


Figure 8.4. The First Stage of the Simulink Model.

Now we want to make use of the differential equation, which can be rewritten in the form $u'' = -5u - 2u'$. So we want to add other blocks to relate u'', the input to the first Integrator block, to u and u' according to this equation. For this purpose we add two Gain blocks, which implement multiplication by a constant, and one Sum block, used for addition. These are all chosen from the "Math Operations" library (seventh from the top in Figure 8.2, or first in the second row in Figure 8.1). Hooking them up the same way we did with the Integrator blocks gives a model window that looks something like Figure 8.5.

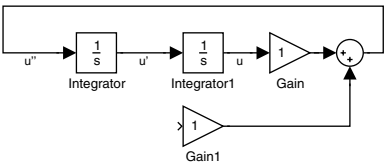


Figure 8.5. The Second Stage of the Simulink Model.

We need to go back and edit the properties of the Gain blocks, to change the

constants by which they multiply from the default of 1 to -5 (in "Gain") and -2 (in "Gain1"). To do this, double-click on each Gain block in turn. A Block Parameters box will open, in which you can change the Gain parameter to whatever you need. Next, we need to send u', the output of the first Integrator block, to the input port of block "Gain1". This presents a problem, since an Integrator block has only one output port and it's already connected to the next Integrator block. So we need to introduce a *branch line*. Position the mouse in the middle of the arrow connecting the two Integrators, hold down the CTRL key with one hand, simultaneously push down the left mouse button with the other hand, and drag the mouse around to the input port of the block entitled "Gain1". At this point we're almost done; we just need a block for viewing the output. Open up the "Sinks" library and drag a copy of the Scope block into the model window. Hook this up with a branch line (again using the CTRL key) to the line connecting the second Integrator and the Gain block. At this point you might want to relabel some of the blocks (by editing the text under each block), and also label some more of the arrows as before. We end up with the model shown in Figure 8.6.

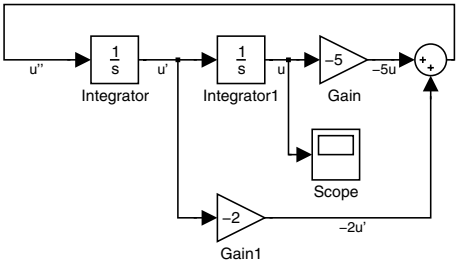


Figure 8.6. A Simulink Model for $u'' = -5u - 2u'$.

Now we're ready to run our simulation. First, it is a good idea to save the model, using **Save as...** from the **File** menu. One might choose to give it the name `dampedosc`. (MATLAB automatically adds the file extension `.mdl`, which stands for "model.") To see what is happening during the simulation, double-click on the Scope block to open an "oscilloscope" that will plot u as a function of t. Of course one needs to set initial conditions also; this can be done by double-clicking on the Integrator blocks and changing the line of the Block Parameters box that reads "Initial condition". For example, suppose that we set the initial condition for u' (in the first Integrator block) to 5 and the condition for u (in the second Integrator block) to 1. In other words, we are solving the system

$$\begin{aligned} u'' + 2u' + 5u &= 0, \\ u(0) &= 1, \\ u'(0) &= 5, \end{aligned}$$

which happens to have the exact solution

$$u(t) = 3e^{-t}\sin(2t) + e^{-t}\cos(2t).$$

Go to the **Simulation** menu and hit **Start**. You should see in the Scope window something like Figure 8.7. This of course is simply the graph of the function $3e^{-t}\sin(2t) + e^{-t}\cos(2t)$. (By the way, you might need to change the scale on the vertical axis of the Scope window. Clicking on the "binoculars" icon does an "automatic" rescale, and right-clicking on the vertical axis opens an **Axes Properties...** menu that enables you to select manually the minimum and maximum values of the dependent variable.) It is easy to go back and change some of the parameters and re-run the simulation again.

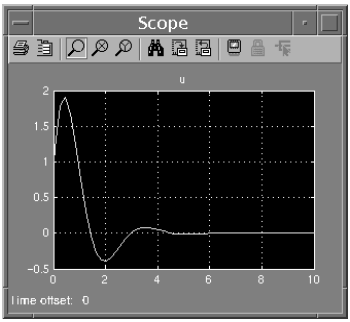


Figure 8.7. Scope Solution of $u'' = -5u - 2u'$, $u(0) = 1$, $u'(0) = 5$.

✓ Your first instinct might be to rely on the Derivative block, rather than the Integrator block, for simulating differential equations. But this has two drawbacks: it is harder to put in the initial conditions, and also numerical differentiation is much less stable than numerical integration.

Finally, suppose that one now wants to study the *inhomogeneous* equation for "forced oscillations," $u'' + 2u' + 5u = g(t)$, where g is a specified "forcing" term. For this, all we have to do is add another block to the model from the Sources library. Click on the shaft of the arrow at the top of the model going into the first Integrator and use **Cut** from the **Edit** menu to remove it. Then drag in another "Sum" block (from the Math Operations library) before the first Integrator and input a suitable source to one input port of the "Sum" block. For example, if $g(t)$ is to represent "noise," drag the Band-Limited White Noise block from the "Sources" library into the model and hook everything up as shown in Figure 8.8.

The output from this revised model (with the default values of 0.1 for the noise power and 0.1 for the noise sample time) looks like Figure 8.9. The effect of noise on the system is clearly visible from the simulation.

☆ An Engineering Example

Now we present a more complicated example, typical of an engineering application of Simulink. Consider a crane used for loading shipping containers at a port. It can be schematically represented by Figure 8.10.

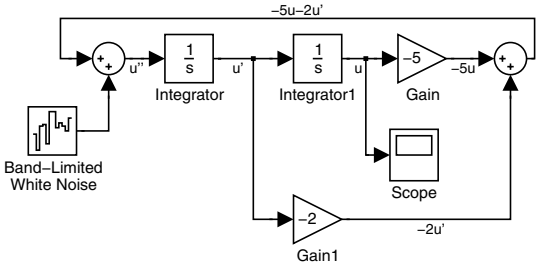


Figure 8.8. A Simulink Model of $u'' = -5u - 2u' + g(t)$.

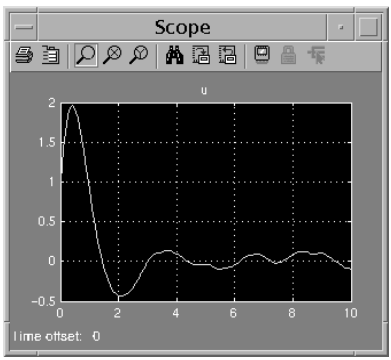


Figure 8.9. Scope Solution of an ODE with Noise.

Here we assume for simplicity that the radii of the pulleys and the size of the container are negligible, and that the pulley in the upper right can be moved horizontally, so that the crane operator can adjust the parameters w (the separation of the two pulleys) and l (the length of the cable holding the container) at will. We place our origin at the fixed pulley, so that, in terms of the parameters w, l, and θ (the angle between the cable and the vertical), the container is located at the point with coordinates

$$(x, y) = (w, 0) + (l \sin\theta, -l\cos\theta) = (w + l\sin\theta, -l\cos\theta). \tag{8.1}$$

We want to simulate the result of lifting the container off the ground at point $(w_0, -l_0)$, dragging it, and then depositing it again at point $(w_1, -l_0)$. In particular, we want to control the side-to-side oscillations, so that the swinging container won't hit something or somebody. For simplicity, we assume that the only forces on the container are tension in the cable (which can vary with time) and gravity. (So we neglect air resistance, for example.)

Let's make a further simplification: we assume that the crane operator moves the container in three steps, where in the first step the container is lifted straight up, in the second step l (the length of the cable connecting the container to the movable pulley) is held fixed, and in the third step the container is lowered straight down. Steps 1 and

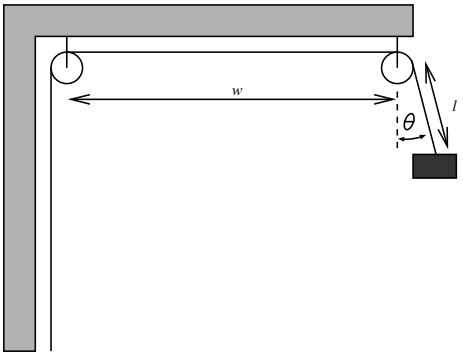


Figure 8.10. A Crane for Loading Containers.

3 are not especially interesting, so we concentrate on step 2, which means we assume that l is held constant and w is a function of time to be selected by the crane operator, with initial value w_0 and final value w_1. Let m be the mass of the container. To get the equations of motion, note that the kinetic energy K and gravitational potential energy V of the container are given by

$$\begin{aligned} K &= \frac{m}{2}\big((x')^2 + (y')^2\big) = \frac{m}{2}\Big(\big(w' + l(\cos\theta)\theta'\big)^2 + \big(l(\sin\theta)\theta'\big)^2\Big) \\ &= \frac{m}{2}\Big((w')^2 + l^2(\theta')^2 + 2lw'(\cos\theta)\theta'\Big) \\ V &= mgy = -mgl\cos\theta. \end{aligned} \tag{8.2}$$

In Lagrangian mechanics, the equation of motion is

$$\frac{\partial L}{\partial \theta} = \frac{d}{dt}\frac{\partial L}{\partial \theta'},$$

where the Lagrangian is $L = K - V$, so (after dividing out by the constant factor of m) we get

$$\begin{aligned} -gl\sin\theta - lw'(t)(\sin\theta)\theta' &= \frac{d}{dt}\Big(l^2\theta' + lw'(t)\cos\theta\Big) \\ &= l^2\theta'' + lw''(t)\cos\theta - lw'(t)(\sin\theta)\theta', \end{aligned}$$

or (after dividing by l^2)

$$\theta'' + \frac{1}{l}w''(t)\cos\theta + \frac{g}{l}\sin\theta = 0, \tag{8.3}$$

which is the usual "pendulum equation" (see the discussion of the nonlinear pendulum in Chapter 10) with an extra term in it involving $w''(t)$, the horizontal acceleration of the crane. Now we can build our Simulink model for the crane. We want to experiment with various possibilities for $w(t)$ and see what effect these have on the measure of "swinging," $\theta(t)$. Let's use meters and seconds as our units and take

$g = 9.81\,\mathrm{m/sec}^2$, $l = 5\,\mathrm{m}$, $w_0 = 0$, and $w_1 = 10\,\mathrm{m}$. The time required to move the container should be on the order of a few minutes, or let's say about $200\,\mathrm{sec}$. So we want w to be a continuous function, preferably with a continuous second derivative (since the force on the motor driving the pulley is proportional to w'' and should be a continuous function of time), with $w(0) = 0$ and $w(200) = 10$. A way to get such a function is to take

$$w(t) = 5 + a(t - 100) + b(t - 100)^3 + c(t - 100)^5.$$

This ensures that $w(100) = 5$ and that the graph of w has odd symmetry around the point $(100, 5)$. In fact, since only odd powers of $t - 100$ appear in the formula for $w(t) - 5$, it follows that $w(100 + t) - 5 = -(w(100 - t) - 5)$, and, on setting $t = 0$, we see that $2w(100) = 10$ or $w(100) = 5$. Putting $t = 100$ into this equation gives $w(200) - 5 = -w(0) + 5$, so we'll have $w(200) = 10$ if $w(0) = 0$. Thus we solve for the coefficients a, b, and c so that $w(0) = w'(0) = w''(0) = 0$. Here is a solution obtained using MATLAB:

```
>> syms t a b c;
>> w = 5 + a*(t-100) + b*(t-100)^3 + c*(t-100)^5;
>> w0 = subs(w, t, 0);
>> w1 = subs(diff(w,t), t, 0);
>> w2 = subs(diff(w,t,2), t, 0);
>> [aa, bb, cc] = solve(w0, w1, w2);
>> w = subs(w, [a,b,c], [aa,bb,cc])
```

```
w =
-35/8+3/32*t-1/160000*(t-100)^3+3/16000000000*(t-100)^5
```

A plot of this function is shown in Figure 8.11. For this function, the formula for

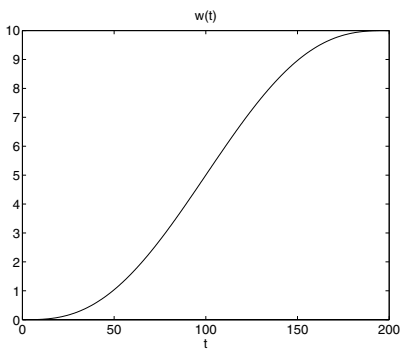


Figure 8.11. A Possible $w(t)$.

$w''(t)$ is:

```
>> diff(w, t, 2)
```

```
ans =
-3/80000*t+3/800+3/800000000*(t-100)^3
```

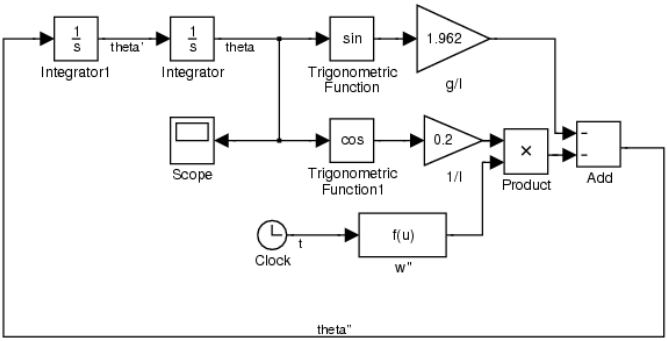


Figure 8.12. A Simulink Model for Studying the Motion of a Crane.

A possible Simulink model for studying this system is shown in Figure 8.12. Let's discuss how this model works and how to assemble it. We begin by rewriting equation (8.3) in the form

$$\theta'' = -\frac{g}{l}\sin\theta - \frac{1}{l}w''(t)\cos\theta.$$

Thus we want to represent θ'' as a sum of two terms, each with a "minus sign" in front, which are assembled together in an Add block. (By the way, there is no difference between the Add and Sum blocks except for the shape of the representing icon.) The signal representing θ'' then enters the block called Integrator1 at the upper left, and the signal exiting the second Integrator represents θ itself. The Clock block outputs the value of t, which goes into a Function block (from the User-Defined Functions library) used for computing $w''(t)$. The Product and Trigonometric Function blocks, both from the Math Operations library, are self-explanatory. There are two Gain blocks, again from the Math Operations library, used to multiply $\sin\theta$ and $\cos\theta$ by the constants g/l and $1/l$, respectively. For convenience, we have renamed these two blocks to indicate which is which. To rename a block, click on the label under the block and the label will be surrounded by a gray box. You can then erase the old name and insert a new one. As in the example in the previous section, the Gain blocks have to be customized for the appropriate constants using the Block Parameters dialog box. Similarly, the Block Parameters dialog box for the "Trigonometric Function1" block can be used to switch from the sine function (the default) to the cosine function that we need. Since the crane starts at rest (with initial conditions $\theta(0) = \theta'(0) = 0$, we do not need to change the default initial condition of 0 in each of the Integrator blocks. One does need to customize the Function block by inserting the formula for $w''(u)$. (Simulink insists here that the independent variable be called u, not t.) Finally, one needs to click on the **Simulation** menu to change the maximum value of t from the default of 5 to the value 200 needed for this problem.

Running the simulation results (after you have clicked on the "binoculars" icon to rescale the graph) in the Scope picture shown in Figure 8.13. If you look at the legend on the picture, you will see that the maximum value of θ is on the order of

2×10^{-4} radians, which is pretty small, even when multiplied by the length of the cable, $l = 5\,\mathrm{m}$. So we conclude that the shaking of the container is not a serious problem in this case.

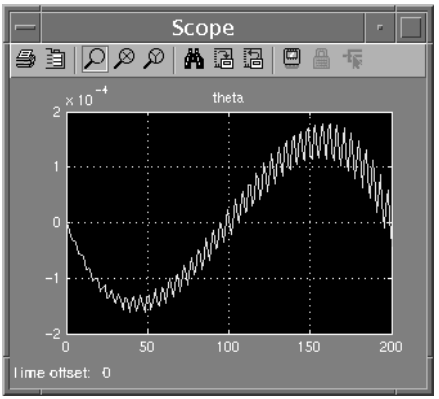


Figure 8.13. Scope Output of the Oscillations of a Crane.

☆ Communication with the Workspace

The examples we have discussed should suffice to give you an idea of how to use Simulink as a stand-alone simulation tool. But part of the power of Simulink comes from the way it can be combined with MATLAB. For example, one can run a Simulink model from within an M-file, or do further processing within MATLAB on the output of a Simulink simulation. In this section, we discuss some of the key commands for combining Simulink with MATLAB.

We'll begin with a practical example. Suppose that we want to recompute the formula for $w(t)$ in the crane example so that w reaches the value of 10 at $t = 20$ instead of at $t = 200$. In other words, we want to retain the shape of $w(t)$ in Figure 8.11, but compress the scale on the horizontal axis by a factor of 10. What happens to the oscillations now that the crane moves the container more rapidly? Is the process now dangerous? To answer this question, we replace the old $w(t)$ by

```
>> w = simplify(subs(w, t, 10*t))
w =
1/80*t^3+3/160000*t^5-3/3200*t^4
```

and replace the old $w''(t)$ by

```
>> diff(w, t, 2)
ans =
3/40*t+3/8000*t^3-9/800*t^2
```

The first thing we need to do is to make a change in the parameters of the block labeled `f(u)` in Figure 8.12. It is possible to do this with the commands **`get_param`** and

set_param from the command line or from an M-file, not just by bringing up the Block Parameters box. This is especially convenient if you need to a run a simulation many times with various values of the parameters. Sometimes it helps to use the commands **gcs** ("get current system") and **find_system** to locate the names of the relevant blocks. Here's an example, based on the supposition that we've already opened the model in Figure 8.12.

```
>> gcs

ans =
crane

>> find_system(gcs, 'Type', 'block')

ans =
    'crane/1//l'
    'crane/Add'
    'crane/Clock'
    'crane/Integrator'
    'crane/Integrator1'
    'crane/Product'
    'crane/Scope'
    'crane/Trigonometric
Function'
    'crane/Trigonometric
Function1'
    'crane/g//l'
    'crane/w"'
```

This gives us the names of all of the blocks; in this case, we need to change the parameters of the block entitled `'crane/w"'`. (If we wanted to change the cable length, we would also have to change the blocks `'crane/1//l'` and `'crane/g//l'`.) To see the current value of the `'Expr'` parameter of this block, which encodes the function $w''(t)$, we type

```
>> get_param('crane/w"', 'Expr')

ans =
-3/80000*u+3/800+3/800000000*(u-100)^3
```

So we can reset this with

```
>> set_param('crane/w"', 'Expr', ...
   '3/40*u+3/8000*u^3-9/800*u^2')
```

To re-run the model, instead of using **Simulation**:**Start**, we can use the command **sim** from the command line or within an M-file. The simplest form of this command just runs the model with the existing parameters. But one can also use this command to set the time interval and to send the output to the workspace. For example, in our situation, we would type

```
>> [t, theta] = sim('crane', [0, 20]);
```

We could then type

```
>> plot(t, theta)
```

to plot the results, giving the picture in Figure 8.14. Here the curve that ends up on top represents $\theta(t)$, and the other curve represents $\theta'(t)$. Or to duplicate what would be

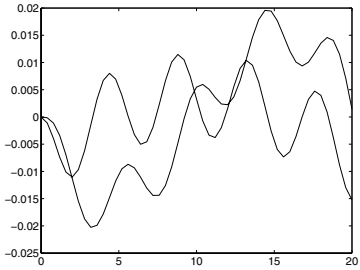


Figure 8.14. Plot of $\theta(t)$ and $\theta'(t)$ for the Revised Crane Model.

seen in the Scope window, we can replace **plot** by **simplot**, getting Figure 8.15:

```
>> simplot(t, theta(:,1))
```

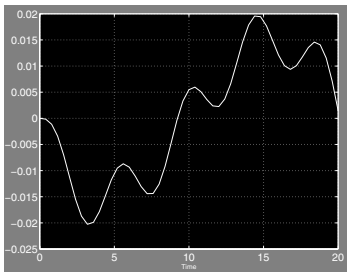


Figure 8.15. "Scope" Plot of $\theta(t)$ for the Revised Crane Model.

✓ Incidentally, if you draw a plot with **simplot**, it always appears in the figure window as in Figure 8.15, with a light-colored plot on a black background. But if you print the plot (either to the printer or to a file), it will sometimes (depending on your **print** defaults) appear with the colors reversed, i.e., as a dark plot on a white background. To undo this color reversal, you need to insert the command

```
>> set(gcf, 'InvertHardcopy', 'off')
```

before executing the **print** command.

The result of our analysis shows that, if the shipping container is moved in 20 seconds instead of 200, the oscillations are on the order of 0.02 radians. Thus the container will swing back and forth about 0.10 m or 10 cm, which is again a reasonable amount. We leave it to the reader to see what happens if the container is moved in only 2 seconds!

Chapter 9

☆ GUIs

With MATLAB you can create your own *Graphical User Interface*, or GUI, which consists of a Figure window containing menus, buttons, text, graphics, etc., that a user can manipulate interactively with the mouse and keyboard. There are two main steps in creating a GUI: one is designing its layout, and the other is writing *callback functions* that perform the desired operations when the user selects different features.

GUI Layout and GUIDE

Specifying the location and properties of various objects in a GUI can be done with commands such as **`uicontrol`**, **`uimenu`**, and **`uicontextmenu`** in an M-file. MATLAB also provides an interactive tool (a GUI itself!) called GUIDE (this stands for Graphical User Interface Development Environment) that greatly simplifies the task of building a GUI. We will describe here how to get started writing GUIs with the MATLAB 7 version of GUIDE, which has some significant enhancements over earlier versions. The version of GUIDE in MATLAB 6 is roughly similar, but some of the menu items and options are different or missing.

✓ One possible drawback of GUIDE is that it equips your GUI with commands that are new in MATLAB 7 and it saves the layout of the GUI in a binary `.fig` file. If your goal is to create a robust GUI that many different users can use with different versions of MATLAB, you may be better off writing the GUI from scratch as an M-file.

To open GUIDE, select **File**:**New**:**GUI** from the Desktop menu bar or type **`guide`** in the Command Window. You will see the GUIDE Quick Start dialog box, shown in Figure 9.1.

Note that there are two tabs at the top. The left-hand one, "Create New GUI", is open by default. You can start by selecting one of the various kinds of GUIs on the left. This will pop open the Layout Editor, in which you design the appearance of your GUI. For purposes of the example here, we'll assume that you select the third item, "GUI with Axes and Menu". The Layout Editor that opens looks like Figure 9.2. (We've shown the version that appears in UNIX or Linux. In Windows, there is one extra button on the left, for "ActiveX" controls.) Let's say that you want to design a GUI that will accept a MATLAB plotting command as input, display the corresponding output, and have buttons for making various changes in the appearance of the output.

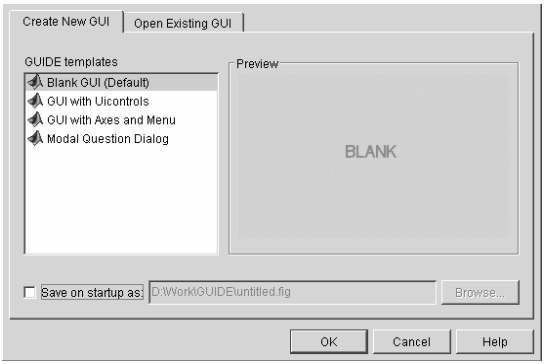


Figure 9.1. The GUIDE Quick Start Dialog Box.

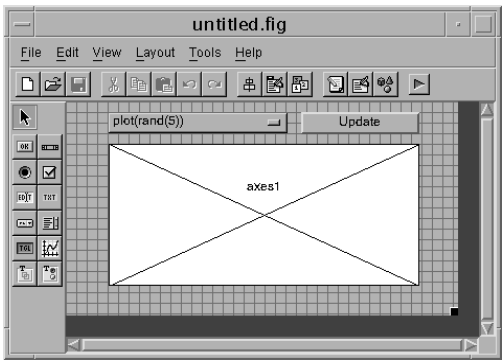


Figure 9.2. The Layout Editor.

The buttons at the left of the Layout Editor are used for inserting various kinds of objects. You build a GUI by clicking on one of these buttons, then moving to a desired location in the grid, and clicking again to place an object on the grid. To see what type of object each button corresponds to, move the mouse over a button but don't click; soon a yellow box with the name of the button will appear. Once you have placed an object on the grid, you can click and drag (hold down the left mouse button and move the mouse) on the middle of the object to move it or click and drag on a corner to resize the object. After you have placed several objects, you can select multiple objects by clicking and dragging on the background grid to enclose them with a rectangle. Then you can move the objects as a block with the mouse, or align them by selecting **Align Objects...** from the **Tools** menu.

To change properties of an object such as its color, the text within it, etc., you must open the *Property Inspector* window. To do so, you can double-click on an object, or choose **Property Inspector** from the **View** menu and then select the object you want to alter with the left mouse button. You can leave the Property Inspector open

throughout your GUIDE session and go back and forth between it and the Layout Editor.

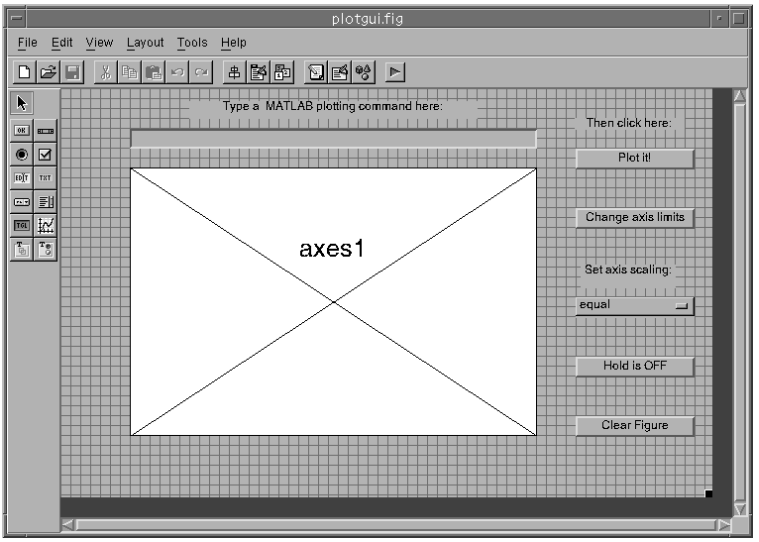


Figure 9.3. The Layout Editor with Design for a "Plot GUI".

Figure 9.3 shows what the Layout Editor window looks like after several objects have been placed and their properties adjusted. Let us describe how we created the objects that make up the GUI.

The two boxes on the top, as well as the one labeled "Set axis scaling:", are *Static Text* boxes, which the user of the GUI will not be allowed to manipulate. To create each of them, we first clicked on the "Static Text" button – the one to the left of the grid labeled "TXT" – and then clicked in the grid where we wanted to add the text. Next, to set the text for the box we opened the Property Inspector and clicked on the square button next to "String", which opens a new window that contains editable default text. Finally, we resized each box according to the length of its text (by clicking on a corner and dragging it).

The buttons labeled "Plot it!", "Change axis limits", and "Clear figure" are all *Push Button* objects, created using the button labeled "OK". To make these buttons all the same size, we first created one of them and then, after sizing it, we duplicated it (twice) by clicking the right mouse button on the existing object and selecting **Duplicate**. We then moved each new Push Button to a different position and changed its text in the same way as we did for the Static Text boxes.

The blank box near the top of the grid is an *Edit Text* box, which allows the user to enter text. We created it with the button labeled "EDIT" and then cleared its default text in the same way that we changed text before. Below the Edit Text box is a large *Axes* box, resized from the one already present in Figure 9.2, and in the lower right the button labeled "Hold is OFF" is a *Toggle Button*, created with the button labeled

"TGL". For toggling (on–off) commands you could also use a *Radio Button* or a *Checkbox*, which are denoted, respectively, by the buttons with a dot and a check mark in them. Finally, the box on the right that says "equal" is a *Popup Menu* – its button in the Layout Editor is right below the *Edit Text* button. Popup Menus and *Listbox* objects allow you to let the user choose among several options.

We moved, resized, and in most cases changed the properties of each object in the fashion described above. In the case of the Popup Menu, after we selected the "String" button in the Property Inspector, we entered into the window that appeared three words on three separate lines: **`equal`**, **`normal`**, and **`square`**. Using multiple lines is necessary in order to give the user multiple choices in a Popup Menu or Listbox object.

✓ In addition to populating your GUI with the objects we described above, you can create a menu bar for it using the Menu Editor, which you can open by selecting **Menu Editor...** from the **Tools** menu. You can also use the Menu Editor to create a *context menu* for an object; this is a menu that appears when you click the right mouse button on the object. See the online documentation for GUIDE to learn how to use the Menu Editor.

We also gave our GUI a title, which will appear in the title bar of its window, as follows. We clicked on the background grid in the Layout Editor to select the entire GUI (as opposed to an object within it) and went to the Property Inspector. There we changed the text to the right of "Name" from "Untitled" to "Simple Plot GUI".

Saving and Running a GUI

To save a GUI, select **Save As...** from the **File** menu. Type a file name for your GUI without any extension; for the GUI described above we chose `plotgui`. When you save, two files are created, an M-file and a binary file with extension `.fig`, so in our case the resulting files were named `plotgui.m` and `plotgui.fig`. When you save a GUI for the first time, the M-file for the GUI will appear in a separate Editor/Debugger window. We will describe how and why to modify this M-file in the next section.

⇨ **The instructions in this and the following section assume the default settings of the GUI Options..., which you may have inspected upon starting GUIDE, as described above. Otherwise, you can access them from the Tools menu. We assume in particular that "Generate FIG-file and M-file", "Generate callback function prototypes", and "GUI allows only one instance to run" are selected.**

Once it has been saved, you can run the GUI from the Command Window by typing its name, in our case **`plotgui`**, whether or not GUIDE is running. Both the `.fig` file and the `.m` file must be in your current directory or MATLAB path. You can also run it from the Layout Editor by typing CTRL+T or selecting **Run** from the **Tools** menu. A copy of the GUI will appear in a separate window, without all the

surrounding menus and buttons of the Layout Editor. (If you have added new objects since the last time you saved or activated the GUI, the M-file associated with the GUI will also be brought to the front.) Figure 9.4 shows how the GUI we created above looks when activated. The plot in the axes window will be explained in the section *GUI Callback Functions*.

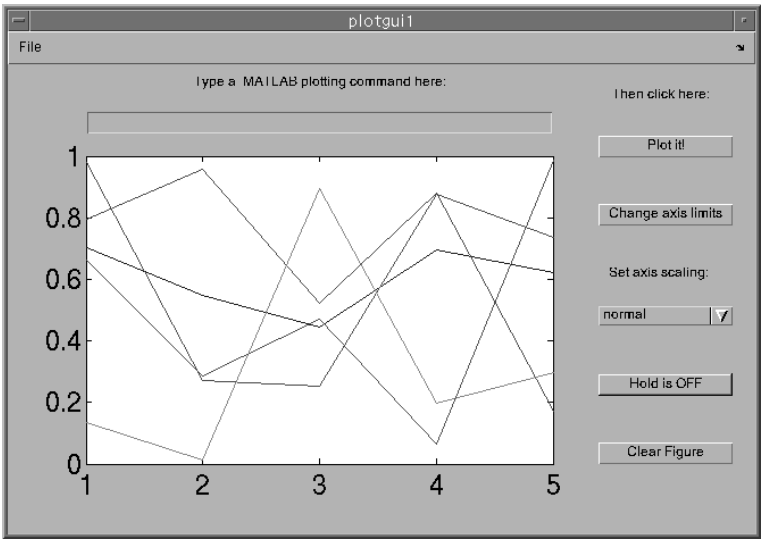


Figure 9.4. The "Plot GUI" Activated.

Notice that the appearance of the GUI differs slightly from that in the GUIDE window; in particular, the font size may differ. For this reason you may have to go back to the GUIDE window after activating a GUI and resize some objects accordingly. The changes you make will not immediately appear in the active GUI; to see their effect you must activate the GUI again.

The objects you create in the Layout Editor are inert within that window – you can't type text in the Edit Text box, you can't see the additional options by clicking on the Popup Menu, etc. But, in an activated GUI window, objects such as Toggle Buttons and Popup Menus will respond to mouse clicks. However, they will not actually perform any operations until you write a callback function for each of them.

GUI Callback Functions

When you are ready to create a callback function for a given object, make sure you've saved your GUI. Then look at the Property Inspector for the object in the Layout Editor, and under Tag you will see the tag by which it is identified in the M-file associated with the GUI. Open the M-file in an Editor/Debugger window (if it's not open already), and go to the corresponding section of the M-file. You will see a small block of text that looks like this:

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

All you need to do now to bring this Push Button to life is to add the commands that you want performed when the user clicks on the button. Of course you also need to save the M-file, which you can do in the usual way from the Editor/Debugger, or by activating the GUI from the Layout Editor. Each time you save or activate a GUI, a block of five lines like the ones above is automatically added to the GUI's M-file for any new objects or menu items that you have added to the GUI and that should have callback functions. In many cases, after you have set enough properties of an object, GUIDE is smart enough to insert some appropriate commands into the M-file automatically, but you may need to modify them. For example, GUIDE puts into the M-file a section that begins

```
% --- Executes just before plotgui is made visible.
function plotgui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
```

and includes the lines

```
% This sets up the initial plot - only do when we are invisible
% so window can get raised using plotgui.
if strcmp(get(hObject,'Visible'),'off')
    plot(rand(5));
end
```

It is this section of the M-file that is responsible for the "random lines" that you see in the axes window in Figure 9.4. Had we taken these lines out or modified them, the GUI could have come loaded with a very different picture, or with no picture at all.

For the Popup Menu on the right-hand side of the GUI, we put the following lines into its callback function (modified from the template that came with the primordial GUI of Figure 9.2).

```
popup_sel_index = get(hObject, 'Value');
switch popup_sel_index
    case 1
        axis equal
    case 2
        axis normal
    case 3
        axis square
end
```

Each time the user of the GUI selects an item from a Popup Menu, MATLAB sets the "Value" property of the object to the line number selected and runs the associated callback function. As we described in Chapter 5, you can use **`get`** to retrieve the current setting of a property of a graphics object. When you use the callback templates provided by GUIDE as we have described, the variable **`hObject`** will contain the handle (the required first argument of **`get`** and **`set`**) for the associated object. (If

you are using another method to write callback functions, you can use the MATLAB command **gcbo** in place of **hObject**.) For our sample GUI, line 1 of the Popup Menu says "equal", and, if the user selects line 1, the callback function above runs **axis equal**; line 2 says "normal", etc.

✓ You may have noticed that the Popup Menu in Figure 9.4 says "normal" rather than "equal" as in Figure 9.3; that's because we set its "Value" property to 2 when we created the GUI, using the Property Inspector. In this way you can make the default selection something other than the first item in a Popup Menu or Listbox.

For the Push Button labeled "Plot it!", we wrote the following callback function:

```
set(handles.figure1, 'HandleVisibility', 'callback');
eval(get(handles.edit1, 'String'))
```

Here **handles.figure1** and **handles.edit1** are the handles for the entire GUI window and for the Edit Text box, respectively. Again these variables are provided by the callback templates in GUIDE, and if you do not use this feature you can generate the appropriate handles with **gcbf** and **findobj(gcbf, 'Tag', 'edit1')**, respectively. The second line of the callback function above uses **get** to find the text in the Edit Text box and then runs the corresponding command with **eval**. The first line uses **set** to make the GUI window accessible to graphics commands used within callback functions; if we did not do this, a plotting command run by the second line would open a separate figure window.

✓ We could also associate a callback function with the Edit Text box; this function would be run each time the user presses the ENTER key after typing text in the box. The callback function **eval(get(hObject, 'String'))** will run the command just typed, providing an alternative to (or making superfluous) the "Plot it!" button.

✓ Another way to enable plotting within a GUI window is to select **GUI Options...** from the **Tools** menu in the Layout Editor, and within the window that appears change "Command-line accessibility" from "Callback" to "On". This has the possible drawback of allowing plotting commands the user types in the Command Window to affect the GUI window. A safer approach is to keep "Command-line accessibility" set to "Callback". With this setting, the line

```
set(handles.figure1, 'HandleVisibility', 'callback');
```

in the callback functions becomes unnecessary. However, we recommend keeping this line in the callback functions in case you decide to change the "Command-line accessibility" later.

In the example **plotgui** from the previous section, there is one case where we used an existing MATLAB command as a callback function. For the Push Button labeled "Change axis limits...", we simply entered **axlimdlg** into its callback function

in `plotgui.m`. This command opens a *dialog box* that allows a user to type new values for the ranges of the x- and y-axes. MATLAB has a number of dialog boxes that you can use either as callback functions or in an ordinary M-file. For example, you can use **`inputdlg`** in place of **`input`**. Type **`help uitools`** for information on available dialog boxes.

Here is our callback function for the Push Button labeled "Clear figure":

```
set(handles.edit1, 'String', '');
set(handles.figure1, 'HandleVisibility', 'callback');
cla reset
```

The first line clears the text in the Edit Text box and the last line clears the Axes box in the GUI window. (If your GUI contains more than one Axes box, you can use **`axes`** to select the one you want to manipulate in each of your callback functions.)

We used the following callback function for the Toggle Button labeled "Hold is OFF":

```
set(handles.figure1, 'HandleVisibility', 'callback');
if get(hObject, 'Value')
    hold on
    set(hObject, 'String', 'Hold is ON');
else
    hold off
    set(hObject, 'String', 'Hold is OFF');
end
```

We get the "Value" property of the Toggle Button in the same way as in the Popup Menu callback function above, but for a Toggle Button this value is either 0 if the button is "out" (the default) or 1 if the button is pressed "in." (Radio Buttons and Checkboxes also have a "Value" property of either 0 or 1.) When the user first presses the Toggle Button, the value is set to 1, so the callback function above runs **`hold on`** and resets the string displayed on the Toggle Button to reflect the change. The next time the user presses the button, these operations are reversed.

Finally, with an object in a GUI you can also associate a variant of a callback function called a "ButtonDownFcn" instead. Such a function will begin something like

```
% --- Executes on mouse press over axes background.
function axes1_ButtonDownFcn(hObject, eventdata, handles)
```

and will be run when the user clicks in the axes window, or in the case of other objects such as a Push Button, clicks with the right mouse button (as opposed to the left mouse button for the callback function). To create such a function, you can right-click on the object in the Layout Editor and select **View Callbacks**:**ButtonDownFcn**. You can associate functions with several other types of user events as well; to learn more, see the online documentation, or experiment by clicking the right mouse button on various objects and on the grid behind them in the Layout Editor.

Chapter 10

Applications

In this chapter, we present examples showing you how to apply MATLAB to problems in several disciplines. Each example is presented in the form of a MATLAB M-file, published to LaTeX. We modified the default style sheet for publishing to LaTeX to adjust the spacing between input, output, and text, and to allow formatting of mathematical formulas not just as displayed equations but also within a paragraph. We also made a few minor adjustments to the published LaTeX code to improve line breaks. Finally, because **`publish`** does not produce italic text, we used bold text instead in places. These examples are illustrations of the kinds of polished, integrated documents that you can create with MATLAB. The examples are:

- Illuminating a Room
- Mortgage Payments
- Monte Carlo Simulation
- Population Dynamics
- Linear Economic Models
- Linear Programming
- The 360° Pendulum
- ☆ Numerical Solution of the Heat Equation
- ☆ A Model of Traffic Flow

We have not explained all the MATLAB commands that we use; you can learn about the new commands from the online help. Simulink is used in *A Model of Traffic Flow* and as an optional accessory in *Population Dynamics* and *Numerical Solution of the Heat Equation*. The example on *Linear Programming* also requires an M-file found (in slightly different forms) in the Simulink and Optimization toolboxes. The examples require different levels of mathematical background and expertise in other subjects. *Illuminating a Room, Mortgage Payments*, and *Population Dynamics* use only high-school mathematics. *Monte Carlo Simulation* uses some probability and statistics; *Linear Economic Models* and *Linear Programming*, some linear algebra; *The* 360° *Pendulum*, some ordinary differential equations; *Numerical Solution of the Heat Equation*, some partial differential equations; and *A Model of Traffic Flow*, differential equations, linear algebra, and familiarity with the function e^z for z a complex number. Even if you don't have the background for a particular example, you should be able to learn something about MATLAB from the presentation.

Illuminating a Room

Suppose that we need to decide where to put light fixtures on the ceiling of a room measuring 10 m by 4 m by 3 m high in order to illuminate it best. For aesthetic reasons, we are asked to use a small number of incandescent bulbs. We want the bulbs to total a maximum of 300 watts. For a given number of bulbs, how should they be placed to maximize the intensity of the light in the darkest part of the room? We also would like to see how much improvement there is in going from one 300-watt bulb to two 150-watt bulbs to three 100-watt bulbs, and so on. To keep things simple, we assume that there is no furniture in the room and that the light reflected from the walls is insignificant compared with the direct light from the bulbs.

One 300-Watt Bulb

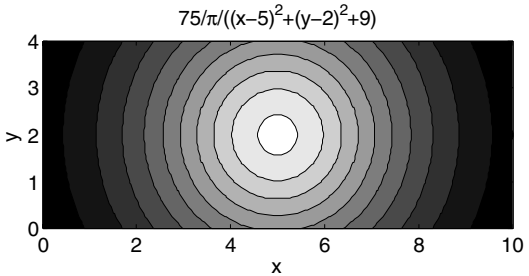
If there is only one bulb, then we want to put the bulb in the center of the ceiling. Let's picture how well the floor is illuminated. We introduce coordinates x running from 0 to 10 in the long direction of the room, and y running from 0 to 4 in the short direction. The intensity at a given point, measured in watts per square meter, is the power of the bulb, 300, divided by 4π times the square of the distance from the bulb. Since the bulb is 3 m above the point $(5, 2)$ on the floor, at a point (x, y) on the floor, we can express the intensity as follows.

```
syms x y; illum = 300/(4*pi*((x - 5)^2 + (y - 2)^2 + 3^2))
```

```
illum =

75/pi/((x-5)^2+(y-2)^2+9)
```

We can use **ezcontourf** to plot this expression over the entire floor. We use the option **colormap** to arrange for a color gradation that helps us to see the illumination. (See the online help for more **colormap** options.)

```
ezcontourf(illum, [0 10 0 4])
colormap('gray'); axis equal tight
```



The darkest parts of the floor are the corners. Let us find the intensity of the light at the corners and at the center of the room.

```
subs(illum, {x, y}, {0, 0})

ans =

    0.6282


subs(illum, {x, y}, {5, 2})

ans =

    2.6526
```

The center of the room, at floor level, is about four times as bright as the corners when there is only one bulb on the ceiling. Our objective is to light the room more uniformly using more bulbs with the same total amount of power. Before proceeding to deal with multiple bulbs, we observe that the use of **ezcontourf** is somewhat confining, as it does not allow us to control the number of contours in our pictures. This will be helpful in seeing the light intensity; therefore we shall plot numerically rather than symbolically, that is we shall use **contourf** instead of **ezcontourf**.

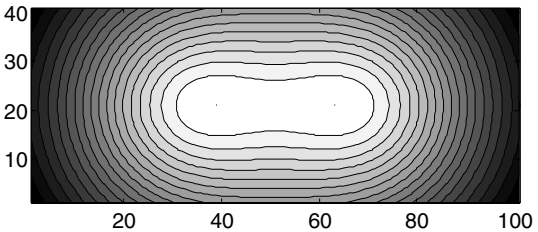
Two 150-Watt Bulbs

In this case we need to decide where to put the two bulbs. Common sense tells us to arrange the bulbs symmetrically along a line down the center of the room in the long direction; that is, along the line $y = 2$. Define a function that gives the intensity of light at a point (x, y) on the floor due to a 150-watt bulb at a position $(d, 2)$ on the ceiling.

```
light2 = @(x,y,d) 150./(4*pi*((x - d).^2 + (y - 2).^2 + 3^2));
```

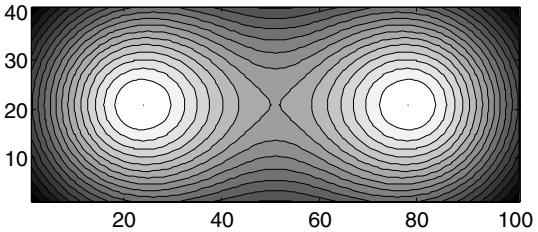
Let's get an idea of the illumination pattern if we put one light at $d = 3$ and the other at $d = 7$. We specify the drawing of 20 contours in this and the following plots.

```
[X,Y] = meshgrid(0:0.1:10, 0:0.1:4);
contourf(light2(X, Y, 3) + light2(X, Y, 7), 20);
colormap('gray'); axis equal tight
```



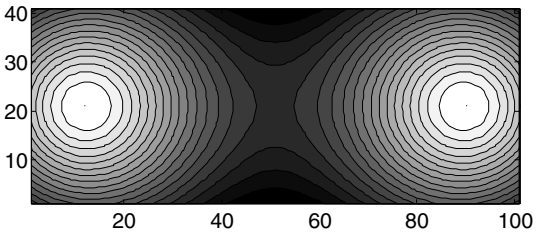
The floor is more evenly lit than with one bulb, but it looks as if the bulbs are closer together than they should be. If we move the bulbs further apart, the center of the room will get dimmer but the corners will get brighter. Let's try changing the location of the lights to $d = 2$ and $d = 8$.

```
contourf(light2(X, Y, 2) + light2(X, Y, 8), 20);
colormap('gray'); axis equal tight
```



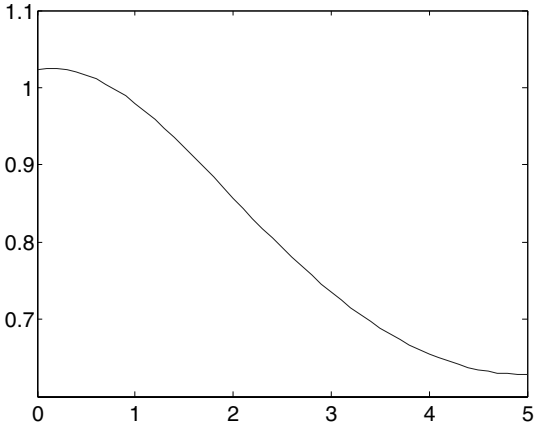
This is an improvement. The corners are still the darkest spots of the room, though the light intensity along the walls toward the middle of the room (near $x = 5$) is diminishing as we move the bulbs further apart. Still, to better illuminate the darkest spots we should keep moving the bulbs apart. Let's try lights at $d = 1$ and $d = 9$.

```
contourf(light2(X, Y, 1) + light2(X, Y, 9), 20);
colormap('gray'); axis equal tight
```



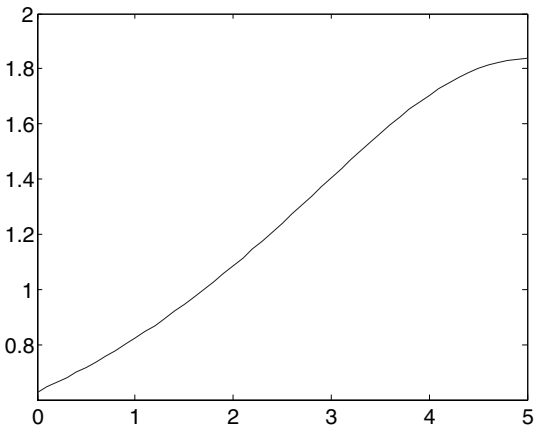
Looking along the long walls, the room is now darker toward the middle than at the corners. This indicates that we have spread the lights too far apart. We could proceed with further contour plots, but instead let's be systematic about finding the best position for the lights. In general, we can put one light at $x = d$ and the other symmetrically at $x = 10 - d$ for d between 0 and 5. Judging from the examples above, the darkest spots will be either at the corners or at the midpoints of the two long walls. By symmetry, the intensity will be the same at all four corners, so let's graph the intensity at one of the corners $(0, 0)$ as a function of d.

```
d = 0:0.1:5;
plot(d, light2(0, 0, d) + light2(0, 0, 10 - d))
```



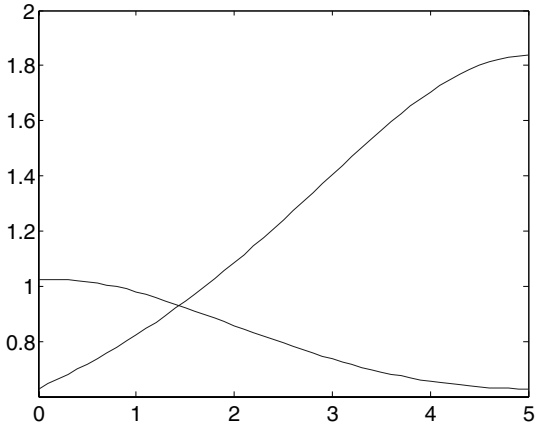
As expected, the smaller d is, the brighter the corners are. In contrast, the graph for the intensity at the midpoint $(5, 0)$ of a long wall (again by symmetry it does not matter which of the two long walls we choose) should grow as d increases toward 5.

```
d = 0:0.1:5;
plot(d, light2(5, 0, d) + light2(5, 0, 10 - d))
```



We are after the value of d for which the lower of the two numbers on the above graphs (corresponding to the darkest spot in the room) is as high as possible. We can find this value by showing both curves on one graph.

```
hold on; plot(d, light2(0, 0, d) + light2(0, 0, 10 - d))
hold off
```



The optimal value of d is at the point of intersection, near 1.4, with minimum intensity a little under 1. To get the optimum value of d, we find exactly where the two curves intersect.

```
syms d;
eqn = @(d) light2(0, 0, d) + light2(0, 0, 10 - d) - ...
    light2(5, 0, d) - light2(5, 0, 10 - d);
dint = fzero(eqn, [0 5])
```

```
dint =

    1.4410
```

So the lights should be placed about 1.44 m from the short walls. For this configuration, the approximate intensity at the darkest spots on the floor is as follows.

```
light2(0, 0, dint) + light2(0, 0, 10 - dint)
```

```
ans =

    0.9301
```

The darkest spots in the room have intensity around 0.93, as opposed to 0.63 for a single bulb. This is an improvement of about 50%.

Three 100-Watt Bulbs

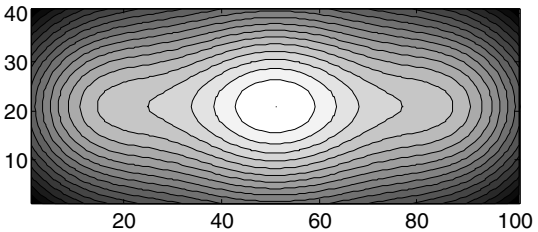
We redefine the intensity function for 100-watt bulbs.

```
light3 = @(x,y,d) 100./(4*pi*((x - d).^2 + (y - 2).^2 + 3^2))
```

```
light3 =

    @(x,y,d) 100./(4*pi*((x - d).^2 + (y - 2).^2 + 3^2))
```

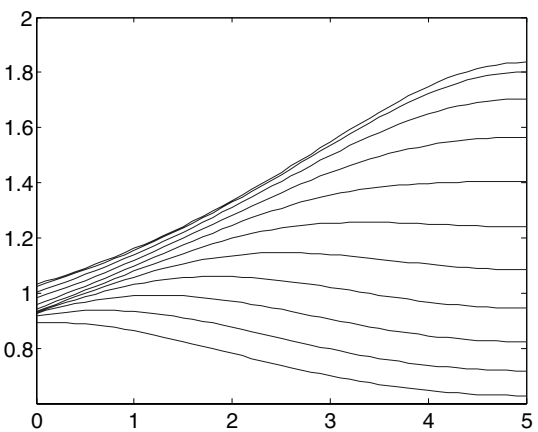
Assume that we put one bulb at the center of the room and place the other two symmetrically as before. Here we show the illumination of the floor when the off-center bulbs are 1 m from the short walls.

```
[X,Y] = meshgrid(0:0.1:10, 0:0.1:4);
contourf(light3(X, Y, 1) + light3(X, Y, 5) + ...
    light3(X, Y, 9), 20);
colormap('gray'); axis equal tight
```



It appears that we should put the bulbs even closer to the walls. (This might not be aesthetically pleasing for everyone!) Let d be the distance of the bulbs from the short walls. We define a function giving the intensity at position x along a long wall and then graph the intensity as a function of d for several values of x.

```
d = 0:0.1:5;
for x = 0:0.5:5
    plot(d, light3(x, 0, d) + light3(x, 0, 5) + ...
        light3(x, 0, 10 - d))
    hold on
end; hold off
```



We know that, for d near 5, the intensity will increase as x increases from 0 to 5, so the bottom curve corresponds to $x = 0$ and the top curve to $x = 5$. Notice that the $x = 0$

curve is the lowest one for all d, and it rises as d decreases. Thus $d = 0$ maximizes the intensity of the darkest spots in the room, which are the corners (corresponding to $x = 0$). There the intensity is as follows.

```
light3(0, 0, 0) + light3(0, 0, 5) + light3(0, 0, 10)
```

```
ans =

    0.8920
```

This is surprising; we do worse than with two bulbs. In going from two bulbs to three, with a decrease in wattage per bulb, we are forced to move wattage away from the ends of the room and bring it back to the center. We could probably improve on the two-bulb scenario if we used brighter bulbs at the ends of the room and a dimmer bulb in the center, or if we used four 75-watt bulbs. But our results so far indicate that the amount to be gained in going to more than two bulbs is likely to be small compared with the amount we gained by going from one bulb to two.

Mortgage Payments

We want to understand the relationships between the mortgage payment of a fixed rate mortgage, the principal (the amount borrowed), the annual interest rate, and the period of the loan. We are going to assume (as is usually the case in the United States) that payments are made monthly, even though the interest rate is given as an annual rate. Let's define

```
peryear = 1/12; percent = 1/100;
```

So the number of payments on a 30-year loan is

```
30*12
```

```
ans =

   360
```

and an annual percentage rate of, say, 8% comes out to a monthly rate of

```
8*percent*peryear
```

```
ans =

    0.0067
```

Now consider what happens with each monthly payment. Some of the payment is applied to interest on the outstanding principal amount, P, and some of the payment is applied to reduce the principal owed. The total amount, R, of the monthly payment, remains constant over the life of the loan. So if J denotes the monthly interest rate, then the amount applied to reduce the principal is $R - JP$, and the new principal after the payment is applied is

$$P - R + JP = P(1 + J) - R = Pm - R,$$

where $m = 1 + J$. So a table of the amount of the principal still outstanding after n payments is tabulated as follows for a loan of initial amount A, for n from 0 to 6:

```
syms m J P R A n
disp('No. of Payments    Remaining Principal');
P = A;
for n = 0:6
    disp([num2str(n), '                    ', char(P)])
    P = simplify(-R + P*m);
end
```

```
No. of Payments    Remaining Principal
0                  A
1                  -R+A*m
2                  -R-m*R+A*m^2
3                  -R-m*R-m^2*R+A*m^3
4                  -R-m*R-m^2*R-m^3*R+A*m^4
5                  -R-m*R-m^2*R-m^3*R-m^4*R+A*m^5
6                  -R-m*R-m^2*R-m^3*R-m^4*R-m^5*R+A*m^6
```

We can write this in a simpler way by noticing that

$$P = Am^n + (\text{terms divisible by } R)$$

For example, with $n = 7$ we have:

```
factor(P - A*m^7)
```

```
ans =

-R*(1+m+m^2+m^3+m^4+m^5+m^6)
```

But on the other hand the quantity inside the parentheses is the sum of a geometric series

$$\sum_{k=1}^{n-1} m^k = \frac{m^n - 1}{m - 1}.$$

So we see that the principal after n payments can be written as

$$P = Am^n - R(m^n - 1)/(m - 1).$$

Now we can solve for the monthly payment amount R under the assumption that the loan is paid off in N installments, i.e., P is reduced to 0 after N payments:

```
syms N;
solve(A*m^N - R*(m^N - 1)/(m - 1), R)
R = subs(ans, m, J + 1)

ans =

A*m^N*(m-1)/(m^N-1)



R =

A*(J+1)^N*J/((J+1)^N-1)
```

For example, with an initial loan amount $A = \$150{,}000$ and a loan lifetime of 30 years (360 payments), we get the following table of payment amounts as a function of annual interest rate:

```
format bank;
disp('      Rate (%)    Monthly Payment ($)');
for rate = 1:10,
  disp([rate,double(subs(R, {A, N, J}, ...
       {150000, 360, rate*percent*peryear}))])
end

      Rate (%)    Monthly Payment ($)
          1.00        482.46

          2.00        554.43

          3.00        632.41

          4.00        716.12
```

```
     5.00        805.23

     6.00        899.33

     7.00        997.95

     8.00       1100.65

     9.00       1206.93

    10.00       1316.36
```

Note the use of **`format bank`** to write the floating-point numbers with two digits after the decimal point.

There's another way to understand these calculations that's a little slicker, and that uses MATLAB's linear-algebra capability. Namely, we can write the fundamental equation

$$P_{\text{new}} = P_{\text{old}} m - R$$

in matrix form as

$$v_{\text{new}} = B v_{\text{old}},$$

where

$$v = \begin{pmatrix} P \\ 1 \end{pmatrix}$$

and

$$B = \begin{pmatrix} m & -R \\ 0 & 1 \end{pmatrix}.$$

We can check this using matrix multiplication:

```
syms R P; B = [m -R; 0 1]; v = [P; 1]; B*v

ans =

 m*P-R
     1
```

which agrees with the formula we had above. Thus the column vector **[P; 1]** resulting after n payments can be computed by left-multiplying the starting vector **[A; 1]** by the matrix B^n. Assuming that $m > 1$, that is, a positive rate of interest, the calculation

```
[eigenvectors, diagonalform] = eig(B)
```

```
eigenvectors =

[        1,        1]
[        0, (m-1)/R]



diagonalform =

[ m, 0]
[ 0, 1]
```

shows us that the matrix B has eigenvalues m, 1, and corresponding eigenvectors

$$\begin{pmatrix}1\\0\end{pmatrix} \text{ and } \begin{pmatrix}1\\(m-1)/R\end{pmatrix} = \begin{pmatrix}1\\J/R\end{pmatrix}.$$

Now we can write the vector **[A; 1]** as a linear combination of the eigenvectors:

$$\begin{pmatrix}A\\1\end{pmatrix} = x\begin{pmatrix}1\\0\end{pmatrix} + y\begin{pmatrix}1\\J/R\end{pmatrix}.$$

We can solve for the coefficients:

```
[x, y] = solve('A = x*1 + y*1', '1 = x*0 + y*J/R')
```

```
x =

(A*J-R)/J



y =

1/J*R
```

and so

$$\begin{pmatrix} A \\ 1 \end{pmatrix} = (A - (R/J)) \begin{pmatrix} 1 \\ 0 \end{pmatrix} + (R/J) \begin{pmatrix} 1 \\ J/R \end{pmatrix}.$$

and

$$B^n \begin{pmatrix} A \\ 1 \end{pmatrix} = (A - (R/J))m^n \begin{pmatrix} 1 \\ 0 \end{pmatrix} + (R/J) \begin{pmatrix} 1 \\ J/R \end{pmatrix}.$$

So the principal remaining after n payments is:

$$P = ((AJ - R)m^n + R)/J = Am^n - R(m^n - 1)/J.$$

This is the same result that we obtained earlier. To conclude, let's determine the amount of money A one can afford to borrow as a function of what one can afford to pay as the monthly payment R. We simply solve for A in the equation that $P = 0$ after N payments.

```
solve(A*m^N - R*(m^N - 1)/(m - 1), A)
```

```
ans =

R*(m^N-1)/(m^N)/(m-1)
```

For example, if one is shopping for a house and can afford to pay $1500 per month for a 30-year fixed-rate mortgage, the maximal loan amount as a function of the interest rate is given by

```
disp('      Rate (%)    Maximal Loan ($)');
for rate = 1:10,
  disp([rate,double(subs(ans, {R, N, m}, ...
       {1500, 360, 1 + rate*percent*peryear}))])
end
```

```
      Rate (%)    Maximal Loan ($)
          1.00     466360.60

          2.00     405822.77

          3.00     355784.07

          4.00     314191.86
```

```
    5.00     279422.43

    6.00     250187.42

    7.00     225461.35

    8.00     204425.24

    9.00     186422.80

   10.00     170926.23
```

Monte Carlo Simulation

In order to make statistical predictions about the long-term results of a random process, it is often useful to do a simulation based on one's understanding of the underlying probabilities. This procedure is referred to as the Monte Carlo method.

As an example, consider a casino game in which a player bets against the casino, and the casino wins 51% of the time. The question is as follows: how many games have to be played before the casino is reasonably sure of coming out ahead? This scenario is common enough that mathematicians long ago figured out very precisely what the statistics are, but here we want to illustrate how to get a good idea of what can happen in practice without having to absorb a lot of mathematics.

First we construct an expression that computes the net revenue to the casino for a single game, based on a random number chosen between 0 and 1 by the MATLAB function **rand**. If the random number is less than or equal to 0.51, the casino wins one betting unit, whereas if the number exceeds 0.51, the casino loses one unit. (In a high-stakes game, each bet may be worth $1000 or more. Thus it is important for the casino to know how bad a losing streak it may have to weather in order to turn a profit so that it doesn't go bankrupt first!) Here is an expression that returns 1 if the output of **rand** is less than 0.51 and -1 if the output of **rand** is greater than 0.51 (it will also return 0 if the output of **rand** is exactly 0.51, but this is extremely unlikely).

```
revenue = sign(0.51 - rand)

revenue =

    -1
```

To simulate several games at once, say ten games, we can generate a vector of ten random numbers with the command **rand(1,10)** and then apply the same operation.

```
revenues = sign(0.51 - rand(1, 10))

revenues =

     1    -1     1    -1    -1     1     1    -1     1    -1
```

Each 1 represents a game that the casino won, and each -1 represents a game that it lost. For a larger number of games, say 100, we can let MATLAB sum the revenue from the individual bets as follows:

```
profit = sum(sign(0.51 - rand(1, 100)))

profit =

    -4
```

The output represents the net profit (or loss, if negative) for the casino after 100 games. On average, every 100 games the casino should win 51 times and the player(s) should win 49 times, so the casino should make a profit of 2 units (on average). Let's see what happens in a few trial runs.

```
profits = sum(sign(0.51 - rand(100, 10)))

profits =

    14   -12     6     2    -4     0   -10    12     0    12
```

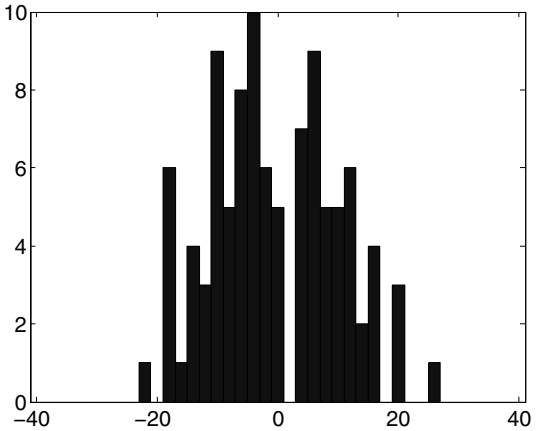
We see that the net profit can fluctuate significantly from one set of 100 games to the next, and there is a sizable probability that the casino has lost money after 100 games. To get an idea of how the net profit is likely to be distributed in general, we can repeat the experiment a large number of times and make a histogram of the results. The following function computes the net profits for k different trials of n games each.

```
profits = @(n,k) sum(sign(0.51 - rand(n, k)))

profits =

    @(n,k) sum(sign(0.51 - rand(n, k)))
```

What this function does is to generate an $n \times k$ matrix of random numbers and then perform the same operations as above on each entry of the matrix to obtain a matrix with entries 1 for bets the casino won and -1 for bets it lost. Finally it sums the columns of the matrix to obtain a row vector of k elements, each of which represents the total profit from a column of **n** bets.

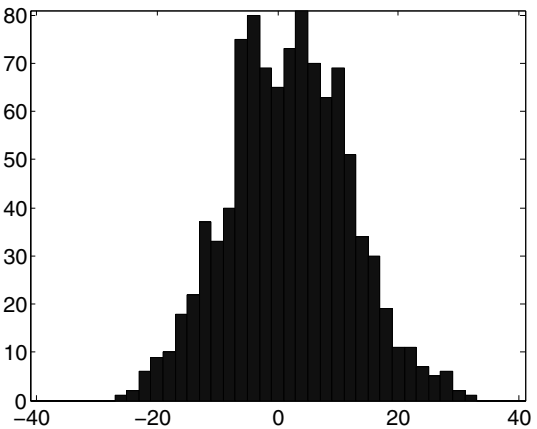
Now we make a histogram of the output of **profits** using $n = 100$ and $k = 100$. Theoretically the casino could win or lose up to 100 units, but in practice we find that the outcomes are almost always within 30 or so of 0. Thus we let the bins of the histogram range from -40 to 40 in increments of 2 (since the net profit is always even after 100 bets).

```
hist(profits(100, 100), -40:2:40); axis tight
```



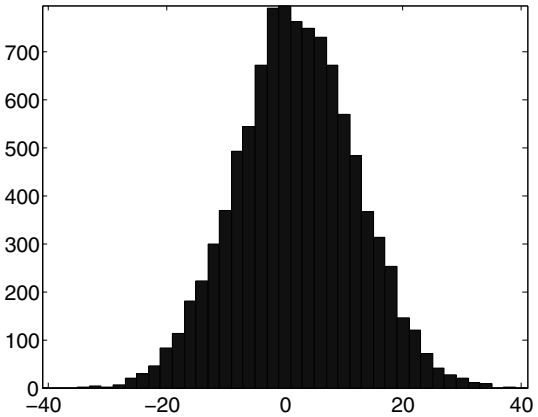
The histogram confirms our impression that there is a wide variation in the outcomes after 100 games. It looks like the casino is about as likely to have lost money as to have profited. However, the distribution shown above is irregular enough to indicate that we really should run more trials to see a better approximation to the actual distribution. Let's try 1000 trials.

```
hist(profits(100, 1000), -40:2:40); axis tight
```



According to the "Central Limit Theorem," when both n and k are large, the histogram should be shaped like a "bell curve," and we begin to see this shape emerging above. Let's move on to $10{,}000$ trials.

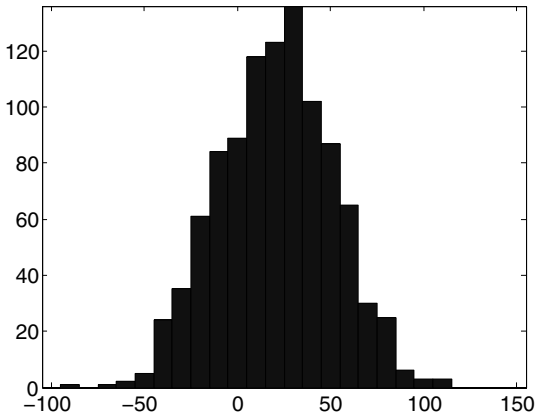
```
hist(profits(100, 10000), -40:2:40); axis tight
```



Here we see very clearly the shape of a bell curve. Though we haven't gained that much in terms of knowing how likely the casino is to be behind after 100 games, and how large its net loss is likely to be in that case, we do gain confidence that our results after 1000 trials are a good depiction of the distribution of possible outcomes.

Now we consider the net profit after 1000 games. We expect on average the casino to win 510 games and the player(s) to win 490, for a net profit of 20 units. Let's start with 1000 trials.

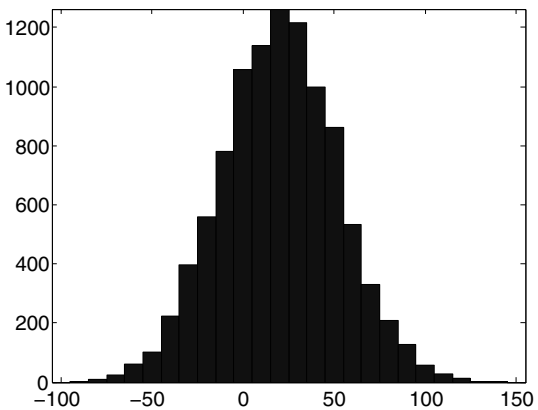
```
hist(profits(1000, 1000), -100:10:150); axis tight
```



Though the range of values we observe for the profit after 1000 games is larger than the range for 100 games, the range of possible values is ten times as large, so that relatively speaking the outcomes are closer together than before. This reflects the theoretical principle (also a consequence of the Central Limit Theorem) that the average "spread" of outcomes after a large number of trials should be proportional to the square root of the number n of games played in each trial. This is important for the casino, since if the spread were proportional to n, then the casino could never be too sure of making a profit. When we increase n by a factor of 10, the spread should only increase by a factor of $\sqrt{10}$, or a little more than 3.

Notice that, after 1000 games, the casino is definitely more likely to be ahead than behind. However, the chances of being behind still look sizable. Let's repeat the simulation with 10,000 trials to be more certain of our results. We might be tempted to type **hist(profits(1000, 10000), -100:10:150)**, but notice that this involves an array of 10 million numbers. While most computers can now store this many numbers in memory, using this much memory can slow MATLAB down. Generally we find that it is best not to go too far over a million numbers in an array when possible, and on our computers it is quicker in this instance to perform the 10,000 trials in batches of 1000, using a loop to assemble the results into a single vector.

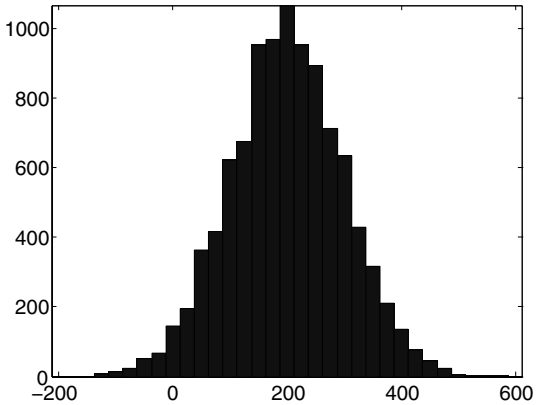
```
profitvec = [];
for j = 1:10
   profitvec = [profitvec profits(1000, 1000)];
end
hist(profitvec, -100:10:150); axis tight
```



We see the bell-curve shape emerging again. Though it is unlikely, the chance that the casino is behind by more than 50 units after 1000 games is not insignificant. If each unit is worth $1000, then we might advise the casino to have at least $100,000 cash on hand in order to be prepared for this possibility. Maybe even that is not enough – to see we have to experiment further.

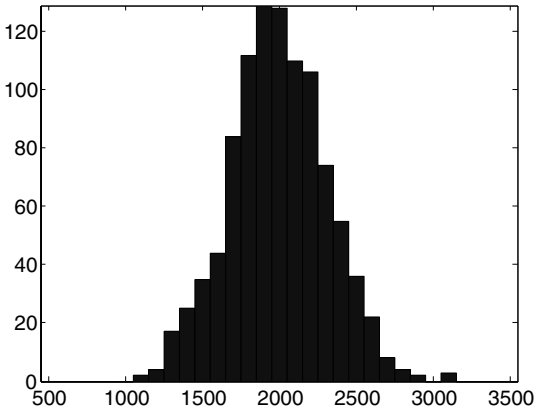
Let's see now what happens after 10,000 games. We expect on average the casino to be ahead by 200 units at this point, and, on the basis of our earlier discussion, the range of values we use to make the histogram need go up only by a factor of three or so from the previous case. Let's go straight to 10,000 trials. This time we do 100 batches of 100 trials each.

```
profitvec = [];
for j = 1:100
   profitvec = [profitvec profits(10000, 100)];
end
hist(profitvec, -200:25:600); axis tight
```



It seems that turning a profit after 10,000 games is highly likely. But though the chance of a loss is quite small at this point, it is not negligible; more than 1% of the trials resulted in a loss, and sometimes the loss was more than 100 units. However, the overall trend toward profitability seems clear, and we expect that after 100,000 games the casino is overwhelmingly likely to have made a profit. On the basis of our previous observations of the growth of the spread of outcomes, we expect that most of the time the net profit will be within 1000 of the expected value of 2000. Since 10,000 trials of 10,000 games each took a while to run, we'll do only 1000 trials this time.

```
profitvec = [];
for j = 1:100
   profitvec = [profitvec profits(100000, 10)];
end
hist(profitvec, 500:100:3500); axis tight
```



The results are consistent with our projections. The casino seems almost certain to have made a profit after 100,000 games, but it should have reserves of several hundred betting units on hand in order to cover the possible losses along the way.

Population Dynamics

We are going to look at two models for population growth of a species. The first is a standard exponential growth/decay model that describes quite well the population of a species becoming extinct, or the short-term behavior of a population growing in an unchecked fashion. The second, more realistic, model describes the growth of a species subject to constraints of space, food supply, and competitors/predators.

Exponential Growth/Decay

We assume that the species starts with an initial population P_0. The population after n time units is denoted P_n. Suppose that, in each time interval, the population increases or decreases by a fixed proportion of its value at the beginning of the interval. Thus

$$P_{n+1} = P_n + rP_n, \qquad n \geq 0.$$

The constant r represents the difference between the birth rate and the death rate. The population increases if r is positive, decreases if r is negative, and remains fixed if $r = 0$.

Here is a simple M-file that we will use to compute the population at stage n, given the population at the previous stage and the rate r.

```
type itseq
```

```
function X = itseq(f, Xinit, n, r)
% Computes an iterative sequence of values.
X = zeros(n + 1, 1);
X(1) = Xinit;
for k = 1:n
     X(k + 1) = f(X(k), r);
end
```

In fact, this is a simple program for computing iteratively the values of a sequence $x_{k+1} = f(x_k, r), n \geq 0$, given any function f, the value of its parameter r, and the initial value x_0 of the sequence.

Now let's use the program to compute two populations at 5-year intervals for $r = 0.1$ and then $r = -0.1$:

```
Xinit = 100; f = @(x, r) x*(1 + r);
X = itseq(f, Xinit, 100, 0.1);
format long; X(1:5:101)
```

```
ans =

   1.0e+06 *

   0.00010000000000
   0.00016105100000
   0.00025937424601
   0.00041772481694
   0.00067274999493
   0.00108347059434
   0.00174494022689
   0.00281024368481
   0.00452592555682
   0.00728904836851
   0.01173908528797
   0.01890591424713
   0.03044816395414
   0.04903707252979
   0.07897469567994
   0.12718953713951
   0.20484002145855
   0.32989690295921
   0.53130226118483
   0.85566760466078
   1.37806123398224
```

```
X = itseq(f, Xinit, 100, -0.1); X(1:5:101)
```

```
ans =

   1.0e+02 *

   1.00000000000000
   0.59049000000000
   0.34867844010000
   0.20589113209465
   0.12157665459057
   0.07178979876919
   0.04239115827522
   0.02503155504993
   0.01478088294143
   0.00872796356809
   0.00515377520732
   0.00304325272217
   0.00179701029991
   0.00106111661200
   0.00062657874822
   0.00036998848504
   0.00021847450053
```

```
0.00012900700782
0.00007617734805
0.00004498196225
0.00002656139889
```

In the first case, the population is growing rapidly; in the second, decaying rapidly. In fact, it is clear from the model that, for any n, the quotient $P_n/P_{n+1} = (1+r)$, and therefore it follows that $P_n = P_0(1+r)^n, n \geq 0$. This accounts for the expression "exponential growth/decay." The model predicts a population growth without bound (for growing populations), and is therefore not realistic. Our next model allows for a check on the population caused by limited space, limited food supply, competitors, and predators.

Logistic Growth

The previous model assumes that the relative change in population is constant, that is

$$(P_{n+1} - P_n)/P_n = r.$$

Now let's build in a term that holds down the growth, namely

$$(P_{n+1} - P_n)/P_n = r - uP_n.$$

We shall simplify matters by assuming that $u = 1 + r$, so that our recursion relation becomes

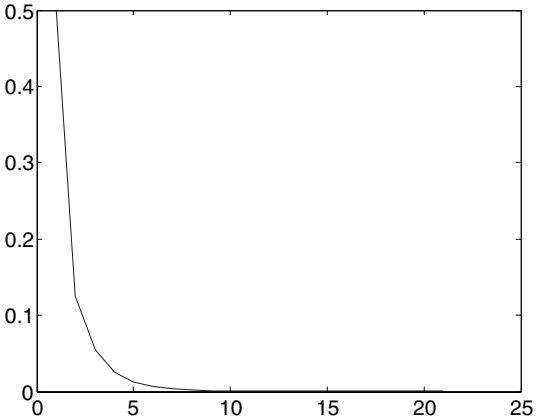
$$P_{n+1} = uP_n(1 - P_n),$$

where u is a positive constant. In this model, the population P is constrained to lie between 0 and 1, and should be interpreted as a percentage of a maximum possible population in the environment in question. So let us define the function we will use in the iterative procedure:

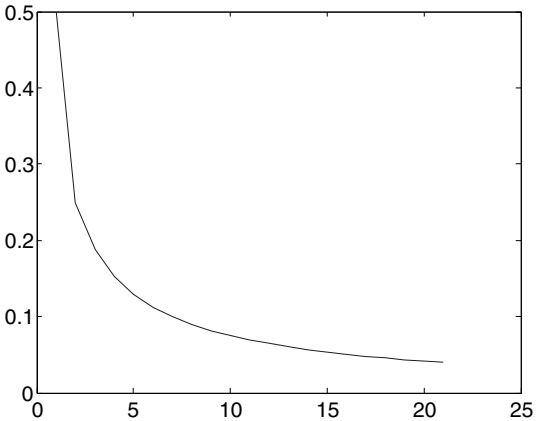
```
f = @(x, u) u*x*(1 - x);
```

Now let's compute a few examples, and use **plot** to display the results.

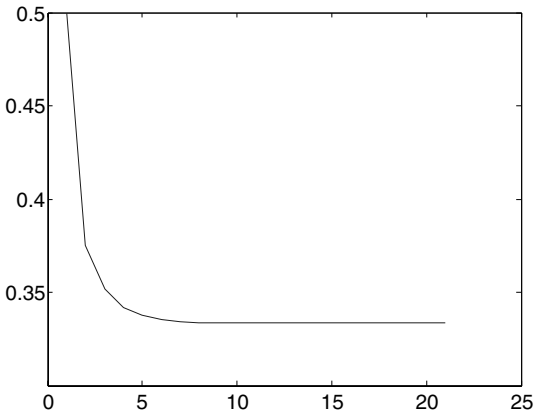
```
Xinit = 0.5; X = itseq(f, Xinit, 20, 0.5); plot(X)
```



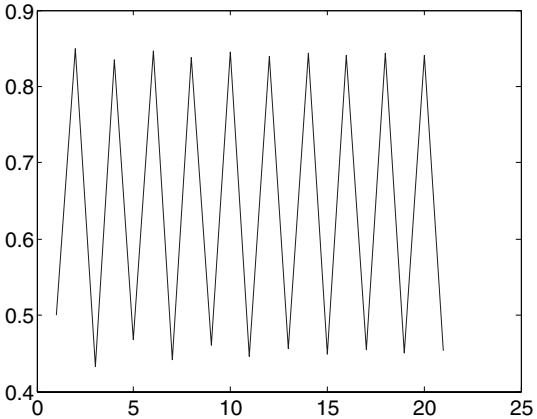

```
X = itseq(f, Xinit, 20, 1); plot(X)
```




```
X = itseq(f, Xinit, 20, 1.5); plot(X)
```




```
X = itseq(f, Xinit, 20, 3.4); plot(X)
```



In the first computation, we have used our iterative program to compute the population density for 20 time intervals, assuming a logistic growth constant $u = 0.5$, and an initial population density of 50%. The population seems to be dying out. In the remaining examples, we kept the initial population density at 50%; the only thing we varied was the logistic growth constant. In the second example, with a growth constant $u = 1$, once again the population is dying out – although more slowly. In the third example, with a growth constant of 1.5 the population seems to be stabilizing at 33.3...%. Finally, in the last example, with a constant of 3.4 the population seems to oscillate between densities of approximately 45% and 84%.

These examples illustrate the remarkable features of the logistic population dynamics model. This model has been studied for more than 150 years, its origins lying in an analysis by the Belgian mathematician Verhulst. Here are some of the facts associated with this model. We will corroborate some of them with MATLAB. In particular, we shall use **`bar`** as well as **`plot`** to display some of the data.

- **The logistic constant cannot be larger than 4.**

In order for the model to work, the output at any point must be between 0 and 1. But the parabola $ux(1-x)$, for $0 \leq x \leq 1$, has its maximum height when $x = 1/2$, where its value is $u/4$. To keep that number between 0 and 1, we must restrict $u \leq 4$. Here is what happens if u is greater than 4:

```
X = itseq(f, 0.5, 10, 4.5)

X =

  1.0e+173 *

   0.00000000000000
   0.00000000000000
  -0.00000000000000
  -0.00000000000000
  -0.00000000000000
```

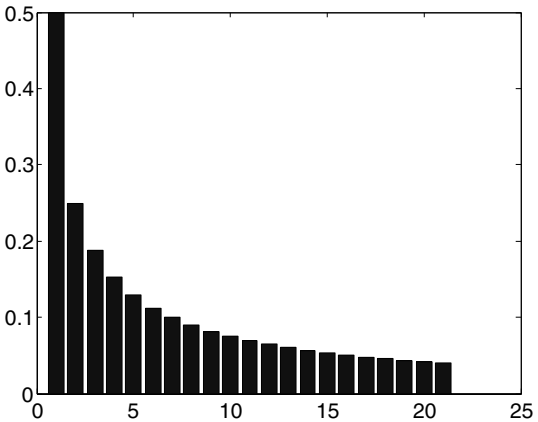
```
-0.00000000000000
-0.00000000000000
-0.00000000000000
-0.00000000000000
-0.00000000000000
-8.31506542713596
```

- **If $0 \leq u \leq 1$, the population density tends to zero for any initial value.**

```
X = itseq(f, 0.5, 100, 0.8); X(101)
```

```
ans =

     2.420473970178059e-11
```

```
X = itseq(f, 0.5, 20, 1); bar(X)
```



- **If $1 < u \leq 3$, the population will stabilize at density $1 - 1/u$ for any initial density other than zero.**

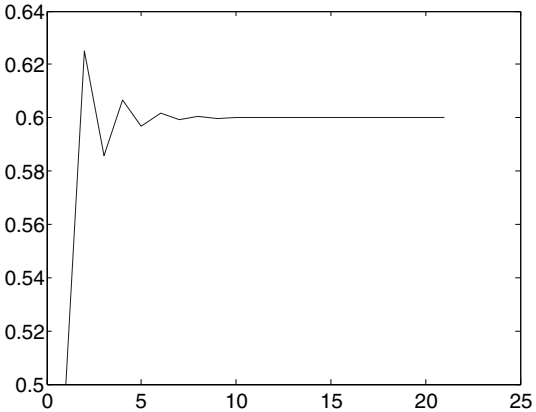
The third of the original four examples corroborates the assertion (with $u = 1.5$ and $1 - 1/u = 1/3$). In the following examples, we set $u = 2$, 2.5, and 3, respectively, so that $1 - 1/u$ equals 0.5, 0.6, and $0.666\ldots$, respectively. The convergence in the last computation is rather slow (as one might expect from a boundary case – or "bifurcation point").

```
X = itseq(f, 0.25, 100, 2); X(101)
```

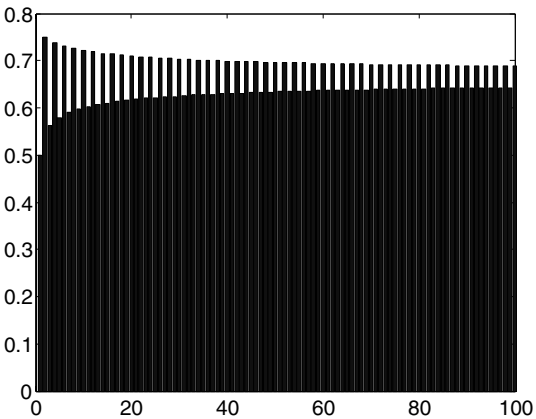
```
ans =

   0.50000000000000
```

```
X = itseq(f, 0.75, 100, 2); X(101)

ans =

   0.50000000000000
```

```
X = itseq(f, 0.5, 20, 2.5);
plot(X)
```




```
X = itseq(f, 0.5, 100, 3);
bar(X); axis([0 100 0 0.8])
```



- **If $3 < u < 3.56994\ldots$, then there is a periodic cycle.**

The theory is quite subtle. For a fuller explanation, the reader may consult **Encounters with Chaos**, by Denny Gulick, McGraw-Hill, New York, 1992, Section 1.5. In fact there is a sequence

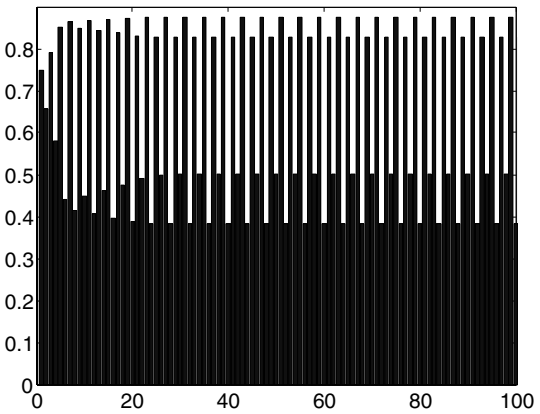
$$u_0 = 3 < u_1 = 1 + \sqrt{6} < u_2 < u_3 < \cdots < 4,$$

such that between u_0 and u_1 there is a cycle of period 2; between u_1 and u_2 there is cycle of period 4; and in general, between u_k and u_{k+1} there is a cycle of period 2^{k+1}. In fact one knows that, at least for small k, one has the approximation $u_{k+1} \approx 1 + \sqrt{3 + u_k}$. So:

```
u1 = 1 + sqrt(6)

u1 =

   3.44948974278318


u2approx = 1 + sqrt(3 + u1)

u2approx =

   3.53958456106175
```

This explains the oscillatory behavior we saw in the last of the original four examples (with $u_0 < u = 3.4 < u_1$). Here is the behavior for $u_1 < u = 3.5 < u_2$. The command **bar** is particularly effective here for spotting the cycle of order 4.

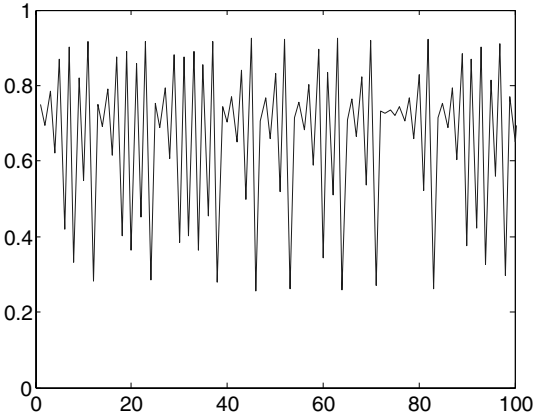
```
X = itseq(f, 0.75, 100, 3.5);
bar(X); axis([0 100 0 0.9])
```



- **There is a value $u < 4$ beyond which – chaos!**

It is possible to prove that the sequence u_k tends to a limit u_∞. The value of u_∞, sometimes called the "Feigenbaum parameter," is approximately $3.56994\ldots$. Let's see what happens if we use a value of u between the Feigenbaum parameter and 4.

```
X = itseq(f, 0.75, 100, 3.7);
plot(X); axis([0 100 0 1])
```

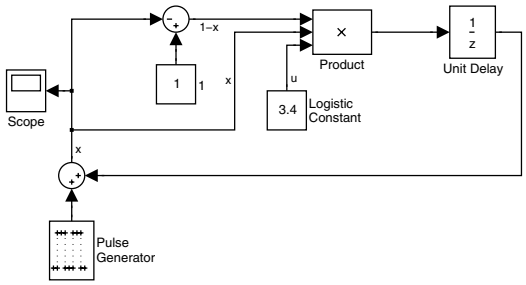


This is an example of what mathematicians call a "chaotic" phenomenon! It is not random – the sequence was generated by a precise, fixed mathematical procedure, but the results manifest no discernible pattern. Chaotic phenomena are unpredictable, but with modern methods (including computer analysis), mathematicians have been able to identify certain patterns of behavior in chaotic phenomena. For example, the last figure suggests the possibility of unstable periodic cycles and other recurring phenomena. Indeed a great deal of information is known. The aforementioned book by Gulick is a fine reference, as well as the source of an excellent bibliography on the subject.

Re-running the Model with Simulink

The logistic growth model that we have been exploring lends itself particularly well to simulation using Simulink. Here is a simple Simulink model that corresponds to the above calculations:

```
open_system popdyn
```

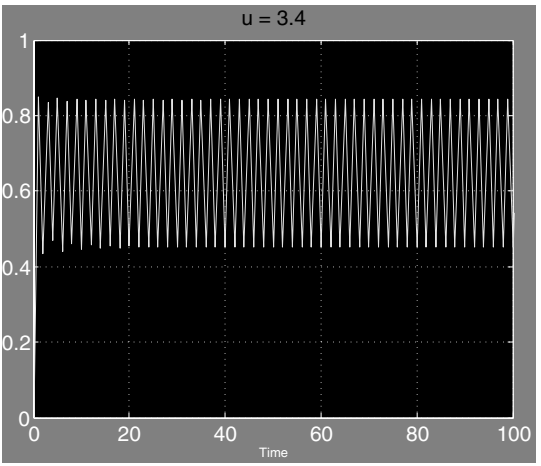


Let's briefly explain how this works. If you ignore the Pulse Generator block and the Sum block in the lower left for a moment, this model implements the equation

$$x \text{ at next time} = ux(1-x) \text{ at current time},$$

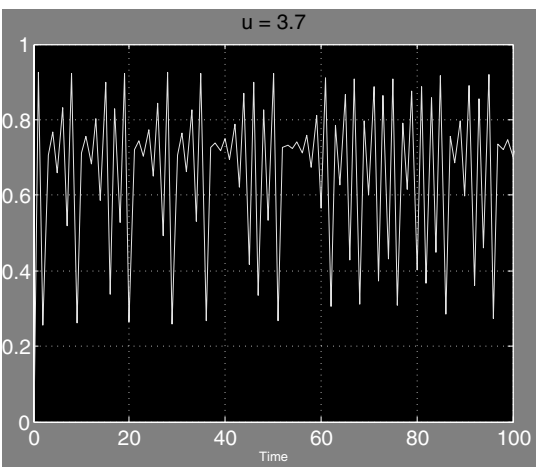
which is the equation for the logistic model. The Scope block displays a plot of x as a function of (discrete) time. However, we need somehow to build in the initial condition for x. The simplest way to do this is as illustrated here: we add to the right-hand side a discrete pulse that is the initial value of x (here we use 0.5) at time $t = 0$ and is 0 thereafter. Since the model is discrete, you can achieve this by setting the Pulse Generator block to "Sample based" mode, setting the period of the pulse to something longer than the length of the simulation, setting the width of the pulse to 1, and setting the amplitude of the pulse to the initial value of x. The outputs from the model in the two interesting cases of $u = 3.4$ and $u = 3.7$ are shown here:

```
[t, x] = sim('popdyn', [0 120]);
simplot(t, x); title('u = 3.4')
```



In the first case of $u = 3.4$, the periodic behavior is clearly visible.

```
set_param('popdyn/Logistic Constant', 'Value', '3.7')
[t, x] = sim('popdyn', [0 120]);
simplot(t, x); title('u = 3.7')
```



On the other hand, when $u = 3.7$, we get chaotic behavior.

Linear Economic Models

MATLAB's linear algebra capabilities make it a good vehicle for studying linear economic models, sometimes called "Leontief models" (after their primary developer, Nobel Prize-winning economist Wassily Leontief) or "input-output models." We will give a few examples. The simplest such model is the "linear exchange model" or "closed Leontief model" of an economy. This model supposes that an economy is divided into, say, n sectors, such as agriculture, manufacturing, service, consumers, etc. Each sector receives input from the various sectors (including itself) and produces an output, which is divided among the various sectors. (For example, agriculture produces food for home consumption and for export, but also seeds and new livestock which are reinvested in the agricultural sector, as well as chemicals that may be used by the manufacturing sector, and so on.) The meaning of a closed model is that total production is equal to total consumption. The economy is in equilibrium when each sector of the economy (at least) breaks even. For this to happen, the prices of the various outputs have to be adjusted by market forces. Let a_{ij} denote the fraction of the output of the jth sector consumed by the ith sector. Then the a_{ij} are the entries of a square matrix, called the "exchange matrix" A, each of whose columns sums to 1. Let p_i be the price of the output of the ith sector of the economy. Since each sector is to break even, p_i cannot be smaller than the value of the inputs consumed by the ith sector, or in other words,

$$p_i \geq \sum_j a_{ij} p_j.$$

But, on summing over i and using the fact that

$$\sum_i a_{ij} = 1,$$

we see that the two sides must be equal. In matrix language, that means that $(I - A)p = 0$, where p is the column vector of prices. Thus p is an eigenvector of A for the eigenvalue 1, and the theory of stochastic matrices implies (assuming that A is "irreducible," meaning that there is no proper subset E of the sectors of the economy such that outputs from E all stay inside E) that p is uniquely determined up to a scalar factor. In other words, a closed irreducible linear economy has an essentially unique equilibrium state. For example, if we have

```
A = [0.3, 0.1, 0.05, 0.2; 0.1, 0.2, 0.3, 0.3; ...
    0.3, 0.5, 0.2, 0.3; 0.3, 0.2, 0.45, 0.2]
```

```
A =

    0.3000    0.1000    0.0500    0.2000
    0.1000    0.2000    0.3000    0.3000
    0.3000    0.5000    0.2000    0.3000
    0.3000    0.2000    0.4500    0.2000
```

then, as required,

```
sum(A)

ans =

     1     1     1     1
```

that is, all the columns sum to 1, and

```
[V, D] = eig(A); D(1, 1)
p = V(:, 1)

ans =

    1.0000


p =

    0.2739
    0.4768
    0.6133
    0.5669
```

shows that 1 is an eigenvalue of A with price eigenvector p as shown.

Somewhat more realistic is the (static, linear) open Leontief model of an economy, which takes labor, consumption, etc., into account. Let's illustrate with an example. The following command inputs an actual input-output transactions table for the economy of the United Kingdom in 1963. (This table is taken from **Input-Output Analysis and its Applications** by R. O'Connor and E. W. Henry, Hafner Press, New York, 1975.) Tables such as this one can be obtained from official government statistics. The table **T** is a 10×9 matrix. Units are millions of British pounds. The rows represent, respectively, agriculture, industry, services, total inter-industry, imports, sales by final buyers, indirect taxes, wages and profits, total primary inputs, and total inputs. The columns represent, respectively, agriculture, industry, services, total inter-industry, consumption, capital formation, exports, total final demand, and output. Thus outputs from each sector can be read off along a row, and inputs into a sector can be read off along a column.

```
T = [ 277   444    14   735  1123    35    51  1209  1944; ...
      587 11148  1884 13619  8174  4497  3934 16605 30224; ...
      236  2915  1572  4723 11657   430  1452 13539 18262; ...
     1100 14507  3470 19077 20954  4962  5437 31353 50430; ...
      133  2844   676  3653  1770   250   273  2293  5946; ...
        3   134    42   179   -90  -177    88  -179     0; ...
     -246   499   442   695  2675   100    17  2792  3487; ...
      954 12240 13632 26826     0     0     0     0 26826; ...
      844 15717 14792 31353  4355   173   378  4906 36259; ...
     1944 30224 18262 50430 25309  5135  5815 36259 86689];
```

A few features of this matrix are apparent from the following:

```
T(4, :) - T(1, :) - T(2, :) - T(3, :)
T(9, :) - T(5, :) - T(6, :) - T(7, :) - T(8, :)
T(10, :) - T(4, :) - T(9, :)
T(10, 1:4) - T(1:4, 9)'

ans =

     0     0     0     0     0     0     0     0     0


ans =

     0     0     0     0     0     0     0     0     0


ans =

     0     0     0     0     0     0     0     0     0


ans =

     0     0     0     0
```

Thus the fourth row, which summarizes inter-industry inputs, is the sum of the first three rows; the ninth row, which summarizes "primary inputs," is the sum of rows 5 through 8; the tenth row, total inputs, is the sum of rows 4 and 9, and the first four entries of the last row agree with the first four entries of the last column (meaning that all output from the industrial sectors is accounted for). Also we have:

```
(T(:, 4) - T(:, 1) - T(:, 2) - T(:, 3))'
(T(:, 8) - T(:, 5) - T(:, 6) - T(:, 7))'
(T(:, 9) - T(:, 4) - T(:, 8))'

ans =
```

```
     0     0     0     0     0     0     0     0     0     0


ans =

     0     0     0     0     0     0     0     0     0     0


ans =

     0     0     0     0     0     0     0     0     0     0
```

so the fourth column, representing total inter-industry output, is the sum of columns 1 through 3; the eighth column, representing total "final demand," is the sum of columns 5 through 7; and the ninth column, representing total output, is the sum of columns 4 and 8. The matrix A of "inter-industry technical coefficients" is obtained by dividing the columns of T corresponding to industrial sectors (in our case there are three of these) by the corresponding total inputs. Thus we have:

```
A = [T(:, 1)/T(10, 1), T(:, 2)/T(10, 2), T(:, 3)/T(10, 3)]

A =

    0.1425    0.0147    0.0008
    0.3020    0.3688    0.1032
    0.1214    0.0964    0.0861
    0.5658    0.4800    0.1900
    0.0684    0.0941    0.0370
    0.0015    0.0044    0.0023
   -0.1265    0.0165    0.0242
    0.4907    0.4050    0.7465
    0.4342    0.5200    0.8100
    1.0000    1.0000    1.0000
```

Here the square upper block (the first three rows) is most important, so we make the replacement

```
A = A(1:3, :)

A =

    0.1425    0.0147    0.0008
    0.3020    0.3688    0.1032
    0.1214    0.0964    0.0861
```

If the vector Y represents total final demand for the various industrial sectors, and the vector X represents total outputs for these sectors, then the fact that the last column

of T is the sum of columns 4 (total inter-industry outputs) and 8 (total final demand) translates into the matrix equation

$$X = AX + Y, \text{ or } Y = (1 - A)X.$$

Let's check this:

```
Y = T(1:3, 8); X = T(1:3, 9); Y - (eye(3) - A)*X

ans =

     0
     0
     0
```

Now one can do various numerical experiments. For example, what would be the effect on output of an increase of £10 billion (10,000 in the units of our problem) in final demand for industrial output, with no corresponding increase in demand for services or for agricultural products? Since the economy is assumed to be linear, the change ΔX in X is obtained by solving the linear equation

$$\Delta Y = (1 - A)\Delta X,$$

and

```
deltaX = (eye(3) - A) \ [0; 10000; 0]

deltaX =

   1.0e+04 *

    0.0280
    1.6265
    0.1754
```

Thus agricultural output would increase by £280 million, industrial output would increase by £16.265 billion, and service output would increase by £1.754 billion. We can illustrate the result of doing this for similar increases in demand for the other sectors with the following pie charts:

```
deltaX1 = (eye(3) - A) \ [10000; 0; 0];
deltaX2 = (eye(3) - A) \ [0; 0; 10000];
subplot(1, 3, 1), pie(deltaX1, {'Ag.', 'Ind.', 'Serv.'})
subplot(1, 3, 2), pie(deltaX, {'Ag.', 'Ind.', 'Serv.'})
```

```
title(['Effect of increases in demand for each of the ' ...
    '3 sectors'], 'FontSize', 13)
subplot(1, 3, 3), pie(deltaX2, {'Ag.', 'Ind.', 'Serv.'})
colormap(gray)
```

Linear Programming

MATLAB is ideally suited to handle so-called linear programming problems. These are problems in which you have a quantity, depending linearly on several variables, that you want to maximize or minimize subject to several constraints that are expressed as linear inequalities in the same variables. If the number of variables and the number of constraints are small, then there are numerous mathematical techniques for solving a linear programming problem – indeed, these techniques are often taught in high-school or university-level courses in finite mathematics. But sometimes these numbers are high, or, even if they are low, the constants in the linear inequalities or the object expression for the quantity to be optimized may be numerically complicated – in which case a software package like MATLAB is required to effect a solution. We shall illustrate the method of linear programming by means of a simple example, giving a combined graphical/numerical solution, and then solve both a slightly as well as a substantially more complicated problem.

Suppose that a farmer has 75 acres on which to plant two crops: wheat and barley. To produce these crops, it costs the farmer (for seed, fertilizer, etc.) \$120 per acre for the wheat and \$210 per acre for the barley. The farmer has \$15,000 available for expenses. But after the harvest, the farmer must store the crops while awaiting favorable market conditions. The farmer has storage space for 4000 bushels. Each acre yields an average of 110 bushels of wheat or 30 bushels of barley. If the net profit per bushel of wheat (after all expenses have been subtracted) is \$1.30 and for barley is \$2.00, how should the farmer plant the 75 acres to maximize profit?

We begin by formulating the problem mathematically. First we express the objective, that is the profit, and the constraints algebraically, then we graph them, and lastly we arrive at the solution by graphical inspection and a minor arithmetic calculation.

Let x denote the number of acres allotted to wheat and y the number of acres allotted to barley. Then the expression to be maximized, that is the profit, is clearly

$$P = (110)(1.30)x + (30)(2.00)y = 143x + 60y.$$

There are three constraint inequalities, specified by the limits on expenses, storage, and acreage. They are, respectively,

$$120x + 210y \leq 15000,$$

$$110x + 30y \leq 4000,$$

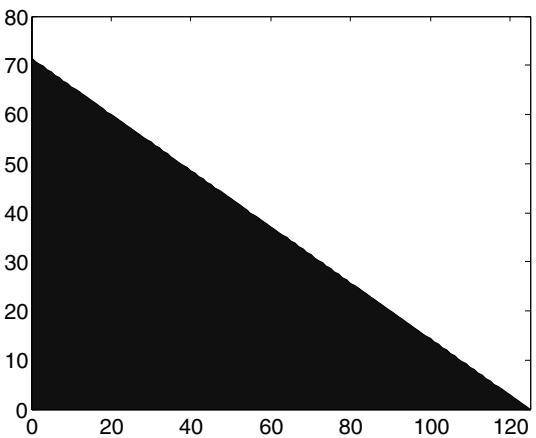
$$x + y \leq 75.$$

Strictly speaking there are two more constraint inequalities forced by the fact that the farmer cannot plant a negative number of acres, namely

$$x \geq 0, \quad y \geq 0.$$

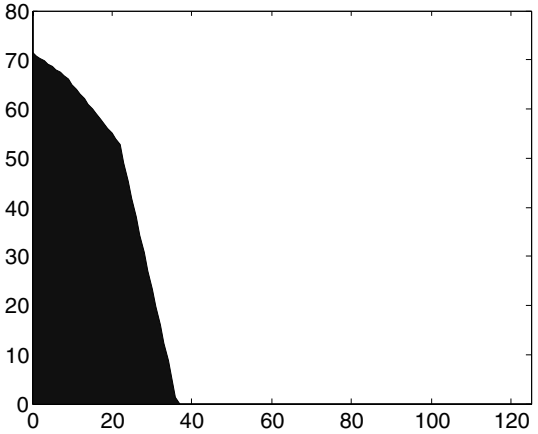
Next we graph the regions specified by the constraints. The last two say that we need to consider only the first quadrant in the x-y plane. Here's a graph delineating the triangular region in the first quadrant determined by the first inequality.

```
X = 0:125;
Y1 = (15000 - 120.*X)./210;
area(X, Y1)
```



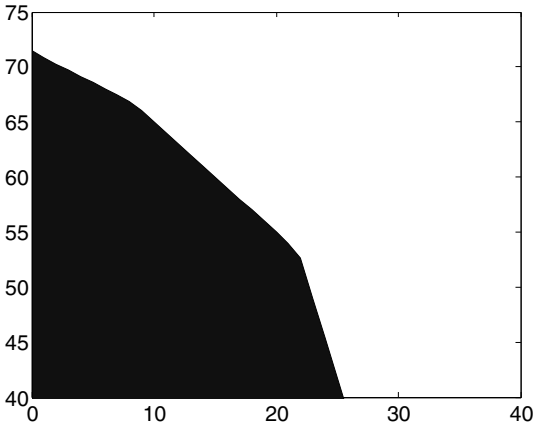
Now let's put in the other two constraint inequalities.

```
Y2 = max((4000 - 110.*X)./30, 0);
Y3 = max(75 - X, 0);
Ytop = min([Y1; Y2; Y3]);
area(X, Ytop)
```



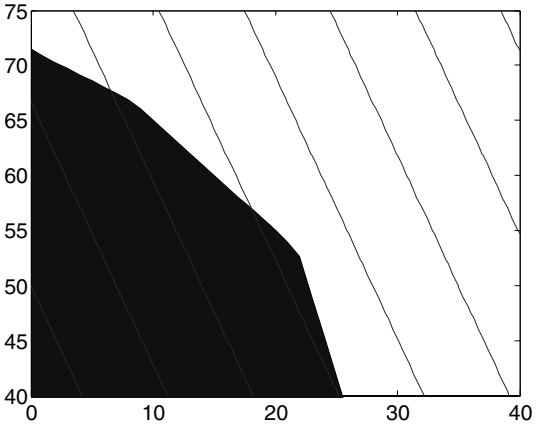
It's a little hard to see the polygonal boundary of the region clearly. Let's home in a bit.

```
area(X, Ytop); axis([0 40 40 75])
```



Now let's superimpose on top of this picture a contour plot of the objective function P.

```
hold on
[U V] = meshgrid(0:40, 40:75);
contour(U, V, 143.*U + 60.*V); hold off
```



It seems apparent that the maximum value of P will occur on the level curve (that is, level line) that passes through the vertex of the polygon that lies near $(22, 53)$. In fact we can compute

```
[x, y] = solve('x + y = 75', '110*x + 30*y = 4000')

x =

175/8


y =

425/8


double([x, y])

ans =

   21.8750   53.1250
```

The acreage that results in the maximum profit is 21.875 for wheat and 53.125 for barley. In that case the profit is

```
P = 143*x + 60*y

P =

50525/8
```

```
format bank; double(P)
```

```
ans =

       6315.62
```

that is, $6,315.63.

This problem illustrates and is governed by the "Fundamental Theorem of Linear Programming," which is stated here for two variables: a linear expression $ax + by$, defined over a closed bounded convex set S whose sides are line segments, takes on its maximum and minimum values at vertices of S. If S is unbounded, there might but need not be an optimum value, but if there is, it occurs at a vertex. (A convex set is one for which any line segment joining two points of the set lies entirely inside the set.)

In fact the Simulink Toolbox has a built-in function, **simlp**, that implements the solution of a linear programming problem. The Optimization Toolbox has an almost identical function called **linprog**. You can learn about either one from the online help. We will use **simlp** on the above problem. After that we will use it to solve two more complicated problems involving more variables and constraints. Here is the beginning of its help text.

```
helptext = help('simlp'); helptext(1:190)
```

```
ans =

 SIMLP Helper function for GETXO; solves linear programming problem.
    X=SIMLP(f,A,b) solves the linear programming problem:

              min f'x    subject to:   Ax <= b
               x
```

So

```
f = [-143 -60];
A = [120 210; 110 30; 1 1; -1 0; 0 -1];
b = [15000; 4000; 75; 0; 0];

format short; simlp(f, A, b)
```

```
ans =

   21.8750
   53.1250
```

This is the same answer that we obtained before. Note that we entered the negative of the coefficients for the objective function P into the vector **f** because **simlp** searches for a minimum rather than a maximum. Note also that the non-negativity constraints are accounted for in the last two rows of **A** and **b**.

Well, we could have done this problem by hand. But suppose that the farmer is dealing with a third crop, say corn, and that the corresponding data are

- Cost per acre: \$150.75,
- Yield per acre: 125 bushels,
- Profit per bushel: \$1.56.

If we denote the number of acres allotted to corn by z, then the objective function becomes

$$P = (110)(1.30)x + (30)(2.00)y + (125)(1.56) = 143x + 60y + 195z,$$

and the constraint inequalities are

$$\begin{gathered} 120x + 210y + 150.75z \leq 15000, \\ 110x + 30y + 125z \leq 4000, \\ x + y + z \leq 75, \\ x \geq 0, \quad y \geq 0, \quad z \geq 0. \end{gathered}$$

The problem is solved with **simlp** as follows:

```
f = [-143 -60 -195];
A = [120 210 150.75; 110 30 125; 1 1 1; -1 0 0; ...
    0 -1 0; 0 0 -1];
b = [15000; 4000; 75; 0; 0; 0];

simlp(f, A, b)

ans =

         0
   56.5789
   18.4211
```

So the farmer should ditch the wheat and plant 56.5789 acres of barley and 18.4211 acres of corn.

There is no practical limit on the number of variables and constraints that MATLAB can handle – certainly none that the relatively unsophisticated user will encounter.

Indeed, in many real applications of the technique of linear programming, one needs to deal with many variables and constraints. The solution of such a problem by hand is not feasible, and software like MATLAB is crucial to success. For example, in the farming problem with which we have been working, one could have more crops than two or three – think agribusiness instead of family farmer. Also one could have constraints that arise from other things beside expenses, storage, and acreage limitations – for example the following.

- Availability of seed. This might lead to constraint inequalities like $x_j \leq k$.
- Personal preferences. Thus the farmer's spouse might have a preference for one variety over another and insist on a corresponding planting, or something similar with a collection of crops; thus constraint inequalities like $x_i \leq x_j$ or $x_1 + x_2 \geq x_3$.
- Government subsidies. It may take a moment's reflection on the reader's part, but this could lead to inequalities like $x_j \geq k$.

Below is a sequence of commands that solves exactly such a problem. You should be able to recognize the objective expression and the constraints from the data that are entered. But as an aid, you might answer the following questions.

- How many crops are under consideration?
- What are the corresponding expenses? How much is available for expenses?
- What are the yields in each case? What is the storage capacity?
- How many acres are available?
- What crops are constrained by seed limitations? To what extent?
- What about preferences?
- What are the minimum acreages for each crop?

```
f = [-110*1.3 -30*2.0 -125*1.56 -75*1.8 -95*.95 ...
    -100*2.25 -50*1.35];
A = [120 210 150.75 115 186 140 85; ...
    110 30 125 75 95 100 50; 1 1 1 1 1 1 1; 1 0 0 0 0 0 0; ...
    0 0 1 0 0 0 0; 0 0 0 0 0 1 0; 1 -1 0 0 0 0 0; ...
    0 0 1 0 -2 0 0; 0 0 0 -1 0 -1 1; -1 0 0 0 0 0 0; ...
    0 -1 0 0 0 0 0; 0 0 -1 0 0 0 0; 0 0 0 -1 0 0 0; ...
    0 0 0 0 -1 0 0; 0 0 0 0 0 -1 0; 0 0 0 0 0 0 -1];
b = [55000; 40000; 400; 100; 50; 250; 0; 0; 0; -10; -10; ...
    -10; -10; -20; -20; -20];
simlp(f, A, b)

ans =

   10.0000
   10.0000
   40.0000
   45.6522
   20.0000
  250.0000
   20.0000
```

Note that, despite the complexity of the problem, MATLAB solves it almost instantaneously. It seems the farmer should bet the farm on crop number 6. We suggest that you alter the expense and/or the storage limit in the problem and see what effect that has on the answer.

The 360° Pendulum

Normally we think of a pendulum as a weight suspended by a flexible string or cable, so that it may swing back and forth. Another type of pendulum consists of a weight attached by a light (but inflexible) rod to an axle, so that it can swing through larger angles, even making a 360° rotation if given enough velocity.

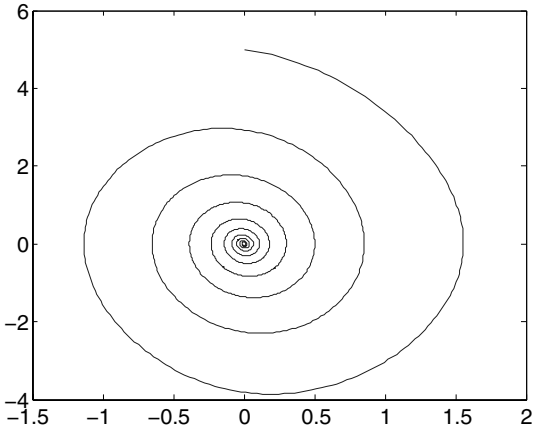
Equations of Motion

Though it is not precisely correct in practice, we often assume that the magnitude of the frictional forces that eventually slow the pendulum to a halt is proportional to the velocity of the pendulum. Assume also that the length of the pendulum is 1 m, the weight at the end of the pendulum has mass 1 kg, and the coefficient of friction is 0.5. In that case, the equations of motion for the pendulum are as follows.

$$x'(t) = y(t), \quad y'(t) = -0.5y(t) - 9.81\sin(x(t)),$$

where t represents time in seconds, x represents the angle of the pendulum from the vertical in radians (so that $x = 0$ is the rest position), y represents the angular velocity of the pendulum in radians per second, and 9.81 is approximately the acceleration due to gravity in meters per second squared. Now here is a phase portrait of the solution with initial position $x(0) = 0$ and initial velocity $y(0) = 5$. This is a graph of x versus y as a function of t, on the time interval $0 < t < 20$. (To use MATLAB's numerical differential-equation solver **ode45**, we combine x and y into a single vector **x**; see the online help for **ode45**.)

```
g = @(t, x) [x(2); -0.5*x(2) - 9.81*sin(x(1))];
[t, xa] = ode45(g, [0:0.01:20], [0 5]);
plot(xa(:, 1), xa(:, 2))
```

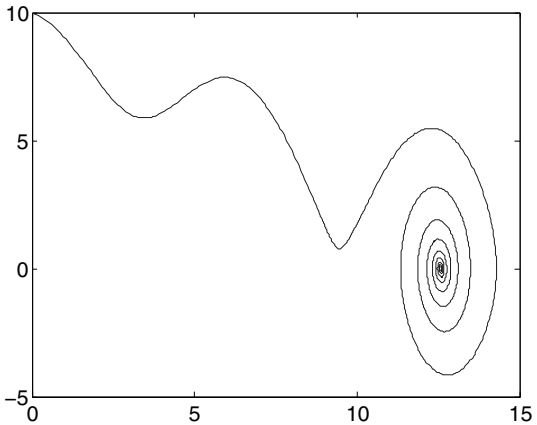


Recall that the x-coordinate corresponds to the angle of the pendulum and the y-coordinate corresponds to its velocity. Starting at $(0, 5)$, as t increases we follow the curve as it spirals clockwise toward $(0, 0)$. The angle oscillates back and forth, but with each swing it gets smaller until the pendulum is virtually at rest by the time $t = 20$. Meanwhile the velocity oscillates as well, taking its maximum value during each oscillation when the pendulum is in the middle of its swing (the angle is near zero) and crossing zero when the pendulum is at the end of its swing.

A Higher Initial Velocity

We increase the initial velocity to 10.

```
[t, xa] = ode45(g, [0:0.01:20], [0 10]);
plot(xa(:,1), xa(:,2))
```



This time the angle increases to over 14 radians before the curve spirals in to a point near $(12.5, 0)$. More precisely, it spirals toward $(4\pi, 0)$, because 4π radians represents the same position for the pendulum as 0 radians does. The pendulum has swung overhead and made two complete revolutions before beginning its damped oscillation toward its rest position. The velocity at first decreases but then rises after the angle

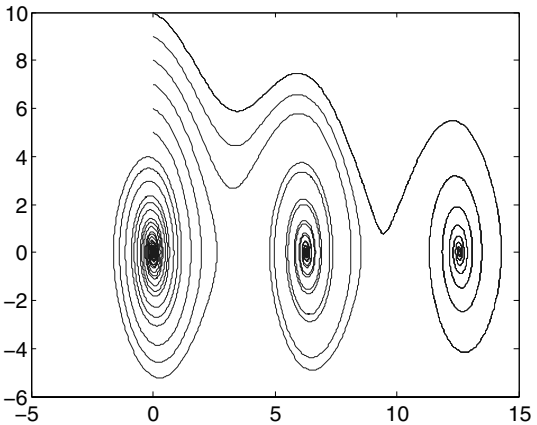
passes through π, as the pendulum passes the upright position and gains momentum. The pendulum has just enough momentum to swing through the upright position once more at the angle 3π.

Finding the Initial Velocity that Causes the Pendulum to Swing Overhead

Suppose that we want to find, to within 0.1, the minimum initial velocity required to make the pendulum, starting from its rest position, swing overhead once. It will be useful to be able to see the solutions corresponding to several different initial velocities on one graph.

First we consider the integer velocities 5 to 10.

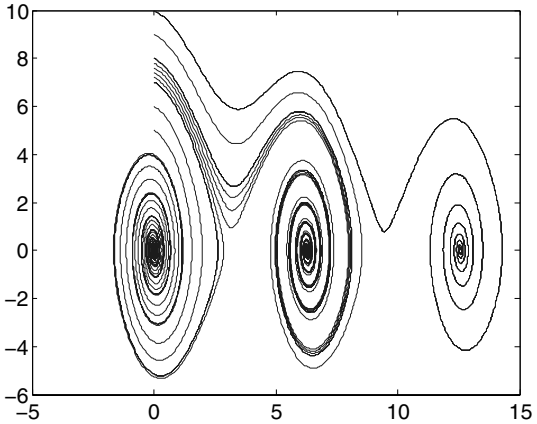
```
hold on
for a = 5:10
    [t, xa] = ode45(g, [0:0.01:20], [0 a]);
    plot(xa(:, 1), xa(:, 2))
end
hold off
```



Initial velocities 5, 6, and 7 are not large enough for the angle to increase past π, but initial velocities 8, 9, and 10 are enough to make the pendulum swing overhead. Let's see what happens between 7 and 8.

Second Iteration

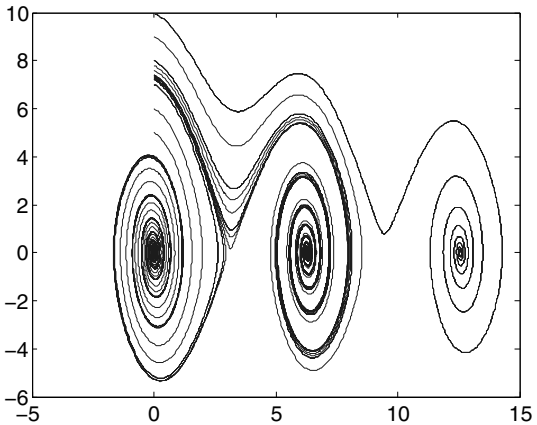
```
hold on
for a = 7.0:0.2:8.0
    [t, xa] = ode45(g, [0:0.01:20], [0 a]);
    plot(xa(:, 1), xa(:, 2))
end
hold off
```



We see that the cut-off is somewhere between 7.2 and 7.4. Let's make one more refinement.

Third Iteration

```
hold on
for a = 7.2:0.05:7.4
    [t, xa] = ode45(g, [0:0.01:20], [0 a]);
    plot(xa(:, 1), xa(:, 2))
end
hold off
```



We conclude that the minimum velocity needed is somewhere between 7.25 and 7.3.

☆ Numerical Solution of the Heat Equation

In this section we will use MATLAB to numerically solve the heat equation (also known as the diffusion equation), a partial differential equation that describes many

physical processes such as conductive heat flow or the diffusion of an impurity in a motionless fluid. You can picture the process of diffusion as a drop of dye spreading in a glass of water. (To a certain extent you could also picture cream in a cup of coffee, but in that case the mixing is generally complicated by the fluid motion caused by pouring the cream into the coffee, and is further accelerated by stirring the coffee.) The dye consists of a large number of individual particles, each of which repeatedly bounces off the surrounding water molecules, following an essentially random path. There are so many dye particles that their individual random motions form an essentially deterministic overall pattern as the dye spreads evenly in all directions (we ignore here the possible effect of gravity). In a similar way, you can imagine heat energy spreading through random interactions of nearby particles.

In a three-dimensional medium, the heat equation is

$$\frac{\partial u}{\partial t} = k\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right).$$

Here u is a function of t, x, y, and z that represents the temperature, or concentration of impurity in the case of diffusion, at time t at position (x, y, z) in the medium. The constant k depends on the materials involved, and is called the thermal conductivity in the case of heat flow, and the diffusion coefficient in the case of diffusion. To simplify matters, let us assume that the medium is instead one-dimensional. This could represent diffusion in a thin water-filled tube or heat flow in a thin insulated wire; let us think primarily of the case of heat flow. Then the partial differential equation becomes

$$\frac{\partial u}{\partial t} = k\,\frac{\partial^2 u}{\partial x^2},$$

where $u(x, t)$ is the temperature at time t a distance x along the wire.

A Finite-Difference Solution

To solve this partial differential equation we need both initial conditions of the form $u(x, 0) = f(x)$, where $f(x)$ gives the temperature distribution in the wire at time 0, and boundary conditions at the endpoints of the wire, call them $x = a$ and $x = b$. We choose so-called Dirichlet boundary conditions $u(a, t) = T_a$ and $u(b, t) = T_b$, which correspond to the temperature being held steady at values T_a and T_b at the two endpoints. Though an exact solution is available in this scenario, let us instead illustrate the numerical method of finite differences.

To begin with, on the computer we can keep track of the temperature u only at a discrete set of times and a discrete set of positions x. Let the times be $0, \Delta t, 2\Delta t, \ldots, N\Delta t$, and let the positions be $a, a + \Delta x, \ldots, a + J\Delta x = b$, and

let $u_j^n = u(a + j\Delta t, n\Delta t)$. After rewriting the partial differential equation in terms of finite-difference approximations to the derivatives, we get

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = k\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2}.$$

(These are the simplest approximations we can use for the derivatives, and this method can be refined by using more accurate approximations, especially for the t derivative.) Thus if, for a particular n, we know the values of u_j^n for all j, we can solve the equation above to find for each j,

$$u_j^{n+1} = u_j^n + \frac{k\Delta t}{\Delta x^2}(u_{j+1}^n - 2u_j^n + u_{j-1}^n) = s(u_{j+1}^n + u_{j-1}^n) + (1 - 2s)u_j^n,$$

where $s = k\Delta t/(\Delta x)^2$. In other words, this equation tells us how to find the temperature distribution at time step $n + 1$ given the temperature distribution at time step n. (At the endpoints $j = 0$ and $j = J$, this equation refers to temperatures outside the prescribed range for x, but at these points we will ignore the equation above and apply the boundary conditions instead.) We can interpret this equation as saying that the temperature at a given location at the next time step is a weighted average of its temperature and the temperatures of its neighbors at the current time step. In other words, in time Δt, a given section of the wire of length Δx transfers to each of its neighbors a portion s of its heat energy and keeps the remaining portion $1 - 2s$. Thus our numerical implementation of the heat equation is a discretized version of the microscopic description of diffusion we gave initially, that heat energy spreads due to random interactions between nearby particles.

The following M-file, which we have named `heat.m`, iterates the procedure described above.

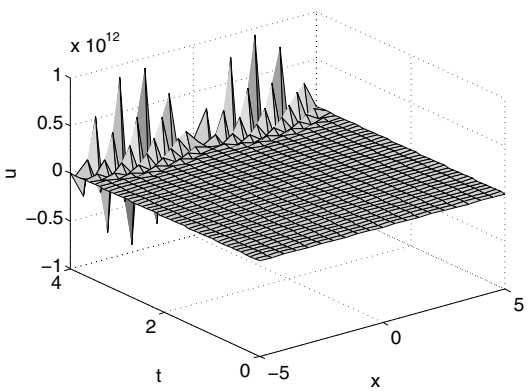
```
type heat

function u = heat(k, t, x, init, bdry)
% Solve the 1D heat equation on the rectangle described by
% vectors x and t with initial condition u(t(1), x) = init
% and Dirichlet boundary conditions u(t, x(1)) = bdry(1),
% u(t, x(end)) = bdry(2).

J = length(x);
N = length(t);
dx = mean(diff(x));
dt = mean(diff(t));
s = k*dt/dx^2;

u = zeros(N,J);
```

```
u(1,:) = init;

for n = 1:N-1
    u(n+1,2:J-1) = s*(u(n,3:J) + u(n,1:J-2)) + ...
        (1-2*s)*u(n,2:J-1);
    u(n+1,1) = bdry(1);
    u(n+1,J) = bdry(2);
end
```

The function `heat` takes as inputs the value of k, vectors of t and x values, a vector **`init`** of initial values (which is assumed to have the same length as **`x`**), and a vector **`bdry`** containing a pair of boundary values. Its output is a matrix of u values. Notice that since indices of arrays in MATLAB must start at 1, not 0, we have deviated slightly from our earlier notation by letting $n = 1$ represent the initial time and $j = 1$ represent the left endpoint. Notice also that, in the first line following the **`for`** statement, we compute an entire row of **`u`**, except for the first and last values, in one line; each term is a vector of length $J - 2$, with the index j increased by 1 in the term **`u(n,3:J)`** and decreased by 1 in the term **`u(n,1:J-2)`**.

Let's use the M-file above to solve the one-dimensional heat equation with $k = 2$ on the interval $-5 \le x \le 5$ from time 0 to time 4, using boundary temperatures 15 and 25, and an initial temperature distribution of 15 for $x < 0$ and 25 for $x > 0$. You can imagine that two separate wires of length 5 with different temperatures are joined at time 0 at position $x = 0$, and each of their far ends remains in an environment that holds it at its initial temperature. We must choose values for Δt and Δx; let's try $\Delta t = 0.1$ and $\Delta x = 0.5$, so that there are 41 values of t ranging from 0 to 4 and 21 values of x ranging from -5 to 5.

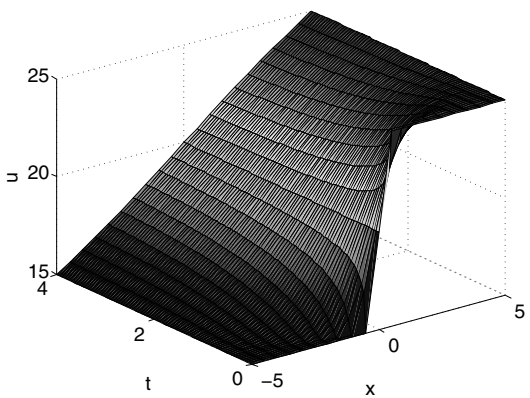
```
tvals = linspace(0, 4, 41);
xvals = linspace(-5, 5, 21);
init = 20 + 5*sign(xvals);
uvals = heat(2, tvals, xvals, init, [15 25]);
surf(xvals, tvals, uvals)
xlabel x; ylabel t; zlabel u
```



Here we used **surf** to show the entire solution $u(x,t)$. The output is clearly unrealistic; notice the scale on the u-axis! The numerical solution of partial differential equations is fraught with dangers, and instability like that seen above is a common problem with finite-difference schemes. For many partial differential equations a finite-difference scheme will not work at all, but for the heat equation and similar equations it will work well with proper choice of Δt and Δx. One might be inclined to think that, since our choice of Δx was larger, it should be reduced, but in fact this would only make matters worse. Ultimately the only parameter in the iteration we're using is the constant s, and one drawback of doing all the computations in an M-file as we did above is that we do not automatically see the intermediate quantities it computes. In this case we can easily calculate that $s = 2(0.1)/(0.5)2 = 0.8$. Notice that this implies that the coefficient $1 - 2s$ of u_j^n in the iteration above is negative. Thus the "weighted average" we described before in our interpretation of the iterative step is not a true average; each section of wire is transferring more energy than it has at each time step!

The solution to the problem above is thus to reduce the time step Δt; for instance, if we cut it in half, then $s = 0.4$, and all coefficients in the iteration are positive.

```
tvals = linspace(0, 4, 81);
uvals = heat(2, tvals, xvals, init, [15 25]);
surf(xvals, tvals, uvals)
xlabel x; ylabel t; zlabel u
```



This looks much better! As time increases, the temperature distribution seems to approach a linear function of x. Indeed, $u(x,t) = 20 + x$ is the limiting "steady state" for this problem; it satisfies the boundary conditions and it yields 0 on both sides of the partial differential equation.

Generally speaking, it is best to understand some of the theory of partial differential equations before attempting a numerical solution as we have done here. However, for this particular case at least, the simple rule of thumb of keeping the coefficients of the iteration positive yields realistic results. A theoretical examination of the stability of this finite-difference scheme for the one-dimensional heat equation shows that indeed any value of s between 0 and 0.5 will work, and suggests that the best value of Δt

to use for a given Δx is the one that makes $s = 0.25$. Notice that, while we can get more accurate results in this case by reducing Δx, if we reduce it by a factor of 10 we must reduce Δt by a factor of 100 to compensate, making the computation take 1000 times as long and use 1000 times the memory!

The Case of Variable Conductivity

Earlier we mentioned that the problem we solved numerically could also be solved analytically. The value of the numerical method is that it can be applied to similar partial differential equations for which an exact solution is not possible or at least not known. For example, consider the one-dimensional heat equation with a variable coefficient, representing an inhomogeneous material with varying thermal conductivity $k(x)$,

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(k(x)\frac{\partial u}{\partial x}\right) = k(x)\frac{\partial^2 u}{\partial x^2} + k'(x)\frac{\partial u}{\partial x}.$$

For the first derivatives on the right-hand side, we use a symmetric finite-difference approximation, so that our discrete approximation to the partial differential equations becomes

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} = k_j \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2} + \frac{k_{j+1} - k_{j-1}}{2\Delta x}\frac{u_{j+1} - u_{j-1}}{2\Delta x},$$

where $k_j = k(a + j\Delta x)$. Then the time iteration for this method is

$$u_j^{n+1} = s_j(u_{j+1}^n + u_{j-1}^n) + (1 - 2s_j)u_j^n + 0.25(s_{j+1} - s_{j-1})(u_{j+1}^n - u_{j-1}^n),$$

where $s_j = k_j \Delta t/(\Delta x)^2$. In the following M-file, which we called `heatvc.m`, we modify our previous M-file to incorporate this iteration.

```
type heatvc
```

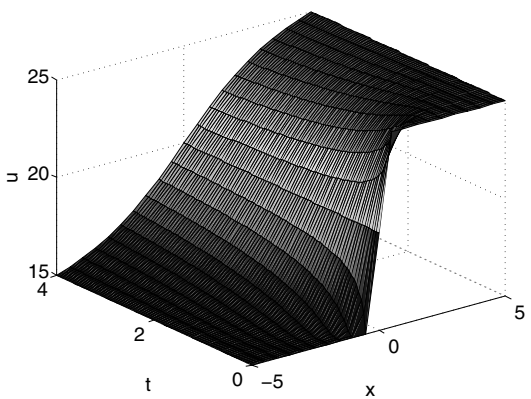
```
function u = heatvc(k, t, x, init, bdry)
% Solve the 1D heat equation with variable-coefficient k
% on the rectangle described by vectors x and t with
% u(t(1), x) = init and Dirichlet boundary conditions
% u(t, x(1)) = bdry(1), u(t, x(end)) = bdry(2).

J = length(x);
N = length(t);
dx = mean(diff(x));
```

```
dt = mean(diff(t));
s = k*dt/dx^2;

u = zeros(N,J);
u(1,:) = init;

for n = 1:N-1
    u(n+1,2:J-1) = s(2:J-1).*(u(n,3:J) + u(n,1:J-2)) + ...
        (1 - 2*s(2:J-1)).*u(n,2:J-1) + ...
        0.25*(s(3:J) - s(1:J-2)).*(u(n,3:J) - u(n,1:J-2));
    u(n+1,1) = bdry(1);
    u(n+1,J) = bdry(2);
end
```

Notice that **k** is now assumed to be a vector with the same length as **x**, and that as a result so is **s**. This in turn requires that we use vectorized multiplication in the main iteration, which we have now split into three lines.

Let's use this M-file to solve the one-dimensional variable coefficient heat equation with the same boundary and initial conditions as before, using $k(x) = 1 + (x/5)^2$. Since the maximum value of k is 2, we can use the same values of Δt and Δx as before.

```
kvals = 1 + (xvals/5).^2;
uvals = heatvc(kvals, tvals, xvals, init, [15 25]);
surf(xvals, tvals, uvals)
xlabel x; ylabel t; zlabel u
```



In this case the limiting temperature distribution is not linear; it has a steeper temperature gradient in the middle, where the thermal conductivity is lower. Again one could find the exact form of this limiting distribution, $u(x,t) = 20(1+(1/\pi)\arctan(x/5))$, by setting the t derivative to zero in the original equation and solving the resulting ordinary differential equation.

You can use the method of finite differences to solve the heat equation in two or three space dimensions as well. For this and other partial differential equations with time

and two space dimensions, you can also use the PDE Toolbox, which implements the more sophisticated "finite-element method."

A Simulink Solution

We can also solve the heat equation using Simulink. To do this we continue to approximate the x-derivatives with finite differences, but think of the equation as a vector-valued ordinary differential equation, with t as the independent variable. Simulink solves the model using MATLAB's ODE solver, **`ode45`**. To illustrate how to do this, let's take the example we started with, the case in which $k = 2$ on the interval $-5 \leq x \leq 5$ from time 0 to time 4, using boundary temperatures 15 and 25, and an initial temperature distribution of 15 for $x < 0$ and 25 for $x > 0$. We replace $u(x,t)$ for fixed t by the vector **`u`** of values of $u(x,t)$, with, say, **`x = -5:5`**. Here there are 11 values of x at which we are sampling u, but, since $u(x,t)$ is pre-determined at the endpoints, we can take **`u`** to be a nine-dimensional vector, and we just tack on the values at the endpoints when we have finished. Since we're replacing $\partial^2/\partial x^2$ by its finite difference approximation and we've taken $\Delta x = 1$ for simplicity, our equation becomes the vector-valued ODE

$$\frac{\partial \mathbf{u}}{\partial t} = k(A\mathbf{u} + c)$$

Here the right-hand side represents our approximation to $k(\partial^2 u/\partial x^2)$. The matrix A is

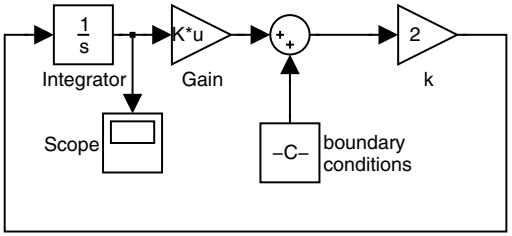
$$\begin{pmatrix} -2 & 1 & \dots & 0 \\ 1 & -2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 0 & \dots & 1 & -2 \end{pmatrix},$$

since we are replacing $\partial^2 u/\partial x^2$ at (n,t) with $u(n-1,t) - 2u(n,t) + u(n+1,t)$. We represent this matrix in MATLAB's notation by **`-2*eye(9) + diag(ones(8,1), 1) + diag(ones(8,1), -1)`**. The vector **`c`** comes from the boundary conditions, and has 15 in its first entry, 25 in its last entry, and 0's in between. We represent it in MATLAB's notation as **`[15; zeros(7,1); 25]`**. The formula for **`c`** comes from the fact that **`u(1)`** represents $u(-4,t)$, and $\partial^2 u/\partial x^2$ at this point is approximated by

$$u(-5,t) - 2u(-4,t) + u(-3,t) = 15 - 2\mathbf{u}(\mathbf{1}) + \mathbf{u}(\mathbf{2})$$

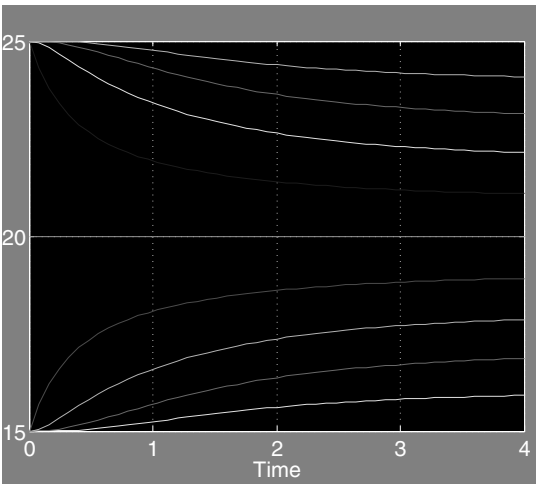
and similarly at the other endpoint. Here's a Simulink model representing this equation:

```
open system heateq.mdl
```



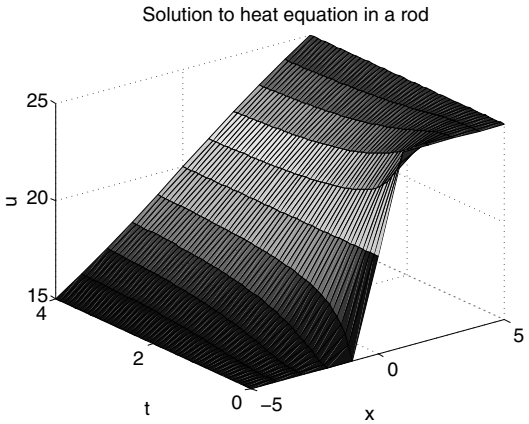
Note that one needs to specify the initial conditions for u as Block Parameters for the Integrator block, and that in the Block Parameters dialog box for the Gain block, one needs to set the multiplication type to "Matrix". Since **`u(1)`** through **`u(4)`** represent the solution $u(x,t)$ at $x = -4$ through -1, and **`u(6)`** through **`u(9)`** represent $u(x,t)$ at $x = 1$ through 4, we take the initial value of u to be **`[15*ones(4,1); 20; 25*ones(4,1)]`**. (We use 20 as a compromise at $x = 0$, since this is right in the middle of the regions where u is 15 and 25.) The output from the model is displayed in the **Scope** block in the form of graphs of the various entries of u as a function of t, but it's more useful to save the output to the MATLAB Workspace and then plot it with **`surf`**. Incidentally, it helps to reset the y-axis limits on the **Scope** block to run from 15 to 25. To make this adjustment and run the model from $t = 0$ to $t = 4$, we execute the commands

```
set_param('heateq/Scope', 'YMin', '15');
set_param('heateq/Scope', 'YMax', '25');
[tout, uout] = sim('heateq.mdl', [0,4]);
simplot(tout, uout)
```



This saves the simulation times (from 0 to 4) as a vector **`tout`** and the computed values of u as **`uout`**, a matrix with nine columns. Each row of these arrays corresponds to a single time step, and each column of **`uout`** corresponds to one value of x. But remember that we have to add in the values of u at the endpoints as additional columns in u. So we plot the data as follows:

```
u = [15*ones(length(tout),1), uout, ...
    25*ones(length(tout),1)];
x = -5:5;
clf reset
set(gcf, 'Color', 'White')
surf(x, tout, u)
xlabel('x'), ylabel('t'), zlabel('u')
title('Solution to heat equation in a rod')
```



Notice how similar this is to the picture obtained before for constant conductivity $k = 2$. We leave it to the reader to modify the model for the case of variable conductivity.

Solution with pdepe

MATLAB has a built-in solver **pdepe** for partial differential equations in one space dimension (as well as time t). To find out more about it, read the online help on **pdepe**. The instructions for use of **pdepe** are quite explicit but somewhat complicated. The method it uses is somewhat similar to that used in the Simulink solution above; i.e., it uses an ODE solver in t and finite differences in x. The following M-file solves the second problem above, the one with variable conductivity. Note the use of function handles and subfunctions.

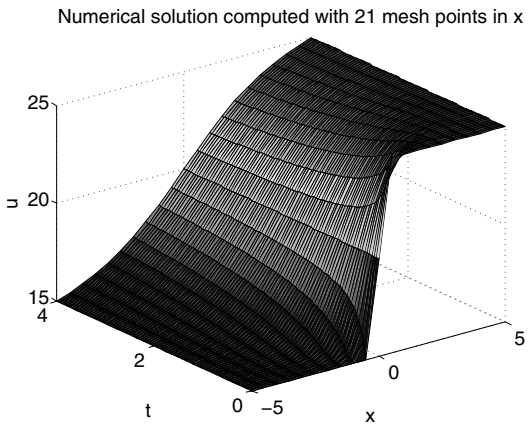
```
type heateqex2
```

```
function heateqex2
% Solves a Dirichlet problem for the heat equation in a rod,
% this time with variable conductivity, 21 mesh points.
m = 0;  % This simply means geometry is linear.
x = linspace(-5, 5, 21);
t = linspace(0, 4, 81);
sol = pdepe(m, @pdex, @pdexic, @pdexbc, x, t);
% Extract the first solution component as u.
u = sol(:,:,1);
```

```
% A surface plot is often a good way to study a solution.
surf(x, t, u)
title('Numerical solution computed with 21 mesh points in x')
xlabel('x'), ylabel('t'), zlabel('u')
% ---------------------------------------------------------
function [c, f, s] = pdex(x, t, u, DuDx)
c = 1;
f = (1 + (x/5).^2)*DuDx;  % flux = conductivity times u_x
s = 0;
% ---------------------------------------------------------
function u0 = pdexic(x)
u0 = 20 + 5*sign(x);  % initial condition at t = 0
% ---------------------------------------------------------
function [pl, ql, pr, qr] = pdexbc(xl, ul, xr, ur, t)
% q's are zero since we have Dirichlet conditions
% pl = 0 at the left, pr = 0 at the right endpoint
pl = ul - 15;
ql = 0;
pr = ur - 25;
qr = 0;
```

Running it gives:

heateqex2



Again the results are very similar to those obtained before.

☆ A Model of Traffic Flow

Everyone has had the experience of sitting in a traffic jam, or of seeing cars bunch up on a road for no apparent good reason. MATLAB and Simulink are good tools for studying models of such behavior. Our analysis here will be based on so-called "follow-the-leader" theories of traffic flow, about which you can read more in **Kinetic Theory of Vehicular Traffic**, by Ilya Prigogine and Robert Herman, Elsevier, New

York, 1971, or in **The Theory of Road Traffic Flow**, by Winifred Ashton, Methuen, London, 1966. We will analyze here an extremely simple model that already exhibits quite complicated behavior. We consider a one-lane, one-way, circular road with a number of cars on it (a very primitive model of, say, the Outer Loop of the Capital Beltway around Washington, DC, since, in very dense traffic, it is hard to change lanes and each lane behaves like a one-lane road). Each driver slows down or speeds up on the basis of his own speed, the speed of the car ahead of him, and the distance to the car ahead of him. But human drivers have a finite reaction time. In other words, it takes them a certain amount of time (usually about a second) to observe what is going on around them and to press the gas pedal or the brake, as appropriate. The standard "follow-the-leader" theory supposes that

$$\ddot{u}_n(t+T) = \lambda\big(\dot{u}_{n-1}(t) - \dot{u}_n(t)\big), \qquad (\dagger)$$

where t is time, T is the reaction time, u_n is the position of the nth car, and the "sensitivity coefficient" λ may depend on $u_{n-1}(t) - u_n(t)$, the spacing between cars, and/or $\dot{u}_n(t)$, the speed of the nth car. The idea behind this equation is this. A driver will tend to decelerate if he is going faster than the car in front of him, or if he is close to the car in front of him, and will tend to accelerate if he is going slower than the car in front of him. In addition, a driver (especially in light traffic) may tend to speed up or slow down depending on whether he is going slower or faster (respectively) than a "reasonable" speed for the road (often, but not always, equal to the posted speed limit). Since our road is circular, in this equation u_0 is interpreted as u_N, where N is the total number of cars.

The simplest version of the model is the one in which the "sensitivity coefficient" λ is a (positive) constant. Then we have a homogeneous linear differential/difference equation with constant coefficients for the velocities $\dot{u}_n(t)$. Obviously there is a "steady-state" solution when all the velocities are equal and constant (i.e., traffic is flowing at a uniform speed), but what we are interested in is the stability of the flow, or the question of what effect is produced by small differences in the velocities of the cars. The solution of (†) will be a superposition of exponential solutions of the form

$$u_n(t) = \exp(\alpha t) v_n,$$

where the v_n's and α are (complex) constants, and the system will be unstable if the velocities are unbounded, i.e., there are any solutions where the real part of α is positive. Using vector notation, we have

$$\dot{\mathbf{u}}(t) = \exp(\alpha t)\mathbf{v}, \qquad \ddot{\mathbf{u}}(t+T) = \alpha \exp(\alpha T)\exp(\alpha t)\mathbf{v}.$$

Substituting back into (†), we get the equation

$$\alpha \exp(\alpha T) \exp(\alpha t)\mathbf{v} = \lambda(S - I) \exp(\alpha t)\mathbf{v},$$

where

$$S = \begin{pmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}$$

is the "shift" matrix that, when it multiplies a vector on the left, cyclically permutes the entries of the vector. We can cancel the $\exp(\alpha t)$ on each side to get

$$\alpha \exp(\alpha T)\mathbf{v} = \lambda(S - I)\mathbf{v},$$

or

$$\left(S - \left(1 + \frac{\alpha}{\lambda} \exp(\alpha T)\right) I\right) \mathbf{v} = 0, \qquad (**)$$

which says that v is an eigenvector for S with eigenvalue

$$1 + \frac{\alpha}{\lambda} \exp(\alpha T).$$

Since the eigenvalues of S are the Nth roots of unity, which are evenly spaced around the unit circle in the complex plane, and closely spaced together for large N, there is potential instability whenever

$$1 + \frac{\alpha}{\lambda} \exp(\alpha T)$$

has absolute value 1 for some α with positive real part; that is, whenever

$$\left(\frac{\alpha T}{\lambda T}\right) e^{\alpha T}$$

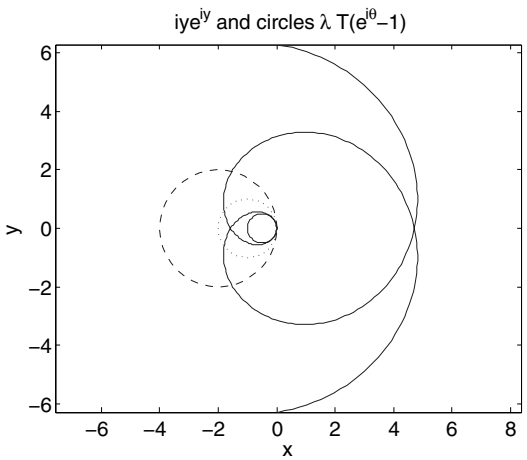
can be of the form $\exp(i\theta) - 1$ for some αT with positive real part. Whether instability occurs depends on the value of the product λT. We can see this by plotting values

of $z\exp(z)$ for $z = \alpha T = iy$ a complex number on the critical line Re $z = 0$, and comparing with plots of $\lambda T(e^{i\theta} - 1)$ for various values of the parameter λT.

```
syms y; expand(i*y*(cos(y) + i*sin(y)))
```

```
ans =

i*y*cos(y)-y*sin(y)
```

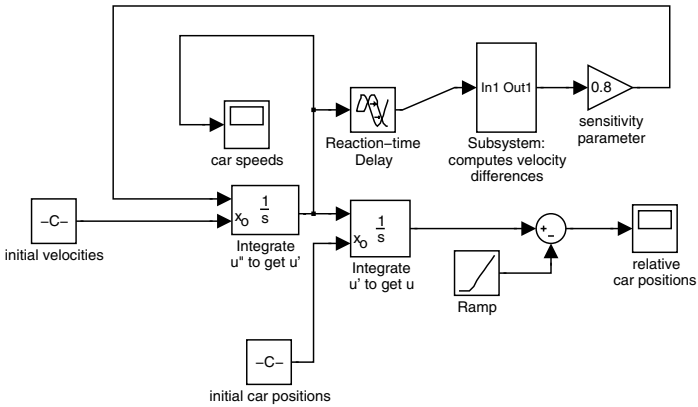
```
ezplot(-y*sin(y), y*cos(y), [-2*pi, 2*pi]); hold on
theta = 0:0.05*pi:2*pi;
plot((1/2)*(cos(theta) - 1), (1/2)*sin(theta), '-');
plot(cos(theta) - 1, sin(theta), ':')
plot(2*(cos(theta) - 1), 2*sin(theta), '--');
title('iye^{iy} and circles \lambda T(e^{i\theta}-1)')
hold off
```



Here the small solid circle corresponds to $\lambda T = 1/2$, and we are just at the limit of stability, since this circle does not cross the spiral produced by $z\exp(z)$ for z a complex number on the critical line Re $z = 0$, though it "hugs" the spiral closely. The dotted and dashed circles, corresponding to $\lambda T = 1$ or 2, do cross the spiral, so they correspond to unstable traffic flow.

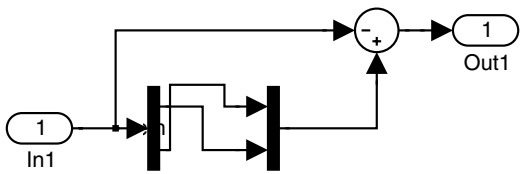
We can check these theoretical predictions with a simulation using Simulink. We'll give a picture of the Simulink model and then explain it.

```
open_system traffic
```



Here the Subsystem, which corresponds to multiplication by $S - I$, looks like this:

```
open_system 'traffic/Subsystem: computes velocity differences'
```



Most of the model is like the example in Chapter 8, except that our unknown function (called u), representing the car positions, is vector-valued, not scalar-valued. The major exceptions are these.

- We need to incorporate the reaction-time delay, so we've inserted a **Transport Delay** block from the **Continuous** block library, with the "Time delay" parameter T set to 0.5.
- The parameter λ shows up as the value of the gain in the "sensitivity parameter" **Gain** block in the upper right.
- Plotting car positions by themselves is not terribly useful, since only the relative positions matter. So before outputting the car positions to the **Scope** block labeled "relative car positions," we've subtracted a constant linear function (corresponding to uniform motion at the average car speed) created by the **Ramp** block from the **Sources** block library.
- We've made use of the option in the **Integrator** blocks to input the initial conditions, instead of having them built into the block. This makes the logical structure a little clearer.
- We've used the subsystem feature of Simulink. If you enclose a bunch of blocks with the mouse and then click on "Create subsystem" in the model's **Edit** menu, Simulink will package them as a subsystem. This is helpful if your model is large or if there is some combination of blocks that you expect to use more than once. Our subsystem sends a vector v to $(S - I)v = Sv - v$. A **Sum** block (with one of the signs changed to a minus) is used for vector subtraction.

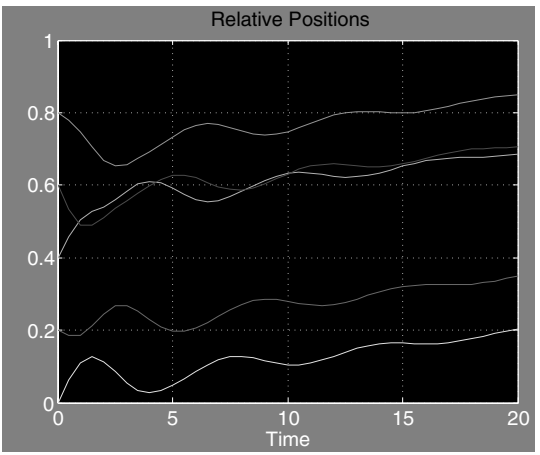
To model the action of S, we've used the **Demux** and **Mux** blocks from the **Signal Routing** block library. The **Demux** block, with the "number of outputs" parameter set to `[4, 1]`, splits a five-dimensional vector into a pair consisting of a four-dimensional vector and a scalar (corresponding to the last car). Then we reverse the order of these and put them back together with the **Mux** block, with the "number of inputs" parameter set to `[1, 4]`.

Once the model has been assembled, it can be run with various inputs. You can see the results yourself in the two **Scope** windows, but here we've run the simulation from the command line and plotted the results with the `simplot` command, that does almost the same thing as a **Scope** but in a regular MATLAB figure window. The following pictures are produced with $\lambda = 0.8$, corresponding to stable flow (though, to be honest, we've let two cars cross through each other briefly!):

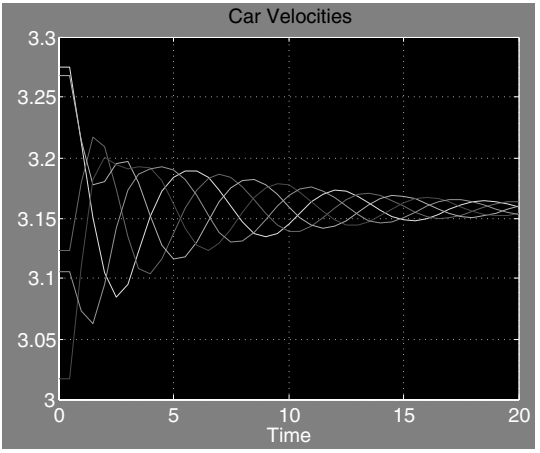
```
set_param('traffic/sensitivity parameter', 'Gain', '0.8');
[t, x] = sim('traffic');
```

The variable `t` stores the time parameter, the variable `x` stores car velocities in its first five columns and car positions in the second five columns. In this example, the average velocity is 3.15. First we plot the relative positions, then we plot the velocities.

```
relpos = x(:,6:10) - 3.15*t*ones(1,5);
simplot(t, relpos), title('Relative Positions')
axis([0, 20, 0, 1]), axis normal
```

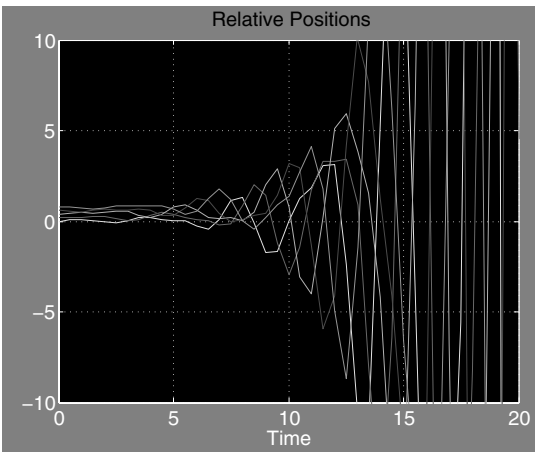



```
simplot(t, x(:,1:5)), title('Car Velocities')
axis([0, 20, 3, 3.3])
```

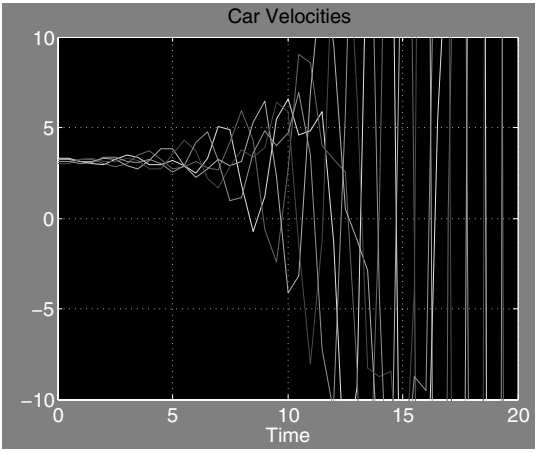


As you can see, the speeds fluctuate but eventually converge to a single value, and the separations between cars eventually stabilize. On the other hand, if λ is increased by changing the "sensitivity parameter" in the **Gain** block in the upper right, say from 0.8 to 2.0, we get the following output, which is typical of instability:

```
set_param('traffic/sensitivity parameter', 'Gain', '2.0');
[t, x] = sim('traffic');
relpos = x(:,6:10) - 3.15*t*ones(1,5);
simplot(t, relpos), title('Relative Positions')
axis([0, 20, -10, 10])
```



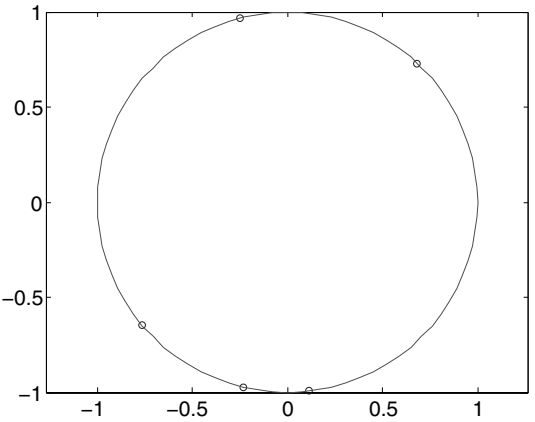

```
simplot(t, x(:,1:5)), title('Car Velocities')
axis([0, 20, -10, 10])
```



We encourage you to go back and tinker with the model (for instance using a sensitivity parameter that is also inversely proportional to the spacing between cars) and study the results.

Finally, you can create a movie with the following code:

```
clf reset
set(gcf, 'Color', 'White')
clear M
theta = (0:0.025:2)*pi;
for j = 1:length(t)
    plot(cos(x(j, 1:5)), sin(x(j, 1:5)), 'o');
    axis([-1, 1, -1, 1]);
    hold on; plot(cos(theta), sin(theta), 'r'); hold off
    axis equal;
    M(j) = getframe;
end
```



The idea here is that we have taken the circular road to have radius 1 (in suitable units), so that the command **`plot(cos(theta),sin(theta),'r')`** draws a red circle (representing the road) in each frame of the movie, and on top of that the

cars are shown with moving little circles. The graph above is the last frame of the movie; you can view the entire movie by typing **`movie(M)`** or **`movieview(M)`**. Try it!

We should mention here one fine point needed to create a realistic movie. Namely, we need the values of **`t`** to be equally spaced – otherwise the cars will appear to be moving faster when the time steps are large, and will appear to be moving slower when the time steps are small. In its default mode of operation, Simulink uses a variable-step differential-equation solver based on MATLAB's command **`ode45`**, so the entries of **`t`** will not be equally spaced. We have fixed this by opening the **Configuration Parameters** dialog box using the **Simulation** menu in the model window, and, under the **Data Import/Export** item, changing the **Output options** box to read `Produce specified output only`, with `Output times` chosen to be **`[0:0.5:20]`**. Then the model will output the car positions only at times that are multiples of 0.5, and the MATLAB program above will produce a 41-frame movie.

Practice Set C

Developing Your MATLAB Skills

Problems 5, 7, 14, and 15 are a bit more advanced than the others. Problems 8 and 9 require either **`simlp`** from Simulink or else the Optimization Toolbox. Problem 11(a) requires the Symbolic Math Toolbox; the others do not. Simulink is needed for Problems 12 and 13.

1. Captain Picard is hiding in a square arena, 50 meters on a side, which is protected by a level-5 force field. Unfortunately, the Cardassians, who are firing on the arena, have a death ray that can penetrate the force field. The point of impact of the death ray is exposed to 10,000 *illumatons* of lethal radiation. It requires only 50 illumatons to dispatch the Captain; anything less has no effect. The number of illumatons that arrive at point (x, y) when the death ray strikes one meter above ground at point (x_0, y_0) is governed by an inverse square law, namely
$$\frac{10000}{4\pi((x-x_0)^2+(y-y_0)^2+1)}.$$
The Cardassian sensors cannot locate Picard's exact position, so they fire at a random point in the arena.

   (a) Use **`contour`** to display the arena after five random bursts of the death ray. The half-life of the radiation is very short, so one can assume that it disappears almost immediately – only its initial burst has any effect. Nevertheless, include all five bursts in your picture, like a time-lapse photo. Where in the arena do you think Captain Picard should hide?

   (b) Suppose that Picard stands in the center of the arena. Moreover, suppose that the Cardassians fire the death ray 100 times, each shot landing at a random point in the arena. Is Picard killed?

   (c) Re-run the "experiment" in part (b) 100 times, and approximate the probability that Captain Picard can survive an attack of 100 shots.

   (d) Redo part (c) but place the Captain halfway to one side (i.e., at $x = 37.5$, $y = 25$ if the coordinates of the arena are $0 \le x \le 50, 0 \le y \le 50$).

   (e) Redo the simulation with the Captain completely to one side, and finally in a corner. What self-evident fact is reinforced for you?

2. Consider an account that has M dollars in it and pays monthly interest J. Suppose that, beginning at a certain point, an amount S is deposited monthly and no withdrawals are made.

   (a) Assume first that $S = 0$. Using the *Mortgage Payments* application in Chapter 10 as a model, derive an equation relating J, M, the number n of months elapsed, and the total T in the account after n months. Assume that the interest is credited on the last day of the month and the total T is computed on the last day after the interest is credited.

   (b) Now assume that $M = 0$, that S is deposited on the first day of the month, and that as before interest is credited on the last day of the month, and the total T is computed on the last day after the interest has been credited. Once again, using the mortgage application as a model, derive an equation relating J, S, the number n of months elapsed, and the total T in the account after n months.

   (c) By combining the last two models derive an equation relating all of M, S, J, n, and T, now of course assuming that there is an initial amount in the account (M) as well as a monthly deposit (S).

   (d) If the annual interest rate is 5%, and no monthly deposits are made, how many years does it take to double your initial stash of money? What if the annual interest rate is 10%?

   (e) In this and the next part, there is no initial stash. Assume an annual interest rate of 8%. How much do you have to deposit monthly to be a millionaire in 35 years (a career)?

   (f) If the interest rate remains as in (e) and you can afford to deposit only \$300 each month, how long do you have to work to retire a millionaire?

   (g) You hit the lottery and win \$100,000. You have two choices: take the money, pay the taxes, and invest what's left; or receive \$100,000/240 monthly for 20 years, depositing what's left after taxes. Assume that a \$100,000 windfall costs you \$35,000 in federal and state taxes, but that the smaller monthly payoff causes only a 20% tax liability. In which way are you better off 20 years later? Assume a 5% annual interest rate here.

   (h) Historically, banks have paid roughly 5%, while the stock market has tended to return 8% on average over a 10-year period. So parts (e) and (f) relate more to investing than to saving. But suppose that the market in a 5-year period returns 13%, 15%, -3%, 5% and 10% in five successive years, and then repeats the cycle. (Note that the [arithmetic] average is 8%, though a geometric mean would be more relevant here.) Assume that \$50,000 is invested at the start of a 5-year market period. How much does it grow to in 5 years? Now recompute four more times, assuming that you enter the cycle at the beginning of the second year, the third year, etc. Which choice yields the best/worst results? Can you explain why? Compare the results with a fixed-rate account paying 8%. Assume sim-

ple annual interest. Redo the five investment computations, assuming that \$10,000 is invested at the start of each year. Again analyze the results.

3. Tony Gwynn had a lifetime batting average of .338. This means that, for every 1000 at bats, he had 338 hits. (For this exercise, we shall ignore walks, hit batsmen, sacrifices, and other plate appearances that do not result in an official at bat.) In an average year he amassed 500 official at bats.

   (a) Design a Monte Carlo simulation of a year in Tony's career. Run it. What is his batting average?

   (b) Now simulate a 20-year career. Assume 500 official at bats every year. What is his best batting average in his career? What is his worst? What is his lifetime average?

   (c) Now run the 20-year career simulation four more times. Answer the questions in part (b) for each of the four simulations.

   (d) Compute the average of the five lifetime averages you computed in parts (b) and (c). What do you think would happen if you ran the 20-year simulation 100 times and took the average of the lifetime averages for all 100 simulations?

The next four problems illustrate some basic MATLAB programming skills.

4. For a positive integer n, let $A(n)$ be the $n \times n$ matrix whose entry in the (i, j)-position is $a_{ij} = 1/(i + j - 1)$. For example,

$$A(3) = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{pmatrix}.$$

   The eigenvalues of $A(n)$ are all real numbers. Write a script M-file that prints the largest eigenvalue of $A(500)$, without any extraneous output. (*Hint*: the M-file may take a while to run if you use a loop within a loop to define A. Try to avoid this!)

5. ☆ Write a script M-file that draws a bulls-eye pattern with a central circle colored red, surrounded by alternating circular strips (annuli) of white and black, say ten of each. Make sure that the final display shows circles, not ellipses. (*Hint*: one way to color the region between two circles black is to color the entire inside of the outer circle black, then color the inside of the inner circle white.)

6. MATLAB has a function **`lcm`** that finds the least common multiple of two numbers. Write a function M-file `mylcm.m` that finds the least common multiple of an arbitrary number of positive integers, which may be given as separate arguments or in a vector. For example, **`mylcm(4, 5, 6)`** and **`mylcm([4 5 6])`** should both produce the answer `60`. The program should produce a

helpful error message if any of the inputs are not positive integers. (*Hint*: for three numbers you could use `lcm` to find the least common multiple m of the first two numbers, then use `lcm` again to find the least common multiple of m and the third number. Your M-file can generalize this approach.)

7. ☆ Write a function M-file that takes as input a string containing the name of a text file and produces a histogram of the number of occurrences of each letter from A to Z in the file. Try to label the figure and axes as usefully as you can.

8. Consider the following linear programming problem. Jane Doe is running for County Commissioner. She wants to personally canvass voters in the four main cities in the county: Gotham, Metropolis, Oz, and River City. She needs to figure out how many residences (private homes, apartments, etc.) to visit in each city. The constraints are as follows.

   (i) She intends to leave a campaign pamphlet at each residence; she has only 50,000 available.

   (ii) The travel costs she incurs for each residence are \$0.50 in each of Gotham and Metropolis, \$1 in Oz, and \$2 in River City; she has \$40,000 available.

   (iii) The number of minutes (on average) that her visits to each residence require are 2 minutes in Gotham, 3 minutes in Metropolis, a minute in Oz, and 4 minutes in River City; she has 300 hours available.

   (iv) Because of political profiles Jane knows that she should not visit any more residences in Gotham than she does in Metropolis, and that, however many residences she visits in Metropolis and Oz, the total of the two should not exceed the number she visits in River City.

   (v) Jane expects to receive, during her visits, on average, campaign contributions of one dollar from each residence in Gotham, a quarter from those in Metropolis, a half-dollar from the Oz residents, and three dollars from the folks in River City. She must raise at least \$10,000 from her entire canvass.

   Jane's goal is to maximize the number of supporters (those likely to vote for her). She estimates that for each residence she visits in Gotham the odds are 0.6 that she picks up a supporter, and the corresponding probabilities in Metropolis, Oz, and River City are, respectively, 0.6, 0.5, and 0.3.

   (a) How many residences should she visit in each of the four cities?

   (b) Suppose that she can double the time she can allot to visits. Now what is the profile for visits?

   (c) But suppose that the extra time (in part (b)) also mandates that she double the contributions she receives. What is the profile now?

9. Consider the following linear programming problem. The famous football coach Joe Glibb is trying to decide how many hours to spend with each component of his offensive unit during the coming week – that is, the quarterback, the running backs, the receivers, and the linemen. The constraints are as follows.

(i) The number of hours available to Joe during the week is 50.

(ii) Joe figures he needs 20 points to win the next game. He estimates that for each hour he spends with the quarterback, he can expect a point return of 0.5. The corresponding numbers for the running backs, receivers and linemen are 0.3, 0.4, and 0.1, respectively.

(iii) In spite of their enormous size, the players are relatively thin skinned. Each hour with the quarterback is likely to require Joe to criticize him once. The corresponding numbers of criticisms per hour for the other three groups are 2 for running backs, 3 for receivers, and 0.5 for linemen. Joe figures he can bleat out only 75 criticisms in a week before he loses control.

(iv) Finally, the players are *prima donnas* who engage in rivalries. Because of that, he must spend exactly the same number of hours with the running backs as he does with the receivers, at least as many hours with the quarterback as he does with the runners and receivers combined, and at least as many hours with the receivers as with the linemen.

Joe worries that he's going to be fired at the end of the season regardless of the outcome of the game, so his primary goal is to maximize his pleasure during the week. (The team's owner should only know.) He estimates that, on a sliding scale from 0 to 1, he gets 0.2 units of personal satisfaction for each hour with the quarterback. The corresponding numbers for the runners, receivers and linemen are 0.4, 0.3, and 0.6, respectively.

(a) How many hours should Joe spend with each group?

(b) Suppose that he needs only 15 points to win; then how many?

(c) Finally suppose, despite needing only 15 points, that the troops are getting restless and he can only dish out only 70 criticisms this week. Is Joe getting the most out of his week?

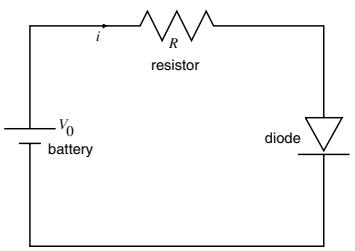


Figure C.1. A Nonlinear Circuit.

10. This problem, suggested to us by our colleague Tom Antonsen, concerns an electrical circuit, one of whose components does not behave linearly. Consider

the circuit in Figure C.1. Unlike the resistor, the diode is a non-linear element – it does not obey Ohm's Law. In fact its behavior is specified by the formula

$$i = I_0 \exp(V_D/V_T), \tag{C.1}$$

where i is the current in the diode (which is the same as that in the resistor by Kirchhoff's Current Law), V_D is the voltage across the diode, I_0 is the leakage current of the diode, and $V_T = kT/e$, where k is Boltzmann's constant, T is the temperature of the diode, and e is the electrical charge.

Now, by application of Ohm's Law to the resistor, we also know that $V_R = iR$, where V_R is the voltage across the resistor and R is its resistance. But, by Kirchhoff's Voltage Law, we also have $V_R = V_0 - V_D$. This gives a second equation relating the diode current and voltage, namely

$$i = (V_0 - V_D)/R. \tag{C.2}$$

Note now that (C.2) says that i is a decreasing linear function of V_D with value V_0/R when V_D is zero. At the same time (C.1) says that i is an exponentially growing function of V_D starting out at I_0. Since, typically, $RI_0 < V_0$, the two resulting curves (for i as a function of V_D) must cross once. Eliminating i from the two equations, we see that the voltage in the diode must satisfy the transcendental equation

$$(V_0 - V_D)/R = I_0 \exp(V_D/V_T),$$

or

$$V_D = V_0 - RI_0 \exp(V_D/V_T).$$

(a) Reasonable values for the electrical constants are $V_0 = 1.5$ volts, $R = 1000$ ohms, $I_0 = 10^{-5}$ amperes, and $V_T = 0.0025$ volts. Use **`fzero`** to find the voltage V_D and current i in the circuit.

(b) In the remainder of the problem, we assume that the voltage V_0 in the battery and the resistance R of the resistor are unchanged. But suppose that we have some freedom to alter the electrical characteristics of the diode. For example, suppose that I_0 is halved. What happens to the voltage?

(c) Suppose that, instead of halving I_0, we halve V_T. Then what is the effect on V_D?

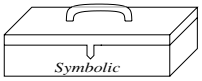
(d) Suppose that both I_0 and V_T are cut in half. What then?

(e) Finally, we want to examine the behavior of the voltage if both I_0 and V_T are decreased toward zero. For definitiveness, assume that we set $I_0 = 10^{-5}x$ and $V_T = 0.0025x$, and let $x \to 0$. Specifically, compute the solution for $x = 10^{-j}$, $j = 0, \ldots, 5$. Then, display a **`loglog`** plot of the solution values, for the voltage as a function of I_0. What do you conclude?

11. This problem is based on the *Population Dynamics* and *The 360° Pendulum* applications from Chapter 10. The growth of a species was modeled in the former by a *difference equation*. In this problem we will model population growth by a *differential equation*, akin to the second application mentioned above. In fact we can give a differential-equation model for the logistic growth of a population x as a function of time t by means of the equation

$$\dot{x} = x(1-x) = x - x^2, \tag{C.3}$$

where $\dot{x}$ denotes the derivative of x with respect to t. We think of x as a fraction of some maximal possible population. One advantage of this continuous model over the discrete model in Chapter 10 is that we can get a "reading" of the population at any point in time (not just on integer intervals).

(a) 

The differential equation (C.3) is solved in any beginning course in ordinary differential equations, but you can do it easily with the MATLAB command **`dsolve`**. (Look up the syntax via online help.)

(b) Now find the solution assuming an initial value $x_0 = x(0)$ of x. Use the values $x_0 = 0, 0.25, \ldots, 2.0$. Graph the solutions and use your picture to justify the statement: "Regardless of $x_0 > 0$, the solution of (C.3) tends to the constant solution $x(t) \equiv 1$ in the long term."

The logistic model presumes two underlying features of population growth: (i) that ideally the population expands at a rate proportional to its current total (i.e., exponential growth – this corresponds to the x term on the right-hand side of (C.3)); and (ii) because of interactions between members of the species and natural limits to growth, unfettered exponential growth is held in check by the logistic term, given by the $-x^2$ expression in (C.3). Now assume that there are two species $x(t)$ and $y(t)$, competing for the same resources to survive. Then there will be another negative term in the differential equation that reflects the interaction between the species. The usual model presumes it to be proportional to the product of the two populations, and the larger the constant of proportionality, the more severe the interaction, as well as the resulting check on population growth.

(c) Here is a typical pair of differential equations that model the growth in population of two competing species $x(t)$ and $y(t)$:

$$\begin{aligned} \dot{x}(t) &= x - x^2 - 0.5xy \\ \dot{y}(t) &= y - y^2 - 0.5xy. \end{aligned} \tag{C.4}$$

The command **`dsolve`** can solve many pairs of ordinary differential equations – especially linear ones. But the mixture of quadratic terms in (C.4) makes it unsolvable symbolically, so we need to use a numerical

ODE solver as we did in the pendulum application. Using the commands in that application as a template, graph numerical solution curves to the system (C.4) for initial data

$$x(0) = 0 : 1/12 : 13/12$$
$$y(0) = 0 : 1/12 : 13/12.$$

(*Hint*: use **`axis`** to limit your view to the square $0 \le x, y \le 13/12$.)

(d) The picture you drew is called a *phase portrait* of the system. Interpret it. Explain the long-term behavior of any population distribution that starts with only one species present. Relate it to part (b). What happens in the long term if both populations are present initially? Is there an initial population distribution that remains undisturbed? What is it? Relate those numbers to the model (C.4).

(e) Now replace 0.5 in the model by 2, that is consider the new model

$$\begin{aligned} \dot{x}(t) &= x - x^2 - 2xy \\ \dot{y}(t) &= y - y^2 - 2xy. \end{aligned} \tag{C.5}$$

Draw the phase portrait. (Use the same initial data and viewing square.) Answer the same questions as in part (d). Do you see a special solution trajectory that emanates from near the origin and proceeds to the special fixed point? Do you see another trajectory from the upper right to the fixed point? What happens to all population distributions that do not start on these trajectories?

(f) Explain why model (C.4) is called "peaceful coexistence" and model (C.5) is called "doomsday." Now explain heuristically why the change in coefficient from 0.5 to 2 converts coexistence into doomsday.

12. Build a Simulink model corresponding to the pendulum equation

$$\ddot{x}(t) = -0.5\dot{x}(t) - 9.81\sin(x(t)) \tag{C.6}$$

from *The 360° Pendulum* in Chapter 10. You will need the Trigonometric Function block from the Math library. Use your model to redraw some of the phase portraits.

13. As you know, Galileo and Newton discovered that all bodies near the Earth's surface fall with the same acceleration g due to gravity, approximately 32.2 ft/sec^2. However, real bodies are also subjected to forces due to air resistance. If we take both gravity and air resistance into account, a moving ball can be modeled by the differential equation

$$\ddot{\mathbf{x}} = [0, -g] - c\,\|\dot{\mathbf{x}}\|\,\dot{\mathbf{x}}. \tag{C.7}$$

Here $\mathbf{x}$, a function of the time t, is the vector giving the position of the ball (the first coordinate is measured horizontally, the second one vertically), $\dot{\mathbf{x}}$ is the velocity vector of the ball, $\ddot{\mathbf{x}}$ is the acceleration of the ball, $\|\dot{\mathbf{x}}\|$ is the magnitude of the velocity, that is, the speed, and c is a constant depending on the surface characteristics and mass of the ball and the density of the air. (We are neglecting the lift force that comes from the ball's rotation, which can also play a major role in some situations, for instance in analyzing the path of a curve ball, as well as forces due to wind currents.) For a baseball, the constant c turns out to be approximately 0.0017, assuming that distances are measured in feet and time is measured in seconds. (See, for example, Chapter 18, "Balls and Strikes and Home Runs," in *Towing Icebergs, Falling Dominoes, and Other Adventures in Applied Mathematics*, by Robert Banks, Princeton University Press, Princeton, 1998.) Build a Simulink model corresponding to equation (C.7), and use it to study the trajectory of a batted baseball. Here are a few hints. Represent $\ddot{\mathbf{x}}$, $\dot{\mathbf{x}}$, and $\mathbf{x}$ as vector signals, joined by two Integrator blocks. The quantity $\ddot{\mathbf{x}}$, according to (C.7), should be computed from a Sum block with two vector inputs. One should be a Constant block with the vector value $[0, -32.2]$, representing gravity, and the other should represent the drag term on the right-hand side of equation (C.7), computed from the value of $\dot{\mathbf{x}}$. You should be able to change one of the parameters to study what happens both with and without air resistance (the cases of $c = 0.0017$ and $c = 0$, respectively). Attach the output to an XY Graph block, with the parameters x-min = 0, y-min = 0, x-max = 500, y-max = 150, so you can see the path of the ball out to a distance of 500 feet from home plate and up to a height of 150 feet.

(a) Let $\mathbf{x}(0) = [0, 4]$ and $\dot{\mathbf{x}}(0) = [80, 80]$. (This corresponds to the ball starting at $t = 0$ from home plate, 4 feet off the ground, with the horizontal and vertical components of its velocity both equal to 80 ft/sec. This corresponds to a speed off the bat of about 77 mph, which is not unrealistic.) How far (approximately – you can read this from your XY Graph output) will the ball travel before it hits the ground, both with and without air resistance? About how long will it take the ball to hit the ground, and how fast will the ball be traveling at that time (again, both with and without air resistance)? (The last parts of the question are relevant for outfielders.)

(b) Suppose that a game is played in Denver, Colorado, where because of thinning of the atmosphere due to the high altitude, c is only 0.0014. How far will the ball travel now (given the same initial velocity as in (a))?

(c) (This is not a MATLAB problem.) Estimate from a comparison of your answers to (a) and (b) what effect altitude might have on the team batting average and team earned run average of the Colorado Rockies.

14. ☆ Write a function M-file (with no arguments or outputs) that scans the current directory for the most recently modified M-file and opens it in the Editor/Debugger. If the current directory contains no M-files, your M-file should produce an error message. Ideally, you should try to write the M-file so it works

in either Windows or UNIX, but at least make it work with your own operating system.

15. ☆ Consider a sequence of complex numbers generated from a starting value z_0 by the rule $z_{n+1} = z_n^2 - 0.75$. For some values of z_0, the sequence of numbers will diverge to infinity as n increases, but for other values of z_0, the sequence will remain inside a bounded region forever. The boundary of the set consisting of the latter values of z_0 is called the *Julia set* for the function $f(z) = z^2 - 0.75$. Use **`image`** or **`imagesc`** to draw a picture of this Julia set. (*Hint*: the Julia set lies within the region where the real part of z_0 is between -2 and 2, and its imaginary part is between -1.5 and 1.5. Form a grid of z_0 values within this rectangle (we suggest using roughly a 300×400 array) and calculate the corresponding values of $z_1, z_2, \ldots, z_n$ for some n. Color the pixel of each z_0 according to how large the corresponding z_n is. The larger you make n, the more accurate (and interesting) the picture will be, but the computation will also take longer, so start with a relatively small value of n and work your way up.)

Chapter 11

Troubleshooting

In this chapter, we offer advice for dealing with some common problems that you may encounter. We also list and describe the most common mistakes that MATLAB users make. Finally, we offer some simple but useful techniques for debugging your M-files.

Common Problems

Problems manifest themselves in various ways: totally unexpected or plainly wrong output appears; MATLAB produces an error message (or at least a warning); MATLAB refuses to process an input line; something that worked earlier stops working; or, worst of all, the computer freezes. Fortunately, these problems are often caused by several easily identifiable and correctable mistakes. What follows is a description of some common problems, together with a presentation of likely causes, suggested solutions, and illustrative examples. We also refer to places in the book where related issues are discussed.

Here is a list of the problems:

- Wrong or unexpected output
- Syntax error
- Spelling error
- Error or warning messages when plotting
- A previously saved M-file evaluates differently
- Computer won't respond.

Wrong or Unexpected Output

There are many possible causes for this problem, but they are likely to be among the following.

CAUSE: **Forgetting to clear or reset variables.**

SOLUTION: Clear or initialize variables before using them, especially in a long session.

☞ *See* Variables and Assignments *in Chapter 2*.

EXAMPLE: Suppose that you set up a loop like

```
>> for i = 1:10
...
end
```

and later in your session you enter the complex number **2+i**; then in fact you have entered the number 12, because at the end of the loop the variable **i** is set equal to 10. In order to restore it to its intrinsic value $\sqrt{-1}$, you need to type **clear i**.

CAUSE: **Conflicting definitions.**

SOLUTION: Do not use the same name for two different functions or variables, and, in particular, do not overwrite the names of any of MATLAB's built-in functions.

You can accidentally mask one of MATLAB's built-in M-files either with your own M-file of the same name or with a variable (including, perhaps, an inline or anonymous function). When unexpected output occurs and you think this might be the cause, it helps to use **which** to find out what M-file, function, or variable is actually being referenced. The preceding example illustrates this problem, but here is perhaps a more extreme example.

EXAMPLE:

```
>> plot = gcf;
>> x = -2:0.1:2;
>> plot(x, x.^2)
??? Subscript indices must either be real positive integers or
logicals.
```

What's wrong, of course, is that **plot** has been masked by a variable with the same name. You could detect this with

```
>> which plot
plot is a variable.
```

If you type **clear plot** and execute the **plot** command again, you'll get a picture of the desired parabola. A more subtle example could occur if you did this on purpose, not thinking you would use **plot**, and then called some other graphics script M-file that uses it indirectly.

CAUSE: **Not keeping track of ans.**

SOLUTION: Assign variable names to any output that you intend to use.

If you decide at some point in a session that you wish to refer to prior output that was unnamed, then give the output a name, and execute the command again. (The UP-ARROW key or Command History Window is useful for recalling the command to edit it.) Do not rely on **ans** since it is likely to be overwritten before you execute the command that references the prior output.

CAUSE: **Improper use of built-in functions.**

SOLUTION: Always use the names of built-in functions exactly as MATLAB specifies them; always enclose inputs in parentheses, not brackets and not braces; always list the inputs in the required order.

☞ *See* Managing Variables *and* Online Help *in Chapter 2*

CAUSE: **Inattention to precedence of arithmetic operations.**

SOLUTION: Use parentheses liberally and correctly when entering arithmetic or algebraic expressions.

EXAMPLE: MATLAB, like any calculator, first exponentiates, then divides and multiplies, and finally adds and subtracts, unless a different order is specified by using parentheses. So if you attempt to compute $5^{2/3} - 25/(2 * 3)$ by typing

```
>> 5^2/3 - 25/2*3
```

```
ans =
   -29.1667
```

the answer MATLAB produces is not what you intended because **5** is raised to the power **2** before the division by **3**, and **25** is divided by **2** before the multiplication by **3**. Here is the correct calculation:

```
>> 5^(2/3) - 25/(2*3)
```

```
ans =
   -1.2426
```

Syntax Error

CAUSE: **Mismatched parentheses, quote marks, braces, or brackets.**

SOLUTION: Look carefully at the input line to find a missing or an extra delimiter.

MATLAB usually catches this kind of mistake. In addition, the MATLAB Desktop automatically highlights matching delimiters as you type, and color-codes strings (expressions enclosed in single quotes) so that you can see where they begin and end. In the Command Window of MATLAB 5 and earlier versions, however, you have to hunt for matching delimiters by hand.

CAUSE: **Wrong delimiters: using parentheses in place of brackets, or vice versa, and so on.**

SOLUTION: Remember the basic rules about delimiters in MATLAB.

Parentheses are used both for grouping arithmetic expressions and for enclosing inputs to a MATLAB command, an M-file, or an inline function. They are also used for referring to an entry in a matrix. *Square brackets* are used for defining vectors or matrices. *Single quote marks* are used for defining strings.

EXAMPLE: The following illustrates what can happen if you don't follow these rules:

```
>> X = -1:0.01:1;
>> X[1]
```

```
??? X[1]
     |
Error: Unbalanced or misused parentheses or brackets.
```

```
>> A = (0,1,2)

??? A = (0,1,2)
          |
Error: Incomplete or misformed expression or statement.
```

These examples are fairly straightforward to understand; in the first case, **`X(1)`** was intended, and in the second case, **`A = [0,1,2]`** was intended. But here's a trickier example:

```
>> abs 3

ans =
   51
```

Here there's no error message, but if one looks closely, one discovers that MATLAB has printed out the absolute value of 51, not of 3. The explanation is as follows: any time a MATLAB command is followed by a space and then an argument to the command (as in the construct **`clear x`**), the argument is always interpreted as a string. Thus MATLAB has interpreted **`3`** not as the number 3, but as the string **`'3'`**! And sure enough, one discovers:

```
>> char(51)

ans =
3
```

In other words, in MATLAB's encoding scheme, the string **`'3'`** is stored as the number **`51`**, which is why **`abs 3`** (or also **`abs('3')`**) produces as output the absolute value of 51.

Braces or *curly brackets* are used less often than either parentheses or square brackets, and are usually not needed by beginners. Their main use is with cell arrays. One example to keep in mind is that, if you want an M-file to take a variable number of inputs or produce a variable number of outputs, then these are stored in the cell arrays **`varargin`** and **`varargout`**, and braces are used to refer to the cells of these arrays. Similarly, **`case`** is sometimes used with braces in the middle of a **`switch`** construct. If you want to construct an array of strings, then it has to be done with braces, since brackets when applied to strings are interpreted as concatenation.

EXAMPLE:

```
>> {'a', 'b'}

ans =
   'a'    'b'

>> ['a', 'b']

ans =
ab
```

CAUSE: **Improper use of arithmetic symbols.**

SOLUTION: When you encounter a syntax error, review your input line carefully for mistakes in typing.

EXAMPLE: If the user, intending to compute 2 times -4, inadvertently switches the symbols, the result is

```
>> 2 - * 4
??? 2 - * 4
        |
Error: Missing variable or function.
```

Here the vertical bar highlights the place where MATLAB believes the error is located. In this case, the actual error is earlier in the input line.

Spelling Error

CAUSE: **Using uppercase instead of lowercase letters in MATLAB commands, or misspelling the command.**

SOLUTION: Fix the spelling.

Consider the following accidental misspellings.

EXAMPLE:

```
>> Sin(pi/2)
??? Undefined command/function 'Sin'.

>> ffzero(@(x) x^2 - 3, 1)
??? Undefined command/function 'ffzero'.

>> fzero(@(x) x^2 - 3, 1)
ans =
   1.7321
```

Error or Warning Messages When Plotting

CAUSE: **There are several possible explanations, but usually the problem is the wrong type of input for the plotting command chosen.**

SOLUTION: Carefully follow the examples in the **`help`** lines of the plotting command, and pay attention to the error and warning messages.

EXAMPLE:

```
>> X = -3:0.05:3;
>> plot(X, X.^(1/3))
Warning: imaginary parts of complex X and/or Y arguments
ignored.
```

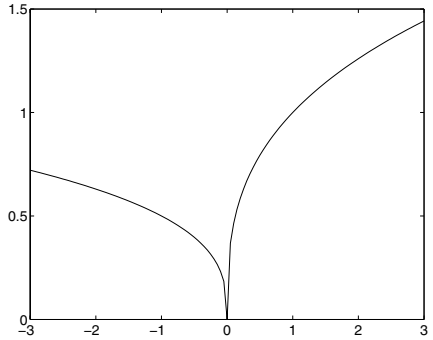


Figure 11.1. A Spurious Graph of $y = x^{1/3}$.

Figure 11.1 is not the graph of $y = x^{1/3}$ on the interval $-3 \leq x \leq 3$. Actually, the graph to the right of the origin looks correct, but to the left of the origin the graph is certainly not right. The warning message provides a clue. What has happened is that MATLAB is plotting the real part of $x^{1/3}$, but for a different branch of the multi-valued function than the one we want. MATLAB is interpreting $x^{1/3}$ for x negative as meaning $[(1 + \sqrt{3}i)/2]|x|^{1/3}$. In order to fix the problem, we must specify the function in MATLAB more carefully. The correct graph in Figure 11.2 results from

```
>> plot(X, sign(X).*abs(X).^(1/3))
```

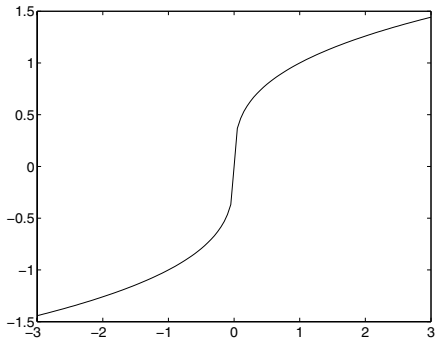


Figure 11.2. Correct Graph of $y = x^{1/3}$.

A Previously Saved M-File Evaluates Differently

One of the most frustrating problems you may encounter occurs when a previously saved M-file, which you are sure is in good shape, won't evaluate or evaluates incor-

rectly when opened in a new session.

CAUSE: **Change in the sequence of evaluation, or failure to initialize variables.**

SOLUTION: Make sure to clear or initialize variables that are not inputs to the M-file.

EXAMPLE: The problem is illustrated by the following simple, but poorly designed, program. To compute $n!$ (n-factorial), a user writes the following script M-file:

```
%% computing n!
for k = 1:n
   f = f*k;
end
f
```

thinking he will initialize $f = 1$ outside the file when choosing n. If he fails to initialize f the second time he runs the file, he is in for a nasty surprise. Can you see why?

Computer Won't Respond

CAUSE: **MATLAB is caught in a very large calculation, or some other calamity has occurred, which has caused it to fail to respond. Perhaps you are using an array that is too large for your computer memory to handle.**

SOLUTION: Abort the calculation with CTRL+C.

If overuse of computer memory is the problem, try to redo your calculation using smaller arrays, e.g., by using fewer grid points in a 3D plot, or by breaking a large vectorized calculation into smaller pieces using a loop. Clearing large arrays from your Workspace may help too.

EXAMPLE: You'll know it when you see it!

The Most Common Mistakes

The most common mistakes are all accounted for in the causes of the problems described earlier. But to help you prevent these mistakes, we compile them here in a single list to which you can refer periodically. Doing so will help you to establish "good MATLAB habits."

- Forgetting to clear or initialize variables
- Improper use of built-in functions
- Inattention to the order of precedence of arithmetic operations
- Improper use of arithmetic symbols
- Mismatched delimiters
- Using the wrong delimiters

- Plotting the wrong kind of object
- Using uppercase instead of lowercase letters in MATLAB commands, or misspelling commands.

Debugging Techniques

Now that we have discussed the most common mistakes, it's time to discuss how to debug your M-files, and how to locate and fix those pesky problems that don't fit into the neat categories above.

If one of your M-files is not working the way you expected, perhaps the easiest thing you can do to debug it is to insert the command **keyboard** somewhere in the middle. This temporarily suspends (but does not stop) execution and returns command to the keyboard, where you are given a special prompt with a **K** in it. You can execute whatever commands you want at this point (for instance, to examine some of the variables). To return to execution of the M-file, type **return** or **dbcont**, short for "debug continue."

A more systematic way to debug M-files is to use the debugging features of the MATLAB M-file Editor/Debugger. Start by going to the menu item **Tools**:**Check Code with M-Lint**. This checks the code of the M-file for common mistakes and syntax errors and prints out a report specifying where potential problems are located. (You can do the same thing from the Command Window with the command **mlint** followed by a space and the name of the file.) Next, use the debugger to insert "breakpoints" in the file. Usually you would do this with the **Breakpoints** menu or with the "Set/clear breakpoint" icon at the top of the Editor/Debugger window, but you can also do it from the Command Window with the command **dbstop**. (See its online help.) Once a breakpoint has been inserted in the M-file, you will see a little red dot next to the appropriate line in the Editor/Debugger. (An example is illustrated in Figure 11.8 below.) Then, when you call the M-file, execution will stop at the breakpoint, and, just as in the case of **keyboard**, control will return to the Command Window, where you will be given a special prompt with a **K** in it. Again, when you are ready to resume execution of the M-file, type **dbcont**. When you have finished the debugging process, **dbclear** "clears" the breakpoint from the M-file.

Let's illustrate these techniques with a real example. Suppose that you want to construct a function M-file that takes as input two expressions f and g (given either as symbolic expressions or as strings) and two numbers a and b, plots the functions f and g between $x = a$ and $x = b$, and shades the region in between them. As a first try, you might start with the nine-line function M-file `shadecurves.m` given as follows:

```
function shadecurves(f, g, a, b)
%SHADECURVES  Draws the region between two curves
%   SHADECURVES(f, g, a, b) takes strings or expressions f
%   and g, interprets them as functions, plots them between
%   x = a and x = b, and shades the region in between.
%   Example: shadecurves('sin(x)', '-sin(x)', 0, pi)
```

```
ffun = inline(vectorize(f)); gfun = inline(vectorize(g));
xvals = a:(b - a)/50:b;
plot([xvals, xvals], [ffun(xvals), gfun(xvals)])
```

Let's run the M-file with the data specified in the help lines:

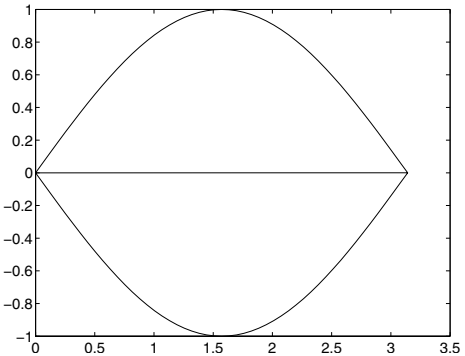


Figure 11.3. A First Attempt at Shading the Region between $\sin x$ and $-\sin x$.

The graph in Figure 11.3 was obtained by executing

```
>> shadecurves('sin(x)', '-sin(x)', 0, pi)
```

or

```
>> syms x; shadecurves(sin(x), -sin(x), 0, pi)
```

This is not really what we wanted; the figure we seek is shown in Figure 11.4.

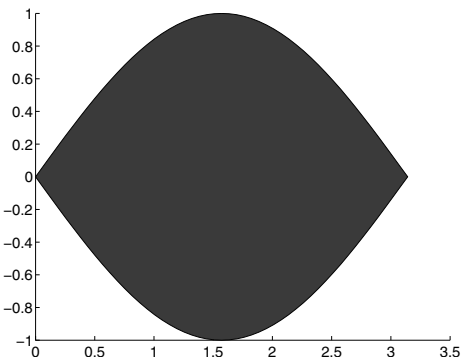


Figure 11.4. The Shaded Region between $\sin x$ and $-\sin x$.

To begin to determine what went wrong, let's try a different example, say

```
>> syms x; shadecurves(x^2, sqrt(x), 0, 1); axis square
```

Now we get the output shown in Figure 11.5.

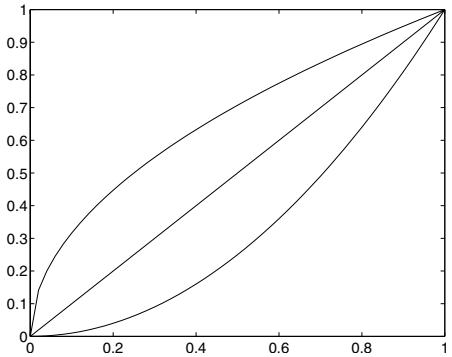


Figure 11.5. A First Attempt at Shading the Region between x^2 and $\sqrt{x}$.

It's not too hard to figure out why our regions aren't shaded; that's because we used **`plot`** (which plots curves) instead of **`patch`** (which plots *filled* patches). So that suggests we should try changing the last line of the M-file to:

```
patch([xvals, xvals], [ffun(xvals), gfun(xvals)])
```

That gives the error message:

```
??? Error using ==> patch
Not enough input arguments.

Error in ==> shadecurves at 9
patch([xvals, xvals], [ffun(xvals), gfun(xvals)])
```

so we go back and try

```
>> help patch
```

to see whether we can get the syntax right. The help lines indicate that **`patch`** requires a third argument, the color (in RGB coordinates) with which our patch is to be filled. So we change our final line to, for instance,

```
patch([xvals, xvals], [ffun(xvals), gfun(xvals)], [.5,0,.7])
```

That gives us now as output to

```
>> syms x; shadecurves(x^2, sqrt(x), 0, 1); axis square
```

the picture shown in Figure 11.6.

That's better, but still not quite right, because we can see a mysterious diagonal line down the middle. Not only that, but if we try

```
>> syms x; shadecurves(x^2, x^4, -1.5, 1.5)
```

we now get the bizarre picture shown in Figure 11.7.
There aren't a lot of lines in the M-file, and lines 7 and 8 seem OK, so the problem must be with the last line. We need to reread the online help for **`patch`**. It indicates that **`patch`** draws a filled two-dimensional polygon defined by the vectors X and

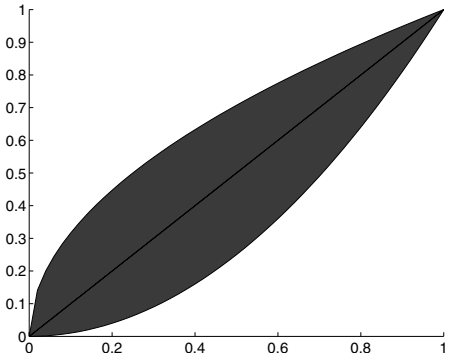


Figure 11.6. A Second Attempt at Shading the Region between x^2 and $\sqrt{x}$.

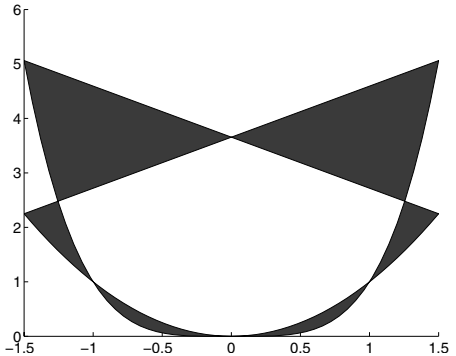


Figure 11.7. A First Attempt at Shading the Region between x^2 and x^4.

Y, which are its first two inputs. A way to see how this is working is to change the "50" in line 9 of the M-file to something much smaller, say 5, and then insert a breakpoint in the M-file before line 9. At this point, our M-file in the Editor/Debugger window now looks like Figure 11.8. Note the large dot to the left of the last line, indicating the breakpoint. When we run the M-file with the same input, we now obtain in the Command Window a **K>>** prompt. At this point, it is logical to try to list the coordinates of the points that are the vertices of our filled polygon, so we try

```
K>> [[xvals, xvals]', [ffun(xvals), gfun(xvals)]']

ans =
   -1.5000    2.2500
   -0.9000    0.8100
   -0.3000    0.0900
    0.3000    0.0900
    0.9000    0.8100
    1.5000    2.2500
   -1.5000    5.0625
```

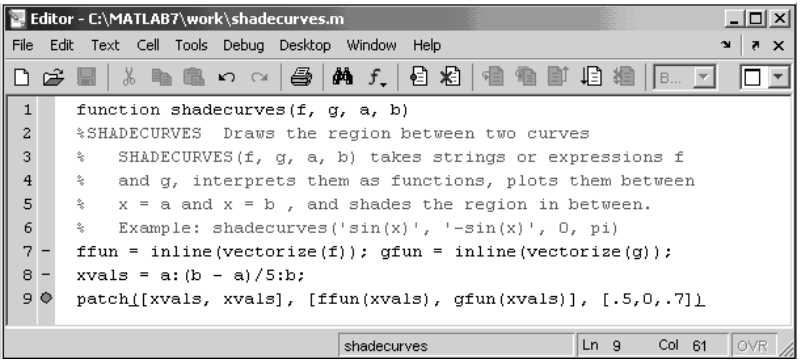


Figure 11.8. The Editor/Debugger.

```
-0.9000    0.6561
-0.3000    0.0081
 0.3000    0.0081
 0.9000    0.6561
 1.5000    5.0625
```

If we now type

```
K>> dbcont
```

we see in the figure window what is shown in Figure 11.9 below.

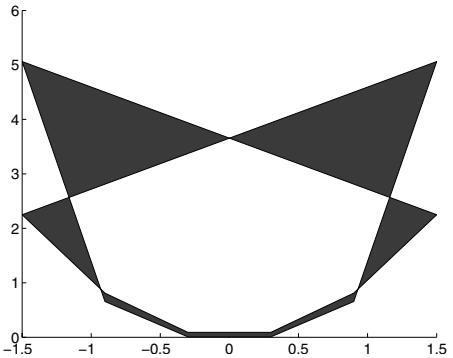


Figure 11.9. A Second Attempt at Shading the Region between x^2 and x^4.

Finally we can understand what is going on; MATLAB has "connected the dots" using the points whose coordinates were just printed out. In particular, MATLAB has drawn a line from the point (1.5, 2.25) to the point (−1.5, 5.0625). This is not what we wanted; we wanted MATLAB to join the point (1.5, 2.25) on the curve $y = x^2$ to the point (1.5, 5.0625) on the curve $y = x^4$. We can fix this by reversing the order of the x-coordinates at which we evaluate the second function g. So, letting **slavx**

denote **xvals** reversed, we correct our M-file to read:

```
function shadecurves(f, g, a, b)
%SHADECURVES  Draws the region between two curves
%   SHADECURVES(f, g, a, b) takes strings or expressions f
%   and g, interprets them as functions, plots them between
%   x = a and x = b, and shades the region in between.
%   Example: shadecurves('sin(x)', '-sin(x)', 0, pi)
ffun = inline(vectorize(f)); gfun = inline(vectorize(g));
xvals = a:(b - a)/50:b; slavx = b:(a - b)/50:a;
patch([xvals, slavx], [ffun(xvals), gfun(slavx)], [.5,0,.7])
```

Now it works properly. Sample output from this M-file is shown in Figure 11.4. Try it out on the other examples we have discussed, or on others of your choice.

Solutions to the Practice Sets

Solutions to Practice Set A: Algebra and Arithmetic

1.

(a)

```
1111 - 345

ans =

   766
```

(b)

```
format long; [exp(14), 382801*pi]

ans =

   1.0e+06 *

   1.20260428416478   1.20260480938683
```

The second number is bigger.

(c)

```
[2709/1024; 10583/4000; 2024/765]

ans =

   2.64550781250000
   2.64575000000000
   2.64575163398693


sqrt(7)

ans =

   2.64575131106459
```

The third, that is 2024/765, is the best approximation.

2.

(a)

```
cosh(0.1)

ans =

   1.00500416805580
```

(b)

```
log(2)

ans =

   0.69314718055995
```

(c)

```
atan(1/2)

ans =

   0.46364760900081

format short
```

3.

```
[x, y, z] = solve('3*x + 4*y + 5*z = 2', ...
    '2*x - 3*y + 7*z = -1', 'x - 6*y + z = 3', ...
    'x', 'y', 'z')

x =

241/92


y =

-21/92
```

```
z =

-91/92
```

Now we'll check the answer.

```
A = [3, 4, 5; 2, -3, 7; 1, -6, 1]; A*[x; y; z]

ans =

  2
 -1
  3
```

It works!

4.

```
[x, y, z] = solve('3*x - 9*y + 8*z = 2', ...
    '2*x - 3*y + 7*z = -1', 'x - 6*y + z = 3', ...
    'x', 'y', 'z')

x =

22/5+39/5*y



y =

y



z =

-9/5*y-7/5
```

We get a one-parameter family of answers depending on the variable y. In fact the three planes determined by the three linear equations are not independent, because the first equation is the sum of the second and third. The locus of points that satisfy the three equations is not a point, the intersection of three independent planes, but rather a line, the intersection of two distinct planes. Once again we check.

```
B = [3, -9, 8; 2, -3, 7; 1, -6, 1]; B*[x; y; z]
```

```
ans =

  2
 -1
  3
```

5.

```
syms x y; factor(x^4 - y^4)
```

```
ans =

(x-y)*(x+y)*(x^2+y^2)
```

6.

(a)

```
simplify(1/(1 + 1/(1 + 1/x)))
```

```
ans =

(x+1)/(2*x+1)
```

(b)

```
simplify(cos(x)^2 - sin(x)^2)
```

```
ans =

2*cos(x)^2-1
```

Let's try **`simple`** instead.

```
[better, how] = simple(cos(x)^2 - sin(x)^2)
```

```
better =

cos(2*x)



how =

combine(trig)
```

7.

```
3^301

pretty(sym('3')^301)
```

```
ans =

  4.1067e+143


  41067443717576512797397808214626494789939108687601230941444057023510699 15\
        324972297814006184670668241647514533217939821284405381982970873236 9\
        8003
```

But note the following:

```
sym('3^301')
```

```
ans =

3^301
```

This does not work because **sym**, by itself, does not cause an evaluation.

8.

(a)

```
solve('67*x + 32 = 0', 'x')
```

```
ans =

-32/67
```

(b)

```
vpa(ans, 15)
```

```
ans =

-.477611940298507
```

(c)

```
pretty(solve('x^3 + p*x + q = 0', 'x'))
```

```
     [                        1/3       p                    ]
     [                 1/6 %1    - 2 -----                   ]
     [                                 1/3                   ]
     [                               %1                      ]
     [                                                       ]
     [         1/3     p            1/2 /      1/3       p  \]
     [- 1/12 %1    + ----- + 1/2 I 3    |1/6 %1    + 2 -----|]
     [                 1/3               |                1/3|]
     [               %1                  \              %1   /]
     [                                                       ]
     [         1/3     p            1/2 /      1/3       p  \]
     [- 1/12 %1    + ----- - 1/2 I 3    |1/6 %1    + 2 -----|]
     [                 1/3               |                1/3|]
     [               %1                  \              %1   /]

                                       3       2 1/2
                %1 := -108 q + 12 (12 p  + 81 q )
```

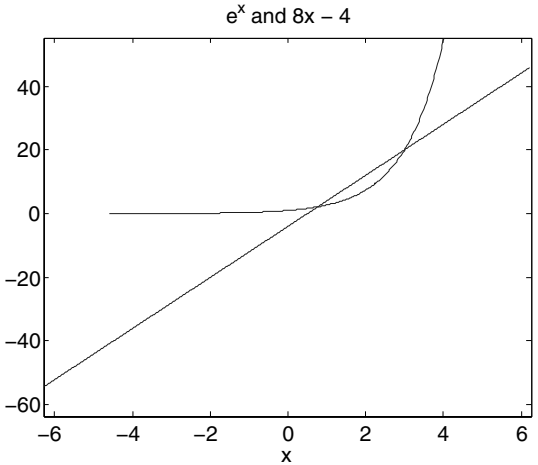
Note how **`pretty`** separates out the subexpression

$$-108q + 12\sqrt{12p^3 + 81q^2},$$

which occurs in the formulas for all three of the roots, and gives it a name. This expression is closely related to the discriminant of the cubic equation.

(d)

```
ezplot('exp(x)'); hold on; ezplot('8*x - 4')
title('e^x and 8x - 4')
```

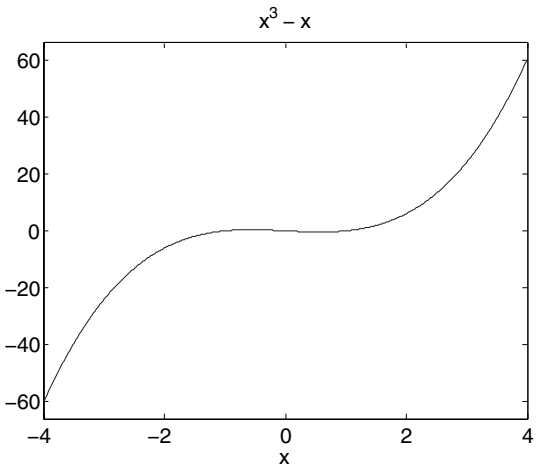



```
fzero('exp(x) - 8*x + 4', 1)
ans =

    0.7700

fzero('exp(x) - 8*x + 4', 3)
ans =

    2.9929
```

9.

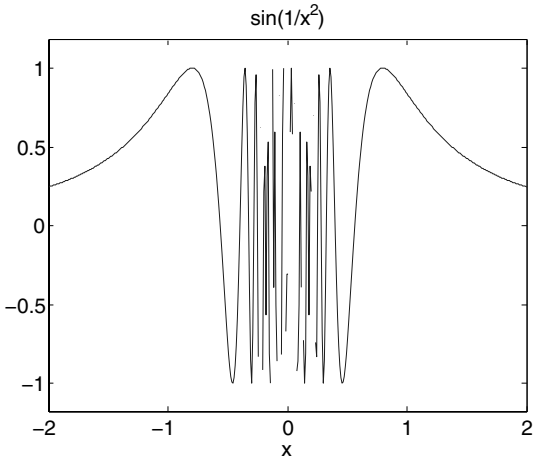
(a)

```
hold off; ezplot('x^3 - x', [-4 4])
```



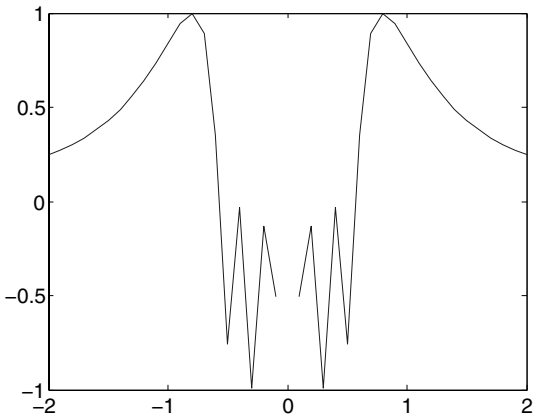
(b)

```
ezplot('sin(1/x^2)', [-2 2])
```




```
X = -2:0.1:2; plot(X, sin(1./X.^2))
```

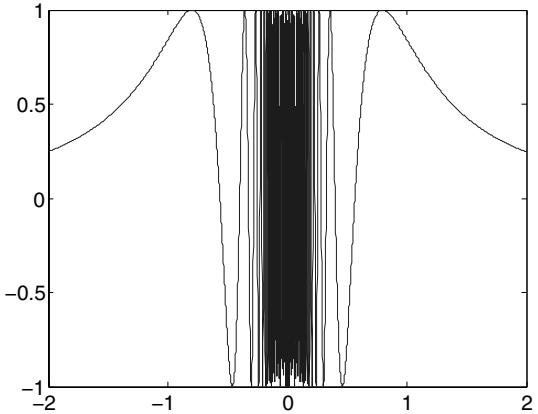
```
Warning: Divide by zero.
```



Both pictures are incomplete. Let's see what happens if we refine the mesh.

```
X = -2:0.001:2; plot(X, sin(1./X.^2))
```

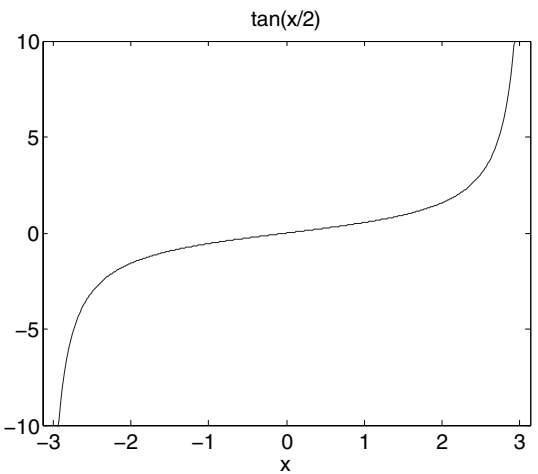
```
Warning: Divide by zero.
```



This is better, but because of the wild oscillation near $x = 0$, neither **`plot`** nor **`ezplot`** gives a totally accurate graph of the function.

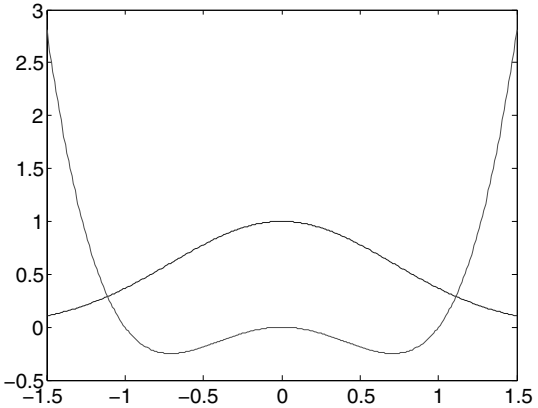
(c)

```
ezplot('tan(x/2)', [-pi pi]); axis([-pi, pi, -10, 10])
```



(d)

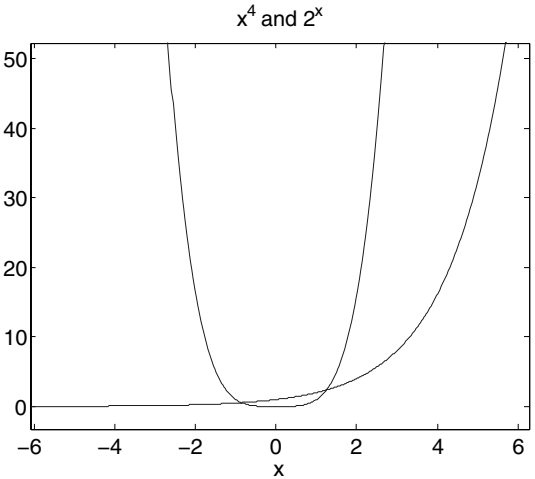
```
X = -1.5:0.01:1.5;
plot(X, exp(-X.^2), X, X.^4 - X.^2)
```



10.

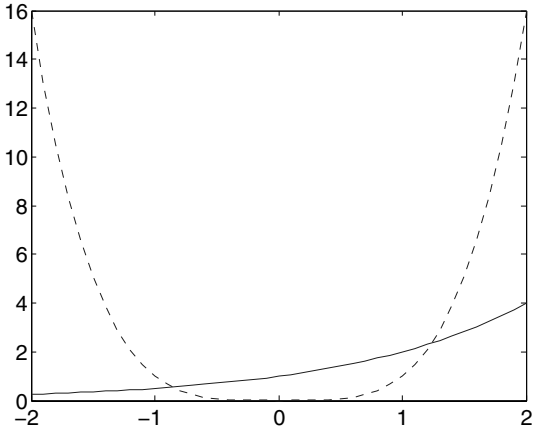
Let's plot 2^x and x^4 and look for points of intersection. We plot them first with **`ezplot`** just to get a feel for the graph.

```
ezplot('x^4'); hold on; ezplot('2^x'); hold off
title('x^4 and 2^x')
```



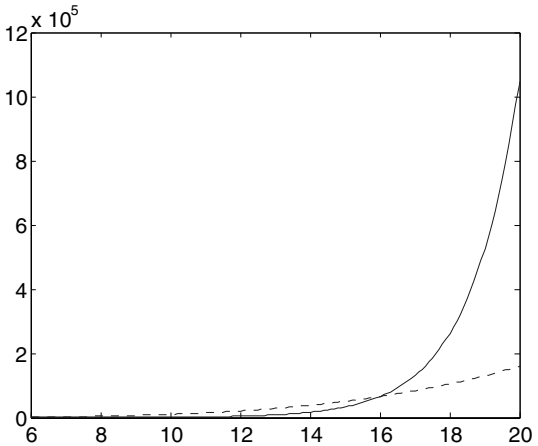
Note the large vertical range. We learn from the plot that there are no points of intersection between 2 and 6 or between -6 and -2; but there are apparently two points of intersection between -2 and 2. Let's change to **`plot`** now and focus on the interval between -2 and 2. We'll plot x^4 dashed.

```
X = -2:0.1:2; plot(X, 2.^X);
hold on; plot(X, X.^4, '--'); hold off
```



We see that there are points of intersection near -0.9 and 1.2. Are there any other points of intersection? To the left of 0, 2^x is always less than 1, whereas x^4 goes to infinity as x goes to negative infinity. On the other hand, both x^4 and 2^x go to infinity as x goes to infinity, so the graphs may cross again to the right of 6. Let's check.

```
X = 6:0.1:20; plot(X, 2.^X);
hold on; plot(X, X.^4, '--'); hold off
```



We see that they do cross again, near $x = 16$. If you know a little calculus, you can show that the graphs never cross again (by taking logarithms, for example), so we have found all the points of intersection. Now let's use **`fzero`** to find these points of intersection numerically. This command looks for a solution near a given starting point. To find the three different points of intersection we will have to use three different starting points. The above graphical analysis suggests appropriate starting points.

```
r1 = fzero('2^x - x^4', -0.9)
r2 = fzero('2^x - x^4', 1.2)
r3 = fzero('2^x - x^4', 16)
```

```
r1 =

   -0.8613


r2 =

    1.2396


r3 =

    16
```

Let's check that these "solutions" satisfy the equation.

```
subs('2^x - x^4', 'x', r1)
subs('2^x - x^4', 'x', r2)
subs('2^x - x^4', 'x', r3)
```

```
ans =

   1.1102e-16


ans =

  -8.8818e-16


ans =

     0
```

So **r1** and **r2** very nearly satisfy the equation, and **r3** satisfies it exactly. It is easily seen that 16 is a solution. It is also interesting to try **solve** on this equation.

```
symroots = solve('2^x - x^4 = 0')
```

```
symroots =

    -4*lambertw(-1/4*log(2))/log(2)
 -4*lambertw(-1,-1/4*log(2))/log(2)
  -4*lambertw(-1/4*i*log(2))/log(2)
     -4*lambertw(1/4*log(2))/log(2)
   -4*lambertw(1/4*i*log(2))/log(2)
```

The function **`lambertw`** is a "special function" that is built into MATLAB. You can learn more about it by typing **`help lambertw`**. Let's see the decimal values of the solutions.

```
double(symroots)
```

```
ans =

   1.2396
  16.0000
  -0.1609 + 0.9591i
  -0.8613
  -0.1609 - 0.9591i
```

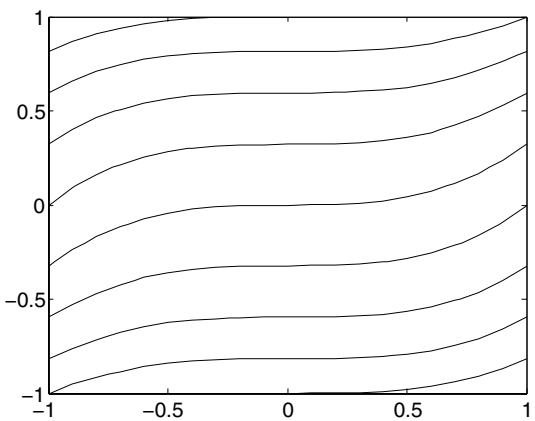
In fact we get the three real solutions already found and two complex solutions. Only the real solutions correspond to points where the graphs intersect.

Solutions to Practice Set B: Calculus, Graphics, and Linear Algebra

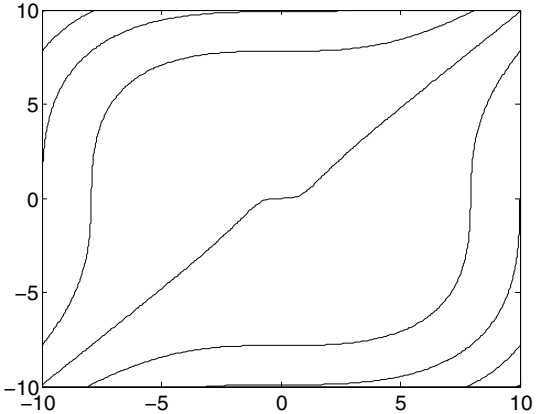
1.

(a)

```
[X, Y] = meshgrid(-1:0.1:1, -1:0.1:1);
contour(X, Y, 3*Y + Y.^3 - X.^3, 'k')
```

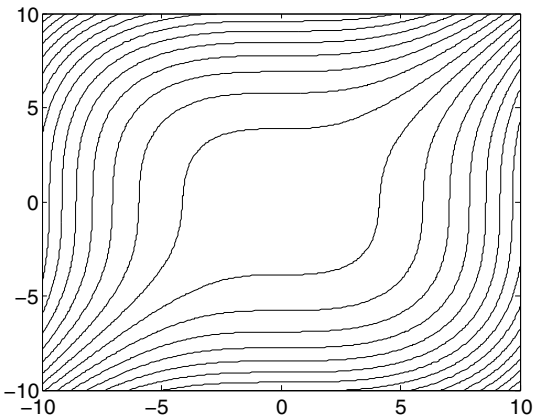



```
[X, Y] = meshgrid(-10:0.1:10, -10:0.1:10);
contour(X, Y, 3*Y + Y.^3 - X.^3, 'k')
```



Here is a plot with more level curves.

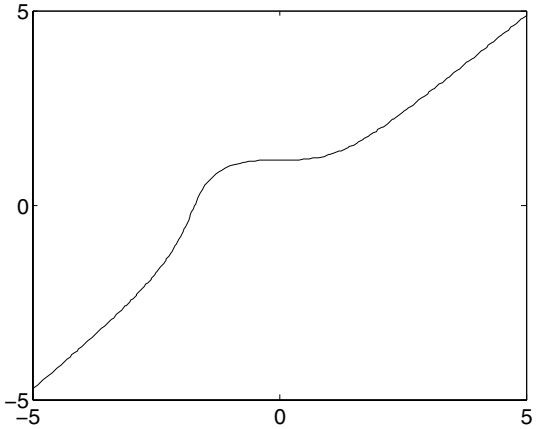
```
[X, Y] = meshgrid(-10:0.1:10, -10:0.1:10);
contour(X, Y, 3*Y + Y.^3 - X.^3, 30, 'k')
```



(b)

Now we plot the level curve through 5.

```
[X, Y] = meshgrid(-5:0.1:5, -5:0.1:5);
contour(X, Y, 3.*Y + Y.^3 - X.^3, [5 5], 'k')
```

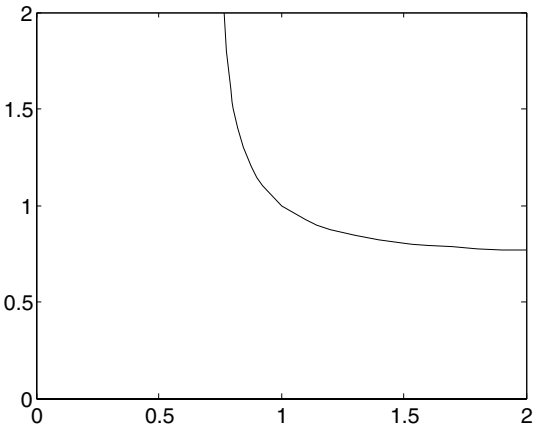


(c)

We note that $f(1,1) = 0$, so the appropriate commands to plot the level curve of f through the point $(1,1)$ are

```
[X, Y] = meshgrid(0:0.1:2, 0:0.1:2);
contour(X, Y, Y.*log(X) + X.*log(Y), [0 0], 'k')
```

```
Warning: Log of zero.
Warning: Log of zero.
```



2.

We find the derivatives of the given functions:

```
syms x r
```

(a)

```
diff(6*x^3 - 5*x^2 + 2*x - 3, x)

ans =

18*x^2-10*x+2
```

(b)

```
diff((2*x - 1)/(x^2 + 1), x)

ans =

2/(x^2+1)-2*(2*x-1)/(x^2+1)^2*x


simplify(ans)

ans =

-2*(x^2-1-x)/(x^2+1)^2
```

(c)

```
diff(sin(3*x^2 + 2), x)

ans =

6*cos(3*x^2+2)*x
```

(d)

```
diff(asin(2*x + 3), x)

ans =

1/(-2-x^2-3*x)^(1/2)
```

(e)

```
diff(sqrt(1 + x^4), x)
```

```
ans =

2/(1+x^4)^(1/2)*x^3
```

(f)

```
diff(x^r, x)
```

```
ans =

x^r*r/x
```

(g)

```
diff(atan(x^2 + 1), x)
```

```
ans =

2*x/(1+(x^2+1)^2)
```

3.

We compute the following integrals.

(a)

```
int(cos(x), x, 0, pi/2)
```

```
ans =

1
```

(b)

```
int(x*sin(x^2), x)

ans =

-1/2*cos(x^2)
```

To check the indefinite integral, we just differentiate.

```
diff(-cos(x^2)/2, x)

ans =

x*sin(x^2)
```

(c)

```
int(sin(3*x)*sqrt(1 - cos(3*x)), x)

ans =

2/9*(1-cos(3*x))^(3/2)
```

```
diff(ans, x)

ans =

sin(3*x)*(1-cos(3*x))^(1/2)
```

(d)

```
int(x^2*sqrt(x + 4), x)

ans =

2/7*(x+4)^(7/2)-16/5*(x+4)^(5/2)+32/3*(x+4)^(3/2)
```

```
diff(ans, x)

ans =

(x+4)^(5/2)-8*(x+4)^(3/2)+16*(x+4)^(1/2)


simplify(ans)

ans =

x^2*(x+4)^(1/2)
```

(e)

```
int(exp(-x^2), x, -Inf, Inf)

ans =

pi^(1/2)
```

4.

(a)

```
syms x; int(exp(sin(x)), x, 0, pi)

Warning: Explicit integral could not be found.

ans =

int(exp(sin(x)),x = 0 .. pi)


format long; quadl(@(x) exp(sin(x)), 0, pi)

ans =

   6.20875803571375
```

(b)

```
quadl(@(x) sqrt(x.^3 + 1), 0, 1)

ans =

   1.11144798484585
```

(c)

MATLAB integrated this one exactly in 3(e) above; let's integrate it numerically and compare answers. Unfortunately, the range is infinite, so to use **quadl** we have to approximate the interval. Note that

```
exp(-35)

ans =

     6.305116760146989e-16
```

which is close to the standard floating-point accuracy, so

```
format long; quadl(@(x) exp(-x.^2), -35, 35)

ans =

   1.77245385102263


sqrt(pi)

ans =

   1.77245385090552
```

The answers agree up to at least eight digits.

5.

(a)

```
limit(sin(x)/x, x, 0)

ans =

1
```

(b)

```
limit((1 + cos(x))/(x + pi), x, -pi)
```

```
ans =

0
```

(c)

```
limit(x^2*exp(-x), x, Inf)
```

```
ans =

0
```

(d)

```
limit(1/(x - 1), x, 1, 'left')
```

```
ans =

-Inf
```

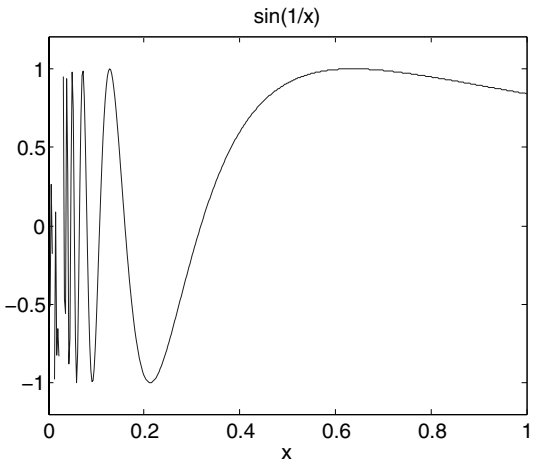
(e)

```
limit(sin(1/x), x, 0, 'right')
```

```
ans =

-1 .. 1
```

This means that every real number in the interval between -1 and $+1$ is a "limit point" of $\sin(1/x)$ as x tends to zero. You can see why if you plot $\sin(1/x)$ on the interval $(0, 1]$.

```
ezplot(sin(1/x), [0 1])
```



6.

(a)

```
syms k n r x z
symsum(k^2, k, 0, n)
```

```
ans =

1/3*(n+1)^3-1/2*(n+1)^2+1/6*n+1/6
```

```
simplify(ans)
```

```
ans =

1/3*n^3+1/2*n^2+1/6*n
```

(b)

```
symsum(r^k, k, 0, n)
```

```
ans =

r^(n+1)/(r-1)-1/(r-1)
```

```
pretty(ans)
```

```
                          (n + 1)
                         r             1
                         -------- - -----
                          r - 1     r - 1
```

(c)

```
symsum(x^k/factorial(k), k, 0, Inf)
```

```
Error using ==> factorial
N must be a matrix of non-negative integers.
```

Here are two ways around this difficulty.

```
symsum(x^k/gamma(k + 1), k, 0, Inf)
```

```
ans =

exp(x)
```

```
symsum(x^k/sym('k!'), k, 0, Inf)
```

```
ans =

exp(x)
```

(d)

```
symsum(1/(z - k)^2, k, -Inf, Inf)
```

```
ans =

pi^2+pi^2*cot(pi*z)^2
```

7.

(a)

```
taylor(exp(x), 7, 0)

ans =

1+x+1/2*x^2+1/6*x^3+1/24*x^4+1/120*x^5+1/720*x^6
```

(b)

```
taylor(sin(x), 5, 0)

ans =

x-1/6*x^3


taylor(sin(x), 6, 0)

ans =

x-1/6*x^3+1/120*x^5
```

(c)

```
pretty(taylor(sin(x), 6, 2))

                                                 2                       3
  sin(2) + cos(2) (x - 2) - 1/2 sin(2) (x - 2)  - 1/6 cos(2) (x - 2)

                                 4                         5
         + 1/24 sin(2) (x - 2)  + 1/120 cos(2) (x - 2)
```

(d)

```
taylor(tan(x), 7, 0)

ans =

x+1/3*x^3+2/15*x^5
```

(e)

```
taylor(log(x), 5, 1)
```

```
ans =

x-1-1/2*(x-1)^2+1/3*(x-1)^3-1/4*(x-1)^4
```

(f)

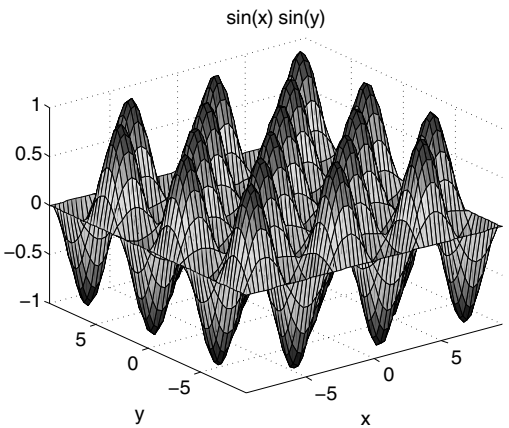
```
pretty(taylor(erf(x), 9, 0))
```

```
                          3            5             7
        x            x            x             x
    2 ----- - 2/3 ----- + 1/5 ----- - 1/21 -----
        1/2          1/2          1/2           1/2
      pi           pi           pi            pi
```

8.

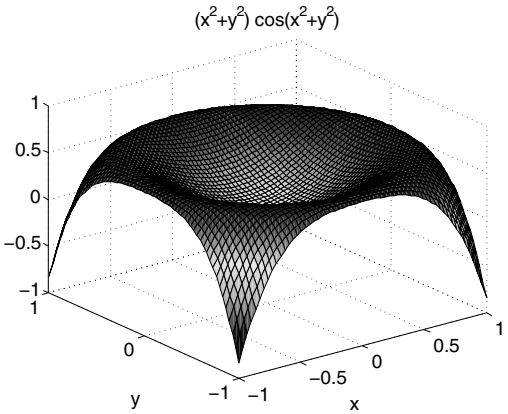
(a)

```
syms x y; ezsurf(sin(x)*sin(y), [-3*pi 3*pi -3*pi 3*pi])
```



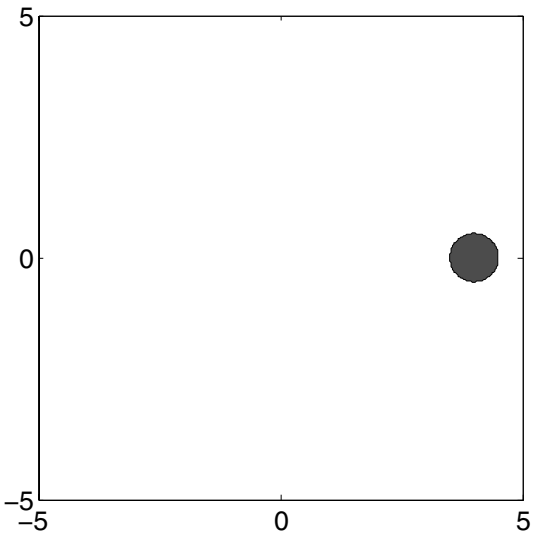
(b)

```
syms x y; ezsurf((x^2 + y^2)*cos(x^2 + y^2), [-1 1 -1 1])
```



9.

```
T = 0:0.01:1;
for j = 0:16
    fill(4*cos(j*pi/8)+(1/2)*cos(2*pi*T), ...
        4*sin(j*pi/8)+(1/2)*sin(2*pi*T), 'r');
    axis equal; axis([-5 5 -5 5]);
    M(j+1) = getframe;
end
movie(M)
```



Obviously we can't show the movie in this book, but you see the last frame above.

10.

(a)

```
A1 = [3 4 5; 2 -3 7; 1 -6 1]; b = [2; -1; 3];
format short; x = A1\b
```

```
x =

    2.6196
   -0.2283
   -0.9891
```

```
A1*x
```

```
ans =

    2.0000
   -1.0000
    3.0000
```

(b)

```
A2 = [3 -9 8; 2 -3 7; 1 -6 1]; b = [2; -1; 3];
x = A2\b
```

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 4.189521e-18.

x =

   -6.0000
   -1.3333
    1.0000
```

```
A2*x
```

```
ans =

     2
    -1
     3
```

The matrix **A2** is singular. In fact,

```
det(A2)
```

```
ans =

     0
```

(c)

```
A3 = [1 3 -2 4; -2 3 4 -1; -4 -3 1 2; 2 3 -4 1];
b3 = [1; 1; 1; 1]; x = A3\b3
```

```
x =

   -0.5714
    0.3333
   -0.2857
         0
```

```
A3*x
```

```
ans =

    1.0000
    1.0000
    1.0000
    1.0000
```

(d)

```
syms a b c d x y u v;
A4 = [a b; c d]; A4\[u; v]
```

```
ans =

  (d*u-b*v)/(d*a-b*c)
 -(c*u-v*a)/(d*a-b*c)
```

```
det(A4)
```

```
ans =

d*a-b*c
```

The determinant of the coefficient matrix is the denominator in the answer. So one gets an answer only if the coefficient matrix is non-singular.

11.

(a)

```
rank(A1)
```

```
ans =

     3
```

```
rank(A2)
```

```
ans =

     2
```

```
rank(A3)
```

```
ans =

     4
```

```
rank(A4)
```

```
ans =

2
```

MATLAB implicitly assumes that $ad - bc$ is not 0 here.

(b)

Only the second one computed is singular.

(c)

```
det(A1)
```

```
ans =

    92
```

```
inv(A1)
```

```
ans =

    0.4239   -0.3696    0.4674
    0.0543   -0.0217   -0.1196
   -0.0978    0.2391   -0.1848
```

```
det(A2)
```

```
ans =

     0
```

The matrix **A2** does not have an inverse.

```
det(A3)
```

```
ans =

   294
```

```
inv(A3)
```

```
ans =

    0.1837   -0.1531   -0.2857   -0.3163
         0    0.1667         0    0.1667
    0.1633    0.0306   -0.1429   -0.3367
    0.2857   -0.0714         0   -0.2143
```

```
det(A4)
```

```
ans =

d*a-b*c
```

```
inv(A4)
```

```
ans =

[  d/(d*a-b*c), -b/(d*a-b*c)]
[ -c/(d*a-b*c),  a/(d*a-b*c)]
```

12.

(a)

```
[U1, R1] = eig(A1)
```

```
U1 =

  -0.9749             0.6036             0.6036
  -0.2003             0.0624 + 0.5401i   0.0624 - 0.5401i
   0.0977            -0.5522 + 0.1877i  -0.5522 - 0.1877i


R1 =

   3.3206                  0                  0
        0            -1.1603 + 5.1342i        0
        0                  0            -1.1603 - 5.1342i
```

```
A1*U1 - U1*R1
```

```
ans =

   1.0e-14 *

   0.3109             0.2776 - 0.3997i   0.2776 + 0.3997i
  -0.0444                  0 - 0.0777i        0 + 0.0777i
  -0.0833             0.0999 - 0.1776i   0.0999 + 0.1776i
```

This is essentially zero. Notice the `e-14`.

```
[U2, R2] = eig(A2)
```

```
U2 =

   0.9669             0.7405             0.7405
   0.1240             0.4574 - 0.2848i   0.4574 + 0.2848i
  -0.2231             0.2831 + 0.2848i   0.2831 - 0.2848i


R2 =

  -0.0000                  0                  0
        0             0.5000 + 6.5383i        0
        0                  0             0.5000 - 6.5383i


A2*U2 - U2*R2

ans =

   1.0e-14 *

  -0.2224            -0.2554 - 0.2665i  -0.2554 + 0.2665i
  -0.1498             0.0888             0.0888
  -0.1156            -0.0222 + 0.0666i  -0.0222 - 0.0666i
```

Same comment as in (a).

```
[U3, R3] = eig(A3);

max(abs(A3*U3 - U3*R3))

ans =

   1.0e-14 *

    0.4529    0.4529    0.5245    0.5245
```

Ditto yet again. (Note that we suppressed the output of the first command, and took the maximum of the absolute values of the entries in each column of **`A3*U3 - U3*R3`**, in order to keep the data from scrolling off the screen.)

```
[U4, R4] = eig(A4);
pretty(U4)

      [                        2            2         1/2
      [  1/2 d - 1/2 a - 1/2 (d  - 2 d a + a  + 4 b c)
      [- ------------------------------------------------ ,
      [                         c
```

```
                                2             2          1/2]
        1/2 d - 1/2 a + 1/2 (d  - 2 d a + a  + 4 b c)    ]
      - ----------------------------------------------]
                                c                        ]

      [1 , 1]
```

```
pretty(R4)
```

```
      [                       2             2          1/2    ]
      [1/2 d + 1/2 a + 1/2 (d  - 2 d a + a  + 4 b c)     , 0]
      [                                                      ]
      [                           2             2          1/2]
      [0 , 1/2 d + 1/2 a - 1/2 (d  - 2 d a + a  + 4 b c)    ]
```

```
simplify(A4*U4 - U4*R4)
```

```
ans =

[ 0, 0]
[ 0, 0]
```

(b)

```
A = [1 0 2; -1 0 4; -1 -1 5];
[U1, R1] = eig(A)
```

```
U1 =

   -0.8165    0.5774    0.7071
   -0.4082    0.5774   -0.7071
   -0.4082    0.5774    0.0000


R1 =

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

```
B = [5 2 -8; 3 6 -10; 3 3 -7];
[U2, R2] = eig(B)
```

```
U2 =

    0.8165   -0.5774    0.7071
    0.4082   -0.5774   -0.7071
    0.4082   -0.5774    0.0000


R2 =

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

We observe that the first and second columns of **U1** are negatives of the corresponding columns of **U2**, and the third columns are identical. Finally,

```
A*B - B*A

ans =

     0     0     0
     0     0     0
     0     0     0
```

13.

(a)

If we set X_n to be the column matrix with entries x_n, y_n, and z_n, and M the square matrix with entries $1, 1/4, 0; 0, 1/2, 0; 0, 1/4, 1$ then $X_{n+1} = MX_n$.

(b)

We have $X_n = MX_{n-1} = M^2X_{n-2} = ... = M^nX_0$.

(c)

```
M = [1, 1/4, 0; 0, 1/2, 0; 0, 1/4, 1];
[U,R] = eig(M)


U =

    1.0000         0   -0.4082
         0         0    0.8165
         0    1.0000   -0.4082
```

```
R =

    1.0000         0         0
         0    1.0000         0
         0         0    0.5000
```

(d)

M should be URU^{-1}. Let's check:

```
M - U*R*inv(U)

ans =

     0     0     0
     0     0     0
     0     0     0
```

Since R^n is the diagonal matrix with entries $1, 1, 1/2^n$, we know that R_∞ is the diagonal matrix with entries $1, 1, 0$. Therefore $M_\infty = UR_\infty U^{-1}$. So

```
Minf = U*diag([1, 1, 0])*inv(U)

Minf =

    1.0000    0.5000         0
         0         0         0
         0    0.5000    1.0000
```

(e)

```
syms x0 y0 z0; X0 = [x0; y0; z0]; Minf*X0

ans =

 x0+1/2*y0
         0
 1/2*y0+z0
```

Half of the mixed genotype migrates to the dominant genotype and the other half of the mixed genotype migrates to the recessive genotype. These are added to the two original pure types, whose proportions are preserved.

(f)

```
M^5*X0
```

```
ans =

 x0+31/64*y0
     1/32*y0
 31/64*y0+z0
```

```
M^10*X0
```

```
ans =

 x0+1023/2048*y0
       1/1024*y0
 1023/2048*y0+z0
```

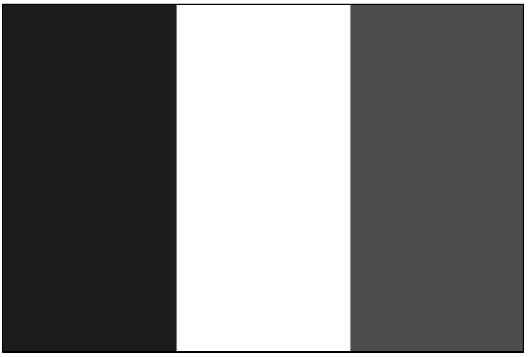
(g)

With the suggested alternate model, only the first three columns of the table are relevant, the transition matrix `M` becomes `M = [1 1/2 0; 0 1/2 1; 0 0 0]`. You can compute that the eventual population distribution is `[1 0 0]`, and is independent of the initial population.

14.

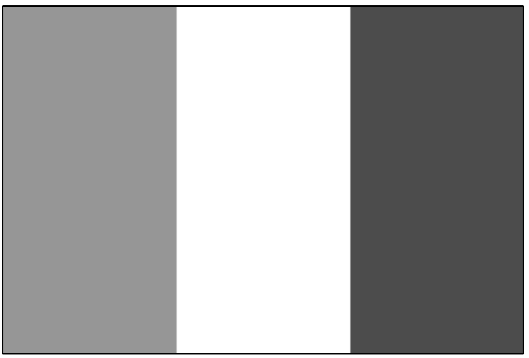
The following commands create a JPEG file of the French flag. The `set` command has no effect on the JPEG file; we include it only to turn off the axis ticks in the figure window.

```
A = zeros(200, 300, 3);
A(:, 1:100, 3) = ones(200, 100);
A(:, 101:200, 1) = ones(200, 100);
A(:, 101:200, 2) = ones(200, 100);
A(:, 101:200, 3) = ones(200, 100);
A(:, 201:300, 1) = ones(200, 100);
image(A); axis equal tight
set(gca, 'XTick', []), set(gca, 'YTick', [])
imwrite(A, 'tricolore.jpg')
```



We now redefine the color of the leftmost third of the array and create a JPEG file of the Italian flag.

```
A(:, 1:100, 3) = zeros(200, 100);
A(:, 1:100, 2) = ones(200, 100);
image(A); axis equal tight
set(gca, 'XTick', []), set(gca, 'YTick', [])
imwrite(A, 'italia.jpg')
```



Of course you will have to run the commands above yourself to see the flags in color. Then you can run the commands below to see a movie transforming the French flag into the Italian flag.

```
clear M
for k = 0:10
    A(:, 1:100, 3) = (10-k)*ones(200, 100)/10;
    A(:, 1:100, 2) = k*ones(200, 100)/10;
    M(k+1) = im2frame(A);
end
movie(M)
```

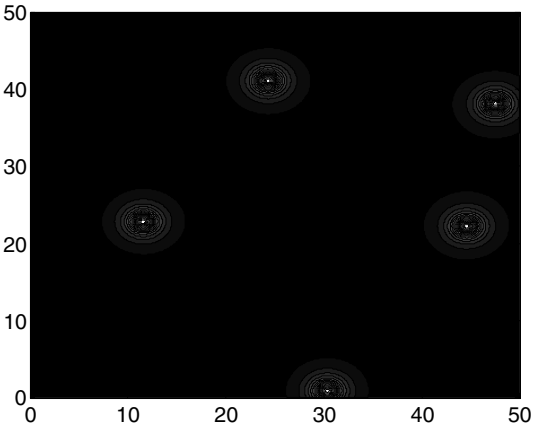
Solutions to Practice Set C: Developing Your MATLAB Skills

```
clear all
```

1.

(a)

```
radiation = @(x, y, x0, y0) 10000./(4*pi*((x - x0).^2 + ...
    (y - y0).^2 + 1));

x0 = 50*rand(1, 5); y0 = 50*rand(1, 5);
[X, Y] = meshgrid(0:0.1:50, 0:0.1:50);
contourf(X, Y, radiation(X, Y, x0(1), y0(1)) + ...
    radiation(X, Y, x0(2), y0(2)) + ...
    radiation(X, Y, x0(3), y0(3)) + ...
    radiation(X, Y, x0(4), y0(4)) + ...
    radiation(X, Y, x0(5), y0(5)), 20)
colormap('gray')
```



It is not so clear from the picture where to hide, although it looks as if the Captain has a pretty good chance of surviving a small number of shots. But 100 shots may be enough to find him. Intuition says he ought to stay close to the boundary.

(b)

Below is a series of commands that places Picard at the center of the arena, fires the death ray 100 times, and then determines the health of Picard. It uses the following function M-file `lifeordeath.m`, which computes the fate of the Captain after a single shot.

```
type lifeordeath
```

```
function r = lifeordeath(x1, y1, x0, y0)
% This function computes the number of illumatons
% that arrive at the point (x1,y1), assuming that the
% death ray strikes 1 meter above the point (x0,y0).
% If that number exceeds 50, a "1" is returned in the
% variable "r"; otherwise a "0" is returned for "r".
```

```
dosage = 10000/(4*pi*((x1 - x0)^2 + (y1 - y0)^2 + 1));
if dosage > 50
    r = 1;
else
    r = 0;
end
```

Here is the series of commands to test the Captain's survival possibilities:

```
x1 = 25; y1 = 25; h = 0;
for n = 1:100
    x0 = 50*rand;
    y0 = 50*rand;
    r = lifeordeath(x1, y1, x0, y0);
    h = h + r;
end
if h > 0
    disp('The Captain is dead!')
else
    disp('Picard lives!')
end
```

```
The Captain is dead!
```

In fact if you run this sequence of commands multiple times, you will see the Captain die far more often than he lives.

(c)

We now use another M-file, `simulation.m`, which runs the loop from part (b) 100 times, and reports how many times Picard survives, given his location as input.

```
type simulation
```

```
function c = simulation(x1, y1)
c = 0;
for k = 1:100
    h = 0;
    for n = 1:100
        x0 = 50*rand;
        y0 = 50*rand;
        r = lifeordeath(x1, y1, x0, y0);
        h = h + r;
    end
    if h == 0
        c = c + 1;
    end
end
```

So let's do a Monte Carlo simulation to see what his odds are:

```
x1 = 25; y1 = 25;
disp(['The chances of Picard surviving are ', ...
    num2str(simulation(x1, y1)/100)])
```

```
The chances of Picard surviving are 0.13
```

We ran this simulation a few times and saw survival chances ranging from 8% to 19%.

(d)

```
x1 = 37.5; y1 = 25;
disp(['The chances of Picard surviving are ', ...
    num2str(simulation(x1, y1)/100)])
```

```
The chances of Picard surviving are 0.15
```

This time the numbers were between 10% and 24%. Let's keep moving him toward the periphery.

(e)

```
x1 = 50; y1 = 25;
disp(['The chances of Picard surviving are ', ...
    num2str(simulation(x1, y1)/100)])
```

```
The chances of Picard surviving are 0.46
```

The numbers now hover between 32% and 47% upon multiple runnings of this scenario; so finally, suppose that he cowers in the corner.

```
x1 = 50; y1 = 50;
disp(['The chances of Picard surviving are ', ...
    num2str(simulation(x1, y1)/100)])
```

```
The chances of Picard surviving are 0.63
```

We saw numbers between 54% and 68%. They say a brave man dies but a single time, but a coward dies a thousand deaths. But the person who said that probably never encountered a Cardassian. Long live Picard!

2.

(a)

Consider the status of the bank account on the last day of each month. At the end of the first month, the account has $M + M \times J = M(1 + J)$ dollars. Then at the end of the second month the account contains $[M(1 + J)](1 + J) = M(1 + J)^2$ dollars. Similarly, at the end of n months, the account will hold $M(1 + J)^n$ dollars. So our formula is

$$T = M(1 + J)^n.$$

(b)

Now we take $M = 0$ and S dollars deposited monthly. At the end of the first month the account has $S + S \times J = S(1 + J)$ dollars. The next day, S dollars are added to that sum, and then at the end of the second month the account contains $[S(1 + J) + S](1 + J) = S[(1 + J)^2 + (1 + J)]$ dollars. Similarly, at the end of n months, the account will hold $S[(1 + J)^n + (1 + J)^{n-1} + \cdots + (1 + J)]$ dollars. Summing the geometric series, the amount T in the account after n months equals

$$T = S[((1 + J)^{n+1} - (J + 1))/((1 + J) - 1)] = S[((1 + J)^{n+1} - 1)/J - 1].$$

(c)

By combining the two models it is clear that, in an account with an initial balance M and monthly deposits S, the amount of money T after n months is given by

$$T = M(1 + J)^n + S[((1 + J)^{n+1} - 1)/J - 1].$$

(d)

We are asked to solve the equation

$$(1 + J)^n = 2$$

with the values $J = 0.05/12$ and $J = 0.1/12$.

```
months = solve('(1 + 0.05/12)^n = 2');
years = double(months)/12
```

```
years =

   13.8918
```

```
months = solve('(1 + 0.1/12)^n = 2');
years = double(months)/12
```

```
years =

    6.9603
```

If you double the interest rate, you roughly halve the time required to achieve the goal.

(e)

```
format bank
1000000/(((1 + 0.08/12)^(12*35 + 1) - 1)/(0.08/12) - 1)
```

```
ans =

        433.06
```

You need to deposit $433.06 every month.

(f)

```
syms n
T = 300*(((1 + 0.08/12)^(n + 1) - 1)/(0.08/12) - 1);
months = solve(T - 1000000);
years = double(months)/12
```

```
years =

         39.37
```

You have to work nearly five more years.

(g)

First, taking the whole bundle at once, after 20 years the $65,000 left after taxes generates

```
option1 = 65000*(1 + 0.05/12)^(12*20)
```

```
option1 =

     176321.62
```

The stash grows to about $176,322. The second option yields

```
S = 0.8*(100000/240);
option2 = S*(((1 + 0.05/12)^(12*20 + 1) - 1)/(0.05/12) - 1)
```

```
option2 =

     137582.10
```

You accumulate only $137,582 this way. Taking the lump sum up front is clearly the better strategy.

(h)

```
rates = [.13, .15, -.03, .05, .10, .13, .15, -.03, .05];

for k = 0:4
    T = 50000;
    for j = 1:5
        T = T*(1 + rates(k + j));
    end
    disp([k + 1, T])
end
```

```
          1.00      72794.74

          2.00      72794.74

          3.00      72794.74

          4.00      72794.74

          5.00      72794.74
```

The results are all the same; you wind up with \$72,795 regardless of where you enter in the cycle, because the product $\prod_{1\le j\le 5}(1+\text{rates}(j))$ is independent of the order in which you place the factors. If you put the \$50,000 in a bank account paying 8%, you get

```
50000*(1 + 0.08)^5

ans =

      73466.40
```

That is, \$73,466 – better than the market. The market's volatility hurts you compared with the bank's stability. But of course that assumes you can find a bank that will pay 8%. Now let's see what happens with no stash, but an annual investment instead. The analysis is more subtle here. Set $S = 10{,}000$. At the end of one year, the account contains $S(1+r_1)$; then at the end of the second year $[S(1+r_1)+S](1+r_2)$, where we have written r_j for **rates(j)**. So at the end of five years, the amount in the account will be the product of S and the number

$$\prod_{j\ge 1}(1+r_j)+\prod_{j\ge 2}(1+r_j)+\prod_{j\ge 3}(1+r_j)+\prod_{j\ge 4}(1+r_j)+(1+r_5).$$

If you enter at a different year in the business cycle the terms get cycled appropriately. So now we can compute

```
for k = 0:4
    T = ones(1, 5);
    for j = 1:5
        TT = 1;
        for l = j:5
            TT = TT*(1 + rates(k + l));
        end
        T(j) = TT;
    end
    disp([k + 1, 10000*sum(T)])
end

          1.00      61196.47

          2.00      64000.40

          3.00      68357.67

          4.00      61884.76

          5.00      60192.11
```

Not surprisingly, all the amounts are less than what one obtains by investing the original $50,000 all at once. But in this model it matters where you enter the business cycle. It's clearly best to start your investment program when a recession is in force and end in a boom. Incidentally, the bank model yields in this case

```
10000*(((1 + 0.08)^6 - 1)/0.08 - 1)

ans =

       63359.29
```

which is better than some investment models and worse than others.

3.

(a)

We can use the expression **`(rand < 0.338)`** to compute whether Tony gets a hit or not during a single at bat, based on a random number chosen between 0 and 1. If the random number is less than 0.338, the expression evaluates to 1 and Tony is credited with a hit; otherwise, the expression evaluates to 0 and Tony is retired by the opposition.

We could simulate a year in Tony's career by evaluating this expression 500 times in a loop. More simply, we can put 500 random numbers into this expression at once and sum the results, dividing by 500 to get his batting average for the year. The following function does just this, allowing more generally for **`n`** at bats in the year, although we shall use only 500.

```
yearbattingaverage = @(n) sum(rand(n, 1) < 0.338)/n;
```

Now we run the function.

```
format short
yearbattingaverage(500)

ans =

    0.3380
```

(b)

Now let's write a function M-file that simulates a 20-year career. As with the number of at bats in a year, we'll allow for a varying-length career.

```
type career
```

```
function ave = career(n, k)
% This function file computes the batting average for each
% year in a k-year career, assuming n at bats in each year.
% Then it lists the average for each of the years in the
% career, and finally computes a lifetime average.
Y = zeros(1, k);
for j = 1:k
    Y(j) = sum(rand(n, 1) < 0.338)/n;
end
ave = sum(Y)/k;
disp(['Best avg: ', num2str(max(Y), 4)])
disp(['Worst average: ', num2str(min(Y), 4)])
disp(['Lifetime avg: ', num2str(ave, 4)])
```

Next we run the simulation.

```
ave1 = career(500, 20);
```

```
Best avg: 0.39
Worst average: 0.292
Lifetime avg: 0.3394
```

(c)

Now we run the simulation four more times:

```
ave2 = career(500, 20);
```

```
Best avg: 0.372
Worst average: 0.294
Lifetime avg: 0.334
```

```
ave3 = career(500, 20);
```

```
Best avg: 0.382
Worst average: 0.306
Lifetime avg: 0.339
```

```
ave4 = career(500, 20);
```

```
Best avg: 0.36
Worst average: 0.292
Lifetime avg: 0.3314
```

```
ave5 = career(500, 20);
```

```
Best avg: 0.38
Worst average: 0.294
Lifetime avg: 0.339
```

(d)

The average for the five different 20-year careers is:

```
(ave1 + ave2 + ave3 + ave4 + ave5)/5

ans =

    0.3366
```

Not bad. In fact if we ran the simulation 100 times and took the average it would be very close to 0.338.

4.

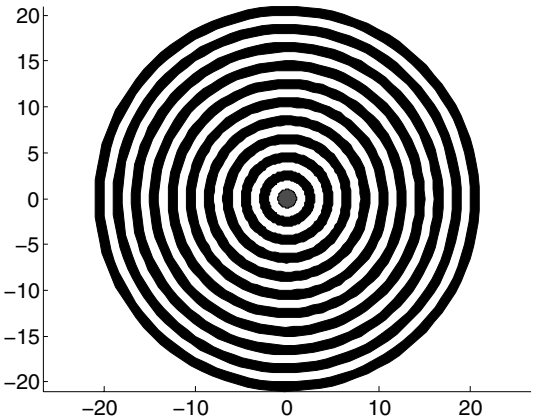
Our solution and its output is below. First we set **n** to 500 in order to save typing in the following lines and make it easier to change this value later. Then we set up a zero matrix **A** of the appropriate sizes and begin a loop that successively defines each row of the matrix. Finally, we extract the maximum value from the list of eigenvalues of **A**.

```
n = 500;
A = zeros(n);
for k = 1:n
    A(k,:) = 1./(k:(k + n - 1));
end
max(eig(A))

ans =

    2.3769
```

5.

Again we display below our solution and its output. First we define a vector t of values between 0 and 2π, in order to be able later to represent circles parametrically as $x = r\cos t, y = r\sin t$. Then we clear any previous figure that might exist and prepare to create the figure in several steps. Let's say that the red circle will have radius 1; then the first black ring should have inner radius 2 and outer radius 3, and thus the tenth black ring should have inner radius 20 and outer radius 21. We start drawing from the outside in because the idea is to fill the largest circle in black, then fill the next largest circle in white leaving only a ring of black, then fill the next largest circle in black leaving a ring of white, etc. The **if** statement tests true when **r** is odd and false when it is even. We stop the alternation of black and white at a radius of 2 in order to make the last circle red instead of black, then we adjust the axes to make the circles appear round.

```
t = linspace(0, 2*pi, 100);
cla reset; hold on
for r = 21:-1:2
    if mod(r,2)
        fill(r*cos(t), r*sin(t), 'k')
    else
        fill(r*cos(t), r*sin(t), 'w')
    end
end
fill(cos(t), sin(t), 'r')
axis equal; hold off
```



6.

Here are the contents of our solution M-file.

```
type mylcm
```

```
function m = mylcm(varargin)

nums = [varargin{:}];
if ~isnumeric(nums) | any(nums ~= round(real(nums))) | ...
        any(nums <= 0)
    error('Arguments must be positive integers.')
end

for k = 2:length(nums);
    nums(k) = lcm(nums(k), nums(k-1));
end
m = nums(end);
```

Here are some examples. First, cases in which it should work:

```
mylcm([4 5 6])
```

```
ans =

    60
```

```
mylcm([6 7 12 15])
```

```
ans =

   420
```

Next, cases in which we expect an error:

```
mylcm(4.5, 6)
```

```
Error using ==> mylcm
Arguments must be positive integers.
```

```
mylcm('a', 'b', 'c')
```

```
Error using ==> mylcm
Arguments must be positive integers.
```

7.

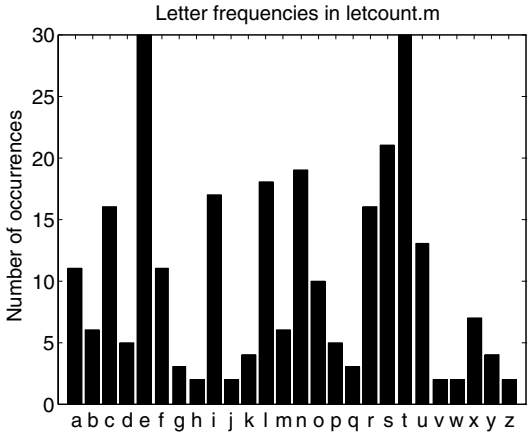
Here are the contents of our solution M-file.

```
type letcount
```

```
function letcount(file)
if isunix
    [stat, str] = unix(['cat ' file]);
else
    [stat, str] = dos(['type ' file]);
end

letters = 'abcdefghijklmnopqrstuvwxyz';
caps = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
for n = 1:26
    count(n) = sum(str == letters(n)) + sum(str == caps(n));
end
bar(count)
ylabel 'Number of occurrences'
title(['Letter frequencies in ' file])
set(gca, 'XLim', [0 27], 'XTick', 1:26, 'XTickLabel', ...
    letters')
```

Here is the output when we run this M-file on itself.

```
letcount('letcount.m')
```



8.

We let w, x, y, and z denote the number of residences canvassed in the four cities Gotham, Metropolis, Oz, and River City, respectively. Then the linear inequalities specified by the given data are as follows:

- Non-negative data: $w \geq 0$, $x \geq 0$, $y \geq 0$, $z \geq 0$;
- Pamphlets: $w + x + y + z \leq 50{,}000$;
- Travel cost: $0.5w + 0.5x + y + 2z \leq 40{,}000$;
- Time available: $2w + 3x + y + 4z \leq 18{,}000$;
- Preferences: $w \leq x$, $x + y \leq z$;
- Contributions: $w + 0.25x + 0.5y + 3z \geq 10{,}000$.

The quantity to be maximized is as follows:

- Voter support: $0.6w + 0.6x + 0.5y + 0.3z$.

(a)

This enables us to set up and solve the Linear Programming problem in MATLAB as follows:

```
f = [-0.6 -0.6 -0.5 -0.3];
A = [1 1 1 1; 0.5 0.5 1 2; 2 3 1 4; 1 -1 0 0; 0 1 1 -1; ...
    -1 -0.25 -0.5 -3; -1 0 0 0; 0 -1 0 0; 0 0 -1 0; 0 0 0 -1];
b = [50000; 40000; 18000; 0; 0; -10000; 0; 0; 0; 0];
simlp(f, A, b)
```

```
ans =

   1.0e+03 *

    1.2683
    1.2683
    1.3171
    2.5854
```

Jane should canvass 1268 residences in each of Gotham and Metropolis, 1317 residences in Oz, and 2585 residences in River City.

(b)

If the allotment for time doubles, then

```
b = [50000; 40000; 36000; 0; 0; -10000; 0; 0; 0; 0];
simlp(f, A, b)
```

```
ans =

   1.0e+03 *

    4.0000
    4.0000
   -0.0000
    4.0000
```

Jane should canvass 4000 residences in each of Gotham, Metropolis, and River City, and ignore Oz.

(c)

Finally, if in addition she needs to raise $20,000 in contributions, then

```
b = [50000; 40000; 36000; 0; 0; -20000; 0; 0; 0; 0];
simlp(f, A, b)
```

```
ans =

   1.0e+03 *

    2.5366
    2.5366
    2.6341
    5.1707
```

Jane needs to canvass 2537 residences in each of Gotham and Metropolis, 2634 residences in Oz, and 5171 in River City.

9.

We let w, x, y, and z denote the numbers of hours that Joe spends with the quarterback, the running backs, the receivers, and the linemen, respectively. Then the linear inequalities specified by the given data are as follows:

- Non-negative data: $w \geq 0$, $x \geq 0$, $y \geq 0$, $z \geq 0$;
- Time available: $w + x + y + z \leq 50$;
- Point production: $0.5w + 0.3x + 0.4y + 0.1z \geq 20$;
- Criticisms: $w + 2x + 3y + 0.5z \leq 75$;
- Prima-donna status: $x = y$, $w \geq x + y$, $x \geq z$.

The quantity to be maximized is:

- Personal satisfaction: $0.2w + 0.4x + 0.3y + 0.6z$.

(a)

This enables us to set up and solve the Linear Programming problem in MATLAB as follows:

```
f = [-0.2 -0.4 -0.3 -0.6];
A = [1 1 1 1; -0.5 -0.3 -0.4 -0.1; 1 2 3 0.5; 0 -1 1 0; ...
    0 1 -1 0; -1 1 1 0; 0 -1 0 1; -1 0 0 0; 0 -1 0 0; ...
    0 0 -1 0; 0 0 0 -1];
b = [50; -20; 75; 0; 0; 0; 0; 0; 0; 0; 0];
simlp(f, A, b)
```

```
ans =

   25.9259
    9.2593
    9.2593
    5.5556
```

Joe should spend 9.3 hours each with the running backs and receivers; 5.6 hours with the linemen; and the majority of his time, 25.9 hours, with the quarterback.

(b)

If the team needs only 15 points to win, then

```
b = [50; -15; 75; 0; 0; 0; 0; 0; 0; 0; 0];
simlp(f, A, b)
```

```
ans =

   20.0000
   10.0000
   10.0000
   10.0000
```

Joe can spread his time more evenly, 10 hours each with the running backs, receivers and linemen; but still the biggest chunk of his time, 20 hours, with the quarterback.

(c)

Finally, if in addition the number of criticisms is reduced to 70, then

```
b = [50; -15; 70; 0; 0; 0; 0; 0; 0; 0; 0];
simlp(f, A, b)
```

```
ans =

   18.6667
    9.3333
    9.3333
    9.3333
```

Joe must spend $18\frac{2}{3}$ hours with the quarterback, and $9\frac{1}{3}$ hours with each of the other three groups. Notice that the total is less than 50, leaving Joe some free time to tend to his NASCAR teams.

10.

We let **f** be the quantity to be set to zero. It is a function of the diode voltage, **VD**, and the parameters **V0**, **R**, **I0**, and **VT**. Notice that **f** should be zero for some value between **0** and **V0**.

```
f = @(V, V0, R, I0, VT) V - V0 + R*I0*exp(V/VT);
```

(a)

```
f1 = @(V) f(V, 1.5, 1000, 10^(-5), .0025);
VD = fzero(f1, [0, 1.5])
```

```
VD =

    0.0125
```

That's the voltage; the current is therefore

```
I = (1.5 - VD)/1000
```

```
I =

    0.0015
```

(b)

```
f2 = @(V) f(V, 1.5, 1000, 10^(-5)/2, .0025);
fzero(f2, [0, 1.5])
```

```
ans =

    0.0142
```

Not surprisingly, the voltage goes up slightly.

(c)

```
f2 = @(V) f(V, 1.5, 1000, 10^(-5), .0025/2);
fzero(f2, [0, 1.5])
```

```
Error using ==> fzero
Function values at interval endpoints must be finite and real.
```

The problem is that the values of the exponential are too big at the right-hand endpoint of the test interval. We have to specify an interval big enough to catch the solution, but small enough to prevent the exponential from blowing up too drastically at the right-hand endpoint. This will be the case even more dramatically in part (e) below.

```
fzero(f2, [0, 0.5])
```

```
ans =

    0.0063
```

This time the voltage goes down.

(d)

Next we halve both:

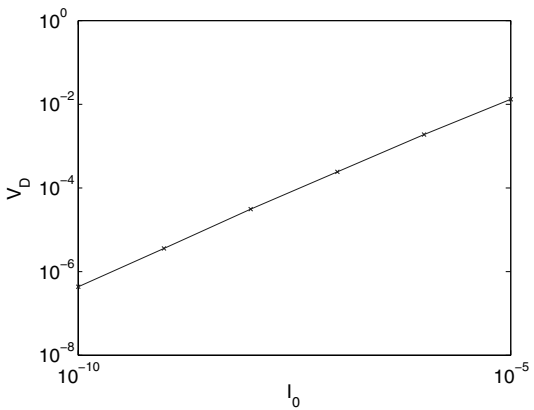
```
f2 = @(V) f(V, 1.5, 1000, 10^(-5)/2, .0025/2);
fzero(f2, [0, 0.5])
```

```
ans =

    0.0071
```

The voltage is less than in part (b) but more than in part (c).

(e)

```
X = 10.^(0:-1:-5);
f3 = @(V, x) f(V, 1.5, 1000, 10^(-5)*x, .0025*x);
VD = 0:5; % initialize vector of VD-values
for j = 1:6
    f4 = @(V) f3(V, X(j));
    VD(j) = fzero(f4, [0, X(j)/10]);
end
loglog(10^(-5)*X, VD, 'x-')
xlabel 'I_0'
ylabel 'V_D'
```



The **loglog** plot appears linear. This suggests that **VD** is roughly a constant times a power of **I0**.

11.

(a)

```
dsolve('Dx = x - x^2')
```

```
ans =

1/(1+exp(-t)*C1)
```

```
syms x0; sol = dsolve('Dx = x - x^2', 'x(0) = x0')
```

```
sol =

1/(1-exp(-t)*(-1+x0)/x0)
```

Note that this includes the zero solution; indeed,

```
bettersol = simplify(sol)
```

```
bettersol =

-x0/(-x0-exp(-t)+exp(-t)*x0)
```

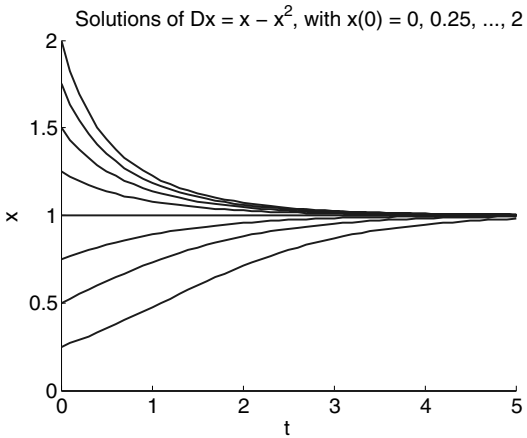
```
subs(bettersol, x0, 0)
```

```
ans =

0
```

(b)

We have already solved the equation in (a) above, so all we need to do is to substitute the initial conditions in for **x0** and plot the results. We increase the **LineWidth** from its default value so that the zero solution stands out better.

```
T = 0:0.1:5;

cla reset; hold on
solcurves = @(t,x0) eval(vectorize(bettersol));
for initval = 0:0.25:2.0
    plot(T, solcurves(T, initval), 'LineWidth', 1.5)
end
axis tight
title 'Solutions of Dx = x - x^2, with x(0) = 0, 0.25, ..., 2'
xlabel 't'
ylabel 'x'
hold off
```

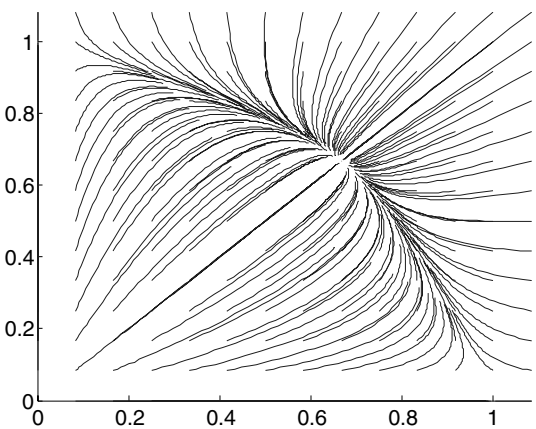


The graphical evidence suggests that the solution that starts at zero stays there; all the others tend toward the constant solution 1.

(c)

To use **`ode45`**, we want to write the differential equations as a single equation in a vector variable **`x`**. Its two components represent the two populations x and y.

```
cla reset; hold on
f = @(t,x) [x(1) - x(1)^2 - 0.5*x(1)*x(2); ...
    x(2) - x(2)^2 - 0.5*x(1)*x(2)];
for a = 0:1/12:13/12
    for b = 0:1/12:13/12
        [t, xa] = ode45(f, [0 3], [a, b]);
        plot(xa(:, 1), xa(:, 2))
    end
end
axis([0 13/12 0 13/12]); hold off
```

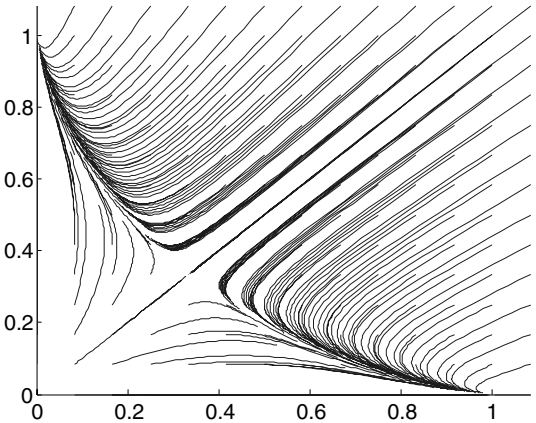


(d)

The endpoints on the curves are the start points. So clearly any curve that starts inside the first quadrant, that is one that corresponds to a situation in which both populations are present at the outset, tends toward a unique point – which from the graph appears to be about $(2/3, 2/3)$. In fact, if $x = y = 2/3$, then the right-hand sides of both equations in (C.4) vanish, so the derivatives are zero and the values of $x(t)$ and $y(t)$ remain constant – they don't depend on t. If only one species is present at the outset, that is you start out on one of the axes, then the solution tends toward either $(1, 0)$ or $(0, 1)$ depending on whether x or y is the species present. That is precisely the behavior we saw in part (b).

(e)

```
cla reset; hold on
f = @(t,x) [x(1) - x(1)^2 - 2*x(1)*x(2); ...
    x(2) - x(2)^2 - 2*x(1)*x(2)];
for a = 0:1/12:13/12
    for b = 0:1/12:13/12
        [t, xa] = ode45(f, [0 3], [a, b]);
        plot(xa(:, 1), xa(:, 2))
    end
end
axis([0 13/12 0 13/12]); hold off
```



This time most of the curves seem to be tending toward one of the points $(1, 0)$ or $(0, 1)$ – in particular, any solution curve that starts on one of the axes (corresponding to no initial population for the other species) does so. It seems that whichever species has a greater population at the outset will eventually take over all the population – the other will die out. But there is a delicate balance in the middle – it appears that, if the two populations are about equal at the outset, then they tend to the unique population distribution at which, if you start there, nothing happens. That value looks like $(1/3, 1/3)$. In fact, that is the value that renders both sides of (C.5) zero – which is analogous to the role $(2/3, 2/3)$ had in part (d).

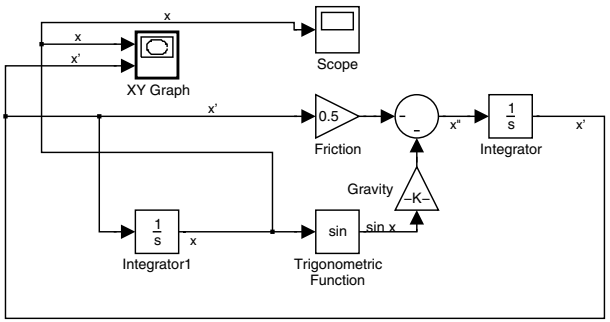
(f)

It makes sense to refer to the model (C.4) as "peaceful coexistence," since whatever initial populations you have – provided that both are present – you wind up with equal populations eventually. "Doomsday" is an appropriate name for model (C.5) since, if you start out with unequal populations, then the smaller group becomes extinct. The lower coefficient 0.5 means relatively small interaction between the species, allowing their coexistence. The larger coefficient 2 means stronger interaction and competition precluding the survival of both.

12.

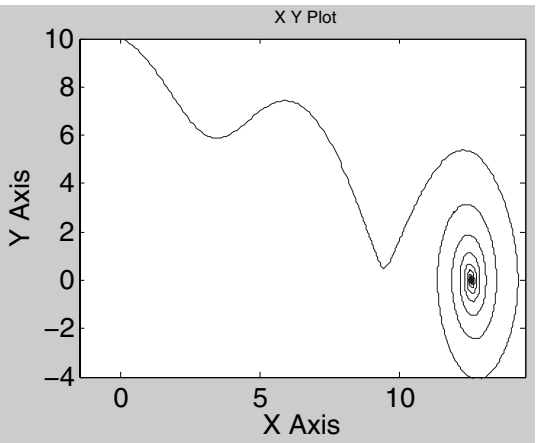
Here is a Simulink model for redoing the pendulum application from Chapter 9:

```
open_system pendulum
```



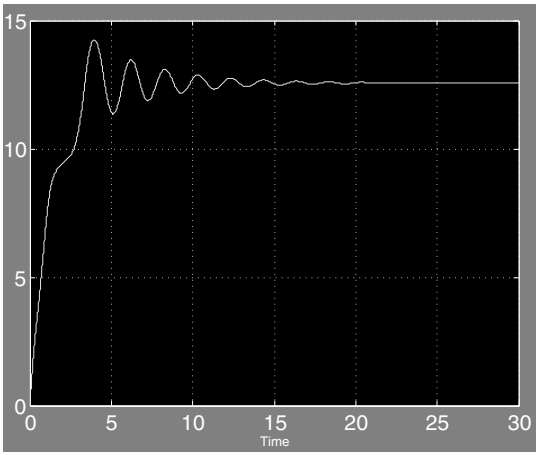
With the initial conditions $x(0) = 0$ and $\dot{x}(0) = 10$, the XY Graph block shows the following phase portrait of x versus $\dot{x}$.

```
[t, x] = sim('pendulum');
```



Meanwhile, the Scope block gives the following graph of x as a function of t.

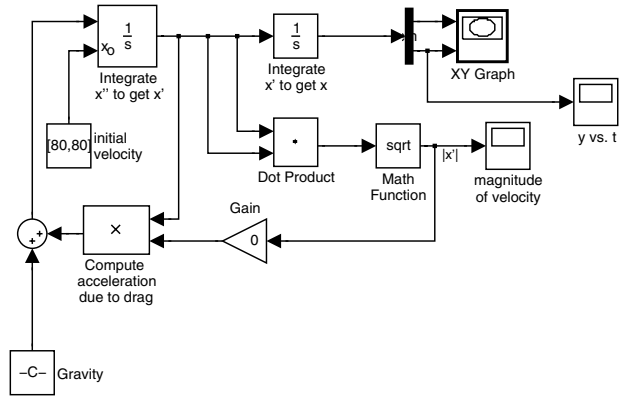
```
simplot(t, x(:,1))
axis([0 30 0 15])
```



13.

Here is a Simulink model for studying the equation of motion of a baseball.

```
open_system baseball
```



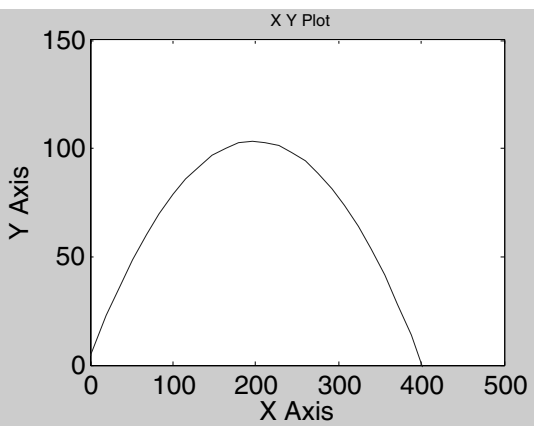
The way this works is fairly straightforward. The Integrator block in the upper left integrates the acceleration (a vector quantity) to get the velocity (also a vector). This block requires the initial value of the velocity as an initial condition; we define it in the "initial velocity" Constant block. Output from the first Integrator goes into the second Integrator, which integrates the velocity to get the position (also a vector). The initial condition for the position, `[0, 4]`, is stored in the parameters of this second Integrator. The position vector is fed into a Demux block, which splits off the horizontal and vertical components of the position. These are fed into the XY Graph block, and also the vertical component is fed into a scope block so that we can see the height of the ball as a function of time. The hardest part is the computation of the right-hand side of (C.7). This is computed by adding the two terms on the

right with the Sum block near the lower left. The value of `[0, -g]` is stored in the "gravity" Constant block. The second term on the right is computed in the Product block labeled "Compute acceleration due to drag," which multiplies the velocity (a vector) by $-c$ times the speed (a scalar). We compute the speed by taking the dot product of the velocity with itself and then taking the square root; then we multiply by $-c$ in the Gain block in the middle bottom of the model. The Scope block in the lower right plots the ball's speed as a function of time.

(a)

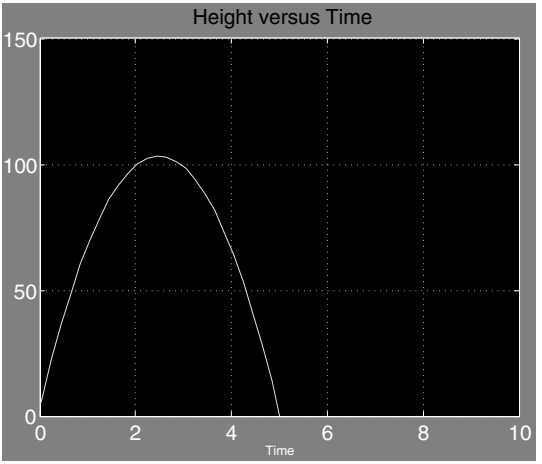
With c set to 0 (no air resistance) and the initial velocity set to $[80, 80]$, the ball follows a familiar parabolic trajectory, as seen in the following picture.

```
[t, x] = sim('baseball');
```

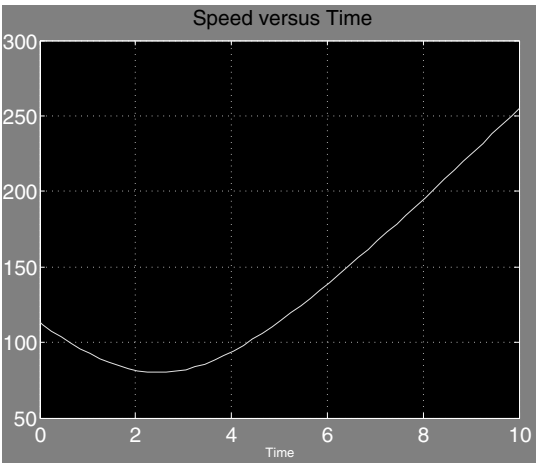


Note that the ball travels about 400 feet before hitting the ground, so the trajectory is just about what is required for a home run in most ballparks. We can read off the flight time and final speed from the other two scopes:

```
simplot(t, x(:,2))
axis([0 10 0 150])
title 'Height versus Time'
```



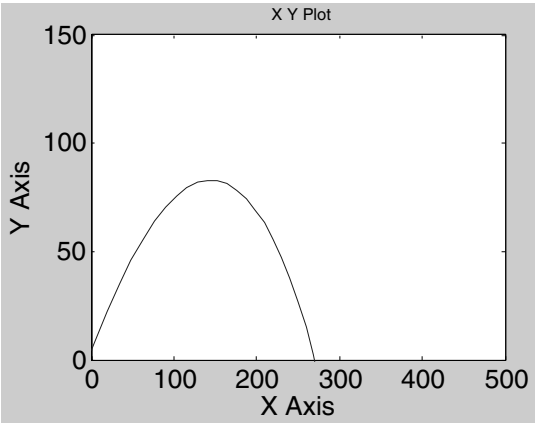

```
simplot(t, sqrt(x(:,3).^2 + x(:,4).^2))
axis([0 10 50 300])
title 'Speed versus Time'
```



Thus the ball stays in the air about 5.0 seconds and is traveling at about 115 ft/sec when it hits the ground. (Notice that the simulation continues beyond this time; since the model doesn't take into account hitting the ground, the height continues to decrease and the velocity continues to increase as if the ball had been batted off a cliff.)

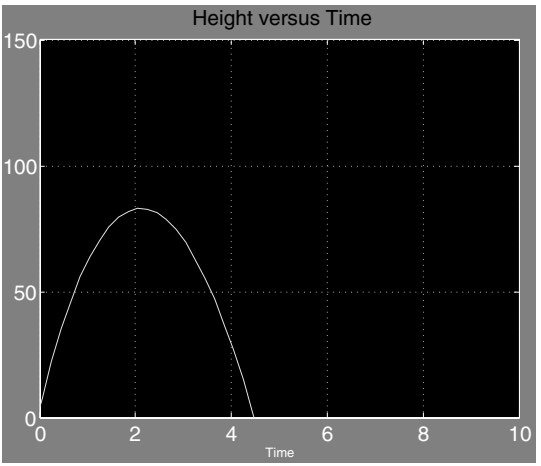
Now let's see what happens when we factor in air resistance, again with the initial velocity set to $[80, 80]$. First we take $c = 0.0017$. The trajectory now looks like this:

```
set_param('baseball/Gain', 'Gain', '-0.0017')
[t, x] = sim('baseball');
```

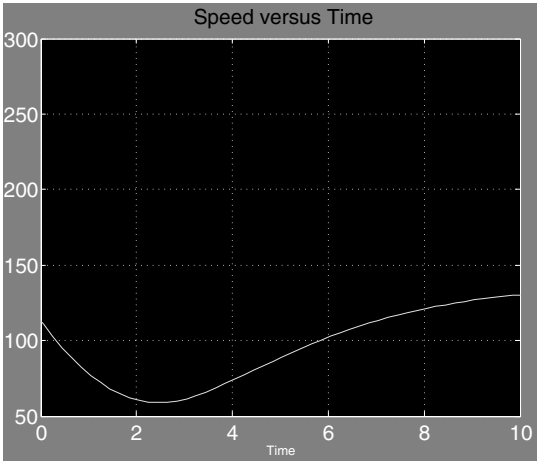


Note the enormous difference air resistance makes; the ball travels only about 270 feet. We can also investigate the flight time and speed with the other two scopes:

```
simplot(t, x(:,2))
axis([0 10 0 150])
title 'Height versus Time'
```




```
simplot(t, sqrt(x(:,3).^2 + x(:,4).^2))
axis([0 10 50 300])
title 'Speed versus Time'
```

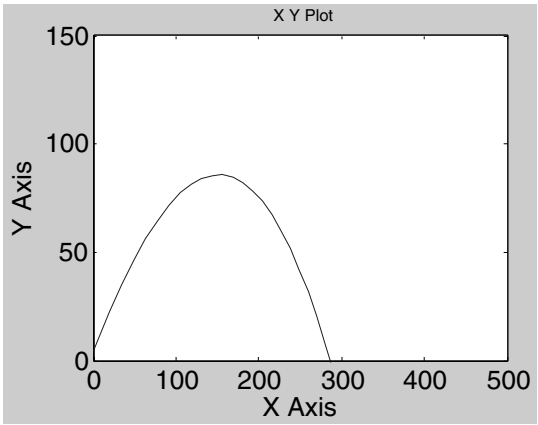


So the ball is about 80 feet up in the air at its peak, and hits the ground in about 4.5 seconds.

(b)

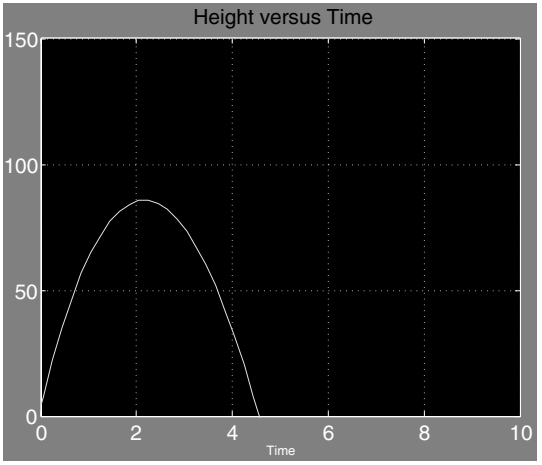
Let's now redo exactly the same calculation with $c = 0.0014$ (corresponding to playing in Denver). The ball's trajectory is now

```
set_param('baseball/Gain', 'Gain', '-0.0014')
[t, x] = sim('baseball');
```



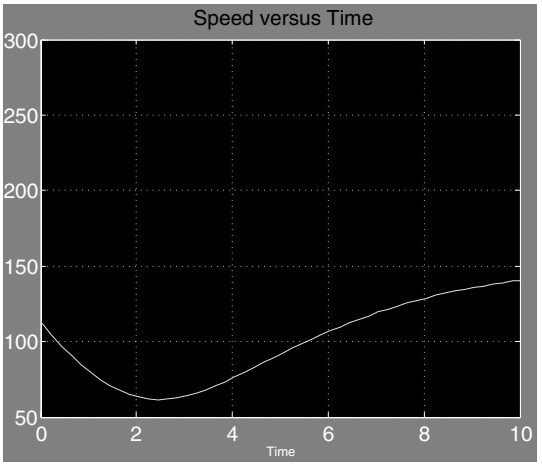
The ball goes about 285 feet, or about 15 feet further than when playing at sea level. This particular ball is probably an easy play, but, with some hard-hit balls, those extra 15 feet could mean the difference between an out and a home run. If we look at the height scope for the Denver calculation, we see:

```
simplot(t, x(:,2))
axis([0 10 0 150])
title 'Height versus Time'
```



so there is a very small increase in the flight time. Similarly, if we look at the speed scope for the Denver calculation, we see:

```
simplot(t, sqrt(x(:,3).^2 + x(:,4).^2))
axis([0 10 50 300])
title 'Speed versus Time'
```



So the final speed is a bit faster, about 83 ft/sec.

(c)

One would expect that batting averages and earned run averages would both be higher in Denver, as indeed is the case according to Major League Baseball statistics.

14.

The following M-file performs the desired task:

```
type editrecent
```

```
function editrecent

directory = dir('*.m');
if any(size(directory) == 0)
    error('No M-files found.')
end
dates = datenum({directory.date});
[maximum, index] = max(dates);
edit(directory(index).name)
```

By assigning the output of **dir** to a variable, we get a structure array that contains among other things the names of all the M-files in the current directory and their modification dates. If the structure array has 0 elements, we print an error message. Otherwise, we form the dates into a cell array of strings, and use **datenum** to convert the strings into numbers. We find the index of the largest number, indicating the most recent date, and finally pass the name of the corresponding file to **edit**.

15.

As suggested in the hint, we start by defining an array **Z** of complex numbers whose real parts are chosen from the vector **xvals** and whose imaginary parts are chosen from the vector **yvals**.

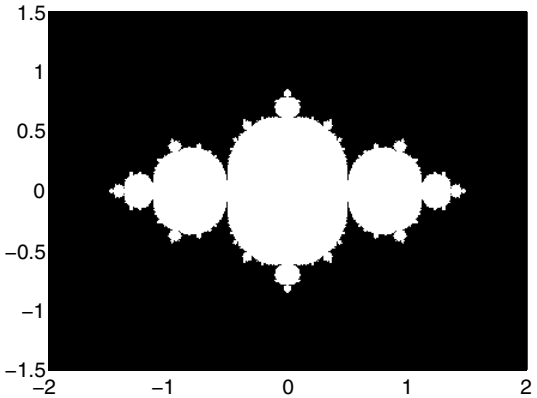
```
xvals = linspace(-2, 2, 400);
yvals = linspace(-1.5, 1.5, 300);
[X, Y] = meshgrid(xvals, yvals);
Z = X + i*Y;
```

Next we then apply the given function 100 times to each number in the array. By that time, most of the sequences that are going to diverge to infinity have already become larger than the largest floating-point number that MATLAB can store.

```
for k = 1:100
    Z = Z.^2 - 0.75;
end
```

We now use **isfinite** to assign a value of 1 to the values in the array that are still finite floating-point numbers and 0 to those that have diverged. We then use **imagesc** to color the points with ones differently from those with zeros, and select a gray colormap to make the colors white and black. Finally, we use **set** to reverse the vertical axis; without this command, the lower numbers would appear at the top of the graph. This is the default for **image** and **imagesc**, but is not the way mathematical graphs are normally drawn.

```
clf reset
set(gcf, 'Color', 'White')
imagesc(xvals, yvals, isfinite(Z))
colormap(gray)
set(gca, 'YDir', 'normal')
axis equal tight
```



The Julia set consists (approximately) of the boundary between the black and white regions. This boundary is an example of a "fractal," and you can draw a variety of other fractals by changing the expression $z^2 - 0.75$. For example, try changing 0.75 to $0.75 + 0.1i$ and see what a difference it makes!

Glossary

We present here the most commonly used MATLAB objects in six categories: operators, built-in constants, built-in functions, commands, graphics commands, and MATLAB programming constructs. After this we list the most important Simulink commands and blocks. Though MATLAB does not distinguish between commands and functions, it is convenient to think of a MATLAB function as we normally think of mathematical functions. A MATLAB function is something that can be evaluated or plotted; a command is something that manipulates data or expressions or that initiates a process.

We list each operator, function, and command together with a short description of its effect, followed by one or more examples. Many MATLAB commands can appear in a number of different forms, because you can apply them to different kinds of objects. In our examples, we have illustrated the most commonly used forms of the commands. Many commands also have numerous optional arguments; in this Glossary, we have included only some very common options. You can find a full description of all forms of a command, and get a more complete accounting of all the optional arguments available for it, by reading the help text – which you can access by typing either **help <commandname>** or **doc <commandname>**.

This Glossary is not comprehensive. We have selected those MATLAB commands that play a prominent role in this book, or that we believe will be most helpful to the user. You can find a more comprehensive list in the Help Browser.

☞ *See* Online Help *in Chapter 2 for a detailed description of the Help Browser.*

MATLAB Operators

**** Left matrix division. **X = A\B** is the solution of the equation **A*X = B**. Type **help slash** for more information.

```
A = [1 0; 2 1]; B = [3; 5];
A\B
```

/ Ordinary scalar division, or right matrix division. For matrices, **A/B** is essentially equivalent to **A*inv(B)**. Type **help slash** for more information.

***** Scalar or matrix multiplication. See the online help for **mtimes**.

. Not a true MATLAB operator. Used in conjunction with arithmetic operators to force element-by-element operations on arrays. Also used to access fields of a structure array.

```
a = [1 2 3]; b = [4 -6 8];
a.*b
syms x y; solve(x + y - 2, x - y); ans.x
```

.* Element-by-element multiplication of arrays. See the previous entry and the online help for **times**.

^ Scalar or matrix powers. See the online help for **mpower**.

.^ Element-by-element powers. See the online help for **power**.

: Range operator, used for defining vectors and matrices. Type **help colon** for more information.

, Separates elements of a row of a matrix, or arguments to a command. Can also be used to separate commands on a command line.

; Suppresses output of a MATLAB command, and can be used to separate commands on a command line. Also used to separate the rows of a matrix or column vector.

```
X = 0:0.1:30;
[1; 2; 3]
```

' Complex conjugate transpose of a matrix. See **ctranspose**. Also delimits the beginning and end of a string.

.' Transpose of a matrix. See **transpose**.

... Line-continuation operator. Cannot be used inside quoted strings. Type **help punct** for more information.

```
1 + 3 + 5 + 7 + 9 + 11 ...
       + 13 + 15 + 17
['This is a way to create very long strings that ', ...
'span more than one line.  Note the square brackets.']
```

! Runs a command from the operating system.

```
!C:\Programs\program.bat
```

% Comment. MATLAB will ignore the rest of the same line.

%% Starts a new cell (if it occurs at the beginning of a line in a script M-file).

@ Creates a function handle or anonymous function.

```
fminbnd(@cos, 0, 2*pi)
f = @(x) x.^2 - x
```

Built-in Constants

eps Roughly the size of the computer's floating-point round-off error; on most computers it is around 2×10^{-16}.

exp(1) $e = 2.71828\ldots$. Note that **e** has no special meaning.

i $i = \sqrt{-1}$. This assignment can be overridden, e.g., if you want to use **i** as an index in a **for** loop. In that case **j** can be used for the imaginary unit.

Inf ∞. Also **inf**.

NaN Not a number. Used for indeterminate expressions like $0/0$.

pi $\pi = 3.14159\ldots$.

Built-in Functions

abs $|x|$.

acos $\arccos x$.

asin $\arcsin x$.

atan $\arctan x$. Use **atan2** instead if you want the angular coordinate θ of the point (x, y).

bessel Bessel functions; **besselj(n, x)** and **bessely(n, x)** are linearly independent solutions of Bessel's equation of order n.

conj Gives the complex conjugate of a complex number.

```
conj(1 - 5*i)
```

cos $\cos x$.

cosh $\cosh x$.

cot $\cot x$.

erf The error function $\mathrm{erf}(x) = (2/\sqrt{\pi}) \int_0^x e^{-t^2}\, dt$.

exp e^x.

expm Matrix exponential.

gamma The gamma function $\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$ (when Re $x > 0$). The property $\Gamma(k+1) = k!$, for non-negative integers k, is sometimes useful.

imag $\mathrm{imag}(z)$, the imaginary part of a complex number.

log The natural logarithm $\ln x = \log_e x$.

real $\mathrm{real}(z)$, the real part of a complex number.

sec $\sec x$.

sech $\mathrm{sech}\, x$.

sign Returns -1, 0, or 1, depending on whether the argument is negative, zero, or positive.

sin $\sin x$.

sinh $\sinh x$.

sqrt $\sqrt{x}$.

tan $\tan x$.

tanh $\tanh x$.

MATLAB Commands

addpath Adds the specified directory to MATLAB's file-search path.

```
addpath C:\my_mfiles
```

ans A variable holding the value of the most recent unassigned output.

cd Makes the specified directory the current (working) directory.

```
cd C:\mydocs\mfiles
```

char Converts a symbolic expression into a string. Useful for defining inline functions.

```
syms x y; f = inline(char(sin(x)*sin(y)))
```

clear Clears values and definitions for variables and functions. If you specify one or more variables, then only those variables are cleared.

```
clear
clear f g
```

collect Collects coefficients of powers of the specified symbolic variable in a given symbolic expression.

```
syms x y; collect(x^2 - 2*y*x^2 + 3*x + x*y, x)
```

compose Composition of functions.

```
syms x y; f = exp(x); g = sin(y); h = compose(f, g)
```

ctranspose Conjugate transpose of a matrix. Usually invoked with the **'** operator. Equivalent to **transpose** for real matrices.

```
A = [1 3 i]
A'
```

D Not a true MATLAB command. Used in **dsolve** to denote differentiation. See **diff**.

```
dsolve('x*Dy + y = sin(x)', 'x')
```

delete Deletes a file.

```
delete <filename>
```

det The determinant of a matrix.

```
det([1 3; 4 5])
```

diag Gives a square matrix with a prescribed diagonal vector, or picks out the diagonal in a square matrix.

```
V = [2 3 4 5]; diag(V)
X = [2 3; 4 5]; diag(X)
```

diary Writes a transcript of a MATLAB session to a file.

```
diary <filename>
diary off
```

diff Symbolic differentiation operator (also difference operator).

```
syms x; diff(x^3)
diff('x*y^2', 'y')
```

dir Lists the files in the current working directory. Similar to **ls**.

disp Displays output without first giving its name.

```
x = 5.6; disp(x)
syms x; disp(x^2)
disp('This will print without quotes.')
```

doc Opens the Help Browser to documentation on a specific command.

```
doc print
```

double Gives a double-precision value for either a numerical or a symbolic quantity. Applied to a string, **double** returns a vector of ASCII codes for the characters in the string.

```
z = sym('pi'); double(z)
double('think')
```

dsolve Symbolic ODE solver. By default, the independent variable is t, but a different variable can be specified as the last argument.

```
dsolve('D2y - x*y = 0', 'x')
dsolve('Dy + y^2 = 0', 'y(0) = 1', 'x')
[x, y] = dsolve('Dx = 2x + y', 'Dy = -x')
```

echo Turns on or off the echoing of commands inside script M-files.

edit Opens the specified M-file in the Editor/Debugger.

```
edit mymfile
```

eig Computes eigenvalues and eigenvectors of a square matrix.

```
eig([2, 3; 4, 5])
[e, v] = eig([1, 0, 0; 1, 1, 1; 1, 2, 4])
```

end Last entry of a vector. Also a programming command.
```
v(end)
v(3:end)
```

eval Evaluates a string as a MATLAB expression. Useful in M-files.
```
eval('cos(x)')
```

expand Expands an algebraic expression.
```
syms x y; expand((x - y)^2)
```

eye The identity matrix of the specified size.
```
eye(5)
```

factor Factors a polynomial or integer.
```
syms x y; factor(x^4 - y^4)
```

feval Evaluates a function specified by a string. Useful in function M-files.
```
feval('exp', 1)
```

find Finds the indices of non-zero elements of a vector or matrix.
```
X = [2 0 5]; find(X)
```

fminbnd Finds the smallest (approximate) value of a function over an interval.
```
fminbnd(@(x) x^4 - x^2 + 1, 0, 1)
```

format Specifies the output format for numerical variables.
```
format long
```

fzero Tries to find a zero of the specified function near a given starting point or on a specified interval.
```
fzero(@(x) cos(x) - x, 1)
fzero(@cos, [-pi 0])
```

guide Opens the GUI Design Environment.
```
guide mygui
```

help Asks for documentation for a MATLAB command. See also **lookfor**.
```
help factor
```

inline Constructs a MATLAB inline function from a string expression.
```
f = inline('x^5 - x'); f(3)
```

int Integration operator for both definite and indefinite integrals.
```
int('1/(1 + x^2)', 'x')
syms x; int(exp(-x), x, 0, Inf)
```

inv Inverse of a square matrix.
```
inv([1 2; 3 5])
```

jacobian Computes the Jacobian matrix, or, for a scalar function, the symbolic gradient.
```
syms x y; f = x^2*y^3; jacobian(f)
```

length Returns the number of elements in a vector or string.
```
length('abcde')
```

limit Finds a two-sided limit, if it exists. Use **'right'** or **'left'** for one-sided limits.
```
syms x; limit(sin(x)/x, x, 0)
syms x; limit(1/x, x, Inf, 'left')
```

linspace Generates a vector of linearly spaced points.
```
linspace(0, 2*pi, 30)
```

load Loads Workspace variables from a disk file.

```
load filename
```

lookfor Searches for a specified string in the first line of all M-files found in the MATLAB path.

```
lookfor ode
```

ls Lists files in the current working directory. Similar to **dir**.

maple Provides direct access to the Maple kernel. Not available in the Student Version.

```
maple('csgn', '-1+i')
```

mhelp Queries the Maple kernel for help on a Maple command. Not available in the Student Version.

```
mhelp csgn
```

max Computes the arithmetic maximum of the entries of a vector.

```
X = [3 5 1 -6 23 -56 100]; max(X)
```

mean Computes the arithmetic average of the entries of a vector.

```
X = [3 5 1 -6 23 -56 100]; mean(X)
```

median Computes the arithmetic median of the entries of a vector.

```
X = [3 5 1 -6 23 -56 100]; median(X)
```

min Computes the arithmetic minimum of the entries of a vector.

```
X = [3 5 1 -6 23 -56 100]; min(X)
```

more Turns on (or off) page-by-page scrolling of MATLAB output. Use the SPACE BAR to advance to the next page, the RETURN key to advance line-by-line, and Q to abort the output.

```
more on, help print
more off
```

notebook Opens an M-Book (Windows only).

```
notebook problem1.doc
notebook -setup
```

num2str Converts a number to a string. Useful in programming.

```
disp(['The value of pi is ', num2str(pi)])
```

ode45 Numerical ODE solver for first-order equations. See MATLAB's online help for **ode45** for a list of other MATLAB ODE solvers.

```
[t, y] = ode45(@(t, y) t^2 + y, [0 10], 1);
plot(t, y)
```

ones Creates a matrix of ones.

```
ones(3)
ones(3, 1)
```

open Opens a file. The way this is done depends on the filename extension.

```
open myfigure.fig
```

path Without an argument, displays the search path. With an argument, sets the search path. Type **help path** for details.

pathtool Opens the "Set Path" tool.

pretty Displays a symbolic expression in a more readable format.

```
syms x y; expr = x/(x - 3)/(x + 2/y)
pretty(expr)
```

prod Computes the product of the entries of a vector.

```
X = [3 5 1 -6 23 -56 100]; prod(X)
```

publish Runs an M-file and "publishes" it. The default is to publish to `html`, but one can specify other formats.

```
publish('mymfile', 'latex')
```

pwd Shows the name of the current (working) directory.

quadl Numerical integration command. In MATLAB 5.3 or earlier, use **quad8** instead.

```
format long; quadl(@(x) sin(exp(x)), 0, 1)
g = inline('sin(exp(x))'); quad8(g, 0, 1)
```

quit Terminates a MATLAB session.

rand Random-number generator; creates arrays of random numbers between 0 and 1.

randn Normal random-number generator; creates arrays of normal random numbers with mean 0 and variance 1.

rank Gives the rank of a matrix.

```
A = [2 3 5; 4 6 8]; rank(A)
```

roots Finds the roots of a polynomial whose coefficients are given by the elements of the vector argument.

```
roots([1 2 2])
```

round Rounds a number to the nearest integer.

save Saves Workspace variables to a specified file. See also **diary** and **load**.

```
save filename
```

simple Attempts to simplify an expression using multiple methods.

```
syms x y;
[expression, how] = simple(sin(x)*cos(y) + cos(x)*sin(y))
```

simplify Attempts to simplify an expression symbolically.

```
syms x; simplify(1/(1 + x)^2 - 1/(1 - x)^2)
```

size Returns the number of rows and the number of columns in a matrix.

```
A = [1 3 2; 4 1 5]
[r, c] = size(A)
```

solve Solves an equation or set of equations. If the right-hand side of the equation is omitted, "0" is assumed.

```
solve('2*x^2 - 3*x + 6')
[x, y] = solve('x + 3*y = 4', '-x - 5*y = 3', 'x', 'y')
```

sound Plays a vector through the computer speakers.

```
sound(sin((0:0.1:1000)*pi))
```

str2num Converts a string to a number. Useful in programming.

```
constant = 'a7'
index = str2num(constant(2))
```

subs Substitutes for parts of an expression.

```
subs('x^3 - 4*x + 1', 'x', 2)
subs('sin(x)^2 + cos(x)', 'sin(x)', 'z')
```

sum Sums a vector, or sums the columns of a matrix.

```
k = 1:10; sum(k)
```

sym Creates a symbolic variable or number.

```
sym pi
```

```
x = sym('x')
constant = sym('1/2')
```

syms Defines symbolic variables – **syms x** is equivalent to **x = sym('x')**.

```
syms x y z
```

symsum Performs a symbolic summation of a vector, possibly with infinitely many entries.

```
syms x k n; symsum(x^k, k, 0, n)
syms n; symsum(n^(-2), n, 1, Inf)
```

taylor Gives a Taylor polynomial approximation with a specified number of terms (the default is 6) at a specified point (default 0). Note: the number of terms includes the constant term, so the default is a polynomial of degree 5, not degree 6.

```
syms x; taylor(cos(x), 8, 0)
taylor(exp(1/x), 10, Inf)
```

transpose Transpose of a matrix (compare **ctranspose**). Converts a column vector into a row vector, and vice versa. Usually invoked with the **.'** operator.

```
A = [1 3 4]
A.'
```

type Displays the contents of a specified file.

```
type myfile.m
```

vectorize Vectorizes a symbolic expression. Useful in defining inline functions.

```
f = inline(vectorize('x^2 - 1/x'))
```

vpa Evaluates an expression to the specified degree of accuracy using variable-precision arithmetic.

```
vpa('1/3', 20)
```

web Opens a web browser.

```
web('http://www.mathworks.com')
```

which Displays the pathname of a command with a given name.

```
which ezplot
which ezplot -all
```

whos Lists current information on all the variables in the Workspace.

zeros Creates a matrix of zeros.

```
zeros(10)
zeros(3, 1)
```

Graphics Commands

area Produces a shaded graph of the area between the x-axis and a curve.

```
X = (0:0.01:4)*pi; Y = sin(X); area(X, Y)
```

axes Creates an empty figure window.

axis Sets axis scaling and appearance.

axis([xmin xmax ymin ymax]) – sets ranges for the axes.
axis tight – sets the axis limits to the full range of the data.
axis equal – makes the horizontal and vertical scales equal.
axis square – makes the axis box square.
axis off – hides the axes and tick marks.

bar Draws a bar graph.

```
bar([2, 7, 1.5, 6])
```

cla Clears axes.

close Closes the current figure window. **close all** closes all figure windows.

colormap Sets the colormap features of the current figure; type **help graph3d** to see examples of colormaps.

```
ezmesh sin(x)*cos(y); colormap cool
```

comet Displays an animated parametric plot.

```
t = (0:0.01:4)*pi; comet(t.*cos(t), t.*sin(t))
```

contour Plots the level curves of a function of two variables; usually used with **meshgrid**.

```
[X, Y] = meshgrid(-3:0.1:3, -3:0.1:3);
contour(X, Y, X.^2 - Y.^2)
```

contourf Filled contour plot. Often used with **colormap**.

```
[X,Y] = meshgrid(-2:0.1:2, -2:0.1:2);
contourf(X, Y, X.^2 - Y.^3);
colormap autumn
```

ezcontour Easy plot command for contour or level curves.

```
ezcontour('x^2 - y^2')
syms x y; ezcontour(x - y^2)
```

ezmesh Easy plot command for mesh view of surfaces.

```
ezmesh('x^2 + y^2')
syms x y; ezmesh(x*y)
```

ezplot Easy plot command for symbolic expressions.

```
ezplot('exp(-x^2)', [-5, 5])
syms x; ezplot(sin(x))
```

ezplot3 Easy plot command for 3D parametric curves.

```
ezplot3('cos(t)', 'sin(t)', 't')
syms t; ezplot3(1 - cos(t), t - sin(t), t, [0 4*pi])
```

ezsurf Easy plot command for standard shaded view of surfaces.

```
ezsurf('(x^2 + y^2)*exp(-(x^2 + y^2))')
syms x y; ezsurf(sin(x*y), [-pi pi -pi pi])
```

figure Creates a new figure window.

fill Creates a filled polygon. See also **patch**.

```
fill([0 1 1 0], [0 0 1 1], 'b'); axis equal tight
```

findobj Finds graphics objects with specified property values.

```
findobj('Type', 'Line')
```

gca Gets current axes.

gcf Gets current figure.

get Gets properties of a figure.

```
get(gcf)
```

getframe Command to get the frames of a movie or animation.

```
T = (0:0.01:2)*pi;
for j = 1:12
    plot(5*cos(j*pi/6) + cos(T), 5*sin(j*pi/6) + sin(T));
```

```
        axis([-6 6 -6 6]);
        M(j) = getframe;
    end
    movie(M)
```

ginput Gathers coordinates from a figure using the mouse (press the ENTER or RETURN key to finish).

```
[X, Y] = ginput
```

grid Puts a grid on a figure.

gtext Places a text label using the mouse.

```
gtext('Region of instability')
```

hist Draws a histogram.

```
hist(rand(200, 1))
```

hold Holds the current graph. Superimpose any new graphics generated by MATLAB on top of the current figure.

```
hold on
hold off
```

image Displays a matrix as an image.

```
image(ones(50,100))
```

imagesc Like **image**, but scales the data if necessary.

```
imagesc(randn(50,100))
```

imread Reads in a graphics file and converts it to a matrix.

```
A = imread('myimage.jpg');
```

imwrite Converts a matrix into a graphics file.

```
imwrite(A, 'picture.jpg')
```

legend Creates a legend for a figure.

```
t = 0:0.1:2*pi;
plot(t, cos(t), t, sin(t))
legend('cos(t)', 'sin(t)')
```

loglog Creates a log-log plot.

```
x = 0.0001:0.1:12; loglog(x, x.^5)
```

mesh Draws a mesh surface.

```
[X,Y] = meshgrid(-2:.1:2, -2:.1:2);
mesh(X, Y, sin(pi*X).*cos(pi*Y))
```

meshgrid Creates a vector array that can be used as input to a graphics command, for example, **contour**, **quiver**, or **surf**.

```
[X, Y] = meshgrid(0:0.1:1, 0:0.1:2)
contour(X, Y, X.^2 + Y.^2)
```

movie Plays back a movie. See the entry for **getframe**.

movieview Similar to **movie**, but has a playback button.

patch Creates a filled polygon or colored surface patch. See also **fill**.

```
t = (0:1:5)*2*pi/5; patch(cos(t), sin(t), 'r'); axis equal
```

pie Draws a pie plot of the data in a vector.

```
Z = [34 5 32 6]; pie(Z)
```

plot Plots vectors of data.

```
X = [0:0.1:2];
plot(X, X.^3)
```

plot3 Plots curves in three-dimensional space.

```
t = [0:0.1:30];
plot3(t, t.*cos(t), t.*sin(t))
```

polar Polar-coordinate plot command.

```
theta = (0:0.01:2)*pi; rho = theta; polar(theta, rho)
```

print Sends the contents of the current figure window to the printer or to a file.

```
print
print -deps picture.eps
```

quiver Plots a (numerical) vector field in the plane.

```
[x, y] = meshgrid(-4:0.5:4, -4:0.5:4);
quiver(x, y, x.*(y - 2), y.*x); axis tight
```

semilogy Creates a semi-log plot, with the logarithmic scale along the vertical axis.

```
x = 0:0.1:12; semilogy(x, exp(x))
```

set Sets properties of a figure.

```
set(gcf, 'Color', [0, 0.8, 0.8])
```

subplot Breaks the figure window into a grid of smaller plots.

```
subplot(2, 2, 1), ezplot('x^2')
subplot(2, 2, 2), ezplot('x^3')
subplot(2, 2, 3), ezplot('x^4')
subplot(2, 2, 4), ezplot('x^5')
```

surf Draws a solid surface.

```
[X,Y] = meshgrid(-2:.1:2, -2:.1:2);
surf(X, Y, sin(pi*X).*cos(pi*Y))
```

text Annotates a figure, by placing text at specified coordinates.

```
text(x, y, 'string')
```

title Assigns a title to the current figure window.

```
title 'Nice Picture'
```

xlabel Assigns a label to the horizontal coordinate axis.

```
xlabel('Year')
```

ylabel Assigns a label to the vertical coordinate axis.

```
ylabel('Population')
```

view Specifies a point from which to view a 3D graph.

```
ezsurf('(x^2 + y^2)*exp(-(x^2 + y^2))'); view([0 0 1])
syms x y; ezmesh(x*y); view([1 0 0])
```

zoom Rescales a figure by a specified factor; **zoom** by itself enables use of the mouse for zooming in or out.

```
zoom
zoom(4)
```

MATLAB Programming

any True if any element of an array is non-zero.

```
if any(imag(x) ~= 0); error('Inputs must be real.'); end
```

all True if all the elements of an array are non-zero.

break Breaks out of a **for** or **while** loop.

case Used to delimit cases after a **switch** statement.

computer Outputs the type of computer on which MATLAB is running.

dbclear Clears breakpoints from a file.

```
dbclear all
```

dbcont Returns to an M-file after stopping at a breakpoint.

dbstep Executes an M-file line-by-line after stopping at a breakpoint.

dbstop Inserts a breakpoint in a file.

```
dbstop in <filename> at <linenumber>
```

dbquit Terminates an M-file after stopping at a breakpoint.

dos Runs a command from the operating system, saving the result in a variable. Similar to **unix**.

end Terminates an **if**, **for**, **while**, or **switch** statement.

else Alternative in a conditional statement. See **if**.

elseif Nested alternative in a conditional statement. See the online help for **if**.

error Displays an error message and aborts execution of an M-file.

find Reports indices of non-zero elements of an array.

```
n = find(isspace(mystring));
if ~isempty(n)
    firstword = mystring(1:n(1)-1);
    restofstring = mystring(n(1)+1:end);
end
```

for Begins a loop. Must be terminated by **end**.

```
close; axes; hold on
t = -1:0.05:1;
for k = 0:10
    plot(t, t.^k)
end
```

function Used on the first line of an M-file to make it a function M-file.

```
function y = myfunction(x)
```

if Allows conditional execution of MATLAB statements. Must be terminated by **end**.

```
if (x >= 0)
    sqrt(x)
else
    error('Invalid input.')
end
```

input Prompts for user input.

```
answer = input('Please enter [x, y] coordinates:  ')
```

isa Checks whether an object is of a given class (**double**, **sym**, etc.).

```
isa(x, 'sym')
```

ischar True if an array is a character string.

isempty True if an array is empty.

isequal Checks whether two arrays are equal. Unlike **==**, does not produce an error message if the arrays have different sizes.
```
isequal([1 2], [1 2 3])
```

isfinite Checks whether elements of an array are finite.
```
isfinite([1 0 1]./[1 0 0])
```

ishold True if **hold on** is in effect.

isinf Checks whether elements of an array are infinite.
```
isinf([1 0 1]./[1 0 0])
```

isletter Checks whether elements of a string are letters of the alphabet.
```
str = 'remove my spaces'; str(isletter(str))
```

isnan Checks whether elements of an array are "not-a-number" (which results from indeterminate forms like $0/0$).
```
isnan([-1 0 1]/0)
```

isnumeric True if an object is of a numerical class.

ispc True if MATLAB is running on a Windows computer.

isreal True if an array consists only of real numbers.

isspace Checks whether elements of a string are spaces, tabs, etc.

isunix True if MATLAB is running on a UNIX computer.

keyboard Returns control from an M-file to the keyboard. Useful for debugging M-files.

mex Compiles a MEX program.

mlint Checks an M-file for common syntax errors.
```
mlint mymfile
```

nargin Returns the number of input arguments passed to a function M-file.
```
if (nargin < 2); error('Wrong number of arguments'); end
```

nargout Returns the number of output arguments requested from a function M-file.

otherwise Used to delimit an alternative case after a **switch** statement.

pause Suspends execution of an M-file until the user presses a key.

return Terminates execution of an M-file early, or returns to an M-file after a **keyboard** command.
```
if abs(err) < tol; return; end
```

switch Alternative to **if** that allows branching to more than two cases. Must be terminated by **end**.
```
switch num
case 1
    disp('Yes.')
case 0
    disp('No.')
otherwise
    disp('Maybe.')
end
```

unix Runs a command from the operating system, saving the result in a variable. Similar to **dos**.

varargin Used in a function M-file to handle a variable number of inputs.

`varargout` Used in a function M-file to allow a variable number of outputs.

`warning` Displays a warning message.

```
warning('Taking the square root of negative number.')
```

`while` Repeats a block of commands until a condition fails to be met. Must be terminated by **`end`**.

```
mysum = 0;
x = 1;
while x > eps
    mysum = mysum + x;
    x = x/2;
end
mysum
```

Simulink Commands

`find_system` Finds name of desired Simulink system.

```
find_system(gcs, 'Type', 'block')
```

`gcs` "Get current system." Returns name of current Simulink model.

`get_param` Gets parameters of a Simulink block or system.

`open_system` Opens a Simulink model.

`set_param` Sets parameters of a Simulink block or system.

`sim` Runs a Simulink model.

```
sim('model', [0, 20])
```

`simlp` Solves a linear programming problem.

```
simlp([-1 -1], [1 4; 2 3; 4 1], [3; 2; 3])
```

`simplot` Plots data in the style of a Simulink Scope window. Similar to **`plot`** in most respects.

`simulink` Opens the Simulink library.

Simulink Blocks

Algebraic Constraint (Math Operations Library) Equation solver like **`fzero`**.

Clock (Sources Library) Outputs the simulation time.

Constant (Sources Library) Outputs a specified constant.

Demux (Signal Routing Library) Disassembles a vector signal into its components.

From Workspace (Sources Library) Takes input from the MATLAB Workspace.

Gain (Math Operations Library) Multiplies by a constant or a constant matrix.

Integrator (Continuous Library) Computes the definite integral, using a specified initial condition.

Math Function (Math Operations Library) Computes exponentials, logarithms, etc.

Mux (Signal Routing Library) Assembles scalar signals into a vector signal.

Polynomial (Math Operations Library) Evaluates a polynomial function.

Product (Math Operations Library) Multiplies or divides signals. Can also invert matrix signals.

Ramp (Sources Library) Outputs a function that is initially constant and then increases linearly starting at the specified time.

Scope (Sinks Library) Plots a signal as a function of time.

Sine Wave (Sources Library) Outputs a sine wave. You can adjust the amplitude, frequency, and phase.

Sum (Math Operations Library) Adds or subtracts inputs.

To Workspace (Sinks Library) Outputs a signal to the Workspace.

Trigonometric Function (Math Operations Library) Computes trigonometric or hyperbolic functions.

Unit Delay (Discrete Library) Useful in modeling difference or differential/difference equations.

XY Graph (Sinks Library) Plots one signal against another.

Index

!, 101, 288
', 13, 15, 288
'*', 70
'+', 70
'-', 70
'--', 70
':', 70
*****, 9, 287
,, 11, 45, 288
., 14, 21, 24, 287
.', 288
.*, 14, 52, 287
..., 38, 47, 109, 288
./, 14, 52
.^, 13, 24, 52, 288
/, 287
:, 13, 53, 90, 96, 254, 265, 288
;, 14, 15, 24, 34, 35, 41, 45, 60, 77, 288
=, 18, 45
==, 87, 92, 99, 299
?, 4
@, 16, 33, 49, 65, 97, 98, 288
%, 32, 51, 288
%%, 32, 36, 104, 288
&, 87, 88
****, 53, 70, 245, 287
^, 287
|, 87, 88
~, 88, 89
~=, 89

aborting calculations, 9
abs, 86, 288
accuracy, floating-point, 93
acos, 288
activate, 127
Add, 119
addition, 8
addpath, 29, 33, 289
agribusiness, 171
air resistance, 202, 203, 279
Algebraic Constraint, 300
all, 89, 298
amperes, 200
amplitude, 78
animation, 77, 295
anonymous function, 16, 18, 22, 49, 65, 97, 98, 288
anonymous-function, 33
ans, 8, 51, 96, 206, 289
any, 89, 297
area, 294
argument, 34, 46, 51, 52, 95, 96
arithmetic, 8
    complex, 51, 52
    exact, 8, 11
    floating-point, 11
    variable-precision, 11
arithmetic operations, 207, 211
arithmetic symbols, 208, 211
array, 14, 18, 46
Array Editor, 28
asin, 55, 288
assignment, 18, 19, 87, 206
asterisk, 8
atan, 15, 43, 289
atan2, 51, 98, 289
Automation Server, 106
axes, 22, 70
axes, 294
Axes Properties..., 23
axis, 23, 44, 59–61, 68, 76, 77, 202, 294
axlimdlg, 129

backslash, 53, 81, 245
Band-Limited White Noise, 115
bank, 141
bank account, 259
bankrupt, 144
bar, 294
baseball, 203
batting average, 197, 203, 263, 283
Beethoven, 78
bell curve, 146
bessel, 289
besselj, 15, 67, 105

bifurcation point, 155
Block Parameters, 114, 119, 121
blocks, 111
Boltzmann's constant, 200
border, 72
braces, 96, 206, 208
brackets, 96, 206–208
    curly, 208
branch line, 114
branching, 85, 87, 89, 93
break, 93, 298
breakpoint, 100, 212, 215
built-in constant, 15
built-in function, 15, 211
bulb, 132
bulls-eye, 197
ButtonDownFcn, 130
Bytes, 18

C, xi, xiii, 101
calculus, 54
callback function, 123, 127–130
Camera Toolbar, 66
Captain Picard, 195
Cardassian, 195
case, 91, 208, 298
casino, 144
cd, 29, 33, 101, 289
Cell, 32, 36
cell, 32, 36, 104
    corrupted, 109
cell, 94
cell array, 94, 96, 208
Cell Markers, 107
Central Limit Theorem, 146, 147
chaos, 157, 158
char, 47, 289
char array, 17, 18
Checkbox, 126, 130
chessboard, 72
Children, 72
circuit, electrical, 199
cla, 295
class, 46
clear, 18, 28, 32, 50, 289
clear variables, 211
Clock, 119, 300
close, 24, 32, 66, 295
coefficient of friction, 172
collect, 289
colon, xiv, 13, 52, 90
colon, 288
Color, 72
color, 70
    background, 72
colorbar, 76
colormap, 74
colormap, 75, 76, 132, 284, 295
column, 52, 53
column vector, 13, 14, 46
comet, 76, 295
comma, 11, 45, 288
command history, 9
Command History Window, 2, 27, 30, 31, 206
command prompt, 2
Command Window, 2, 7, 18, 27, 28, 37, 38, 87, 103, 107, 108, 111
comment, 32, 36
common mistakes, 211
competing species, 201
complex arithmetic, 51, 52
complex number, 15, 21, 51, 52, 86, 88, 96
compose, 290
computer, 101, 298
computer won't respond, 205, 211
concatenation, 47
conductivity, thermal, 176, 180
conj, 289
Constant, 203, 300
constant, built-in, 15
containers, shipping, 115
Continuous library, 112
continuous model, 201
contour, 60, 64, 79, 195, 231, 295
contour plot, 60
contourf, 133, 256, 295
convex set, 169
cos, 15, 289
cosh, 289
cot, 289
crane, 115
ctranspose, 13, 15, 290
CTRL+C, 9, 37, 92, 211
CTRL+ENTER, 41, 107, 109
current directory, 28, 29, 31, 33, 39
Current Directory Browser, 2, 18, 27–29, 31
current, electrical, 200
CurrentAxes, 72

curve, 79
    level, 60
cylindrical coordinates, 64

D, 54, 290
damped harmonic oscillator, 111
dashed line, 70
data class, 18, 46, 49
Data Cursor, 66
data type, 94
datenum, 284
dbclear, 100, 212, 298
dbcont, 100, 212, 298
dblquad, 55
dbquit, 100, 298
dbstep, 100, 298
dbstop, 100, 212, 298
deal, 97
debug, 41, 100, 212
debugger, 31, 212
default variables, 57
definite integral, 54
definitions, conflicting, 206
delete, 37, 101, 290
delimiters, 207, 211
demo, 5
Demux, 300
Denver, Colorado, 203
Derivative, 115
derivative, 54, 79
descendents, 72
Desktop, xiii, 2, 3, 6, 7, 18, 27, 28, 111, 207
Desktop, 27
det, 81, 245, 290
determinant, 82, 246
diag, 253, 290
dialog box, 130
diary, 35, 37
diary, 37, 290
diff, 54, 290
difference equation, 201
differential equation, 111, 115, 201
differential/difference equation, 186, 301
differentiation, 54
diffusion coefficient, 176
diffusion equation, 175
digit, 19, 43
digits, 12
dingbats, xiv
diode, 200
dir, 28, 101, 284, 290
Dirichlet boundary condition, 176
discrete model, 201
discriminant, 224
disp, 38, 45, 98, 99, 141, 257, 290
display pane, 4
division, 8, 207
doc, 4, 290
Dock, 27
documentation, 5
doomsday, 202, 277
DOS, 101
dos, 101, 102, 298
dotted line, 70
double, 20, 47, 48, 290
double array, 17, 18
double-precision, 8, 18
down-arrow key, 9, 30
dsolve, 58, 201, 273, 290
dynamics, 149

e, 288
earned run average, 203
echo, 33, 35–38, 41, 290
Edit, 23
edit, 31, 85, 284, 290
Edit Text, 125, 127, 129
editing input, 9
Editor/Debugger, 31, 36, 41, 86, 100, 103, 104, 126–128, 212, 215, 216
editpath, 29
eig, 53, 142, 249, 290
eigenpair, 53
eigenvalue, 53, 83, 142, 187, 197, 265
eigenvector, 53, 83, 142, 187
element, 14, 52
element-by-element, 14, 24, 52
else, 85, 86, 89, 256, 298
elseif, 86–88, 92, 298
Enable Cell Mode, 32
end, 35, 86, 89, 91–93, 256, 291, 298
energy
    kinetic, 117
    potential, 117
engineering, 115
eps, 288
equal sign, 18, 45
erf, 15, 289
error, 19, 105, 241
    syntax, 9
error, 95, 98, 99, 284, 298

error function, 289
error message, 9, 205, 214, 284
    plotting, 209
error messages, multiple, 209
errors, common, 9
`eval`, 97, 129, 291
Evaluate Cell, 41
Evaluate Current Cell, 32
evaluation, 97
    suppressing, 48, 49
exact arithmetic, 8, 11
exchange matrix, 160
execute, 31
`exp`, 10, 14, 15, 289
`expand`, 10, 291
`expm`, 14, 289
exponential function, 14
    matrix, 14
exponential growth, 201
exponential growth/decay, 150
expression, 48
external program, 101
extinct, 150
`eye`, 53, 291
`ezcontour`, 61, 295
`ezcontourf`, 132, 133
`ezmesh`, 64, 65, 295
`ezplot`, 22, 25, 44, 59–61, 63, 94, 295
`ezplot3`, 63, 295
`ezsurf`, 64, 65, 243, 295

`factor`, 10, 47, 222, 291
factorial, 35
`factorial`, 80, 241
Feigenbaum parameter, 157
`feval`, 98, 291
field, vector, 62
figure, 77
    current, 66
`figure`, 37, 39, 42, 66, 108, 295
Figure Palette, 67, 70
Figure Toolbar, 67
figure window, 6
file extension, 28, 31, 114, 123, 126, 292
`fill`, 72, 73, 265, 295
`find`, 291, 298
`find_system`, 121, 300
`findobj`, 72, 129, 295
finite difference, 176
finite element method, 182
flag
    French, 83
    Italian, 83
`flipdim`, 75
`fliplr`, 75
`flipud`, 75
floating-point arithmetic, 8, 11
floating-point number, 46, 48
`fminbnd`, 291
`FontName`, 69, 71
`FontSize`, 69, 70
`FontWeight`, 69
football coach, 198
`fopen`, 102
`for`, 35, 39, 73, 85, 87, 92, 93, 257, 298
forced oscillations, 115
`format`, 30, 32, 33, 291
`format long`, 8
`format short`, 8
Fortran, xi, xiii, 101
`fprintf`, 102
fractal, 285
fraction, 8
frequency, 78
friction, 172
From Workspace, 300
Function, 119
function, 48
    built-in, 15
    user-defined, 16
`function`, 34, 94, 263, 298
function handle, 17, 18, 46, 49, 97, 98, 288
function M-file, 31, 33, 49, 50, 85, 94, 95
Fundamental Theorem of Linear Programming, 169
`fzero`, 20, 21, 44, 200, 225, 271, 291

Gain, 113, 119, 300
Galileo, 202
`gamma`, 15, 80, 241, 289
gamma function, 80, 289
`gca`, 70, 72, 295
`gcbf`, 129
`gcbo`, 129
`gcf`, 66, 71, 122, 295
`gcs`, 121, 300
genotype, 82, 83
geometric series, 57, 259
`get`, 71, 72, 128, 129, 295
`get_param`, 120, 300
`getframe`, 77, 295

ginput, 67, 99, 100, 296
gradient, 291
graph, 44
graph2d, 59
graph3d, 59
Graphical User Interface, xiii
graphics, 22, 38, 59, 66
    customizing, 71
    M-book, 108
gravity, 202
Greek letters, 70
grid, 296
gtext, 296
GUI, xiii, 123–130
GUIDE, 123, 125–129
guide, 123, 291
Gwynn, Tony, 197

half-life, 195
handle, 71, 72, 128, 129
heat equation, 131, 175
heat flow, 175
helix, 63
help, 3
help, 4, 5, 10, 23, 34, 51, 108, 209, 291
Help Browser, 4, 5, 58, 68, 70
Help Navigator, 4
help text, 46, 47, 51
helpdesk, 5, 68
helpwin, 5
hist, 296
histogram, 145, 198
hold, 25, 39, 42, 66, 76, 77, 99, 100, 296
homogeneous equation, 111
HTML, 35, 36, 103

i, 15, 20, 52, 288
if, 85–89, 91–93, 95, 256, 298
illumatons, 195
illumination, 131
im2frame, 255
imag, 289
image, 74
image, 72, 74–76, 204, 255, 296
imagesc, 76, 204, 284, 296
imaginary number, 52
implicit plot, 61
Import Wizard, 102
improper integral, 55
imread, 74, 296
imwrite, 76, 255, 296
increment, 13
    fractional, 13
    negative, 13
ind2rgb, 74
indefinite integral, 54, 236
indeterminate form, 89
indexed image, 74
Inf, 15, 56, 288
infinite loop, 92
inhomogeneous equation, 115
initial condition, 114, 115
initial value, 201
initialize variables, 211
inline, 16, 33, 49, 291
inline function, 18, 46, 49, 97, 98
inline object, 17, 18
input, 7, 51, 95, 96
    editing, 9
input, 38, 99, 130, 298
input-output model, 160
inputdlg, 130
Insert Cell Divider, 32
Insert Text Markup, 36
installation, 2
int, 47, 49, 54, 55, 291
int32, 95
integral, 79
    definite, 54
    improper, 55
    indefinite, 54, 236
    multiple, 55
    symbolic, 79
Integrator, 112, 114, 115, 119, 203, 300
interest, 196
interest rate, 138, 260
interrupting calculations, 211
intersection point, 44
inv, 82, 248, 291
inverse, 248
InvertHardcopy, 122
isa, 298
ischar, 95, 298
isempty, 99, 298
isequal, 92, 99, 299
isfinite, 284, 299
ishold, 99, 299
isinf, 299
isletter, 299
isnan, 299
isnumeric, 95, 96, 299

ispc, 101, 299
isreal, 299
isspace, 299
isunix, 101, 299

j, 288
jacobian, 291
Julia set, 204, 285

keyboard, 42, 99, 100, 212, 299
keyboard shortcuts, 27
kinetic energy, 117
Kirchhoff's Current Law, 200
Kirchhoff's Voltage Law, 200

label, 23
Lagrangian, 117
Lagrangian mechanics, 117
LaTeX, 36, 131
Launch Pad, 111
Layout Editor, 123–130
lcm, 197
least common multiple, 197
left-arrow key, 9, 30
left-matrix-divide, 81
legend, 69, 108, 296
lemniscate, 61, 69
length, 50, 90, 291
Leontief model, 160
Leontief, Wassily, 160
level curve, 60, 79, 232
limit, 55
    one-sided, 55
limit, 55, 238, 291
Line, 72
line
    dashed, 70
    dotted, 70
    solid, 70
line, 72
line style, 70
linear algebra, 12, 82, 141, 160
linear economic model, 160
linear inequality, 165
linear programming, 131, 165, 198, 268
LineStyle, 72
LineWidth, 274
linprog, 169
linspace, 178, 184, 291
Linux, 1, 101
Listbox, 126
ln, 15
load, 28, 102, 291
loan, 138
local variable, 34, 50
location of MATLAB, 2
log, 10, 15, 43, 52, 56, 289
logical, 105
logical array, 88
logical expression, 87, 89
logistic constant, 154
logistic growth, 152, 201
loglog, 200, 273, 296
lookfor, 4, 29, 292
loop, 35, 39, 73, 85, 92, 93
loops, avoiding, 90, 91, 197
lowercase, 19, 209, 212
ls, 28, 292

M-book, 35, 40, 103, 108
M-books, enabling, 40
M-file, xiii, 16, 27, 31, 39, 49–51, 68, 85, 94, 99, 100, 108, 120, 123, 126, 127, 206
    function, 31, 33, 49, 51
    problems with, 205
    script, 31, 32, 35, 50
Macintosh, 1
Maple, 58
maple, 58, 292
Maple kernel, 58, 292
Math Function, 300
Math library, 202
Math Operations library, 113, 119
mating, 83
MATLAB Desktop, 2
matrix, 14, 46, 52, 53
matrix multiplication, 14
max, 284, 292
mean, 180, 292
median, 292
memory, managing, 211
menu bar, 2, 4, 27, 66, 126
Menu Editor, 126
menu items, xiv
mesh, 64, 296
meshgrid, 60, 61, 64, 231, 296
mex, 299
mhelp, 58, 292
Microsoft Word, xiii, 106
millionaire, 196
min, 292

minus sign, 109
mismatched delimiters, 207, 211
mistake, 205
mkdir, 29
mldivide, 53
mlint, 212, 299
model window, 111
Monte Carlo simulation, 131, 144, 197, 258
monthly payment, 139
more, 4, 109, 292
more on, 23
mortgage, 131, 138
movie, 81, 84, 192, 295
movie, 77, 192, 244, 255, 296
movie2avi, 77
movieview, 77, 192, 296
multiple curves, 25
multiple integral, 55
multiplication, 8
Mux, 300

NaN, 89, 288
nargin, 95, 299
nargout, 95, 96, 299
Newton, 202
nodesktop, 28
noise, 115
non-singular, 82
Notebook, 41
notebook, 40, 106, 292
num2str, 47, 99, 292
numerical differentiation, 115
numerical integration, 55, 115
numerical ODE solver, 202
numerical output, 10
numerical solution, 175

objective function, 170
ode45, 49, 172, 182, 275, 276, 292
Ohm's Law, 200
ohms, 200
ones, 53, 67, 89, 90, 254, 292
online help, 3, 47, 131
open, 292
open_system, 300
openvar, 28
optimal value, 136
Optimization Toolbox, 169
order of integration, 55
ordinary differential equation, 182
oscillation, 173
otherwise, 91, 299
output, 7, 51, 95, 96
    numerical, 10
    suppressing, 15
    symbolic, 10
    unexpected, 205
    wrong, 205
overloaded, 47

pane, 4
parabolic trajectory, 279
parameter, 59
parametric plot, 59
parametric surface, 64
parentheses, 87, 206, 207
parsing, 95
partial derivative, 54
partial differential equation, 175, 179, 184
patch, 214, 296
path, 29, 33
path, 29, 292
pathtool, 29, 292
pause, 37, 41, 42, 99, 299
PDE Toolbox, 182
pdepe, 184
PDF, 68, 111
peaceful coexistence, 202, 277
pendulum, 117, 131, 172, 202, 277
percent sign, 32, 51
period, 14
periodic cycle, 156, 158
phase portrait, 202
pi, 11, 12, 15, 288
pie, 296
pitch, of sound, 78
pixel, 74
platforms, 1
plot
    contour, 60
    log-log, 296
    parametric, 59
    semi-log, 297
    three-dimensional, 62
    two-dimensional, 59
plot, 24, 25, 39, 44, 59, 62, 70, 73, 76, 206, 214, 296
Plot Browser, 67
plot style, 70
plot3, 62, 70, 297
plotting several functions, 25

plottools, 67
polar, 297
polar coordinates, 51, 64, 96
Polynomial, 300
polynomial, 43
population density, 155
population growth, 150, 201
Popup Menu, 126–128
PostScript, 39
potential energy, 117
PowerPoint, 103
precedence, 207, 211
pretty, 38, 250, 292
pretty print, 38
previously saved M-file, 210
prima donna, 199
prime number, 47
principal, 138
print, 38, 39, 42, 122, 297
Printable Documentation, 68
printing, suppressing, 15
problems, 205
prod, 56, 89, 90, 292
Product, 119, 300
product, element-by-element, 52
Professional Version, 1, 58
profit, 144, 165
prompt, 35
Property Editor, 67, 68, 70–72
Property Inspector, 124–127, 129
publish, 32
publish, xiii, 27, 35, 36, 41, 103, 105, 131, 293
punct, 288
Push Button, 125, 129
pwd, 28, 101, 293

quad, 55
quad8, 293
quadl, 55, 80, 237, 293
quit, 293
quit MATLAB, 6
quiver, 62, 297
quote marks, 207

Radio Button, 126, 130
Ramp, 301
rand, 144, 256, 293
randn, 293
random number, 144
range, 23
rank, 81
rank, 81, 247, 293
real, 289
rectangle, 72
relational operator, 87–89, 92
relop, 87, 88
rescale, 297
reshape, 90
resistor, 200
return, 42, 90, 212, 299
RGB, 72, 74
right-arrow key, 9, 30
right-click, 30
rmpath, 33
roots, 293
round, 293
round-off error, 288
row, 52, 53
row vector, 13, 14, 34, 46

saddle point, 62
saddle surface, 64
save, 28, 102, 293
Save As..., 39
scalar, 14, 18, 46
scale-factor, 62
Scope, 114, 115, 119, 122, 301
    rescaling axes in, 115
script M-file, 31, 32, 34, 35, 50
sec, 289
sech, 289
semicolon, 14, 15, 24, 25, 34, 35, 41, 45, 60, 77, 288
semilogy, 297
sensitivity coefficient, 186
sequence of evaluation, 211
set, 70–73, 128, 129, 297
Set Path, 29
set_param, 121, 300
sign, 86, 289
signal, 112
sim, 121, 300
simlp, 169, 268, 300
simple, 10, 43, 222, 293
simplify, 10, 43, 222, 293
simplot, 122, 300
simulation, 264
    Monte Carlo, 131, 144, 258
Simulink, xi, xiii, 112, 120, 131, 158, 182, 188, 203, 277
simulink, 111, 300

Simulink Toolbox, 169
Simulink Library Browser, 111
Simulink library window, 111
sin, 4, 15, 18, 54, 289
Sine Wave, 301
singular, 247
sinh, 289
Sinks library, 114
size, 67, 89, 90, 293
slash, 52
slash, 30
smiley face, 109
Solaris, 101
solve, 20, 21, 44, 52, 58, 220, 293
solving equations, 20
sound, 78
sound, 77, 78, 293
soundsc, 78
Sources library, 115
space, in argument of command, 208
specgraph, 59
species, 150
spelling error, 205, 209
sphere, 64
spherical coordinates, 64
sqrt, 8, 15, 52, 289
square, 23, 60, 61
sscanf, 101
starting MATLAB, 2
Static Text, 125
steady state, 179, 186
stock market, 196
str2num, 47, 101, 293
strcmp, 92
string, 12, 20, 46, 47, 49, 70, 207, 208
struct, 94, 105
structure array, 94, 105
Student Version, xiv, 1, 292
style, line, 70
subfunction, 94
submatrix, 52
submenu items, xiv
subplot, 67, 297
subs, 11, 50, 230, 293
substitution, 50
subtraction, 8
Sum, 113, 119, 203, 301
sum, 56, 293
summation sign, 70
suppressing output, 34, 35, 41, 45
surf, 64, 178, 297
surface, 72
switch, 85, 91, 208, 299
sym, 12, 47, 48, 53, 80, 223, 293
sym object, 17
symbol, 70
Symbol font, 71
symbolic, 10
symbolic expression, 11, 22, 46, 48, 49
Symbolic Math Toolbox, xiv, 2, 10, 11, 43, 54, 58, 79, 93, 195
symbolic output, 10
symbolic variable, 18
syms, 10, 12, 22, 48, 294
symsum, 56, 58, 93, 294
syntax error, 205, 207, 209

Tag, 127
tan, 15, 289
tanh, 289
taylor, 57, 242, 294
Taylor polynomial, 57, 80
TEX, 70
text, 69, 297
thermal conductivity, 176, 180
tick mark, 70
tight, 23
time-lapse photo, 195
title, 23, 25, 39, 61, 68, 69, 108, 297
title bar, 2
To Workspace, 301
Toggle Button, 125, 130
tool bar, 2, 4, 27, 31, 66
toolboxes, 2, 5, 29, 131
traffic flow, 131, 185
transpose, 13, 15
transpose, 15, 294
Trigonometric Function, 119, 202, 301
type, 49, 85, 94, 101, 105, 266, 294

uicontextmenu, 123
uicontrol, 123
uiimport, 102
uimenu, 123
uint32, 95
underscore, 19
Unit Delay, 301
UNIX, 1, 101, 299
unix, 101, 102, 299
up-arrow key, 9, 30
uppercase, 19, 209, 212

user-defined function, 16
User-Defined Functions library, 119

valid name, 19
varargin, 95, 96, 208, 299
varargout, 95, 97, 208, 300
variable, 8, 10, 19
    symbolic, 10
variable-precision arithmetic, 11, 12
variables
    clearing, 205
    default, 57
vector, 13, 16, 21, 24, 52
    column, 14, 46
    element of, 52
    row, 14, 46
vector field, 62
vectorize, 47, 294
vectorized, 16
Verhulst, 154
versions, 1
vertex, 169
vertical range, 23
view, 66, 297
viewpoint, 66
voltage, 200
volts, 200
vpa, 12, 294

warning, 205, 226
warning, 98, 300
watts, 132
wavread, 78
wavwrite, 78
web, 103, 294
which, 206, 294
while, 92, 93, 300
whos, 17, 28, 46, 88, 294
window, figure, 66
Windows, 1, 101, 299
Windows 2000, 1
Windows Explorer, 106
Windows NT, 1
Windows XP, 1
word processor, 106
Word, Microsoft, 39
Workspace, 28, 46, 50, 51, 102, 211
workspace, 18, 28
Workspace Browser, 2, 18, 27, 28, 46
wrap, 38
wrong output, 205

XData, 72
xlabel, 23, 61, 69, 108, 178, 297
XTick, 70
XTickLabel, 70
XY Graph, 203, 301

YData, 72
ylabel, 23, 61, 69, 108, 178, 297

zeros, 53, 254, 294
zlabel, 69, 178
zoom, 66, 297
zoom in, 66