

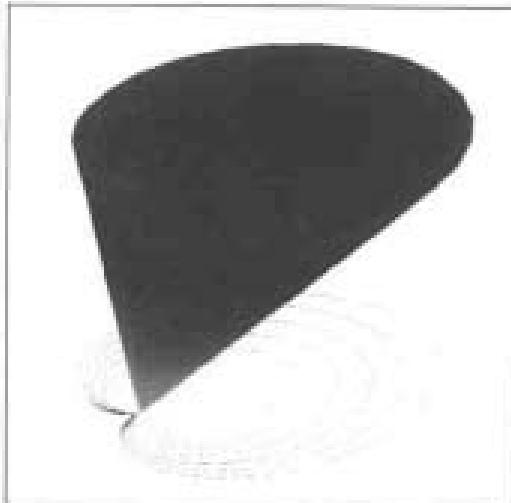


MATLAB 编程

(第二版)

MATLAB Programming for Engineers
(Second Edition)

(英文影印版)



Stephen J. Chapman 著

责任编辑：巴建芳 李 宇 封面设计：黄华斌 陈 敏 责任印制：刘秀平

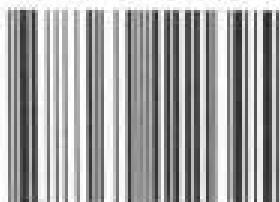
- Covers advanced material such as advanced I/O and the design of Graphical User Interfaces (GUIs) for programs. This material is not typically introduced in a first course but will be useful for students as they advance in the field of engineering.
- Emphasizes the use of functions to logically deconstruct tasks into smaller sub-tasks.
- Offers pedagogical tools such as Good Programming Practice and Programming Pitfalls boxes to help with syntax issues.
- Enhances student learning with effective in-text pedagogy: more than 28 quizzes, 300 programming exercises with selected solutions, and helpful programming hints throughout.
- Teachers can get instructor's manuals for teaching.

国外高校电子信息类优秀教材（英文影印版）

- | | |
|---|---|
| <ul style="list-style-type: none"> ■ 数字基础（第七版） ■ 现代控制系统（第九版） ■ 现代工业电子学（第四版） ■ 射频电路设计理论及应用 ■ 光纤通信技术 ■ 信号与系统基础——应用Web和MATLAB（第二版） ■ 现代VLSI电路设计（第二版） ■ 网络通信技术 ■ 数据与网络通信 ■ 微电子电路分析与设计 ■ 数字系统设计入门教程——集成方法 ■ 应用电磁学基础 ■ 过程控制仪表技术（第六版） ■ 应用数字信号处理进行数字控制 | <ul style="list-style-type: none"> ■ 电机、拖动及电力系统（第五版） ■ 电路（第六版） ■ 电路分析（第二版） ■ 现代VLSI电路设计——芯片系统设计（第三版） ■ 工程电磁学 ■ 数字信号处理引论 ■ 数字信号处理——应用MATLAB ■ 数字信号处理 ■ 自适应控制（第二版） ■ 无线网络原理 ■ 数字与模拟通信系统（第六版） ■ 电子通信技术（第四版） ■ 现代通信系统——应用MATLAB ■ MATLAB编程（第二版） |
|---|---|

限中国大陆地区销售

ISBN 7-03-011139-7



9 787030 111395 >

ISBN 7-03-011139-7

定 价：41.00 元

国外高校电子信息类优秀教材(英文影印版)

MATLAB 编程

(第二版)

MATLAB Programming for Engineers
(Second Edition)

Stephen J. Chapman 著

科学出版社
北京

内 容 简 介

本书为国外高校电子信息类优秀教材(英文影印版)之一。

本书详细讲述了如何用 MATLAB 进行程序设计,如何编写清楚、高效的程序。书中强调了自上而下的程序设计方法、函数的使用、MATLAB 内部工具的使用和数据结构,并指出了一些使用技巧和编程者常犯的错误。

本书可作为工科各专业本科生的教学辅导书,也可作为工程技术人员的参考书。

MATLAB Programming for Engineers, 2nded.

By Stephen J. Chapman

Copyright ©2002

First published by Brooks/Cole, a division of Thomson Learning, United States of America.

All Rights Reserved.

Reprint for the People's Republic of China by Science Press and Thomson Asia Pte Ltd under the authorization of Thomson Learning. No part of this book may be reproduced in any form without the prior written permission of Science Press and Thomson Learning.
此影印版只限在中国大陆地区销售(不包括香港、澳门、台湾地区)。

This edition is only for sale in the People's Republic of China (excluding Hong Kong, Macau SARs and Taiwan).

981-243-385-6

THOMSON  <http://www.thomsonlearning.com>

图字:01-2002-5256

图书在版编目(CIP)数据

MATLAB 编程:第 2 版/(美)查普曼(Chapman, S. J.)等著. —影印本.
—北京:科学出版社,2003

(国外高校电子信息类优秀教材)

ISBN 7-03-011139-7

I . M… II . 查… III . 计算机辅助计算—软件包, MATLAB—程序设计—
高等学校—教材—英文 IV . TN391.75

中国版本图书馆 CIP 数据核字(2003)第 005593 号

责任编辑:巴建芬 李 宇/封面设计:黄华斌 陈 敬/责任印制:刘秀平

科学出版社出版

北京市黄城根北街 16 号

邮政编码:100717

<http://www.sciencep.com>

双青印刷厂 印刷

科学出版社发行 各地新华书店经销

*

2003 年 3 月第 一 版 开本: 787×1092 1/16

2003 年 3 月第一次印刷 印张: 31 1/4

印数: 1—3 000 字数: 711 000

定价: 41.00 元

(如有印装质量问题, 我社负责调换(环伟))

国外高校电子信息类优秀教材(英文影印版)

丛书编委会

(按姓氏笔画排序)

王兆安	西安交通大学	王成华	南京航空航天大学
田 良	东南大学	申功璋	北京航空航天大学
吕志伟	哈尔滨工业大学	吴 刚	中国科学技术大学
吴 澄	清华大学	宋文涛	上海交通大学
张延华	北京工业大学	李哲英	北方交通大学
李瀚荪	北京理工大学	郑大钟	清华大学
姚建铨	天津大学	赵光宙	浙江大学
郭从良	中国科学技术大学	崔一平	东南大学

This book is dedicated to my wife, Rosa, after 25 wonderful years together.

Preface

MATLAB (short for MATrix LABoratory) is a special-purpose computer program optimized to perform engineering and scientific calculations. It started life as a program designed to perform matrix mathematics, but over the years it has grown into a flexible computing system capable of solving essentially any technical problem.

The MATLAB program implements the MATLAB language, and provides a very extensive library of predefined functions to make technical programming tasks easier and more efficient. This extremely wide variety of functions makes it much easier to solve technical problems in MATLAB than in other languages such as Fortran or C. This book introduces the MATLAB language, and shows how to use it to solve typical technical problems.

This book teaches MATLAB as a technical programming language, showing students how to write clean, efficient, and well-documented programs. It makes no pretense at being a complete description of all of MATLAB's hundreds of functions. Instead, it teaches the student how to use MATLAB as a language, and how to locate any desired function with MATLAB's extensive online help facilities.

The first six chapters of the text are designed to serve as the text for an "Introduction to Programming / Problem Solving" course for freshman engineering students. This material should fit comfortably into a 9-week, 3-hour course. The remaining chapters cover advanced topics such as input/output and graphical user interfaces. These chapters may be covered in a longer course, or used as a reference by engineering students or practicing engineers who use MATLAB as a part of their coursework or employment.

Changes in the Second Edition

The second edition of this book is specifically devoted to MATLAB versions 6.0 and 6.1. While the basic MATLAB language has been largely constant since the

release of version 5.0, the integrated tools, the windows, and the help subsystem have changed dramatically. In addition, a completely new paradigm for the design of MATLAB GUIs was introduced in version 6.0. Users working with versions of MATLAB before version 6.0 should be aware that the description of GUI development in Chapter 10 does not apply to them.

The book also covers the minor but important improvements in the language itself, such as the introduction of the `continue` statement.

The Advantages of MATLAB for Technical Programming

MATLAB has many advantages compared to conventional computer languages for technical problem solving. Among them are:

1. Ease of Use

MATLAB is an interpreted language, like many versions of Basic. Like Basic, it is very easy to use. The program can be used as a scratch pad to evaluate expressions typed at the command line, or it can be used to execute large prewritten programs. Programs may be easily written and modified with the built-in integrated development environment, and debugged with the MATLAB debugger. Because the language is so easy to use, it is ideal for educational use, and for the rapid prototyping of new programs.

Many program development tools are provided to make the program easy to use. They include an integrated editor/debugger, online documentation and manuals, a workspace browser, and extensive demos.

2. Platform Independence

MATLAB is supported on many different computer systems, providing a large measure of platform independence. At the time of this writing, the language is supported on Windows 9x/NT/2000 and many different versions of UNIX. Programs written on any platform will run on all of the other platforms, and data files written on any platform may be read transparently on any other platform. As a result, programs written in MATLAB can migrate to new platforms when the needs of the user change.

3. Predefined Functions

MATLAB comes complete with an extensive library of predefined functions that provide tested and prepackaged solutions to many basic technical tasks. For example, suppose that you are writing a program that must calculate the statistics associated with an input data set. In most languages, you would need to write your own subroutines or functions to implement calculations such as the arithmetic mean, standard deviation, median, etc. These and hundreds of other functions are built right into the MATLAB language, making your job much easier.

In addition to the large library of functions built into the basic MATLAB language, there are many special-purpose toolboxes available to

help solve complex problems in specific areas. For example, a user can buy standard toolboxes to solve problems in signal processing, control systems, communications, image processing, and neural networks, among many others.

4. Device-Independent Plotting

Unlike other computer languages, MATLAB has many integral plotting and imaging commands. The plots and images can be displayed on any graphical output device supported by the computer on which MATLAB is running. This capability makes MATLAB an outstanding tool for visualizing technical data.

5. Graphical User Interface

MATLAB includes tools that allow a programmer to interactively construct a graphical user interface (GUI) for his or her program. With this capability, the programmer can design sophisticated data analysis programs that can be operated by relatively inexperienced users.

6. MATLAB Compiler

MATLAB's flexibility and platform independence is achieved by compiling MATLAB programs into a device-independent p-code, and then interpreting the p-code instructions at run-time. This approach is similar to that used by Microsoft's Visual Basic language. Unfortunately, the resulting programs can sometimes execute slowly because the MATLAB code is interpreted rather than compiled. We will point out features that tend to slow program execution when we encounter them.

A separate MATLAB compiler is available. This compiler can compile a MATLAB program into a true executable that runs faster than the interpreted code. It is a great way to convert a prototype MATLAB program into an executable suitable for sale and distribution to users.

Features of This Book

Many features of this book are designed to emphasize the proper way to write reliable MATLAB programs. These features should serve a student well as he or she is first learning MATLAB, and should also be useful to the practitioner on the job. They include:

1. Emphasis on Top-Down Design Methodology

The book introduces a top-down design methodology in Chapter 3, and then uses it consistently throughout the rest of the book. This methodology encourages a student to think about the proper design of a program *before* beginning to code. It emphasizes the importance of clearly defining the problem to be solved and the required inputs and outputs before any other work is begun. Once the problem is properly defined, it teaches the student to employ stepwise refinement to break the task down into successively smaller subtasks, and to implement the subtasks as separate

subroutines or functions. Finally, it teaches the importance of testing at all stages of the process, both unit testing of the component routines and exhaustive testing of the final product.

The formal design process taught by the book may be summarized as follows:

- Clearly state the problem that you are trying to solve.
- Define the inputs required by the program and the outputs to be produced by the program.
- Describe the algorithm that you intend to implement in the program. This step involves top-down design and stepwise decomposition, using pseudocode or flow charts.
- Turn the algorithm into MATLAB statements.
- Test the MATLAB program. This step includes unit testing of specific functions, and also exhaustive testing of the final program with many different data sets.

2. Emphasis on Functions

The book emphasizes the use of functions to logically decompose tasks into smaller subtasks. It teaches the advantages of functions for data hiding. It also emphasizes the importance of unit testing functions before they are combined into the final program. In addition, the book teaches about the common mistakes made with functions, and how to avoid them.

3. Emphasis on MATLAB Tools

The book teaches the proper use of MATLAB's built-in tools to make programming and debugging easier. The tools covered include the Launch Pad, Editor / Debugger, Workspace Browser, Help Browser, and GUI design tools.

4. Good Programming Practice Boxes

These boxes highlight good programming practices when they are introduced for the convenience of the student. In addition, the good programming practices introduced in a chapter are summarized at the end of the chapter. An example Good Programming Practice box is shown below.

Good Programming Practice

Always indent the body of an `if` construct by 2 or more spaces to improve the readability of the code.

5. Programming Pitfalls Boxes

These boxes highlight common errors so that they can be avoided. An example Programming Pitfalls box follows.

Programming Pitfalls

Make sure that your variable names are unique in the first 31 characters. Otherwise, MATLAB will not be able to tell the difference between them.

6. Emphasis on Data Structures

Chapter 7 contains a detailed discussion of MATLAB data structures, including sparse arrays, cell arrays, and structure arrays. The proper use of these data structures is illustrated in the chapters on Handle Graphics and Graphical User Interfaces.

Pedagogical Features

The first six chapters of this book are specifically designed to be used in a freshman “Introduction to Programming / Problem Solving” course. It should be possible to cover this material comfortably in a 9-week, 3-hour course. If there is insufficient time to cover all of the material in a particular Engineering program, Chapter 6 may be deleted, and the remaining material will still teach the fundamentals of programming and using MATLAB to solve problems. This feature should appeal to harassed engineering educators trying to cram ever more material into a finite curriculum.

The remaining chapters cover advanced material that will be useful to the engineer and students as they progress in their careers. This material includes advanced input/output and the design of graphical user interfaces for programs.

The book includes several features designed to aid student comprehension. A total of 15 quizzes appear scattered throughout the chapters, with answers to all questions included in Appendix B. These quizzes can serve as a useful self-test of comprehension. In addition, there are approximately 140 end-of-chapter exercises. Answers to selected exercises are available at the book’s Web site, and of course answers to all exercises are included in the Instructor’s Manual. Good programming practices are highlighted in all chapters with special Good Programming Practice boxes, and common errors are highlighted in Programming Pitfalls boxes. End-of-chapter materials include Summaries of Good Programming Practice and Summaries of MATLAB Commands and Functions.

The book is accompanied by an Instructor’s Manual containing the solutions to all end-of-chapter exercises. The source code for all examples in the book is available from the book’s Web site, and the source code for all solutions in the Instructor’s Manual is available separately to instructors.

A Final Note to the User

No matter how hard I try to proofread a document like this book, it is inevitable that some typographical errors will slip through and appear in print. If you should

spot any such errors, please drop me a note via the publisher, and I will do my best to get them eliminated from subsequent printings and editions. Thank you very much for your help in this matter.

I will maintain a complete list of errata and corrections at the book's Web site, which is <http://info.brookscole.com/chapman>. Please check that site for any updates and/or corrections.

Acknowledgments

I would like to thank Bill Stenquist and the crew at Brooks/Cole for the support they have given me in getting this book to market. It has been gratifying to see the user response to the first edition, which was the result of our joint efforts.

In addition, I would like to thank my wife, Rosa, and our children, Avi, David, Rachel, Aaron, Sarah, Naomi, Shira, and Devorah, for being such delightful people, and the inspiration for my efforts.

Stephen J. Chapman

Contents

| Introduction to MATLAB |

1.1	The Advantages of MATLAB	1
1.2	Disadvantages of MATLAB	3
1.3	The MATLAB Environment	3
1.3.1	The MATLAB Desktop	4
1.3.2	The Command Window	4
1.3.3	The Command History Window	7
1.3.4	The Launch Pad	7
1.3.5	The Edit/Debug Window	8
1.3.6	Figure Windows	9
1.3.7	The MATLAB Workspace	10
1.3.8	The Workspace Browser	11
1.3.9	Getting Help	13
1.3.10	A Few Important Commands	14
1.3.11	The MATLAB Search Path	15
1.4	Using MATLAB as a Scratchpad	16
1.5	Summary	18
1.5.1	MATLAB Summary	19
1.6	Exercises	19

2 MATLAB Basics

21

2.1	Variables and Arrays	21
2.2	Initializing Variables in MATLAB	24
2.2.1	Initializing Variables in Assignment Statements	25
2.2.2	Initializing with Shortcut Expressions	27
2.2.3	Initializing with Built-in Functions	28
2.2.4	Initializing Variables with Keyboard Input	29
2.3	Multidimensional Arrays	31
2.3.1	Storing Multidimensional Arrays in Memory	32
2.3.2	Accessing Multidimensional Arrays with a Single Subscript	32
2.4	Subarrays	34
2.4.1	The <code>end</code> Function	34
2.4.2	Using Subarrays on the Left-Hand Side of an Assignment Statement	35
2.4.3	Assigning a Scalar to a Subarray	36
2.5	Special Values	37
2.6	Displaying Output Data	39
2.6.1	Changing the Default Format	39
2.6.2	The <code>disp</code> Function	40
2.6.3	Formatted Output with the <code>fprintf</code> Function	40
2.7	Data Files	42
2.8	Scalar and Array Operations	44
2.8.1	Scalar Operations	45
2.8.2	Array and Matrix Operations	45
2.9	Hierarchy of Operations	48
2.10	Built-in MATLAB Functions	51
2.10.1	Optional Results	51
2.10.2	Using MATLAB Functions with Array Inputs	52
2.10.3	Common MATLAB Functions	52
2.11	Introduction to Plotting	52
2.11.1	Using Simple <i>xy</i> Plots	54
2.11.2	Printing a Plot	55
2.11.3	Multiple Plots	56
2.11.4	Line Color, Line Style, Marker Style, and Legends	56
2.11.5	Logarithmic Scales	58
2.12	Examples	59
2.13	Debugging MATLAB Programs	67
2.14	Summary	69
2.14.1	Summary of Good Programming Practice	69
2.14.2	MATLAB Summary	70
2.15	Exercises	73

3 Branching Statements and Program Design 81

3.1 Introduction to Top-Down Design Techniques	81
3.2 Use of Pseudocode	86
3.3 Relational and Logical Operators	87
3.3.1 Relational Operators	87
3.3.2 A Caution about the == and ~= Operators	89
3.3.3 Logic Operators	90
3.3.4 Logical Functions	92
3.4 Branches	94
3.4.1 The if Construct	94
3.4.2 Examples Using if Constructs	96
3.4.3 Notes Concerning the Use of if Constructs	102
3.4.4 The switch Construct	104
3.4.5 The try/catch Construct	106
3.5 Additional Plotting Features	108
3.5.1 Controlling x- and y-axis Plotting Limits	108
3.5.2 Plotting Multiple Plots on the Same Axes	111
3.5.3 Creating Multiple Figures	111
3.5.4 Subplots	112
3.5.5 Enhanced Control of Plotted Lines	114
3.5.6 Enhanced Control of Text Strings	115
3.5.7 Polar Plots	115
3.5.8 Annotating and Saving Plots	123
3.6 More on Debugging MATLAB Programs	125
3.7 Summary	128
3.7.1 Summary of Good Programming Practice	129
3.7.2 MATLAB Summary	129
3.8 Exercises	130

4 Loops 137

4.1 The while Loop	137
4.2 The for Loop	143
4.2.1 Details of Operation	150
4.2.2 The break and continue Statements	154
4.2.3 Nesting Loops	156
4.3 Logical Arrays and Vectorization	157
4.3.1 The Significance of Logical Arrays	158
4.3.2 Creating the Equivalent of if/else Constructs with Logical Arrays	161

4.4 Additional Examples	163
4.5 Summary	178
4.5.1 Summary of Good Programming Practice	178
4.5.2 MATLAB Summary	179
4.6 Exercises	179

5 User-Defined Functions 187

5.1 Introduction to MATLAB Functions	189
5.2 Variable Passing in MATLAB: The Pass-By-Value Scheme	194
5.3 Optional Arguments	204
5.4 Sharing Data Using Global Memory	209
5.5 Preserving Data Between Calls to a Function	217
5.6 Function Functions	222
5.7 Subfunctions and Private Functions	225
5.8 Summary	227
5.8.1 Summary of Good Programming Practice	228
5.8.2 MATLAB Summary	229
5.9 Exercises	229

6 Complex Data, Character Data, and Additional Plot Types 241

6.1 Complex Data	241
6.1.1 Complex Variables	243
6.1.2 Using Complex Numbers with Relational Operators	244
6.1.3 Complex Functions	244
6.1.4 Plotting Complex Data	248
6.2 String Functions	252
6.2.1 String Conversion Functions	252
6.2.2 Creating Two-Dimensional Character Arrays	252
6.2.3 Concatenating Strings	253
6.2.4 Comparing Strings	254
6.2.5 Searching/Replacing Characters within a String	256
6.2.6 Uppercase and Lowercase Conversion	257
6.2.7 Numeric-to-String Conversions	258
6.2.8 String-to-Numeric Conversions	259
6.2.9 Summary	260

6.3 Multidimensional Arrays	266
6.4 Additional Two-Dimensional Plots	268
6.4.1 Additional Types of Two-Dimensional Plots	268
6.4.2 Plotting Functions	273
6.4.3 Histograms	274
6.5 Three-Dimensional Plots	276
6.5.1 Three-Dimensional Line Plots	276
6.5.2 Three-Dimensional Surface, Mesh, and Contour Plots	278
6.6 Summary	281
6.6.1 Summary of Good Programming Practice	281
6.6.2 MATLAB Summary	282
6.7 Exercises	283

7 Sparse Arrays, Cell Arrays, and Structures 287

7.1 Sparse Arrays	287
7.1.1 The <code>sparse</code> Data Type	289
7.2 Cell Arrays	294
7.2.1 Creating Cell Arrays	296
7.2.2 Using Braces {} as Cell Constructors	297
7.2.3 Viewing the Contents of Cell Arrays	298
7.2.4 Extending Cell Arrays	298
7.2.5 Deleting Cells in Arrays	300
7.2.6 Using Data in Cell Arrays	300
7.2.7 Cell Arrays of Strings	301
7.2.8 The Significance of Cell Arrays	302
7.2.9 Summary of <code>cell</code> Array Functions	305
7.3 Structure Arrays	306
7.3.1 Creating Structures	306
7.3.2 Adding Fields to Structures	308
7.3.3 Removing Fields from Structures	309
7.3.4 Using Data in Structure Arrays	310
7.3.5 The <code>getfield</code> and <code>setfield</code> Functions	311
7.3.6 Using the <code>size</code> Function with Structure Arrays	312
7.3.7 Nesting Structure Arrays	312
7.3.8 Summary of structure Functions	313
7.4 Summary	314
7.4.1 Summary of Good Programming Practice	315
7.4.2 MATLAB Summary	315
7.5 Exercises	316

8 Input/Output Functions**319**

8.1	The <code>textread</code> Function	319
8.2	More about the <code>load</code> and <code>save</code> Commands	321
8.3	An Introduction to MATLAB File Processing	323
8.4	File Opening and Closing	325
8.4.1	The <code>fopen</code> Function	325
8.4.2	The <code>fclose</code> Function	328
8.5	Binary I/O Functions	328
8.5.1	The <code>fwrite</code> Function	328
8.5.2	The <code>fread</code> Function	329
8.6	Formatted I/O Functions	332
8.6.1	The <code>fprint</code> Function	332
8.6.2	Understanding Format Conversion Specifiers	334
8.6.3	How Format Strings Are Used	336
8.6.4	The <code>fscanf</code> Function	339
8.6.5	The <code>fgetl</code> Function	341
8.6.6	The <code>fgets</code> Function	341
8.7	Comparing Formatted and Binary I/O Functions	342
8.8	File Positioning and Status Functions	347
8.8.1	The <code>exist</code> Function	347
8.8.2	The <code>ferror</code> Function	349
8.8.3	The <code>feof</code> Function	350
8.8.4	The <code>ftell</code> Function	350
8.8.5	The <code>frewind</code> Function	350
8.8.6	The <code>fseek</code> Function	350
8.9	Function <code>uiimport</code>	356
8.10	Summary	358
8.10.1	Summary of Good Programming Practice	359
8.10.2	MATLAB Summary	359
8.11	Exercises	360

9 Handle Graphics**363**

9.1	The MATLAB Graphics System	363
9.2	Object Handles	365
9.3	Examining and Changing Object Properties	365
9.3.1	Changing Object Properties at Creation Time	365
9.3.2	Changing Object Properties after Creation Time	366
9.4	Using <code>set</code> to List Possible Property Values	372
9.5	User-Defined Date	374
9.6	Finding Objects	375

9.7	Selecting Objects with the Mouse	377
9.8	Position and Units	380
9.7.1	Positions of figure Objects	380
9.7.2	Positions of axes and uicontrol Objects	381
9.7.3	Positions of text Objects	381
9.9	Printer Positions	384
9.10	Default and Factory Properties	385
9.11	Graphics Object Properties	387
9.12	Summary	387
9.12.1	Summary of Good Programming Practice	388
9.12.2	MATLAB Summary	388
9.13	Exercises	387

10 Graphical User Interfaces

391

10.1	How a Graphical User Interface Works	391
10.2	Creating and Displaying a Graphical User Interface	392
10.2.1	A Look Under the Hood	402
10.2.2	The Structure of a Callback Subfunction	404
10.2.3	Adding Application Data to a Figure	405
10.2.4	A Few Useful Functions	406
10.3	Object Properties	406
10.4	Graphical User Interface Components	407
10.4.1	Text Fields	409
10.4.2	Edit Boxes	409
10.4.3	Frames	409
10.4.4	Pushbuttons	409
10.4.5	Toggle Buttons	411
10.4.6	Checkboxes and Radio Buttons	411
10.4.7	Popup Menus	414
10.4.8	List Boxes	414
10.4.9	Sliders	417
10.5	Dialog Boxes	422
10.5.1	Error and Warning Dialog Boxes	423
10.5.2	Input Dialog Boxes	423
10.5.3	The <code>uigetfile</code> and <code>uisetfile</code> Dialog Boxes	424
10.6	Menus	425
10.6.1	Suppressing the Default Menu	428
10.6.2	Creating Your Own Menus	427
10.6.3	Accelerator Keys and Keyboard Mnemonics	430
10.6.4	Creating Context Menus	430

10.7	Tips for Creating Efficient GUIs	436
10.7.1	Tool Tips	436
10.7.2	Pcode	437
10.7.3	Additional Enhancements	439
10.8	Summary	443
10.8.1	Summary of Good Programming Practice	445
10.8.2	MATLAB Summary	445
10.9	Exercises	446

Appendix A	ASCII Character Set	449
-------------------	----------------------------	------------

Appendix B	Answers to Quizzes	451
-------------------	---------------------------	------------

Index		465
--------------	--	------------

C H A P T E R

Introduction to MATLAB

MATLAB (short for MATrix LABoratory) is a special-purpose computer program optimized to perform engineering and scientific calculations. It started life as a program designed to perform matrix mathematics, but over the years it has grown into a flexible computing system capable of solving essentially any technical problem.

The MATLAB program implements the MATLAB programming language, and provides an extensive library of predefined functions that make technical programming tasks easier and more efficient. This book introduces the MATLAB language and shows how to use it to solve typical technical problems.

MATLAB is a huge program, with an incredibly rich variety of functions. Even the basic version of MATLAB without any toolkits is much richer than other technical programming languages. There are more than 1000 functions in the basic MATLAB product alone, and the toolkits extend this capability with many more functions in various specialties. This book makes no attempt to introduce the user to all of MATLAB's functions. Instead, a user learns the basics of how to write, debug, and optimize good MATLAB programs, plus a subset of the most important functions. Just as importantly, the programmer learns how to use MATLAB's own tools to locate the correct function for a specific purpose from the enormous choice available.

I.1 The Advantages of MATLAB

MATLAB has many advantages compared with conventional computer languages for technical problem solving. Among them are the following:

1. Ease of Use

MATLAB is an interpreted language, like many versions of Basic. Like Basic, it is very easy to use. The program can be used as a scratch pad to evaluate expressions typed at the command line, or it can be used to execute large prewritten programs. Programs may be easily written and modified with the built-in integrated development environment and debugged with the MATLAB debugger. Because the language is so easy to use, it is ideal for the rapid prototyping of new programs.

Many program development tools are provided to make the program easy to use. They include an integrated editor/debugger, online documentation and manuals, a workspace browser, and extensive demos.

2. Platform Independence

MATLAB is supported on many different computer systems, providing a large measure of platform independence. At the time of this writing, the language is supported on Windows 95/98/ME/NT/2000 and many different versions of UNIX. (Older versions of MATLAB were available for the Macintosh and VAX computers, and they are still in use in many places.) Programs written on any platform will run on all the other platforms, and data files written on any platform may be read transparently on any other platform. As a result, programs written in MATLAB can migrate to new platforms when the user's needs change.

3. Predefined Functions

MATLAB comes complete with an extensive library of predefined functions that provide tested and prepackaged solutions to many basic technical tasks. For example, suppose that you are writing a program that must calculate the statistics associated with an input data set. In most languages, you would need to write your own subroutines or functions to implement calculations such as the arithmetic mean, standard deviation, median, and so forth. These and hundreds of other functions are built right into the MATLAB language, making your job much easier.

In addition to the large library of functions built into the basic MATLAB language, many special-purpose toolboxes are available to help solve complex problems in specific areas. For example, a user can buy standard toolboxes to solve problems in Signal Processing, Control Systems, Communications, Image Processing, and Neural Networks, among many others. There is also an extensive collection of free user-contributed MATLAB programs that are shared through the MATLAB Web site.

4. Device-Independent Plotting

Unlike most other computer languages, MATLAB has many integral plotting and imaging commands. The plots and images can be displayed on any graphical output device supported by the computer on which MATLAB is running. This capability makes MATLAB an outstanding tool for visualizing technical data.

5. Graphical User Interface

MATLAB includes tools that allow a programmer to interactively construct a graphical user interface (GUI) for his or her program. With this capability, the programmer can design sophisticated data analysis programs that can be operated by relatively inexperienced users.

6. MATLAB Compiler

MATLAB's flexibility and platform independence is achieved by compiling MATLAB programs into a device-independent p-code, and then interpreting the p-code instructions at run time. This approach is similar to that used by Microsoft's Visual Basic language. Unfortunately, the resulting programs can sometimes execute slowly because the MATLAB code is interpreted rather than compiled. We will point out features that tend to slow program execution when we encounter them.

A separate MATLAB compiler is available. This compiler can compile a MATLAB program into a true executable program that runs faster than the interpreted code. It is a great way to convert a prototype MATLAB program into an executable program suitable for sale and distribution to users.

1.2 Disadvantages of MATLAB

MATLAB has two principal disadvantages. The first is that it is an interpreted language, and therefore can execute more slowly than compiled languages. This problem can be mitigated by properly structuring the MATLAB program and by the use of the MATLAB compiler to compile the final MATLAB program before distribution and general use.

The second disadvantage is cost: A full copy of MATLAB is 5 to 10 times more expensive than a conventional C or Fortran compiler. This relatively high cost is more than offset by the reduced time required for an engineer or scientist to create a working program, so MATLAB is cost-effective for businesses. However, it is too expensive for most individuals to consider purchasing. Fortunately, there is also an inexpensive Student Edition of MATLAB, which is a great tool for students wanting to learn the language. The Student Edition of MATLAB is essentially identical to the full edition.

1.3 The MATLAB Environment

The fundamental unit of data in any MATLAB program is the **array**. An array is a collection of data values organized into rows and columns, and known by a single name. Individual data values within an array may be accessed by including the name of the array followed by subscripts in parentheses that identify the row and column of the particular value. Even scalars are treated as arrays by MATLAB—

they are simply arrays with only one row and one column. We will learn how to create and manipulate MATLAB arrays in Chapter 2.

When MATLAB executes, it can display several types of windows that accept commands or display information. The three most important types of windows are Command Windows, where commands may be entered; Figure Windows, which display plots and graphs; and Edit/Debug Windows, which permit a user to create and modify MATLAB programs. We will see examples of all three types of windows in this section.

In addition, MATLAB can display other windows that provide help and that allow the user to examine the values of variables defined in memory. We will examine some of these additional windows here and examine the others when we discuss how to debug MATLAB programs.

1.3.1 The MATLAB Desktop

When you start MATLAB Version 6, a special window called the MATLAB desktop appears. The default configuration of the MATLAB desktop is shown in Figure 1.1. It integrates many tools for managing files, variables, and applications within the MATLAB environment.

The major tools within or accessible from the MATLAB desktop are the following:

- The Command Window
- The Command History Window
- Launch Pad
- The Edit/Debug Window
- Figure Windows
- Workspace Browser and Array Editor
- Help Browser
- Current Directory Browser

We will discuss the functions of these tools in later sections of this chapter.

1.3.2 The Command Window

The right hand side of the MATLAB desktop contains the **Command Window**. A user can enter interactive commands at the command prompt (`>>`) in the Command Window, and they will be executed on the spot.

As an example of a simple interactive calculation, suppose that you want to calculate the area of a circle with a radius of 2.5 m. This can be done in the MATLAB command window by typing:

```
>> area = pi * 2.5^2
area =
19.6350
```

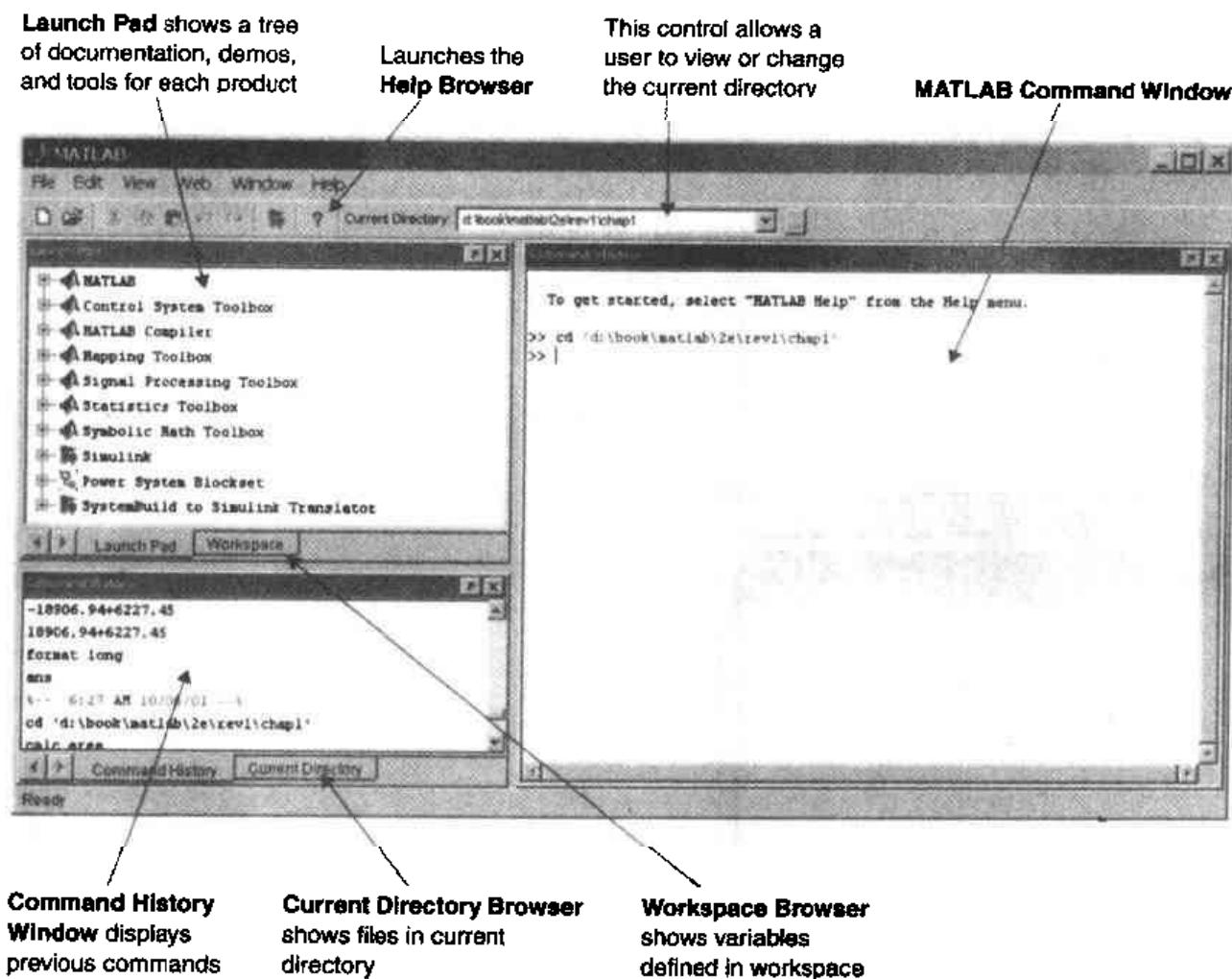


Figure 1.1 The MATLAB desktop. The exact appearance of the window may differ slightly on different types of computers.

MATLAB calculates the answer as soon as the Enter key is pressed and stores the answer in a variable (really a 1×1 array) called *area*. The contents of the variable are displayed in the Command Window as shown in Figure 1.2, and the variable can be used in further calculations. (Note that π is predefined in MATLAB, so we can just use *pi* without first declaring it to be 3.141592 . . .).

If a statement is too long to type on a single line, it may be continued on successive lines by typing an ellipsis (...) at the end of the first line, and then

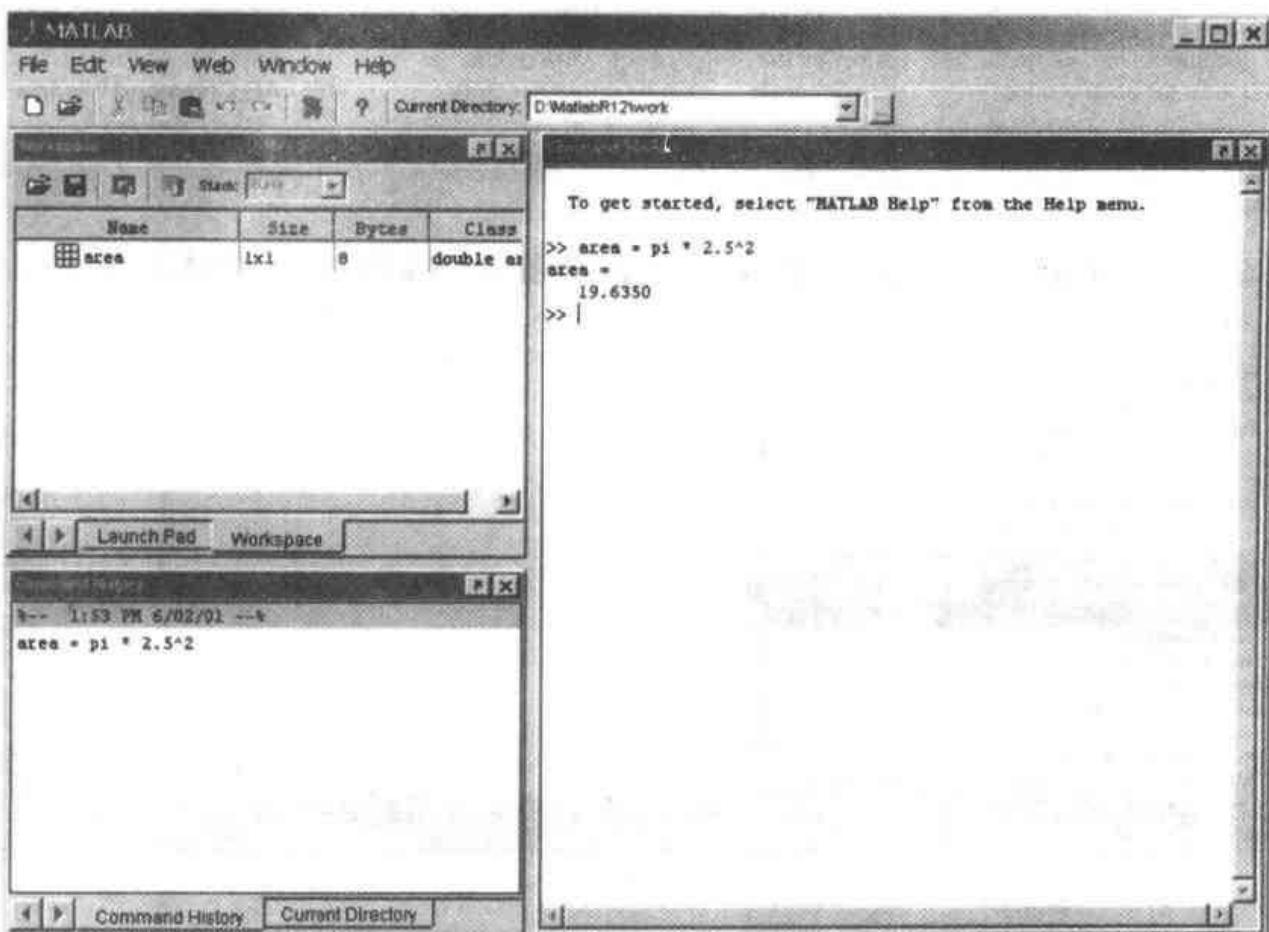


Figure 1.2 The Command Window is the rightmost part of the desktop. Users enter commands and see responses here.

continuing on the next line. For example, the following two statements are identical.

```
x1 = 1 + 1/2 + 1/3 + 1/4 + 1/5 + 1/6;
```

and

```
x1 = 1 + 1/2 + 1/3 + 1/4 ...
+ 1/5 + 1/6;
```

Instead of typing commands directly in the Command Window, a series of commands may be placed into a file, and the entire file may be executed by typing its name in the Command Window. Such files are called **script files**. Script files (and functions, which we will see later) are also known as **M-files** because they have a file extension of ".m".

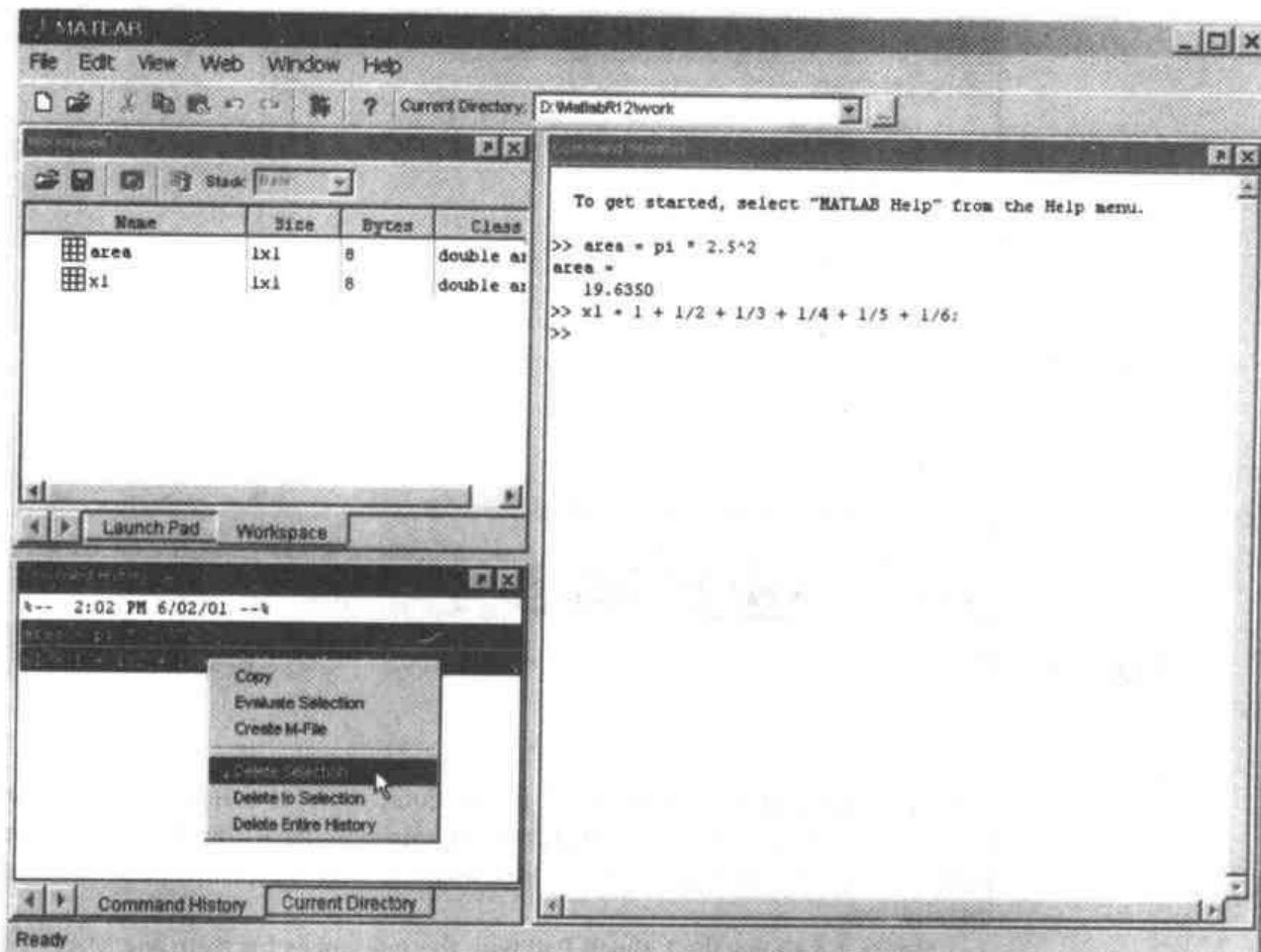


Figure 1.3 The Command History Window, showing two commands being deleted.

1.3.3 The Command History Window

The Command History Window displays a list of the commands that a user has entered in the Command Window. The list of previous commands can extend back to previous executions of the program. Commands remain in the list until they are deleted. To re-execute any command, simply double-click it with the left mouse button. To delete one or more commands from the Command History Window, select the commands and right-click them with the mouse. A popup menu will be displayed that allows the user to delete the items (see Figure 1.3).

1.3.4 The Launch Pad

The Launch Pad is a special tool that collects references to the documentation, demos, and related tools for MATLAB itself and for each toolkit that you

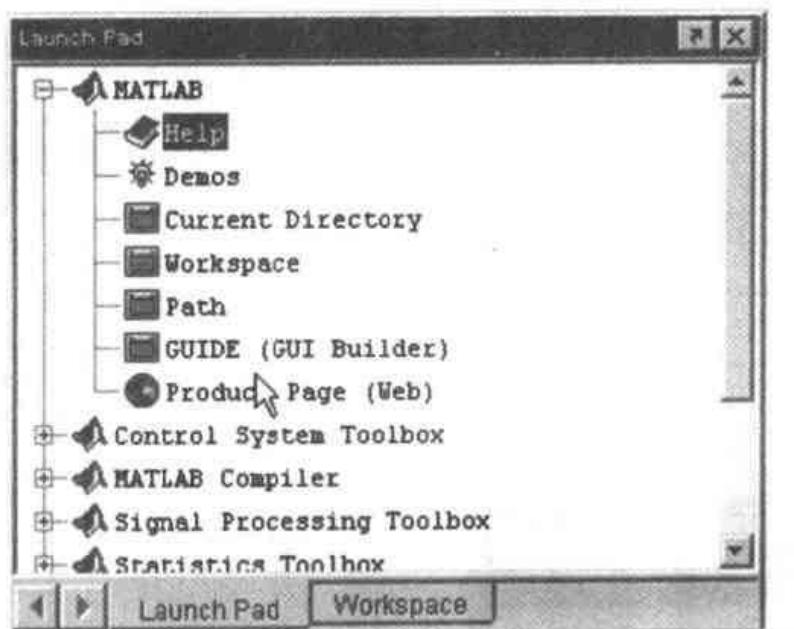


Figure 1.4 The Launch Pad.

own. The information is organized by product, with all references listed under each product or toolbox. Different people will have purchased different products, so the exact contents of this window will vary from installation to installation.

Figure 1.4 shows the Launch Pad with the references for the basic MATLAB product expanded. By double-clicking on one of these items, you can get help with MATLAB, run the MATLAB demos, access the standard tools supplied with the program, or go to the MATLAB Internet Web site.

1.3.5 The Edit/Debug Window

An **Edit/Debug Window** is used to create new M-files, or to modify existing ones. An Edit/Debug Window is created automatically when you create a new M-file or open an existing one. You can create a new M-file with the “File/New/M-file” selection from the desktop menu, or by clicking the Toolbar icon. You can open an existing M-file with the “File/Open” selection from the desktop menu or by clicking the Toolbar icon.

The Edit/Debug Window is essentially a programming text editor, with the MATLAB language features highlighted in different colors. Comments in an M-file file appear in green, variables and numbers appear in black, character strings appear in red, and language keywords appear in blue. An example Edit Window containing a simple M-file is shown in Figure 1.5. This file calculates the area of a circle given its radius and displays the result.

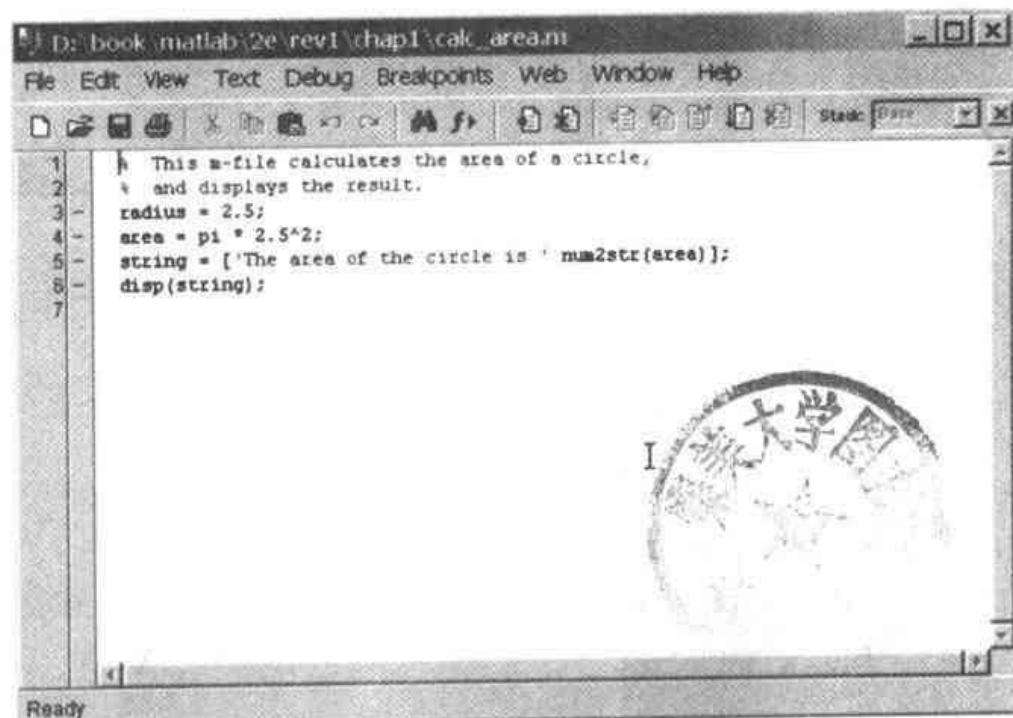


Figure 1.5 The Edit/Debug Window, showing file `calc_area.m`.

After an M-file is saved, it may be executed by typing its name in the Command Window. For the M-file in Figure 1.5, the results are

```

>> calc_area
The area of the circle is 19.635

```

The Edit/Debug Window also doubles as a debugger, as we shall see in Chapter 2.

1.3.6 Figure Windows

A **Figure Window** is used to display MATLAB graphics. A figure can be a two- or three-dimensional plot of data, an image, or a GUI. A simple script file that calculates and plots the function $\sin x$ is shown here:

```

% sin_x.m: This M-file calculates and plots the
% function sin(x) for 0 <= x <= 6.
x = 0:0.1:6;
y = sin(x);
plot(x,y);

```

If this file is saved under the name `sin_x.m`, then a user can execute the file by typing “`sin_x`” in the Command Window. When this script file is executed,

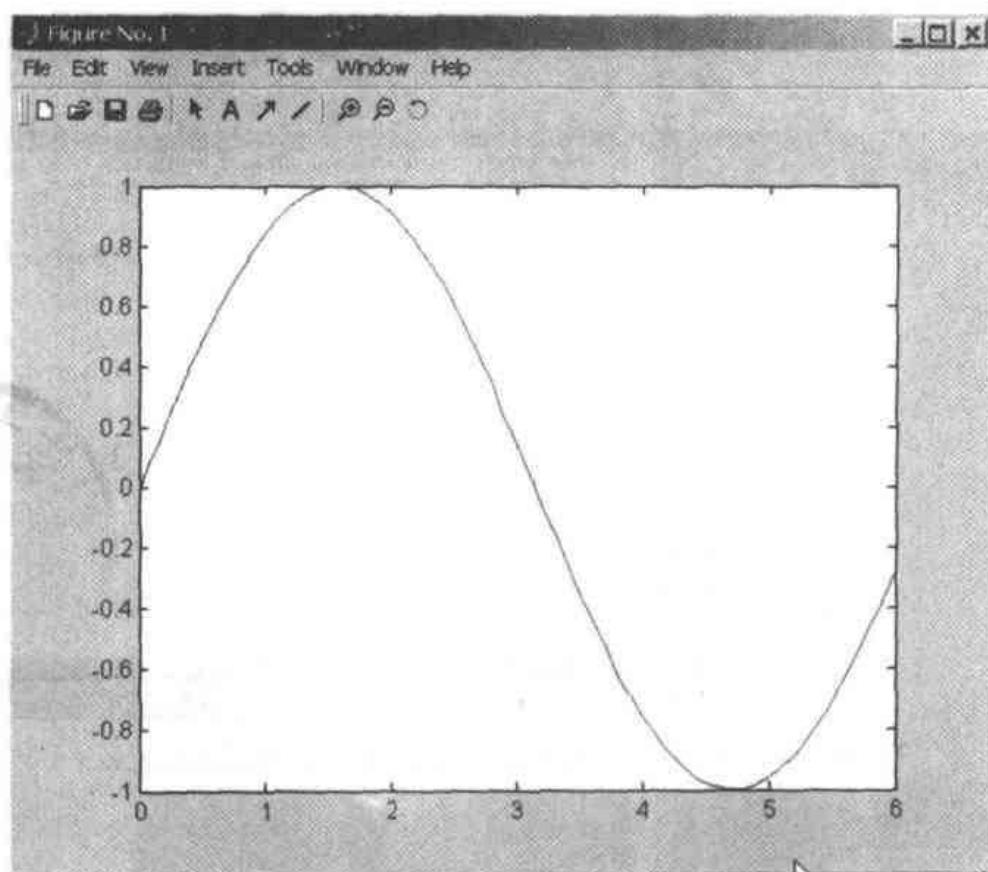


Figure 1.6 MATLAB plot of $\sin x$ versus x .

MATLAB opens a Figure Window and plots the function $\sin x$ in it. The resulting plot is shown in Figure 1.6.

1.3.7 The MATLAB Workspace

A statement like

```
z = 10;
```

creates a variable named *z*, stores the value 10 in it, and saves it in a part of computer memory known as the **workspace**. A workspace is the collection of all the variables and arrays that can be used by MATLAB when a particular command, M-file, or function is executing. All commands executed in the Command Window (and all script files executed from the Command Window) share a common workspace, so they can all share variables. As we will see later, MATLAB functions differ from script files in that each function has its own separate workspace.

A list of the variables and arrays in the current workspace can be generated with the `whos` command. For example, after M-files `calc_area` and `sin_x` are executed, the workspace contains the following variables:

```
>> whos
  Name      Size      Bytes  Class
  area      1x1        8  double array
  radius    1x1        8  double array
  string    1x32       64  char array
  x         1x61      488  double array
  y         1x61      488  double array

Grand total is 155 elements using 1056 bytes
```

The script file `calc_area` created variables `area`, `radius`, and `string`, and the script file `sin_x` created variables `x` and `y`. Note that all of the variables are in the same workspace, so if two script files are executed in succession, the second script file can use variables created by the first script file.

The contents of any variable or array may be determined by typing the appropriate name in the Command Window. For example, the contents of `string` can be found as follows:

```
>> string
string =
The area of the circle is 19.635
```

A variable may be deleted from the workspace with the `clear` command. The `clear` command takes the form

```
clear var1 var2 ...
```

where `var1` and `var2` are the names of the variables to be deleted. The command `clear variables` or simply `clear` deletes all variables from the current workspace.

1.3.8 The Workspace Browser

The contents of the current workspace can also be examined with the GUI-based Workspace Browser. The Workspace Browser appears by default in the upper left-hand corner of the desktop. It provides a graphic display of the same information as the `whos` command, and the display is dynamically updated whenever the contents of the workspace change. The Workspace Browser also allows a user to change the contents of any variable in the workspace.

A typical Workspace Browser Window is shown in Figure 1.7. As you can see, it displays the same information as the `whos` command. Double-clicking on any variable in the window will bring up the Array Editor (Figure 1.8), which allows the user to modify the information stored in the variable.

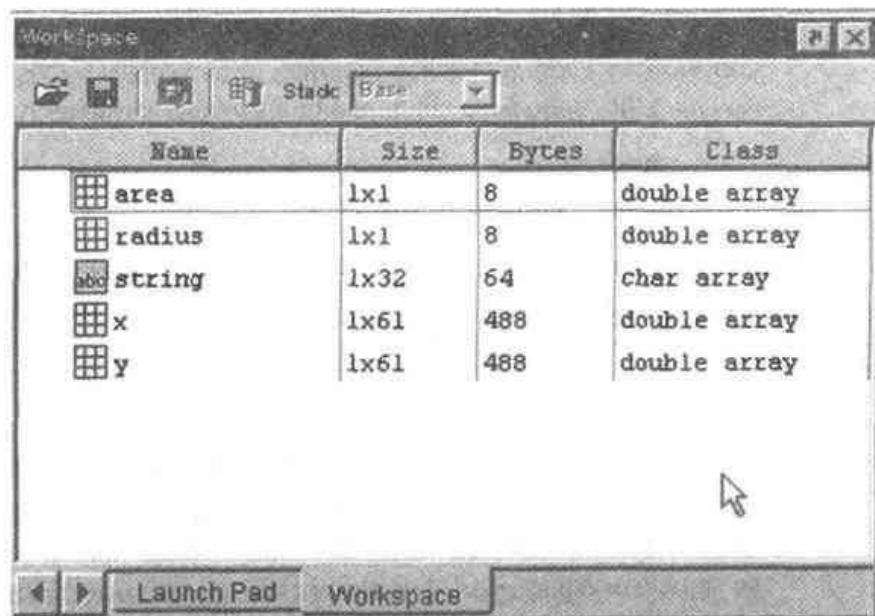


Figure 1.7 The Workspace Browser.

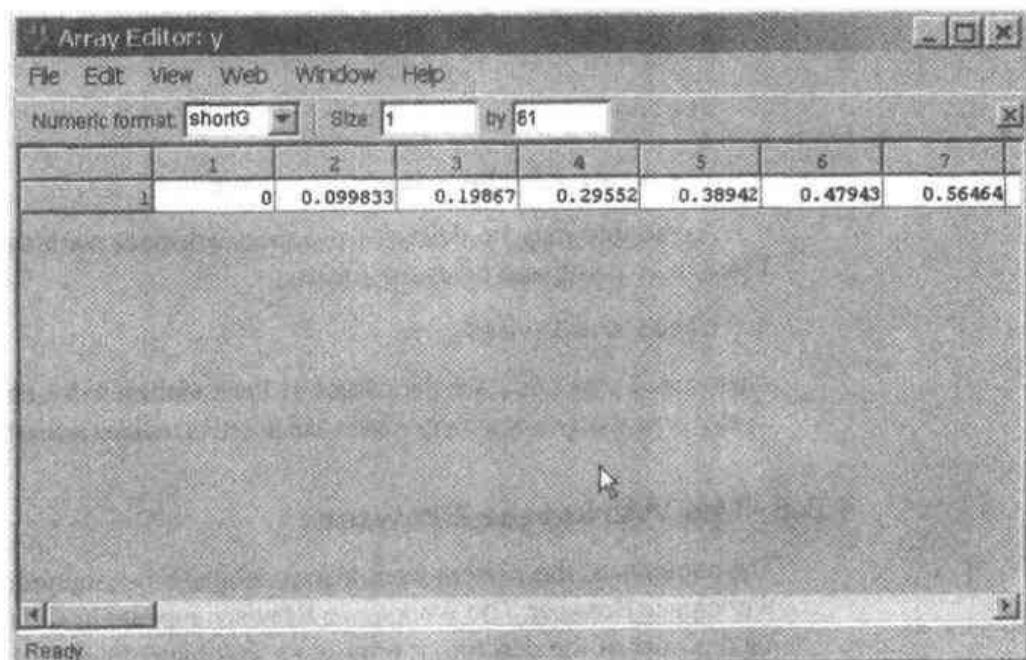


Figure 1.8 The Array Editor is invoked by double-clicking a variable in the Workspace Browser. The Array Editor allows users to change the values contained in a variable or array.

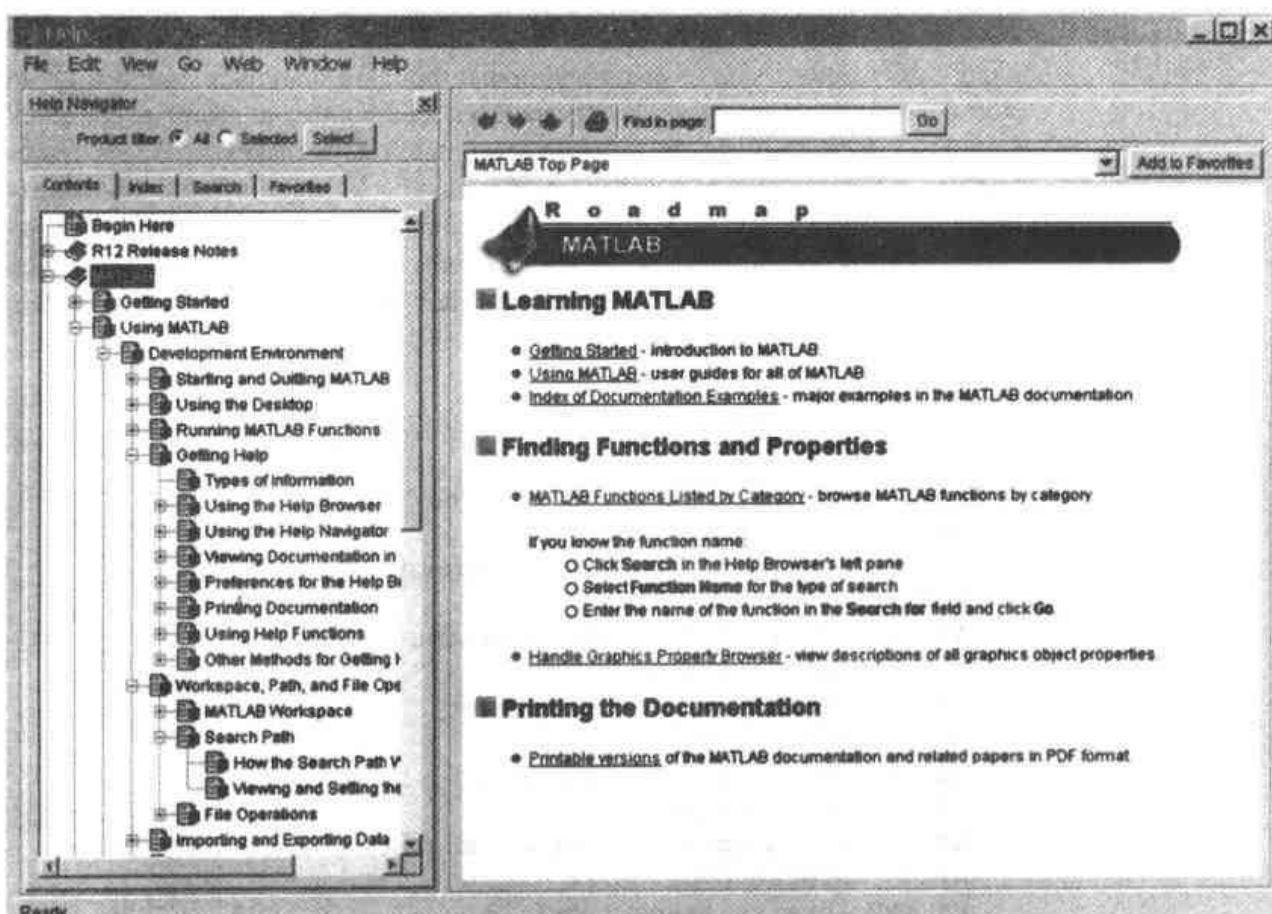


Figure 1.9 The Help Browser.

One or more variables may be deleted from the workspace by selecting them with the mouse and pressing the Delete key, or by right-clicking with the mouse and selecting the delete option.

1.3.9 Getting Help

You can get help in MATLAB in three ways. The preferred method is to use the Help Browser. You can start the Help Browser by selecting the Help icon from the desktop toolbar, or by typing `helpdesk` or `helpwin` in the Command Window. You can get help by browsing the MATLAB documentation, or you can search for the details of a particular command. The Help Browser is shown in Figure 1.9.

There are also two command-line oriented ways to get help. The first way is to type `help` or `help` followed by a function name in the Command Window. If you just type `help`, MATLAB will display a list of possible help topics in the Command Window. If a specific function or a toolbox name is included, `help` will be provided for that particular function or toolbox.

The second way to get help is the `lookfor` command. The `lookfor` command differs from the `help` command in that the `help` command searches for an exact function name match, whereas the `lookfor` command searches the quick summary information in each function for a match. This makes `lookfor` slower than `help`, but it improves the chances of getting back useful information. For example, suppose that you were looking for a function to take the inverse of a matrix. MATLAB does not have a function named `inverse`, so the command `help inverse` will produce nothing. On the other hand, the command `lookfor inverse` will produce the following results:

```
> lookfor inverse
INVHILB Inverse Hilbert matrix.
ACOS Inverse cosine.
ACOSH Inverse hyperbolic cosine.
ACOT Inverse cotangent.
ACOTH Inverse hyperbolic cotangent.
ACSC Inverse cosecant.
ACSCH Inverse hyperbolic cosecant.
ASEC Inverse secant.
ASECH Inverse hyperbolic secant.
ASIN Inverse sine.
ASINH Inverse hyperbolic sine.
ATAN Inverse tangent.
ATAN2 Four quadrant inverse tangent.
ATANH Inverse hyperbolic tangent.
ERFINV Inverse error function.
INV Matrix inverse.
PINV Pseudoinverse.
IFFT Inverse discrete Fourier transform.
IFFT2 Two-dimensional inverse discrete Fourier transform.
IFFTN N-dimensional inverse discrete Fourier transform.
IPERMUTE Inverse permute array dimensions.
```

From this list, we can see that the function of interest is named `inv`.

1.3.10 A Few Important Commands

If you are new to MATLAB, a few demonstrations may help to give you a feel for its capabilities. To run MATLAB's built-in demonstrations, type `demos` in the Command Window, or select "demos" from the Launch Pad.

The contents of the Command Window can be cleared at any time using the `clc` command, and the contents of the current Figure Window can be cleared at any time using the `clf` command. The variables in the workspace can be cleared with the `clear` command. As we have seen, the contents of the workspace persist between the executions of separate commands and M-files, so it is possible for the results of one problem to have an effect on the next one that you attempt

to solve. To avoid this possibility, it is a good idea to issue the `clear` command at the start of each new independent calculation.

Another important command is the `abort` command. If an M-file appears to be running for too long, it might contain an infinite loop, and it will never terminate. In this case, the user can regain control by typing `control-c` (abbreviated `^c`) in the Command Window. Enter this command by holding down the Control key while typing a “c”. When MATLAB detects a `^c`, it interrupts the running program and returns a command prompt.

The exclamation point (!) is another important special character. Its special purpose is to send a command to the computer’s operating system. Any characters after the exclamation point will be sent to the operating system and executed as though they had been typed at the operating system’s command prompt. This feature lets you embed operating system commands directly into MATLAB programs.

Finally, you can keep track of everything done during a MATLAB session using the `diary` command. The form of this command is

```
diary filename
```

After this command is typed, a copy of all input and most output typed in the Command Window is echoed in the diary file. This is a great tool for recreating events when something goes wrong during a MATLAB session. The command `diary off` suspends input into the diary file, and the command `diary on` resumes input again.

1.3.11 The MATLAB Search Path

MATLAB uses a search path to find M-files. MATLAB’s M-files are organized in directories on your file system. Many of these directories of M-files are provided along with MATLAB, and users may add others. If a user enters a name at the MATLAB prompt, the MATLAB interpreter attempts to find the name as follows:

1. It looks for the name as a variable. If it is a variable, MATLAB displays the current contents of the variable.
2. It checks to see if the name is a built-in function or command. If it is, MATLAB executes that function or command.
3. It checks to see if the name is an M-file in the current directory. If it is, MATLAB executes that function or command.
4. It checks to see if the name is an M-file in any directory in the search path. If it is, MATLAB executes that function or command.

Note that MATLAB checks for variable names first, so if you define a variable with the same name as a MATLAB function or command, that function or command becomes inaccessible. This is a common mistake made by novice users.

Programming Pitfall

Never use a variable with the same name as a MATLAB function or command. If you do so, that function or command will become inaccessible.

Also, if there is more than one function or command with the same name, the first one found on the search path will be executed, and all the others will be inaccessible. This is a common problem for novice users because they sometimes create M-files with the same names as standard MATLAB functions, making them inaccessible.

Programming Pitfall

Never create an M-file with the same name as a MATLAB function or command.

MATLAB includes a special command (`which`) to help you find out just which version of a file is being executed, and where it is located. This can be useful in finding filename conflicts. The format of this command is `which functionname`, where `functionname` is the name of the function that you are trying to locate. For example, the cross-product function `cross.m` can be located as follows:

```
>> which cross
D:\MatlabR12\toolbox\matlab\specfun\cross.m
```

The MATLAB search path can be examined and modified at any time by selecting the Path Tool from the Launch Pad, or by typing `editpath` in the Command Window. The Path Tool is shown in Figure 1.10. It allows a user to add, delete, or change the order of directories in the path. Other path-related functions include:

- `addpath` Add directory to MATLAB search path.
- `path` Display MATLAB search path.
- `path2rc` Adds current directory to MATLAB search path.
- `rmpath` Remove directory from MATLAB search path.

1.4 Using MATLAB as a Scratchpad

In its simplest form, MATLAB can be used as a scratchpad to perform mathematical calculations. The calculations to be performed are typed directly into the Command Window, using the symbols `+`, `-`, `*`, `/`, and `^` for addition, subtraction, multiplication, division, and exponentiation, respectively. After an expression is typed, the results of

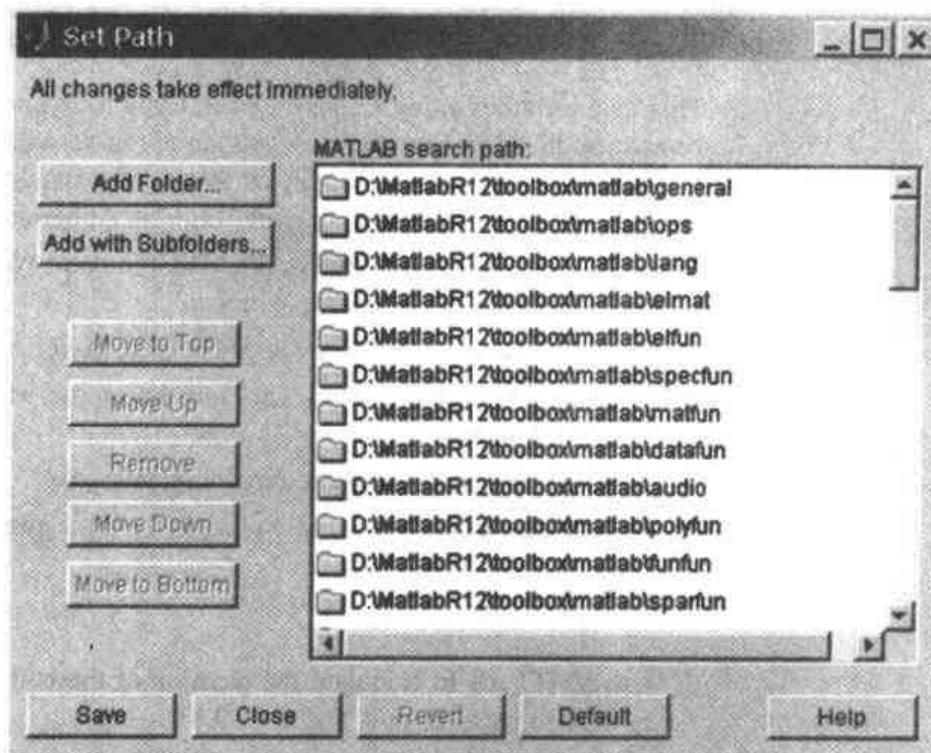


Figure 1.10 The Path Tool.

the expression will be automatically calculated and displayed. For example, suppose we would like to calculate the volume of a cylinder of radius r and length l . The area of the circle at the base of the cylinder is given by the equation

$$A = \pi r^2 \quad (1.1)$$

and the total volume of the cylinder will be

$$V = Al \quad (1.2)$$

If the radius of the cylinder is 0.1 m and the length is 0.5 m, then the volume of the cylinder can be found using the MATLAB statements (user inputs are shown in bold face):

```
>> A = pi * 0.1^2
A =
    0.0314
>> V = A * 0.5
V =
    0.0157
```

Note that `pi` is predefined to be the value 3.141592... Also, note that the value stored in `A` was saved by MATLAB, and reused when we calculated `V`.

Quiz 1.1

This quiz provides a quick check to see if you have understood the concepts introduced in Chapter 1. If you have trouble with the quiz, reread the sections, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in Appendix B.

1. What is the purpose of the MATLAB Command Window? The Edit/Debug Window? The Figure Window?
2. List the different ways that you get help in MATLAB.
3. What is a workspace? How can you determine what is stored in a MATLAB workspace?
4. How can you clear the contents of a workspace?
5. The distance traveled by a ball falling in the air is given by the equation

$$x = x_0 + v_0 t + \frac{1}{2} at^2$$

Use MATLAB to calculate the position of the ball at time $t = 5$ s if $x_0 = 10$ m, $v_0 = 15$ m/s, and $a = -9.81$ m/sec².

6. Suppose that $x = 3$ and $y = 4$. Use MATLAB to evaluate the following expression:

$$\frac{x^2 y^3}{(x - y)^2}$$

The following questions are intended to help you become familiar with MATLAB tools. (If you have a version of MATLAB older than version 6.0, you may some difficulty with some of the questions because some tools were different in older versions of MATLAB.)

7. Execute the M-files `calc_area.m` and `sin_x.m` in the Command Window (these M-files are available from this book's Web site). Then use the Workspace Browser to determine what variables are defined in the current workspace.
8. Use the Array Editor to examine and modify the contents of variable x in the workspace. Then type the command `plot(x,y)` in the Command Window. What happens to the data displayed in the Figure Window?

1.5 Summary

In this chapter, we learned about the basic types of MATLAB windows, the workspace, and how to get online help. The MATLAB desktop appears when the program is started. It integrates many of the MATLAB tools in single location. These tools include the Command Window, the Command History Window, the Launch

Pad, the Workspace Browser, the Array Editor, and the Current Directory viewer. The Command Window is the most important of the windows because all commands are typed and results are displayed in the Command Window.

The Edit/Debug Window is used to create or modify M-files. It displays the contents of the M-file with the contents of the file color-coded according to function: comments, keywords, strings, and so forth.

The Figure Window is used to display graphics.

A MATLAB user can get help by either using the Help Browser or the command-line help functions `help` and `lookfor`. The Help Browser allows full access to the entire MATLAB documentation set. The command-line function `help` displays help about a specific function in the Command Window. Unfortunately, you must know the name of the function to get help about it. The function `lookfor` searches for a given string in the first comment line of every MATLAB function, and displays any matches.

When a user types a command in the Command Window, MATLAB searches for that command in the directories specified in the MATLAB path. MATLAB will execute the *first* M-file in the path that matches the command, and any further M-files with the same name will never be found. The Path Tool can be used to add, delete, or modify directories in the MATLAB path.

1.5.1 MATLAB Summary

The following summary lists all of the MATLAB special symbols described in this chapter, along with a brief description of each one.

Special Symbols

- + Addition
- Subtraction
- * Multiplication
- / Division
- ^ Exponentiation

1.6 Exercises

- 1.1** The following MATLAB statements plot the function $y(x) = 2e^{-0.2x}$ for the range $0 \leq x \leq 10$.

```
x = 0:0.1:10;
y = 2 * exp( -0.2 * x);
plot(x,y);
```

Use the MATLAB Edit Window to create a new empty M-file, type these statements into the file, and save the file with the name `test1.m`. Then, execute the program by typing the name `test1` in the Command Window. What result do you get?

- 1.2** Get help on the MATLAB function `exp` using: (a) The “`help exp`” command typed in the Command Window, and (b) the Help Browser.
- 1.3** Use the `lookfor` command to determine how to take the base-10 logarithm of a number in MATLAB.
- 1.4** Suppose that $u = 1$ and $v = 3$. Evaluate the following expressions using MATLAB.
- $\frac{4u}{3v}$
 - $\frac{2v^{-2}}{(u + v)^2}$
 - $\frac{v^3}{v^3 - u^3}$
 - $\frac{4}{3} \pi v^2$
- 1.5** Use the MATLAB Help Browser to find the command required to show MATLAB’s current directory. What is the current directory when MATLAB starts up?
- 1.6** Use the MATLAB Help Browser to find out how to create a new directory from within MATLAB. Then, create a new directory called `mynewdir` under the current directory. Add the new directory to the top of MATLAB’s path.
- 1.7** Change the current directory to `mynewdir`. Then open an Edit Window and add the following lines:

```
% Create an input array from -2*pi to 2*pi
t = -2*pi:pi/10:2*pi;

% Calculate |sin(t)|
x = abs(sin(t));

% Plot result
plot(t,x);
```

Save the file with the name `test2.m`, and execute it by typing `test2` in the Command Window. What happens?

- 1.8** Close the Figure Window, and change back to the original directory that MATLAB started in. Next type “`test2`” in the Command Window. What happens, and why?

CHAPTER 2

MATLAB Basics

In this chapter, we will introduce some basic elements of the MATLAB language. By the end of the chapter, you will be able to write simple but functional MATLAB programs.

2.1 Variables and Arrays

The fundamental unit of data in any MATLAB program is the **array**. An array is a collection of data values organized into rows and columns, and known by a single name (see Figure 2.1). Individual data values within an array are accessed by including the name of the array followed by subscripts in parentheses that identify the row and column of the particular value. Even scalars are treated as arrays by MATLAB—they are simply arrays with only one row and one column.

Arrays can be classified as either **vectors** or **matrices**. The term “vector” is usually used to describe an array with only one dimension, while the term “matrix” is usually used to describe an array with two or more dimensions. In this text, we will use the term vector when discussing one-dimensional arrays, and the term matrix when discussing arrays with two or more dimensions. If a particular discussion applies to both types of arrays, we will use the generic term “array.”

The **size** of an array is specified by the number of rows and the number of columns in the array, with the number of rows mentioned first. The total number of elements in the array will be the product of the number of rows and the number of columns. For example, the sizes of the following arrays are:

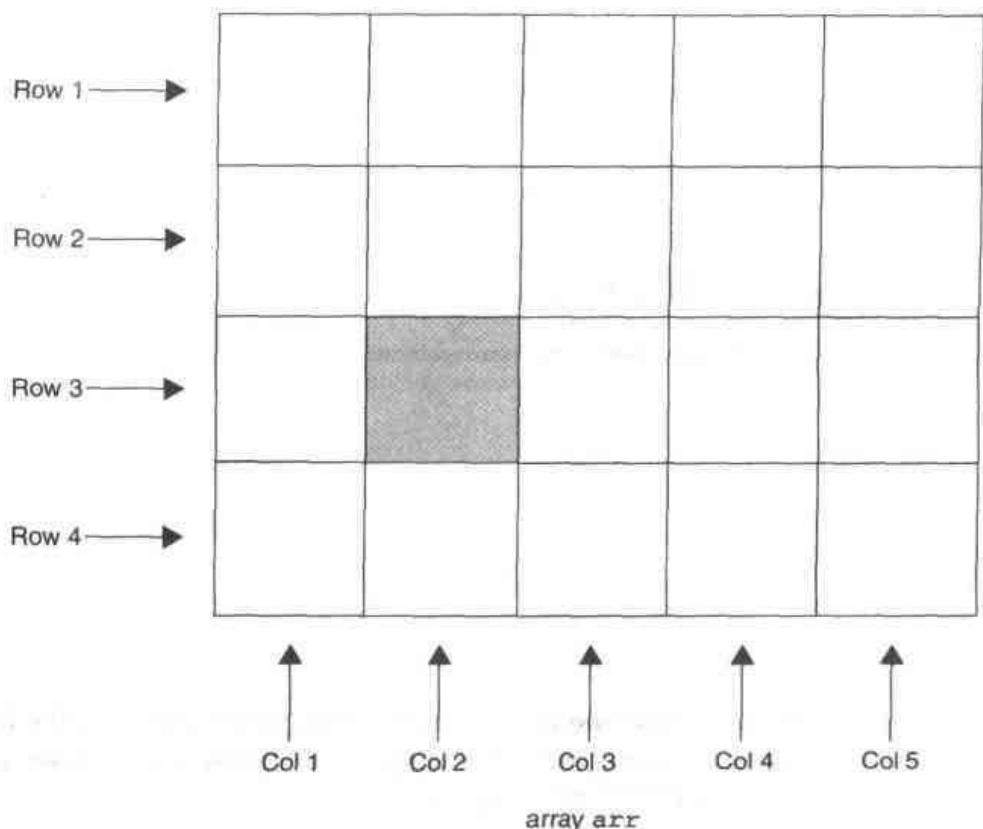


Figure 2.1 An array is a collection of data values organized into rows and columns. The array `arr` shown here consists of 20 elements, organized into four rows and five columns. The shaded element in this array would be addressed as `arr(3, 2)`.

Array	Size
$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$	This is a 3×2 matrix, containing 6 elements.
$b = [1 \ 2 \ 3 \ 4]$	This is a 1×4 array, known as a row vector , containing 4 elements.
$c = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$	This is a 3×1 array, known as a column vector , containing 3 elements.

Individual elements in an array are addressed by the array name followed by the row and column of the particular element. If the array is a row or column vector, then only one subscript is required. For example, in the above arrays $a(2, 1)$ is 3 and $c(2) = 2$.

A **MATLAB variable** is a region of memory containing an array, which is known by a user-specified name. The contents of the array can be used or modified at any time by including its name in an appropriate MATLAB command.

MATLAB variable names must begin with a letter, followed by any combination of letters, numbers, and the underscore (_) character. Only the first 31 characters are significant; if more than 31 are used, the remaining characters will be ignored. If two variables are declared with names that only differ in the 32nd character, MATLAB will treat them as the same variable.

Programming Pitfalls

Make sure that your variable names are unique in the first 31 characters. Otherwise, MATLAB will not be able to tell the difference between them.

When writing a program, it is important to pick meaningful names for the variables. Meaningful names make a program *much* easier to read and to maintain. Names such as day, month, and year are quite clear even to a person seeing a program for the first time. Since spaces cannot be used in MATLAB variable names, underscore characters can be substituted to create meaningful names. For example, *exchange rate* might become *exchange_rate*.

Good Programming Practice

Always give your variables descriptive and easy-to-remember names. For example, a currency exchange rate could be given the name *exchange_rate*. This practice will make your programs clearer and easier to understand.

It is also important to include a **data dictionary** in the header of any program that you write. A data dictionary lists the definition of each variable used in a program. The definition should include both a description of the contents of the item and the units in which it is measured. A data dictionary may seem unnecessary while the program is being written, but it is invaluable when you or another person have to go back and modify the program at a later time.

Good Programming Practice

Create a data dictionary for each program to make program maintenance easier.

The MATLAB language is case-sensitive, which means that uppercase and lowercase letters are not the same. Thus the variables *name*, *NAME*, and *Name* are all different in MATLAB. You must be careful to use the same capitalization every time that a variable name is used. While it is not required, it is customary to use all lowercase letters for ordinary variable names.

Good Programming Practice

Be sure to capitalize a variable exactly the same way each time that it is used. It is good practice to use only lowercase letters in variable names.

The most common types of MATLAB variables are `double` and `char`. Variables of type `double` consist of scalars or arrays of 64-bit double-precision floating-point numbers. They can hold real, imaginary, or complex values. The real and imaginary components of each variable can be positive or negative numbers in the range 10^{-308} to 10^{308} , with 15 to 16 significant decimal digits of accuracy. They are the principal numerical data type in MATLAB.

A variable of type `double` is automatically created whenever a numerical value is assigned to a variable name. For example, the following statement creates a variable named `var` containing a single element of type `double`, and stores the complex value $(1 + i)$ in it.

```
var = 1 + i;
```

Variables of type `char` consist of scalars or arrays of 16-bit values, each representing a single character. Arrays of this type are used to hold character strings. They are automatically created whenever a single character or a character string is assigned to a variable name. For example, the following statement creates a variable of type `char` whose name is `comment` and stores the specified string in it. After the statement is executed, `comment` will be a 1×26 character array.

```
comment = 'This is a character string';
```

In a language such as C, the type and name of every variable must be explicitly declared in a program before it is used. These languages are said to be **strongly typed**. In contrast, MATLAB is a **weakly typed** language. Variables can be created at any time by simply assigning values to them, and the type of data assigned to the variable determines the type of variable that is created.

2.2 Initializing Variables in MATLAB

MATLAB variables are automatically created when they are initialized. There are three common ways to initialize a variable in MATLAB:

1. Assign data to the variable in an assignment statement.
2. Input data into the variable from the keyboard.
3. Read data from a file.

The first two ways will be discussed here, and the third approach will be discussed in Section 2.7.

2.2.1 Initializing Variables in Assignment Statements

The simplest way to create and initialize a variable is to assign it one or more values in an **assignment statement**. An assignment statement has the general form

```
var = expression
```

where `var` is the name of a variable, and `expression` is a scalar constant, an array, or combination of constants, other variables, and mathematical operations (+, -, etc.). The value of the expression is calculated using the normal rules of mathematics, and the resulting values are stored in named variable. Simple examples of initializing variables with assignment statements include the following.

```
var = 40*i;
var2 = var / 5;
array = [1 2 3 4];
x = 1; y = 2;
```

The first example creates a scalar variable of type `double` and stores the imaginary number $40i$ in it. The second example creates a scalar variable `var2` and stores the result of the expression `var / 5` in it. The third example creates a variable `array` and stores a 4-element row vector in it. The last example shows that multiple assignment statements can be placed on a single line, provided that they are separated by semicolons or commas. Note that if any of the variables had already existed when the statements were executed, then their old contents would have been lost.

As the third example shows, variables can also be initialized with arrays of data. Such arrays are constructed using brackets (`[]`) and semicolons. All of the elements of an array are listed in **row order**. In other words, the values in each row are listed from left to right, with the topmost row first and the bottommost row last. Individual values within a row are separated by blank spaces or commas, and the rows themselves are separated by semicolons or new lines. The following expressions are all legal arrays that can be used to initialize a variable:

[3.4]

This expression creates a 1×1 array (a scalar) containing the value 3.4. The brackets are not required in this case.

[1.0 2.0 3.0]

This expression creates a 1×3 array containing the row vector [1 2 3].

[1.0; 2.0; 3.0]

This expression creates a 3×1 array containing the column vector $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$.

[1, 2, 3; 4, 5, 6]

This expression creates a 2×3 array containing the matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$.

[1, 2, 3

This expression creates a 2×3 array containing the matrix $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$. The end of the first line terminates the first row.

4, 5, 6]

[]

This expression creates an **empty array**, which contains no rows and no columns. (Note that this is not the same as an array containing zeros.)

Note that the number of elements in every row of an array must be the same, and the number of elements in every column must be the same. An expression such as

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

is illegal because row 1 has three elements while row 2 has only two elements.

Programming Pitfalls

The number of elements in every row of an array must be the same, and the number of elements in every column must be the same. Attempts to define an array with different numbers of elements in its rows or different numbers of elements in its columns will produce an error when the statement is executed.

The expressions used to initialize arrays can include algebraic operations and all or portions of previously defined arrays. For example, the assignment statements

```
a = [0 1+7];
b = [a(2) 7 a];
```

will define an array $a = [0 \ 8]$ and an array $b = [8 \ 7 \ 0 \ 8]$.

Also, not all of the elements in an array must be defined when it is created. If a specific array element is defined and one or more of the elements before it are not, then the earlier elements will automatically be created and initialized to zero. For example, if c is not previously defined, the statement

```
c(2,3) = 5;
```

will produce the matrix $c = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 5 \end{bmatrix}$. Similarly, an array can be extended by specifying a value for an element beyond the currently defined size. For example, suppose that array $d = [1 \ 2]$. Then the statement

```
d(4) = 4;
```

will produce the array $d = [1 \ 2 \ 0 \ 4]$.

The semicolon at the end of each assignment statement shown above has a special purpose: it *suppresses the automatic echoing of values* that normally occurs whenever an expression is evaluated in an assignment statement. If an assignment statement is typed without the semicolon, the results of the statement are automatically displayed in the Command Window.

```
>> e = [1, 2, 3; 4, 5, 6]
e =
    1     2     3
    4     5     6
```

If a semicolon is added at the end of the statement, the echoing disappears. Echoing is an excellent way to quickly check your work, but it seriously slows down the execution of MATLAB programs. For that reason, we normally suppress echoing at all times.

However, echoing the results of calculations makes a great quick-and-dirty debugging tool. If you are not certain what the results of a specific assignment statement are, just leave off the semicolon from that statement, and the results will be displayed in the Command Window as the statement is executed.

Good Programming Practice

Use a semicolon at the end of all MATLAB assignment statements to suppress echoing of assigned values in the Command Window. This greatly speeds program execution.

Good Programming Practice

If you need to examine the results of a statement during program debugging, you may remove the semicolon from that statement only so that its results are echoed in the Command Window.

2.2.2 Initializing with Shortcut Expressions

It is easy to create small arrays by explicitly listing each term in the array, but what happens when the array contains hundreds or even thousands of elements? It is just not practical to write out each element in the array separately!

MATLAB provides a special shortcut notation for these circumstances using the **colon operator**. The colon operator specifies a whole series of values by specifying the first value in the series, the stepping increment, and the last value in the series. The general form of a colon operator is

`first:incr:last`

where `first` is the first value in the series, `incr` is the stepping increment, and `last` is the last value in the series. If the increment is one, it may be omitted. For

example, the expression `1:2:10` is a shortcut for a 1×5 row vector containing the values 1, 3, 5, 7, and 9.

```
>> x = 1:2:10
x =
    1     3     5     7     9
```

With colon notation, an array can be initialized to have the hundred values $\frac{\pi}{100}, \frac{2\pi}{100}, \dots, \pi$ as follows.

```
angles = (0.01:0.01:1) * pi;
```

Shortcut expressions can be combined with the **transpose operator** (`'`) to initialize column vectors and more complex matrices. The transpose operator swaps the row and columns of any array that it is applied to. Thus the expression

```
f = [1:4]';
```

generates a 4-element row vector `[1 2 3 4]`, and then transposes it into the

4-element column vector $f = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$. Similarly, the expressions

```
g = 1:4;
h = [g' g'];
```

will produce the matrix $h = \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \\ 4 & 4 \end{bmatrix}$.

2.2.3 Initializing with Built-in Functions

Arrays can also be initialized using built-in MATLAB functions. For example, the function `zeros` can be used to create an all-zero array of any desired size. There are several forms of the `zeros` function. If the function has a single scalar argument, it will produce a square array using the single argument as both the number of rows and the number of columns. If the function has two scalar arguments, the first argument will be the number of rows, and the second argument will be the number of columns. Since the `size` function returns two values containing the number of rows and columns in an array, it can be combined with the `zeros` function to generate an array of zeros that is the same size as another array. Some examples using the `zeros` function follow.

```
a = zeros(2);
b = zeros(2,3);
c = [1 2; 3 4];
d = zeros(size(c));
```

Table 2.1 MATLAB Functions Useful for Initializing Variables

Function	Purpose
<code>zeros(n)</code>	Generates an $n \times n$ matrix of zeros.
<code>zeros(n,m)</code>	Generates an $n \times m$ matrix of zeros.
<code>zeros(size(arr))</code>	Generates a matrix of zeros of the same size as <code>arr</code> .
<code>ones(n)</code>	Generates an $n \times n$ matrix of ones.
<code>ones(n,m)</code>	Generates an $n \times m$ matrix of ones.
<code>ones(size(arr))</code>	Generates a matrix of ones of the same size as <code>arr</code> .
<code>eye(n)</code>	Generates an $n \times n$ identity matrix.
<code>eye(n,m)</code>	Generates an $n \times m$ identity matrix.
<code>length(arr)</code>	Returns the length of a vector or the longest dimension of a 2-D array.
<code>size(arr)</code>	Returns two values specifying the number of rows and columns in <code>arr</code> .

These statements generate the following arrays.

$$\begin{aligned} a &= \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & b &= \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ c &= \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} & d &= \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \end{aligned}$$

Similarly, the `ones` function can be used to generate arrays containing all ones, and the `eye` function can be used to generate arrays containing **identity matrices**, in which all on-diagonal elements are one, while all off-diagonal elements are zero. Table 2.1 contains a list of common MATLAB functions useful for initializing variables.

2.2.4 Initializing Variables with Keyboard Input

It is also possible to prompt a user and initialize a variable with data that he or she types directly at the keyboard. This option allows a script file to prompt a user for input data values while it is executing. The `input` function displays a prompt string in the Command Window and then waits for the user to type in a response. For example, consider the following statement.

```
my_val = input('Enter an input value:');
```

When this statement is executed, MATLAB prints out the string 'Enter an input value:', and then waits for the user to respond. If the user enters a single number, it may just be typed in. If the user enters an array, it must be enclosed in brackets. In either case, whatever is typed will be stored in variable `my_val` when the Enter key is pressed. If *only* the Enter key is pressed, then an empty matrix will be created and stored in the variable.

If the `input` function includes the character '`s`' as a second argument, then the input data is returned to the user as a character string. Thus, the statement

```
>> in1 = input('Enter data: ');
Enter data: 1.23
```

stores the value 1.23 into `in1`, while the statement

```
>> in2 = input('Enter data: ','s');
Enter data: 1.23
```

stores the character string '1.23' into `in2`.

Quiz 2.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 2.1 and 2.2. If you have trouble with the quiz, reread the sections, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. What is the difference between an array, a matrix, and a vector?
2. Answer the following questions for the array shown below.

$$c = \begin{bmatrix} 1.1 & -3.2 & 3.4 & 0.6 \\ 0.6 & 1.1 & -0.6 & 3.1 \\ 1.3 & 0.6 & 5.5 & 0.0 \end{bmatrix}$$

- (a) What is the size of `c`?
- (b) What is the value of `c(2, 3)`?
- (c) List the subscripts of all elements containing the value 0.6.
3. Determine the size of the following arrays. Check your answers by entering the arrays into MATLAB and using the `whos` command or the Workspace Browser. Note that the later arrays may depend on the definitions of arrays defined earlier in this exercise.
 - (a) `u = [10 20*i 10+20];`
 - (b) `v = [-1; 20; 3];`
 - (c) `w = [1 0 -9; 2 -2 0; 1 2 3];`
 - (d) `x = [u' v];`
 - (e) `y(3,3) = -7;`
 - (f) `z = [zeros(4,1) ones(4,1) zeros(1,4)'];`
 - (g) `v(4) = x(2,1);`
4. What is the value of `w(2,1)` above?
5. What is the value of `x(2,1)` above?
6. What is the value of `y(2,1)` above?
7. What is the value of `v(3)` after statement (g) is executed?

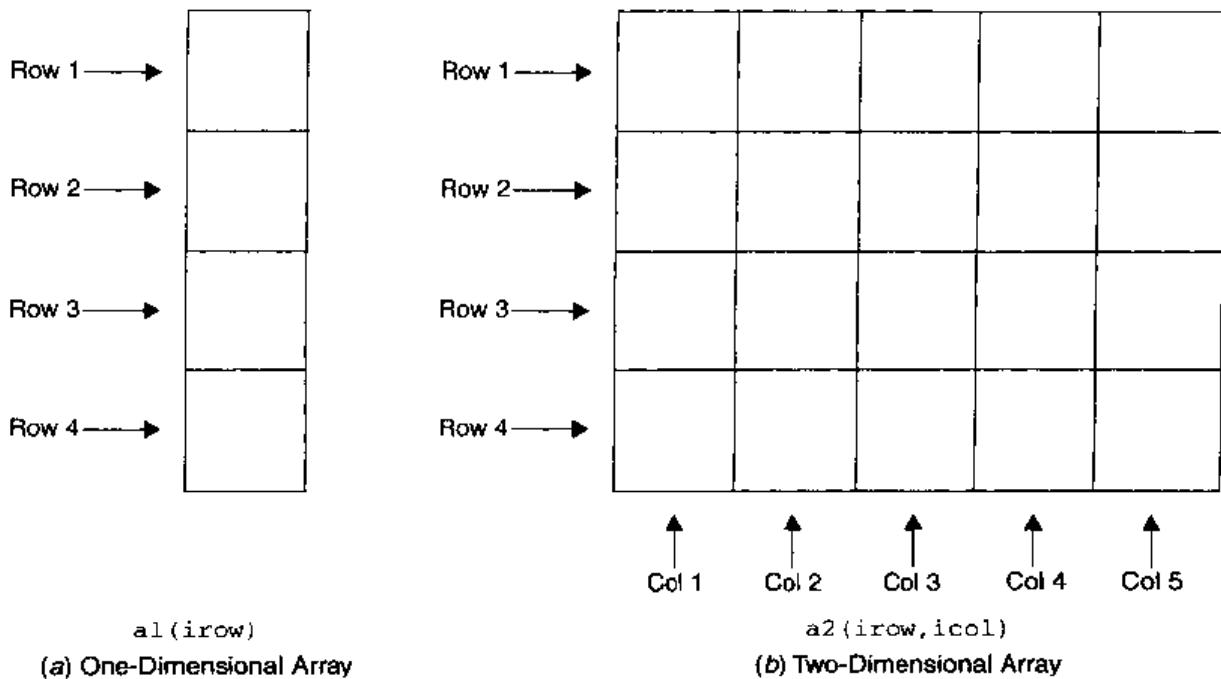


Figure 2.2 Representations of one- and two-dimensional arrays.

2.3 Multidimensional Arrays

As we have seen, MATLAB arrays can have one or more dimensions. One-dimensional arrays can be visualized as a series of values laid out in a column, with a single subscript used to select the individual array elements (Figure 2.2a). Such arrays are useful to describe data that is a function of one independent variable, such as a series of temperature measurements made at fixed intervals of time.

Some types of data are functions of more than one independent variable. For example, we might want to measure the temperature at five different locations at four different times. In this case, our 20 measurements could logically be grouped into five different columns of four measurements each, with a separate column for each location (Figure 2.2b). In this case, we will use two subscripts to access a given element in the array: the first one to select the row and the second one to select the column. Such arrays are called **two-dimensional arrays**. The number of elements in a two-dimensional array will be the product of the number of rows and the number of columns in the array.

MATLAB allows us to create arrays with as many dimensions as necessary for any given problem. These arrays have one subscript for each dimension, and an individual element is selected by specifying a value for each subscript. The total number of elements in the array will be the product of the maximum value of each subscript. For example, the following two statements create a $2 \times 3 \times 2$ array c:

```

>> c(:,:,1)=[1 2 3; 4 5 6];
>> c(:,:,2)=[7 8 9; 10 11 12];
>> whos c
  Name      Size            Bytes  Class
  c            2x3x2          96   double array

```

This array contains 12 elements organized ($2 \times 3 \times 2$), and its contents can be displayed just like that of any other array.

```

>> c
c(:,:,1) =
    1     2     3
    4     5     6
c(:,:,2) =
    7     8     9
   10    11    12

```

2.3.1 Storing Multidimensional Arrays in Memory

A two-dimensional array with m rows and n columns will contain $m \times n$ elements, and these elements will occupy $m \times n$ successive locations in the computer's memory. How are the elements of the array arranged in the computer's memory? MATLAB always allocates array elements in **column major order**. That is, MATLAB allocates the first column in memory, then the second, then the third, and so on, until all of the columns have been allocated. Figure 2.3 illustrates this memory allocation scheme for a 4×3 array a . As we can see, element $a(1, 2)$ is really the fifth element allocated in memory. The order in which elements are allocated in memory will become important when we discuss single-subscript addressing in the next section, and low-level I/O functions in Chapter 8.

This same allocation scheme applies to arrays with more than two dimensions. The first array subscript is incremented most rapidly, the second subscript is incremented less rapidly, and so forth, and the last subscript is incremented most slowly. For example, in a $2 \times 2 \times 2$ array, the elements would be allocated in the following order: $(1,1,1)$, $(2,1,1)$, $(1,2,1)$, $(2,2,1)$, $(1,1,2)$, $(2,1,2)$, $(1,2,2)$, $(2,2,2)$.

2.3.2 Accessing Multidimensional Arrays with a Single Subscript

One of MATLAB's peculiarities is that it will permit a user or programmer to treat a multidimensional array as though it were a one-dimensional array whose length is equal to the number of elements in the multidimensional array. If a multidimensional array is addressed with a single dimension, then the elements will be accessed in the order in which they were allocated in memory.

For example, suppose that we declare the 4×3 element array a as follows:

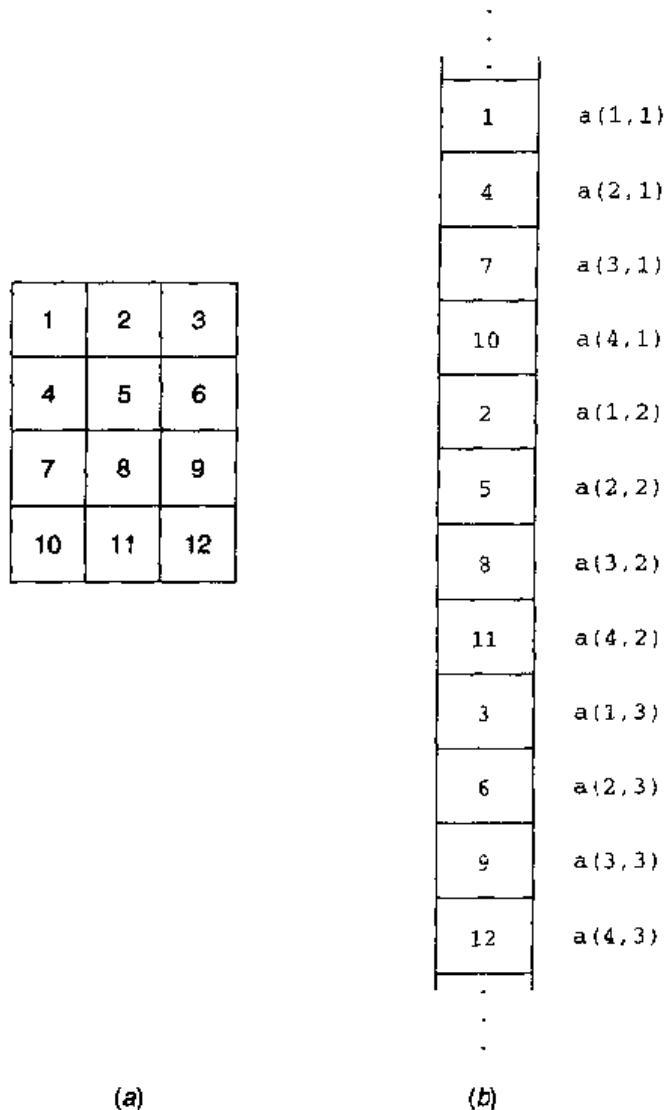


Figure 2.3 (a) Data values for array a. (b) Layout of values in memory for array a.

```
>> 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
```

Then the value of $a(5)$ will be 2, which is the value of element $a(1,2)$ because $a(1,2)$ was allocated fifth in memory.

Under normal circumstances, you should never use this feature of MATLAB. Addressing multidimensional arrays with a single subscript is a recipe for confusion.

Good Programming Practice

Always use the proper number of dimensions when addressing a multidimensional array.

2.4 Subarrays

It is possible to select and use subsets of MATLAB arrays as though they were separate arrays. To select a portion of an array, just include a list of all the elements to be selected in the parentheses after the array name. For example, suppose array arr1 is defined as follows.

```
arr1 = [1.1 -2.2 3.3 -4.4 5.5];
```

Then arr1(3) is just 3.3, arr1([1 4]) is the array [1.1 -4.4], and arr1(1:2:5) is the array [1.1 3.3 5.5].

For a two-dimensional array, a colon can be used in a subscript to select all of the values of that subscript. For example, suppose

```
arr2 = [1 2 3; -2 -3 -4; 3 4 5];
```

This statement would create an array arr2 containing the values $\begin{bmatrix} 1 & 2 & 3 \\ -2 & -3 & -4 \\ 3 & 4 & 5 \end{bmatrix}$.

With this definition, the subarray arr2(1,:) would be [1 2 3], and the

subarray arr2(:,1:2:3) would be $\begin{bmatrix} 1 & 3 \\ -2 & -4 \\ 3 & 5 \end{bmatrix}$.

2.4.1 The end Function

MATLAB includes a special function named `end` that is very useful for creating array subscripts. When used in an array subscript, `end` *returns the highest value taken on by that subscript*. For example, suppose that array arr3 is defined as follows:

```
arr3 = [1 2 3 4 5 6 7 8];
```

Then arr1(5:`end`) would be the array [5 6 7 8], and `array(end)` would be the value 8.

The value returned by `end` is always the *highest value* of a given subscript. If `end` appears in different subscripts, it can return *different* values within the same expression. For example, suppose that the 3×4 array arr4 is defined as follows:

```
arr4 = [1 2 3 4; 5 6 7 8; 9 10 11 12];
```

Then the expression `arr4(2:end, 2:end)` would return the array $\begin{bmatrix} 6 & 7 & 8 \\ 10 & 11 & 12 \end{bmatrix}$. Note that the first `end` returned the value 3, while the second `end` returned the value 4!

2.4.2 Using Subarrays on the Left-Hand Side of an Assignment Statement

It is also possible to use subarrays on the left-hand side of an assignment statement to update only some of the values in an array, as long as the `shape` (the number of rows and columns) of the values being assigned matches the shape of the subarray. If the shapes do not match, then an error will occur. For example, suppose that the 3×4 array `arr4` is defined as follows:

```
>> arr4 = [1 2 3 4; 5 6 7 8; 9 10 11 12]
arr4 =
    1     2     3     4
    5     6     7     8
    9    10    11    12
```

Then the following assignment statement is legal, since the expressions on both sides of the equal sign have the same shape (2×2):

```
>> arr4(1:2,[1 4]) = [20 21; 22 23]
arr4 =
    20     2     3     21
    22     6     7     23
    9    10    11    12
```

Note that the array elements (1,1), (1,4), (2,1), and (2,4) were updated. In contrast, the following expression is illegal because the two sides do not have the same shape.

```
>> arr5(1:2,1:2) = [3 4]
??? In an assignment A(matrix,matrix) = B, the number
of rows in B and the number of elements in the A row
index matrix must be the same.
```

Programming Pitfalls

For assignment statements involving subarrays, the *shapes of the subarrays on both sides of the equal sign must match*. MATLAB will produce an error if they do not match.

There is a major difference in MATLAB between assigning values to a subarray and assigning values to an array. If values are assigned to a subarray, *only those values are updated, and all other values in the array remain unchanged*. On the other hand, if values are assigned to an array, *the entire contents of the array are deleted and replaced by the new values*. For example, suppose that the 3×4 array arr4 is defined as follows:

```
>> arr4 = [1 2 3 4; 5 6 7 8; 9 10 11 12]
arr4 =
    1     2     3     4
    5     6     7     8
    9    10    11    12
```

Then the following assignment statement replaces the *specified elements* of arr4:

```
>> arr4(1:2,[1 4]) = [20 21; 22 23]
arr4 =
    20     2     3     21
    22     6     7     23
    9     10    11    12
```

In contrast, the following assignment statement replaces the *entire contents* of arr4 with a 2×2 array:

```
>> arr4 = [20 21; 22 23]
arr4 =
    20     21
    22     23
```

Be sure to distinguish between assigning values to a subarray and assigning values to an array. MATLAB behaves differently in these two cases.

2.4.3 Assigning a Scalar to a Subarray

A scalar value on the right-hand side of an assignment statement always matches the shape specified on the left-hand side. The scalar value is copied into every element specified on the left-hand side of the statement. For example, assume that the 3×4 array arr4 is defined as follows:

```
arr4 = [1 2 3 4; 5 6 7 8; 9 10 11 12];
```

Then the expression shown below assigns the value one to four elements of the array.

```
>> arr4(1:2,1:2) = 1
arr5 =
    1     1     3     4
    1     1     7     8
    9    10    11    12
```

2.5 Special Values

MATLAB includes a number of predefined special values. These predefined values can be used at any time in MATLAB, without initializing them first. A list of the most common predefined values is given in Table 2.2.

These predefined values are stored in ordinary variables, so they can be overwritten or modified by a user. If a new value is assigned to one of the predefined variables, then that new value will replace the default one in all later calculations. For example, consider the following statements that calculate the circumference of a circle with a 10-cm radius.

```
circ1 = 2 * pi * 10
pi = 3;
circ2 = 2 * pi * 10
```

Table 2.2 Predefined Special Values

Function	Purpose
pi	Contains π to 15 significant digits.
i, j	Contain the value $i (\sqrt{-1})$.
Inf	This symbol represents machine infinity. It is usually generated as a result of a division by 0.
NaN	This symbol stands for Not-a-Number. It is the result of an undefined mathematical operation, such as the division of zero by zero.
clock	This special variable contains the current date and time in the form of a 6-element row vector containing the year, month, day, hour, minute, and second.
date	Contains the current data in a character string format, such as 24-Nov-1999.
eps	This variable name is short for <i>epsilon</i> . It is the smallest difference between two numbers that can be represented on the computer.
ans	A special variable used to store the result of an expression if that result is not explicitly assigned to some other variable.

In the first statement, `pi` has its default value of `3.14159...`, so `circ1` is 62.8319, which is the correct circumference. The second statement redefines `pi` to be 3, so in the third statement `circ2` is 60. Changing a predefined value in the program has created an incorrect answer, and also introduced a subtle and hard-to-find bug. Imagine trying to locate the source of such a hidden error in a 10,000-line program!

Programming Pitfalls

Never redefine the meaning of a predefined variable in MATLAB. It is a recipe for disaster, producing subtle and hard-to-find bugs.

Quiz 2.2

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 2.3 through 2.5. If you have trouble with the quiz, reread the sections, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

- Assume that array `c` is defined as shown, and determine the contents of the following subarrays:

$$c = \begin{bmatrix} 1.1 & -3.2 & 3.4 & 0.6 \\ 0.6 & 1.1 & -0.6 & 3.1 \\ 1.3 & 0.6 & 5.5 & 0.0 \end{bmatrix}$$

- (a) `c(2,:)`
 - (b) `c(:,end)`
 - (c) `c(1:2,2:end)`
 - (d) `c(6)`
 - (e) `c(4:end)`
 - (f) `c(1:2,2:4)`
 - (g) `c([1 4],2)`
 - (h) `c([2 2],[3 3])`
- Determine the contents of array `a` after the following statements are executed.
 - (a) `a = [1 2 3; 4 5 6; 7 8 9];`
`a([3 1],:) = a([1 3],:);`
 - (b) `a = [1 2 3; 4 5 6; 7 8 9];`
`a([1 3],:) = a([2 2],:);`
 - (c) `a = [1 2 3; 4 5 6; 7 8 9];`
`a = a([2 2],:);`

3. Determine the contents of array *a* after the following statements are executed.

```
(a) a = eye(3,3);
   b = [1 2 3];
   a(2,:) = b;

(b) a = eye(3,3);
   b = [4 5 6];
   a(:,3) = b';

(c) a = eye(3,3);
   b = [7 8 9];
   a(3,:) = b([3 1 2]);

(d) a = eye(3,3);
   b = [7 8 9];
   a(3,:) = b([3 1 2]);
```

2.6 Displaying Output Data

There are several ways to display output data in MATLAB. This simplest way is one we have already seen—just leave the semicolon off the end of a statement and it will be echoed to the Command Window. We will now explore a few other ways to display data.

2.6.1 Changing the Default Format

When data is echoed in the Command Window, integer values are always displayed as integers, and other values are printed using a **default format**. The default format for MATLAB shows four digits after the decimal point, and it may be displayed in scientific notation with an exponent if the number is too large or too small. For example, the statements

```
x = 100.11
y = 1001.1
z = 0.00010011
```

produce the following output

```
x =
100.1100

y =
1.0011e+003

z =
1.0011e-004
```

This default format can be changed using the **format** command. The **format** command changes the default format according to the values given in Table 2.3.

Table 2.3 Output Display Formats

Format Command	Results	Example ^a
format short	4 digits after decimal (default format)	12.3457
format long	14 digits after decimal	12.34567890123457
format short e	5 digits plus exponent	1.2346e+001
format short g	5 total digits with or without exponent	12.346
format long e	15 digits plus exponent	1.234567890123457e+001
format long g	15 total digits with or without exponent	12.3456789012346
format bank	“dollars and cents” format	12.35
format hex	hexadecimal display of bits	4028b0fc3d32f707a
format rat	approximate ratio of small integers	1000/81
format compact	suppress extra line feeds	
format loose	restore extra line feeds	
format +	Only the <i>signs</i> of the numbers are printed	+

^aThe data value used for the example is 12.345678901234567 in all cases.

The default format can be modified to display more significant digits of data, force the display to be in scientific notation, to display data to two decimal digits, or to eliminate extra line feeds to get more data visible in the Command Window at a single time. Experiment with the commands in Table 2.3 for yourself.

2.6.2 The `disp` Function

Another way to display data is with the `disp` function. The `disp` function accepts an array argument and displays the value of the array in the Command Window. If the array is of type `char`, then the character string contained in the array is printed out.

This function is often combined with the functions `num2str` (convert a number to a string) and `int2str` (convert an integer to a string) to create messages to be displayed in the Command Window. For example, the following MATLAB statements will display “The value of pi = 3.1416” in the Command Window. The first statement creates a string array containing the message, and the second statement displays the message.

```
str = ['The value of pi = ' num2str(pi)];
disp (str);
```

2.6.3 Formatted Output with the `fprintf` Function

An even more flexible way to display data is with the `fprintf` function. The `fprintf` function displays one or more values together with related text and lets

the programmer control the way that the displayed value appears. The general form of this function when it is used to print to the Command Window is:

```
fprintf(format,data)
```

where **format** is a string describing the way the **data** is to be printed, and **data** is one or more scalars or arrays to be printed. The **format** is a character string containing text to be printed plus special characters describing the format of the data. For example, the function

```
fprintf('The value of pi is %f \n',pi)
```

will print out 'The value of pi is 3.141593' followed by a line feed. The characters **%f** are called **conversion characters**; they indicate that a value in the data list should be printed out in floating point format at that location in the format string. The characters such as **\n** are called **escape characters**. **\n** indicates that a line feed should be issued so that the following text starts on a new line. There are many types of conversion characters and escape characters that can be used in an **fprintf** function. A few of them are listed in Table 2.4, and a complete list can be found in Chapter 8.

It is also possible to specify the width of the field in which a number will be displayed and the number of decimal places to display. This is done by specifying the width and precision after the **%** sign and before the **f**. For example, the function

```
fprintf('The value of pi is %6.2f \n',pi)
```

will print out 'The value of pi is 3.14' followed by a line feed. The conversion characters **%6.2f** indicate that the first data item in the function should be printed out in floating point format in a field six characters wide, including two digits after the decimal point.

The **fprintf** function has one very significant limitation: *it only displays the real portion of a complex value*. This limitation can lead to misleading results when calculations produce complex answers. In those cases, it is better to use the **disp** function to display answers.

Table 2.4 Common Special Characters in **fprintf Format Strings**

Format String	Results
%d	Display value as an integer
%e	Display value in exponential format.
%f	Display value in floating point format.
%g	Display value in either floating point or exponential format, whichever is shorter.
\n	Skip to a new line.

For example, the following statements calculate a complex value x and display it using both `fprintf` and `disp`.

```
x = 2 * ( 1 - 2*i )^3;
str = ['disp: x = ' num2str(x)];
disp(str);
fprintf('fprintf: x = %8.4f\n',x);
```

The results printed out by these statements are

```
disp: x = -22+4i
fprintf: x = -22.0000
```

Note that the `fprintf` function ignored the imaginary part of the answer.

Programming Pitfalls

The `fprintf` function only displays the *real* part of a complex number, which can produce misleading answers when working with complex values.

2.7 Data Files

There are many ways to load and save data files in MATLAB, most of which will be addressed in Chapter 8. For the moment, we will consider only the `load` and `save` commands, which are the simplest ones to use.

The `save` command saves data from the current MATLAB workspace into a disk file. The most common form of this command is

```
save filename var1 var2 var3
```

where `filename` is the name of the file where the variables are saved, and `var1`, `var2`, and so forth, are the variables to be saved in the file. By default, the file name will be given the extent “`mat`” and such data files are called MAT-files. If no variables are specified, then the entire contents of the workspace are saved.

MATLAB saves MAT-files in a special compact format, which preserves many details, including the name and type of each variable, the size of each array, and all data values. A MAT-file created on any platform (PC, Mac, or Unix) can be read on any other platform, so MAT-files are a good way to exchange data between computers if both computers run MATLAB. Unfortunately, the MAT-file is in a format that cannot be read by other programs. If data must be shared with other programs, then the `-ascii` option should be specified, and the data values will be written to the file as ASCII character strings separated by spaces. However, the special information such as variable names and types are lost when the data is saved in ASCII format, and the resulting data file will be much larger.

For example, suppose the array `x` is defined as

```
x=[1.23 3.14 6.28; -5.1 7.00 0];
```

The command “`save x.dat x -ascii`” will produce a file named `x.dat` containing the following data.

```
1.230000e+000 3.140000e+000 6.280000e+000
-5.100000e+000 7.000000e+000 0.000000e+000
```

This data is in a format that can be read by spreadsheets or by programs written in other computer languages, so it makes it easy to share data between MATLAB programs and other applications.

Good Programming Practice

If data must be exchanged between MATLAB and other programs, save the MATLAB data in ASCII format. If the data will only be used in MATLAB, save the data in MAT-file format.

MATLAB does not care what file extent is used for ASCII files. However, it is better for the user if a consistent naming convention is used, and an extent of “`dat`” is a common choice for ASCII files.

Good Programming Practice

Save ASCII data files with a “`dat`” file extent to distinguish them from MAT-files, which have a “`mat`” file extent.

The `load` command is the opposite of the `save` command. It loads data from a disk file into the current MATLAB workspace. The most common form of this command is

```
load filename
```

where `filename` is the name of the file to be loaded. If the file is a MAT-file, then all of the variables in the file will be restored, with the names and types the same as before. If a list of variables is included in the command, then only those variables will be restored.

MATLAB can load data created by other programs in space-separated ASCII format. It will examine each file to determine whether it is a MAT-file or an ASCII file, and treat the data accordingly. MATLAB can be forced to treat a file as an ASCII file by specifying the `-ascii` option on the `load` statement. The contents of an ASCII file will be converted into a MATLAB array having the same name as the file that the data was loaded from. For example, suppose that an ASCII data file named `x.dat` contains the following data.

```
1.23 3.14 6.28
-5.1 7.00 0
```

Then the command “`load x.dat`” will create a 2×3 array named `x` in the current workspace, containing these data values.

Quiz 2.3

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 2.6 and 2.7. If you have trouble with the quiz, reread the sections, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. How would you tell MATLAB to display all real values in exponential format with 15 significant digits?

2. What do the following sets of statements do? What is the output from them?

```
(a) radius = input('Enter circle radius:\n');
area = pi * radius^2;
str = ['The area is ' num2str(area)];
disp(str);
(b) value = int2str(pi);
disp(['The value is ' value '!']);
```

3. What do the following sets of statements do? What is the output from them?

```
value = 123.4567e2;
fprintf('value = %e\n',value);
fprintf('value = %f\n',value);
fprintf('value = %g\n',value);
fprintf('value = %12.4f\n',value);
```

2.8 Scalar and Array Operations

Calculations are specified in MATLAB with an assignment statement, whose general form is

```
variable_name = expression;
```

The assignment statement calculates the value of the expression to the right of the equal sign, and *assigns* that value to the variable named on the left of the equal sign. Note that the equal sign does not mean equality in the usual sense of the word. Instead, it means: *store the value of expression into location variable_name*. For this reason, the equal sign is called the **assignment operator**. A statement like

```
ii = ii + 1;
```

is complete nonsense in ordinary algebra, but makes perfect sense in MATLAB. It means: take the current value stored in variable `i`, add one to it, and store the result back into variable `i`.

2.8.1 Scalar Operations

The expression to the right of the assignment operator can be any valid combination of scalars, arrays, parentheses, and arithmetic operators. The standard arithmetic operations between two scalars are given in Table 2.5.

Parentheses can be used to group terms whenever desired. When parentheses are used, the expressions inside the parentheses are evaluated before the expressions outside the parentheses. For example, the expression $2 ^ ((8+2)/5)$ is evaluated as shown below.

$$\begin{aligned} 2 ^ ((8+2)/5) &= 2 ^ (10/5) \\ &= 2 ^ 2 \\ &= 4 \end{aligned}$$

2.8.2 Array and Matrix Operations

MATLAB supports two types of operations between arrays, known as *array operations* and *matrix operations*. **Array operations** are operations performed between arrays on an **element-by-element basis**. That is, the operation is performed on corresponding elements in the two arrays. For example, if $a = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$ and $b = \begin{bmatrix} -1 & 3 \\ -2 & 1 \end{bmatrix}$, then $a+b = \begin{bmatrix} 0 & 5 \\ 1 & 5 \end{bmatrix}$. Note that for these operations to work, *the number of rows and columns in both arrays must be the same*. If not, MATLAB will generate an error message.

Array operations can also occur between an array and a scalar. If the operation is performed between an array and a scalar, the value of the scalar is applied to every element of the array. For example, if $a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, then $a+4 = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$.

Table 2.5 Arithmetic Operations between Two Scalars

Operation	Algebraic Form	MATLAB Form
Addition	$a + b$	<code>a + b</code>
Subtraction	$a - b$	<code>a - b</code>
Multiplication	$a \times b$	<code>a * b</code>
Division	$\frac{a}{b}$	<code>a / b</code>
Exponentiation	a^b	<code>a ^ b</code>

In contrast, **matrix operations** follow the normal rules of linear algebra, such as matrix multiplication. In linear algebra, the product $c = a \times b$ is defined by the equation

$$c(i, j) = \sum_{k=1}^n a(i, k) b(k, j)$$

For example, if $a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $b = \begin{bmatrix} -1 & 3 \\ -2 & 1 \end{bmatrix}$, then $a \times b = \begin{bmatrix} -1 & -2 \\ 3 & 1 \end{bmatrix}$. Note that for matrix multiplication to work, *the number of columns in matrix a must be equal to the number of rows in matrix b*.

MATLAB uses a special symbol to distinguish array operations from matrix operations. In the cases where array operations and matrix operations have a different definition, MATLAB uses a period before the symbol to indicate an array operation (for example, $\cdot *$). A list of common array and matrix operations is given in Table 2.6.

Table 2.6 Common Array and Matrix Operations

Operation	MATLAB Form	Comments
Array Addition	$a + b$	Array addition and matrix addition are identical.
Array Subtraction	$a - b$	Array subtraction and matrix subtraction are identical.
Array Multiplication	$a \cdot * b$	Element-by-element multiplication of a and b. Both arrays must be the same shape, or one of them must be a scalar.
Matrix Multiplication	$a * b$	Matrix multiplication of a and b. The number of columns in a must equal the number of rows in b.
Array Right Division	$a \cdot / b$	Element-by-element division of a and b: $a(i, j) / b(i, j)$. Both arrays must be the same shape, or one of them must be a scalar.
Array Left Division	$a . \backslash b$	Element-by-element division of a and b, but with b in the numerator: $b(i, j) / a(i, j)$. Both arrays must be the same shape, or one of them must be a scalar.
Matrix Right Division	a / b	Matrix division defined by $a * \text{inv}(b)$, where $\text{inv}(b)$ is the inverse of matrix b.
Matrix Left Division	$a \backslash b$	Matrix division defined by $\text{inv}(a) * b$, where $\text{inv}(a)$ is the inverse of matrix a.
Array Exponentiation	$a . ^ b$	Element-by-element exponentiation of a and b: $a(i, j) ^ b(i, j)$. Both arrays must be the same shape, or one of them must be a scalar.

New users often confuse array operations and matrix operations. In some cases, substituting one for the other will produce an illegal operation, and MATLAB will report an error. In other cases, both operations are legal, and MATLAB will perform the wrong operation and come up with a wrong answer. The most common problem happens when working with square matrices. Both array multiplication and matrix multiplication are legal for two square matrices of the same size, but the resulting answers are totally different. Be careful to specify exactly what you want!

Programming Pitfalls

Be careful to distinguish between array operations and matrix operations in your MATLAB code. It is especially common to confuse array multiplication with matrix multiplication.



Example 2.1

Assume that a , b , c , and d are defined as follows.

$$a = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \quad b = \begin{bmatrix} -1 & 2 \\ 0 & 1 \end{bmatrix}$$

$$c = \begin{bmatrix} 3 \\ 2 \end{bmatrix} \quad d = 5$$

What is the result of each of the following expressions?

- | | |
|---------------|---------------|
| (a) $a + b$ | (e) $a + c$ |
| (b) $a . * b$ | (f) $a + d$ |
| (c) $a * b$ | (g) $a . * d$ |
| (d) $a * c$ | (h) $a * d$ |

Solution

- (a) This is array or matrix addition: $a + b = \begin{bmatrix} 0 & 2 \\ 2 & 2 \end{bmatrix}$.
- (b) This is element-by-element array multiplication: $a . * b = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$.
- (c) This is matrix multiplication: $a * b = \begin{bmatrix} -1 & 2 \\ -2 & 5 \end{bmatrix}$.
- (d) This is matrix multiplication: $a * c = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$.
- (e) This operation is illegal, since a and c have different numbers of columns.
- (f) This is addition of an array to a scalar: $a + d = \begin{bmatrix} 6 & 5 \\ 7 & 6 \end{bmatrix}$.

(g) This is array multiplication: $a.*d = \begin{bmatrix} 5 & 0 \\ 10 & 5 \end{bmatrix}$.

(h) This is matrix multiplication: $a*d = \begin{bmatrix} 5 & 0 \\ 10 & 5 \end{bmatrix}$.

The matrix left division operation has a special significance that we must understand. A 3×3 set of simultaneous linear equations takes the form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \quad (2.1)$$

which can be expressed as

$$Ax = B \quad (2.2)$$

where $A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$, $B = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$, and $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$.

Equation (2.2) can be solved for x using linear algebra. The result is

$$x = A^{-1}B \quad (2.3)$$

Since the left division operator $A \setminus B$ is defined to be $\text{inv}(A) \times B$, the left division operator solves a system of simultaneous equations in a single statement!

2.9 Hierarchy of Operations

Often, many arithmetic operations are combined into a single expression. For example, consider the equation for the distance traveled by an object starting from rest and subjected to a constant acceleration.

```
distance = 0.5 * accel * time ^ 2
```

There are two multiplications and an exponentiation in this expression. In such an expression, it is important to know the order in which the operations are evaluated. If exponentiation is evaluated before multiplication, this expression is equivalent to

```
distance = 0.5 * accel * (time ^ 2)
```

But if multiplication is evaluated before exponentiation, this expression is equivalent to

```
distance = (0.5 * accel * time) ^ 2
```

These two equations have different results, and we must be able to unambiguously distinguish between them.

Table 2.7 Hierarchy of Arithmetic Operations

Precedence	Operation
1	The contents of all parentheses are evaluated, starting from the innermost parentheses and working outward.
2	All exponentials are evaluated, working from left to right.
3	All multiplications and divisions are evaluated, working from left to right.
4	All additions and subtractions are evaluated, working from left to right.

To make the evaluation of expressions unambiguous, MATLAB has established a series of rules governing the hierarchy, or order, in which operations are evaluated within an expression. The rules generally follow the normal rules of algebra. The order in which the arithmetic operations are evaluated is given in Table 2.7.

Example 2.2

Variables a , b , c , and d have been initialized to the following values.

$a = 3; b = 2; c = 5; d = 3;$

Evaluate the following MATLAB assignment statements.

- (a) $output = a*b+c*d;$
- (b) $output = a*(b+c)*d;$
- (c) $output = (a*b)+(c*d);$
- (d) $output = a^b^d;$
- (e) $output = a^{(b^d)};$

Solution

- | | |
|---|---|
| (a) Expression to evaluate:
Fill in numbers:
First, evaluate multiplications
and divisions from left to right: | $output = a*b+c*d;$
$output = 3*2+5*3;$

$output = 6 + 15;$
$output = 21$ |
| Now evaluate additions: | |
| (b) Expression to evaluate:
Fill in numbers:
First, evaluate parentheses:
Now, evaluate multiplications
and divisions from left to right: | $output = a*(b+c)*d;$
$output = 3*(2+5)*3;$
$output = 3*7*3;$

$output = 21*3;$
$output = 63;$ |

(c) Expression to evaluate: Fill in numbers: First, evaluate parentheses: Now evaluate additions:	output = (a*b)+(c*d); output = (3*2)+(5*3); output = 6 + 15; output = 21
(d) Expression to evaluate: Fill in numbers: Evaluate exponentials from left to right:	output = a^b^d; output = 3^2^3; output = 9^3; output = 729;
(e) Expression to evaluate: Fill in numbers: First, evaluate parentheses: Now, evaluate exponential:	output = a^(b^d); output = 3^(2^3); output = 3^8; output = 6561;

As we see, the order in which operations are performed has a major effect on the final result of an algebraic expression.

It is important that every expression in a program be made as clear as possible. Any program of value must not only be written but also be maintained and modified when necessary. You should always ask yourself: "Will I easily understand this expression if I come back to it in six months? Can another programmer look at my code and easily understand what I am doing?" If there is any doubt in your mind, use extra parentheses in the expression to make it as clear as possible.

Good Programming Practice

Use parentheses as necessary to make your equations clear and easy to understand.

If parentheses are used within an expression, then the parentheses must be balanced. That is, there must be an equal number of open parentheses and closed parentheses within the expression. It is an error to have more of one type than the other. Errors of this sort are usually typographical, and they are caught by the MATLAB interpreter when the command is executed. For example, the expression

$(2 + 4) / 2)$

produces an error when the expression is executed.

Quiz 2.4

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 2.8 and 2.9. If you have trouble with the quiz, reread

the sections, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. Assume that a , b , c , and d are defined as follows, and calculate the results of the following operations if they are legal. If an operation is illegal, explain why it is illegal.

$$a = \begin{bmatrix} 2 & 1 \\ -1 & 2 \end{bmatrix} \quad b = \begin{bmatrix} 0 & -1 \\ 3 & 1 \end{bmatrix} \quad c = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad d = -3$$

- (a) `result = a .* c;`
- (b) `result = a * [c c];`
- (c) `result = a .* [c c];`
- (d) `result = a + b * c;`
- (e) `result = a + b .* c;`

2. Solve for x in the equation $Ax = B$, where $A = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 3 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ and $B = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$
-

2.10 Built-in MATLAB Functions

In mathematics, a **function** is an expression that accepts one or more input values and calculates a single result from them. Scientific and technical calculations usually require functions that are more complex than the simple addition, subtraction, multiplication, division, and exponentiation operations that we have discussed so far. Some of these functions are very common and are used in many different technical disciplines. Others are rarer and specific to a single problem or a small number of problems. Examples of very common functions are the trigonometric functions, logarithms, and square roots. Examples of rarer functions include the hyperbolic functions, Bessel functions, and so forth. One of MATLAB's greatest strengths is that it comes with an incredible variety of built-in functions ready for use.

2.10.1 Optional Results

Unlike mathematical functions, MATLAB functions can return more than one result to the calling program. The function `max` is an example of such a function. This function normally returns the maximum value of an input vector, but it can also return a second argument containing the location in the input vector where the maximum value was found. For example, the statement

```
maxval = max ([1 -5 6 -3])
```

returns the result `maxval = 6`. However, if two variables are provided to store results in, the function returns *both* the maximum value *and* the location of the maximum value.

```
[maxval index] = max ([1 -5 6 -3])
```

produces the results `maxval = 6` and `index = 3`.

2.10.2 Using MATLAB Functions with Array Inputs

Many MATLAB functions are defined for one or more scalar inputs, and produce a scalar output. For example, the statement $y = \sin(x)$ calculates the sine of x and stores the result in y . If these functions receive an array of input values, then they will calculate an array of output values on an element-by-element basis. For example, if $x = [0 \text{ pi}/2 \text{ pi} 3\text{*pi}/2 2\text{*pi}]$, then the statement

```
y = sin(x)
```

will produce the result $y = [0 \text{ } 1 \text{ } 0 \text{ } -1 \text{ } 0]$.

2.10.3 Common MATLAB Functions

A few of the most common and useful MATLAB functions are shown in Table 2.8. These functions will be used in many examples and homework problems. If you need to locate a specific function not on this list, you can search for the function alphabetically or by subject using the MATLAB Help Browser.

Note that unlike most computer languages, many MATLAB functions work correctly for both real and complex inputs. MATLAB functions automatically calculate the correct answer, even if the result is imaginary or complex. For example, the function `sqrt(-2)` will produce a runtime error in languages such as C or Fortran. In contrast, MATLAB correctly calculates the imaginary answer.

```
> sqrt(-2)
ans =
0 + 1.4142i
```

2.11 Introduction to Plotting

MATLAB's extensive, device-independent plotting capabilities are one of its most powerful features. They make it very easy to plot any data at any time. To plot a data set, just create two vectors containing the x and y values to be plotted, and use the `plot` function.

For example, suppose that we wish to plot the function $y = x^2 - 10x + 15$ for values of x between 0 and 10. It takes only three statements to create this plot. The first statement creates a vector of x values between 0 and 10 using the colon operator. The second statement calculates the y values from the equation (note that we are using array operators here so that this equation is applied to each x value on an element-by-element basis). Finally, the third statement creates the plot.

```
x = 0:1:10;
y = x.^2 - 10.*x + 15;
plot(x,y);
```

Table 2.8 Common MATLAB Functions

Function	Description
Mathematical Functions	
<code>abs(x)</code>	Calculates $ x $.
<code>acos(x)</code>	Calculates $\cos^{-1}x$, with the results in <i>radians</i> .
<code>angle(x)</code>	Returns the phase angle of the complex value x , in <i>radians</i> .
<code>asin(x)</code>	Calculates $\sin^{-1}x$, with the results in <i>radians</i> .
<code>atan(x)</code>	Calculates $\tan^{-1}x$, with the results in <i>radians</i> .
<code>atan2(y,x)</code>	Calculates $\tan^{-1}\frac{y}{x}$ over all four quadrants of the circle (results in <i>radians</i> in the range $-\pi \leq \tan^{-1}\frac{y}{x} \leq \pi$).
<code>cos(x)</code>	Calculates $\cos x$, with x in radians.
<code>exp(x)</code>	Calculates e^x .
<code>log(x)</code>	Calculates the natural logarithm $\log_e x$.
<code>[value, index] = max(x)</code>	Returns the maximum value in vector x , and optionally the location of that value.
<code>[value, index] = min(x)</code>	Returns the minimum value in vector x , and optionally the location of that value.
<code>mod(x,y)</code>	Remainder, or modulo, function.
<code>sin(x)</code>	Calculates $\sin x$, with x in radians.
<code>sqrt(x)</code>	Calculates the square root of x .
<code>tan(x)</code>	Calculates $\tan x$, with x in radians.
Rounding Functions	
<code>ceil(x)</code>	Rounds x to the nearest integer towards positive infinity: <code>ceil(3.1) = 4</code> and <code>ceil(-3.1) = -3</code> .
<code>fix(x)</code>	Rounds x to the nearest integer towards zero: <code>fix(3.1) = 3</code> and <code>fix(-3.1) = -3</code> .
<code>floor(x)</code>	Rounds x to the nearest integer towards minus infinity: <code>floor(3.1) = 3</code> and <code>floor(-3.1) = -4</code> .
<code>round(x)</code>	Rounds x to the nearest integer.
String Conversion Functions	
<code>char(x)</code>	Converts a matrix of numbers into a character string. For ASCII characters the matrix should contain numbers ≤ 127 .
<code>double(x)</code>	Converts a character string into a matrix of numbers.
<code>int2str(x)</code>	Converts x into a character string representing the number as an integer.
<code>num2str(x)</code>	Converts x into a character string representing the number with a decimal point.
<code>str2num(s)</code>	Converts character string s into a number.

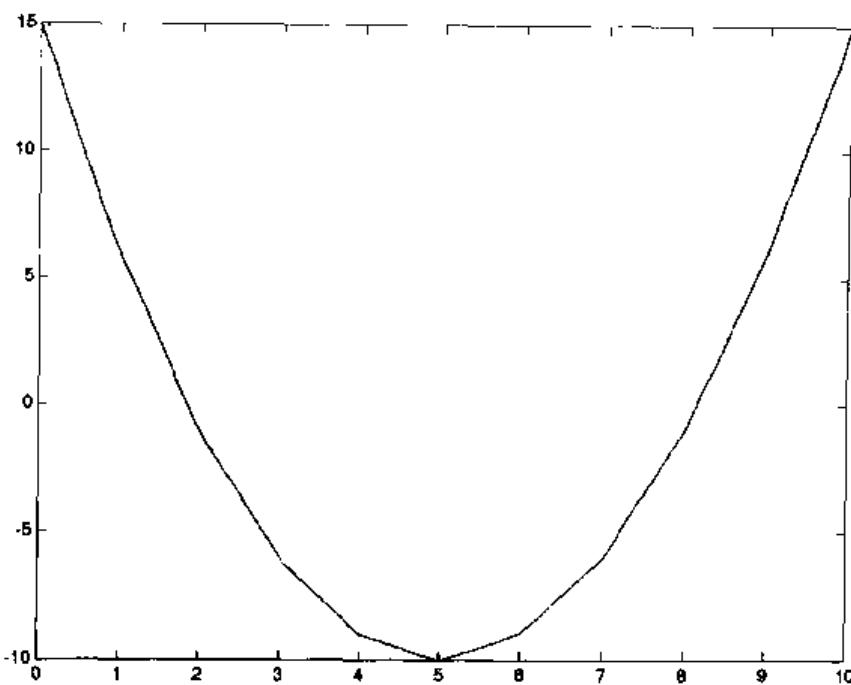


Figure 2.4 Plot of $y = x^2 - 10x + 15$ from 0 to 10.

When the `plot` function is executed, MATLAB opens a Figure Window and displays the plot in that window. The plot produced by these statements is shown in Figure 2.4.

2.11.1 Using Simple xy Plots

As we saw above, plotting is *very* easy in MATLAB. Any pair of vectors can be plotted versus each other as long as both vectors have the same length. However, the result is not a finished product, since there are no titles, axis labels, or grid lines on the plot.

Titles and axis labels can be added to a plot with the `title`, `xlabel`, and `ylabel` functions. Each function is called with a string containing the title or label to be applied to the plot. Grid lines can be added or removed from the plot with the `grid` command: `grid on` turns on grid lines, and `grid off` turns off grid lines. For example, the following statements generate a plot of the function $y = x^2 - 10x + 15$ with titles, labels, and gridlines. The resulting plot is shown in Figure 2.5.

```

x = 0:1:10;
y = x.^2 - 10.*x + 15;
plot(x,y);
title ('Plot of y = x.^2 - 10.*x + 15');
xlabel ('x');
ylabel ('y');
grid on;

```

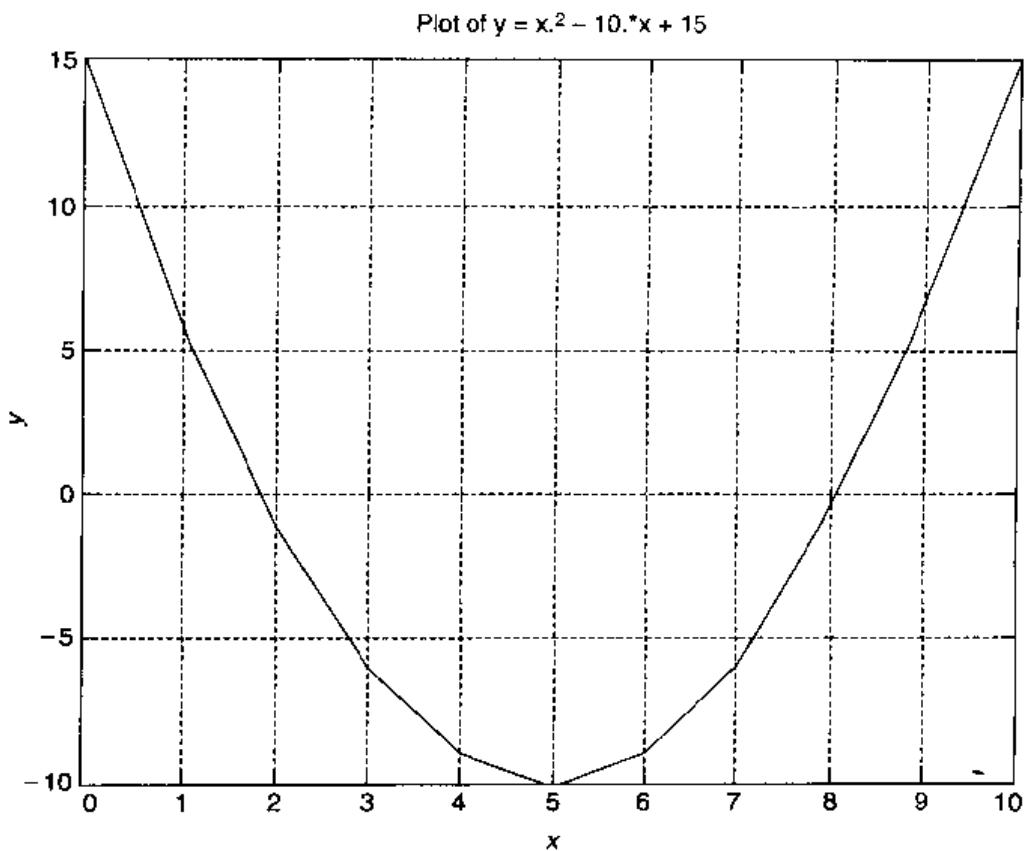


Figure 2.5 Plot of $y = x^2 - 10x + 15$ with a title, axis labels, and gridlines.

2.11.2 Printing a Plot

Once created, a plot can be printed on a printer with the `print` command, by clicking on the “print” icon in the Figure Window, or by selecting the “File/Print” menu option in the Figure Window.

The form of the `print` command is:

```
print <options> <filename>
```

If no filename is included, this command prints a copy of the current figure on the system printer. If a filename is specified, the command prints a copy of the current figure to the specified file.

There are many different options that specify the format of the output sent to a file or printer. One very important option is `-dtiff`. This option specifies that the output will be an image in tagged image file format (TIFF). Since this format can be imported into all of the important word processors on PC, Mac, and UNIX

platforms, it is a great way to include MATLAB plots in a document. The following command will create a TIFF image of the current figure and store it in a file called `my_image.tif`:

```
print -dtiff my_image.tif
```

It is also possible to create TIFF images by selecting the “File/Export” menu option in the Figure Window.

2.11.3 Multiple Plots

It is possible to plot multiple functions on the same graph simply by including more than one set of (x,y) values in the `plot` function. For example, suppose that we wanted to plot the function $f(x) = \sin 2x$ and its derivative on the same plot. The derivative of $f(x) = \sin 2x$ is

$$\frac{d}{dt} \sin 2x = 2 \cos 2x \quad (2.4)$$

To plot both functions on the same axes, we must generate a set of x values and the corresponding y values for each function. Then to plot the functions, we would simply list both sets of (x,y) values in the `plot` function as shown below.

```
x = 0:pi/100:2*pi;
y1 = sin(2*x);
y2 = 2*cos(2*x);
plot(x,y1,x,y2);
```

The resulting plot is shown in Figure 2.6.

2.11.4 Line Color, Line Style, Marker Style, and Legends

MATLAB allows a programmer to select the color of a line to be plotted, the style of the line to be plotted, and the type of marker to be used for data points on the line. These traits can be selected using an attribute character string after the x and y vectors in the `plot` function.

The attribute character string can have up to three components, with the first component specifying the color of the line, the second component specifying the style of the marker, and the last component specifying the style of the line. The characters for various colors, markers, and line styles are shown in Table 2.9.

The attribute characters can be mixed in any combination, and more than one attribute string can be specified if more than one pair of (x,y) vectors are included in a single `plot` function call. For example, the following statements will plot the function $y = x^2 - 10x + 15$ with a dashed red line and include the actual data points as blue circles.

```
x = 0:1:10;
y = x.^2 - 10.*x + 15;
plot(x,y,'r--',x,y,'bo');
```

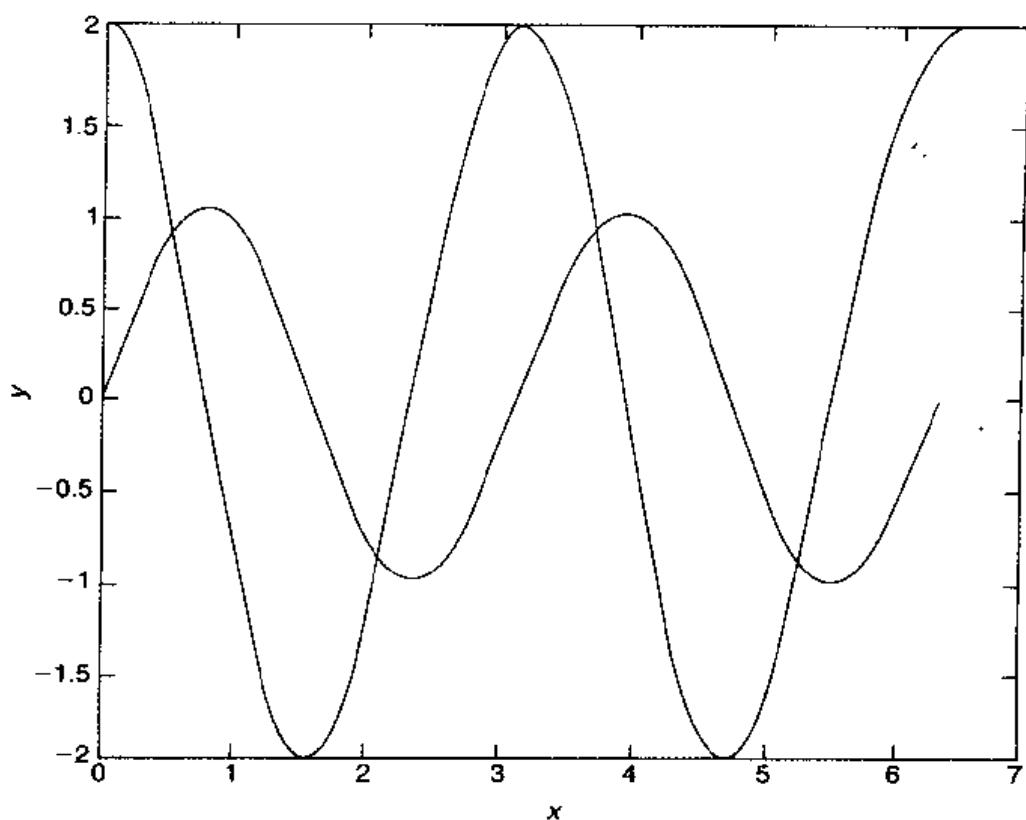


Figure 2.6 Plot of $f(x) = \sin 2x$ and $f(x) = 2 \cos 2x$ on the same axes.

Table 2.9 Plot Colors, Marker Styles, and Line Styles

Color		Marker Style		Line Style	
y	yellow	.	point	-	solid
m	magenta	o	circle	:	dotted
c	cyan	x	x-mark	-.	dash-dot
r	red	+	plus	---	dashed
g	green	*	star	<none>	no line
b	blue	s	square		
w	white	d	diamond		
k	black	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		
		<none>	no marker		

Legends can be created with the `legend` function. The basic form of this function is

```
legend('string1','string2',..., pos)
```

where `string1`, `string2`, and so forth, are the labels associated with the lines plotted, and `pos` is an integer specifying where to place the legend. The possible values for `pos` are given in Table 2.10. The command `legend off` will remove an existing legend.

An example of a complete plot is shown in Figure 2.7, and the statements to produce that plot are shown below. They plot the function $f(x) = \sin 2x$ and its derivative on the same axes, with a solid black line for $f(x)$ and a dashed red line for its derivative. The plot includes a title, axis labels, a legend, and grid lines.

```
x = 0:pi/100:2*pi;
y1 = sin(2*x);
y2 = 2*cos(2*x);
plot(x,y1,'k-',x,y2,'b--');
title ('Plot of f(x) = sin(2x) and its derivative');
xlabel ('x');
ylabel ('y');
legend ('f(x)', 'd/dx f(x)')
grid on;
```

2.11.5 Logarithmic Scales

It is possible to plot data on logarithmic scales as well as linear scales. There are four possible combinations of linear and logarithmic scales on the x and y axes, and each combination is produced by a separate function.

1. The `plot` function plots both x and y data on linear axes.
2. The `semilogx` function plots x data on logarithmic axes and y data on linear axes.

Table 2.10 Values of pos in the legend Command

Value	Meaning
0	Automatic “best” placement (least conflict with data)
1	Upper right-hand corner (default)
2	Upper left-hand corner
3	Lower left-hand corner
4	Lower right-hand corner
-1	To the right of the plot

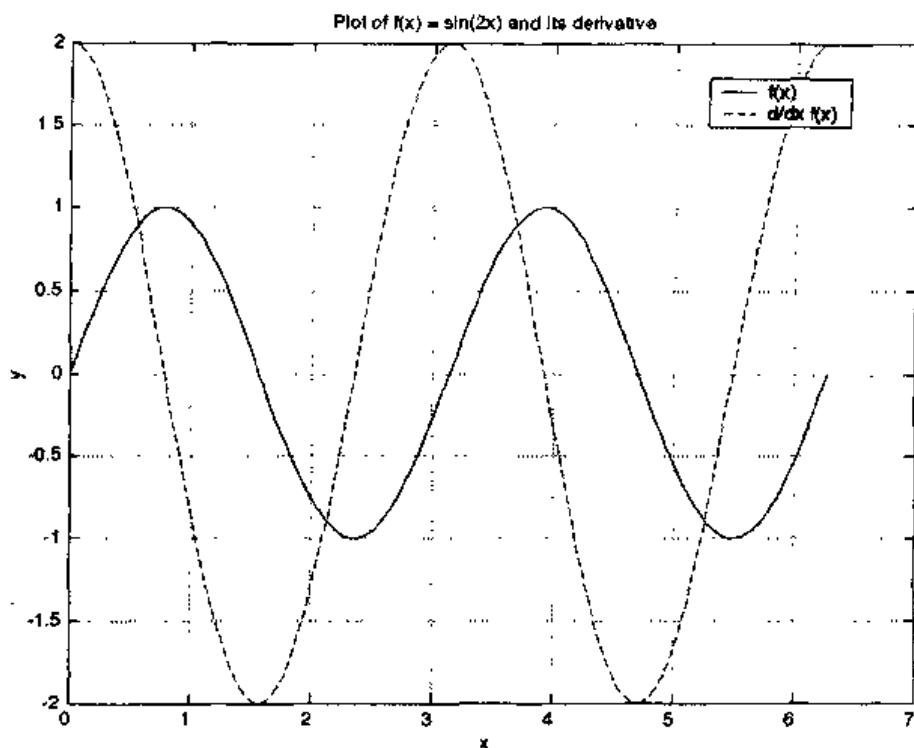


Figure 2.7 A complete plot with title, axis labels, legend, grid, and multiple line styles.

3. The `semilogx` function plots x data on linear axes and y data on logarithmic axes.
4. The `loglog` function plots both x and y data on logarithmic axes.

All of these functions have identical calling sequences—the only difference is the type of axis used to plot the data. Examples of each plot are shown in Figure 2.8.

2.12 Examples

The following examples illustrate problem-solving with MATLAB.



Example 2.3—Temperature Conversion

Design a MATLAB program that reads an input temperature in degrees Fahrenheit, converts it to an absolute temperature in kelvins, and writes out the result.

Solution

The relationship between temperature in degrees Fahrenheit ($^{\circ}\text{F}$) and temperature in kelvins (K) can be found in any physics textbook. It is

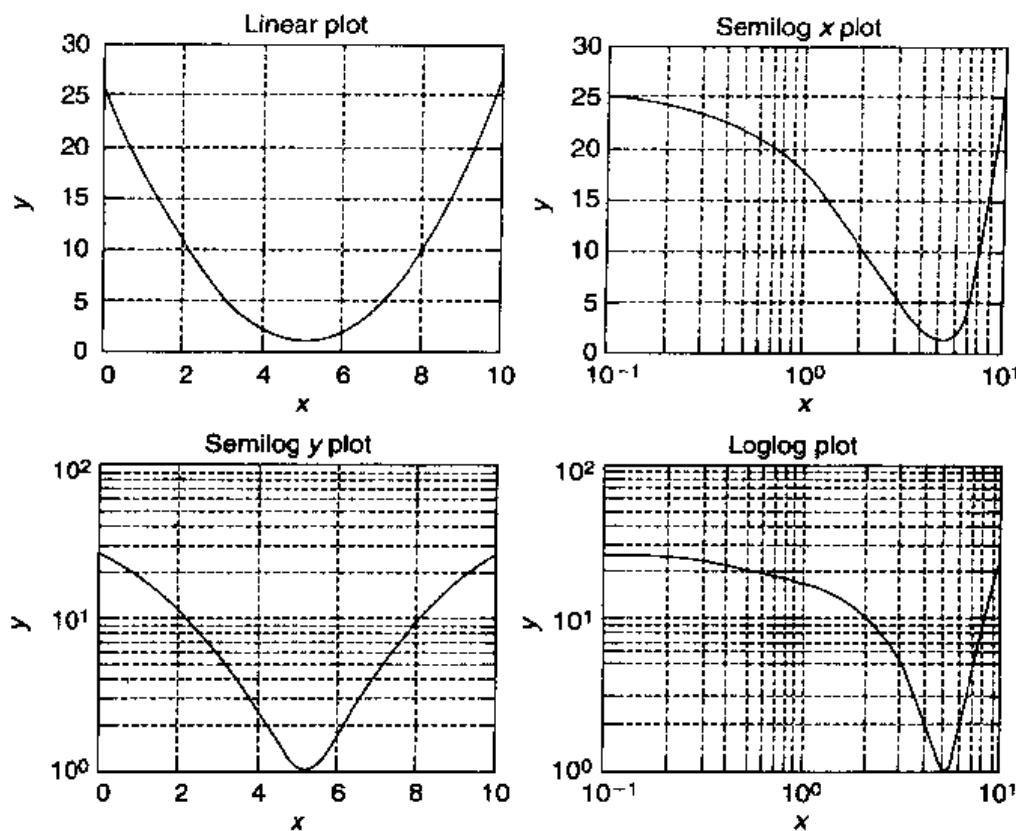


Figure 2.8 Comparison of linear, semilog x , semilog y , and loglog plots.

$$T \text{ (in Kelvins)} = \left(\frac{5}{9} T \text{ (in } ^\circ\text{F)} - 32.0 \right) + 273.15 \quad (2.5)$$

The physics books also give us sample values on both temperature scales, which we can use to check the operation of our program. Two such values are:

The boiling point of water	212° F	373.15 K
The sublimation point of dry ice	-110° F	194.26 K

Our program must perform the following steps.

1. Prompt the user to enter an input temperature in $^\circ\text{F}$.
2. Read the input temperature.
3. Calculate the temperature in kelvins from Equation (2.5).
4. Write out the result, and stop.

We will use function `input` to get the temperature in degrees Fahrenheit and function `fprintf` to print the answer. The resulting program follows.

```
% Script file: temp_conversion
%
% Purpose:
```

```

% To convert an input temperature from degrees Fahrenheit to
% an output temperature in kelvins.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   =====    ======          =====
%   12/01/98  S. J. Chapman  Original code
%
% Define variables:
%   temp_f . -- Temperature in degrees Fahrenheit
%   temp_k   -- Temperature in kelvins

% Prompt the user for the input temperature.
temp_f = input('Enter the temperature in degrees Fahrenheit: ');

% Convert to kelvins.
temp_k = (5/9) * (temp_f - 32) + 273.15;

% Write out the result.
fprintf('%6.2f degrees Fahrenheit = %6.2f kelvins.\n', ...
        temp_f,temp_k);

```

To test the completed program, we will run it with the known input values given above. Note that user inputs appear in bold face below.

```

>> temp_conversion
Enter the temperature in degrees Fahrenheit: 212
212.00 degrees Fahrenheit = 373.15 kelvins.
>> temp_conversion
Enter the temperature in degrees Fahrenheit: -110
-110.00 degrees Fahrenheit = 194.26 kelvins.

```

The results of the program match the values from the physics book.

In this program, we echoed the input values and printed the output values together with their units. The results of this program only make sense if the units (degrees Fahrenheit and kelvins) are included together with their values. As a general rule, the units associated with any input value should always be printed along with the prompt that requests the value, and the units associated with any output value should always be printed along with that value.

Good Programming Practice

Always include the appropriate units with any values that you read or write in a program.

The previous example program exhibits many of the good programming practices that we have described in this chapter. It includes a data dictionary defining the meanings of all the variables in the program. It also uses descriptive variable names, and appropriate units are attached to all printed values.

Example 2.4—Electrical Engineering: Maximum Power Transfer to a Load

Figure 2.9 shows a voltage source $V = 120$ V with an internal resistance R_S of 50Ω supplying a load of resistance R_L . Find the value of load resistance R_L that will result in the maximum possible power being supplied by the source to the load. How much power will be supplied in this case? Also, plot the power supplied to the load as a function of the load resistance R_L .

Solution

In this program, we need to vary the load resistance R_L and compute the power supplied to the load at each value of R_L . The power supplied to the load resistance is given by the equation

$$P_L = I^2 R_L \quad (2.6)$$

where I is the current supplied to the load. The current supplied to the load can be calculated by Ohm's law.

$$I = \frac{V}{R_{TOT}} = \frac{V}{R_S + R_L} \quad (2.7)$$

The program must perform the following steps.

1. Create an array of possible values for the load resistance R_L . The array will vary R_L from 1Ω to 100Ω in 1Ω steps.
2. Calculate the current for each value of R_L .

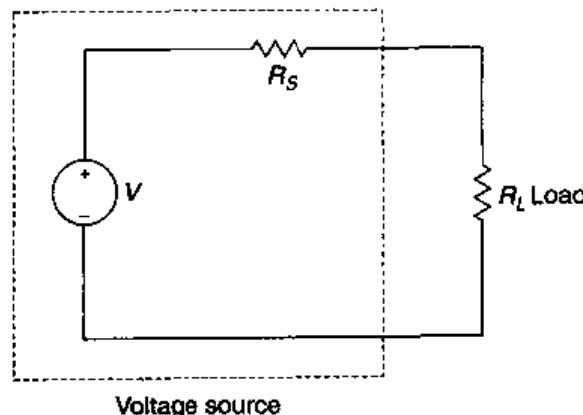


Figure 2.9 A voltage source with a voltage V and an internal resistance R_S supplying a load of resistance R_L .

3. Calculate the power supplied to the load for each value of R_L .
4. Plot the power supplied to the load for each value of R_L , and determine the value of load resistance resulting in the maximum power.

The final MATLAB program is shown below.

```
% Script file: calc_power.m
%
% Purpose:
%   To calculate and plot the power supplied to a load as
%   a function of the load resistance.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   ----      ======      =====
%   12/01/98    S. J. Chapman  Original code
%
% Define variables:
%   amps      -- Current flow to load (amps)
%   pl        -- Power supplied to load (watts)
%   rl        -- Resistance of the load (ohms)
%   rs        -- Internal resistance of the power source (ohms)
%   volts     -- Voltage of the power source (volts)

% Set the values of source voltage and internal resistance
volts = 120;
rs = 50;

% Create an array of load resistances
rl = 1:1:100;

% Calculate the current flow for each resistance
amps = volts ./ ( rs + rl );

% Calculate the power supplied to the load
pl = (amps .^ 2) .* rl;

% Plot the power versus load resistance
plot(rl,pl);
title('Plot of power versus load resistance');
xlabel('Load resistance (ohms)');
ylabel('Power (watts)');
grid on;
```

When this program is executed, the resulting plot is shown in Figure 2.10 on the following page. From this plot, we can see that the maximum power is supplied to the load when the load's resistance is 50Ω . The power supplied to the load at this resistance is 72 watts.

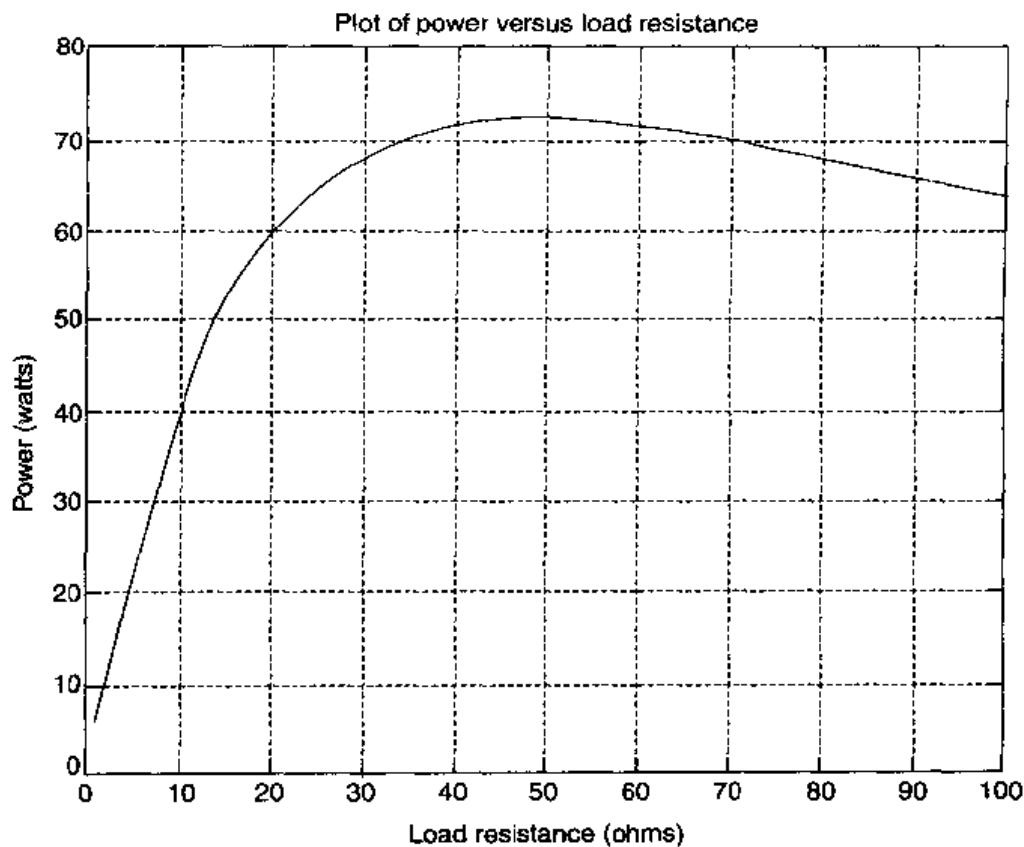


Figure 2.10 Plot of power supplied to load versus load resistance.

Note the use of the array operators `.*`, `.^`, and `./` in this example. These operators cause the arrays `amps` and `p1` to be calculated on an element-by-element basis.

Example 2.5—Carbon 14 Dating

A radioactive isotope of an element is a form of the element that is not stable. Instead, it spontaneously decays into another element over a period of time. Radioactive decay is an exponential process. If Q_0 is the initial quantity of a radioactive substance at time $t = 0$, then the amount of that substance which will be present at any time t in the future is given by

$$Q(t) = Q_0 e^{-\lambda t} \quad (2.8)$$

where λ is the radioactive decay constant.

Because radioactive decay occurs at a known rate, it can be used as a clock to measure the time since the decay started. If we know the initial amount of the radioactive material Q_0 present in a sample, and the amount of the material Q left

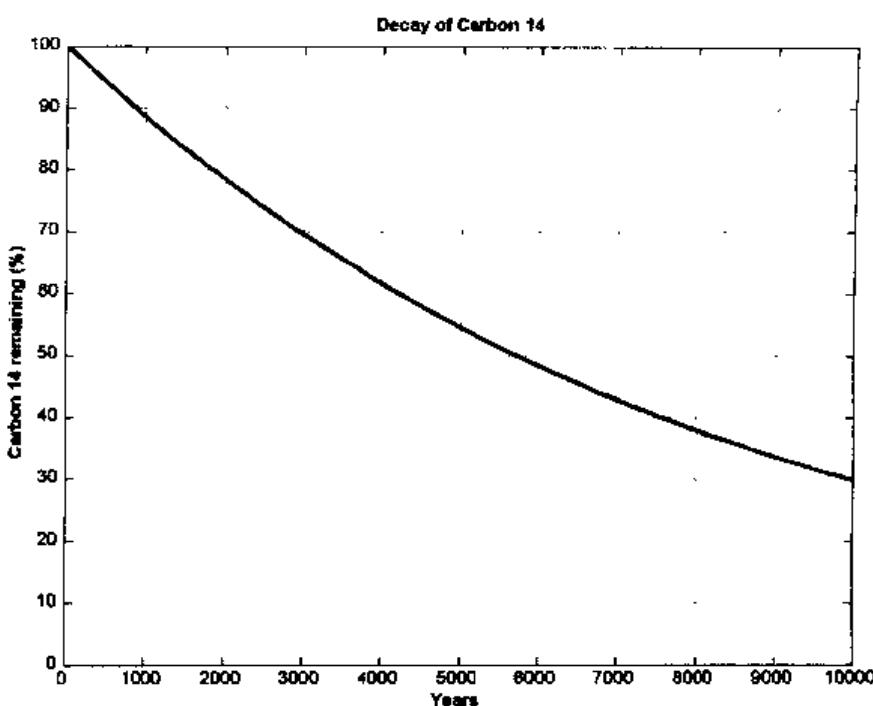


Figure 2.11 The radioactive decay of carbon 14 as a function of time. Notice that 50 percent of the original carbon 14 is left after 5730 years have elapsed.

at the current time, we can solve for t in Equation (2.8) to determine how long the decay has been going on. The resulting equation is

$$t_{\text{decay}} = -\frac{1}{\lambda} \log_e \frac{Q}{Q_0} \quad (2.9)$$

Equation (2.9) has practical applications in many areas of science. For example, archaeologists use a radioactive clock based on carbon 14 to determine the time that has passed since a once-living thing died. Carbon 14 is continually taken into the body while a plant or animal is living, so the amount of it present in the body at the time of death is assumed to be known. The decay constant λ of carbon 14 is well known to be 0.00012097/year, so if the amount of carbon 14 remaining now can be accurately measured, then Equation (2.9) can be used to determine how long ago the living thing died. The amount of carbon 14 remaining as a function of time is shown in Figure 2.11.

Write a program that reads the percentage of carbon 14 remaining in a sample, calculates the age of the sample from it, and prints out the result with proper units.

Solution

Our program must perform the following steps.

- Prompt the user to enter the percentage of carbon 14 remaining in the sample.

2. Read in the percentage.
3. Convert the percentage into the fraction $\frac{Q}{Q_0}$.
4. Calculate the age of the sample in years using Equation (2.9).
5. Write out the result, and stop.

The resulting code is shown below.

```
% Script file: c14_date.m
%
% Purpose:
%   To calculate the age of an organic sample from the percentage
%   of the original carbon 14 remaining in the sample.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   =====    ======      =====
%   12/02/98  S. J. Chapman  Original code
%
% Define variables:
%   age        -- The age of the sample in years.
%   lamda      -- The radioactive decay constant for carbon-14,
%                 in units of 1/years.
%   percent    -- The percentage of carbon 14 remaining at the time
%                 of the measurement.
%   ratio      -- The ratio of the carbon 14 remaining at the time
%                 of the measurement to the original amount of
%                 carbon 14.

% Set decay constant for carbon-14
lamda = 0.00012097;

% Prompt the user for the percentage of C-14 remaining.
percent = input('Enter the percentage of carbon 14 remaining:\n');

% Perform calculations
ratio = percent / 100; % Convert to fractional ratio
age = (-1.0 / lamda) * log(ratio); % Get age in years

% Tell the user about the age of the sample.
string = ['The age of the sample is ' num2str(age) ' years.'];
disp(string);
```

To test the completed program, we will calculate the time it takes for half of the carbon 14 to disappear. This time is known as the *half-life* of carbon 14.

```
> c14_date
Enter the percentage of carbon 14 remaining:
50
The age of the sample is 5729.9097 years.
```

The *CRC Handbook of Chemistry and Physics* states that the half-life of carbon 14 is 5730 years, so output of the program agrees with the reference book.

2.13 Debugging MATLAB Programs

There is an old saying that the only sure things in life are death and taxes. We can add one more certainty to that list: if you write a program of any significant size, it will not work the first time you try it! Errors in programs are known as **bugs**, and the process of locating and eliminating them is known as **debugging**. Given that we have written a program and it is not working, how do we debug it?

Three types of errors are found in MATLAB programs. The first type of error is a **syntax error**. Syntax errors are errors in the MATLAB statement itself, such as spelling errors or punctuation errors. These errors are detected by the MATLAB compiler the first time that an M-file is executed. For example, the statement

```
x = (y + 3) / 2;
```

contains a syntax error because it has unbalanced parentheses. If this statement appears in an M-file named `test.m`, the following message appears when `test` is executed.

```
> test
??? x = (y + 3) / 2
                                |
Missing operator, comma, or semi-colon.

Error in ==> d:\book\matlab\chap2\test.m
On line 2 ==>
```

The second type of error is the **run-time error**. A run-time error occurs when an illegal mathematical operation is attempted during program execution (for example, attempting to divide by 0). These errors cause the program to return `Inf` or `Nan`, which are then used in further calculations. The results of a program that contains calculations using `Inf` or `Nan` are usually invalid.

The third type of error is a **logical error**. Logical errors occur when the program compiles and runs successfully but produces the wrong answer.

The most common mistakes made during programming are *typographical errors*. Some typographical errors create invalid MATLAB statements. These errors produce syntax errors that are caught by the compiler. Other typographical errors occur in variable names. For example, the letters in some variable names might have been transposed, or an incorrect letter might be typed. The result will be a new variable, and MATLAB simply creates the new variable the first time that it is referenced. MATLAB cannot detect this type of error. Typographical errors can also produce logical errors. For example, if variables `vel1` and `vel2` are both

used for velocities in the program, then one of them might be inadvertently used instead of the other one at some point. You must check for that sort of error by manually inspecting the code.

Sometimes a program will start to execute, but run-time errors or logical errors occur during execution. In this case, there is either something wrong with the input data or something wrong with the logical structure of the program. The first step in locating this sort of bug should be to *check the input data to the program*. Either remove semicolons from input statements or add extra output statements to verify that the input values are what you expect them to be.

If the variable names seem to be correct and the input data is correct, then you are probably dealing with a logical error. You should check each of your assignment statements.

1. If an assignment statement is very long, break it into several smaller assignment statements. Smaller statements are easier to verify.
2. Check the placement of parentheses in your assignment statements. It is a very common error to have the operations in an assignment statement evaluated in the wrong order. If you have any doubts as to the order in which the variables are being evaluated, add extra sets of parentheses to make your intentions clear.
3. Make sure that you have initialized all of your variables properly.
4. Be sure that any functions you use are in the correct units. For example, the input to trigonometric functions must be in units of radians, not degrees.

If you are still getting the wrong answer, add output statements at various points in your program to see the results of intermediate calculations. If you can locate the point where the calculations go bad, then you know just where to look for the problem, which is 95% of the battle.

If you still cannot find the problem after taking all of these steps, explain what you are doing to another student or to your instructor, and let them look at the code. It is very common for a person to see just what he or she expects to see when they look at their own code. Another person can often quickly spot an error that you have overlooked time after time.

To reduce your debugging effort, make sure that during your program design you:

1. Initialize all variables.
2. Use parentheses to make the functions of assignment statements clear.

MATLAB includes a special debugging tool called a *symbolic debugger*. A symbolic debugger is a tool that allows you to walk through the execution of your

program one statement at a time and to examine the values of any variables at each step along the way. Symbolic debuggers allow you to see all of the intermediate results without having to insert a lot of output statements into your code. We will learn how to use MATLAB's symbolic debugger in Chapter 3.

2.14 Summary

In this chapter, we introduced two data types: `double` and `char`. We also introduced assignment statements, arithmetic calculations, intrinsic functions, input/output statements, and data files.

The order in which MATLAB expressions are evaluated follows a fixed hierarchy, with operations at a higher level evaluated before operations at lower levels. The hierarchy of operations is summarized in Table 2.11.

The MATLAB language includes an extremely large number of built-in functions to help us solve problems. This list of functions is *much* richer than the list of functions found in other languages like Fortran or C, and it includes device-independent plotting capabilities. A few of the common intrinsic functions are summarized in Table 2.8, and many others will be introduced throughout the remainder of the book. A complete list of all MATLAB functions is available through the on-line Help Browser.

2.14.1 Summary of Good Programming Practice

Every MATLAB program should be designed so that another person who is familiar with MATLAB can easily understand it. This is very important since a good program can be used for a long time. Over that time, conditions will change, and the program will need to be modified to reflect the changes. The program modifications may be done by someone other than the original programmer. The programmer making the modifications must understand the original program well before attempting to change it.

Table 2.11 Hierarchy of Operations

Precedence	Operation
1	The contents of all parentheses are evaluated, starting from the innermost parentheses and working outward.
2	All exponentials are evaluated, working from left to right.
3	All multiplications and divisions are evaluated, working from left to right.
4	All additions and subtractions are evaluated, working from left to right.

It is much harder to design clear, understandable, and maintainable programs than it is to simply write programs. To do so, a programmer must develop the discipline to properly document his or her work. In addition, the programmer must be careful to avoid known pitfalls along the path to good programs. The following guidelines will help you to develop good programs.

1. Use meaningful variable names whenever possible. Use names that can be understood at a glance, like day, month, and year.
2. Create a data dictionary for each program to make program maintenance easier.
3. Use only lowercase letters in variable names, so that there will not be errors due to capitalization differences in different occurrences of a variable name.
4. Use a semicolon at the end of all MATLAB assignment statements to suppress echoing of assigned values in the Command Window. If you need to examine the results of a statement during program debugging, you may remove the semicolon from that statement only.
5. If data must be exchanged between MATLAB and other programs, save the MATLAB data in ASCII format. If the data will only be used in MATLAB, save the data in MAT-file format.
6. Save ASCII data files with a “dat” file extent to distinguish them from MAT-files, which have a “mat” file extent.
7. Use parentheses as necessary to make your equations clear and easy to understand.
8. Always include the appropriate units with any values that you read or write in a program.

2.14.2 MATLAB Summary

The following summary lists all of the MATLAB special symbols, commands, and functions described in this chapter, along with a brief description of each one.

Special Symbols

[]	Array constructor
()	Forms subscripts
'	Marks the limits of a character string
,	(comma) Separates subscripts or matrix elements
:	<ol style="list-style-type: none"> 1. Suppresses echoing in Command Window 2. Separates matrix rows 3. Separates assignment statements on a line
%	Marks the beginning of a comment
:	Colon operator, used to create shorthand lists
+	Array and matrix addition
-	Array and matrix subtraction
.*	Array multiplication

*	Matrix multiplication
. /	Array right division
\ /	Array left division
/	Matrix right division
\	Matrix left division
. ^	Array exponentiation
'	Transpose operator

Commands and Functions

...	Continues a MATLAB statement on the following line.
abs(x)	Calculates the absolute value of x .
ans	Default variable used to store the result of expressions not assigned to another variable.
acos(x)	Calculates the inverse cosine of x . The resulting angle is in radians between 0 and π .
asin(x)	Calculates the inverse sine of x . The resulting angle is in radians between $-\pi/2$ and $\pi/2$.
atan(x)	Calculates the inverse tangent of x . The resulting angle is in radians between $-\pi/2$ and $\pi/2$.
atan2(y,x)	Calculates the inverse tangent of y/x , valid over the entire circle. The resulting angle is in radians between $-\pi$ and π .
ceil(x)	Rounds x to the nearest integer toward positive infinity: $\text{ceil}(3.1) = 4$ and $\text{ceil}(-3.1) = -3$.
char	Converts a matrix of numbers into a character string. For ASCII characters the matrix should contain numbers ≤ 127 .
clock	Current time.
cos(x)	Calculates cosine of x , where x is in radians.
date	Current date.
disp	Displays data in Command Window.
doc	Open HTML Help Desk directly at a particular function description.
double	Converts a character string into a matrix of numbers.
eps	Represents machine precision.
exp(x)	Calculates e^x .
eye(n,m)	Generates an identity matrix.
fix(x)	Rounds x to the nearest integer towards zero: $\text{fix}(3.1) = 3$ and $\text{fix}(-3.1) = -3$.
floor(x)	Rounds x to the nearest integer towards minus infinity: $\text{floor}(3.1) = 3$ and $\text{floor}(-3.1) = -4$.
format +	Print + and - signs only.
format bank	Print in "dollars and cents" format.

<code>format compact</code>	Suppress extra linefeeds in output.
<code>format hex</code>	Print hexadecimal display of bits.
<code>format long</code>	Print with 14 digits after the decimal.
<code>format long e</code>	Print with 15 digits plus exponent.
<code>format long g</code>	Print with 15 digits with or without exponent.
<code>format loose</code>	Print with extra linefeeds in output.
<code>format rat</code>	Print as an approximate ratio of small integers.
<code>format short</code>	Print with 4 digits after the decimal.
<code>format short e</code>	Print with 5 digits plus exponent.
<code>format short g</code>	Print with 5 digits with or without exponent.
<code>fprintf</code>	Print formatted information.
<code>grid</code>	Add / remove a grid from a plot.
<code>i</code>	$\sqrt{-1}$
<code>Inf</code>	Represents machine infinity (∞).
<code>input</code>	Writes a prompt and reads a value from the keyboard.
<code>int2str</code>	Converts x into an integer character string.
<code>j</code>	$\sqrt{-1}$
<code>legend</code>	Adds a legend to a plot.
<code>length(arr)</code>	Returns the length of a vector, or the longest dimension of a 2-D array.
<code>load</code>	Load data from a file.
<code>log(x)</code>	Calculates the natural logarithm of x .
<code>loglog</code>	Generates a log-log plot.
<code>lookfor</code>	Look for a matching term in the one-line MATLAB function descriptions.
<code>max(x)</code>	Returns the maximum value in vector x , and optionally the location of that value.
<code>min(x)</code>	Returns the minimum value in vector x , and optionally the location of that value.
<code>mod(n,m)</code>	Remainder or modulo function.
<code>NaN</code>	Represents not-a-number.
<code>num2str(x)</code>	Converts x into a character string.
<code>ones(n,m)</code>	Generates an array of ones.
<code>pi</code>	Represents the number π .
<code>plot</code>	Generates a linear xy plot.
<code>print</code>	Prints a Figure Window.
<code>round(x)</code>	Rounds x to the nearest integer.
<code>save</code>	Saves data from workspace into a file.
<code>semilogx</code>	Generates a log-linear plot.
<code>semilogy</code>	Generates a linear-log plot.
<code>sin(x)</code>	Calculates sine of x , where x is in radians.
<code>size</code>	Get number of rows and columns in an array.
<code>sqrt</code>	Calculates the square root of a number.
<code>str2num</code>	Converts a character string into a number.

<code>tan(x)</code>	Calculates tangent of x , where x is in radians.
<code>title</code>	Adds a title to a plot.
<code>zeros</code>	Generate an array of zeros.

2.15 Exercises

- 2.1** Answer the following questions for the array shown below.

$$\text{array1} = \begin{bmatrix} 1.1 & 0.0 & 2.1 & -3.5 & 6.0 \\ 0.0 & 1.1 & -6.6 & 2.8 & 3.4 \\ 2.1 & 0.1 & 0.3 & -0.4 & 1.3 \\ -1.4 & 5.1 & 0.0 & 1.1 & 0.0 \end{bmatrix}$$

- a. What is the size of `array1`?
- b. What is the value of `array1(4,1)`?
- c. What is the size and value of `array1(:,1:2)`?
- d. What is the size and value of `array1([1 3],end)`?

- 2.2** Are the following MATLAB variable names legal or illegal? Why?

- a. `dog1`
- b. `1dog`
- c. `Do_you_know_the_way_to_san_jose`
- d. `_help`
- e. `What's_up?`

- 2.3** Determine the size and contents of the following arrays. Note that the later arrays may depend on the definitions of arrays defined earlier in this exercise.

- a. `a = 1:2:5;`
- b. `b = [a' a' a'];`
- c. `c = b(1:2:3,1:2:3);`
- d. `d = a + b(2,:);`
- e. `w = [zeros(1,3) ones(3,1)' 3:5'];`
- f. `b([1 3],2) = b([3 1],2);`

- 2.4** Assume that array `array1` is defined as shown, and determine the contents of the following sub-arrays.

$$\text{array1} = \begin{bmatrix} 1.1 & 0.0 & 2.1 & -3.5 & 6.0 \\ 0.0 & 1.1 & -6.6 & 2.8 & 3.4 \\ 2.1 & 0.1 & 0.3 & -0.4 & 1.3 \\ -1.4 & 5.1 & 0.0 & 1.1 & 0.0 \end{bmatrix}$$

- a. `array1(3,:)`
- b. `array1(:,3)`
- c. `array1(1:2:3,[3 3 4])`
- d. `array1([1 1],:)`

- 2.5** Assume that `value` has been initialized to 10π , and determine what is printed out by each of the following statements.

```
disp(['value = ' num2str(value)]);
disp(['value = ' int2str(value)]);
fprintf('value = %e\n',value);
fprintf('value = %f\n',value);
fprintf('value = %g\n',value);
fprintf('value = %12.4f\n',value);
```

- 2.6** Assume that `a`, `b`, `c`, and `d` are defined as follows, and calculate the results of the following operations if they are legal. If an operation is illegal, explain why it is illegal.

$$\mathbf{a} = \begin{bmatrix} 2 & -2 \\ -1 & 2 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 & -1 \\ 0 & 2 \end{bmatrix}$$

$$\mathbf{c} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \quad \mathbf{d} = \text{eye}(2)$$

- a. `result = a + b;`
- b. `result = a * d;`
- c. `result = a .* d;`
- d. `result = a * c;`
- e. `result = a .* c;`
- f. `result = a \ b;`
- g. `result = a .\ b;`
- h. `result = a .^ b;`

- 2.7** Evaluate each of the following expressions.

- a. `11 / 5 + 6`
- b. `(11 / 5) + 6`
- c. `11 / (5 + 6)`
- d. `3 ^ 2 ^ 3`
- e. `3 ^ (2 ^ 3)`
- f. `(3 ^ 2) ^ 3`
- g. `round(-11/5) + 6`
- h. `ceil(-11/5) + 6`
- i. `floor(-11/5) + 6`

- 2.8** Use MATLAB to evaluate each of the following expressions.

- a. $(3 - 5i)(-4 + 6i)$
- b. $\cos^{-1}(1.2)$

- 2.9** Solve the following system of simultaneous equations for x .

$$\begin{aligned}
 -2.0x_1 + 5.0x_2 + 1.0x_3 + 3.0x_4 + 4.0x_5 - 1.0x_6 &= 0.0 \\
 2.0x_1 - 1.0x_2 - 5.0x_3 - 2.0x_4 + 6.0x_5 + 4.0x_6 &= 1.0 \\
 -1.0x_1 + 6.0x_2 - 4.0x_3 - 5.0x_4 + 3.0x_5 - 1.0x_6 &= -6.0 \\
 4.0x_1 + 3.0x_2 - 6.0x_3 - 5.0x_4 - 2.0x_5 - 2.0x_6 &= 10.0 \\
 -3.0x_1 + 6.0x_2 + 4.0x_3 + 2.0x_4 - 6.0x_5 + 4.0x_6 &= -6.0 \\
 2.0x_1 + 4.0x_2 + 4.0x_3 + 4.0x_4 + 5.0x_5 - 4.0x_6 &= -2.0
 \end{aligned}$$

- 2.10 Position and Velocity of a Ball.** If a stationary ball is released at a height h_0 above the surface of the Earth with a vertical velocity v_0 , the position and velocity of the ball as a function of time will be given by the equations

$$h(t) = \frac{1}{2}gt^2 + v_0t + h_0 \quad (2.10)$$

$$v(t) = gt + v_0 \quad (2.11)$$

where g is the acceleration due to gravity (-9.81 m/s^2), h is the height above the surface of the Earth (assuming no air friction), and v is the vertical component of velocity. Write a MATLAB program that prompts a user for the initial height of the ball in meters and velocity of the ball in meters per second, and plots the height and velocity as a function of time. Be sure to include proper labels in your plots.

- 2.11** The distance between two points (x_1, y_1) and (x_2, y_2) on a Cartesian coordinate plane is given by the equation

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (2.12)$$

Write a program to calculate the distance between any two points (x_1, y_1) and (x_2, y_2) specified by the user. Use good programming practices in your program. Use the program to calculate the distance between the points $(2, 3)$ and $(8, -5)$.

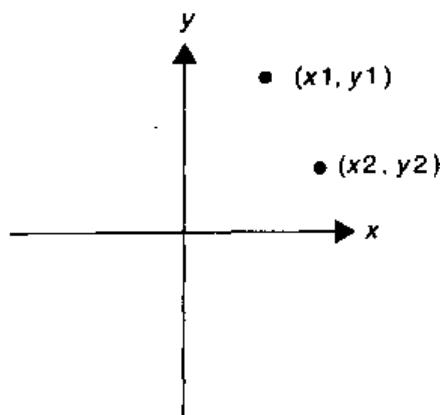


Figure 2.12 Distance between two points on a Cartesian plane.

- 2.12 Decibels.** Engineers often measure the ratio of two power measurements in *decibels*, or dB. The equation for the ratio of two power measurements in decibels is

$$\text{dB} = 10 \log_{10} \frac{P_2}{P_1} \quad (2.13)$$

where P_2 is the power level being measured, and P_1 is some reference power level.

- Assume that the reference power level P_1 is 1 milliwatt, and write a program that accepts an input power P_2 and converts it into dB with respect to the 1 mW reference level. (Engineers have a special unit for dB power levels with respect to a 1 mW reference: dBm.) Use good programming practices in your program.
- Write a program that creates a plot of power in watts versus power in dBm with respect to a 1 mW reference level. Create both a linear xy plot and a log-linear xy plot.

- 2.13 Hyperbolic cosine.** The hyperbolic cosine function is defined by the equation

$$\cosh x = \frac{e^x + e^{-x}}{2} \quad (2.14)$$

Write a program to calculate the hyperbolic cosine of a user-supplied value x . Use the program to calculate the hyperbolic cosine of 3.0. Compare the answer that your program produces to the answer produced by the MATLAB intrinsic function `cosh(x)`. Also, use MATLAB to plot the function `cosh(x)`. What is the smallest value that this function can have? At what value of x does it occur?

- 2.14 Energy Stored in a Spring.** The force required to compress a linear spring is given by the equation

$$F = kx \quad (2.15)$$

where F is the force in newtons and k is the spring constant in newtons per meter. The potential energy stored in the compressed spring is given by the equation

$$E = \frac{1}{2} kx^2 \quad (2.16)$$

where E is the energy in joules. The following information is available for four springs.

	Spring 1	Spring 2	Spring 3	Spring 4
Force (N)	20	24	22	20
Spring constant k (N/m)	500	600	700	800

Determine the compression of each spring, and the potential energy stored in each spring. Which spring has the most energy stored in it?

- 2.15 Radio Receiver.** A simplified version of the front end of an AM radio receiver is shown in Figure 2.13. This receiver consists of an *RLC* tuned circuit containing a resistor, capacitor, and an inductor connected in series. The *RLC* circuit is connected to an external antenna and ground as shown in the figure.

The tuned circuit allows the radio to select a specific station out of all the stations transmitting on the AM band. At the resonant frequency of the circuit, essentially all of the signal V_0 appearing at the antenna appears across the resistor, which represents the rest of the radio. In other words, the radio receives its strongest signal at the resonant frequency. The resonant frequency of the LC circuit is given by the equation

$$f_0 = \frac{1}{2\pi\sqrt{LC}} \quad (2.17)$$

where L is inductance in henrys (H) and C is capacitance in farads (F). Write a program that calculates the resonant frequency of this radio set given specific values of L and C . Test your program by calculating the frequency of the radio when $L = 0.1$ mH and $C = 0.25$ nF.

- 2.16 Radio Receiver.** The voltage across the resistive load in Figure 2.13 varies as a function of frequency according to Equation (2.18).

$$V_R = \frac{R}{\sqrt{R^2 + \left(\omega L - \frac{1}{\omega C}\right)^2}} V_0 \quad (2.18)$$

where $\omega = 2\pi f$ and f is the frequency in hertz. Assume that $L = 0.1$ mH, $C = 0.25$ nF, $R = 50 \Omega$, and $V_0 = 10$ mV.

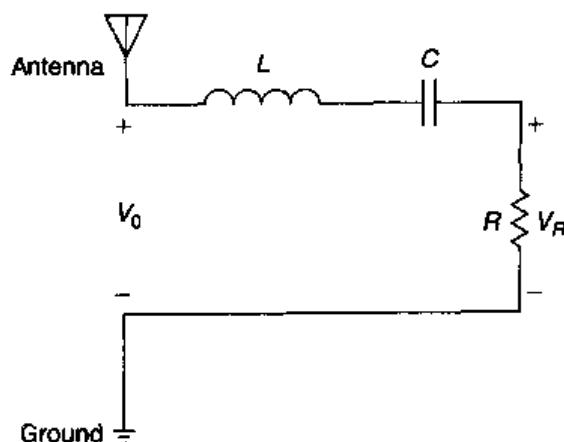


Figure 2.13 A simplified version of the front end of an AM radio receiver.

- a. Plot the voltage on the resistive load as a function of frequency. At what frequency does the voltage on the resistive load peak? What is the voltage on the load at this frequency? This frequency is called the resonant frequency f_o of the circuit.
- b. If the frequency is changed to 10% greater than the resonant frequency, what is the voltage on the load? How selective is this radio receiver?
- c. At what frequencies will the voltage on the load drop to half of the voltage at the resonant frequency?
- 2.17** Suppose two signals were received at the antenna of the radio receiver described in the previous problem. One signal has a strength of 1 V at a frequency of 1000 kHz, and the other signal has a strength of 1 V at 950 kHz. How much power will the first signal supply to the resistive load R ? How much power will the second signal supply to the resistive load R ? Express the ratio of the power supplied by signal 1 to the power supplied by signal 2 in decibels. How much is the second signal enhanced or suppressed compared to the first signal?
- 2.18 Aircraft Turning Radius.** An object moving in a circular path at a constant tangential velocity v is shown in Figure 2.14. The radial acceleration required for the object to move in the circular path is given by the equation

$$a = \frac{v^2}{r} \quad (2.19)$$

where a is the centripetal acceleration of the object in m/s^2 , v is the tangential velocity of the object in m/s , and r is the turning radius in meters. Suppose that the object is an aircraft, and answer the following questions about it:

- a. Suppose that the aircraft is moving at Mach 0.85, or 85% of the speed of sound. If the centripetal acceleration is 2 g, what is the turning radius of the aircraft? (Note: For this problem, you may assume that Mach 1 is equal to 340 m/s, and that 1 g = 9.81 m/s^2 .)
- b. Suppose that the speed of the aircraft increases to Mach 1.5. What is the turning radius of the aircraft now?
- c. Plot the turning radius as a function of aircraft speed for speeds between Mach 0.5 and Mach 2.0, assuming that the acceleration remains 2 g.
- d. Suppose that the maximum acceleration that the pilot can stand is 7 g. What is the minimum possible turning radius of the aircraft at Mach 1.5?
- e. Plot the turning radius as a function of centripetal acceleration for accelerations between 2 g and 8 g, assuming a constant speed of Mach 0.85.

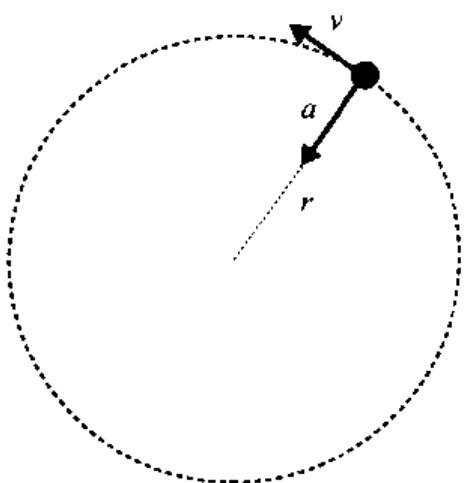


Figure 2.14 An object moving in uniform circular motion due to the centripetal acceleration a .

CHAPTER 3

Branching Statements and Program Design

In the previous chapter, we developed several complete working MATLAB programs. However, all of the programs were very simple, consisting of a series of MATLAB statements that were executed one after another in a fixed order. Such programs are called **sequential programs**. They read input data, process it to produce a desired answer, print out the answer, and quit. There is no way to repeat sections of the program more than once, and there is no way to selectively execute only certain portions of the program depending on values of the input data.

In the next two chapters, we will introduce a number of MATLAB statements that allow us to control the order in which statements are executed in a program. There are two broad categories of control statements: **branches**, which select specific sections of the code to execute; and **loops**, which cause specific sections of the code to be repeated. Branches will be discussed in this chapter, and loops will be discussed in Chapter 4.

With the introduction of branches and loops, our programs are going to become more complex, and it will get easier to make mistakes. To help avoid programming errors, we will introduce a formal program design procedure based on the technique known as top-down design. We will also introduce a common algorithm development tool known as pseudocode.

3.1 Introduction to Top-Down Design Techniques

Suppose that you are an engineer working in industry, and that you need to write a program to solve some problem. How do you begin?

When given a new problem, there is a natural tendency to sit down at a keyboard and start programming without “wasting” a lot of time thinking about the problem first. It is often possible to get away with this “on the fly” approach to programming for very small problems, such as many of the examples in this book. In the real world, however, problems are larger, and a programmer attempting this approach will become hopelessly bogged down. For larger problems, it pays to completely think out the problem and the approach you are going to take to it before writing a single line of code.

We introduce a formal program design process in this section, and then apply that process to every major application developed in the remainder of the book. For some of the simple examples that we will be doing, the design process will seem like overkill. However, as the problems that we solve get larger and larger, the process becomes more and more essential to successful programming.

When I was an undergraduate, one of my professors was fond of saying, “Programming is easy. It’s knowing what to program that’s hard.” His point was forcefully driven home to me after I left university and began working in industry on larger scale software projects. I found that the most difficult part of my job was to *understand the problem* I was trying to solve. Once I really understood the problem, it became easy to break the problem apart into smaller, more easily manageable pieces with well-defined functions, and then to tackle those pieces one at a time.

Top-down design is the process of starting with a large task and breaking it down into smaller, more easily understandable pieces (subtasks), which perform a portion of the desired task. Each subtask may in turn be subdivided into smaller subtasks if necessary. Once the program is divided into small pieces, each piece can be coded and tested independently. We do not attempt to combine the subtasks into a complete task until each of the subtasks has been verified to work properly by itself.

The concept of top-down design is the basis of our formal program design process. We will now introduce the details of the process, which is illustrated in Figure 3.1. The steps involved are:

1. *Clearly state the problem that you are trying to solve.*

Programs are usually written to fill some perceived need, but that need may not be articulated clearly by the person requesting the program. For example, a user may ask for a program to solve a system of simultaneous linear equations. This request is not clear enough to allow a programmer to design a program to meet the need; he or she must first know much more about the problem to be solved. Is the system of equations to be solved real or complex? What is the maximum number of equations and unknowns that the program must handle? Are there any symmetries in the equations that might be exploited to make the task easier? The program designer will have to talk with the user requesting the program, and the two of them will have to come up with a clear statement of exactly what they are trying to accomplish. A clear statement of the problem will prevent misunderstandings, and it will also help the program designer to properly organize his or her thoughts. In the example we were describing, a proper statement of the problem might have been:

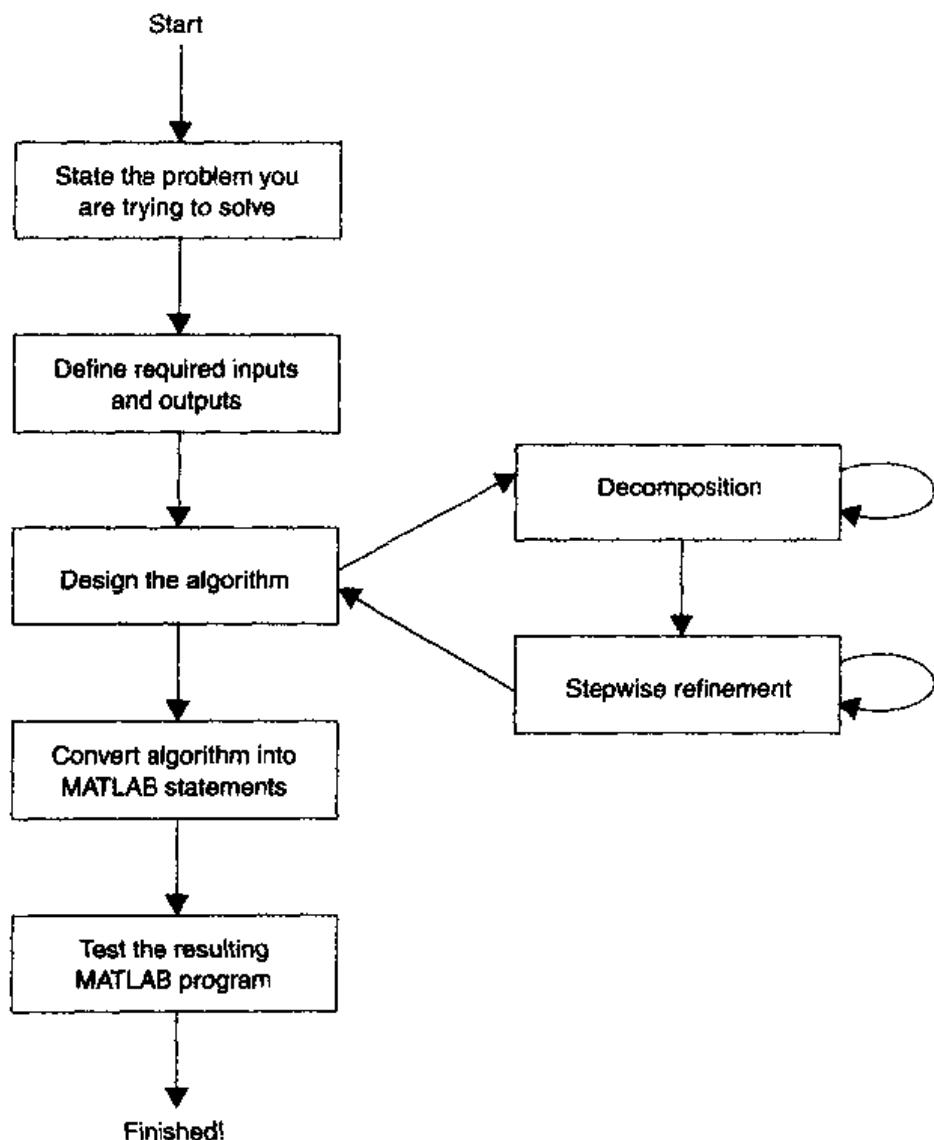


Figure 3.1 The program design process used in this book.

Design and write a program to solve a system of simultaneous linear equations having real coefficients and with up to 20 equations in 20 unknowns.

2. *Define the inputs required by the program and the outputs to be produced by the program.*

The inputs to the program and the outputs produced by the program must be specified so that the new program will properly fit into the overall processing scheme. In this example, the coefficients of the equations to be solved are probably in some pre-existing order, and our new program needs to be able to read them in that order. Similarly, it needs to produce the answers required by the programs that may follow it in the overall processing scheme, and to write out those answers in the format needed by the programs following it.

3. *Design the algorithm that you intend to implement in the program.*

An **algorithm** is a step-by-step procedure for finding the solution to a problem. It is at this stage in the process that top-down design techniques come into play. The designer looks for logical divisions within the problem, and divides it up into subtasks along those lines. This process is called *decomposition*. If the subtasks are large, the designer can break them up into even smaller sub-subtasks. This process continues until the problem has been divided into many small pieces, each of which does a simple, clearly understandable job.

After the problem has been decomposed into small pieces, each piece is further refined through a process called *stepwise refinement*. In stepwise refinement, a designer starts with a general description of what the piece of code should do, and then defines the functions of the piece in greater and greater detail until they are specific enough to be turned into MATLAB statements. Stepwise refinement is usually done with **pseudocode**, which will be described in the next section.

It is often helpful to solve a simple example of the problem by hand during the algorithm development process. If the designer understands the steps that he or she went through in solving the problem by hand, then he or she will be better able to apply decomposition and stepwise refinement to the problem.

4. *Turn the algorithm into MATLAB statements.*

If the decomposition and refinement process was carried out properly, this step will be very simple. All the programmer will have to do is to replace pseudocode with the corresponding MATLAB statements on a one-for-one basis.

5. *Test the resulting MATLAB program.*

This step is the real killer. The components of the program must first be tested individually, if possible, and then the program as a whole must be tested. When testing a program, we must verify that it works correctly for *all legal input data sets*. It is very common for a program to be written, tested with some standard data set, and released for use, only to find that it produces the wrong answers (or crashes) with a different input data set. If the algorithm implemented in a program includes different branches, we must test all of the possible branches to confirm that the program operates correctly under every possible circumstance.

Large programs typically go through a series of tests before they are released for general use (see Figure 3.2). The first stage of testing is sometimes called **unit testing**. During unit testing, the individual subtasks of the program are tested separately to confirm that they work correctly. After the unit testing is completed, the program goes through a series of *builds*, during which the individual subtasks are combined to produce the final program. The first build of the program typically

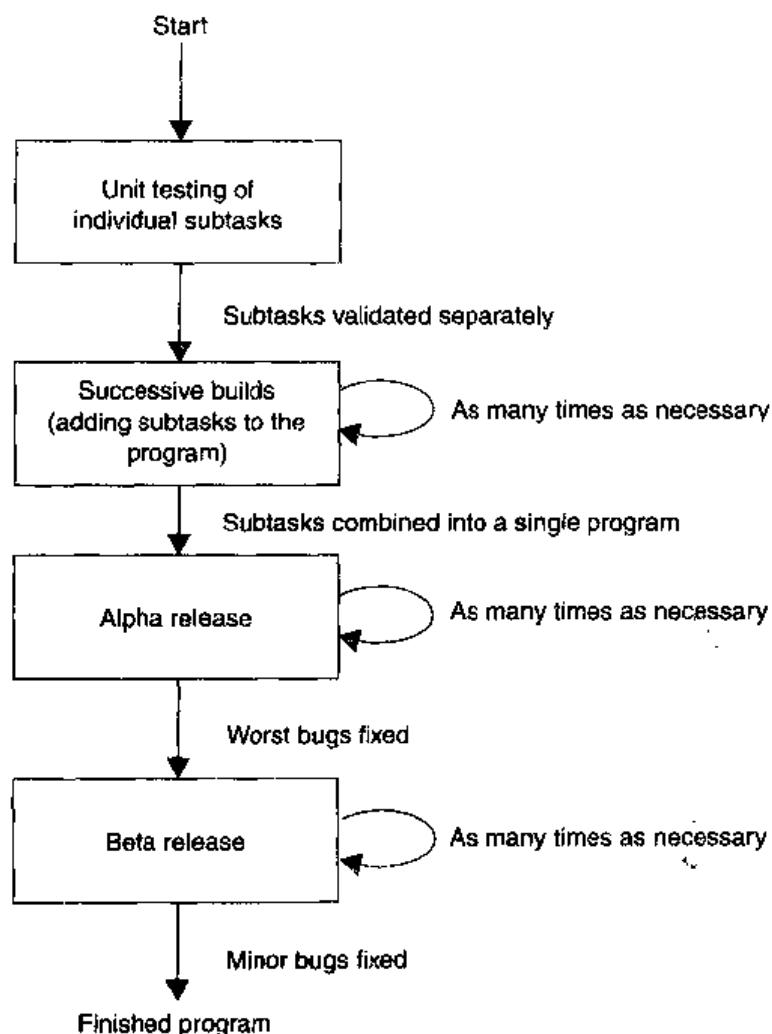


Figure 3.2 A typical testing process for a large program.

includes only a few of the subtasks. It is used to check the interactions among those subtasks and the functions performed by the combinations of the subtasks. In successive builds, more and more subtasks are added, until the entire program is complete. Testing is performed on each build, and any errors (bugs) detected are corrected before moving on to the next build.

Testing continues even after the program is complete. The first complete version of the program is usually called the **alpha release**. It is exercised by the programmers and others very close to them in as many different ways as possible, and the bugs discovered during the testing are corrected. When the most serious bugs have been removed from the program, a new version called the **beta release** is prepared. The beta release is normally given to “friendly” outside users who have a need for the program in their normal day-to-day jobs. These users put the program

through its paces under many different conditions and with many different input data sets, and they report any bugs that they find to the programmers. When those bugs have been corrected, the program is ready to be released for general use. Because the programs in this book are fairly small, we will not go through the sort of extensive testing described above. However, we will follow the basic principles in testing all of our programs.

The program design process can be summarized as follows.

1. Clearly state the problem that you are trying to solve.
2. Define the inputs required by the program and the outputs to be produced by the program.
3. Design the algorithm that you intend to implement in the program.
4. Turn the algorithm into MATLAB statements.
5. Test the MATLAB program.

Good Programming Practice

Follow the steps of the program design process to produce reliable, understandable MATLAB programs.

In a large programming project, the time actually spent programming is surprisingly small. In the book *The Mythical Man-Month*¹, Frederick P. Brooks, Jr. suggests that in a typical large software project, one third of the time is spent planning what to do (steps 1 through 3), one sixth of the time is spent actually writing the program (step 4), and fully half of the time is spent in testing and debugging the program! Clearly, anything that we can do to reduce the testing and debugging time will be very helpful. We can best reduce the testing and debugging time by doing a very careful job in the planning phase and by using good programming practices. Good programming practices will reduce the number of bugs in the program and will make the ones that do creep in easier to find.

3.2 Use of Pseudocode

- As a part of the design process, it is necessary to describe the algorithm that you intend to implement. The description of the algorithm should be in a standard form that is easy for both you and other people to understand, and the description should aid you in turning your concept into MATLAB code. The standard forms that we use to describe algorithms are called **constructs** (or sometimes struc-

¹*The Mythical Man-Month*, by Frederick P. Brooks Jr., Addison-Wesley, 1974.

tures), and an algorithm described using these constructs is called a structured algorithm. When the algorithm is implemented in a MATLAB program, the resulting program is called a **structured program**.

The constructs used to build algorithms can be described in a special way called pseudocode. **Pseudocode** is a hybrid mixture of MATLAB and English. It is structured like MATLAB, with a separate line for each distinct idea or segment of code, but the descriptions on each line are in English. Each line of the pseudocode should describe its idea in plain, easily understandable English. Pseudocode is very useful for developing algorithms since it is flexible and easy to modify. It is especially useful since pseudocode can be written and modified with the same editor or word processor used to write the MATLAB program—no special graphical capabilities are required.

For example, the pseudocode for the algorithm in Example 2.3 is:

```
Prompt user to enter temperature in degrees Fahrenheit
Read temperature in degrees Fahrenheit (temp_f)
temp_k in kelvins ← (5/9) * (temp_f - 32) + 273.15
Write temperature in kelvins
```

Notice that a left arrow (\leftarrow) is used instead of an equal sign (=) to indicate that a value is stored in a variable, since this avoids any confusion between assignment and equality. Pseudocode is intended to aid you in organizing your thoughts before converting them into MATLAB code.

3.3 Relational and Logical Operators

The operation of many branching constructs is controlled by an expression whose result is either true (1) or false (0). There are two types of operators that produce true/false results in MATLAB: relational operators and logic operators.

Like the C language, MATLAB does not have a boolean or logical data type. Instead, MATLAB interprets a zero value as false result, and any nonzero value as true result.

3.3.1 Relational Operators

Relational operators are operators with two numerical or string operands that yield either a true (1) or a false (0) result, depending on the relationship between the two operands. The general form of a relational operator is

$$a_1 \text{ op } a_2$$

where a_1 and a_2 are arithmetic expressions, variables, or strings, and op is one of the relational operators in Table 3.1.

If the relationship between a_1 and a_2 expressed by the operator is true, then the operation returns a value of 1; otherwise, the operation returns a value of 0.

Table 3.1 Relational Operators

Operator	Operation
<code>==</code>	Equal to
<code>~=</code>	Not equal to
<code>></code>	Greater than
<code>>=</code>	Greater than or equal to
<code><</code>	Less than
<code><=</code>	Less than or equal to

Some relational operations and their results follow.

Operation	Result
<code>3 < 4</code>	1
<code>3 <= 4</code>	1
<code>3 == 4</code>	0
<code>3 > 4</code>	0
<code>4 <= 4</code>	1
<code>'A' < 'B'</code>	1

The last relational operation produces a 1 because characters are evaluated in alphabetical order.

Relational operators can be used to compare a scalar value with an array. For example, if $a = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}$ and $b = 0$, then the expression $a > b$ will yield the result $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Relational operators can also be used to compare two arrays, as long as both arrays have the same size. For example, if $a = \begin{bmatrix} 1 & 0 \\ -2 & 1 \end{bmatrix}$ and $b = \begin{bmatrix} 0 & 2 \\ -2 & -1 \end{bmatrix}$, then the expression $a >= b$ will yield the result $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$. If the arrays have different sizes, a runtime error will result.

Note that since strings are really arrays of characters, *relational operators can only compare two strings if they are of equal lengths*. If they are of unequal lengths, the comparison operation will produce an error. We will learn of a more general way to compare strings in Chapter 6.

The equivalence relational operator is written with two equal signs, while the assignment operator is written with a single equal sign. These are very different operators, which beginning programmers often confuse. The `==` symbol is a *comparison* operation, which returns a logical result, whereas the `=` symbol *assigns* the value of the expression to the right of the equal sign to the variable on the left of the equal sign. It is a very common mistake for

beginning programmers to use a single equal sign when trying to do a comparison.

Programming Pitfalls

Be careful not to confuse the equivalence relational operator (`==`) with the assignment operator (`=`).

In the hierarchy of operations, relational operators are evaluated after all arithmetic operators have been evaluated. Therefore, the following two expressions are equivalent (both are true and produce a value of 1).

$$\begin{aligned} 7 + 3 &< 2 + 11 \\ (7 + 3) &< (2 + 11) \end{aligned}$$

3.3.2 A Caution about the `==` and `~=` Operators

The equivalence operator (`==`) returns a 1 when the two values being compared are equal, and a 0 when the two values being compared are different. Similarly, non-equivalence operator (`~=`) returns a 0 when the two values being compared are equal, and a 1 when the two values being compared are different. These operators are generally safe to use for comparing strings, but they can sometimes produce surprising results when two numeric values are compared. Due to **roundoff errors** during computer calculations, two theoretically equal numbers can differ slightly, causing an equality or inequality test to fail.

For example, consider the following two numbers, both of which should be equal to 0.0.

```
a = 0;
b = sin(pi);
```

Since these numbers are theoretically the same, the relational operation `a == b` should produce a 1. In fact, the results of this MATLAB calculation are

```
>> a = 0;
>> b = sin(pi);
>> a == b
ans =
0
```

MATLAB reports that `a` and `b` are different because a slight roundoff error in the calculation of `sin(pi)` makes the result be 1.2246×10^{-16} instead of exactly zero. The two theoretically equal values differ slightly due to roundoff error!

Instead of comparing two numbers for *exact* equality, you should set up your tests to determine if the two numbers *nearly* equal to each other within some

accuracy that takes into account the roundoff error expected for the numbers being compared. The test

```
>> abs(a - b) < 1.0E-14
ans =
1
```

produces the correct answer, despite the roundoff errors in calculating *a* and *b*.

Be cautious about testing for equality with numeric values, since roundoff errors can cause two variables that should be equal to fail a test for equality. Instead, test to see if the variables are *nearly* equal within the roundoff error to be expected on the computer you are working with.

3.3.3 Logic Operators

Logic operators are operators with one or two logical operands that yield a logical result. There are three binary logic operators: AND, OR, and exclusive OR; and one unary operator: NOT. The general form of a binary logic operation is

$$l_1 \text{ op } l_2$$

and the general form of a unary logic operation is

$$\text{op } l_1$$

where *l*₁ and *l*₂ are expressions or variables, and op is one of the logic operators shown in Table 3.2. If the relationship between *l*₁ and *l*₂ expressed by the operator is true, then the operation returns a value of 1; otherwise, the operation returns a value of 0.

The results of the operators are summarized in the following truth tables shown in Table 3.3, which show the result of each operation for all possible com-

Table 3.2 Logic Operators

Operator	Operation
&	Logical AND
	Logical OR
xor	Logical Exclusive OR
~	Logical NOT

Table 3.3 Truth Tables Logic Operators

Inputs		and $I_1 \& I_2$	or $I_1 I_2$	xor $xor(I_1, I_2)$	not $\sim I_1$
I_1	I_2				
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

binations of I_1 and I_2 . For the purposes of logic operations, MATLAB treats an operand as true if it is any non-zero value, and false if it is zero. Thus, the result of ~ 5 is 0 and the result of ~ 0 is 1.

Logic operators can be used to compare a scalar value with an array. For example, if $a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $b = 0$, then the expression $a \& b$ will yield the result $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$. Logic operators can also be used to compare two arrays, as long as both arrays have the same size. For example, if $a = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $b = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$, then the expression $a | b$ will yield the result $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$. If the arrays have different sizes, a runtime error will result.

In the hierarchy of operations, logic operators are evaluated *after all arithmetic operations and all relational operators have been evaluated*. The order in which the operators in an expression are evaluated is as follows:

1. All arithmetic operators are evaluated first in the order previously described.
2. All relational operators ($==$, $\sim=$, $>$, \geq , $<$, \leq) are evaluated, working from left to right.
3. All \sim operators are evaluated.
4. All $\&$ operators are evaluated, working from left to right.
5. All $|$ operators are evaluated, working from left to right.²

As with arithmetic operations, parentheses can be used to change the default order of evaluation. Examples of some logic operators and their results are given in Example 3.1 below.

²Note: Before MATLAB 6, the $\&$ and $|$ operators had equal precedence, and were evaluated in order from left to right. Beginning with MATLAB 6, $\&$ is evaluated before $|$. Thus, an expression such as $a | b \& c$ would be evaluated differently in MATLAB 5.3.1 and earlier than in MATLAB 6.0 and later. Beware! To be safe, always use parentheses to indicate the order in which you want an expression to be evaluated.

Example 3.1

Assume that the following variables are initialized with the values shown, and calculate the result of the specified expressions.

```
value1 = 1
value2 = 0
value3 = -10
```

Logical Expression	Result
(a) ~value1	0
(b) value1 value2	1
(c) value1 & value2	0
(d) value1 & value2 value3	1
(e) value1 & (value2 value3)	1
(f) ~(value1 & value3)	0

The `~` operator is evaluated before other logic operators. Therefore, the parentheses in part (f) of the above example were required. If they had been absent, the expression in part (f) would have been evaluated in the order `(~value1) & value3`.

3.3.4 Logical Functions

MATLAB includes a number of logical functions that return a 1 whenever the condition they test for is true, and a 0 whenever the condition they test for is false. These functions can be used with relation and logic operators to control the operation of branches and loops.

A few of the more important logical functions are given in Table 3.4.

Table 3.4 MATLAB Logical Functions

Function	Purpose
<code>ischar(a)</code>	Returns a 1 if <code>a</code> is a character array and a 0 otherwise.
<code>isempty(a)</code>	Returns a 1 if <code>a</code> is an empty array and a 0 otherwise.
<code>isinf(a)</code>	Returns a 1 if the value of <code>a</code> is infinite (<code>Inf</code>) and a 0 otherwise.
<code>isnan(a)</code>	Returns a 1 if the value of <code>a</code> is NaN (not a number) and a 0 otherwise.
<code>isnumeric(a)</code>	Returns a 1 if the <code>a</code> is a numeric array and a 0 otherwise.

Quiz 3.1

This quiz provides a quick check to see if you have understood the concepts introduced in Section 3.3. If you have trouble with the quiz, reread the sections, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

Assume that `a`, `b`, `c`, and `d` are as defined, and evaluate the following expressions.

```
a = 20;      b = -2;
c = 0;      d = 1;
```

1. `a > b`
2. `b > d;`
3. `a > b & c > d`
4. `a == b`
5. `a & b > c`
6. `--b`

Assume that `a`, `b`, `c`, and `d` are as defined, and evaluate the following expressions.

$$a = 2; \quad b = \begin{bmatrix} 1 & -2 \\ -0 & 10 \end{bmatrix};$$

$$c = \begin{bmatrix} 0 & 1 \\ 2 & 0 \end{bmatrix}; \quad d = \begin{bmatrix} -2 & 1 & 2 \\ 0 & 1 & 0 \end{bmatrix};$$

7. `~(a > b)`
8. `a > c & b > c;`
9. `c <= d`

Assume that `a`, `b`, `c`, and `d` are as defined. Explain the order in which each of the following expressions are evaluated, and specify the results in each case:

```
a = 2;          b = 3;
c = 10;         d = 0;
```

10. `a*b^2 > a*c`
11. `d | b > a`
12. `(d | b) > a`

Assume that `a`, `b`, `c`, and `d` are as defined, and evaluate the following expressions.

```
a = 20;          b = -2;
c = 0;          d = 'Test';
```

13. `isinf(a/b)`
14. `isinf(a/c)`
15. `a > b & ischar(d)`
16. `isempty(c)`

3.4 Branches

Branches are MATLAB statements that permit us to select and execute specific sections of code (called *blocks*) while skipping other sections of code. They are variations of the `if` construct, the `switch` construct, and the `try/catch` construct.

3.4.1 The `if` Construct

The `if` construct has the form

```
if control_expr_1
    Statement 1
    Statement 2 } Block 1
    ...
elseif control_expr_2
    Statement 1
    Statement 2 } Block 2
    ...
else
    Statement 1
    Statement 2 } Block 3
    ...
end
```

where the control expressions control the operation of the `if` construct. If `control_expr_1` is nonzero, then the program executes the statements in Block 1, and skips to the first executable statement following the `end`. Otherwise, the program checks for the status of `control_expr_2`. If `control_expr_2` is nonzero, then the program executes the statements in Block 2, and skips to the first executable statement following the `end`. If all control expressions are zero, then the program executes the statements in the block associated with the `else` clause.

There can be any number of `elseif` clauses (0 or more) in an `if` construct, but there can be at most one `else` clause. The control expression in each clause will be tested only if the control expressions in every clause above it evaluate to zero. Once one of the expressions proves to be nonzero and the corresponding code block is executed, the program skips to the first executable statement following the `end`. If all control expressions are zero, then the program executes the statements in the block associated with the `else` clause. If there is no `else` clause, then execution continues after the `end` statement without executing any part of the `if` construct.

Note that the MATLAB keyword `end` in this construct is *completely different* from the MATLAB function `end` that we used in Chapter 2 to return the highest value of a given subscript. MATLAB differentiates between these two uses of `end` from the context in which the word appears within an M-file.

In most circumstances, *the control expressions will be some combination of relational and logic operators*. As we learned earlier in this chapter, relational and logic operators produce a 1 when the corresponding condition is true and a 0 when the corresponding condition is false. Thus when an operator

is true, its result is nonzero, and the corresponding block of code will be executed.

As an example of an `if` construct, consider the solution of a quadratic equation of the form

$$ax^2 + bx + c = 0 \quad (3.1)$$

The solution to this equation is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (3.2)$$

The term $b^2 - 4ac$ is known as the *discriminant* of the equation. If $b^2 - 4ac > 0$, then there are two distinct real roots to the quadratic equation. If $b^2 - 4ac = 0$, then there is a single repeated root to the equation; and if $b^2 - 4ac < 0$, then there are two complex roots to the quadratic equation.

Suppose that we wanted to examine the discriminant of a quadratic equation and to tell a user whether the equation has two complex roots, two identical real roots, or two distinct real roots. In pseudocode, this construct would take the form

```
if (b^2 - 4*a*c) < 0
    Write msg that equation has two complex roots.
elseif (b^2 - 4*a*c) == 0
    Write msg that equation has two identical real roots.
else
    Write msg that equation has two distinct real roots.
end
```

The MATLAB statements to do this are

```
if (b^2 - 4*a*c) < 0
    disp('This equation has two complex roots.');
elseif (b^2 - 4*a*c) == 0
    disp('This equation has two identical real roots.');
else
    disp('This equation has two distinct real roots.');
end
```

Recall that relational operators return a non-zero value (1) when the test is true, so a true test will cause the corresponding code block to be executed.

For readability, the blocks of code within an `if` construct are usually indented by 2 or 3 spaces, but this is not actually required.

Good Programming Practice

Always indent the body of an `if` construct by 2 or more spaces to improve the readability of the code.

It is possible to write a complete `if` construct on a single line by separating the parts of the construct by commas or semicolons. Thus, the following two constructs are identical:

```
if x < 0
    y = abs(x);
end
```

and

```
if x < 0; y = abs(x); end
```

However, this should only be done for very simple constructs.

3.4.2 Examples Using `if` Constructs

We now look at two examples that illustrate the use of `if` constructs.

Example 3.2—The Quadratic Equation

Design and write a program to solve for the roots of a quadratic equation, regardless of type.

Solution

We will follow the design steps outlined earlier in the chapter.

1. State the problem.

The problem statement for this example is very simple. We want to write a program that will solve for the roots of a quadratic equation, whether they are distinct real roots, repeated real roots, or complex roots.

2. Define the inputs and outputs.

The inputs required by this program are the coefficients a , b , and c of the quadratic equation

$$ax^2 + bx + c = 0 \quad (3.1)$$

The output from the program will be the roots of the quadratic equation, whether they are distinct real roots, repeated real roots, or complex roots.

3. Design the algorithm.

This task can be broken down into three major sections, whose functions are input, processing, and output.

```
Read the input data
Calculate the roots
Write out the roots
```

We now break each of the above major sections into smaller, more detailed pieces. There are three possible ways to calculate the roots, depending on the value of the discriminant, so it is logical to implement this algorithm with a three-branched `if` construct. The resulting pseudo-code is

```

Prompt the user for the coefficients a, b, and c.
Read a, b, and c
discriminant ← b^2 - 4 * a * c
if discriminant > 0
    x1 ← ( -b + sqrt(discriminant) ) / ( 2 * a )
    x2 ← ( -b - sqrt(discriminant) ) / ( 2 * a )
    Write msg that equation has two distinct real roots.
    Write out the two roots.
elseif discriminant == 0
    x1 ← -b / ( 2 * a )
    Write msg that equation has two identical real roots.
    Write out the repeated root.
else
    real_part ← -b / ( 2 * a )
    imag_part ← sqrt( abs( discriminant ) ) / ( 2 * a )
    Write msg that equation has two complex roots.
    Write out the two roots.
end

```

4. Turn the algorithm into MATLAB statements.

The final MATLAB code follows.

```

% Script file: calc_roots.m
%
% Purpose:
% This program solves for the roots of a quadratic equation
% of the form a*x^2 + b*x + c = 0. It calculates the answers
% regardless of the type of roots that the equation possesses.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   ===       =====       =====
%   12/04/98  S. J. Chapman  Original code
%
% Define variables:
% a          -- Coefficient of x^2 term of equation
% b          -- Coefficient of x term of equation
% c          -- Constant term of equation
% discriminant -- Discriminant of the equation
% imag_part  -- Imag part of equation (for complex roots)
% real_part   -- Real part of equation (for complex roots)
% x1         -- First solution of equation (for real roots)
% x2         -- Second solution of equation (for real roots)

% Prompt the user for the coefficients of the equation
disp ('This program solves for the roots of a quadratic ');
disp ('equation of the form A*X^2 + B*X + C = 0. ');
a = input ('Enter the coefficient A: ');
b = input ('Enter the coefficient B: ');
c = input ('Enter the coefficient C: ');

```

```

% Calculate discriminant
discriminant = b^2 - 4 * a * c;

% Solve for the roots, depending on the value of the discriminant
if discriminant > 0 % there are two real roots, so...

    x1 = ( -b + sqrt(discriminant) ) / ( 2 * a );
    x2 = ( -b - sqrt(discriminant) ) / ( 2 * a );
    disp ('This equation has two real roots:');
    fprintf ('x1 = %f\n', x1);
    fprintf ('x2 = %f\n', x2);

elseif discriminant == 0 % there is one repeated root, so...

    x1 = ( -b ) / ( 2 * a );
    disp ('This equation has two identical real roots:');
    fprintf ('x1 = x2 = %f\n', x1);

else % there are complex roots, so ...

    real_part = ( -b ) / ( 2 * a );
    imag_part = sqrt ( abs ( discriminant ) ) / ( 2 * a );
    disp ('This equation has complex roots:');
    fprintf('x1 = %f +i %f\n', real_part, imag_part );
    fprintf('x1 = %f -i %f\n', real_part, imag_part );

end

```

5. Test the program.

Next, we must test the program using real input data. Since there are three possible paths through the program, we must test all three paths before we can be certain that the program is working properly. From Equation (3.2), it is possible to verify the solutions to the equations that follow.

$$\begin{array}{ll} x^2 + 5x + 6 = 0 & x = -2, \text{ and } x = -3 \\ x^2 + 4x + 4 = 0 & x = -2 \\ x^2 + 2x + 5 = 0 & x = -1 \pm i\sqrt{2} \end{array}$$

If this program is executed three times with the preceding coefficients, the results are as shown (user inputs are shown in boldface).

```

> calc_roots
This program solves for the roots of a quadratic
equation of the form A*X^2 + B*X + C = 0.
Enter the coefficient A: 1
Enter the coefficient B: 5
Enter the coefficient C: 6

```

```

This equation has two real roots:
x1 = -2.000000
x2 = -3.000000
» calc_roots
This program solves for the roots of a quadratic
equation of the form A*X^2 + B*X + C = 0.
Enter the coefficient A: 1
Enter the coefficient B: 4
Enter the coefficient C: 4
This equation has two identical real roots:
x1 = x2 = -2.000000
» calc_roots
This program solves for the roots of a quadratic
equation of the form A*X^2 + B*X + C = 0.
Enter the coefficient A: 1
Enter the coefficient B: 2
Enter the coefficient C: 5
This equation has complex roots:
x1 = -1.000000 +i 2.000000
x1 = -1.000000 -i 2.000000

```

The program gives the correct answers for our test data in all three possible cases.

Example 3.3—Evaluating a Function of Two Variables

Write a MATLAB program to evaluate a function $f(x,y)$ for any two user-specified values x and y . The function $f(x,y)$ is defined as follows.

$$f(x,y) = \begin{cases} x + y & x \geq 0 \text{ and } y \geq 0 \\ x + y^2 & x \geq 0 \text{ and } y < 0 \\ x^2 + y & x < 0 \text{ and } y \geq 0 \\ x^2 + y^2 & x < 0 \text{ and } y < 0 \end{cases}$$

Solution

The function $f(x,y)$ is evaluated differently depending on the signs of the two independent variables x and y . To determine the proper equation to apply, it will be necessary to check for the signs of the x and y values supplied by the user.

1. State the problem.

This problem statement is very simple: Evaluate the function $f(x,y)$ for any user-supplied values of x and y .

2. Define the inputs and outputs.

The inputs required by this program are the values of the independent variables x and y . The output from the program will be the value of the function $f(x,y)$.

3. Design the algorithm.

This task can be broken down into three major sections, whose functions are input, processing, and output.

```
Read the input values x and y
Calculate f(x,y)
Write out f(x,y)
```

We now break each of the above major sections into smaller, more detailed pieces. There are four possible ways to calculate the function $f(x,y)$, depending on the values of x and y , so it is logical to implement this algorithm with a four-branched if construct. The resulting pseudocode is

```
Prompt the user for the values x and y.
Read x and y
if x ≥ 0 and y ≥ 0
    fun ← x + y
elseif x ≥ 0 and y < 0
    fun ← x + y^2
elseif x < 0 and y ≥ 0
    fun ← x^2 + y
else
    fun ← x^2 + y^2
end
Write out f(x,y)
```

4. Turn the algorithm into MATLAB statements.

The final MATLAB code follows.

```
% Script file: funxy.m
%
% Purpose:
%   This program solves the function f(x,y) for a
%   user-specified x and y, where f(x,y) is defined as:
%
%   f(x,y) = | x + y      x >= 0 and y >= 0
%             | x + y^2    x >= 0 and y < 0
%             | x^2 + y    x < 0 and y >= 0
%             | x^2 + y^2  x < 0 and y < 0
%
% Record of revisions:
%   Date      Programmer      Description of change
%   =====      =====      =====
%   12/05/98  S. J. Chapman  Original code
%
```

```

% Define variables:
%   x -- First independent variable
%   y -- Second independent variable
%   fun -- Resulting function

% Prompt the user for the values x and y
x = input ('Enter the x coefficient: ');
y = input ('Enter the y coefficient: ');

% Calculate the function f(x,y) based upon
% the signs of x and y.
if x >= 0 & y >= 0
    fun = x + y;
elseif x >= 0 & y < 0
    fun = x + y^2;
elseif x < 0 & y >= 0
    fun = x^2 + y;
else
    fun = x^2 + y^2;
end

% Write the value of the function.
disp (['The value of the function is ' num2str(fun)]);

```

5. Test the program.

Next, we must test the program using real input data. Since there are four possible paths through the program, we must test all four paths before we can be certain that the program is working properly. To test all four possible paths, we will execute the program with the four sets of input values $(x,y) = (2, 3), (2, -3), (-2, 3)$, and $(-2, -3)$. Calculating by hand, we see that

$$\begin{aligned}f(2, 3) &= 2 + 3 = 5 \\f(2, -3) &= 2 + (-3)^2 = 11 \\f(-2, 3) &= (-2)^2 + 3 = 7 \\f(-2, -3) &= (-2)^2 + (-3)^2 = 13\end{aligned}$$

If this program is compiled and then run four times with the preceding values, the results are:

```

* funny
Enter the x coefficient: 2
Enter the y coefficient: 3
The value of the function is 5

```

```

» funxy
Enter the x coefficient: 2
Enter the y coefficient: -3
The value of the function is 11
» funxy
Enter the x coefficient: -2
Enter the y coefficient: 3
The value of the function is 7
» funxy
Enter the x coefficient: -2
Enter the y coefficient: -3
The value of the function is 13

```

The program gives the correct answers for our test values in all four possible cases.

3.4.3 Notes Concerning the Use of **if** Constructs

The **if** construct is very flexible. It must have one **if** statement and one **end** statement. In between, it can have any number of **elseif** clauses, and can also have one **else** clause. With this combination of features, it is possible to implement any desired branching construct.

In addition, **if** constructs can be **nested**. Two **if** constructs are said to be nested if one of them lies entirely within a single code block of the other one. The following two **if** constructs are properly nested.

```

if x > 0
    ...
    if y < 0
        ...
    end
    ...
end

```

The MATLAB interpreter always associates a given **end** statement with the most recent **if** statement, so the first **end** closes the **if y < 0** statement, while the second **end** closes the **if x > 0** statement. This works well for a properly written program, but can cause the interpreter to produce confusing error messages in cases where the programmer makes a coding error. For example, suppose we have a large program containing a construct like the one that follows.

```

...
if (test1)
    ...
    if (test2)
        ...

```

```

if (test3)
    ...
end
...
end
...
end

```

This program contains three nested `if` constructs that may span hundreds of lines of code. Now suppose that the first `end` statement is accidentally deleted during an editing session. When that happens, the MATLAB interpreter will automatically associate the second `end` with the innermost `if (test3)` construct and the third `end` with the middle `if (test2)`. When the interpreter reaches the end of the file, it will notice that the first `if (test1)` construct was never ended, and it will generate an error message saying that there is a missing `end`. Unfortunately, it cannot tell *where* the problem occurred, so we will have to go back and manually search the entire program to locate the problem.

It is sometimes possible to implement an algorithm using either multiple `elseif` clauses or nested `if` statements. In that case, a programmer can choose whichever style he or she prefers.

Example 3.4—Assigning Letter Grades

Suppose that we are writing a program which reads in a numerical grade and assigns a letter grade to it according to the following table.

grade > 95	A
$95 \geq \text{grade} > 86$	B
$86 \geq \text{grade} > 76$	C
$76 \geq \text{grade} > 66$	D
$66 \geq \text{grade} > 0$	F

Write an `if` construct that will assign the grades as described above using (a) multiple `elseif` clauses and (b) nested `if` constructs.

Solution

(a) One possible structure using `elseif` clauses is

```

if grade > 95.0
    disp('The grade is A.');
elseif grade > 86.0
    disp('The grade is B.');
elseif grade > 76.0
    disp('The grade is C.');
elseif grade > 66.0
    disp('The grade is D.');

```

```

    else
        disp('The grade is F.');
    end

```

(b) One possible structure using nested if constructs is

```

if grade > 95.0
    disp('The grade is A.');
else
    if grade > 86.0
        disp('The grade is B.');
    else
        if grade > 76.0
            disp('The grade is C.');
        else
            if grade > 66.0
                disp('The grade is D.');
            else
                disp('The grade is F.');
            end
        end
    end
end

```

It should be clear from the previous example that if there are a lot of mutually exclusive options, a single if construct with multiple elseif clauses will be simpler than a nested if construct.

Good Programming Practice

For branches in which there are many mutually exclusive options, use a single if construct with multiple elseif clauses in preference to nested if constructs.

3.4.4 The switch Construct

The switch construct is another form of branching construct. It permits a programmer to select a particular code block to execute based on the value of a single integer, character, or logical expression. The general form of a switch construct is:

```

switch (switch_expr)
case case_expr_1,
    Statement 1
    Statement 2
    ...
}
} Block 1

```

```

case case_expr_2,
    Statement 1
    Statement 2
    ...
    ...
otherwise,
    Statement 1
    Statement 2
    ...
end

```

} Block 2 } Block n

If the value of *switch_expr* is equal to *case_expr_1*, then the first code block will be executed, and the program will jump to the first statement following the end of the *switch* construct. Similarly, if the value of *switch_expr* is equal to *case_expr_2*, then the second code block will be executed, and the program will jump to the first statement following the end of the *switch* construct. The same idea applies for any other cases in the construct. The *otherwise* code block is optional. If it is present, it will be executed whenever the value of *switch_expr* is outside the range of all the case selectors. If it is not present and the value of *switch_expr* is outside the range of all the case selectors, then none of the code blocks will be executed. The pseudocode for the *case* construct looks just like its MATLAB implementation.

If many values of the *switch_expr* should cause the same code to execute, all of those values may be included in a single block by enclosing them in brackets, as shown below. If the switch expression matches any of the case expressions in the list, then the block will be executed.

```

switch (switch_expr)
case {case_expr_1, case_expr_2, case_expr_3},
    Statement 1
    Statement 2
    ...
otherwise,
    Statement 1
    Statement 2
    ...
end

```

} Block 1 } Block n

The *switch_expr* and each *case_expr* may be either numerical or string values.

Note that at most one code block can be executed. After a code block is executed, execution skips to the first executable statement after the *end* statement. Thus if the switch expression matches more than one case expression, *only the first one of them will be executed*.

Let's look at a simple example of a *switch* construct. The following statements determine whether an integer between 1 and 10 is even or odd, and print out an appropriate message. It illustrates the use of a list of values as case selectors and also the use of the *otherwise* block.

```

switch (value)
case {1,3,5,7,9},
    disp('The value is odd.');
case {2,4,6,8,10},
    disp('The value is even.');
otherwise,
    disp('The value is out of range.');
end

```

3.4.5 The try/catch Construct

The **try/catch** construct is a special form branching construct designed to trap errors. Ordinarily, when a MATLAB program encounters an error while running, the program aborts. The **try/catch** construct modifies this default behavior. If an error occurs in a statement in the **try** block of this construct, then instead of aborting, the code in the **catch** block is executed and the program keeps running. This allows a programmer to handle errors within the program without causing the program to stop.

The general form of a **try/catch** construct is:

```

try
    Statement 1
    Statement 2
    ...
}
catch
    Statement 1
    Statement 2
    ...
}
end

```

When a **try/catch** construct is reached, the statements in the **try** block of a will be executed. If no error occurs, the statements in the **catch** block will be skipped, and execution will continue at the first statement following the end of the construct. On the other hand, if an error *does* occur in the **try** block, the program will stop executing the statements in the **try** block, and immediately execute the statements in the **catch** block.

An example program containing a **try/catch** construct follows. This program creates an array, and asks the user to specify an element of the array to display. The user will supply a subscript number, and the program displays the corresponding array element. The statements in the **try** block will always be executed in this program, while the statements in the **catch** block will only be executed if an error occurs in the **try** block.

```
% Initialize array
a = [ 1 -3 2 5];
try
    % Try to display an element
    index = input('Enter subscript of element to display: ');
    disp( ['a(' int2str(index) ') = ' num2str(a(index))]);
catch
    % If we get here an error occurred
    disp( ['Illegal subscript: ' int2str(index)]);
end
```

When this program is executed, the results are:

```
>> try_catch
Enter subscript of element to display: 3
a(3) = 2
>> try_catch
Enter subscript of element to display: 8
Illegal subscript: 8
```

Quiz 3.2

This quiz provides a quick check to see if you have understood the concepts introduced in Section 3.4. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book. Write MATLAB statements that perform the functions described below.

1. If x is greater than or equal to zero, then assign the square root of x to variable `sqrt_x` and print out the result. Otherwise, print out an error message about the argument of the square root function and set `sqrt_x` to zero.
2. A variable `fun` is calculated as `numerator / denominator`. If the absolute value of `denominator` is less than `1.0E-300`, write "Divide by 0 error." Otherwise, calculate and print out `fun`.
3. The cost per mile for a rented vehicle is \$0.50 for the first 100 miles, \$0.30 for the next 200 miles, and \$0.20 for all miles in excess of 300 miles. Write MATLAB statements that determine the total cost and the average cost per mile for a given number of miles (stored in variable `distance`).

Examine the following MATLAB statements. Are they correct or incorrect? If they are correct, what do they output? If they are incorrect, what is wrong with them?

4. `if volts > 125`
 `disp('WARNING: High voltage on line.');`

```

if volts < 105
    disp('WARNING: Low voltage on line.');
else
    disp('Line voltage is within tolerances.');
end

5. color = 'yellow';
switch ( color )
case 'red',
    disp('Stop now!');
case 'yellow',
    disp('Prepare to stop.');
case 'green',
    disp('Proceed through intersection.');
otherwise,
    disp('Illegal color encountered.');
end

6. if temperature > 37
    disp('Human body temperature exceeded.');
elseif temperature > 100
    disp('Boiling point of water exceeded.');
end

```

3.5 Additional Plotting Features

This section describes additional features of the simple two-dimensional plots introduced in Chapter 2. These features permit us to control the range of x and y values displayed on a plot, to lay multiple plots on top of each other, to create multiple figures, to create multiple subplots within a figure, and to provide greater control of the plotted lines and text strings. In addition, we will learn how to create polar plots.

3.5.1 Controlling x - and y -axis Plotting Limits

By default, a plot is displayed with x - and y -axis ranges wide enough to show every point in an input data set. However, it is sometimes useful to display only the subset of the data that is of particular interest. This can be done using the **axis** command/function (see the Sidebar about the relationship between MATLAB commands and functions).

Some of the forms of the **axis** command / function are shown in Table 3.5. The two most important forms are shown in **bold** type—they let a programmer get the current limits of a plot and modify them. A complete list of all options can be found in the MATLAB on-line documentation.

To illustrate the use of **axis**, we will plot the function $f(x) = \sin x$ from -2π to 2π , and then restrict the axes to the region to $0 \leq x \leq \pi$ and $0 \leq y \leq 1$. The

Table 3.5 Forms of the axis Function / Command

Command	Description
<code>v = axis;</code>	This function returns a 4-element row vector containing [xmin xmax ymin ymax], where xmin, xmax, ymin, and ymax are the current limits of the plot.
<code>axis ([xmin xmax ymin ymax]);</code>	This function sets the x and y limits of the plot to the specified values.
<code>axis equal</code>	This command sets the axis increments to be equal on both axes.
<code>axis square</code>	This command makes the current axis box square.
<code>axis normal</code>	This command cancels the effect of axis equal and axis square.
<code>axis off</code>	This command turns off all axis labeling, tick marks, and background.
<code>axis on</code>	This command turns on all axis labeling, tick marks, and background (default case).

Good Programming Practice

Some MATLAB commands seem to be unable to make up their minds whether they are commands or functions. For example, sometimes `axis` seems to be a command and sometimes it seems to be a function. Sometimes we treat it as a command: `axis on`, and other times we might treat it as a function: `axis ([0 20 0 35])`. How is this possible?

The short answer is that MATLAB commands are really implemented by functions, and the MATLAB interpreter is smart enough to substitute the function call whenever it encounters the command. It is always possible to call the command directly as a function instead of using the command syntax. Thus the following two statements are identical:

```
axis on;
axis ('on');
```

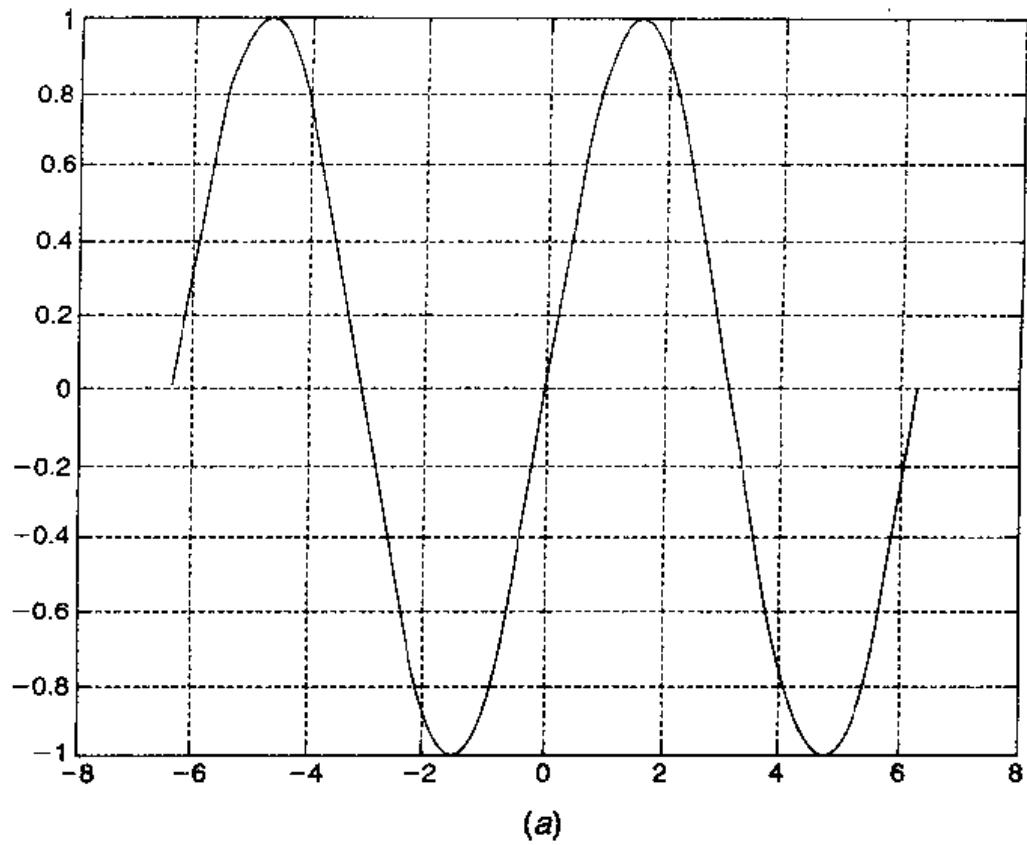
Whenever MATLAB encounters a command, it forms a function from the command by treating each command argument as a character string and calling the equivalent function with those character strings as arguments. Thus MATLAB interprets the command

```
garbage 1 2 3
```

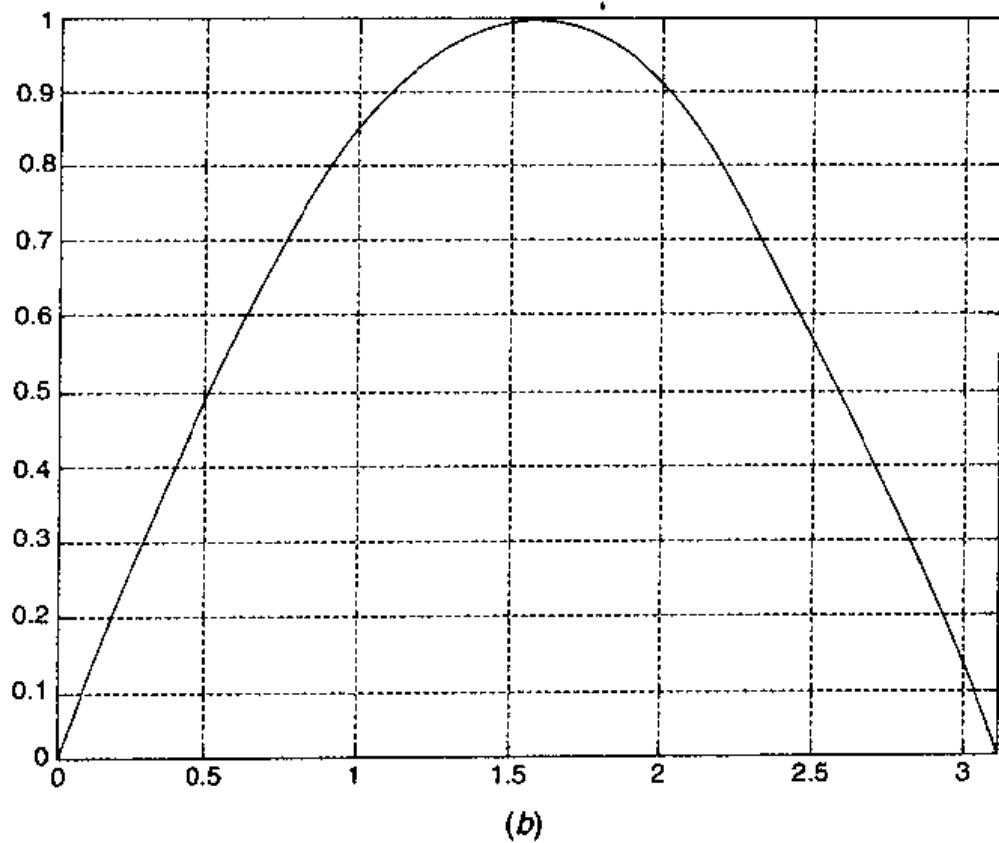
as the following function call:

```
garbage ('1', '2', '3')
```

Note that *only functions with character arguments can be treated as commands*. Functions with numerical arguments must be used in function form only. This fact explains why `axis` is sometimes treated as a command and sometimes treated as a function.



(a)



(b)

Figure 3.3 (a) Plot of $\sin x$ versus x . (b) Closeup of the region $[0 \leq \pi \leq 0 \leq 1]$.

statements to create this plot are shown below, and the resulting plot is shown in Figure 3.3a.

```
x = -2*pi:pi/20:2*pi;
y = sin(x);
plot(x,y);
title ('Plot of sin(x) vs x');
grid on;
```

The current limits of this plot can be determined from the **basic axis** function.

```
>> limits=axis
limits =
    -8     8     -1      1
```

These limits can be modified with the function call `axis([0 pi 0 1])`. After that function is executed, the resulting plot is shown in Figure 3.3b.

3.5.2 Plotting Multiple Plots on the Same Axes

Normally, a new plot is created each time that a `plot` command is issued, and the previous data is lost. This behavior can be modified with the **hold** command. After a **hold on** command is issued, all additional plots will be laid on top of the previously existing plots. A **hold off** command switches plotting behavior back to the default situation, in which a new plot replaces the previous one.

For example, the following commands plot $\sin x$ and $\cos x$ on the same axes. The resulting plot is shown in Figure 3.4.

```
x = -pi:pi/20:pi;
y1 = sin(x);
y2 = cos(x);
plot(x,y1,'b-');
hold on;
plot(x,y2,'k--');
hold off;
legend ('sin x','cos x');
```

3.5.3 Creating Multiple Figures

MATLAB can create multiple Figure Windows, with different data displayed in each window. Each Figure Window is identified by a *figure number*, which is a small positive integer. The first Figure Window is Figure 1, the second is Figure 2, and so forth. One of the Figure Windows will be the **current figure**, and all new plotting commands will be displayed in that window.

The current figure is selected with the **figure** function. This function takes the form “`figure(n)`”, where *n* is a figure number. When this command is executed, Figure *n* becomes the current figure and is used for all plotting commands. The figure is automatically created if it does not already exist. The cur-

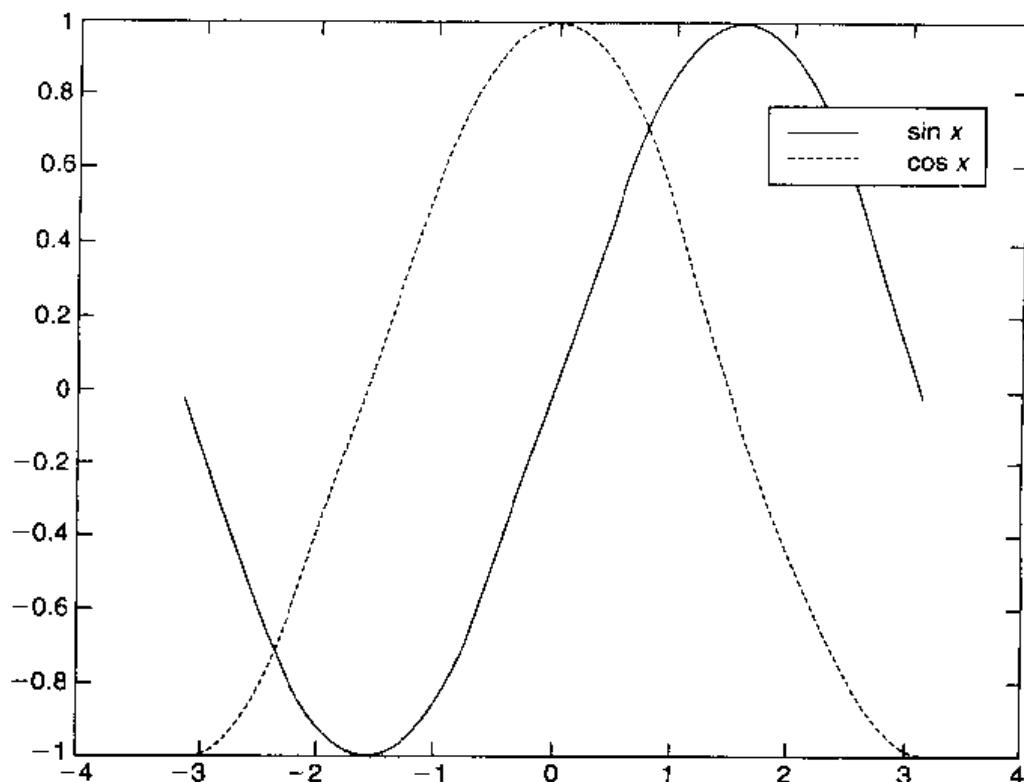


Figure 3.4 Multiple curves plotted on a single set of axes using the `hold` command.

rent figure can also be selected by clicking on it with the mouse.

The function `gcf` returns the number of the current figure. This function can be used by an M-file if it needs to know the current figure number.

The following commands illustrate the use of the `figure` function. They create two figures, displaying e^x in the first figure and e^{-x} in the second one.

```
figure(1)
x = 0:0.05:2;
y1 = exp(x);
plot(x,y1);
figure(2)
y2 = exp(-x);
plot(x,y2);
```

3.5.4 Subplots

It is possible to place more than one set of axes on a single figure, creating multiple **subplots**. Subplots are created with a `subplot` command of the form

```
subplot(m, n, p)
```

This command creates $m \times n$ subplots in current figure, arranged in m rows and n columns, and selects subplot p to receive all current plotting commands. The

subplots are numbered from left to right and from top to bottom. For example, the command `subplot(2, 3, 4)` would create six subplots in the current figure, and make subplot 4 (the lower left one) the current subplot.

If a `subplot` command creates a new set of axes that conflicts with a previously existing set, then the older axes are automatically deleted.

The commands shown below create two subplots within a single window, and display separate graphs in each subplot. The resulting figure is shown in Figure 3.5.

```
figure(1)
subplot(2,1,1)
x = -pi:pi/20:pi;
y = sin(x);
plot(x,y);
title('Subplot 1 title');
subplot(2,1,2)
x = -pi:pi/20:pi;
y = cos(x);
plot(x,y);
title('Subplot 2 title');
```

3.5.5 Enhanced Control of Plotted Lines

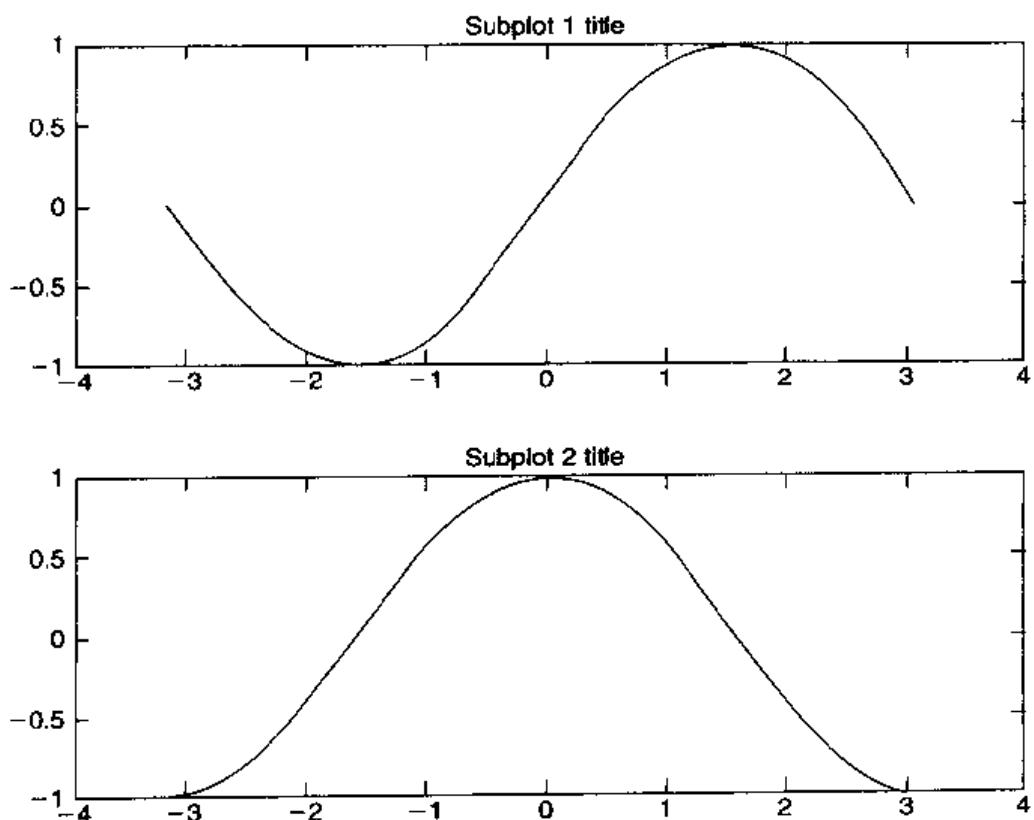


Figure 3.5 A plot containing multiple subplots.

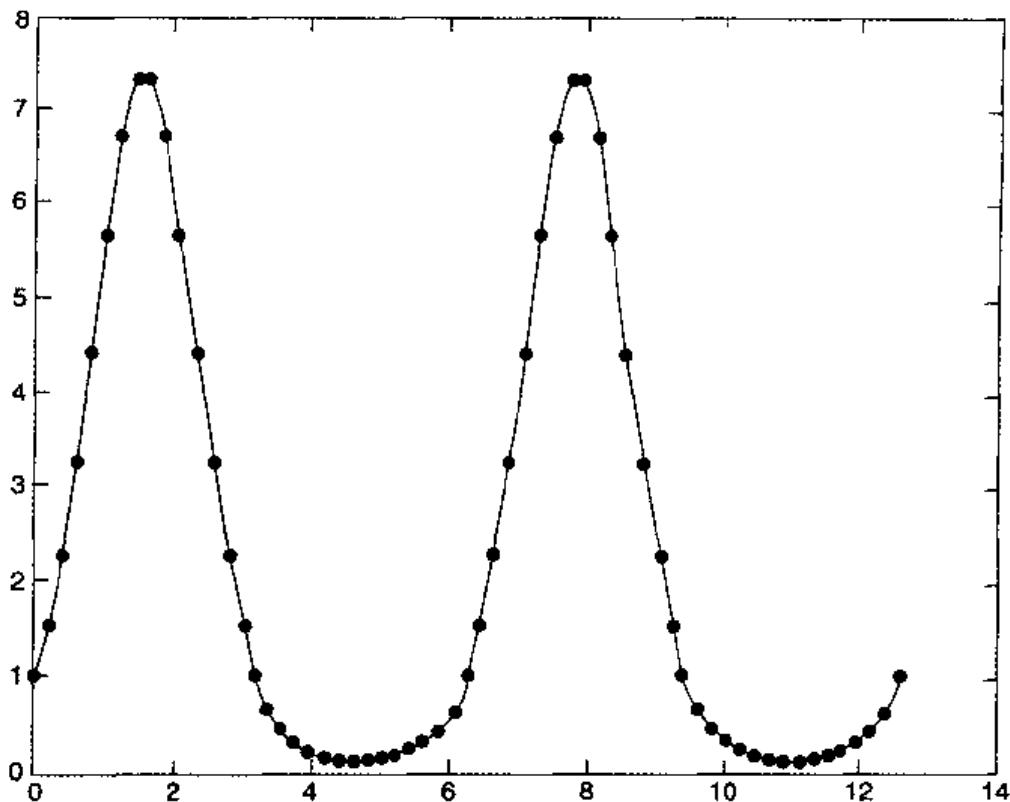


Figure 3.6 A plot illustrating the use of the `LineWidth` and `Marker` properties (a color version of this plot appears in the insert).

In Chapter 1, we learned how to set the color, style, and marker type for a line. It is also possible to set four additional properties associated with each line.

- `LineWidth` – Specifies the width of each line in points.
- `MarkerEdgeColor` – Specifies the color of the marker or the edge color for filled markers.
- `MarkerFaceColor` – Specifies the color of the face of filled markers.
- `MarkerSize` – Specifies the size of the marker in points.

These properties are specified in the `plot` command after the data to be plotted in the following fashion.

```
plot(x,y,'PropertyName',value,...)
```

For example, the following command plots a 3-point wide solid black line with 6-point wide circular markers at the data points. Each marker has a red edge and a green center, as shown in Figure 3.6 (see the color insert).

```
x = 0:pi/15:4*pi;
y = exp(2*sin(x));
plot(x,y,'ko','LineWidth',3.0,'MarkerSize',6,...
'MarkerEdgeColor','r','MarkerFaceColor','g')
```

3.5.6 Enhanced Control of Text Strings

It is possible to enhance plotted text strings (titles, axis labels, etc.) with formatting such as boldface, italics, or both, as well as with special characters such as Greek and mathematical symbols.

The font used to display the text can be modified by **stream modifiers**. A stream modifier is a special sequence of characters that tells the MATLAB interpreter to change its behavior. The most common stream modifiers are:

- `\bf` – Boldface
- `\it` – Italics
- `\rm` – Restore normal font.
- `\fontname{fontname}` – Specify the font name to use.
- `\fontsize{fontsize}` – Specify font size.
- `_{xxx}` – The characters inside the braces are subscripts.
- `\^{}{xxx}` – The characters inside the braces are superscripts.

Once a stream modifier has been inserted into a text string, it will remain in effect until the end of the string or until cancelled. If a modifier is followed by brackets {}, only the text in the brackets is affected.

Special Greek and mathematical symbols can also be used in text strings. They are created by embedding special escape sequences into the text string. These escape sequences are a subset of the special sequences supported by the T_EX language. A sample of the possible escape codes is shown in Table 3.6 on the following page. The full set of possibilities is included in the MATLAB online documentation.

If one of the special escape characters \, {, }, _, or ^ must be printed, precede it by a backslash character.

The following examples illustrate the use of stream modifiers and special characters.

String	Result
<code>\tau_{ind} versus \omega_m(\it{tm})</code>	τ_{ind} versus ω_m
<code>\theta varies from 0\circ to 90\circ</code>	θ varies from 0° to 90°
<code>\bf{B}_s_{\it{ts}}</code>	B_s

3.5.7 Polar Plots

MATLAB includes a special function called `polar`, which plots data in polar coordinates. The basic form of this function is

```
polar(theta,r)
```

where `theta` is an array of angles in radians, and `r` is an array of distances. It is useful for plotting data that is intrinsically a function of angle.

Example 3.5—Cardioid Microphone

Most microphones designed for use on a stage are directional microphones, which are specifically built to enhance the signals received from the singer in

Table 3.6 Selected Greek and Mathematical Symbols

Character Sequence	Symbol	Character Sequence	Symbol	Character Sequence	Symbol
\alpha	α			\int	\int
\beta	β			\cong	\cong
\gamma	γ	\Gamma	Γ	\sim	\sim
\delta	δ	\Delta	Δ	\infty	∞
\epsilon	ϵ			\pm	\pm
\eta	η			\leq	\leq
\theta	θ			\geq	\geq
\lambda	λ	\Lambda	Λ	\neq	\neq
\mu	μ			\propto	\propto
\nu	ν			\div	\div
\pi	π	\Pi	Π	\circ	\circ
\phi	ϕ			\rightarrowleftarrow	\leftrightarrow
\rho	ρ			\leftarrow	\leftarrow
\sigma	σ	\Sigma	Σ	\rightarrow	\rightarrow
\tau	τ			\uparrow	\uparrow
\omega	ω	\Omega	Ω	\downarrow	\downarrow

front of the microphone while suppressing the audience noise from behind the microphone. The gain of such a microphone varies as a function of angle according to the equation

$$Gain = 2g(1 + \cos \theta) \quad (3.3)$$

where g is a constant associated with a particular microphone, and θ is the angle from the axis of the microphone to the sound source. Assume that g is 0.5 for a particular microphone, and make a polar plot the gain of the microphone as a function of the direction of the sound source.

Solution

We must calculate the gain of the microphone versus angle and then plot it with a polar plot. The MATLAB code to do this is:

```
% Script file: microphone.m
%
% Purpose:
%   This program plots the gain pattern of a cardioid
%   microphone.
%
```

```
% Record of revisions:
%   Date      Programmer      Description of change
%   ===       ======      =====
%   12/10/98    S. J. Chapman  Original code
%
% Define variables:
%   g          -- Microphone gain constant
%   gain       -- Gain as a function of angle
%   theta      -- Angle from microphone axis (radians)

% Calculate gain versus angle
g = 0.5;
theta = 0:pi/20:2*pi;
gain = 2*g*(1+cos(theta));

% Plot gain
polar(theta,gain,'r-');
title ('\bfGain versus angle \theta');
```

The resulting plot is shown in Figure 3.7. Note that this type of microphone is called a cardioid microphone because its gain pattern is heart-shaped.

Example 3.6—Electrical Engineering: Frequency Response of a Low-Pass Filter

A simple low-pass filter circuit is shown in Figure 3.8. This circuit consists of a resistor and capacitor in series, and the ratio of the output voltage V_o to the input voltage V_i is given by the equation

$$\frac{V_o}{V_i} = \frac{1}{1 + j2\pi f RC} \quad (3.4)$$

where V_i is a sinusoidal input voltage of frequency f , R is the resistance in ohms, C is the capacitance in farads, and j is $\sqrt{-1}$ (electrical engineers use j instead of i for $\sqrt{-1}$, because the letter i is traditionally reserved for the current in a circuit).

Assume that the resistance $R = 16 \text{ k}\Omega$, and capacitance $C = 1 \mu\text{F}$, and plot the amplitude and frequency response of this filter.

Solution

The amplitude response of a filter is the ratio of the amplitude of the output voltage to the amplitude of the input voltage, and the phase response of the filter is the difference between the phase of the output voltage and the phase of the input voltage. The simplest way to calculate the amplitude and phase response of the filter is to evaluate Equation (3.4) at many different frequencies. The plot of the magnitude of Equation (3.4) versus frequency is the amplitude response of the filter, and the plot of the angle of Equation (3.4) versus frequency is the phase response of the filter.

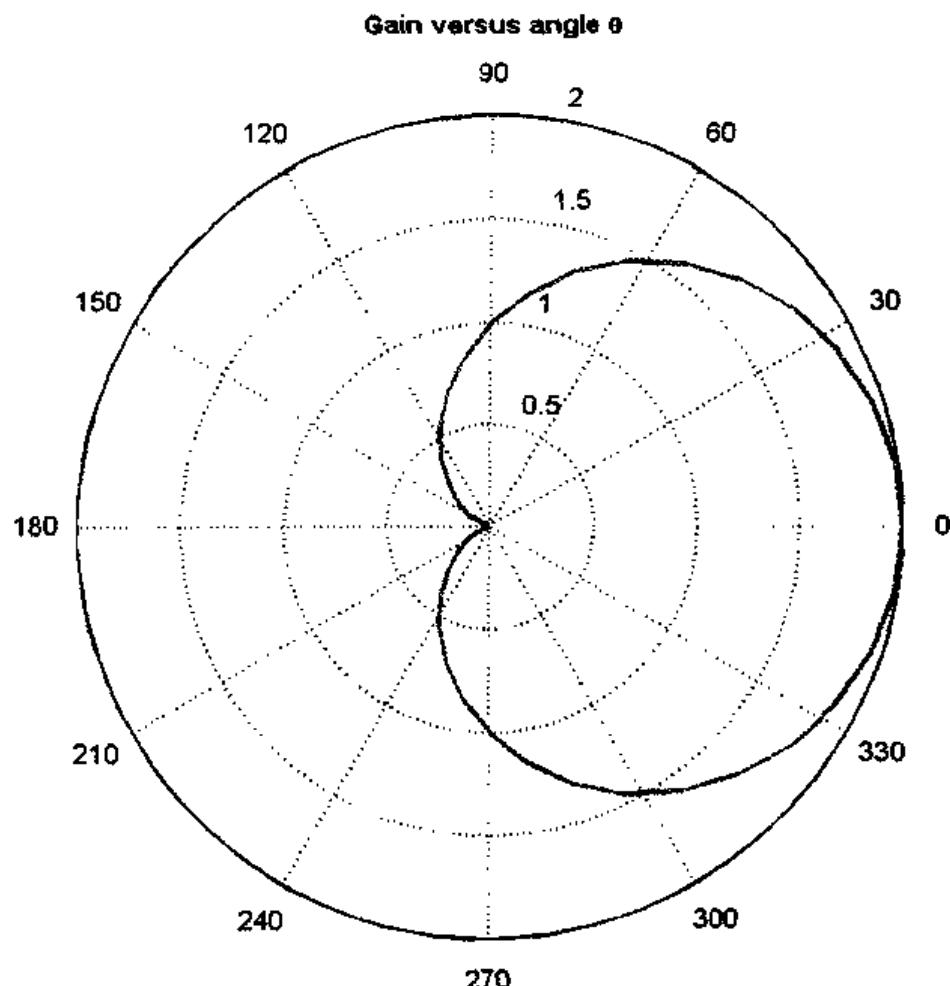


Figure 3.7 Gain of a cardioid microphone.

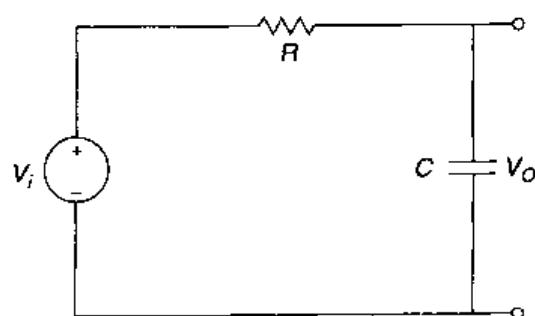


Figure 3.8 A simple low-pass filter circuit.

Because the frequency and amplitude response of a filter can vary over a wide range, it is customary to plot both of these values on logarithmic scales. On the other hand, the phase varies over a very limited range, so it is customary to plot the phase of the filter on linear scales. Therefore, we will use a log-log plot

for the amplitude response, and a `semilogx` plot for the phase response of the filter. We will display both responses as two subplots within a figure.

The MATLAB code required to create and plot the responses is shown below.

```
% Script file: plot_filter.m
%
% Purpose:
%   This program plots the amplitude and phase responses
%   of a low-pass RC filter.
%
% Record of revisions:
%   Date        Programmer      Description of change
%   ===         ======         =====
%   12/29/98    S. J. Chapman  Original code
%
% Define variables:
%   amp        -- Amplitude response
%   C          -- Capacitance (farads)
%   f          -- Frequency of input signal (Hz)
%   phase      -- Phase response
%   R          -- Resistance (ohms)
%   res        -- Vo/Vi

% Initialize R & C
R = 16000;           % 16 k ohms
C = 1.0E-6;          % 1 uF

% Create array of input frequencies
f = 1:2:1000;

% Calculate response
res = 1 ./ ( 1 + j*2*pi*f*R*C );

% Calculate amplitude response
amp = abs(res);

% Calculate phase response
phase = angle(res);

% Create plots
subplot(2,1,1);
loglog( f, amp );
title('Amplitude Response');
xlabel('Frequency (Hz)');
ylabel('Output/Input Ratio');
grid on;

subplot(2,1,2);
```

```

semilogx( f, phase );
title('Phase Response');
xlabel('Frequency (Hz)');
ylabel('Output-Input Phase (rad)');
grid on;

```

The resulting amplitude and phase responses are shown in Figure 3.9. Note that this circuit is called a low-pass filter because low frequencies are passed through with little attenuation, while high frequencies are strongly attenuated.

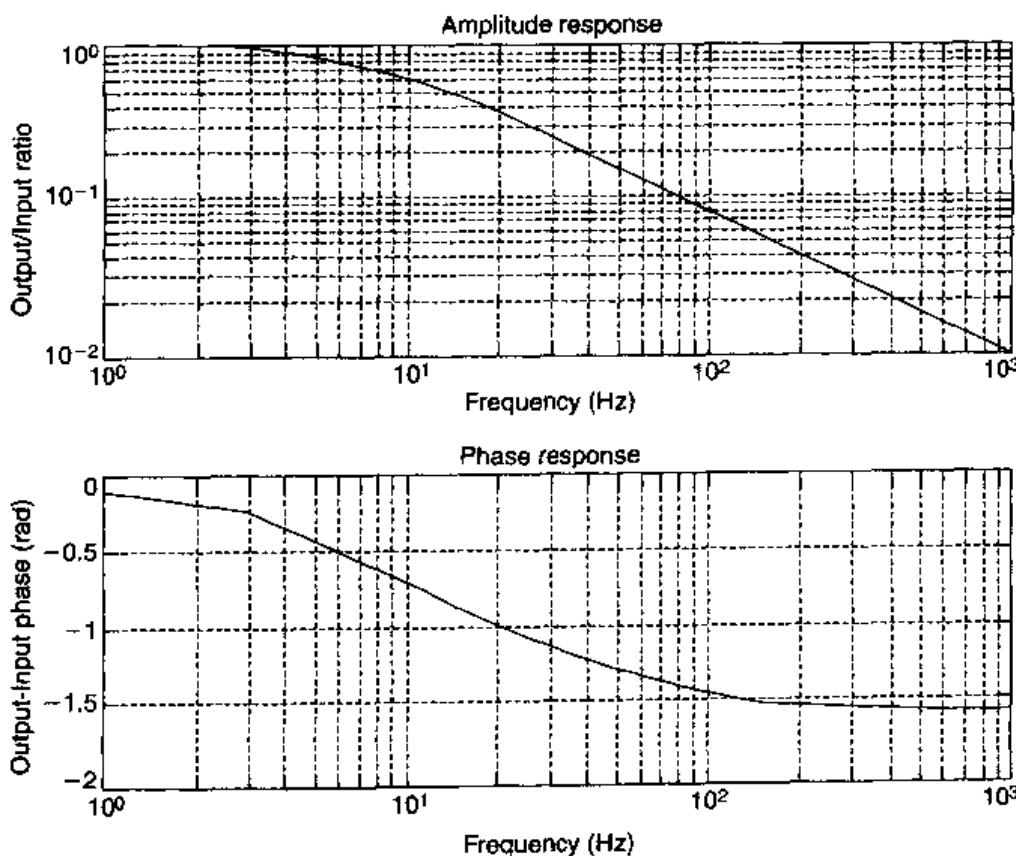


Figure 3.9 The amplitude and phase response of the low-pass filter circuit.

Example 3.7—Thermodynamics: The Ideal Gas Law

An ideal gas is one in which all collisions between molecules are perfectly elastic. It is possible to think of the molecules in an ideal gas as perfectly hard billiard balls that collide and bounce off each other without losing kinetic energy.

Such a gas can be characterized by three quantities: absolute pressure (P), volume (V), and absolute temperature (T). The relationship among these quantities in an ideal gas is known as the Ideal Gas Law:

$$PV = nRT \quad (3.5)$$

where P is the pressure of the gas in kilopascals (kPa), V is the volume of the gas in liters (L), n is the number of molecules of the gas in units of moles (mol), R is the universal gas constant (8.314 L·kPa/mol K), and T is the absolute temperature in kelvins (K). (Note: 1 mol = 6.02×10^{23} molecules.)

Assume that a sample of an ideal gas contains 1 mole of molecules at a temperature of 273 K, and answer the following questions.

- (a) How does the volume of this gas vary as its pressure varies from 1 to 1000 kPa? Plot pressure versus volume for this gas on an appropriate set of axes. Use a solid red line, with a width of 2 pixels.
- (b) Suppose that the temperature of the gas is increased to 373 K. How does the volume of this gas vary with pressure now? Plot pressure versus volume for this gas on the same set of axes as part (a). Use a dashed blue line, with a width of 2 pixels.

Include a bold title and x - and y -axis labels on the plot, as well as legends for each line.

Solution

The values that we want to plot both vary by a factor of 1000, so an ordinary linear plot will not produce a useful plot. Therefore, we will plot the data on a log-log scale.

Note that we must plot two curves on the same set of axes, so we must issue the command `hold on` after the first one is plotted, and `hold off` after the plot is complete. We also have to specify the color, style, and width of each line and specify that labels be in bold.

A program that calculates the volume of the gas as a function of pressure and creates the appropriate plot is shown below. Note that the special features controlling the style of the plot are shown in bold.

```
% Script file: ideal_gas.m
%
% Purpose:
%   This program plots the pressure versus volume of an
%   ideal gas.
%
% Record of revisions:
%   Date      Programmer          Description of change
%   =====  ======  =====
%   07/17/00  S. J. Chapman      Original code
```

```

% Define variables:
% n           -- Number of atoms (mol)
% P           -- Pressure (kPa)
% R           -- Ideal gas constant (L kPa/mol K)
% T           -- Temperature (K)
% V           -- volume (L)

% Initialize nRT
n = 1;                      % Moles of atoms
R = 8.314;                   % Ideal gas constant
T = 273;                     % Temperature (K)

% Create array of input pressures. Note that this
% array must be quite dense to catch the major
% changes in volume at low pressures.
P = 1:0.1:1000;

% Calculate volumes
V = (n * R * T) ./ P;

% Create first plot
figure(1);
loglog(P, V, 'r-', 'LineWidth', 2);
title('Volume vs Pressure in an Ideal Gas');
xlabel('Pressure (kPa)');
ylabel('Volume (L)');
grid on;
hold on;

% Now increase temperature
T = 373;                     % Temperature (K)

% Calculate volumes
V = (n * R * T) ./ P;

% Add second line to plot
figure(1);
loglog(P, V, 'b--', 'LineWidth', 2);
hold off;

% Add legend
legend('T = 273 K', 'T = 373 K');

```

The resulting volume versus pressure plot is shown in Figure 3.10.

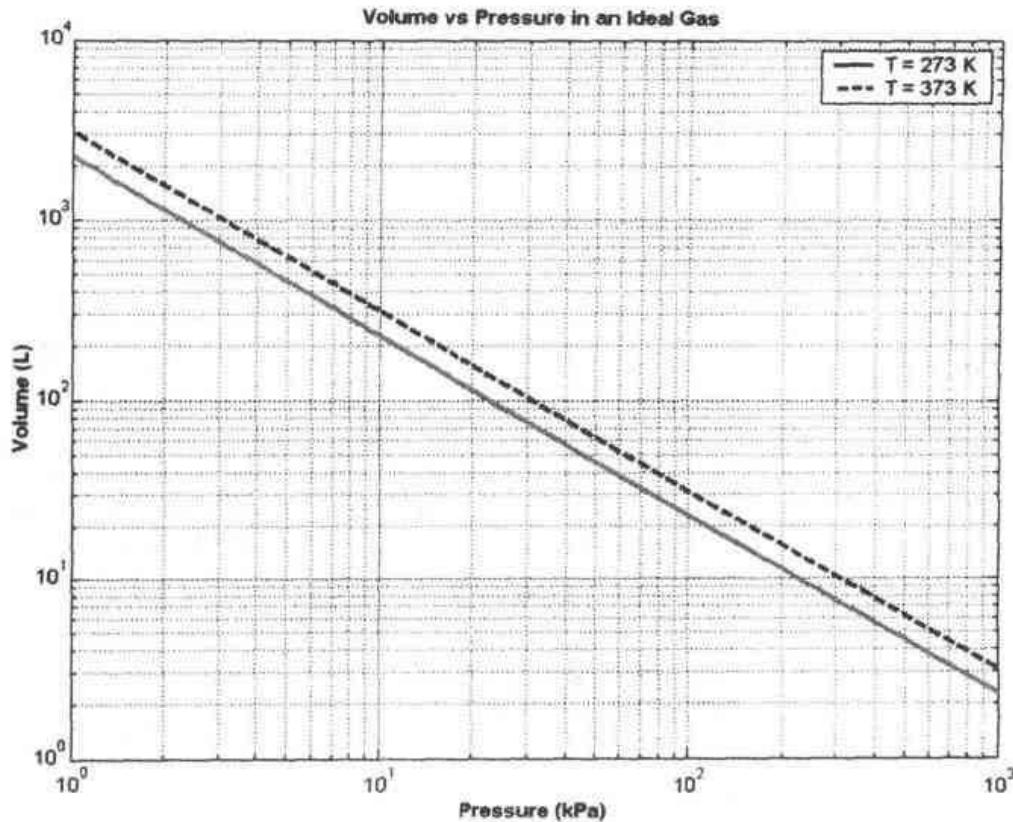


Figure 3.10 Pressure versus volume for an ideal gas.

3.5.8 Annotating and Saving Plots

Once a plot has been created by a MATLAB program, a user can edit and annotate the plot using the GUI-based tools available from the plot toolbar. Figure 3.11 shows the available tools, which allow the user to add lines, arrows, and text. When the Editing button () is selected from the toolbar, the annotation and editing tools become available for use.

Also, when the Editing button is depressed, clicking any line or text on the figure will cause it to be selected for editing, and double-clicking the line or text will open a properties window that allows you to modify any or all of the characteristics of that object. Figure 3.12 shows Figure 3.10 after a user has edited it to change the blue line to a 3-pixel-wide dashed line and added an arrow and comment.

When the plot has been edited and annotated, you can save the entire plot in a modifiable form using the "File/Save As" menu item from the Figure Window. The resulting figure file (*.fig) contains all the information required to re-create the figure plus annotations at any time in the future.

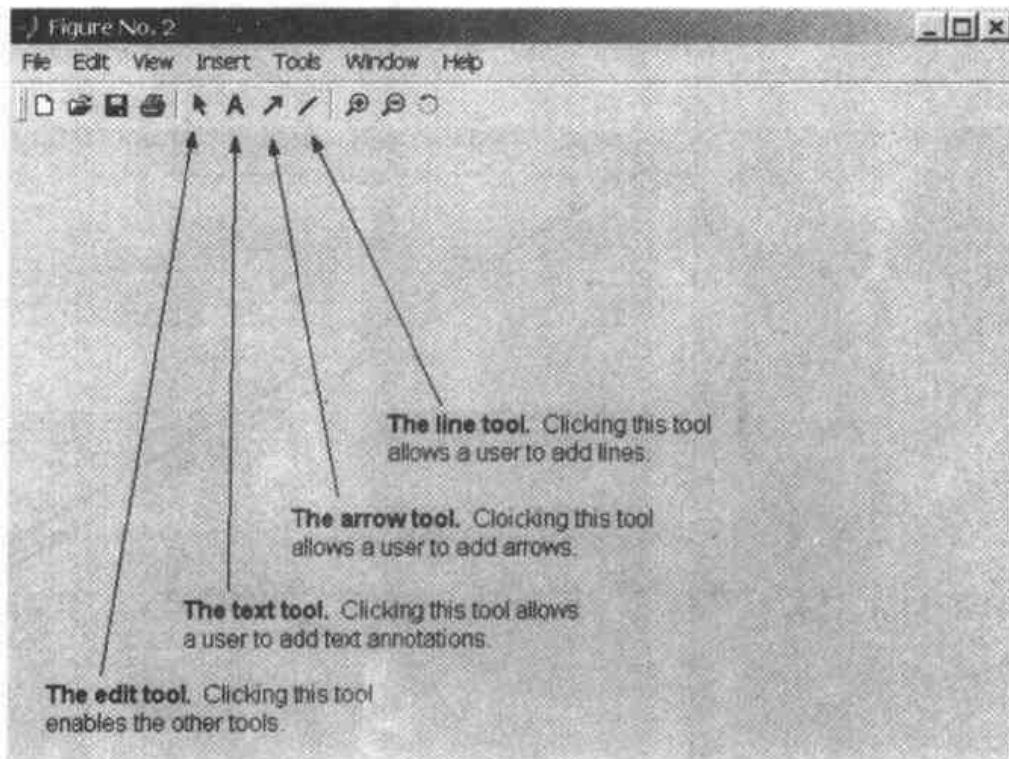


Figure 3.11 The editing tools on the Figure toolbar.

Quiz 3.3

This quiz provides a quick check to see if you have understood the concepts introduced in Section 3.5. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. Write the MATLAB statements required to plot $\sin x$ versus $\cos 2x$ from 0 to 2π in steps of $\pi/10$. The points should be connected by a 2-pixel-wide red line, and each point should be marked with a 6-pixel-wide blue circular marker.
2. Use the Figure editing tools to change the markers on the previous plot into black squares. Add an arrow and annotation pointing to the location $x = \pi$ on the plot.

Write the MATLAB text string that will produce the following expressions:

3. $f(x) = \sin \theta \cos 2\phi$
4. Plot of $\sum x^2$ versus x

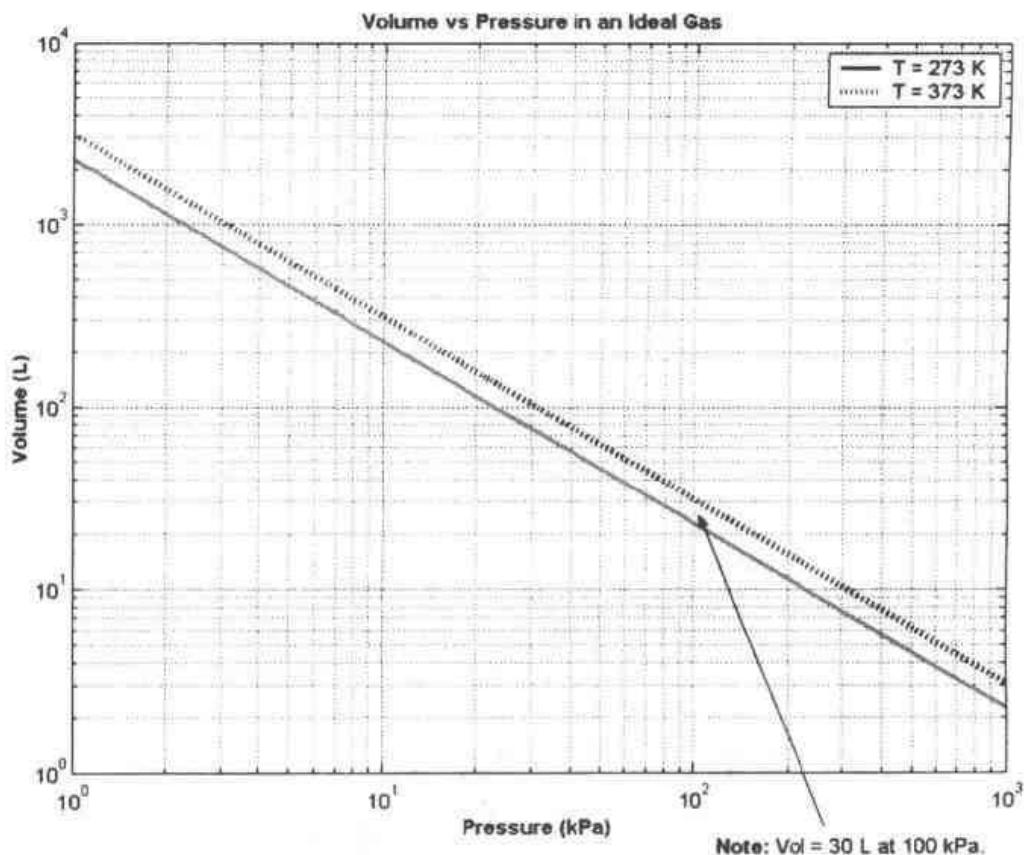


Figure 3.12 Figure 3.10 after the blue line has been modified and an annotation has been added using the editing tools built into the Figure toolbar.

Write the expression produced by the following text strings:

5. '\tau\it_{m}'
6. '\bf\it{x_{1}}^{2} + x_{2}^{2} \rm{units:}\n\bf m^{2}\rm{'}
7. How do you display the backslash (\) character in a text string?

3.6 More on Debugging MATLAB Programs

It is much easier to make a mistake when writing a program containing branches and loops than it is when writing simple sequential programs. Even after going through the full design process, a program of any size is almost guaranteed not to be completely correct the first time it is used. Suppose that we have built the program and tested it, only to find that the output values are in error. How do we go about finding the bugs and fixing them?

The screenshot shows a MATLAB editor window titled 'd:\book\matlab\2e\rev1\chap3\calc_roots.m'. The window contains the following MATLAB code:

```

23 % Prompt the user for the coefficients of the equation
24 disp ('This program solves for the roots of a quadratic ');
25 disp ('equation of the form A*X^2 + B*X + C = 0. ');
26 a = input ('Enter the coefficient A: ');
27 b = input ('Enter the coefficient B: ');
28 c = input ('Enter the coefficient C: ');

29 % Calculate discriminant
30 discriminant = b^2 - 4 * a * c;

31 % Solve for the roots, depending on the value of the discriminant
32 if discriminant > 0 % there are two real roots, so...
33
34 x1 = (-b + sqrt(discriminant)) / (2 * a);
35 x2 = (-b - sqrt(discriminant)) / (2 * a);
36 disp ('This equation has two real roots:');
37 fprintf ('x1 = %f\n', x1);
38 fprintf ('x2 = %f\n', x2);

39 elseif discriminant == 0 % there is one repeated root, so...
40
41 x1 = (-b) / (2 * a);
42 disp ('This equation has two identical real roots:');
43 fprintf ('x1 = x2 = %f\n', x1);

44 else % there are complex roots, so ...
45
46 real_part = (-b) / (2 * a);
47 imag_part = sqrt (abs (discriminant)) / (2 * a);
48 disp ('This equation has complex roots:');
49 fprintf('x1 = %f +i %f\n', real_part, imag_part);
50 fprintf('x1 = %f -i %f\n', real_part, imag_part);
51
52 end

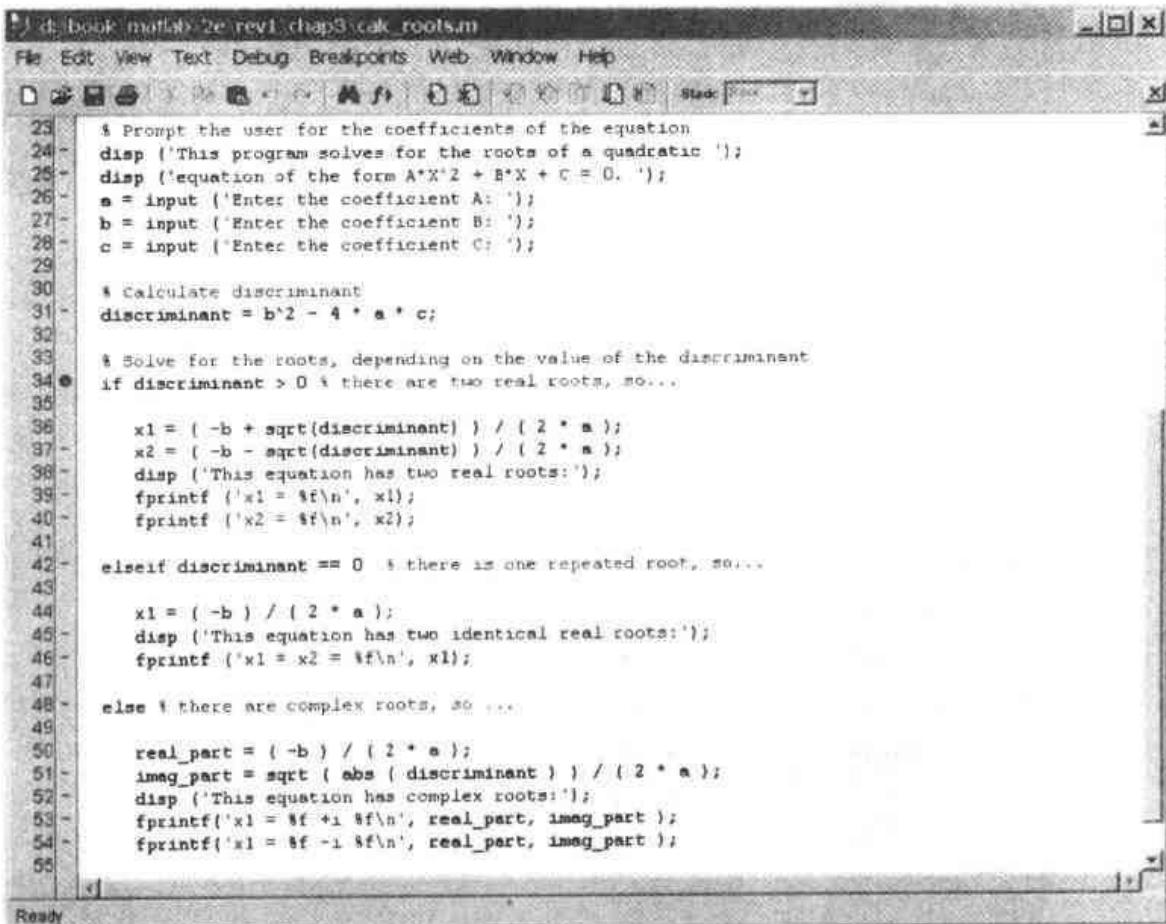
```

Figure 3.13 An Edit / Debug window with a MATLAB program loaded.

Once programs start to include loops and branches, the best way to locate an error is to use the symbolic debugger supplied with MATLAB. This debugger is integrated with the MATLAB editor.

To use the debugger, first open the file that you would like to debug using the “File/Open” menu selection in the MATLAB Command Window. When the file is opened, it is loaded into the editor and the syntax is automatically color-coded. Comments in the file appear in green, variables and numbers appear in black, character strings appear in red, and language keywords appear in blue. Figure 3.13 shows an example Edit/Debug Window containing the file `calc_roots.m`.

Let's say that we would like to determine what happens when the program is executed. To do this, we can set one or more **breakpoints** by right-clicking the mouse on the lines of interest and choosing the “Set/Clear Breakpoint” option. When a breakpoint is set, a red dot appears to the left of that line containing the breakpoint, as shown in Figure 3.14.



The screenshot shows the MATLAB Editor window with the file `calc_roots.m` open. The code is a script for solving quadratic equations. A red dot is placed on the line number 34, indicating it is the current breakpoint. The code uses `input` to get coefficients from the user, calculates the discriminant, and then branches based on its value to calculate and print the roots.

```

1 % Prompt the user for the coefficients of the equation
2 disp ('This program solves for the roots of a quadratic ');
3 disp ('equation of the form A*X^2 + B*X + C = 0. ');
4 a = input ('Enter the coefficient A: ');
5 b = input ('Enter the coefficient B: ');
6 c = input ('Enter the coefficient C: ');
7
8 % Calculate discriminant
9 discriminant = b^2 - 4 * a * c;
10
11 % Solve for the roots, depending on the value of the discriminant
12 if discriminant > 0 % there are two real roots, so...
13
14     x1 = ( -b + sqrt(discriminant) ) / ( 2 * a );
15     x2 = ( -b - sqrt(discriminant) ) / ( 2 * a );
16     disp ('This equation has two real roots:');
17     fprintf ('x1 = %f\n', x1);
18     fprintf ('x2 = %f\n', x2);
19
20 elseif discriminant == 0 % there is one repeated root, so...
21
22     x1 = ( -b ) / ( 2 * a );
23     disp ('This equation has two identical real roots:');
24     fprintf ('x1 = x2 = %f\n', x1);
25
26 else % there are complex roots, so ...
27
28     real_part = ( -b ) / ( 2 * a );
29     imag_part = sqrt ( abs ( discriminant ) ) / ( 2 * a );
30     disp ('This equation has complex roots:');
31     fprintf('x1 = %f +i %f\n', real_part, imag_part );
32     fprintf('x1 = %f -i %f\n', real_part, imag_part );
33
34 end

```

Figure 3.14 The window after a breakpoint has been set. Note the red dot to the left of the line with the breakpoint.

Once the breakpoints have been set, execute the program as usual by typing `calc_roots` in the Command Window. The program will run until it reaches the first breakpoint and stop there. A green arrow will appear by the current line during the debugging process, as shown in Figure 3.15. Once the breakpoint is reached, the programmer can examine and/or modify any variable in the workspace by typing its name in the Command Window. When the programmer is satisfied with the program at that point, he or she can either step through the program a line at a time by repeatedly pressing F10, or else run to the next breakpoint by pressing F5. It is always possible to examine the values of any variable at any point in the program.

Another very important feature of the debugger is found on the Breakpoints menu. This menu includes the two items: “Stop if Error” and “Stop if Warning.” If an error is occurring in a program that causes it to crash or generate warning messages, the programmer can turn these items on and execute the program. The program will run to the point of the error or warning and stop there, allowing the

```

d:\book\matlab\2e\rev1\chap3\calc_rootsm.m
File Edit View Text Debug Breakpoints Web Window Help
File Edit View Text Debug Breakpoints Web Window Help Stack calc_rootsm
23 % Prompt the user for the coefficients of the equation
24 disp ('This program solves for the roots of a quadratic ');
25 disp ('equation of the form A*X^2 + B*X + C = 0. ');
26 a = input ('Enter the coefficient A: ');
27 b = input ('Enter the coefficient B: ');
28 c = input ('Enter the coefficient C: ');
29
30 % Calculate discriminant
31 discriminant = b^2 - 4 * a * c;
32
33 % Solve for the roots, depending on the value of the discriminant
34 if discriminant > 0 % there are two real roots, so...
35
36     x1 = (-b + sqrt(discriminant)) / (2 * a);
37     x2 = (-b - sqrt(discriminant)) / (2 * a);
38     disp ('This equation has two real roots:');
39     fprintf ('x1 = %f\n', x1);
40     fprintf ('x2 = %f\n', x2);
41
42 elseif discriminant == 0 % there is one repeated root, so...
43
44     x1 = (-b) / (2 * a);
45     disp ('This equation has two identical real roots:');
46     fprintf ('x1 = x2 = %f\n', x1);
47
48 else % there are complex roots, so ...
49
50     real_part = (-b) / (2 * a);
51     imag_part = sqrt (abs (discriminant)) / (2 * a);
52     disp ('This equation has complex roots:');
53     fprintf('x1 = %f +i %f\n', real_part, imag_part );
54     fprintf('x1 = %f -i %f\n', real_part, imag_part );
55

```

Figure 3.15 A yellow arrow will appear by the current line during the debugging process.

programmer to examine the values of variables and see exactly what is causing the problem.

When a bug is found, the programmer can use the Editor to correct the MATLAB program and save the modified version to disk. Note that all breakpoints may be lost when the program is saved to disk, so they might have to be set again before debugging can continue. This process is repeated until the program appears to be bug-free.

Take some time now to become familiar with the debugger—it is a very worthwhile investment.

3.7 Summary

In Chapter 3, we presented the basic types of MATLAB branches and the relational and logic operations used to control them. The principal type of branch is the **if** construct. This construct is very flexible. It can have as many **elseif**

clauses as needed to construct any desired test. Furthermore, `if` constructs can be nested to produce more complex tests. A second type of branch is the `switch` construct. It can be used to select among mutually exclusive alternatives specified by a control expression. A third type of branch is the `try/catch` construct. It is used to trap errors that might occur during execution.

Chapter 3 also included additional information about plots. The `axis` command allows a programmer to select the specific range of x and y data to be plotted. The `hold` command allows later plots to be plotted on top of earlier ones so that elements can be added to a graph a piece at a time. The `figure` command allows the programmer to create and select among multiple Figure Windows so that a program can create multiple plots in separate windows. The `subplot` command allows the programmer to create and select among multiple plots within a single Figure Window.

In addition, we learned how to control additional characteristics of our plots, such as the line width and marker color. These properties can be controlled by specifying '`PropertyName`',`value` pairs in the `plot` command after the data to be plotted.

Text strings in plots can be enhanced with stream modifiers and escape sequences. Stream modifiers allow a programmer to specify features like `boldface`, `italic`, `superscripts`, `subscripts`, font size, and font name. Escape sequences allow the programmer to include special characters such as Greek and mathematical symbols in the text string.

The MATLAB symbolic debugger makes debugging MATLAB code much easier. You should invest some time to become familiar with this tool.

3.7.1 Summary of Good Programming Practice

The following guidelines should be adhered to when programming with branch or loop constructs. By following them consistently, your code will contain fewer bugs, will be easier to debug, and will be more understandable to others who may need to work with it in the future.

1. Be cautious about testing for equality with numeric values, since round-off errors may cause two variables that should be equal to fail a test for equality. Instead, test to see if the variables are *nearly* equal within the roundoff error to be expected on the computer you are using.
2. Follow the steps of the program design process to produce reliable, understandable MATLAB programs.
3. Always indent code blocks in `if` and `switch` constructs to make them more readable.

3.7.2 MATLAB Summary

The following summary lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one.

Commands and Functions

<code>axis</code>	(a) Set the x and y limits of the data to be plotted. (b) Get the x and y limits of the data to be plotted. (c) Set other axis-related properties.
<code>figure</code>	Select a Figure Window to be the current Figure Window. If the selected Figure Window does not exist, it is automatically created.
<code>hold</code>	Allows multiple plot commands to write on top of each other.
<code>if construct</code>	Selects a block of statements to execute if a specified condition is satisfied.
<code>ischar(a)</code>	Returns a 1 if a is a character array and a 0 otherwise.
<code>isempty(a)</code>	Returns a 1 if a is an empty array and a 0 otherwise.
<code>isinf(a)</code>	Returns a 1 if the value of a is infinite (<code>Inf</code>) and a 0 otherwise.
<code>isnan(a)</code>	Returns a 1 if the value of a is <code>Nan</code> (not a number) and a 0 otherwise.
<code>isnumeric(a)</code>	Returns a 1 if the a is a numeric array and a 0 otherwise.
<code>polar</code>	Create a polar plot.
<code>subplot</code>	Select a subplot in the current Figure Window. If the selected subplot does not exist, it is automatically created. If the new subplot conflicts with a previously existing set of axes, they are automatically deleted.
<code>switch construct</code>	Selects a block of statements to execute from a set of mutually exclusive choices based on the results of a single expression.
<code>try / catch construct</code>	A special construct used to trap errors. It executes the code in the <code>try</code> block. If an error occurs, execution stops immediately and transfers to the code in the <code>catch</code> construct.

3.8 Exercises

- 3.1 The tangent function is defined as $\tan \theta = \sin \theta / \cos \theta$. This expression can be evaluated to solve for the tangent as long as the magnitude of $\cos \theta$ is not too near to 0. (If $\cos \theta$ is 0, evaluating the equation for $\tan \theta$ will produce the non-numerical value `Inf`.) Assume that θ is given in *degrees*, and write the MATLAB statements to evaluate $\tan \theta$ as long as the magnitude of $\cos \theta$ is greater than or equal to 10^{-20} . If the magnitude of $\cos \theta$ is less than 10^{-20} , write out an error message instead.

- 3.2** The following statements are intended to alert a user to dangerously high oral thermometer readings (values are in degrees Fahrenheit). Are they correct or incorrect? If they are incorrect, explain why and correct them.

```

if temp < 97.5
    disp('Temperature below normal');
elseif temp > 97.5
    disp('Temperature normal');
elseif temp > 99.5
    disp('Temperature slightly high');
elseif temp > 103.0
    disp('Temperature dangerously high');
end

```

- 3.3** The cost of sending a package by an express delivery service is \$10.00 for the first two pounds, and \$3.75 for each pound or fraction thereof over two pounds. If the package weighs more than 70 pounds, a \$10.00 excess weight surcharge is added to the cost. No package over 100 pounds will be accepted. Write a program that accepts the weight of a package in pounds and computes the cost of mailing the package. Be sure to handle the case of overweight packages.
- 3.4** In Example 3.3, we wrote a program to evaluate the function $f(x,y)$ for any two user-specified values x and y , where the function $f(x,y)$ was defined as follows.

$$f(x,y) = \begin{cases} x + y & x \geq 0 \text{ and } y \geq 0 \\ x + y^2 & x \geq 0 \text{ and } y < 0 \\ x^2 + y & x < 0 \text{ and } y \geq 0 \\ x^2 + y^2 & x < 0 \text{ and } y < 0 \end{cases}$$

The problem was solved by using a single `if` construct with 4 code blocks to calculate $f(x,y)$ for all possible combinations of x and y . Rewrite program `funxy` to use nested `if` constructs, where the outer construct evaluates the value of x and the inner constructs evaluate the value of y .

- 3.5** Write a MATLAB program to evaluate the function

$$y(x) = \ln \frac{1}{1-x}$$

for any user-specified value of x , where x is a number < 1 (note that `ln` is the natural logarithm, the logarithm to the base e). Use an `if` structure to verify that the value passed to the program is legal. If the value of x is legal, calculate $y(x)$. If not, write a suitable error message and quit.

- 3.6** Write a program that allows a user to enter a string containing a day of the week ("Sunday," "Monday," "Tuesday," etc.) and uses a `switch` construct to convert the day to its corresponding number, where Sunday is considered the first day of the week and Saturday is considered the last day of the week. Print out the resulting day number. Also, be sure to handle the

case of an illegal day name! (*Note:* Be sure to use the ‘s’ option on function input so that the input is treated as a string.)

- 3.7 Ideal Gas Law.** The Ideal Gas Law was defined in Example 3.7. Assume that the volume of 1 mole of this gas is 10 L, and plot the pressure of the gas as a function of temperature as the temperature is changed from 250 to 400 kelvins. What sort of plot (linear, semilogx, etc.) is most appropriate for this data?
- 3.8 Antenna Gain Pattern.** The gain G of a certain microwave dish antenna can be expressed as a function of angle by the equation

$$G(\theta) = |\operatorname{sinc} 4\theta| \quad \text{for } -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2} \quad (3.6)$$

where θ is measured in radians from the boresite of the dish, and $\operatorname{sinc} x = \sin x/x$. Plot this gain function on a polar plot, with the title “**Antenna Gain vs θ** ” in boldface.

- 3.9 Refraction.** When a ray of light passes from a region with an index of refraction n_1 into a region with a different index of refraction n_2 , the light ray is bent (see Figure 3.16). The angle at which the light is bent is given by Snell’s law.

$$n_1 \sin \theta_1 = n_2 \sin \theta_2 \quad (3.7)$$

where θ_1 is the angle of incidence of the light in the first region, and θ_2 is the angle of incidence of the light in the second region. Using Snell’s law, it is possible to predict the angle of incidence of a light ray in Region 2 if the angle of incidence θ_1 in Region 1 and the indices of refraction n_1 and n_2 are known. The equation to perform this calculation is

$$\theta_2 = \sin^{-1}\left(\frac{n_1}{n_2} \sin \theta_1\right) \quad (3.8)$$

Write a program to calculate the angle of incidence (in degrees) of a light ray in Region 2 given the angle of incidence θ_1 in Region 1 and the indices of refraction n_1 and n_2 . (*Note:* If $n_1 > n_2$, then for some angles θ_1 , Equation 3.8 will have no real solution because the absolute value of the quantity $\left(\frac{n_1}{n_2} \sin \theta_1\right)$ will be greater than 1.0. When this occurs, all light is reflected back into Region 1, and no light passes into Region 2 at all. Your program must be able to recognize and properly handle this condition.)

The program should also create a plot showing the incident ray, the boundary between the two regions, and the refracted ray on the other side of the boundary.

Test your program by running it for the following two cases: (a) $n_1 = 1.0$, $n_2 = 1.7$, and $\theta_1 = 45^\circ$, (b) $n_1 = 1.7$, $n_2 = 1.0$, and $\theta_1 = 45^\circ$.

- 3.10** Assume that the complex function $f(t)$ is defined by the equation

$$f(t) = (0.5 - 0.25i)t - 1.0$$

Plot the amplitude and phase of function f for $0 \leq t \leq 4$.

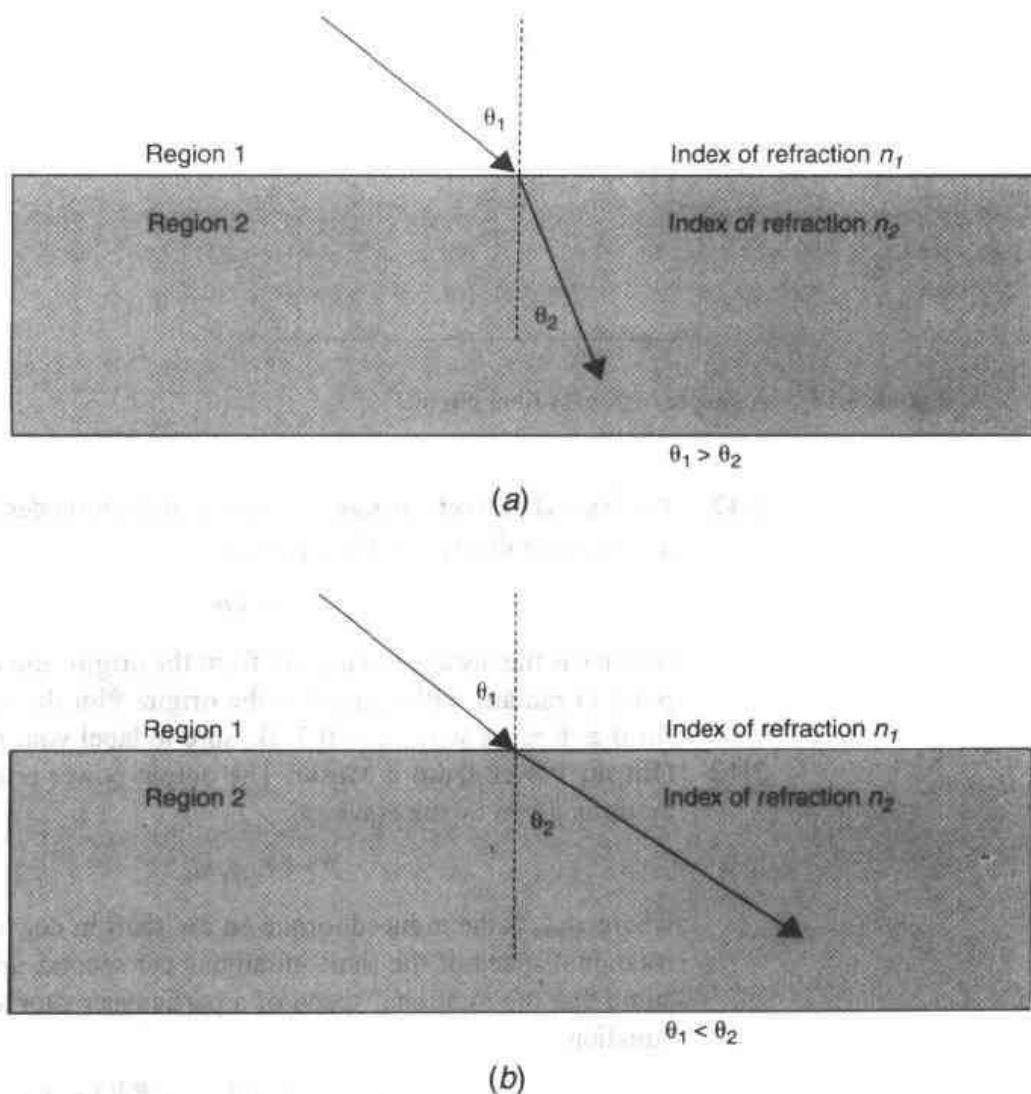


Figure 3.16 A ray of light bends as it passes from one medium into another one. (a) If the ray of light passes from a region with a low index of refraction into a region with a higher index of refraction, the ray of light bends more toward the vertical. (b) If the ray of light passes from a region with a high index of refraction into a region with a lower index of refraction, the ray of light bends away from the vertical.

3.11 High-Pass Filter. Figure 3.17 shows a simple high-pass filter consisting of a resistor and a capacitor. The ratio of the output voltage V_o to the input voltage V_i is given by the equation

$$\frac{V_o}{V_i} = \frac{j 2\pi f R C}{1 + j 2\pi f R C} \quad (3.9)$$

Assume that $R = 16 \text{ k}\Omega$ and $C = 1 \mu\text{F}$. Calculate and plot the amplitude and phase response of this filter as a function of frequency.

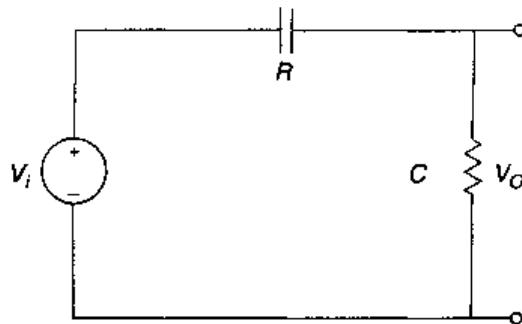


Figure 3.17 A simple high-pass filter circuit.

- 3.12 The Spiral of Archimedes.** The spiral of Archimedes is a curve described in polar coordinates by the equation

$$r = k\theta \quad (3.10)$$

where r is the distance of a point from the origin, and θ is the angle of that point in radians with respect to the origin. Plot the spiral of Archimedes for $0 \leq \theta \leq 6\pi$ when $k = 0.5$. Be sure to label your plot properly.

- 3.13 Output Power from a Motor.** The output power produced by a rotating motor is given by the equation

$$P = \tau_{\text{IND}} \omega_m \quad (3.11)$$

where τ_{IND} is the induced torque on the shaft in newton-meters, ω_m is the rotational speed of the shaft in radians per second, and P is in watts. Assume that the rotational speed of a particular motor shaft is given by the equation

$$\omega_m = 188.5(1 - e^{-0.2t}) \text{ rad/s}$$

and the induced torque on the shaft is given by

$$\tau_{\text{IND}} = 10e^{-0.2t} \text{ N} \cdot \text{m}$$

Plot the torque, speed, and power supplied by this shaft versus time for $0 \leq t \leq 10$ s. Be sure to label your plot properly with the symbols τ_{IND} and ω_m where appropriate. Create two plots, one with the power displayed on a linear scale, and one with the output power displayed on a logarithmic scale. Time should always be displayed on a linear scale.

- 3.14 Plotting Orbits** When a satellite orbits the Earth, the satellite's orbit will form an ellipse with the Earth located at one of the focal points of the ellipse. The satellite's orbit can be expressed in polar coordinates as

$$r = \frac{P}{1 - e \cos \theta} \quad (3.12)$$

where r and θ are the distance and angle of the satellite from the center of the Earth, p is a parameter specifying the size of the orbit, and ϵ is a parameter representing the eccentricity of the orbit. A circular orbit has an eccentricity ϵ of 0. An elliptical orbit has an eccentricity of $0 \leq \epsilon \leq 1$. If $\epsilon > 1$, the satellite follows a hyperbolic path and escapes from the Earth's gravitational field.

Consider a satellite with a size parameter $p = 1000$ km. Plot the orbit of this satellite if (a) $\epsilon = 0$; (b) $\epsilon = 0.25$; (c) $\epsilon = 0.5$. How close does each orbit come to the Earth? How far away does each orbit get from the Earth? Compare the three plots you created. Can you determine what the parameter p means from looking at the plots?

Loops

Loops are MATLAB constructs that permit us to execute a sequence of statements more than once. There are two basic forms of loop constructs: **while loops** and **for loops**. The major difference between these two types of loops is in how the repetition is controlled. The code in a **while loop** is repeated an indefinite number of times until some user-specified condition is satisfied. By contrast, the code in a **for loop** is repeated a specified number of times, and the number of repetitions is known before the loops starts.

4.1 The while Loop

A **while loop** is a block of statements that is repeated indefinitely as long as some condition is satisfied. The general form of a **while loop** is

```
while expression
    ...
    ...
}
end
```

If the *expression* is non-zero (true), the code block will be executed, and then control will return to the **while** statement. If the *expression* is still non-zero (true), the statements will be executed again. This process will be repeated until the *expression* becomes zero (false). When control returns to the **while** statement and the expression is zero, the program will execute the first statement after the **end**.

The pseudocode corresponding to a **while** loop is

```

while expr
    ...
    ...
    ...
end

```

We will now show an example statistical analysis program that is implemented using a `while` loop.

Example 4.1—Statistical Analysis

It is very common in science and engineering to work with large sets of numbers, each of which is a measurement of some particular property in which we are interested. A simple example would be the grades on the first test in this course. Each grade would be a measurement of how much a particular student has learned in the course to date.

Much of the time, we are not interested in looking closely at every single measurement that we make. Instead, we want to summarize the results of a set of measurements with a few numbers that tell us a lot about the overall data set. Two such numbers are the *average* (or *arithmetic mean*) and the *standard deviation* of the set of measurements. The average or arithmetic mean of a set of numbers is defined as

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i \quad (4.1)$$

where x_i is sample i out of N samples. If all of the input values are available in an array, the average of a set of numbers can be calculated either directly from Equation (4.1) or else by the built-in MATLAB function `mean`.

The standard deviation of a set of numbers is defined as

$$s = \sqrt{\frac{N \sum_{i=1}^N x_i^2 - \left(\sum_{i=1}^N x_i\right)^2}{N(N-1)}} \quad (4.2)$$

Standard deviation is a measure of the amount of scatter on the measurements; the greater the standard deviation, the more scattered are the points in the data set. If all of the input values are available in an array, the standard deviation of a set of numbers can be calculated either directly from Equation (4.2), or else by the built-in MATLAB function `std`. For the purposes of this example, we will calculate the mean and standard deviation directly from Equations (4.1) and (4.2), instead of using the built-in functions.

Implement an algorithm that reads in a set of measurements and calculates the mean and the standard deviation of the input data set.

Solution

This program must be able to read in an arbitrary number of measurements, and then calculate the mean and standard deviation of those measurements. We will use a while loop to accumulate the input measurements before performing the calculations.

When all of the measurements have been read, we must have some way of telling the program that there is no more data to enter. For now, we will assume that all the input measurements are either positive or zero, and we will use a negative input value as a *flag* to indicate that there is no more data to read. If a negative value is entered, then the program will stop reading input values and will calculate the mean and standard deviation of the data set.

1. State the problem

Since we assume that the input numbers must be positive or zero, a proper statement of this problem would be: *calculate the average and the standard deviation of a set of measurements, assuming that all of the measurements are either positive or zero, and assuming that we do not know in advance how many measurements are included in the data set. A negative input value will mark the end of the set of measurements.*

2. Define the inputs and outputs

The inputs required by this program are an unknown number of positive or zero numbers. The outputs from this program are a printout of the mean and the standard deviation of the input data set. In addition, we will print out the number of data points input to the program, since this is a useful check that the input data was read correctly.

3. Design the algorithm

This program can be broken down into three major steps.

Accumulate the input data
 Calculate the mean and standard deviation
 Write out the mean, standard deviation, and number of points

The first major step of the program is to accumulate the input data. To do this, we will have to prompt the user to enter the desired numbers. When the numbers are entered, we will have to keep track of the number of values entered, plus the sum and the sum of the squares of those values. The pseudocode for these steps is:

```

Initialize n, sum_x, and sum_x2 to 0
Prompt user for first number
Read in first x
while x >= 0
    n ← n + 1
    sum_x ← sum_x + x

```

```

sum_x2 ← sum_x2 + x^2
Prompt user for next number
Read in next x
end

```

Note that we have to read in the first value before the while loop starts so that the while loop can have a value to test the first time it executes.

Next, we must calculate the mean and standard deviation. The pseudocode for this step is just the MATLAB versions of Equations (4.1) and (4.2).

```

x_bar ← sum_x / n
std_dev ← sqrt((n*sum_x2 - sum_x^2) / (n*(n-1)))

```

Finally, we must write out the results.

```

Write out the mean value x_bar
Write out the standard deviation std_dev
Write out the number of input data points n

```

4. Turn the algorithm into MATLAB statements

The final MATLAB program follows:

```

% Script file: stats_1.m
%
% Purpose:
%   To calculate mean and the standard deviation of
%   an input data set containing an arbitrary number
%   of input values.
%
% Record of revisions:
%   Date        Programmer            Description of change
%   ===        ======            ======
%   12/05/98    S. J. Chapman      Original code
%
% Define variables:
%   n          -- The number of input samples
%   std_dev    -- The standard deviation of the input samples
%   sum_x      -- The sum of the input values
%   sum_x2     -- The sum of the squares of the input values
%   x          -- An input data value
%   xbar       -- The average of the input samples

% Initialize sums.
n = 0; sum_x = 0; sum_x2 = 0;

% Read in first value
x = input('Enter first value: ');

% While loop to read input values.
while x >= 0

```

```

% Accumulate sums.
n      = n + 1;
sum_x  = sum_x + x;
sum_x2 = sum_x2 + x^2;

% Read in next value
x = input('Enter next value: ');

end

% Calculate the mean and standard deviation
x_bar = sum_x / n;
std_dev = sqrt( (n * sum_x2 - sum_x^2) / (n * (n-1)) );

% Tell user.
fprintf('The mean of this data set is: %f\n', x_bar);
fprintf('The standard deviation is:    %f\n', std_dev);
fprintf('The number of data points is: %f\n', n);

```

5. Test the program

To test this program, we will calculate the answers by hand for a simple data set, and then compare the answers to the results of the program. If we used three input values: 3, 4, and 5, then the mean and standard deviation would be

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i = \frac{1}{3} 12 = 4$$

$$s = \sqrt{\frac{N \sum_{i=1}^N x_i^2 - \left(\sum_{i=1}^N x_i\right)^2}{N(N-1)}} = 1$$

When the above values are fed into the program, the results are

```

> stats_1
Enter first value: 3
Enter next value: 4
Enter next value: 5
Enter next value: -1
The mean of this data set is: 4.000000
The standard deviation is: 1.000000
The number of data points is: 3.000000

```

The program gives the correct answers for our test data set.

In this example, we failed to follow the design process completely. This failure has left the program with a fatal flaw! Did you spot it?

We have failed because *we did not completely test the program for all possible types of inputs*. Look at the example once again. If we enter either no num-

bers or only one number, then we will be dividing by zero in the above equations! The division-by-zero error will cause divide-by-zero warnings to be printed, and the output values will be NaN. We need to modify the program to detect this problem, tell the user what the problem is, and stop gracefully.

A modified version of the program called `stats_2` follows. Here, we check to see if there are enough input values before performing the calculations. If not, the program will print out an intelligent error message and quit. Test the modified program for yourself.

```
% Script file: stats_2.m
%
% Purpose:
%   To calculate mean and the standard deviation of
%   an input data set containing an arbitrary number
%   of input values.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   ===       ======        =====
%   12/05/98  S. J. Chapman  Original code
%   1. 12/06/98 S. J. Chapman  Correct divide-by-0 error if
%                               0 or 1 input values given.
%
% Define variables:
%   n          -- The number of input samples
%   std_dev    -- The standard deviation of the input samples
%   sum_x      -- The sum of the input values
%   sum_x2     -- The sum of the squares of the input values
%   x          -- An input data value
%   xbar       -- The average of the input samples

% Initialize sums.
n = 0; sum_x = 0; sum_x2 = 0;

% Read in first value
x = input('Enter first value: ');

% While loop to read input values.
while x >= 0

    % Accumulate sums.
    n      = n + 1;
    sum_x = sum_x + x;
    sum_x2 = sum_x2 + x^2;

    % Read in next value
    x = input('Enter next value: ');

end
```

```
% Check to see if we have enough input data.
if n < 2 % Insufficient information
    disp('At least 2 values must be entered!');
else % There is enough information, so
    % calculate the mean and standard deviation

    x_bar = sum_x / n;
    std_dev = sqrt( (n * sum_x2 - sum_x^2) / (n * (n-1)) );

    % Tell user.
    fprintf('The mean of this data set is: %f\n', x_bar);
    fprintf('The standard deviation is: %f\n', std_dev);
    fprintf('The number of data points is: %f\n', n);

end
```

Note that the average and standard deviation could have been calculated with the built-in MATLAB functions `mean` and `std` if all of the input values are saved in a vector and that vector is passed to these functions. You will be asked to create a version of the program that uses the standard MATLAB functions in an exercise at the end of this chapter.

4.2 The for Loop

The **for** loop is a loop that executes a block of statements a specified number of times. The for loop has the form

```
for index = expr
    Statement 1
    ...
    Statement n
end
```

} Body

where `index` is the loop variable (also known as the **loop index**) and `expr` is the loop control expression. The columns in `expr` are stored one at a time in the variable `index` and then the loop body is executed, so that the loop is executed once for each column in `expr`. The expression usually takes the form of a vector in shortcut notation `first:incr:last`.

The statements between the `for` statement and the `end` statement are known as the **body** of the loop. They are executed repeatedly during each pass of the `for` loop. The `for` loop construct functions as follows.

1. At the beginning of the loop, MATLAB generates the control expression.
2. The first time through the loop, the program assigns the first column of the expression to the loop variable `index`, and the program executes the statements within the body of the loop.

3. After the statements in the body of the loop have been executed, the program assigns the next column of the expression to the loop variable index, and the program executes the statements within the body of the loop again.
4. Step 3 is repeated over and over as long as there are additional columns in the control expression.

Let's look at a number of specific examples to make the operation of the `for` loop clearer. First, consider the following example.

```
for ii = 1:10
    Statement 1
    ...
    Statement n
end
```

In this case, the control expression generates a 1×10 array, so statements 1 through n will be executed 10 times. The loop index `ii` will be 1 the first time, 2 the second time, and so on. The loop index will be 10 during the last pass through the statements. When control is returned to the `for` statement after the tenth pass, there are no more columns in the control expression, so execution transfers to the first statement after the `end` statement. Note that the loop index `ii` is still set to 10 after the loop finishes executing.

Second, consider the following example.

```
for ii = 1:2:10
    Statement 1
    ...
    Statement n
END DO
```

In this case, the control expression generates a 1×5 array, so statements 1 through n will be executed 5 times. The loop index `ii` will be 1 the first time, 3 the second time, and so on. The loop index will be 9 during the fifth and last pass through the statements. When control is returned to the `for` statement after the fifth pass, there are no more columns in the control expression, so execution transfers to the first statement after the `end` statement. Note that the loop index `ii` is still set to 9 after the loop finishes executing.

Third, consider the following example.

```
for ii = [5 9 7]
    Statement 1
    ...
    Statement n
end
```

Here, the control expression is an explicitly written 1×3 array, so statements 1 through n will be executed 3 times with the loop index set to 5 the first time, 9 the second time, and 7 the final time. The loop index `ii` is still set to 7 after the loop finishes executing.

Finally, consider the example:

```
for ii = [1 2 3;4 5 6]
    Statement 1
    ...
    Statement n
end
```

In this case, the control expression is a 2×3 array, so statements 1 through n will be executed 3 times. The loop index *ii* will be the column vector $\begin{bmatrix} 1 \\ 4 \end{bmatrix}$ the first time, $\begin{bmatrix} 2 \\ 5 \end{bmatrix}$ the second time, and $\begin{bmatrix} 3 \\ 6 \end{bmatrix}$ the third time. This example illustrates the fact that a loop index can be a vector.

The pseudocode corresponding to a *for* loop looks like the loop itself.

```
for index = expression
    Statement 1
    ...
    Statement n
end
```

Example 4.2—The Factorial Function

To illustrate the operation of a *for* loop, we will use a *for* loop to calculate the factorial function. The factorial function is defined as

$$\begin{array}{ll} N! = 1 & N = 0 \\ N! = N * (N-1) * (N-2) * \dots * 3 * 2 * 1 & N > 0 \end{array}$$

The MATLAB code to calculate *N* factorial for positive value of *N* would be

```
n_factorial = 1
for ii = 1:n
    n_factorial = n_factorial * ii;
end
```

Suppose that we wish to calculate the value of $5!$. If *n* is 5, the *for* loop control expression would be the row vector [1 2 3 4 5]. This loop will be executed 5 times, with the variable *ii* taking on values of 1, 2, 3, 4, and 5 in the successive loops. The resulting value of *n_factorial* will be $1 \times 2 \times 3 \times 4 \times 5 = 120$.

Example 4.3—Calculating the Day of Year

The *day of year* is the number of days (including the current day) that have elapsed since the beginning of a given year. It is a number in the range 1 to 365 for ordinary years, and 1 to 366 for leap years. Write a MATLAB program that accepts a day, month, and year, and calculates the day of year corresponding to that date.

Solution

To determine the day of year, this program will need to sum up the number of days in each month preceding the current month, plus the number of elapsed days in the current month. A `for` loop will be used to perform this sum. Since the number of days in each month varies, it is necessary to determine the correct number of days to add for each month. A `switch` construct will be used to determine the proper number of days to add for each month.

During a leap year, an extra day must be added to the day of year for any month after February. This extra day accounts for the presence of February 29 in the leap year. Therefore, to perform the day of year calculation correctly, we must determine which years are leap years. In the Gregorian calendar, leap years are determined by the following rules.

1. Years evenly divisible by 400 are leap years.
2. Years evenly divisible by 100 but *not* by 400 are not leap years.
3. All years divisible by 4 but *not* by 100 are leap years.
4. All other years are not leap years.

We will use the `mod` (for modulo) function to determine whether or not a year is evenly divisible by a given number. If the result of the `mod` function is zero, then the year was evenly divisible.

A program to calculate the day of year is shown below. Note that the program sums up the number of days in each month before the current month, and that it uses a `switch` construct to determine the number of days in each month.

```
% Script file: doy.m
%
% Purpose:
%   This program calculates the day of year corresponding
%   to a specified date. It illustrates the use switch and
%   for constructs.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   ----      ======      =====
%   12/07/98  S. J. Chapman  Original code
%
% Define variables:
%   day          -- Day (dd)
%   day_of_year  -- Day of year
%   ii           -- Loop index
%   month        -- Month (mm)
%   year         -- Year (YYYY)

% Get day, month, and year to convert
disp('This program calculates the day of year given the ');
disp('current date.');
month = input('Enter current month (1-12): ');
```

```

day    = input('Enter current day(1-31):   ');
year   = input('Enter current year(yyyy):   ');

% Check for leap year, and add extra day if necessary
if mod(year,400) == 0
    leap_day = 1;           % Years divisible by 400 are leap years
elseif mod(year,100) == 0
    leap_day = 0;           % Other centuries are not leap years
elseif mod(year,4) == 0
    leap_day = 1;           % Otherwise every 4th year is a leap year
else
    leap_day = 0;           % Other years are not leap years
end

% Calculate day of year by adding current day to the
% days in previous months.
day_of_year = day;
for ii = 1:month-1

    % Add days in months from January to last month
    switch (ii)
        case {1,3,5,7,8,10,12},
            day_of_year = day_of_year + 31;
        case {4,6,9,11},
            day_of_year = day_of_year + 30;
        case 2,
            day_of_year = day_of_year + 28 + leap_day;
    end

end

% Tell user
fprintf('The date %2d/%2d/%4d is day of year %d.\n', ...
    month, day, year, day_of_year);

```

We will use the following known results to test the program.

1. Year 1999 is not a leap year. January 1 must be day of year 1, and December 31 must be day of year 365.
2. Year 2000 is a leap year. January 1 must be day of year 1, and December 31 must be day of year 366.
3. Year 2001 is not a leap year. March 1 must be day of year 60, since January has 31 days, February has 28 days, and this is the first day of March.

If this program is executed five times with the given dates, the results are

```

* doy
This program calculates the day of year given the
current date.
Enter current month (1-12): 1
Enter current day(1-31): 1
Enter current year(yyyy): 1999

```

```

The date 1/ 1/1999 is day of year 1.
» doy
This program calculates the day of year given the
current date.
Enter current month (1-12): 12
Enter current day(1-31): 31
Enter current year(yyyy): 1999
The date 12/31/1999 is day of year 365.
» doy
This program calculates the day of year given the
current date.
Enter current month (1-12): 1
Enter current day(1-31): 1
Enter current year(yyyy): 2000
The date 1/ 1/2000 is day of year 1.
» doy
This program calculates the day of year given the
current date.
Enter current month (1-12): 12
Enter current day(1-31): 31
Enter current year(yyyy): 2000
The date 12/31/2000 is day of year 366.
» doy
This program calculates the day of year given the
current date.
Enter current month (1-12): 3
Enter current day(1-31): 1
Enter current year(yyyy): 2001
The date 3/ 1/2001 is day of year 60.

```

The program gives the correct answers for our test dates in all five test cases.



Example 4.4—Statistical Analysis

Implement an algorithm that reads in a set of measurements and calculates the mean and the standard deviation of the input data set, when any value in the data set can be positive, negative, or zero.

Solution

This program must be able to read in an arbitrary number of measurements and then calculate the mean and standard deviation of those measurements. Each measurement can be positive, negative, or zero.

Since we cannot use a data value as a flag this time, we will ask the user for the number of input values, and then use a **for** loop to read in those values. The

modified program that permits the use of any input value is shown below. Verify its operation for yourself by finding the mean and standard deviation of the following 5 input values: 3, -1, 0, 1, and -2.

```
% Script file: stats_3.m
%
% Purpose:
%   To calculate mean and the standard deviation of
%   an input data set, where each input value can be
%   positive, negative, or zero.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   ----      ======      =====
%   12/08/98    S. J. Chapman  Original code
%
% Define variables:
%   ii      -- Loop index
%   n       -- The number of input samples
%   std_dev -- The standard deviation of the input samples
%   sum_x   -- The sum of the input values
%   sum_x2  -- The sum of the squares of the input values
%   x       -- An input data value
%   xbar    -- The average of the input samples

% Initialize sums.
sum_x = 0; sum_x2 = 0;

% Get the number of points to input.
n = input('Enter number of points: ');

% Check to see if we have enough input data.
if n < 2 % Insufficient data

    disp ('At least 2 values must be entered.');
else % we will have enough data, so let's get it.

    % Loop to read input values.
    for ii = 1:n

        % Read in next value
        x = input('Enter value: ');

        % Accumulate sums.
        sum_x = sum_x + x;
        sum_x2 = sum_x2 + x^2;

    end

    % Now calculate statistics.
    x_bar = sum_x / n;
    std_dev = sqrt( (n * sum_x2 - sum_x^2) / (n * (n-1)) );

```

```
% Tell user.
fprintf('The mean of this data set is: %f\n', x_bar);
fprintf('The standard deviation is:    %f\n', std_dev);
fprintf('The number of data points is: %f\n', n);

end
```



4.2.1 Details of Operation

Now that we have seen examples of a `for` loop in operation, we must examine some important details required to use `for` loops properly.

1. It is not necessary to indent the body of a `for` loop, as shown in the examples above. MATLAB will recognize the loop even if every statement in it starts in column 1. However, the code is much more readable if the body of the `for` loop is indented, so you should always indent the bodies of loops.

Good Programming Practice

Always indent the body of a `for` loop by 2 or more spaces to improve the readability of the code.

2. The loop index of a `for` loop *should not be modified anywhere within the loop*. The index variable is often used as a counter within the loop and modifying its value can cause strange and hard-to-find errors. The example shown below is intended to initialize the elements of an array, but the statement “`ii = 5`” has been accidentally inserted into the body of the loop. As a result, only `a(5)` is initialized, and it gets the values that should have gone into `a(1)`, `a(2)`, and so on.

```
for ii = 1:10
    ...
    ii = 5;      % Error!
    ...
    a(ii) = <calculation>
end
```

Good Programming Practice

Never modify the value of a loop index within the body of the loop.

3. We learned in Chapter 2 that it is possible to extend an existing array simply by assigning a value to a higher array element. For example, the statement

```
arr = 1:4;
```

defines a 4-element array containing the values [1 2 3 4]. If the statement

```
arr(8) = 6;
```

is executed, the array will be automatically extended to 8 elements and will contain the values [1 2 3 4 0 0 0 6]. Unfortunately, each time that an array is extended, MATLAB has to (1) create a new array, (2) copy the contents of the old array to the new longer array, (3) add the new value to the array, and then (4) delete the old array. This process is very time-consuming for long arrays.

When a for loop stores values in a previously undefined array, the loop forces MATLAB to go through this process each time the loop is executed. On the other hand, if the array is **preallocated** to its maximum size before the loop starts executing, no copying is required, and the code executes much faster. The following code fragment shows how to preallocate an array before starting the loop.

```
square = zeros(1,100);
for ii = 1:100
    square(ii) = ii^2;
end
```

Good Programming Practice

Always preallocate all arrays used in a loop before executing the loop. This practice greatly increases the execution speed of the loop.

4. It is often possible to perform calculations with either for loops or vectors. For example, the following code fragment calculates the squares, square roots, and cube roots of all integers between 1 and 100 using a for loop.

```
for ii = 1:100
    square(ii) = ii^2;
    square_root(ii) = ii^(1/2);
    cube_root(ii) = ii^(1/3);
end
```

The following code fragment performs the same calculation with vectors.

```
ii = 1:100;
square = ii.^2;
square_root = ii.^(1/2);
cube_root(ii) = ii.^(1/3);
```

Even though these two calculations produce the same answers, they are *not equivalent*. The version with the `for` loop is *more than 15 times slower* than the vectorized version! This happens because the statements in the `for` loop must be interpreted and executed a line at a time by MATLAB during each pass of the loop. In effect, MATLAB must interpret and execute 300 separate lines of code. In contrast, MATLAB only has to interpret and execute 4 lines in the vectorized case. Since MATLAB is designed to implement vectorized statements in a very efficient fashion, it is much faster in that mode.

The disadvantage of vectorizing the statements is that more memory is needed, since some intermediate arrays must be created. This is usually a very small penalty, so vectorizing calculations is almost always better than implementing them with `for` loops.

In MATLAB, the process of replacing loops by vectorized statements is known as **vectorization**. Vectorization can yield dramatic improvements in performance for many MATLAB programs.

Good Programming Practice

If it is possible to implement a calculation either with a `for` loop or using vectors, *always implement the calculation with vectors*. Your program will be *much faster*.

Example 4.5—Comparing Loops and Vectorization

To compare the execution speeds of loops and vectorized statements, perform and time the following three sets of calculations.

1. Calculate the squares of every integer from 1 to 10,000 in a `for` loop without initializing the array of squares first.
2. Calculate the squares of every integer from 1 to 10,000 in a `for` loop, using the `zeros` function to pre-allocate the array of squares first.
3. Calculate the squares of every integer from 1 to 10,000 with vectors.

Solution

This program must calculate the squares of the integers from 1 to 10,000 in each of the three ways described, timing the executions in each case. The timing can be accomplished using the MATLAB functions `tic` and `toc`. Function `tic` resets

the built-in elapsed time counter, and function `toc` returns the elapsed time in seconds since the last call to function `tic`.

Since the real-time clocks in many computers have a fairly coarse granularity, it may be necessary to execute each set of instructions multiple times to get a valid average time.

A MATLAB program to compare the speeds of the three approaches follows.

```
% Script file: timings.m
%
% Purpose:
% This program calculates the time required to
% calculate the squares of all integers from 1 to
% 10,000 in three different ways:
% 1. Using a for loop with an uninitialized output
%    array.
% 2. Using a for loop with a pre-allocated output
%    array.
% 3. Using vectors.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   ----      ======          ======
% 12/08/98  S. J. Chapman  Original code
%
% Define variables:
% ii, jj      -- Loop index
% average1    -- Average time for calculation 1
% average2    -- Average time for calculation 2
% average3    -- Average time for calculation 3
% maxcount    -- Number of times to loop calculation
% square      -- Array of squares
% leap_day    -- Extra day for leap year
% month       -- Month (mm)
% year        -- Year (YYYY)

% Perform calculation with an uninitialized array
% "square".  This calculation is done only once
% because it is so slow.
maxcount = 1;                      % One repetition
tic;                                % Start timer
for jj = 1:maxcount
    clear square;                   % Clear output array
    for ii = 1:10000
        square(ii) = ii^2;         % Calculate square
    end
end
average1 = (toc)/maxcount;          % Calculate average time
```

```

% Perform calculation with a pre-allocated array
% "square". This calculation is averaged over 10
% loops.
maxcount = 10;                      % One repetition
tic;                                  % Start timer
for jj = 1:maxcount
    clear square                     % Clear output array
    square = zeros(1,10000);          % Pre-initialize array
    for ii = 1:10000
        square(ii) = ii^2;           % Calculate square
    end
end
average2 = (toc)/maxcount;  % Calculate average time

% Perform calculation with vectors. This calculation
% averaged over 100 executions.
maxcount = 100;                      % One repetition
tic;                                  % Start timer
for jj = 1:maxcount
    clear square                     % Clear output array
    ii = 1:10000;                   % Set up vector
    square = ii.^2;                 % Calculate square
end
average3 = (toc)/maxcount;  % Calculate average time

% Display results
fprintf('Loop / uninitialized array = %8.4f\n', ...
        average1);
fprintf('Loop / initialized array =   %8.4f\n', ...
        average2);
fprintf('Vectorized =                  %8.4f\n', ...
        average3);

```

When this program is executed using MATLAB 6 on a 733 MHz Pentium computer, the results are

```

> timings
Loop / uninitialized array = 2.9640
Loop / initialized array =   0.1002
Vectorized =                  0.0014

```

As you can see, the proper use of initializations and vectorized calculations can make an incredible difference in the speed of your MATLAB code!

4.2.2 The break and continue Statements

There are two additional statements that can be used to control the operation of while loops and for loops: the break and continue statements. The break

statement terminates the execution of a loop and passes control to the next statement after the end of the loop, while the `continue` statement terminates the current pass through the loop and returns control to the top of the loop.

If a `break` statement is executed in the body of a loop, the execution of the body will stop and control will be transferred to the first executable statement after the loop. An example of the `break` statement in a `for` loop is shown below.

```
for ii = 1:5
    if ii == 3;
        break;
    end
    fprintf('ii = %d\n',ii);
end
disp('End of loop!');
```

When this program is executed, the output is

```
>> test_break
ii = 1
ii = 2
End of loop!
```

Note that the `break` statement was executed on the iteration when `ii` was 3, and control transferred to the first executable statement after the loop without executing the `fprintf` statement.

If a `continue` statement is executed in the body of a loop, the execution of the current pass through the loop will stop and control will return to the top of the loop. The controlling variable in the `for` loop will take on its next value, and the loop will be executed again. An example of the `continue` statement in a `for` loop is shown below.

```
for ii = 1:5
    if ii == 3;
        break;
    end
    fprintf('ii = %d\n',ii);
end
disp('End of loop!');
```

When this program is executed, the output is

```
>> test_continue
ii = 1
ii = 2
ii = 4
ii = 5
End of loop!
```

Note that the `continue` statement was executed on the iteration when `ii` was 3, and control transferred to the top of the loop without executing the `fprintf` statement.

The `break` and `continue` statements work with both `while` loops and `for` loops.

4.2.3 Nesting Loops

It is possible for one loop to be completely inside another loop. If one loop is completely inside another one, the two loops are called **nested loops**. The following example shows two nested for loops used to calculate and write out the product of two integers.

```
for ii = 1:3
    for jj = 1:3
        product = ii * jj;
        fprintf('%d * %d = %d\n',ii,jj,product);
    end
end
```

In this example, the outer for loop will assign a value of 1 to index variable *ii*, and then the inner for loop will be executed. The inner for loop will be executed 3 times with index variable *jj* having values 1, 2, and 3. When the entire inner for loop has been completed, the outer for loop will assign a value of 2 to index variable *ii*, and the inner for loop will be executed again. This process repeats until the outer for loop has executed 3 times, and the resulting output is

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
```

Note that the inner for loop executes completely before the index variable of the outer for loop is incremented.

When MATLAB encounters an end statement, it associates that statement with the innermost currently open construct. Therefore, the first end statement above closes the “`for jj = 1:3`” loop, and the second end statement above closes the “`for ii = 1:3`” loop. This fact can produce hard-to-find errors if an end statement is accidentally deleted somewhere within a nested loop construct.

If for loops are nested, they should have independent loop index variables. If they have the same index variable, then the inner loop will change the value of the loop index that the outer loop just set.

If a break or continue statement appears inside a set of nested loops, then the break statement refers to the *innermost* of the loops containing it. For example, consider the following program.

```
for ii = 1:3
    for jj = 1:3
        if jj == 3;
            break;
        end
        product = ii * jj;
```

```

        fprintf('%d * %d = %d\n',ii,jj,product);
    end
    fprintf('End of inner loop\n');
end
fprintf('End of outer loop\n');

```

If the inner loop counter `jj` is equal to 3, then the `break` statement will be executed. This will cause the program to exit the innermost loop. The program will print out “End of inner loop,” the index of the outer loop will be increased by 1, and execution of the innermost loop will start over. The resulting output values are

```

1 * 1 = 1
1 * 2 = 2
End of inner loop
2 * 1 = 2
2 * 2 = 4
End of inner loop
3 * 1 = 3
3 * 2 = 6
End of inner loop
End of outer loop

```

4.3 Logical Arrays and Vectorization

In Chapter 2, we mentioned that MATLAB has two fundamental data types: numeric and string. Numeric data contains numbers, and string data contains characters. In addition to these two data types, MATLAB contains a third “sort of” data type: logical.

The “logical” data type is not really distinct in MATLAB. Instead, it is a standard numeric array with a special “logical” property added. **Logical arrays** are created by all relational and logic operators. They can be distinguished from standard arrays because the `(logical)` modifier appears after them in the `whos` command or the Workspace Browser.

For example, consider the following statements:

```

a = [1 2 3; 4 5 6; 7 8 9];
b = a > 5;

```

These statements produced two arrays `a` and `b`. Array `a` is a numeric array containing the values

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$
, and array `b` is a special numeric array with the

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$
. When the `whos` command

is executed, the results are as shown below. Note that array `b` has the `(logical)` modifier applied to it.

```
>> whos
  Name      Size            Bytes  Class
  a          3x3              72  double array
  b          3x3              72  double array (logical)
```

Grand total is 18 elements using 144 bytes

It is also possible to add the logical property to an array by using the logical function. For example, the statement `c = logical(a)` will assign the values in array `a` to array `c`, while setting the logical property for the array:

```
>> c = logical(a)
c =
  1     2     3
  4     5     6
  7     8     9
>> whos
  Name      Size            Bytes  Class
  a          3x3              72  double array
  b          3x3              72  double array (logical)
  c          3x3              72  double array (logical)
```

Grand total is 27 elements using 216 bytes

The logical property can be removed from an array by performing almost any arithmetic operation on it. For example, if we add zero to array `c`, the values in the array will be unchanged, but the logical property will be lost.

```
>> c = c + 0
c =
  1     2     3
  4     5     6
  7     8     9
>> whos
  Name      Size            Bytes  Class
  a          3x3              72  double array
  b          3x3              72  double array (logical)
  c          3x3              72  double array
```

Grand total is 27 elements using 216 bytes

4.3.1 The Significance of Logical Arrays

Logical arrays have a very important special property—*they can serve as a mask for arithmetic operations*. A mask is an array that selects the elements of another

array for use in an operation. The specified operation will be applied to the selected elements, but *not* to the remaining elements.

For example, suppose that arrays *a* and *b* are as defined in the previous section. Then the statement *a(b) = sqrt(a(b))* will take the square root of all elements for which the logical array *b* is nonzero and leave all the other elements in the array unchanged.

```
>> a(b) = sqrt(a(b))
a =
    1.0000    2.0000    3.0000
    4.0000    5.0000    2.4495
    2.6458    2.8284    3.0000
```

This is a very fast and very clever way of performing an operation on a subset of an array without needing loops and branches.

The following two code fragments both take the square root of all elements in array *a* whose value is greater than 5, but the vectorized approach is much faster than the loop approach.

```
for ii = 1:size(a,1)
    for jj = 1:size(a,2)
        if a(ii,jj) > 5
            a(ii,jj) = sqrt(a(ii,jj));
        end
    end
end

b = a > 5;
a(b) = sqrt(a(b));
```

Example 4.6—Using Logical Arrays to Mask Operations

To compare the execution speeds of loops and branches versus vectorized code using a logical array, we will perform and time the following two sets of calculations.

1. Create a 10,000 element array containing the values, 1, 2, . . . , 10000. Then take the square root of all elements whose value is greater than 5,000 using a *for* loop and an *if* construct.
2. Create a 10,000 element array containing the values, 1, 2, . . . , 10000. Then take the square root of all elements whose value is greater than 5,000 using a logical array.

Solution

This program must create an array containing the integers from 1 to 10000, and take the square roots of those values that are greater than 5000 in each of the two ways described above.

A MATLAB program to compare the speeds of the two approaches is shown below:

```
% Script file: logical1.m
%
% Purpose:
%   This program calculates the time required to
%   calculate the square roots of all elements in
%   array a whose value exceeds 5000.  This is done
%   in two different ways:
%   1. Using a for loop and if construct.
%   2. Using a logical array.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   =====    ======        =====
%   06/01/01  S. J. Chapman  Original code
%
% Define variables:
%   a          -- Array of input values
%   b          -- Logical array to serve as a mask
%   ii, jj     -- Loop index
%   average1   -- Average time for calculation 1
%   average2   -- Average time for calculation 2
%   maxcount   -- Number of times to loop calculation
%   month      -- Month (mm)
%   year       -- Year (yyyy)

% Perform calculation using loops and branches.
maxcount = 1;                                % One repetition
tic;                                         % Start timer
for jj = 1:maxcount
    a = 1:10000;                               % Declare array a
    for ii = 1:10000
        if a(ii) > 5000
            a(ii) = sqrt(a(ii));
        end
    end
end
average1 = (toc)/maxcount; % Calculate average time

% Perform calculation using logical arrays.
maxcount = 10;                                % One repetition
tic;                                         % Start timer
for jj = 1:maxcount
    a = 1:10000;                               % Declare array a
    b = a > 5000;                            % Create mask
```

```

    a(b) = sqrt(a(b));      % Take square root
end
average2 = (toc)/maxcount; % Calculate average time

% Display results
fprintf('Loop / if approach =     %8.4f\n', ...
        average1);
fprintf('Logical array approach = %8.4f\n', ...
        average2);

```

When this program is executed using MATLAB 6 on a 733 MHz Pentium III computer, the results are as follows:

```

» logical1
Loop / if approach =     0.1200
Logical array approach =  0.0060

```

As you can see, the use of logical arrays can speed up code execution by a factor of 20!

Good Programming Practice

Where possible, use logical arrays as masks to select the elements of an array for processing. If logical arrays are used instead of loops and *if* constructs, your program will be *much* faster.

4.3.2 Creating the Equivalent of *if/else* Constructs with Logical Arrays

Logical arrays can also be used to implement the equivalent of an *if/else* construct inside a set of *for* loops. As we saw in the last section, it is possible to apply an operation to selected elements of an array using a logical array as a mask. It is also possible to apply a different set of operations to the *unselected* elements of the array by simply adding the *not* operator (~) to the logical mask. For example, suppose that we wanted to take the square root of any elements in a two-dimensional array whose value is greater than 5 and to square the remaining elements in the array. The code for this operation using loops and branches is

```

for ii = 1:size(a,1)
    for jj = 1:size(a,2)
        if a(ii,jj) > 5
            a(ii,jj) = sqrt(a(ii,jj));
        else

```

```

        a(ii,jj) = a(ii,jj)^2;
    end
end
end

```

The vectorized code for this operation is

```

b = a > 5;
a(b) = sqrt(a(b));
a(~b) = a(~b).^2;

```

The vectorized code is enormously faster than the loops-and-branches version.

Quiz 4.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 4.1 through 4.3. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

Examine the following `for` loops and determine how many times each loop will be executed.

1. `for index = 7:10`
2. `for jj = 7:-1:10`
3. `for index = 1:10:10`
4. `for ii = -10:3:-7`
5. `for kk = [0 5 ; 3 3]`

Examine the following loops and determine the value in `ires` at the end of each of the loops.

6. `iress = 0;`
`for index = 1:10`
 `iress = iress + 1;`
`end`
7. `iress = 0;`
`for index = 1:10`
 `iress = iress + index;`
`end`
8. `iress = 0;`
`for index1 = 1:10`
 `for index2 = index1:10`
 `if index2 == 6`
 `break;`
 `end`
 `iress = iress + 1;`

```

        end
    end
9. ires = 0;
for index1 = 1:10
    for index2 = index1:10
        if index2 == 6
            continue;
        end
        ires = ires + 1;
    end
end

```

10. Write the MATLAB statements to calculate the values of the function

$$f(t) = \begin{cases} \sin t & \text{for all } t \text{ where } \sin t > 0 \\ 0 & \text{elsewhere} \end{cases}$$

for $-6\pi < t < 6\pi$ at intervals of $\pi/10$. Do this twice, once using loops and branches, and once using vectorized code.

4.4 Additional Examples



Example 4.7—Fitting a Line to a Set of Noisy Measurements

The velocity of a falling object in the presence of a constant gravitational field is given by the equation

$$v(t) = a t + v_0 \quad (4.3)$$

where $v(t)$ is the velocity at any time t , a is the acceleration due to gravity, and v_0 is the velocity at time 0. This equation is derived from elementary physics—it is known to every freshman physics student. If we plot velocity versus time for the falling object, our (v,t) measurement points should fall along a straight line. However, the same freshman physics student also knows that if we go out into the laboratory and attempt to *measure* the velocity versus time of an object, our measurements will *not* fall along a straight line. They may come close, but they will never line up perfectly. Why not? Because we can never make perfect measurements. There is always some *noise* included in the measurements that distorts them.

There are many cases in science and engineering where there are noisy sets of data such as this, and we wish to estimate the straight line which “best fits” the

data. This problem is called the *linear regression* problem. Given a noisy set of measurements (x,y) that appear to fall along a straight line, how can we find the equation of the line

$$y = m x + b \quad (4.4)$$

which “best fits” the measurements? If we can determine the regression coefficients m and b , then we can use this equation to predict the value of y at any given x by evaluating Equation (4.4) for that value of x .

A standard method for finding the regression coefficients m and b is the *method of least squares*. This method is named *least squares* because it produces the line $y = m x + b$, for which the sum of the squares of the differences between the observed y values and the predicted y values is as small as possible. The slope of the least squares line is given by

$$m = \frac{(\sum xy) - (\sum x)\bar{y}}{(\sum x^2) - (\sum x)\bar{x}} \quad (4.5)$$

and the intercept of the least squares line is given by

$$b = \bar{y} - m \bar{x} \quad (4.6)$$

where

$\sum x$ is the sum of the x values.

$\sum x^2$ is the sum of the squares of the x values.

$\sum xy$ is the sum of the products of the corresponding x and y values.

\bar{x} is the mean (average) of the x values.

\bar{y} is the mean (average) of the y values.

Write a program that will calculate the least-squares slope m and y -axis intercept b for a given set of noisy measured data points (x,y) . The data points should be read from the keyboard, and both the individual data points and the resulting least-squares fitted line should be plotted.

Solution

1. State the problem

Calculate the slope m and intercept b of a least-squares line that best fits an input data set consisting of an arbitrary number of (x,y) pairs. The input (x,y) data is read from the keyboard. Plot both the input data points and the fitted line on a single plot.

2. Define the inputs and outputs

The inputs required by this program are the number of points to read, plus the pairs of points (x,y) .

The outputs from this program are the slope and intercept of the least-squares fitted line, the number of points going into the fit, and a plot of the input data and the fitted line.

3. Describe the algorithm

This program can be broken down into six major steps.

- Get the number of input data points
- Read the input data values
- Calculate the required statistics
- Calculate the slope and intercept
- Write out the slope and intercept
- Plot the input points and the fitted line

The first major step of the program is to get the number of points to read in. To do this, we will prompt the user and read his or her answer with an input function. Next we will read the input (x, y) pairs one pair at a time using an input function in a for loop. Each pair of input values will be placed in an array ($[x \ y]$), and the function will return that array to the calling program. Note that a for loop is appropriate because we know in advance how many times the loop will be executed.

The pseudocode for these steps is shown below.

```
Print message describing purpose of the program
n_points ← input('Enter number of [x y] pairs: ');
for ii = 1:n_points
    temp ← input('Enter [x y] pair: ');
    x(ii) ← temp(1);
    y(ii) ← temp(2);
end
```

Next, we must accumulate the statistics required for the calculation. These statistics are the sums $\sum x$, $\sum y$, $\sum x^2$, and $\sum xy$. The pseudocode for these steps is

```
Clear the variables sum_x, sum_y, sum_x2, and sum_xy
for ii = 1:n_points
    sum_x ← sum_x + x(ii)
    sum_y ← sum_y + y(ii)
    sum_x2 ← sum_x2 + x(ii)^2
    sum_xy ← sum_xy + x(ii)*y(ii)
end
```

Next, we must calculate the slope and intercept of the least-squares line. The pseudocode for this step is just the MATLAB versions of Equations (4.4) and (4.5).

```
x_bar ← sum_x / n_points
y_bar ← sum_y / n_points
slope ← (sum_xy - sum_x * y_bar) / (sum_x2 - sum_x * x_bar)
y_int ← y_bar - slope * x_bar
```

Finally, we must write out and plot the results. The input data points should be plotted with circular markers and without a connecting line,

while the fitted line should be plotted as a solid 2-pixel wide line. To do this, we will need to plot the points first, set `hold on`, plot the fitted line, and set `hold off`. We will add titles and a legend to the plot for completeness.

4. Turn the algorithm into MATLAB statements

The final MATLAB program follows.

```
% Purpose:
% To perform a least-squares fit of an input data set
% to a straight line, and print out the resulting slope
% and intercept values. The input data for this fit
% is input from the keyboard.

%
% Record of revisions:
% Date      Programmer      Description of change
% =====    ======      =====
% 01/03/99  S. J. Chapman  Original code

%
% Define variables:
% ii          -- Loop index
% n_points    -- Number in input [x y] points
% slope       -- Slope of the line
% sum_x       -- Sum of all input x values
% sum_x2      -- Sum of all input x values squared
% sum_xy      -- Sum of all input x*y values
% sum_y       -- Sum of all input y values
% temp        -- Variable to read user input
% x           -- Array of x values
% x_bar       -- Average x value
% y           -- Array of y values
% y_bar       -- Average y value
% y_int       -- y-axis intercept of the line

disp('This program performs a least-squares fit of an ');
disp('input data set to a straight line.');
n_points = input('Enter the number of input [x y] points: ');

% Read the input data
for ii = 1:n_points
    temp = input('Enter [x y] pair: ');
    x(ii) = temp(1);
    y(ii) = temp(2);
end

% Accumulate statistics
sum_x = 0;
sum_y = 0;
sum_x2 = 0;
sum_xy = 0;
```

```

for ii = 1:n_points
    sum_x = sum_x + x(ii);
    sum_y = sum_y + y(ii);
    sum_x2 = sum_x2 + x(ii)^2;
    sum_xy = sum_xy + x(ii) * y(ii);
end

% Now calculate the slope and intercept.
x_bar = sum_x / n_points;
y_bar = sum_y / n_points;
slope = (sum_xy - sum_x * y_bar) / (sum_x2 - sum_x * x_bar);
y_int = y_bar - slope * x_bar;

% Tell user.
disp('Regression coefficients for the least-squares line:');
fprintf(' Slope (m) = %8.3f\n', slope);
fprintf(' Intercept (b) = %8.3f\n', y_int);
fprintf(' No of points = %8d\n', n_points);

% Plot the data points as blue circles with no
% connecting lines.
plot(x,y,'bo');
hold on;

% Create the fitted line
xmin = min(x);
xmax = max(x);
ymin = slope * xmin + y_int;
ymax = slope * xmax + y_int;

% Plot a solid red line with no markers
plot([xmin xmax],[ymin ymax],'r-','LineWidth',2);
hold off;

% Add a title and legend
title ('\bfLeast-Squares Fit');
xlabel('\bf\itx');
ylabel('\bf\ity');
legend('Input data','Fitted line');
grid on

```

5. Test the program

To test this program, we will try a simple data set. For example, if every point in the input data set actually falls along a line, then the resulting slope and intercept should be exactly the slope and intercept of that line.

Thus the data set

```
[1.1 1.1]
[2.2 2.2]
[3.3 3.3]
```

```
[4.4 4.4]
[5.5 5.5]
[6.6 6.6]
[7.7 7.7]
```

should produce a slope of 1.0 and an intercept of 0.0. If we run the program with these values, the results are as follows.

```
> lsqfit
This program performs a least-squares fit of an
input data set to a straight line.
Enter the number of input [x y] points: 7
Enter [x y] pair: [1.1 1.1]
Enter [x y] pair: [2.2 2.2]
Enter [x y] pair: [3.3 3.3]
Enter [x y] pair: [4.4 4.4]
Enter [x y] pair: [5.5 5.5]
Enter [x y] pair: [6.6 6.6]
Enter [x y] pair: [7.7 7.7]
Regression coefficients for the least-squares line:
Slope (m)      =    1.000
Intercept (b)   =    0.000
No of points    =      7
```

Now we will add some noise to the measurements. The data set becomes

```
[1.1 1.01]
[2.2 2.30]
[3.3 3.05]
[4.4 4.28]
[5.5 5.75]
[6.6 6.48]
[7.7 7.84]
```

If we run the program with these values, the results are

```
> lsqfit
This program performs a least-squares fit of an
input data set to a straight line.
Enter the number of input [x y] points: 7
Enter [x y] pair: [1.1 1.01]
Enter [x y] pair: [2.2 2.30]
Enter [x y] pair: [3.3 3.05]
Enter [x y] pair: [4.4 4.28]
Enter [x y] pair: [5.5 5.75]
Enter [x y] pair: [6.6 6.48]
Enter [x y] pair: [7.7 7.84]
Regression coefficients for the least-squares line:
Slope (m)      =    1.024
Intercept (b)   =   -0.120
No of points    =      7
```

If we calculate the answer by hand, it is easy to show that the program gives the correct answers for our two test data sets. The noisy input data set and the resulting least-squares fitted line are shown in Figure 4.1.

This example uses several of the plotting capabilities that we introduced in Chapter 3. It uses the `hold` command to allow multiple plots to be placed on the same axes, the `LineWidth` property to set the width of the least-squares fitted line, and escape sequences to make the title ***boldface*** and the axis labels ***italic***.

Example 4.8—Physics: The Flight of a Ball

If we assume negligible air friction and ignore the curvature of the Earth, a ball that is thrown into the air from any point on the Earth's surface will follow a parabolic flight path (Figure 4.2a). The height of the ball at any time t after it is thrown is given by Equation (4.7)

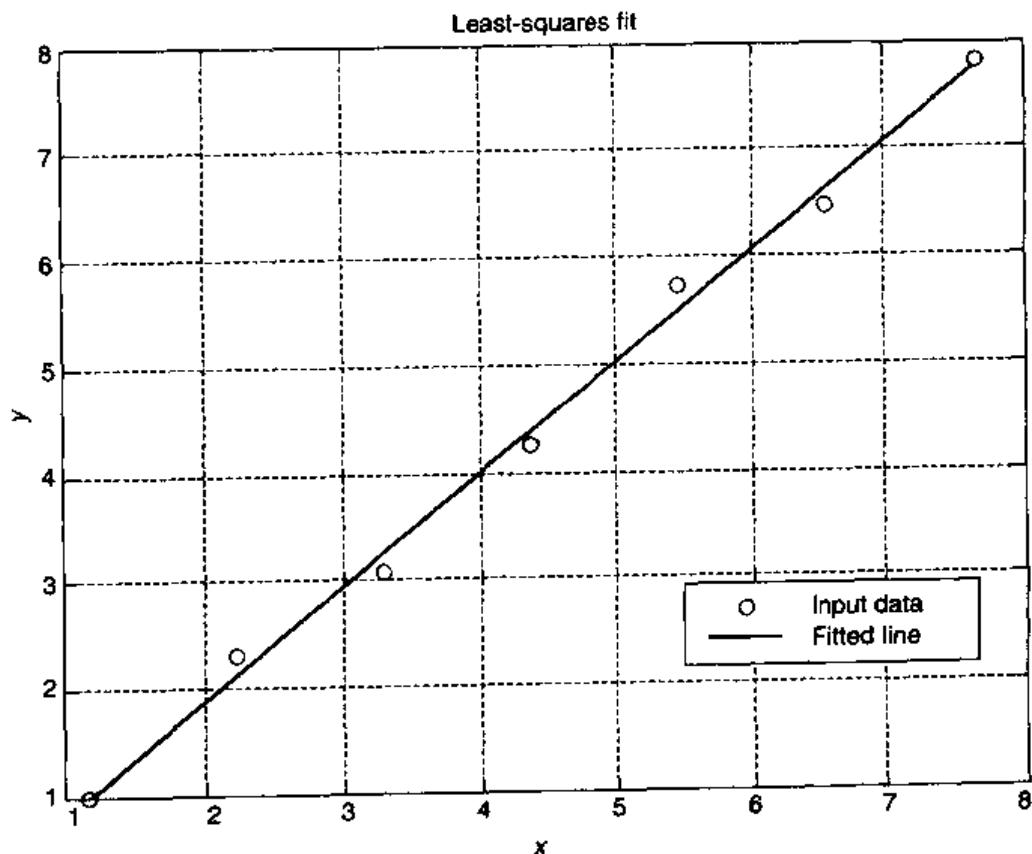


Figure 4.1 A noisy data set with a least-squares fitted line.

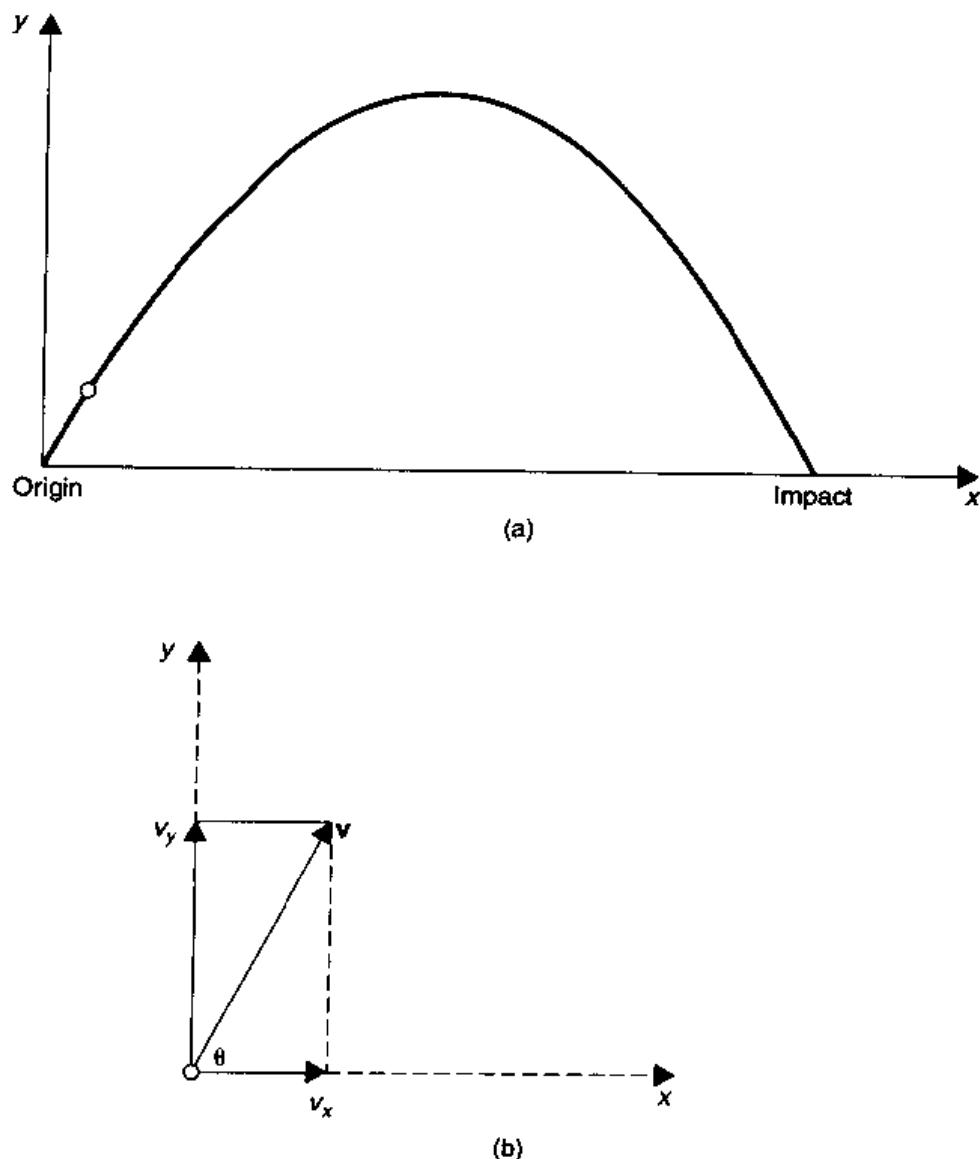


Figure 4.2 (a) When a ball is thrown upward, it follows a parabolic trajectory. (b) The horizontal and vertical components of a velocity vector v at an angle θ with respect to the horizontal.

$$y(t) = y_o + v_{yo} t + \frac{1}{2} g t^2 \quad (4.7)$$

where y_o is the initial height of the object above the ground, v_{yo} is the initial vertical velocity of the object, and g is the acceleration due to the Earth's gravity. The horizontal distance (range) traveled by the ball as a function of time after it is thrown is given by Equation (4.8)

$$x(t) = x_o + v_{xo} t \quad (4.8)$$

where x_o is the initial horizontal position of the ball on the ground, and v_{xo} is the initial horizontal velocity of the ball.

If the ball is thrown with some initial velocity v_o at an angle of θ degrees with respect to the Earth's surface, then the initial horizontal and vertical components of velocity will be

$$v_{xo} = v_o \cos \theta \quad (4.9)$$

$$v_{yo} = v_o \sin \theta \quad (4.10)$$

Assume that the ball is initially thrown from position $(x_o, y_o) = (0, 0)$ with an initial velocity v_o of 20 meters per second at an initial angle of θ degrees. Write a program that will plot the trajectory of the ball and also determine the horizontal distance traveled before it touches the ground again. The program should plot the trajectories of the ball for all angles θ from 5° to 85° , in 10° steps, and should determine the horizontal distance traveled for all angles θ from 0° to 90° , in 1° steps. Finally, it should determine the angle θ that maximizes the range of the ball, and plot that particular trajectory in a different color with a thicker line.

Solution

To solve this problem, we must determine an equation for the time that the ball returns to the ground. Then, we can calculate the (x, y) position of the ball using Equations (4.7) through (4.10). If we do this for many times between 0 and the time that the ball returns to the ground, we can use those points to plot the ball's trajectory.

The time that the ball will remain in the air after it is thrown can be calculated from Equation (4.7). The ball will touch the ground at the time t for which $y(t) = 0$. Remembering that the ball will start from ground level ($y(0) = 0$), and solving for t , we get

$$y(t) = y_o + v_{yo} t - \frac{1}{2} g t^2 \quad (4.7)$$

$$0 = 0 + v_{yo} t + \frac{1}{2} g t^2$$

$$0 = \left(v_{yo} + \frac{1}{2} g t \right) t$$

so the ball will be at ground level at time $t_1 = 0$ (when we threw it), and at time

$$t_2 = -\frac{2 v_{yo}}{g} \quad (4.11)$$

From the problem statement, we know that the initial velocity v_o is 20 meters per second, and that the ball will be thrown at all angles from 0° to 90° in 1° steps. Finally, any elementary physics textbook will tell us that the acceleration due to the Earth's gravity is 9.81 meters per second squared.

Now we apply our design technique to this problem.

1. State the problem

A proper statement of this problem would be: *Calculate the range that a ball would travel when it is thrown with an initial velocity of v_0 of 20 m/s at an initial angle θ . Calculate this range for all angles between 0° and 90° , in 1° steps. Determine the angle θ that will result in the maximum range for the ball. Plot the trajectory of the ball for angles between 5° and 85° , in 10° increments. Plot the maximum-range trajectory in a different color and with a thicker line. Assume that there is no air friction.*

2. Define the inputs and outputs

As the problem is defined, no inputs are required. We know from the problem statement what v_0 and θ will be, so there is no need to input them. The outputs from this program will be a table showing the range of the ball for each angle θ , the angle θ for which the range is maximum, and a plot of the specified trajectories.

3. Design the algorithm

This program can be broken down into the following major steps.

Calculate the range of the ball for θ between 0 and 90°
Write a table of ranges

Determine the maximum range and write it out

Plot the trajectories for θ between 5 and 85°

Plot the maximum-range trajectory

Since we know the exact number of times that the loops will be repeated, for loops are appropriate for this algorithm. We will now refine the pseudocode for each of the major steps.

To calculate the maximum range of the ball for each angle, we will first calculate the initial horizontal and vertical velocity from Equations (4.9) and (4.10). Then, we will determine the time when the ball returns to Earth from Equation (4.11). Finally, we will calculate the range at that time from Equation (4.7). The detailed pseudocode for these steps follows. Note that we must convert all angles to radians before using the trigonometric functions!

```
Create and initialize an array to hold ranges
for ii = 1:91
    theta ← ii - 1;
    vxo ← vo * cos(theta*conv);
    vyo ← vo * sin(theta*conv);
    max_time ← -2 * vyo / g;
    range(ii) ← vxo * max_time;
end
```

Next, we must write a table of ranges. The pseudocode for this step is:

```

Write heading
for ii = 1:91
    theta ← ii - 1;
    print theta and range;
end

```

The maximum range can be found with the `max` function. Recall that this function returns both the maximum value and its location. The pseudocode for this step is:

```

{maxrange index} ← max(range);
Print out maximum range and angle (=index-1);

```

We will use nested `for` loops to calculate and plot the trajectories. To get all of the plots to appear on the screen, we must plot the first trajectory and then set `hold on` before plotting any other trajectories. After plotting the last trajectory, we must set `hold off`. To perform this calculation, we will divide each trajectory into 21 time steps and find the x and y positions of the ball for each time step. Then, we will plot those (x,y) positions. The pseudocode for this step is:

```

for ii = 5:10:85
    % Get velocities and max time for this angle
    theta ← ii - 1;
    vxo ← vo * cos(theta*conv);
    vyo ← vo * sin(theta*conv);
    max_time ← -2 * vyo / g;

    Initialize x and y arrays
    for jj = 1:21
        time ← (jj-1) * max_time/20;
        x(time) ← vxo * time;
        y(time) ← vyo * time + 0.5 * g * time^2;
    end
    plot(x,y) with thin green lines
    Set 'hold on' after first plot
end
Add titles and axis labels

```

Finally, we must plot the maximum range trajectory in a different color and with a thicker line.

```

vxo ← vo * cos(max_angle*conv);
vyo ← vo * sin(max_angle*conv);
max_time ← -2 * vyo / g;

Initialize x and y arrays
for jj = 1:21
    time ← (jj-1) * max_time/20;

```

```

        x( jj ) ← vxo * time;
        y( jj ) ← vyo * time + 0.5 * g * time^2;
    end
    plot(x,y) with a thick red line
    hold off

```

4. Turn the algorithm into MATLAB statements

The final MATLAB program follows.

```

% Script file: ball.m
%
% Purpose:
% This program calculates the distance traveled by a ball
% thrown at a specified angle "theta" and a specified
% velocity "vo" from a point on the surface of the Earth,
% ignoring air friction and the Earth's curvature. It
% calculates the angle yielding maximum range, and also
% plots selected trajectories.
%
% Record of revisions:
% Date      Programmer      Description of change
% =====      ======      =====
% 12/10/98   S. J. Chapman  Original code
%
% Define variables:
% conv          -- Degrees to radians conv factor
% gravity       -- Accel. due to gravity (m/s^2)
% ii, jj        -- Loop index
% index         -- Location of maximum range in array
% maxangle      -- Angle that gives maximum range (deg)
% maxrange     -- Maximum range (m)
% range         -- Range for a particular angle (m)
% time          -- Time (s)
% theta         -- Initial angle (deg)
% traj_time    -- Total trajectory time (s)
% vo            -- Initial velocity (m/s)
% vxo           -- X-component of initial velocity (m/s)
% vyo           -- Y-component of initial velocity (m/s)
% x              -- X-position of ball (m)
% y              -- Y-position of ball (m)

% Constants
conv = pi / 180;      % Degrees-to-radians conversion factor
g = -9.81;             % Accel. due to gravity
vo = 20;                % Initial velocity

% Create an array to hold ranges
range = zeros(1,91);

% Calculate maximum ranges
for ii = 1:91

```

```

theta = ii - 1;
vxo = vo * cos(theta*conv);
vyo = vo * sin(theta*conv);
max_time = -2 * vyo / g;
range(ii) = vxo * max_time;
end

% Write out table of ranges
fprintf ('Range versus angle theta:\n');
for ii = 1:91
    theta = ii - 1;
    fprintf(' %2d %8.4f\n',theta, range(ii));
end

% Calculate the maximum range and angle
[maxrange index] = max(range);
maxangle = index - 1;
fprintf ('\nMax range is %8.4f at %2d degrees.\n',...
    maxrange, maxangle);

% Now plot the trajectories
for ii = 5:10:85

    % Get velocities and max time for this angle
    theta = ii;
    vxo = vo * cos(theta*conv);
    vyo = vo * sin(theta*conv);
    max_time = -2 * vyo / g;

    % Calculate the (x,y) positions
    x = zeros(1,21);
    y = zeros(1,21);
    for jj = 1:21
        time = (jj-1) * max_time/20;
        x(jj) = vxo * time;
        y(jj) = vyo * time + 0.5 * g * time^2;
    end
    plot(x,y,'b');
    if ii == 5
        hold on;
    end
end

% Add titles and axis labels
title ('\bfTrajectory of Ball vs Initial Angle \theta');
xlabel ('\bf\itx \rm\bf(meters)');
ylabel ('\bf\ity \rm\bf(meters)');
axis ([0 45 0 25]);
grid on;

% Now plot the max range trajectory
vxo = vo * cos(maxangle*conv);

```

```

vyo = vo * sin(maxangle*conv);
max_time = -2 * vyo / g;

% Calculate the (x,y) positions
x = zeros(1,21);
y = zeros(1,21);
for jj = 1:21
    time = (jj-1) * max_time/20;
    x(jj) = vxo * time;
    y(jj) = vyo * time + 0.5 * g * time^2;
end
plot(x,y,'r','LineWidth',3.0);
hold off

```

The acceleration due to gravity at sea level can be found in any physics text. It is about 9.81 m/sec^2 , directed downward.

5. Test the program

To test this program, we will calculate the answers by hand for a few of the angles, and compare the results with the output of the program.

θ	$v_{xo} = v_o \cos \theta$	$v_{yo} = v_o \sin \theta$	$t_2 = -\frac{2v_{yo}}{g}$	$x = v_{xo}t_2$
0°	20 m/s	0 m/s	0 s	0 m
5°	19.92 m/s	1.74 m/s	0.355 s	7.08 m
40°	15.32 m/s	12.86 m/s	2.621 s	40.15 m
45°	14.14 m/s	14.14 m/s	2.883 m/s	40.77 m

When program ball is executed, a 91-line table of angles and ranges is produced. To save space, only a portion of the table is reproduced.

```

> ball
Range versus angle theta:
      0      0.0000
      1      1.4230
      2      2.8443
      3      4.2621
      4      5.6747
      5      7.0805
      .
      40     40.1553
      41     40.3779
      42     40.5514
      43     40.6754
      44     40.7499
      45     40.7747

```

46	40.7499
47	40.6754
48	40.5514
49	40.3779
50	40.1553
...	
85	7.0805
86	5.6747
87	4.2621
88	2.8443
89	1.4230
90	0.0000

Max range is 40.7747 at 45 degrees.

The resulting plot is shown in Figure 4.3. The program output matches our hand calculation for the angles calculated to the 4-digit accuracy of the hand calculation. Note that the maximum range occurred at an angle of 45° .

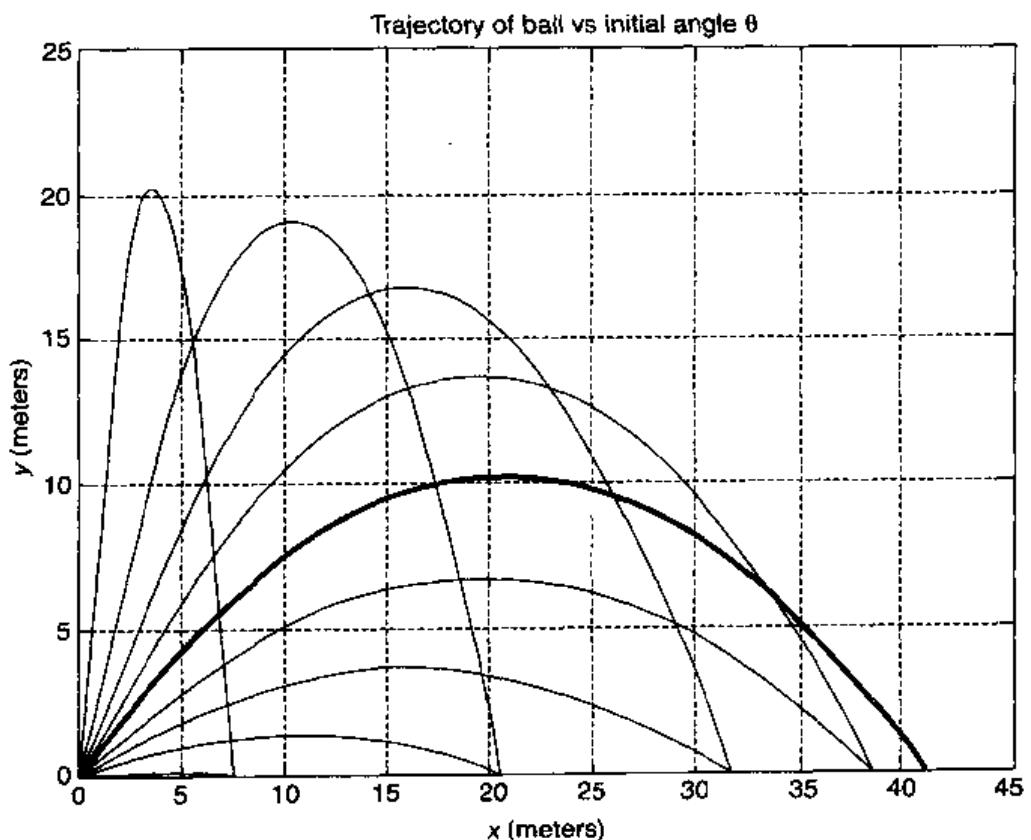


Figure 4.3 Possible trajectories for the ball.

This example uses several of the plotting capabilities that we introduced in Chapter 3. It uses the `axis` command to set the range of data to display, the `hold` command to allow multiple plots to be placed on the same axes, the `LineWidth` property to set the width of the line corresponding to the maximum-range trajectory, and escape sequences to create the desired title and *x*- and *y*-axis labels.

However, this program is not written in the most efficient manner, since there are a number of loops that could have been better replaced by vectorized statements. You will be asked to rewrite and improve `ball.m` in Exercise 4.11 at the end of this chapter.

4.5 Summary

There are two basic types of loops in MATLAB, the `while` loop and the `for` loop. The `while` loop is used to repeat a section of code in cases where we do not know in advance how many times the loop must be repeated. The `for` loop is used to repeat a section of code in cases where we know in advance how many times the loop should be repeated. It is possible to exit from either type of loop at any time using the `break` statement.

4.5.1 Summary of Good Programming Practice

The following guidelines should be adhered to when programming with branch or loop constructs. By following them consistently, your code will contain fewer bugs, will be easier to debug, and will be more understandable to others who may need to work with it in the future.

1. Always indent code blocks in `while` and `for` constructs to make them more readable.
2. Use a `while` loop to repeat sections of code when you do not know in advance how often the loop will be executed.
3. Use a `for` loop to repeat sections of code when you know in advance how often the loop will be executed.
4. Never attempt to modify the values of a `for` loop index while inside the loop.
5. Always initialize all arrays used in a loop before executing the loop. This practice greatly increases the execution speed of the loop.
6. If it is possible to implement a calculation either with a `for` loop or using vectors, *always implement the calculation with vectors*. Your program will be *much faster*.
7. Where possible, use logical arrays as masks to select the elements of an array for processing. If logical arrays are used instead of loops and `if` constructs, your program will be *much faster*.

4.5.2 MATLAB Summary

The following summary lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one.

Commands and Functions

<code>break</code>	Stop the execution of a loop, and transfer control to the first statement after the end of the loop.
<code>continue</code>	Stop the execution of a loop, and transfer control to the top of the loop for the next iteration.
<code>for</code> loop	Loops over a block of statements a specified number of times.
<code>logical</code>	Set the logical attribute on a numeric array.
<code>tic</code>	Resets elapsed time counter.
<code>toc</code>	Returns elapsed time since last call to <code>tic</code> .
<code>while</code> loop	Loops over a block of statements until a test condition becomes 0 (false).

4.6 Exercises

- 4.1** Write the MATLAB statements required to calculate $y(t)$ from the equation

$$y(t) = \begin{cases} -3t^2 + 5 & t \geq 0 \\ 3t^2 + 5 & t < 0 \end{cases}$$

for values of t between -9 and 9 in steps of 0.5 . Use loops and branches to perform this calculation.

- 4.2** Rewrite the statements required to solve Exercise 4.1 using vectorization.
- 4.3** Write the MATLAB statements required to calculate and print out the squares of all the even integers between 0 and 50 . Create a table consisting of each integer and its square, with appropriate labels over each column.
- 4.4** Write an M-file to evaluate the equation $y(x) = x^2 - 3x + 2$ for all values of x between 0.1 and 3 , in steps of 0.1 . Do this twice, once with a `for` loop and once with vectors. Plot the resulting function using a 3-point thick dashed red line.
- 4.5** Write an M-file to calculate the factorial function $N!$, as defined in Example 4.2. Be sure to handle the special case of $0!$ Also, be sure to report an error if N is negative or not an integer.
- 4.6** Examine the following `for` statements and determine how many times each loop will be executed.
- `for ii = -32768:32767`
 - `for ii = 32768:32767`
 - `for kk = 2:4:3`
 - `for jj = ones(5,5)`

- 4.7** Examine the following for loops and determine the value of ires at the end of each of the loops, and also the number of times each loop executes.

```

a. ires = 0;
   for index = -10:10
      ires = ires + 1;
   end
b. ires = 0;
   for index = 10:-2:4
      if index == 0
         continue;
      end
      ires = ires + index;
   end
c. ires = 0;
   for index = 10:-2:4
      if index == 0
         break;
      end
      ires = ires + index;
   end
d. ires = 0;
   for index1 = 10:-2:4
      for index2 = 2:2:index1
         if index2 == 6
            break
         end
         ires = ires + index2;
      end
   end

```

- 4.8** Examine the following while loops and determine the value of ires at the end of each of the loops, and the number of times each loop executes.

```

a. ires = 1;
   while mod(ires,10) ~= 0
      ires = ires + 1;
   end
b. ires = 2;
   while ires <= 200
      ires = ires^2;
   end
c. ires = 2;
   while ires > 200
      ires = ires^2;
   end

```

- 4.9** What is contained in array arr1 after each of the following sets of statements are executed?

```

a. arr1 = [1 2 3 4; 5 6 7 8; 9 10 11 12];
mask = mod(arr1,2) == 0;
arr1(mask) = -arr1(mask);
b. arr1 = [1 2 3 4; 5 6 7 8; 9 10 11 12];
arr2 = arr1 <= 5;
arr1(arr2) = 0;
arr1(~arr2) = arr1(~arr2).^2;

```

- 4.10** How can a numerical array be made to behave as a logical mask for vector operations? How can the logical attribute be removed from a numeric array?
- 4.11** Modify program ball from Example 4.8 by replacing the inner `for` loops with vectorized calculations.
- 4.12** Modify program ball from Example 4.8 to read in the acceleration due to gravity at a particular location and to calculate the maximum range of the ball for that acceleration. After modifying the program, run it with accelerations of -9.8 m/sec^2 , -9.7 m/sec^2 , and -9.6 m/sec^2 . What effect does the reduction in gravitational attraction have on the range of the ball? What effect does the reduction in gravitational attraction have on the best angle θ at which to throw the ball?
- 4.13** Modify program ball from Example 4.8 to read in the initial velocity with which the ball is thrown. After modifying the program, run it with initial velocities of 10 m/sec, 20 m/sec, and 30 m/sec. What effect does changing the initial velocity v_0 have on the range of the ball? What effect does it have on the best angle θ at which to throw the ball?
- 4.14** Program lsqfit from Example 4.7 required the user to specify the number of input data points before entering the values. Modify the program so that it reads an arbitrary number of data values using a `while` loop and stops reading input values when the user presses the Enter key without typing any values. Test your program using the same two data sets that were used in Example 4.7. (*Hint:* The `input` function returns an empty array `[]` if a user presses Enter without supplying any data. You can use function `isempty` to test for an empty array and stop reading data when one is detected.)
- 4.15** Modify program lsqfit from Example 4.7 to read its input values from an ASCII file named `input1.dat`. The data in the file will be organized in rows, with one pair of (x,y) values on each row, as shown below:

```

1.1 2.2
2.2 3.3
...

```

Test your program using the same two data sets that were used in Example 4.7. (*Hint:* Use the `load` command to read the data into an array named `input1`, and then store the first column of `input1` into array `x` and the second column of `input1` into array `y`.)

- 4.16 MATLAB Least-Squares Fit Function.** MATLAB includes a standard function that performs a least-squares fit to a polynomial. Function `polyfit` calculates the least-squares fit of a data set to a polynomial of order N

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 \quad (4.12)$$

where N can be any value greater than or equal to 1. Note that for $N = 1$, this polynomial is a linear equation, with the slope being the coefficient a_1 and the y -intercept being the coefficient a_0 . The form of this function is

```
p = polyfit(x,y,n)
```

where x and y are vectors of x and y components, and n is the order of the fit.

Write a program that calculates the least-squares fit of a data set to a straight line using `polyfit`. Plot the input data points and the resulting fitted line. Compare the result produced by the program using `polyfit` with the result produced by `lsqfit` for the input data set in Example 4.7.

- 4.17** Program `doy` in Example 4.3 calculates the day of year associated with any given month, day, and year. As written, this program does not check to see if the data entered by the user is valid. It will accept nonsense values for months and days and do calculations with them to produce meaningless results. Modify the program so that it checks the input values for validity before using them. If the inputs are invalid, the program should tell the user what is wrong and quit. The year should be a number greater than zero, the month should be a number between 1 and 12, and the day should be a number between 1 and a maximum that depends on the month. Use a `switch` construct to implement the bounds checking performed on the day.

- 4.18** Write a MATLAB program to evaluate the function

$$y(x) = \ln \frac{1}{1-x}$$

- 4.18** for any user-specified value of x , where \ln is the natural logarithm (logarithm to the base e). Write the program with a `while` loop, so that the program repeats the calculation for each legal value of x entered into the program. When an illegal value of x is entered, terminate the program. (Any $x \geq 1$ is considered an illegal value.)
- 4.19** **Fibonacci Numbers.** The n th Fibonacci number is defined by the following recursive equations:

$$\begin{aligned} f(1) &= 1 \\ f(2) &= 2 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

Therefore, $f(3) = f(2) + f(1) = 2 + 1 = 3$, and so forth for higher numbers. Write an M-file to calculate and write out the n th Fibonacci number for $n > 2$, where n is input by the user. Use a `while` loop to perform the calculation.

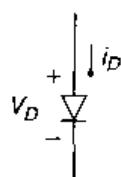


Figure 4.4 A semiconductor diode.

4.20 Current Through a Diode. The current flowing through the semiconductor diode shown in Figure 4.4 is given by the equation

$$i_D = I_o \left(e^{\frac{qV_D}{kT}} - 1 \right) \quad (4.13)$$

where

i_D = the voltage across the diode, in volts

v_D = the current flow through the diode, in amps

I_o = the leakage current of the diode, in amps

q = the charge on an electron, 1.602×10^{-19} coulombs

k = Boltzmann's constant, 1.38×10^{-23} joule/K

T = temperature, in kelvins (K)

The leakage current I_o of the diode is $2.0 \mu\text{A}$. Write a program to calculate the current flowing through this diode for all voltages from -1.0 V to $+0.6 \text{ V}$, in 0.1 V steps. Repeat this process for the following temperatures: 75°F , 100°F , and 125°F . Create a plot of the current as a function of applied voltage, with the curves for the three different temperatures appearing as different colors.

4.21 Tension on a Cable. A 200-pound object is to be hung from the end of a rigid 8-foot horizontal pole of negligible weight, as shown in Figure 4.5. The pole is attached to a wall by a pivot and is supported by an 8-foot cable that is attached to the wall at a higher point. The tension on this cable is given by the equation

$$T = \frac{W \cdot lc \cdot lp}{d \sqrt{lp^2 - d^2}} \quad (4.14)$$

where T is the tension on the cable, W is the weight of the object, lc is the length of the cable, lp is the length of the pole, and d is the distance along the pole at which the cable is attached. Write a program to determine the distance d at which to attach the cable to the pole to minimize the tension on the cable. To do this, the program should calculate the tension on the cable at regular 1-foot intervals from $d = 1$ foot to $d = 7$ feet, and should locate the position d that produces the minimum tension.

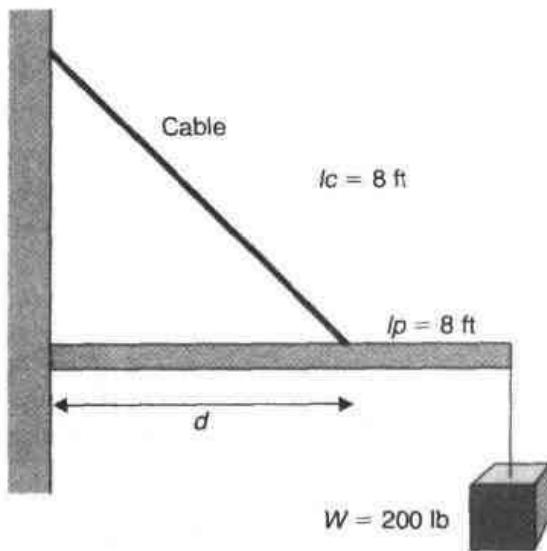


Figure 4.5 A 200-pound weight suspended from a rigid bar supported by a cable.

Also, the program should plot the tension on the cable as a function of d , with appropriate titles and axis labels.

- 4.22 Bacterial Growth.** Suppose that a biologist performs an experiment in which he or she measures the rate at which a specific type of bacterium reproduces asexually in different culture media. The experiment shows that in Medium A the bacteria reproduce once every 60 minutes, and in Medium B the bacteria reproduce once every 90 minutes. Assume that a single bacterium is placed on each culture medium at the beginning of the experiment. Write a program that calculates and plots the number of bacteria present in each culture at intervals of 3 hours from the beginning of the experiment until 24 hours have elapsed. Make two plots, one a linear xy plot and the other a linear-log (semilogy) plot. How do the numbers of bacteria compare on the two media after 24 hours?
- 4.23 Decibels.** Engineers often measure the ratio of two power measurements in *decibels*, or dB. The equation for the ratio of two power measurements in decibels is

$$\text{dB} = 10 \log_{10} \frac{P_2}{P_1} \quad (4.15)$$

where P_2 is the power level being measured, and P_1 is some reference power level. Assume that the reference power level P_1 is 1 watt, and write a program that calculates the decibel level corresponding to power levels between 1 and 20 watts, in 0.5 W steps. Plot the dB-versus-power curve on a log-linear scale.

- 4.24 Geometric Mean.** The *geometric mean* of a set of numbers x_1 through x_n is defined as the n th root of the product of the numbers

$$\text{geometric mean} = \sqrt[n]{x_1 x_2 x_3 \dots x_n} \quad (4.16)$$

Write a MATLAB program that will accept an arbitrary number of positive input values and calculate both the arithmetic mean (*i.e.*, the average) and the geometric mean of the numbers. Use a `while` loop to get the input values and terminate the inputs when a user enters a negative number. Test your program by calculating the average and geometric mean of the four numbers 10, 5, 2, and 5.

- 4.25 RMS Average.** The *root-mean-square* (rms) *average* is another way of calculating a mean for a set of numbers. The rms average of a series of numbers is the square root of the arithmetic mean of the squares of the numbers

$$\text{rms average} = \sqrt{\frac{1}{N} \sum_{i=1}^N x_i^2} \quad (4.17)$$

Write a MATLAB program that will accept an arbitrary number of positive input values and calculate the rms average of the numbers. Prompt the user for the number of values to be entered, and use a `for` loop to read in the numbers. Test your program by calculating the rms average of the four numbers 10, 5, 2, and 5.

- 4.26 Harmonic Mean.** The *harmonic mean* is yet another way of calculating a mean for a set of numbers. The harmonic mean of a set of numbers is given by the equation

$$\text{harmonic mean} = \frac{N}{\frac{1}{x_1} + \frac{1}{x_2} + \dots + \frac{1}{x_N}} \quad (4.18)$$

Write a MATLAB program that will read in an arbitrary number of positive input values and calculate the harmonic mean of the numbers. Use any method that you desire to read in the input values. Test your program by calculating the harmonic mean of the four numbers 10, 5, 2, and 5.

- 4.27** Write a single program that calculates the arithmetic mean (average), rms average, geometric mean, and harmonic mean for a set of positive numbers. Use any method that you desire to read in the input values. Compare these values for each of the following sets of numbers.

- a. 4, 4, 4, 4, 4, 4
- b. 4, 3, 4, 5, 4, 3, 5
- c. 4, 1, 4, 7, 4, 1, 7
- d. 1, 2, 3, 4, 5, 6, 7

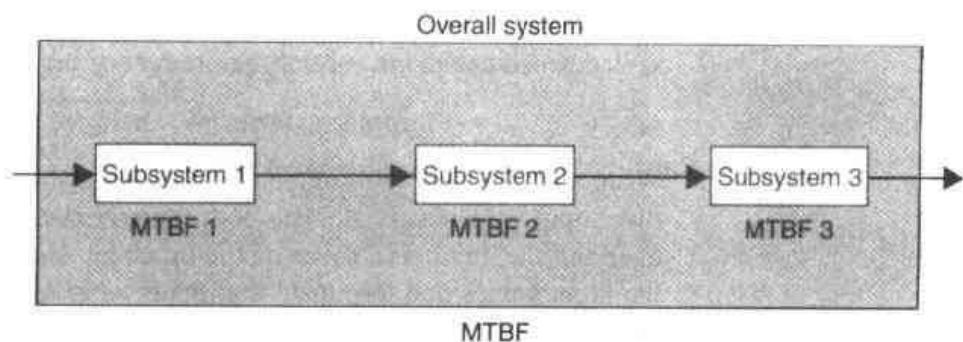


Figure 4.6 An electronic system containing three subsystems with known MTBFs.

4.28 Mean Time Between Failure Calculations. The reliability of a piece of electronic equipment is usually measured in terms of mean time between failures (MTBF), where MTBF is the average time that the piece of equipment can operate before a failure occurs in it. For large systems containing many pieces of electronic equipment, it is customary to determine the MTBFs of each component and to calculate the overall MTBF of the system from the failure rates of the individual components. If the system is structured like the one shown in Figure 4.6, every component must work in order for the whole system to work, and the overall system MTBF can be calculated as

$$\text{MTBF}_{\text{sys}} = \frac{1}{\frac{1}{\text{MTBF}_1} + \frac{1}{\text{MTBF}_2} + \dots + \frac{1}{\text{MTBF}_n}} \quad (4.19)$$

Write a program that reads in the number of series components in a system and the MTBFs for each component, and then calculates the overall MTBF for the system. To test your program, determine the MTBF for a radar system consisting of an antenna subsystem with an MTBF of 2000 hours, a transmitter with an MTBF of 800 hours, a receiver with an MTBF of 3000 hours, and a computer with an MTBF of 5000 hours.

User-Defined Functions

In Chapter 3, we learned the importance of good program design. The basic technique that we employed is **top-down design**. In top-down design, the programmer starts with a statement of the problem to be solved and the required inputs and outputs. Next, he or she describes the algorithm to be implemented by the program in broad outline, and applies **decomposition** to break the algorithm down into logical subdivisions called subtasks. Then, the programmer breaks down each subtask until he or she winds up with many small pieces, each of which does a simple, clearly understandable job. Finally, the individual pieces are turned into MATLAB code.

Although we have followed this design process in our examples, the results have been somewhat restricted because we have had to combine the final MATLAB code generated for each subtask into a single large program. There has been no way to code, verify, and test each subtask independently before combining them into the final program.

Fortunately, MATLAB has a special mechanism designed to make subtasks easy to develop and debug independently before building the final program. It is possible to code each subtask as a separate **function**, and each function can be tested and debugged independently of all the other subtasks in the program.

Well-designed functions enormously reduce the effort required on a large programming project. Their benefits include:

1. Independent Testing of Subtasks

Each subtask can be written as an independent unit. The subtask can be tested separately to ensure that it performs properly by itself before combining it into the larger program. This step is known as **unit**

testing. It eliminates a major source of problems before the final program is even built.

2. Reusable Code

In many cases, the same basic subtask is needed in many parts of a program. For example, it may be necessary to sort a list of values into ascending order many different times within a program, or even in other programs. It is possible to design, code, test, and debug a single function to do the sorting, and then to reuse that function whenever sorting is required. This reusable code has two major advantages: it reduces the total programming effort required and it simplifies debugging, since the sorting function only needs to be debugged once.

3. Isolation from Unintended Side Effects

Functions receive input data from the program that invokes them through a list of variables called an **input argument list** and return results to the program through an **output argument list**. *The only variables in the calling program that can be seen by the function are those in the input argument list, and the only variables in the function that can be seen by the calling program are those in the output argument list.* This is very important, since accidental programming mistakes within a function can only affect the variables within the function in which the mistake occurred.

Once a large program is written and released, it has to be maintained. Program maintenance involves fixing bugs and modifying the program to handle new and unforeseen circumstances. The programmer who modifies a program during maintenance is often not the person who originally wrote it. In poorly written programs, it is common for the programmer modifying the program to make a change in one region of the code and to have that change cause unintended side effects in a totally different part of the program. This happens because variable names are re-used in different portions of the program. When the programmer changes the values left behind in some of the variables, those values are accidentally picked up and used in other portions of the code.

The use of well-designed functions minimizes this problem by **data hiding**. The variables in the main program are not visible to the function (except for those in the input argument list), and the variables in the main program cannot be accidentally modified by anything occurring in the function. Therefore, mistakes or changes in the function's variables cannot accidentally cause unintended side effects in the other parts of the program.

Good Programming Practice

Break large program tasks into functions whenever practical to achieve the important benefits of independent component testing, reusability, and isolation from undesired side effects.

5.1 Introduction to MATLAB Functions

All of the M-files that we have seen so far have been **script files**. Script files are just collections of MATLAB statements that are stored in a file. When a script file is executed, the result is the same as it would be if all of the commands had been typed directly into the Command Window. Script files share the Command Window's workspace, so any variables that were defined before the script file starts are visible to the script file, and any variables created by the script file remain in the workspace after the script file finishes executing. A script file has no input arguments and returns no results, but script files can communicate with other script files through the data left behind in the workspace.

In contrast, a **MATLAB function** is a special type of M-file that runs in its own independent workspace. It receives input data through an **input argument list**, and returns results to the caller through an **output argument list**. The general form of a MATLAB function is

```
function [outarg1, outarg2, ...] = fname(inarg1, inarg2, ...)
% H1 comment line
% Other comment lines
...
(Executable code)
...
(return)
```

The `function` statement marks the beginning of the function. It specifies the name of the function and the input and output argument lists. The input argument list appears in parentheses after the function name, and the output argument list appears in brackets to the left of the equal sign. (If there is only one output argument, the brackets can be dropped.)

The input argument list is a list of names representing values that will be passed from the caller to the function. These names are called **dummy arguments**. They are just placeholders for actual values that are passed from the caller when the function is invoked. Similarly, the output argument list contains a list of dummy arguments that are placeholders for the values returned to the caller when the function finishes executing.

A function is invoked by naming it in an expression together with a list of **actual arguments**. A function can be invoked by typing its name directly in the Command Window, or by including it in a script file or another function. When the function is invoked, the value of the first actual argument is used in place of the first dummy argument, and so forth for each other actual argument / dummy argument pair.

Execution begins at the top of the function, and ends when either a `return` statement or the end of the function is reached. Because execution stops at the end of a function anyway, the `return` statement is not actually required in most functions, and is rarely used. Each item in the output argument list must appear on the left side of at least one assignment statement in the function. When the function returns, the values stored in the output argument list are returned to the caller and can be used in further calculations.

The initial comment lines in a function serve a special purpose. The first comment line after the function statement is called the **H1 comment line**. It should always contain a one-line summary of the purpose of the function. The special significance of this line is that it is searched and displayed by the `lookfor` command. The remaining comment lines from the H1 line until the first blank line or the first executable statement are displayed by the `help` command and the Help Window. They should contain a brief summary of how to use the function.

A simple example of a user-defined function follows. Function `dist2` calculates the distance between points (x_1, y_1) and (x_2, y_2) in a Cartesian coordinate system.

```
function distance = dist2 (x1, y1, x2, y2)
%DIST2 Calculate the distance between two points
% Function DIST2 calculates the distance between
% two points (x1,y1) and (x2,y2) in a Cartesian
% coordinate system.
%
% Calling sequence:
%     res = dist2(x1, y1, x2, y2)

% Define variables:
%     x1      -- x-position of point 1
%     y1      -- y-position of point 1
%     x2      -- x-position of point 2
%     y2      -- y-position of point 2
%     distance -- Distance between points

% Record of revisions:
%     Date      Programmer          Description of change
%     ===       ======             =====
%     12/15/98   S. J. Chapman    Original code

% Calculate distance.
distance = sqrt((x2-x1).^2 + (y2-y1).^2);
```

This function has four input arguments and one output argument. A simple script file using this function is shown below.

```
% Script file: test_dist2.m
%
% Purpose:
%     This program tests function dist2.
%
% Record of revisions:
%     Date      Programmer          Description of change
%     ===       ======             =====
%     12/15/98   S. J. Chapman    Original code
%
% Define variables:
%     ax      -- x-position of point a
```

```

% ay -- y-position of point a
% bx -- x-position of point b
% by -- y-position of point b
% result -- Distance between the points

% Get input data.
disp('Calculate the distance between two points:');
ax = input('Enter x value of point a: ');
ay = input('Enter y value of point a: ');
bx = input('Enter x value of point b: ');
by = input('Enter y value of point b: ');

% Evaluate function
result = dist2 (ax, ay, bx, by);

% Write out result.
fprintf('The distance between points a and b is %f\n',result);

```

When this script file is executed, the results are

```

>> test_dist2
Calculate the distance between two points:
Enter x value of point a: 1
Enter y value of point a: 1
Enter x value of point b: 4
Enter y value of point b: 5
The distance between points a and b is 5.000000

```

These results are correct, as we can verify from simple hand calculations.

Function `dist2` also supports the MATLAB help subsystem. If we type “`help dist2`”, the results are:

```

>> help dist2
DIST2 Calculate the distance between two points
    Function DIST2 calculates the distance between
    two points (x1,y1) and (x2,y2) in a Cartesian
    coordinate system.

    Calling sequence:
        res = dist2(x1, y1, x2, y2)

```

Similarly, “`lookfor distance`” produces the result

```

>> lookfor distance
DIST2 Calculate the distance between two points
MAHAL Mahalanobis distance.
DIST Distances between vectors.
NBDIST Neighborhood matrix using vector distance.
NBGRID Neighborhood matrix using grid distance.
NBMAN Neighborhood matrix using Manhattan-distance.

```

To observe the behavior of the workspace before, during, and after the function is executed, we will load function `dist2` and the script file `test_dist2` into

```

d:\book\matlab\2e\rev1\chap5\test_dist2.m
File Edit View Text Debug Breakpoints Web Window Help
Stack: test_dist2
1 % Script file: test_dist2.m
2 %
3 % Purpose:
4 %   This program tests function dist2.
5 %
6 % Record of revisions:
7 %   Date      Programmer      Description of change
8 %   ===       =====          =====
9 %   12/15/98  S. J. Chapman  Original code
10 %
11 % Define variables:
12 %   ax      -- x-position of point a
13 %   ay      -- y-position of point a
14 %   bx      -- x-position of point b
15 %   by      -- y-position of point b
16 %   result -- Distance between the points
17 %
18 % Get input data.
19 - disp('Calculate the distance between two points:');
20 - ax = input('Enter x value of point a: ');
21 - ay = input('Enter y value of point a: ');
22 - bx = input('Enter x value of point b: ');
23 - by = input('Enter y value of point b: ');
24 -
25 % Evaluate function
26 • result = dist2 (ax, ay, bx, by);
27 -
28 % Write out result.
29 • fprintf('The distance between points a and b is %f\n',result);
30

```

test_dist2.m dist2.m

Ready

Figure 5.1 M-file `test_dist2` and function `dist2` are loaded into the debugger, with breakpoints set before, during, and after the function call.

the MATLAB debugger, and set breakpoints before, during, and after the function call (see Figure 5.1). When the program stops at the breakpoint *before* the function call, the workspace is as shown in Figure 5.2(a). Note that variables `ax`, `ay`, `bx`, and `by` are defined in the workspace. When the program stops at the breakpoint *within* the function call, the workspace is as shown in Figure 5.2(b). Note that variables `x1`, `x2`, `y1`, `y2`, and `distance` are defined in the workspace, and the variables defined in the calling M-file are gone. When the program stops at the breakpoint *after* the function call, the workspace is as shown in Figure 5.2(c). Note that the original variables are back, with the variable `result` added to contain the value returned by the function. These figures show that the workspace of the function is different than the workspace of the calling M-file.

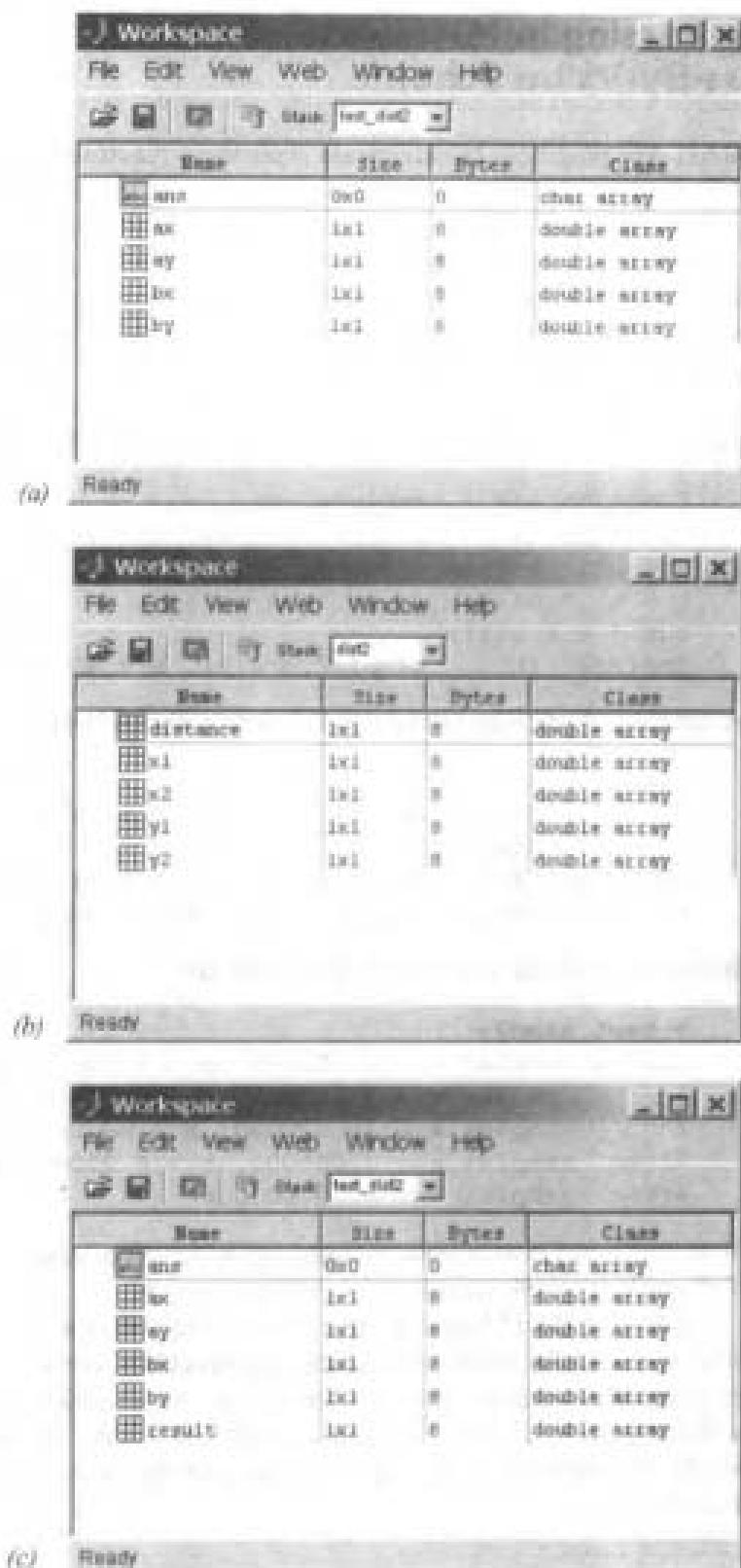


Figure 5.2 (a) The workspace before the function call. (b) The workspace during the function call. (c) The workspace after the function call.

5.2 Variable Passing in MATLAB: The Pass-By-Value Scheme

MATLAB programs communicate with their functions using a **pass-by-value** scheme. When a function call occurs, MATLAB makes a *copy* of the actual arguments and passes them to the function. This copying is very significant because it means that even if the function modifies the input arguments, it will not affect the original data in the caller. This feature helps to prevent unintended side effects in which an error in the function might unintentionally modify variables in the calling program.

This behavior is illustrated in the following function. This function has two input arguments: *a* and *b*. During its calculations, it modifies both input arguments.

```
function out = sample(a, b)
fprintf('In      sample: a = %f, b = %f %f\n',a,b);
a = b(1) + 2*a;
b = a .* b;
out = a + b(1);
fprintf('In      sample: a = %f, b = %f %f\n',a,b);
```

A simple test program to call this function is shown below.

```
a = 2; b = [6 4];
fprintf('Before sample: a = %f, b = %f %f\n',a,b);
out = sample(a,b);
fprintf('After  sample: a = %f, b = %f %f\n',a,b);
fprintf('After  sample: out = %f\n',out);
```

When this program is executed, the results are

```
> test_sample
Before sample: a = 2.000000, b = 6.000000 4.000000
In      sample: a = 2.000000, b = 6.000000 4.000000
In      sample: a = 10.000000, b = 60.000000 40.000000
After  sample: a = 2.000000, b = 6.000000 4.000000
After  sample: out = 70.000000
```

Note that *a* and *b* were both changed inside function *sample*, but those changes had *no effect on the values in the calling program*.

Users of the C language will be familiar with the pass-by-value scheme, since C uses it for scalar values passed to functions. However C does *not* use the pass-by-value scheme when passing arrays, so an unintended modification to a dummy array in a C function causes side effects in the calling program. MATLAB improves on this by using the pass-by-value scheme for both scalars and arrays.¹

¹ The implementation of argument passing in MATLAB is actually more sophisticated than this discussion indicates. As pointed out above, the copying associated with pass-by-value takes up a lot of time, but it provides protection against unintended side effects. MATLAB actually uses the best of

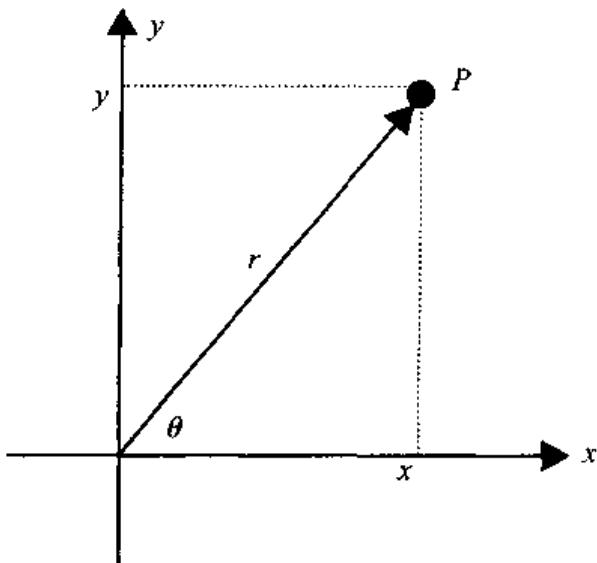


Figure 5.3 A point P in a Cartesian plane can be located by either the rectangular coordinates (x,y) or the polar coordinates (r,θ) .



Example 5.1—Rectangular-to-Polar Conversion

The location of a point in a cartesian plane can be expressed in either the rectangular coordinates (x,y) or the polar coordinates (r,θ) , as shown in Figure 5.3. The relationships among these two sets of coordinates are given by the following equations:

$$x = r \cos \theta \quad (5.1)$$

$$y = r \sin \theta \quad (5.2)$$

$$r = \sqrt{x^2 + y^2} \quad (5.3)$$

$$\theta = \tan^{-1} \frac{y}{x} \quad (5.4)$$

Write two functions `rect2polar` and `polar2rect` that convert coordinates from rectangular to polar form, and vice versa, where the angle θ is expressed in degrees.

Solution

We will now apply our standard problem-solving approach to creating these functions. Note that MATLAB's trigonometric functions work in units of radians,

both approaches: it analyzes each argument of each function and determines whether or not the function modifies that argument. If the function modifies the argument, then MATLAB makes a copy of it. If it does not modify the argument, then MATLAB simply points to the existing value in the calling program. This practice increases speed while still providing protection against side effects!

so we must convert from degrees to radians and vice versa when solving this problem. The basic relationship between degrees and radians is

$$180^\circ = \pi \text{ radians} \quad (5.5)$$

1. State the problem

A succinct statement of the problem is:

Write a function that converts a location on a Cartesian plane expressed in rectangular coordinates into the corresponding polar coordinates, with the angle θ expressed in degrees. Also, write a function that converts a location on a Cartesian plane expressed in polar coordinates with the angle θ expressed in degrees into the corresponding rectangular coordinates.

2. Define the inputs and outputs

The inputs to function `rect2polar` are the rectangular (x,y) location of a point. The outputs of the function are the polar (r, θ) location of the point. The inputs to function `polar2rect` are the polar (r, θ) location of a point. The outputs of the function are the rectangular (x,y) location of the point.

3. Describe the algorithm

These functions are very simple, so we can directly write the final pseudocode for them. The pseudocode for function `polar2rect` is

```
x ← r * cos(theta * pi/180)
y ← r * sin(theta * pi/180)
```

The pseudocode for function `rect2polar` will use the function `atan2` because that function works over all four quadrants of the Cartesian plane. (Look for that function in the MATLAB help systems!)

```
r ← sqrt(x.^2 + y.^2)
theta ← 180/pi * atan2(y,x)
```

4. Turn the algorithm into MATLAB statements

The MATLAB code for the selection `polar2rect` function is shown below.

```
function [x, y] = polar2rect(r, theta)
%POLAR2RECT Convert rectangular to polar coordinates
% Function POLAR2RECT accepts the polar coordinates
% (r,theta), where theta is expressed in degrees,
% and converts them into the rectangular coordinates
% (x,y).
%
% Calling sequence:
%   [x, y] = polar2rect(r,theta)

% Define variables:
```

```

% r          -- Length of polar vector
% theta      -- Angle of vector in degrees
% x          -- x-position of point
% y          -- y-position of point

% Record of revisions:
% Date      Programmer      Description of change
% =====    ======        =====
% 09/19/00   S. J. Chapman  Original code

x = r * cos(theta * pi/180);
y = r * sin(theta * pi/180);

```

The MATLAB code for the selection `rect2polar` function is shown below.

```

function [r, theta] = rect2polar(x,y)
%RECT2POLAR Convert rectangular to polar coordinates
% Function RECT2POLAR accepts the rectangular coordinates
% (x,y) and converts them into the polar coordinates
% (r,theta), where theta is expressed in degrees.
%
% Calling sequence:
% [r, theta] = rect2polar(x,y)

% Define variables:
% r          -- Length of polar vector
% theta      -- Angle of vector in degrees
% x          -- x-position of point
% y          -- y-position of point

% Record of revisions:
% Date      Programmer      Description of change
% =====    ======        =====
% 09/19/00   S. J. Chapman  Original code

r = sqrt( x.^2 + y .^2 );
theta = 180/pi * atan2(y,x);

```

Note that these functions both include help information, so they will work properly with MATLAB's help subsystem and with the `lookfor` command.

5. Test the program

To test these functions, we will execute them directly in the MATLAB Command Window. We will test the functions using the 3-4-5 triangle, which is familiar to most people from secondary school. The smaller angle within a 3-4-5 triangle is approximately 36.87° . We will also test the

function in all four quadrants of the Cartesian plane to ensure that the conversion are correct everywhere.

```
>> [r, theta] = rect2polar(4,3)
r =
    5
theta =
  36.8699
>> [r, theta] = rect2polar(-4,3)
r =
    5
theta =
  143.1301
>> [r, theta] = rect2polar(-4,-3)
r =
    5
theta =
 -143.1301
>> [r, theta] = rect2polar(4,-3)
r =
    5
theta =
 -36.8699
>> [x, y] = polar2rect(5,36.8699)
x =
  4.0000
y =
  3.0000
>> [x, y] = polar2rect(5,143.1301)
x =
 -4.0000
y =
  3.0000
>> [x, y] = polar2rect(5,-143.1301)
x =
 -4.0000
y =
 -3.0000
>> [x, y] = polar2rect(5,-36.8699)
x =
  4.0000
y =
 -3.0000
>>
```

These functions appear to be working correctly in all quadrants of the Cartesian plane.



Example 5.2—Sorting Data

In many scientific and engineering applications, it is necessary to take a random input data set and to sort it so that the numbers in the data set are either all in *ascending order* (lowest-to-highest) or all in *descending order* (highest-to-lowest). For example, suppose that you were a zoologist studying a large population of animals and that you wanted to identify the largest 5% of the animals in the population. The most straightforward way to approach this problem would be to sort the sizes of all the animals in the population into ascending order and take the top 5% of the values.

Sorting data into ascending or descending order seems to be an easy job. After all, we do it all the time. It is a simple matter for us to sort the data (10, 3, 6, 4, 9) into the order (3, 4, 6, 9, 10). How do we do it? We first scan the input data list (10, 3, 6, 4, 9) to find the smallest value in the list (3), and then scan the remaining input data (10, 6, 4, 9) to find the next smallest value (4), and continue until the complete list is sorted.

In fact, sorting can be a very difficult job. As the number of values to be sorted increases, the time required to perform the simple sort described above increases rapidly, since we must scan the input data set once for each value sorted. For very large data sets, this technique just takes too long to be practical. Even worse, how would we sort the data if there were too many numbers to fit into the main memory of the computer? The development of efficient sorting techniques for large data sets is an active area of research and is the subject of courses all by itself.

In this example, we will confine ourselves to the simplest possible algorithm to illustrate the concept of sorting. This simplest algorithm is called the **selection sort**. It is just a computer implementation of the mental math described previously. The basic algorithm for the selection sort is as follows.

1. Scan the list of numbers to be sorted to locate the smallest value in the list. Place that value at the front of the list by swapping it with the value currently at the front of the list. If the value at the front of the list is already the smallest value, then do nothing.
2. Scan the list of numbers from position 2 to the end to locate the next smallest value in the list. Place that value in position 2 of the list by swapping it with the value currently at that position. If the value in position 2 is already the next smallest value, then do nothing.
3. Scan the list of numbers from position 3 to the end to locate the third smallest value in the list. Place that value in position 3 of the list by swapping it with the value currently at that position. If the value in position 3 is already the third smallest value, then do nothing.
4. Repeat this process until the next-to-last position in the list is reached. After the next-to-last position in the list has been processed, the sort is complete.

Note that if we are sorting N values, this sorting algorithm requires $N-1$ scans through the data to accomplish the sort.

This process is illustrated in Figure 5.4. Since there are 5 values in the data set to be sorted, we will make 4 scans through the data. During the first pass through the entire data set, the minimum value is 3, so the 3 is swapped with the 10 that was in position 1. Pass 2 searches for the minimum value in positions 2 through 5. That minimum is 4, so the 4 is swapped with the 10 in position 2. Pass 3 searches for the minimum value in positions 3 through 5. That minimum is 6, which is already in position 3, so no swapping is required. Finally, pass 4 searches for the minimum value in positions 4 through 5. That minimum is 9, so the 9 is swapped with the 10 in position 4, and the sort is completed.

Programming Pitfalls

The selection sort algorithm is the easiest sorting algorithm to understand, but it is computationally inefficient. *It should never be applied to sort large data sets* (say, sets with more than 1000 elements). Over the years, computer scientists have developed much more efficient sorting algorithms. The `sort` and `sortrows` functions built into MATLAB are extremely efficient and should be used for all real work.

We will now develop a program to read in a data set from the Command Window, sort it into ascending order, and display the sorted data set. The sorting will be done by a separate user-defined function.

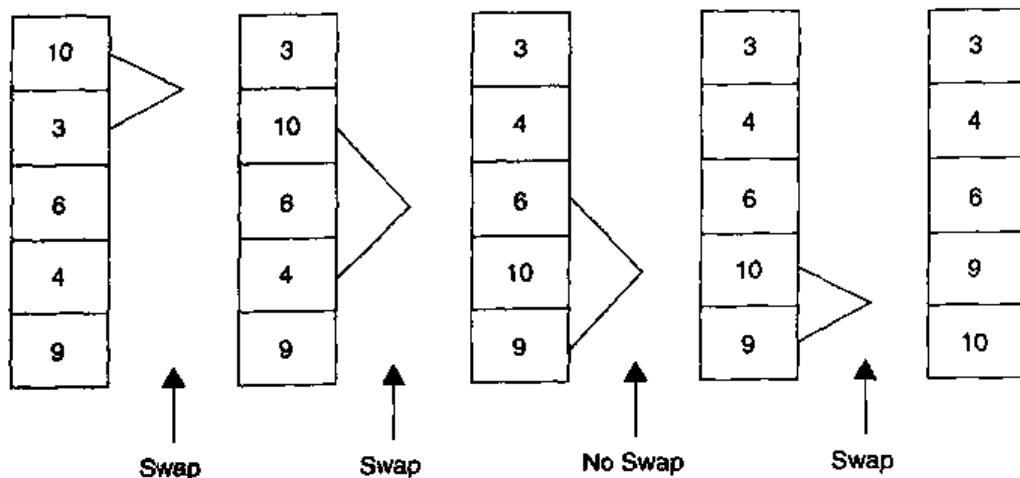


Figure 5.4 An example problem demonstrating the selection sort algorithm.

Solution

This program must be able to ask the user for the input data, sort the data, and write out the sorted data. The design process for this problem follows.

1. State the problem

We have not yet specified the type of data to be sorted. If the data is numeric, then the problem can be stated as follows.

Develop a program to read an arbitrary number of numeric input values from the Command Window, sort the data into ascending order using a separate sorting function, and write the sorted data to the Command Window.

2. Define the inputs and outputs

The inputs to this program are the numeric values typed in the Command Window by the user. The outputs from this program are the sorted data values written to the Command Window.

3. Describe the algorithm

This program can be broken down into three major steps.

Read the input data into an array
 Sort the data in ascending order
 Write the sorted data

The first major step is to read in the data. We must prompt the user for the number of input data values, and then read in the data. Since we will know how many input values there are to read, a `for` loop is appropriate for reading in the data. The detailed pseudocode follows.

```
Prompt user for the number of data values
Read the number of data values
Preallocate an input array
for ii = 1:number of values
    Prompt for next value
    Read value
end
```

Next, we must sort the data in a separate function. We will need to make `nvals-1` passes through the data, finding the smallest remaining value each time. We will use a pointer to locate the smallest value in each pass. Once the smallest value is found, it will be swapped to the top of the list if it is not already there. The detailed pseudocode follows:

```
for ii = 1:nvals-1
    % Find the minimum value in a(ii) through a(nvals)
    iptr ← ii
    for jj = ii+1 to nvals
        if a(jj) < a(iptr)
            iptr ← jj
        end
    end
end
```

```

        % iptr now points to the min value, so swap a(iptr)
        % with a(ii) if iptr ~= ii.
        if ii ~= iptr
            temp ← a(ii)
            a(ii) ← a(iptr)
            a(iptr) ← temp
        end
    end

```

The final step is writing out the sorted values. No refinement of the pseudocode is required for that step. The final pseudocode is the combination of the reading, sorting, and writing steps.

4. Turn the algorithm into MATLAB statements

The MATLAB code for the selection sort function follows.

```

function out = ssort(a)
%SSORT Selection sort data in ascending order
% Function SSORT sorts a numeric data set into
% ascending order. Note that the selection sort
% is relatively inefficient. DO NOT USE THIS
% FUNCTION FOR LARGE DATA SETS. Use MATLAB's
% "sort" function instead.

% Define variables:
%   a           -- Input array to sort
%   ii          -- Index variable
%   iptr        -- Pointer to min value
%   jj          -- Index variable
%   nvals       -- Number of values in "a"
%   out         -- Sorted output array
%   temp        -- Temp variable for swapping

% Record of revisions:
%      Date      Programmer      Description of change
%      =====      ======      =====
%      12/19/98    S. J. Chapman  Original code

% Get the length of the array to sort
nvals = size(a,2);

% Sort the input array
for ii = 1:nvals-1

    % Find the minimum value in a(ii) through a(n)
    iptr = ii;
    for jj = ii+1:nvals
        if a(jj) < a(iptr)
            iptr = jj;
        end
    end

```

```

% iptr now points to the minimum value, so swap a(iptr)
% with a(ii) if ii ~= iptr.
if ii ~= iptr
    temp      = a(ii);
    a(ii)    = a(iptr);
    a(iptr) = temp;
end

end

% Pass data back to caller
out = a;

```

The program to invoke the selection sort function follows.

```

% Script file: test_ssort.m
%
% Purpose:
%   To read in an input data set, sort it into ascending
%   order using the selection sort algorithm, and to
%   write the sorted data to the Command Window. This
%   program calls function "ssort" to do the actual
%   sorting.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   ===       ======          =====
%   12/19/98   S. J. Chapman  Original code
%
% Define variables:
%   array -- Input data array
%   ii    -- Index variable
%   nvals -- Number of input values
%   sorted -- Sorted data array

% Prompt for the number of values in the data set
nvals = input('Enter number of values to sort: ');

% Preallocate array
array = zeros(1,nvals);

% Get input values
for ii = 1:nvals

    % Prompt for next value
    string = ['Enter value ' int2str(ii) ': '];
    array(ii) = input(string);

end

% Now sort the data
sorted = ssort(array);

```

```

% Display the sorted result.
fprintf('\nSorted data:\n');
for ii = 1:nvals
    fprintf(' %8.4f\n',sorted(ii));
end

```

5. Test the program

To test this program, we will create an input data set and run the program with it. The data set should contain a mixture of positive and negative numbers as well as at least one duplicated value to see if the program works properly under those conditions.

```

>> test_msort
Enter number of values to sort: 6
Enter value 1: -5
Enter value 2: 4
Enter value 3: -2
Enter value 4: 3
Enter value 5: -2
Enter value 6: 0

Sorted data:
-5.0000
-2.0000
-2.0000
0.0000
3.0000
4.0000

```

The program gives the correct answers for our test data set. Note that it works for both positive and negative numbers as well as for repeated numbers.

5.3 Optional Arguments

Many MATLAB functions support optional input arguments and output arguments. For example, we have seen calls to the `plot` function with as few as two or as many as seven input arguments. On the other hand, the function `max` supports either one or two output arguments. If there is only one output argument, `max` returns the maximum value of an array. If there are two output arguments, `max` returns both the maximum value and the location of the maximum value in an array. How do MATLAB functions know how many input and output arguments are present, and how do they adjust their behavior accordingly?

There are eight special functions that can be used by MATLAB functions to get information about their optional arguments and to report errors in those

arguments. Six of these functions are introduced here, and the remaining two will be introduced in Chapter 7 after we learn about the cell array data type.

■ nargin	This function returns the number of actual input arguments that were used to call the function.
■ nargout	This function returns the number of actual output arguments that were used to call the function.
■ nargchk	This function returns a standard error message if a function is called with too few or too many arguments.
■ error	Display error message and abort the function producing the error. This function is used if the argument errors are fatal.
■ warning	Display warning message and continue function execution. This function is used if the argument errors are not fatal, and execution can continue.
■ inputname	This function returns the actual name of the variable that corresponds to a particular argument number.

Functions nargin and nargout can only be used within a user-defined function. Whenever they are called, these functions return the number of actual input arguments and output arguments respectively. Function nargchk generates a string containing a standard error message if a function is called with too few or too many arguments. The syntax of this function is

```
message = nargchk(min_args,max_args,num_args);
```

where min_args is the minimum number of arguments, max_args is the maximum number of arguments, and num_args is the actual number of arguments. If the number of arguments is outside the acceptable limits, a standard error message is produced. If the number of arguments is within acceptable limits, then an empty string is returned.

Function error is a standard way to display an error message and abort the user-defined function causing the error. The syntax of this function is error('msg'), where msg is a character string containing an error message. When error is executed, it halts the current function and returns to the keyboard, displaying the error message in the Command Window. If the message string is empty, error does nothing, and execution continues. This function works well with nargchk, which produces a message string when an error occurs and an empty string when there is no error.

Function warning is a standard way to display a warning message that includes the function and line number where the problem occurred, but lets execution continue. The syntax of this function is warning('msg'), where msg is a character string containing a warning message. When warning is executed, it displays the warning message in the Command Window, and lists the function name and line number where the warning came from. If the message string is empty, warning does nothing. In either case, execution of the function continues.

Function `inputname` returns the name of the actual argument used when a function is called. The syntax of this function is

```
name = inputname(argno);
```

where `argno` is the number of the argument. If argument is a variable, then its name is returned. If the argument is an expression, then this function will return an empty string. For example, consider the function

```
function myfun(x,y,z)
name = inputname(2);
disp(['The second argument is named ' name]);
```

When this function is called, the results are

```
> myfun(dog,cat)
The second argument is named cat
> myfun(1,2+cat)
The second argument is named
```

Function `inputname` is useful for displaying argument names in warning and error messages.

Example 5.3—Using Optional Arguments

We will illustrate the use of optional arguments by creating a function that accepts an (x,y) value in rectangular coordinates and produces the equivalent polar representation consisting of a magnitude and an angle in degrees. The function is designed to support two input arguments, x and y . However, if only one argument is supplied, the function assumes that the y value is zero and proceeds with the calculation. The function normally returns both the magnitude and the angle in degrees, but if only one output argument is present, it returns only the magnitude. This function is shown below.

```
function [mag, angle] = polar_value(x,y)
% POLAR_VALUE Converts (x,y) to (r,theta)
% Function POLAR_VALUE converts an input (x,y)
% value into (r,theta), with theta in degrees.
% It illustrates the use of optional arguments.

% Define variables:
% angle -- Angle in degrees
% msg -- Error message
% mag -- Magnitude
% x -- Input x value
% y -- Input y value (optional)

% Record of revisions:
% Date Programmer Description of change
% ===== ===== =====
% 12/16/98 S. J. Chapman Original code
```

```

% Check for a legal number of input arguments.
msg = narginchk(1,2,nargin);
error(msg);

% If the y argument is missing, set it to 0.
if nargin < 2
    y = 0;
end

% Check for (0,0) input arguments, and print out
% a warning message.
if x == 0 & y == 0
    msg = 'Both x and y are zero: angle is meaningless!';
    warning(msg);
end

% Now calculate the magnitude.
mag = sqrt(x.^2 + y.^2);

% If the second output argument is present, calculate
% angle in degrees.
if nargout == 2
    angle = atan2(y,x) * 180/pi;
end

```

We will test this function by calling it repeatedly from the Command Window. First, we will try to call the function with too few or too many arguments.

```

» [mag angle] = polar_value
??? Error using ==> polar_value
Not enough input arguments.

» [mag angle] = polar_value(1,-1,1)
??? Error using ==> polar_value
Too many input arguments.

```

The function provides proper error messages in both cases. Next, we will try to call the function with one or two input arguments.

```

» [mag angle] = polar_value(1)
mag =
    1
angle =
    0
» [mag angle] = polar_value(1,-1)
mag =
    1.4142
angle =
   -45

```

The function provides the correct answer in both cases. Next, we will try to call the function with one or two output arguments.

```
> mag = polar_value(1,-1)
mag =
    1.4142
> [mag angle] = polar_value(1,-1)
mag =
    1.4142
angle =
   -45
```

The function provides the correct answer in both cases. Finally, we will try to call the function with both x and y equal to zero.

```
> [mag angle] = polar_value(0,0)

Warning: Both x and y are zero: angle is meaningless!
> In d:\book\matlab\chap5\polar_value.m at line 32
mag =
    0
angle =
    0
```

In this case, the function displays the warning message, but execution continues.

Note that a MATLAB function can be declared to have more output arguments than are actually used, and this is not an error. The function does not actually have to check `nargout` to determine if an output argument is present. For example, consider the following function.

```
function [z1, z2] = junk(x,y)
z1 = x + y;
z2 = x - y;
```

This function can be called successfully with one or two output arguments.

```
> a = junk(2,1)
a =
    3
> [a b] = junk(2,1)
a =
    3
b =
    1
```

The reason for checking `nargout` in a function is to prevent useless work. If a result is going to be thrown away anyway, why bother to calculate it in the first place? A programmer can speed up the operation of a program by not bothering with useless calculations.

Quiz 5.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 5.1 through 5.3. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. What are the differences between a script file and a function?
2. How does the help command work with user-defined functions?
3. What is the significance of the H1 comment line in a function?
4. What is the pass-by-value scheme? How does it contribute to good program design?
5. How can a MATLAB function be designed to have optional arguments?

For questions 6 and 7, determine whether the function calls are correct or not. If they are in error, specify what is wrong with them.

6. `out = test1(6);`

```
function res = test1(x,y)
res = sqrt(x.^2 + y.^2);
```

7. `out = test2(12);`

```
function res = test2(x,y)
error(nargchk(1,2,nargin));
if nargin == 2
    res = sqrt(x.^2 + y.^2);
else
    res = x;
end
```

5.4 Sharing Data Using Global Memory

We have seen that programs exchange data with the functions they call through argument lists. When a function is called, each actual argument is copied, and the copy is used by the function.

In addition to the argument list, MATLAB functions can exchange data with each other and with the base workspace through global memory. **Global memory** is a special type of memory that can be accessed from any workspace. If a variable is declared to be global in a function, then it will be placed in the global memory instead of the local workspace. If the same variable is declared to be global in another function, then that variable will refer to the *same memory location* as the variable in the first function. Each script file or function that declares the global variable will have access to the same data values, so *global memory provides a way to share data between functions*.

A global variable is declared with the **global** statement. The form of a **global** statement is

```
global var1 var2 var3 ...
```

where *var1*, *var2*, *var3*, and so forth, are the variables to be placed in global memory. By convention, global variables are declared in all capital letters, but this is not actually a requirement.

Good Programming Practice

Declare global variables in all capital letters to make them easy to distinguish from local variables.

Each global variable must be declared to be global before it is used for the first time in a function—it is an error to declare a variable to be global after it has already been created in the local workspace. To avoid this error, it is customary to declare global variables immediately after the initial comments and before the first executable statement in a function.

Good Programming Practice

Declare global variables immediately after the initial comments and before the first executable statement in each function that uses them.

Global variables are especially useful for sharing very large volumes of data among many functions because the entire data set does not have to be copied each time that a function is called. The downside of using global memory to exchange data among functions is that the functions will only work for that specific data set. A function that exchanges data through input arguments can be reused by simply calling it with different arguments, but a function that exchanges data through global memory must actually be modified to allow it to work with a different data set.

Global variables are also useful for sharing hidden data among a group of related functions, while keeping it invisible from the calling program.

Good Programming Practice

You can use global memory to pass large amounts of data among functions within a program.

Example 5.4—Random Number Generator

It is impossible to make perfect measurements in the real world. There will always be some *measurement noise* associated with each measurement. This fact is an important consideration in the design of systems to control the operation of such real-world devices as airplanes, refineries, and so forth. A good engineering design must take these measurement errors into account so that the noise in the measurements will not lead to unstable behavior (no plane crashes or refinery explosions!).

Most engineering designs are tested by running *simulations* of the operation of the system before it is ever built. These simulations involve creating mathematical models of the behavior of the system and feeding the models a realistic string of input data. If the models respond correctly to the simulated input data, then we can have reasonable confidence that the real-world system will respond correctly to the real-world input data.

The simulated input data supplied to the models must be corrupted by a simulated measurement noise, which is just a string of random numbers added to the ideal input data. The simulated noise is usually produced by a *random number generator*.

A random number generator is a function that will return a different and apparently random number each time it is called. Since the numbers are in fact generated by a deterministic algorithm, they only appear to be random.² However, if the algorithm used to generate them is complex enough, the numbers will be random enough to use in the simulation.

One simple random number generator algorithm is shown below.³ It relies on the unpredictability of the modulo function when applied to large numbers. Consider the following equation.

$$n_{i+1} = \text{mod}(8121 n_i + 28411, 134456) \quad (5.6)$$

Assume that n_i is a non-negative integer. Then because of the modulo function, n_{i+1} will be a number between 0 and 134,455 inclusive. Next, this value can be used as n_i and fed back into the equation to produce a new number n_{i+1} that is also between 0 and 134,455. This process can be repeated forever to produce a series of numbers in the range [0,134455]. If we did not know the numbers 8121, 28,411, and 134,456 in advance, it would be impossible to guess the order in which the values of n would be produced. Furthermore, it turns out that there is an equal (or uniform) probability that any given number will appear in the sequence. Because of these properties, Equation (5.6) can serve as the basis for a simple random number generator with a uniform distribution.

² For this reason, some people refer to these functions as *pseudorandom number generators*.

³ This algorithm is adapted from the discussion found in Chapter 7 of *Numerical Recipes: The Art of Scientific Programming*, by Press, Flannery, Teukolsky, and Vetterling, Cambridge University Press, 1986.

We will now use Equation (5.6) to design a random number generator whose output is a real number in the range [0.0, 1.0].⁴

Solution

We will write a function that generates one random number in the range $0 \leq ran < 1.0$ each time that it is called. The random number will be based on the equation

$$ran_i = \frac{n_i}{134456} \quad (5.7)$$

where n_i is a number in the range 0 to 134,455 produced by Equation (5.6).

The particular sequence produced by Equations (5.6) and (5.7) will depend on the initial value of n_0 (called the *seed*) of the sequence. We must provide a way for the user to specify n_0 so that the sequence can be varied from run to run.

1. State the problem

Write a function `random0` that will generate and return an array `ran` containing one or more numbers with a uniform probability distribution in the range $0 \leq ran < 1.0$, based on the sequence specified by Equations (5.6) and (5.7). The function should have one or two input arguments (`n` and `m`) specifying the size of the array to return. If there is one argument, the function should generate square array of size $n \times n$. If there are two arguments, the function should generate an array of size $n \times m$. The initial value of the seed n_0 will be specified by a call to a function called `seed`.

2. Define the inputs and outputs

There are two functions in this problem: `seed` and `random0`. The input to function `seed` is an integer to serve as the starting point of the sequence. There is no output from this function. The input to function `random0` is one or two integers specifying the size of the array of random numbers to be generated. If only argument `n` is supplied, the function should generate a square array of size $n \times n$. If both arguments `n` and `m` are supplied, the function should generate an array of size $n \times m$. The output from the function is the array of random values in the range [0.0, 1.0].

3. Describe the algorithm

The pseudocode for function `random0` is:

```
function ran = random0 ( n, m )
Check for valid arguments
Set m ← n if not supplied
```

⁴ The notation [0.0,1.0) implies that the range of the random numbers is between 0.0 and 1.0, including the number 0.0, but excluding the number 1.0.

```

Create output array with "zeros" function
for ii = 1:number of rows
    for jj = 1:number of columns
        ISEED ← mod (8121 * ISEED + 28411, 134456 )
        ran(ii,jj) ← ISEED / 134456
    end
end

```

where the value of ISEED is placed in global memory so that it is saved between calls to the function. The pseudocode for function seed is trivial.

```

function seed ( new_seed )
new_seed ← round( new_seed )
ISEED ← abs( new_seed )

```

The round function is used in case the user fails to supply an integer, and the absolute value function is used in case the user supplies a negative seed. The user will not have to know in advance that only positive integers are legal seeds.

The variable ISEED will be placed in global memory so that it can be accessed by both functions.

4. Turn the algorithm into MATLAB statements

Function random0 follows.

```

function ran = random0(n,m)
%RANDOM0 Generate uniform random numbers in [0,1)
% Function RANDOM0 generates an array of uniform
% random numbers in the range [0,1). The usage
% is:
%
% random0(n)      -- Generate an n x n array
% random0(n,m)    -- Generate an n x m array

% Define variables:
% ii              -- Index variable
% ISEED           -- Random number seed (global)
% jj              -- Index variable
% m               -- Number of columns
% msg             -- Error message
% n               -- Number of rows
% ran             -- Output array

% Record of revisions:
% Date          Programmer          Description of change
% =====         ======          =====
% 12/16/98       S. J. Chapman     Original code

```

```
% Declare global values
global ISEED          % Seed for random number generator

% Check for a legal number of input arguments.
msg = nargchk(1,2,nargin);
error(msg);

% If the m argument is missing, set it to n.
if nargin < 2
    m = n;
end

% Initialize the output array
ran = zeros(n,m);

% Now calculate random values
for ii = 1:n
    for jj = 1:m
        ISEED = mod(8121*ISEED + 28411, 134456 );
        ran(ii,jj) = ISEED / 134456;
    end
end
```

Function seed follows.

```
function seed(new_seed)
%SEED Set new seed for function RANDOM0
% Function SEED sets a new seed for function
% RANDOM0. The new seed should be a positive
% integer.

% Define variables:
%   ISEED -- Random number seed (global)
%   new_seed -- New seed

% Record of revisions:
%   Date      Programmer      Description of change
%   ===       ======           =====
%   12/16/98   S. J. Chapman  Original code

% Declare global values
global ISEED          % Seed for random number generator

% Check for a legal number of input arguments.
msg = nargchk(1,1,nargin);
error(msg);

% Save seed
new_seed = round(new_seed);
ISEED = abs(new_seed);
```

5. Test the resulting MATLAB programs

If the numbers generated by these functions are truly uniformly distributed random numbers in the range $0 \leq \text{ran} < 1.0$, then the average of

many numbers should be close to 0.5 and the standard deviation of the numbers should be close to $\frac{1}{\sqrt{12}}$.

Furthermore, if the range between 0 and 1 is divided into a number of equal-size bins, the number of random values falling in each bin should be about the same. A **histogram** is a plot of the number of values falling in each bin. MATLAB function `hist` will create and plot a histogram from an input data set, so we will use it to verify the distribution of random number generated by `random0`.

To test the results of these functions, we will perform the following tests.

1. Call `seed` with `new_seed` set to 1024.
2. Call `random0(4)` to see that the results appear random.
3. Call `random0(4)` to verify that the results differ from call to call.
4. Call `seed` again with `new_seed` set to 1024.
5. Call `random0(4)` to see that the results are the same as in (2) above. This verifies that the seed is properly being reset.
6. Call `random0(2,3)` to verify that both input arguments are being used correctly.
7. Call `random0(1,20000)` and calculate the average and standard deviation of the resulting data set using MATLAB functions `mean` and `std`. Compare the results to 0.5 and $\frac{1}{\sqrt{12}}$.
8. Create a histogram of the data from (7) to see if approximately equal numbers of values fall in each bin.

We will perform these tests interactively, checking the results as we go.

```
>> seed(1024)
>> random0(4)
ans =
    0.0598    1.0000    0.0905    0.2060
    0.2620    0.6432    0.6325    0.8392
    0.6278    0.5463    0.7551    0.4554
    0.3177    0.9105    0.1289    0.6230
>> random0(4)
ans =
    0.2266    0.3858    0.5876    0.7880
    0.8415    0.9287    0.9855    0.1314
    0.0982    0.6585    0.0543    0.4256
    0.2387    0.7153    0.2606    0.8922
>> seed(1024)
>> random0(4)
ans =
    0.0598    1.0000    0.0905    0.2060
    0.2620    0.6432    0.6325    0.8392
    0.6278    0.5463    0.7551    0.4554
    0.3177    0.9105    0.1289    0.6230
```

```
> random0(2,3)
ans =
    0.2266    0.3858    0.5876
    0.7880    0.8415    0.9287
> arr = random0(1,20000);
> mean(arr)
ans =
    0.5020
> std(arr)
ans =
    0.2881
> hist(arr,10);
> title('\bfHistogram of the Output of random0');
> xlabel('Bin')
> ylabel('Count')
```

The results of these tests look reasonable, so the function appears to be working. The average of the data set was 0.5020, which is quite close to the theoretical value of 0.5000, and the standard deviation of the data set was 0.2881, which is quite close to the theoretical value of 0.2887. The histogram is shown in Figure 5.5, and the distribution of the random values is roughly even across all of the bins.

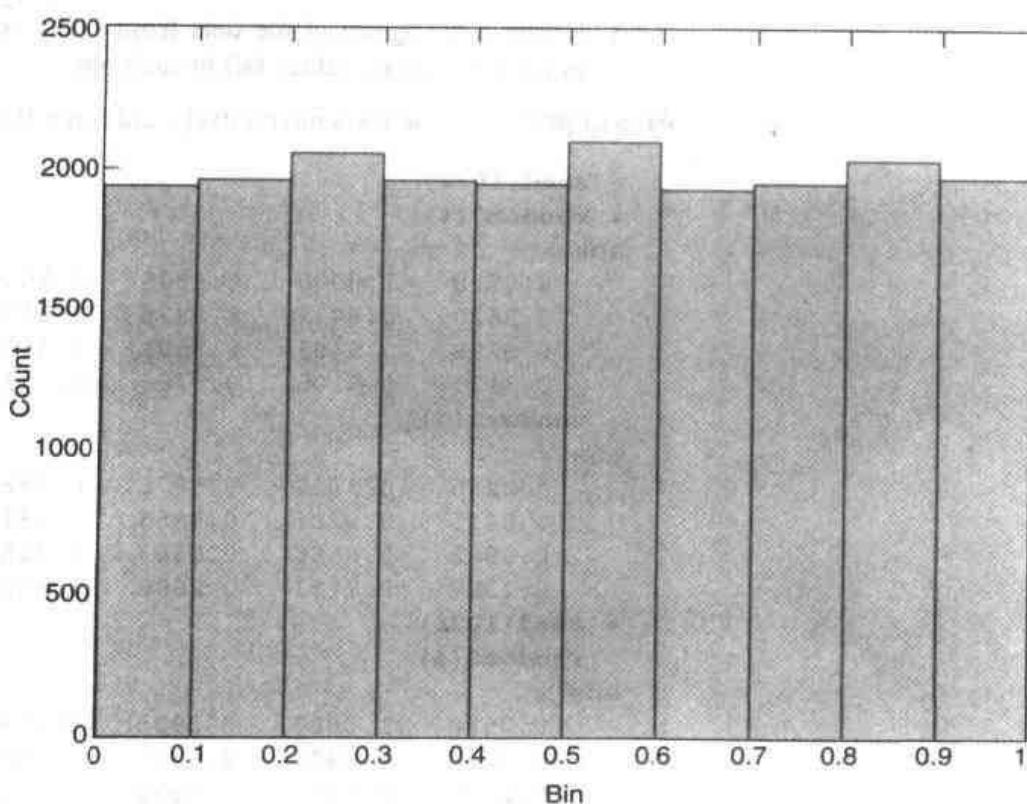


Figure 5.5 Histogram of the output of function random0.

MATLAB includes two intrinsic functions that generate random values from different distributions. They are

- `rand` Generates random values from a uniform distribution.
- `randn` Generates random values from a normal distribution.

Both of them are much faster and much more “random” than the simple function that we have created. If you really need random numbers in your programs, use one of these functions.

Functions `rand` and `randn` have the following calling sequences (the same for both functions):

- `rand`—Generates a single random value
- `rand(n)`—Generates an $n \times n$ array of random values
- `rand(n,m)`—Generates an $n \times m$ array of random values

5.5 Preserving Data Between Calls to a Function

When a function finishes executing, the special workspace created for that function is destroyed, so the contents of all local variables within the function will disappear. The next time that the function is called, a new workspace will be created, and all of the local variables will be returned to their default values. This behavior is usually desirable, since it ensures that MATLAB functions behave in a repeatable fashion every time they are called.

However, it is sometimes useful to preserve some local information within a function between calls to the function. For example, we might wish to create a counter to count the number of times that the function has been called. If such a counter were destroyed every time the function exited, the count would never exceed 1!

Beginning with Version 5.1, MATLAB includes a special mechanism to allow local variables to be preserved between calls to a function. **Persistent memory** is a special type of memory that can only be accessed from within the function, but is preserved unchanged between calls to the function.

A persistent variable is declared with the **`persistent`** statement. The form of a persistent statement is

```
persistent var1 var2 var3 ...
```

where `var1`, `var2`, `var3`, and so forth, are the variables to be placed in persistent memory.

Good Programming Practice

Use persistent memory to preserve the values of local variables within a function between calls to the function.

Example 5.5—Running Averages

It is sometimes desirable to calculate running statistics on a data set on-the-fly as the values are being entered. The built-in MATLAB functions `mean` and `std` could perform this function, but we would have to pass the entire data set to them for re-calculation after each new data value is entered. A better result can be achieved by writing a special function that keeps tracks of the appropriate running sums between calls, and only needs the latest value to calculate the current average and standard deviation.

The average or arithmetic mean of a set of numbers is defined as

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i \quad (5.8)$$

where x_i is sample i out of N samples. The standard deviation of a set of numbers is defined as

$$s = \sqrt{\frac{N \sum_{i=1}^N x_i^2 - \left(\sum_{i=1}^N x_i\right)^2}{N(N-1)}} \quad (5.9)$$

Standard deviation is a measure of the amount of scatter on the measurements; the greater the standard deviation, the more scattered are the points in the data set. If we can keep track of the number of values N , the sum of the values Σx , and the sum of the squares of the values Σx^2 , then we can calculate the average and standard deviation at any time from Equations (5.8) and (5.9).

Write a function to calculate the running average and standard deviation of a data set as it is being entered.

Solution

This function must be able to accept input values one at a time and keep running sums of N , Σx , and Σx^2 , which will be used to calculate the current average and standard deviation. It must store the running sums in persistent memory so that they are preserved between calls. Finally, there must be a mechanism to reset the running sums.

1. State the problem

Create a function to calculate the running average and standard deviation of a data set as new values are entered. The function must also include a feature to reset the running sums when desired.

2. Define the inputs and outputs

There are two types of inputs required by this function.

1. The character string 'reset' to reset running sums to zero.
2. The numeric values from the input data set presented one value per function call.

The outputs from this function are the mean and standard deviation of the data supplied to the function so far.

3. Design the algorithm

This function can be broken down into four major steps.

```

Check for a legal number of arguments
Check for a 'reset', and reset sums if present
Otherwise, add current value to running sums
Calculate and return running average and std dev
    if enough data is available. Return zeros if
        not enough data is available.

```

The detailed pseudocode for these steps is:

```

Check for a legal number of arguments
if x == 'reset'
    n ← 0
    sum_x ← 0
    sum_x2 ← 0
else
    n ← n + 1
    sum_x ← sum_x + x
    sum_x2 ← sum_x2 + x^2
end

% Calculate ave and sd
if n == 0
    ave ← 0
    std ← 0
elseif n == 1
    ave ← sum_x
    std ← 0
else
    ave ← sum_x / n
    std ← sqrt((n*sum_x2 - sum_x^2) / (n*(n-1)))
end

```

4. Turn the algorithm into MATLAB statements

The final MATLAB function follows.

```

function [ave, std] = runstats(x)
%RUNSTATS Generate running ave / std deviation
% Function RUNSTATS generates a running average
% and standard deviation of a data set. The
% values x must be passed to this function one
% at a time. A call to RUNSTATS with the argument
% 'reset' will reset the running sums.

% Define variables:
%   ave      -- Running average
%   msg      -- Error message

```

```

% n          -- Number of data values
% std        -- Running standard deviation
% sum_x     -- Running sum of data values
% sum_x2    -- Running sum of data values squared
% x          -- Input value

% Record of revisions:
%      Date      Programmer      Description of change
%      ===       ======      =====
% 12/16/98   S. J. Chapman   Original code

% Declare persistent values
persistent n          % Number of input values
persistent sum_x       % Running sum of values
persistent sum_x2      % Running sum of values squared

% Check for a legal number of input arguments.
msg = narginchk(1,1,nargin);
error(msg);

% If the argument is 'reset', reset the running sums.
if x == 'reset'
    n = 0;
    sum_x = 0;
    sum_x2 = 0;
else
    n = n + 1;
    sum_x = sum_x + x;
    sum_x2 = sum_x2 + x^2;
end

% Calculate ave and sd
if n == 0
    ave = 0;
    std = 0;
elseif n == 1
    ave = sum_x;
    std = 0;
else
    ave = sum_x / n;
    std = sqrt((n*sum_x2 - sum_x^2) / (n*(n-1)));
end

```

5. Test the program

To test this function, we must create a script file that resets `runstats`, reads input values, calls `runstats`, and displays the running statistics. An appropriate script file follows.

```

% Script file: test_runstats.m
%
% Purpose:
%   To read in an input data set and calculate the

```

```

% running statistics on the data set as the values
% are read in. The running stats will be written
% to the Command Window.

%
% Record of revisions:
%      Date          Programmer           Description of change
%      ===          ======           =====
%      12/16/98      S. J. Chapman       Original code
%
% Define variables:
%      array -- Input data array
%      ave   -- Running average
%      std   -- Running standard deviation
%      ii    -- Index variable
%      nvals -- Number of input values
%      std   -- Running standard deviation

% First reset running sums
[ave std] = runstats('reset');

% Prompt for the number of values in the data set
nvals = input('Enter number of values in data set: ');

% Get input values
for ii = 1:nvals

    % Prompt for next value
    string = ['Enter value ' int2str(ii) ': '];
    x = input(string);

    % Get running statistics
    [ave std] = runstats(x);

    % Display running statistics
    fprintf('Average = %.4f; Std dev = %.4f\n',ave, std);
end

```

To test this function, we will calculate running statistics by hand for a set of 5 numbers, and compare the hand calculations to the results from the program. If a data set is created with the following 5 input values

3.0, 2.0, 3.0, 4.0, 2.8

then the running statistics calculated by hand would be

Value	n	Σx	Σx^2	Average	Std Dev
3.0	1	3.0	9.0	3.00	0.000
2.0	2	5.0	13.0	2.50	0.707
3.0	3	8.0	22.0	2.67	0.577
4.0	4	12.0	38.0	3.00	0.816
2.8	5	14.8	45.84	2.96	0.713

The output of the test program for the same data set is

```
>> test_runstats
Enter number of values in data set: 5
Enter value 1: 3
Average = 3.0000; Std dev = 0.0000
Enter value 2: 2
Average = 2.5000; Std dev = 0.7071
Enter value 3: 3
Average = 2.6667; Std dev = 0.5774
Enter value 4: 4
Average = 3.0000; Std dev = 0.8165
Enter value 5: 2.8
Average = 2.9600; Std dev = 0.7127
```

so the results check to the accuracy shown in the hand calculations.

5.6 Function Functions

Function functions are functions with input arguments that include the names of other functions. The function names that are passed to the function function are normally used during the function's execution.

For example, MATLAB contains a function function called `fzero`. This function locates a zero of the function that is passed to it. For example, the statement `fzero('cos',[0 pi])` locates a zero of the function `cos` between 0 and π , and `fzero('exp(x)-2',[0 1])` locates a zero of the function `exp(x)-2` between 0 and 1. When these statements are executed, the result is

```
>> fzero('cos',[0 pi])
ans =
    1.5708
>> fzero('exp(x)-2',[0 1])
ans =
    0.6931
```

The keys to the operation of function functions are two special MATLAB functions, `eval` and `feval`. Function `eval` evaluates a character string as though it had been typed in the Command Window, and function `feval` evaluates a named function at a specific input value.

Function `eval` evaluates a character string as though it has been typed in the Command Window. This function gives MATLAB functions a chance to construct executable statements during execution. The form of the `eval` function is

```
eval(string)
```

For example, the statement `x = eval('sin(pi/4)')` produces the result

```
>> x = eval('sin(pi/4)')
x =
    0.7071
```

Table 5.1 Common MATLAB Function Functions

Function Name	Description
fminbnd	Minimize a function of one variable.
fzero	Find a zero of a function of one variable.
quad	Numerically integrate a function.
ezplot	Easy to use function plotter.
fplot	Plot a function by name.

An example where a character string is constructed and evaluated using the eval function follows:

```
x = 1;
str = ['exp(' num2str(x) ') - 1'];
res = eval(str);
```

In this case, str contains the character string 'exp(1) - 1', which eval evaluates to get the result 1.7183.

Function feval evaluates a *named function* defined by an M-file at a specified input value. The general form of the feval function is

```
feval(fun,value),
```

For example, the statement x = feval('sin',pi/4) produces the result

```
>> x = feval('sin',pi/4)
x =
    0.7071
```

Some of the more common MATLAB function functions are listed in Table 5.1. Type help *function_name* to learn how to use one of these functions.

Example 5.6—Creating a Function Function

Create a function function that will plot any MATLAB function of a single variable between specified starting and ending values.

Solution

This function has two input arguments, the first one containing the name of the function to plot and the second one containing a 2-element vector with the range of values to plot.

1. State the problem

Create a function to plot any MATLAB function of a single variable between two user-specified limits.

2. Define the inputs and outputs

There are two inputs required by this function.

1. A character string containing the name of a function.

2. A 2-element vector containing the first and last values to plot.

The output from this function is a plot of the function specified in the first input argument.

3. Design the algorithm

This function can be broken down into four major steps.

Check for a legal number of arguments

Check that the second argument has two elements

Calculate the value of the function between the start and stop points

Plot and label the function

The detailed pseudocode for the evaluation and plotting steps is

```
n_steps ← 100
step_size ← (xlim(2) - xlim(1)) / n_steps
x ← xlim(1):step_size:xlim(2)
y ← feval(fun,x)
plot(x,y)
title(['\bf Plot of function ' fun '(x)'])
xlabel('\bf x')
ylabel(['\bf ' fun '(x)'])
```

4. Turn the algorithm into MATLAB statements

The final MATLAB function follows.

```
function quickplot(fun,xlim)
%QUICKPLOT Generate quick plot of a function
% Function QUICKPLOT generates a quick plot
% of a function contained in an external M-file,
% between user-specified x limits.

% Define variables:
%   fun      -- Function to plot
%   msg      -- Error message
%   n_steps  -- Number of steps to plot
%   step_size -- Step size
%   x        -- X-values to plot
%   y        -- Y-values to plot
%   xlim    -- Plot x limits

% Record of revisions:
%   Date      Programmer      Description of change
%   =====    ======      =====
%   12/17/98  S. J. Chapman  Original code

% Check for a legal number of input arguments.
```

```

msg = nargchk(2,2,nargin);
error(msg);

% Check the second argument to see if it has two
% elements. Note that this double test allows the
% argument to be either a row or a column vector.
if ( size(xlim,1) == 1 & size(xlim,2) == 2 ) | ...
( size(xlim,1) == 2 & size(xlim,2) == 1 )

    % Ok--continue processing.
    n_steps = 100;
    step_size = (xlim(2) - xlim(1)) / n_steps;
    x = xlim(1):step_size:xlim(2);
    y = feval(fun,x);
    plot(x,y);
    title(['\bf Plot of function ' fun '(x)'']);
    xlabel('\bf x');
    ylabel(['\bf ' fun '(x)'']);
else
    % Else wrong number of elements in xlim.
    error('Incorrect number of elements in xlim.');
end

```

5. Test the program

To test this function, we must call it with correct and incorrect input arguments, verifying that it handles both correct inputs and errors properly. The results follow.

```

» quickplot('sin')
??? Error using ==> quickplot
Not enough input arguments.

» quickplot('sin',[-2*pi 2*pi],3)
??? Error using ==> quickplot
Too many input arguments.

» quickplot('sin',-2*pi)
??? Error using ==> quickplot
Incorrect number of elements in xlim.

» quickplot('sin',[-2*pi 2*pi])

```

The last call was correct, and it produced the plot shown in Figure 5.6.

5.7 Subfunctions and Private Functions

It is possible to place more than one function in a single file. If more than one function is present in a file, the top function is a normal function, while the ones below it are **subfunctions** or **internal functions**. Subfunctions look just like

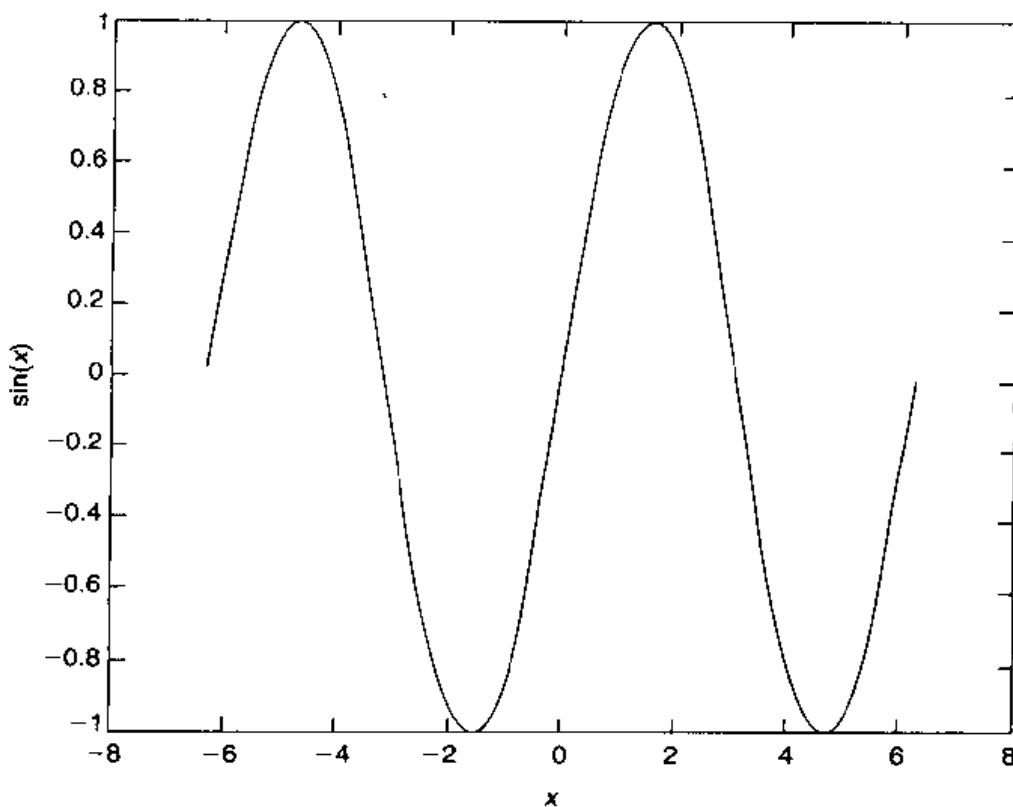


Figure 5.6 Plot of $\sin x$ versus x generated by function quickplot.

ordinary functions, but they are only accessible to the other functions within the same file.

The following example defines a function mystats and two subfunctions mean and median. Function mystats can be called by any other MATLAB function, but functions mean and median can only be called by other functions in the same file.

```

function [avg, med] = mystats(u)
%MYSTATS Find mean and median with internal functions.
% Function MYSTATS calculates the average and median
% of a data set using subfunctions.

n = length(u);
avg = mean(u,n);
med = median(u,n);

function a = mean(v,n)
% Subfunction to calculate average.
a = sum(v)/n;

function m = median(v,n)
% Subfunction to calculate median.

```

```
w = sort(v);
if rem(n,2) == 1
    m = w((n+1)/2);
else
    m = (w(n/2)+w(n/2+1))/2;
end
```

Private functions are functions that reside in subdirectories with the special name `private`. They are only visible to functions in the parent directory. For example, assume the directory `testing` is on the MATLAB search path. A subdirectory of `testing` called `private` can contain functions that only the functions in `testing` can call. Because private functions are invisible outside of the parent directory, they can use the same names as functions in other directories. This is useful if you want to create your own version of a particular function while retaining the original in another directory. Because MATLAB looks for private functions before standard M-file functions, it will find a private function named `test.m` before a non-private M-file named `test.m`.

You can create your own private directories simply by creating a subdirectory called `private` under the directory containing your functions. Do not place these private directories on your search path.

When a function is called from within an M-file, MATLAB first checks the file to see if the function is a subfunction. If not, it checks for a private function with that name. If it is not a private function, MATLAB checks the standard search path for the function.

If you have special-purpose MATLAB functions that should only be used by other functions and never be called directly by the user, consider hiding them as subfunctions or private functions. Hiding the functions will prevent their accidental use, and will also prevent conflicts with other public functions of the same name.

Good Programming Practice

Use subfunctions and private functions to hide special-purpose MATLAB functions that should only be called by other functions. Hiding the functions will prevent their accidental use, and will also prevent conflicts with other public functions of the same name.

5.8 Summary

In Chapter 5, we presented an introduction to user-defined functions. Functions are special types of M-files that receive data through input arguments and return results through output arguments. Each function has its own independent workspace.

Arguments are passed to functions using a pass-by-value scheme, meaning that MATLAB copies each argument and passes the copy to the function. This copying is important, because the function can freely modify its input arguments without affecting the actual arguments in the calling program.

MATLAB functions can support varying numbers of input and output arguments. Function `nargin` reports the number of actual input arguments used in a function call, and function `nargout` reports the number of actual output arguments used in a function call.

Data can also be shared between MATLAB functions by placing the data in global memory. Global variables are declared using the `global` statement. Global variables can be shared by all functions that declare them. By convention, global variable names are written in all capital letters.

Internal data within a function can be preserved between calls to that function by placing the data in persistent memory. Persistent variables are declared using the `persistent` statement.

Function functions are MATLAB functions with input arguments that include the names of other functions. The functions whose names are passed to the `function` function are normally used during that function's execution. Examples are some root-solving and plotting functions.

Subfunctions are additional functions placed within a single file. Subfunctions are only accessible from other functions within the same file. Private functions are functions placed in a special subdirectory called `private`. They are only accessible to functions in the parent directory. Both subfunctions and private functions can be used to restrict access to MATLAB functions.

5.8.1 Summary of Good Programming Practice

The following guidelines should be adhered to when working with MATLAB functions.

1. Break large program tasks into smaller, more understandable functions whenever possible.
2. Declare global variables in all capital letters to make them easy to distinguish from local variables.
3. Declare global variables immediately after the initial comments and before the first executable statement in each function that uses them.
4. You can use global memory to pass large amounts of data among functions within a program.
5. Use persistent memory to preserve the values of local variables within a function between calls to the function.
6. Use subfunctions and private functions to hide special-purpose MATLAB functions that should only be called by other functions. Hiding the functions will prevent their accidental use and will also prevent conflicts with other public functions of the same name.

5.8.2 MATLAB Summary

The following summary lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one.

Commands and Functions

<code>error</code>	Displays error message and aborts the function producing the error. This function is used if the argument errors are fatal.
<code>eval</code>	Evaluates a character string as though it had been typed in the Command Window.
<code>ezplot</code>	Easy-to-use function plotter.
<code>feval</code>	Calculates the value of a function $f(x)$ defined by an M-file at a specific x .
<code>fmin</code>	Minimize a function of one variable.
<code>fplot</code>	Plot a function by name.
<code>fzero</code>	Find a zero of a function of one variable.
<code>global</code>	Declares global variables.
<code>hist</code>	Calculate and plot a histogram of a data set.
<code>inputname</code>	Returns the actual name of the variable that corresponds to a particular argument number.
<code>nargchk</code>	Returns a standard error message if a function is called with too few or too many arguments.
<code>nargin</code>	Returns the number of actual input arguments that were used to call the function.
<code>nargout</code>	Returns the number of actual output arguments that were used to call the function.
<code>persistent</code>	Declares persistent variables.
<code>quad</code>	Numerically integrates a function.
<code>rand</code>	Generates random values from a uniform distribution.
<code>randn</code>	Generates random values from a normal distribution.
<code>return</code>	Stops executing a function and returns to caller.
<code>warning</code>	Displays a warning message and continues function execution. This function is used if the argument errors are not fatal, and execution can continue.

5.9 Exercises

- 5.1 What is the difference between a script file and a function?
- 5.2 When a function is called, how is data passed from the caller to the function, and how are the results of the function returned to the caller?
- 5.3 What are the advantages and disadvantages of the pass-by-value scheme used in MATLAB?

- 5.4** Modify the selection sort function developed in this chapter so that it accepts a second optional argument, which may be either 'up' or 'down'. If the argument is 'up', sort the data in ascending order. If the argument is 'down', sort the data in descending order. If the argument is missing, the default case is to sort the data in ascending order. (Be sure to handle the case of invalid arguments, and be sure to include the proper help information in your function.)
- 5.5** Write a function that uses function `random0` to generate a random value in the range [-1.0,1.0). Make `random0` a subfunction of your new function.
- 5.6** Write a function that uses function `random0` to generate a random value in the range [low, high), where low and high are passed as calling arguments. Make `random0` a private function called by your new function.
- 5.7** **Dice Simulation.** It is often useful to be able to simulate the throw of a fair die. Write a MATLAB function `dice` that simulates the throw of a fair die by returning some random integer between 1 and 6 every time that it is called. (*Hint:* Call `random0` to generate a random number. Divide the possible values out of `random0` into six equal intervals, and return the number of the interval that a given random value falls into.)
- 5.8** **Road Traffic Density.** Function `random0` produces a number with a *uniform* probability distribution in the range [0.0, 1.0). This function is suitable for simulating random events if each outcome has an equal probability of occurring. However, in many events the probability of occurrence is *not* equal for every event, and a uniform probability distribution is not suitable for simulating such events.

For example, when traffic engineers studied the number of cars passing a given location in a time interval of length t , they discovered that the probability of k cars passing during the interval is given by the equation

$$P(k, t) = e^{-\lambda t} \frac{(\lambda t)^k}{k!} \text{ for } t \geq 0, \lambda > 0, \text{ and } k = 0, 1, 2, \dots \quad (5.10)$$

This probability distribution is known as the *Poisson distribution*; it occurs in many applications in science and engineering. For example, the number of calls k to a telephone switchboard in time interval t , the number of bacteria k in a specified volume t of liquid, and the number of failures k of a complicated system in time interval t all have Poisson distributions.

Write a function to evaluate the Poisson distribution for any k , t , and λ . Test your function by calculating the probability of 0, 1, 2, ..., 5 cars passing a particular point on a highway in 1 minute, given that λ is 1.6 per minute for that highway. Plot the Poisson distribution for $t = 1$ and $\lambda = 1.6$.

- 5.9** Write three MATLAB functions to calculate the hyperbolic sine, cosine, and tangent functions.

$$\sinh(x) = \frac{e^x - e^{-x}}{2}, \quad \cosh(x) = \frac{e^x + e^{-x}}{2}, \quad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Use your functions to plot the shapes of the hyperbolic sine, cosine, and tangent functions.

- 5.10 Cross Product.** Write a function to calculate the cross product of two vectors \mathbf{V}_1 and \mathbf{V}_2 .

$$\mathbf{V}_1 \times \mathbf{V}_2 = (V_{y1}V_{z2} - V_{y2}V_{z1})\mathbf{i} + (V_{z1}V_{x2} - V_{z2}V_{x1})\mathbf{j} + (V_{x1}V_{y2} - V_{x2}V_{y1})\mathbf{k}$$

where $\mathbf{V}_1 = V_{x1}\mathbf{i} + V_{y1}\mathbf{j} + V_{z1}\mathbf{k}$ and $\mathbf{V}_2 = V_{x2}\mathbf{i} + V_{y2}\mathbf{j} + V_{z2}\mathbf{k}$. Note that this function will return a real array as its result. Use the function to calculate the cross product of the two vectors $\mathbf{V}_1 = [-2, 4, 0.5]$ and $\mathbf{V}_2 = [0.5, 3, 2]$.

- 5.11 Sort with Carry.** It is often useful to sort an array `arr1` into ascending order, while simultaneously carrying along a second array `arr2`. In such a sort, each time an element of array `arr1` is exchanged with another element of `arr1`, the corresponding elements of array `arr2` are also swapped. When the sort is over, the elements of array `arr1` are in ascending order, whereas the elements of array `arr2` that were associated with particular elements of array `arr1` are still associated with them. For example, suppose we have the following two arrays.

<u>Element</u>	<u>arr1</u>	<u>arr2</u>
1.	6.	1.
2.	1.	0.
3.	2.	10.

After sorting array `arr1` while carrying along array `arr2`, the contents of the two arrays will be

<u>Element</u>	<u>arr1</u>	<u>arr2</u>
1.	1.	0.
2.	2.	10.
3.	6.	1.

Write a function to sort one real array into ascending order while carrying along a second one. Test the function with the following two 9-element arrays.

```
a = [-1, 11, -6, 17, -23, 0, 5, 1, -1];
b = [-31, 101, 36, -17, 0, 10, -8, -1, -1];
```

- 5.12** Use the Help Browser to look up information about the standard MATLAB function `sortrows` and compare the performance of `sortrows` with the sort-with-carry function created in the previous exercise. To do this, create two copies of a 1000×2 element array containing random values, and sort column 1 of each array while carrying along column 2 using both functions. Determine the execution times of each sort function using `tic` and `toc`. How does the speed of your function compare with the speed of the standard function `sortrows`?

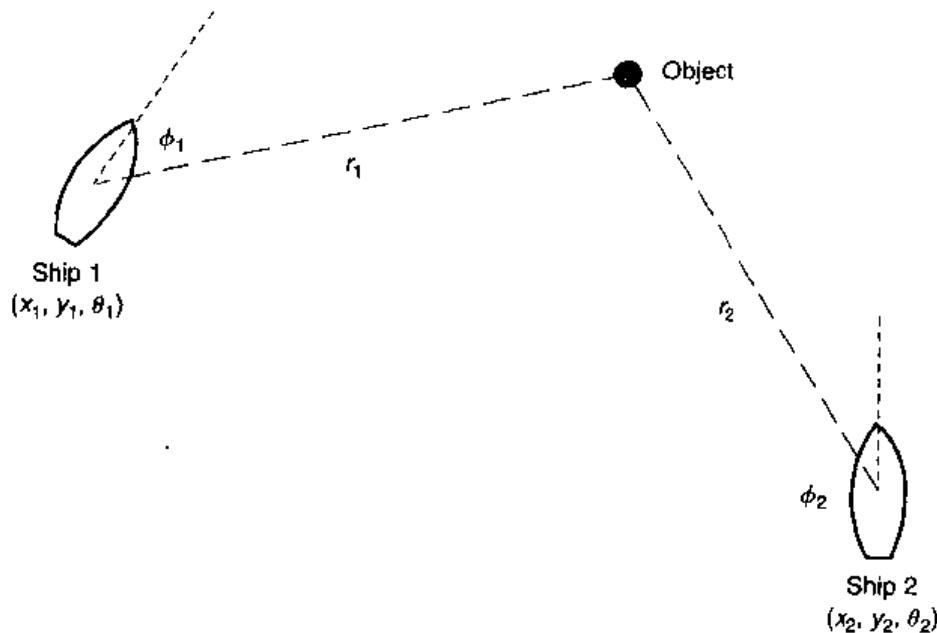


Figure 5.7 Two ships at positions (x_1, y_1) and (x_2, y_2) , respectively. Ship 1 is traveling at heading θ_1 , and Ship 2 is traveling at heading θ_2 .

5.13 Figure 5.7 shows two ships steaming on the ocean. Ship 1 is at position (x_1, y_1) and steaming on heading θ_1 . Ship 2 is at position (x_2, y_2) and steaming on heading θ_2 . Suppose that Ship 1 makes radar contact with an object at range r_1 and bearing ϕ_1 . Write a MATLAB function that will calculate the range r_2 and bearing ϕ_2 at which Ship 2 should see the object.

5.14 Minima and Maxima of a Function. Write a function that attempts to locate the maximum and minimum values of an arbitrary function $f(x)$ over a certain range. The function being evaluated should be passed to the function as a calling argument. The function should have the following input arguments.

```

first_value -- The first value of x to search
last_value  -- The last value of x to search
num_steps    -- The number of steps to include in the search
func         -- The name of the function to search
  
```

The function should have the following output arguments.

```

xmin        -- The value of x at which the minimum was found
min_value   -- The minimum value of f(x) found
xmax        -- The value of x at which the maximum was found
max_value   -- The maximum value f(x) found
  
```

Be sure to check that there are a valid number of input arguments and that the MATLAB help and lookfor commands are properly supported.

5.15 Write a test program for the function generated in the previous exercise. The test program should pass to the function function the user-defined

function $f(x) = x^3 - 5x^2 + 5x - 2$, and search for the minimum and maximum in 200 steps over the range $-1 \leq x \leq 3$. It should print out the resulting minimum and maximum values.

- 5.16 Derivative of a Function.** The *derivative* of a continuous function $f(x)$ is defined by the equation

$$\frac{d}{dx} f(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \quad (5.11)$$

In a sampled function, this definition becomes

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{\Delta x} \quad (5.12)$$

where $\Delta x = x_{i+1} - x_i$. Assume that a vector `vect` contains `nsamp` samples of a function taken at a spacing of `dx` per sample. Write a function that will calculate the derivative of this vector from Equation (5.12). The function should check to make sure that `dx` is greater than zero to prevent divide-by-zero errors in the function.

To check your function, you should generate a data set whose derivative is known, and compare the result of the function with the known correct answer. A good choice for a test function is $\sin x$. From elementary calculus, we know that $\frac{d}{dx} (\sin x) = \cos x$. Generate an input vector containing 100 values of the function $\sin x$ starting at $x = 0$ and using a step size Δx of 0.05. Take the derivative of the vector with your function, and then compare the resulting answers to the known correct answer. How close did your function come to calculating the correct value for the derivative?

- 5.17 Derivative in the Presence of Noise.** We will now explore the effects of input noise on the quality of a numerical derivative. First, generate an input vector containing 100 values of the function $\sin x$ starting at $x = 0$, using a step size Δx of 0.05, just as you did in the previous problem. Next, use function `random0` to generate a small amount of random noise with a maximum amplitude of ± 0.02 , and add that random noise to the samples in your input vector (see Figure 5.8). Note that the peak amplitude of the noise is only 2% of the peak amplitude of your signal, since the maximum value of $\sin x$ is 1. Now take the derivative of the function using the derivative function that you developed in the last problem. How close to the theoretical value of the derivative did you come?

- 5.18 Linear Least-Squares Fit.** Develop a function that will calculate slope m and intercept b of the least-squares line that best fits an input data set. The input data points (x, y) will be passed to the function in two input arrays, `x` and `y`. (The equations describing the slope and intercept of the least-squares line are given in Example 4.7 in the previous chapter.) Test your function using a test program and the 20-point input data set given in Table 5.2.

- 5.19 Correlation Coefficient of Least-Squares Fit.** Develop a function that will calculate both the slope m and intercept b of the least-squares line that

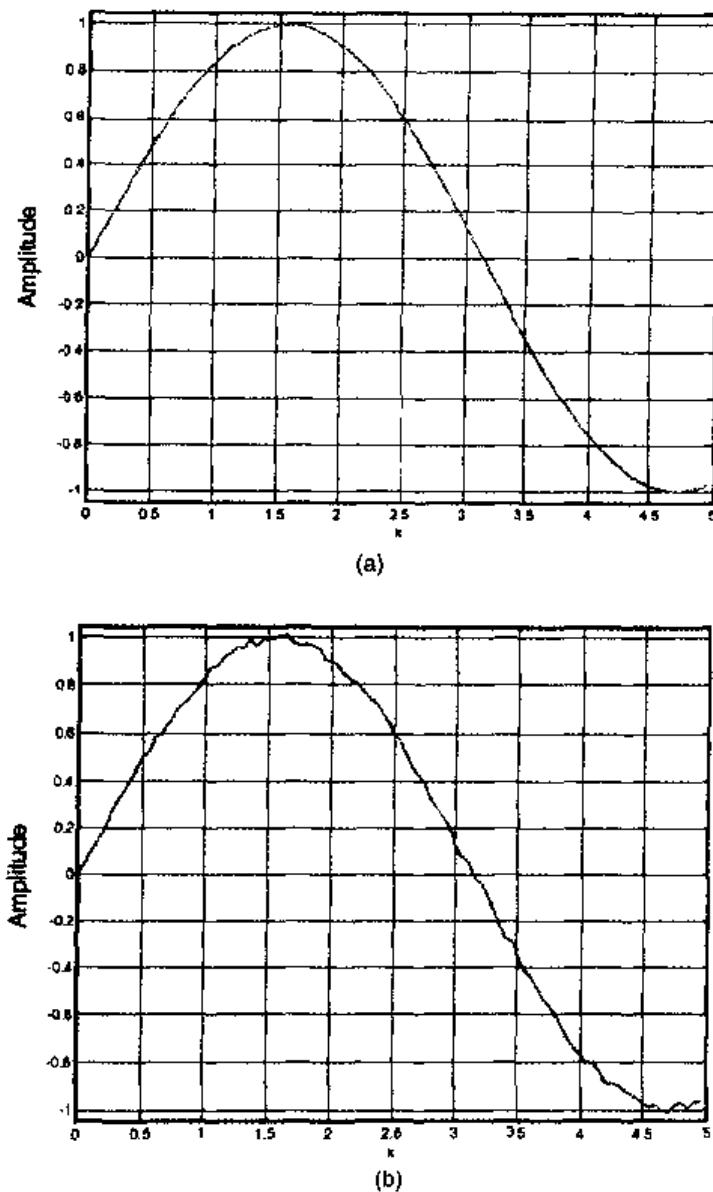


Figure 5.8 (a) A plot of $\sin x$ as a function of x with no noise added to the data. (b) A plot of $\sin x$ as a function of x with a 2% peak amplitude uniform random noise added to the data.

best fits an input data set, and also the correlation coefficient of the fit. The input data points (x, y) will be passed to the function in two input arrays, x and y . The equations describing the slope and intercept of the least-squares line are given in Example 4.7, and the equation for the correlation coefficient is

$$r = \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{[(n\sum x^2) - (\sum x)^2][(n\sum y^2) - (\sum y)^2]}} \quad (5.13)$$

Table 5.2 Sample Data to Test the Least-Squares Fit Function in Ex. 5.18

No.	x	y	No.	x	y
1	-4.91	-8.18	11	-0.94	0.21
2	-3.84	-7.49	12	0.59	1.73
3	-2.41	-7.11	13	0.69	3.96
4	-2.62	-6.15	14	3.04	4.26
5	-3.78	-5.62	15	1.01	5.75
6	-0.52	-3.30	16	3.60	6.67
7	-1.83	-2.05	17	4.53	7.70
8	-2.01	-2.83	18	5.13	7.31
9	0.28	-1.16	19	4.43	9.05
10	1.08	0.52	20	4.12	10.95

where

$\sum x$ is the sum of the x values.

$\sum y$ is the sum of the y values.

$\sum x^2$ is the sum of the squares of the x values.

$\sum y^2$ is the sum of the squares of the y values.

$\sum xy$ is the sum of the products of the corresponding x and y values.

n is the number of points included in the fit.

Test your function using a test driver program and the 20-point input data set given in the previous problem.

- 5.20 The Birthday Problem.** The birthday problem is: if there is a group of n people in a room, what is the probability that two or more of them have the same birthday? It is possible to determine the answer to this question by simulation. Write a function that calculates the probability that two or more of n people will have the same birthday, where n is a calling argument. (*Hint:* To do this, the function should create an array of size n and generate n birthdays in the range 1 to 365 randomly. It should then check to see if any of the n birthdays are identical. The function should perform this experiment at least 5000 times and calculate the fraction of those times in which two or more people had the same birthday.) Write a test program that calculates and prints out the probability that 2 or more of n people will have the same birthday for $n = 2, 3, \dots, 40$.
- 5.21** Use function `random0` to generate a set of three arrays of random numbers. The three arrays should be 100, 1000, and 2000 elements long. Then, use functions `tic` and `toc` to determine the time that it takes function `ssort` to sort each array. How does the elapsed time to sort increase as a

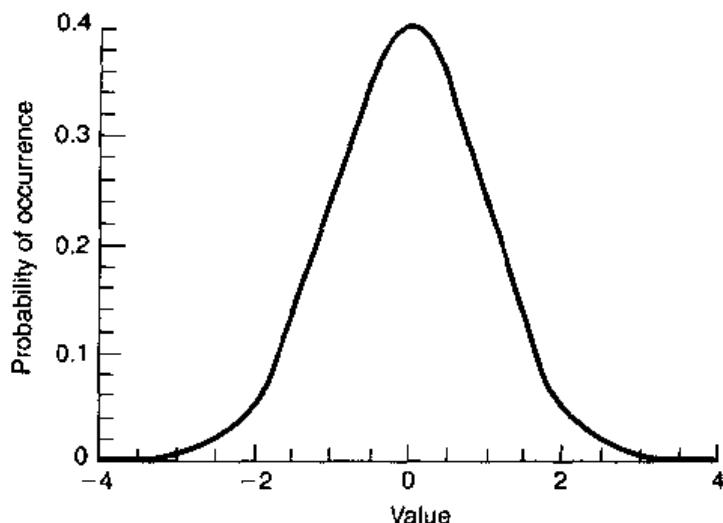


Figure 5.9 A normal probability distribution.

function of the number of elements being sorted? (*Hint:* On a fast computer, you will need to sort each array many times and calculate the average sorting time in order to overcome the quantization error of the system clock.)

5.22 Gaussian (Normal) Distribution. Function `random0` returns a uniformly distributed random variable in the range $[0, 1]$, which means that there is an equal probability of any given number in the range occurring on a given call to the function. Another type of random distribution is the Gaussian distribution, in which the random value takes on the classic bell-shaped curve shown in Figure 5.9. A Gaussian distribution with an average of 0.0 and a standard deviation of 1.0 is called a *standardized normal distribution*, and the probability of any given value occurring in the standardized normal distribution is given by the equation

$$p(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \quad (5.14)$$

It is possible to generate a random variable with a standardized normal distribution starting from a random variable with a uniform distribution in the range $[-1, 1]$ as follows.

1. Select two uniform random variables x_1 and x_2 from the range $[-1, 1]$ such that $x_1^2 + x_2^2 < 1$. To do this, generate two uniform random variables in the range $[-1, 1]$, and see if the sum of their squares happens to be less than 1.0. If so, use them. If not, try again.
2. Then each of the values y_1 and y_2 in the equations that follow will be a normally distributed random variable.

$$y_1 = \sqrt{\frac{-2 \ln r}{r}} x_1 \quad (5.15)$$

$$y_2 = \sqrt{\frac{-2 \ln r}{r}} x_2 \quad (5.16)$$

where

$$r = x_1^2 + x_2^2 \quad (5.17)$$

a \ln is the natural logarithm (log to the base e). Write a function that returns a normally distributed random value each time that it is called. Test your function by getting 1000 random values, calculating the standard deviation, and plotting a histogram of the distribution. How close to 1.0 was the standard deviation?

- 5.23 Gravitational Force.** The gravitational force F between two bodies of masses m_1 and m_2 is given by the equation

$$F = \frac{G m_1 m_2}{r^2} \quad (5.18)$$

where G is the gravitational constant (6.672×10^{-11} N m 2 / kg 2), m_1 and m_2 are the masses of the bodies in kilograms, and r is the distance between the two bodies. Write a function to calculate the gravitational force between two bodies given their masses and the distance between them. Test your function by determining the force on an 800-kg satellite in orbit 38,000 km above the Earth. (The mass of the Earth is 5.98×10^{24} kg.)

- 5.24 Rayleigh Distribution.** The Rayleigh distribution is another random number distribution that appears in many practical problems. A Rayleigh-distributed random value can be created by taking the square root of the sum of the squares of two normally distributed random values. In other words, to generate a Rayleigh-distributed random value r , get two normally distributed random values (n_1 and n_2), and perform the following calculation.

$$r = \sqrt{n_1^2 + n_2^2} \quad (5.19)$$

- Create a function `rayleigh(n,m)` that returns an $n \times m$ array of Rayleigh-distributed random numbers. If only one argument is supplied [`rayleigh(n)`], the function should return an $n \times n$ array of Rayleigh-distributed random numbers. Be sure to design your function with input argument checking and with proper documentation for the MATLAB help system.
- Test your function by creating an array of 20,000 Rayleigh-distributed random values and plotting a histogram of the distribution. What does the distribution look like?
- Determine the mean and standard deviation of the Rayleigh distribution.

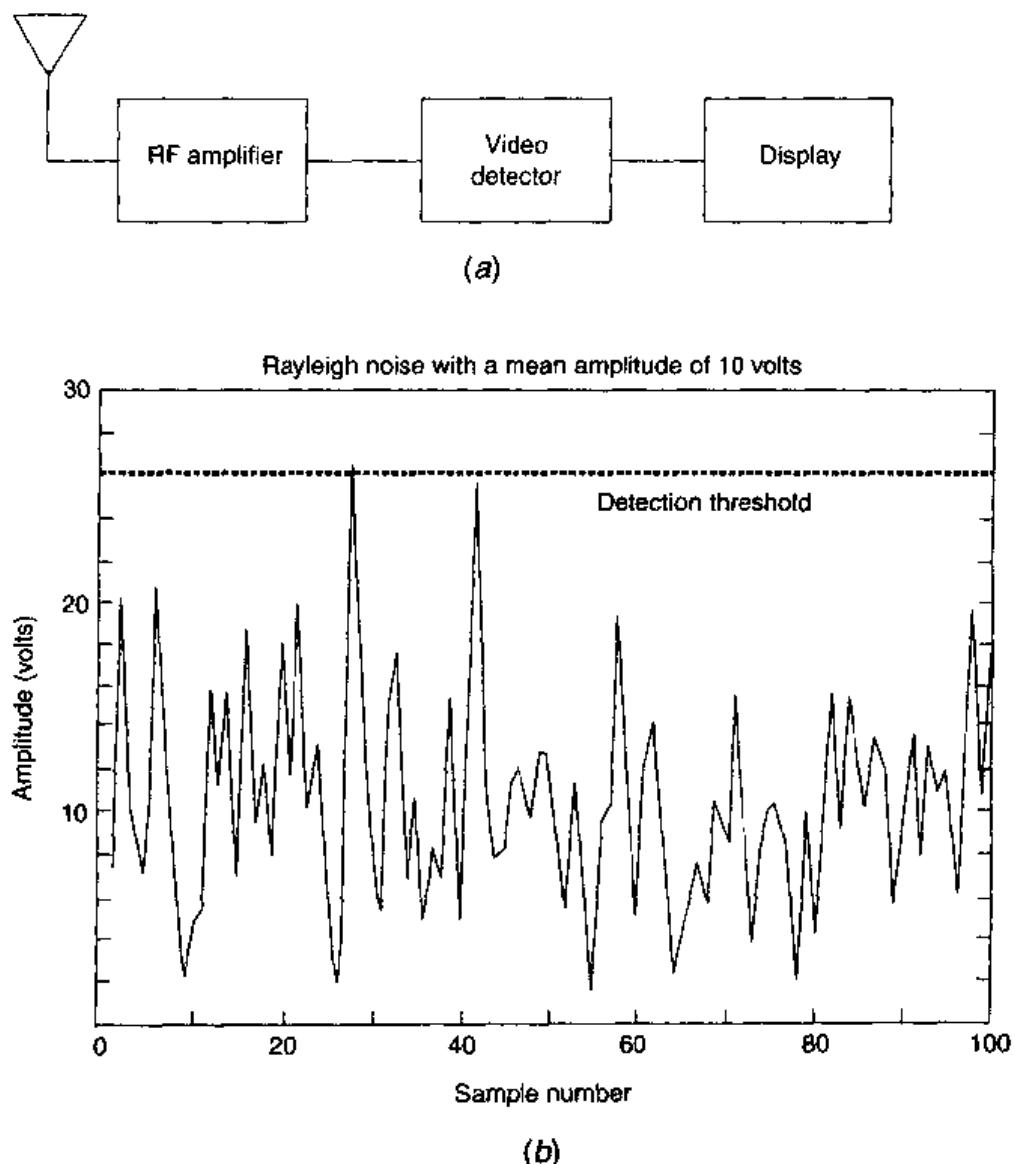


Figure 5.10 (a) A typical radar receiver. (b) Thermal noise with a mean of 10 volts output from the detector. The noise sometimes crosses the detection threshold.
(c) Probability distribution of the noise out of the detector.

5.25 Constant False Alarm Rate (CFAR). A simplified radar receiver chain is shown in Figure 5.10a. When a signal is received in this receiver, it contains both the desired information (returns from targets) and thermal noise. After the detection step in the receiver, we would like to be able to pick out received target returns from the thermal noise background. We can do this by setting a threshold level, and then declaring that we see a target whenever the signal crosses that threshold. Unfortunately, it is

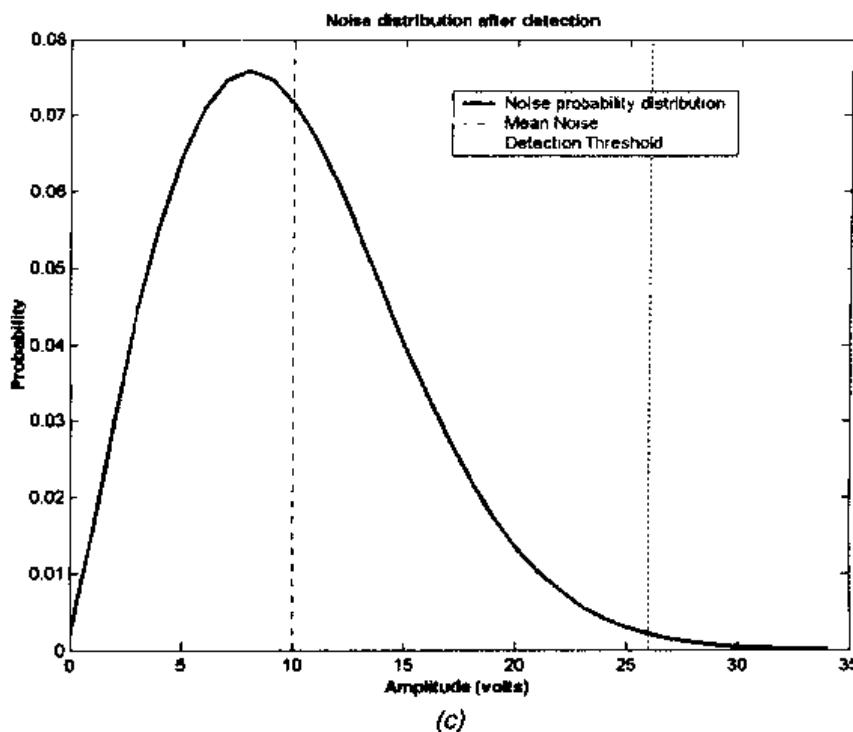


Figure 5.10 (Continued)

occasionally possible for the receiver noise to cross the detection threshold even if no target is present. If that happens, we will declare the noise spike to be a target, creating a *false alarm*. The detection threshold needs to be set as low as possible so that we can detect weak targets, but it must not be set too low, or we get many false alarms.

After video detection, the thermal noise in the receiver has a Rayleigh distribution. Figure 5.10b shows 100 samples of a Rayleigh-distributed noise with a mean amplitude of 10 volts. Note that there would be one false alarm even if the detection threshold were as high as 26! The probability distribution of these noise samples is shown in Figure 5.10c.

Detection thresholds are usually calculated as a multiple of the mean noise level, so that if the noise level changes, the detection threshold will change with it to keep false alarms under control. This is known as *constant false alarm rate* (CFAR) detection. A detection threshold is typically quoted in decibels. The relationship between the threshold in dB and the threshold in volts is

$$\text{Threshold (volts)} = \text{Mean Noise Level (volts)} \times 10^{\frac{\text{dB}}{20}} \quad (5.20)$$

or

$$\text{dB} = 20 \log_{10} \left(\frac{\text{Threshold (volts)}}{\text{Mean Noise Level (volts)}} \right) \quad (5.21)$$

The false alarm rate for a given detection threshold is calculated as

$$P_{fa} = \frac{\text{Number of False Alarms}}{\text{Total Number of Samples}} \quad (5.22)$$

Write a program that generates 1,000,000 random noise samples with a mean amplitude of 10 volts and a Rayleigh noise distribution. Determine the false alarm rates when the detection threshold is set to 5, 6, 7, 8, 9, 10, 11, 12, and 13 dB above the mean noise level. At what level should the threshold be set to achieve a false alarm rate of 10^{-4} ?

- 5.26 Probability of Detection (P_d) versus Probability of False Alarm (P_{fa}).** The signal strength returned by a radar target usually fluctuates over time. The target will be detected if its signal strength exceeds the detection threshold for any given look. The probability that the target will be detected can be calculated as

$$P_d = \frac{\text{Number of Target Detections}}{\text{Total Number of Looks}} \quad (5.23)$$

Suppose that a specific radar looks repeatedly in a given direction. On each look, the range between 10 km and 20 km is divided into 100 independent range samples (called *range gates*). One of these range gates contains a target whose amplitude has a normal distribution with a mean amplitude of 7 volts and a standard deviation of 1 volt. All 100 of the range gates contain system noise with a mean amplitude of 2 volts and a Rayleigh distribution. Determine both the probability of target detection P_d and the probability of a false alarm P_{fa} on any given look for detection thresholds of 8.0, 8.5, 9.0, 9.5, 10.0, 10.5, 11.0, 11.5, and 12.0 dB. What threshold would you use for detection in this radar? (*Hint:* Perform the experiment many times for each threshold and average the results to determine valid probabilities.)

Complex Data, Character Data, and Additional Plot Types

In Chapter 2, we learned about some of the fundamental data types of MATLAB: `double` and `char`. MATLAB includes a number of additional data types, and we will learn about one of them in this chapter. The additional data type that we will discuss is MATLAB support for complex data. We will also learn more about how to use the `char` data type, and how to extend MATLAB arrays to more than two dimensions.

The chapter concludes with a discussion of additional types of plots available in MATLAB.

6.1 Complex Data

Complex numbers are numbers with both a real and an imaginary component. Complex numbers occur in many problems in science and engineering. For example, complex numbers are used in electrical engineering to represent alternating current voltages, currents, and impedances. The differential equations that describe the behavior of most electrical and mechanical systems also give rise to complex numbers. Because they are so ubiquitous, it is impossible to work as an engineer without a good understanding of the use and manipulation of complex numbers.

A complex number has the general form

$$c = a + bi \quad (6.1)$$

where c is a complex number, a and b are both real numbers, and i is $\sqrt{-1}$. The number a is called the *real part* and b is called the *imaginary part* of the complex number c . Since a complex number has two components, it can be plotted as a point

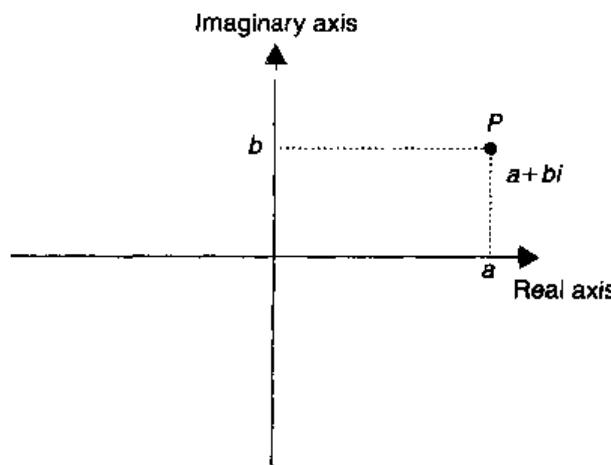


Figure 6.1 Representing a complex number in rectangular coordinates.

on a plane (Figure 6.1). The horizontal axis of the plane is the real axis, and the vertical axis of the plane is the imaginary axis, so that any complex number $a + bi$ can be represented as a single point a units along the real axis and b units along the imaginary axis. A complex number represented this way is said to be in *rectangular coordinates*, since the real and imaginary axes define the sides of a rectangle.

A complex number can also be represented as a vector of length z and angle θ pointing from the origin of the plane to the point P (Figure 6.2). A complex number represented this way is said to be in *polar coordinates*.

$$c = a + bi = z \angle \theta$$

The relationships among the rectangular and polar coordinate terms a , b , z , and θ are

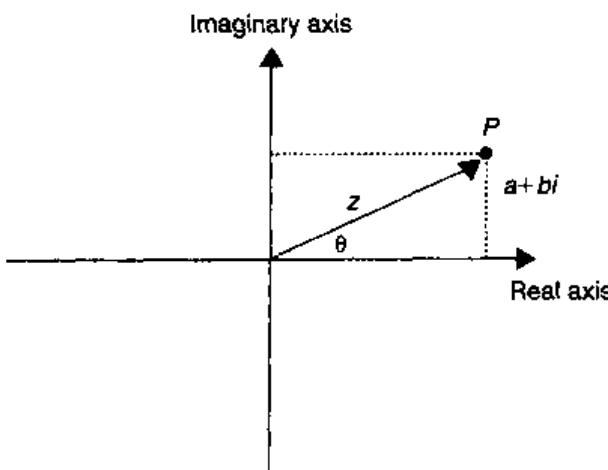


Figure 6.2 Representing a complex number in polar coordinates.

$$a = z \cos \theta \quad (6.2)$$

$$b = z \sin \theta \quad (6.3)$$

$$z = \sqrt{a^2 + b^2} \quad (6.4)$$

$$\theta = \tan^{-1} \frac{b}{a} \quad (6.5)$$

MATLAB uses rectangular coordinates to represent complex numbers. Each complex number consists of a pair of real numbers (a, b). The first number (a) is the real part of the complex number, and the second number (b) is the imaginary part of the complex number.

If complex numbers c_1 and c_2 are defined as $c_1 = a_1 + b_1i$ and $c_2 = a_2 + b_2i$, then the addition, subtraction, multiplication, and division of c_1 and c_2 are defined as

$$c_1 + c_2 = (a_1 + a_2) + (b_1 + b_2)i \quad (6.6)$$

$$c_1 - c_2 = (a_1 - a_2) + (b_1 - b_2)i \quad (6.7)$$

$$c_1 \times c_2 = (a_1a_2 - b_1b_2) + (a_1b_2 + b_1a_2)i \quad (6.8)$$

$$\frac{c_1}{c_2} = \frac{a_1a_2 + b_1b_2}{a_2^2 + b_2^2} + \frac{b_1a_2 - a_1b_2}{a_2^2 + b_2^2}i \quad (6.9)$$

When two complex numbers appear in a binary operation, MATLAB performs the required additions, subtractions, multiplications, or divisions between the two complex numbers using versions of these formulas.

6.1.1 Complex Variables

A complex variable is created automatically when a complex value is assigned to a variable name. The easiest way to create a complex value is to use the intrinsic values i or j , both of which are predefined to be $\sqrt{-1}$. For example, the following statement stores the complex value $4 + i3$ into variable $c1$.

```
>> c1 = 4 + i*3
c1 =
    4.0000 + 3.0000i
```

The function `isreal` can be used to determine whether a given array is real or complex. If any element of an array has an imaginary component, then the array is complex, and `isreal(array)` returns a 0.

6.1.2 Using Complex Numbers with Relational Operators

It is possible to compare two complex numbers with the `==` relational operator to see if they are equal to each other, and to compare them with the `~=` operator to see if they are not equal to each other. Both of these operators produce the expected results. For example, if $c_1 = 4 + i3$ and $c_2 = 4 - i3$, then the relational operation $c_1 == c_2$ produces a 0 and the relational operation $c_1 ~=~ c_2$ produces a 1.

However, *comparisons with the `>`, `<`, `>=`, or `<=` operators do not produce the expected results*. When complex numbers are compared with these relational operators, only the *real parts* of the numbers are compared. For example, if $c_1 = 4 + i3$ and $c_2 = 3 + i8$, then the relational operation $c_1 > c_2$ produces a 1, even though the magnitude of c_1 is really smaller than the magnitude of c_2 .

If we ever need to compare two complex numbers with these operators, we will probably be more interested in the total magnitude of the number than we are in the magnitude of only its real part. The magnitude of a complex number can be calculated with the `abs` intrinsic function (see the next section, or directly from Equation (6.4)).

$$|c| = \sqrt{a^2 + b^2} \quad (6.4)$$

If we compare the *magnitudes* of c_1 and c_2 , the results are more reasonable. `abs(c1) > abs(c2)` produces a 0, since the magnitude of c_2 is greater than the magnitude of c_1 .

Programming Pitfalls

Be careful when using the relational operators with complex numbers. The relational operators `>`, `>=`, `<`, and `<=` only compare the *real parts* of complex numbers, not their magnitudes. If you need these relational operators with a complex number, it will probably be more sensible to compare the total magnitudes rather than only the real components.

6.1.3 Complex Functions

MATLAB includes many functions that support complex calculations. These functions fall into three general categories.

1. Type Conversion Functions

These functions convert data from the complex data type to the real (`double`) data type. Function `real` converts the *real part* of a complex number into the double data type and throws away the imaginary part of the complex number. Function `imag` converts the *imaginary part* of a complex number into a real number.

Table 6.1 Some Functions That Support Complex Numbers

Function	Description
<code>conj(c)</code>	Computes the complex conjugate of a number c . If $c = a + bi$, then $\text{conj}(c) = a - bi$.
<code>real(c)</code>	Returns the real portion of the complex number c .
<code>imag(c)</code>	Returns the imaginary portion of the complex number c .
<code>isreal(c)</code>	Returns true (1) if no element of array c has an imaginary component. Therefore, $\sim\text{isreal}(c)$ returns true (1) if an array is complex.
<code>abs(c)</code>	Returns the magnitude of the complex number c .
<code>angle(c)</code>	Returns the angle of the complex number c , computed from the expression <code>atan2(imag(c), real(c))</code> .

2. Absolute Value and Angle Functions

These functions convert a complex number to its polar representation. Function `abs(c)` calculates the absolute value of a complex number using the equation

$$\text{abs}(c) = \sqrt{a^2 + b^2}$$

where $c = a + bi$. Function `angle(c)` calculates the angle of a complex number using the equation

$$\text{angle}(c) = \text{atan2}(\text{imag}(c), \text{real}(c))$$

producing an answer in the range $-\pi < \theta \leq \pi$.

3. Mathematical Functions

Most elementary mathematical functions are defined for complex values. These functions include exponential functions, logarithms, trigonometric functions, and square roots. The functions `sin`, `cos`, `log`, `sqrt`, and so on, will work as well with complex data as they will with real data.

Some of the intrinsic functions that support complex numbers are listed in Table 6.1.

Example 6.1—The Quadratic Equation (Revisited)

The availability of complex numbers often simplifies the calculations required to solve problems. For example, when we solved the quadratic equation in Example 3.2, it was necessary to take three separate branches through the program depending on the sign of the discriminant. With complex numbers available, the square root of a negative number presents no difficulties, so we can greatly simplify these calculations.

Write a general program to solve for the roots of a quadratic equation, regardless of type. Use complex variables so that no branches will be required based on the value of the discriminant.

Solution

1. State the Problem

Write a program that will solve for the roots of a quadratic equation, whether they are distinct real roots, repeated real roots, or complex roots, without requiring tests on the value of the discriminant.

2. Define the Inputs and Outputs

The inputs required by this program are the coefficients a , b , and c of the quadratic equation

$$ax^2 + bx + c = 0 \quad (3.1)$$

The output from the program will be the roots of the quadratic equation, whether they are real, repeated, or complex.

3. Describe the Algorithm

This task can be broken down into three major sections, whose functions are input, processing, and output.

Read the input data
Calculate the roots
Write out the roots

We now break each of these major sections into smaller, more detailed pieces. In this algorithm, the value of the discriminant is unimportant in determining how to proceed. The resulting pseudocode is:

```
Prompt the user for the coefficients a, b, and c.  
Read a, b, and c  
discriminant ← b^2 - 4 * a * c  
x1 ← ( -b + sqrt(discriminant) ) / ( 2 * a )  
x2 ← ( -b - sqrt(discriminant) ) / ( 2 * a )  
Print 'The roots of this equation are: '  
Print 'x1 = ', real(x1), ' +i ', imag(x1)  
Print 'x2 = ', real(x2), ' +i ', imag(x2)
```

4. Turn the Algorithm into MATLAB Statements

The final MATLAB code follows.

```
% Script file: calc_roots2.m  
%  
% Purpose:  
% This program solves for the roots of a quadratic equation  
% of the form a*x^2 + b*x + c = 0. It calculates the answers  
% regardless of the type of roots that the equation possesses.  
%
```

```

% Record of revisions:
%   Date      Programmer      Description of change
%   ----      ======      =====
%   12/06/98  S. J. Chapman  Original code
%
% Define variables:
%   a          -- Coefficient of x^2 term of equation
%   b          -- Coefficient of x term of equation
%   c          -- Constant term of equation
%   discriminant -- Discriminant of the equation
%   x1         -- First solution of equation
%   x2         -- Second solution of equation

% Prompt the user for the coefficients of the equation
disp ('This program solves for the roots of a quadratic ');
disp ('equation of the form A*X^2 + B*X + C = 0. ');
a = input ('Enter the coefficient A: ');
b = input ('Enter the coefficient B: ');
c = input ('Enter the coefficient C: ');

% Calculate discriminant
discriminant = b^2 - 4 * a * c;

% Solve for the roots
x1 = ( -b + sqrt(discriminant) ) / ( 2 * a );
x2 = ( -b - sqrt(discriminant) ) / ( 2 * a );

% Display results
disp ('The roots of this equation are:');
fprintf ('x1 = (%f) +i (%f)\n', real(x1), imag(x1));
fprintf ('x2 = (%f) +i (%f)\n', real(x2), imag(x2));

```

5. Test the Program

Next, we must test the program using real input data. We will test cases in which the discriminant is greater than, less than, and equal to 0 to be certain that the program is working properly under all circumstances. From Equation (3.1), it is possible to verify the solutions to the following equations.

$$x^2 + 5x + 6 = 0 \quad x = -2, \text{ and } x = -3$$

$$x^2 + 4x + 4 = 0 \quad x = -2$$

$$x^2 + 2x + 5 = 0 \quad x = -1 \pm 2i$$

When the above coefficients are fed into the program, the results are

```

> calc_roots2
This program solves for the roots of a quadratic
equation of the form A*X^2 + B*X + C = 0.
Enter the coefficient A: 1
Enter the coefficient B: 5

```

```

Enter the coefficient C: 6
The roots of this equation are:
x1 = (-2.000000) +i (0.000000)
x2 = (-3.000000) +i (0.000000)
» calc_roots2
This program solves for the roots of a quadratic
equation of the form A*X^2 + B*X + C = 0.
Enter the coefficient A: 1
Enter the coefficient B: 4
Enter the coefficient C: 4
The roots of this equation are:
x1 = (-2.000000) +i (0.000000)
x2 = (-2.000000) +i (0.000000)
» calc_roots2
This program solves for the roots of a quadratic
equation of the form A*X^2 + B*X + C = 0.
Enter the coefficient A: 1
Enter the coefficient B: 2
Enter the coefficient C: 5
The roots of this equation are:
x1 = (-1.000000) +i (2.000000)
x2 = (-1.000000) +i (-2.000000)

```

The program gives the correct answers for our test data in all three possible cases. Note how much simpler this program is compared to the quadratic root solver found in Example 3.2. The use of the complex data type has greatly simplified our program.

6.1.4 Plotting Complex Data

Complex data has both real and imaginary components, and plotting complex data with MATLAB is a bit different than plotting real data. For example, consider the function

$$y(t) = e^{-0.2t} (\cos t + i \sin t) \quad (6.10)$$

If this function is plotted with the conventional `plot` command, only the real data will be plotted—the imaginary part will be ignored. The following statements produce the plot shown in Figure 6.3, together with a warning message that the imaginary part of the data is being ignored.

```

t = 0:pi/20:4*pi;
y = exp(-0.2*t).*(cos(t)+i*sin(t));
plot(t,y);
title('Plot of Complex Function vs Time');
xlabel('itt');
ylabel('ity(t)');

```

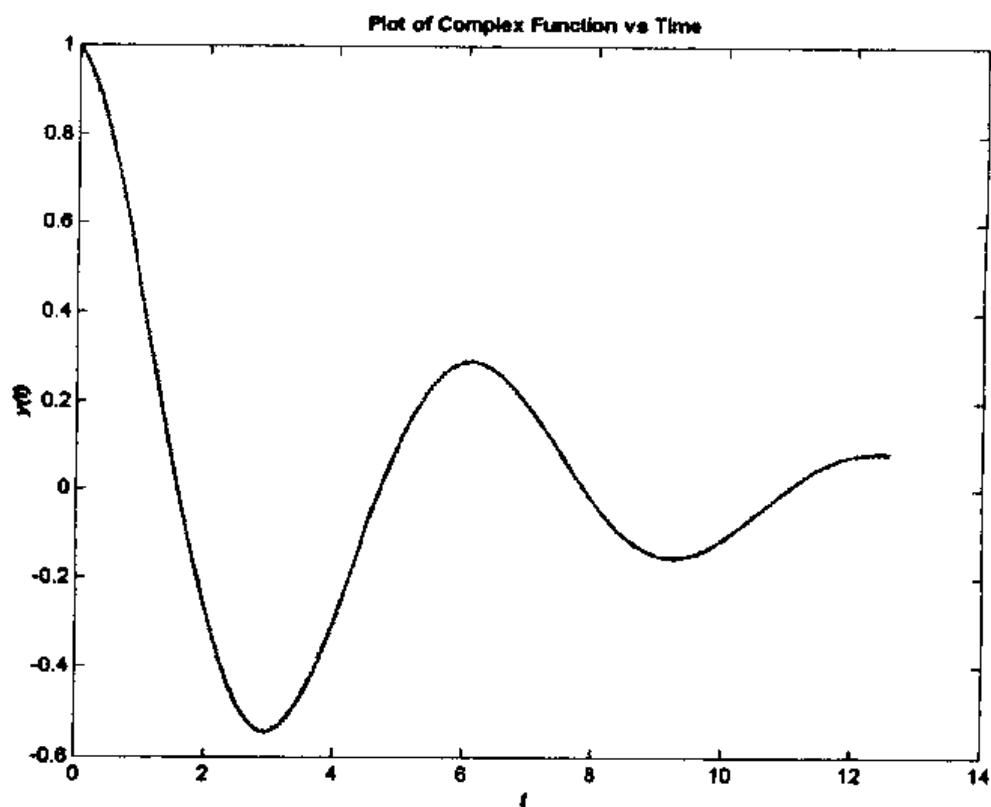


Figure 6.3 Plot of $y(t) = e^{-0.2t}(\cos t + i \sin t)$ using the command `plot (t,y)`.

If both the real and imaginary parts of the function are of interest, then the user has several choices. Both parts can be plotted as a function of time on the same axes using the following statements (Figure 6.4).

```
t = 0:pi/20:4*pi;
y = exp(-0.2*t).*(cos(t)+i*sin(t));
plot(t,real(y), 'b-');
hold on;
plot(t,imag(y), 'r--');
title('Plot of Complex Function vs Time');
xlabel('t');
ylabel('y(t)');
legend ('real','imaginary');
hold off;
```

Alternatively, the real part of the function can be plotted versus the imaginary part. If a single complex argument is supplied to the `plot` function, it automatically generates a plot of the real part versus the imaginary part. The statements to generate this plot follow, and the result is shown in Figure 6.5.

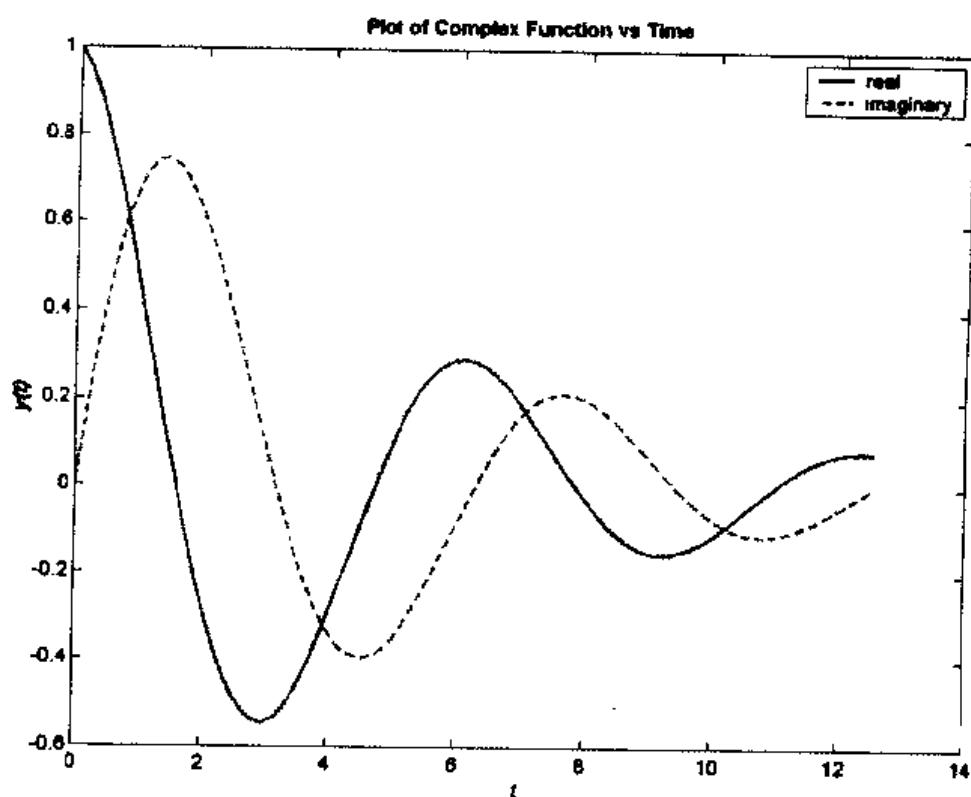


Figure 6.4 Plot of real and imaginary parts of $y(t)$ versus time.

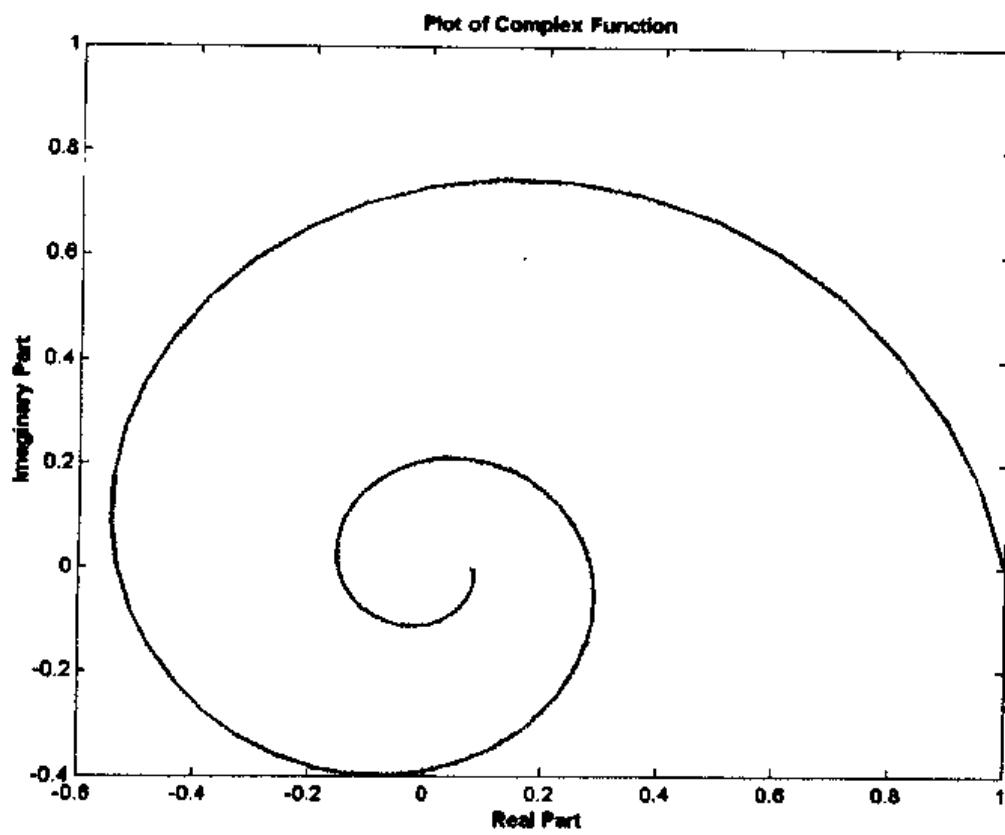


Figure 6.5 Plot of real versus imaginary parts of $y(t)$.

```
t = 0:pi/20:4*pi;
y = exp(-0.2*t).*(cos(t)+i*sin(t));
plot(y,'b-');
title('Plot of Complex Function');
xlabel('Real Part');
ylabel('Imaginary Part');
```

Finally, the function can be plotted as a polar plot showing magnitude versus angle. The statements to generate this plot follow, and the result is shown in Figure 6.6.

```
t = 0:pi/20:4*pi;
y = exp(-0.2*t).*(cos(t)+i*sin(t));
polar(angle(y),abs(y));
title('Plot of Complex Function');
```

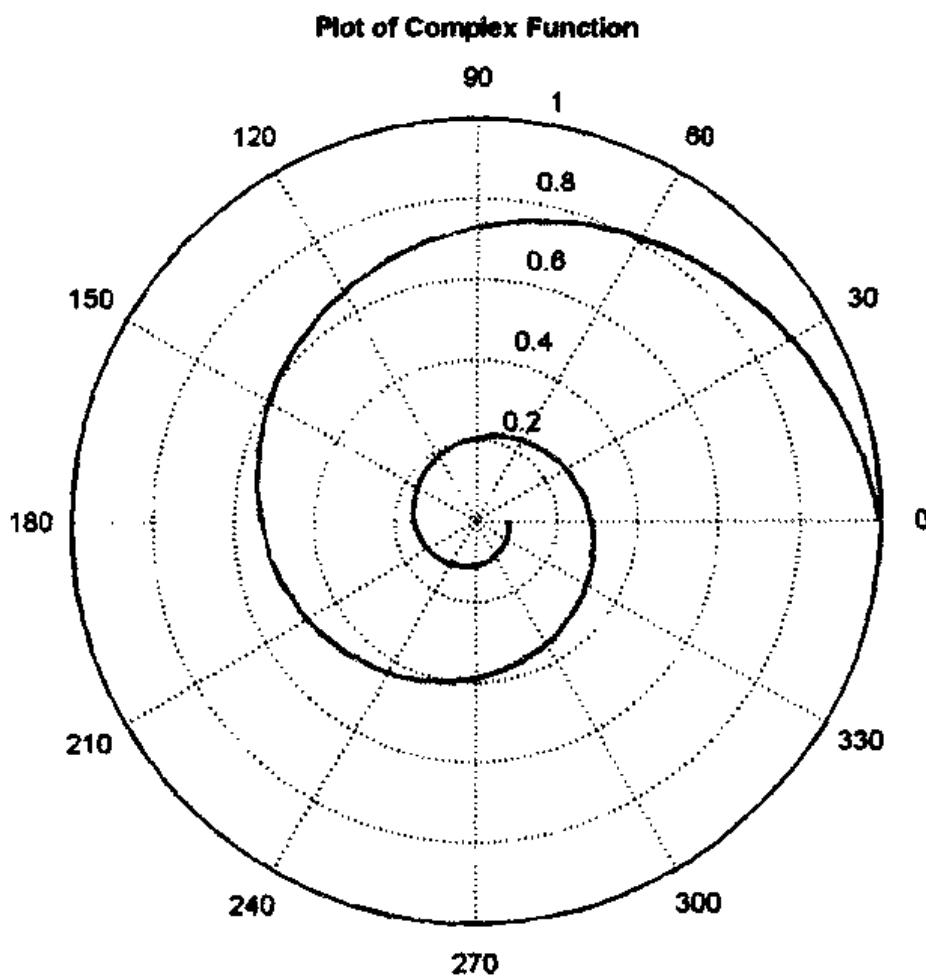


Figure 6.6 Polar plot of magnitude of $y(t)$ versus angle.

6.2 String Functions

A MATLAB string is an array of type `char`. Each character is stored in two bytes of memory.

A character variable is automatically created when a string is assigned to it. For example, the statement

```
str = 'This is a test';
```

creates a 14-element character array. The output of `whos` for this array is

```
> whos str
  Name      Size            Bytes  Class
  str       1x14           28  char array
                                         Grand total is 14 elements using 28 bytes
```

A special function `ischar` can be used to check for character arrays. If a given variable is a character array, then `ischar` returns a true (1) value. If it is not, `ischar` returns a false value.

The following subsections describe MATLAB functions useful for manipulating character strings.

6.2.1 String Conversion Functions

Variables can be converted from the character data type to the double data type using the `double` function. Thus, the function `double(str)` yields the result

```
> x = double(str)
x =
  Columns 1 through 12
    84 104 105 115  32 105 115  32  97  32 116 101
  Columns 13 through 14
    115 116
```

Variables can be converted from the double data type to the character data type using the `char` function. Thus, the function `char(x)` yields the result

```
> z = char(x)
z =
  This is a test
```

6.2.2 Creating Two-Dimensional Character Arrays

It is possible to create two-dimensional character arrays, but *each row of such an array must have exactly the same length*. If one of the rows is shorter than the other rows, the character array is invalid and will produce an error. For example, the following statements are illegal because the two rows have different lengths.

```
name = ['Stephen J. Chapman'; 'Senior Engineer'];
```

The easiest way to produce two-dimensional character arrays is with the `char` function. This function will automatically pad all strings to the length of the largest input string.

```
>> name = char('Stephen J. Chapman', 'Senior Engineer')
name =
Stephen J. Chapman
Senior Engineer
```

Two-dimensional character arrays can also be created with function `strvcat`, which is described in the next section.

Good Programming Practice

Use the `char` function to create two-dimensional character arrays without worrying about padding each row to the same length.

It is possible to remove any extra blanks from a string when it is extracted from an array using the `deblank` function. For example, the following statements remove the second line from array `name`, and compare the results with and without blank trimming.

```
>> line2 = name(2,:)
line2 =
Senior Engineer
>> line2_trim = deblank(name(2,:))
line2_trim =
Senior Engineer
>> size(line2)
ans =
    1    18
>> size(line2_trim)
ans =
    1    15
```

6.2.3 Concatenating Strings

Function `strcat` concatenates two or more strings horizontally, ignoring any trailing blanks but preserving blanks within the strings. For example, the statement that follows

```
>> result = strcat('String 1 ', 'String 2')
result =
String 1String 2
```

produces the result 'String 1String 2'.

Function `strvcat` concatenates two or more strings vertically, automatically padding the strings to make a valid two-dimensional array. This function produces the result

```
> result = strvcat('Long String 1 ', 'String 2')
result =
Long String 1
String 2
```

6.2.4 Comparing Strings

Strings and substrings can be compared in several ways.

- Two strings, or parts of two strings, can be compared for equality.
- Two individual characters can be compared for equality.
- Strings can be examined to determine whether each character is a letter or whitespace.

6.2.4.1 Comparing Strings for Equality

You can use four MATLAB functions to compare two strings as a whole for equality. They are

- `strcmp` Determines if two strings are identical.
- `strcmpi` Determines if two strings are identical ignoring case.
- `strncmp` Determines if the first n characters of two strings are identical.
- `strncmpi` Determines if the first n characters of two strings are identical ignoring case.

Function `strcmp` compares two strings, including any leading and trailing blanks, and returns a 1 (true) if the strings are identical¹. Otherwise, it returns a 0. Function `strcmpi` is the same as `strcmp`, except that it ignores the case of letters (that is, it treats 'a' as equal to 'A').

Function `strncmp` compares the first n characters of two strings, including any leading blanks, and returns a 1 (true) if the characters are identical. Otherwise, it returns a 0. Function `strncmpi` is the same as `strncmp`, except that it ignores the case of letters.

To understand these functions, consider the two strings

```
str1 = 'hello';
str2 = 'Hello';
str3 = 'help';
```

¹ Caution: The behavior of this function is different from that of the `strcmp` in C. C programmers can be tripped up by this difference.

Strings `str1` and `str2` are not identical, but they differ only in the case of one letter. Therefore, `strcmp` returns 0 (false), while `strcmpi` returns 1 (true).

```
>> c = strcmp(str1,str2)
c =
    0
>> c = strcmpi(str1,str2)
c =
    1
```

Strings `str1` and `str3` are also not identical, and both `strcmp` and `strcmpi` will return a 0. However, the first three characters of `str1` and `str3` are identical, so invoking `strncmp` with any value up to 3 returns 1.

```
>> c = strncmp(str1,str3,2)
c =
    1
```

6.2.4.2 Comparing Individual Characters for Equality and Inequality

You can use MATLAB relational operators on character arrays to test for equality *one character at a time*, as long as the arrays you are comparing have equal dimensions, or one is a scalar. For example, you can use the equality operator (`==`) to determine which characters in two strings match.

```
>> a = 'fate';
>> b = 'cake';
>> result = a == b
result =
0 1 0 1
```

All of the relational operators (`>`, `>=`, `<`, `<=`, `==`, `~=`) compare the ASCII values of corresponding characters.

Unlike C, MATLAB does not have an intrinsic function to define a “greater than” or “less than” relationship between two strings taken as a whole. We will create such a function in an example at the end of this section.

6.2.4.3 Categorizing Characters within a String

There are two functions for categorizing characters on a character-by-character basis inside a string.

- `isletter` determines if a character is a letter.
- `isspace` determines if a character is whitespace (blank, tab, or new line).

For example, let us create a string named `mystring`.

```
mystring = 'Room 23a';
```

Function `isletter` examines each character in the string, producing an output vector of the same length as `mystring` containing a 1 in each location corresponding to a character.

```
>> a = isletter(mystring)
a =
1 1 1 1 0 0 0 1
```

The first four and the last elements in `a` are 1 (true) because the corresponding characters of `mystring` are letters.

Function `isspace` also examines each character in the string, producing an output vector of the same length as `mystring` containing a 1 in each location corresponding to a whitespace.

```
>> a = isspace(mystring)
a =
0 0 0 0 1 0 0 0
```

The fifth element in `a` is 1 (true) because the corresponding character of `mystring` is a blank.

6.2.5 Searching / Replacing Characters within a String

MATLAB provides several functions for searching and replacing characters in a string. Consider a string named `test`.

```
test = 'This is a test!';
```

Function `findstr` returns the starting position of all occurrences of the shorter of two strings within a longer string. For example, to find all occurrences of the string 'is' inside `test`,

```
>> position = findstr(test,'is')
position =
3      6    .
```

The string 'is' occurs twice within `test`, starting at positions 3 and 6.²

Function `strmatch` is another matching function. This one looks at the beginning characters of the rows of a 2-D character array, and returns a list of those rows that start with the specified character sequence. The form of this function is

```
result = strmatch(str,array);
```

For example, suppose that we create a 2-D character array with the function `strvcat`:

```
array = strvcat('maxarray','min value','max value');
```

² A new function `strfind` was added to MATLAB beginning with version 6.1 (Release 12.1). This function largely duplicates the functionality of `findstr`, and it is not backwards compatible with earlier versions of MATLAB, so for now I recommend that it not be used. A description of function `strfind` can be found in the MATLAB online documentation, if you are using version 6.1 or later.

then the following statement will return the row numbers of all rows beginning with the letters 'max'.

```
>> result = strmatch('max',array)
result =
    1
    3
```

Function `strrep` performs the standard search-and-replace operation. It finds all occurrences of one string within another one, and replaces them by a third string. The form of this function is

```
result = strrep(str,srch,repl)
```

where `str` is the string being checked, `srch` is the character string to search for, and `repl` is the replacement character string. For example,

```
>> result = strrep('test','test','pest')
result =
This is a pest!
```

The `strtok` function returns the characters before the first occurrence of a delimiting character in an input string. The default delimiting characters are the set of whitespace characters. The form of `strtok` is

```
[token,remainder] = strtok(string,delim)
```

where `string` is the input character string, `delim` is the (optional) set of delimiting characters, `token` is the first set of characters delimited by a character in `delim`, and `remainder` is the rest of the line. For example,

```
>> [token,remainder] = strtok('This is a test!')
token =
This
remainder =
    is a test!
```

You can use the `strtok` function to parse a sentence into words. For example, the following code separates out each word in the character array `input_string` and places it in a separate row of the character array `all_words`.

```
function all_words = words(input_string)
remainder = input_string;
all_words = '';
while (any(remainder))
    [chopped,remainder] = strtok(remainder);
    all_words = strvcat(all_words,chopped);
end
```

6.2.6 Uppercase and Lowercase Conversion

Functions `upper` and `lower` convert all of the alphabetic characters within a string to uppercase and lowercase respectively. For example

```

>> result = upper('This is test 1')
result =
THIS IS TEST 1!
>> result = lower('This is test 2')
result =
this is test 2!
```

Note that the alphabetic characters were converted to the proper case, while the numbers and punctuation were unaffected.

6.2.7 Numeric-to-String Conversions

MATLAB contains several string/numeric conversion functions to change numeric values into character strings. We have already seen two such functions, `num2str` and `int2str`. Consider a scalar `x`.

```
x = 5317;
```

By default, MATLAB stores the number `x` as a 1×1 double array containing the value 5317. The `int2str` (integer to string) function converts this scalar into a 1-by-4 char array containing the string '5317'.

```

>> x = 5317;
>> y = int2str(x);
>> whos

  Name      Size            Bytes  Class
  x         1x1                  8  double array
  y         1x4                  8  char array

Grand total is 5 elements using 16 bytes
```

The function `num2str` provides more control over the format of the output string. An optional second argument sets the number of digits in the output string, or specifies an actual format to use. For example

```

>> p = num2str(pi,7)
p =
3.141593
>> p = num2str(pi,'%10.5e')
p =
3.14159e+000
```

Both `int2str` and `num2str` are handy for labeling plots. For example, the following lines use `num2str` to prepare automated labels for the `x`-axis of a plot.

```

function plotlabel(x,y)
plot(x,y)
str1 = num2str(min(x));
str2 = num2str(max(x));
out = ['Value of f from ' str1 ' to ' str2];
xlabel(out);
```

There are also conversion functions designed to change numeric values into strings representing a decimal value in another base, such as a binary or hexadecimal representation. For example, the `dec2hex` function converts a decimal value into the corresponding hexadecimal string.

```
dec_num = 4035;
hex_num = dec2hex(dec_num)
hex_num =
FC3
```

Other functions of this type include `hex2num`, `hex2dec`, `bin2dec`, `dec2bin`, `base2dec`, and `dec2base`. MATLAB includes on-line help for all of these functions.

MATLAB function `mat2str` converts an array to a string that MATLAB can evaluate. This string is useful input for a function such as `eval`, which evaluates input strings just as if they were typed at the MATLAB command line. For example, if we define array `a` as

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

then the function `mat2str` will return the result

```
>> b = mat2str(a)
b =
[1 2 3; 4 5 6]
```

Finally, MATLAB includes a special function `sprintf` that is identical to function `fprintf`, except that the output goes into a character string instead of the Command Window. This function provides complete control over the formatting of the character string. For example

```
>> str = sprintf('The value of pi = %8.6f.',pi)
str =
The value of pi = 3.141593.
```

This function is extremely useful in creating complex titles and labels for plots.

6.2.8 String-to-Numeric Conversions

MATLAB converts strings containing numbers to numeric form using the function `eval`. For example, the string `a` containing the characters '3.141592' can be converted to numeric form by the following statements.

```
>> a = '3.141592';
>> b = eval(a)
b =
3.1416
```

```
>> whos
  Name      Size            Bytes  Class
  a          1x8             16   char array
  b          1x1              8   double array

Grand total is 9 elements using 24 bytes
```

Strings can also be converted to numeric form using the function `sscanf`. This function converts a string into a number according to a format conversion character. The simplest form of this function is

```
value = sscanf(string,format)
```

where `string` is the string to scan, and `format` specifies the type of conversion to occur. The two most common conversion specifiers for `sscanf` are '`%d`' for decimals and '`%g`' for floating-point numbers. This function will be covered in much greater detail in Chapter 8. It is extremely useful in creating complex titles and labels for plots.

The following examples illustrate the use of `sscanf`.

```
>> value1 = sscanf('3.141593','%g')
value1 =
    3.1416
>> value2 = sscanf('3.141593','%d')
value2 =
    3
```

6.2.9 Summary

The common MATLAB string functions are summarized in Table 6.2.

Table 6.2 Common MATLAB String Functions

Category	Function	Description
General		
	char	(1) Convert numbers to the corresponding character values. (2) Create a 2-D character array from a series of strings.
	double	Convert characters to the corresponding numeric codes.
	blanks	Create a string of blanks.
	deblank	Remove trailing blanks.
String Tests		
	ischar	Returns true (1) for a character array.
	isletter	Returns true (1) for letters of the alphabet.
	isspace	Returns true (1) for whitespace.

Continued

Table 6.2 (Continued)

Category	Function	Description
String Operations		
	<code>strcat</code>	Concatenate strings.
	<code>strvcat</code>	Concatenate strings vertically.
	<code>strcmp</code>	Returns true (1) if two strings are identical.
	<code>strcmpi</code>	Returns true (1) if two strings are identical, ignoring case.
	<code>strncmp</code>	Returns true (1) if first n characters of two strings are identical.
	<code>strncmpi</code>	Returns true (1) if first n characters of two strings are identical, ignoring case.
	<code>findstr</code>	Find one string within another one.
	<code>strfind</code>	Find one string within another one (Version 6.1 or later).
	<code>strjust</code>	Justify string.
	<code>strmatch</code>	Find matches for string.
	<code>strrep</code>	Replace one string with another.
	<code>strtok</code>	Find token in string.
	<code>upper</code>	Convert string to uppercase.
	<code>lower</code>	Convert string to lowercase.
Number-to-String Conversion		
	<code>int2str</code>	Convert integer to string.
	<code>num2str</code>	Convert number to string.
	<code>mat2str</code>	Convert matrix to string.
	<code>sprintf</code>	Write formatted data to string.
String-to-Number Conversion		
	<code>str2double</code>	Convert string to a double value. (This function is new in MATLAB 5.3.)
	<code>str2num</code>	Convert string to number.
	<code>sscanf</code>	Read formatted data from string.
Base Number Conversion		
	<code>hex2num</code>	Convert IEEE hexadecimal string to double.
	<code>hex2dec</code>	Convert hexadecimal string to decimal integer.
	<code>dec2hex</code>	Convert decimal to hexadecimal string.
	<code>bin2dec</code>	Convert binary string to decimal integer.
	<code>dec2bin</code>	Convert decimal integer to binary string.
	<code>base2dec</code>	Convert base B string to decimal integer.
	<code>dec2base</code>	Convert decimal integer to base B string.

Example 6.2—String Comparison Function

In C, function `strcmp` compares two strings according to the order of their characters in the ASCII table (called the **lexicographic order** of the characters), and returns a -1 if the first string is lexicographically less than the second string, a 0 if the strings are equal, and a $+1$ if the first string is lexicographically greater than the second string. This function is extremely useful for such purposes as sorting strings in alphabetic order.

Create a new MATLAB function `c_strcmp` that compares two strings in a similar fashion to the C function and returns similar results. The function should ignore trailing blanks in doing its comparisons. Note that the function must be able to handle the situation where the two strings are of different lengths.

Solution

1. State the Problem

Write a function that will compare two strings `str1` and `str2`, and return the following results.

- -1 if `str1` is lexicographically less than `str2`.
- 0 if `str1` is lexicographically less than `str2`.
- $+1$ if `str1` is lexicographically greater than `str2`.

The function must work properly if `str1` and `str2` do not have the same length, and the function should ignore trailing blanks.

2. Define the Inputs and Outputs

The inputs required by this function are two strings, `str1` and `str2`. The output from the function will be a -1 , 0 , or $+1$, as appropriate.

3. Describe the Algorithm

This task can be broken down into four major sections:

```

Verify input strings
Pad strings to be equal length
Compare characters from beginning to end, looking
    for the first difference
Return a value based on the first difference

```

We will now break each of the above major sections into smaller, more detailed pieces. First, we must verify that the data passed to the function is correct. The function must have exactly two arguments, and the arguments must both be characters. The pseudocode for this step is

```

% Check for a legal number of input arguments.
msg = nargchk(2,2,nargin)
error(msg)

% Check to see if the arguments are strings
if either argument is not a string

```

```

        error('str1 and str2 must both be strings')
else
    (add code here)
end

```

Next, we must pad the strings to equal lengths. The easiest way to do this is to combine both strings into a 2-D array using `strvcat`. Note that this step effectively results in the function ignoring trailing blanks because both strings are padded out to the same length. The pseudocode for this step is:

```

% Pad strings
strings = strvcat(str1,str2)

```

Now we must compare each character until we find a difference, and return a value based on that difference. One way to do this is to use relational operators to compare the two strings, creating an array of 0's and 1's. We can then look for the first one, which will correspond to the first difference between the two strings. The pseudocode for this step is

```

% Compare strings
diff = strings(1,:) ~= strings(2,:)
if sum(diff) == 0
    % Strings match
    result = 0
else
    % Find first difference
    ival = find(diff)
    if strings(1,ival(1)) > strings(2,ival(1))
        result = 1
    else
        result = -1
    end
end

```

4. Turn the Algorithm into MATLAB Statements

The final MATLAB code follows.

```

function result = c_strcmp(str1,str2)
%C_strcmp Compare strings like C function "strcmp"
% Function C_strcmp compares two strings, and returns
% a -1 if str1 < str2, a 0 if str1 == str2, and a
% +1 if str1 > str2.

% Define variables:
% diff      -- Logical array of string differences
% msg       -- Error message
% result    -- Result of function
% str1      -- First string to compare

```

```

% str2      -- Second string to compare
% strings   -- Padded array of strings

% Record of revisions:
% Date      Programmer      Description of change
% =====    ======          =====
% 05/18/99  S. J. Chapman  Original code

% Check for a legal number of input arguments.
msg = nargchk(2,2,nargin);
error(msg);

% Check to see if the arguments are strings
if ~isstr(str1) & isstr(str2))
    error('Both str1 and str2 must both be strings!')
else

    % Pad strings
    strings = strvcat(str1,str2);

    % Compare strings
    diff = strings(1,:) ~= strings(2,:);
    if sum(diff) == 0

        % Strings match, so return a zero!
        result = 0;
    else
        % Find first difference between strings
        ival = find(diff);
        if strings(1,ival(1)) > strings(2,ival(1))
            result = 1;
        else
            result = -1;
        end
    end
end

```

5. Test the Program

Next, we must test the function using various strings.

```

>> result = c_strcmp('String 1','String 1')
result =
     0
>> result = c_strcmp('String 1','String 1    ')
result =
     0
>> result = c_strcmp('String 1','String 2')
result =
    -1
>> result = c_strcmp('String 1','String 0')

```

```

result =
    1
» result = c_strcmp('String','str')
result =
    -1

```

The first test returns a zero because the two strings are identical. The second test also returns a zero because the two strings are identical *except for trailing blanks*, and trailing blanks are ignored. The third test returns a -1 because the two strings first differ in position 8, and '`1`' $<$ '`2`' at that position. The fourth test returns a `1` because the two strings first differ in position 8, and '`1`' $>$ '`0`' at that position. The fifth test returns a -1 because the two strings first differ in position 1, and '`s`' $<$ '`s`' in the ASCII collating sequence.

This function appears to be working properly.

Quiz 6.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 6.1 through 6.2. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. What is the value of `result` in the following statements?
 - (a) `x = 12 + i*5;`
`y = 5 - i*13;`
`result = x > y;`
 - (b) `x = 12 + i*5;`
`y = 5 - i*13;`
`result = abs(x) > abs(y);`
 - (c) `x = 12 + i*5;`
`y = 5 - i*13;`
`result = real(x) - imag(y);`
2. If `array` is a complex array, what does the function `plot(array)` do?
3. How can you convert a vector of the `char` data type into a vector of the `double` data type?

For questions 4 through 11, determine whether these statements are correct. If they are, what is produced by each set of statements?

4. `str1 = 'This is a test! ';`
`str2 = 'This line, too.';`
`res = strcat(str1,str2);`

```

5. str1 = 'Line 1';
str2 = 'line 2';
res = strcat(str1,str2);
6. str1 = 'This is a test! ';
str2 = 'This line, too.';
res = [str1; str2];
7. str1 = 'This is a test! ';
str2 = 'This line, too.';
res = strvcat(str1,str2);
8. str1 = 'This is a test! ';
str2 = 'This line, too.';
res = strncmp(str1,str2,5);
9. str1 = 'This is a test! ';
res = findstr(str1,'s');
10. str1 = 'This is a test! ';
str1(4:7) = upper(str1(4:7));
11. str1 = 'This way to the egress.';
str2 = 'This way to the egret.'
res = strncmp(str1,str2);

```

6.3 Multidimensional Arrays

Beginning with Version 5.0, MATLAB supports arrays with more than two dimensions. These **multidimensional arrays** are very useful for displaying data that intrinsically has more than two dimensions or for displaying multiple versions of 2-D data sets. For example, measurements of pressure and velocity throughout a three-dimensional volume are very important in such studies as aerodynamics and fluid dynamics. These sorts of areas naturally use multidimensional arrays.

Multidimensional arrays are a natural extension of two-dimensional arrays. Each additional dimension is represented by one additional subscript used to address the data.

It is very easy to create a multidimensional array. They can be created either by assigning values directly in assignment statements or by using the same functions that are used to create one- and two-dimensional arrays. For example, suppose that you have a two-dimensional array created by the assignment statement

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

This is a 2×4 array, with each element addressed by two subscripts. The array can be extended to be a three-dimensional $2 \times 4 \times 3$ array with the following assignment statements.

```

>> a(:,:,2) = [ 9 10 11 12; 13 14 15 16];
>> a(:,:,3) = [ 17 18 19 20; 21 22 23 24]
a(:,:,1) =
    1     2     3     4
    5     6     7     8
a(:,:,2) =
    9    10    11    12
   13    14    15    16
a(:,:,3) =
   17    18    19    20
   21    22    23    24

```

Individual elements in this multidimensional array can be addressed by the array name followed by three subscripts, and subsets of the data can be created using the colon operators. For example, the value of $a(2,2,2)$ is

```

>> a(2,2,2)
ans =
    14

```

and the vector $a(1,1,:)$ is

```

>> a(1,1,:)
ans(:,:,1) =
    1
ans(:,:,2) =
    9
ans(:,:,3) =
   17

```

Multidimensional arrays can also be created using the same functions as other arrays, for example:

```

>> b = ones(4,4,2)
b(:,:,1) =
    1     1     1     1
    1     1     1     1
    1     1     1     1
    1     1     1     1
b(:,:,2) =
    1     1     1     1
    1     1     1     1
    1     1     1     1
    1     1     1     1
>> c = randn(2,2,3)
c(:,:,1) =
   -0.4326    0.1253
   -1.6656    0.2877

```

```
c(:,:,2) =
    -1.1465    1.1892
     1.1909   -0.0376
c(:,:,3) =
    0.3273   -0.1867
    0.1746    0.7258
```

The number of dimensions in a multidimensional array can be found using the `ndims` function, and the size of the array can be found using the `size` function.

```
>> ndims(c)
ans =
    3
>> size(c)
ans =
    2      2      3
```

If you are writing applications that need multidimensional arrays, see the *MATLAB Users' Guide* for more details on the behavior of various MATLAB functions with multidimensional arrays.

Good Programming Practice

Use multidimensional arrays to solve problems that are naturally multivariate in nature, such as aerodynamics and fluid flows.

6.4 Additional Two-Dimensional Plots

In previous chapters, we learned to create linear, log-log, semilog, and polar plots. MATLAB supports many additional types of plots that you can use to display your data. This section will introduce you to some of these additional plotting options.

6.4.1 Additional Types of Two-Dimensional Plots

In addition to the two-dimensional plots that we have already seen, MATLAB supports *many* other more specialized plots. In fact, the MATLAB Help Browser lists more than 20 types of two-dimensional plots! Examples include **stem plots**, **stair plots**, **bar plots**, **pie plots**, and **compass plots**. A *stem plot* is a plot in which each data value is represented by a marker and a line connecting the marker vertically to the *x* axis. A *stair plot* is a plot in which each data point is represented by a horizontal line, and successive points are connected by vertical lines, producing a stair-step effect. A *bar plot* is a plot in which each point is represented

by a vertical bar or horizontal bar. A *pie plot* is a plot represented by “pie slices” of various sizes. Finally, a *compass plot* is a type of polar plot in which each value is represented by an arrow whose length is proportional to its value. These types of plots are summarized in Table 6.3, and examples of all of the plots are shown in Figure 6.7.

Stair, stem, vertical bar, horizontal bar, and compass plots are all similar to plot, and they are used in the same manner. For example, the following code produces the stem plot shown in Figure 6.7a.

```
x = [ 1 2 3 4 5 6];
y = [ 2 6 8 7 8 5];
stem(x,y);
title('Example of a Stem Plot');
xlabel('itx');
ylabel('ity');
axis([0 7 0 10]);
```

Stair, bar, and compass plots can be created by substituting stairs, bar, barh, or compass for stem in the above code. The details of all of these plots, including any optional parameters, can be found in the MATLAB on-line help system.

Table 6.3 Additional Two-Dimensional Plotting Functions

Function	Description
bar(x,y)	This function creates a <i>vertical</i> bar plot, with the values in x used to label each bar and the values in y used to determine the height of the bar.
barh(x,y)	This function creates a <i>horizontal</i> bar plot, with the values in x used to label each bar and the values in y used to determine the horizontal length of the bar.
compass(x,y)	This function creates a polar plot, with an arrow drawn from the origin to the location of each (x,y) point. Note that the locations of the points to plot are specified in Cartesian coordinates, not polar coordinates.
pie(x) pie(x,explode)	This function creates a pie plot. This function determines the percentage of the total pie corresponding to each value of x and plots pie slices of that size. The optional array explode controls whether or not individual pie slices are separated from the remainder of the pie.
stairs(x,y)	This function creates a stair plot, with each stair step centered on an (x,y) point.
stem(x,y)	This function creates a stem plot, with a marker at each (x,y) point and a stem drawn vertically from that point to the x axis.

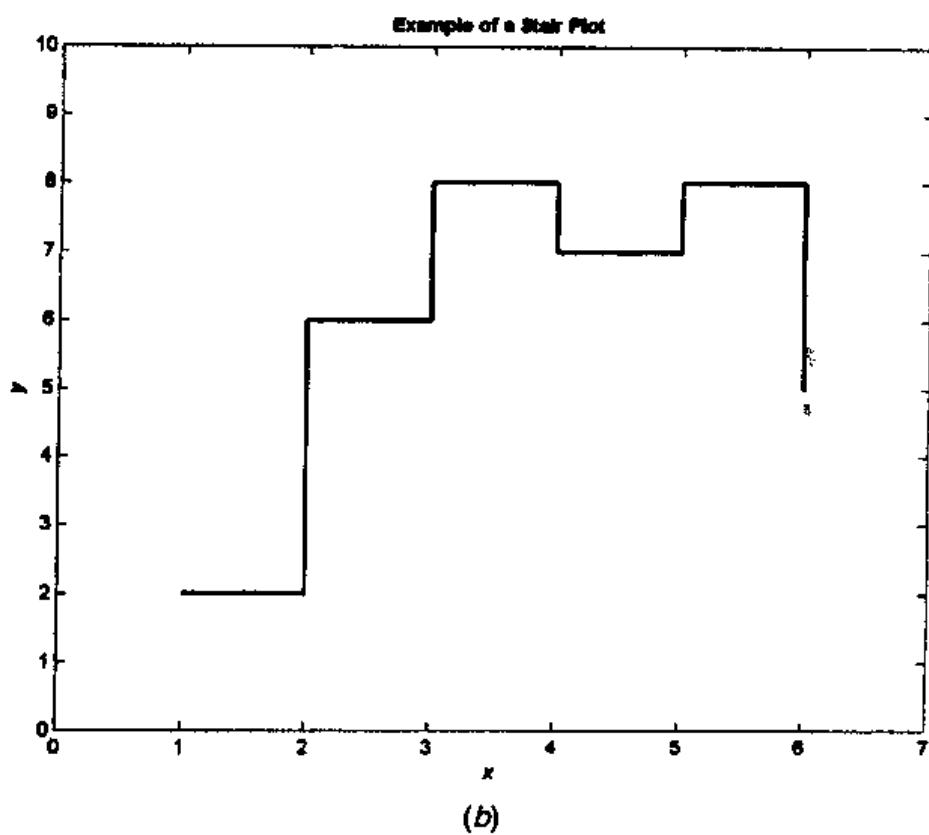
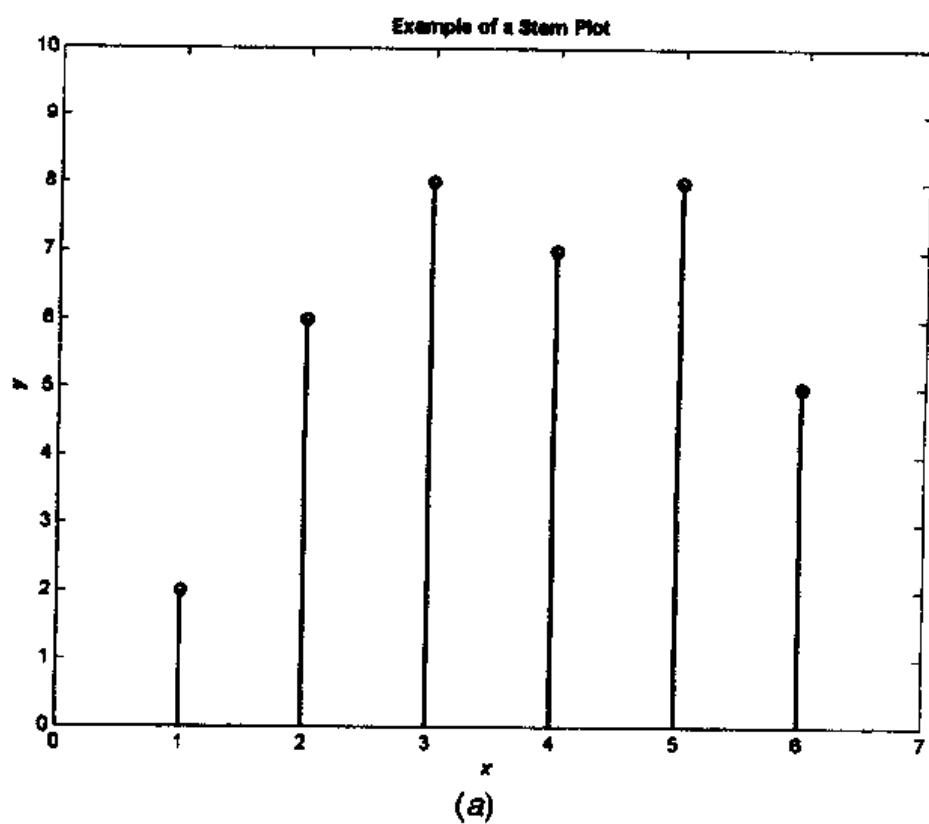


Figure 6.7 Additional types of 2D plots: (a) stem plot; (b) stair plot.

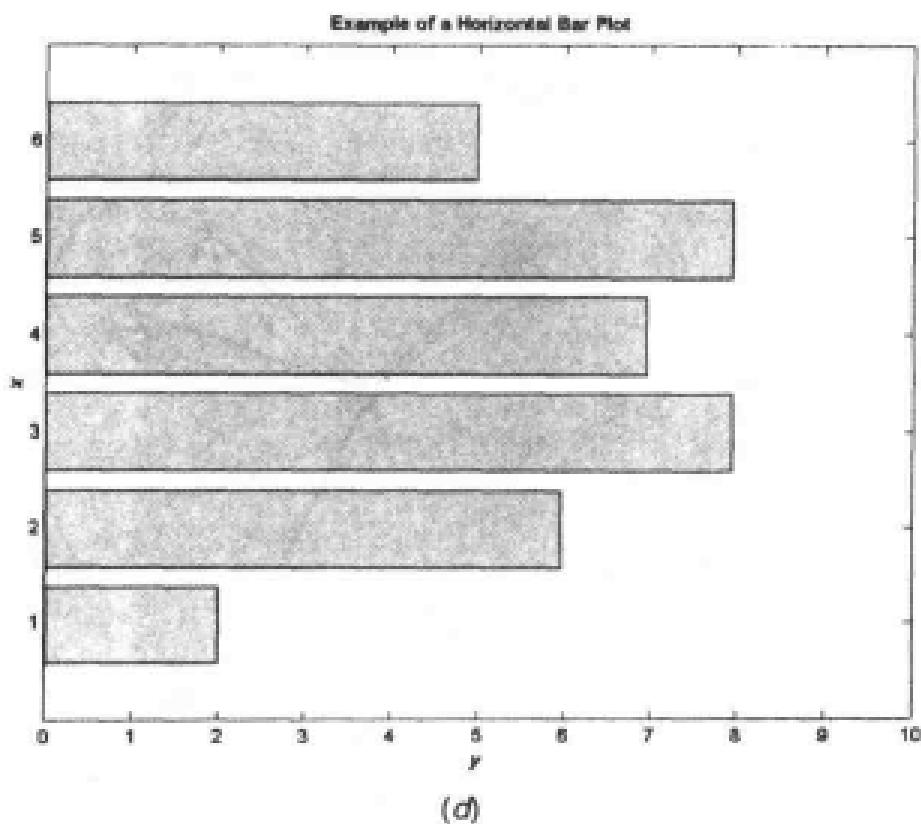
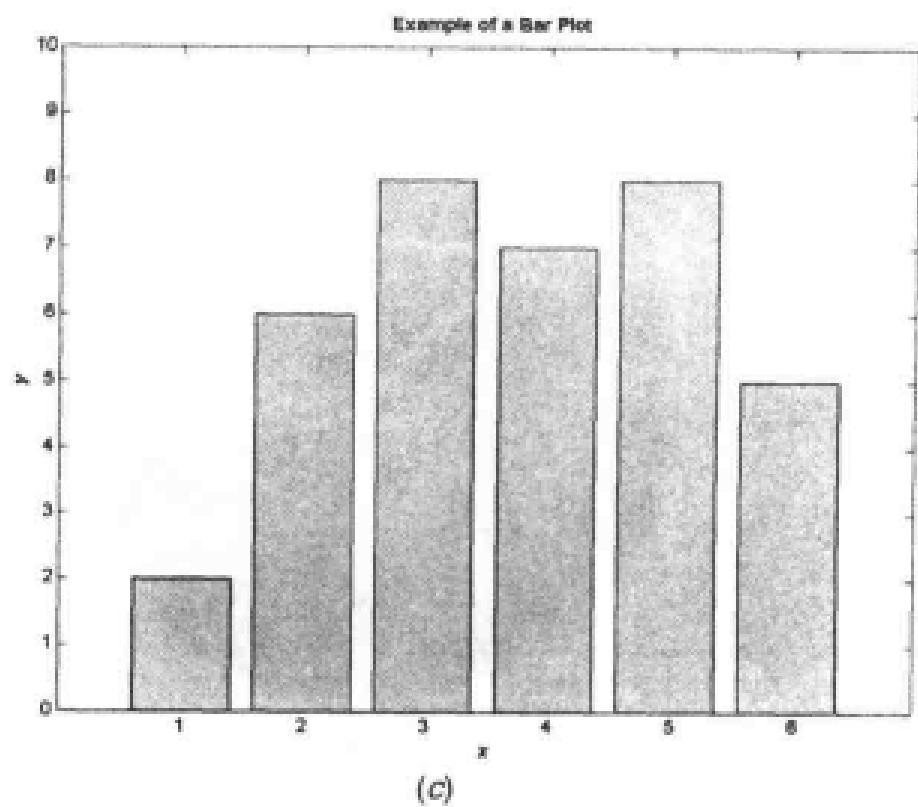


Figure 6.7 (Continued) Additional types of 2D plots: (c) vertical bar plot; (d) horizontal bar plot.

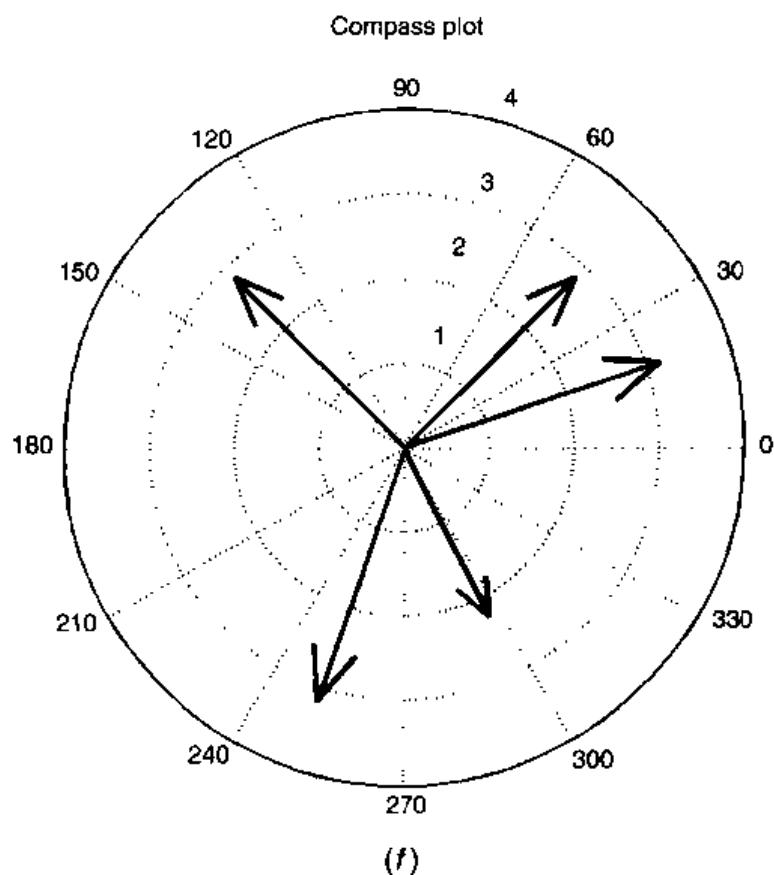
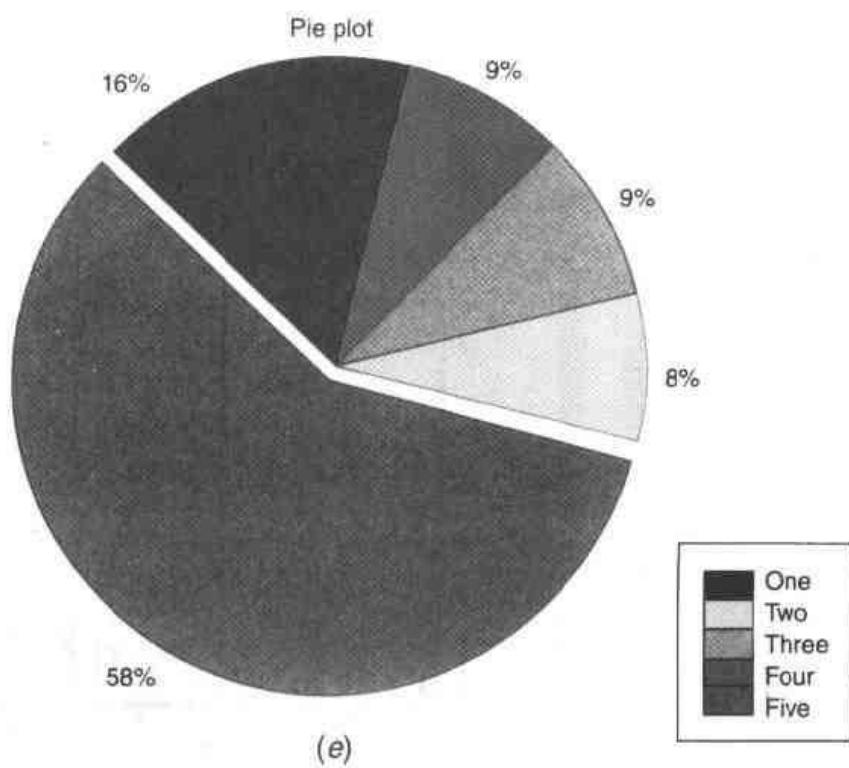


Figure 6.7 (Continued) Additional types of 2D plots: (e) pie plot; (f) compass plot.

Function `pie` behaves differently than the other plots previously described. To create a pie plot, a programmer passes an array `x` containing the data to be plotted, and function `pie` determines the *percentage of the total pie* that each element of `x` represents. For example, if the array `x` is [1 2 3 4], then `pie` will calculate that the first element `x(1)` is 1/10 or 10% of the pie, the second element `x(2)` is 2/10 or 20% of the pie, and so forth. The function then plots those percentages as pie slices.

Function `pie` also supports an optional parameter, `explode`. If present, `explode` is a logical array of 1's and 0's, with an element for each element in array `x`. If a value in `explode` is 1, then the corresponding pie slice is drawn slightly separated from the pie. For example, the following code produces the pie plot in Figure 6.7e. Note that the second slice of the pie is “exploded.”

```
data = [ 10 37 5 6 6];
explode = [ 0 1 0 0 0];
pie(data=explode);
title('Example of a Pie Plot');
legend('One','Two','Three','Four','Five');
```

6.4.2 Plotting Functions

In all previous plots, we have created arrays of data and passed those arrays to the plotting function. MATLAB also includes two functions that will plot a function directly, without the necessity of creating intermediate data arrays. These functions are `ezplot` and `fplot`.

Function `ezplot` takes one of the following forms.

```
ezplot( fun );
ezplot( fun, [xmin xmax] );
ezplot( fun, [xmin xmax], figure );
```

In each case, `fun` is a *character string* containing the functional expression to be evaluated. The optional parameter `[xmin xmax]` specifies the range of the function to plot. If it is absent, the function will be plotted between -2π and 2π . The optional parameter `figure` specifies the figure number to plot the function on.

For example, the following statements plot the function $f(x) = \sin x/x$ between -4π and 4π . The output of these statements is shown in Figure 6.8.

```
ezplot('sin(x)/x',[-4*pi 4*pi]);
title('Plot of sin x / x');
grid on;
```

Function `fplot` is similar to but more sophisticated than `ezplot`. The first two arguments are the same for both functions, but `fplot` has the following advantages.

1. Function `fplot` is *adaptive*, meaning that it calculates and displays more data points in the regions where the function being plotted is changing

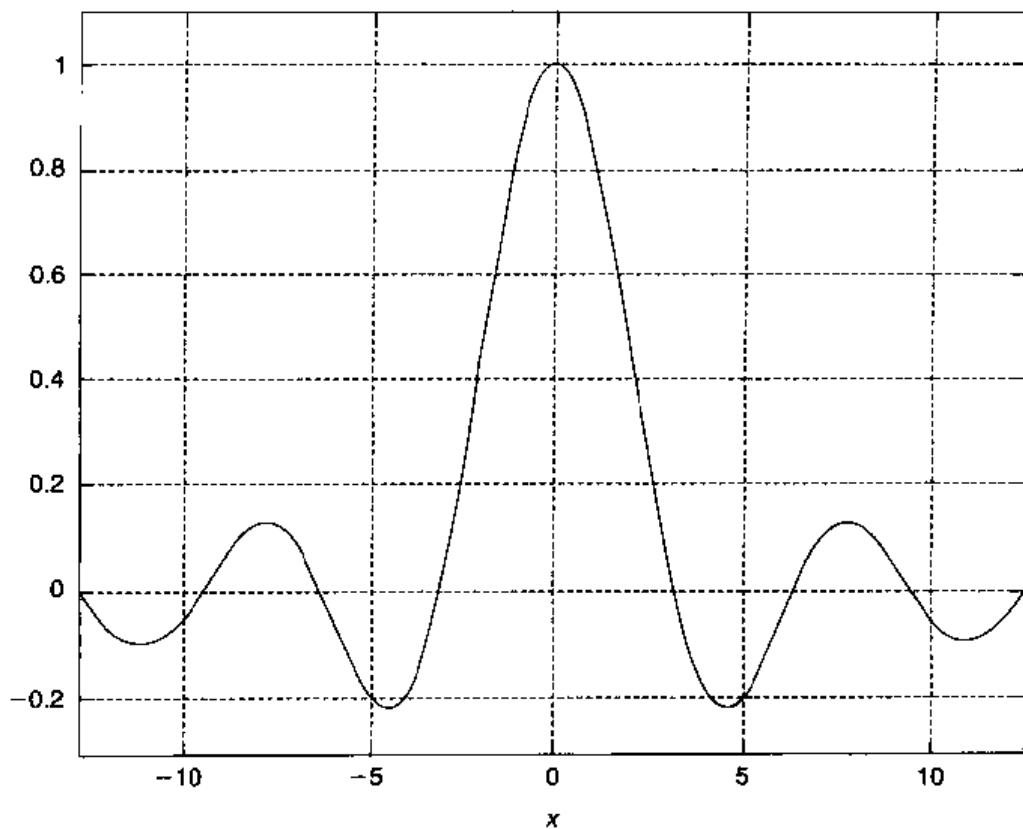


Figure 6.8 The function $\sin x/x$, plotted with function `ezplot`.

most rapidly. The resulting plot is more accurate at locations where a function's behavior changes suddenly.

2. Function `fplot` supports the use of \TeX commands in titles and axis labels, while function `ezplot` does not.

In general, you should use `fplot` in preference to `ezplot` whenever you plot functions.

Functions `ezplot` and `fplot` are examples of the “function functions” described in Chapter 5.

Good Programming Practice

Use function `fplot` to plot functions directly without having to create intermediate data arrays.

6.4.3 Histograms

A *histogram* is a plot showing the distribution of values within a data set. To create a histogram, the range of values within the data set is divided into evenly

spaced bins, and the number of data values falling into each bin is determined. The resulting count can then be plotted as a function of bin number.

The standard MATLAB histogram function is `hist`. The forms of this function follow.

```
hist(y)
hist(y,nbins)
hist(y,x);
[n,xout] = hist(y,...)
```

The first form of the function creates and plots a histogram with 10 equally spaced bins, while the second form creates and plots a histogram with `nbins` equally spaced bins. The third form of the function allows the user to specify the bin centers to use in an array `x`; the function creates a bin centered around each element in the array. In all three of these cases, the function both creates and plots the histogram. The last form of the function creates a histogram and returns the bin centers in array `xout` and the count in each bin in array `n`, without actually creating a plot.

For example, the following statements create a data set containing 10,000 Gaussian random values and generate a histogram of the data using 15 evenly spaced bins. The resulting histogram is shown in Figure 6.9.

```
y = randn(10000,1);
hist(y,15);
```

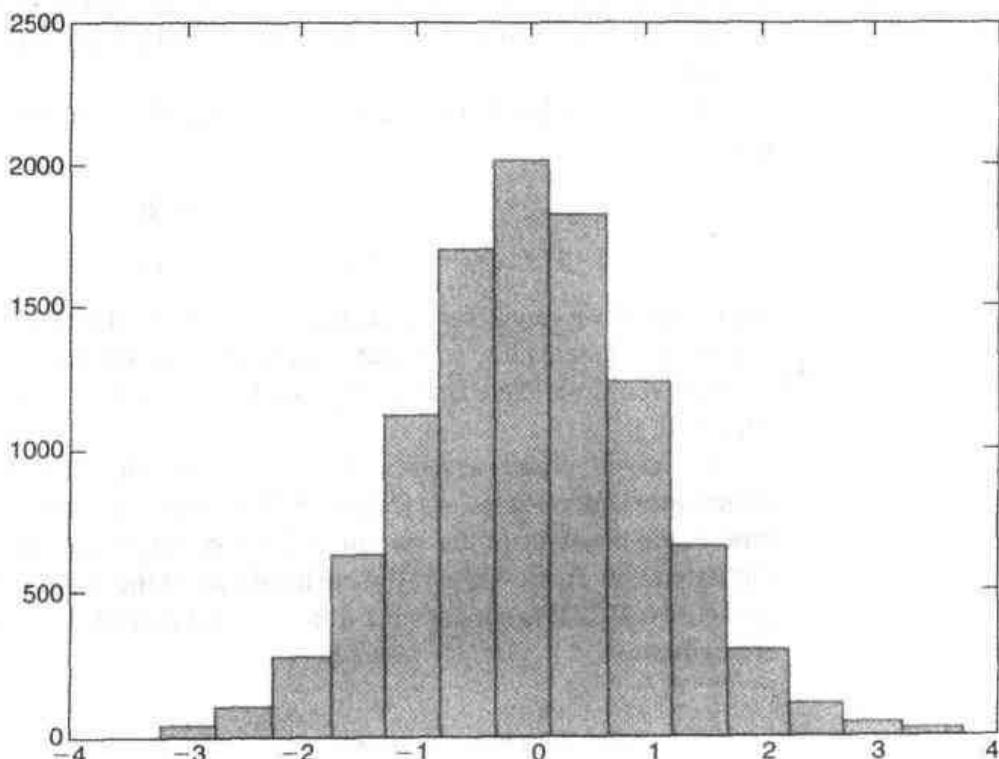


Figure 6.9 A histogram.

MATLAB also includes a function `rose` to create and plot a histogram on radial axes. It is especially useful for distributions of angular data. You will be asked to use this function in an end-of-chapter exercise.

6.5 Three-Dimensional Plots

MATLAB also includes a rich variety of three-dimensional plots that can be useful for displaying certain types of data. In general, three-dimensional plots are useful for displaying two types of data.

1. Two variables that are functions of the same independent variable, when you wish to emphasize the importance of the independent variable.
2. A single variable that is a function of two independent variables.

6.5.1 Three-Dimensional Line Plots

A three-dimensional line plot can be created with the `plot3` function. This function is exactly like the two-dimensional `plot` function, except that each point is represented by x , y , and z values instead of just x and y values. The simplest form of this function is

```
plot(x,y,z);
```

where x , y , and z are equal-sized arrays containing the locations of data points to plot. Function `plot3` supports all the same line size, line style, and color options as `plot`, and you can use it immediately using the knowledge acquired in earlier chapters.

As an example of a three-dimensional line plot, consider the following functions.

$$\begin{aligned}x(t) &= e^{-0.2t} \cos 2t \\y(t) &= e^{-0.2t} \sin 2t\end{aligned}\tag{6.11}$$

These functions might represent the decaying oscillations of a mechanical system in two dimensions, so x and y together represent the location of the system at any given time. Note that x and y are both functions of the *same* independent variable t .

We could create a series of (x,y) points and plot them using the two-dimensional function `plot` (Figure 6.10a), but if we do so, the importance of time to the behavior of the system will not be obvious in the graph. The following statements create the two-dimensional plot of the location of the object shown in Figure 6.10a. It is not possible from this plot to tell how rapidly the oscillations are dying out.

```
t = 0:0.1:10;
x = exp(-0.2*t) .* cos(2*t);
y = exp(-0.2*t) .* sin(2*t);
```

```

plot(x,y);
title('Two-Dimensional Line Plot');
xlabel('x');
ylabel('y');
grid on;

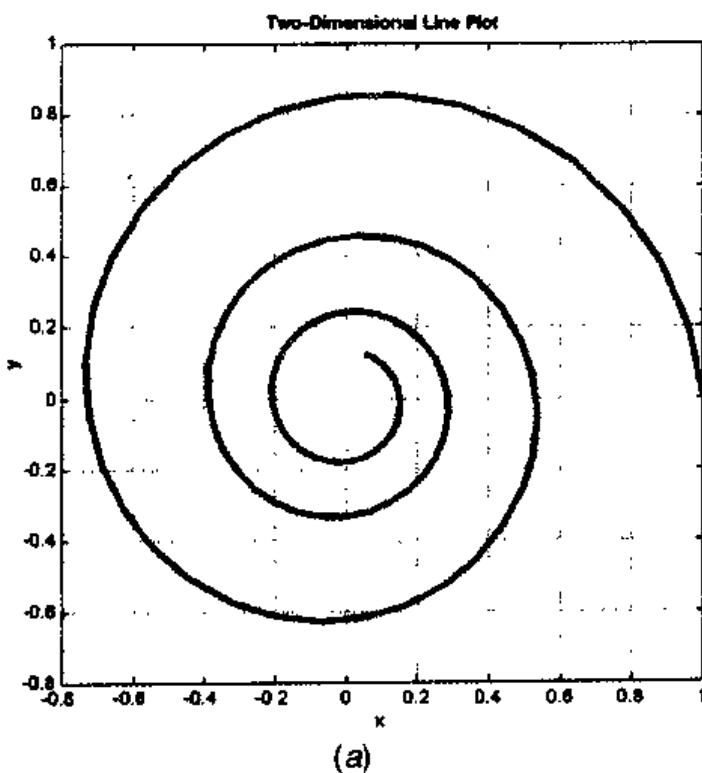
```

Instead, we could plot the variables with `plot3` to preserve the time information as well as the two-dimensional position of the object. The following statements will create a three-dimensional plot of Equations 6.11.

```

t = 0:0.1:10;
x = exp(-0.2*t) .* cos(2*t);
y = exp(-0.2*t) .* sin(2*t);
plot3(x,y,t);
title('Three-Dimensional Line Plot');
xlabel('x');
ylabel('y');
zlabel('time');
axis square;
grid on;

```



(a)

Figure 6.10 (a) A two-dimensional line plot showing the motion in (x,y) space of a mechanical system. This plot reveals nothing about the time behavior of the system.
 (b) A three-dimensional line plot showing the motion in (x,y) space versus time for the mechanical system. This plot clearly shows the time behavior of the system.

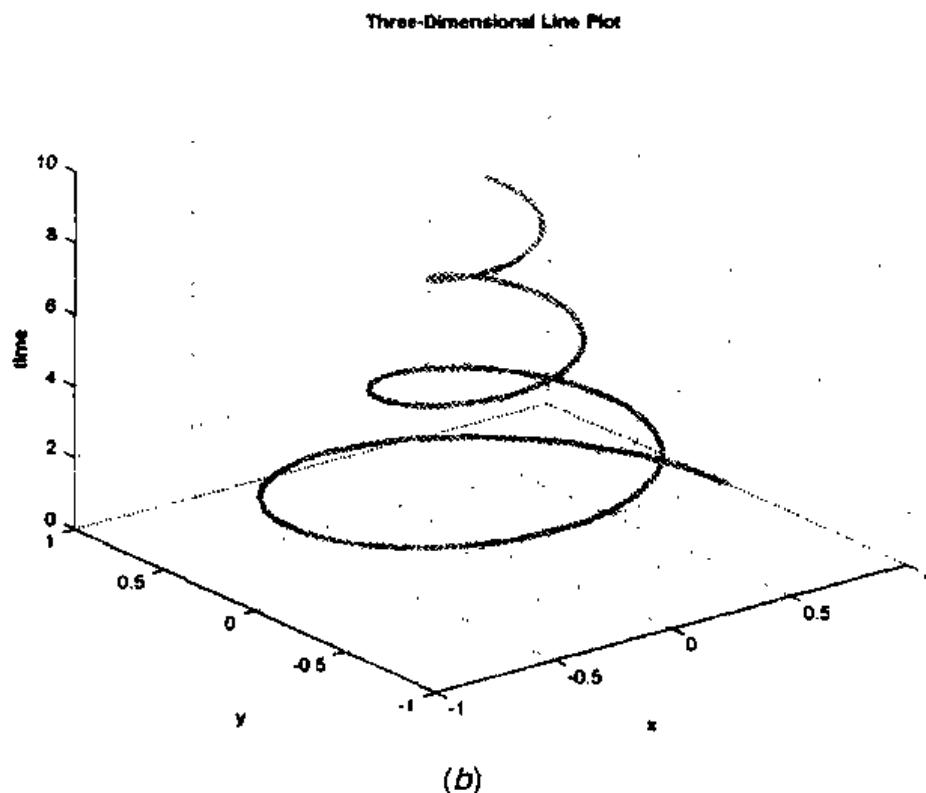


Figure 6.10 (Continued)

The resulting plot is shown in Figure 6.10b. Note how this plot emphasizes time-dependence of the two variables x and y .

6.5.2 Three-Dimensional Surface, Mesh, and Contour Plots

Surface, mesh, and contour plots are convenient ways to represent data that is a function of *two* independent variables. For example, the temperature at a point is a function of both the East–West location (x) and the North–South (y) location of the point. Any value that is a function of two independent variables can be displayed on a three-dimensional surface, mesh, or contour plot. The more common types of plots are summarized in Table 6.4, and examples of each plot are shown in Figure 6.11.²

To plot data using one of these functions, a user must create three equal-sized arrays. The three arrays must contain the x , y , and z values of every point to be plotted. As a simple example, suppose that we wanted to plot the four points $(-1, -1, 1)$, $(1, -1, 2)$, $(-1, 1, 1)$, and $(1, 1, 0)$. To plot these four points, we must create the arrays $\mathbf{x} = \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$, $\mathbf{y} = \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix}$, and $\mathbf{z} = \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix}$. Array \mathbf{x} con-

² There are many variations on these basic plot types. Consult the MATLAB Help Browser documentation for a complete description of these variations.

Table 6.4 Selected Mesh, Surface, and Contour Plot Functions

Function	Description
<code>mesh(x, y, z)</code>	This function creates a mesh or wireframe plot, where x is a two-dimensional array containing the x values of every point to display, y is a two-dimensional array containing the y values of every point to display, and z is a two-dimensional array containing the z values of every point to display.
<code>surf(x, y, z)</code>	This function creates a surface plot. Arrays x , y , and z have the same meaning as for a mesh plot.
<code>contour(x, y, z)</code>	This function creates a contour plot. Arrays x , y , and z have the same meaning as for a mesh plot.

tains the x values associated with every point to plot, array y contains the y values associated with every point to plot, and array z contains the z values associated with every point to plot. These arrays are then passed to the plotting function.

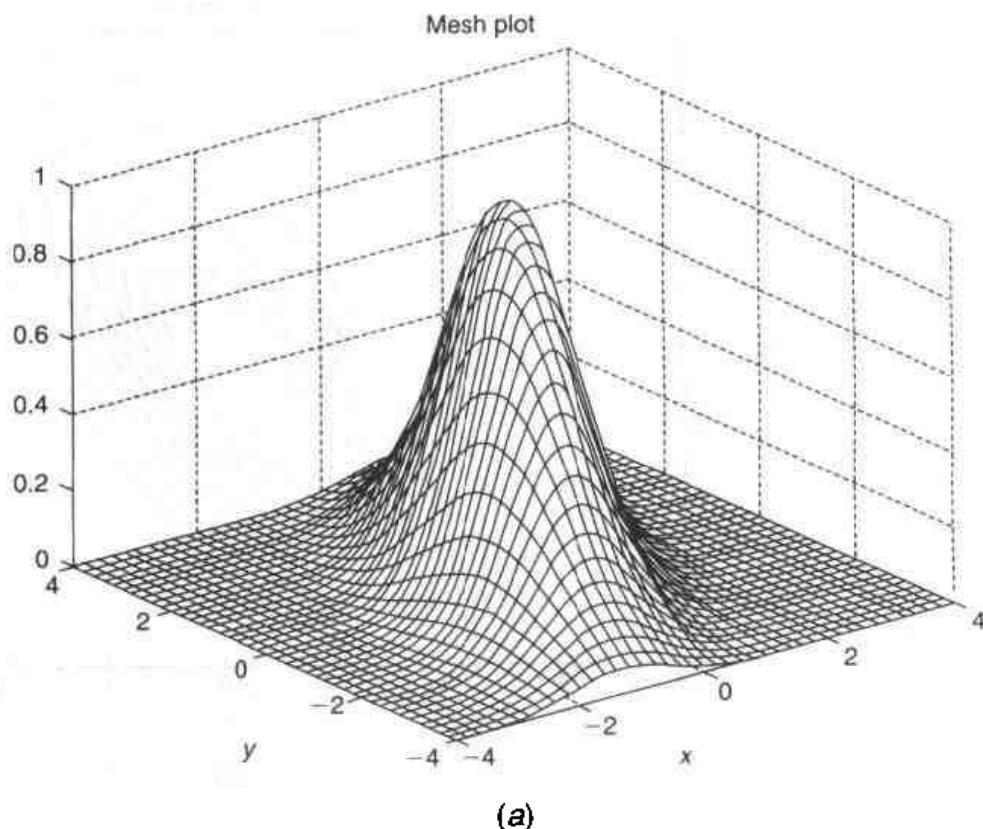


Figure 6.11 (a) A mesh plot of the function $z(x,y) = e^{-0.5[x^2 + 0.5(y-1)^2]}$. A color version of this plot appears in the insert.

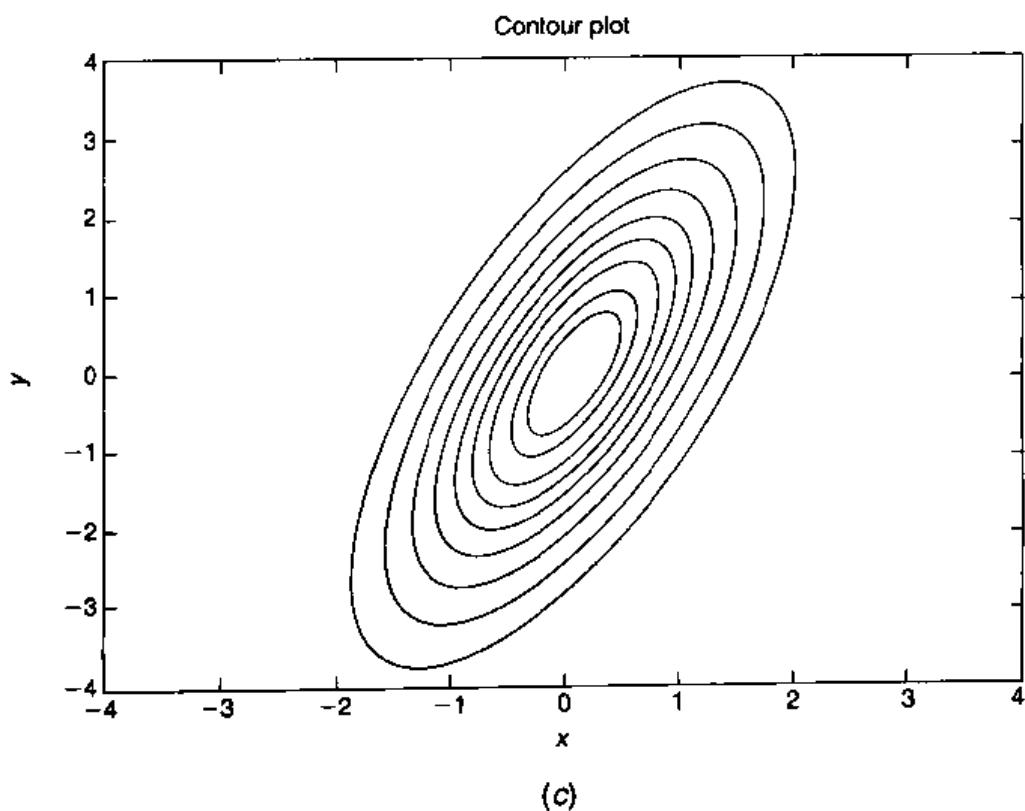
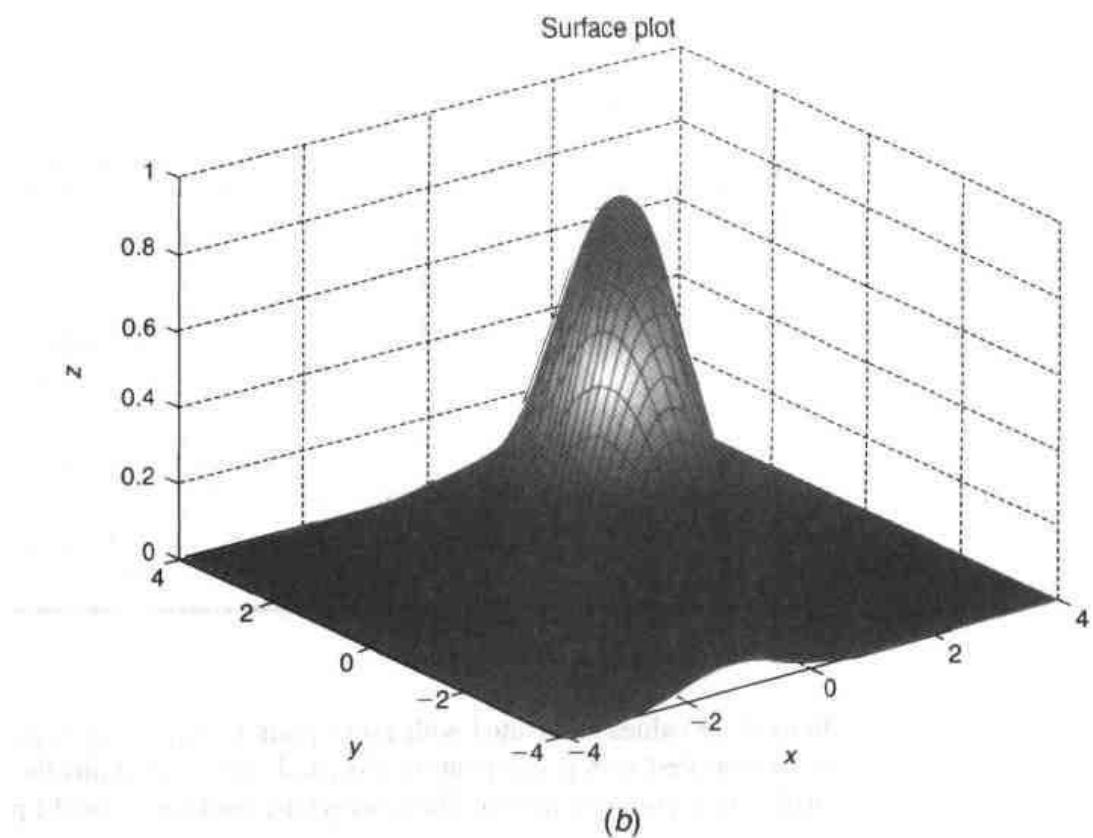


Figure 6.11 (Continued) (b) A surface plot of the same function. (c) A contour plot of the same function. Color version of these plots appears in the insert.

The MATLAB function `meshgrid` makes it easy to create the `x` and `y` arrays required for these plots. The form of this function is

```
[x y] = meshgrid( xstart:xinc:xend, ystart:yinc:yend);
```

where `xstart:xinc:xend` specifies the `x` values to include in the grid, and `ystart:yinc:yend` specifies the `y` values to be included in the grid.

To create a plot, we use `meshgrid` to create the arrays of `x` and `y` values, and then evaluate the function to plot at each of those (x,y) locations. Finally, we call function `mesh`, `surf`, or `contour` to create the plot.

For example, suppose that we wish to create a mesh plot of the function

$$z(x,y) = e^{-0.5(x^2+0.5(y-v)^2)} \quad (6.12)$$

over the interval $-4 \leq x \leq 4$ and $-4 \leq y \leq 4$. The following statements will create the plot, which is shown in Figure 6.11a.

```
[x,y] = meshgrid(-4:0.2:4, -4:0.2:4);
z = exp(-0.5*(x.^2+y.^2));
mesh(x,y,z);
xlabel('x');
ylabel('y');
zlabel('z');
```

Surface and contour plots can be created by substituting the appropriate function for the `mesh` function.

6.6 Summary

MATLAB supports complex numbers as an extension of the `double` data type. They can be defined using the `i` or `j`, both of which are predefined to be $\sqrt{-1}$. Using complex numbers is straightforward, except that the relational operators `>`, `>=`, `<`, and `<=` only compare the *real parts* of complex numbers, not their magnitudes. They must be used with caution when working with complex values.

String functions are functions designed to work with strings, which are arrays of type `char`. These functions allow a user to manipulate strings in a variety of useful ways, including concatenation, comparison, replacement, case conversion, and numeric-to-string and string-to-numeric type conversions.

Multidimensional arrays are arrays with more than two dimensions. They may be created and used in a fashion similar to one- and two-dimensional arrays. Multidimensional arrays appear naturally in certain classes of physical problems.

MATLAB includes a rich variety of two- and three-dimensional plots. In this chapter, we introduced `stem`, `stair`, `bar`, `compass`, `mesh`, `surface`, and `contour` plots.

6.6.1 Summary of Good Programming Practice

The following guidelines should be followed.

1. Use the `char` function to create two-dimensional character arrays without worrying about padding each row to the same length.

2. Use multidimensional arrays to solve problems that are naturally multivariate in nature, such as aerodynamics and fluid flows.
3. Use function `fplot` to plot functions directly without having to create intermediate data arrays.

6.6.2 MATLAB Summary

The following summary lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one.

Commands and Functions

<code>abs</code>	Returns absolute value (magnitude) of a number.
<code>angle</code>	Returns the angle of a complex number, in radians.
<code>bar(x,y)</code>	Create a vertical bar plot.
<code>barh(x,y)</code>	Create a horizontal bar plot.
<code>base2dec</code>	Convert base B string to decimal integer.
<code>bin2dec</code>	Convert binary string to decimal integer.
<code>blanks</code>	Create a string of blanks.
<code>char</code>	(1) Convert numbers to the corresponding character values. (2) Create a 2-D character array from a series of strings.
<code>compass(x,y)</code>	Create a compass plot.
<code>conj</code>	Compute complex conjugate of a number.
<code>contour</code>	Create a contour plot.
<code>deblank</code>	Remove trailing blanks.
<code>dec2base</code>	Convert decimal integer to base B string.
<code>dec2bin</code>	Convert decimal integer to binary string.
<code>double</code>	Convert characters to the corresponding numeric codes.
<code>find</code>	Find indices and values of non-zero elements in a matrix.
<code>findstr</code>	Find one string within another one.
<code>hex2num</code>	Convert IEEE hexadecimal string to double.
<code>hex2dec</code>	Convert hexadecimal string to decimal integer.
<code>hist</code>	Create a histogram of a data set.
<code>imag</code>	Returns the imaginary portion of the complex number.
<code>int2str</code>	Convert integer to string.
<code>ischar</code>	Returns true (1) for a character array.
<code>isletter</code>	Returns true (1) for letters of the alphabet.
<code>isreal</code>	Returns true (!) if no element of array has an imaginary component.
<code>isspace</code>	Returns true (1) for whitespace.
<code>lower</code>	Convert string to lowercase.
<code>mat2str</code>	Convert matrix to string.
<code>mesh</code>	Create a mesh plot.

<code>meshgrid</code>	Create the (x,y) grid required for mesh, surface, and contour plots.
<code>nnz</code>	Number of non-zero matrix elements.
<code>nonzeros</code>	Return a column vector containing the non-zero elements in a matrix.
<code>num2str</code>	Convert number to string.
<code>nzmax</code>	Amount of storage allocated for non-zero matrix elements.
<code>pie(x)</code>	Create a pie plot.
<code>plot(c)</code>	Plots the real versus the imaginary part of a complex array.
<code>real</code>	Returns the real portion of the complex number.
<code>rose</code>	Create a radial histogram of a data set.
<code>sscanf</code>	Read formatted data from string.
<code>stairs(x,y)</code>	Create a stair plot.
<code>stem(x,y)</code>	Create a stem plot.
<code>str2num</code>	Convert string to number.
<code>strcat</code>	Concatenate strings.
<code>strcmp</code>	Returns true (1) if two strings are identical.
<code>strcmpi</code>	Returns true (1) if two strings are identical, ignoring case.
<code>strfind</code>	Find one string within another one (MATLAB Version 6.1 or later only).
<code>strjust</code>	Justify string.
<code>strncmp</code>	Returns true (1) if first n characters of two strings are identical.
<code>strncmpi</code>	Returns true (1) if first n characters of two strings are identical, ignoring case.
<code>strmatch</code>	Find matches for string.
<code>strrep</code>	Replace one string with another.
<code>strtok</code>	Find token in string.
<code>struct</code>	Pre-define a structure array.
<code>strvcat</code>	Concatenate strings vertically.
<code>surf</code>	Create a surface plot.
<code>upper</code>	Convert string to uppercase.

6.7 Exercises

- 6.1** Figure 6.12 shows a series *RLC* circuit driven by a sinusoidal AC voltage source whose value is $120\angle0^\circ$ volts. The impedance of the inductor in this circuit is $Z_L = j2\pi fL$, where j is $\sqrt{-1}$, f is the frequency of the voltage source in hertz, and L is the inductance in henrys. The impedance of the capacitor in this circuit is $Z_C = -j\frac{1}{2\pi fC}$, where C is the capacitance in farads. Assume that $R = 100 \Omega$, $L = 0.1$ mH, and $C = 0.25$ nF.

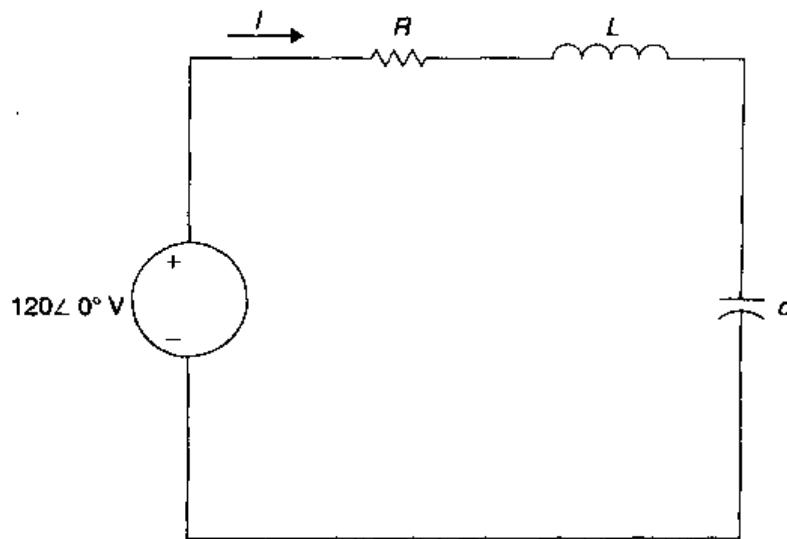


Figure 6.12 A series RLC circuit driven by a sinusoidal ac voltage source.

The current I flowing in this circuit is given by Kirchhoff's Voltage Law to be

$$I = \frac{120\angle 0^\circ \text{ V}}{R + j2\pi fL - j\frac{1}{2\pi fC}}$$

- a. Calculate and plot the magnitude of this current as a function of frequency as the frequency changes from 100 kHz to 10 MHz. Plot this information on both a linear and a log-linear scale. Be sure to include a title and axis labels.
- b. Calculate and plot the phase angle in degrees of this current as a function of frequency as the frequency changes from 100 kHz to 10 MHz. Plot this information on both a linear and a log-linear scale. Be sure to include a title and axis labels.
- c. Plot both the magnitude and phase angle of the current as a function of frequency on two sub-plots of a single figure. Use log-linear scales.
- 6.2 Write a function `to_polar` that accepts a complex number c , and returns two output arguments containing the magnitude `mag` and angle `theta` of the complex number. The output angle should be in degrees.
- 6.3 Write a function `to_complex` that accepts two input arguments containing the magnitude `mag` and angle `theta` of the complex number in degrees, and returns the complex number c .
- 6.4 In a sinusoidal steady-state AC circuit, the voltage across a passive element is given by Ohm's law:

$$V = IZ \quad (6.13)$$

where V is the voltage across the element, I is the current through the element, and Z is the impedance of the element. Note that all three of these values are complex, and that these complex numbers are usually specified in the form of a magnitude at a specific phase angle expressed in degrees. For example, the voltage might be $V = 120\angle 30^\circ V$.

Write a program that reads the voltage across an element and the impedance of the element, and calculates the resulting current flow. The input values should be given as magnitudes and angles expressed in degrees, and the resulting answer should be in the same form. Use the function `to_complex` from Exercise 6.3 to convert the numbers to rectangular for the actual computation of the current, and the function `to_polar` from Exercise 6.2 to convert the answer into polar form for display (see Figure 6.13).

- 6.5** Write a function that will accept a complex number c and plot that point on a Cartesian coordinate system with a circular marker. The plot should include both the x and y axes, plus a vector drawn from the origin to the location of c .
- 6.6** Plot the function $v(t) = 10e^{(-0.2+j\pi)t}$ for $0 \leq t \leq 10$ using the function `plot(t, v)`. What is displayed on the plot?
- 6.7** Plot the function $v(t) = 10e^{(-0.2+j\pi)t}$ for $0 \leq t \leq 10$ using the function `plot(v)`. What is displayed on the plot this time?
- 6.8** Create a polar plot of the function $v(t) = 10e^{(-0.2+j\pi)t}$ for $0 \leq t \leq 10$.
- 6.9** Plot of the function $v(t) = 10e^{(-0.2+j\pi)t}$ for $0 \leq t \leq 10$ using function `plot3`, where the three dimensions to plot are the real part of the function, the imaginary part of the function, and time.

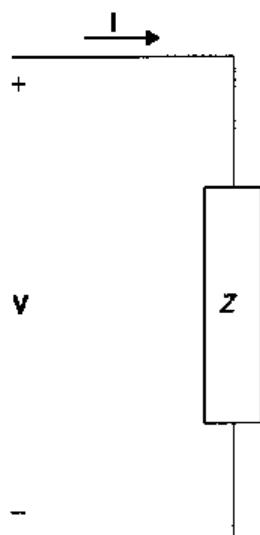


Figure 6.13 The voltage and current relationship on a passive ac circuit element.

- 6.10 Euler's Equation** Euler's equation defines e raised to an imaginary power in terms of sinusoidal functions as follows

$$e^{i\theta} = \cos \theta + i \sin \theta \quad (6.14)$$

Create a two-dimensional plot of this function as θ varies from 0 to 2π . Create a three-dimensional line plot using function `plot3` as θ varies from 0 to 2π (the three dimensions are the real part of the expression, the imaginary part of the expression, and θ).

- 6.11** Create a mesh plot, surface plot, and contour plot of the function $z = e^{x+iy}$ for the interval $-1 \leq x \leq 1$ and $-2\pi \leq y \leq 2\pi$. In each case, plot the real part of z versus x and y .
- 6.12** Write a program that accepts an input string from the user and determines how many times a user-specified character appears within the string. (*Hint:* Look up the 's' option of the `input` function using the MATLAB Help Browser.)
- 6.13** Modify the previous program so that it determines how many times a user-specified character appears within the string without regard to the case of the character.
- 6.14** Write a program that accepts a string from a user with the `input` function, chops that string into a series of tokens, sorts the tokens into ascending order, and prints them out.
- 6.15** Write a program that accepts a series of strings from a user with the `input` function, sorts the strings into ascending order, and prints them out.
- 6.16** Write a program that accepts a series of strings from a user with the `input` function, sorts the strings into ascending order disregarding case, and prints them out.
- 6.17** MATLAB includes functions `upper` and `lower`, which shift a string to uppercase and lowercase respectively. Create a new function called `caps`, which capitalizes the first letter in each word, and forces all other letters to be lowercase.
- 6.18** Plot the function $y = e^{-x} \sin x$ for x between 0 and 2 in steps of 0.1. Create the following plot types: (a) stem plot; (b) stair plot; (c) bar plot; (d) compass plot. Be sure to include titles and axis labels on all plots.
- 6.19** Suppose that George, Sam, Betty, Charlie, and Suzie contributed \$5, \$10, \$7, \$5, and \$15 respectively to a colleague's going-away present. Create a pie chart of their contributions. What percentage of the gift was paid for by Sam?
- 6.20** Plot the function $f(x) = 1/\sqrt{x}$ over the range $0.1 \leq x \leq 10.0$ using function `fplot`. Be sure to label your plot properly.

Sparse Arrays, Cell Arrays, and Structures

This chapter deals with three additional data types: sparse arrays, cell arrays, and structures. Sparse arrays are a special type of array in which memory is only allocated for the nonzero elements in the array. Cell arrays are arrays, each element of which can hold any type of MATLAB data. They are used extensively in MATLAB Graphical User Interface (GUI) functions. Finally, structures provide a way to store different types of data in a single variable, with a separate name for each data item within the variable.

7.1 Sparse Arrays

We learned about ordinary MATLAB arrays in Chapter 2. When an ordinary array is declared, MATLAB creates a memory location for every element in the array. For example, the function `a = eye(10)` creates 100 elements arranged as a 10×10 structure, with the diagonal terms set to 1 and all other terms set to 0. Note that 90 of those elements in the array are zero! This matrix requires 100 elements, but only 10 of them contain nonzero data. This is an example of a **sparse array** or **sparse matrix**. A sparse matrix is a large matrix in which the vast majority of the elements are zero.

```
> a = 2 * eye(10);  
a =  
    2     0     0     0     0     0     0     0     0     0  
    0     2     0     0     0     0     0     0     0     0  
    0     0     2     0     0     0     0     0     0     0  
    0     0     0     2     0     0     0     0     0     0  
    0     0     0     0     2     0     0     0     0     0  
    0     0     0     0     0     2     0     0     0     0  
    0     0     0     0     0     0     2     0     0     0  
    0     0     0     0     0     0     0     2     0     0  
    0     0     0     0     0     0     0     0     2     0  
    0     0     0     0     0     0     0     0     0     2
```

Now suppose that we create another 10×10 matrix **b** defined as follows.

```
b =
 1   0   0   0   0   0   0   0   0   0
 0   2   0   0   0   0   0   0   0   0
 0   0   2   0   0   0   0   0   0   0
 0   0   0   1   0   0   0   0   0   0
 0   0   0   0   5   0   0   0   0   0
 0   0   0   0   0   1   0   0   0   0
 0   0   0   0   0   0   1   0   0   0
 0   0   0   0   0   0   0   1   0   0
 0   0   0   0   0   0   0   0   1   0
 0   0   0   0   0   0   0   0   0   1
```

If these two matrices are multiplied together, the result is

```
> c = a * b
c =
 2   0   0   0   0   0   0   0   0   0
 0   4   0   0   0   0   0   0   0   0
 0   0   4   0   0   0   0   0   0   0
 0   0   0   2   0   0   0   0   0   0
 0   0   0   0   10  0   0   0   0   0
 0   0   0   0   0   2   0   0   0   0
 0   0   0   0   0   0   2   0   0   0
 0   0   0   0   0   0   0   2   0   0
 0   0   0   0   0   0   0   0   2   0
 0   0   0   0   0   0   0   0   0   2
```

The process of multiplying these two sparse matrices together requires 1900 multiplications and additions, but most of the terms being added and multiplied are zeros, so it was largely wasted effort.

This problem gets worse rapidly as matrix size increases. For example, suppose that we were to generate two 200×200 sparse matrices **a** and **b** as follows.

```
a = 5 * eye(200);
b = 3 * eye(200);
```

Each matrix now contains 20,000 elements, of which 19,800 are zero! Furthermore, multiplying these two matrices together requires 7,980,000 additions and multiplications!

It should be apparent that storing and working with large sparse matrices, most of whose elements are zero, is a serious waste of both computer

memory and CPU time. Unfortunately, many real-world problems naturally create sparse matrices, so we need some efficient way to solve problems involving them.

A large electric power system is an excellent example of a real-world problem involving sparse matrices. Large electric power systems can have a thousand or more electrical busses at generating plants and transmission and distribution substations. If we wish to know the voltages, currents, and power flows in the system, we must first solve for the voltage at every bus. For a 1000-bus system, this involves the simultaneous solution of 1000 equations in 1000 unknowns, which is equivalent to inverting a matrix with 1,000,000 elements. Solving this matrix requires millions of floating point operations!

However, each bus in the power system is probably connected to an average of only three other busses, so 996 of the 1000 terms in each row of the matrix will be zeros, and most of the operations involved in inverting the matrix will be additions and multiplications by zeros. The calculation of the voltages and currents in this power system would be much simpler and more efficient if the zeros could be ignored in the solution process.

7.1.1 The **sparse** Data Type

MATLAB has a special data type designed to work with sparse arrays. The **sparse** data type differs from the **double** data type in that only the nonzero elements of an array are allocated memory locations. The **sparse** data type actually saves three values for each nonzero element: the value of the element and the row and column numbers where the element is located. Even though three values must be saved for each nonzero element, this approach is much more memory efficient than allocating full arrays for sparse matrices.

To illustrate the use of sparse matrices, let's create a 10×10 identity matrix.

```
> a = eye(10)
a =
    1   0   0   0   0   0   0   0   0   0
    0   1   0   0   0   0   0   0   0   0
    0   0   1   0   0   0   0   0   0   0
    0   0   0   1   0   0   0   0   0   0
    0   0   0   0   1   0   0   0   0   0
    0   0   0   0   0   1   0   0   0   0
    0   0   0   0   0   0   1   0   0   0
    0   0   0   0   0   0   0   1   0   0
    0   0   0   0   0   0   0   0   1   0
    0   0   0   0   0   0   0   0   0   1
```

If this matrix is converted to a sparse matrix using function **sparse**, the results are

```
> as = sparse(a)
as =
(1,1)      1
(2,2)      1
(3,3)      1
(4,4)      1
(5,5)      1
(6,6)      1
(7,7)      1
(8,8)      1
(9,9)      1
(10,10)    1
```

Note that the data in the sparse matrix is a list of row and column addresses, followed by the nonzero data value at that point. This is a very efficient way to store data as long as most of the matrix is zero, but if there are many nonzero elements, it can take up even more space than the full matrix because of the need to store the addresses.

The function `issparse` can be used to determine whether or not a given array is sparse. If an array is sparse, then `issparse(array)` returns a 1.

The power of the sparse data type can be seen by considering a 1000×1000 matrix z with an average of 4 nonzero elements per row. If this matrix is stored as a full matrix, it will require 8,000,000 bytes of space. On the other hand, if it is converted to a sparse matrix, the memory usage will drop dramatically.

```
> zs=sparse(z);
> whos
  Name      Size           Bytes  Class
  z         1000x1000     8000000  double array
  zs        1000x1000      51188  sparse array
Grand total is 1003932 elements using 8051188 bytes
```

7.1.1.1 Generating Sparse Matrices

MATLAB can generate sparse matrices by converting a full matrix into a sparse matrix with the `sparse` function, or by directly generating sparse matrices with the MATLAB functions `speye`, `sprand`, and `sprandn`, which are the sparse equivalents of `eye`, `rand`, and `randn`. For example, the expression `a = speye(4)` generates a 4×4 sparse matrix.

```
> a = speye(4)
a =
(1,1)      1
(2,2)      1
(3,3)      1
(4,4)      1
```

The expression `b = full(a)` converts the sparse matrix into a full matrix.

```
> b = full(a)
b =
```

```

1     0     0     0
0     1     0     0
0     0     1     0
0     0     0     1

```

7.1.1.2 Working with Sparse Matrices

Once a matrix is sparse, individual elements can be added to it or deleted from it using simple assignment statements. For example, the following statement generates a 4×4 sparse matrix and then adds another nonzero element to it.

```

>> a = speye(4)
a =
    (1,1)      1
    (2,2)      1
    (3,3)      1
    (4,4)      1
>> a(2,1) = -2
a =
    (1,1)      1
    (2,1)     -2
    (2,2)      1
    (3,3)      1
    (4,4)      1

```

MATLAB allows full and sparse matrices to be freely mixed and used in any combination. The result of an operation between a full matrix and a sparse matrix can be either a full matrix or a sparse matrix, depending on which result is the most efficient. Essentially, any matrix technique that is supported for full matrices is also available for sparse matrices.

A few of the common sparse matrix functions are listed in Table 7.1.

Example 7.1—Solving Simultaneous Equations with Sparse Matrices

To illustrate the ease with which sparse matrices can be used in MATLAB, we will solve the following simultaneous system of equations with both full and sparse matrices.

$$\begin{aligned}
1.0x_1 + 0.0x_2 + 1.0x_3 + 0.0x_4 + 0.0x_5 + 2.0x_6 + 0.0x_7 - 1.0x_8 &= 3.0 \\
0.0x_1 + 1.0x_2 + 0.0x_3 + 0.4x_4 + 0.0x_5 + 0.0x_6 + 0.0x_7 + 0.0x_8 &= 2.0 \\
0.5x_1 + 0.0x_2 + 2.0x_3 + 0.0x_4 + 0.0x_5 + 0.0x_6 - 1.0x_7 + 0.0x_8 &= -1.5 \\
0.0x_1 + 0.0x_2 + 0.0x_3 + 2.0x_4 + 0.0x_5 + 1.0x_6 + 0.0x_7 + 0.0x_8 &= 1.0 \\
0.0x_1 + 0.0x_2 + 1.0x_3 + 1.0x_4 + 1.0x_5 + 0.0x_6 + 0.0x_7 + 0.0x_8 &= -2.0 \\
0.0x_1 + 0.0x_2 + 0.0x_3 + 1.0x_4 + 0.0x_5 + 1.0x_6 + 0.0x_7 + 0.0x_8 &= 1.0 \\
0.5x_1 + 0.0x_2 + 0.0x_3 + 0.0x_4 + 0.0x_5 + 0.0x_6 + 1.0x_7 + 0.0x_8 &= 1.0 \\
0.0x_1 + 1.0x_2 + 0.0x_3 + 0.0x_4 + 0.0x_5 + 0.0x_6 + 0.0x_7 + 1.0x_8 &= 1.0
\end{aligned}$$

Table 7.1 Common MATLAB Sparse Matrix Functions

Category	Function	Description
Create sparse matrices	speye	Create a sparse identity matrix.
	sprand	Create a sparse matrix containing uniformly distributed random values.
	sprandn	Create a sparse matrix containing normally distributed random values.
Full-to-sparse conversion functions	sparse	Convert a full matrix into a sparse matrix.
	full	Convert a sparse matrix into a full matrix.
	find	Find indices and values of nonzero elements in a matrix.
Working with sparse matrices	nnz	Number of nonzero matrix elements.
	nonzeros	Return a column vector containing the nonzero elements in a matrix.
	nzmax	Amount of storage allocated for nonzero matrix elements.
	spones	Replace nonzero sparse matrix elements with ones.
	spalloc	Allocate space for a sparse matrix.
	issparse	Returns 1 (true) for sparse matrix.
	sfun	Apply function to nonzero matrix elements.
	spy	Visualize sparsity pattern as a plot.

Solution

To solve this problem, we will create full matrices of the equation coefficients and then convert them to sparse form using the `sparse` function. Then we will solve the equation both ways comparing the results and the memory required.

The script file to perform these calculations follows.

```
% Script file: simul.m
%
% Purpose:
% This program solves a system of 8 linear equations in 8
% unknowns ( $a \cdot x = b$ ), using both full and sparse matrices.
%
% Record of revisions:
% Date      Programmer          Description of change
% =====    ======          =====
% 10/14/98   S. J. Chapman      Original code
```

```

% Define variables:
% a           -- Coefficients of x (full matrix)
% as          -- Coefficients of x (sparse matrix)
% b           -- Constant coefficients (full matrix)
% bs          -- Constant coefficients (sparse matrix)
% x           -- Solution (full matrix)
% xs          -- Solution (sparse matrix)

% Define coefficients of the equation a*x = b for
% the full matrix solution.
a = [ 1.0  0.0  1.0  0.0  0.0  2.0  0.0 -1.0; ...
       0.0  1.0  0.0  0.4  0.0  0.0  0.0  0.0; ...
       0.5  0.0  2.0  0.0  0.0  0.0 -1.0  0.0; ...
       0.0  0.0  0.0  2.0  0.0  1.0  0.0  0.0; ...
       0.0  0.0  1.0  1.0  1.0  0.0  0.0  0.0; ...
       0.0  0.0  0.0  1.0  0.0  1.0  0.0  0.0; ...
       0.5  0.0  0.0  0.0  0.0  0.0  1.0  0.0; ...
       0.0  1.0  0.0  0.0  0.0  0.0  0.0  1.0];

b = [ 3.0  2.0 -1.5  1.0 -2.0  1.0  1.0  1.0]';

% Define coefficients of the equation a*x = b for
% the sparse matrix solution.
as = sparse(a);
bs = sparse(b);

% Solve the system both ways
disp ('Full matrix solution:');
x = a\b

disp ('Sparse matrix solution:');
xs = as\bs

% Show workspace
disp('Workspace contents after the solutions:')
whos

```

When this program is executed, the results are

```

> simul
Full matrix solution:
x =
    0.5000
    2.0000
   -0.5000
    0.0000
   -1.5000
    1.0000
    0.7500
   -1.0000

```

```

Sparse matrix solution:
xs =
    (1,1)      0.5000
    (2,1)      2.0000
    (3,1)     -0.5000
    (5,1)     -1.5000
    (6,1)      1.0000
    (7,1)      0.7500
    (8,1)     -1.0000

Workspace contents after the solutions:
  Name      Size      Bytes  Class
  a         8x8        512  double array
  as        8x8        276  sparse array
  b         8x1         64  double array
  bs        8x1'       104  sparse array
  x         8x1         64  double array
  xs        8x1         92  sparse array

Grand total is 115 elements using 1112 bytes

```

The answers are the same for both solutions. Note that the sparse solution does not contain a solution for x_4 because that value is zero, and zeros are not carried in a sparse matrix! Also, note that the sparse form of matrix b actually takes up more space than the full form. This happens because the sparse representation must store the indices as well as the values in the arrays, so it is less efficient if most of the elements in an array are nonzero.

7.2 Cell Arrays

A **cell array** is a special MATLAB array whose elements are *cells*, containers that can hold other MATLAB arrays. For example, one cell of a cell array might contain an array of real numbers, another an array of strings, and yet another an array of complex numbers (Figure 7.1).

In programming terms, each element of a cell array is a *pointer* to another data structure, and those data structures can be of different types. Figure 7.2 illustrates this concept. Cell arrays are great ways to collect information about a problem since all of the information can be kept together and accessed by a single name.

Cell arrays use braces {} instead of parentheses () for selecting and displaying the contents of cells. This difference is due to the fact that *cell arrays contain data structures instead of data*. Suppose that the cell array a is defined as shown in Figure 7.2. Then the contents of element $a(1,1)$ is a data structure containing a 3×3 array of numeric data, and a reference to $a(1,1)$ displays the *contents* of the cell, which is the data structure.

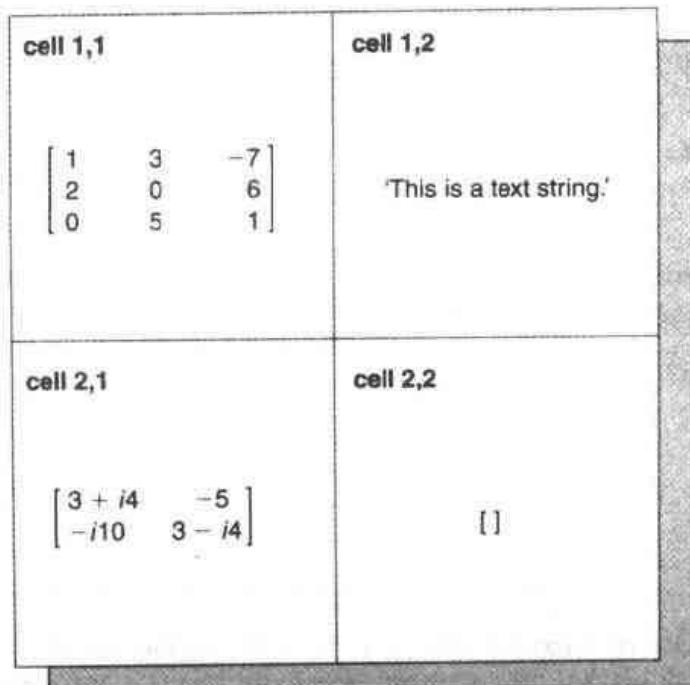


Figure 7.1 The individual elements of a cell array may point to real arrays, complex arrays, character strings, other cell arrays, or even empty arrays.

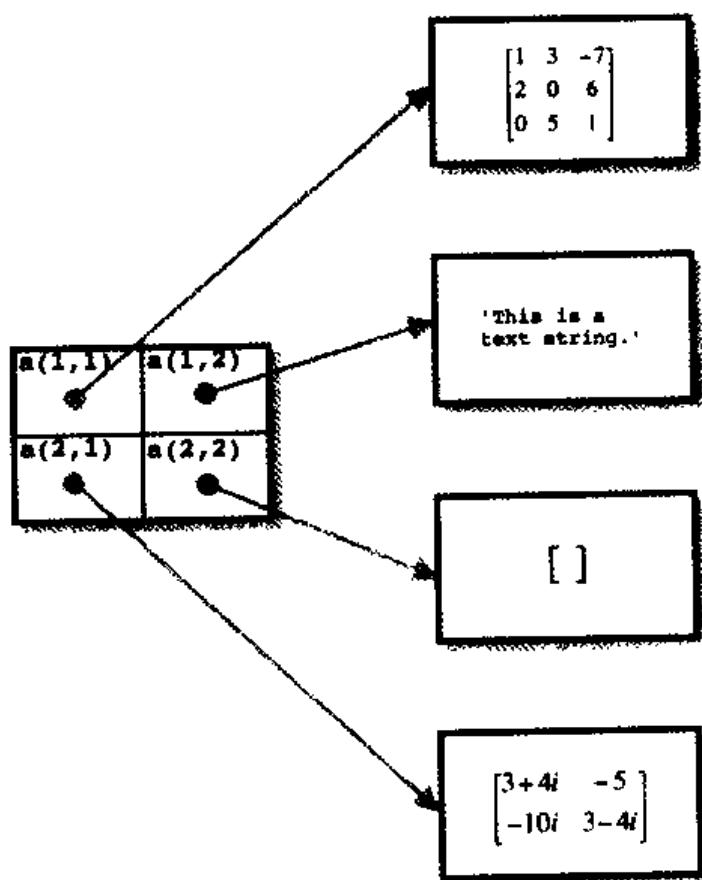


Figure 7.2 Each element of a cell array holds a *pointer* to another data structure, and different cells in the same cell array can point to different types of data structures.

```
>> a(1,1)
ans =
[3x3 double]
```

By contrast, a reference to `a{1,1}` displays *the contents of the data structure contained in the cell*.

```
>> a{1,1}
ans =
1     3    -7
2     0     6
0     5     1
```

In summary, the notation `a(1,1)` refers to the contents of cell `a(1,1)` (which is a data structure), while the notation `a{1,1}` refers to the contents of the data structure within the cell.

Programming Pitfalls

Be careful not to confuse `()` with `{ }` when addressing cell arrays. They are very different operations!

7.2.1 Creating Cell Arrays

Cell arrays can be created in two ways.

- By using assignment statements.
- By preallocating a cell array using the `cell` function.

The simplest way to create a cell array is to directly assign data structures to individual cells, one cell at a time. However, preallocating cell arrays is more efficient, so you should preallocate really large cell arrays.

7.2.1.1 Allocating Cell Arrays Using Assignment Statements

You can assign values to cell arrays one cell at a time using assignment statements. There are two ways to assign data to cells, known as **content indexing** and **cell indexing**.

Content indexing involves placing braces “`{ }` ” around the cell subscripts, together with cell contents in ordinary notation. For example, the following statement creates the 2×2 cell array in Figure 7.2.

```
a(1,1) = [1 3 -7; 2 0 6; 0 5 1];
a(1,2) = 'This is a text string.';
a(2,1) = [3+4*i -5; -10*i 3 - 4*i];
a(2,2) = [];
```

This type of indexing defines the *contents of the data structure contained in a cell*.

Cell indexing involves placing braces “{}” around the data to be stored in a cell, together with cell subscripts in ordinary subscript notation. For example, the following statement creates the 2×2 cell array in Figure 7.2.

```
a(1,1) = {[1 3 -7; 2 0 6; 0 5 1]};
a(1,2) = {'This is a text string.'};
a(2,1) = {[3+4*i -5; -10*i 3 - 4*i]};
a(2,2) = {};
```

This type of indexing *creates a data structure containing the specified data, and then assigns that data structure to a cell*.

These two forms of indexing are completely equivalent, and they can be freely mixed in any program.

Programming Pitfalls

Do not attempt to create a cell array with the same name as an existing numeric array. If you do this, MATLAB will assume that you are trying to assign cell contents to an ordinary array, and it will generate an error message. Be sure to clear the numeric array before trying to create a cell array with the same name.

7.2.1.2 Preallocating Cell Arrays with the `cell` Function

The `cell` function allows you to preallocate empty cell arrays of the specified size. For example, the following statement creates an empty 2×2 cell array.

```
a = cell(2,2)
```

Once a cell array is created, you can use assignment statements to fill values in the cells.

7.2.2 Using Braces {} as Cell Constructors

It is possible to define many cells at once by placing all of the cell contents between a single set of braces. Individual cells on a row are separated by commas, and rows are separated by semicolons. For example, the following statement creates a 2×3 cell array.

```
b = {[1 2], 17, [2;4]; 3-4*i, 'Hello', eye(3)}
```

7.2.3 Viewing the Contents of Cell Arrays

MATLAB displays the data structures in each element of a cell array in a condensed form that limits each data structure to a single line. If the entire data structure can be displayed on the single line, it is. Otherwise, a summary is displayed. For example, cell arrays *a* and *b* are displayed as

```
> a
a =
    [3x3 double]    [1x22 char]
    [2x2 double]    []
> b
b =
    [1x2 double]    [    17]    [2x1 double]
    [3.0000- 4.0000i]  'Hello'  [3x3 double]
```

Note that MATLAB is displaying the *data structures*, complete with brackets or apostrophes, not the contents of the data structures.

If you would like to see the full contents of a cell array, use the `celldisp` function. This function displays the *contents of the data structures in each cell*.

```
> celldisp(a)
a{1,1} =
    1      3      -7
    2      0       6
    0      5       1
a{2,1} =
    3.0000 + 4.0000i  -5.0000
    0 -10.0000i   3.0000 - 4.0000i
a{1,2} =
This is a text string.
a{2,2} =
[]
```

For a high-level graphical display of the structure of a cell array, use function `cellplot`. For example, the function `cellplot(b)` produces the plot shown in Figure 7.3.

7.2.4 Extending Cell Arrays

If a value is assigned to a cell array element that does not currently exist, the element will be automatically created, and any additional cells necessary to preserve the shape of the array will be automatically created. For example, suppose that array *a* has been defined to be a 2×2 cell array as shown in Figure 7.1. If the following statement is executed

```
a{3,3} = 5
```

the cell array will be automatically extended to 3×3 , as shown in Figure 7.4.

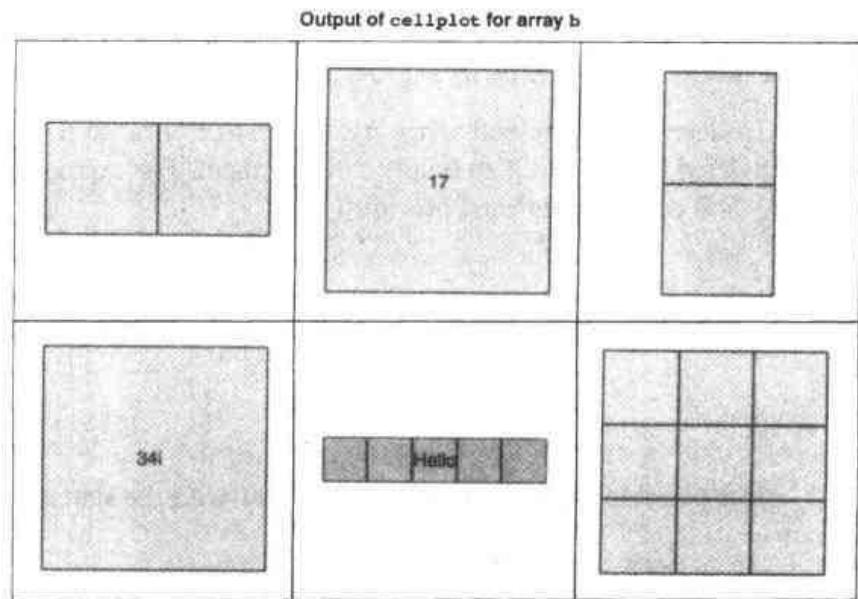


Figure 7.3 The structure of cell array b is displayed as a nested series of boxes by function `cellplot`.

cell 1,1 $\begin{bmatrix} 1 & 3 & -7 \\ 2 & 0 & 8 \\ 0 & 5 & 1 \end{bmatrix}$	cell 1,2 'This is a text string.'	cell 1,3 {}
cell 2,1 $\begin{bmatrix} 3 + i4 & -5 \\ -i10 & 3 - i4 \end{bmatrix}$	cell 2,2 ()	cell 2,3 ()
cell 3,1 ()	cell 3,2 ()	cell 3,3 [5]

Figure 7.4 The result of assigning a value to a (3, 3). Note that four other empty cells were created to preserve the shape of the cell array.

7.2.5 Deleting Cells in Arrays

To delete an entire cell array, use the `clear` command. Subsets of cells can be deleted by assigning an empty array to them. For example, assume that `a` is the 3×3 cell array defined previously.

```
> a
a =
    [3x3 double]    [1x22 char]    []
    [2x2 double]    []            []
    []              []            [5]
```

It is possible to delete the entire third row with the statement

```
> a(3,:) = []
a =
    [3x3 double]    [1x22 char]    []
    [2x2 double]    []            []
```

7.2.6 Using Data in Cell Arrays

The data inside the data structures in a cell array can be used at any time using either content indexing or cell indexing. For example, suppose a cell array `c` is defined as

```
c = {[1 2;3 4], 'dogs'; 'cats', i}
```

The contents of the array stored in cell `c(1,1)` can be accessed as follows

```
> c{1,1}
ans =
    1     2
    3     4
```

and the contents of the array in cell `c(2,1)` can be accessed as follows.

```
> c{2,1}
ans =
cats
```

Subsets of a cell's contents can be obtained by concatenating the two sets of subscripts. For example, suppose that we would like to get the element (1,2) from the array stored in cell `c(1,1)` of cell array `c`. To do this, we would use the expression `c{1,1}(1,2)`, which says: select element (1,2) from the contents of the data structure contained in cell `c(1,1)`.

```
>> c{1,1}(1,2)
ans =
2
```

7.2.7 Cell Arrays of Strings

It is often convenient to store groups of strings in a cell array, instead of storing them in rows of a standard character array, because each string in a cell array can have a different length, while every row of a standard character array must have an identical length. This fact means that *strings in cell arrays do not have to be padded with blanks*. Many MATLAB Graphical User Interface functions use cell arrays for precisely this reason, as we will see in Chapter 10.

Cell arrays of strings can be created in one of two ways. Either the individual strings can be inserted into the array with brackets, or else function `cellstr` can be used to convert a 2-D string array into a cell array of strings.

The following example creates a cell array of strings, by inserting the strings into the cell array one at a time, and displays the resulting cell array. Note that the individual strings can be of different lengths.

```
>> cellstring(1) = 'Stephen J. Chapman';
>> cellstring(2) = 'Male';
>> cellstring(3) = 'SSN 999-99-9999';
>> cellstring
'Stephen J. Chapman'    'Male'    'SSN 999-99-9999'
```

Function `cellstr` creates a cell array of strings from a 2-D string array. Consider the character array

```
>> data = ['Line 1'      ';' 'Additional Line']
data =
Line 1
Additional Line
```

This 2×15 character array can be converted into an cell array of strings with the function `cellstr` as follows.

```
>> c = cellstr(data)
c =
'Line 1'
'Additional Line'
```

and it can be converted back to a standard character array using function `char`.

```
>> newdata = char(c)
newdata =
Line 1
Additional Line
```

7.2.8 The Significance of Cell Arrays

Cell arrays are extremely flexible, since any amount of any type of data can be stored in each cell. As a result, cell arrays are used in many internal MATLAB data structures. We must understand them in order to use many features of the MATLAB Graphical User Interface, which we will study in Chapter 10.

In addition, the flexibility of cell arrays makes them regular features of functions with variable numbers of input arguments and output arguments. A special input argument, `varargin`, is available within user-defined MATLAB functions to support variable numbers of input arguments. This argument appears as the last item in an input argument list, and it returns a cell array, so *a single dummy input argument can support any number of actual arguments*. Each actual argument becomes one element of the cell array returned by `varargin`. If it is used, `varargin` must be the *last* input argument in a function after all of the required input arguments.

For example, suppose we are writing a function that may have any number of input arguments. This function could be implemented as shown.

```
function test1(varargin)
    disp(['There are ' int2str(nargin) ' arguments.']);
    disp('The input arguments are:');
    disp(varargin);
```

When this function is executed with varying numbers of arguments, the results are

```
> test1
There are 0 arguments.
The input arguments are:
> test1(6)
There are 1 arguments.
The input arguments are:
[6]
> test1(1,'test 1',[1 2;3 4])
There are 3 arguments.
The input arguments are:
[1]    'test 1'    [2x2 double]
```

As you can see, the arguments become elements of a cell array within the function.

A sample function making use of variable numbers of arguments follows. Function `plotline` accepts an arbitrary number of 1×2 row vectors, with each vector containing the (x,y) position of one point to plot. The function plots a line connecting all of the (x,y) values together. Note that this function also accepts an optional line specification string and passes that specification on to the `plot` function.

```
function plotline(varargin)
%PLOTLINE Plot points specified by [x,y] pairs.
% Function PLOTLINE accepts an arbitrary number of
```

```

% [x,y] points and plots a line connecting them.
% In addition, it can accept a line specification
% string, and pass that string on to the plot function.

% Define variables:
% ii          -- Index variable
% jj          -- Index variable
% linespec    -- String defining plot characteristics
% msg         -- Error message
% varargin   -- Cell array containing input arguments
% x           -- x values to plot
% y           -- y values to plot

% Record of revisions:
% Date        Programmer            Description of change
% ===        ======              =====
% 10/20/98    S. J. Chapman      Original code

% Check for a legal number of input arguments.
% We need at least 2 points to plot a line...
msg = nargchk(2,Inf,nargin);
error(msg);

% Initialize values
jj = 0;
linespec = '';

% Get the x and y values, making sure to save the line
% specification string, if one exists.
for ii = 1:nargin

    % Is this argument an [x,y] pair or the line
    % specification?
    if ischar(varargin{ii})

        % Save line specification
        linespec = varargin{ii};

    else

        % This is an [x,y] pair. Recover the values.
        jj = jj + 1;
        x(jj) = varargin{ii}(1);
        y(jj) = varargin{ii}(2);

    end
end

% Plot function.
if isempty(linespec)
    plot(x,y);
else
    plot(x,y,linespec);
end

```

When this function is called with the arguments that follow, the resulting plot is shown in Figure 7.5. Try the function with different numbers of arguments and see for yourself how it behaves.

```
plotline([0 0], [1 1], [2 4], [3 9], 'k--');
```

There is also a special output argument, varargout, to support variable numbers of output arguments. This argument appears as the last item in an output argument list, and it returns a cell array, so *a single dummy output argument can support any number of actual arguments*. Each actual argument becomes one element of the cell array stored in varargout.

If it is used, varargout must be the *last* output argument in a function, after all of the required input arguments. The number of values to be stored in varargout can be determined from function nargout, which specifies the number of actual output arguments for any given function call. For example, suppose we are writing a function that returns random values in any number of output arguments. Our function can detect the number of actual arguments used during any specific call with function nargout and can store that number of values as cell array elements in varargout.

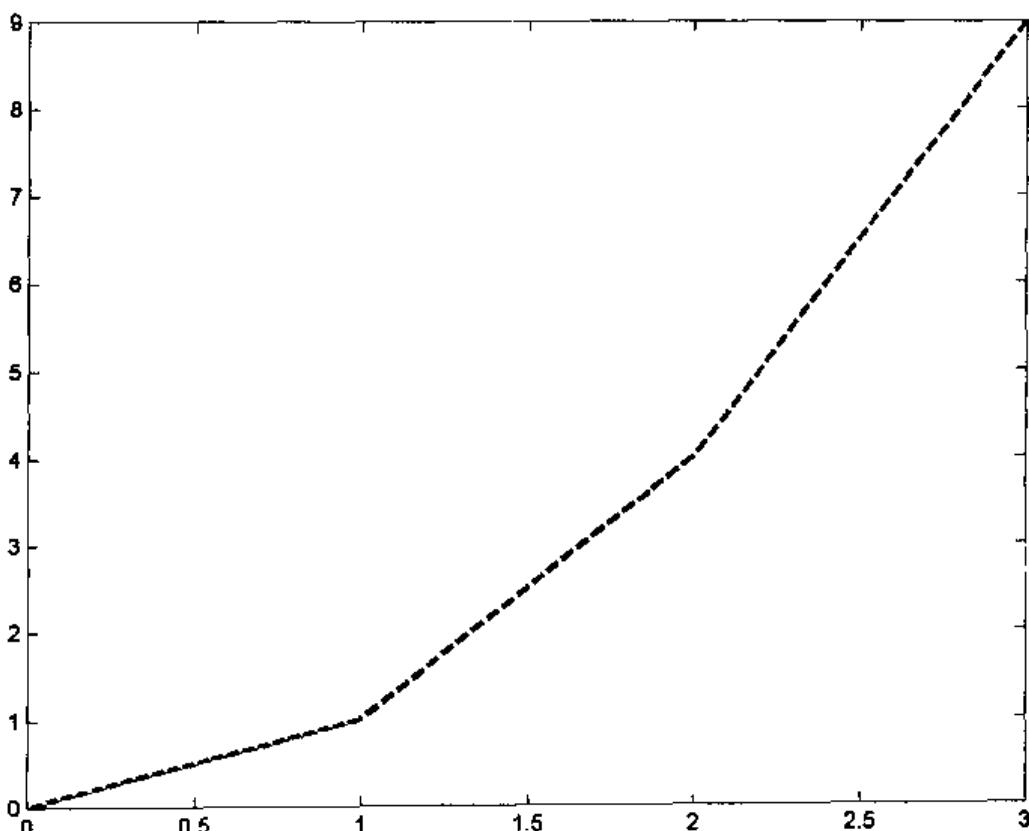


Figure 7.5 The plot produced by function plotline.

```

function [nvals,varargout] = test2(mult)
% nvals is the number of random values returned
% varargout contains the random values returned
nvals = nargin - 1;
for ii = 1:nargout-1
    varargout(ii) = randn * mult;
end

```

When this function is executed, the results are as follows.

```

> test2(4)
ans =
    -
    -1
> [a b c d] = test2(4)
a =
    3
b =
    -1.7303
c =
    -6.6623
d =
    0.5013

```

Good Programming Practice

Use cell array arguments varargin and varargout to create functions that support varying numbers of input and output arguments.

7.2.9 Summary of cell Array Functions

The common MATLAB functions that support cell arrays are summarized in Table 7.2.

Table 7.2 Common MATLAB Cell Array Functions

Function	Description
cell	Predefine a cell array structure.
celldisp	Display contents of a cell array.
cellplot	Plot structure of a cell array.
cellstr	Convert a 2-D character array to a cell array of strings.
char	Convert a cell array of strings into a 2-D character array.

7.3 Structure Arrays

An *array* is a data type in which there is a name for the whole data structure, but individual elements within the array are only known by number. Thus the fifth element in the array named arr would be accessed as `arr(5)`. Note that all of the individual elements in an array *must be of the same type* (numeric or character).

A *cell array* is also a data type in which there is a name for the whole data structure, and the individual elements within the array are only known by number. However, the individual elements in the cell array *may be of different types*.

In contrast, a **structure** is a data type in which each individual element is given a *name*. The individual elements of a structure are known as **fields**, and each field in a structure may have a different type. The individual fields are addressed by combining the name of the structure with the name of the field, separated by a period.

Figure 7.6 shows a sample structure named `student`. This structure has five fields, called `name`, `addr1`, `city`, `state`, and `zip`. The field called “`name`” would be addressed as `student.name`.

A **structure array** is an array of structures. Each structure in the array will have identically the same fields, but the data stored in each field can differ. For example, a class could be described by an array of the structure `student`. The first student’s name would be addressed as `student(1).name`, the second student’s city would be addressed as `student(2).city`, and so forth.

7.3.1 Creating Structures

Structures can be created in two ways.

- A field at a time using assignment statements.
- All at once using the `struct` function.

7.3.1.1 Building a Structure with Assignment Statements

You can build a structure a field at a time using assignment statements. Each time that data is assigned to a field, that field is automatically created. For example, the structure shown in Figure 7.6 can be created with the following statements.

```
> student.name='John Doe';
> student.addr1='123 Main Street';
> student.city ='Anytown';
> student.zip='71211'
student =
    name: 'John Doe'
    addr1: '123 Main Street'
    city: 'Anytown'
    state: 'LA'
    zip: '71211'
```

A second student can be added to the structure by adding a subscript to the structure name (*before* the period).

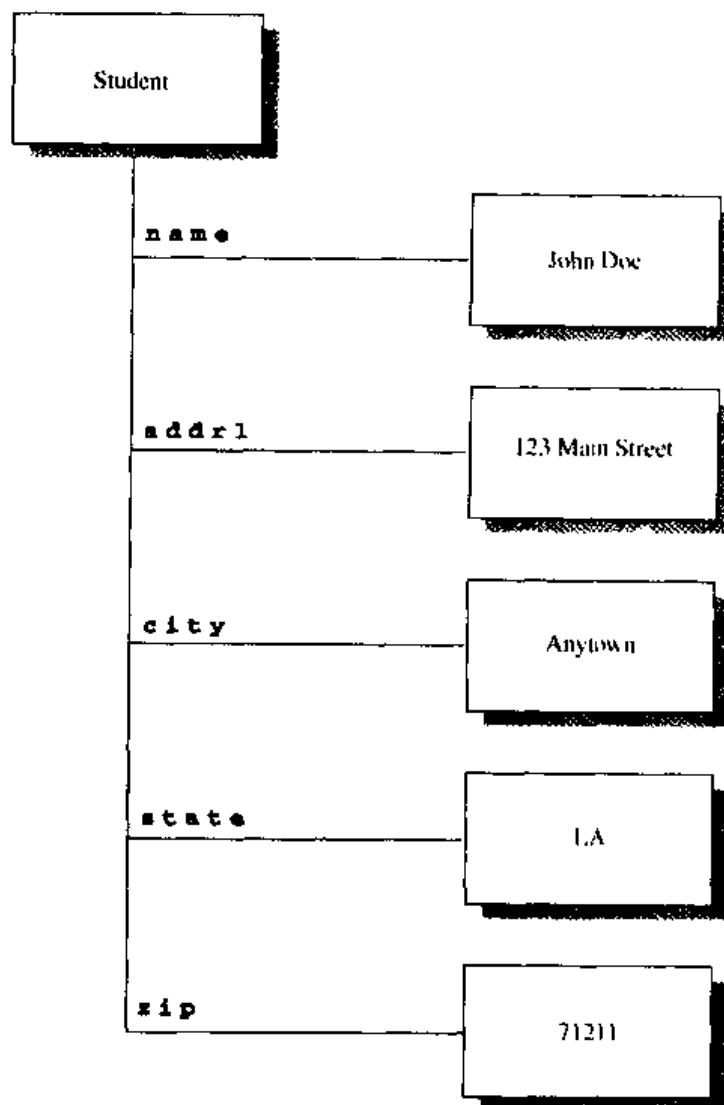


Figure 7.6 A sample structure. Each element within the structure is called a field, and each field is addressed by name.

```

> student(2).name = 'Jane Q. Public'
student =
1x2 struct array with fields:
  name
  addr1
  city
  state
  zip

```

student is now a 1×2 array. Note that when a structure array has more than one element, only the field names are listed, not their contents. The contents of each element can be listed by typing the element separately in the Command Window.

```

> student(1)
ans =
    name: 'John Doe'
    addr1: '123 Main Street'
    city: 'Anytown'
    state: 'LA'
    zip: '71211'
> student(2)
ans =
    name: 'Jane Q. Public'
    addr1: []
    city: []
    state: []
    zip: []

```

Note that *all of the fields of a structure are created for each array element whenever that element is defined*, even if they are not initialized. The uninitialized fields will contain empty arrays, which can be initialized with assignment statements at a later time.

The field names used in a structure can be recovered at any time using the `fieldnames` function. This function returns a list of the field names in a cell arrays of strings and is very useful for working with structure arrays within a program.

7.3.1.2 Creating Structures with the `struct` Function

The `struct` function allows you to preallocate an array of structures. The basic form of this function is

```
structure_array = struct(fields)
```

where `fields` is a padded string array or cell array containing the field names for the structures. With this syntax, function `struct` initializes every field in the array to an empty matrix.

It is also possible to initialize the fields as they are defined. This form of the function is

```
str_array = struct('field1',val1,'field2',val2, ...)
```

where the arguments are field names and their values.

7.3.2 Adding Fields to Structures

If a new field name is defined for any element in a structure array, the field is *automatically added to all of the elements in the array*. For example, suppose that we add some exam scores to Jane Public's record.

```

> student(2).exams = [90 82 88]
student =

```

```
1x2 struct array with fields:
  name
  addr1
  city
  state
  zip
  exams
```

There is now a field called exams in every record of the array, as shown below. This field will be initialized for student(2) and will be an empty array for all other students until appropriate assignment statements are issued.

```
> student(1)
ans =
  name: 'John Doe'
  addr1: '123 Main Street'
  city: 'Anytown'
  state: 'LA'
  zip: '71211'
  exams: []
> student(2)
ans =
  name: 'Jane Q. Public'
  addr1: []
  city: []
  state: []
  zip: []
  exams: [90 82 88]
```

7.3.3 Removing Fields from Structures

A field can be removed from a structure array using the `rmfield` function. The form of this function is

```
struct2 = rmfield(struct_array, 'field')
```

where `struct_array` is a structure array, '`field`' is the field to remove, and `struct2` is the name of a new structure with that field removed. For example, we can remove the field '`zip`' from structure array `student` with the following statement.

```
> stu2 = rmfield(student, 'zip')
stu2 =
1x2 struct array with fields:
  name
  addr1
  city
  state
  exams
```

7.3.4 Using Data in Structure Arrays

Now let us assume that structure array `student` has been extended to include 3 students, and all data has been filled in as shown in Figure 7.7. How do we use the data in this structure array?

The information in any field of any array element can be accessed by naming the array element followed by a period and the field name.

```
> student(2).addr1
ans =
P. O. Box 17
> student(3).exams
ans =
65     84     81
```

Any individual item with a field can be accessed by adding a subscript after the field name. For example, the second exam of the third student is

```
> student(3).exams(2)
ans =
84
```

The fields in a structure array can be used as arguments in any function that supports that type of data. For example, to calculate `student(2)`'s exam average, we could use the function

```
> mean(student(2).exams)
ans =
86.6667
```

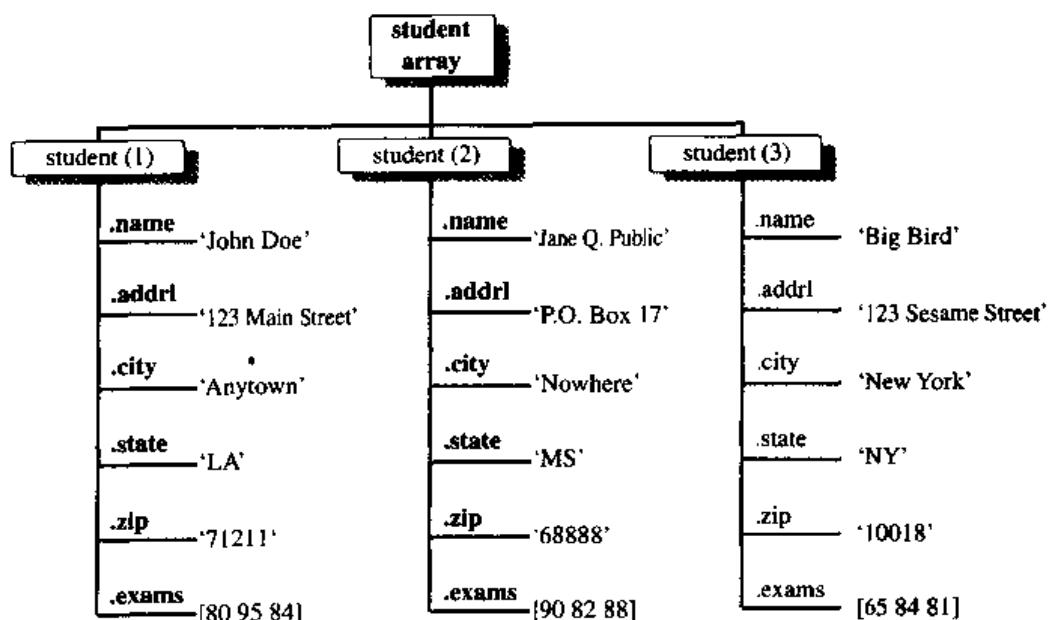


Figure 7.7 The student array with three elements and all fields filled in.

Unfortunately, we *cannot* get the values from a given field across multiple elements at the same time. For example, the statement `student.name` is meaningless and leads to an error. If we want to get the names of all the students, we must use a `for` loop.

```
for ii = 1:length(student)
    disp(student(ii).name);
end
```

Similarly, if we wanted to get the average of all exams from all students, we cannot use the function `mean(student.exams)`. Instead, we must access each student's exams separately and build a list of all exams.

```
exam_list = [];
for ii = 1:length(student)
    exam_list = [exam_list student(ii).exams];
end
mean(exam_list)
```

7.3.5 The `getfield` and `setfield` Functions

Two MATLAB functions are available to make structure arrays easier to use in user-defined functions. Function `getfield` gets the current value stored in a field, and function `setfield` inserts a new value into a field. The structure of function `getfield` is

```
f = getfield(array,{array_index},'field',{field_index})
```

where the `field_index` is optional, and `array_index` is optional for a 1×1 structure array. The function call corresponds to the statement

```
f = array(array_index).field(field_index);
```

but it can be used even if the programmer does not know the names of the fields in the structure array at the time the program is written.

For example, suppose that we needed to write a function to read and manipulate the data in an unknown structure array. This function could determine the field names in the structure using a call to `fieldnames` and then could read the data using function `getfield`. To read the zip code of the second student, the function would be

```
* zip = getfield(student,{2},'zip')
zip =
68888
```

Similarly, a program could modify values in the structure using function `setfield`. The structure of function `setfield` is

```
f = setfield(array,{array_index},'field',{field_index},value)
```

where *f* is the output structure array, the *field_index* is optional, and *array_index* is optional for a 1×1 structure array. The function call corresponds to the statement

```
array(array_index).field(field_index) = value;
```

7.3.6 Using the *size* Function with Structure Arrays

When the *size* function is used with a structure array, it returns the size of the structure array itself. When the *size* function is used with a *field* from a particular element in a structure array, it returns the size of that field instead of the size of the whole array. For example

```
> size(student)
ans =
    1      3
> size(student(1).name)
ans =
    1      8
```

7.3.7 Nesting Structure Arrays

Each field of a structure array can be of any data type, including a cell array or a structure array. For example, the following statements define a new structure array as a field under array *student* to carry information about each class that the student is enrolled in.

```
student(1).class(1).name = 'COSC 2021'
student(1).class(2).name = 'PHYS 1001'
student(1).class(1).instructor = 'Mr. Jones'
student(1).class(2).instructor = 'Mrs. Smith'
```

After these statements are issued, *student*(1) contains the following data. Note the technique used to access the data in the nested structures.

```
> student(1)
ans =
    name: 'John Doe'
    addr1: '123 Main Street'
    city: 'Anytown'
    state: 'LA'
    zip: '71211'
    exams: [80 95 84]
    class: [1x2 struct]
*> student(1).class
ans =
1x2 struct array with fields:
    name
    instructor
```

Table 7.3 Common MATLAB Functions That Support Structures

Function	Description
fieldnames	Return a list of field names in a cell array of strings.
getfield	Get current value from a field.
rmfield	Remove a field from a structure array.
setfield	Set new value into a field.
struct	Predefine a structure array.

```

>> student(1).class(1)
ans =
    name: 'COSC 2021'
    instructor: 'Mr. Jones'
>> student(1).class(2)
ans =
    name: 'PHYS 1001'
    instructor: 'Mrs. Smith'
>> student(1).class(2).name
ans =
    PHYS 1001

```

7.3.8 Summary of structure Functions

The common MATLAB functions that support structures are summarized in Table 7.3.

Quiz 7.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 7.1 through 7.3. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. What is a sparse array? How does it differ from a full array? How can you convert from a sparse array to a full array, and vice versa?
2. What is a cell array? How does it differ from an ordinary array?
3. What is the difference between content indexing and cell indexing?
4. What is a structure? How does it differ from ordinary arrays and cell arrays?
5. What is the purpose of varargin? How does it work?

6. Given the definition of array *a* that follows, what will be produced by each of the following sets of statements? (Note: some of these statements may be illegal. If a statement is illegal, explain why.)

```
a{1,1} = [1 2 3; 4 5 6; 7 8 9];
a(1,2) = {'Comment line'};
a{2,1} = j;
a{2,2} = a{1,1} - a{1,1}(2,2);
```

- (a) *a*(1,1)
- (b) *a*{1,1}
- (c) 2*a(1,1)
- (d) 2*a{1,1}
- (e) *a*{2,2}
- (f) *a*(2,3) = {[−17; 17]}
- (g) *a*{2,2}(2,2)

7. Given the definition of structure array *b* that follows, what will be produced by each of the following sets of statements? (Note: some of these statements may be illegal. If a statement is illegal, explain why.)

```
b(1).a = −2*eye(3);
b(1).b = 'Element 1';
b(1).c = [1 2 3];
b(2).a = {b(1).c' [-1; -2; -3] b(1).c'};;
b(2).b = 'Element 2';
b(2).c = [1 0 .1];
```

- (a) *b*(1).a - *b*(2).a
 - (b) strcmp(*b*(1).b,*b*(2).b,6)
 - (c) mean(*b*(1).c)
 - (d) mean(*b*.c)
 - (e) *b*
 - (f) *b*(1)
-

7.4 Summary

Sparse arrays are special arrays in which memory is only allocated for non-zero elements. Three values are saved for each nonzero element—a row number, a column number, and the value itself. This form of storage is much more efficient than ordinary full arrays for the situation where only a tiny fraction of the elements are nonzero. MATLAB includes functions and intrinsic calculations for sparse arrays, so they can be freely and transparently mixed with full arrays.

Cell arrays are arrays whose elements are *cells*, containers that can hold other MATLAB arrays. Any sort of data can be stored in a cell, including structure ar-

rays and other cell arrays. They are a very flexible way to store data and are used in many internal MATLAB Graphical User Interface functions.

Structure arrays are a data type in which each individual element is given a name. The individual elements of a structure are known as fields, and each field in a structure may have a different type. The individual fields are addressed by combining the name of the structure with the name of the field, separated by a period. Structure arrays are useful for grouping together all of the data related to a particular person or thing into a single location.

7.4.1 Summary of Good Programming Practice

The following guideline should be followed.

1. Use cell array arguments `varargin` and `varargout` to create functions that support varying numbers of input and output arguments.

7.4.2 MATLAB Summary

The following summary lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one.

Commands and Functions

<code>cell</code>	Predefine a cell array structure.
<code>celldisp</code>	Display contents of a cell array.
<code>cellplot</code>	Plot structure of a cell array.
<code>cellstr</code>	Convert a 2-D character array to a cell array of strings.
<code>fieldnames</code>	Return a list of field names in a cell array of strings.
<code>getfield</code>	Get current value from a field.
<code>full</code>	Convert a sparse matrix into a full matrix.
<code>nnz</code>	Number of nonzero matrix elements.
<code>nonzeros</code>	Return a column vector containing the nonzero elements in a matrix.
<code>nzmax</code>	Amount of storage allocated for nonzero matrix elements.
<code>rmfield</code>	Remove a field from a structure array.
<code>setfield</code>	Set new value into a field.
<code>spalloc</code>	Allocate space for a sparse matrix.
<code>sparse</code>	Convert a full matrix into a sparse matrix.
<code>speye</code>	Create a sparse identity matrix.
<code>spfun</code>	Apply function to nonzero matrix elements.
<code>spones</code>	Replace nonzero sparse matrix elements with ones.
<code>sprand</code>	Create a sparse uniformly distributed random matrix.
<code>sprandn</code>	Create a sparse normally distributed random matrix.
<code>sprintf</code>	Write formatted data to string.
<code>spy</code>	Visualize sparsity pattern as a plot.

7.5 Exercises

- 7.1** Write a MATLAB function that will accept a cell array of strings and sort them into ascending order according to the lexicographic order of the ASCII character set. (You may use function `c_strcmp` from Chapter 6 for the comparisons if you wish.)
- 7.2** Write a MATLAB function that will accept a cell array of strings and sort them into ascending order according to *alphabetical order*. (This implies that you must treat 'A' and 'a' as the same letter.)
- 7.3** Create a sparse 100×100 array `a` in which about 5% of the elements contain normally distributed random values, and all of the other elements are zero (use function `sprandn` to generate these values). Next, set all of the diagonal elements in the array to 1. Next, define a 100-element sparse column array `b`, and initialize that array with 100 uniformly distributed values produced by function `rand`. Answer the following questions about these arrays.
- Create a full array `a_full` from the sparse array `a`. Compare the memory required to store the full array and the sparse array. Which is more efficient?
 - Plot the distribution of values in array `a` using function `spy`.
 - Create a full array `b_full` from the sparse array `b`. Compare the memory required to store the full array and the sparse array. Which is more efficient?
 - Solve the system of equations $a * x = b$ for x using both the full arrays and the sparse arrays. How do the two sets of answers compare? Time the two solutions. Which one is faster?
- 7.4** Create a function that accepts any number of numeric input arguments, and sums up all of individual elements in all of the arguments. Test your function by passing it the four arguments $a = 10$, $b = \begin{bmatrix} 4 \\ -2 \\ 2 \end{bmatrix}$, $c = \begin{bmatrix} 1 & 0 & 3 \\ -5 & 1 & 2 \\ 1 & 2 & 0 \end{bmatrix}$, and $d = [1 \ 5 \ -2]$.
- 7.5** Modify the function of the previous exercise so that it can accept either ordinary numeric arrays or cell arrays containing numeric values. Test your function by passing it the two arguments `a` and `b`, where $a = \begin{bmatrix} 1 & 4 \\ -2 & 3 \end{bmatrix}$, $b\{1\} = [1 \ 5 \ 2]$, and $b\{2\} = \begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix}$.
- 7.6** Create a structure array containing all of the information needed to plot a data set. At a minimum, the structure array should have the following fields.
- | | |
|-------------------------|--|
| <code>x_data</code> | <i>x</i> -data (one or more data sets in separate cells) |
| <code>y_data</code> | <i>y</i> -data (one or more data sets in separate cells) |
| <code>type</code> | linear, semilogx, and so on |
| <code>plot_title</code> | plot title |
| <code>x_label</code> | <i>x</i> -axis label |

y_label	<i>y</i> -axis label
x_range	<i>x</i> -axis range to plot
y_range	<i>y</i> -axis range to plot

You may add additional fields that would enhance your control of the final plot.

After this structure array is created, create a MATLAB function that accepts an array of this structure and produces one plot for each structure in the array. The function should apply intelligent defaults if some data fields are missing. For example, if the `plot_title` field is an empty matrix, then the function should not place a title on the graph. Think carefully about the proper defaults before starting to write your function!

To test your function, create a structure array containing the data for three plots of three different types and pass that structure array to your function. The function should correctly plot all three data sets in three different Figure Windows.

Input / Output Functions

In Chapter 2, we learned how to load and save MATLAB data using the `load` and `save` commands, and how to write out formatted data using the `fprintf` function. In this chapter, we will learn more about MATLAB's input/output (I/O) capabilities. First, we will learn about `textread`, a very useful function that was added to MATLAB 5.3. Then, we will spend a bit more time examining the `load` and `save` commands. Finally, we will look at the other file I/O options available in MATLAB.

Those readers familiar with C will find much of this material very familiar. Be careful, however, because there are subtle differences between MATLAB and C functions that can trip you up.

8.1 The `textread` Function

The `textread` command is new beginning with MATLAB 5.3. It is designed to read ASCII files that are formatted into columns of data, where each column can be of a different type. This command is *very* useful for importing tables of data printed out by other applications.

The form of the `textread` command is

```
[a,b,c,...] = textread(filename,format,n)
```

where `filename` is the name of the file to open, `format` is a string containing a description of the type of data in each column, and `n` is the number of lines to read. (If `n` is missing, the command reads to the end of the file.) The `format` string contains the same types of format descriptors as function `fprintf`. Note that the

number of output arguments must match the number of columns that you are reading.

For example, suppose that file `test_input.dat` contains the following data.

```
James Jones O+ 3.51 22 Yes
Sally Smith A+ 3.28 23 No
```

This data could be read into a series of arrays with the following function.

```
[first,last,blood,gpa,age,answer] = ...
    textread('test_input.dat','%s %s %s %f %d %s')
```

When this command is executed, the results are

```
> [first,last,blood,gpa,age,answer] = ...
    textread('test_input.dat','%s %s %s %f %d %s')

first =
    'James'
    'Sally'
last =
    'Jones'
    'Smith'
blood =
    'O+'
    'A+'
gpa =
    3.5100
    3.2800
age =
    22
    23
answer =
    'Yes'
    'No'
```

This function can also skip selected columns by adding an asterisk to the corresponding format descriptor (for example, `%*s`). For example, the following statement reads only the first name, last name, and gpa from the file.

```
> [first,last,gpa] = ...
    textread('test_input.dat','%s %s %*s %f %*d %*s')

first =
    'James'
    'Sally'
last =
    'Jones'
    'Smith'
gpa =
```

```
3.5100
3.2800
```

Function `textread` is much more useful and flexible than the `load` command. The `load` command assumes that all of the data in the input file is of a single type—it cannot support different types of data in different columns. In addition, it stores all of the data into a single array. In contrast, the `textread` function allows each column to go into a separate variable, which is *much* more convenient when working with columns of mixed data.

Function `textread` has a number of additional options that increase its flexibility. Consult the MATLAB on-line documentation for details of these options.

Tip
Use function `textread` to import ASCII data in column format from programs written in other languages or exported from applications such as spreadsheets.

8.2 More about the load and save Commands

The `save` command saves MATLAB workspace data to disk, and the `load` command loads data from disk into the workspace. The `save` command can save data either in a special binary format called a MAT-file or in an ordinary ASCII file. The form of the `save` command is

```
save filename [list of variables] [options]
```

The command `save all` by itself saves all of the data in the current workspace to a file named `matlab.mat` in the current directory. If a file name is included, the data will be saved in file “`filename.mat`”. If a list of variables is included, then only those particular variables will be saved.

The options supported by the `save` command are shown in Table 8.1.

Table 8.1 `save` Command Options

Option	Description
<code>-mat</code>	Save data in MAT-file format (default).
<code>-ascii</code>	Save data in space-separated ASCII format.
<code>-append</code>	Adds the specified variables to an existing MAT-file.
<code>-v4</code>	Save the MAT-file in a format readable by MATLAB version 4.

The `load` command can load data from MAT-files or in an ordinary ASCII file. The form of the `load` command is

```
save filename [options]
```

The command `load all` by itself loads all of the data in file `matlab.mat` into the current workspace. If a file name is included, the data will be loaded from that file name.

The options supported by the `load` command are shown in Table 8.2.

Although it is not immediately obvious, the `save` and `load` commands are the most powerful and useful I/O commands in MATLAB. Among their advantages are:

1. These commands are very easy to use.
2. MAT-files are *platform independent*. A MAT-file written on any type of computer that supports MATLAB can be read on any other computer supporting MATLAB. This format transfers freely among PCs, Macs, and many different versions of Unix.
3. MAT-files are relatively efficient users of disk space, and they store the full precision of every variable—no precision is lost due to conversion to and from ASCII format.
4. MAT-files preserve all of the information about each variable in the workspace, including its class, name, and whether or not it is global. All of this information is lost in other types of I/O. For example, suppose that the workspace contains the following information.

```
* whos
  Name      Size      Bytes  Class
  a          10x10    800  double array (global)
  ans        1x1       8  double array
  b          10x10    800  double array
  c          2x2       332  cell array
  string     1x16      32  char array
  student    1x3      2152 struct array

Grand total is 372 elements using 4124 bytes
```

If this workspace is saved with the command `save workspace.mat`, a file named `workspace.mat` will be created. When this file is loaded, all

Table 8.2 `load` Command Options

Option	Description
<code>-mat</code>	Treat file as a MAT-file (default if file extent is <code>mat</code>).
<code>-ascii</code>	Treat file as a space-separated ASCII file (default if file extent is <i>not</i> <code>mat</code>).

of the information will be restored, including the type of each item and whether or not it is global.

A disadvantage of these commands is that the MAT-file format is unique to MATLAB, and cannot be used to share data with other programs. The `-ascii` option can be used if you wish to share data with other programs, but it has serious limitations.

Good Programming Practice

Unless you must exchange data with non-MATLAB programs, always use the `load` and `save` commands to save data sets in MAT-file format. This format is efficient and transportable across MATLAB implementations, and it preserves all details of all MATLAB data types.

The `save -ascii` command will not save cell array or structure array data at all, and it converts string data to numbers before saving it. The `load -ascii` command will only load space-separated data with an equal number of elements on each row, and it will place all of the data into a single variable with the same name as the input file. If you need anything more elaborate (such as, saving and loading strings, cells, and structure arrays in formats suitable for exchanging with other programs), then it will be necessary to use the other file I/O commands and functions described in this chapter.

If the file name and the names of the variables to be loaded or saved are in strings, then you should use the function forms of these commands. For example, the following fragment of code asks the user for a file name and saves the workspace in that file.

```
filename = input('Enter save file name: ','s');
save (filename);
```

8.3 An Introduction to MATLAB File Processing

To use files within a MATLAB program, we need some way to select the desired file and to read from or write to it. MATLAB has a very flexible method for reading and writing files, whether they are on disk, magnetic tape, or some other device attached to the computer. This mechanism is known as the `file id` (sometimes known as `fid`). The file id is a number assigned to a file when it is opened, and it is used for all reading, writing, and control operations on that file. The file id is a positive integer. Two file ids are always open—file id 1 is the standard output device (`stdout`) and file id 2 is the standard error (`stderr`) device for the com-

puter on which MATLAB is executing. Additional file ids are assigned as files are opened and released as files are closed.

Several MATLAB statements can be used to control disk file input and output. The file I/O functions are summarized in Table 8.3.

File ids are assigned to disk files or devices using the `fopen` statement, and detached from them using the `fclose` statement. Once a file is attached to a file id using the `fopen` statement, we can read and write to that file using MATLAB file input and output statements. When we are through with the file, the `fclose` statement closes the file and makes the file id invalid. The `frewind` and `fseek` statements can be used to change the current reading or writing position in a file while it is open.

Data can be written to and read from files in two possible ways: as binary data or as formatted character data. Binary data consists of the actual bit patterns that are used to store the data in computer memory. Reading and writing binary data is very efficient, but a user cannot read the data stored in the file. Data in formatted files is translated into characters that can be read directly by a user. However, formatted I/O operations are slower and less efficient than binary I/O operations. We will discuss both types of I/O operations later in this chapter.

Table 8.3 MATLAB Input / Output Statements

Category	Function	Description
Load / Save workspace	<code>load</code>	Load workspace.
	<code>save</code>	Save workspace.
File opening and closing	<code>fopen</code>	Open file.
	<code>fclose</code>	Close file.
Binary I/O	<code>fread</code>	Read binary data from file.
	<code>fwrite</code>	Write binary data to file.
Formatted I/O	<code>fscanf</code>	Read formatted data from file.
	<code>fprintf</code>	Write formatted data to file.
	<code>fgetl</code>	Read line from file; discard newline character.
	<code>fgets</code>	Read line from file; keep newline character.
File positioning, status, and miscellaneous	<code>delete</code>	Delete file.
	<code>exist</code>	Check for the existence of a file.
	<code>ferror</code>	Inquire file I/O error status.
	<code>feof</code>	Test for end-of-file.
	<code>fseek</code>	Set file position.
	<code>ftell</code>	Check file position.
	<code>frewind</code>	Rewind file.
Temporary files	<code>tempdir</code>	Get temporary directory name.
	<code>tempname</code>	Get temporary file name.

8.4 File Opening and Closing

The file opening and closing functions, `fopen` and `fclose`, are described in this section.

8.4.1 The `fopen` Function

The `fopen` function opens a file and returns a file id number for use with the file. The basic forms of this statement are

```
fid = fopen(filename,permission)
[fid, message] = fopen(filename,permission)
[fid, message] = fopen(filename,permission,format)
```

where `filename` is a string specifying the name of the file to open, `permission` is a character string specifying the mode in which the file is opened, and `format` is an optional string specifying the numeric format of the data in the file. If the open is successful, `fid` will contain a positive integer after this statement is executed, and `message` will be an empty string. If the open fails, `fid` will contain a -1 after this statement is executed, and `message` will be a string explaining the error. If a file is opened for reading and it is not in the current directory, MATLAB will search for it along the MATLAB search path.

The possible permission strings are shown in Table 8.4.

On some platforms such as PCs, it is important to distinguish between text files and binary files. If a file is to be opened in text mode, then a `t` should be

Table 8.4 `fopen` File Permissions

File Permission	Meaning
'r'	Open an existing file for reading only (default).
'r+'	Open an existing file for reading and writing.
'w'	Delete the contents of an existing file (or create a new file) and open it for writing only.
'w+'	Delete the contents of an existing file (or create a new file) and open it for reading and writing.
'a'	Open an existing file (or create a new file) and open it for writing only, appending to the end of the file.
'a+'	Open an existing file (or create a new file) and open it for reading and writing, appending to the end of the file.
'w'	Write without automatic flushing (special command for tape drives).
'A'	Append without automatic flushing (special command for tape drives).

added to the permissions string (for example, 'rt' or 'rt+'). If a file is to be opened in binary mode, a b may be added to the permissions string (for example, 'rb'). This is not actually required since files are opened in binary mode by default. This distinction between text and binary files does not exist on Unix computers, so the t or b is never needed on those systems.

The *format* string in the fopen function specifies the numeric format of the data stored in the file. This string is only needed when transferring files between computers with incompatible numeric data formats, so it is rarely used. A few of the possible numeric formats are shown in Table 8.5; see the MATLAB *Language Reference Manual* for a complete list of possible numeric formats.

There are also two forms of this function that provide information rather than open files. The function

```
fids = fopen('all')
```

returns a row vector containing a list of all file id's for currently open files (except for `stdout` and `stderr`). The number of elements in this vector is equal to the number of open files. The function

```
[filename, permission, format] = fopen(fid)
```

returns the file name, permission string, and numeric format for an open file specified by file id.

Some examples of correct fopen functions follow.

8.4.1.1 Case I: Opening a Binary File for Input

The following function opens a file named `example.dat` for binary input only.

```
fid = fopen('example.dat','r')
```

The permission string is 'r', indicating that the file is to be opened for reading only. The string could have been 'rb', but this is not required because binary access is the default case.

Table 8.5 Selected fopen Format Strings

File Permission	Meaning
'native' or 'n'	Numeric format for the machine MATLAB is executing on (default).
'ieee-le' or 'l'	IEEE floating point with little-endian byte ordering.
'ieee-be' or 'b'	IEEE floating point with big-endian byte ordering.
'ieee-le.164' or 'a'	IEEE floating point with little-endian byte ordering and 64-bit long data type.
'ieee-le.b64' or 's'	IEEE floating point with big-endian byte ordering and 64-bit long data type.

8.4.1.2 Case 2: Opening a File for Text Output

The functions below open a file named `outdat` for text output only.

```
fid = fopen('outdat', 'wt')
or
fid = fopen('outdat', 'at')
```

The '`wt`' permissions string specifies that the file is a new text file; if it already exists, then the old file will be deleted, and a new empty file will be opened for writing. This is the proper form of the `fopen` function for an *output file* if we want to replace preexisting data.

The '`at`' permissions string specifies that we want to append to an existing text file. If it already exists, then it will be opened and new data will be appended to the currently existing information. This is the proper form of the `fopen` function for an *output file* if we don't want to replace preexisting data.

8.4.1.3 Case 3: Opening a Binary File for Read / Write Access

The following function opens a file named `junk` for binary input and output.

```
fid = fopen('junk', 'r+')
```

The following function also opens the file for binary input and output.

```
fid = fopen('junk', 'w+')
```

The difference between the first and the second statements is that the first statement required the file to exist before it is opened, while the second statement will delete any preexisting file.

Common Programming Practice

Always be careful to specify the proper permissions in `fopen` statements, depending on whether you are reading from or writing to a file. This practice will help prevent errors such as accidentally overwriting data files that you want to keep.

It is important to check for errors after you attempt to open a file. If the `fid` is `-1`, then the file failed to open. You should report this problem to the user, and allow him or her to either select another file or else quit the program.

Common Programming Practice

Always check the status after a file open operation to make sure that it is successful. If the file open fails, tell the user and provide a way to recover from the problem.

8.4.2 The `fclose` Function

The `fclose` function closes a file. Its form is

```
status = fclose(fid)
status = fclose('all')
```

where `fid` is a file id and `status` is the result of the operation. If the operation is successful, `status` will be 0, and if it is unsuccessful, `status` will be -1.

The form `status = fclose('all')` closes all open files except for `stdout` (`fid = 1`) and `stderr` (`fid = 2`). It returns a status of 0 if all files close successfully, and -1 otherwise.

8.5 Binary I/O Functions

The binary I/O functions, `fwrite` and `fread`, are described in this section.

8.5.1 The `fwrite` Function

The `fwrite` function writes binary data in a user-specified format to a file. Its form is

```
count = fwrite(fid,array,precision)
count = fwrite(fid,array,precision,skip)
```

where `fid` is the file id of a file opened with the `fopen` function, `array` is the array of values to write out, and `count` is the number of values written to the file.

MATLAB writes out data in *column order*, which means that the entire first column is written out, followed by the entire second column, and so forth. For

example, if `array = [1 2; 3 4; 5 6]`, then the data will be written out in the order

1, 3, 5, 2, 4, 6.

The optional `precision` string specifies the format in which the data will be output. MATLAB supports both platform-independent precision strings, which are the same for all computers on which MATLAB runs, and platform-dependent precision strings, which vary among different types of computers. You should only use the platform-independent strings, and those are the only forms presented in this book.

For convenience, MATLAB accepts some C and Fortran data-type equivalents for the MATLAB precision strings. If you are a C or Fortran programmer, you may find it more convenient to use the names of the data types in the language that you are most familiar with.

The possible platform-independent precisions are presented in Table 8.6. All of these precisions work in units of bytes, except for '`bitN`' or '`ubitN`', which work in units of bits.

Table 8.6 MATLAB Precision Strings

MATLAB Precision String	C / Fortran Equivalent	Meaning
'char'	'char*1'	8-bit characters
'schar'	'signed char'	8-bit signed character
'uchar'	'unsigned char'	8-bit unsigned character
'int8'	'integer*1'	8-bit integer
'int16'	'integer*2'	16-bit integer
'int32'	'integer*4'	32-bit integer
'int64'	'integer*8'	64-bit integer
'uint8'	'integer*1'	8-bit unsigned integer
'uint16'	'integer*2'	16-bit unsigned integer
'uint32'	'integer*4'	32-bit unsigned integer
'uint64'	'integer*8'	64-bit unsigned integer
'float32'	'real*4'	32-bit floating point
'float64'	'real*8'	64-bit floating point
'bitN'		N-bit signed integer, $1 \leq N \leq 64$
'ubitN'		N-bit unsigned integer, $1 \leq N \leq 64$

The optional argument *skip* specifies the number of bytes to skip in the output file before each write. This option is useful for placing values at certain points in fixed-length records. Note that if *precision* is a bit format like 'bitN' or 'ubitN', *skip* is specified in bits instead of bytes.

8.5.2 The fread Function

The fread function reads binary data in a user-specified format from a file. Its form is

```
[array, count] = fread(fid, size, precision)
[array, count] = fread(fid, size, precision, skip)
```

where *fid* is the file id of a file opened with the fopen function, *size* is the number of values to read, *array* is the array to contain the data, and *count* is the number of values read from the file.

The optional argument *size* specifies the amount of data to be read from the file. There are three versions of this argument.

- *n* Read exactly *n* values. After this statement, *array* will be a column vector containing *n* values read from the file.

- Inf Read until the end of the file. After this statement, array will be a column vector containing all of the data until the end of the file.
- [n m] Read exactly $n \times m$ values, and format the data as an $n \times m$ array.

If `fread` reaches the end of the file and the input stream does not contain enough bits to write out a complete array element of the specified precision, `fread` pads the last byte or element with zero bits until the full value is obtained. If an error occurs, reading is done up to the last full value.

The arguments `precision` and `size` have the same meaning for `fread` as they have for `fwrite`.

Example 8.1—Writing and Reading Binary Data

The script file shown in this example creates an array containing 10,000 random values, opens a user-specified file for writing only, writes the array to disk in 64-bit floating-point format, and closes the file. It then opens the file for reading and reads the data back into a 100×100 array. It illustrates the use of binary I/O operations.

```
% Script file: binary_io.m
%
% Purpose:
%   To illustrate the use of binary i/o functions.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   ===       =====          =====
%   12/19/98  S. J. Chapman  Original code
%
% Define variables:
%   count    -- Number of values read / written
%   fid      -- File id
%   filename -- File name
%   in_array -- Input array
%   msg      -- Open error message
%   out_array -- Output array
%   status   -- Operation status

% Prompt for file name
filename = input('Enter file name: ','s');

% Generate the data array
out_array = randn(1,10000);

% Open the output file for writing.
[fid,msg] = fopen(filename,'w');
```

```

% Was the open successful?
if fid > 0

    % Write the output data.
    count = fwrite(fid,out_array,'float64');

    % Tell user
    disp([int2str(count) ' values written...']);

    % Close the file
    status = fclose(fid);

else

    % Output file open failed. Display message.
    disp(msg);

end

% Now try to recover the data. Open the
% file for reading.
[fid,msg] = fopen(filename,'r');

% Was the open successful?
if fid > 0

    % Read the input data.
    [in_array, count] = fread(fid,[100 100],'float64');

    % Tell user
    disp([int2str(count) ' values read...']);

    % Close the file
    status = fclose(fid);

else

    % Input file open failed. Display message.
    disp(msg);

end

```

When this program is executed, the result is

```

> binary_io
Enter file name: testfile
10000 values written...
10000 values read...

```

An 80,000-byte file named `testfile` was created in the current directory. This file is 80,000 bytes long because it contains 10,000 64-bit values, and each value occupies 8 bytes.

Quiz 8.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 8.1 through 8.5. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. Why is the `textread` function especially useful for reading data created by programs written in other languages?
2. What are the advantages and disadvantages of saving data in a MAT-file?
3. What MATLAB functions are used to open and close files? What is the difference between opening a binary file and opening a text file?
4. Write the MATLAB statement to open a preexisting file named `myinput.dat` for appending new text data.
5. Write the MATLAB statements required to open an unformatted binary input file named `input.dat` for reading only. Check to see if the file exists and generate an appropriate error message if it does not.

For questions 6 and 7, determine whether the MATLAB statements are correct or not. If they are in error, specify what is wrong with them.

6. `fid = fopen('file1','rt');`
`array = fread(fid,Inf)`
`fclose(fid);`
7. `fid = fopen('file1','w');`
`x = 1:10;`
`count = fwrite (fid, x);`
`fclose (fid);`
`fid = fopen ('file1','r');`
`array = fread(fid,[2 Inf])`
`fclose(fid);`

8.6 Formatted I/O Functions

The formatted I/O functions are described in this section.

8.6.1 The `fprintf` Function

The `fprintf` function writes formatted data in a user-specified format to a file. Its form is

```
count = fprintf(fid,format,val1,val2,...)
fprintf(format,val1,val2,...)
```

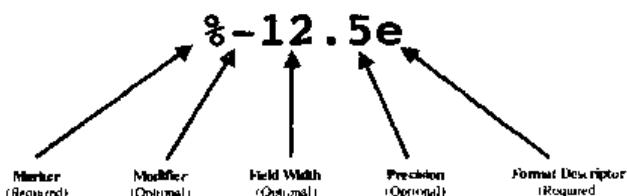


Figure 8.1 The components of a typical format specifier.

where *fid* is the file id of a file to which the data will be written, and *format* is the format string controlling the appearance of the data. If *fid* is missing, the data is written to the standard output device (the Command Window). This is the form of `fprintf` that we have been using since Chapter 2.

The format string specifies the alignment, significant digits, field width, and other aspects of output format. It can contain ordinary alphanumeric characters along with special sequences of characters that specify the exact format in which the output data will be displayed. The structure of a typical format specifier is shown in Figure 8.1. A single % character always marks the beginning of a format specifier—if an ordinary % sign is to be printed out, then it must appear in the format string as %. After the % character, the format can have a flag, a field width and precision specifier, and a conversion specifier. The % character and the conversion specifier are always required in any format, while the flag, field width, and precision specifiers are optional.

The possible conversion specifiers are listed in Table 8.7, and the possible flags are listed in Table 8.8. If a field width and precision are specified in a for-

Table 8.7 Format Conversion Specifiers for `fprintf`

Specifier	Description
%c	Single character
%d	Decimal notation (signed)
%e	Exponential notation (using a lowercase e as in 3.1416e+00)
%E	Exponential notation (using an uppercase E as in 3.1416E+00)
%f	Fixed-point notation
%g	The more compact of %e or %f; insignificant zeros do not print
%G	Same as %g, but using an uppercase E
%o	Octal notation (unsigned)
%s	String of characters
%u	Decimal notation (unsigned)
%x	Hexadecimal notation (using lowercase letters a-f)
%X	Hexadecimal notation (using uppercase letters A-F)

Table 8.8 Format Flags

Flag	Description
Minus sign (-)	Left-justifies the converted argument in its field (Example: %-5.2d). If this flag is not present, the argument is right-justified.
+	Always print a + or - sign (Example: %+5.2d).
0	Pad argument with leading zeros instead of blanks (Example: %05.2d).

mat, then the number before the decimal point is the field width, which is the number of characters used to display the number. The number after the decimal point is the precision, which is the minimum number of significant digits to display after the decimal point.

In addition to ordinary characters and formats, certain special escape characters can be used in a format string. These special characters are listed in Table 8.9.

8.6.2 Understanding Format Conversion Specifiers

The best way to understand the wide variety of format conversion specifiers is by example, so we will now present several examples along with their results.

8.6.2.1 Case 1: Displaying Decimal Data

Decimal (integer) data can be displayed with the %d format conversion specifier. The d may be preceded by a flag and a field width and precision specifier, if desired. If used, the precision specifier specifies a minimum number of

Table 8.9 Escape Characters in Format Strings

Escape Sequences	Description
\n	New line
\t	Horizontal tab
\b	Backspace
\r	Carriage return
\f	Form feed
\\\	Print an ordinary backslash (\) symbol
' or ''	Print an apostrophe or single quote
%	Print an ordinary percent (%) symbol

digits to display. If there are not enough digits, leading zeros will be added to the number.

Function	Result	Comment
<code>fprintf('%d\n', 123)</code>	---- ---- 123	Display the number using as many characters as required. For the number 123, three characters are required.
<code>fprintf('%6d\n', 123)</code>	---- ---- 123	Display the number in a 6-character-wide field. By default the number is <i>right justified</i> in the field.
<code>fprintf('%6.4d\n', 123)</code>	---- ---- 0123	Display the number in a 6-character-wide field using a minimum of 4 characters. By default the number is <i>right justified</i> in the field.
<code>fprintf('%-6.4d\n', 123)</code>	---- ---- 0123	Display the number in a 6-character-wide field using a minimum of 4 characters. The number is <i>left justified</i> in the field.
<code>fprintf('%+6.4d\n', 123)</code>	---- ---- +0123	Display the number in a 6-character-wide field using a minimum of 4 characters plus a sign character. By default the number is <i>right justified</i> in the field.

If a nondecimal number is displayed with the %d conversion specifier, the specifier will be ignored and the number will be displayed in exponential format. For example

```
fprintf('%6d\n', 123.4)
```

produces the result 1.234000e+002.

8.6.2.2 Case 2: Displaying Floating-Point Data

Floating-point data can be displayed with the %e, %f, or %g format conversion specifiers. They may be preceded by a flag and a field width and precision specifier, if desired. If the specified field width is too small to display the number, it is ignored. Otherwise, the specified field width is used.

Function	Result	Comment
<code>fprintf('%f\n', 123.4)</code>	---- ---- 123.400000	Displays the number using as many characters as required. The default case for %f is to display 6 digits after the decimal place.
<code>fprintf('%8.2f\n', 123.4)</code>	---- ---- 123.40	Displays the number in an 8-character wide field, with two places after the decimal point. The number is <i>right justified</i> in the field.
<code>fprintf('%4.2f\n', 123.4)</code>	---- ---- 123.40	Displays the number in a 6-character-wide field. The width specification was ignored because it was too small to display the number.

Function	Result	Comment
<code>fprintf('%10.2e\n', 123.4)</code>	----- ----- 1.23e+002	Displays the number in exponential format in a 10-character-wide field using 2 decimal places. By default the number is <i>right justified</i> in the field.
<code>fprintf('%10.2E\n', 123.4)</code>	----- ----- 1.23E+002	The same, but with a capital E for the exponent.

8.6.2.3 Case 3: Displaying Character Data

Character data can be displayed with the %c or %s format conversion specifiers. They may be preceded by field width specifier, if desired. If the specified field width is too small to display the number, it is ignored. Otherwise, the specified field width is used.

Function	Result	Comment
<code>fprintf('%c\n', 's')</code>	----- ----- s	Displays a single character.
<code>fprintf('%s\n', 'string')</code>	----- ----- string	Displays the character string.
<code>fprintf('%8s\n', 'string')</code>	----- ----- string	Displays the character string in an 8-character-wide field. By default the string is <i>right justified</i> in the field.
<code>fprintf('%-8s\n', 'string')</code>	----- ----- string	Displays the character string in an 8-character-wide field. The string is <i>left justified</i> in the field.

8.6.3 How Format Strings Are Used

The `fprintf` function contains a format string followed by zero or more values to print out. When the `fprintf` function is executed, the list of output values associated with the `fprintf` function is processed together with the format string. The function begins at the left end of the variable list and the left end of the format string, and scans from left to right, associating the first value in the output list with the first format descriptor in the format string, and so forth. The variables in the output list must be of the same type and in the same order as the format descriptors in the format, or unexpected results may be produced. For example, if we attempt to display a floating-point number such as 123.4 with a %c or %d descriptor, the descriptor is ignored totally and the number is printed in exponential notation.

Make sure that there is a one-to-one correspondence between the types of the data in a `fprintf` function and the types of the format conversion specifiers

in the associated format string, or your program will produce unexpected results.

As the program moves from left to right through the variable list of a `fprint` function, it also scans from left to right through the associated format string. Format strings are scanned according to the following rules.

1. *Format strings are scanned in order from left to right.* The first format conversion specifier in the format string is associated with the first value in the output list of the `fprint` function, and so forth. The type of each format conversion specifier must match the type of the data being output. In the following example, specifier `%d` is associated with variable `a`, `%f` with variable `b`, and `%s` with variable `c`. Note that the specifier types match the data types.

```
a = 10; b = pi; c = 'Hello';
fprintf('Output: %d %f %s\n',a,b,c);
```

2. If the scan reaches the end of the format string before the `fprintf` function runs out of values, the program starts over *at the beginning of the format string*. For example, the statements

```
a = [10 20 30 40];
fprintf('Output = %4d %4d\n',a);
```

will produce the output

```
Output =    10    20
Output =    30    40
```

When the function reaches the end of the format string after printing `a(2)`, it starts over at the beginning of the string to print `a(3)` and `a(4)`.

3. If the `fprintf` function runs out of variables before the end of the format string, *the use of the format string stops at the first format conversion specifier without a corresponding variable, or at the end of the format string, whichever comes first*. For example, the statements

```
a = 10; b = 15; c = 20;
fprintf('Output = %4d\nOutput = %4.1f\n',a,b,c);
```

will produce the output

```
Output =    10
Output = 15.0
Output =    20
Output = >
```

The use of the format string stops at `%4.1f`, which is the first unmatched format conversion specifier. On the other hand, the statements

```
voltage = 20;
fprintf('Voltage = %6.2f kV.\n',voltage);
```

will produce the output

```
Voltage = 20.00 kV.
```

since there are no unmatched format conversion specifiers, and the use of the format stops at the end of the format string.

Example 8.2—Generating a Table of Information

A good way to illustrate the use of `fprintf` functions is to generate and print out a table of data. The example script file that follows generates the square roots, squares, and cubes of all integers between 1 and 10, and presents the data in a table with appropriate headings.

```
% Script file: table.m
%
% Purpose:
%   To create a table of square roots, squares, and
%   cubes.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   =====    ======      =====
%   12/20/98  S. J. Chapman  Original code
%
% Define variables:
%   cube      -- Cubes
%   ii        -- Index variable
%   square    -- Squares
%   square_root -- Square roots
%   out       -- Output array

% Print the title of the table.
fprintf(' Table of Square Roots, Squares, and Cubes\n\n');

% Print column headings
fprintf(' Number  Square Root      Square      Cube\n');
fprintf(' -----  ======      =====      =====\n');

% Generate the required data
ii = 1:10;
square_root = sqrt(ii);
square = ii.^2;
cube = ii.^3;

% Create the output array
out = [ii' square_root' square' cube'];
```

```

% Print the data
for ii = 1:10
    fprintf (' %2d %11.4f %6d %8d\n',out(ii,:));
end

```

When this program is executed, the result is

```

> table
Table of Square Roots, Squares, and Cubes

Number   Square Root      Square      Cube
=====  ======  ======  =====
    1       1.0000        1          1
    2       1.4142        4          8
    3       1.7321        9         27
    4       2.0000       16         64
    5       2.2361       25        125
    6       2.4495       36        216
    7       2.6458       49        343
    8       2.8284       64        512
    9       3.0000       81        729
   10      3.1623      100       1000

```

8.6.4 The fscanf Function

The `fscanf` function reads formatted data in a user-specified format from a file. Its form is

```

array = fscanf(fid,format)
[array, count] = fscanf(fid,format,size)

```

where `fid` is the file id of a file from which the data will be read, `format` is the format string controlling how the data is read, and `array` is the array that receives the data. The output argument `count` returns the number of values read from the file.

The optional argument `size` specifies the amount of data to be read from the file. There are three versions of this argument.

- `n` Read exactly `n` values. After this statement, `array` will be a column vector containing `n` values read from the file.
- `Inf` Read until the end of the file. After this statement, `array` will be a column vector containing all of the data until the end of the file.
- `[n m]` Read exactly $n \times m$ values, and format the data as an $n \times m$ array.

The format string specifies the format of the data to be read. It can contain ordinary characters along with format conversion specifiers. The `fscanf` func-

tion compares the data in the file with the format conversion specifiers in the format string. As long as the two match, `fscanf` converts the value and stores it in the output array. This process continues until the end of the file or until the amount of data in `size` has been read, whichever comes first.

If the data in the file does not match the format conversion specifiers, the operation of `fscanf` stops immediately.

The format conversion specifiers for `fscanf` are basically the same as those for `fprintf`. The most common specifiers are shown in Table 8.10.

To illustrate the use of `fscanf`, we will attempt to read a file called `x.dat` containing the following values on two lines.

```
10.00 20.00
30.00 40.00
```

1. If the file is read with the statement

```
[z, count] = fscanf(fid, '%f');
```

variable `z` will be the column vector $\begin{bmatrix} 10 \\ 20 \\ 30 \\ 40 \end{bmatrix}$ and `count` will be 4.

2. If the file is read with the statement

```
[z, count] = fscanf(fid, '%f', [2 2]);
```

variable `z` will be the array $\begin{bmatrix} 10 & 30 \\ 20 & 40 \end{bmatrix}$ and `count` will be 4.

3. Next, let us try to read this file as decimal values. If the file is read with the statement

```
[z, count] = fscanf(fid, '%d', Inf);
```

variable `z` will be the single value 10 and `count` will be 1. This happens

Table 8.10 Format Conversion Specifiers for `fscanf`

Specifier	Description
%c	Read a single character. This specifier reads any character including blanks, new lines, and so forth.
%Nc	Read N characters.
%d	Read a decimal number (ignores blanks).
%e %f %g	Read a floating-point number (ignores blanks).
%i	Read a signed integer (ignores blanks).
%s	Read a string of characters. The string is terminated by blanks or other special characters such as new lines.

because the decimal point in the 10.00 does not match the format conversion specifier, and `fscanf` stops at the first mismatch.

4. If the file is read with the statement

```
[z, count] = fscanf(fid, '%d.%d', [1 Inf]);
```

variable `z` will be the row vector [10 0 20 0 30 0 40 0] and `count` will be 8. This happens because the decimal point is now matched in the format conversion specifier and the numbers on either side of the decimal point are interpreted as separate integers!

5. Now let us try to read the file as individual characters. If the file is read with the statement

```
[z, count] = fscanf(fid, '%c');
```

variable `z` will be a row vector containing every character in the file, including all spaces and newline characters! Variable `count` will be equal to the number of characters in the file.

6. Finally, let us try to read the file as a character string. If the file is read with the statement

```
[z, count] = fscanf(fid, '%s');
```

variable `z` will be a row vector containing the 20 characters 10.0020.0030.0040.00, and `count` will be 4. This happens because the string specifier ignores white space, and the function found four separate strings in the file.

8.6.5 The `fgetl` Function

The `fgetl` function reads the next line *excluding the end-of-line characters* from a file as a character string. Its form is

```
line = fgetl(fid)
```

where `fid` is the file id of a file from which the data will be read, and `line` is the character array that receives the data. If `fgetl` encounters the end of a file, the value of `line` is set to -1.

8.6.6 The `fgets` Function

The `fgets` function reads the next line *including the end-of-line characters* from a file as a character string. Its form is

```
line = fgets(fid)
```

where `fid` is the file id of a file from which the data will be read, and `line` is the character array that receives the data. If `fgets` encounters the end of a file, the value of `line` is set to -1.

8.7 Comparing Formatted and Binary I/O Functions

Formatted I/O operations produce formatted files. A **formatted file** contains recognizable characters, numbers, and so on, stored as ASCII text. These files are easy to distinguish because we can see the characters and numbers in the file when we display them on the screen or print them on a printer. However, to use data in a formatted file, a MATLAB program must translate the characters in the file into the internal data format used by the computer. Format conversion specifiers provide the instructions for this translation.

Formatted files have the advantages that we can readily see what sort of data they contain, and it is easy to exchange data between different types of programs using them. However, they also have disadvantages. A program must do a good deal of work to convert a number between the computer's internal representation and the characters contained in the file. All of this work is just wasted effort if we are going to be reading the data back into another MATLAB program. Also, the internal representation of a number usually requires much less space than the corresponding representation of the number found in a formatted file. For example, the internal representation of a 64-bit floating-point value requires 8 bytes of space. The character representation of the same value would be $\pm d. d d d d d d d d E \pm e e$, which requires 21 bytes of space (one byte per character). Thus, storing data in character format is inefficient and wasteful of disk space.

Unformatted files (or **binary files**) overcome these disadvantages by copying the information from the computer's memory directly to the disk file with no conversions at all. Since no conversions occur, no computer time is wasted formatting the data. In MATLAB, binary I/O operations are *much* faster than formatted I/O operations because there is no conversion. Furthermore, the data occupies a much smaller amount of disk space. On the other hand, unformatted data cannot be examined and interpreted directly by humans. In addition, it usually cannot be moved between different types of computers, because those types of computers have different internal ways to represent integers and floating-point values.

Formatted and unformatted files are compared in Table 8.11. In general, formatted files are best for data that people must examine or data that may have to be moved between different programs on different computers. Unformatted files are best for storing information that will not need to be examined by human beings, and that will be created and used on the same type of computer. Under those circumstances, unformatted files are both faster and occupy less disk space.

Good Programming Practice

Use formatted files to create data that must be readable by humans or that must be transferable between programs on computers of different types. Use unformatted files to efficiently store large quantities of data that do not have to be directly examined and that will remain on only one type of computer. Also, use unformatted files when I/O speed is critical.

Table 8.11 Comparison of Formatted and Unformatted Files

Formatted Files	Unformatted Files
Can display data on output devices.	Cannot display data on output devices.
Can easily transport data between different computers.	Cannot easily transport data between computers with different internal data representations.
Require a relatively large amount of disk space.	Require relatively little disk space.
Slow: requires a lot of computer time.	Fast: requires little computer time.
Truncation or rounding errors possible in formatting.	No truncation or rounding errors.

Example 8.3—Comparing Formatted and Binary I/O

The program shown in this example compares the time required to read and write a 10,000-element array using both formatted and binary I/O operations. Note that each operation is repeated 10 times and the average time is reported.

```
% Script file: compare.m
%
% Purpose:
%   To compare binary and formatted I/O operations.
%   This program generates an array of 10,000 random
%   values and writes it to disk both as a binary and
%   as a formatted file.
%
% Record of revisions:
%   Date      Programmer          Description of change
%   ----      ======          =====
%   12/19/98    S. J. Chapman    Original code
%
% Define variables:
%   count      -- Number of values read / written
%   fid        -- File id
%   in_array   -- Input array
%   msg        -- Open error message
%   out_array  -- Output array
%   status     -- Operation status
%   time       -- Elapsed time in seconds

%%%%%%%%%%%%%
% Generate the data array.
%%%%%%%%%%%%%
out_array = randn(1,10000);
```

```
%%%%%%  
% First, time the binary output operation.  
%%%%%  
% Reset timer  
tic;  
  
% Loop for 10 times  
for ii = 1:10  
  
    % Open the binary output file for writing.  
    [fid,msg] = fopen('unformatted.dat','w');  
  
    % Write the data  
    count = fwrite(fid,out_array,'float64');  
  
    % Close the file  
    status = fclose(fid);  
  
end  
  
% Get the average time  
time = toc / 10;  
fprintf ('Write time for unformatted file = %6.3f\n',time);  
  
%%%%%  
% Next, time the formatted output operation.  
%%%%%  
% Reset timer  
tic;  
  
% Loop for 10 times  
for ii = 1:10  
  
    % Open the formatted output file for writing.  
    [fid,msg] = fopen('formatted.dat','wt');  
  
    % Write the data  
    count = fprintf(fid,'%23.15e\n',out_array);  
  
    % Close the file  
    status = fclose(fid);  
end  
  
% Get the average time  
time = toc / 10;  
fprintf ('Write time for formatted file = %6.3f\n',time);  
  
%%%%%  
% Time the binary input operation.  
%%%%%  
% Reset timer  
tic;
```

```

% Loop for 10 times
for ii = 1:10

    % Open the binary file for reading.
    [fid,msg] = fopen('unformatted.dat','r');

    % Read the data
    [in_array, count] = fread(fid,Inf,'float64');

    % Close the file
    status = fclose(fid);

end

% Get the average time
time = toc / 10;
fprintf ('Read time for unformatted file = %6.3f\n',time);

%%%%%%%%%%%%%
% Time the formatted input operation.
%%%%%%%%%%%%%
% Reset timer
tic;

% Loop for 10 times
for ii = 1:10

    % Open the formatted file for reading.
    [fid,msg] = fopen('unformatted.dat','rt');

    % Read the data
    [in_array, count] = fscanf(fid,'%f',Inf);

    % Close the file
    status = fclose(fid);

end

% Get the average time
time = toc / 10;
fprintf ('Read time for unformatted file = %6.3f\n',time);

```

When this program is executed in a 733 MHz Pentium running Windows NT 2000 Professional, the results are:

```

> compare
Write time for unformatted file = 0.002
Write time for formatted file = 0.132
Read time for unformatted file = 0.002
Read time for formatted file = 0.157

```

The files written to disk are shown below.

```
D:\book\matlab\chap8>dir *.dat
Volume in drive D is Data
Volume Serial Number is CC1C-2FAA
Directory of D:\book\matlab\chap8

13/06/2001  04:24p      <DIR> .
13/06/2001  04:24p      <DIR> ..
13/06/2001  05:26p          250,000 formatted.dat
13/06/2001  05:26p          80,000 unformatted.dat
                23 File(s)       330,000 bytes
                2 Dir(s)        206,955,520 bytes free
```

Note that the write time for the formatted file was 60 times slower than the write time for the unformatted file, and the read time for the formatted file was more than 75 times slower than the read time for the unformatted file. Furthermore, the formatted file was 3 times larger than the unformatted file.

It is clear from these results that unless you *really* need formatted data, binary I/O operations are the preferred way to save data in MATLAB.

Quiz 8.2

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 8.6 and 8.7. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. What is the difference between unformatted (binary) and formatted I/O operations?
2. When should formatted I/O be used? When should unformatted I/O be used?
3. Write the MATLAB statements required to create a table that contains the sine and cosine of x for $x = 0, 0.1\pi, \dots, \pi$. Be sure to include a title and label on the table.

For questions 4 and 5, determine whether the MATLAB statements are correct or not. If they are in error, specify what is wrong with them.

4.

```
a = 2*pi;
b = 6;
c = 'hello';
fprintf(fid,'%s %d %g\n',a,b,c);
```
5.

```
data1 = 1:20;
data2 = 1:20;
fid = fopen('xxx','w+');
fwrite(fid,data1);
fprintf(fid,'%g\n',data2);
```

8.8 File Positioning and Status Functions

As we stated previously, MATLAB files are sequential—they are read in order from the first record in the file to the last record in the file. However, we sometimes need to read a piece of data more than once or to process a whole file more than once during a program. How can we skip around within a sequential file?

The MATLAB function `exist` can determine whether or not a file exists before it is opened. There are two functions to tell us where we are within a file once it is opened: `f.eof` and `f.tell`. In addition, there are two functions to help us move around within the file: `frewind` and `fseek`.

Finally, MATLAB includes a function `ferror` that provides a detailed description of cause of I/O errors when they occur. We will now explore these six functions, looking at `ferror` first, since it can be used with all of the other functions.

8.8.1 The `exist` Function

The MATLAB function `exist` checks for the existence of a variable in a workspace, a built-in function, or a file in the MATLAB search path. The forms of the `ferror` function are

```
ident = exist('item');
ident = exist('item', 'kind');
```

If 'item' exists, this function returns a value based on its type. The possible results are shown in Table 8.12.

The second form of the `exist` function restricts the search for an item to a specified kind. The legal types are '`var`', '`file`', '`builtin`', and '`dir`'.

The `exist` function is very important because we can use it to check for the existence of a file before it is overwritten by `fopen`. The permissions '`w`' and

Table 8.12 Values Returned by the `exist` Function

Value	Meaning
0	Item not found.
1	Item is a variable in the current workspace.
2	Item is an m-file or a file of unknown type.
3	Item is a MEX file.
4	Item is a MDL file.
5	Item is a built-in function.
6	Item is a pcode file.
7	Item is a directory.

'wt' delete the contents of an existing file when they open it. Before a programmer allows fopen to delete the file, he or she should check with the user to confirm that the file really should be deleted.

Example 8.4—Opening an Output File

The program shown gets an output file name from the user and checks to see if it exists. If it exists, the program checks to see if the user wants to delete the existing file or to append the new data to it. If the file does not exist, then the program simply opens the output file.

```
% Script file: output.m
%
% Purpose:
%   To demonstrate opening an output file properly.
%   This program checks for the existence of an output
%   file. If it exists, the program checks to see if
%   the old file should be deleted, or if the new data
%   should be appended to the old file.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   ----      ======          =====
%   11/29/98   S. J. Chapman  Original code
%
% Define variables:
%   fid        -- File id
%   out_filename -- Output file name
%   yn         -- Yes/No response

% Get the output file name.
out_filename = input('Enter output filename: ','s');

% Check to see if the file exists.
if exist(out_filename,'file')

    % The file exists
    disp('Output file already exists.');
    yn = input('Keep existing file? (y/n) ','s');

    if yn == 'n'
        fid = fopen(out_filename,'wt');
    else
        fid = fopen(out_filename,'at');
    end

else

    % File doesn't exist
    fid = fopen(out_filename,'wt');
end
```

```
% Output data
fprintf(fid, '%s\n', date);

% Close file
fclose(fid);
```

When this program is executed, the results are

```
> output
Enter output filename: xxxx          (Non-existent file)
> type xxxx
29-Nov-1998

> output
Enter output filename: xxxx
Output file already exists.
Keep existing file? (y/n) y      (Keep current file)
> type xxxx
29-Nov-1998
29-Nov-1998                      (Note new data added)

> output
Enter output filename: xxxx
Output file already exists.
Keep existing file? (y/n) n      (Replace current file)
> type xxxx
29-Nov-1998
```

The program appears to be functioning correctly in all three cases.

Good Programming Practice

Do not overwrite an output file without confirming that the user would like to delete the preexisting information.

8.8.2 The ferror Function

The MATLAB I/O system has several internal variables, including a special error indicator that is associated with each open file. This error indicator is updated by every I/O operation. The **ferror** function gets the error indicator and translates it into an easy-to-understand character message. The forms of the **ferror** function are:

```
message = ferror(fid)
message = ferror(fid, 'clear')
[message,errnum] = ferror(fid)
```

This function returns the most recent error message (and optionally, error number) associated with the file attached to `fid`. It can be called at any time after any I/O operation to get a more detailed description of what went wrong. If this function is called after a successful operation, the message will be '...' and the error number will be 0.

The 'clear' argument clears the error indicator for a particular file id.

8.8.3 The `feof` Function

The `feof` function tests to see if the current file position is at the end of the file. The form of the `feof` function is

```
eofstat = feof(fid)
```

This function returns a 1 if the current file position is at the end of the file, and a 0 otherwise.

8.8.4 The `ftell` Function

The `ftell` function returns the current location of the file position indicator for the file specified by `fid`. The position is a non-negative integer specified in bytes from the beginning of the file. A returned value of -1 for position indicates that the query was unsuccessful. If this happens, use `ferror` to determine why the request failed. The form of the `ftell` function is

```
position = ftell(fid)
```

8.8.5 The `frewind` Function

The `frewind` function allows a programmer to reset a file's position indicator to the beginning of the file. The form of the `frewind` function is

```
frewind(fid)
```

This function does not return status information.

8.8.6 The `fseek` Function

The `fseek` function allows a programmer to set a file's position indicator to an arbitrary location within a file. The form of the `fseek` function is

```
status = fseek(fid, offset, origin)
```

This function repositions the file position indicator in the file with the given `fid` to the byte with the specified `offset` relative to `origin`. The `offset` is measured in bytes, with a positive number indicating motion towards the end of the

file and a negative number indicating motion towards the head of the file. The origin is a string that can have one of three possible values.

- 'bof' This is the beginning of the file.
- 'cof' This is the current position within the file.
- 'eof' This is the end of the file.

The returned status is zero if the operation is successful and -1 if the operation fails. If the returned status is -1 , use `ferror` to determine why the request failed.

As an example of using `fseek` and `ferror` together, consider the following statements.

```
[fid,msg] = fopen('x','r');
status = fseek(fid,-10,'bof');
if status ~= 0
    msg = ferror(fid);
    disp(msg);
end
```

These commands open a file and attempt to set the file pointer 10 bytes before the beginning of the file. Since this is impossible, `fseek` returns a status of -1 , and `ferror` gets an appropriate error message. When these statements are executed, the result is an informative error message.

Offset is bad - before beginning-of-file.



Example 8.5—Fitting a Line to a Set of Noisy Measurements

In Example 4.7, we learned how to perform a fit of a noisy set of measurements (x,y) to a line of the form

$$y = mx + b \quad (8.1)$$

The standard method for finding the regression coefficients m and b is the method of least-squares. This method is named “least-squares” because it produces the line $y = mx + b$ for which the sum of the squares of the differences between the observed y values and the predicted y values is as small as possible. The slope of the least-squares line is given by

$$m = \frac{(\sum xy) - (\sum x)\bar{y}}{(\sum x^2) - (\sum x)\bar{x}} \quad (8.2)$$

and the intercept of the least-squares line is given by

$$b = \bar{y} - m\bar{x} \quad (8.3)$$

where

Σx is the sum of the x values.

Σx^2 is the sum of the squares of the x values.

Σxy is the sum of the products of the corresponding x and y values.

\bar{x} is the mean (average) of the x values.

\bar{y} is the mean (average) of the y values.

Write a program that will calculate the least-squares slope m and y -axis intercept b for a given set of noisy measured data points (x,y) that are to be found in an input data file.

Solution

1. State the Problem

Calculate the slope m and intercept b of a least-squares line that best fits an input data set consisting of an arbitrary number of (x,y) pairs. The input (x,y) data resides in a user-specified input file.

2. Define the Inputs and Outputs

The inputs required by this program are pairs of points (x,y) , where x and y are real quantities. Each pair of points will be located on a separate line in the input disk file. The number of points in the disk file is not known in advance.

The outputs from this program are the slope and intercept of the least-squares fitted line, plus the number of points going into the fit.

3. Describe the Algorithm

This program can be broken down into four major steps.

```
Get the name of the input file and open it
Accumulate the input statistics
Calculate the slope and intercept
Write out the slope and intercept
```

The first major step of the program is to get the name of the input file and to open the file. To do this, we will have to prompt the user to enter the name of the input file. After the file is opened, we must check to see that the open was successful. Next, we must read the file and keep track of the number of values entered, plus the sums Σx , Σy , Σx^2 , and Σxy . The pseudocode for these steps is

```
Initialize n, sum_x, sum_x2, sum_y, and sum_xy to 0
Prompt user for input file name
Open file 'filename'
Check for error on open
if no error
    READ x, y from file 'filename'
    while not at end-of-file
        n ← n + 1
        sum_x ← sum_x + x
```

```

        sum_y ← sum_y + y
        sum_x2 ← sum_x2 + x^2
        sum_xy ← sum_xy + x*y
        READ x, y from file 'filename'
    end
    (further processing)
end

```

Next, we must calculate the slope and intercept of the least-squares line. The pseudocode for this step is just the MATLAB versions of Equations (8.2) and (8.3).

```

x_bar ← sum_x / n
y_bar ← sum_y / n
slope ← (sum_xy - sum_x*y_bar) / (sum_x2 - sum_x*x_bar)
y_int ← y_bar - slope * x_bar

```

Finally, we must write out the results.

```
Write out slope 'slope' and intercept 'y_int'.
```

4. Turn the Algorithm into MATLAB Statements

The final MATLAB program follows.

```

% Script file: lsqfit.m
%
% Purpose:
%   To perform a least-squares fit of an input data set
%   to a straight line, and print out the resulting slope
%   and intercept values. The input data for this fit
%   comes from a user-specified input data file.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   ----      ======      =====
%   12/20/98  S. J. Chapman  Original code
%
% Define variables:
%   count      -- number of values read
%   filename   -- Input file name
%   fid        -- File id
%   msg        -- Open error message
%   n          -- Number of input data pairs (x,y)
%   slope      -- Slope of the line
%   sum_x     -- Sum of all input X values
%   sum_x2    -- Sum of all input X values squared
%   sum_xy    -- Sum of all input X*Y values
%   sum_y     -- Sum of all input Y values
%   x          -- An input X value
%   x_bar     -- Average X value

```

```

% y           -- An input Y value
% y_bar       -- Average Y value
% y_int       -- Y-axis intercept of the line

% Initialize sums
n = 0; sum_x = 0; sum_y = 0; sum_x2 = 0; sum_xy = 0;

% Prompt user and get the name of the input file.
disp('This program performs a least-squares fit of an');
disp('input data set to a straight line. Enter the name');
disp('of the file containing the input (x,y) pairs: ');
filename = input(' ', 's');

% Open the input file
[fid,msg] = fopen(filename,'rt');

% Check to see if the open failed.
if fid < 0

    % There was an error--tell user.
    disp(msg);

else

    % File opened successfully. Read the (x,y) pairs from
    % the input file. Get first (x,y) pair before the
    % loop starts.
    [in,count] = fscanf(fid,'%g %g',2);

    while ~feof(fid)
        x = in(1);
        y = in(2);
        n      = n + 1;                      %
        sum_x = sum_x + x;                  % Calculate
        sum_y = sum_y + y;                  %   statistics
        sum_x2 = sum_x2 + x.^2;            %
        sum_xy = sum_xy + x * y;            %

        % Get next (x,y) pair
        [in,count] = fscanf(fid,'%f',[1 2]);

    end

    % Now calculate the slope and intercept.
    x_bar = sum_x / n;
    y_bar = sum_y / n;
    slope = (sum_xy - sum_x*y_bar) / (sum_x2 - sum_x*x_bar);
    y_int = y_bar - slope * x_bar;

    % Tell user.
    fprintf('Regression coefficients for the least-squares line:\n');
    fprintf(' Slope (m) = %12.3f\n',slope);

```

```

fprintf(' Intercept (b) = %12.3f\n',y_int);
fprintf(' No of points = %12d\n',n);
end

```

5. Test the Program

To test this program, we will try a simple data set. For example, if every point in the input data set actually falls along a line, then the resulting slope and intercept should be exactly the slope and intercept of that line. Thus the data set

```

1.1  1.1
2.2  2.2
3.3  3.3
4.4  4.4
5.5  5.5
6.6  6.6
7.7  7.7

```

should produce a slope of 1.0 and an intercept of 0.0. If we place these values in a file called `input1`, and run the program, the results are

» lsqfit

This program performs a least-squares fit of an input data set to a straight line. Enter the name of the file containing the input (x,y) pairs:

input1

Regression coefficients for the least-squares line:

```

Slope (m)      =      1.000
Intercept (b) =      0.000
No of points  =      7

```

Now we add some noise to the measurements. The data set becomes

```

1.1  1.01
2.2  2.30
3.3  3.05
4.4  4.28
5.5  5.75
6.6  6.48
7.7  7.84

```

If these values are placed in a file called `input2`, and the program is run on that file, the results are

» lsqfit

This program performs a least-squares fit of an input data set to a straight line. Enter the name of the file containing the input (x,y) pairs:

input2

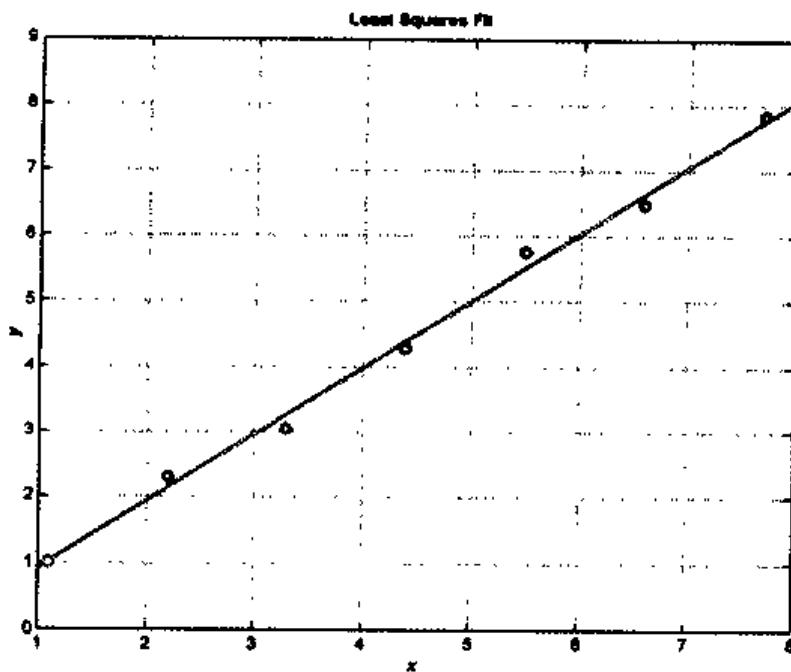


Figure 8.2 A noisy input data set and the resulting least-squares fitted line.

```
Regression coefficients for the least-squares line:
Slope (m) = 1.024
Intercept (b) = -0.120
No of points = 7
```

If we calculate the answer by hand, it is easy to show that the program gives the correct answers for our two test data sets. The noisy input data set and the resulting least-squares fitted line are shown in Figure 8.2.

8.9 Function uiimport

Function `uiimport` is a GUI-based way to import data from a file or from the clipboard. This command takes the forms

```
uiimport
structure = uiimport;
```

In the first case, the imported data is inserted directly into the current MATLAB workspace. In the second case, the data is converted into a structure and saved in variable `structure`.

When the command `uiimport` is typed, the Import Wizard is displayed in a window (see Figure 8.3 for the PC version of this window). The user can then select the file or clipboard that he or she would like to import from and the specific

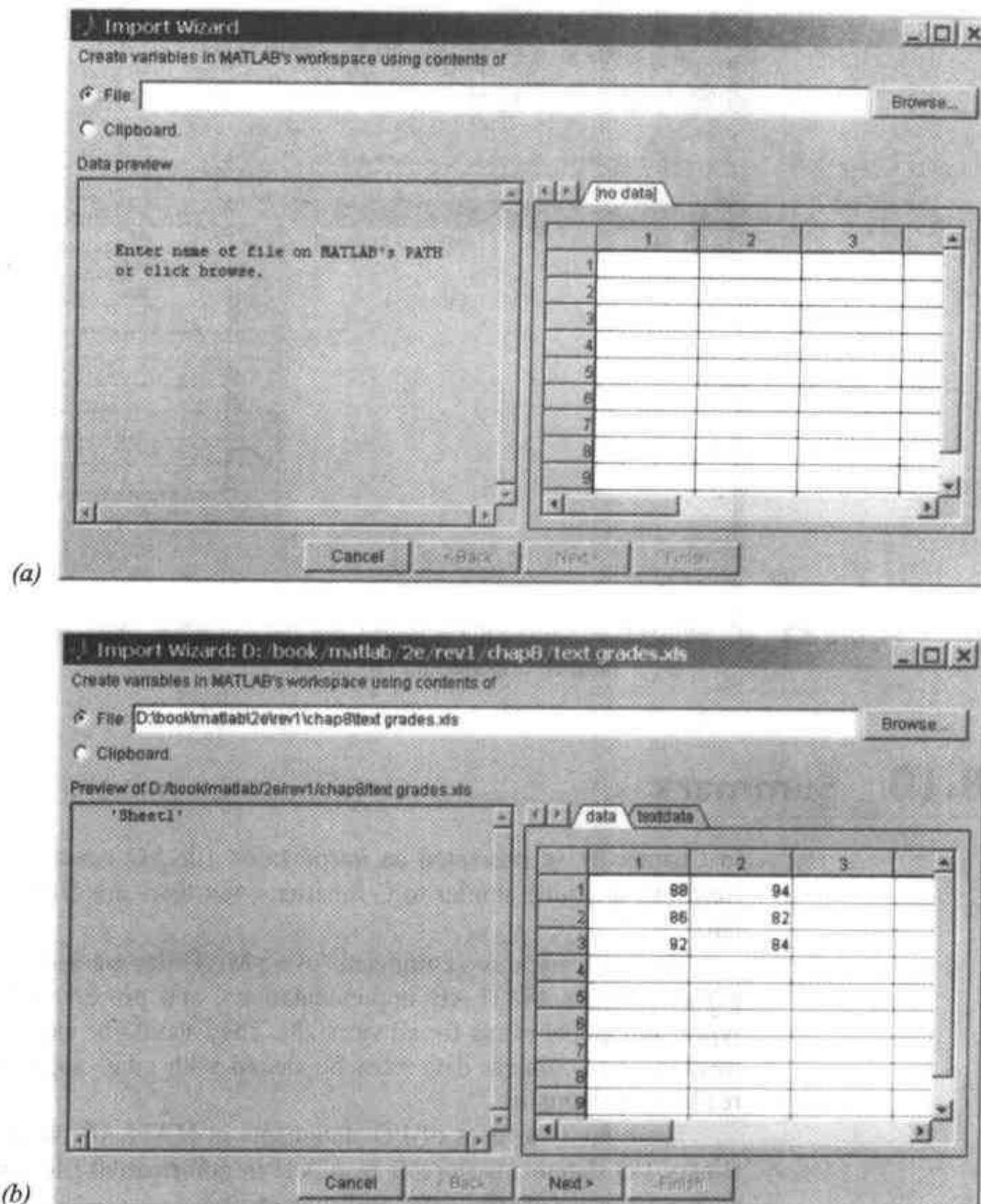


Figure 8.3 Using `uiimport`: (a) The Import Wizard after it is started. (b) After a data file has been selected, one or more data arrays are created, and their contents can be examined.

data within that file. Many different formats are supported, and data can be imported from almost *any* application by saving the data on the clipboard. This flexibility can be very useful when you are trying to get data into MATLAB for analysis.

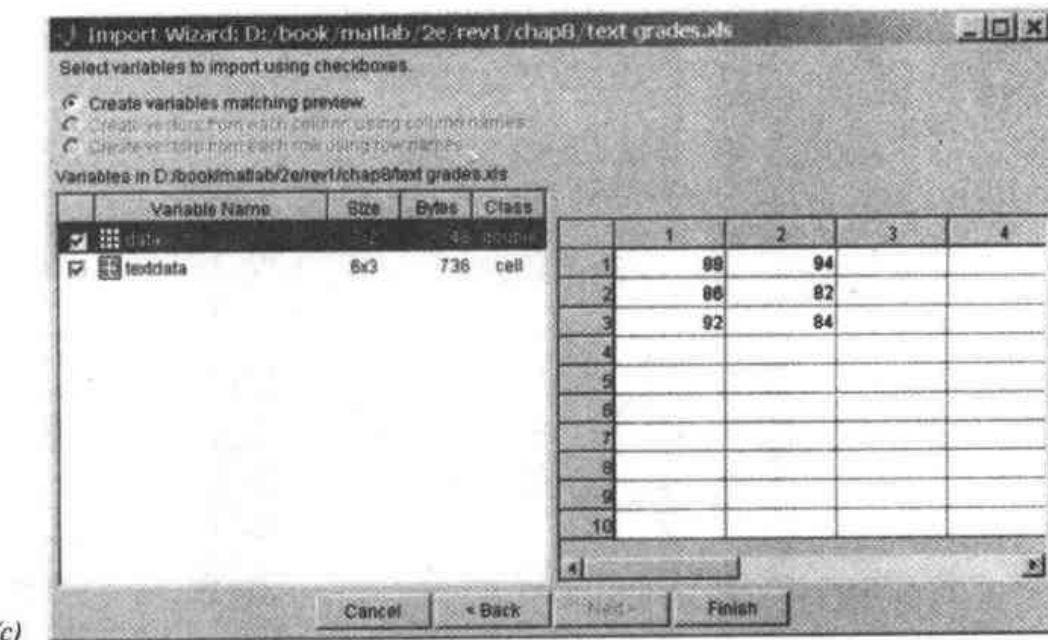


Figure 8.3 *Continued.* (c) Once a file is loaded, the user can select which of the data arrays will be imported into MATLAB.

8.10 Summary

In Chapter 8, we presented an introduction file I/O operations. MATLAB I/O functions are quite similar to C functions, but there are differences in some details.

The `load` and `save` commands using MAT-files are very efficient, are transportable across MATLAB implementations, and preserve full precision, data types, and global status for all variables. They should be used as the first-choice method of I/O, unless data must be shared with other applications or must be readable by humans.

There are two types of I/O statements in MATLAB: binary and formatted. Binary I/O statements store or read data in unformatted files, and formatted I/O statements store or read data in formatted files.

MATLAB files are opened with the `fopen` function and closed with the `fclose` function. Binary reads and writes are performed with the `fread` and `fwrite` functions, whereas formatted reads and writes are performed with the `fscanf` and `fprintf` functions. Functions `fgets` and `fgetl` simply transfer a line of text from a formatted file into a character string.

The `exist` function can be used to determine if a file exists before it is opened. This is useful to ensure that existing data is not accidentally overwritten.

It is possible to move around within a disk file using the `frewind` and `fseek` functions. The `frewind` function moves the current file position to the beginning of the file, whereas the `fseek` function moves the current file position to a point

a specified number of bytes ahead or behind a reference point. The reference point can be the current file position, the beginning of the file, or the end of the file.

8.10.1 Summary of Good Programming Practice

The following guidelines should be adhered to when working with MATLAB I/O functions.

1. Use function `textread` to import ASCII data in column format from programs written in other languages or exported from applications such as spreadsheets.
2. Unless you must exchange data with non-MATLAB programs, always use the `load` and `save` commands to save data sets in MAT-file format. This format is efficient and transportable across MATLAB implementations, and it preserves all details of all MATLAB data types.
3. Always be careful to specify the proper permissions in `fopen` statements, depending on whether you are reading from or writing to a file. This practice will help prevent errors such as accidentally overwriting data files that you want to keep.
4. Always check the status after a file open operation to make sure that it is successful. If the file open fails, tell the user and provide a way to recover from the problem.
5. Use formatted files to create data that must be readable by humans, or that must be transferable between programs on computers of different types. Use unformatted files to efficiently store large quantities of data that do not have to be directly examined and that will remain on only one type of computer. Also, use unformatted files when I/O speed is critical.
6. Do not overwrite an output file without confirming that the user would like to delete the preexisting information.

8.10.2 MATLAB Summary

The following summary lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one.

Commands and Functions

<code>exist</code>	Check for the existence of a file.
<code>fclose</code>	Close file.
<code>feof</code>	Test for end-of-file.
<code>ferror</code>	Inquire file I/O error status.
<code>fgetl</code>	Read line from file, discard newline character.
<code>fgets</code>	Read line from file, keep newline character.
<code>fopen</code>	Open file.
<code>fprintf</code>	Write formatted data to file.
<code>fread</code>	Read binary data from file.

<code>frewind</code>	Rewind file.
<code>fscanf</code>	Read formatted data from file.
<code>fseek</code>	Set file position.
<code>ftell</code>	Check file position.
<code>fwrite</code>	Write binary data to file.
<code>textread</code>	Read data of various types organized in column format from an ASCII file.
<code>uiimport</code>	Import data from a file or the clipboard using the Import Wizard

8.11 Exercises

- 8.1** What is the difference between binary and formatted I/O? Which MATLAB functions perform each type of I/O?
- 8.2** **Table of Logarithms.** Write a MATLAB program to generate a table of the base-10 logarithms between 1 and 10 in steps of 0.1. The table should start on a new page, and it should include a title describing the table and row and column headings. This table should be organized as shown below.

	X.0	X.1	X.2	X.3	X.4	X.5	X.6	X.7	X.8	X.9
1.0	0.000	0.041	0.079	0.114	...					
2.0	0.301	0.322	0.342	0.362	...					
3.0	...									
4.0	...									
5.0	...									
6.0	...									
7.0	...									
8.0	...									
9.0	...									
10.0	...									

- 8.3** Write a MATLAB program that reads in a time in seconds since the start of the day (this value will be somewhere between 0.0 and 86400.0), and prints a character string containing time in the form HH:MM:SS using the 24-hour clock convention. Use the proper format converter to ensure that leading zeros are preserved in the MM and SS fields. Also, be sure to check the input number of seconds for validity, and write an appropriate error message if an invalid number is entered.
- 8.4** **Gravitational Acceleration.** The acceleration due to the Earth's gravity at any height h above the surface of the Earth is given by the equation

$$g = -G \frac{M}{(R + h)^2} \quad (8.4)$$

where G is the gravitational constant ($6.672 \times 10^{-11} \text{ N m}^2 / \text{kg}^2$), M is the mass of the Earth ($5.98 \times 10^{24} \text{ kg}$), R is the mean radius of the Earth (6371 km), and h is the height above the Earth's surface. If M is measured in kg and R and h in meters, then the resulting acceleration will be in units of meters per second squared. Write a program to calculate the acceleration due to the Earth's gravity in 500-km increments at heights from 0 km to 40,000 km above the surface of the Earth. Print out the results in a table of height versus acceleration with appropriate labels, including the units of the output values. Plot the data as well.

- 8.5** The program in Example 8.5 illustrated the use of formatted I/O commands to read (x,y) pairs of data from disk. This could also be done with the `load -ascii` function. Rewrite this program to use `load` instead of the formatted I/O functions. Test your rewritten program to confirm that it gives the same answers as Example 8.5.
- 8.6** Rewrite the program in Example 8.5 to use the `textread` function instead of the formatted I/O functions. How difficult was it to use `textread`, compared to using `load -ascii` or the formatted I/O functions?
- 8.7** Write a program that reads an arbitrary number of real values from a user-specified input data file, rounds the values to the nearest integer, and writes the integers out to a user-specified output file. Make sure that the input file exists, and if not, tell the user and ask for another input file. If the output file exists, ask the user whether or not to delete it. If not, prompt for a different output file name.
- 8.8** **Table of Sines and Cosines.** Write a program to generate a table containing the sine and cosine of θ for θ between 0° and 90° , in 1° increments. The program should properly label each of the columns in the table.
- 8.9** **Interest Calculations.** Suppose that you have a sum of money P in an interest-bearing account at a local bank (P stands for *present value*). If the bank pays you interest on the money at a rate of i percent per year and compounds the interest monthly, the amount of money that you will have in the bank after n months is given by the equation

$$F = P \left(1 + \frac{i}{1200}\right)^n \quad (8.5)$$

where F is the future value of the account and $\frac{i}{12}$ is the monthly percentage interest rate (the extra factor of 100 in the denominator converts the interest rate from percentages to fractional amounts). Write a MATLAB program that will read an initial amount of money P and an annual interest rate i , and will calculate and write out a table showing the future value of the account every month for the next 5 years. The table should be written to an output file called 'interest'. Be sure to properly label the columns of your table.

- 8.10** Write a program to read a set of integers from an input data file, and locate the largest and smallest values within the data file. Print out the largest and smallest values, together with the lines on which they were found. Assume that you do not know the number of values in the file before the file is read.
- 8.11 Means.** In Exercise 4.27, we wrote a MATLAB program that calculated the arithmetic mean (average), rms average, geometric mean, and harmonic mean for a set of numbers. Modify that program to read an arbitrary number of values from an input data file, and calculate the means of those numbers. To test the program, place the following values into an input data file and run the program on that file: 1.0, 2.0, 5.0, 4.0, 3.0, 2.1, 4.7, 3.0.
- 8.12 Converting Radians to Degrees/Minutes/Seconds.** Angles are often measured in degrees ($^{\circ}$), minutes ($'$), and seconds ($"$), with 360 degrees in a circle, 60 minutes in a degree, and 60 seconds in a minute. Write a program that reads angles in radians from an input disk file, and converts them into degrees, minutes, and seconds. Test your program by placing the following four angles expressed in radians into an input file, and reading that file into the program: 0.0, 1.0, 3.141593, 6.0.
- 8.13** Create a data set in some other program on your computer, such as Microsoft Word, Microsoft Excel, a text editor, etc. Copy the data set to the clipboard using the Windows or UNIX copy function, and then use function `uiimport` to load the data set into MATLAB.

Handle Graphics

Handle graphics is the name of a set of low-level graphics functions that control the characteristics of graphics objects generated by MATLAB. These functions are normally hidden inside M-files, but they are very important to the programmer, since they allow him or her to have fine control of the appearance of the plots and graphs that they generate. For example, it is possible to use handle graphics to turn on a grid on the x-axis only, or to choose a line color like orange, which is not supported by the standard `LineSpec` option of the `plot` command. In addition, handle graphics enable a programmer to create graphical user interfaces (GUIs) for programs, as we will see in the next chapter.

This chapter introduces the structure of the MATLAB graphics system, and explains how to control the properties of graphical objects to create a desired display.

9.1 The MATLAB Graphics System

The MATLAB graphics system is based on a hierarchical system of **graphics objects**, each of which is known by a unique name called a **handle**. Each graphics object has special data known as **properties** associated with it, and modifying those properties will modify the behavior of the object. For example, a `line` is one type of graphics object. The properties associated with a line object include: `x`-data, `y`-data, color, line style, line width, marker type, and so forth. Modifying any of these properties will change the way that the line is displayed in a Figure Window.

Every component of a MATLAB graph is a graphical object. For example, each line, axes, and text string is a separate object with its own unique name (handle) and characteristics. All graphical objects are arranged in a hierarchy with **parent objects** and **child objects**, as shown in Figure 9.1. When a child object is created, it inherits many of its properties from its parent.

The highest-level graphics object in MATLAB is the **root**, which can be thought of as the entire computer screen. The root object is created automatically when MATLAB starts up and is always present until the program is shut down. The properties associated with the root object are the defaults that apply to all MATLAB windows.

Under the root object, there can be one or more **Figure Windows**, or just **figures**. Each **figure** is a separate window on the computer screen that can display graphical data, and each figure has its own properties. The properties associated with a **figure** include color, color map, paper size, paper orientation, pointer type, and so forth.

Each **figure** can contain four types of objects: **uimenus**, **uicontextmenus**, **uicontrols**, and **axes**. **Uimenus**, **uicontextmenus**, and **uicontrols** are special graphics objects used to create graphical user interfaces—they will be described in the next chapter. **Axes** are regions within a figure where data is actually plotted. There can be more than one set of axes in a single figure.

Each set of axes can contain as many lines, text strings, patches, and other graphical objects as necessary to create the plot of interest.

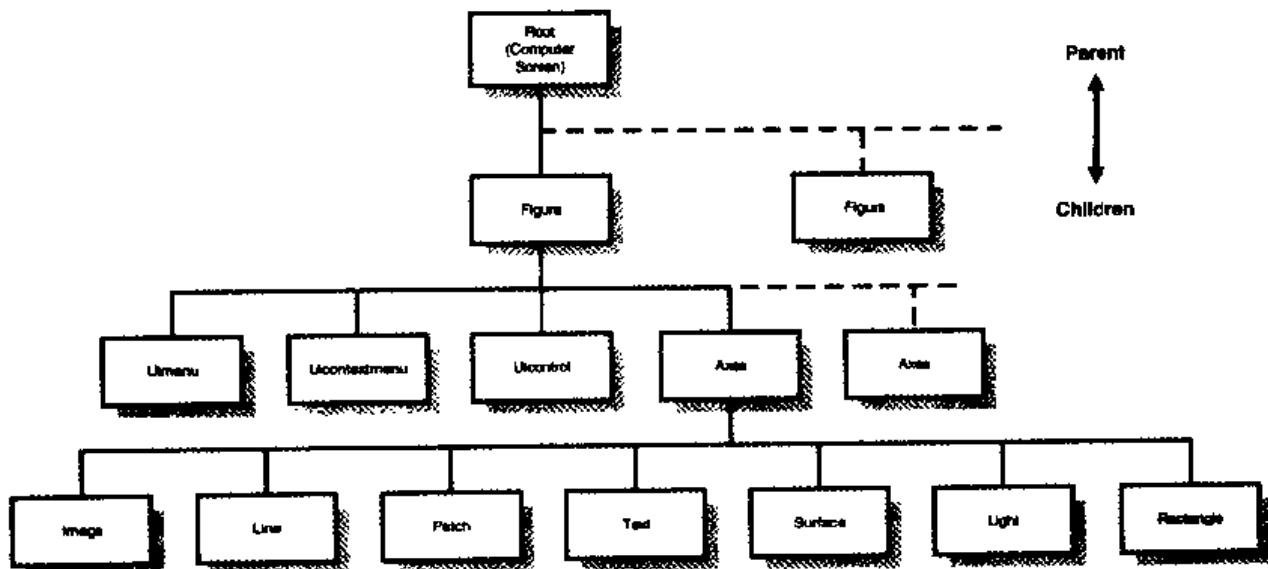


Figure 9.1 The hierarchy of handle graphics objects.

9.2 Object Handles

Each graphics object has a unique name called a **handle**. The handle is a unique integer or real number that is used by MATLAB to identify the object. A handle is automatically returned by any command that creates a graphics object. For example, the command

```
>> Hndl = figure;
```

creates a new figure and returns the handle of that figure in variable `Hndl`. The handle of the root object is always 0, and the handle of each figure is normally a small positive integer, such as 1, 2, 3, The handles of all other graphics objects are arbitrary floating-point numbers.

There are MATLAB functions available to get the handles of figures, axes, and other objects. For example, the function `gcf` returns the handle of the currently selected figure, and `gca` returns the handle of the current axes within the currently selected figure, and `gco` returns the handle of the currently selected object. These functions will be discussed in more detail later.

By convention, handles are usually stored in variables that begin with a capital `H` followed by lowercase letters. This distinguishes them from both ordinary variables, which are all lowercase, and global variables, which are all uppercase.

9.3 Examining and Changing Object Properties

Object properties are special values associated with an object that control some aspect of how that object behaves. Each property has a **property name** and an associated value. The property names are strings that are typically displayed in mixed case with the first letter of each word capitalized, but MATLAB recognizes a property name regardless of the case in which it is written.

9.3.1 Changing Object Properties at Creation Time

When an object is created, all of its properties are automatically initialized to default values. These default values can be overridden at creation time by including '`'PropertyName'`, `value` pairs in the object creation function¹. For example, we saw in Chapter 2 that the width of a line could be modified in the `plot` command as follows.

```
plot(x,y,'LineWidth',2);
```

¹ Examples of object creation functions include `figure`, which creates a new figure; `axes`, which creates a new set of axes with a figure; and `line`, which creates a line within a set of axes. High-level functions such as `plot` are also object creation functions.

This function overrides the default `LineWidth` property with the value 2 at the time that the line object is created.

9.3.2 Changing Object Properties after Creation Time

The properties of any object can be examined at any time using the `get` function, and modified using the `set` function. The most common forms of `get` function are

```
value = get(handle, 'PropertyName');
value = get(handle);
```

where `value` is the value contained in the specified property of the object whose handle is supplied. If only the handle is included in the function call, then the function returns a structure in which the field names are all of the properties of the object, and the field values are the property values.

The most common form of the `set` function is

```
set(handle, 'PropertyName1', value1, ...);
```

where there can be any number of '`PropertyName`', `value` pairs in a single function.

For example, suppose that we plot the function $y(x) = x^2$ from 0 to 2 with the following statements

```
x = 0:0.1:2;
y = x.^2;
Hndl = plot(x,y);
```

The resulting plot is shown in Figure 9.2a. The handle of the plotted line is stored in `Hndl`, and we can use it to examine or modify the properties of the line. The function `get(Hndl)` will return all of the properties of this line in a structure, with each property name being an element of the structure.

```
> result=get(Hndl)
result =
    BusyAction: 'queue'
    ButtonDownFcn: ''
    Children: []
    Clipping: 'on'
    Color: [0 0 1]
    CreateFcn: ''
    DeleteFcn: ''
    EraseMode: 'normal'
    HandleVisibility: 'on'
    Interruptible: 'on'
    LineStyle: '-'
    LineWidth: 0.5
    Marker: 'none'
    MarkerEdgeColor: 'auto'
```

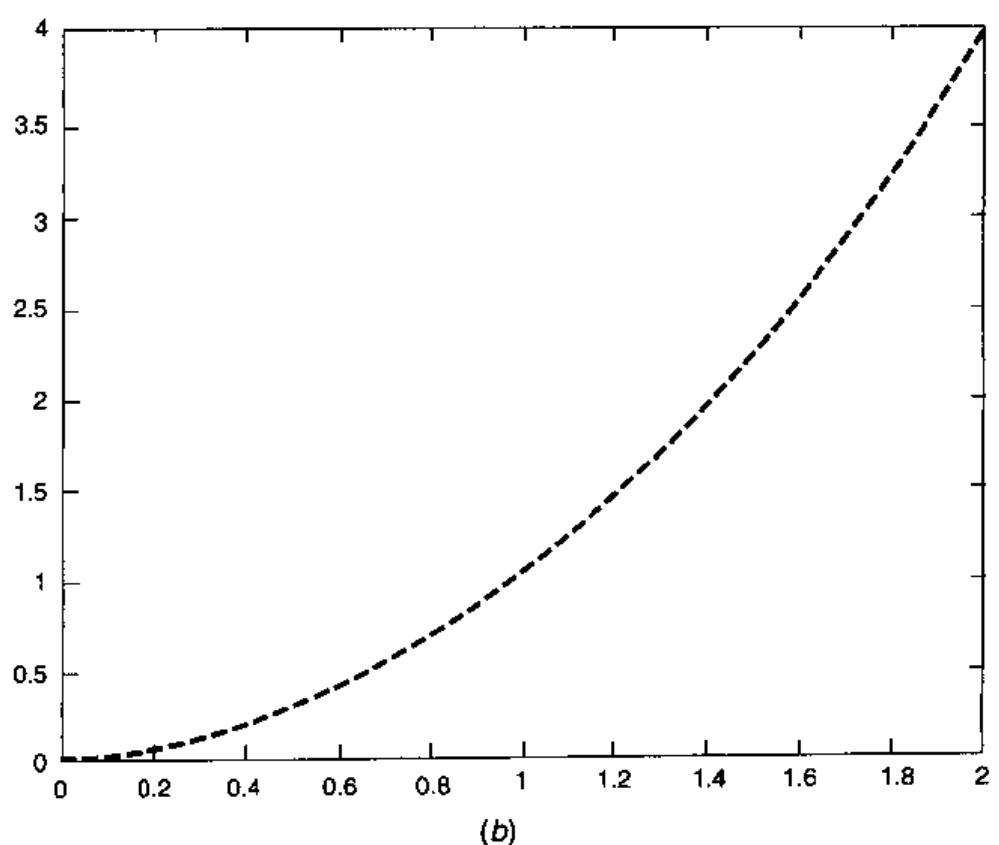
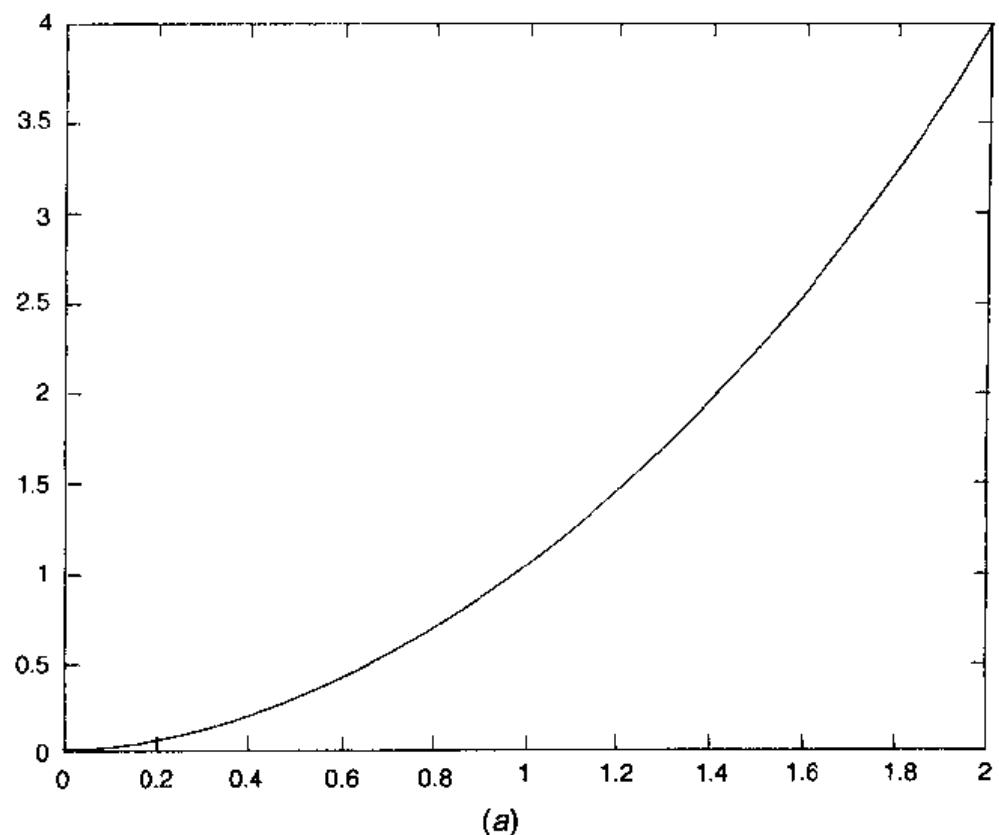


Figure 9.2 (a) Plot of the function $y = x^2$ using the default linewidth. (b) Plot of the function after modifying the `LineWidth` and `LineStyle` properties.

```

    MarkerFaceColor: 'none'
    MarkerSize: 6
    Parent: 2.0001
    Selected: 'off'
    SelectionHighlight: 'on'
    Tag: ''
    Type: 'line'
    UserData: []
    Visible: 'on'
    XData: [1x21 double]
    YData: [1x21 double]
    ZData: []

```

Note that the current line width is 0.5 pixels and the current line style is a solid line. We can change the line width and the line style with the commands

```
» set(Hndl, 'LineWidth', 4, 'LineStyle', '--')
```

The plot after this command is issued is shown in Figure 9.2b.

The `get` and `set` functions are especially useful for programmers because they can be directly inserted into MATLAB programs to modify a figure based on a user's input. As we shall see in the next chapter, these functions are used extensively in GUI programming.

For the end user, however, it is often easier to change the properties of a MATLAB object interactively. The Property Editor is a GUI-based tool designed for this purpose. The property editor is started by the command

```
propedit(HandleList);
propedit;
```

The first version of this function edits the properties of the listed handles, and the second version of this function edits the properties of the currently selected figure. For example, the following statements will create a plot containing the line $y = x^2$ over the range 0 to 2 and open the Property Editor to allow the user to interactively change the properties of the line.

```

figure(2);
x = 0:0.1:2;
y = x.^2;
Hndl = plot(x,y);
propedit(Hndl);

```

The Property Editor invoked by these statements is shown in Figure 9.3. The Property Editor contains a series of tabbed panes that vary depending on the type of object being modified. In the line object example shown here, the tabs are "Data," "Style," and "Info." The "Data" pane allows a user to select or modify the data being displayed—it modifies the `XData`, `YData`, and `ZData` properties of the line. The "Style" pane controls line style and marker properties, and the "Info" sets other miscellaneous information about the line object.

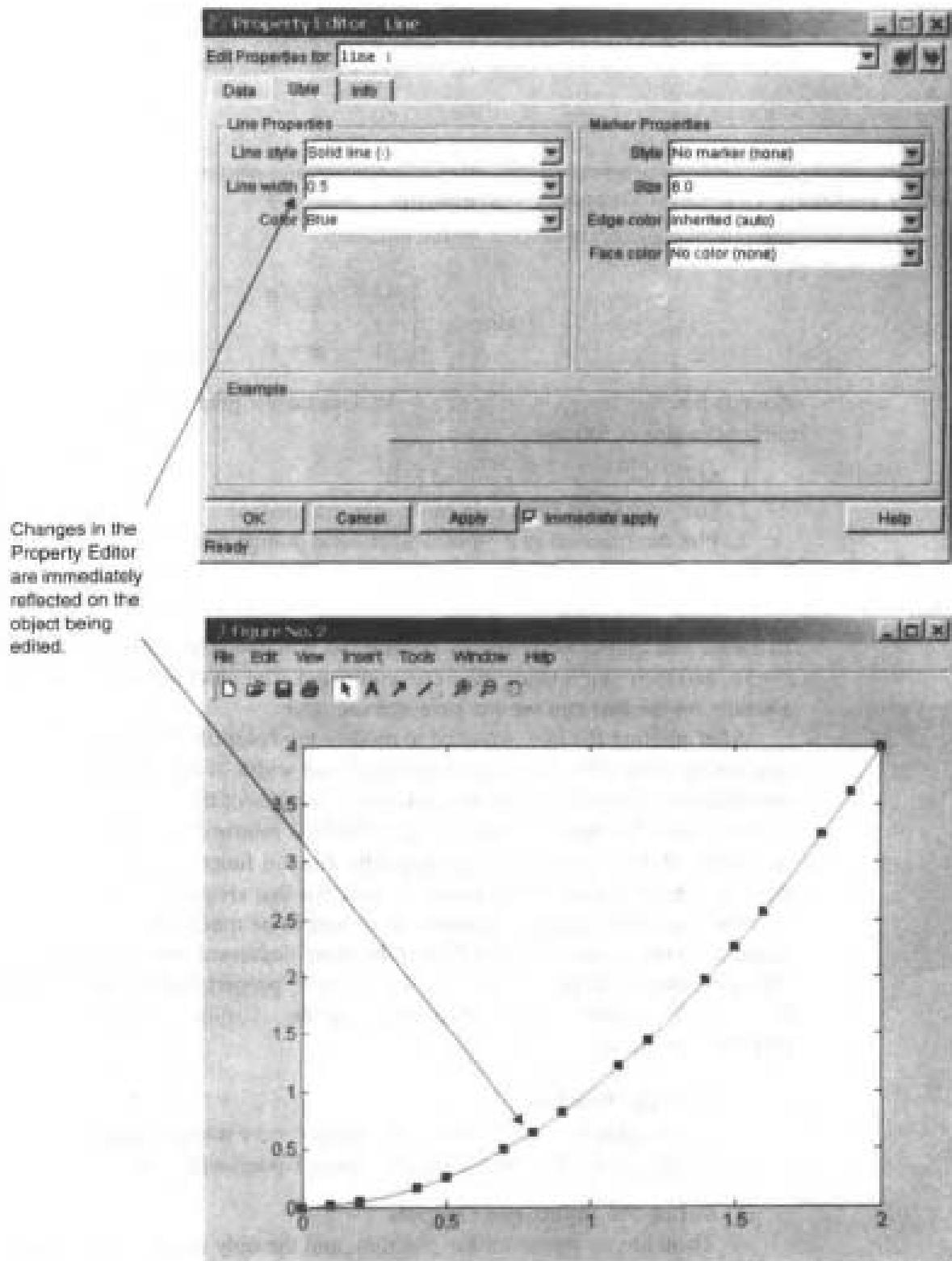


Figure 9.3 The Property Editor when editing a line object. Changes in style are immediately displayed on the figure as the object is edited.

The Property Editor can also be invoked by selecting the Edit button () on a figure toolbar, and then double-clicking the object that you want to edit.

Example 9.1—Using Low-Level Graphics Commands

The function $\text{sinc}(x)$ is defined by the equation

$$\text{sinc } x = \begin{cases} \frac{\sin x}{x} & x \neq 0 \\ 1 & x = 0 \end{cases} \quad (9.1)$$

Plot this function from $x = -3\pi$ to $x = 3\pi$. Use handle graphics functions to customize the plot as follows.

1. Make the figure background pink.
2. Use y -axis grid lines only (no x -axis grid marks).
3. Plot the function as a 3-point-wide solid orange line.

Solution

To create this graph, we need to calculate the function $\text{sinc } x$ from $x = -3\pi$ to $x = 3\pi$, and then plot it using the `plot` command. The `plot` command will return a handle for the line that we can save and use later.

After plotting the line, we need to modify the color of the `figure` object, the grid status of the `axes` object, and the color and width of the `line` object. These modifications require us to have access to the handles of the `figure`, `axes`, and `line` objects. The handle of the `figure` object is returned by the `gcf` function, the handle of the `axes` object is returned by the `gca` function, and the handle of the `line` object is returned by the `plot` function that created it.

The low-level graphics properties that need to be modified can be found by referring to the on-line MATLAB Help Browser documentation, under the topic "Handle Graphics Objects." They are the '`Color`' property of the current figure, the '`YGrid`' property of the current axes, and the '`LineWidth`' and '`Color`' properties of the line.

1. State the Problem

Plot the function $\text{sinc } x$ from $x = -3\pi$ to $x = 3\pi$ using a figure with a pink background, y -axis grid lines only, and a 3-point-wide solid orange line.

2. Define the Inputs and Outputs

There are no inputs to this program, and the only output is the specified figure.

3. Describe the Algorithm

This program can be broken down into three major steps.

```

Calculate sinc(x)
Plot sinc(x)
Modify the required graphics object properties

```

The first major step is to calculate $\text{sinc } x$ from $x = -3\pi$ to $x = 3\pi$. This can be done with vectorized statements, but the vectorized statements will produce a NaN at $x = 0$, since the division of 0/0 is undefined. We must replace the NaN with a 1.0 before plotting the function. The detailed pseudocode for this step is

```

% Calculate sinc(x)
x = -3*pi:pi/10:3*pi
y = sin(x) ./ x

% Find the zero value and fix it up. The zero is
% located in the middle of the x array.
index = fix(length(y)/2) + 1
y(index) = 1

```

Next, we must plot the function, saving the handle of the resulting line for further modifications. The detailed pseudocode for this step is

```
Hndl = plot(x,y);
```

Now we must use handle graphics commands to modify the figure background, y-axis grid, and line width and color. Note that the figure handle can be recovered with the function `gcf` and the axis handle can be recovered with the function `gca`. The color pink can be created with the RGB vector [1 0.8 0.8], and the color orange can be created with the RGB vector [1 0.5 0]. The detailed pseudocode for this step is

```

set(gcf,'Color',[1 0.8 0.8])
set(gca,'YGrid','on')
set(Hndl,'Color',[1 0.5 0],'LineWidth',3)

```

4. Turn the Algorithm into MATLAB Statements

The final MATLAB program follows.

```

% Script file: plotsinc.m
%
% Purpose:
%   This program illustrates the use of handle graphics
%   commands by creating a plot of sinc(x) from -3*pi to
%   3*pi, and modifying the characteristics of the figure,
%   axes, and line using the "set" function.
%
% Record of revisions:
%   Date      Programmer          Description of change
%   =====    ======          =====
%   11/23/98  S. J. Chapman    Original code

```

```

%
% Define variables:
%   Hndl          -- Handle of line
%   x              -- Independent variable
%   y              -- sinc(x)

% Calculate sinc(x)
x = -3*pi:pi/10:3*pi;
y = sin(x) ./ x;

% Find the zero value and fix it up. The zero is
% located in the middle of the x array.
index = fix(length(y)/2) + 1;
y(index) = 1;

% Plot the function.
Hndl = plot(x,y);

% Now modify the figure to create a pink background,
% modify the axis to turn on y-axis grid lines, and
% modify the line to be a 3-point wide orange line.
set(gcf,'Color',[1 0.8 0.8]);
set(gca,'YGrid','on');
set(Hndl,'Color',[1 0.5 0],'LineWidth',3);

```

5. Test the Program

Testing this program is very simple—we just execute it and examine the resulting plot. The plot created is shown in Figure 9.4, and it does have the characteristics that we wanted.

9.4 Using set to List Possible Property Values

The `set` function can be used to provide lists of possible property values. If a `set` function call contains a property name but not a corresponding value, `set` returns a list of all of the legal choices for that property. For example, the command `set(Hndl, 'LineStyle')` will return a list of all legal line styles with the default choice in brackets.

```
>> set(Hndl, 'LineStyle')
[ (--) | -- | : | -. | none ]
```

This function shows that the legal line styles are `'--'`, `'-'`, `:`, `'-.'`, and `'none'`, with the first choice as the default.

If the property does not have a fixed set of values, MATLAB says so.

```
>> set(Hndl, 'LineWidth')
A line's "LineWidth" property does not have a fixed set
of property values.
```

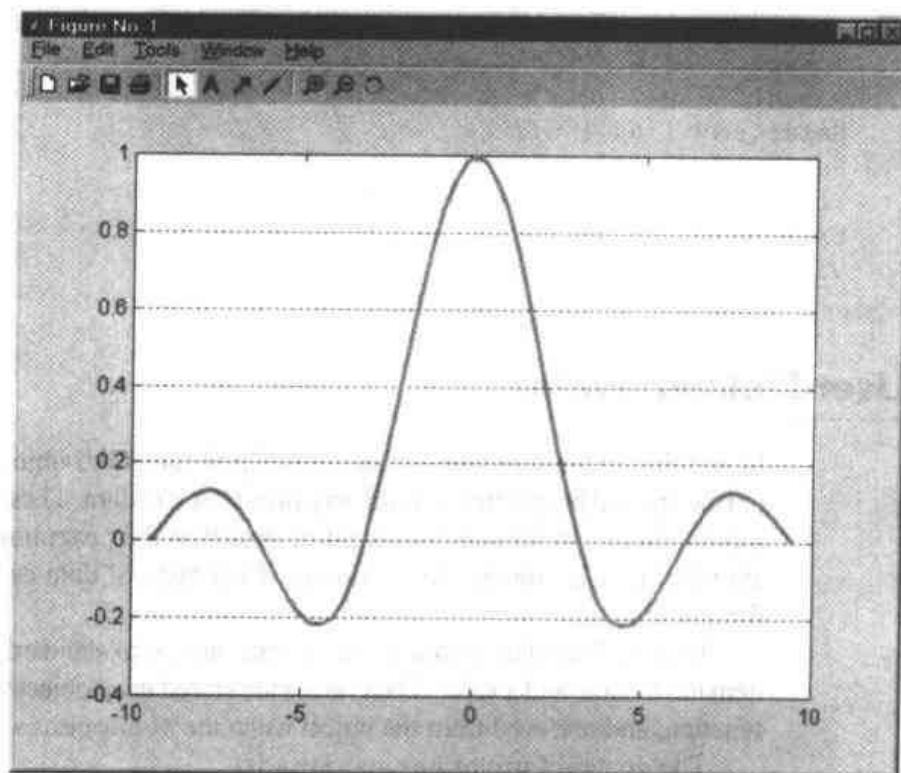


Figure 9.4 Plot of $\text{sinc } x$ versus x . A color version of this plot appears in the insert.

The function `set(Hndl)` will return all of the possible choices for all the properties of an object.

```
> set(Hndl)
Color
EraseMode: [ {normal} | background | xor | none ]
LineStyle: [ {-} | -- | : | -. | none ]
LineWidth
Marker: [ + | o | * | . | x | square | diamond | v | ^ | > | <
| pentagram | hexagram | {none} ]
MarkerSize
MarkerEdgeColor: [ none | {auto} ] -or- a ColorSpec.
MarkerFaceColor: [ {none} | auto ] -or- a ColorSpec.
XData
YData
ZData

ButtonDownFcn
Children
Clipping: [ {on} | off ]
CreateFcn
DeleteFcn
BusyAction: [ {queue} | cancel ]
```

```

HandleVisibility: [ {on} | callback | off ]
Interruptible: [ {on} | off ]
Parent
Selected: [ on | off ]
SelectionHighlight: [ {on} | off ]
Tag
UserData
Visible: [ {on} | off ]

```

9.5 User-Defined Data

In addition to the standard properties defined for a GUI object, a programmer can define special properties to hold program-specific data. These extra properties are a convenient way to store any kind of data that the programmer wants to associate with the GUI object. Any amount of any type of data can be stored and used for any purpose.

User-defined data is stored in a manner similar to standard properties. Each data item has a name and a value. Data values are stored in an object with the `setappdata` function, and retrieved from the object using the `getappdata` function.

The general form of `setappdata` is

```
setappdata(Hndl, 'DataName', DataValue);
```

where `Hndl` is the handle of the object to store the data into, '`DataName`' is the name given to the data, and `DataValue` is the value assigned to that name. Note that the data value can be either a numeric value or a character string.

For example, suppose that we wanted to define two special data values, one containing the number of errors that have occurred on a particular figure, and the other containing a string describing the last error detected. Such data values could be given the names '`ErrorCount`' and '`LastError`'. If we assume that `H1` is the handle of the figure, then command to create these data items and initialize them would be

```
setappdata(H1, 'ErrorCount', 0);
setappdata(H1, 'LastError', 'No error');
```

Application data can be retrieved at any time using the function `getappdata`. The two forms of `getappdata` are

```
value = getappdata(Hndl, 'DataName');
struct = getappdata(Hndl);
```

where `Hndl` is the handle of the object containing the data, and '`DataName`' is the name of the data to be retrieved. If a '`DataName`' is specified, then the value associated with that data name will be returned. If it is not specified, then *all* user-defined data associated with that object will be returned in a structure. The names of the data items will be structure element names in the returned structure.

Table 9.1 Functions for Manipulating User-Defined Data

Function	Description
<code>setappdata(Hndl, 'DataName', DataValue)</code>	Stores DataValue in an item named 'DataName' within the object specified by the handle Hndl.
<code>value = getappdata(Hndl, 'DataName')</code> <code>struct = getappdata(Hndl)</code>	Retrieves user-defined data from the object specified by the handle Hndl. The first form retrieves the value associated with 'DataName' only, and the second form retrieves all user-defined data.
<code>isappdata(Hndl, 'DataName')</code>	A logical function that returns a 1 if 'DataName' is defined within the object specified by the handle Hndl, and 0 otherwise.
<code>isappdata(Hndl, 'DataName')</code>	Removes the user-defined data item named 'DataName' from the object specified by the handle Hndl.

For the example given above, `getappdata` will produce the following results:

```
>> value = getappdata(H1, 'ErrorCount')
value =
    0
>> struct = getappdata(H1)
struct =
    ErrorCount: 0
    LastError: 'No error'
```

The functions associated with user-defined data are summarized in Table 9.1.

9.6 Finding Objects

Each new graphics object that is created has its own handle, and that handle is returned by the creating function.

Code Programming Practice

If you intend to modify the properties of an object that you create, save the handle of that object for later use with `get` and `set`.

However, sometimes we might not have access to the handle. Suppose that we lost a handle for some reason. How can we examine and modify the graphics objects? MATLAB provides four special functions to help find the handles of objects.

- `gcf` Returns the handle of the current *figure*.
- `gca` Returns the handle of the current *axes* in the current *figure*.
- `gco` Returns the handle of the current *object*.
- `findobj` Finds a graphics object with a specified property value.

The function `gcf` returns the handle of the current figure. If no figure exists, `gcf` will create one and return its handle. The function `gca` returns the handle of the current axes within the current figure. If no figure exists, or if the current figure exists but contains no axes, `gcf` will create a set of axes and return its handle. The function `gco` has the form

```
H_obj = gco;
H_obj = gco(H_fig);
```

where `H_obj` is the handle of the object and `H_fig` is the handle of a figure. The first form of this function returns the handle of the *current object in the current figure*. The second form of the function returns the handle of the *current object in a specified figure*.

The current object is defined as the last object clicked on with the mouse. This object can be any graphics object except the root. There will not be a current object within a figure until a mouse click has occurred within that figure. Before the first mouse click, function `gco` will return an empty array `[]`. Unlike `gcf` and `gca`, `gco` does not create objects if they do not exist.

Once the handle of an object is known, we can determine the type of the object by examining its 'Type' property. The 'Type' property will be a character string, such as '`figure`', '`line`', '`text`', and so forth.

```
H_obj = gco;
type = get(H_obj, 'Type')
```

The easiest way to find an arbitrary MATLAB object is with the `findobj` function. The basic form of this function is

```
Hndls = findobj('PropertyName1', value1, ...)
```

This command starts at the root object and searches the entire tree for all objects that have the specified values for the specified properties. Note that multiple property/value pairs can be specified, and `findobj` will only return the handles of objects that match all of them.

For example, suppose that we have created Figures 1 and 3. Then the function `findobj('Type', 'figure')` will return the results

```
>> H_fig = findobj('Type', 'figure')
H_fig =
    3
    1
```

This form of the `findobj` function is very useful, but it is slow, since it must search through the entire object tree to locate any matches. If you must use an object multiple times, only make one call to `findobj` and save the handle for re-use.

Restricting the number of objects that must be searched can increase the execution speed of this function. This can be done with the following form of the function.

```
Hndls = findobj(SrchHndls, 'PropertyName1', value1, ...)
```

Here, only the handles listed in array `SrchHndls` and their children will be searched to find the object. For example, suppose that you wanted to find all of the dashed lines in Figure 1. The command to do this would be:

```
Hndls = findobj(1, 'Type', 'line', 'LineStyle', '--');
```

Good Programming Practice

If possible, restrict the scope of your searches with `findobj` to make them faster.

9.7 Selecting Objects with the Mouse

Function `gco` returns the handle of the current object, which is the last object clicked on by the mouse. Each object has a **selection region** associated with it, and any mouse click within that selection region is assumed to be a click on that object. This is very important for thin objects like lines or points—the selection region allows the user to be slightly sloppy in mouse position and still select the line. The width of and shape of the selection region varies for different types of objects. For instance, the selection region for a line is 5 pixels on either side of the line, whereas the selection region for a surface, patch, or text object is the smallest rectangle that can contain the object.

The selection region for an axes object is the area of the axes plus the area of the titles and labels. However, lines or other objects inside the axes have a higher priority, so to select the axes, you must click on a point within the axes that is *not* near lines or text. Clicking on a figure outside of the axes region will select the figure itself.

What happens if a user clicks on a point that has two or more objects, such as the intersection of two lines? The answer depends on the **stacking order** of the objects. The stacking order is the order in which MATLAB selects objects. This order is specified by the order of the handles listed in the 'Children' property of a figure. If a click is in the selection region of two or more objects, the one with the highest position in the 'Children' list will be selected.

MATLAB includes a function called `waitforbuttonpress` that is sometimes used when selecting graphics objects. The form of this function is

```
k = waitforbuttonpress
```

When this function is executed, it halts the program until either a key is pressed or a mouse button is clicked. The function returns 0 if it detects a mouse button press or 1 if it detects a key press.

The function can be used to pause a program until a mouse click occurs. After the mouse click occurs, the program can recover the handle of the selected object using the `gco` function.

Example 9.2—Selecting Graphics Objects

The program shown in this example explores the properties of graphics objects, and incidentally shows how to select objects using `waitforbuttonpress` and `gco`. The program allows objects to be selected repeatedly until a key press occurs.

```
% Script file: select_object.m
%
% Purpose:
% This program illustrates the use of waitforbuttonpress
% and gco to select graphics objects. It creates a plot
% of sin(x) and cos(x), and then allows a user to select
% any object and examine its properties. The program
% terminates when a key press occurs.
%
% Record of revisions:
% Date      Programmer      Description of change
% =====      ======      =====
% 11/23/98    S. J. Chapman  Original code
%
% Define variables:
% details      -- Object details
% H1           -- Handle of sine line
% H2           -- Handle of cosine line
% Handle       -- Handle of current object
% k             -- Result of waitforbuttonpress
% type          -- Object type
% x             -- Independent variable
% y1            -- sin(x)
% y2            -- cos(x)
% yn            -- Yes/No

% Calculate sin(x) and cos(x)
x = -3*pi:pi/10:3*pi;
y1 = sin(x);
y2 = cos(x);

% Plot the functions.
H1 = plot(x,y1);
set(H1,'LineWidth',2);
hold on;
H2 = plot(x,y2);
set(H2,'LineWidth',2,'LineStyle',':', 'Color','r');
title('\bfPlot of sin \itx \rm\bf and cos \itx');
xlabel('\bf\itx');
ylabel('\bfsin \itx \rm\bf and cos \itx');
legend('sine','cosine');
hold off;
```

```

% Now set up a loop and wait for a mouse click.
k = waitforbuttonpress;

while k == 0

    % Get the handle of the object
    Handle = gco;

    % Get the type of this object.
    type = get(Handle,'Type');

    % Display object type
    disp(['Object type = ' type '.']);

    % Do we display the details?
    yn = input('Do you want to display details? (y/n) ','s');

    if yn == 'y'
        details = get(Handle);
        disp(details);
    end

    % Check for another mouse click
    k = waitforbuttonpress;
end

```

When this program is executed, it produces the plot shown in Figure 9.5.

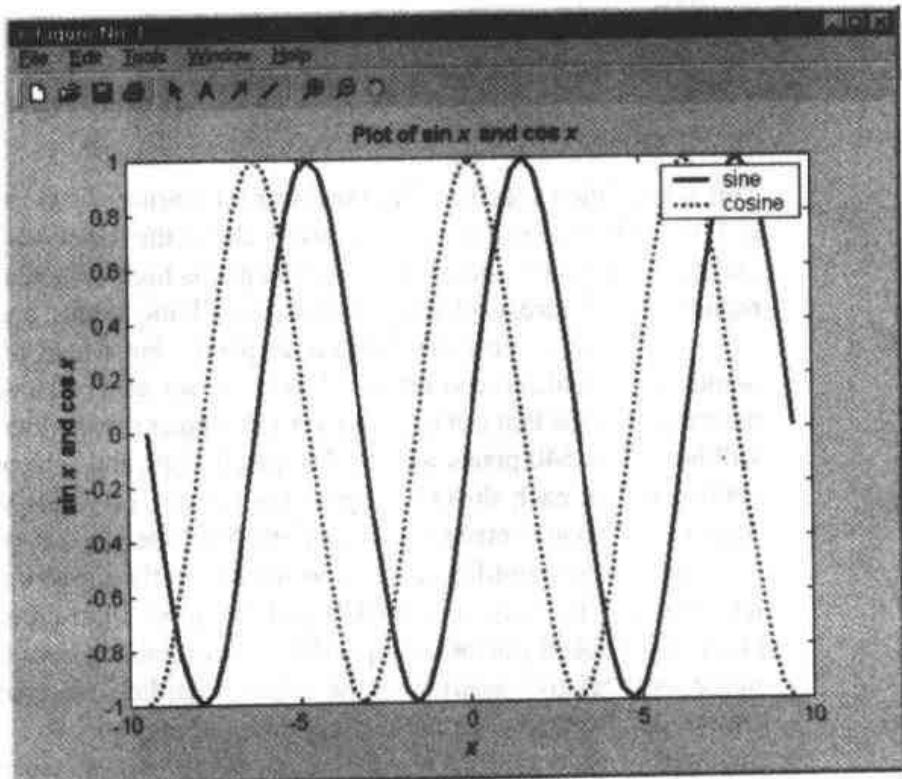


Figure 9.5 Plot of $\sin x$ and $\cos x$.

Experiment by clicking on various objects on the plot and viewing the resulting characteristics.

9.8 Position and Units

Many MATLAB objects have a 'position' property, which specifies the size and position of the object on the computer screen. This property differs slightly for different kinds of objects, as described in this section.

9.8.1 Positions of figure Objects

The 'position' property for a figure specifies the location of that figure on the computer screen using a 4-element row vector. The values in this vector are [left bottom width height], where *left* is the leftmost edge of the figure, *bottom* is the bottom edge of the figure, *width* is the width of the figure, and *height* is the height of the figure. These position values are in the units specified in the 'Units' property for the object. For example, the position and units associated with the current figure can be found as follows.

```
> get(gcf, 'Position')
ans =
    176    204    672    504
> get(gcf, 'Units')
ans =
pixels
```

This information specifies that the lower left corner of the current Figure Window is 176 pixels to the right and 204 pixels above the lower left corner of the screen, and the figure is 672 pixels wide by 504 pixels high. Note that this is the drawable region of the figure, excluding borders, scrollbars, menus, and the figure title area.

The 'units' property defaults to pixels, but it can be inches, centimeters, points, or normalized coordinates. Pixels are screen pixels, which are the smallest rectangular shape that can be drawn on a computer screen. Typical computer screens will be at least 640 pixels wide \times 480 pixels high, and screens can have more than 1000 pixels in each direction. Since the number of pixels varies from computer screen to computer screen, the size of an object specified in pixels will also vary.

Normalized coordinates are coordinates in the range 0 to 1, where the lower left corner of the screen is at (0,0) and the upper right corner of the screen is at (1,1). If an object position is specified in normalized coordinates, it will appear in the same relative position on the screen regardless of screen resolution. For example, the following statements create a figure and place it into the upper left quadrant of the screen on any computer, regardless of screen size².

² The normalized height of this figure is reduced to 0.45 to allow room for the Figure title and menu bar, both of which are above the drawing area.

```
H = figure(1)
set(H,'units','normalized','position',[0 .5 .5 .45])
```

Good Programming Practice

If you would like to place a window in a specific location, it is easier to place the window at the desired location using normalized coordinates, and the results will be the same regardless of the computer's screen resolution.

9.8.2 Positions of axes and uicontrol Objects

The position of axes and `uicontrol` objects is also specified by a 4-element vector, but the object position is specified relative to the lower left corner of the *figure* containing the objects, instead of the lower left hand corner of the screen. In general, the 'Position' property of any child object is relative to the position of its parent.

By default, the positions of axes objects are specified in *normalized* units within a figure, with (0,0) representing the lower left hand corner of the figure and (1,1) representing the upper right hand corner of the figure.

9.8.3 Positions of text Objects

Unlike other objects, `text` objects have a `Position` property containing only two or three elements. These elements correspond to the *x*, *y*, and *z* values of the `text` object *within* an axes object. Note that these values are in the units being displayed on the axes themselves.

The position of the `text` object with respect to the specified point is controlled by the object's `HorizontalAlignment` and `VerticalAlignment` properties. The `HorizontalAlignment` can be {Left}, Center, or Right, and the `VerticalAlignment` can be Top, Cap, {Middle}, Baseline, or Bottom.

The size of `text` objects is determined by the font size and the number of characters being displayed, so there are no height and width values associated with them.

Example 9.3—Positioning Objects within a Figure

As we mentioned earlier, axes positions are defined relative to the lower left corner of the frame that they are contained in, and `text` object positions are defined within axes in the data units being displayed on the axes.

To illustrate the positioning of graphics objects within a figure, we will write a program that creates two overlapping sets of axes within a single figure. The

first set of axes will display $\sin x$ versus x and have a text comment attached to the display line. The second set of axes will display $\cos x$ versus x and have a text comment in the lower left hand corner.

A program to create the figure is shown below. Note that we are using the `figure` function to create an empty figure and then two `axes` functions to create the two sets of axes within the figure. The position of the `axes` functions is specified in normalized units within the figure, so the first set of axes, which starts at (0.05,0.05), is in the lower left corner of the figure, and the second set of axes, which starts at (0.45,0.45), is in the upper right corner of the figure. Each set of axes has the appropriate function plotted on it.

The first `text` object is attached to the first set of axes at position $(-\pi, 0)$, which is a point on the curve. The '`HorizontalAlignment`', '`right`' property is selected, so the *attachment point* $(-\pi, 0)$ is on the *right side* of the text string. As a result, the text appears to the *left* of the attachment point in the final figure. (This can be confusing for new programmers!)

The second `text` object is attached to the second set of axes at position $(-7.5, -0.9)$, which is near the lower left corner of the axes. This string uses the default horizontal alignment, which is '`left`', so the attachment point $(-7.5, -0.9)$ is on the *left side* of the text string. As a result, the text appears to the right of the attachment point in the final figure.

```
% Script file: position_object.m
%
% Purpose:
%   This program illustrates the positioning of graphics
%   objects. It creates a figure, and then places
%   two overlapping sets of axes on the figure. The first
%   set of axes is placed in the lower left corner of
%   the figure, and contains a plot of  $\sin(x)$ . The second
%   set of axes is placed in the upper right corner of
%   the figure, and contains a plot of  $\cos(x)$ . Then two
%   text strings are added to the axes, illustrating the
%   positioning of text within axes.
%
% Record of revisions:
%   Date      Programmer      Description of change
%   =====    ======      =====
%   02/26/99  S. J. Chapman  Original code
%
% Define variables:
%   H1          -- Handle of sine line
%   H2          -- Handle of cosine line
%   Ha1         -- Handle of first axes
%   Ha2         -- Handle of second axes
%   x           -- Independent variable
```

```

%   y1           -- sin(x)
%   y2           -- cos(x)

% Calculate sin(x) and cos(x)
x = -2*pi:pi/10:2*pi;
y1 = sin(x);
y2 = cos(x);

% Create a new figure
figure;

% Create the first set of axes and plot sin(x).
% Note that the position of the axes is expressed
% in normalized units.
H1 = axes('Position',[.05 .05 .5 .5]);
H1 = plot(x,y1);
set(H1,'LineWidth',2);
title('\bfPlot of sin \itx');
xlabel('\bf\itx');
ylabel('\bfsin \itx');
axis([-8 8 -1 1]);

% Create the second set of axes and plot cos(x).
% Note that the position of the axes is expressed
% in normalized units.
H2 = axes('Position',[.45 .45 .5 .5]);
H2 = plot(x,y1);
set(H2,'LineWidth',2,'Color','r','LineStyle','--');
title('\bfPlot of cos \itx');
xlabel('\bf\itx');
ylabel('\bfcos \itx');
axis([-8 8 -1 1]);

% Create a text string attached to the line on the first
% set of axes.
axes(H1);
text(-pi,0.0,'sin(x)\rightarrow','HorizontalAlignment','right');

% Create a text string in the lower left corner
% of the second set of axes.
axes(H2);
text(-7.5,-0.9,'Text string 2');

```

When this program is executed, it produces the plot shown in Figure 9.6. You should execute this program again on your computer, changing the size or location of the objects being plotted, and observing the results.

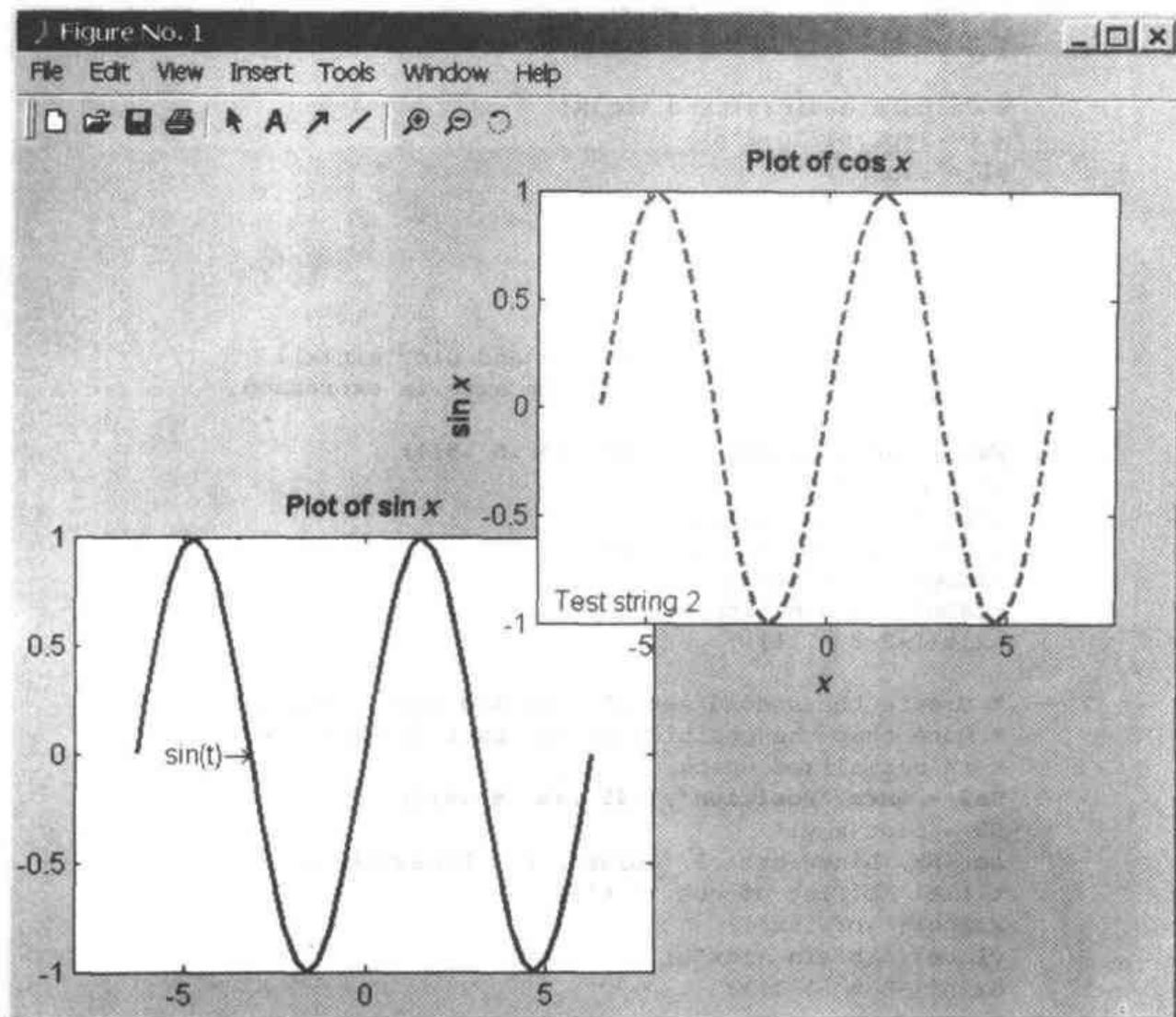


Figure 9.6 The output of program position_object.

9.9 Printer Positions

The 'Position' and 'Units' properties specify the location of a figure on the *computer screen*. There are also five other properties that specify the location of a figure on a sheet of paper *when it is printed*. These properties are summarized in Table 9.2.

Table 9.2 Printing-related Figure Properties

Option	Description
PaperUnits	Units for paper measurements: [{inches} centimeters normalized points]
PaperOrientation	[(portrait) landscape]
PaperPosition	A position vector of the form [left, bottom, width, height] where all units are as specified in PaperUnits.
PaperSize	A two-element vector containing the paper size, for example [8.5 11]
PaperType	Sets paper type. Note that setting this property automatically updates the PaperSize property. [{usletter} uslegal a3 a4letter a5 b4 tabloid]

For example, to set a plot to print out in landscape mode, on A4 paper, in normalized units, we could set the following properties.

```
set(Hndl, 'PaperType', 'a4letter')
set(Hndl, 'PaperOrientation', 'landscape')
set(Hndl, 'PaperUnits', 'normalized');
```

9.10 Default and Factory Properties

MATLAB assigns default properties to each object when it is created. If those properties are not what you want, then you must use `set` to select the desired values. If you wanted to change a property in every object that you create, this process could become very tedious. For those cases, MATLAB allows you to modify the default property itself, so that all objects will inherit the correct value of the property when they are created.

When a graphics object is created, MATLAB looks for a default value for each property by examining the object's parent. If the parent sets a default value, that value is used. If not, MATLAB examines the parent's parent to see if that object sets a default value, and so on back to the root object. MATLAB uses the first default value that it encounters when working back up the tree.

Default properties can be set at any point in the graphics object hierarchy that is *higher* than the level at which the object is created. For example, a default figure color would be set in the root object, and then all figures created after that time would have the new default color. On the other hand, a default axes color could be set in either the root object or the figure object. If the default

axes color is set in the root object, it will apply to all new axes in all figures. If the default axes color is set in the figure object, it will apply to all new axes in the current figure only.

Default values are set using a string consisting of 'Default' followed by the object type and the property name. Thus the default figure color would be set with the property 'DefaultFigureColor' and the default axes color would be set with the property 'DefaultAxesColor'. Some examples of setting default values follow:

<code>set(0, 'DefaultFigureColor', 'y')</code>	Yellow figure background—all new figures.
<code>set(0, 'DefaultAxesColor', 'r')</code>	Red axes background—all new axes in all figures.
<code>set(gcf, 'DefaultAxesColor', 'r')</code>	Red axes background—all axes in current figure only.
<code>set(gca, 'DefaultLineStyle', ':')</code>	Set default line style to dashed, in current axes only.

If you are working with the properties of existing objects, it is always a good idea to restore them to their original condition after they are used. *If you change the default properties of an object in a function, save the original values and restore them before exiting the function.* For example, suppose that we wish to create a series of figures in normalized units. We could save and restore the original units as follows.

```
saveunits = get(0, 'DefaultFigureUnits');
set(0, 'DefaultFigureUnits', 'normalized');
...
<MATLAB statements>
...
set(0, 'DefaultFigureUnits', saveunits);
```

If you want to customize MATLAB to use different default values at all times, then you should set the defaults in the root object every time that MATLAB starts up. The easiest way to do this is to place the default values into the `startup.m` file, which is automatically executed every time MATLAB starts. For example, suppose you always use A4 paper and you always want a grid displayed on your plots. Then you could set the following lines into `startup.m`.

```
set(0, 'DefaultFigurePaperType', 'a4letter');
set(0, 'DefaultAxesXGrid', 'on');
set(0, 'DefaultAxesYGrid', 'on');
set(0, 'DefaultAxesZGrid', 'on');
```

There are three special value strings that are used with handle graphics: 'remove', 'factory', and 'default'. If you have set a default value for a property, the 'remove' value will remove the default that you set. For example, suppose that you set the default figure color to yellow.

```
set(0,'DefaultFigureColor','y');
```

The following function call will cancel this default setting and restore the previous default setting.

```
set(0,'DefaultFigureColor','remove');
```

The string 'factory' allows a user to temporarily override a default value and use the original MATLAB default value instead. For example, the following figure is created with the factory default color despite a default color of yellow being previously defined.

```
set(0,'DefaultFigureColor','y');
figure('Color','factory')
```

The string 'default' forces MATLAB to search up the object hierarchy until it finds a default value for the desired property. It uses the first default value that it finds. If it fails to find a default value, then it uses the factory default value for that property. This use is illustrated below:

```
* Set default values
set(0,'DefaultLineColor','k'); % root default = black
set(gcf,'DefaultLineColor','g'); % figure default = green

% Create a line on the current axes. This line is green.
Hndl = plot(randn(1,10));
set(Hndl,'Color','default');
pause(2);

% Now clear the figures default and set the line color to the new
% default. The line is now black.
set(gcf,'DefaultLineColor','remove');
set(Hndl,'Color','default');
```

9.11 Graphics Object Properties

There are hundreds of different graphic object properties, far too many to discuss in detail here. The best place to find a complete list of graphics object properties is in the Help Browser distributed with MATLAB.

We will mention a few of the most important properties for each type of graphic object as we need them. A complete set of properties is given in the MATLAB Help Browser documentation under the descriptions of each type of object.

9.12 Summary

Every element of a MATLAB plot is a graphics object. Each object is identified by a unique handle, and each object has many properties associated with it, which affect the way the object is displayed.

MATLAB objects are arranged in a hierarchy with **parent objects** and **child objects**. When a child object is created, it inherits many of its properties from its parent.

The highest-level graphics object in MATLAB is the **root**, which can be thought of as the entire computer screen. Under the root there can be one or more **Figure Windows**. Each **figure** is a separate window on the computer screen that can display graphical data, and each figure has its own properties.

Each **figure** can contain four types of objects: **uimenus**, **uicontextmenus**, **uicontrols**, and **axes**. **Uimenus**, **uicontextmenus**, and **uicontrols** are special graphics objects used to create graphical user interfaces—they will be described in the next chapter. **Axes** are regions within a figure where data is actually plotted. There can be more than one set of axes in a single figure.

Each set of axes can contain as many lines, text strings, patches, and other graphical objects as necessary to create the plot of interest.

The handles of the current figure, current axes, and current object can be recovered with the **gcf**, **gca**, and **gco** functions respectively. The properties of any object can be examined and modified using the **get** and **set** functions.

There are literally hundreds of properties associated with MATLAB graphics functions, and the best place to find the details of these functions is the MATLAB on-line documentation.

9.12.1 Summary of Good Programming Practice

The following guidelines should be adhered to when working with MATLAB handle graphics.

1. If you intend to modify the properties of an object that you create, save the handle of that object for later use with **get** and **set**.
2. If possible, restrict the scope of your searches with **findobj** to make them faster.
3. If you would like to place a window in a specific location, it is easier to place the window at the desired location using normalized coordinates, and the results will be the same regardless of the computer's screen resolution.

9.12.2 MATLAB Summary

The following summary lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one.

axes	Creates new axes / makes axes current.
figure	Creates a new figure / makes figure current.
findobj	Finds an object based on one or more property values.

<code>gca</code>	Get handle of current axes.
<code>gcf</code>	Get handle of current figure.
<code>gco</code>	Get handle of current object.
<code>get</code>	Gets object properties.
<code>getappdata</code>	Gets user-defined data in an object.
<code>isappdata</code>	Tests to see if an object contains user-defined data with the specified name.
<code>rmpappdata</code>	Removes user-defined data from an object.
<code>set</code>	Sets object properties.
<code>setappdata</code>	Stores user-defined data in an object.
<code>waitforbuttonpress</code>	Pauses program, waiting for a mouse click or keyboard input.

9.13 Exercises

- 9.1** What is meant by the term *handle graphics*? Sketch the hierarchy of MATLAB graphics objects.
- 9.2** Use the MATLAB Help Browser to learn about the Name and Number Title properties of a figure object. Create a figure containing a plot of the function $y(x) = e^x$ for $-2 \leq x \leq 2$. Change the properties mentioned above to suppress the figure number and to add the title “Plot Window” to the figure.
- 9.3** Write a program that modifies the default figure color to orange and the default line width to 3.0 points. Then create a figure plotting the ellipse defined by the equations

$$\begin{aligned}x(t) &= 10 \cos t \\y(t) &= 6 \sin t\end{aligned}$$

from $t = 0$ to $t = 2\pi$. What color and width was the resulting line?

- 9.4** Use the MATLAB Help Browser to learn about the CurrentPoint property of an axes object. Use this property to create a program that creates an axes object and plots a line connecting the locations of successive mouse clicks within the axes. Use the function `waitforbuttonpress` to wait for mouse clicks, and update the plot after each click. Terminate the program when a keyboard press occurs.
- 9.5** Create a MATLAB program that plots the functions

$$x(t) = \cos \frac{t}{\pi}$$

$$x(t) = 2 \sin \frac{t}{2\pi}$$

for the range $-2 \leq t \leq 2$. The program should then wait for mouse clicks, and if the mouse has clicked on one of the two lines, the program should change the line’s color randomly from a choice of red, green, blue, yellow,

cyan, magenta, or black. Use the function `waitforbuttonpress` to wait for mouse clicks, and update the plot after each click. Use the function `gco` to determine the object clicked on, and use the `Type` property of the object to determine if the click was on a line.

- 9.6** Create a MATLAB figure and store some user-defined data values in the figure. Then, save the figure in a figure file (*.fig) using the File/Save As menu option on the Figure Window. Next close down and restart MATLAB. Reload the figure using the File/Open menu option on a Figure Window, and try to recover the user-defined data using the `getappdata` function. Was the information preserved?

Graphical User Interfaces

A graphical user interface (GUI) is a pictorial interface to a program. A good GUI can make programs easier to use by providing them with a consistent appearance and with intuitive controls like pushbuttons, list boxes, sliders, menus, and so forth. The GUI should behave in an understandable and predictable manner, so that a user knows what to expect when he or she performs an action. For example, when a mouse click occurs on a pushbutton, the GUI should initiate the action described on the label of the button.

This chapter introduces the basic elements of the MATLAB GUIs. The chapter does not contain a complete description of components or GUI features, but it does provide the basics required to create functional GUIs for your programs.

10.1 How a Graphical User Interface Works

A graphical user interface provides the user with a familiar environment in which to work. This environment contains pushbuttons, toggle buttons, lists, menus, text boxes, and so forth, all of which are already familiar to the user, so that he or she can concentrate on using the application rather than on the mechanics involved in doing things. However, GUIs are harder for the programmer because a GUI-based program must be prepared for mouse clicks (or possibly keyboard input) for any GUI element at any time. Such inputs are known as **events**, and a program that responds to events is said to be *event driven*.

The three principal elements required to create a MATLAB Graphical User Interface are

1. **Components.** Each item on a MATLAB GUI (pushbuttons, labels, edit boxes, etc.) is a graphical component. The types of components include graphical **controls** (pushbuttons, edit boxes, lists, sliders, etc.), **static elements** (frames and text strings), **menus**, and **axes**. Graphical controls and static elements are created by the function `uicontrol`, and menus are created by the functions `uimenu` and `uicontextmenu`. Axes, which are used to display graphical data, are created by the function `axes`.
2. **Figures.** The components of a GUI must be arranged within a **figure**, which is a window on the computer screen. In the past, figures have been created automatically whenever we have plotted data. However, empty figures can be created with the function `figure` and can be used to hold any combination of components.
3. **Callbacks.** Finally, there must be some way to perform an action if a user clicks a mouse on a button or types information on a keyboard. A mouse click or a key press is an event, and the MATLAB program must respond to each event if the program is to perform its function. For example, if a user clicks on a button, that event must cause the MATLAB code that implements the function of the button to be executed. The code executed in response to an event is known as a **callback**. There must be a callback to implement the function of each graphical component on the GUI.

The basic GUI elements are summarized in Table 10.1, and sample elements are shown in Figure 10.1. We will be studying examples of these elements and then build working GUIs from them.

10.2 Creating and Displaying a Graphical User Interface

MATLAB GUIs are created using a tool called `guide`, the GUI Development Environment. This tool allows a programmer to lay out the GUI, selecting and aligning the GUI components to be placed in it. Once the components are in place, the programmer can edit their properties: name, color, size, font, text to display, and so forth. When `guide` saves the GUI, it creates working program including skeleton functions that the programmer can modify to implement the behavior of the GUI.

When `guide` is executed, it creates the Layout Editor, shown in Figure 10.2. The large white area with grid lines is the *layout area*, where a programmer can lay out the GUI. The Layout Editor window has a palette of GUI components along the left side of the layout area. A user can create any number of GUI components by first clicking on the desired component, and then dragging its outline in the layout area. The top of the window has a toolbar with a series of useful tools that allow the user to distribute and align GUI components, modify the properties of GUI components, add menus to GUIs, and so on.

The basic steps required to create a MATLAB GUI are

1. Decide what elements are required for the GUI and what the function of each element will be. Make a rough layout of the components by hand on a piece of paper.

Table 10.1 Some Basic GUI Components

Element	Created By	Description
Graphical Controls		
Pushbutton	uicontrol	A graphical component that implements a pushbutton. It triggers a callback when clicked with a mouse.
Toggle button	uicontrol	A graphical component that implements a toggle button. A toggle button is either “on” or “off,” and it changes state each time that it is clicked. Each mouse button click also triggers a callback.
Radio button	uicontrol	A radio button is a type of toggle button that appears as a small circle with a dot in the middle when it is “on.” Groups of radio buttons are used to implement mutually exclusive choices. Each mouse click on a radio button triggers a callback.
Check box	uicontrol	A check box is a type of toggle button that appears as a small square with a check mark in it when it is “on.” Each mouse click on a check box triggers a callback.
Edit box	uicontrol	An edit box displays a text string and allows the user to modify the information displayed. A callback is triggered when the user presses the Enter key.
List box	uicontrol	A list box is a graphical control that displays a series of text strings. A user can select one of the text strings by single- or double-clicking on it. A callback is triggered when the user selects a string.
Popup menus	uicontrol	A popup menu is a graphical control that displays a series of text strings in response to a mouse click. When the popup menu is not clicked on, only the currently selected string is visible.
Slider	uicontrol	A slider is a graphical control to adjust a value in a smooth, continuous fashion by dragging the control with a mouse. Each slider change triggers a callback.
Static Elements		
Frame	uicontrol	Creates a frame, which is a rectangular box within a figure. Frames are used to group sets of controls together. Frames never trigger callbacks.
Text field	uicontrol	Creates a label, which is a text string located at a point on the figure. Text fields never trigger callbacks.
Menus and Axes		
Menu items	uimenu	Creates a menu item. Menu items trigger a callback when a mouse button is released over them.
Context menus	uicontextmenu	Creates a context menu, which is a menu that appears over a graphical object when a user right-clicks the mouse on that object.
Axes	axes	Creates a new set of axes to display data on. Axes never trigger callbacks.

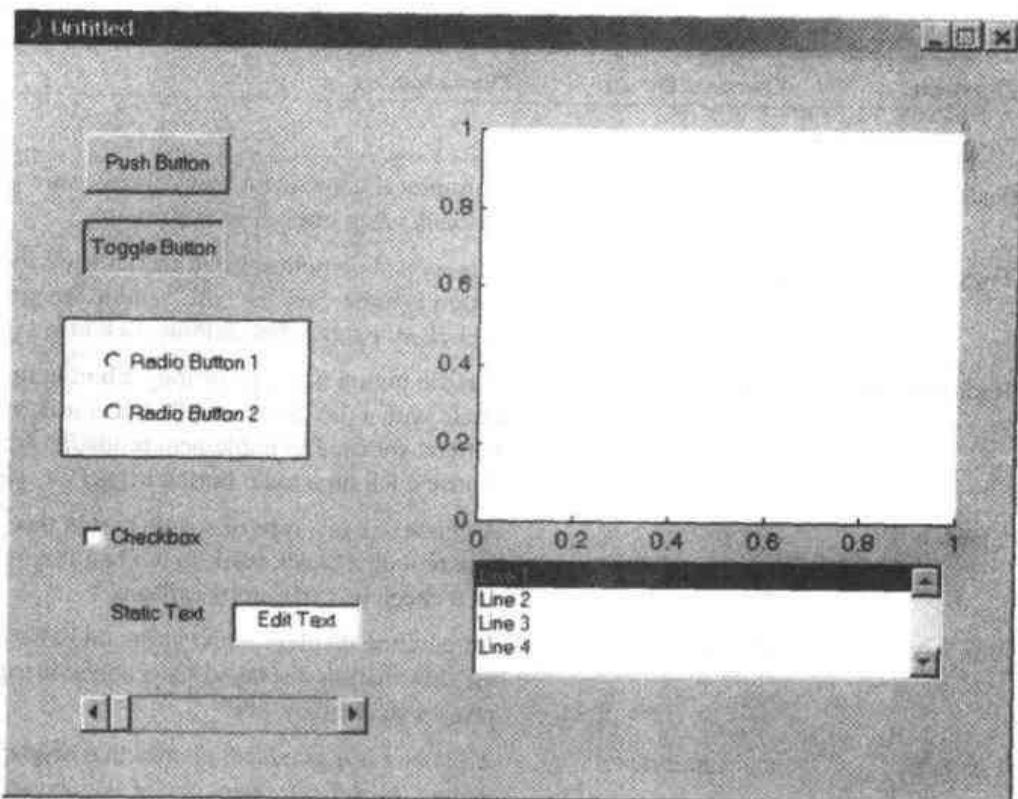


Figure 10.1 A Figure Window showing examples of MATLAB GUI elements. From top to bottom and left to right, the elements are: (1) a pushbutton; (2) a toggle button in the ‘on’ state; (3) two radio buttons surrounded by a frame; (4) a check box; (5) a text field and an edit box; (6) a slider; (7) a set of axes; and (8) a list box.

2. Use a MATLAB tool called `guide` (GUI Development Environment) to lay out the components on a figure. The size of the figure and the alignment and spacing of components on the figure can be adjusted using the tools built into `guide`.
3. Use a MATLAB tool called the Property Inspector (built into `guide`) to give each component a name (a “tag”) and to set the characteristics of each component, such as its color, the text it displays, and so on.
4. Save the figure to a file. When the figure is saved, two files will be created on disk with the same name but different extents. The `.fig` file contains the actual GUI that you have created, and the M-file contains the code to load the figure and skeleton callbacks for each GUI element.
5. Write code to implement the behavior associated with each callback function.

As an example of these steps, let’s consider a simple GUI that contains a single pushbutton and a single text string. Each time that the pushbutton is clicked, the text string will be updated to show the total number of clicks since the GUI started.

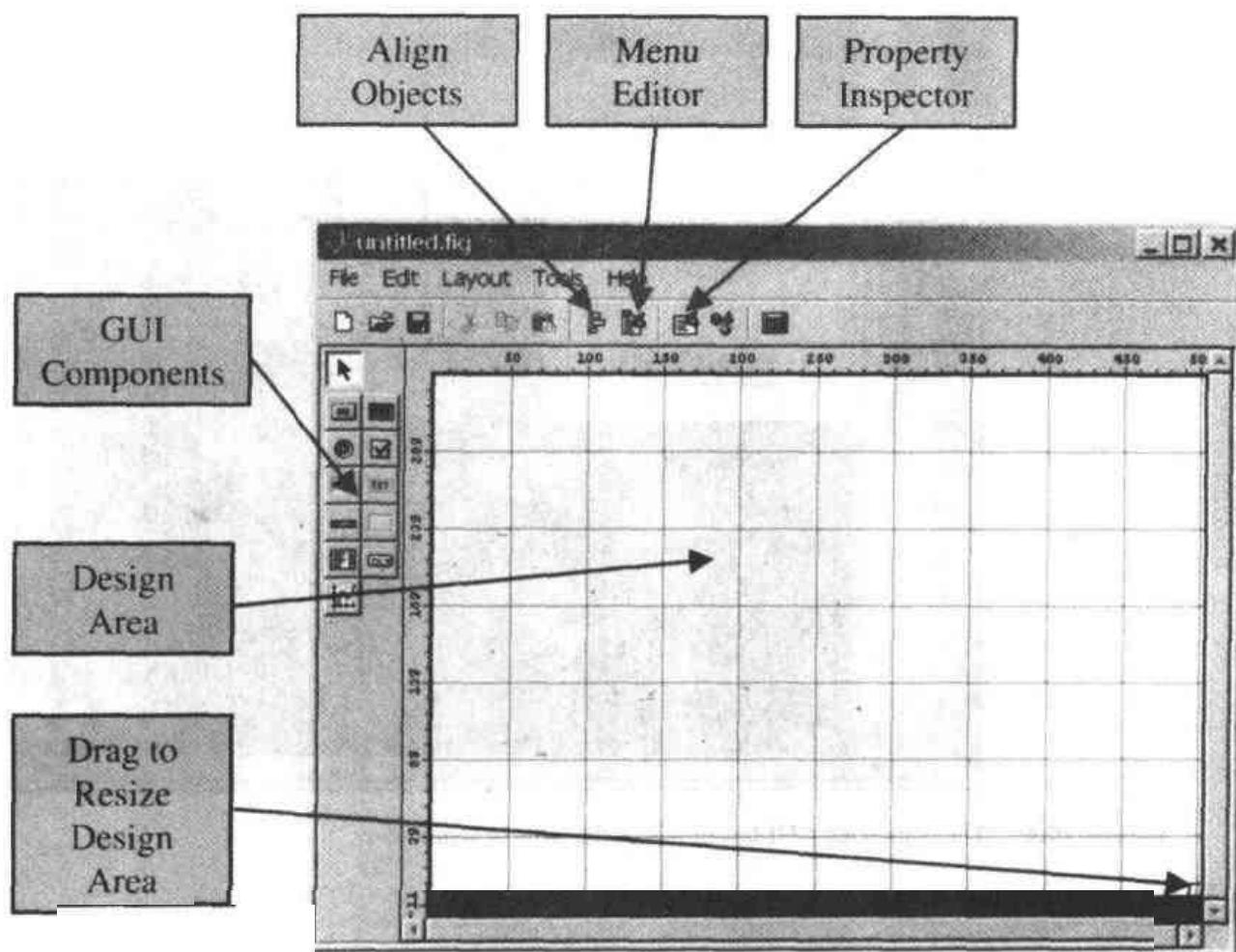


Figure 10.2 The guide tool window.

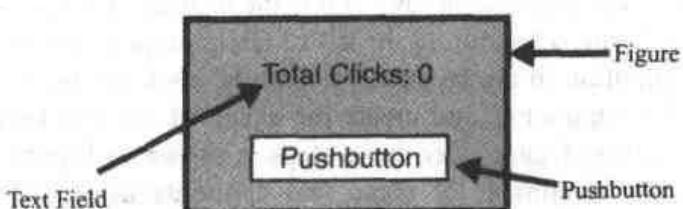


Figure 10.3 Rough layout for a GUI containing a single pushbutton and a single label field.

Step 1: The design of this GUI is very simple. It contains a single pushbutton and a single text field. The callback from the pushbutton will cause the number displayed in the text field to increase by one each time that the button is pressed. A rough sketch of the GUI is shown in Figure 10.3.

Step 2: To lay out the components on the GUI, run the MATLAB function `guide`. When `guide` is executed, it creates the window shown in Figure 10.2.

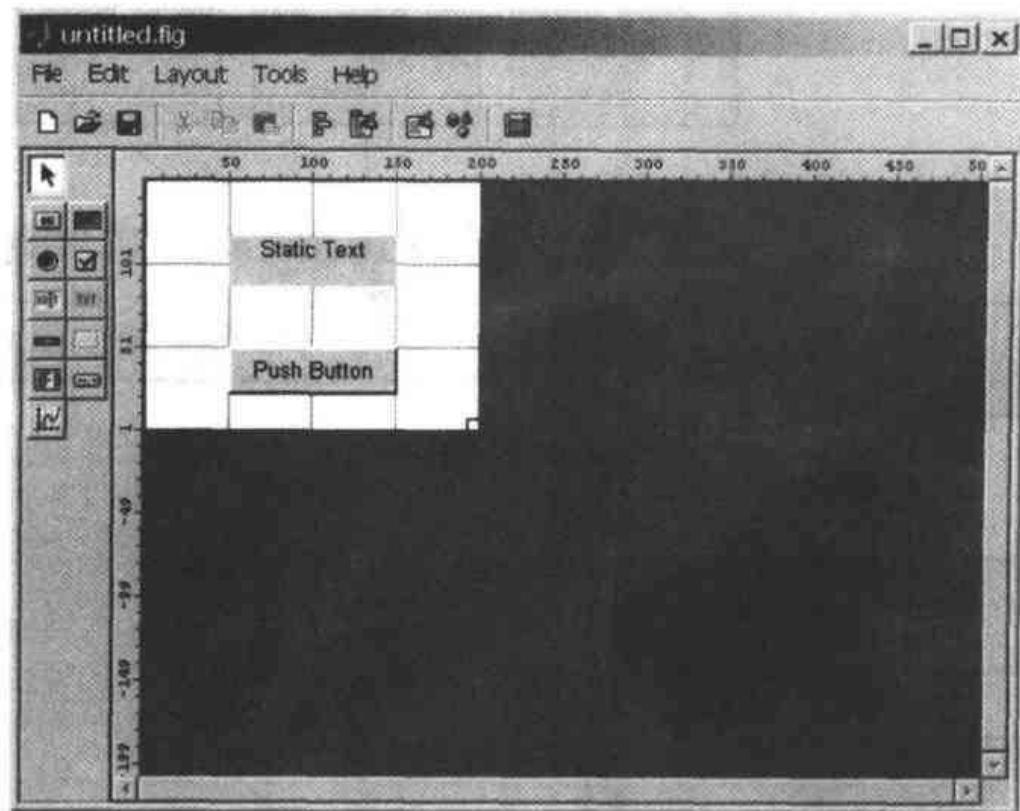


Figure 10.4 The completed GUI layout within the guide window.

First, we must set the size of the layout area, which will become the size of the final GUI. We do this by dragging the small square on the lower right corner of the layout area until it has the desired size and shape. Then, click on the “pushbutton” button in the list of GUI components, and create the shape of the pushbutton in the layout area. Finally, click on the “text” button in the list of GUI components, and create the shape of the text field in the layout area. The resulting figure after these steps is shown in Figure 10.4. We could now adjust the alignment of these two elements using the Alignment Tool, if desired.

Step 3: To set the properties of the pushbutton, click on the button in the layout area and then select “Property Inspector” () from the toolbar. Alternatively, right-click on the button and select “Inspect Properties” from the popup menu. The Property Inspector window shown in Figure 10.5 will appear. Note that this window lists every property available for the pushbutton and allows us to set each value using a GUI interface. The Property Inspector performs the same function as the get and set functions introduced in Chapter 9, but in a much more convenient form.

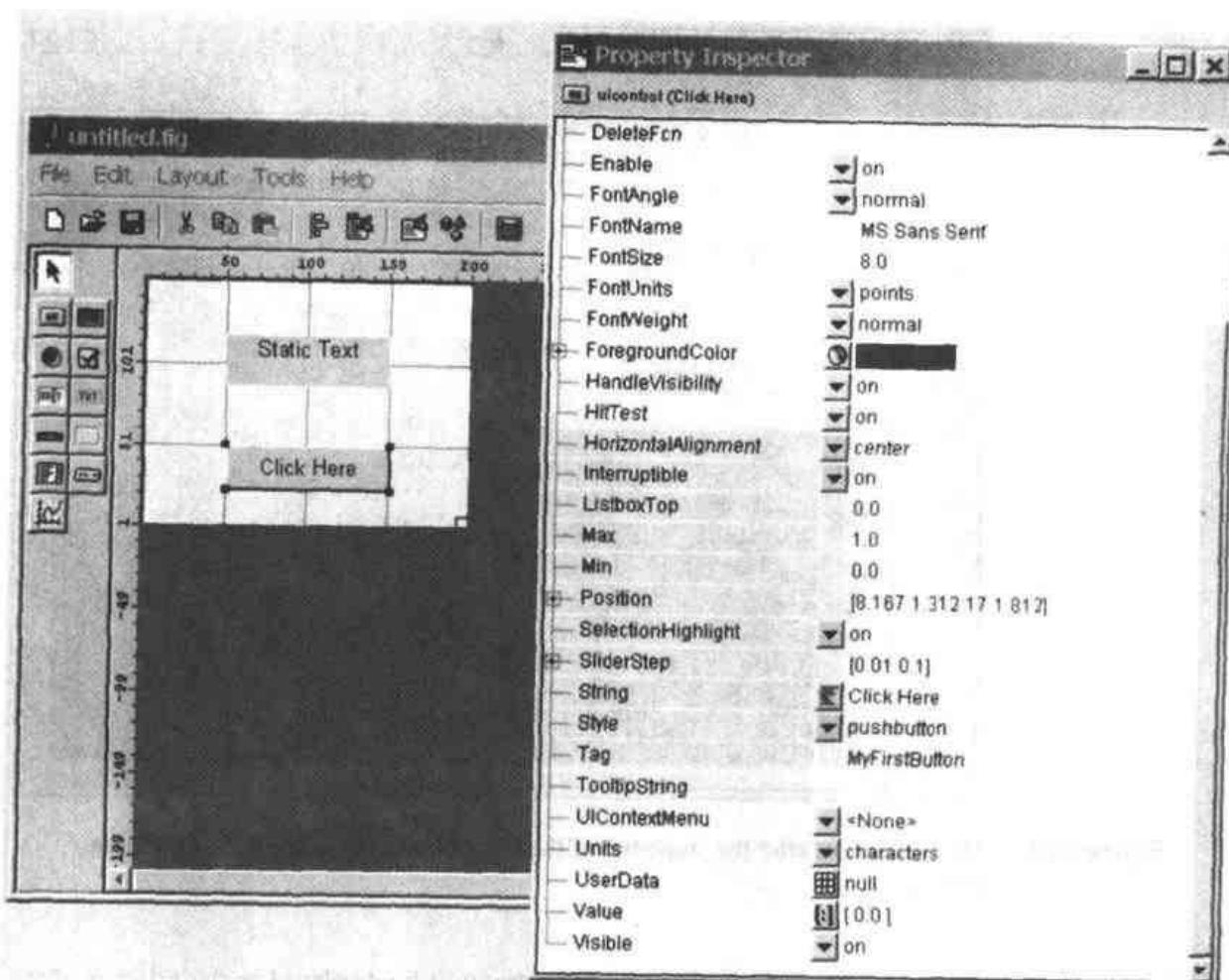


Figure 10.5 The Property Inspector showing the properties of the pushbutton. Note that the String is set to 'Click Here', and the Tag is set to 'MyFirstButton'.

For the pushbutton, we may set many properties such as color, size, font, text alignment, and so on. However, we *must* set two properties: the String property, which contains the text to be displayed, and the Tag property, which is the name of the pushbutton. In this case, the String property will be set to 'Click Here', and the Tag property will be set to MyFirstButton.

For the text field, we *must* set two properties: the String property, which contains the text to be displayed, and the Tag property, which is the name of the text field. This name will be needed by the callback function to locate and update the text field. In this case, the String property will be set to 'Total Clicks: 0', and the Tag property defaulted to 'MyFirstText'. The layout area after these steps is shown in Figure 10.6.

It is possible to set the properties of the figure itself by clicking on a clear spot in the Layout Editor, and then using the Property Inspector to examine and set the figure's properties. Although not required, it is a good idea to set the figure's Name

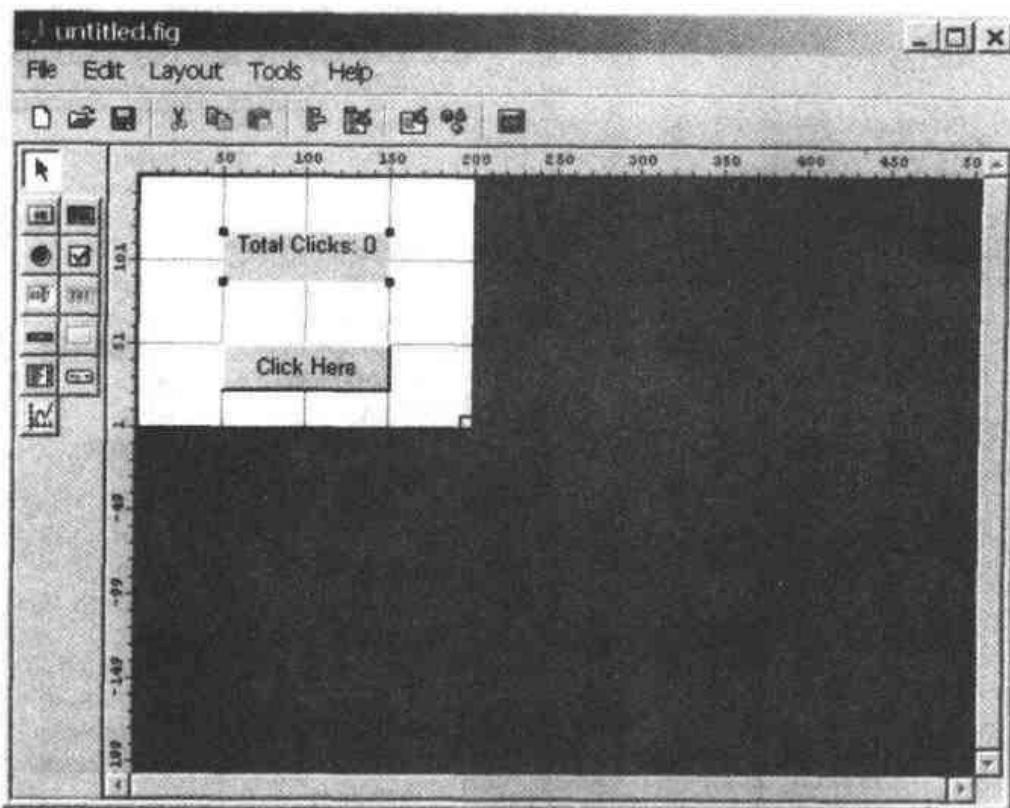


Figure 10.6 The design area after the properties of the pushbutton and the text field have been modified.

property. The string in the Name property will be displayed in the title bar of the resulting GUI when it is executed.

Step 4: We will now save the layout area under the name MyFirstGUI. Select the “File/Save As” menu item, type the name MyFirstGUI as the file name, and click “Save”. This action will automatically create two files, MyFirstGUI.fig and MyFirstGUI.m. The figure file contains the actual GUI that we have created. The M-file contains code that loads the figure file and creates the GUI, plus a skeleton callback function for each active GUI component.

At this point, we have a complete GUI, but one that does not yet do the job it was designed to do. You can start this GUI by typing MyFirstGUI in the Command Window, as shown in Figure 10.7. If the button is clicked on this GUI, the following message will appear in the Command Window:

```
MyFirstButton Callback not implemented yet.
```

A portion of the M-file automatically created by guide is shown in Figure 10.8. This file contains function MyFirstGUI, plus dummy subfunctions implementing the callbacks for each active GUI component. If function MyFirstGUI is called *without* arguments, then the function displays the GUI contained in file

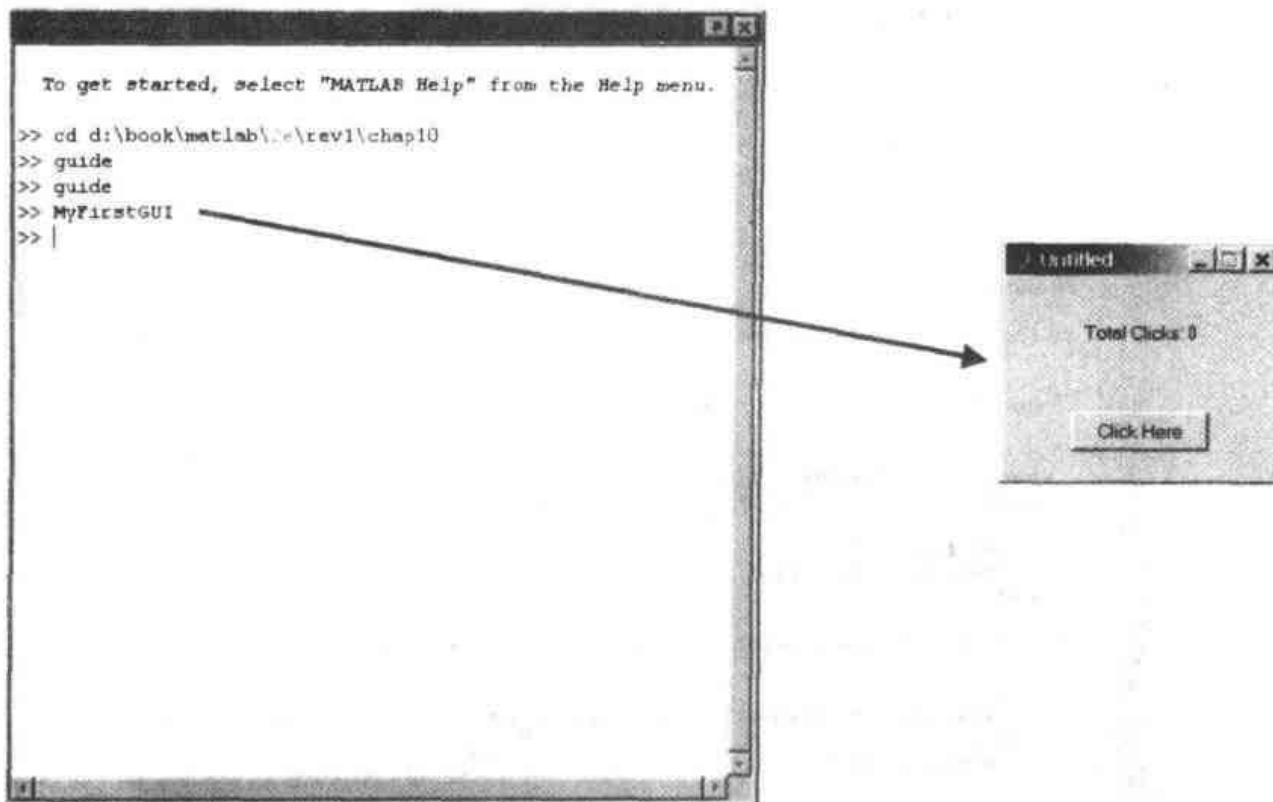


Figure 10.7 Typing `MyFirstGUI` in the Command Window starts the GUI.

`MyFirstGUI.fig`. If function `MyFirstGUI` is called *with* arguments, then the function assumes that the first argument is the name of a subfunction, and it calls that function using `feval`, passing the other arguments on to that function.

Each callback function handles events from a single GUI component. If a mouse click (or keyboard input for Edit Fields) occurs on the GUI component, then the component's callback function will be automatically called by MATLAB. The name of the callback function will be the value in the Tag property of the GUI component plus the characters “_Callback”. Thus, the callback function for `MyFirstButton` will be named `MyFirstButton_Callback`.

M-files created by `guide` contain callbacks for each active GUI component, but these callbacks simply display a message saying that the function of the callback has not been implemented yet.

Step 5: Now, we need to implement the callback subfunction for the pushbutton. This function will include a persistent variable that can be used to count the number of clicks that have occurred. When a click occurs on the pushbutton, MATLAB will call the function `MyFirstGUI` with `MyFirstButton_Callback` as the first argument. Then function `MyFirstGUI` will call subfunction `MyFirstButton_Callback`, as shown in Figure 10.9. This

```

1 function varargout = MyFirstGUI(varargin)
2 % MYFIRSTGUI Application M-file for MyFirstGUI.fig
3 % FIG = MYFIRSTGUI Launch MyFirstGUI.m
4 % MYFIRSTGUI('Callback name', ...) invoke the named callback
5 %
6 % Last Modified by GUIDE v2.0 22 Jun 2001 21:14:46
7
8 if nargin == 0 % LAUNCH GUI
9    fig = openfig(mfilename,'reuse'); % If called without an argument, open the GUI.
10   % use system color scheme for figure
11   set(fig,'Color',get(0,'defaultFigureBackgroundColor'));
12
13   % Generate a structure of handles to pass to callbacks, and store it.
14   handles = guihandles(fig);
15   guidata(fig, handles);
16
17   if nargin > 0
18      varargout{1} = fig;
19   end
20
21 elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION OR CALLBACK
22
23   try
24      [varargout{1:nargin}] = feval(varargin{:}); % FEVAL switchyard
25   catch
26      disp(lasterr);
27   end
28
29 end
30
31 % Callbacks are subfunctions.
32
33 %
34 function varargout = MyFirstButton_Callback(h, eventdata, handles, varargin)
35 % Stub for Callback of the uicontrol handles.MyFirstButton
36 disp('MyFirstButton Callback not implemented yet.')
37

```

Figure 10.8 The M-file for MyFirstGUI, automatically created by guide.

function should increase the count of clicks by one, create a new text string containing the count, and store the new string in the String property of the text field MyFirstText. A function to perform this step is shown below:

```

function varargout = MyFirstButton_Callback(h, eventdata, ...
                                         handles, varargin)

% Declare and initialize variable to store the count
persistent count
if isempty(count)
    count = 0;
end

```

```

% Update count
count = count + 1;

% Create new string
str = sprintf('Total Clicks: %d',count);

% Update the text field
set (handles.MyFirstText,'String',str);

```

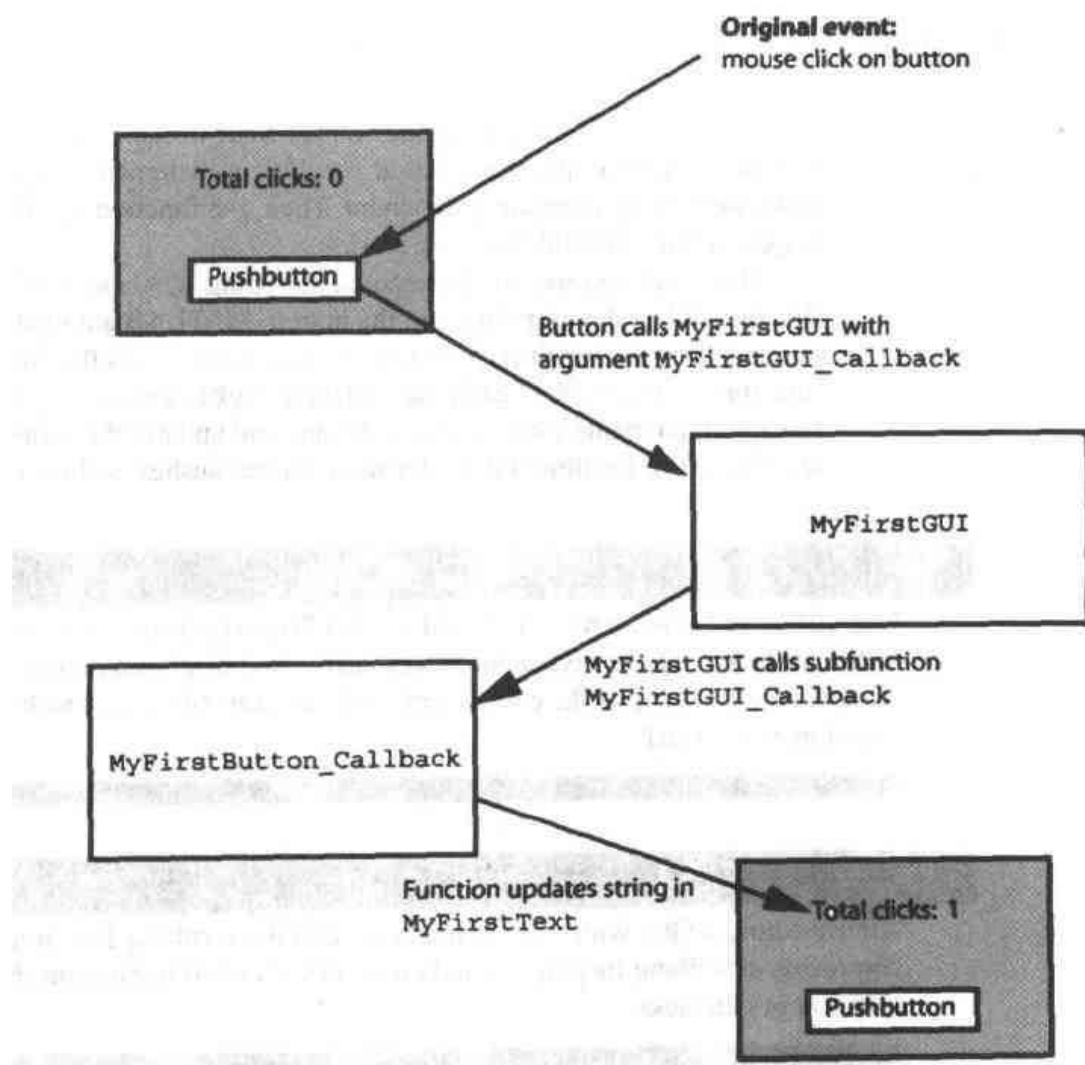


Figure 10.9 Event handling in program `MyFirstGUI`. When a user clicks on the button with the mouse, the function `MyFirstGUI` is called automatically with the argument `MyFirstButton_Callback`. Function `MyFirstGUI` in turn calls subfunction `MyFirstButton_Callback`. This function increments `count`, and then saves the new count in the text field on the GUI.



Figure 10.10 The resulting program after three button pushes.

Note that this function declares a persistent variable `count` and initializes it to zero. Each time that the function is called, it increments `count` by 1 and creates a new string containing the count. Then, the function updates the string displayed in the text field `MyFirstText`.

The resulting program is executed by typing `MyFirstGUI` in the Command Window. When the user clicks on the button, MATLAB automatically calls function `MyFirstGUI` with `MyFirstButton_Callback` as the first argument, and function `MyFirstGUI` calls subfunction `MyFirstButton_Callback`. This function increments variable `count` by one and updates the value displayed in the text field. The resulting GUI after three button pushes is shown in Figure 10.10.

Good Programming Practice

Use `guide` to lay out a new GUI, and use the Property Inspector to set the initial properties of each component such as the text displayed on the component, the color of the component, and the name of the callback function, if required.

Good Programming Practice

After creating a GUI with `guide`, manually edit the resulting function to add comments describing its purpose and components and to implement the function of callbacks.

10.2.1 A Look Under the Hood

Figure 10.8 shows the M-file that was automatically generated by `guide` for `MyFirstGUI`. We will now examine this M-file more closely to understand how it works.

First, let's look at the function declaration itself. Note that this function uses `varargin` to represent its input arguments and `varargout` to represent its output results. As we learned in Chapter 7, function `varargin` can represent an arbitrary number of input arguments, and function `varargout` can represent a varying number of output arguments. Therefore, a user can call function `MyFirstGUI` with any number of arguments.

Calling the M-File without Arguments

If the user calls `MyFirstGUI` without arguments, the value returned by `nargin` will be zero. In this case, the program loads the GUI from the figure file `MyFirstGUI.fig` using the `openfig` function. The form of this function is

```
fig = openfig(mfilename, 'reuse');
```

where `mfilename` is the name of the figure file to load. The second argument in the function specifies whether there can be only one copy of the figure running at a given time, or whether multiple copies can be run. If the argument is '`reuse`', then only one copy of the figure can be run. If `openfig` is called with the '`reuse`' option and the specified figure already exists, the preexisting figure will be brought to the top of the screen and reused. In contrast, if the argument is '`new`', multiple copies of the figure can be run. If `openfig` is called with the '`new`' option, a new copy of the figure will be created each time. By default, a GUI created by `guide` has the '`reuse`' option, so only one copy of the figure can exist at any time. If you want to have multiple copies of the GUI, you must manually edit this function call.

After the figure is loaded, the function executes the statement

```
set(fig, 'Color', get(0, 'defaultUicontrolBackgroundColor'));
```

This function sets the background color of the figure to match the default background color used by the computer on which MATLAB is executing. This function makes the color of the GUI match the color of native windows on the computer. Therefore, a GUI can be written on a Windows-based PC and used on a UNIX-based computer, and vice versa. It will look natural in either environment.

The next two statements create a structure containing the handles of all the objects in the current figure and store that structure as application data in the figure.

```
handles = guihandles(fig);
guidata(fig, handles);
```

Function `guihandles` creates a structure containing handles to all of the objects within the specified figure. The element names in the structure correspond to the `Tag` properties of each GUI component, and the values are the handles of each component. For example, the handle structure returned in `MyFirstGUI.m` is

```
>> handles = guihandles(fig)
handles =
    figure1: 99.0005
```

```
MyFirstText: 3.0021
MyFirstButton: 100.0007
```

There are three GUI components in this figure—the figure itself, plus a text field and a pushbutton. Function `guidata` saves the handles structure as application data in the figure, using the `setappdata` function that we studied in Chapter 9.

The final set of statements here return the figure handle to the caller if an output argument was specified in the call to `MyFirstGUI`.

```
if nargin > 0
    varargout{1} = fig;
end
```

Calling the M-File with Arguments

If the user calls `MyFirstGUI` with arguments, the value returned by `nargin` will be greater than zero. In this case, the program treats the first calling argument as a callback function name and executes the function using `feval`. This function executes the function named in `varargin{1}` and passes all of the remaining arguments (`varargin{2}`, `varargin{3}`, etc.) to the function. This mechanism allows callback functions to be subfunctions that cannot be accidentally called from some other part of the program.

10.2.2 The Structure of a Callback Subfunction

Every callback subfunction has the standard form

```
function varargout = ComponentTag_Callback(h, eventdata, ...
    handles, varargin)
```

where `ComponentTag` is the name of the component generating the callback (the string in its `Tag` property). The arguments of this subfunction are

- **`h`**—The handle of the parent figure
- **`eventdata`**—A currently unused (in MATLAB Version 6) array.
- **`handles`**—The `handles` structure contains the handles of all GUI components on the figure.
- **`varargin`**—A supplemental argument passing any additional calling arguments to the callback function. A programmer can use this feature to provide additional information to the callback function if needed.

Note that each callback function has full access to the `handles` structure, so each callback function can modify any GUI component in the figure. We took advantage of this structure in the callback function for the pushbutton in `MyFirstGUI`, where the callback function for the pushbutton modified the text displayed in the text field.

```
% Update the text field
set (handles.MyFirstText, 'String', str);
```

10.2.3 Adding Application Data to a Figure

It is possible to store application-specific information needed by a GUI program in the `handles` structure instead of using global or persistent memory for that data. The resulting GUI design is more robust because other MATLAB programs cannot accidentally modify the global GUI data and because multiple copies of the same GUI cannot interfere with each other.

To add local data to the `handles` structure, we must manually modify the M-file after it is created by `guide`. A programmer adds the application data to the `handles` structure after the call to `guihandles` and before the call to `guidata`. For example, to add the number of mouse clicks `count` to the `handles` structure, we would modify the program as follows:

```
% Generate a structure of handles to pass to callbacks
handles = guihandles(fig);

% Add count to the structure.
handles.count = 0;

% Store the structure
guidata(fig, handles);
```

This application data will now be passed with the `handles` structure to every callback function, where it can be used.

A version of the pushbutton callback function that uses the `count` value in the `handles` data structure is shown below. Note that we must save the `handles` structure with a call to `guidata` if any of the information in it has been modified.

```
function varargout = MyFirstButton_Callback(h, eventdata, ...
    handles, varargin)

% Update count
handles.count = handles.count + 1;

% Save the updated handles structure
guidata(h, handles);

% Create new string
str = sprintf('Total Clicks: %d', handles.count);

% Update the text field
set(handles.MyFirstText, 'String', str);
```

Good Programming Practice

Store GUI application data in the `handles` structure, so that it will automatically be available to any callback function.

Good Programming Practice

If you modify any of the GUI application data in the handles structure, be sure to save the structure with a call to guidata before exiting the function where the modifications occurred.

10.2.4 A Few Useful Functions

Three special functions are used occasionally in the design of callback functions: `gcbo`, `gcbf`, and `findobj`. Although these functions are less needed with MATLAB 6 GUIs than with earlier versions, they are still very useful, and a programmer is sure to encounter them.

Function `gcbo` (*get callback object*) returns the handle of the object that generated the callback, and function `gcbf` (*get callback figure*) returns the handle of the figure containing that object. These functions can be used by the callback function to determine the object and figure producing the callback, so that it can modify objects on that figure.

Function `findobj` searches through all of the child objects within a parent object, looking for ones that have a specific value of a specified property. It returns a handle to any objects with the matching characteristics. The most common form of `findobj` is

```
Hndl = findobj(parent, 'Property', Value);
```

where `parent` is the handle of a parent object such as a figure, '`Property`' is the property to examine, and '`value`' is the value to look for.

For example, suppose that a programmer would like to change the text on a pushbutton with the tag '`Button1`' when a callback function executes. The programmer could find the pushbutton and replace the text with the following statements

```
Hndl = findobj( gcbf, 'Tag', 'Button1' );
set( Hndl, 'String', 'New text' );
```

10.3 Object Properties

Every GUI object includes an extensive list of properties that can be used to customize the object. These properties are slightly different for each type of object (figures, axes, uicontrols, etc.). All of the properties for all types of objects are documented on the online Help Browser, but a few of the more important properties for figure and uicontrol objects are summarized in Tables 10.2 and 10.3.

Object properties may be modified using either the Property Inspector or the `get` and `set` functions. Although the Property Inspector is a convenient way to adjust properties during GUI design, we must use `get` and `set` to adjust them dynamically from within a program, such as in a callback function.

Table 10.2 Important figure Properties

Property	Description
Color	Specifies the color of the figure. The value is either a predefined color such as 'r', 'g', or 'b', or else a 3-element vector specifying the red, green, and blue components of the color on a 0-1 scale. For example, the color magenta would be specified by [1 0 1].
MenuBar	Specifies whether or not the default set of menus appears on the figure. Possible values are 'figure' to display the default menus, or 'none' to delete them.
Name	A string containing the name that appears in the title bar of a figure.
NumberTitle	Specifies whether or not the figure number appears in the title bar. Possible values are 'on' or 'off'.
Position	Specifies the position of a figure on the screen, in the units specified by the 'units' property. This value accepts a 4-element vector in which the first two elements are the x and y positions of the lower left-hand corner of the figure, and the next two elements are the width and height of the figure.
SelectionType	Specifies the type of selection for the last mouse click on this figure. A single click returns type 'normal'. A double click returns type 'open'. There are additional options; see the MATLAB on-line documentation.
Tag	The 'name' of the figure, which can be used to locate it.
Units	The units used to describe the position of the figure. Possible choices are 'inches', 'centimeters', 'normalized', 'points', 'pixels', or 'characters'. The default units are 'pixels'.
Visible	Specifies whether or not this figure is visible. Possible values are 'on' or 'off'.
WindowStyle	Specifies whether this figure is normal or modal (see discussion of Dialog Boxes). Possible values are 'normal' or 'modal'.

10.4 Graphical User Interface Components

This section summarizes the basic characteristics of common graphical user interface components. It describes how to create and use each component, as well as the types of events each component can generate. The components discussed in this section are

- Text Fields
- Edit Boxes
- Frames
- Pushbuttons
- Toggle Buttons
- Checkboxes
- Radio Buttons
- Popup Menus
- List Boxes
- Sliders

Table 10.3 Important uicontrol Properties

Property	Description
BackgroundColor	Specifies the background color of the object. The value is either a predefined color such as 'r', 'g', or 'b', or else a 3-element vector specifying the red, green, and blue components of the color on a 0-1 scale. For example, the color magenta would be specified by [1 0 1].
Callback	Specifies the name and parameters of the function to be called when the object is activated by a keyboard or text input.
Enable	Specifies whether or not this object is selectable. If it is not enabled, it will not respond to mouse or keyboard input. Possible values are 'on' or 'off'.
FontAngle	A string containing the font angle for text displayed on the object. Possible values are 'normal', 'italic', and 'oblique'.
FontName	A string containing the font name for text displayed on the object.
FontSize	A number specifying the font size for text displayed on the object. By default, the font size is specified in points.
FontWeight	A string containing the font weight for text displayed on the object. Possible values are 'light', 'normal', 'demi', and 'bold'.
ForegroundColor	Specifies the foreground color of the object.
HorizontalAlignment	Specifies the horizontal alignment of a text string within the object. Possible values are 'left', 'center', and 'right'.
Max	The maximum size of the value property for this object.
Min	The minimum size of the value property for this object.
Parent	The handle of the figure containing this object.
Position	Specifies the position of the object on the screen, in the units specified by the 'units' property. This value accepts a 4-element vector in which the first two elements are the x and y positions of the lower left-hand corner of the object <i>relative to the figure containing it</i> , and the next two elements are the width and height of the object.
Tag	The 'name' of the object, which can be used to locate it.
TooltipString	Specifies the help text to be displayed when a user places the mouse pointer over an object.
Units	The units used to describe the position of the figure. Possible choices are 'inches', 'centimeters', 'normalized', 'points', 'pixels', or 'characters'. The default units are 'pixels'.
Value	The current value of the uicontrol. For toggle buttons, check boxes, and radio buttons, the value is max when the button is on and min when the button is off. Other controls have different meanings for this term.
Visible	Specifies whether or not this object is visible. Possible values are 'on' or 'off'.

10.4.1 Text Fields

A **text field** is a graphical object that displays a text string. You can specify how the text is aligned in the display area by setting the horizontal alignment property. By default, text fields are horizontally centered. A text field is created by creating a `uicontrol` whose style property is 'edit'. A text field may be added to a GUI by using the text tool () in the Layout Editor.

Text fields do not create callbacks, but the value displayed in the text field can be updated in a callback function by changing the text field's `String` property, as shown in Section 10.2.

10.4.2 Edit Boxes

An **edit box** is a graphical object that allows a user to enter a text string. The edit box generates a callback when the user presses the Enter key after typing a string into the box. An edit box is created by creating a `uicontrol` whose style property is 'edit'. An edit box may be added to a GUI by using the edit box tool () in the Layout Editor.

Figure 10.11a shows a simple GUI containing an edit box named 'EditBox' and a text field named 'TextBox'. When a user types a string into the edit box, it automatically calls the function `EditBox_Callback`, which is shown in Figure 10.11b. This function locates the edit box using the handles structure and recovers the string typed by the user. Then, it locates the text field and displays the string in the text field. Figure 10.12 shows this GUI just after it has started and after the user has typed the word "Hello" in the edit box.

10.4.3 Frames

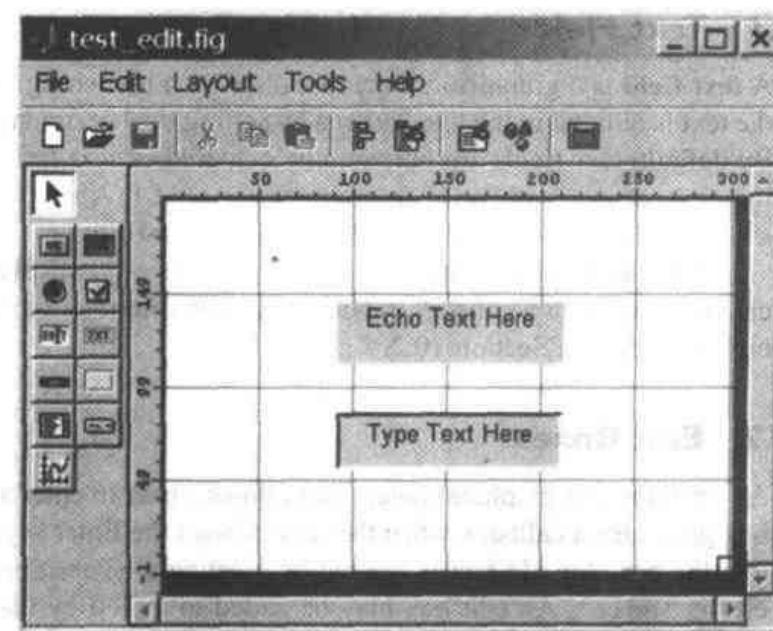
A **frame** is a graphical object that displays a rectangle on the GUI. You can use frames to draw boxes around groups of logically related objects. For example, a frame is used to group the radio buttons together on Figure 10.1.

A frame is created by creating a `uicontrol` whose style property is 'frame'. A frame may be added to a GUI by using the frame tool () in the Layout Editor. Frames do not generate callbacks.

10.4.4 Pushbuttons

A **pushbutton** is a component that a user can click on to trigger a specific action. The pushbutton generates a callback when the user clicks the mouse on it. A pushbutton is created by creating a `uicontrol` whose style property is 'pushbutton'. A pushbutton may be added to a GUI by using the pushbutton tool () in the Layout Editor.

Function `MyFirstGUI` in Figure 10.10 illustrated the use of pushbuttons.



(a)

```

function varargout = EditBox_Callback(h, eventdata, ...
    handles, varargin)

% Find the value typed into the edit box
str = get (handles.EditBox, 'String');

% Place the value into the text field
set (handles.TextBox, 'String',str);

```

(b)

Figure 10.11 (a) Layout of a simple GUI with an edit box and a text field. (b) The callback function for this GUI.

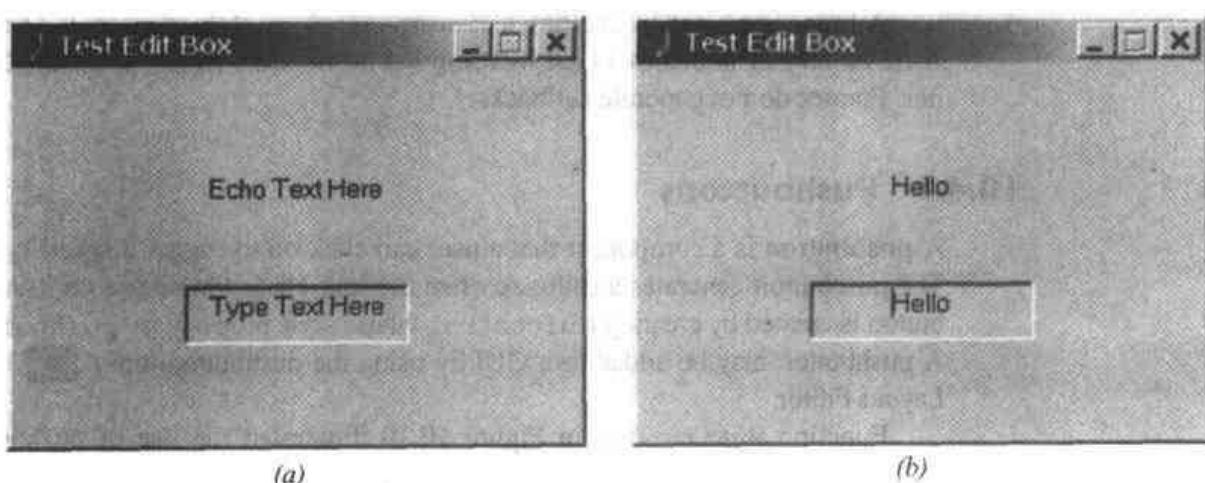


Figure 10.12 (a) The GUI produced by program test_edit. (b) The GUI after a user types Hello into the edit box and presses Enter.

10.4.5 Toggle Buttons

A **toggle button** is a type of button that has two states: on (depressed) and off (not depressed). A toggle button switches between these two states whenever the mouse clicks on it, and it generates a callback each time. The 'Value' property of the toggle button is set to max (usually 1) when the button is on, and min (usually 0) when the button is off.

A toggle button is created by creating a uicontrol whose style property is 'togglebutton'. A toggle button may be added to a GUI by using the toggle button tool (■) in the Layout Editor.

Figure 10.13a shows a simple GUI containing a toggle button named 'ToggleButton' and a text field named 'TextBox'. When a user clicks on the toggle button, it automatically calls the function ToggleButton_Callback, which is shown in Figure 10.13b. This function locates the toggle button using the handles structure and recovers its state from the 'Value' property. Then, the function locates the text field and displays the state in the text field. Figure 10.14 shows this GUI just after it has started, and after the user has clicked on the toggle button for the first time.

10.4.6 Checkboxes and Radio Buttons

Checkboxes and radio buttons are essentially identical to toggle buttons except that they have different shapes. Like toggle buttons, checkboxes and radio buttons have two states: on and off. They switch between these two states whenever the mouse clicks on them, generating a callback each time. The 'Value' property of the checkbox or radio button is set to max (usually 1) when they are on, and min (usually 0) when they are off. Both checkboxes and radio buttons are illustrated in Figure 10.1.

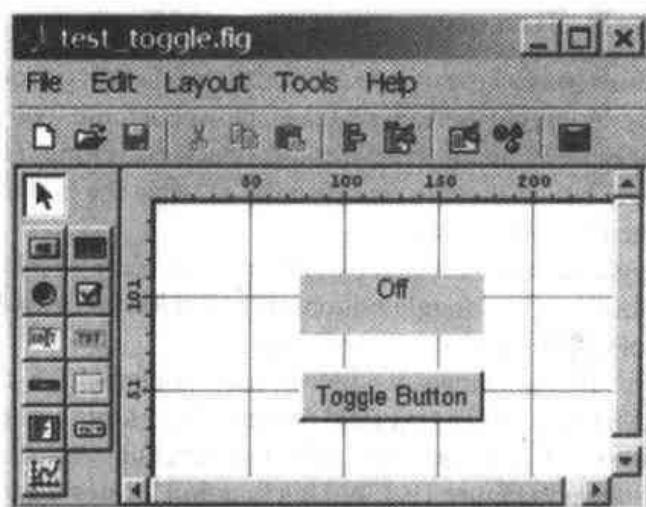
A checkbox is created by creating a uicontrol whose style property is 'checkbox', and a radio button is created by creating a uicontrol whose style property is 'radiobutton'. A checkbox may be added to a GUI by using the checkbox tool (☒) in the Layout Editor, and a radio button may be added to a GUI by using the radio button tool (●) in the Layout Editor.

Checkboxes are traditionally used to display on/off options, and groups of radio buttons are traditionally used to select among mutually exclusive options.

Figure 10.15a shows an example of how to create a group of mutually exclusive options with radio buttons. The GUI in this figure creates three radio buttons, labeled "Option 1," "Option 2," and "Option 3." Each radio button uses the same callback function, but with a separate parameter.

The corresponding callback functions are shown in Figure 10.15b. When the user clicks on a radio button, the corresponding callback function is executed. That function sets the text box to display the current option, turns on that radio button, and turns off all other radio buttons.

Note that the GUI uses a frame to group the radio buttons together, making it obvious that they are a set. Figure 10.16 shows this GUI after Option 2 has been selected.



(a)

```

function varargout = ToggleButton_Callback(h, eventdata, ...
    handles, varargin)

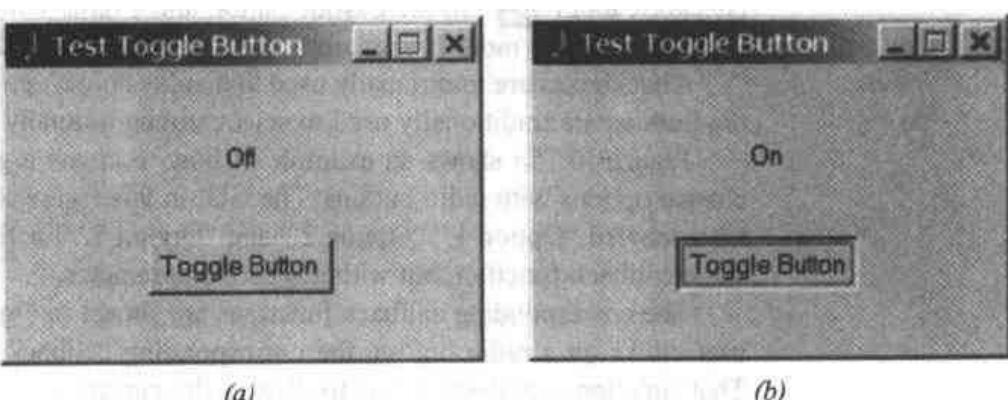
% Find the state of the toggle button
state = get(handles.ToggleButton, 'Value');

% Place the value into the text field
if state == 0
    set (handles.TextBox, 'String', 'Off');
else
    set (handles.TextBox, 'String', 'On');
end

```

(b)

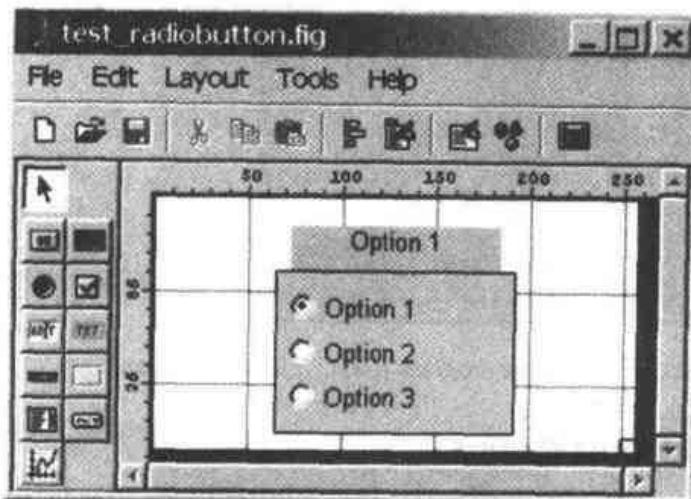
Figure 10.13 (a) Layout of a simple GUI with a toggle button and a text field. (b) The callback function for this GUI.



(a)

(b)

Figure 10.14 (a) The GUI produced by program test_togglebutton when the toggle button is off. (b) The GUI when the toggle button is on.



(a)

```

function varargout = radiobutton1_Callback(hObject, eventdata, handles, varargin)
set(handles.Label1,'String','Option 1');
set(handles.radiobutton1,'Value',1);
set(handles.radiobutton2,'Value',0);
set(handles.radiobutton3,'Value',0);

function varargout = radiobutton2_Callback(hObject, eventdata, handles, varargin)
set(handles.Label1,'String','Option 2');
set(handles.radiobutton1,'Value',0);
set(handles.radiobutton2,'Value',1);
set(handles.radiobutton3,'Value',0);

function varargout = radiobutton3_Callback(hObject, eventdata, handles, varargin)
set(handles.Label1,'String','Option 3');
set(handles.radiobutton1,'Value',0);
set(handles.radiobutton2,'Value',0);
set(handles.radiobutton3,'Value',1);

```

(b)

Figure 10.15 (a) Layout of a simple GUI with three radio buttons in a frame, plus a text field to display the current selection. (b) The callback functions for this GUI.

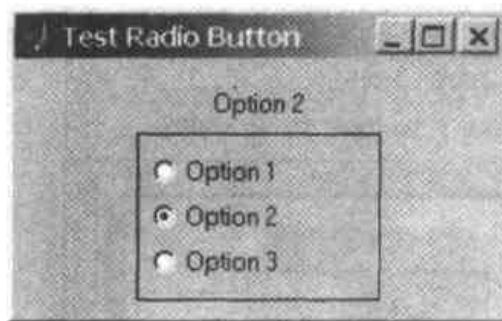


Figure 10.16 The GUI produced by program `test_radiobutton`.

10.4.7 Popup Menus

Popup menus are graphical objects that allow a user to select one of a mutually exclusive list of options. The list of options that the user can select among is specified by a cell array of strings, and the 'Value' property indicates which of the strings is currently selected. A popup menu may be added to a GUI by using the popup menu tool () in the Layout Editor.

Figure 10.17a shows an example of a popup menu. The GUI in this figure creates a popup menu with five options, labeled "Option 1," "Option 2," and so forth.

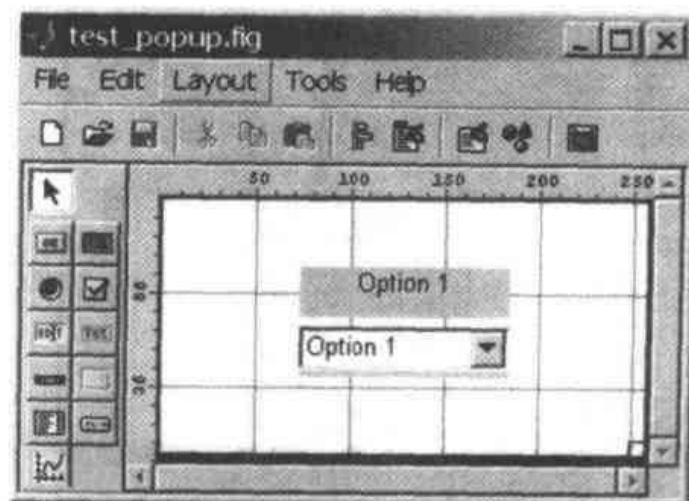
The corresponding callback function is shown in Figure 10.17b. The callback function recovers the selected option by checking the 'Value' parameter of the popup menu, and creates and displays a string containing that value in the text field. Figure 10.18 shows this GUI after Option 4 has been selected.

10.4.8 List Boxes

List boxes are graphical objects that display many lines of text and allow a user to select one or more of those lines. If there are more lines of text than can fit in the list box, a scroll bar will be created to allow the user to scroll up and down within the list box. The lines of text that the user can select among are specified by a cell array of strings, and the 'Value' property indicates which of the strings are currently selected.

A list box is created by creating a `uicontrol` whose style property is '`listbox`'. A list box may be added to a GUI by using the listbox tool () in the Layout Editor.

List boxes can be used to select a single item from a selection of possible choices. In normal GUI usage, a single mouse click on a list item selects that item but does not cause an action to occur. Instead, the action waits on some external trigger, such as a pushbutton. However, a mouse double-click causes an action to happen immediately. Single-click and double-click events can be distinguished using the `SelectionType` property of the figure in which the clicks occurred. A single mouse click will place the string '`normal`' in the `SelectionType` property, and a double mouse click will place the string '`open`' in the `SelectionType` property.



(a)

```
function varargout = radiobutton1_Callback(hObject, eventdata, handles, varargin)

% Find the value of the popup menu
value = get(handles.Popup1, 'Value');

% Place the value into the text field
str = ['Option ' num2str(value)];
set (handles.Label1, 'String', str);
```

(b)

Figure 10.17 (a) Layout of a simple GUI with a popup menu and a text field to display the current selection. (b) The callback functions for this GUI.

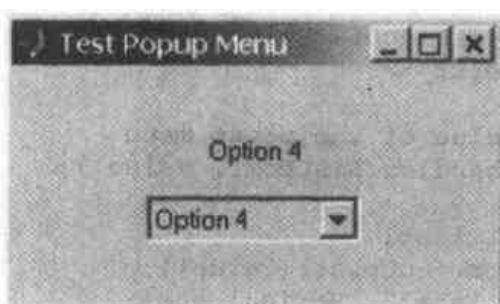
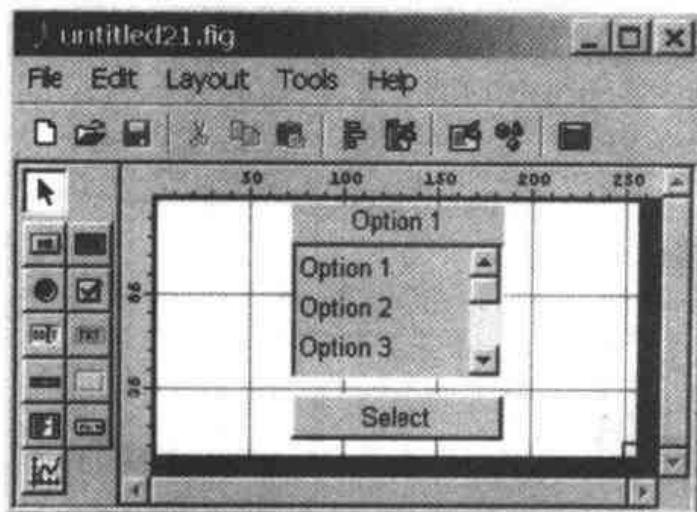


Figure 10.18 The GUI produced by program test_popup.

It is also possible for a list box to allow multiple selections from the list. If the difference between the `max` and `min` properties of the list box is greater than one, then multiple selection is allowed. Otherwise, only one item may be selected from the list.

Figure 10.19a shows an example of a single-selection list box. The GUI in this figure creates a list box with eight options, labeled “Option 1”, “Option 2,”



(a)

```

function varargout = Button1_Callback(h, eventdata, ...
    handles, varargin)

% Find the value of the popup menu
value = get(handles.Listbox1, 'Value');

% Update text label
str = ['Option ' num2str(value)];
set (handles.Label1, 'String',str);

function varargout = Listbox1_Callback(h, eventdata, ...
    handles, varargin)

% If this was a double click, update the label.
selectiontype = get(gcbf, 'SelectionType');
if selectiontype(1) == 'o'

    % Find the value of the popup menu
    value = get(handles.Listbox1, 'Value');

    % Update text label
    str = ['Option ' num2str(value)];
    set (handles.Label1, 'String',str);
end

```

(b)

Figure 10.19 (a) Layout of a simple GUI with a list box, a pushbutton, and a text field. (b) The callback functions for this GUI. Note that selection can occur either by clicking the button or double-clicking on an item in the listbox.

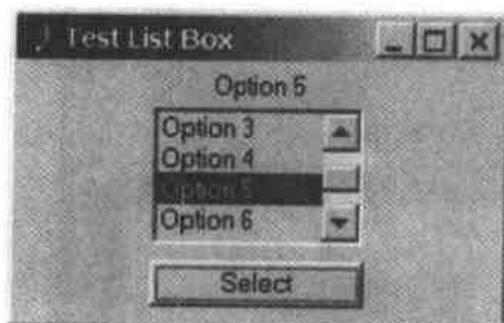


Figure 10.20 The GUI produced by program `test_listbox`.

and so forth. In addition, the GUI creates a pushbutton to perform selection and a text field to display the selected choice. Both the list box and the pushbutton generate callbacks.

The corresponding callback functions are shown in Figure 10.19b. If a selection is made in the list box, then function `Listbox1_Callback` will be executed. This function will check the *figure producing the callback* (using function `gcbf`) to see if the selecting action was a single-click or a double-click. If it was a single-click, the function does nothing. If it was a double-click, then the function gets the selected value from the listbox, and writes an appropriate string into the text field.

If the pushbutton is selected, then function `Button1_Callback` will be executed. This function gets the selected value from the listbox, and writes an appropriate string into the text field. The GUI produced by program `test_listbox` is shown in Figure 10.20.

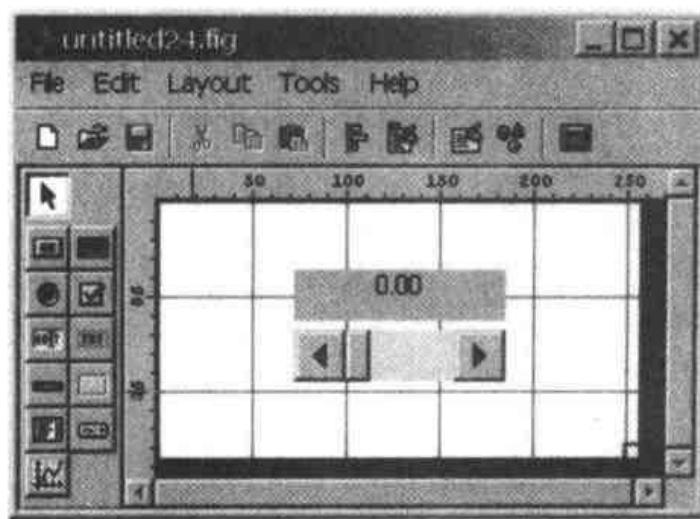
In an end of chapter exercise, you will be asked to modify this example to allow multiple selections in the list box.

10.4.9 Sliders

Sliders are graphical objects that allow a user to select values from a continuous range between a specified minimum value and a specified maximum value by moving a bar with a mouse. The 'Value' property of the slider is set to a value between min and max depending on the position of the slider.

A slider is created by creating a `uicontrol` whose style property is 'slider'. A slider may be added to a GUI by using the slider tool (■) in the Layout Editor.

Figure 10.21a shows the layout for a simple GUI containing a slider and a text field. The 'Min' property for this slider is set to zero, and the 'Max' property is set to ten. When a user drags the slider, it automatically calls the function `Slider1_Callback`, which is shown in Figure 10.21b. This function gets the value of the slider from the 'Value' property and displays the value in the text field. Figure 10.22 shows this GUI with the slider at some intermediate position in its range.



(a)

```

function varargout = Slider1_Callback(h, eventdata, ...
    handles, varargin)

% Find the value of the slider
value = get(handles.Slider1,'Value');

% Place the value in the text field
str = sprintf('%2f',value);
set (handles.Label1,'String',str);

```

(b)

Figure 10.21 (a) Layout of a simple GUI with a slider and a text field. (b) The callback function for this GUI.

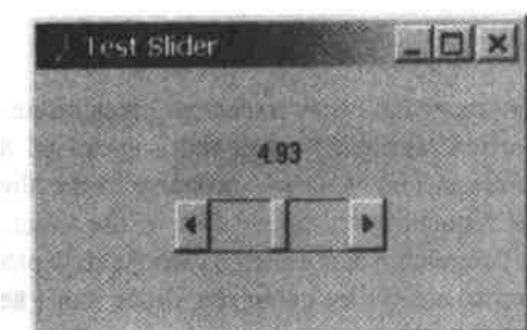


Figure 10.22 The GUI produced by program `test_slider`.

Example 10.1—Temperature Conversion

Write a program that converts temperature from degrees Fahrenheit to degrees Celsius and vice versa over the range 0–100° C, using a GUI to accept data and display results. The program should include an edit box for the temperature in degrees Fahrenheit, an edit box for the temperature in degrees Celsius, and a slider to allow for the continuous adjustment of temperature. The user should be able to enter temperatures in either edit box, or by moving the slider, and all GUI elements should adjust to the corresponding values.

Solution

To create this program, we will need a text field and an edit box for the temperature in degrees Fahrenheit, another text field and an edit box for the temperature in degrees Celsius, and a slider. We will also need a function to convert degrees Fahrenheit to degrees Celsius, and a function to convert degrees Celsius to degrees Fahrenheit. Finally, we will need to write a callback function to support user inputs.

The range of values to convert will be 32–212° F or 0–100° C, so it will be convenient to set up the slider to cover the range 0–100, and to treat the value of the slider as a temperature in degrees C.

The first step in this process is to use guide to design the GUI. We can use guide to create the five required GUI elements and locate them in approximately the correct positions. Then, we can use the Property Inspector to perform the following steps:

1. Select appropriate names for each GUI element and store them in the appropriate Tag properties. The names will be 'Label1', 'Label2', 'Edit1', 'Edit2', and 'Slider1'.
2. Store 'Degrees F' and 'Degrees C' in the String properties of the two labels.
3. Set the slider's minimum and maximum limits to 0 and 100, respectively.
4. Store initial values in the String property of the two edit fields and in the Value property of the slider. We will initialize the temperature to 32° F or 0° C, which corresponds to a slider value of 0.
5. Set the Name property of the figure containing the GUI to 'Temperature Conversion'.

Once these changes have been made, the GUI should be saved to file `temp_conversion.fig`. This will produce both both a figure file and a matching M-file. The M-file will contain stubs for the three callback functions needed by the edit fields and the slider. The resulting GUI is shown during the layout process in Figure 10.23.

The next step in the process is to create the functions to convert degrees Fahrenheit to degrees Celsius. Function `to_c` will convert temperature from degrees Fahrenheit to degrees Celsius. It must implement the equation

$$\text{deg } C = \frac{5}{9} (\text{deg } F - 32) \quad (10.1)$$

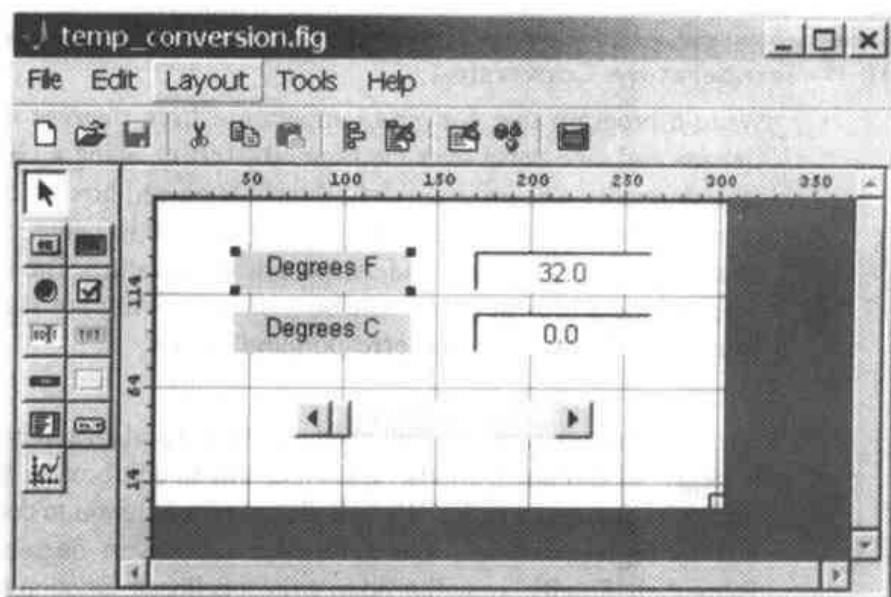


Figure 10.23 Layout of the temperature conversion GUI.

The code for this function is

```
function deg_c = to_c(deg_f)

% Convert degrees Fahrenheit to degrees C.
deg_c = (5/9) * (deg_f - 32);
```

Function `to_f` will convert temperature from degrees Celsius to degrees Fahrenheit. It must implement the equation

$$\text{deg } F = \frac{9}{5} (\text{deg } C) + 32 \quad (10.2)$$

The code for this function is

```
function deg_f = to_f(deg_c)

% Convert degrees Celsius to degrees Fahrenheit.
deg_f = (9/5) * deg_c + 32;
```

Finally, we must write the callback functions to tie it all together. The functions must respond to either edit box or the slider and update all three components. (Note that we will update even the edit box that the user types into, so that the data can be displayed with a consistent format at all times and to correct errors if the user types an out-of-range input value.)

There is an extra complication here because the values entered into edit boxes are *strings*, and we want to treat them as *numbers*. If a user types the value 100 into an edit box, he or she has really created the string '100', not the number 100. The callback function must convert the strings into numbers so that the

```

function varargout = Edit1_Callback(h, eventdata, ...
    handles, varargin)

% Update all temperature values
deg_f = str2num( get(h,'String') );
deg_f = max( [ 32 deg_f] );
deg_f = min( [212 deg_f] );
deg_c = to_c(deg_f);

% Now update the fields
set (handles.Edit1,'String',sprintf('%1f',deg_f));
set (handles.Edit2,'String',sprintf('%1f',deg_c));
set (handles.Slider1,'Value',deg_c);

function varargout = Edit2_Callback(h, eventdata, ...
    handles, varargin)

% Update all temperature values
deg_c = str2num( get(h,'String') );
deg_c = max( [ 0 deg_c] );
deg_c = min( [100 deg_c] );
deg_f = to_f(deg_c);

% Now update the fields
set (handles.Edit1,'String',sprintf('%1f',deg_f));
set (handles.Edit2,'String',sprintf('%1f',deg_c));
set (handles.Slider1,'Value',deg_c);

function varargout = Slider1_Callback(h, eventdata, ...
    handles, varargin)

% Update all temperature values
deg_c = get(h,'Value');
deg_f = to_f(deg_c);

% Now update the fields
set (handles.Edit1,'String',sprintf('%1f',deg_f));
set (handles.Edit2,'String',sprintf('%1f',deg_c));
set (handles.Slider1,'Value',deg_c);

```

Figure 10.24 Callback functions for the temperature conversion GUI.

conversion can be calculated. This conversion is done with the `str2num` function, which converts a string into a numerical value.

Also, the callback function will have to limit user entries to the valid temperature range, which is 0–100° C and 32–212° F.

The resulting callback functions are shown in Figure 10.24.

The program is now complete. Execute it and enter several different values using both the edit boxes and the sliders. Be sure to use some out-of-range values. Does it appear to be functioning properly?

10.5 Dialog Boxes

A dialog box is a special type of figure that is used to display information or to get input from a user. Dialog boxes are used to display errors, provide warnings, ask questions, or get user input. They are also used to select files or printer properties.

Dialog boxes may be **modal** or **non-modal**. A modal dialog box does not allow any other window in the application to be accessed until it is dismissed, whereas a nonmodal dialog box does not block access to other windows. Modal dialog boxes are typically used for warning and error messages that need urgent attention and cannot be ignored. By default, most dialog boxes are nonmodal.

MATLAB includes many types of dialog boxes, the most important of which are summarized in Table 10.4. We will examine only a few of the types of dialog boxes here, but you can consult the MATLAB online documentation for the details of the others.

Table 10.4 Selected Dialog Boxes

Property	Description
dialog	Creates a generic dialog box.
errordlg	Displays an error message in a dialog box. The user must click the OK button to continue.
helpdlg	Displays a help message in a dialog box. The user must click the OK button to continue.
inputdlg	Displays a request for input data and accepts the user's input values
listdlg	Allows a user to make one or more selections from a list.
printdlg	Displays a printer selection dialog box.
questdlg	Asks a question. This dialog box can contain either two or three buttons, which by default are labeled Yes, No, and Cancel.
uigetfile	Displays a file open dialog box. This box allows a user to select a file to open <i>but does not actually open the file</i> .
uiputfile	Displays a file save dialog box. This box allows a user to select a file to save, <i>but does not actually save the file</i> .
uisetcolor	Displays a color selection dialog box.
uisetfont	Displays a font selection dialog box.
warndlg	Displays a warning message in a dialog box. The user must click the OK button to continue.

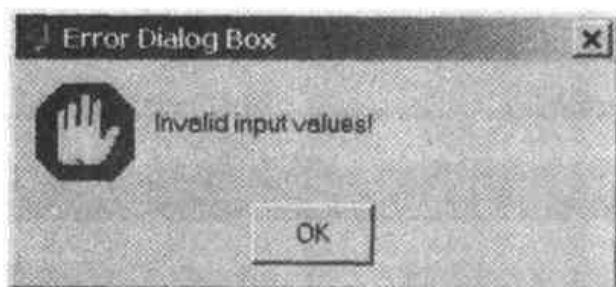


Figure 10.25 An error dialog box.

10.5.1 Error and Warning Dialog Boxes

Error and warning dialog boxes have similar calling parameters and behavior. In fact, the only difference between them is the icon displayed in the dialog box. The most common calling sequence for these dialog boxes is

```
errordlg(error_string,box_title,create_mode);
warndlg(warning_string,box_title,create_mode);
```

The `error_string` or `warning_string` is the message to display to the user, and the `box_title` is the title of the dialog box. Finally, `create_mode` is a string that can be '`modal`' or '`non-modal`', depending on the type of dialog box you want to create.

For example, the following statement creates a modal error message that cannot be ignored by the user. The dialog box produced by this statement is shown in Figure 10.25.

```
errordlg('Invalid input values!', 'Error Dialog Box', 'modal');
```

10.5.2 Input Dialog Boxes

Input dialog boxes prompt a user to enter one or more values that may be used by a program. Input dialog boxes may be created with one of the following calling sequences.

```
answer = inputdlg(prompt)
answer = inputdlg(prompt,title)
answer = inputdlg(prompt,title,line_no)
answer = inputdlg(prompt,title,line_no,default_answer)
```

Here, `prompt` is a cell array of strings, with each element of the array corresponding to one value that the user will be asked to enter. The parameter `title` specifies the title of the dialog box, and `line_no` specifies the number of lines to be allowed for each answer. Finally, `default_answer` is a cell array containing the default answers that will be used if the user fails to enter data for a particular item. Note that there must be as many default answers as there are prompts.

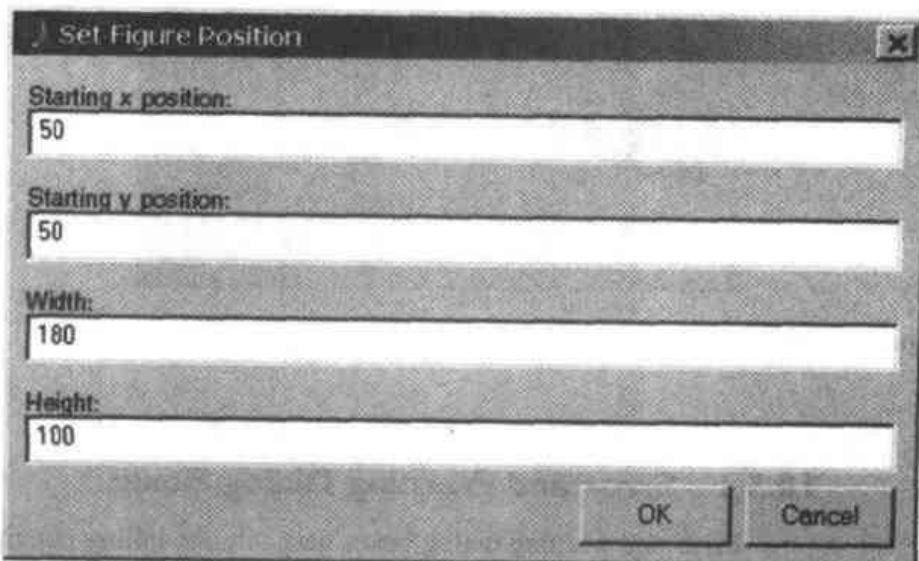


Figure 10.26 An input dialog box.

When the user clicks the **OK** button on the dialog box, his or her answers will be returned as a cell array of strings in variable **answer**.

As an example of an input dialog box, suppose that we wanted to allow a user to specify the position of a figure using an input dialog. The code to perform this function would be

```

prompt{1} = 'Starting x position:';
prompt{2} = 'Starting y position:';
prompt{3} = 'Width:';
prompt{4} = 'Height:';
title = 'Set Figure Position';
default_ans = {'50','50','180','100'};
answer = inputdlg(prompt,title,1,default_ans);

```

The resulting dialog box is shown in Figure 10.26.

10.5.3 The **uigetfile** and **uisetfile** Dialog Boxes

The **uigetfile** and **uisetfile** dialog boxes are designed to allow a user to interactively pick files to open or save. These dialog boxes return the name and the path of the file but do not actually read or save it. The programmer is responsible for writing additional code for that purpose.

The form of these two dialog boxes is

```

[filename, pathname] = uigetfile(filter_spec,title);
[filename, pathname] = uisetfile(filter_spec,title);

```

Parameter **filter_spec** is a string specifying the type of files to display in the dialog box, such as `'*.m'`, `'*.mat'`, and so on. Parameter **title** is a string

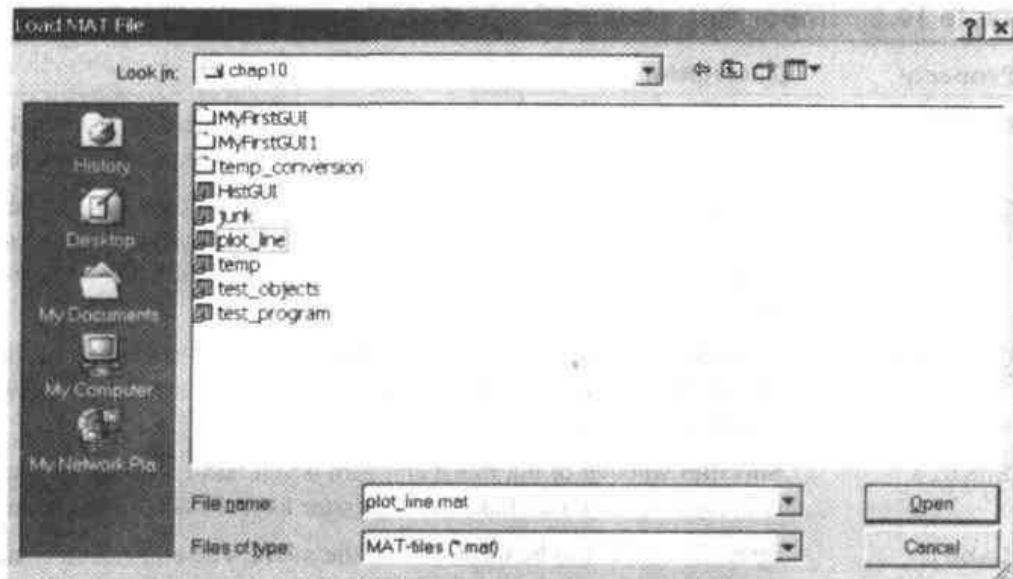


Figure 10.27 A file open dialog box created by `uigetfile` on a Windows 2000 Professional system.

specifying the title of the dialog box. After the dialog box executes, `filename` contains the name of the selected file and `pathname` contains the path of the file. If the user cancels the dialog box, `filename` is set to zero.

The following script file illustrates the use of these dialog boxes. It prompts the user to enter the name of a mat-file, then reads the contents of that file. The dialog box created by this code on a Windows 2000 system is shown in Figure 10.27. (The style of the box varies for different operating systems. It will appear slightly different on a Windows NT 4.0 or UNIX system).

```
[filename, pathname] = uigetfile('.mat', 'Load MAT File');
if filename ~= 0
    load([pathname filename])
end
```

Coding Programming Practice

Use dialog boxes to provide information or request input in GUI-based programs. If the information is urgent and should not be ignored, make the dialog boxes modal.

10.6 Menus

Menus can also be added to MATLAB GUIs. A menu allows a user to select actions without additional components appearing on the GUI display. Menus are

Table 10.5 Important uimenu Properties

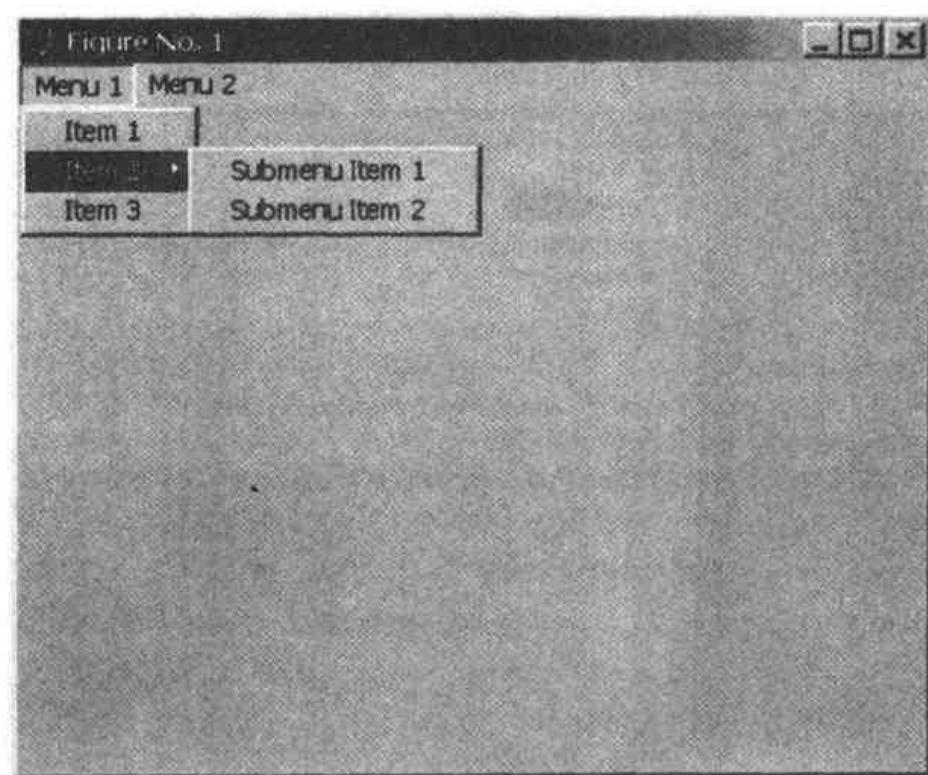
Property	Description
Accelerator	A single character specifying the keyboard equivalent for the menu item. The keyboard combination CTRL+key allows a user to activate the menu item from the keyboard.
Callback	Specifies the name and parameters of the function to be called when the menu item is activated. If the menu item has a submenu, the callback executes <i>before the submenu is displayed</i> . If the menu item does not have submenus, then the callback executes when the mouse button is <i>released</i> .
Checked	When this property is 'on', a checkmark is placed to the left of the menu item. This property can be used to indicate the status of menu items that toggle between two states. Possible values are 'on' or 'off'.
Enable	Specifies whether or not this menu item is selectable. If it is not enabled, the menu item will not respond to mouse clicks or accelerator keys. Possible values are 'on' or 'off'.
Label	Specifies the text to be displayed on the menu. The ampersand character (&) can be used to specify a keyboard mnemonic for this menu item; it will not appear on the label. For example, the string '&File' will create a menu item displaying the text 'File' and responding to the F key.
Parent	The handle of the parent object for this menu item. The parent object could be a figure or another menu item.
Position	Specifies the position of a menu item on the menu bar or within a menu. Position 1 is the left-most menu position for a top-level menu, and the highest position within a submenu.
Separator	When this property is 'on', a separating line is drawn above this menu item. Possible values are 'on' or 'off'.
Tag	The 'name' of the menu item, which can be used to locate it.
Visible	Specifies whether or not this menu item is visible. Possible values are 'on' or 'off'.

useful for selecting less commonly used options without cluttering up the GUI with a lot of extra buttons.

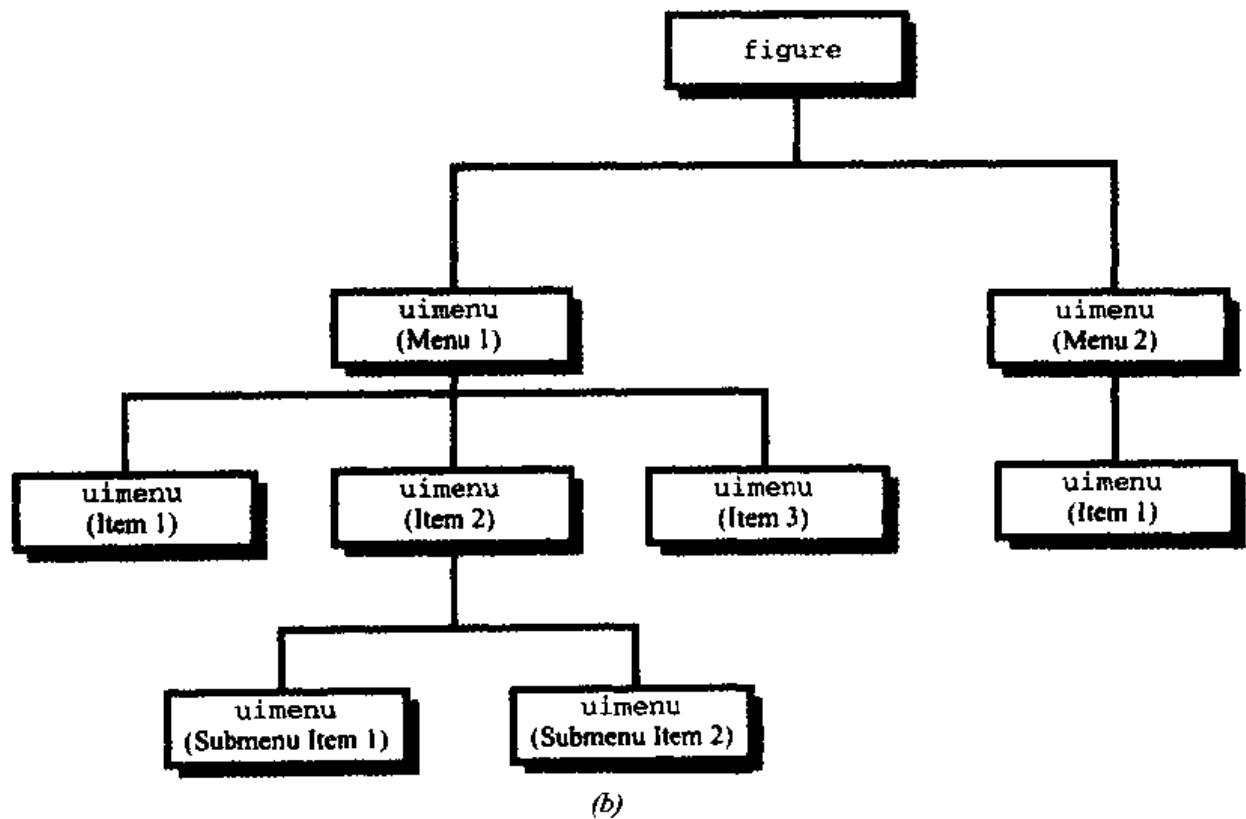
There are two types of menus in MATLAB: **standard menus**, which are pulled down from the menu bar at the top of a figure, and **context menus**, which pop up over the figure when a user right-clicks the mouse over a graphical object. We will learn how to create and use both types of menus in this section.

Standard menus are created with **uimenu** objects. Each item in a menu is a separate **uimenu** object, including items in submenus. These **uimenu** objects are similar to **uicontrol** objects, and they have many of the same properties such as **Parent**, **Callback**, **Enable**, and so forth. A list of the more important **uimenu** properties is given in Table 10.5.

Each menu item is attached to a parent object, which is a figure for the top-level menus, or another menu item for submenus. All of the **uimenus** connected to the same parent appear on the same menu, and the cascade of items forms a tree



(a)



(b)

Figure 10.28 (a) A typical menu structure. (b) The relationships among the uimenu items creating the menu.

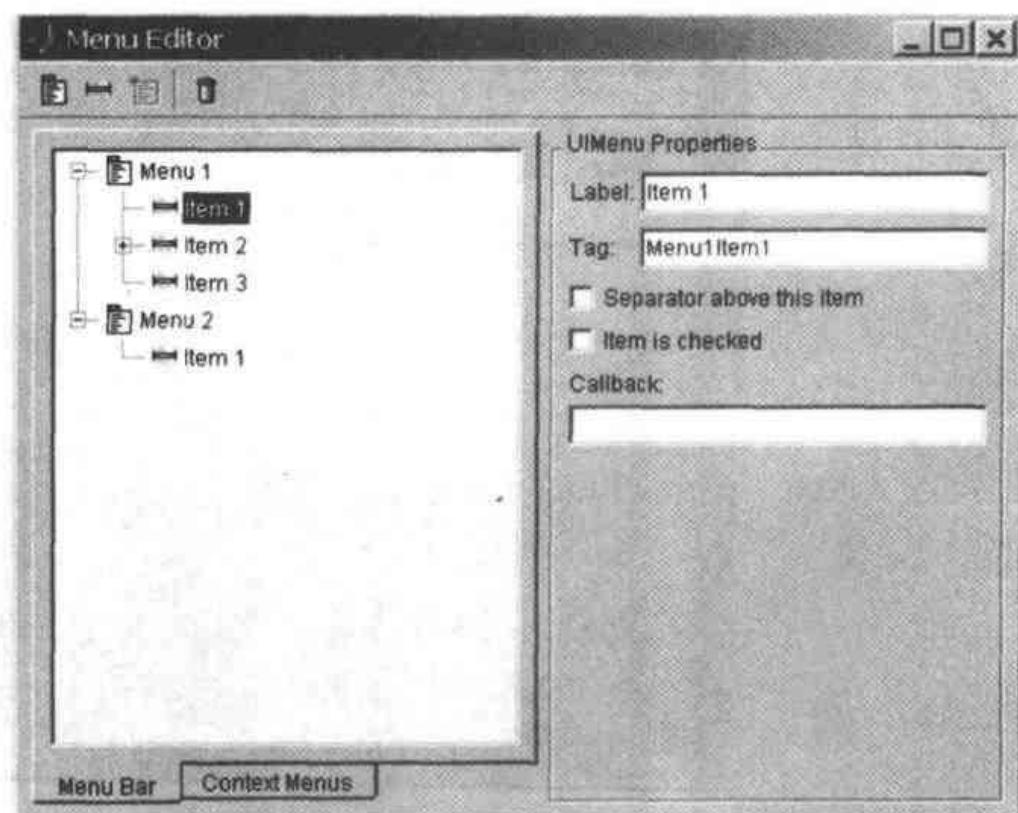


Figure 10.28 *Continued. (c) The Menu Editor structure that generated these menus.*

of submenus. Figure 10.28a shows a typical MATLAB menu in operation, while Figure 10.28b shows the relationship among the objects making up the menu.

MATLAB menus are created using the Menu Editor, which can be selected by clicking the icon on the toolbar in the guide Layout Editor. Figure 10.28c shows the Menu Editor with the menu items that generate this menu structure. The additional properties in Table 10.5 that are not shown in the Menu Editor can be set with the Property Editor (`propedit`).

Top-level context menus are created by `uicontextmenu` objects, and the lower level items within context menus are created by `uimenu` objects. Context menus are basically the same as standard menus, except that they can be associated with any GUI object (axes, lines, text, figures, etc.). A list of the more important `uicontextmenu` properties is given in Table 10.6.

10.6.1 Suppressing the Default Menu

Every MATLAB figure comes with a default set of standard menus. If you want to delete these menus from a figure and create your own menus, you must first turn the default menus off. The display of default menus is controlled by the fig-

Table 10.6 Important `uicontextmenu` Properties

Property	Description
Callback	Specifies the name and parameters of the function to be called when the context menu is activated. The callback executes before the context menu is displayed.
Parent	The handle of the parent object for this context menu.
Tag	The ‘name’ of the context menu, which can be used to locate it.
Visible	Specifies whether or not this context menu is visible. This property is set automatically and should normally not be modified.

ure’s `MenuBar` property. The possible values of this property are ‘figure’ and ‘none’. If the property is set to ‘figure’, then the default menus are displayed. If the property is set to ‘none’, then the default menus are suppressed. You can use the Property Inspector to set the `MenuBar` property for your GUIs when you create them.

10.6.2 Creating Your Own Menus

Creating your own standard menus for a GUI is basically a three-step process.

1. First, create a new menu structure with the Menu Editor. Use the Menu Editor to define the structure, giving each menu item a `Label` to display and a unique `Tag` value. Also, you must *manually* create the callback string for menu items. This can be tricky—the best way to create a menu callback is to examine the callback automatically created for a `uicontrol`, and use it as an example. The proper form of a `uimenu` callback string is

```
MyGUI('MenuItemTag_Callback',gcbo,[],guidata(gcbo))
```

where you should substitute the name of your GUI for `MyGUI`, and the tag value of the menu item for `MenuItemTag`.

2. Second, edit the properties of each menu item using the Property Editor (`propedit`), if necessary. The most important properties to set are the `Callback`, `Label`, and `Tag`, which can be set from the Menu Editor, so the Property Editor is usually not needed. However, if you need to set any of the other properties listed in Table 10.5, you will need to use the Property Editor.
3. Third, implement a callback function to perform the actions required by your menu items. You must *manually* create the callback function for menu items.

The process of building menus will be illustrated in an example at the end of this section.

Creating Plugins

Unlike GUI objects, MATLAB does not automatically create callback strings and stub functions for menu items. You must perform this function manually.

Creating Plugins

Only the Label, Tag, Callback, Checked, and Separator properties of a menu item can be set from the Menu Editor. If you need to set any of the other properties, you will have to use the Property Editor (`propedit`) on the figure, and select the appropriate menu item to edit.

10.6.3 Accelerator Keys and Keyboard Mnemonics

MATLAB menus support accelerator keys and keyboard mnemonics. **Accelerator keys** are “**CTRL+key**” combinations that cause a menu item to be executed without opening the menu first. For example, the accelerator key “o” might be assigned to the File/Open menu item. In that case, the keyboard combination **CTRL+o** will cause the File/Open callback function to be executed.

A few **CTRL+key** combinations are reserved for the use of the host operating system. These combinations differ between PC and UNIX systems; consult the MATLAB online documentation to determine which combinations are legal for your type of computer.

Accelerator keys are defined by setting the **Accelerator** property in a **uimenu** object.

Keyboard mnemonics are single letters that can be pressed to cause a menu item to execute once the menu is open. The keyboard mnemonic letter for a given menu item is underlined. For top-level menus, the keyboard mnemonic is executed by pressing **ALT** plus the mnemonic key at the same time. Once the top-level menu is open, simply pressing the mnemonic key will cause a menu item to execute.

Figure 10.29 illustrates the use of keyboard mnemonics. The **File** menu is opened with the keys **ALT+f**, and once it is opened, the **Exit** menu item can be executed by simply typing “x”.

Keyboard mnemonics are defined by placing the ampersand character (&) before the desired mnemonic letter in the **Label** property. The ampersand will not be displayed, but the following letter will be underlined, and it will act as a mnemonic key. For example, the **Label** property of the **Exit** menu item in Figure 10.29 is ‘**E&xit**’.

10.6.4 Creating Context Menus

Context menus are created in the same fashion as ordinary menus, except that the top-level menu item is a **uicontextmenu**. The parent of a **uicontextmenu**

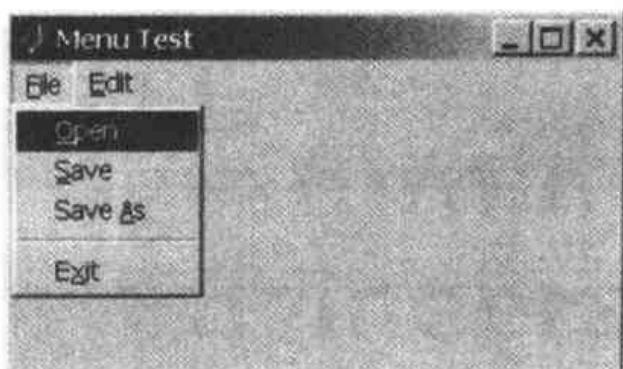


Figure 10.29 The menu shown was opened by typing the keys ALT+f, and the Exit option could be executed by simply typing “x”.

must be a figure, but the context menu can be associated with and respond to right mouse clicks on any graphical object. Context menus are created using the “Context Menu” selection on the Menu Editor. Once the context menu is created, any number of menu items can be created under it.

To associate a context menu with a specific object, you must set the object’s `UIContextMenu` property to the handle of the `uicontextmenu`. This is normally done using the Property Inspector, but it can be done with the `set` command as shown below. If `Hcm` is the handle to a context menu, the following statements will associate the context menu with a line created by a `plot` command.

```
H1 = plot(x,y);
set (H1, 'UIContextMenu', Hcm);
```

We will create a context menu and associate it with a graphical object in the following example.

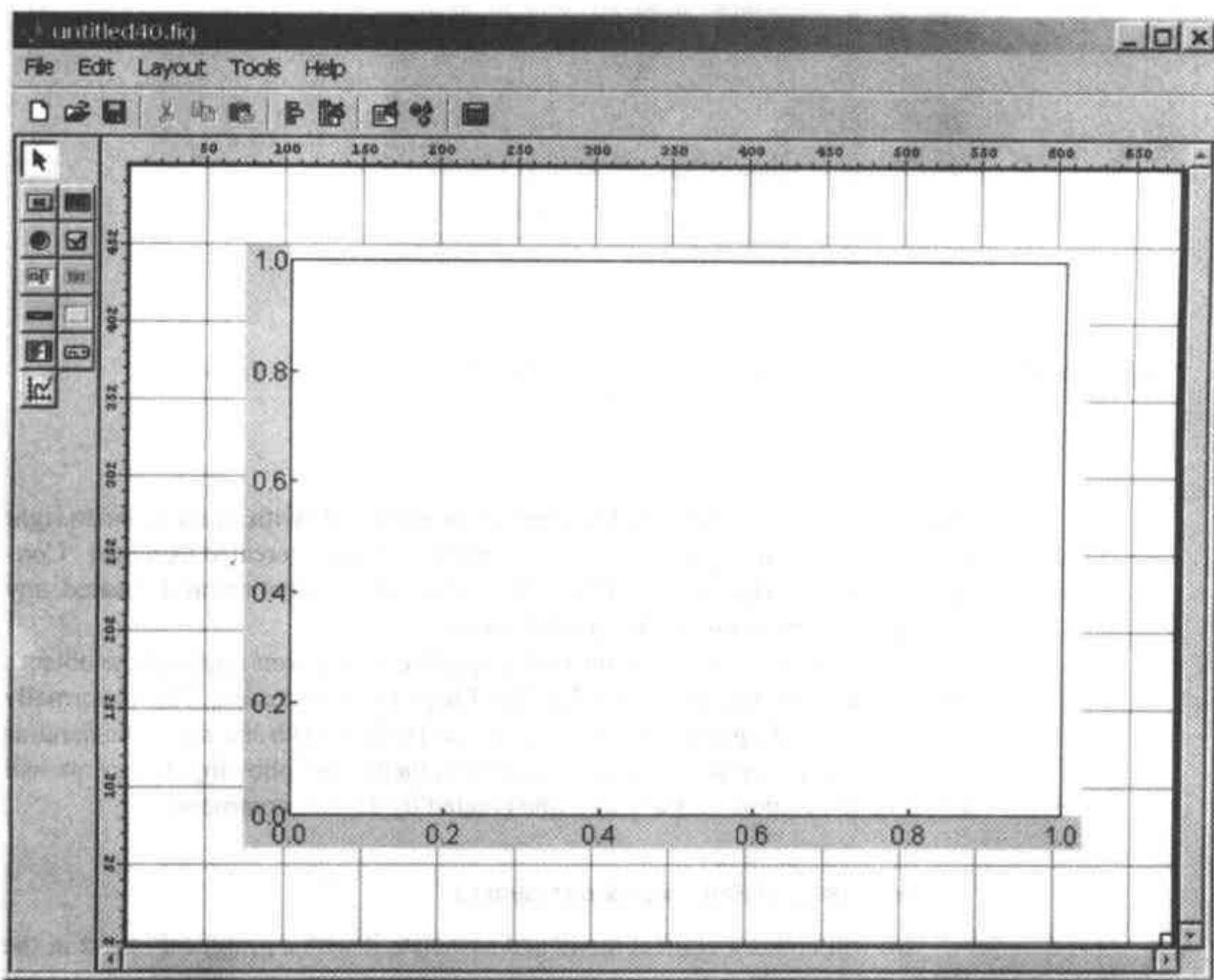
Example 10.2—Plotting Data Points

Write a program that opens a user-specified data file and plots the line specified by the points in the file. The program should include a File menu, with Open and Exit menu items. The program should also include a context menu attached to the line, with options to change the line style. Assume that the data in the file is in the form of (x,y) pairs, with one pair of data values per line.

Solution

This program should include a standard menu with Open and Exit menu items, plus a set of axes on which to plot the data. It should also include a context menu specifying various line styles, which can be attached to the line after it is plotted. The options should include solid, dashed, dotted, and dash-dot line styles.

The first step in creating this program is to use `guide` to create the required GUI, which is only a set of axes in this case (see Figure 10.30a). Then, we must use the Menu Editor to create the File menu. This menu must contain Open and



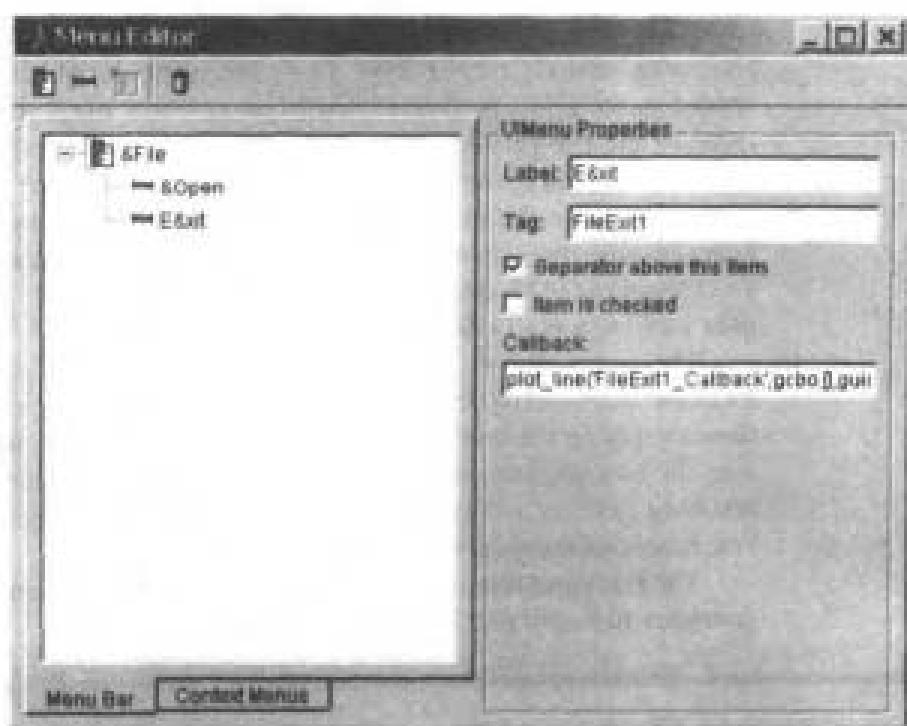
(a)

Figure 10.30 (a) The layout for `plot_line`.

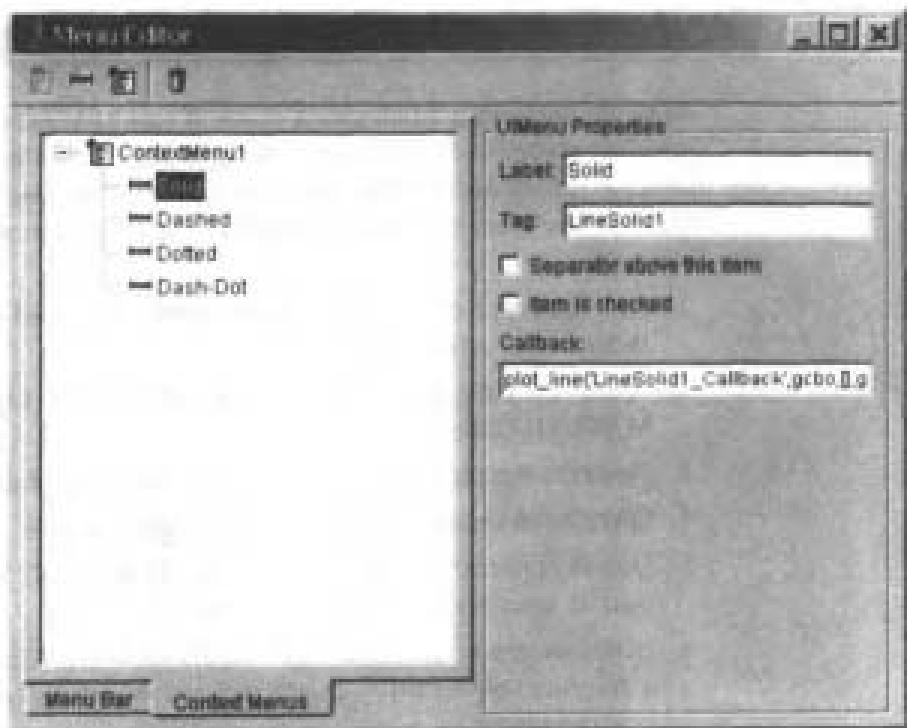
Exit menu items, as shown in Figure 10.30b. Note that we must use the Menu Editor to set the Label, Tag, and callback string for each of these menu items. We will also define keyboard mnemonics ‘F’ for File, ‘O’ for Open and ‘x’ for Exit, and place a separator between the Open and Exit menu items. Figure 10.30b shows the Exit menu item. Note that “x” is the keyboard mnemonic, and that the separator switch is turned on.

Next, we must use the Menu Editor to create the context menu. This menu starts with a `uicontextmenu` object, with four menu items attached to it (see Figure 10.30c). Again, we must set the Label, Tag, and callback string for each of these menu items.

At this point, the GUI should be saved as `plot_line.fig`, and `plot_line.m` will be automatically created. However, dummy callback functions will not be automatically created for the menu items. They must be made by hand.



(b)



(c)

Figure 10.30 *Continued.* (b) The File menu in the Menu Editor. (c) The context menu in the Menu Editor.

After the GUI is created, we must create six callback functions for the Open, Exit, and linestyle menu items. The most difficult callback function is the response to the File/Open menu item. This callback must prompt the user for the name of the file (using a `uigetfile` dialog box), open the file, read the data, save it into `x` and `y` arrays, and close the file. Then, it must plot the line and save the line's handle as application data so that we can use it to modify the line style later. Finally, the callback must associate the context menu with the line. The `FileOpen1_Callback` function is shown in Figure 10.31. Note that the function uses a dialog box to inform the user of file open errors.

The remaining callback functions are very simple. The `FileExit1_Callback` function simply closes the figure, and the line style functions simply set the line style. When the user right-clicks a mouse button over the line, the context menu will appear. If the user selects an item from the menu, the resulting callback will use the line's saved handle to change its properties. These five functions are shown in Figure 10.32.

The final program is shown in Figure 10.33. Experiment with it on your own computer to verify that it behaves properly.

QUIZ 10.1

This quiz provides a quick check to see if you have understood the concepts introduced in Sections 10.1 through 10.6. If you have trouble with the quiz, reread the section, ask your instructor, or discuss the material with a fellow student. The answers to this quiz are found in the back of the book.

1. List the types of graphical components discussed in this chapter. What is the purpose of each one?
2. What is a callback function? How are callback functions used in MATLAB GUIs?
3. Describe the steps required to create a GUI-based program.
4. Describe the purpose of the handles data structure.
5. How is application data saved in a MATLAB GUI? Why would you want to save application data in a GUI?
6. How can you make a graphical object invisible? How can you turn a graphical object off so that it will not respond to mouse clicks or keyboard input?
7. Which of the GUI components described in this chapter respond to mouse clicks? Which ones respond to keyboard inputs?
8. What are dialog boxes? How can you create a dialog box?

```

function varargout = FileOpen_Callback(h, eventdata, ...
    handles, varargin)

% Get the file to open
[filename, pathname] = uigetfile('.dat','Load Data');
if filename == 0

    % Open the input file
    filename = [pathname filename];
    [fid,msg] = fopen(filename,'rt'); ←
    % Check to see if the open failed.
    if fid < 0

        % There was an error--tell user.
        str = ['File ' filename ' could not be opened.'];
        title = 'File Open Failed';
        errordlg(str,title,'modal'); ←
    else

        % File opened successfully. Read the (x,y) pairs from
        % the input file. Get first (x,y) pair before the
        % loop starts.
        [in,count] = fscanf(fid,'%g',2); ←
        ii = 0; ←
        Read data

        while ~feof(fid)
            ii = ii + 1;
            x(ii) = in(1);
            y(ii) = in(2);

            % Get next (x,y) pair
            [in,count] = fscanf(fid,'%g',2); ←
        end

        % Data read in. Close file.
        fclose(fid);

        % Now plot the data.
        hline = plot(x,y,'LineWidth',3); ←
        xlabel('x');
        ylabel('y');
        grid on; ←
        Plot line

        % Associate the context menu with line
        set(hline,'Uicontextmenu',handles.ContextMenu1); ←
        Set context
        menu

        % Save the line's handle as application data
        handles.hline = hline; ←
        guidata(gcbf, handles); ←
        Save handle
        as app data

    end
end

```

Figure 10.31 The File/Open callback function.

```

function varargout = FileExit1_Callback(h, eventdata, ...
                                         handles, varargin)
close(gcf);

function varargout = LineSolid1_Callback(h, eventdata, ...
                                         handles, varargin)
set(handles.hline,'LineStyle','-' );

function varargout = LineDashed1_Callback(h, eventdata, ...
                                         handles, varargin)
set(handles.hline,'LineStyle','--');

function varargout = LineDotted1_Callback(h, eventdata, ...
                                         handles, varargin)
set(handles.hline,'LineStyle',':' );

function varargout = LineDashDot1_Callback(h, eventdata, ...
                                         handles, varargin)
set(handles.hline,'LineStyle','-.');

```

Figure 10.32 The remaining callback functions in `plot_line`.

9. What is the difference between a modal and a non-modal dialog box?
10. What is the difference between a standard menu and a context menu? What components are used to create these menus?
11. What are accelerator keys? What are mnemonics?

10.7 Tips for Creating Efficient GUIs

This section lists a few miscellaneous tips for creating efficient graphical user interfaces.

10.7.1 Tool Tips

Versions of MATLAB beginning with version 5.2 support **tool tips**, which are small help windows that pop up beside a `uicontrol` GUI object whenever the mouse is held over the object for a while. Tool tips are used to provide a user with quick help about the purpose of each object on a GUI.

A tool tip is defined by setting an object's `TooltipString` property to the string that you want to display. You will be asked to create tool tips in the end of chapter exercises.

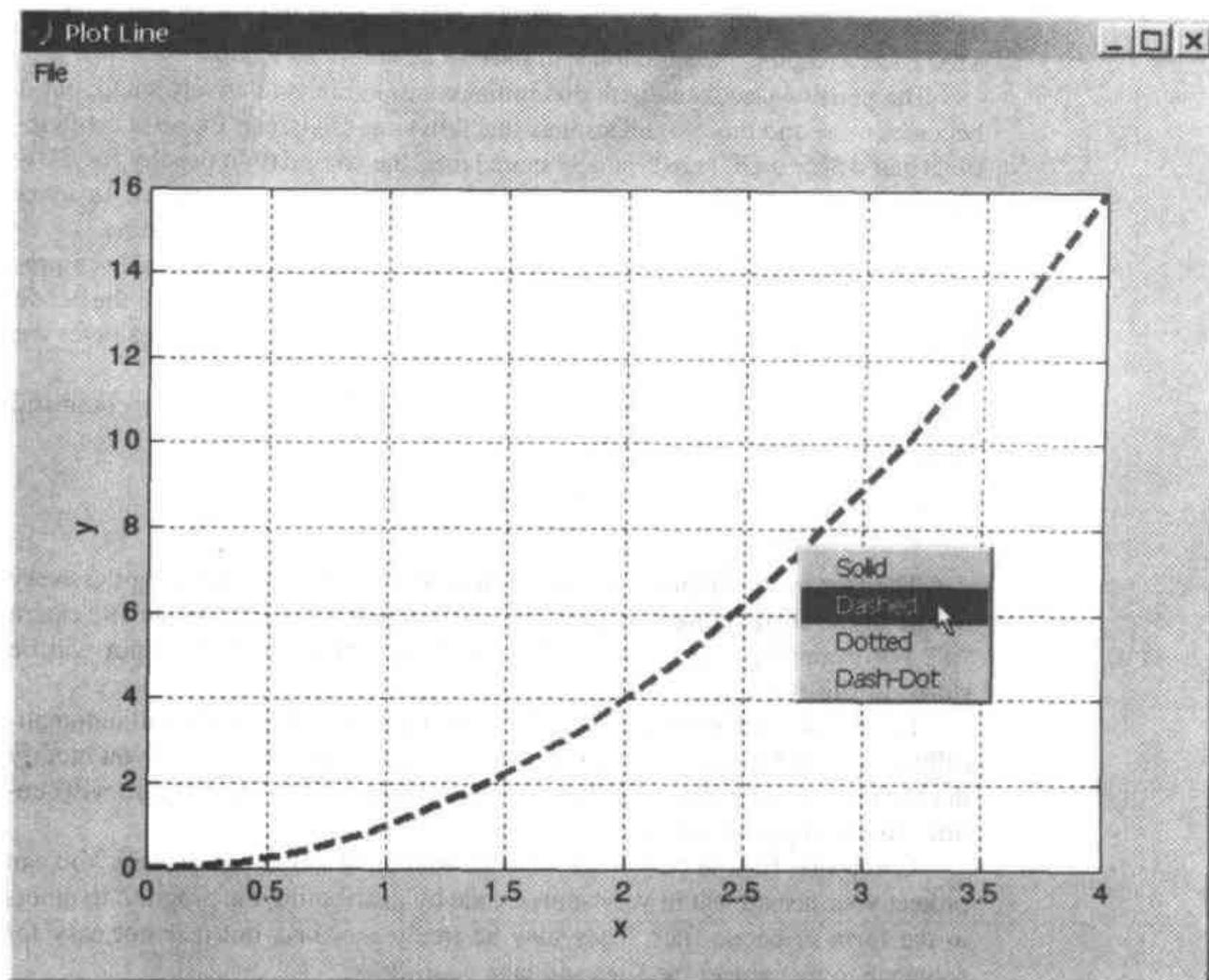


Figure 10.33 The GUI produced by program `plot_line`.

Good Programming Practice

Define tool tips to provide users with helpful hints about the functions of your GUI components.

10.7.2 Pcode

The first time that MATLAB executes a function during a program's execution, it compiles (or parses) the function into an intermediate code called **pcode** (which is short for pseudocode), and then executes the pcode in its run-time interpreter. Once a function has been compiled, it remains in MATLAB's memory, and it can

be executed repeatedly without having to be re-compiled. However, the next time MATLAB executes, the function will have to be compiled again.

The penalty associated with this initial compilation is relatively small, but it becomes more and more significant as function sizes get larger. Because the functions that define a GUI are typically quite large, the compilation penalty for GUI-based programs is relatively larger than for other types of programs. In other words, the GUIs run more slowly because of this initial compilation time.

Fortunately, there is a way to avoid this extra penalty. It is possible to pre-compile your MATLAB functions and script files into pcode, and save the pcode in files for immediate execution in the future. Executing the pcode files saves the initial compilation time, and so makes your programs faster.

MATLAB creates pcode files with the command `pcode`. This command takes one of the following forms:

```
pcode fun1.m fun2.m fun3.m ...
pcode *.m
```

The first form compiles the named files, and the second form compiles every M-file in the current directory. The compiled output is saved with the file extent “p”. For example, if you compile the file `foo.m`, the compiled output will be stored in file `foo.p`.

If a function exists as both an M-file and a p-file, MATLAB will automatically execute the p-file version because it will be faster. However, if you modify the M-file, you *must* remember to manually recompile it, or the program will continue to execute your old code!

Compiling files to pcode can have an additional advantage as well. You can protect your investment in your source code by distributing the program to others in the form of pcode files. They may be freely executed, but it is not easy for someone to reengineer the files and take your ideas.

Good Programming Practice

Once a program is working properly, you may use the `pcode` command to pre-compile its M-files and increase the program’s speed.

Programming Pitfalls

If you change an M-file that has been compiled into pcode, always remember to recompile the file. Otherwise, you will continue to execute your old, unmodified code!

10.7.3 Additional Enhancements

GUI-based programs can be much more sophisticated than we have described in this introductory chapter. In addition to the `Callback` property that we have been using in the chapter, `uicontrols` support three other types of callbacks: `CreateFcn`, `DeleteFcn`, and `ButtonDownFcn`. MATLAB figures also support three important types of callbacks: `WindowButtonDownFcn`, `WindowButtonMotionFcn`, and `WindowButtonUpFcn`.

The `CreateFcn` property defines a callback that is automatically called whenever an object is created. This property allows a programmer to customize his or her objects as they are created during program execution. Because this callback is executed before the object is completely defined, a programmer must specify the function to execute as a default property of the root before the object is created. For example, the following statement will cause the function `function_name` to be executed each time that a `uicontrol` is created. The function will be called after MATLAB creates the object's properties, so they will be available to the function when it executes.

```
Sset(0, 'DefaultUicontrolCreateFcn', 'function_name')
```

The `DeleteFcn` property defines a callback that is automatically called whenever an object is destroyed. It is executed before the object's properties are destroyed, so they will be available to the function when it executes. This callback provides the programmer with an opportunity to do custom clean-up work.

The `ButtonDownFcn` property defines a callback that is automatically called whenever a mouse button is pressed within a five-pixel border around a `uicontrol`. If the mouse button is pressed on the `uicontrol`, the `Callback` is executed. Otherwise, if it is near the border, the `ButtonDownFcn` is executed. If the `uicontrol` is not enabled, the `ButtonDownFcn` is executed even for clicks on the control.

The figure-level callback functions `WindowButtonDownFcn`, `WindowButtonMotionFcn`, and `WindowButtonUpFcn` allow a programmer to implement features such as animations and drag-and-drop because the callbacks can detect the initial, intermediate, and final locations at which the mouse button is pressed. These functions are beyond the scope of this book, but are well worth learning about. Refer to the *Creating Graphical User Interfaces* volume in the MATLAB user documentation for a description of these callbacks.

Example 10.3—Creating a Histogram GUI

Write a program that opens a user-specified data file and calculates a histogram of the data in the file. The program should calculate the mean, median, and standard deviation of the data in the file. It should include a File menu, with Open and Exit menu items. It should also include a means to allow the user to change the number of bins in the histogram.

Select a color other than the default color for the figure and the text label backgrounds, use keyboard accelerators for menu items, and add tool tips where appropriate.

Solution

This program should include a standard menu with Open and Exit menu items, a set of axes on which to plot the histogram, and a set of six text fields for the mean, median, and standard deviation of the data. Three of these text fields will hold labels, and three will hold the read-only mean, median and standard deviation values. The program must also include a label and an edit field to allow the user to select the number of bins to display in the histogram.

We will select a light blue color [0.6 1.0 1.0] for the background of this GUI. To make the GUI have a light blue background, this color vector must be loaded into the 'Color' property of the figure and into the 'BackgroundColor' property of each text label with the Property Inspector during the GUI layout.

The first step in creating this program is to use guide to lay out the required GUI (see Figure 10.34a). Then, use the Property Inspector to set the properties of the seven text fields and the edit field. The fields must be given unique tags so that we can locate them from the callback functions. Next, use the Menu Editor to create the File menu (see Figure 10.34b). Finally, the resulting GUI should be saved as histGUI, creating histGUI.fig and histGUI.m.

After histGUI.m is saved, it must be edited to add the number of bins in the histogram to the application data in the handles structure, so that it will be available for use by the callbacks. The modified code to save nbins is

```

fig = openfig(mfilename, 'reuse');

% Generate a structure of handles to pass to callbacks.
handles = guihandles(fig);

% Add the number of bins to this structure.
handles.nbins = str2num(get(handles.NBins, 'String'));

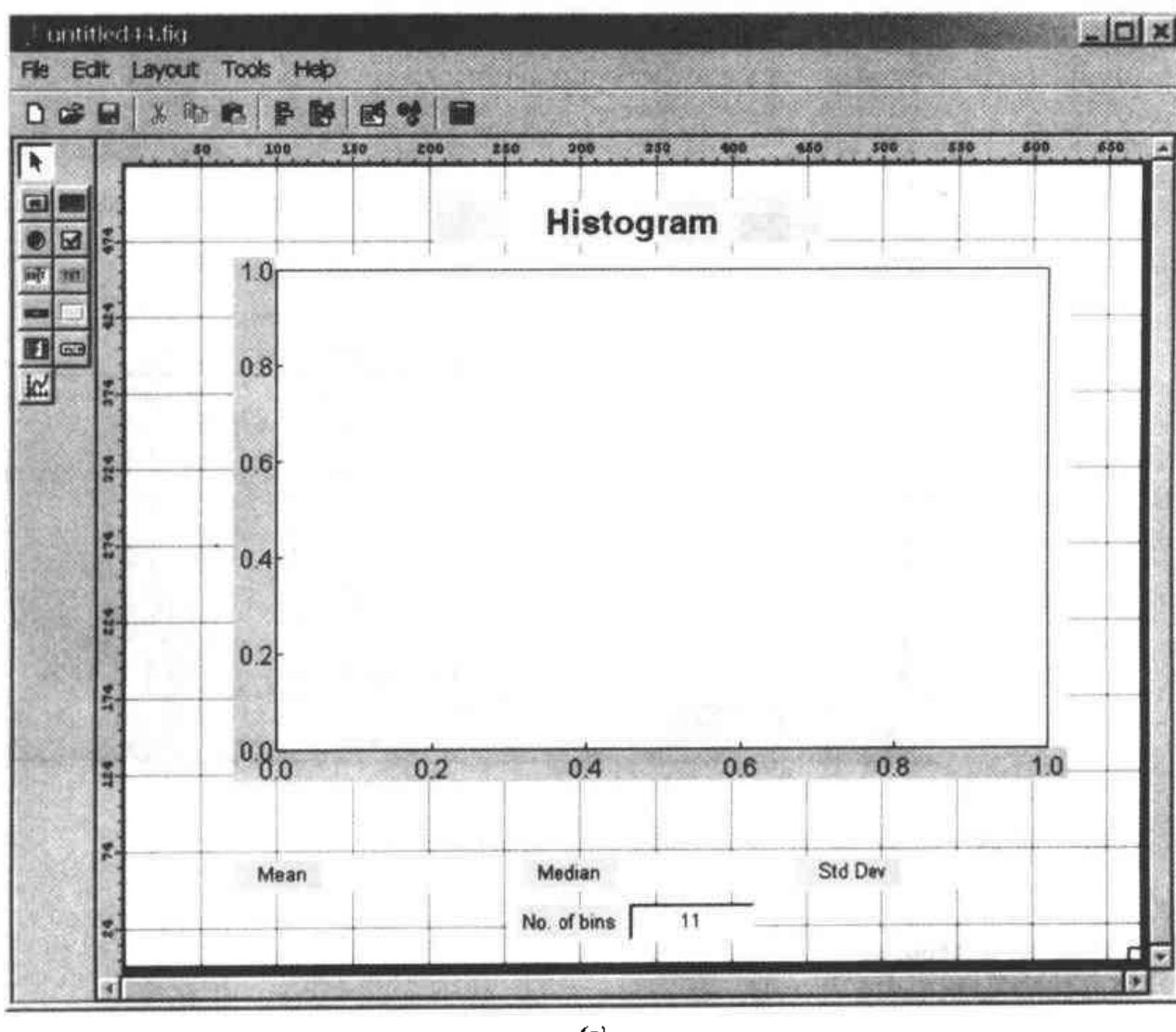
% Save handles structure
guidata(fig, handles);

if nargout > 0
    varargout{1} = fig;
end

```

Next, we must create callback functions for the File/Open menu item, the File/Exit menu item, and the “number of bins” edit box. Only the last function will be created automatically—the menu item callbacks must be added by hand.

The File/Open callback must prompt the user for a file name, then read the data from the file. It must calculate and display the histogram and update the statistics text fields. Note that the data in the file must also be saved in the handles structure, so that it will be available for re-calculation if the user changes the number of bins in the histogram. The callback function to perform these steps is shown below:



(a)

Figure 10.34 (a) The layout for histGUI.

```

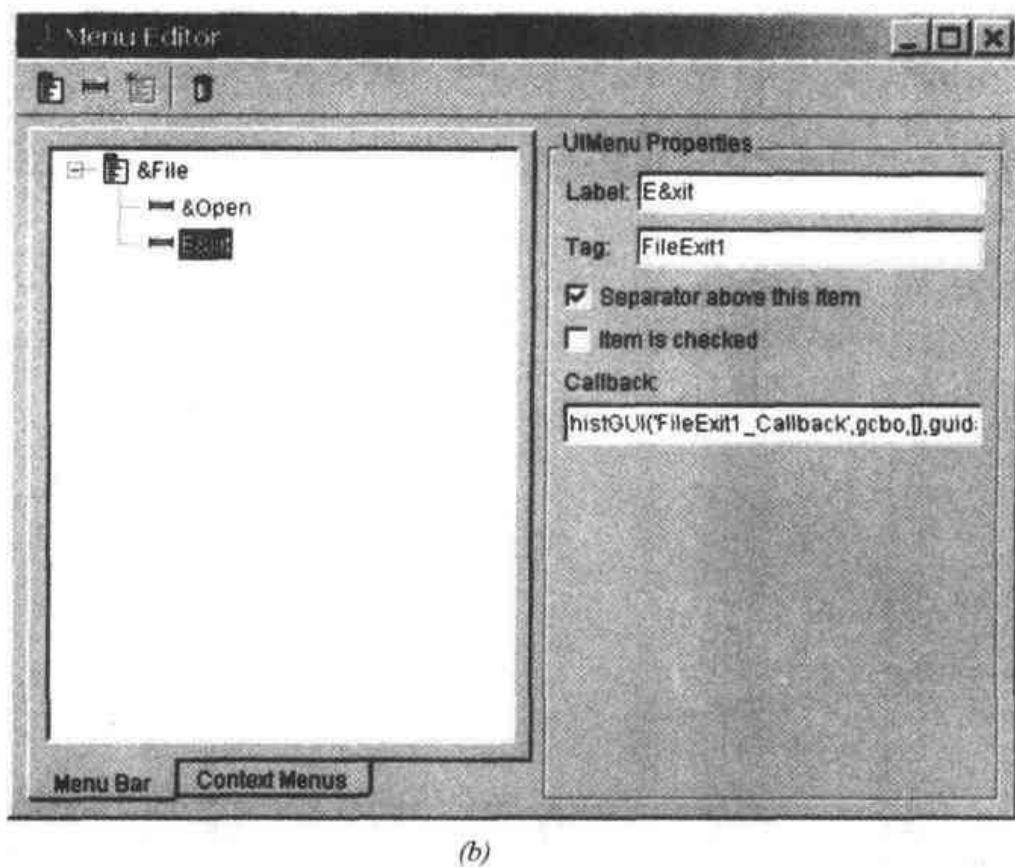
function varargout = FileOpen1_Callback(h, eventdata, ...
    handles, varargin)

% Get file name
[filename,path] = uigetfile('.dat','Load Data File');
if filename ~= 0

    % Read data
    x = textread([path filename], '%f');

    % Save in handles structure
    handles.x = x;
    guidata(gcbf, handles);

```



(b)

Figure 10.34 *Continued.* (b) The File menu in the Menu Editor.

```
% Create histogram
hist(handles.x,handles.nbins);

% Set axis labels
xlabel('\bfValue');
ylabel('\bfCount');

% Calculate statistics
ave = mean(x);
med = median(x);
sd = std(x);
n = length(x);

% Update fields
set (handles.MeanData,'String',sprintf('%7.2f',ave));
set (handles.MedianData,'String',sprintf('%7.2f',med));
set (handles.StdDevData,'String',sprintf('%7.2f',sd));
set (handles.TitleString,'String',[['Histogram (N = ' int2str(n) ')']]);

end
```

The File/Exit callback is trivial. All it has to do is close the figure.

```
function varargout = FileExit1_Callback(h, eventdata, ...
    handles, varargin)
close(gcbf);
```

The NBins callback must read a numeric input value, round it off to the nearest integer, display that integer in the Edit Box, and re-calculate and display the histogram. Note that the number of bins must also be saved in the handles structure, so that it will be available for re-calculation if the user loads a new data file. The callback function to perform these steps is shown below:

```
function varargout = NBins_Callback(h, eventdata, ...
    handles, varargin)

% Get number of bins, round to integer, and update field
nbins = str2num(get(gcbo,'String'));
nbins = round(nbins);
if nbins < 1
    nbins = 1;
end
set(handles.NBins,'String',int2str(nbins));

% Save in handles structure
handles.nbins = nbins;
guidata(gcf, handles);

% Re-display data, if available
if handles.nbins > 0 & ~isempty(handles.x)

    % Create histogram
    hist(handles.x,handles.nbins);

end
```

The final program is shown in Figure 10.35. Experiment with it on your own computer to verify that it behaves properly.

10.8 Summary

In this chapter, we learned how to create MATLAB graphical user interfaces. The three fundamental parts of a GUI are components (uicontrols, uimenus, and uicontextmenus), figures to contain them, and callbacks to implement actions in response to mouse clicks or keyboard inputs.

The standard GUI components created by uicontrol include text fields, edit boxes, frames, pushbuttons, toggle buttons, check boxes, radio buttons,

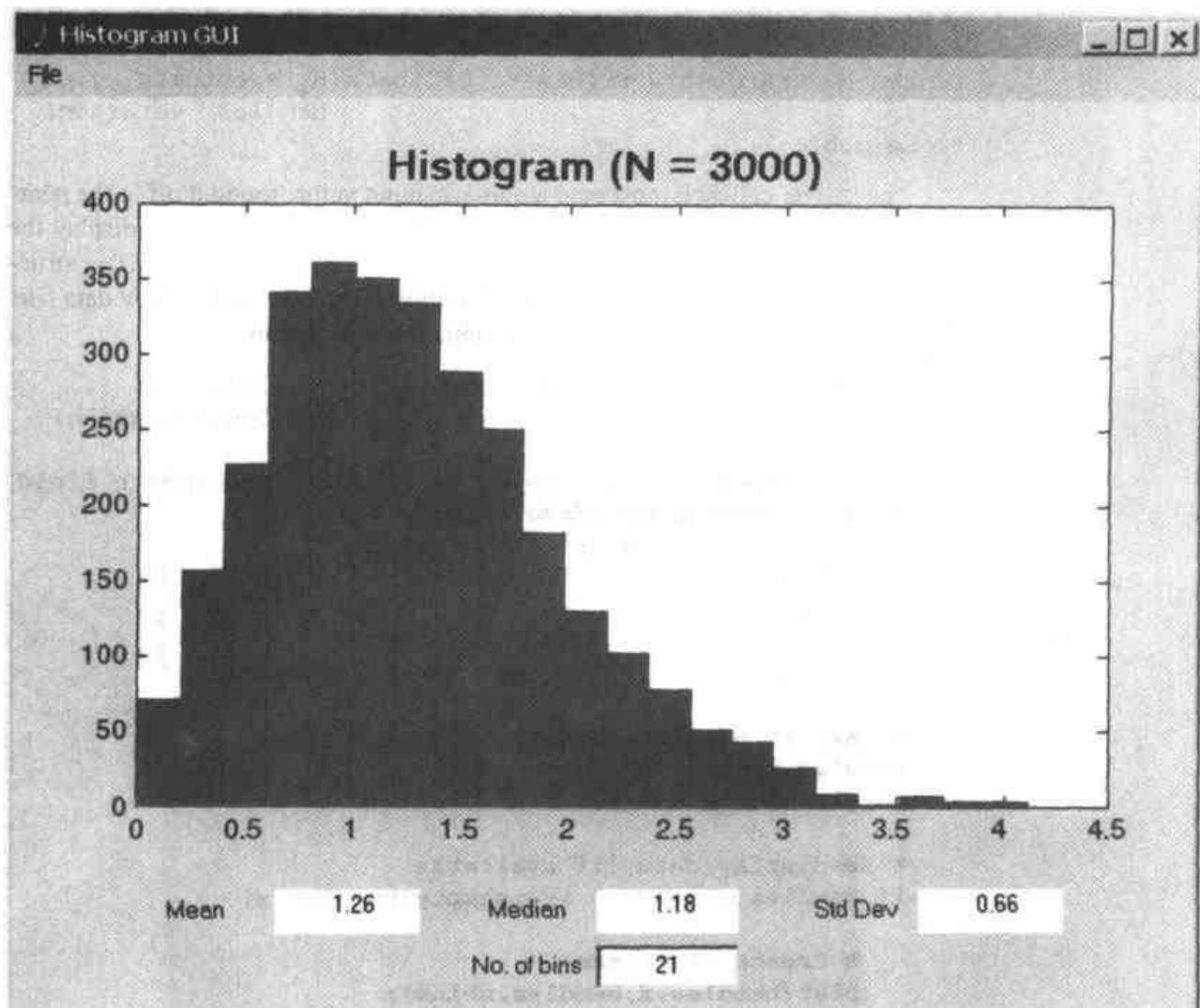


Figure 10.35 The GUI produced by program HistGUI. (Also shown in color insert.)

popup menus, list boxes, and sliders. The standard GUI components created by `uimenu` and `uicontextmenu` are standard menus and context menus.

Any of these components can be placed on a figure using `guide` (the GUI Development Environment tool). Once the GUI layout has been completed, the user must edit the object properties with the Property Inspector, then write a callback function to implement the actions associated with each GUI object.

Dialog boxes are special figures used to display information or to get input from a user. Dialog boxes are used to display errors, provide warnings, ask questions, or get user input. They are also used to select files or printer properties.

Dialog boxes may be modal or non-modal. A modal dialog box does not allow any other window in the application to be accessed until it is dismissed,

whereas a non-modal dialog box does not block access to other windows. Modal dialog boxes are typically used for warning and error messages that need urgent attention and cannot be ignored.

Menus can also be added to MATLAB GUIs. A menu allows a user to select actions without additional components appearing on the GUI display. Menus are useful for selecting less commonly used options without cluttering up the GUI with a lot of extra buttons. Menus are created with the Menu Editor. For each menu item, the user must use the Menu Editor to set the Label, Tag, and the callback string. Unlike GUI components, guide does not generate automatic callback strings for menu items. The user is completely responsible for defining the callback string and creating the corresponding callback function.

Accelerator keys and keyboard mnemonics can be used to speed the operation of windows.

10.8.1 Summary of Good Programming Practice

The following guidelines should be adhered to when working with MATLAB GUIs.

1. Use `guide` to lay out a new GUI, and use the Property Inspector to set the initial properties of each component such as the text displayed on the component, the color of the component, and the name of the callback function, if required.
2. After creating a GUI with `guide`, manually edit the resulting function to add comments describing its purpose and components, and to implement the function of callbacks.
3. Store GUI application data in the `handles` structure, so that it will automatically be available to any callback function.
4. If you modify any of the GUI application data in the `handles` structure, be sure to save the structure with a call to `guidata` before exiting the function where the modifications occurred.
5. Use dialog boxes to provide information or request input in GUI-based programs. If the information is urgent and should not be ignored, make the dialog boxes modal.
6. Define tool tips to provide users with helpful hints about the functions of your GUI components.
7. Once a program is working properly, you may use the `pcode` command to pre-compile its M-files and increase the program's speed.

10.8.2 MATLAB Summary

The following summary lists all of the MATLAB commands and functions described in this chapter, along with a brief description of each one. Also, refer to the summaries of Graphical object properties in Tables 10.2, 10.3, 10.5, and 10.6.

Commands and Functions

<code>axes</code>	Function to create a set of axes.
<code>dialog</code>	Create a generic dialog box.
<code>errordlg</code>	Display an error message.
<code>helpdlg</code>	Display a help message.
<code>findobj</code>	Find a GUI object by matching one or more of its properties.
<code>gcbf</code>	Get callback figure.
<code>gcbo</code>	Get callback object.
<code>guidata</code>	Save GUI application data in a figure.
<code>guihandles</code>	Get the handles structure from the application data stored in a figure.
<code>guide</code>	GUI Development Environment tool.
<code>inputdlg</code>	Dialog to get input data from the user.
<code>printdlg</code>	Print dialog box.
<code>questdlg</code>	Dialog box to ask a question.
<code>uicontrol</code>	Function to create a GUI object.
<code>uicontextmenu</code>	Function to create a context menu.
<code>uigetfile</code>	Dialog box to select an input file.
<code>uimenu</code>	Function to create a standard menu, or a menu item on either a standard menu or a context menu.
<code>uiputfile</code>	Dialog box to select an output file.
<code>uisetcolor</code>	Displays a color selection dialog box.
<code>uisetfont</code>	Displays a font selection dialog box.
<code>warndlg</code>	Displays a warning message.

10.9 Exercises

- 10.1** Explain the steps required to create a GUI in MATLAB.
- 10.2** How does a callback function work? How can a callback function locate the figures and objects that it needs to manipulate?
- 10.3** Create a GUI that uses a popup menu to select the background color displayed by the GUI.
- 10.4** Create a GUI that uses a standard menu to select the background color displayed by the GUI.
- 10.5** Write a GUI program that plots the equation $y(x) = ax^2 + bx + c$. The program should have a set of axes for the plot and should have GUI elements to input the values of a , b , c , and the minimum and maximum x to plot. Include tool tips for each of your GUI elements.
- 10.6** Modify the GUI of Exercise 10.5 to include a menu. The menu should include two submenus to select the color and line style of the plotted line, with a check mark beside the currently selected menu choices. The menu should also include an “Exit” option. If the user selects this op-

- tion, the program should create a modal question dialog box asking “Are You Sure?”, with the appropriate responses.
- 10.7** Modify the List Box example in Section 10.4.8 to allow for multiple selections in the list box. The text field should display a list of all selections whenever the “Select” button is clicked.
- 10.8** **Random Number Distributions.** Create a GUI to display the distributions of different types of random numbers. The program should create the distributions by generating an array of 200,000 random values from a distribution and using function `hist` to create a histogram. Be sure to label the title and axes of the histogram properly.
The program should support uniform, Gaussian, and Rayleigh distributions, with the distribution selection made by a popup menu. In addition, it should have an edit box to allow the user to select the number of bins in the histogram. Make sure that the values entered in the edit box are legal (the number of bins must be a positive integer).
- 10.9** Modify the temperature conversion GUI of Example 10.1 to add a “thermometer.” The thermometer should be a set of rectangular axes with a red “fluid” level corresponding to the current temperature in degrees Celsius. The range of the thermometer should be 0°–100° C.
- 10.10** Modify the temperature conversion GUI of Exercise 10.9 to allow you to adjust the displayed temperature by clicking the mouse. (Warning: This exercise requires material not discussed in this chapter. Refer to the `CurrentPoint` property of figure objects in the on-line MATLAB documentation.)
- 10.11** **Least Squares Fit.** Create a GUI that can read an input data set from a file and perform a least-squares fit to the data. The data will be stored in a disk file in (x,y) format, with one x and one y value per line. Perform the least-squares fit with the MATLAB function `polyfit`, and plot both the original data and the least-squares fitted line. Include two menus: File and Edit. The File menu should include File/Open and File/Exit menu items, and the user should receive an “Are You Sure?” prompt before exiting. The Edit menu item should allow the user to customize the display, including line style, line color, and grid status.
- 10.12** Modify the GUI of the previous exercise to include an Edit/Preferences menu item that allows the user to suppress the “Are You Sure?” exit prompt.
- 10.13** Modify the GUI of the previous exercise to read and write an initialization file. The file should contain the line style, line color, grid choice (on/off), and exit prompt choice made by the user on previous runs. These choices should be automatically written out and saved when the program exits via the File/Exit menu item, and they should be read in and used whenever the program is started again.

ASCII Character Set

MATLAB strings use the ASCII character set, which consists of the 127 characters shown in the table below. The results of MATLAB string comparison operations depend on the *relative lexicographic positions* of the characters being compared. For example, the character 'a' in the ASCII character set is at position 97 in the table, while the character 'A' is at position 65. Therefore, the relational operator 'a' > 'A' will return a 1 (true), since 97 > 65.

Each MATLAB character is stored in a 16-bit field, which means that in the future MATLAB could support the entire Unicode character set. However, Unicode character support is not yet a part of MATLAB Version 6.

The table shown below contains the ASCII character set, with the first two digits of the character number defined by the row and the third digit defined by the column. Thus, the letter 'R' is on row 8 and column 2, so it is character 82 in the ASCII character set.

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	de4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	,	}	~	del		

**Answers
to
Quizzes**

This appendix contains the answers to all of the quizzes in the book.

Quiz 1.1, page 18

1. The MATLAB Command Window is the first window seen when MATLAB starts up. A user can enter interactive commands at the command prompt ($>$) in the Command Window, and they will be executed on the spot. The Command Window is also used to start M-files executing. The Edit/Debug Window is an editor used to create, modify, and debug M-files. The Figure Window is used to display MATLAB graphical output.
2. You can get help in MATLAB by:
 - Typing `help <command_name>` in the Command Window. This command will display information about a command or function in the Command Window.
 - Typing `lookfor <keyword>` in the Command Window. This command will display in the Command Window a list of all commands or functions containing the keyword in their first comment line.
 - Starting the Help Browser by typing `helpwin` or `helpdesk` in the Command Window, or by selecting “Help” from the Launch Pad. The Help Browser contains an extensive hypertext-based description of all of the features in MATLAB, plus a complete copy of all manuals on-line in HTML and Adobe PDF formats. It is the most comprehensive source of help in MATLAB.

3. A workspace is the collection of all the variables and arrays that can be used by MATLAB when a particular command, M-file, or function is executing. All commands executed in the Command Window (and all script files executed from the Command Window) share a common workspace, so they can all share variables. The contents of the workspace can be examined with the `whos` command, or graphically with the Workspace Browser.

4. To clear the contents of a workspace, type `clear` or `clear variables` in the Command Window.

5. The commands to perform this calculation are:

```
>> t = 5;
>> x0 = 10;
>> v0 = 15;
>> a = -9.81;
>> x = x0 + v0 * t + 1/2 * a * t^2
x =
-37.6250
```

6. The commands to perform this calculation are:

```
>> x = 3;
>> y = 4;
>> res = x^2 * y^3 / (x - y)^2
res =
576
```

Questions 7 and 8 are intended to get you to explore the features of MATLAB. There is no single “right” answer for them.

Quiz 2.1, page 30

1. An array is a collection of data values organized into rows and columns, and known by a single name. Individual data values within an array are accessed by including the name of the array followed by subscripts in parentheses that identify the row and column of the particular value. The term “vector” is usually used to describe an array with only one dimension, while the term “matrix” is usually used to describe an array with two or more dimensions.

2. (a) This is a 3×4 array; (b) $c(2,3) = -0.6$; (c) The array elements whose value is 0.6 are $c(1,4)$, $c(2,1)$, and $c(3,2)$.

3. (a) 1×3 ; (b) 3×1 ; (c) 3×3 ; (d) 3×2 ; (e) 3×3 ; (f) 4×3 ; (g) 4×1 .

4. $w(2,1) = 2$

5. $x(2,1) = -20i$

6. $y(2,1) = 0$

7. $v(3) = 3$

Quiz 2.2, page 38

1. (a) $c(2, :) = [0.6 \quad 1.1 \quad -0.6 \quad 3.1]$

(b) $c(:, 4) = \begin{bmatrix} 0.6 \\ 3.1 \\ 0.0 \end{bmatrix}$

(c) $c(1:2, 2:4) = \begin{bmatrix} -3.2 & 3.4 & 0.6 \\ 1.1 & -0.6 & 3.1 \end{bmatrix}$

(d) $c(6) = 0.6$

(e) $c(4, \text{end}) = [-3.2 \quad 1.1 \quad 0.6 \quad 3.4 \quad -0.6 \quad 5.5 \quad 0.6 \quad 3.1 \quad 0.0]$

(f) $c(1:2, 2:\text{end}) = \begin{bmatrix} -3.2 & 3.4 & 0.6 \\ 1.1 & -0.6 & 3.1 \end{bmatrix}$

(g) $c([1 \ 3], 2) = \begin{bmatrix} -3.2 \\ 0.6 \end{bmatrix}$

(h) $c([2 \ 2], [3 \ 3]) = \begin{bmatrix} -0.6 & -0.6 \\ -0.6 & -0.6 \end{bmatrix}$

2.

(a) $a = \begin{bmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix}$

(b) $a = \begin{bmatrix} 4 & 5 & 6 \\ 4 & 5 & 6 \\ 4 & 5 & 6 \end{bmatrix}$

(c) $a = \begin{bmatrix} 4 & 5 & 6 \\ 4 & 5 & 6 \end{bmatrix}$

3.

(a) $a = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 3 \\ 0 & 0 & 1 \end{bmatrix}$

(b) $a = \begin{bmatrix} 1 & 0 & 4 \\ 0 & 1 & 5 \\ 0 & 0 & 6 \end{bmatrix}$

(c) $a = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 9 & 7 & 8 \end{bmatrix}$

Quiz 2.3, page 44

1. The required command is “format long e”.

2. (a) These statements get the radius of a circle from the user, and calculate and display the area of the circle.

(b) These statements display the value of π as an integer, so they display the string: “The value is 3!”.

3. The first statement outputs the value 12345.67 in exponential format; the second statement outputs the value in floating point format; the third statement outputs the value in general format; and the fourth statement outputs the value in floating point format in a field 12 characters wide, with four places after the decimal point. The results of these statements are:

```
value = 1.234567e+004
value = 12345.670000
value = 12345.7
value = 12345.6700
```

Quiz 2.4, page 50

1. (a) This operation is illegal. Array multiplication must be between arrays of the same shape, or between an array and a scalar.
 - (b) Legal matrix multiplication: result = $\begin{bmatrix} 4 & 4 \\ 3 & 3 \end{bmatrix}$
 - (c) Legal array multiplication: result = $\begin{bmatrix} 2 & 1 \\ -2 & 4 \end{bmatrix}$
 - (d) This operation is illegal. The matrix multiplication $b * c$ yields a 1×2 array, and a is a 2×2 array, so the addition is illegal.
 - (e) This operation is illegal. The array multiplication $b . * c$ is between two arrays of different sizes, so the multiplication is illegal.
2. This result can be found from the operation $x = A/B$: $x = \begin{bmatrix} -0.5 \\ 1.0 \\ -0.5 \end{bmatrix}$

Quiz 3.1, page 92

- | | |
|---------------------|--|
| 1. $a > b$ | 1 |
| 2. $b > d$ | 0 |
| 3. $a > b \& c > d$ | 0 |
| 4. $a == b$ | 0 |
| 5. $a \& b > c$ | 0 |
| 6. $\sim\sim b$ | 1 |
| 7. $a \& b > c$ | $\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$ |
| 8. $a > c \& b > c$ | $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ |

9. This is illegal, because the two arrays have different sizes.

- | | |
|---------------------------------|---|
| 10. $a^*b^2 > a*c$ | 0 |
| 11. $d \mid b > a$ | 1 |
| 12. $(d \mid b) > a$ | 0 |
| 13. $\text{isinf}(a/b)$ | 0 |
| 14. $\text{isinf}(a/c)$ | 1 |
| 15. $a > b \& \text{ischar}(d)$ | 1 |
| 16. $\text{isempty}(c)$ | 0 |

Quiz 3.2, page 107

1. if $x \geq 0$
 $\quad \quad \quad \text{sqrt_x} = \text{sqrt}(x);$
 else
 $\quad \quad \quad \text{disp('ERROR: } x < 0\text{')};$
 $\quad \quad \quad \text{sqrt_x} = 0;$
 end
2. if $\text{abs}(\text{denominator}) < 1.0E-300$
 $\quad \quad \quad \text{disp('Divide by 0 error.')};$
 else
 $\quad \quad \quad \text{fun} = \text{numerator} / \text{denominator};$
 $\quad \quad \quad \text{disp(fun)};$
 end
3. if $\text{distance} \leq 100$
 $\quad \quad \quad \text{cost} = 0.50 * \text{distance};$
 elseif $\text{distance} \leq 300$
 $\quad \quad \quad \text{cost} = 50 + 0.30 * (\text{distance} - 100);$
 else
 $\quad \quad \quad \text{cost} = 110 + 0.20 * (\text{distance} - 300);$
 end
4. These statements are incorrect. For this structure to work, the second if statement would need to be an elseif statement.
5. These statements are legal. They will display the message "Prepare to stop."
6. These statements will execute, but they will not do what the programmer intended. If the temperature is 150, these statements will print out "Human body temperature exceeded." instead of "Boiling point of water exceeded.", because the if structure executes the first true condition and skips the rest. To get proper behavior, the order of these tests should be reversed.

Quiz 3.3, page 124

1.

```
x = 0:pi/10:2*pi;
x1 = cos(2*x);
y1 = sin(x);
plot(x1,y1,'-
ro','LineWidth',2.0,'MarkerSize',6,...,
'MarkerEdgeColor','b','MarkerFaceColor','b')
```
3. $\text{itf}(\text{rm}(\text{itx}\text{rm}) = \sin \theta \cos 2\phi)$
4. 'bfPlot of $\Sigma \text{itx}\text{rm}\text{bf}^2$ versus itx '
5. This string creates the characters: τ_m
6. This string creates the characters: $x_1^2 + x_2^2$ (units : m²)
7. The backslash character is displayed using a double backslash ('\\').

Quiz 4.1, page 162

1. 4 times
2. 0 times
3. 1 time
4. 2 times
5. 2 times
6. ires = 10
7. ires = 55
8. ires = 25;
9. ires = 49;
10. With loops and branches:

```
for ii = -6*pi:pi/10:6*pi
    if sin(ii) > 0
        res(ii) = sin(ii);
    else
        res(ii) = 0;
end
```

With vectorized code:

```
arr1 = sin(-6*pi:pi/10:6*pi);
res = zeros(size(arr1));
res(arr1>0) = arr1(arr1>0);
```

Quiz 5.1, page 209

1. Script files are collections of MATLAB statements that are stored in a file. Script files share the Command Window's workspace, so any variables that were defined before the script file starts are visible to the script file, and any variables created by the script file remain in the workspace after the script file finishes executing. A script file has no input arguments and returns no results, but script files can communicate with other script files through the data left behind in the workspace. In contrast, each MATLAB function runs in its own independent workspace. It receives input data through an input argument list, and returns results to the caller through an output argument list.
2. The `help` command displays all of the comment lines in a function until either the first blank line or the first executable statement is reached.
3. The H1 comment line is the first comment line in the file. This line is searched by and displayed by the `lookfor` command. It should always contain a one-line summary of the purpose of a function.
4. In the pass-by-value scheme, a *copy* of each input argument is passed from a caller to a function, instead of the original argument itself. This practice contributes to good program design because the input arguments may be freely modified in the function without causing unintended side effects in the caller.
5. A MATLAB function can have any number of arguments, and not all arguments need to be present each time the function is called. Function `nargin` is used to determine the number of input arguments actually present when a function is called, and function `nargout` is used to determine the number of output arguments actually present when a function is called.
6. This function call is incorrect. Function `test1` must be called with two input arguments. In this case, variable `y` will be undefined in function `test1`, and the function will abort.
7. This function call is correct.

Quiz 6.1, page 265

1. (a) `result = 1`, because the comparison is made between the real parts of the numbers
(b) `result = 0`, because the absolute values of the two numbers are identical
(c) `result = 25`

2. The function `plot(array)` plots the imaginary part of the array versus the real part of the array, with the real part on the *x* axis and the imaginary part on the *y* axis.
3. The vector can be converted using the `double` function.
4. These statements concatenate the two lines together, and variable `res` contains the string 'This is a test!This line, too.'.
5. These statements are illegal—there is no function `strcati`.
6. These statements are illegal—the two strings must have the same number of columns, and these strings are of different lengths.
7. These statements are legal, producing the result

$$\text{res} = \begin{bmatrix} \text{This is a test!} \\ \text{This line, too.} \end{bmatrix}.$$

Note that each line is now 17 characters long, with the line 2 padded out to that length.
8. These statements are legal, and the result `res = 1`, since the two strings are identical in their first 5 characters.
9. These statements are legal, and the result is `res = [4 7 13]`, since the letter "s" is at those locations in the string.
10. These statements are legal, and the result `res = 'This IS a test!'`.
11. These statements are illegal—you must specify the number of characters to compare in the two strings when using `strcmp`.

Quiz 7.1, page 313

1. A sparse array is a special type of array in which memory is only allocated for the non-zero elements in the array. Memory values are allocated for both the subscripts and the value of each element in a sparse array. By contrast, a memory location is allocated for every value in a full array, whether the value is 0 or not. Sparse arrays can be converted to full arrays using the `full` function, and full arrays can be converted to sparse arrays using the `sparse` function.
2. A cell array is an array of “pointers”, each element of which can point to any type of MATLAB data. It differs from an ordinary array in that each element of a cell array can point to a different type of data, such as a numeric array, a string, another cell array, or a structure. Also, cell arrays use braces {} instead of parentheses () for selecting and displaying the contents of cells.
3. *Content indexing* involves placing braces {} around the cell subscripts, together with cell contents in ordinary notation. This type of indexing defines the contents of the data structure contained in a cell.

Cell indexing involves placing braces {} around the data to be stored in a cell, together with cell subscripts in ordinary subscript notation. This type of indexing creates a data structure containing the specified data, and then assigns that data structure to a cell.

4. A structure is a data type in which each individual element is given a name. The individual elements of a structure are known as fields, and each field in a structure may have a different type. The individual fields are addressed by combining the name of the structure with the name of the field, separated by a period. Structures differ from ordinary arrays and cell arrays in that ordinary arrays and cell array elements are addressed by subscript, while structure elements are addressed by name.
5. Function varargin appears as the last item in an input argument list, and it returns a cell array containing all of the actual arguments specified when the function is called, each in an individual element of a cell array. This function allows a MATLAB function to support any number of input arguments.
6. (a) $a(1,1) = [3 \times 3 \text{ double}]$. The contents of cell array element $a(1,1)$ is a 3×3 double array, and this data structure is displayed.
 (b) $a(1,1) = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$. This statement displays the *value* of the data structure stored in element $a(1,1)$.
 (c) These statements are illegal, since you can not multiply a data structure by a value.
 (d) These statements are legal, since you *can* multiply the contents of the data structure by a value. The result is $\begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}$.
 (e) $a\{2,2\} = \begin{bmatrix} -4 & -3 & -2 \\ -1 & 0 & 1 \\ 2 & 3 & 4 \end{bmatrix}$.
 (f) This statement is legal. It initializes cell array element $a(2,3)$ to be a 2×1 double array containing the values $\begin{bmatrix} -17 \\ 17 \end{bmatrix}$.
 (g) $a\{2,2\}\{2,2\} = 0$.
7. (a) $b(1).a - b(2).a = \begin{bmatrix} -3 & 1 & -1 \\ -2 & 0 & -2 \\ -3 & 3 & 5 \end{bmatrix}$.
 (b) $\text{strcmp}(b(1).b, b(2).b, 6) = 1$, since the two structure elements contain character strings that are identical in their first 6 characters.
 (c) $\text{mean}(b(1).c) = 2$

(d) This statement is illegal, since you cannot treat individual elements of a structure array as though it were an array itself.

(e) `b = 1x2 struct array with fields:`

```
a  
b  
c
```

(f) `b(1) =`
 `a: [3x3 double]`
 `b: 'Element 1'`
 `c: [1 2 3]`

Quiz 8.1, page 332

1. The `textread` function is designed to read ASCII files that are formatted into columns of data, where each column can be of a different type. This command is very useful for importing tables of data printed out by other applications, since it can handle data of mixed types within a single file.
2. MAT-files are relatively efficient users of disk space, and they store the full precision of every variable—no precision is lost due to conversion to and from ASCII format. In addition, MAT-files preserve all of the information about each variable in the workspace, including its class, name, and whether or not it is global. A disadvantage of MAT-files is that they are unique to MATLAB, and cannot be used to share data with other programs.
3. Function `fopen` is used to open files, and function `fclose` is used to close files. On PCs (but not on Unix computers), there is a difference between the format of a text file and a binary file. In order to open files in text mode a '`t`' must be appended to the permission string in the `fopen` function.
4. `fid = fopen('myinput.dat','at')`
5. `fid = fopen('input.dat','r');`
`if fid < 0;`
 `disp('File input.dat does not exist.');`
`end`
6. These statements are incorrect. They open a file as a text file, but then read the data in binary format. (Function `fscanf` should be used to read text data, as we see later in this chapter.)
7. These statements are correct. They create a 10-element array `x`, open a binary output file `file1`, write the array to the file, and close the

file. Next, it opens the file again for reading, and reads the data into array `array` in a [2 Inf] format. The resulting contents of the array

$$\text{are } \begin{bmatrix} 1 & 3 & 5 & 7 & 9 \\ 2 & 4 & 6 & 8 & 10 \end{bmatrix}.$$

Quiz 8.2, page 346

- Formatted I/O operations produce formatted files. A formatted file contains recognizable characters, numbers, etc. stored as ASCII text. Formatted files have the advantages that we can readily see what sort of data they contain, and it is easy to exchange data between different types of programs using them. However, formatted I/O operations take longer to read and write, and formatted files take up more space than unformatted files. Unformatted I/O operations copy the information from a computer's memory directly to the disk file with no conversions at all. These operations are much faster than formatted I/O operations because there is no conversion. In addition, the data occupies a much smaller amount of disk space. However, unformatted data cannot be examined and interpreted directly by humans.
- Formatted I/O should be used whenever we need to exchange data between MATLAB and other programs, or when a person needs to be able to examine and/or modify the data in the file. Otherwise, unformatted I/O should be used.
- ```
fprintf(' Table of Cosines and Sines\n\n');
fprintf(' theta cos(theta) sin(theta)\n');
fprintf(' ===== ====== ======\n');
for ii = 0:0.1:1
 theta = pi * ii;
 fprintf('%7.4f %11.5f %11.5f\n', theta,cos(theta),sin(theta));
end
```
- These statements are incorrect. Variables `a` and `b` will be printed properly, but the `%d` format descriptor will not display a string properly. The first character of the string will be displayed as a decimal number because of the `%d` format descriptor, and the remaining part of the string will be displayed properly because it will be displayed by the `%g` descriptor.
- These statements are technically correct, but the results are undesirable. It is possible to mix binary and formatted data in a single file the way that these statements do, but the file is then very hard to use for any purpose. Normally, binary data and formatted data should be written to separate files.

### Quiz 10.1, page 434

- The types of graphical components discussed in this chapter are listed below, together with their purposes.

**Table B-1. GUI Components Discussed in Chapter 10**

| <b>Component</b>          | <b>Created By</b> | <b>Description</b>                                                                                                                                                                                                                                   |
|---------------------------|-------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Graphical Controls</b> |                   |                                                                                                                                                                                                                                                      |
| Pushbutton                | uicontrol         | A graphical component that implements a pushbutton. It triggers a callback when clicked with a mouse.                                                                                                                                                |
| Toggle Button             | uicontrol         | A graphical component that implements a toggle button. A toggle button is either "on" or "off", and it changes state each time that it is clicked. Each mouse button click also triggers a callback.                                                 |
| Radio Button              | uicontrol         | A radio button is a type of toggle button that appears as a small circle with a dot in the middle when it is "on". Groups of radio buttons are used to implement mutually exclusive choices. Each mouse click on a radio button triggers a callback. |
| Check Box                 | uicontrol         | A check box is a type of toggle button that appears as a small square with a check mark in it when it is "on". Each mouse click on a check box triggers a callback.                                                                                  |
| Edit Box                  | uicontrol         | An edit box displays a text string, and allows the user to modify the information displayed. A callback is triggered when the user presses the <code>Enter</code> key.                                                                               |
| List Box                  | uicontrol         | A list box is a graphical control that displays a series of text strings. A user may select one of the text strings by single- or double-clicking on it. A callback is triggered when the user selects a string.                                     |
| Popup Menus               | uicontrol         | A popup menu is a graphical control that displays a series of text strings in response to a mouse click. When the popup menu is not clicked on, only the currently-selected string is visible.                                                       |
| Slider                    | uicontrol         | A slider is a graphical control to adjust a value in a smooth, continuous fashion by dragging the control with a mouse. Each slider change triggers a callback.                                                                                      |
| <b>Static Elements</b>    |                   |                                                                                                                                                                                                                                                      |
| Frame                     | uicontrol         | Creates a frame, which is a rectangular box within a figure. Frames are used to group sets of controls together. Frames never trigger callbacks.                                                                                                     |
| Text Field                | uicontrol         | Creates a label, which is a text string located at a point on the figure. Text fields never trigger callbacks.                                                                                                                                       |
| <b>Menus and Axes</b>     |                   |                                                                                                                                                                                                                                                      |
| Menu Items                | uimenu            | Creates a menu item. Menu items trigger a callback when a mouse button is released over them.                                                                                                                                                        |
| Context Menus             | uicontextmenu     | Creates a context menu, which is a menu that appears over a graphical object when a user right-clicks the mouse on that object.                                                                                                                      |
| Axes                      | axes              | Creates a new set of axes to display data on. Axes never trigger callbacks.                                                                                                                                                                          |

2. A callback function is a function that is executed whenever an action (mouse click, keyboard input, etc.) occurs on a specific GUI component. They are used to perform an action when a user clicks on or types in a GUI component. Callback functions are specified by the 'Callback' property in a uicontrol, uimenu, or uicontextmenu. When a new GUI is created, the uicontrol callbacks are set automatically by guide to be `xxx_Callback`, where `xxx` is the value of the Tag property of the corresponding GUI component. Menu callbacks are not created automatically—they must be defined manually by the programmer.
3. The basic steps required to create a MATLAB GUI are:
  1. Decide what elements are required for the GUI, and what the function of each element will be. Make a rough layout of the components by hand on a piece of paper.
  2. Use a MATLAB tool called `guide` (GUI Development Environment) to lay out the components on a figure. The size of the figure, and the alignment and spacing of components on the figure, can be adjusted using the tools built into `guide`.
  3. Use a MATLAB tool called the Property Inspector (built into `guide`) to give each component a name (a "tag"), and to set the characteristics of each component, such as its color, the text it displays, etc.
  4. Save the figure to a file. When the figure is saved, two files will be created on disk with the same name but different extents. The `.fig` file contains the actual GUI that you have created, and the `.M`-file contains the code to load the figure, and also skeleton callbacks for each GUI element.
  5. Write code to implement the behavior associated with each callback function.
4. The `handles` data structure is a structure containing the handles of all components within a figure. Each structure element has the name of a component, and the value of the component's handle. This structure is passed to every callback function, allowing each function to have access to every component in the figure.
5. Application data can be saved in a GUI by adding it to the `handles` structure, and saving that structure after it is modified using function `guidata`. Since the `handles` structure is automatically passed to every callback function, and additional data added to the structure will be available to any callback function in the GUI. (Each function that modifies the `handles` structure must be sure to save the modified version with a call to `guidata` before the function exits.)

6. A graphical object can be made invisible by setting its 'Visible' property to 'off'. A graphical object can be disabled so that it will not respond to mouse clicks or keyboard input by setting its 'Enable' property to 'off'.
7. Pushbuttons, toggle buttons, radio buttons, check boxes, list boxes, popup menus, and sliders all respond to mouse clicks. Edit boxes respond to keyboard inputs.
8. A dialog box is a special type of figure that is used to display information or to get input from a user. Dialog boxes are used to display errors, provide warnings, ask questions, or get user input. Dialog boxes can be created by any of the functions listed in Table 10.4, including `errordlg`, `warndlg`, `inputdlg`, `uigetfile`, etc.
9. A modal dialog box does not allow any other window in the application to be accessed until it is dismissed, while a normal dialog box does not block access to other windows.
10. A standard menu is tied to a menu bar running across the top of a figure, while a context menu can be attached to any GUI component. Standard menus are activated by normal mouse clicks on the menu bar, while context menus are activated by mouse right-clicks over the associated GUI component. Menus are built out of `uimenu` components. Context menus are built out of both `uicontextmenu` and `uimenu` components.
11. Accelerator keys are keys that may be typed on the keyboard to cause a menu item to be selected. Keyboard mnemonic keys are **CTRL+key** combinations that cause a menu item to be executed. The principal difference between accelerator keys and keyboard mnemonics is that accelerator keys only work to select a menu item if a menu has already been opened, while keyboard mnemonics can trigger an action even if a menu has not been opened.

# Index

---

Note **Boldface numbers** indicate illustrations.

- & logical AND operator, 90
- : to suppress automatic echo of values, 26–27, 70
- : operator for shortcut expressions, 27–28, 70
- [ ] constructor brackets, arrays, 70
- ... continuation characters (ellipsis), 5, 71
- = and == operators, 87–90, 130, 244, 245
- ! character, 15
- >= operator, 88
- ~= nonequivalence operator, 88, 139, 244
- ~ logical NOT operator, 90
- % as comment delimiter, 70
- | logical OR operator, 90
  
- Abort command, 15
- abs, 53, 71, 245, 282
- accelerator keys, 426, 430
- acos function, 53, 71
- actual arguments, 189
- addition, 16, 70
  - complex numbers, 243
  - hierarchy of operations, 48–50, 69
  - matrix addition, 46, 70
- addpath command, 16
- advantages of MATLAB, 1–3
- algorithms, 84
- alpha releases, in program development, 85–86
- AND logic operator (&), 90–92
- angle, 53, 245, 282
- annotating plots, 123–125, **123, 124, 125**
- ans predefined special value, 37, 71
  
- argument lists, 188, 189, 228
- arguments, 189
- arithmetic functions, 2
- arithmetic mean
  - running-average calculation using persistent memory, 217–222
  - statistical analysis using for loop, 148–150
  - statistical analysis using while loop, 138–143
- arithmetic operations
  - hierarchy of operations, 48–50, 69, 89
  - logical arrays, 158–159
  - parentheses, 48–50, 69, 89
  - precedence of operators, 48–50, 69, 89
- Array Editor, 11, 12
- arrays, 3, 21, 45–48
  - addition, matrix addition, 46
  - array operations, 45–48
  - cell arrays, 287, 294–305, 314–315
  - colon operator for shortcut expressions, 27–28
  - constructor brackets, 70
  - division, matrix division, 46
  - Editor, 11, 12
  - element-by-element performance of operations, 45
  - empty arrays, 26
  - exponentiation, 46
  - eye function to initialize variables/arrays, 29
  - for loops and preallocated arrays, 151, 178
  - functions using array input, 52
  - identity matrices to initialize variables/arrays, 29
  - initializing arrays, 25–26

- arrays (continued)**
- initializing variables using arrays, 25
  - Input** function to initialize variables/arrays, 29
  - keyboard input initialization of variables/arrays, 29–30
  - Length** function to initialize variables/arrays, 29
  - logical and vectorization, 157–162
  - matrix arrays, 21, 25
  - matrix operations, 45–48
  - multidimensional arrays, 31–34, 31
  - multiplication, matrix multiplication, 45
  - naming arrays, 22
  - ndims** function, 268
  - number of elements, number of rows in array, 26
  - ones** function to initialize variables/arrays, 29
  - relational operators, 89
  - scalar value through use of relational operators, 89
  - shortcut expression to initialize arrays, 27–28
  - Size** function to initialize variables/arrays, 28, 29
  - size of arrays, 21
  - sparse arrays, 287–294, 314–315
  - structure arrays, 287, 306–313, 307, 315
  - subarrays, 34–37
  - subtraction, matrix subtraction, 46
  - transpose operator for shortcut expressions, 28, 71
  - two-dimensional arrays, 31, 252–253, 268–276, 281
  - variables, 22
  - vector arrays, 21, 25
  - zeros** function to initialize variables/arrays, 28, 29
- Arrow tool**, 124
- ASCII format/ASCII files**, 43, 70, 342–346, 359
- character set, 449
  - lexicographic order of characters in ASCII tables, 260–261
  - load** command, 321
  - save** command, 42, 72, 319, 321, 323, 324, 358, 359
  - textread** function, 321
- asin** function, 53, 71
- assignment operator (=)**, 44, 89
- assignment statements**
- cell arrays, 296, 297
  - initialize variables, 25–27
  - scalar operations, 45
  - structure arrays, 306–308
  - subarrays, 35
  - suppress automatic echo of values, semicolon, 26, 70
- atan** function, 53, 71
- atan2** function, 53, 71
- averages**
- running-average calculation using persistent memory, 217–222
  - statistical analysis using **for** loop, 145–150
  - statistical analysis using **while** loop, 138–143
- axes**, 108, 111, 130, 365, 393
- graphical user interface (GUI), 392
  - axes objects, 393, 394
  - graphical user interface (GUI), 392
  - handle graphics, 381, 388
  - axis** command, 108
- bar**, 269, 282
- bar plots**, 268–269, 271, 282
- bash**, 269, 282
- base2dec**, 259, 261, 282
- bin2dec**, 259, 262, 282
- binary I/O functions**, 328–332, 342–346, 358
- binary logic**, 90
- blank removal from string, 253
- blanks**, 269, 282
- blocks of code**, in program development, 94
- boldface** text, plotting, 115, 130
- braces as cell delimiters**, 292, 294, 296, 298
- branching statements**, 81, 128–130
- assigning letter grades example, using **if** construct, 103–104
  - case** statements, 104–106
  - constructs, 86
  - debugging, 125–128, 126, 127, 128
  - else** clause, 94–95, 102
  - elseif** clause, 94–95, 102–104, 130
  - end** statement, 102
  - evaluating function of two variables using **if** construct, 99–100
  - if** construct, 94–104, 130
  - indenting code, program development, 94, 130
  - nested statements, 102–104
  - pseudocoding, 84, 86–87
  - quadratic equation using **if** construct, 96–99
  - relational operators, 87–89
  - structured programming, 86–87
  - switch construct, 104–106, 130
  - top-down program design, 81–86, 83, 85
  - try/catch** construct, 106–107
  - break** statement, 154–155, 179
  - breakpoints, debugging, 125–126, 127, 128
  - builds, in program development, 84–86
  - built-in functions, 51–52
  - ButtonDownFcn** property (GUI), 439
  - callbacks in GUI, 388, 392, 394, 398
  - carbon-14 dating example, 64–67, 65
  - cardioid microphone example of polar plot, 115–117, 118, 381–384
  - case** statements, 104–106
  - case-sensitivity of names, 23, 70
  - categorize characters within a string, 254–255
  - ceil**, 53, 71
  - cell**, 296, 297, 305, 315
  - cell arrays, 287, 294–305, 314–315
  - arrays vs. cell arrays, 297

- braces as cell delimiters, 294, 296, 297, 298  
`cell`, 296, 297, 305, 315  
 cell indexing, 297, 300–301  
`celldisp`, 298, 305, 315  
`cellplot`, 298, 299, 305, 315  
`cellstr`, 301, 305, 315  
`char` function, 301, 305  
 content indexing, 297, 300–310  
 contents of cells, 294  
 creating cell arrays using assignment statements, 296, 297  
 data structures vs. data in cell arrays, 294, 295, 295  
 data vs. data structures, 300–301  
 deleting cells in array, 299  
 extending cell arrays, 298–299, 299  
 graphical user interface (GUI), 287, 392  
 indexing, 297, 300–301  
 plotting, 302, 304  
 pointers, 294, 295  
 preallocating cell arrays using `cell` function, 296, 297  
 strings in cell arrays, 301  
`varargin`, 302–304, 315  
`varargout`, 304–305, 315  
 viewing contents of cell arrays, 298  
 cell indexing, cell arrays, 297, 300–301  
`celldisp`, 298, 305, 315  
`cellplot`, 298, 299, 305, 315  
`cellstr`, 301, 305, 315  
`char`, 24, 53, 71, 241, 252–253, 258, 260  
 character data, 336  
 check boxes in GUI, 393, 394, 411  
 child objects, handle graphics, 364, 388  
`clc` command, 14  
`clear` Command, 14–15, 300  
`clf` Command, 14  
 clock predefined special value, 37, 71  
 colon operator for shortcut expressions, 27–28, 70  
 color, plotting, 56–58  
 column major order, 32  
 column order of data, write operations, 328  
 Command History Window, 7, 7  
 Command Window, 4–6, 6, 18  
     `clear`, `cfc` command, 14  
     `diary` command, 15  
 commands, 70–73  
     command/function duality, 109  
 comments, 70, 190  
     H1 comment lines, 190  
 comparing strings, substrings, 254–256, 260, 261, 262–265  
 compass, 269, 282  
 compass plots, 268–269, 272, 282  
 compiler, 3, 67  
 complex data, complex numbers, 241–251, 281  
     addition, 243  
     `angle` function, 245  
     complex functions, 244–248  
     `conj` function, 245  
     division, 243  
     `imag` function, 244, 245  
     imaginary part of complex number, 241–242, 244, 281  
     `isreal` function, 243, 245  
     mathematical functions, 245  
     multiplication, 243  
     `plot` function, 72, 248, 250, 276, 288, 365  
     plotting, 241–242, 242, 248–251, 249, 250, 251  
     polar coordinate plot, 242, 242, 251, 251  
     `real` function, 244, 245  
     real part of complex number, 241–242, 244, 281  
     rectangular coordinates plot, 242, 242, 243  
     relational operators used with complex numbers, 244  
     subtraction, 243  
     type conversion functions, 244  
     variables, complex variables, 243  
 complex functions, 244–248  
 complex variables, 243  
 concatenating strings, 253–254  
`conj`, 245, 282  
 constructor brackets, arrays, 70  
 constructs, in program development, 86–87  
 content indexing, cell arrays, 297, 300–301  
 context menus, 430–431, 433  
 continuation characters (ellipsis), 5, 71  
`continue` statement, 154–155  
`contour`, 279, 281, 282  
 contour plots, 278–280, 280  
 controls in the GUI, 392  
 conversion characters for `fprintf` function, 40–41  
 conversion functions, string functions, 252, 258–259  
`cos`, 53, 71, 245  
`createFcn` property (GUI), 439  
 current object, handle graphics, 376, 388  
 data dictionaries, variables, 23, 70  
 data files, 42–44  
     ASCII format, 23, 70  
     saving files, 43–45, 70  
 data hiding, 188  
 data structures vs. data in cell arrays, 294, 295, 295  
 data types, 24, 241–281  
     cell arrays, 287, 294–305, 315  
     sparse arrays, 287–294, 314–315  
     structure arrays, 287, 306–313, 307, 315  
     type conversion functions, 244  
 date predefined special value, 37, 71  
`deblank`, 253, 269, 282  
 debugging MATLAB programs, 67–69, 125–128, 126, 127, 128, 130, 189  
 branching statements, 125  
 breakpoints, 126–128, 126, 127, 128  
 Edit/Debug Window, 8–9

debugging MATLAB programs (*continued*)  
 input validation, 68  
 logical errors, 67  
 loops, 125–126  
 parentheses, 67  
 roundoff errors, 89, 130  
 run-time errors, 67  
 symbolic debugger tool, 68–69, 125–128, 127, 128, 130  
 syntax errors, 67  
 typographical errors, 67–68  
**dec2base**, 259, 261, 282  
**dec2bin**, 259, 261, 282  
**dec2hex**, 259, 261  
**decay**, radioactive, 64–65  
**decimal data**, 334–335  
**decomposition** in program design, 84, 187  
**default format**, 39–40  
**default properties**, handle graphics, 385–388  
**delete**, 324  
**DeleteFcn** property (GUI), 439  
**demo** command, 14  
**Desktop**, 4, 5  
 device-independent plotting, 2  
**dialog boxes** in GUI, 422–425  
 error dialog boxes, 423, 423  
 input dialog boxes, 423–424  
 modal vs. non-modal dialog boxes, 422  
 properties for dialog boxes, 423  
 warning dialog boxes, 423  
**diary** command, 15  
**directories**  
 path, search path, 15–16  
 private subdirectories, private functions, 228  
**disadvantages of MATLAB**, 3  
**discriminants** in quadratic equations, 95  
**disp**, 40, 71  
**displaying output data** (See *formatting output data*)  
**divide-by-zero error**, 142  
**division**, 16  
 complex numbers, 243  
 hierarchy of operations, 48–51, 69  
 matrix division, 45, 71  
**doc**, 71  
**double**, 24, 53, 71, 241, 252, 260, 282  
**duality of functions/commands in MATLAB**, 109  
**dummy arguments**, 189  
**ease of use of MATLAB**, 2  
**Edit box** in GUI, 393, 394, 407, 409  
**Edit/Debug Window**, 8–9, 9, 19  
**Editing button**, 123  
**editpath** command, 16  
**Editing tools**, 123, 124  
**element-by-element performance of operations**, 45  
**ellipsis** to wrap around line of code, 5, 71  
**else clause**, 94–95, 102  
**elseif clause**, 94–95, 102–104  
**empty arrays**, 26  
**end function**, subarray, 34  
**end statements**, 95, 156  
**end-of-file indicator** (`feof`), 350  
**engineering applications**  
 frequency response of low-pass filter plot example, 117–120, 118, 120  
 maximum power transfer to load example, 62–64, 62, 64  
**environment of MATLAB**, 3–16  
**eps** predefined special value, 37, 71  
**equality operator** (`==`), 88–90, 130, 244, 254  
**error**, 205, 207, 229  
**error dialog boxes**, 423, 423  
**escape characters** for `fprintf` function, 41  
**escape codes** for text, plotting, 115, 130  
**eval**, 222–223, 229, 259–260  
**evaluating function of two variables using `zf` construct**, 99–102  
**events and event handling in GUI**, 392  
**exclamation point character**, 15  
**exclusive OR (XOR) logic operator**, 90–92  
**execution of functions**, 189  
**exist**, 324, 347–349, 360  
**exp**, 53, 71  
**explode parameter** for `pie` function, 273  
**exponentiation**, 16, 43, 71  
 hierarchy of operations, 48–51, 69  
**expressions and assignment statements**, 25–27  
**eye**, 71  
 initialize variables/arrays, 29–30  
**ezplot**, 223, 229, 273–274, 274  
**factorial function** using `for` loop, 145  
**factory properties**, handle graphics, 385–387  
**fclose**, 324, 328, 359  
**feof**, 324, 350, 359  
**ferror**, 324, 347, 349–350, 359  
**feval**, 222–223, 229  
**fgetl**, 324, 341, 359  
**fgets**, 324, 341, 359  
**fieldnames**, 313, 315  
**fields in structure arrays**, 306, 308–309, 310–311, 310, 315  
**figure**, 111, 112, 112, 130, 365, 388, 392  
**Figure Windows**, 9–10, 10, 19  
 clear, `clf` command, 14  
 figure function, 111–112, 112, 130, 364, 388, 392  
 multiple Figure Windows, 111–112, 130  
**figures**  
 graphical user interface (GUI), 392  
 handle graphics, 364, 388  
**Figure toolbar**, 123, 124  
**file id (fid)**, 323

file positioning functions, 347–356  
 file processing, 323–324  
 filters, frequency response of low-pass filter plot example, 117–120, 118, 120  
`find`, 282, 292  
 finding objects, 375–377  
`findobj`, 376–377, 388  
`findstr`, 255, 260, 282  
`fix`, 53, 71  
 floating point data, 335–336  
`floor`, 53, 71  
`fmin`, 229  
`fminbnd`, 223  
 font selection, plotting, 115, 130  
`fopen`, 324, 325–327, 359  
 for loop, 137, 143–157, 178–179  
     array values used in for loops, 151, 178  
     break statement, 154–155  
     continue statement, 154–155  
     end statements, 156  
     indenting code, 150, 178  
     index, loop index, 143–145, 150, 178  
     nested loops, 156–157, 172  
     preallocated arrays in for loops, 151, 178  
     vectors vs. for loops, 151, 178  
`format`, 39–40, 71–72  
 format conversion specifiers, 333, 334–336  
 format flags, 333, 334  
 format strings, 336–339  
 formatted I/O functions, 332–346  
 formatted vs. unformatted files, 342–346, 358, 359  
 formatting output data, 39–42, 72  
     default format, 39  
     `disp` function, 40  
     escape characters for `fprintf` function, 41  
     `format` and options, 39–40, 72  
     formatted I/O functions, 332–346  
     formatted vs. unformatted files, 342–346, 358, 359  
     `fprintf`, 40–42, 72  
`fplot`, 223, 229, 273–274, 282  
`fprintf`, 40–42, 72, 259, 319, 324, 332–333, 336–339, 360  
 frame in GUI, 393, 409  
`fread`, 324, 329–330, 359  
 free MATLAB programs, 2  
 frequency response of low-pass filter plot example, 117–120, 118, 120  
`frewind`, 324, 347, 350, 360  
`fscanf`, 324, 339–341, 360  
`fseek`, 324, 350–351, 360  
`ftell`, 324, 350, 360  
`full`, 282, 290, 292, 315  
 function functions, 222–224, 228  
 function statement, 189  
 functions, 1, 2, 51–54, 54, 72–73  
     actual arguments, 189  
     argument lists, 188, 189, 228  
     arguments, 189  
     array input for functions, 52  
     beginning of function, Function statement, 189  
     binary I/O functions, 328–332  
     built-in functions, 51–54  
     command/function duality, 109  
     comment lines, 190  
     complex functions, 244–248  
     dummy arguments, 189  
     `eval` function, 222–223, 229, 259–260  
     execution of functions, 189  
     `feval` function, 222–223, 229  
     formatted I/O functions, 332–341  
     function functions, 222–223, 229  
     function statement, 189  
     `fzero` function, 222  
     global memory to share data, 209–217, 228  
     global variables, 210, 228  
     H1 comment lines, 190  
     initializing variables using functions, 23–29  
     input argument lists, 188, 228  
     input/output functions, 319–362  
     internal functions (See subfunctions)  
     logical functions, 92–93  
     magnitude and angle plot using optional arguments, 206–208  
     `nargin`, 205, 228, 229  
     `nargout` function, 205, 208, 228, 229  
     optional arguments, 204–208  
     output argument lists, 188, 189, 228  
     pass-by-value variable passing, 194–204, 228  
     persistent memory, 217–222, 228  
     persistent statement, 217–222, 228  
     plotting functions, 273–274  
     preserving data between calls to function, 217–222, 228  
     private functions, 227, 228  
     random-number generator using global variables, 211–217, 216  
     rectangular-to-polar conversion, 195–198  
     return statements, 189  
     reusable code, 188  
     script files, 189  
     sorting data, 199–204, 200  
     subtasks and functions, 187  
     subfunctions, 225–227, 228  
     type conversion functions, 244  
     user-defined functions, 187–240  
`fwrite`, 324, 328–329, 360  
`fzero`, 222, 223, 229  
  
`gca` function, 376–377  
`gcbf`, graphical user interface (GUI), 406  
`gcbo`, graphical user interface (GUI), 406  
`gcf`, 376–377, 388, 389

gco, 376–377, 388, 389  
 get, 366, 389  
**getfield**, 311–312, 313, 315  
**global**, 210, 229  
 global memory to share data, 209–219, 229  
 global variables, 210, 228  
 graphical user interface (GUI), 3, 391–447  
     accelerator keys, 426, 430  
     adding application data to a figure, 405  
     axes function, 108–111, 130, 365, 393, 446  
     axes objects, 392, 393, 394  
     **ButtonDownFcn** property, 439  
     callbacks, 392, 394, 399–402, 401, 404, 406, 413, 426, 434, 435, 436  
     cell arrays, 287, 302  
     check boxes in GUI, 393, 394, 411  
     components of a GUI, 392, 395, 407–422  
     context menus, 430–431, 433  
     controls in the GUI, 392  
     **createFcn** property, 439  
     creating a MATLAB GUI, 392–406  
     **DeleteFcn** property, 439  
     dialog, 446  
     dialog boxes in GUI, 422–425, 423, 424, 425, 445  
     Edit box in GUI, 393, 394, 409  
     error dialog boxes, 423, 423  
     **errordlg**, 446  
      eventdata, 404  
     event handling, 401  
     event-driven programs, 391  
     events, 391, 392  
     figure function, 392  
     figure properties for GUI, 407  
     **findobj** function, 376–377, 388, 406, 446  
     frame in GUI, 393, 409  
     **gcbf** function, 406, 446  
     **gcbo** function, 406, 446  
     guidata, 405, 445, 446  
**GUI Development Environment (See guide tool)**  
 guide tool for GUI creation, 392, 394, 399, 492, 445, 446  
 guihandles, 403, 446  
**handle graphics**, 363  
 handles, 494–405, 445  
 helpdlg, 446  
 histogram GUI example, 439–443, 441, 442, 444  
 input, 446  
 input dialog boxes, 423–424  
 keyboard mnemonic keys, 430, 431  
 Layout Editor, 414  
 list box in GUI, 393, 394, 414–417, 416  
 Menu Editor, 395, 428, 428, 433  
 menus in GUI, 392, 393, 425–436, 427, 428  
 M-file, 402–404  
 modifying GUI manually, 402, 403, 405, 429–430, 438  
 pcode for GUI, 437–438  
 persistent variables, 399  
 plots, annotating and saving, 123–125, 123, 124, 125  
 plotting data points, 431–434, 432, 436, 437  
 popup menus in GUI, 393, 414, 415  
 printdlg, 446  
 properties for GUI objects, 406–407  
 Property Editor (propedit) tool, 394, 428, 429, 430  
 Property Inspector, 394, 395, 396, 397, 406, 445  
 pushbutton in GUI, 393, 394, 394, 395, 397, 398  
 questdlg, 446  
 radio buttons in GUI, 393, 394, 411, 413  
 slider in GUI, 393, 394, , 417, 418  
 static elements of GUI, 352, 393  
 String property, 397  
 Tag property, 397  
 temperature conversion example using slider, 418–422, 420  
 text field in GUI, 393, 394, 395, 398, 409, 412  
 toggle buttons in GUI, 393, 394, 411, 412  
 tool tips in GUI, 436–437  
 uicontextmenu objects, 428, 429, 430–434  
 unicontrol and uicontrol objects, 392, 393, 406, 408, 446  
 uigetfile dialog boxes, 424–425, 425, 446  
 uimenu and uimenu objects, 393, 426, 446  
 uiputfile, 446  
 uisetcolor, 446  
 uisetfile dialog boxes, 424–425  
 uisetfont, 446  
 varargin, 404  
 warndlg, 446  
 warning dialog boxes, 423  
 WindowButtonDownFcn, 439  
 WindowButtonMotionFcn, 439  
 WindowButtonUpFcn, 439  
 graphics (See handle graphics)  
 gravity  
     velocity of falling object in gravitational field, 169–178, 170, 177  
 greater than equal to operator, 88  
 greater than operator, 88  
 grid command, 54, 72  
 guide tool for GUI creation, 392, 394–395, 399, 445, 446  
  
 H1 comment lines, 190  
 handle graphics, 363–390  
     axes function, 108–111, 130, 365, 388, 393  
     axes objects, 381, 388  
     changing properties after creation, 366, 368  
     changing properties at creation, 365–366  
     child objects, 364, 388  
     current object, 376, 388  
     default properties, 385–387  
     factory properties, 385–387

**figure**, 365, 389  
**figure objects**, 380–384, **384**  
**Figure Windows**, 9–10, **10**, 364  
**figures**, 364, 388  
 finding objects, 375–377  
**findobj**, 376–377, 388  
**gca**, 365, 376–377, 388, 389  
**gcf**, 365, 376–377, 388, 389  
**gco**, 365, 376–377, 388, 389  
**get**, 366, 389  
**getappdata**, 389  
**graphical user interface (GUI)**, 363  
**graphic object properties**, 387  
**graphics objects in MATLAB**, 363  
**handles**, 365  
 hierarchy of handle graphics, **364**  
**hdl1 command**, 365, 374  
**isappdata**, 375  
**line function**, 365  
 lines, properties of lines, 363, **367**, 369  
 listing possible property values, **set** function, 370,  
     372–376, 388  
 object handles, 365  
 parent objects, 364, 388  
**plot function**, 72, 248–251, 276, 288, 365  
 positions of objects, 380–384, **384**  
 printer position, 384–385  
 properties of graphics objects, 363, 388  
**Property Editor**, 368–370, **369**  
 property names for objects, 365  
 root objects, 364, 388  
**rmappdata**, 389  
 selecting objects with mouse, 377–380  
 selection region of objects, 377–380  
**set**, 370, 372–376, 389  
**setappdata**, 375  
**struct**, 375  
 text objects, 381  
**uicontextmenus**, 364, 388  
**uicontrol objects**, 364, 381, 388  
**uimenus objects**, 364, 388  
 units of measure, 380–384  
 user-defined data, 374–375  
**value**, 375  
 values for objects, 365  
**waitforbuttonpress** function, 377–379, 389  
**handles, handle graphics**, 365  
**Help**, 13–14, 19  
**help command**, 13, **13**  
**Help Browser**, 13, **13**  
**Help Window**, 13, 13–14  
**hex2dec**, 259, 261, 282  
**hex2num**, 259, 261, 282  
**hierarchy (See precedence of operators)**  
**hierarchy of handle graphics**, **364**  
**hist**, 215–217, 229, 275–276, 282  
**histogram GUI example**, 439–443, **441**, **442**, **444**  
**histogram plot of random-number generator**, 215–217, **216**  
**histograms**, 274–276, **275**  
**hdl1 command, handle graphics**, 365  
**hold**, 111–112, **112**, 130, 172  
**i predefined value**, 37, 72  
 identity matrices to initialize variables/arrays, 29  
**if construct**, 94–104, 130  
     assigning letter grades example, using **if construct**,  
     103–104  
     **else clause**, 94–95, 102  
     **elseif clause**, 94–95, 102–104, 130  
     **end statement**, 102  
     indenting code, program development, 97, 130  
     **nested ifs**, 102–104  
     quadratic equation using **if construct**, 96–99  
**if/else construct**  
     creating with logical arrays, 161–162  
**imag**, 245, 282  
**imaginary part of complex number**, 241–242, 244, 281  
**importing data, textread function**, 319–321  
 indenting code, program design, 96, 130, 150, 178  
**index, loop index**, 143–145, 150, 178  
**indexing, cell arrays**, 297, 300–301  
**inf predefined special value**, 37, 72  
**initializing arrays**, 25–27  
**initializing variables/arrays**, 24–30  
     arrays to initialize variables, 25  
     assignment statements to initialize variables, 25–27  
     built-in functions to initialize variables, 28–29  
     colon operator for shortcut expressions, 27–28  
     eye function to initialize variables/arrays, 29  
     identity matrices to initialize variables/arrays, 29  
     Input function to initialize variables/arrays, 29  
     keyboard input initialization of variables/arrays, 29–30  
     length function to initialize variables/arrays, 29  
     ones function to initialize variables/arrays, 29  
     shortcut expression to initialize variables, 27–28  
     size function to initialize variables/arrays, 28, 29  
     transpose operator for shortcut expressions, 29, 71  
     zeros function to initialize variables/arrays, 28, 29  
**input**, 29, 72  
**input argument lists**, 188, 189, 228  
     **input/output functions**, 319–362  
         **varargin function**, 302–304, 315  
**input dialog boxes**, 423–424  
**inputname**, 205–206, 229  
**input/output functions**, 319–362  
     ASCII files, 342–346, 359  
     binary I/O functions, 328–332, 342–346, 358  
     character data, 336  
     closing files with **fclose**, 324, 328, 360  
     column order of data, write operations, 328

**input/output functions (*continued*)**

- decimal data, 334–335
- delete, 324
- escape characters for format strings, 334
- exist, 324, 347–349, 359
- `fclose`, 324, 328, 359
- `feof`, 324, 350, 359
- `ferror`, 324, 347, 349–350, 359
- `fgetl`, 324, 341, 359
- `fgets`, 324, 341, 359
- file id (`fid`), 323, 327
- file positioning functions, 347–356
- file processing, 323–324
- floating point data, 335–336
- `fopen`, 324, 325–327, 359
- format conversion specifiers, 333, 334–336
- format flags, 333, 334
- format for `fopen` strings, 324, 325–327, 359, 360
- format strings, 336–339
- formatted I/O functions, 332–341
- formatted vs. unformatted files, 342–346, 358, 359
- `fprintf`, 324, 332–333, 336–339, 359
- `fread`, 324, 329–330, 359
- `frewind`, 324, 347, 350, 360
- `fscanf`, 324, 339–341, 360
- `fseek`, 324, 350–351, 360
- `fTell`, 324, 350, 360
- `fwrite`, 324, 328–329, 360
- load, 321, 322, 324, 358, 359
- opening binary file for input, 326
- opening binary file for read/write access, 327
- opening file for text output, 326
- opening files with `fopen`, 324, 325–327, 359, 360
- overwriting files, 347–349, 359
- permissions, `fopen`, 325, 327, 359
- precision strings, read operations, 330
- precision strings, write operations, 328–329
- save, 42, 72, 319, 321, 323, 324, 358, 359
- status functions, 347–358, 359
- `stderr` file, 323
- `stdout` file, 323
- table of data using `fprintf` function, 338–339
- `tempdir`, 324
- `tempname`, 324
- `textread`, 319–321, 359, 360
- `uiimport`, 356–358, 357, 358
- `int2str`, 40, 53, 72, 258–259, 260, 282
- internal functions (See subfunctions)
- `isappdata`, 375
- `ischar`, 92, 130, 252, 260, 282
- `isempty`, 92, 130
- `isinf`, 92, 130
- `isletter`, 254–255, 260, 282
- `isnan`, 92, 130
- `isnumeric`, 92, 130
- `ireal`, 243, 245, 282
- `isspace`, 254–255, 261, 282
- `isspare`, 282, 292
- italic text, plotting, 115, 130
- `j` predefined value, 37, 72
- keyboard mnemonic keys, 430, 431
- landscape mode, 385
- Launch Pad, 7–8, 8
- least squares method, regression analysis, 164, 351
- legend, 58, 72
- legends, plotting, 56–58
- length, 29, 72
- less than equal operator, 88
- less than operator, 88
- lexicographic order of characters in ASCII tables, 260–261
- line color, plotting, 56–58
- line function, 365
- line style, line properties, plotting, 56–58, 114, 130, 363
- Line tool, 124
- linear regression
  - for loops, 163–169
  - I/O functions, 351–358, 358
- LineWidth property, plotting, 114
- list box in GUI, 393, 394, 414–417, 416
- load, 42, 72, 319, 321, 323, 324, 358, 359
- log, 53, 72, 245
- logarithmic scales, 58–59
- logfiles, diary command, 15
- logical arrays, 157–162
  - creating the equivalent of `if/else` constructs, 161–162
  - mask for arithmetic operations, 158–159
  - using to mask operations, 159–161
  - significance, 158–159
- logic operators, 90–92
- logical errors, 67
- logical functions, 92–94
- `loglog`, 59, 72
- `lookfor`, 14, 72, 190
- loop index, 143–145, 150, 178
- loops, 137–186
  - Abort command, 15
  - `break` statement, 154–155
  - `continue` statement, 154–155
  - debugging, 105, 125–126
  - end statements, 134, 156
  - factorial function using `for` loop, 145
  - `for` loop, 137, 143–157, 178–179
  - index, loop index, 143–145, 150, 178
  - logical arrays and vectorization, 157–162
  - linear regression using `for` loops, 163–169
  - nested loops, 156–157, 172
  - preallocated arrays in `for` loops, 151, 178

- statistical analysis using `for` loop, 148–150  
 statistical analysis using `while` loop, 138–143  
 vectors vs. `for` loops, 150–154, 178  
`while` loop, 137–143, 178–179  
`lower`, 257–258, 260, 282  
**low-pass filters, frequency response of low-pass filter plot,** 117–120, **118, 129**
- Macintosh**, 2  
 magnitude and angle plot using optional arguments, 206–208  
 marker properties and styles, plotting, 56–58, 114, 130  
`mat2str`, 259, 261, 282  
**MAT-files**, 322–323  
 mathematical calculations, 16  
 mathematical functions, 53  
     optional (varying) results, 51–52  
 matrix, sparse arrays/sparse matrices, 287–294  
 matrix arrays, 21, 25  
 matrix operations, 45–48  
`max`, 51, 53, 72  
**maximum power transfer to load example**, 62–64, **62, 64**  
`maxval`, 51  
`mean`, 136, 142  
`median`, 2  
**memory**  
     global memory to share data, 209–217, 228  
     persistent memory, 217–222, 228  
     menus in GUI, 392, 393, 425–436, **427, 428**  
     accelerator keys, 426, 430  
     context menus, 430–431, **433**  
     creating a menu, 429  
     default menu, 428–429  
     keyboard mnemonic keys, 430, **431**  
     parent objects for menus, 388  
     properties for menus, 425–426  
     standard menus, 426  
     storing multidimensional arrays, 32  
     structure of typical menu, 426, **427, 428**  
     suppressing default menu, 428–429  
     uicontextmenu properties, 428, 429  
     uicontrol objects, 392  
     uimenu objects, 393  
`mesh`, 279, 281, 282  
**mesh plots**, 278–281, **279**  
`meshgrid`, 281, 283  
**method of least squares, regression analysis**, 164, 351  
**M-file**, 8, 15, 16, 19  
`min`, 53, 72  
`mod`, 53, 72  
**modal vs. nonmodal dialog boxes**, 422  
**modulo**, 211  
**mouse, selecting objects with mouse**, 377–380  
**multidimensional arrays**, 31–34, 31, 266–268, 281, 282  
     accessing with a single subscript, 32–34  
     storing in memory, 32  
**multiplication**, 16  
     complex numbers, 243  
     hierarchy of operations, 48–50, 69  
     matrix multiplication, 46, 71  
**naming conventions**  
     arrays, 22  
     case-sensitivity of names, 23, 70  
     data dictionary, 46, 71  
     variables, 16, 23, 70  
**NaN predefined special value**, 37, 72, 142  
`nargchk`, 205, 229  
 `nargin`, 205, 229  
`nargout`, 205, 208, 228, 229  
`ndims`, 268  
     nested loops, 156–157, 172  
     nested statements, `if` construct, 102–104  
     nested structure arrays, 312  
`nnz`, 283, 292, 315  
**noise in values**, 164, 211, 351–358, **358**  
**noisy measurements line plot example**, 163–169, **169**  
`nonzeros`, 283, 292, 315  
**not equal to operator (`~=`)**, 88, 130  
     complex numbers, 244  
**NOT logic operator (`~`)**, 90–92  
`num2str`, 40, 53, 72, 258–259, 283  
**numeric-to-string conversion**, 258–259  
`nzmax`, 283, 292, 315  
**object handles, handle graphics**, 365  
**Ohm's law**, 62  
`ones`, 72, 267  
     initialize variables/arrays, 29  
**operating systems**  
     independence of MATLAB to OS, 2  
     send command to OS, exclamation point character, 15  
**optional arguments**, 204–208  
     magnitude and angle plot using optional arguments, 206  
**OR logic operator (`|`)**, 90–92  
**output argument lists**, 188, 189, 228  
     input/output functions, 319–362  
     varargout function, 304–305, 315  
**output data (See *formatting output data*)**  
**overwriting files**, 347–349, 359  
**padding strings**, 253, 281  
**parent objects, handle graphics**, 364, 388  
**parentheses, use of**, 45, 48–51, 67, 69, 70, 89  
**pass-by-value variable passing**, 194–204, 228  
**passing variables/arguments**, 194–204, 228  
     global memory to share data, 209–217, 228  
**path command**, 16  
**path, search path**, 15–16, 19  
**Path Tool**, 16, 17  
**path2rc command**, 16

pcode, 3, 84, 87–88, 437–438  
 percent sign as comment delimiter, 70  
 permissions, fopen, 325, 327, 359  
**persistent**, 217–222, 228, 229  
 persistent memory, 217–222, 228  
 persistent variables, 399  
 physics applications  
     trajectories, 169–178, **170, 177**  
     velocity of falling object in gravitational field, 169–178,  
         **170, 177**  
**pi** predefined special value, 16–17, 37, 38, 72  
**pie**, 269, 273, 283  
 pie plots, 268–269, **272, 283**  
 platform independence, 2  
**plot**, 72, 249, 251, 276, 283, 365  
**plot3** function, 276–278, **277, 278**  
 plotting, 2, 40, 52–59, 108–125, 130, 268–281  
     adaptive fplot function, 273–274  
     annotating, 123–125, **123, 124, 125**  
     axes command, 108–111, 130, 365, 388, 393  
     axis labels, 54  
     bar function, 269  
     bar plots, 268–269, **271, 281**  
     barh function, 269  
     boldface text, 115, 130  
     carbon-14 dating example, 64–67, **65**  
     cardioid microphone example of polar plot, 115–117, **118**  
     cell arrays, 301, 304  
     color, 56–58  
     compass function, 269  
     compass plots, 268–269, **272, 282**  
     complex numbers, 241, **242, 248–250, 248, 250, 251**  
     contour function, 279, 281  
     contour plots, 278–281  
     escape codes for text, 115, !30  
     explode parameter for pie function, 273  
     exporting in TIFF format, 56  
     ezplot function, 223, 229, 273–274, **274**  
     figure function, 111–112  
     Figure Windows, 9–10, **10**  
     figures, multiple figures, 111–112, 130  
     font selection, 115, 130  
     fplot function, 223, 229, 273–274, 282  
     function function plotting, 223–225, **226**  
     functions for plotting, 273–274  
     grid command, 54, 72  
     grid lines on/off, 54  
     hist function, 215–217, 229, 253–276, 282  
     histogram plot of random-number generator, 215–217,  
         **216**  
     histograms, 274–276, **275, 439–443, 441, 442, 444**  
     hold command, 111–112, **112, 130, 172**  
     italic text, 115, 130  
     legend command, 58  
     legends, 56–58  
     line properties, 56–58, 114, 130  
     linear regression, 163–169, 351–358, **358**  
     LineWidth property, 114  
     logarithmic scales, 58–59  
     loglog function, 59–72  
     marker properties, 56–58, 114, 130  
     maximum power transfer to load example, 62–64, **62, 64**  
     mesh function, 279, 281  
     mesh plots, 278–281, **279**  
     meshgrid function, 281  
     multiple plots on one graph, 56, **57**  
     multiple plots on same axes, 111–112  
     noisy measurements line plot example, 163–169, **169**  
     pie function, 269, 273, 283  
     pie plots, 268–269, **272, 282**  
     plot function, 72, 249, 251, 276  
     plot3 function, 276–278, **277, 278**  
     polar coordinate plot, **242, 242, 251, 251**  
     polar function, 115–117, **118, 130, 381–384**  
     polar plots, 115–117, **118, 381–384**  
     printing a plot, print command, 55–56  
     properties for lines and markers, 114, 130  
     ranges or limits for x and y axis, 108–111, **110, 130**  
     rectangular coordinates plot, 242, **242, 243**  
     saving, 123–125, **123, 124, 125**  
     semilogx function, 58–59  
     size of font (point size), 115, 130  
     stair plots, 268–269, **270, 281**  
     stairs function, 269  
     stem function, 269  
     stem plots, 268–269, **270, 283**  
     stream modifiers for text, 115, 130  
     subplot, 112–113, **113, 130**  
     subplots, 112–113, **113, 130**  
     subscript text, 115, 130  
     superscript text, 115, 130  
     surf function, 279, 283  
     surface plots, 278, 280, 283  
     symbols in plots, 115, 116, 130  
     text in plots, 115, 130  
     thermodynamics: the ideal gas law example, 120–122, **120**  
     three-dimensional plots, 276–281, **277, 278, 279, 280**  
     title command, 54  
     titles, 54–55, **55**  
     units of measure, 61  
     velocity of falling object in gravitational field example,  
         **169–178, 170, 177**  
     xlabel command, 54  
     xy plots, 52, **54–55, 55**  
     ylabel command, 54  
     pointers, 294, 295  
     polar, 115–117, **118, 130, 381–384**  
     polar2rect, 195–198  
     polar plots, 115–117, **118, 242, 242, 250, 250, 381–384**  
     popup menus in GUI, 393, 414, 415

portrait mode, 385  
 positions of objects, handle graphics, 380–384, 384  
 power, maximum power transfer to load example, 62–64, 62, 64  
 preallocated arrays in for loops, 151, 178  
 precedence of operators, 48–50, 69, 89
 

- logic operators, 91
- relational operators, 91

 precision strings, read operations, 330  
 precision strings, write operations, 328–329  
**print**, 55, 72  
 printing
 

- dtiff** option for TIFF file format, 55
- conversion characters for **fprintf** function, 41
- escape characters for **fprintf** function, 41
- format and options, 72
- fprintf**, 40–42, 72, 332–333
- handle graphics, 384–385
- landscape mode, 384–385
- plotting, 52, 54
- portrait mode, 385
- position of printer, 384–385
- print** command, 52–54

 private directories, subdirectories, 228  
 private functions, 227  
 properties for GUI objects, 388–407  
 properties of graphics objects, 363, 388  
 Property Editor (**propedit**) tool, 368–370, 369, 392  
 pseudocoding, 3, 84, 86–87  
 pseudorandom number generators, 212  
 pushbutton in GUI, 393, 394, 394, 395, 397, 398  
  
**quad**, 223, 224  
 quadratic equation using complex numbers, 245–248  
 quadratic equation using if construct, 95–99  
  
 radio buttons in GUI, 393, 394, 411, 413  
 radioactive decay, 64–65  
**rand**, 212–217, 229  
**randn**, 212–217, 229  
 random-number generator using global variables, 212–217, 216  
 ranges or limits for x and y axis, 108–110, 130  
**real**, 245, 283  
 real part of complex number, 241–242, 244, 281  
 rectangular coordinates plot, complex numbers, 242, 242, 243  
 regression analysis, 164, 351  
 relational operators, 87–92
 

- complex numbers, 244
- precedence of operators, 91
- roundoff errors, 89, 130
- string functions, 254

 removing fields from structures, 307  
**rect2polar**, 195–198  
  
 rectangular-to-polar conversion, 195–198  
**return**, 189, 229  
 reusable code, 188  
**rmfield**, 309, 313, 315  
**rmpath** command, 16, 19  
 root objects, handle graphics, 364, 388  
**rose**, 283  
**round**, 53, 72, 215  
 rounding functions, 53  
 roundoff errors, 89, 130  
 row order, 25  
 running-average calculation using persistent memory, 217–222  
 run-time errors, 67  
  
**save**, 42, 72, 319, 321, 323, 324, 358, 359  
 saving files, 42–43, 70  
 saving plots, 123–125, 123, 124, 125  
 scalar operations, 3, 44–48
 

- assigning to a subarray, 36
- relational operators, 88

 scratchpad use of MATLAB, 16–17  
 script files, 6, 189  
 search path, 15–16  
 search/replace characters within a string, 255–257  
 selection region of objects, handle graphics, 377–380  
 selection sorts, 199–204, 200  
 semicolon to suppress automatic echo of values, 26–27, 70  
**semilogx**, 58–59, 72  
**semilogy**, 72  
 sequential programming, 81  
**set**, 370, 372–374, 389  
**setappdata**, 375  
**setfield**, 311–312, 313, 315  
 sharing data, global memory to share data, 209–217, 228  
 shorthand list operator, colon, 70  
 simulations, 211  
 simultaneous equations solved using sparse matrices, 291–294  
**sin**, 9, 10, 53, 72, 245
 

- plotting sin x using low-level graphics commands, 370–372

**size**, 21, 72, 268, 312
 

- initialize variables/arrays, 28, 30

 size of font (point size), plotting, 115, 130  
 slider in GUI, 393, 394, 417, 418
 

- temperature conversion example using slider, 419–422, 420

 sorting data, 199–204, 200  
**spalloc**, 292, 315  
**sparse**, 289–290, 292, 315  
 sparse arrays/matrices, 287–294, 314
 

- find**, 292
- full**, 252, 290, 292, 315
- generating sparse matrices, 290

sparse arrays/matrices (*continued*)  
**issparse**, 292  
**nnz**, 283, 292, 315  
**nonzeros**, 283, 292, 315  
**nzmax**, 283, 292, 315  
**spalloc**, 292  
**speye**, 290, 291, 292  
**spfun**, 292  
**spones**, 292  
**sprand**, 290, 292  
**sprandn**, 290, 292  
**spy**, 292  
**special values, predefined**, 37–38  
**speye**, 290, 291, 292, 315  
**spfun**, 292, 315  
**spones**, 292, 315  
**sprand**, 290, 292, 315  
**sprandn**, 290, 292, 315  
**sprintf**, 259, 260, 315  
**spy**, 292, 315  
**sqrt**, 52, 53, 72, 245  
**sscanf**, 260, 261, 283  
**stair plots**, 268–269, 270, 281  
**stairs**, 269, 288  
**standard deviation**, 2  
**standard menus**, 426  
**static elements of GUI**, 392, 393  
**statistical analysis**  
  using **for loop**, 145–150  
  using **while loop**, 138–143  
**status functions**, 347–358, 359  
**std function**, 138, 143  
**stderr file**, 323  
**stdout file**, 323  
**stem**, 269, 283  
**stem plots**, 268–269, 270, 283  
**stepwise refinement in program design**, 84  
**str function**, 252  
**str2double**, 261  
**str2num**, 53, 72, 261, 283  
**strcat**, 253–254, 261, 283  
**strcmp**, 254, 261–265, 283  
**strcmpi**, 254–255, 261, 283  
**stream modifiers for text, plotting**, 115, 130  
**string function**, 11, 53  
**string functions**, 252–265, 281  
  **base2dec**, 259, 261, 282  
  **bin2dec**, 259, 261, 282  
  blank removal from string, 253  
  blanks, 260, 282  
  categorize characters within a string, 254–255  
  cell arrays using strings, 301–302  
  **cellstr function**, 301  
  **char**, 53, 252, 253, 260, 301, 305  
  comparing strings, substrings, 254–255, 260, 261–265  
  concatenating strings, 253–254  
  conversion functions, 252, 258–259  
  **deblank**, 253, 260, 282  
  **dec2base**, 259, 261, 282  
  **dec2bin**, 259, 261, 282  
  **dec2hex**, 259, 261  
  delimiters, single quotes, 79  
  **double**, 24, 53, 71, 241, 252, 260, 282  
  **equality operator**, 255  
  **eval function**, 222–223, 229, 259–260  
  **findstr function**, 255, 261, 282  
  **fprintf function**, 259  
  **hex2dec**, 259, 261, 282  
  **hex2num**, 259, 261, 282  
  **int2str**, 40, 53, 72, 258–259, 260, 282  
  **ischar**, 92, 130, 252, 260, 282  
  **isletter**, 255–256, 260, 282  
  **isspace**, 255–256, 260, 282  
  lexicographic order of characters in ASCII tables, 260–261  
  **lower**, 257–258, 261, 282  
  **mat2str**, 258, 261, 288  
  **num2str**, 40, 53, 72, 258–259, 261, 283  
  numeric-to-string conversion, 258–259  
  padding strings, 253, 281  
  relational operators, 255  
  search/replace characters within a string, 255–257  
  **sprintf**, 261, 370, 372–376, 388  
  **sscanf**, 260, 261, 283  
  **str function**, 252  
  **str2double**, 261  
  **str2num**, 53, 72, 261, 283  
  **strcat**, 253–254, 261, 283  
  **strcmp**, 254–255, 260, 261, 262–265, 283  
  **strcmpi**, 254–255, 261, 283  
  string-to-numeric conversion, 259–260  
  **strjust**, 261, 283  
  **strmatch**, 256, 261, 283  
  **strncmp**, 254–255, 261, 283  
  **strncmpi**, 254–255, 261, 283  
  **strrep**, 257, 261, 283  
  **strtok**, 257, 261, 283  
  **strvcat**, 254, 256, 261, 283  
  test for equality, 255  
  two-dimensional character arrays, 252–253  
  **upper**, 257–258, 261, 283  
  upper/lowercase conversion, 257–258  
  **whos function**, 252  
  **strjust**, 261, 283  
  **strmatch**, 256, 261, 283  
  **strncmp**, 254–255, 261, 283  
  **strncmpi**, 254–255, 261, 283  
  strongly vs. weakly typed language variables, 24  
  **strrep**, 257, 261, 283  
  **strtok**, 257, 261, 283

**struct.** 283, 308, 313, 375  
**structure arrays**, 306–313, 307, 315  
  adding fields to structures, 308–309  
  assignment statement to create, 306–308  
  data in structure arrays, 310–311  
  `fieldnames`, 313  
  fields, 306, 310–311, 310, 315  
  `getfield`, 311–312, 313  
  nested structure arrays, 312  
  removing fields from structures, 307  
  `rmfield`, 309, 313  
  `setfield`, 311–312, 313  
  `size` function, 312  
  **struct**, 308, 313  
**structured programming**, 87  
**structures**, 287  
`strvcat`, 254, 256, 261, 283  
**subtasks and functions**, 84, 187  
**subarrays**, 34–37  
  assigning a scalar, 36  
  end function, 34–35  
  shape, 35  
  using on an assignment statement, 35  
**subfunctions**, 225–227, 228  
**subplot**, 112–113, 113, 130  
**subplots**, 112–113, 113, 130  
**subscript text, plotting**, 115, 130  
**subscripts**, 70  
**subtraction**, 16, 71  
  complex numbers, 243  
  hierarchy of operations, 48–50, 69  
  matrix subtraction, 46, 71  
**superscript text, plotting**, 115, 130  
**surf**, 279, 281, 289  
**surface plots**, 278, 280, 281  
**switch construct**, 104–106, 130  
  indenting code, program development, 96, 130  
**symbolic debugger tool**, 68–69, 125–128, 126, 127, 128, 130  
**symbols in plots**, 115, 116, 130  
**symbols used in MATLAB**, 70–71  
**syntax errors**, 67  
  
**table of data using fprintf function**, 338–339  
**tan**, 53, 72  
**tempdir**, 324  
**temperature conversion example using slider**, 419–422, 420  
**temperature conversion problem example**, 59–62  
**temprame**, 324  
**testing programming**, 84  
**text field in GUI**, 393, 394, 395, 398, 409, 412  
**text in plots**, 115, 130  
**text objects, handle graphics**, 381  
**Text tool**, 124  
**textread**, 319–321, 359, 360  
**thermodynamics: the ideal gas law example**, 120–122, 120  
**three-dimensional line plots**, 276–281, 277, 278, 279, 280  
**tic**, 152–154, 179  
**TIFF**  
  `-dtiff` option for TIFF file format, 55  
  exporting TIFF file format, 56  
**time required to compile MATLAB code**, 3  
**title**, 73  
  **title** command, 73  
**toc**, 152–154, 179  
**toggle buttons in GUI**, 393, 394, 411, 412  
**Tool bar, Figure** 123, 124  
**tool tips in GUI**, 123, 124, 436–437  
**toolkits/toolboxes**, 1, 2  
**top-down program design**, 81–87, 83, 85, 187  
**transpose operator for shortcut expressions**, 28, 71  
**true/false operators**, 88–89  
**truth tables**, 90–91  
**try block**, 106  
**try/catch construct**, 106–107  
**two-dimensional arrays**, 31, 252–253, 268–276, 281  
**type conversion functions**, 244  
**typographical errors, debugging**, 67–68  
**uicontextmenus**, 364, 388  
**uicontrol**, 364, 381, 388, 392, 393, 406, 408  
**uigetfile dialog boxes**, 424–425, 425  
**uimenu**, 364, 388, 392, 393  
**uisetfile dialog boxes**, 424–425  
**unary logic**, 90  
**unit testing in program development**, 84, 187  
**units of measure**, 61, 70  
  **handle graphics**, 380–384  
**UNIX**, 2  
**upper**, 257–258, 260, 288  
**upper/lowercase conversion**, 257–258  
**user-defined functions**, 187–240  
  actual arguments, 189  
  argument lists, 188, 189  
  arguments, 189  
  beginning of function, **function statement**, 189  
  comment lines, 190  
  data hiding, 188  
  dummy arguments, 189  
  error function, 205, 207, 229  
  execution of functions, 189  
  `feval`, 222–223, 229  
  **function statement**, 189  
  **function call**, 191–193, 192, 193  
  **function functions**, 222–225  
  `fzero` function, 222  
  global memory to share data, 209–217, 228  
  H1 comment lines, 190  
  input argument lists, 188  
  **inputname** function, 205–206  
  internal functions, 225–226  
  **lookfor** command, 14, 72, 190

**user-defined functions (*continued*)**

magnitude and angle plot using optional arguments, 206  
**nargchk** function, 205, 229  
**margin** function, 205, 229  
**nargout** function, 205, 208, 228, 229  
 optional arguments, 204–208  
 output argument lists, 188, 189  
 output arguments, multiple, 208  
 pass-by-value variable passing, 194–204  
 passing variables/arguments, 194–204  
 persistent memory, 217–222  
 persistent statement, 217–222  
**polar2rect**, 195–198  
 preserving data, 217–222  
 private functions, 227  
 random number generator, 211–217, 216  
**rect2polar**, 195–198  
 rectangular-to-polar conversion, 195–198  
 return statements, 189  
 reusable code, 188  
 script files, 189  
 sorting data, 199–204, 200  
 subfunctions, 225–227  
**Warning** function, 205, 208

**validation of input, debugging**, 67–68

**value**, 375  
**varargin**, 302–304, 315  
**varargout**, 304–305, 315  
**variables**, 22  
 arrays, 22  
 arrays to initialize variables, 25  
 assignment statements to initialize variables, 25–27  
 built-in functions to initialize variables, 28–29  
 case-sensitivity of names, 23, 70  
 cell arrays, 294–305  
 char variables, 24  
 colon operator for shortcut expressions, 27–28  
 complex variables, 243  
 data dictionary for variables, 23, 70  
 double variables, 24  
 eye function to initialize variables/arrays, 29–30  
 global memory to share data, 209–217, 228  
 global variables, 210, 228  
 identity matrices to initialize variables/arrays, 29  
 initialization of variables, 24–30  
 input function to initialize variables/arrays, 29  
**isreal** function, 243  
 keyboard input initialization of variables/arrays, 29–30

**length function to initialize variables/arrays**, 29

list of, **whos** command, 7  
 naming variables, 15–16, 23, 70  
**ones** function to initialize variables/arrays, 29  
 pass-by-value variable passing, 194–204, 228  
 path, search path, 15–16  
 persistent variables, 399  
 preserving data between calls to function, 217–222, 228  
 shortcut expression to initialize variables, 27–28  
**size** function to initialize variables/arrays, 21  
 special values, predefined, 37–39  
 strongly vs. weakly typed language variables, 24  
 subarrays, 34–37  
 transpose operator for shortcut expressions, 28, 71  
 types, data types of variables, 24, 241–286  
**varargin**, 302–304, 315  
**varargout**, 304–305, 315  
**zeros** function to initialize variables/arrays, 28, 29  
 vector arrays, 21, 25  
 vectorization and logical arrays, 157–162  
 vectors, for loops vs. vector calculation, 151–154, 178  
 velocity  
 falling object in gravitational field, 169–178, 177, 179  
 trajectory plotting using for loop, 169–178, 177, 179

**waitForbuttonpress**, 377–379, 389

**warning**, 205, 208, 229  
 warning dialog boxes, 423  
**while** loop, 137–143, 178–179  
 break statement, 154–155  
 end statements, 156  
 nested loops, 156–157  
 statistical analysis using while loop, 138–143  
**whos**, 11, 252  
**WindowButtonDownFcn**, 439  
**WindowButtonMotionFcn**, 439  
**WindowButtonUpFcn**, 439  
 windows in MATLAB, 4–10  
 Windows, 2  
**Workspace Browser**, 11–13, 12  
**Workspace**, 10–13, 13  
 wrap around line of code (ellipsis), 5, 71

**xlabel command**, 54**ylabel command**, 54**zeros**, 73, 152

initialize variables/arrays, 28, 29

sparse arrays/sparse matrices, 287–294