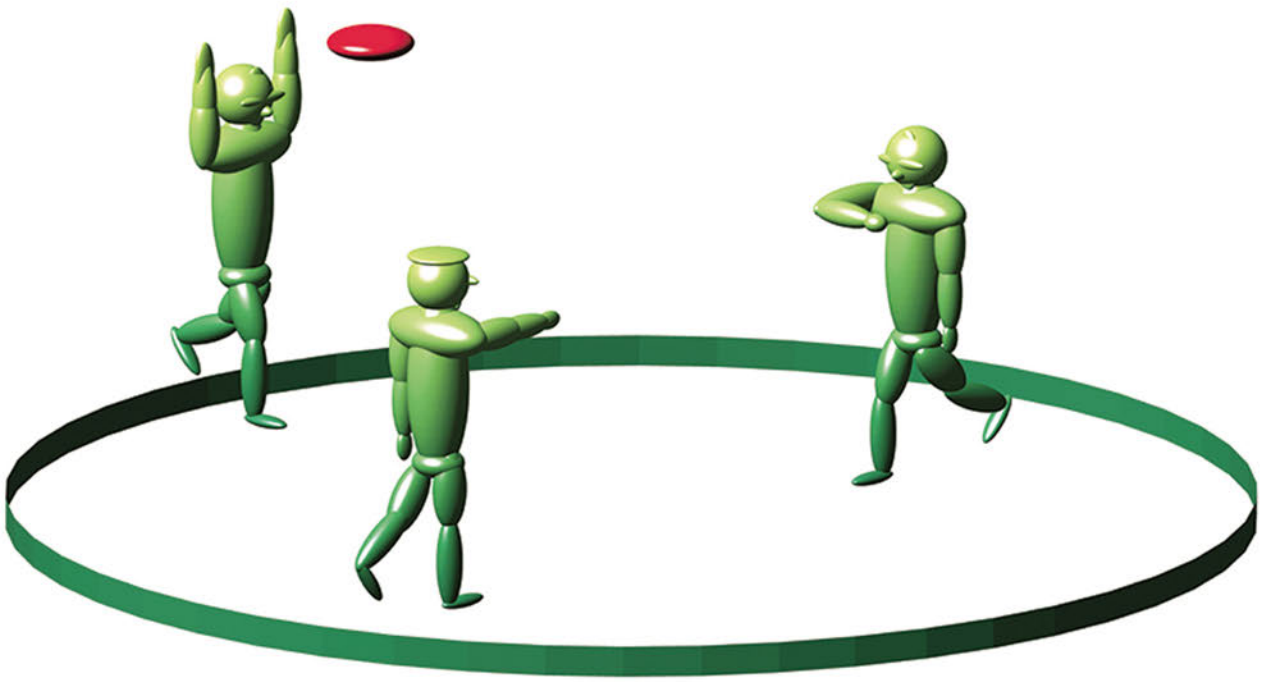
SECOND EDITION

MATLAB for Behavioral Scientists



David A. Rosenbaum,
Jonathan Vaughan, & Brad Wyble

MATLAB

for Behavioral Scientists

Second Edition

Written specifically for those with no prior programming experience and minimal quantitative training, this accessible text walks behavioral science students and researchers through the process of programming using MATLAB. The book explores examples, terms, and programming needs relevant to those in the behavioral sciences, and helps readers perform virtually any computational function in solving their research problems. Principles are illustrated with usable code. Each chapter opens with a list of objectives followed by new commands required to accomplish those goals. The objectives also serve as a reference to help readers easily relocate a section of interest. Sample code and output and chapter problems demonstrate how to write a program and explore a model so readers can see the results using different equations and values. A website provides solutions to selected problems as well as the book's program code output and examples so readers can modify them as needed. The outputs on the website have color, motion, and sound.

Highlights of the new edition follow:

- Updated to reflect changes in the most recent version of MATLAB, including special tricks and new functions.
- More information on debugging and common errors as well as more basic problems in the rudiments of MATLAB to help novices get up and running more quickly.
- A new chapter on Psychtoolbox, a suite of programs specifically geared to behavioral science research.
- A new chapter on Graphical User Interfaces (GUIs) for user-friendly communication.
- Increased emphasis on pre-allocation of memory, recursion, handles, and matrix algebra operators.

Intended as a primary text for MATLAB courses for advanced undergraduate and/or graduate students in experimental and cognitive psychology and/or neuroscience, as well as a supplementary text for labs in data (statistical) analysis, research methods, and

computational modeling (programming), the book also appeals to individual researchers in these disciplines who wish to get up and running in MATLAB.

David A. Rosenbaum is a Distinguished Professor of Psychology at Pennsylvania State University.

Jonathan Vaughan is the James L. Ferguson Professor of Psychology and Neuroscience at Hamilton College.

Brad Wyble is Assistant Professor of Psychology at Pennsylvania State University.

MATLAB

for Behavioral Scientists

Second Edition

David A. Rosenbaum,
Jonathan Vaughan, and Brad Wyble

Routledge
Taylor & Francis Group
NEW YORK AND LONDON

First published 2015
by Routledge
711 Third Avenue, New York, NY 10017

Simultaneously published in the UK
by Routledge
27 Church Road, Hove, East Sussex BN3 2FA

Routledge is an imprint of the Taylor & Francis Group, an informa business

Library of Congress Cataloging-in-Publication Data

Rosenbaum, David A.
MATLAB for behavioral scientists / authored by David A. Rosenbaum, Jonathan Vaughan, and Brad Wyble. — Second edition
pages cm
1. Psychology—Data processing. 2. MATLAB. I. Vaughan, Jonathan (Professor) II. Wyble, Brad.
BF39.5.R67 2014
150.285'536—dc23 2014003997

ISBN 978-0-415-53591-5 (hbk)
ISBN 978-0-415-53594-6 (pbk)
ISBN 978-0-203-11210-6 (ebk)

Typeset in Times and Courier New
by Apex CoVantage, LLC

Dedication Code

```
% Dedication.m
clc
for author = {'Brad' 'Jon' 'David'}
    authorstring = char(author);
    switch authorstring
        case 'Brad'
            Dedication.to = 'Elizabeth Spillman-Wyble';
            Dedication.features = ...
                {'inspiration','storytelling',...
                 'mastery of folklore',...
                 'extraordinary cooking'};
        case 'Jon'
            Dedication.to = 'Virginia Vaughan';
            Dedication.features = ...
               {'intelligence','strength of character',...
                'unfailing support','generosity'};
        case 'David'
            Dedication.to = 'Judith Kroll';
            Dedication.features = ...
                {'brilliance', 'bravery', 'beauty'};
    end
    fprintf('%s dedicates this work to %s',...
          authorstring,Dedication.to);
    fprintf(' in grateful recognition of her ');
    for featurecount = 1:length(Dedication.features)-1
        fprintf('%s, ',...
           Dedication.features{featurecount});
    end
    fprintf('and %s.\n\n',Dedication.features{end})
end
commandwindow
```

Dedication Output

```
Brad dedicates this work to Elizabeth Spillman-Wyble in
grateful recognition of her inspiration, storytelling,
mastery of folklore, and extraordinary cooking.

Jon dedicates this work to Virginia Vaughan in grateful
recognition of her intelligence, strength of character,
unfailing support, and generosity.

David dedicates this work to Judith Kroll in grateful
recognition of her brilliance, bravery, and beauty.
```

This page intentionally left blank

Contents

	Preface	ix
	Acknowledgements	xii
	About the Authors	xiii
1	**Introduction**	1
2	**Interacting With MATLAB**	22
3	**Matrices**	38
4	**Calculations**	62
5	**Contingencies**	99
6	**Input-Output**	128
7	**Data Types**	156
8	**Modules and Functions**	182
9	**Plots**	208
10	**Lines, Shapes, and Images**	248
11	**Animation and Sound**	287
12	**Enhanced User Interaction**	304
13	**Psychtoolbox**	323
14	**Debugging**	355
15	**Going On**	370

References 375

Commands Index 377

Name Index 381

Subject Index 382

Preface

The first edition of MATLAB for Behavioral Scientists (published in 2007) was the result of a rebellious thought. The prevailing view before then was that most behavioral scientists shouldn't or couldn't write their own computer programs. This irked the first author, who decided to pursue the notion that all behavioral scientists, including students in the relevant fields (psychology, cognitive and affective neuroscience, economics, and so on), could and should learn to program for themselves.

Behavioral scientists need to be able to program as much as scientists in other fields. They need to be able to program to do whatever they want, computationally speaking, without having to rely on the kindness of strangers or the largesse of granting agencies to pay others to program for them.

To give some examples, a behavioral scientist—a behavioral economist, say—wishing to model decision making should be able to roll up her sleeves and graph data showing observed and expected data in the way she prefers. A personality psychologist interested in designing a new questionnaire requiring a special computer interface should be able to pursue that aim. A psychotherapist wanting to model changing relations between members of a family should be able to characterize that process with custom-made animations that show network links with dynamically changing thicknesses and colors, growing and shrinking over time, if that's what she wants. A cognitive psychologist interested in setting up and conducting behavioral experiments should be able to create any kind of stimuli and response recording capabilities he or she cares to, not being limited by what's possible with off-the-shelf commercial products.

This book is meant to help behavioral scientists (and especially students entering this field) to do these things. The authors of this book assume you have no prior familiarity with computer programming, and we assume you have no knowledge of mathematics beyond what is generally learned in high school. The text is meant to be as friendly and encouraging as possible. Our aim is to draw you in and help you feel comfortable within a domain that may at first seem foreign and maybe even scary.

Programming can be humbling. If you set out to learn to program, you should prepare yourself emotionally as well as intellectually for what will happen because you will be dealing with an unfeeling machine. It takes a tough hide to believe you have a program that does what you want, only to discover that the program doesn't run, generates unexpected results, or produces outputs that seem reasonable at first but then turn out to be wrong. Everyone who has programmed has gone through this, including the authors of this book, so don't feel like you need to be able to program perfectly. No one does!

Programming needn't be unpleasant, however. The attitude to have is to keep an open mind about the value of mistakes. If you treat errors as windows for improvement, you will learn a lot. Availing yourself of that learning, when you see a program work and especially

when it does something that, to your knowledge, has not been done before, can let you feel rightly proud of your achievement.

There are many computer programming languages. Why is this book about MATLAB? MATLAB (short for Matrix Laboratory), is a commercial product of a company called The MathWorks (Natick, Massachusetts), for which we authors do not work and have no commercial connection. The following, therefore, can be taken as our honest opinion of their product: MATLAB is a simple yet powerful language for computer programming. It has an active community of users, engaged in many branches of science and engineering. One of MATLAB's most attractive features is that it offers high-level commands for performing calculations with large as well as small data sets and for generating publication-quality graphics. Another attraction of MATLAB is that it allows for the presentation of stimuli and the collection of responses with precise timing. Yet another attraction is that MATLAB is platform-independent. It runs on PCs, Macs, and Linux machines. For these and other reasons, MATLAB is a very good language for behavioral scientists. A growing number of behavioral scientists, along with neuroscientists, engineers, and investigators in other disciplines, have therefore chosen to learn MATLAB. Owing to the health and vitality of the MATLAB programming community, it is likely that more and more people will want to learn MATLAB in the future. You will be part of that active community if you choose to dive into the material provided here.

How did it come to pass that there is a second edition of this book? As is always true of a second edition, its predecessor was successful enough to keep the work alive, but changes in the field suggested a face-lift was needed. Among the needed changes was the appearance of other MATLAB books for psychologists and neuroscientists (Fine & Boynton, 2013; Madan, 2014; Wallisch et al., 2009), which are welcome additions, though they are different in style, tone, level of coverage, and organization from the first edition of this book (but not so perfect, in our view, that they obviate this second edition).

As the author of the first edition (Rosenbaum, 2007) contemplated the second edition, he realized that the process of revising and updating the book would benefit from the involvement of his long-time friend and collaborator, Jonathan Vaughan, the James L. Ferguson Professor of Psychology and Neuroscience at Hamilton College. Jon has decades of experience with computer programming. He has served as the editor of *Behavior Research Methods, Instruments, & Computers*, a peer-reviewed publication of the Psychonomic Society. The first author basically learned MATLAB from Jon. He continued to learn from Jon in preparing this second edition.

When Jon agreed to join in, he and David began to map out the ways the second edition would differ from the first. Among the things they agreed to were the following: (1) All known errors in the first edition would be corrected; (2) more would be said about debugging; (3) more problems would be given, including problems that would help students confront very basic issues in the rudiments of MATLAB; (4) solutions to selected problems would appear with downloadable code on the book's new website (www.routledge.com/9780415535946) rather than in the back of the book to allow for more extensive code, updating of the programs if necessary, and addition of new programs as needs and curiosities arose; (5) there would be a tutorial on designing Graphical User Interfaces, or GUIs, which enable a user to interact with a program using graphics to run experiments within MATLAB; (6) there would be a tutorial in designing experiments using Psychtoolbox, a freely available MATLAB toolbox that is specifically geared to behavioral science

research; and (7) special tricks and new functions, developed or discovered since 2007, would be featured, including several developed by the authors to solve sometimes thorny problems that arise in data collection and data presentation.

In preparing the second edition, Jon and David made these changes while retaining the main organization of the book's first edition. As before, readers are ushered to the material slowly and in as a welcoming a way as possible, with more specialized topics coming as the chapters continue. Also as in the first edition, there is continued use of a style that worked well before—introducing a new problem or challenge, presenting associated code, and then presenting the output. In addition, as in the first edition, each chapter starts with a list of things to be done followed by commands that get them done. These start-of-chapter lists let you use the book as a reference once you understand the basics of MATLAB. Thus, after you have worked your way through the book, you will be able to turn to a section and quickly get the detailed information you need to complete the programming task you are undertaking. All the commands are listed as well in a single Command Index near the back of the book, another innovation of the second edition relative to the first.

Another way we have made the text as user-friendly as possible is to update the website for this book. On this site, you will be able to find and copy the programs and program outputs in this volume. The outputs on the website have color, motion, and sound, whereas those modalities are absent from the printed edition.

As shown in the list of new features, the second edition has a chapter on Psychtoolbox. This is a free, popular, MATLAB-based toolbox for running behavioral experiments. Neither Jon nor David had used Psychtoolbox before, simply because it wasn't essential for their work. It happened, however, that Brad Wyble, a newly hired faculty member in the Penn State Psychology Department (the department where David works), had extensive experience with Psychtoolbox. Jon and David invited Brad to prepare a chapter for the book on Psychtoolbox, and, to their great satisfaction, he agreed.

Brad's area of expertise is vision, the domain of behavioral science in which, it happens, Psychtoolbox is used the most. With his extensive background in computer science—Brad was a computer science major as an undergrad and did research in computer science labs after completing his PhD at Harvard—he proved to be a wonderful addition to the team. His involvement in the book was limited to the one chapter he prepared, plus his review of this Preface, as per the agreement he made with Jon and David. Any errors in the book, then, outside of the Psychtoolbox chapter and the Preface are not due to Brad. By the same token, any errors in the Psychtoolbox chapter and in the Preface are as much Jon's and David's fault as they are, or might be, Brad's. In general, any mistakes rest squarely with Jon and David, or most especially David, who, after having had several years to mull over the transition from the first edition to the second, should have gotten things right by now!

The last thing we want to say in this preface echoes what we say in the main text about responsiveness to feedback. It is fine to be open to feedback from a *computer*, as we urge you to be, but it is also good to be open to feedback from *people*. If you spot something that you think could be better, please let us know. If you have suggestions for things to include in a future edition, give us those suggestions. If you want help with your programming, we cannot serve as consultants to you. We appreciate understanding on that last point. To get in touch with us, you can use one or more of our e-mail addresses: dar12@psu.edu, jaughan@hamilton.edu, or bpw10@psu.edu. We hope you will find this book useful.

Acknowledgements

There are others who deserve praise and thanks for their contributions, direct and indirect. First, we express our appreciation to the students who took the MATLAB courses offered by David at Penn State and by Jon at Hamilton, and who also were exposed to MATLAB by Brad. Teaching these students helped us see which programming concepts are transparent and which are less so.

Several students in our classes and in our labs played especially important roles in helping us hone our MATLAB instruction. We thank Penn State students Max Bay, Katie Chapman, Chase Coelho, Rajal Cohen, Samantha Debes, Jeff Eder, Jason Gullifer, Lanyun Gong, Derek Henig, Joe Santamaria, Garrett Swan, Matt Walsh, and Robrecht van der Wel. We thank Hamilton College students Deborah Barany, Julia Brandt, Hallie Brown, Drew Linsley, Ramya Ramnath, and Anthony Sali. Others who provided valuable feedback are Debra Boutin, Gillian Dale, Mike Frederick, Michael Romano, and Doug Weldon. Mario Kleiner provided helpful information about Psychtoolbox.

We also wish to thank the reviewers who provided feedback on the revision plan: Simon Farrell, University of Bristol, UK; Alen Hajnal, University of Southern Mississippi, USA; and an anonymous reviewer.

This book was completed while the first author was on sabbatical in Los Angeles, at UCLA and USC, where he was supported in part by a fellowship from the John Simon Guggenheim Memorial Foundation. Brad Wyble's research was supported at the time of this writing by NSF grant BCS #1331073. Jon's research has been supported by grants from the National Science Foundation and the National Institutes of Health, as has David's. We all appreciate this support, not to mention the support of the institutions that have paid our salaries.

We also wish to express our thanks to Paul Dukes at Psychology Press/Taylor & Francis, who was instrumental in opening the door for the second edition of the book. Paul called on his colleague, Debra Riegert, to work with us to bring the work to completion. Debra was responsive and helpful at every stage. We appreciate her help as well as the further assistance of Angela Halliday at Routledge/Taylor & Francis, who helped with the book's and website's production.

About the Authors

David A. Rosenbaum is a cognitive psychologist whose main interests are human perception and performance. His main research contribution has been joining cognitive psychology and motor control. Rosenbaum attended public schools in Philadelphia and then attended Swarthmore College (B.A., 1970–1973) and Stanford University (Ph.D., 1973–1977). He worked at Bell Laboratories (1977–1981), Hampshire College (1981–1987), and the University of Massachusetts, Amherst (1987–1994). He has been at Pennsylvania State University since 1994, where he was named Distinguished Professor of Psychology in 2000. Rosenbaum was a recipient of a National Science Foundation Graduate Fellowship (1973–1976), a National Institutes of Health Research Career Development Award (1985–1990), and a National Institutes of Health Research Scientist Development Award (1992–1997). His work been supported by grants from the National Science Foundation (NSF) and the National Institutes of Health, as well as grants from the Dutch, French, and German equivalents of NSF. Rosenbaum is a Fellow of the American Association for the Advancement of Science, the American Psychological Association, the American Psychological Society, and the Society of Experimental Psychologists. He served as Editor of *Journal of Experimental Psychology: Human Perception and Performance* (a publication of the American Psychological Association) from 2000 to 2005. He was awarded a Guggenheim Foundation Fellowship in 2012 for the 2013–2014 academic year. Besides being the author of the first edition of this book, David is the author of a textbook on motor control [Rosenbaum, 2010] and the author of a book applying Darwin's theory of natural selection to cognitive psychology [Rosenbaum, 2013].

Jonathan Vaughan is a broadly trained experimental psychologist (B.A., Swarthmore College, 1962–1966; Ph.D., Brown University, 1966–1970) whose research interests focus on the planning and execution of motor actions, eye movements and attentional processes, human and animal learning, and cognitive neuropsychology. He has taught at Hamilton College since 1971. His work with David Rosenbaum and Ruud G. J. Meulenbroek, initiated under an AREA grant from the NINDS, has produced computational models of reaching, grasping, tapping, and manual circumvention of obstacles. Other research support has come from the NSF and NIMH. Vaughan has published more than 60 journal articles and book chapters, and given more than 100 research presentations, many in collaboration with Hamilton undergraduates. He has contributed in many ways to computer applications in psychological research, including tutorial materials for the use of PsyScope and SPSS. He edited the Psychonomic Society's international quarterly, *Behavior Research Methods, Instruments, and Computers* [1994–2004] and founded the Psychonomic Society's Archive of Norms, Stimuli, and Data, an online repository of computer programs, data, and stimulus norms that has served as an important resource for researchers in the field.

Brad Wyble studies attention, perception, and memory. He attended public schools in Lancaster, Pennsylvania, after which he obtained a B.A. in computer science from Brandeis University (1991–1995) and a Ph.D. in psychology from Harvard University (1996–2003). He was a postdoctoral fellow at the University of Kent in Canterbury, England (2003–2007), University College, London (2007), and MIT (2007–2009). He was subsequently an assistant professor at Syracuse University (2009–2012) and is now an assistant professor at Pennsylvania State University in the Department of Psychology. Wyble was a recipient of a National Science Foundation Graduate Fellowship (1997–2000), he was a Sackler Fellow (2001–2002), and he has been supported by grants from the National Science Foundation, the Office of Naval Research, and the National Institutes of Health. He serves as a consulting editor for the *Journal of Experimental Psychology: Human Perception and Performance*, and as an associate editor for the journal *Frontiers in Cognition.*

1. Introduction

This chapter covers the following topics:

1.1 Getting oriented
1.2 Getting an overview of this book
1.3 Understanding computer architecture
1.4 Programming principles
1.5 Deciding if a program is needed and whether you should write it
1.6 Being as clear as possible about what your program should do
1.7 Working incrementally
1.8 Being open to negative feedback
1.9 Programming with a friend
1.10 Writing concise programs
1.11 Writing clear programs
1.12 Writing correct programs
1.13 Understanding how the chapters of this book are organized
1.14 Using the website associated with this book
1.15 Obtaining and installing MATLAB
1.16 Acknowledging limits

1.1 Getting Oriented

Computers are vital in every branch of science today, and behavioral science is no exception. When behavioral scientists use computers to obtain responses in questionnaires, present visual stimuli, display brain images, generate data graphs, or write manuscripts, their ability to make efficient progress in their research depends largely on their ability to use computers effectively.

Many specialized computer packages let behavioral scientists do their work, and each one takes some time to learn. It is useful to know how to use these specialized packages, but it is also tantalizing to consider the possibility of learning how to program for yourself. The reason is that all specialized computer packages rely on underlying code, and knowing how to generate such code yourself can allow you to be self-sufficient or nearly so in your own research.

Suppose, for example, that you want to develop a mathematical model of some cognitive process. It is convenient to be able to write a program that lets you explore the mathematical model freely, seeing the results obtained with different equations, different parameter values, and so on. Similarly, to analyze data in ways that would be cumbersome with existing spreadsheet applications, it is refreshing to be able to write the analysis program to your own specifications. For example, to view graphs of obtained or theoretical data in a variety of forms, it is useful to be able to generate the graphs quickly and easily, however you please, not just as stipulated by an existing graphics package.

The computer language introduced here, MATLAB, provides you with these capabilities. MATLAB is available from The MathWorks (www.mathworks.com), a company with which

we authors have no affiliation. MATLAB has become popular in several branches of engineering and science, including behavioral science. Nonetheless, to the best of our knowledge, no book has appeared about MATLAB that is written specifically with behavioral scientists in mind. Nor for that matter has a book come out for behavioral scientists about any other general-purpose programming language. The need for such a volume motivated the first edition of this book. Its positive reception encouraged us to revise the text and expand the coverage in this second edition.

Will it be worth your time to read this book? Once you have gone through the text and generated your own MATLAB programs based on the material presented here, you should have enough programming skill to do most of what you need to for your own behavioral research needs. Most importantly, a working knowledge of MATLAB will allow you to perform some analyses that would be tedious, difficult, or impossible otherwise. In addition, you will be able to understand and build upon the work of colleagues who use MATLAB in their work.

You will probably find this book most useful if you use it in two stages. In the first, you will want to go through it, or the parts of it most relevant to your needs, in considerable detail, working problems and developing the hands-on skills that will make you a MATLAB *user*, not just a MATLAB *appreciator*. In the second stage, you will be able to rely on the book as a reference, turning quickly to those sections that provide examples you can adapt for your own programming needs.

To make the book as useful as possible as a reference source, we have designed it so you can get the examples you need quickly and easily. You can do so by turning to the opening page of any chapter and finding there a list of things you may want to do. Beneath that list is a compendium of associated commands. The text itself provides examples you can adapt for your own purposes. You can copy those examples by hand into your own programs, or, to avoid typographical errors, you can copy and paste them from the website associated with this book (www.routledge.com/9780415535946), where the programs and their outputs are available, along with the solution to selected problems. Finally, the list of commands introduced in each chapter is listed as well in the Commands Index.

1.2 Getting an Overview of This Book

Acquiring a new skill such as computer programming can be daunting, so it helps to have an overview of what you can expect as you proceed. Here, then, is a roadmap of the contents of this book. Besides signposts, we also provide brief explanations of the goals of each chapter.

1. **Introduction.** By reading the present chapter, you will learn more than you may already know about how computers work and what computer programming languages do. You will also learn about the ways you should approach computer programming. Finally, by reading this chapter, you will understand how this book is organized. That information can help you use the book efficiently.
2. **Interacting With MATLAB.** By delving into the second chapter, you will learn how to activate MATLAB's windows in order to open, edit, save, and run MATLAB programs.

3. **Matrices.** By studying the third chapter, you will learn how MATLAB enables you to store and access data. Briefly, MATLAB lets you store data in matrices consisting of one or more rows and one or more columns. Matrices are so fundamental to MATLAB that the name of the language is actually short for "Matrix Laboratory." You can think of a two-dimensional matrix (one having both rows and columns) as analogous to the rows and columns in a spreadsheet.
4. **Calculations.** Computers are good at calculating. Chapter 4 shows how to get your computer to carry out calculations with MATLAB.
5. **Contingencies.** One of the main purposes of a computer program is to perform different actions depending on existing conditions. The logic of a program involves not only calculations but also decision making, such as evaluating variables differently (or not evaluating them at all), depending on their values.
6. **Input-Output.** Chapter 6 shows you how to control your computer's interactions with the external world. By studying Chapter 6, you will be able to design programs that let you create dialogs with users, including participants in behavioral studies, and to read and write data to and from external files.
7. **Data Types.** One of the biggest challenges in using computers in research is determining how best to represent the data you are working with. It is important to understand what data types are available in MATLAB so you can choose and manipulate your data types accordingly.
8. **Modules and Functions.** Simple programs are usually easy to understand, but when they become more complex it often helps to deal with them in chunks. Some higher level structure is often helpful. Chapter 8 shows you how to write programs that have this property. Those programs often have stand-alone modules and functions. Such modules and functions can be called by a variety of programs. Using modules and functions can help you approach programming from a top-down rather than a bottom-up perspective. Modules and functions can also facilitate the reuse of programs in the future.
9. **Plots.** The ability to generate and manipulate complex graphics for the exploration and presentation of data is widely regarded as one of the special strengths of MATLAB. Chapter 9 exposes you to those strengths by showing you how to make line graphs, bar graphs, and other types of graphs that are suitable for professional presentations and publications.
10. **Lines, Shapes, and Images.** Here you will learn how to create, import, or reshape lines, shapes, and other images that can either stand alone or be included in graphs. Chapter 10 will also show you how to generate three-dimensional graphs.
11. **Animation and Sound.** Chapter 11 builds on the static graphics of the tenth chapter to manipulate figures using simple animation techniques, generate movies, and generate auditory stimuli.
12. **Enhanced User Interaction.** When you think of a typical computer application, what comes to mind is how the program interacts with the user, typically through graphics, the keyboard, the mouse, or touchscreen. Chapter 12 introduces you to some of the tools available in MATLAB for user interactions.

13. **Psychtoolbox.** For real-time work, there are some features that MATLAB ordinarily lacks that are needed for precise and flexible stimulus presentation and data acquisition. Chapter 13 describes a sophisticated extension to MATLAB, *Psychtoolbox*, which adds features to facilitate research using MATLAB, especially in vision research. This chapter also touches on related packages of interest to behavioral scientists in related areas.
14. **Debugging.** Programs often have bugs because, for better or worse, programming is often a trial-and-error process. While it is hard to know in advance how to address every possible bug, it is possible, based on the authors' many goofs of their own, to convey advice about debugging techniques which you may find useful. These are offered in Chapter 13 . . . oops, Chapter 14 (☺).
15. **Going On.** Chapter 15, the last chapter of the book, provides pointers for going further with MATLAB. This chapter also directs you to other resources you may want to draw on.

A lot of material will be covered in this book. Do you need to go through all of it? If you have no need to play sounds, show animations, or generate three-dimensional graphics, you may safely ignore large parts of Chapters 9 through 13, though leafing through these chapters may help you overcome any prejudices or fears you might have regarding these topics. At the same time, there are chapters you cannot avoid, at least if you don't want to emerge from this book the way Woody Allen emerged from his speed-reading of Tolstoy's epic novel, *War and Peace*. "It was about Russia" was all he could recall.

The truly essential chapters of this book are Chapters 2 through 5. You cannot go on to the later chapters and expect to have control of your programs if you don't have command of the material in Chapters 2 through 5, and the only way to gain that command is to work your way through the examples and exercises slowly and carefully. We promise that even if you think you understand how things work, the only way to be sure is to try them out and expose yourself to the feedback you will receive.

As you gain expertise, Chapters 6 through 8 will allow you to write more sophisticated code. Chapters 9 through 13 will provide you with specialized tools for your work and enjoyment. And Chapter 14, as already mentioned, will suggest ways to help you debug efficiently.

A word of advice: Don't hesitate to revisit earlier sections of the book as you move through it. No one remembers perfectly, and no one understands material quite as fully the first time as in revisits. Your understanding of what may seem very obscure the first time through will be enhanced by the top-down knowledge and context you will acquire touring later material.

1.3 Understanding Computer Architecture

As a first step toward learning to program, it can be helpful to know a bit about computer architecture. Knowing about the main components of a computer can help you understand what features of the environment your program must deal with.

All working computers have five basic elements. As shown in Figure 1.3.1, these are (1) input devices (not only the conventional keyboards and mice, but also the microphones, response buttons, and video and voltage recorders that are useful in the laboratory); (2) output devices (screens, printers, loudspeakers, etc.); (3) storage devices (hard disks, thumb drives, DVDs, the "Cloud," etc.); (4) primary memory; and (5) the central processing unit. The first three components should need no further explanation. The last two components merit more discussion.

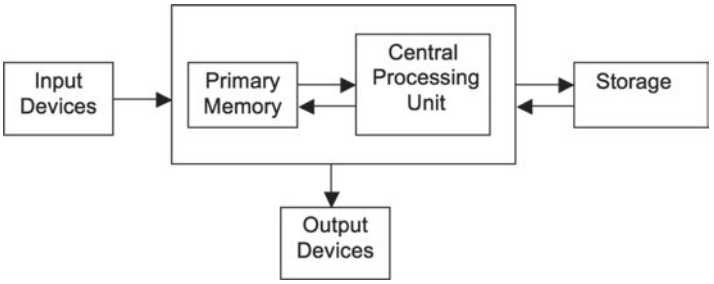


Figure 1.3.1

Primary memory (item 4 on the list) is like human or animal working memory. Its contents are currently active information. The amount of information that can be kept in this active state is limited, both in biological agents (humans and animals) and in computers. The amount of information a computer can maintain in primary memory is hardware dependent.

Because the capacity of primary memory is limited, it is important to be mindful of the amount of information a computer can keep active at once. The amount of information made active by a program, such as one written in MATLAB, depends on the number of variables that are declared and the number of bits (the number of 1s and 0s) required to represent each variable.

Essentially, there are three ways of using primary memory efficiently: (1) defining just the variables that are needed; (2) clearing variables once they are no longer needed; and (3) defining the types of the variables so the amount of memory initially reserved for them is large enough but not substantially larger than needed. We will return to these topics in Chapter 7 ("Data Types").

Returning to the components of computer architecture, the fifth component is the central processing unit. This is the part of the computer that executes instructions. For present purposes, the central processing unit, or CPU, can be likened to consciousness, for which, it is said, only one thought can exist at a time (James, 1890). The same can be said of a computer's CPU. It can handle only one instruction at a time, at least in a conventional digital computer. Handling just one instruction at a time is called *serial* processing. Handling more than one instruction at a time is called *parallel* processing.

Serial processing can occur at high rates in modern computers. For example, the computer on which this text was prepared (a Dell laptop) runs at 2 gigahertz (2 billion cycles per second).

Regardless of the speed at which a CPU runs, serial processing imposes constraints on the kinds of programs you can run, and therefore write, in MATLAB. Suppose, for example, that you want to find the largest value among a set of numbers. Parallel processing is a natural way to solve this problem. If the values are plotted as in Figure 1.3.2, for example, a brief glance at the bars lets you pick the biggest one. The tallest bar seems to jump out at you. Once it does, you can look down to find the associated element (element 3 in this case), or you can look to the left to find the largest value (39 in this case).

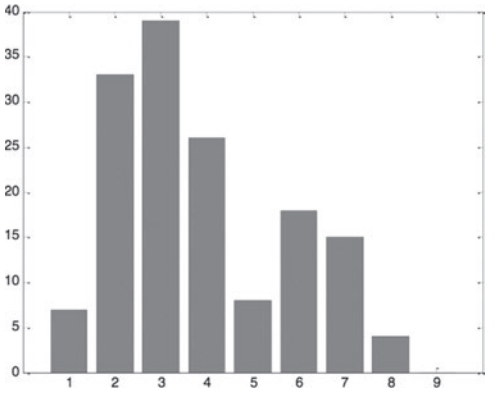


Figure 1.3.2

You might object that parallel evaluation of the heights of all the bars in this case is not actually possible, and even it were for this particular figure, it wouldn't be for all other sets of numbers, such as those whose largest values are similar. You might also say that the method outlined above is not a truly parallel process because distinct stages are associated with looking down the tallest bar or looking sidewise from the top of the tallest bar. These objections are well taken, especially considering that serial processing is inescapable in MATLAB, at least in a program that uses MATLAB in its usual configuration. To sort values or do anything else in MATLAB, everything must be done one step at a time (serially). Knowing this can help you approach the task of programming. (Many recent computers have multiple processors, or *cores*, that make parallel computing possible. Advanced users can take advantage of these to speed complex computations by having two or more cores compute different things at once, using additional tools available from The MathWorks. If you are beginning your programming skills with this book, you can safely save parallel programming for another time.)

1.4 Programming Principles

How should you approach the task of programming? We have come to believe in the following principles:

- Decide if a program is actually needed and, if so, whether you should write it.
- Be as clear as possible about what your program should do.

- Work incrementally.
- Be open to negative feedback.
- Program with a friend.
- Write concise programs.
- Write clear programs.
- Write correct programs.

Consider each of these principles in turn.

1.5 Deciding If a Program Is Needed and Whether You Should Write It

The first principle is less obvious than you might suppose. Consider the problem discussed above (finding the largest of a set of values). The numbers corresponding to the bars in Figure 1.3.2 are as follows:

```
7  33  39  26  8  18  15  4  0
```

Do you need a computer program to find the largest of these values? Obviously not. You know that the largest of these numbers is 39 and that this largest number occupies the third slot in the series. If you only had to find the largest value in this particular array, you would be foolish to write a program for this task, except as an exercise. On the other hand, if you were quite sure you would often need to find the largest number in each of a large number of arrays of unpredictable sizes, writing a program would make more sense. A program is useful, then, for performing a well-defined task that would be too taxing to perform by hand.

The second part of the first principle, whether you should write the program yourself, also deserves comment. If you decide you need a program, it may or may not make sense for you to write the program yourself. Why should you write a program for a task if someone else has done so before?

Our answer to this question is analogous to the answer a math teacher might give to a rebellious student: "Why should I prove this theorem if it's been proved before?" "Practice makes perfect," the teacher may reply. He or she may go on: "Even if true perfection is beyond your reach, practice will increase the chance of your proving something new yourself."

Our view of programming is the same. You might be able to locate programs that already do things you need to, and it may make sense for you to use those programs, especially for problems that seem very complicated or that are beyond your technical ability. But the more practice you get programming, the more likely it will be that you will be able to generate programs that either solve new problems or solve old problems in new ways. Don't be discouraged if it takes an hour or more to get your first "real" program up and running, even if you might have done the same computation by hand in a minute or less. As you develop

programming expertise, you will become more efficient and productive, and you'll be able to apply your new skills to other problems.

1.6 Being as Clear as Possible About What Your Program Should Do

If you decide that you need a program and that you should write it yourself, you will need to be as clear as possible about what your program should do. This is easier said than done. Thinking through the workings of a program can be one of the hardest aspects of programming, even harder in some cases than getting the syntax right.

Return to the problem of finding the largest value in an array. It turns out that MATLAB provides a program (or more precisely, a *function*), called `max`, that lets you find the maximum of a set of values (see Chapter 4). You can use this function to get the largest value in a matrix without having to reinvent the function yourself. Nevertheless, it is worth thinking through the way you would identify the largest value in an array. Working through this example—however simple it may seem—will help you begin to "think programmatically."

To think through what a program must do to find the largest value in an array of numbers, imagine that you have a row of numbers like the one above, but you can only see one of the numbers at a time—say, by sliding the hole in a card across the row. Under this circumstance, you can determine the largest value by finding the largest value *so far*. If you were actually doing this, you'd first place the hole in the card over the first number, which is 7. Then, you'd remember that 7 is the largest value you've seen, and move the card to reveal the 33. Thirty-three is larger than 7, so now you'd note that 33 is the largest number you've seen, and you'd move the card again. After seeing 39, you would revise the largest number seen to that value. Continuing and not encountering any number larger than 39 for the rest of the series, that would be the number you'd report.

Now translate this algorithm into a program. Assign some very small value to a variable called, for instance, `Largest_Value_So_Far`. Then, proceeding from left to right, every time you encounter a value larger than `Largest_Value_So_Far,` reassign that new value to `Largest_Value_So_Far`. After you have evaluated the last item on the list, `Largest_Value_So_Far` will be the largest of all the values.

Here is a flow chart for the procedure, along with some other items you'd need to get the job done. One of these other items is telling the program how many values there are in the list. We give the list the name `V`. There are `n = 9` values in `V`.

Another thing that needs to be done is initializing `Largest_Value_So_Far` to an extremely small value, namely, minus infinity (which can be expressed in MATLAB as `-inf`). We do this because whenever a new number is tested, it must be compared to some prior value. Starting with `-inf` ensures that the first value will be called the largest provided it is larger than `-inf`. It may stay that way if no larger value comes along.

The third thing that needs to be done is providing an index, `i`, for each successively encountered value in `V`. An index for a value is the position of the value in the matrix. For the first item, `i = 1`, for the second item, `i = 2`, and so forth. Initially, `i` is set to `0`. Each time a new number is compared to `Largest_Value_So_Far,` the variable `i` is incremented by `1,` until `i` is greater than `n`. The `i`-th value of `V`, denoted `V(i)`, is assigned to `Largest_Value_So_Far` if `V(i)` is larger than the current value of

`Largest_Value_So_Far`. When `i` is larger than `n`, the program stops and the value of `Largest_Value_So_Far` is printed out.

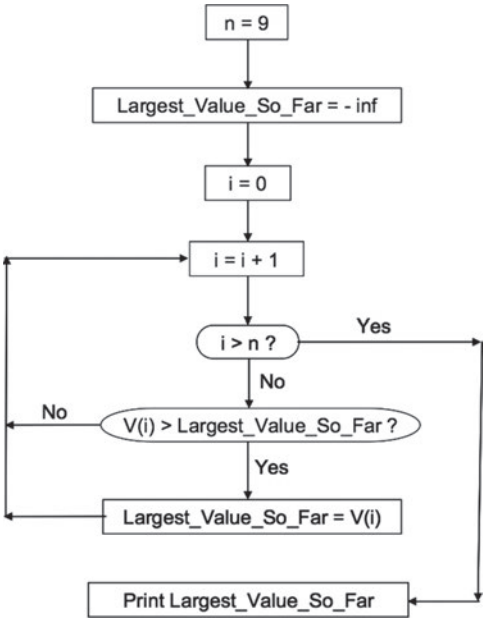


Figure 1.6.1

A flowchart like this can serve as the conceptual foundation for the code needed to get a computer to find the largest value in an array. You don't *have* to draw a flowchart before you write MATLAB code, however. Some people only imagine flowcharts or the steps corresponding to them. Drawing flowcharts in your head obviously gets easier as you get more practice with programming. Early in practice, however, it is advisable to sketch the steps your programs will follow.

How do you come up with a flowchart or its corresponding steps in the first place? The honest answer is that no one knows. Anyone who could give the answer would, in effect, know how thoughts originate, and no one at this time has a clue about that. If you solve this problem, a Nobel Prize awaits you.

You can, however, consider some practical advice about how to come up with the procedures for computer programs. One suggestion is to talk out loud as you imagine yourself doing the task you wish to program, step by step, much as we did with the imaginary card above. Talking out loud may enable you to make explicit whatever implicit knowledge you bring to bear as you do the task, as if you were explaining the task to a friend. Hearing your own words will also help you identify those things you're not clear about. If you hear yourself say, "OK, next I'll somehow figure out which of the values might be OK based on some criterion I can't quite articulate but I have a vague feeling about," then you're not quite ready to write all the code you need. Ultimately, you'll need to be completely explicit about the instructions your programs contain. Relying on a miracle just won't work, and the reason, just to be explicit, is that computers, for all their speed, are ignorant and inflexible. They do exactly and only what they're told to do.

This is one way in which programming is very different from other forms of communication. When you speak to other people, you assume—usually correctly—that they have some knowledge that lets them fill in missing information. Not so with computers, or at least conventional computers given stand-alone programs. Writing successful computer programs requires a degree of explicitness that is unparalleled in other aspects of human experience. This is one reason why learning to write computer programs can be challenging. On the other hand, being explicit to the point that a computer can carry out instructions may sometimes carry over well to other things you do, like writing papers or reaching agreements with others about who will do what in connection with some project.

1.7 Working Incrementally

Another challenge of programming is translating your procedural ideas into language the computer can understand. Here it is useful to work incrementally. By this we mean you should build your program a little at a time, making sure each part works before you go on to another part that depends on what you've just written. You should build your program the way a reliable contractor builds a house, by making sure the foundation is solid before the basement is added, by making sure the basement is solid before the first floor is added, and so on. During program development, you will often find it useful to generate intermediate output to verify that each step works as expected. You may later inhibit that output when the program is completed and is no longer needed. Think of this incremental programming process as the digital equivalent of the ancient woodworking adage (attributed to John Florio, 1591), *Alwaies measure manie, before you cut anie* ("Measure twice, cut once.").

When you're reasonably sure your program works, and before you add another component or make other significant changes, save the program with a file name unique to the last working version. The moment you prepare to make changes to the program, save the file with a new name or version number before putting in any changes. Follow the American folk adage, "If it ain't broke, don't fix it." Too often, attempts to further develop a program *will*, in fact, break it, or otherwise reveal some weakness in it, and you might want to go back to an earlier version. You'll be glad you have one!

Remember, too, that computer storage is cheap. There is no harm in having a folder full of documents called `Max_Program_01.m`, `Max_Program_02.m`, `Max_Program_03.m`, and so on. It may be that the version you'll use for actual work is `Max_Program_101.m`. There is nothing wrong with such a high number. You can tuck away the earlier versions in a sub-folder until you're sure you'll never need to look back. Having sequential versions of a program in development makes it easy to compare the changes. In this connection, it is useful to note that MATLAB has a comparison tool that highlights all differences between two versions of a program, similar to "track changes" in Microsoft Word.

1.8 Being Open to Negative Feedback

How can you tell if your program works? As you consider this question, one attitude should rule over all others: *Be open to negative feedback*. If you treat negative feedback as a help rather than a hindrance, you will become a better, and certainly happier, programmer than if you treat negative feedback in a negative way.

The research of psychologists Carol Dweck and Janine Bempechat (1980) is relevant in this regard. Dweck and Bempechat distinguished between people who take negative feedback

as signs of their lack of talent (*entity* learners) and people who treat negative feedback as cues for ways to improve their performance (*incremental* learners). It is important while programming to have the attitude of an incremental learner rather than an entity learner. You will learn more if you take negative feedback constructively than if you read such feedback as a sign that you weren't "cut out" for programming. MATLAB will not give you an error message that says

```
??? You don't deserve oxygen!
```

A more likely message is something prosaic like

```
??? Subscript indices must either be real positive integers
or logicals.
```

You might get an error message like the latter one in response to code such as

```
Reaction_Time_For_Trial(0) = 680;
```

All you have to do here is appreciate that it makes no sense to have the zero-th element of an array. An array can have a first element, a second element, a third element, and so on, but it can't have an element numbered zero. Whether the 0 was entered in the code based on a misunderstanding or simply as a typo, you can correct the error without indicting your genes. If when you typed 0, you were referring to the first trial, you can replace the 0 with a 1 and all will be fine:

```
Reaction_Time_For_Trial(1) = 680;
```

One reason for saying these things is that it bears remembering that the error messages you receive while programming come from a machine, not from a person who knows what you are trying to say. When you receive an error message, it will help you to take the message as a piece of advice. Over time, you will get fewer error messages concerning low-level aspects of coding (e.g., when you have an unequal number of opening and closing parentheses in a line of code), and you will learn what the error messages mean. More about error messages and debugging (correcting your programs) will come later in the text.

Over time you will also learn to guard against disaster when you program. We encourage you to do so by writing programs that are resilient rather than brittle. If you write a program that crashes or yields crazy results when it gets input of a different sort than what you anticipated, your program won't be of much good. For example, if you write a program that is used to collect questionnaire data, and a participant types in an age of -83, that could wreak havoc with subsequent data analyses. It doesn't matter why the participant put a minus sign in front of his or her age (if he or she is actually 83). Perhaps the participant thought this might help you see the number more clearly, perhaps it was just a typo, or perhaps the participant thought he or she was being cute. The point is that you must anticipate such eventualities. All sorts of things can go wrong when a program is being run. A good programmer guards against as such eventualities. In this sense, being open to negative feedback means more than not letting your feelings be hurt when the computer beeps because you left out a punctuation mark or because you mistyped the name of a function. Responding constructively to negative feedback also means being open to all sorts of

unwanted events and building safeguards into your programs so you're not confronted with bogus results later on.

The final sense in which it is important to be open to negative results is that you should not be complacent when your program runs and gives you results, especially beautiful ones, that cause you to blush with quixotic pride. Here is an example.

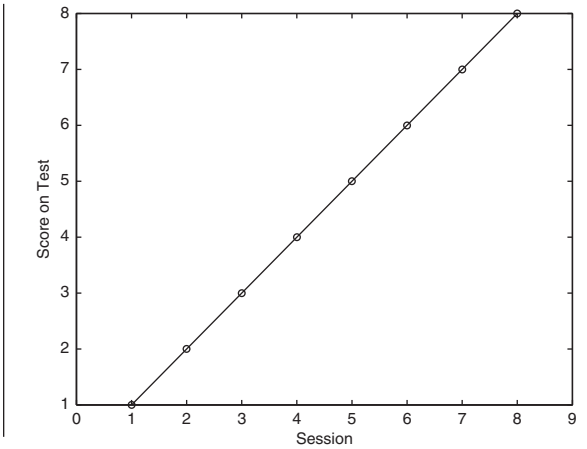
The numbers 1 through 8 are assigned to a matrix called `x`. These numbers are session numbers, which comprise the independent variable of a fictional behavioral science study. The dependent variable is `y`, a set of fictional scores. After `x` and `y` have been defined, a command is used to `plot` the data. This command ends with a special instruction, in quotes, to plot the data in black (`k`), using circles (`o`), with connecting lines (`-`). Within the plot command, you accidentally (or on purpose for this example) tell MATLAB to plot `x` along the horizontal axis and to plot `x` along the vertical axis, rather than telling MATLAB to plot `x` along the horizontal axis and `y` along the vertical axis. Three more lines of code follow. One sets the limits of the `x` axis to ensure that the first point is plotted (a need that arises for this particular graph). The second specifies the label for the `x` axis, using the `xlabel` command. The third specifies the label for the `y` axis, using the `ylabel` command. (More details about these commands will be given in Chapter 9. You can just skim over them here.)

Code 1.8.1:

```
x = [1 2 3 4 5 6 7 8];
y = [0.39  0.47  0.60  0.21  0.57  0.36  0.64  0.32];
plot(x,x,'ko-')
xlim([0 9])
xlabel('Session')
ylabel('Score on Test')
```

When you look at the output, you are impressed by the beauty of the results.

Output 1.8.1:

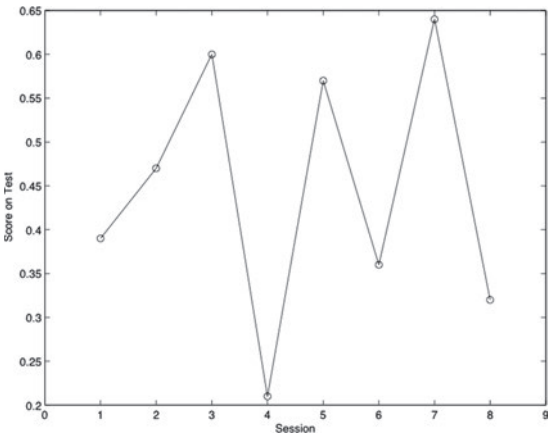


Before calling a press conference, however, it would be advisable for you to check your work. In this case, the results look too good to be true, and in fact, they are. An error was made. Once the error has been found and fixed (with a comment inserted in the program accordingly), the results look quite different.

Code 1.8.2:

```
x = [1 2 3 4 5 6 7 8];
y = [0.39  0.47  0.60  0.21  0.57  0.36  0.64  0.32];
plot(x,y,'ko-')  % Correction made here!
xlim([0 9])
xlabel('Session')
ylabel('Score on Test')
```

Output 1.8.2:



The point of this example is that you should avoid being too self-congratulatory, at least until you know you have something to be very proud of. We hope you will reach that point! Be open to negative feedback. In that connection, we authors welcome corrections and suggestions about ways to improve this book. Feel free to contact us. We will welcome constructive comments.

1.9 Programming With a Friend

No matter how open you may be to negative feedback, it is hard to catch all the mistakes you may make. And no matter how useful it may be to talk aloud in forming your plan for a computer program, you may feel uncomfortable speaking to no one in particular, especially when others are in earshot.

A good way to avoid these problems is to have a friend by your side while you program. This is one of the best ways to program, in our opinion. Apart from the fact that the interactions can be fun, having two pairs of eyes and ears on a problem can spur creativity.

We encourage you to program with someone else. The co-authors of this text often share questions and suggest solutions with each other, even though we usually collaborate at a distance. If you are using this book in a course, we encourage your instructor to find ways of grading your work so cooperation with others counts for you, not against you.

1.10 Writing Concise Programs

It is fairly easy to write a program that has many unnecessary variables and superfluous lines. It is harder, at least early in training, to write a program that does the same job with few variables and lines. It becomes a source of pride to programmers when they write concise programs. Such programs do more than appeal to programmers' aesthetic sense. Concise programs also tend to finish in less time than programs that are verbose, go on and on, are redundant, and have far too many words in them, as in this needlessly long sentence that should have ended long ago had we not wanted to make the point that excess verbiage isn't helpful.

Sometimes, but not always, a concise program can reduce the time to run a program by seconds, minutes, hours, or even days. If the program must solve a problem on which people's lives depend, finding a quick solution can literally mean the difference between life and death. In more mundane terms, when a program is used to acquire behavioral data, if it runs too slowly, not all potential data can be captured. That said, it is of course possible to write *too* concisely, so the code is obscure to other readers and maybe even to yourself once you've set it aside for a while. Our advice, then, is to be concise, but only to the extent necessary. Don't obsess about writing code that's ultra-brief if it makes it harder for you or others to understand it.

1.11 Writing Clear Programs

As just said, program conciseness can enhance clarity, but that's not always the case. Just as you should be as lucid as possible about what your program must do (the second principle in the list above), you should write programs that are as easy as possible to read and understand. Program clarity becomes especially important when you have written many programs. If you return to a program that you wrote days or weeks ago and find yourself unable to understand it, you will be very frustrated.

There are several things you can do to make your programs clear. One is to use extra lines of code or extra variables to make the structure of the program transparent. For example, if you need to divide one term by another and the numerator and denominator both contain complex expressions, it usually helps to have one variable for the terms in the numerator and another variable for the terms in the denominator. The quotient can then be expressed as the ratio of the two variables. The program might have a few more variables than are strictly required, but it will be easier for you and others to understand the code later.

A second practice to make your code clear is to give your variables meaningful names. For example, in the program presented earlier (Codes 1.8.1 and 1.8.2), it would have helped to call the independent variable `session` rather than `x` and to call the dependent variable `test_score` rather than `y`. Using those meaningful variable names might have prevented the "accidental" plotting of `x` against `x` rather than the more appropriate plotting of `y` against `x`.

A third practice to improve program clarity is to add comments. Comments are nonexecutable statements that provide information for the programmer (or reader) instead of for the computer. In MATLAB, comments are preceded by a percent sign (`%`), as shown in Code 1.8.2.

Programmers comment in different ways. Some interleave comments and executable lines of code. Others tend to provide comments above the executable code (at the start of the program), putting relatively few comments in the body of the program. The first author of this book prefers the latter method because it allows him to provide a conceptual plan for the program to follow, along with introductions of the variables he will be using. He prefers not to have too many comments interspersed with code within his programs because he finds them distracting to read and, frankly, a pain to write.

Providing comments at the start of a program can help you start your programming session by combining the need for commenting with the need for "speaking aloud." Developing a plan for a program is often aided by putting the plan into words, as stated earlier (Section 1.5). Being able to say what your program should do will help you write the code you need. The first author often sits down and starts typing the description of what his program will do, editing the emerging comment until he reaches the point where he thinks the procedure he's describing is as clear and mechanically doable as he can make it. Then he begins coding, testing one part of the code at a time, saving successive edits in files with higher and higher version numbers.

Here is an example of one such program. The comments in the opening section (before any executable statements) are typical of what the first author writes. In a short program like this, no further comments are usually needed, because once you gain familiarity with MATLAB, the meanings of the executable statements can usually be understood if the context is clear. All the commands used below will be explained in more detail later in this book.

Code 1.11.1:

```
% Largest_So_Far_01

% Find the largest value in the one-row matrix V.
% Initialize largest_so_far to minus infinity.
% Then go through the matrix by first setting i to 1
% and then letting i increase to the value equal
% to the number of elements of V, given by length(V).
% If the i-th value of V is greater than largest_so_far,
% reassign largest_so_far as the i-th value of V.
% After going through the whole array, print out
% largest_so_far.

V = [7 33 39 26 8 18 15 4 0];
largest_so_far = -inf;
for i = 1:length(V)
    if V(i) > largest_so_far
        largest_so_far = V(i);
    end
end
largest_so_far
```

Output 1.11.1:

```
largest_so_far =
    39
```

The foregoing program can be adapted to find the largest value of other arrays, including much larger ones. We include the program here to give you a taste for what MATLAB programs look like. We also want to convey the idea that it's advisable to test programs on small scales. In general, it's advisable to work on "toy" problems before scaling up to larger ones. This program was tested with an array of length 9. Nine numbers is a more tractable length to use at first than 9,000,000. Just to be sure there are no problems, the program should also be tested with sample data sets in which the largest value is in the first or last position of the matrix because many program errors only reveal themselves at such boundaries.

One last point about program clarity follows. Like all writing, a program is composed for several audiences. Apart from yourself (the person writing and using the code), there are three audiences to keep in mind.

First, there is the computer. The computer, the machine, must be able to deal with the program in the way you wish. At the very least, the program supplied to the computer must be syntactically and logically correct.

The second audience is a colleague, who may wish to evaluate or adapt your program for a related purpose. The colleague may need to understand your program and its logic, with or without your direct advice, and without any particular insights into how you addressed the problem beyond the comments you provided.

The third audience is your future self who, another day, may look back at the prior work. At that later time, you may be faced with understanding what you did without a detailed memory of how you addressed the problem. In the urgency of writing your program to solve an immediate problem, you may take shortcuts, such as using very brief mnemonics for variable names, the meaning of which may be forgotten in the future. To ensure against this unhappy outcome, you may find that once the program is completed, it will serve you to spend a little time clarifying the variable names and adding a few judicious comments. Once you have made these changes, be sure to test the program again, lest your clarification inadvertently produced a new error.

In that spirit, here is the program from Code 1.11.1, with the variable name `V` replaced by `theDataArray`. A couple of other variables and comments have been added as well. Is it clearer to read? Is the result different? Try to make your own programs "self-documenting" by selecting variable names and comments that are as self-explanatory as possible.

Code 1.11.2:

```
% Largest_So_Far_02

% Find the largest value in the one-row matrix theDataArray.
% Initialize largest_so_far to minus infinity.
% Then go through the matrix, by first setting i to 1
% and then letting i increase to the value equal
```

```
% to the number of elements of theDataArray, given by
% length(theDataArray).
% If the i-th value of theDataArray is greater than
% largest_so_far,reassign largest_so_far with the i-th
% value of theDataArray.
% After having gone through the whole array, print out
% largest_so_far, which will be the largest value found.

theDataArray = [7 33 39 26 8 18 15 4 0];
%start with an absurdly small maximum
largest_so_far = -inf;

for i = 1:length(theDataArray)
    if theDataArray(i) > largest_so_far
        %Got a new candidate!
        largest_so_far = theDataArray(i);
    end
end

% All done...so what's the maximum?
largest_of_them_all = largest_so_far
```

Output 1.11.2:

```
largest_of_them_all =
    39
```

1.12 Writing Correct Programs

If your program does not generate any error messages and generates plausible output, does that mean the results are correct? You will find that the MATLAB programming environment, introduced in Chapter 2, serves as an excellent source of feedback as you write and then try to run your own programs. You will be told, indirectly or directly, if your syntax (word use and punctuation) is acceptable or unacceptable. If your syntax is unacceptable, you will get an error message. Otherwise, your program will run. If you get an error message, it will be up to you to figure out what needs to be done to resolve the error. It takes some time to learn to interpret error messages, but over time you will learn to do so.

If your syntax is acceptable, it is your responsibility to confirm that the output you get is correct, because correct syntax alone does not guarantee correct program logic. You will find that judging the correctness of your program's output is often as challenging as generating acceptable syntax. As in natural language, an expression can be syntactically correct but not mean what you intend. Sometimes a program seems to work, but lurking within it is some subtle error that makes the output obviously wrong or, much worse, seemingly correct but actually flawed.

Detecting such mistakes is one of the most challenging aspects of programming. In general, developing a program that works correctly requires more than an understanding of programming syntax. It also requires greater clarity and explicitness about procedures to

be followed than is usually required in daily life. Additionally, it requires some way of verifying the output. Striving for such clarity and explicitness is one of the things that makes programming a humbling, though educational, experience.

As you plan your program, pay attention to the eventual means of verification as well as the logic of computation. For instance, suppose you have a set of reaction-time data from a within-subjects design experiment. In such a design, the number of trials observed in each condition may be determined by the experimental design. Part of the output of the analysis program that you write can be the number of trials in each condition (`n_trials`) for each subject, even if those values do not enter into subsequent analyses. You can take getting the correct (i.e., predicted) values of `n_trials` in each condition as evidence that all trials have been considered in the analysis. Conversely, any apparent anomaly in the values of `n_trials` may alert you to an error somewhere, whether in data acquisition or in the summary computation.

Relatedly, if a program analyzes the data of dozens of participants, it is well worth performing the analysis of at least one or two participants by hand, if possible, to verify the match between the computer's computations and your own. In fact, beginning by doing one subject by hand will give you insights into how best to approach the programming problem. Similarly, it's not a bad idea to have two researchers each independently write a program to analyze the same data. If the two programmers' results agree in every detail, you can be reasonably confident in the correctness of the analysis. If it turns out that there is some detail in which the two results do not agree, that outcome provides an opportunity to explore the difference to see if it is due to a programming error, a difference in understanding the data, or error(s) in the analysis logic.

Another useful shortcut for data verification is to exploit a different analysis environment to serve as that "second programmer." The results of analyzing a small subset of the data in a spreadsheet or statistical package should agree perfectly with the corresponding output of your MATLAB program. If the results differ, even apparently trivially, you will want to track down the source of the disagreement.

1.13 Understanding How the Chapters of This Book Are Organized

If you are persuaded that it makes sense for you to go further with this book, it will help you to understand how the book's chapters are organized.

Each chapter begins with the sentence, "This chapter covers the following topics," after which those subjects are listed. The way the subjects are listed is via presentation of the chapters' section names. All the section names of this book begin with gerunds, such as "Understanding . . . ," "Approaching . . . ,", or "Deciding . . . ." The sections are titled this way because we want you to learn by doing. You should be actively engaged in understanding, approaching, and deciding (to name some activities) as you pursue the material presented here.

Continuing with the layout of the chapters, after all the section titles are given, each chapter continues with the sentence, "The commands that are introduced and the sections in which they are premiered are:." This sentence precedes a list of all the new commands introduced in the chapter, along with the sections in which those new command are first discussed. If you run your finger down the list and find the activity to which it corresponds, you should

be able to turn to that section and find an example of how the command is used. The commands discussed are also listed alphabetically, with reference to their first mention, in the Commands Index.

Every program shown in this book has a code number. The first number (to the left of the decimal point) corresponds to the chapter in which the code appears. The second number (between the two decimal points) corresponds to the section in which the code appears. The third number (to the right of the second decimal point) is the number of the code within the section. All MATLAB code appears in `Courier` font, as do all words taken from the code shown in the text body of this book.

Every program that yields output has its output shown in the same format as the code. The output has a number that corresponds to the code that produced it.

One thing that is missing from the programs shown in this book are extensive comments. We have left them out not because comments are unimportant but because, for most of the programs in this book, the comments are, in effect, presented in the text leading up to the programs. If you imagine percent signs in front of the lines of text preceding the code for a program shown here, you effectively have the kind of comment that can be supplied in a program.

Does it make sense for you to read the code shown in this book? Shouldn't you just dive in code for yourself, sinking or swimming as the case may be? We don't want you to sink. We want you to swim, and we think there is much to be learned by reading successful code to figure out what it does and how it does it. You can learn by example. Starting with examples of code can be one of the best ways to learn to program. You can always edit the working example for your own needs, much as a cook can edit a recipe he or she reads in a cookbook.

1.14 Using the Website Associated With This Book

As you leaf through this book, you will see that all the graphs and images are in grayscale. The programs that yield these graphs and images allow for color graphics. The reason the book has grayscale images is to keep the cost of production down, which translates into a lower price for you. You can see the color images generated by the programs, and animations, by going to the website associated with this book (www.routledge.com/9780415535946). You will be able to copy the programs and outputs as you wish.

1.15 Obtaining and Installing MATLAB

How can you access MATLAB? You or your institution can purchase individual or shared licenses. Students can also purchase the educational version for their own use.

MATLAB is, formally, a cross-platform programming environment with versions for Windows, Mac OS, and Unix. There are superficial differences between the Windows version of MATLAB and the version that runs under the Mac OS or Unix operating system. If a program involves certain kinds of input-output, there may be differences across platforms, but these will not interfere with your mastery of the basics of the language.

The differences between the Windows and Mac OS platforms relate primarily to common platform-specific GUI (graphical user interface) conventions and aspects of interfacing for real-time data acquisition. Most of the computational features of MATLAB are equivalent across platforms, so programs written on one platform should work on another. Where there are important platform differences that can cause problems, we will point them out, though we cannot anticipate all problems that might arise.

As of this writing, we have used versions MATLAB installed in the following contexts:

- As a stand-alone program, individually licensed to a particular researcher under an academic license.
- As a stand-alone program (student version), individually licensed to an undergraduate or graduate student.
- Under an educational site license in which the number of simultaneous users on a campus is monitored by a local server.
- As a program that runs remotely on a central server, to which a limited number of simultaneous users may log on.
- Using an open-source alternative to MATLAB, called OCTAVE (www.gnu.org/software/octave/), that allows the running of much of the code of MATLAB. OCTAVE lacks the closely coordinated debugging and program management tools of MATLAB, and we have found that its graphics are less sophisticated, but it is capable of most of the computational operations of MATLAB.

The examples in this text should almost all run identically regardless of the environment and MATLAB version ("release") that is used. For the most part, we have relied on the current Windows release, R2013a, released March 1, 2013. Because successive MATLAB releases are upward compatible (later versions are compatible with earlier versions), what you learn here should apply to later releases.

How should MATLAB be installed? It is outside our scope to describe the installation procedures needed to get MATLAB to run wherever you are, in part because the details vary depending on the version you are using, the platform you are running on, and the type of license you hold. Ideally, you will have local knowledge to draw on, but MATLAB support through The MathWorks, Inc., is typically very responsive to calls for installation assistance, provided you have your license number handy; see the `ver` command in Chapter 2, Section 2.2.

1.16 Acknowledging Limits

The final section of this chapter is concerned with the limits of this book, our limits as the book's authors, and the limits of MATLAB itself. It is important for you to know what these limits are so you won't form unrealistic expectations.

First, with regard to the book, you should know that you will not be able to program in MATLAB if you just read this book without also trying to program yourself. Reading how

to program is a little like reading how to ride a bike. You have to get on and try it yourself. Don't worry if you fall off a few times. Indeed, experienced as we authors may be, in preparing the examples in this book we had to spend quite a bit of time getting the syntax to work just right, often with many cycles of the edit-run-error-edit loop. It's no reflection on your skills, then, if you have lots of false starts when putting together a new programming project. We, the authors of this book, have gone through those false starts ourselves.

You should also know that the material presented in this book is meant to *acquaint* you with MATLAB but not to convey every aspect of this vast language and its associated applications. This book would be much denser if it went into many more detailed and advanced aspects of the MATLAB programming language. You should be able to delve into these topics on your own having worked through the material provided here.

Third, you should know about the limits of MATLAB. The "word on the street" is that MATLAB is terrific for graphics and for creating conceptual models. Its reputation is less secure when it comes to real-time data gathering, where commercial or free alternatives like E-Prime, PsyScope, and SuperLab are often favored. For large-scale number crunching or statistics, C/C++, R, SPSS, or SAS may be better than MATLAB. On the other hand, MATLAB is being actively enhanced in so many quarters that its limitations, whatever they may be, will probably wane over time as needed tools are being developed to address deficiencies that are spotted by the MATLAB community.

Three examples of such tools can be mentioned here. One is Psychtoolbox (discussed in Chapter 13), which has methods for precise real-time control in psychophysical research. Another tool is an add-on toolbox, MATLAB Coder (not discussed further in this book), which enables MATLAB programs to be converted and distributed as C++ code. A third toolbox from The MathWorks, Parallel Computing, enables intensive computation to be distributed across multiple processors if your computer has more than one. You can learn more about these and other toolboxes provided by The MathWorks by going to their website.

Another comment about the limits of the book is that while the program examples presented here should be comprehensible to you as a behavioral scientist (veteran or fledgling), the program examples are not drawn from a particular approach or finding. The interests of behavioral scientists are highly varied, so the examples offered here are generic rather than specific. They are selected more to highlight particular features of MATLAB than to address specific scientific questions.

2. Interacting With MATLAB

This chapter covers the following topics:

2.1 Using MATLAB's windows
2.2 Using the Command window
2.3 Writing tiny programs in the Command window
2.4 Allowing or suppressing outputs by omitting or including end-of-line semi-colons
2.5 Correcting errors in the Command window
2.6 Writing, saving, and running larger programs as scripts (.m files)
2.7 Running and debugging MATLAB programs
2.8 Keeping a diary
2.9 Practicing interacting with MATLAB

The commands that are introduced and the sections in which they are premiered are:

`calendar`	(2.2)
`clc`	(2.2)
`ctrl-c`	(2.2)
`date`	(2.2)
`disp`	(2.2)
`doc`	(2.2)
`exit`	(2.2)
`help`	(2.2)
`ls`	(2.2)
`open`	(2.2)
`pwd`	(2.2)
`quit`	(2.2)
`ver`	(2.2)
`who`	(2.2)
`;` *(output suppression)*	(2.4)
`up-arrow`	(2.5)
`%`	(2.6)
`...`	(2.6)
`commandwindow`	(2.6)
`ctrl-[`	(2.6)
`ctrl-]`	(2.6)
`ctrl-0` *(zero)*	(2.6)
`ctrl-i`	(2.6)

`edit`	(2.6)
`F5 key`	(2.6)
`New Script button`	(2.6)
`Run button`	(2.6)
`diary`	(2.8)
`type`	(2.8)

2.1 Using MATLAB's Windows

To use MATLAB, you must launch the program. MATLAB is activated, as are most computer applications, by clicking on its icon on the computer desktop or wherever its icon is located. When MATLAB is running, a number of windows will be opened, often as panes docked together in a single window.

When MATLAB is first launched, the **Command** window appears as a pane in the composite window (the one with the name beginning "MATLAB . . . ," followed by the version of MATLAB that you are running). The Command window is the most important window in MATLAB. It is where you control what happens and where you see the results of your programming efforts. The Command window will be described in more detail in Section 2.2.

The second most important window is the **Editor** window, which usually appears as a separate window (the one named "Editor -. . ." followed by the location and name of the file you are editing). Here you exploit MATLAB's editing capabilities by writing, revising, and saving program scripts and functions, both of which are files that end with a `.m` suffix. The Editor window will be discussed in Section 2.3. Suggestions for how best to arrange these windows will be given in Section 2.5.

The two windows just mentioned are the ones that are most critical. Both are normally used to write and run MATLAB programs. There are also several other windows, however, which are more specialized and are described briefly below.

One is the **Help** window. This window provides a portal to MATLAB's tutorials. The Help window can be opened directly by entering a command in the search bar at the top right of the MATLAB window, or it can be opened indirectly by typing the `doc` command in the Command window.

The **Command History** window chronicles the commands used in the Command window. You can use this information to remind you what commands you have issued in a MATLAB session.

The **Current Folder** window lists the contents of the working directory. You will learn how to change the Current Folder in Chapter 6 ("Input-Output"). By default, the Current Folder is set to `My Documents/MATLAB` in Windows, and `Documents/MATLAB` in Mac OS.

The **Workspace** window lists the variables that are currently active, giving their names and values. The values of a variable can be viewed in this window in spreadsheet form by clicking on the grid icon to the left of its name.

Other windows, called **Figure** windows, can be created, opened, and closed in your programs to show graphics, text, and other related information (e.g., sounds). Details will be given in Chapter 9 ("Plots").

2.2 Using the Command Window

As mentioned above, after MATLAB is activated, it brings up the Command window. This is the window where you can issue commands. You do so by typing after the >> prompt.

Some useful commands that can be typed after the >> prompt are given below, followed by the purposes they serve. It will be helpful for you to read through this list now because the commands are listed more or less "chronologically," in a way that corresponds to what occurs in a typical MATLAB session. Some of the commands tend to be used more than others. The most frequent ones, in our experience, are `help`, `ls`, `pwd`, `edit`, `open`, `ctrl-c`, and `exit`.

`ver`	Information about your license, computer, and MATLAB version, together in a convenient summary. If you consult with MathWorks support, you will need this information.
`date`	The current date (in a format you can specify).
`disp`	The value of an expression (numeric or string), displayed in the Command window.
`calendar`	The calendar for the current month.
`help`	Topics for which help can be provided within the command window. Adding a topic name after `help` (followed by a space) brings up help about that topic, provided it is known to MATLAB. You can find out what topics are known to MATLAB by first typing `help` alone. This brings up all the categories for which `help` is available.
`doc`	This is a shortcut to the Help window, where all the help that can be viewed in the Command window is available, plus more. The Help navigator can also be accessed via the Help tab at the top of the main MATLAB window.
`pwd`	Identifies the current directory, the one listed in the Current Folder window, and the default location for saving a script. (`pwd` stands for "print working directory".)
`ls`	Lists the contents of the current directory. Adding just part of a file name after `ls` (following a space) with an asterisk

	replacing part of the file name causes all the files with that named part to be listed. Thus, `ls tim*` lists `tim_program_01.m`, `tim_program_02.m`, `timmy_program_101.m`, and `timothy.doc`, provided these files exist in the current directory. `ls tim*.m` lists `tim_program_02.m`, and `timmy_program_101.m`, but not `timothy.doc`.
`open`	Opens a file in the current directory or invokes other programs as needed (e.g., Adobe Acrobat for `.pdf` files).
`who`	Lists the names of the currently active variables.
`whos`	Lists the names of the currently active variables along with their sizes and other attributes.
`ctrl-c`	Holding down the ctrl key and then pressing the c key interrupts the program that is currently running, provided the Command window is the active window (the window in front of any others that are open). This is very useful when you have "runaway" programs and unwanted data are being spewed on the screen or when you have a program that is running for a long time without any output that you actually want.
`clc`	Clears the Command window.
`exit`	Terminates MATLAB.
`quit`	Runs an optional program called `finish.m`, whose contents can be customized by the user, then terminates MATLAB, just as `exit` does.

2.3 Writing Tiny Programs in the Command Window

The preceding list of commands is just a small subset of those that can potentially be typed in the Command window. In fact, the number of possible commands that can be typed in the Command window is infinite, because a series of commands of arbitrary length and complexity can be typed or pasted after the command line prompt (>>).

In practice, typing or pasting very long series of commands is not a good idea, however, because the longer and more complex the commands, the greater the chance of error. Once your sequence of commands has grown to a few lines (or is expected to be several lines long), it is better to generate program scripts "off-line" in MATLAB's Editor. There, the scripts can be saved and modified. We will turn to the Editor in the next section. In this section, setting the stage for what will come when we turn to the Editor per se and to acquaint you with some elementary programming, we will consider a few tiny programs that can be written in the Command window. The rules governing acceptable command syntax are the same whether the commands are typed into the command line "by hand"

or are part of a file in the Editor. Therefore, typing commands into the Command window can be a good way to experiment with getting the syntax right before you add the lines to an edited program.

One of the most fundamental programming tasks is to assign a value to a variable. Suppose you want to assign the number 2 to some variable, arbitrarily called `A`. This can be done by typing `A = 2` after the command line prompt as follows:

Code 2.3.1:

```
>> A = 2
```

Output 2.3.1:

```
A =
     2
```

The ordering of terms in the assignment is important, as shown below.

Code 2.3.2:

```
>> 2 = A
```

Output 2.3.2:

```
??? 2 = A

Error: The expression to the left of the equals sign is
not a valid target for an assignment.
```

The error message indicates that, in contrast to mathematics, where an equation means the same thing regardless of whether terms appear to the left or right of the equal sign, order matters in MATLAB. Thus, `2 = A` does not mean the same thing as `A = 2`. Programmers often say "A gets 2" when referring to statements such as `A = 2` to indicate that they are referring to a variable assignment rather than to a conventional mathematical equation.

In MATLAB, variable names, program names, and other file names are case sensitive. Consequently, if you query MATLAB about the value of `A`, you can get a satisfying, if not terribly exciting, result:

Code 2.3.3:

```
>> A
```

Output 2.3.3:

```
A =
     2
```