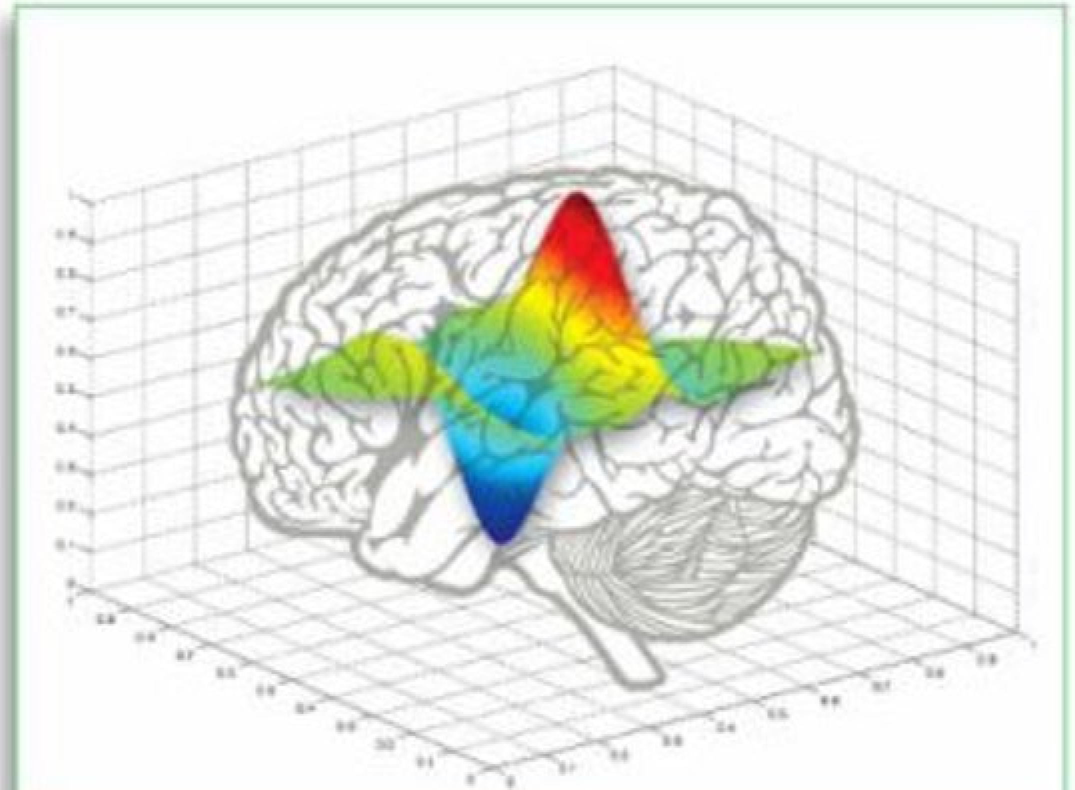
Matlab® *for* Neuroscientists

An Introduction to Scientific Computing in Matlab®



Pascal Wallisch • Michael Lusignan
Marc Benayoun • Tanya I. Baker
Adam S. Dickey • Nicholas G. Hatsopoulos

Academic Press. is an imprint of Elsevier

30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
525 B Street, Suite 1900, San Diego, California 92101–4495, USA
84 Theobald's Road, London WC1X 8RR, UK

This book is printed on acid-free paper. ♾

Library of Congress Cataloging-in-Publication Data
MATLAB for neuroscientists : an introduction to scientific computing in MATLAB / Pascal Wallisch ... [et al.].
p. ; cm.
Includes bibliographical references and index.
ISBN 978-0-12-374551-4 (hardcover : alk. paper)
1. Neuroscience–Data processing. 2. MATLAB. I. Wallisch, Pascal, 1978-
[DNLM: 1. Computing Methodologies. 2. Neurosciences. WL 26.5 M433 2009]
QP357.5.M38 2009
612.80285–dc22

2008028494

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

IBSN: 978-0-12-374551-4

For information on all Academic Press publications visit our Web site at www.elsevierdirect.com

Printed in China
08 09 10 9 8 7 6 5 4 3 2 1

Preface

I hear and I forget.

I see and I remember.

I do and I understand.

Confucius

The creation of this book stems from a set of courses offered over the past several years in quantitative neuroscience, particularly within the graduate program in computational neuroscience at the University of Chicago. This program started in 2001 and is one of the few programs focused on computational neuroscience with a complete curriculum including courses in cellular, systems, behavioral, and cognitive neuroscience; neuronal modeling; and mathematical foundations in computational neuroscience. Many of these courses include not only lectures but also lab sessions in which students get hands-on experience using the MATLAB® software to solve various neuroscientific problems.

The content of our book is oriented along the philosophy of using MATLAB as a comprehensive platform that spans the entire cycle of experimental neuroscience: stimulus generation, data collection and experimental control, data analysis, and finally data modeling. We realize that this approach is not universally followed. Quite a number of labs use different—and specialized—software for stimulus generation, data collection, data analysis, and data modeling, respectively. Although this alternative is a feasible strategy, it does introduce a number of problems: namely, the need to convert data between different platforms and formats and to keep up with a wide range of software packages as well as the need to learn ever-new specialized home-cooked "local" software when entering a new lab. As we have realized in our own professional life as scientists, these obstacles can be far from trivial and a significant detriment to productivity.

We also believe that our comprehensive MATLAB "strategy" makes particular sense for educational purposes, as it empowers users to progressively solve a wide variety of computational problems and challenges within a single programming environment. It has the added advantage of an elegant progression within the problem space. Our experience in teaching has led us to this approach that does not focus on the inherent structure of MATLAB as a computer programming language but rather as a tool for solving problems within neuroscience. In addition, it is well founded in our current understanding of the learning process. Constant use of the information forces the repeated retrieval of the introduced concepts, which—in turn—facilitates learning (Karpicke & Roediger, 2008).

The book is structured in four parts, each with several chapters. The first part serves as a brief introduction to some of the most commonly used functions of the MATLAB software, as well as to basic programming in MATLAB. Users who are already familiar with MATLAB may skip it. It serves the important purpose of a friendly invitation to the power of the MATLAB environment. It is elementary insofar as it is necessary to have mastered the content within before progressing any further. Later parts focus on the use of MATLAB to solve computational problems in neuroscience. The second part focuses on MATLAB as a tool for the collection of data. For the sake of generality, we focus on the collection of data from human subjects in these chapters, although the user can easily adapt them for the collection of animal data as well. The third part focuses on MATLAB as a tool for data analysis and graphing. This part forms the core of the book, as this is also how MATLAB is most commonly used. In particular, we explore the analysis of a variety of datasets, including "real" data from electrophysiology as well as neuroimaging. The fourth part focuses on data modeling with MATLAB, and appendices address the philosophy of MATLAB as well as the underlying mathematics. Each chapter begins with the goals of the chapter and a brief background of the problem of interest (neuroscientific or psychological), followed by an introduction to the MATLAB concepts necessary to address the problem by breaking it down into smaller parts and providing sample code. You are invited to modify, expand, and improvise on these examples in a set of exercises. Finally, a project is assigned at the end of the chapter which requires integrating the parts into a coherent whole. Based on our experience, we believe that these chapters can serve as self-contained "lab" components of a course if this book is used in the context of teaching.

In essence, we strived to write the book that we wished to have had when first learning MATLAB ourselves, as well as the book that we would have liked to have had when teaching MATLAB to our students in the past. Our hope is that this is the very book you are holding in your hands right now.

We could have not written this book without the continuous support of a large number of friends. First and foremost, we would like to thank our families for their kind support, their endless patience, as well as their untiring encouragement. We also would like to extend thanks to our students who provided the initial impetus for this undertaking as well as for providing constant feedback on previous versions of our manuscript. Steve Shevell deserves thanks for suggesting that the project is worth pursuing in the first place. In addition, we would like to thank everyone at Elsevier who was involved in the production and development of this book—in particular our various editors, Johannes Menzel, Sarah Hajduk, Clare Caruana, Christie Jozwiak, Chuck Hutchinson, Megan Wickline, and Meg Day—their resourcefulness, professionalism and patience really did make a big difference. Curiously, there was another Meg involved with this project, specifically Meg Vulliez from The MathWorks™ book program. In addition, we would like to thank Kori Lusignan and Amber Martell for help with illustrations and Wim van Drongelen for advice and guidance in the early stages of this project. Moreover, we thank Armen Kherlopian and Gopathy Purushothaman who were kind enough to provide us with valuable insights throughout our undertaking. We also would like to thank Kristine Mosier for providing the finger-tapping functional magnetic imaging data that we used in the fMRI lab and would like to thank Aaron Suminski for his help in the post-processing of that data. Importantly, we thank everyone whom we neglected to name explicitly but deserves our praise. Finally, we would like to thank you, the reader, for your willingness to join us on this exciting journey. We sincerely hope that we can help you reach your desired destination.

The authors

About the Authors

Pascal Wallisch, PhD, Center for Neural Science, New York University
Pascal received his PhD from the University of Chicago and is now a postdoctoral fellow at New York University. He is currently studying the processing of visual motion. Pascal is passionate about teaching, as well as the communication of scientific concepts to a wider audience. He was recognized for his distinguished teaching record by the University of Chicago Booth Prize.

Michael Lusignan, Committee on Computational Neuroscience, University of Chicago
Michael is an advanced graduate student who has enjoyed teaching several courses involving MATLAB to graduate, as well as undergraduate students. He infuses his teaching with eight years of experience in active software development. His current interests include sensory encoding in neuroethological model systems.

Marc Benayoun, Committee on Computational Neuroscience, University of Chicago
Marc is an MD/PhD student currently interested in applying statistical field theory to study neural networks with applications to epilepsy. He has an extensive teaching record and was also awarded the University of Chicago Booth Prize.

Tanya I. Baker, PhD, Junior Research Fellow, Crick-Jacobs Center for Theoretical Neurobiology, The Salk Institute for Biological Studies, La Jolla, California
Tanya is a junior research fellow modelling large-scale neuronal population dynamics using modern statistical methods. Previously, she was a post-doctoral lecturer at the University of Chicago where she developed and taught *Mathematical Methods for the Biological Sciences*, a new year-long course with a computer lab component. She received her PhD in Physics at the University of Chicago and her BS in Physics and Applied Mathematics at UCLA.

Adam Dickey, Committee on Computational Neuroscience, University of Chicago
Adam is an MD/PhD candidate at the University of Chicago. He is currently a graduate student in the laboratory of Dr. Nicholas Hatsopoulos. Adam is interested in improving decoding techniques used for neural prosthetic control.

Nicholas G. Hatsopoulos, PhD, Department of Organismal Biology and Anatomy & Department of Neurology, University of Chicago
Nicholas is Associate Professor and Chairman of the graduate program on Computational Neuroscience. He teaches a course in

Cognitive Neuroscience which formed the basis for some of the chapters in the book. His research focuses on how ensembles of cortical neurons work together to control, coordinate, and learn complex movements of the arm and hand. He is also developing brain-machine interfaces by which patients with severe motor disabilities could activate large groups of neurons to control external devices.

How to Use This Book

A text of a technical nature tends to be more readily understood if its design principles are clear from the very outset. This is also the case with this book. Hence, we will use this space to briefly discuss what we had in mind when writing the chapters. Hopefully, this will improve usability and allows you to get most out of the book.

STRUCTURAL AND CONCEPTUAL CONSIDERATIONS

A chapter typically begins with a concise overview of what material will be covered. Moreover, we usually put the chapter in the broader context of practical applications. This brief introduction is followed by a discussion of the conceptual and theoretical background of the topic in question. The heart of each chapter is a larger section in which we introduce relevant MATLAB® functions that allow you to implement methods or solve problems that tend to come up in the context of the chapter topic. This part of the chapter is enriched by small exercises and suggestions for exploration. We believe that doing the exercises is imperative to attain a sufficiently deep understanding of the function in question, while the suggestions for exploration are aimed at readers who are particularly interested in broadening their understanding of a given function. In this spirit, the exercises are usually rather specific, while the suggestions for exploration tend to be of a rather sweeping nature. This process of successive introduction and reinforcement of functions and concepts culminates in a "project", a large programming task that ties all the material covered in the book together. This will allow you to put the learned materials to immediate use in a larger goal, often utilizing "real" experimental data. Finally, we list the MATLAB functions introduced in the chapter at the very end. It almost goes without saying that you will get the most out of this book if you have a version of MATLAB open and running while going through the chapters. That way, you can just try out the functions we introduce, try out new code, etc.

Hence, we implicitly assumed this to be the case when writing the book.

Moreover, we made sure that all the code works when running the latest version of MATLAB (currently 7.7). Don't let this concern you too much, though. The vast majority of code should work if you use anything above version 6.0. We did highlight some important changes where appropriate.

LAYOUT AND STYLE

The reader can utilize not only the conceptual structure of each chapter as outlined above, but also profit from the fact that we systematically encoded information about the function of different text parts in the layout and style of the book.

The main text is set in 10/12 Palatino-Roman. In contrast, executable code is **bolded** and offset by >>, such as this:

```
>> figure
>> subplot(2,2,1)
>> image(test_disp)
```

The idea is to type this text (without the >>) directly into MATLAB. Moreover, functions that are first introduced at this point are **bolded** in the text. Exercises and Suggestions for exploration are set in italics and separated from the main text by boxes.

Equations are set in 10/12 Palatino-Roman. Sample solutions in 10/12 Palatino-Bold.

COMPANION WEBSITE

The successful completion of many chapters of this book depends on additional material (experimental data, sample solutions and other supplementary information) which is accessible from the website that accompanies this book. For example, a database of executable code will be maintained as long as the book is in print. For information on how to access this online repository, please see page ii.

CHAPTER

1 Introduction

Neuroscience is at a critical juncture. In the past few decades, the essentially biological nature of the field has been infused by the tools provided by mathematics. At first, the use of mathematics was mostly methodological in nature—primarily aiding the analysis of data. Soon, this influence turned conceptual, framing the very issues that characterize modern neuroscience today. Naturally, this development has not remained uncontroversial. Some neurobiologists of yore resent what they perceive to be a hostile takeover of the field, as many quantitative methods applied to neurobiology were pioneered by nonbiologists with a background in physics, engineering, mathematics, statistics, and computer science. Their concerns are not entirely without merit. For example, Hubel and Wiesel (2004) warn of the faddish nature that the idol of "computation" has taken on, even likening it to a dangerous disease that has befallen the field and that we should overcome quickly in order to restore its health.

While these concerns are valid to some degree and while excesses do happen, we strongly believe that—all in all—the effect of mathematics in the neurosciences has been very positive. Moreover, we believe that our science is and will continue to be one that is computational at its very core. Historically, this notion stems in part from the influence that cognitive psychology has had in the study of the mind. Cognitive psychology and cognitive science, more generally, posited that the mind and, by extension, the brain should be viewed as an information processing device that receives inputs and transforms these inputs into intermediate representations which ultimately generate observable outputs. At the same time that cognitive science was taking hold in psychology in the 1950s and 1960s, computer science was developing beyond mere number crunching and considering the possibility that intelligence could be modeled computationally, leading to the birth of artificial intelligence. The information processing perspective, in turn, ultimately influenced the study of the brain and is best exemplified by an influential book by David Marr titled *Vision,* published in 1982. In that book, Marr proposed that vision and, more generally, the brain should be studied at three levels of analysis: the computational, algorithmic, and implementational levels. The challenge at the computational level is to determine what computational problem a neuron, neural circuit, or part of the brain is solving. The algorithmic level identifies the inputs, the outputs, their representational format, and the algorithm that takes the input representation and transforms it into an output representation. Finally, the implementational level identifies the neural "hardware" and biophysical mechanisms that underlie the algorithm which solves the problem. Today, this perspective has permeated not only cognitive neuroscience but also systems, cellular, and even molecular neuroscience.

Importantly, such a conceptualization of our field places chief importance on the issues surrounding scientific computing. For someone to participate in or even appreciate state of the art debates in modern neuroscience, that person has to be well versed in the language of computation. Of course, it is the task of education—if it is to be truly liberal—to enable students to do so. Yet, this poses a quite formidable challenge.

For most students interested in neuroscience, mathematics amounts to what is essentially a foreign language. Similarly, the language of scientific computing is typically as foreign to students as it is powerful. The prospects of learning both at the same time can be daunting and—at times—overwhelming. So what is a student or educator to do?

Immersion has been shown to be a powerful way to learn foreign languages (Genesee, 1985). Hence, it is imperative that students are using these languages as often as possible when facing a problem in the field. For immersion to work, the learning experience has to be positive, yielding useful results that solve some real or perceived problem. Unfortunately, the inherent complexity as well as the seemingly arcane formalisms that characterize both are usually very off-putting to students, requiring much effort with little tangible yield, reducing the likelihood of further voluntary immersion.

To break this catch-22, the utility of learning these languages has to be drastically increased while making the learning process more accessible and manageable at the same time, even during the learning process itself. As we alluded to previously, this is a tall order. Fortunately, there is a way out of this conundrum. Recent advances in software, as well as hardware, have instantiated scientific computing within the framework of a unified computational environment. One of these environments is provided by the MATLAB® software. For reasons that will become readily apparent in this book, MATLAB fulfills the requirements that are necessary to meet and overcome the challenges outlined earlier. In addition—and partly for these reasons—MATLAB has become the de facto standard of scientific computing in our field. More strongly, MATLAB really has become the lingua franca that all serious students of neuroscience are expected to understand in the very near future, if not already today.

This, in turn, introduces a new—albeit more tractable—problem. How does one teach MATLAB to a useful level of proficiency without making the study of MATLAB itself an additional problem and simply another chore for students? Overcoming this problem as a key to reaching the deeper goals of fluency in mathematics and scientific computing is a crucial goal of this book. We reason that a gentle introduction to MATLAB with a special emphasis on immediate results will computationally empower you to such a degree that the practice of MATLAB becomes self-sustaining by the end of the book. We carefully picked the content such that the result constitutes a confluence of ease (gradually increasing sophistication and complexity) and relevance. We are confident that at the end of the book you will be at a level where you will be able to venture out on your own, convinced of the utility of MATLAB as a tool as well as your abilities to harness this power henceforth. We have tested the various parts of the contents of this book on our students and believe that our approach has been successful. It is our sincere wish and hope that the material contained will be as beneficial to you as it was to those students.

With this in mind, we would like to outline two additional specific goals of this book. First, the material covered in the chapters to follow gives a MATLAB perspective on many topics within computational neuroscience across multiple levels of neuroscientific inquiry from decision-making and attentional mechanisms to retinal circuits and ion channels. It is well known that an active engagement with new material facilitates both understanding and long-time retention of said material. The secondary aim of this book is to acquire proficiency in programming using MATLAB while going through the chapters. If you are already proficient in MATLAB, you can go right to the chapters following the tutorial. For the rest, the tutorial chapter will provide a gentle introduction to the empowering qualities that the mastery of a language of scientific computing affords.

We take a project-based approach in each chapter so that you will be encouraged to write a MATLAB program that implements the ideas introduced in the chapter. Each chapter begins with background information related to a particular neuroscientific or psychological problem, followed by an introduction to the MATLAB concepts necessary to address that problem with sample code and output included in the text. You are invited to modify, expand, and improvise on these examples in a set of exercises. Finally, the project assignment introduced at the end of the chapter requires integrating the exercises. Most of the projects will involve genuine experimental data that are either collected as part of the project or were collected through experiments in research labs. In some rare cases, we use published data from classical papers to illustrate important concepts, giving you a computational understanding of critically important research.

In addition, solutions to exercises as well as executable code can be found in the online repository accompanying this book.

Finally, we would like to point out that we are well aware that there is more than one way to teach—and learn—MATLAB in a reasonably successful and efficient manner. This book represents a manifestation of our approach; it is the path we chose, for the reasons we outlined here.

CHAPTER

2

MATLAB Tutorial

2.1. GOAL OF THIS CHAPTER

The primary goal of this chapter is to help you to become familiar with the MATLAB® software, a powerful tool. It is particularly important to familiarize yourself with the user interface and some basic functionality of MATLAB. To this end, it is worthwhile to at least work through the examples in this chapter (actually type them in and see what happens). Of course, it is even more useful to experiment with the principles discussed in this chapter instead of just sticking to the examples. The chapter is set up in such a way that it affords you time to do this.

If desired, you can work with a partner, although it is advisable to select a partner of similar skill to avoid frustrations and maximize your learning. Advanced MATLAB users can skip this tutorial altogether, while the rest are encouraged to start at a point where they feel comfortable.

The basic structure of this tutorial is as follows: each new concept is introduced through an example, an exercise, and some suggestions on how to explore the principles that guide the implementation of the concept in MATLAB. While working through the examples and exercises is indispensable, taking the suggestions for exploration seriously is also highly recommended. It has been shown that negative examples are very conducive to learning; in other words, it is very important to find out what does not work, in addition to what does work (the examples and exercises will—we hope—work). Since there are infinite ways in which something might not work, we can't spell out exceptions explicitly here. That's why the suggestions are formulated very broadly.

2.2. BASIC CONCEPTS

2.2.1. Purpose and Philosophy of MATLAB

MATLAB is a high-performance programming environment for numerical and technical applications. The first version was written at the University of New Mexico in the 1970s. The "MATrix LABoratory" program was invented by Cleve Moler to provide a simple and interactive way to write programs using the Linpack and Eispack libraries of FORTRAN subroutines for matrix manipulation. MATLAB has since evolved to become an effective and powerful tool for programming, data visualization and analysis, education, engineering and research.

The strengths of MATLAB include extensive data handling and graphics capabilities, powerful programming tools and highly advanced algorithms. Although it specializes in numerical computation, MATLAB is also capable of performing symbolic computation by having an interface with Maple (a leading symbolic mathematics computing environment). Besides fast numerics for linear algebra and the availability of a large number of domain-specific built-in functions and libraries (e.g., for statistics, optimization, image processing, neural networks), another useful feature of MATLAB is its capability to easily generate various kinds of visualizations of your data and/or simulation results.

For every MATLAB feature in general, and for graphics in particular, the usefulness of MATLAB is mainly based on the large number of built-in functions and libraries. The intention of this tutorial is not to provide a comprehensive coverage of all MATLAB features but rather to prepare you for your own exploration of its functionality. The *online help system* is an immensely powerful tool in explaining the vast collection of functions and libraries available to you, and should be the most frequently used tool when programming in MATLAB. Note that this tutorial will not cover any of the functions provided in any of the hundreds of toolboxes, since each toolbox is licensed separately and their availability to you can vary. We will indicate in each section if a particular toolbox is required. If you have additional toolboxes available to you, we recommend using the online help system to familiarize yourself with the additional functions provided. Another tool for help is the Internet. A quick online search will usually bring up numerous useful web pages designed by other MATLAB users trying to help out each other.

As stated previously, MATLAB is essentially a tool—a sophisticated one, but a tool nevertheless. Used properly, it enables you to express and solve computational and analytic problems from a wide variety of domains. The MATLAB environment combines computation, visualization, and programming around the central concept of the matrix. Almost everything in MATLAB is represented in terms of matrices and matrix-manipulations. If you would like a refresher on matrix-manipulations, a brief overview of the main linear algebra concepts needed is given in Appendix B, "Linear Algebra Review." We will start to explore this concept and its power in detail later in this tutorial. For now, it is important to note that, properly learned, MATLAB will help you get your job done in a very efficient way. Giving it a serious shot is worth the effort.

2.2.2. Getting Started

You can start MATLAB by simply clicking on the MATLAB icon on your desktop or taskbar. The command window will pop up, awaiting your commands and instructions.

In the context of this tutorial, all commands that are supposed to be typed into the MATLAB command window, as well as expected MATLAB responses, are typeset in **bold**. The beginnings of these commands are indicated by the **>>** prompt. You press Enter at the end of this line, after typing the instructions for MATLAB. All instructions discussed in this tutorial will be in MATLAB notation, to enhance your familiarity with the MATLAB environment.

Don't be afraid as you delve into this new programming world. Help is readily at hand. Using the command **help** followed by the name of the command (for example, **help save**) in the command window gives you a brief overview on how to use the corresponding command (i.e., the command **save**). You can also easily access these help files for functions or commands by highlighting the command for which you need assistance in either the command window or in an M-file and right-clicking to select the Help on Selection option. Entering the commands **helpwin**, **helpdesk,** or **helpbrowser** will also open the MATLAB help browser. Besides these resources provided directly by the MATLAB program, do not forget the usefulness of the Internet. Not only is additional online help available within MATLAB, but numerous tutorials and advice can be found posted online by other programmers in the MATLAB community.

2.2.3. MATLAB as a Calculator

MATLAB implements and affords all the functionality that you have come to expect from a fine scientific calculator. While MATLAB can, of course, do much more than that, this is probably a good place to start. This functionality also demonstrates the basic philosophy behind this tutorial—discussing the principles behind MATLAB by showing how MATLAB can make your life easier, in this case by replicating the functionality of a scientific calculator.

Elementary mathematical operations: Addition, subtraction, multiplication, division.

These operations are straightforward:

Addition:

```
>> 2 + 3

ans =

   5
```

Subtraction:

```
>> 7 - 5

ans = 2
```

Multiplication:

```
>> 17 * 4

ans =

   68
```

Division:

```
>> 24 / 7

ans =

   3.4286
```

Following are some points to note:

1. It doesn't matter how many spaces are between the numbers and operators, if only numbers and operators are involved (this does not hold for characters):

```
>> 5          + 12

ans =

   17
```

2. Of course, operators can be concatenated, making a statement arbitrarily complex:

```
>> 2 + 3 + 4 - 7 * 5 + 8 / 9 + 1 - 5 * 6 / 3

ans =

   -34.1111
```

3. Parentheses disambiguate statements in the usual way:

```
>> 5 + 3 * 8

ans =

   29

>> (5 +; 3) * 8

ans =

   64
```

"Advanced" mathematical operators: Powers, log, exponentials, trigonometry. Power: x ^ p is x to the power p:

```
>> 2 ^ 3

ans =

   8
```

Natural logarithm: log:

```
>> log (2.7183)

ans =

   1.0000

>> log(1)

ans =

   0
```

Exponential: exp(x) is e^x

```
>> exp(1)

ans =

   2.7183
```

Trigonometric functions; for example, sine:

```
>> sin(0)

ans =

   0

>> sin(pi/2)

ans =

   1

>> sin(3/2 * pi)

ans =

   -1
```

Note: Many of these operations are dependent on the desired accuracy. Internally, MATLAB works with 16 significant decimal digits (for floating point numbers—see Appendix A), but you can determine how many should be displayed. You do this by using the **format** command. The **format short** command displays 4 digits after the decimal point; **format long** displays 14 or 15 (depending on the version of Matlab). Example:

```
>> log(2.7183)

ans =

   1.0000

>> format long
>> log(2.7183)

ans =

   1.000006684913988

>> format short
>> log(2.7183)

ans =

   1.0000
```

As an exercise, try to "verify" numerically that $x * y = \exp(\log(x) + \log(y))$. A possible example follows:

```
>> 5*7

ans =

   35

>> exp(log(5)+log(7))

ans =

   35.0000
```

Hint: Keep track of the number of your parentheses. This practice will come in handy later.

One of the reasons MATLAB is a good calculator is that—on modern machines—it is very fast and has a remarkable numeric range.

For example:

```
>>2^500

ans =

   3.2734e + 150
```

Note: e is scientific notation for the number of digits of a number.

x e + y means x * 10 ^ y.

Example:

```
>> 2e3

ans =

   2000

>> 2*10^3

ans =

   2000
```

Note that in the preceding exercises MATLAB has responded to a command entered by defining a new variable *ans* and assigning to it the value of the result of the command. The variable *ans* can then be used again:

```
>> ans + ans

ans =

   4000
```

The variable *ans* has now been reassigned to the value 4000. We will explore this idea of variable assignments in more detail in the next section.

Exercise 2.1: Try to find the numeric range of MATLAB. For which values of x in $2 \wedge x$ does MATLAB return a numeric value? For which values does it return infinity or negative infinity, **Inf** or **-Inf**, respectively?

2.2.4. Defining Matrices

Of course, MATLAB can do much more than described in the preceding section. A central concept in this regard is that of vectors and matrices—arrays of vectors. Vectors and matrices are designated by square brackets: [ ]. Everything between the brackets is part of the vector or matrix.

A simple row vector is as follows:

```
>> [1 2 3]

ans =

   1   2   3
```

It contains the elements 1, 2, and 3.

A simple matrix is as follows:

```
>> [2 2 2; 3 3 3]

ans =

   2   2   2
   3   3   3
```

This matrix contains two rows and three columns. When you are entering the elements of the matrix, a semicolon separates rows, whereas spaces separate the columns.

Make sure that all rows have the same number of column elements:

```
>> [2 2 2; 3 3]
??? Error using ==> vertcat
CAT arguments dimensions are not consistent.
```

In MATLAB, the concept of a variable is closely associated with the concept of matrices. MATLAB stores matrices in variables, and variables typically manifest themselves as matrices. *Caution:* This variable is not the same as a mathematical variable, just a place in memory.

Assigning a particular matrix to a specific variable is straightforward. In practice, you do this with the equal operator (=). Following are some examples:

```
>> a = [1 2 3 4 5]

a =

   1   2   3   4   5
```

```
>> b = [6 7 8 9]

b =

   6   7   8   9
```

Once in memory, the matrix can be manipulated, recalled, or deleted.

The process of recalling and displaying the contents of the variable is simple. Just type its name:

```
>> a

a =

   1   2   3   4   5

>> b

b =

   6   7   8   9
```

Note:

1. Variable names are case-sensitive. Watch out what you assign and recall:

   ```
   >> A
   ??? Undefined function or variable 'A'.
   ```

   In this case, MATLAB—rightfully—complains that there is no such variable, since you haven't assigned *A* yet.
2. Variable names can be of almost arbitrary length. Try to assign meaningful variable names for matrices:

   ```
   >> uno = [1 1 1; 1 1 1; 1 1 1]

   uno =

      1   1   1
      1   1   1
      1   1   1

   >> thismatrixisreallyempty = [ ]

   thismatrixisreallyempty =

      [ ]
   ```

You can easily create some commonly used matrices by using the functions **eye**, **ones**, **zeros**, **rand**, and **randn**. The function **eye(*n*)** will create an *nxn* identity matrix. The function **ones(*n*,*m*)** will generate an *n* by *m* matrix whose elements are all equal to 1, and the function **zeros(*n*,*m*)** will generate an *n* by *m* matrix whose elements are all equal to 0. When you leave out the second entry, *m,* in calling those functions, they will generate square matrices of either zeros or ones. So, for example, the matrix *uno* could have been more easily created using the command **uno = ones(3).**

In a similar way, MATLAB will generate matrices of random numbers pulled from a uniform distribution between 0 and 1 through the **rand** function, and matrices of random numbers pulled from a normal distribution with 0 mean and variance 1 through the **randn** function.

MATLAB uses so-called workspaces to store variables. The command **who** will allow you to see which variables are in your workspace, and the command **whos** will return additional information regarding the size, class, and bytes of the variables stored in the active workspace.

Now create two variables, x and y, and assign to them the values 23 and 57, respectively:

```
>> x=23; y=57;
```

Note that when you add a semicolon to the end of your statement, MATLAB suppresses the display of the result of that statement. Next, create a third variable, z, and assign to it the value of $x + y$.

```
>> z = x + y

z=80
```

Let's see what's in the working memory, i.e., the workspace:

```
>> who

Your variables are:

a  ans  b thismatrixisreallyempty  uno  x  y  z

>> whos

Name                      Size     Bytes   Class

a                         1×5       40     double

ans                       2×3       48     double

b                         1×4       32     double

thismatrixisreallyempty   0×0        0     double

uno                       3×3       72     double

x                         1×1        8     double

y                         1×1        8     double

z                         1×1        8     double
```

When you use the command **save**, all the variables in your workspace can be saved into a file. MATLAB data files have the .mat ending. The command **save** is followed by the filename and a list of the variables to be saved in the file. If no variables are listed after the filename, then the entire workspace is saved. For example,

```
save my_workspace x y z
```

will create a file named my_workspace.mat that contains the variables x, y, and z. Now rewrite that file with one that includes all the variables in the workspace. Again, you do this by omitting a list of the variables to be saved:

```
>> save my_workspace
```

You can now clear the workspace using the command **clear all**:

```
>> clear all
>> who
>> x
??? Undefined function or variable 'x'.
```

Note that nothing is returned by the command **who**, as is expected because all the variables and their corresponding values have been removed from memory. For the same reason, MATLAB complains that there is no variable named x because it has been cleared from the workspace. You can now reload the workspace with the variables using the command **load**:

```
>> load my_workspace
>> who

Your variables are:

a ans b thismatrixisreallyempty uno x y z
```

If they are no longer needed, specific variables and their corresponding values can be removed from memory. The command **clear** followed by specific variable names will delete only those variables:

```
>> clear x y z
>> who

Your variables are:

a ans b thismatrixisreallyempty uno
```

Try using the command **help** (i.e., via **help save**, **help load**, and **help clear**) in the command window to learn about some of the additional options these functions provide.

The size of the matrix assigned to a given variable can be obtained by using the function **size**. The function **length** is also useful when only the size of the largest dimension of a matrix is desired:

```
>> size(a)

ans =

   1   5

>> length(a)

ans =

   5
```

The content of matrices and variables in your workspace can be reassigned and changed on the fly, as follows:

```
>> thismatrixisreallyempty = [5]

thismatrixisreallyempty =

   5
```

It is very common to have MATLAB create long vectors of incremental elements just by specifying a start and end element:

```
>> thisiscool = 4:18

thisiscool =

   4   5   6   7   8   9   10   11   12   13   14   15   16   17   18
```

The size of the increment of the vector can be changed by specifying the step size in between the start and end element:

```
>> thisiscool = 4:2:18

thisiscool =

   4   6   8   10   12   14   16   18
```

Two convenient functions that MATLAB has for creating vectors are **linspace** and **logspace**. The command **linspace(*a*,*b*,*n*)** will create a vector of n evenly spaced elements whose first value is a and whose last value is b. Similarly, **logspace(*a*,*b*,*n*)** will generate a vector of n equally spaced elements between decades 10^a and 10^b:

```
>> v= logspace(1,5,5)

v =

         10   100   1000   10000   100000
```

Transposing a matrix or a vector is quite simple: It's done with the ' (apostrophe) command:

```
>> a

a =

   1   2   3   4   5

>> a'

ans =

   1
   2
   3
   4
   5
```

Variables can be copied into each other, using the = command. In this case, the right side is assigned to the left side. What was on the left side before is overwritten and lost, as shown here:

```
>> b

b =

   6   7   8   9

>> b = a

b =

   1   2   3   4   5
```

Note: Don't confuse the = (equal) sign with its mathematical meaning. In MATLAB, this symbol does not denote the equality of terms, but is an assignment instruction. Again, the right side of the = will be assigned to the left side, while the left side will be lost. This is the source of many errors and misunderstandings and is emphasized again here. The conceptual difference is nowhere clearer than in the case of "self-assignment":

```
>> a

a =

   1   2   3   4   5

>> a = a'

a =

   1
   2
   3
   4
   5

>> a

a =

   1
   2
   3
   4
   5
```

The assignment of the transpose eliminates the original row vector and renders *a* as a column vector.

This reassignment also works for elements of matrices and vectors:

```
>> a(2,1) = 9
```

```
a =

    1
    9
    3
    4
    5
```

Generally, you can access a particular element of a two-dimensional matrix with the indices i and j, where i denotes the row and j denotes the column. Specifying a single index i accesses the i^{th} element of the array counted column-wise:

```
>> a(2)

ans =

    9
```

We will explore indexing more in Section 2.2.6.

Exercise 2.2: Clear the workspace. Create a variable A and assign to it the following matrix value:

$$A = \begin{pmatrix} 7 & 5 \\ 2 & 3 \\ 1 & 8 \end{pmatrix}.$$

Access the element $i = 2, j = 1$, and change it to a number twice its original value. Create a variable B and assign to it the transpose of A. Verify that the fifth element of the matrix B counted column-wise is the same as the $i = 1, j = 3$ element.

Exercise 2.3: Using the function **linspace** generates a row vector *v1* with seven elements which uniformly cover the interval between 0 and 1. Now generate a vector *v2* which also covers the interval between 0 and 1, but with a fixed discretization of 0.1. Use either the function **length** or **size** to determine how many elements the vector *v2* has. What is the value of the third element of the vector *v2*?

Suggestions for solutions to many exercises are available on the companion website.

2.2.5. Basic Matrix Algebra

Almost everything that you learned in the previous section on mathematical operators in MATLAB can now be applied to what you just learned about matrices and variables. In this section we explore how this synthesis is accomplished—with the necessary modifications.

First, define a simple matrix and then add 2 to all elements of the matrix, like this:

```
>> p = [1 2; 3 4]

p =

   1   2
   3   4

>> p = p + 2

p =

   3   4
   5   6
```

As a quick exercise, check whether this principle extends to the other basic arithmetic operations such as subtraction, division, or multiplication.

What if you want to add a different number to each element in the matrix? It is not inconceivable that this operation could be very useful in practice. Of course, you could do it element by element, as in the end of the preceding section. But doing this would be very tedious and time-consuming. One of the strengths of MATLAB is its matrix operations, allowing you to do many things at once.

Here, you will define a new matrix with the elements that will be added to the old matrix and assign the result to a new matrix to preserve the original matrices:

```
>> q = [2 1; 1 1]

q =

   2   1
   1   1

>> m = p + q

m =

   5   5
   6   7
```

Note: The number of elements has to be the same for this element-wise addition to work. Specifically, the matrices that are added to each other must have the same number of rows and columns. Otherwise, nothing is added, and the new matrix is not created. Instead, MATLAB reports an error and gives a brief hint what went wrong:

```
>> r = [2 1; 1 1; 1 1]

r =

   2   1
   1   1
   1   1

>> n = p + r
??? Error using ==> plus
Matrix dimensions must agree.
```

As a quick exercise, see whether this method of simultaneous, element-wise addition generalizes to other basic operations such as subtraction, multiplication, and division.

Note: It is advisable to assign a variable to the result of a computation, if the result is needed later. If this is not done, the result will be assigned to the MATLAB default variable *ans*, which is overwritten every time a new calculation without explicit assignment of a variable is performed. Hence, *ans* is at best a temporary storage.

Note that in the preceding exercise, you get consistent results for addition and subtraction, but not for multiplication and division. The reason is that * and / really symbolize a different level of operations than + or -. Specifically, they refer to matrix multiplication and division, respectively, which can be used to calculate outer products, etc. Refer to Appendix B, "Linear Algebra Review," for a refresher if necessary. If you want an analogous operation to + and -, you have to preface the * or / with a dot (.). This is known as *element-wise operations*:

```
>> p

p =

   3   4
   5   6

>> q

q =

   2   1
   1   1

>> p*q

ans =

   10    7
   16   11

>> p.*q

ans =

   6   4
   5   6
```

Due to the nature of outer products, this effect is even more dramatic if you want to multiply or divide a vector by another vector:

```
>> a = [1 2 3 4 5]

a =

   1   2   3   4   5

>> b = [5 4 5 4 5]

b =

   5   4   5   4   5
```

```
>> c = a.*b

c =

   5   8   15   16   25

>> c = a*b
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Raising a matrix to a power is similar to matrix multiplication; thus, if you wish to raise each element of a matrix to a given power, the dot (.) must be included in the command. Therefore, to generate a vector c having the same length as the vector a, but for each element i in c, it holds that $c(i) = [a(i)]^\wedge 2$ you use the following command:

```
>> c = a.^2

c =

   1   4   9   16   25
```

As you might expect, there exists a function **sqrt** that will raise every element of its input to the power 0.5. Note that the omission of the dot (.) to indicate element-wise operations when it is intended is one of the most common errors when beginning to program in MATLAB. Keep this point in mind when troubleshooting your code.

Of course, you do not have to use matrix algebra to manipulate the content of a matrix. Instead, you can "manually" manipulate individual elements of the matrix. For example, if A is a matrix with four rows and three columns, you can permanently add 5 to the element in the third row and second column of this matrix by using the following command:

```
>> A(3,2) = 5 + A(3,2);
```

We will explore indexing further in the next section.

Earlier, we rather casually introduced matrix operations like outer products versus element-wise operations. Now, we will briefly take the liberty to rectify this state of affairs in a systematic way. MATLAB is built around the concept of the matrix. As such, it is ideally suited to implement mathematical operations from linear algebra. MATLAB distinguishes between *matrix operations* and *array operations*. Basically, the former are the subject of linear algebra and denoted in MATLAB with symbols such as +, **-**, *****, **/**, or ^. These operators typically involve the entire matrix. Array operations, on the other hand, are indicated by the same symbols prefaced by a dot, such as **.***, **./**, or **.**^. Array operators operate element-wise, or one by one. The rest of the sections will mostly deal with array operations. Hence, we will give the more arcane matrix operations—and the linear algebra that is tied to it—a brief introduction. Linear algebra has many useful applications, most of which are beyond the scope of this tutorial. One of its uses is the elegant and painless (particularly with MATLAB) solution of systems of equations. Consider, for example, the system

$$x + y + 2z = 9$$

$$2x + 4y - 3z = 1$$

$$3x + 6y - 5z = 0$$

You can solve this system with the operations you learned in middle school, or you can represent the preceding system with a matrix and use a MATLAB function that produces the reduced row echelon form of A to solve it, as follows:

```
>> A = [1 1 2 9; 2 4 -3 1; 3 6 -5 0]

A =

   1   1    2   9
   2   4   -3   1
   3   6   -5   0

>> rref(A)

ans =

   1   0   0   1
   0   1   0   2
   0   0   1   3
```

From the preceding, it is now obvious that $x = 1$, $y = 2$, $z = 3$. As you can see, tackling algebraic problems with MATLAB is quick and painless—at least for you.

Similarly, matrix multiplication can be used for quick calculations. Suppose you sell five items, with five different prices, and you sell them in five different quantities. This can be represented in terms of matrices. The revenue can be calculated using a matrix operation:

```
>> Prices = [10 20 30 40 50];
>> Sales = [50; 30; 20; 10; 1];
>> Revenue = Prices*Sales

Revenue =
   2150
```

Note: Due to the way in which matrix multiplication is defined, one of the vectors (**Prices**) has to be a row vector, while the other (**Sales**) is a column vector.

Exercise 2.4: Double-check whether the matrix multiplication accurately determined revenue.

Exercise 2.5: Which set of array operations achieves the same effect as this simple matrix multiplication?

Exploration: As opposed to array multiplication (.*), matrix multiplication i NOT commutative. In other words, **Prices** * **Sales** $\neq$ **Sales** * **Prices**. Try it by typing the latter. What does the result represent?

Exercise 2.6: Create a variable C and assign to it a 5×5 identity matrix using the function **eye**. Create a variable D and assign to it a 5×5 matrix of ones using the function **ones**. Create a third variable E and assign to it the square of the sum of C and D.

Exercise 2.7: Clear your workspace. Create the following variables and assign to them the given matrix values (superscript T indicates transpose):

(a) $x = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$ (b) $y = x^T \cdot 17 \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ (c) $A = \begin{pmatrix} 3 & 7 \\ 2 & 1 \end{pmatrix}$

(d) $b = yA$ (e) $c = x^T A^{-1} b^T$ (f) $E = cA^T$

Exercise 2.8: Create a time vector t that goes from 0 to 100 in increments of 5. Now create a vector q whose length is that of t and each element of q is equal to $2 + 5$ times the corresponding element of t raised to the 1.7 power.

2.2.6. Indexing

Individual elements of a matrix can be identified and manipulated by the explicit use of their index values. When indexing a matrix, A, you may identify an element with two numbers using the format *A(row, column)*. You could also identify an element with a single number, *A(num)*, where the elements of the matrix are counted column-wise. Let's explore this a bit through a series of exercises. First, remove all variables from the workspace (use the command **clear all**) and create a variable A:

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 10 & 20 & 30 & 40 \\ 50 & 60 & 70 & 80 \end{pmatrix}.$$

```
>> clear all
>> A=[1 2 3 4; 5 6 7 8; 10 20 30 40; 50 60 70 80];
```

Now assign the value 23 to each entry in the first row:

```
>> A(1,:)=23
```

```
A =

   23   23   23   23
    5    6    7    8
   10   20   30   40
   50   60   70   80
```

The colon (:) in the col position indicates all column values. Similarly, you can assign the value 23 to each entry in the first column:

```
>> A(:,1)=23

A =

   23   23   23   23
   23    6    7    8
   23   20   30   40
   23   60   70   80
```

Suppose you didn't know the index values for the elements that you wanted to change. For example, suppose you wanted to assign the value 57 to each entry of *A* that is equal or larger than 7 in the second row. What are the column indices for the elements of the second row of the matrix *A* [i.e., *A(2,:)*] which satisfy the criteria to change? For this task, the **find** function comes in handy:

```
>> find(A(2,:)>=7)

ans =

   1   3   4
```

Thus, the following command will produce the desired result:

```
>> A(2,find(A(2,:)>=7))=57

A =

   23   23   23   23

   57    6   57   57

   23   20   30   40

   23   60   70   80
```

To further illustrate the use of the function **find** and indexing, consider the following task. Assign the value 7 to each entry in the fourth column of the matrix *A* that is equal or larger than 40 and lower than 60. For this example, it is clearer to split this operation into two lines:

```
>> i=find((A(:,4)>=40)&(A(:,4)<60))

i =

   2
   3
```

```
>> A(i,4) = 7

A =

   23   23   23   23
   57    6   57    7
   23   20   30    7
   23   60   70   80
```

Back to a nice and simple task, assign the value 15 to the entry in the third row, second column:

```
>> A(3,2)=15

A =

   23   23   23   23
   57    6   57    7
   23   15   30    7
   23   60   70   80
```

Similarly, you could have used the command **A(7) = 15**. If you try entering the command **find(A==15)**, you will get the answer 7. The reason is that MATLAB stores the elements of a matrix column after column, so 15 is stored in the seventh element of the matrix when counted this way. Had you entered the command **[r,c]=find(A==15)**; you would see that r is now assigned the row index value and c the column index value of the element whose value is 15; that is, $r = 3$, $c = 2$.

```
>> [r,c]=find(A==15)

r =

   3

c =

   2
```

The **find** function is often used with relational and logical operators. We used a few of these in the preceding examples and will summarize them all here. The relational operators are as follows:

== (equal to)
~= (not equal to)
< (less than)
> (greater than)
<= (less than or equal to)
>= (greater than or equal to)

MATLAB also used the following syntax for logical operators:

& (AND)
| (OR)

~ (NOT)
xor (EXCLUSIVE OR)
any (true if any element is nonzero)
all (true if all elements are nonzero.)

Exercise 2.9: Find the row and column indices of the matrix elements of A whose values are less than 20. Set all elements of the third row equal to the value 17. Assign the value 2 to each of the last three entries in the second column.

2.3. GRAPHICS AND VISUALIZATION

2.3.1. Basic Visualization

Whereas we re-created the functionality of a scientific calculator in the previous sections, here we will explore MATLAB as a graphing calculator. As you will see, visualization of data and data structures is one of the great strengths of MATLAB. In this section, it will be particularly valuable to experiment with possibilities other than the ones suggested in the examples, since the examples can cover only a very small number of possibilities that will have a profound impact on the graphs produced.

For aesthetic purposes, start with a trigonometric function, which was introduced before—sine. First, generate a vector x, take the sine of that vector, and then plot the result:

```
>> x = 0:10

x =

   0   1   2   3   4   5   6   7   8   9   10

>> y = sin(x)

y =

      0   0.8415    0.9093   0.1411   -0.7568   -0.9589   -0.2794   0.6570   0.9894
0.4121   -0.5440

>> plot(x,y)
```

The result of this series of commands will look something like Figure 2.1.

A quick result was reached, but the graphic produced is admittedly rather crude, albeit sinusoidal in nature. Note the values on the x-axis (0 to 10), as desired, and the values on the y-axis, between –1 and 1, as it's supposed to be, for a sine function. The problem seems to be with sampling. So let's redraw the sine wave with a finer mesh.

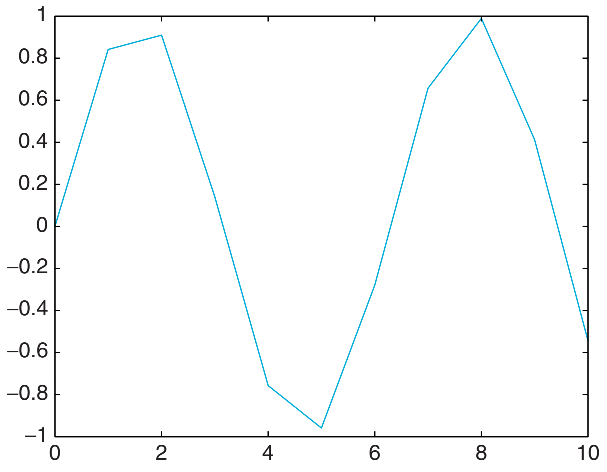


FIGURE 2.1 Crude sinusoid.

Recall that a third parameter in the quick generation of vectors indicates the step size. If nothing is indicated, MATLAB assumes 1 by default. This time, you will make the mesh 10 times finer, with a step size of 0.1.

> ***Exercise 2.10:*** Use **>> x = 0:0.1:10** to create the finer mesh.

Notice that MATLAB displays a long series of incremental elements in a vector that is 101 elements long. MATLAB did exactly what you told it to do, but you don't necessarily want to see all that. Recall that the ; (semicolon) command at the end of a command suppresses the "echo," the display of all elements of the vector, while the vector is still created in memory. You can operate on it and display it later, like any other vector.

So try this:

```
>> x = 0:0.1:10;
>> y = sin(x);
>> plot(x,y)
```

This yields something like that shown in Figure 2.2, which is arguably much smoother.

> ***Exercise 2.11:*** Plot the sine wave on the interval from 0 to 20, in 0.1 steps.

Upon completing Exercise 2.11 enter the following commands:

```
>> hold on
>> z = cos(x);
>> plot(x,z,'color','k')
```

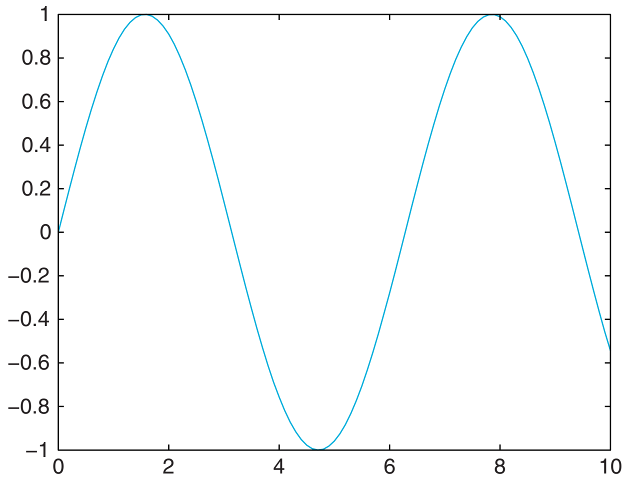


FIGURE 2.2 Smooth sinusoid.

The result should look something like that shown in Figure 2.3.

Now you have two plots on the canvas, the sine and cosine from 0 to 20, in different colors. The command **hold on** is responsible for the fact that the drawing of the sine wave didn't just vanish when you drew the cosine function. If you want to erase your drawing board, type **hold off** and start from scratch. Alternatively, you can draw to a new figure, by typing **figure**, but be careful. Only a limited number of figures can be opened, since every single one will draw from the resources of your computer. Under normal

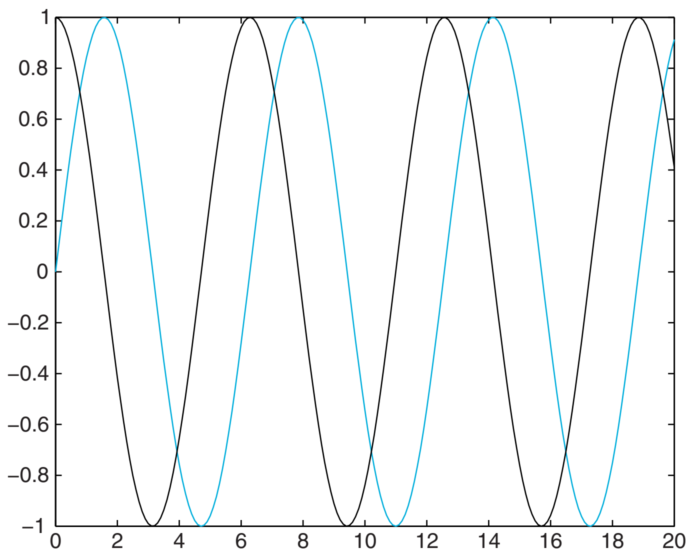


FIGURE 2.3 Sine vs. cosine.

circumstances, you should not have more than about 20 figures open—if that. The command **close all** closes all the figures.

> *Exercise 2.12:* Draw different functions with different colors into the same figure. Things will start to look interesting very soon. MATLAB can draw lines in a large number of colors, eight of which are predefined: *r* for red, *k* for black, *w* for white, *g* for green, *b* for blue, *y* for yellow, *c* for cyan, and *m* for magenta.

Give your drawing an appropriate name. Type something like the following:

```
>> title('My trigonometric functions')
```

Now watch the title appear in the top of the figure.

Of course, you don't just want to draw lines. Say there is an election and you want to quickly visualize the results. You could create a quick matrix with the hypothetical results for the respective candidates and then make a bar graph, like this:

```
>> results = [55 30 10 5]

results =
   55   30   10   5

>> bar(results)
```

The result should look something like that shown in Figure 2.4.

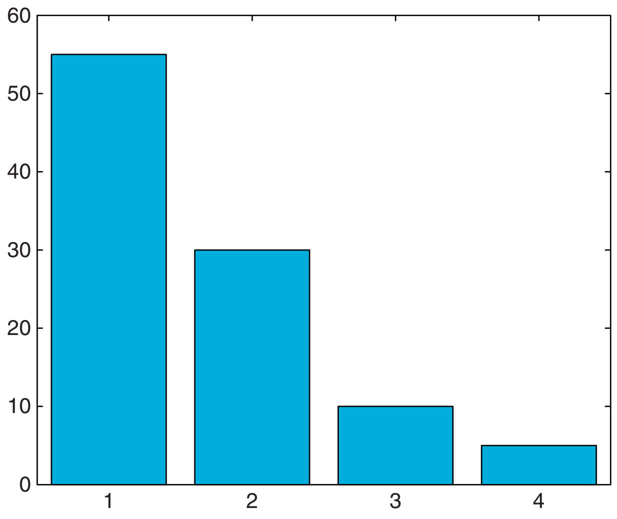


FIGURE 2.4 Lowering the bar.

Exercise 2.13: To get control over the properties of your graph, you will have to assign a handle to the drawing object. This can be an arbitrary variable, for example, *h*:

```
>> h = bar(results)
h =
   298.0027

>> set(h,'linewidth', 3)
>> set(h,'FaceColor', [1 1 1])
```

The result should be white bars with thick lines. Try **get(*h*)** to see more properties of the bar graph. Then try manipulating them with **set(*h*, '*Propertyname*', *Propertyvalue*)**.

Finally, let's consider histograms. Say you have a suspicion that the random number generator of MATLAB is not working that well. You can test this hunch by visual inspection.

First, you generate a large number of random numbers from a normal distribution, say 100,000. Don't forget the ; (semicolon). Then you draw a histogram with 100 bins, and you're done. Try this, for example:

```
>> suspicious = randn(100000,1);
>> figure
>> hist(suspicious, 100)
```

The result should look something like that shown in Figure 2.5. No systematic deviations from the normal distribution are readily apparent. Of course, statistical tests could yield a more conclusive evaluation of your hypothesis.

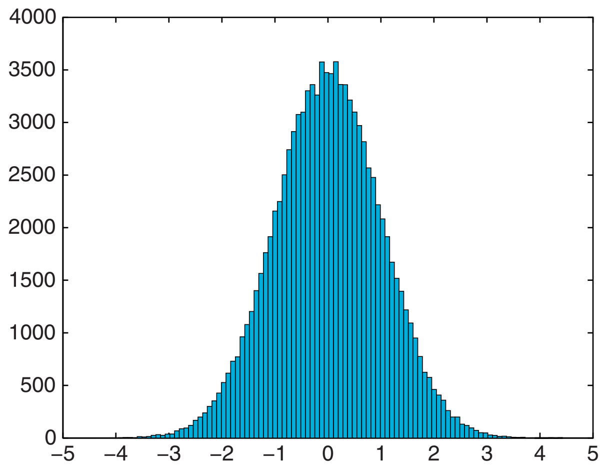


FIGURE 2.5 Gaussian normal distribution.

Exercise 2.14: You might want to run this test a couple of times to convince yourself that the deviations from a normal distribution are truly random and inconsistent from trial to trial.

A final remark on the display outputs: most of the commands that affect the display of the output are permanent. In other words, the output stays like that until another command down the road changes it again. Examples are the **hold** command and **format** command. Typing **hold** will hold plot and allow something else to be plotted on it. Typing **hold** again toggles hold and releases the plot. This is similarly true for the **format** commands, which keep the format of the output in a certain way.

We have thus far introduced you to only a small number of the many visualization tools that give MATLAB its strength. In addition to the functions **plot**, **bar**, and **hist**, you can explore other plotting commands and get a feel for more display options by viewing the help files for the following plotting commands that you might find useful: **loglog**, **semilogx**, **semilogy**, **stairs**, and **pie**.

Want to know how your functions sound? MATLAB can send your data to the computer's speakers, allowing you to visually manipulate your data and *listen* to it at the same time. To hear an example, load the built-in chirp.mat data file by typing **load chirp**. Use **plot(*y*)** to see these data and **sound(*y*)** to listen to the data.

We will cover more advanced plotting methods in the following section as well as in future chapters.

2.4. FUNCTION AND SCRIPTS

Until now, we have driven MATLAB by typing commands directly in the command window. This is fine for simple tasks, but for more complex ones you can store the typed input commands into a file and tell MATLAB to get its input commands from the file. Such files must have the extension .m and are thus called *M-files.* If an M-file contains statements just as you would type them into the MATLAB command window, they are called *scripts*. If, however, they accept input arguments and produce output arguments, they are called *functions*.

The primary goal of this section is to help you become familiar with M-files within MATLAB. M-files are the primary vehicle to implement programming in MATLAB. So while the previous sections showed how MATLAB can double as a scientific calculator and as a calculator with graphing functions, this section will introduce you to a high-level programming language that should be able to satisfy most of your programming needs if you are a casual user. It will become apparent in this section of the tutorial how MATLAB can aid the researcher in all stages of an experiment or study. By no means is this tutorial the last word on M-files and programming. Later we will elaborate on all concepts introduced in this section—particularly in terms of improving efficiency and performance of the programs you are writing. One final goal of this tutorial is to demonstrate the remarkable versatility of MATLAB—and don’t worry, we’ll move on to neuroscience-heavy topics soon enough.

2.4.1. Scripts

Scripts typically contain a sequence of commands to be executed by MATLAB when the filename of the M-file is typed into the command window.

M-files are at the heart of programming in MATLAB. Hence, most of our future examples will take place in the context of M-files. You can work on M-files with the M-file editor, which comes with MATLAB. To go to the editor, open the File menu (top left), select New, and then select M-File (see Figure 2.6).

The first thing to do after a new M-file is created is to name it. For this purpose, you have to save it to the hard disk. There are several ways of doing this. The most common is to click the editor’s File menu and then click Save As. You can then save the file with a certain name. Using **myfirstmfile.m** would probably be appropriate for the occasion.

As a script, an M-file is just a repository for a sequence of commands that you want to execute repeatedly. Putting them in an M-file can save you the effort of typing the commands anew every time. This is the first purpose for which you will use M-files.

Type the commands into your M-file editor as they appear in Figure 2.7. Make sure to save your work after you are done (by pressing Ctrl+S), if you already named it. If you now type **myfirstmfile** into the MATLAB command window (not the editor), this sequence will be executed by MATLAB. You can do this repeatedly, tweaking things as you go along (don’t forget to save).

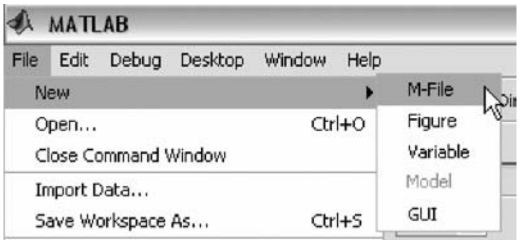
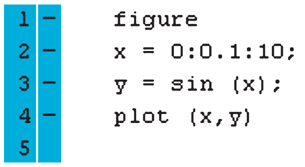


FIGURE 2.6 Creating a new M-File.



```
figure
x = 0:0.1:10;
y = sin (x);
plot (x,y)
```

FIGURE 2.7 The editor.

2.4.2. Functions

We have already been using many of the functions built into MATLAB, such as **sin** and **plot**. You can extend the MATLAB language by writing your own functions as well. MATLAB has a very specific syntax for declaring and defining a function. Function M-files must start with the word **function**, followed by the output variable(s), and equal sign, the name of the function, and the input variable(s). Functions do not have to have input or output arguments. The following is a valid function definition line for a function named **flower** that has three inputs, *a, b,* and *c*, and two outputs, *out_1* and *out_2*:

```
function [out_1, out_2] =flower(a,b,c)
```

To demonstrate this further, you will write a function named **triple** that computes and returns three times the input, i.e., **triple(2)=6, triple(3)=9**, etc. First, type the following two lines into an M-file and save it as **triple.m**:

```
function r = triple(i)

r=3*i;
```

If you want to avoid confusion, it is strongly advised to match the name of the M-file with the name of the function. The input to the function is *i* and the output is *r*. You can now test this function:

```
>> a=triple(7)

a =

   21

>> b=triple([10 20 30])

b =

   30   60   90
```

Note: Of course, the implementation of this function is trivial. Here, however, you should learn to apply the syntax only for defining and calling functions. Also note that function variables do not appear in the main workspace; rather they are local to themselves.

2.4.3. Control Structures

Of course, what you just saw is only the most primitive form to use M-files. M-files will allow you to harness the full power of MATLAB as a programming language. For this to happen, you need to familiarize yourself with loops and conditionals—in other words, with statements that allow you to control the flow of your program.

Loops: The two most common statements to implement loops in MATLAB are **for** and **while**.

The structure of all loops is as follows (in terms of a **while** loop):

```
while certain-conditions-are-true
```

```
Statements
. . .
Statements
end
```

All statements between **while** and **end** are executed repeatedly until the conditions that are checked in **while** are no longer the case. This is best demonstrated with a simple example: open a new M-file in the editor and give it a name. Then type the following code and save. Finally, type the name of the M-file in the command window (not the editor) to execute the loop.

This is a good place to introduce comments. As your programs become more complex, it is highly recommended that you add comments. After a week or two, you will not necessarily remember what a particular variable represents or how a particular computation was done. Comments can help, and MATLAB affords functionality for it. Specifically, it ignores everything that is written after the percent sign (%) in a line. In the editor itself, comments are represented in green. So here is the program that you should write now, implementing a simple **while** loop. If you want to, you can save yourself the comments (everything after % in each line). We placed them here to explain the program flow (what the program will do) to you:

```
%A simple counter
i = 1          %Initializing our counter as 1
while i < 6    %While i is smaller than 6, statements are executed
   i = i + 1   %Incrementing the counter and displaying new value
end            %Ending the loop, it contains only one statement
```

What happened after you executed the program? Did it count from 1 to 6?

Exercise 2.15: Let your program count from 50 to 1050.

If you execute this program on a slow machine, chances are that this operation will take a while.

Exercise 2.16: Let your program count from 1 to 1,000,000.

If you did everything right, you will be sitting for at least a minute, watching the numbers go by. While we set up this exercise deliberately, chances are that you will underestimate the time it takes to execute a given loop sometime in the future. Instead of just biding your time, you have several options at your disposal to terminate runaway computations. The most benign of these options is pressing Ctrl+C in the MATLAB command window. That key press should stop a process that hasn't yet completely taken over the resources of your machine. Try it.

Note: The display takes most of the time. The computation itself is relatively quick. Make the following modifications to your program; then save and run it:

```
%A silent counter
i = 1           %Initializing our counter as 1
while i < 1000000
   i = i + 1;   %Incrementing the counter without displaying new value
end             %Ending the loop, it contains only one statement
i               %Displaying the final value of i
```

Note: One of the most typical ways to get logical errors in complex programs is to forget to initialize the counter (after doing something else with the variable). This is particularly likely if you reuse the same few variable names (*i*, *j*, etc.) throughout the program. In this case, it would not execute the loop, since the conditions are not met. Hence, you should make sure to always initialize the variables that you use in the loop before the loop. As a cautionary exercise, reduce your program to the following:

```
%A simple counter, without initialization
while i < 1000000  %While i is smaller than 1 M, statements are executed
  i = i + 1     %Incrementing the counter and displaying new value
end             %Ending the loop, it contains only one statement
```

Save and run this new program. If you ran one of the previous versions, nothing will happen. The reason is that the loop won't be entered because the condition is not met; *i* is already larger than 1,000,000.

Of course, the most common way to get runaway computations is to create infinite loops—in other words, loops with conditions that are always true after they are entered. If that is the case, they will never be exited. A simple case of such an infinite loop is a modified version of the initial loop program—one without an increment of the counter; hence, *i* will always be smaller than the conditional value and never exit.

Try this, save, and run:

```
%An infinite loop
i = 1           %Initializing our counter as 1
while i < 6     %While i is smaller than 6, statements are executed
  i = i         %NOT incrementing the counter, yet displaying its value
end             %Ending the loop, it contains only one statement
```

If you're lucky, you can also exit this process by pressing Ctrl+C. If you're not quick enough or if the process already consumed too many resources—this is particularly likely for loops with many statements, not necessarily this one—your best bet is to summon the Task Manager by pressing Ctrl+Alt+Delete simultaneously in Windows (for a Mac, the corresponding key press is Command+Option+Escape to call the Force Quit menu). There, you can kill your running version of MATLAB. The drawbacks of this method are that you have to restart MATLAB and your unsaved work will be lost. So beware the infinite loop.

If statements: In a way, **if** statements are pure conditionals. Statements within **if** statements are either executed once (if the conditions are met) or not (if they are not met). Their syntax is similar to loops:

if these-conditions-are-met

Execute-these-Statements

else

Execute-these-Statements

end

It is hard to create a good example consisting solely of **if** statements. They are typically used in conjunction with loops: the program loops through several cases, and when it hits a special case, the **if** statement kicks in and does something special. We will see instances of this in later examples. For now, it is enough to note the syntax.

Fun with loops—How to make an American quilt

This is a rather baroque but nevertheless valid exercise on how to simply save time writing all the statements explicitly by using nested loops. If you want to, you can try replicating all the effects without the use of loops. It's definitely possible—just very tedious.

Open a new window in the editor, name it, type the following statements (without comments if you prefer), save it, and see what happens when you run it:

```
figure          %Open a new figure
x = 0:0.1:20;   %Have an x-vector with 201 elements
y = sin(x);     %Take the sine of x, put it in y
k = 1;          %Initialize our counter variable k with 1
while k < 3;    %For k = 1 and 2
QUILT1(1,:) = x;      %Put x into row 1 of the matrix QUILT1
QUILT2(1,:) = y;      %Put y into row 1 of the matrix QUILT2
QUILT1(2,:) = x;      %Put x into row 2 of the matrix QUILT1
QUILT2(2,:) = -y;     %Put –y into row 2 of the matrix QUILT2
QUILT1(3,:) = -x;     %Put –x into row 3 of the matrix QUILT1
QUILT2(3,:) = y;      %Put y into row 3 of the matrix QUILT2
QUILT1(4,:) = -x;     %Put -x into row 4 of the matrix QUILT1
QUILT2(4,:) = -y;     %Put –y into row 4 of the matrix QUILT2

hold on               %Always plot into the same figure
for i = 1:4           %A nested loop, with i as counter, from 1 to 4
  plot(QUILT1(i,:),QUILT2(i,:))   %Plot the ith row of QUILT1 vs. QUILT2
  pause               %Waiting for user input (key press)
end                   %End of i-loop

for i = 1:4           %Another nested loop, with i as counter, from 1 to 4
  plot(QUILT2(i,:),QUILT1(i,:))   %Plot the ith row of QUILT2 vs. QUILT1
```

```
   pause                %Waiting for user input (key press)
end                     %End of i-loop

y = y + 19;             %Incrementing y by 19 (for every increment of k)
k = k + 1;              %Incrementing k by 1
end                     %End of k-loop
```

Note: This program is the first time we use the **pause** function. If the program pauses after encountering a **pause** statement, press a key to continue until the program is done. This is also the first time that we wrote a program that depends on user input—albeit a very small and limited form—to execute its flow. We will expand on this later.

Note: This program used both **for** and **while** loops. The **for** loops increment their counter automatically, whereas the **while** loops must have their counter incremented explicitly.

Now that you know what the program does and how it operates, you might want to take out the two **pause** functions to complete the following exercises more smoothly.

Exercise 2.17: What happens if you allow the conditional for k to assume values larger than 1 or 2?

Exercise 2.18: Do you know why the program increments y by 19 at the end of the k loop? What happens if you make that increment smaller or larger than 19?

Exercise 2.19: Do you remember how to color your quilt? Try it.

2.4.4. Advanced Plotting

We introduced basic plotting of two-dimensional figures previously. This time, our plotting section will deal with subplots and three-dimensional figures. Subplots are an efficient way to present data. You probably have seen the use of the subplot function in published papers. The syntax of the subplot command is simply **subplot(*a*,*b*,*c*),** where a is the number of rows the subplots are arranged in, b is the number of columns, and c is the particular subplot you are drawing to. It's probably best to illustrate this command in terms of an example. This requires you to open a new program, name it, etc.

Then type the following:

```
figure                    %Open a new figure
for i = 1:9               %Start loop, have counter i run from 1 to 9
  subplot(3,3,i)          %Draw into the subplot i, arranged in 3 rows, 3 columns
  h = bar(1,1);           %This is just going to fill the plot with a uniform color
  set(h,'FaceColor',[0 0 i/9]); %Draw each in a slightly different color
end                       %End loop
```

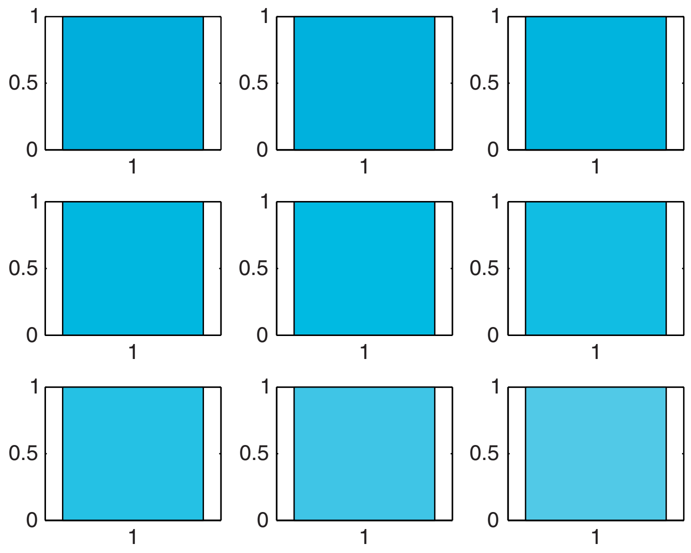


FIGURE 2.8 Color gradient subplots.

This program will draw nine colored squares in subplots in a single figure, specifically, different shades of blue (from dark blue to light blue) and should look something like Figure 2.8.

Note: The three numbers within the square brackets in the **set(h,'FaceColor',[0 0 i/9]);** statement represent the red, green, and blue color components of the bar that is plotted. Each color component can take on a value between 0 and 1. A bar whose color components are [0 0 0] is displayed black and [1 1 1] is white. By setting the color components of the pixels of your image to different combinations of values, you can create virtually any color you desire.

Exercise 2.20: Make the blocks go from black to red instead of black to blue.

Exercise 2.21: Make the blocks go from black to white (via gray).

Suggestion for Exploration: Can you create more complex gradations? It is possible, given this simple program and your recently established knowledge about RGB values in MATLAB as a basis.

Three-dimensional plotting is a simple extension of two-dimensional plotting. To appreciate this, we will introduce a classic example: drawing a two-dimensional exponential function. The two most common three-dimensional plotting functions in MATLAB are probably **surf** and **mesh**. They operate on a grid. Magnitudes of values are represented

FIGURE 2.9 Maximizing a figure.

by different heights and colors. These concepts are probably best understood through an example.

Open a new program in the MATLAB editor, name it, and type the following; then save and run the program:

```
a = -2:0.2:2;              %Creating a vector a with 21 elements
[x, y] = meshgrid(a, a);   %Creating x and y as a meshgrid of a
z = exp (-x.^2 - y.^2);    %Take the 2-dimensional exponential of x and y
figure                     %Open a new figure
subplot(1,2,1)             %Create a left subplot
mesh(z)                    %Draw a wire mesh plot of the data in z
subplot(1,2,2)             %Create a right subplot
surf(z)                    %Draw a surface plot of the data in z
```

After running this program, you probably need to maximize the figure to be able to see it properly. To do this, click the maximize icon in the upper right of your figure (see Figure 2.9; or if using a Mac, click on the green button in the upper left corner). Both the left and right figures illustrate the same data, but in different manners. On the left is a wire mesh; on the right, a surface plot.

If you did everything right, you should see something like that shown in Figure 2.10.

Exercise 2.22: Improve the resolution of the meshgrid. Then redraw.

Exercise 2.23: Can you improve the look of your figure? Try shading it in different ways by using the following:

```
shading interp
```

Now try the following:

```
colormap hot
```

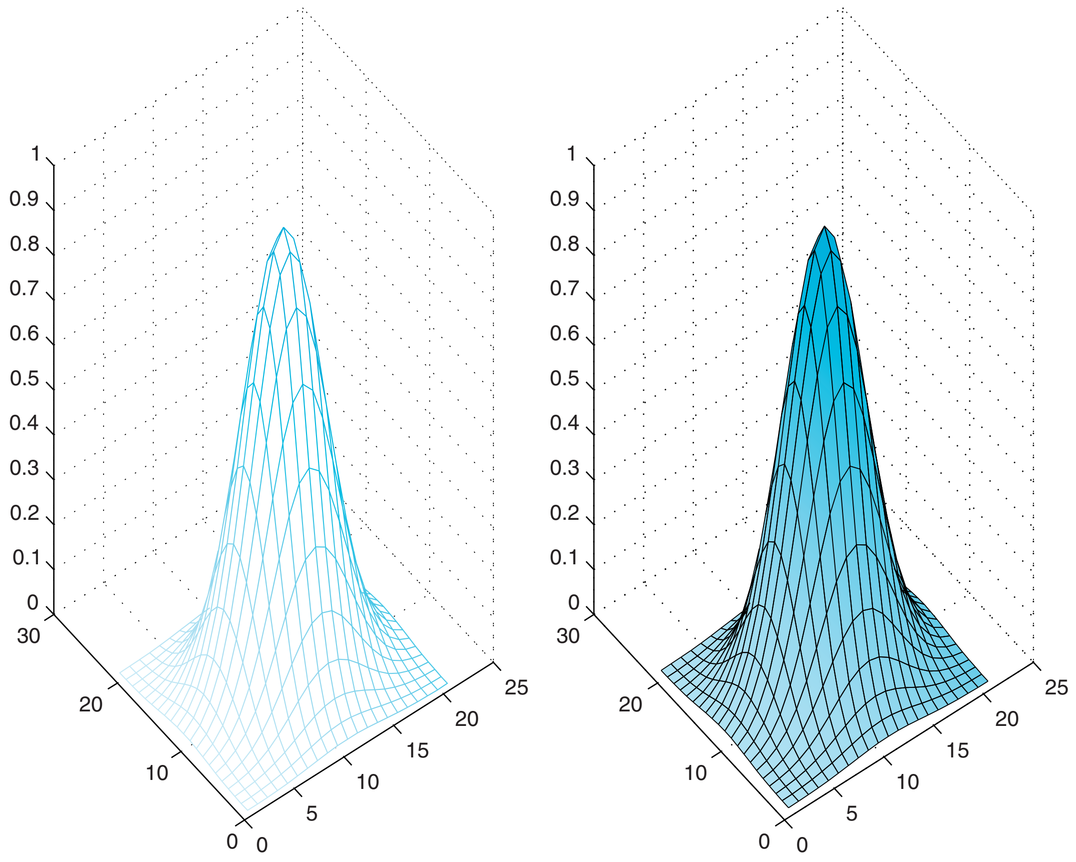


FIGURE 2.10 Three dimensional plotting of a Gaussian.

Suggestion for Exploration: As you can see, meshgrid is extremely powerful. With its help, you can visualize any quantity as a function of several independent variables. This capability is at the very heart of what makes MATLAB so powerful and appealing. Some say that one is not using MATLAB unless one is using meshgrid. While this statement is certainly rather strong, it does capture the central importance of the meshgrid function. We recommend trying to visualize a large number of functions to try and get a good handle on it. Start with something simple, such as a variable that is the result of the addition of a sine wave and a quadratic function. Use meshgrid, then surf to visualize it. This makes for a lot of very appealing graphs.

2.4.5. Interactive Programs

Many programs that are actually useful crucially depend on user input. This input comes mostly in one of two forms: from the mouse or from the keyboard. We will explore both forms in this section.

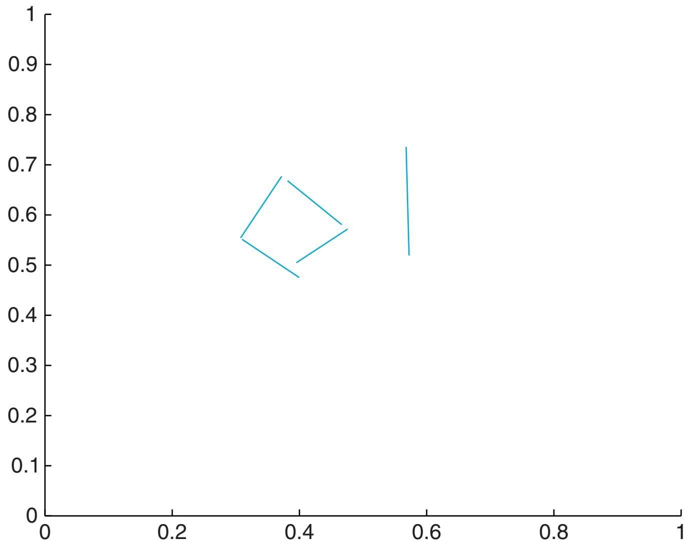


FIGURE 2.11 The luck of the draw.

First, create a program that allows you to draw lines. Open a new program in the editor, write the following code, then save and run the program:

```
figure              %Opens a new figure
hold on;            %Make sure to draw all lines in same figure
xlim([0 1])         %Establish x-limits
ylim([0 1])         %Establish y-limits

for i = 1:5         %Start for-loop. Allow to draw 5 lines
a = ginput(2);      %Get user input for 2 points
plot(a(:,1),a(:,2)); %Draw the line
end                 %End the loop
```

The program will open a new figure and then allow you to draw five lines. When the cross-hairs appear, click the start point of your line and then on the end point of your line. Repeat until you're done. The result should look something like that shown in Figure 2.11.

Exercise 2.24: Allow the program's user to draw 10 lines, instead of five.

Exercise 2.25: Allow the user to draw "lines" that are defined by three points instead of two.

Remember to use **close all** if you opened too many figures.
Of course, most user input will come from the keyboard, not the mouse. So let's create a little program that exemplifies user input very well. In effect, we are striving to re-create the "sugar factory" experiment by Berry and Broadbent (1984). In this experiment, subjects were told that they are the manager of a sugar factory and instructed to keep sugar output

at 12,000 tons per month. They were also told that their main instrument of steering the output is to determine the number of workers per month. The experiment showed that subjects are particularly bad at controlling recursive systems. Try this exercise on a friend or classmate (after you're done programming it). Each month, you ask the subject to indicate the number of workers, and each month, you give feedback on the production so far.

Here is the code:

```
P = [ ];                  %Assigning an empty matrix. Making P empty.
a0 = 6000;                %a0 is 6000;
m0 = 0;                   %m0 is 0;
w0 = 300;                 %w0 is 300;
P(1,:) = [m0, w0, a0];    %First production values
figure                     %Open a new figure
plot(0,a0,'.', 'markersize', 24); %Plot the first value
hold on;                   %Make sure that the plot is held
xlim([0 25])               %Indicate the right x-limits
i = 1;                    %Initialize our counter
while i < 25              %The subject is in charge for 24 months = 2 years.
P                         %Show the subject the production values thus far
a = input('How many workers this month?') %Get the user input
b = 20 * a - a0           %This is the engine. Determines how much sugar is produced
a0 = b;                   %Assign a new a0
plot (i,a0,'.', 'markersize', 24); %Plot it in the big figure
P(i+1,:) = [i, a, b];     %Assign a new row in the P(roduction) matrix
plot (P(:,1),P(:,3),'color','k'); %Connect the dots
i = i + 1;                %Increment counter
end                       %End loop
```

The result (of a successful subject!) should look something like that shown in Figure 2.12.

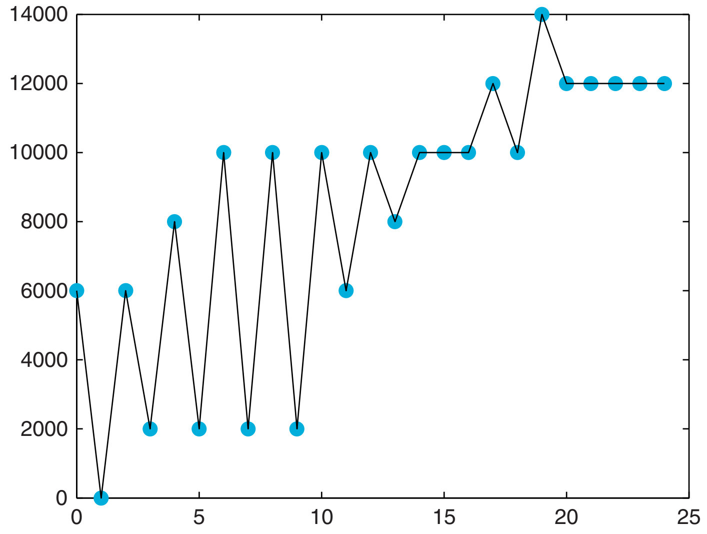


FIGURE 2.12 Game over.

Exercise 2.26: Add more components to the production term, like a trend that tends to increase production over time (efficiency) or decrease production over time (attrition).

Exercise 2.27: Add another plot (a subplot) that tracks the development of the workforce (in addition to the development of production; refer to Figure 2.12).

2.5. DATA ANALYSIS

2.5.1. Importing and Storing Data

Section 2.4.5 described a good way to get data in MATLAB: via user input. Conversely, this section is concerned with data analysis after you have data. One of the primary uses of MATLAB in experimental neuroscience is the analysis of data.

Of course, data analysis is fun only if you already have large amounts of data. Cases in which you will have to manually enter the data before analyzing them will (we hope) be rare. For this scenario, suppose that you are in the marketing department of a major motion picture studio. You just produced a series of movies and asked people how they like these movies.

Specifically, the movies are *Matrix I*, *Matrix II: Matrix Reloaded*, and *Matrix III: Matrix Revolutions*. You asked 1603 subjects how much they liked any of these movies. Subjects were instructed to use a nine-point scale (0 for awful, 4 for great and everything in between, in 0.5 steps). Also, subjects were instructed to abstain from giving a rating if they hadn't seen the movie. Now you will construct a program that analyzes these data, piece by piece. So open a new program in the editor and then add commands as we add them in our discussion here.

Data import: Download the data from the companion website to a suitable directory in your hard disk. Try using a directory that MATLAB will read without additional specifications of a path (file location) in the following code. First, import the data into MATLAB. To do this, add the following pieces of code to your new analysis program:

```
%Data import
M1 = xlsread('Matrix1.xls') %Importing data for Matrix I
M2 = xlsread('Matrix2.xls') %Importing data for Matrix II
```

These commands will create two matrices, M1 and M2, containing the ratings of the subjects for the respective movies. Type **M1** and **M2** to inspect the matrices. You can also click on them in the workspace to get a spreadsheet view. One of the things that you will notice quickly is cells that contain "NaN." These are subjects that didn't see the movie or didn't submit a rating for this movie for other reasons. In any case, you don't have ratings for these subjects, and MATLAB indicates "NaN," which means "not a number"—or empty, in our case. The problem is that retaining this entry will defeat all attempts at data analysis if you don't get rid of these missing values. For example, try a correlation:

```
>> corrcoef(M1,M2)
ans =
  NaN NaN
  NaN NaN
```

You want to know how much an average person likes *Matrix II* if he or she saw *Matrix I* and vice versa. Correlating the two matrices is a good start to answering this question. However, the correlation function (**corrcoef**) in MATLAB assumes two vectors that consist only of numbers, not NaNs. A single NaN somewhere in the two vectors will render the entire answer NaN. This result is not very useful. So the first thing you need to do is to prune the data and retain only those subjects that submitted ratings for both movies.

Data Pruning: There are many ways of pruning data, and the way that we're suggesting here is certainly not the most efficient one. It does, however, require the least amount of introduction of new concepts and is based most on what you already know, namely loops. As a side note, loops are generally slow (compared to matrix operations); therefore, it is almost always more efficient to substitute the loop with such an operation, particularly when calculating things that take too long with loops. We'll discuss this issue more later. For now, you should be fine if you add the following code to the program you already started:

```
%Data pruning
Movies = [ ];           %New Movies variable. Initializing it as empty.
temp = [M1 M2];         %Create a joint temporary Matrix, consisting of two long vectors
k = 1;                  %Initializing index k as 1
for i = 1:length(temp)  %Could have said 1603, this is flexible. Start i loop
if isnan(temp(i,1)) == 0 & isnan(temp(i,2)) == 0 %If both are numbers (=valid)
  Movies(k,:) = temp(i,:);    %Fill with valid entries only
  k = k + 1;            %Update k index only in this case
end                     %End if clause
end                     %End for loop
```

The **isnan** function tests the elements of its input. It returns 1 if the element is not a number and returns 0 if the element is a number. By inspecting M1, you can verify visually that M1(2,1) is a number but that M1(3,1) is not. So you can test the function by typing the following in the command window:

```
>> isnan(M1(2,1))

ans =
  0

>> isnan(M1(3,1))

ans =
  1
```

Recall that the **&** is the MATLAB symbol for logical AND. The symbol for logical OR is **|**. So you are effectively telling MATLAB in the **if** statement that you want to execute the statements it contains only if both vectors contain numbers at that row using **isnan** in combination with **&**.

Exercise 2.28: What would have happened if you had made everything contingent on the index *i*, instead of declaring another specialized and independent index *k*? Would the program have worked?

It's time to look at the result. In fact, it seems to have worked: There is a new matrix, "Movies," which is 805 entries long. In other words, about half the subjects have seen both movies.

After these preliminaries (data import and data pruning), you're ready to move to data analysis and the presentation of the results. The next step is to calculate the correlation you were looking for before, so add that to the code:

```
corrcoef(Movies(:,1),Movies(:,2)) %Correlation between Matrix I and Matrix II
```

The correlation is 0.503. That's not substantial, but not bad, either. The good news is that it's positive (if you like one, you tend to like the other) and that it's moderately large (definitely not 0). To get a better idea of what the correlation means, use a scatterplot to visualize it:

```
figure %Create a new figure
plot(Movies(:,1), Movies(:,2),'.', 'markersize', 24) %Plot ratings vs. each other
```

The result looks something like that shown in Figure 2.13.

The problem is that the space is very coarse. You have only nine steps per dimension—or 81 cells overall. Since you have 805 subjects, it is not surprising that almost every cell is taken by at least one rating. This plot is clearly not satisfactory. We will improve on it later.

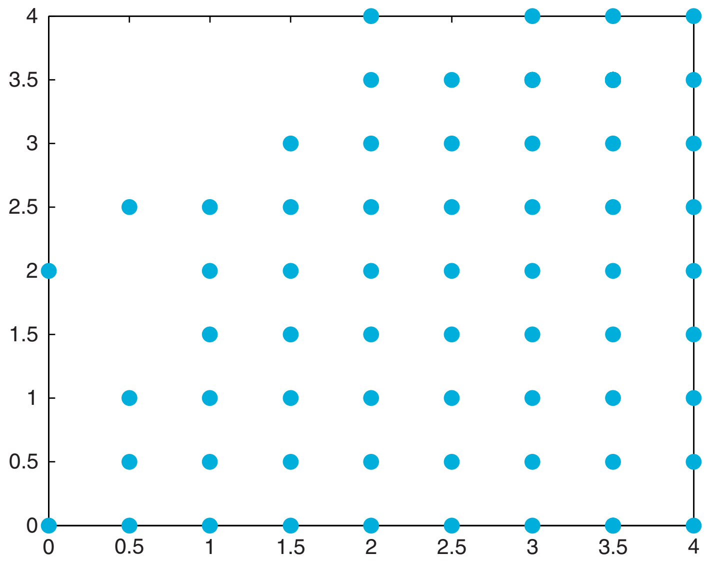


FIGURE 2.13 Low resolution.

The white space on the top left of the figure is, however, significant. It means that there was no one in the sample who disliked the first *Matrix* movie but liked the second one. The opposite seems to be very common.

Let's look at this in more detail and add the following line to the code:

```
averages = mean(Movies) %Take the average of the Movie matrix
```

mean is a MATLAB function that takes the average of a vector. The **averages** variable contains both means.

As it turns out, the average rating for *Matrix I* is 3.26 (out of 4), while the average rating for *Matrix II* is only about 2.28. Figure 2.13 makes sense in light of these data. This can be further impressively illustrated in a bar graph, as shown in Figure 2.14.

However, this graph doesn't tell about the variance among the means. Let's rectify this in a quick histogram. Now add the following code:

```
figure                    %Open new figure
subplot(1,2,1)            %Open new subplot
hold on;                  %Hold the plot
hist(Movies(:,1),9)       %Matrix I data. 9 bins is enough, since we only have 9 ratings
histfit(Movies(:,1),9)    %Let's fit a gaussian
xlim([0 4]) ;             %Let's make sure that plotting range is fine
title('Matrix I')         %Add a title
subplot(1,2,2)            %Open new subplot
hold on;                  %Hold the plot
hist(Movies(:,2),9)       %Matrix II data. 9 bins is enough, since we only have 9 ratings
histfit(Movies(:,2),9)    %Let's fit a gaussian
xlim([0 4]) ;             %Let's make sure that plotting range is fine
title('Matrix II: reloaded') %Add a title
```

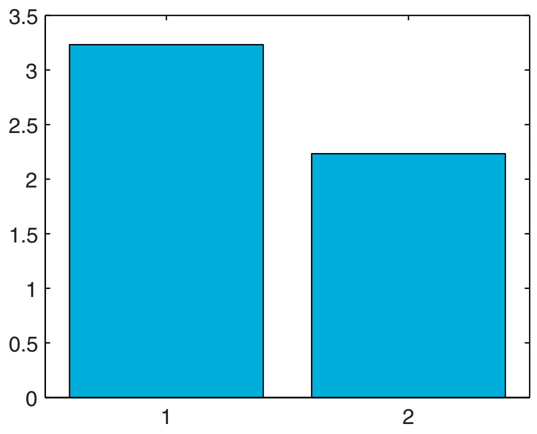


FIGURE 2.14 Means.

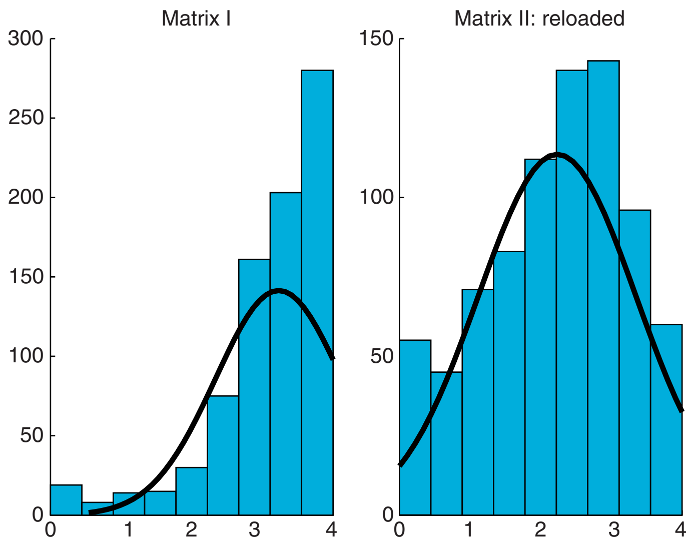


FIGURE 2.15 Variance.

As you can see in Figure 2.15, it looks as though almost everyone really liked the first *Matrix* movie, but the second one was just okay (with a wide spread of opinion). Plus, a fewer people actually report having seen the second movie.

The last thing to do—for now—is to fix the scatterplot that you obtained in Figure 2.13. You will do that by using what you learned about surface plots, keeping in mind that you will have only a very coarse plot (9×9 cells).

Nevertheless, add the following code to the program:

```
MT1 = (Movies(:,1)*2)+1; % Assign a temporary matrix, multiplying ratings by 2 to get
MT2 = (Movies(:,2)*2)+1; %integral steps and adding 1 matrix indices start w/ 1, not 0.
c = zeros(9,9);  %Creates a matrix "c" filled with zeros with the indicated dimensions
i = 1;           %Initialize index
for i = 1:length(Movies) %Start i loop. This loop fills c matrix with movie rating counts
   c(10-MT1(i,1),MT2(i,1)) = c(10-MT1(i,1),MT2(i,1)) + 1; %Adding one in the cell count
end %End loop
figure %New figure
surf(c) %Create a surface
shading interp %Interpolate the shading
xlabel('Ratings for matrix I') %Label for the x-axis
ylabel('Ratings for matrix II: reloaded') %Label for the y-axis
zlabel('Frequency') %Get in the habit of labeling your axes.
```

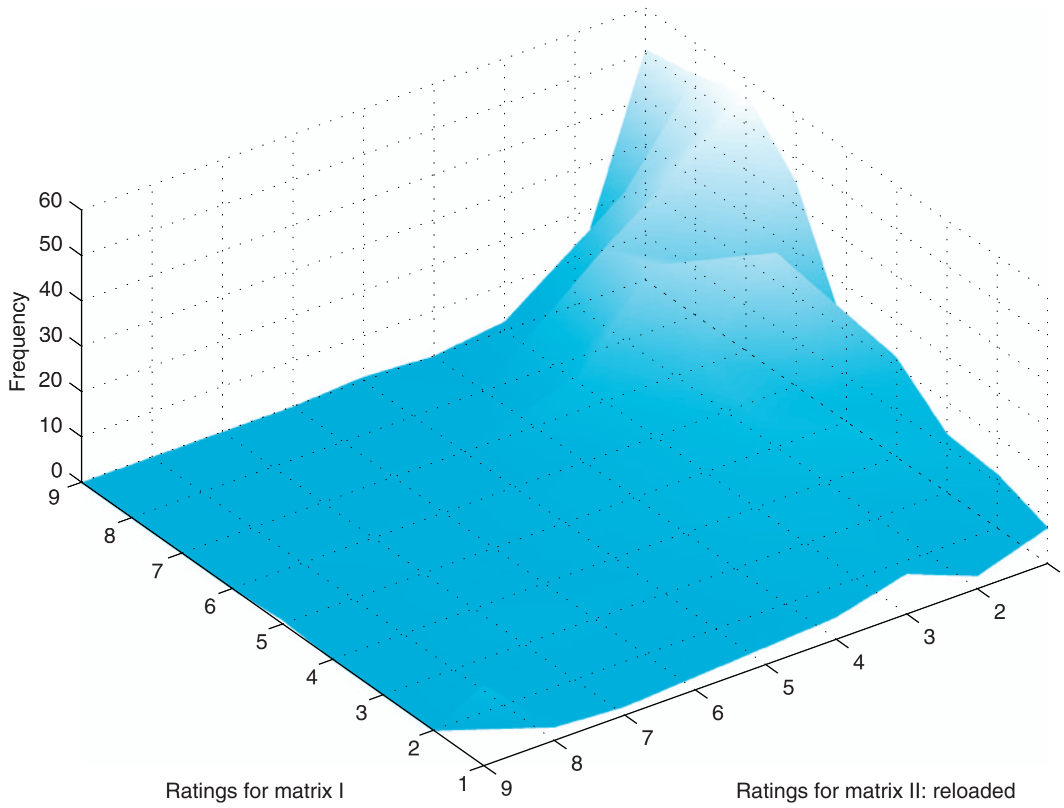


FIGURE 2.16 The real deal.

The result looks rather appealing—something like that shown in Figure 2.16. It gives much more information than the simple scatterplot shown previously—namely, how often a given cell was filled and how often a given combination of ratings was given.

Exercise 2.29: Import the data for the third *Matrix* movie, prune it, and include it in the analysis. In particular, explore the relations between *Matrix I* and *Matrix III* and between *Matrix II* and *Matrix III*. The plots between *Matrix II* and *Matrix III* are particularly nice.

Can you now predict how much someone will like *Matrix II*, given how much he or she liked *Matrix I*? It looks as though you can. But the relationship is much stronger for *Matrix II*→*III*.

2.6. A WORD ON FUNCTION HANDLES

Before we conclude, it is worthwhile to mention function handles, as you will likely need them—either in your own code or when interpreting the code of others.

In this tutorial, we talked a lot about functions. Mostly, we did so in the context of the arguments they take. Up to this point, the arguments have been numbers—sometimes individual numbers, sometimes sequences of numbers—but they were always numbers.

However, there are quite a few functions in MATLAB that expect other functions as part of their input arguments. This concept will take awhile to get used to if it is unfamiliar from your previous programming experience, but once you have used it a couple of times, the power and flexibility of this hierarchical nestedness will be obvious.

There are several ways to pass a function as an argument to another function. A straightforward and commonly used approach is to declare a function handle. Let's explore this concept in the light of specific examples. Say you would like to evaluate the sine function at different points. As you saw previously, you could do this by just typing

sin(*x*)

where *x* is the value of interest.

For example, type

```
sin([0 pi/2 pi 3/2*pi 2*pi])
```

to evaluate the sine function at some significant points of interest.

Predictably, the result is

```
ans =

   0    1.0000    0.0000    -1.0000    -0.0000
```

Now, you can do this with function handles. To do so, type

```
h = @sin
```

You now have a function handle *h* in your workspace. It represents the sine function. As you can see in your workspace, it takes memory and should be considered analogous to other handles that you have already encountered, namely figure handles.

The function **feval** evaluates a function at certain values. It expects a function as its first input and the values to-be-evaluated as the second. For example, typing

```
feval(h,[0 pi/2 pi 3/2*pi 2*pi])
```

yields

```
ans =
   0    1.0000    0.0000    -1.0000    -0.0000
```

Comparing this with the previous result illustrates that passing the function handle worked as expected.

You might wonder what the big deal is. It is arguably as easy—if not easier—to just type the values directly into the **sin** function than to formally declare a function handle.

Of course, you would be right to be skeptical. However—at the very least—you will save time typing when you use the same function over and over again—given that you use function handles that are shorter than the function itself. Moreover, you can create more succinct code, which is always a concern as your programs get longer and more intricate.

More importantly, there are functions that actually do useful stuff with function handles. For example, **fplot** plots a given function over a specified range. Typing

```
fplot(h,[0 2*pi])
```

should give you a result that looks something like that shown in Figure 2.17.

Now let's consider another function that expects a function as input. The function **quad** performs numeric integration of a given function over a certain interval. You need a way to tell **quad** which function you want to integrate. This makes **quad** very powerful because it can integrate any number of functions (as opposed to your writing a whole library of specific integrated functions).

Now integrate the sine function numerically. Conveniently, you already have the function handle *h* in memory. Then type

```
>> quad(h,0,pi)

ans =

   2.0000

>> quad(h,0,2*pi)

ans =

   0
```

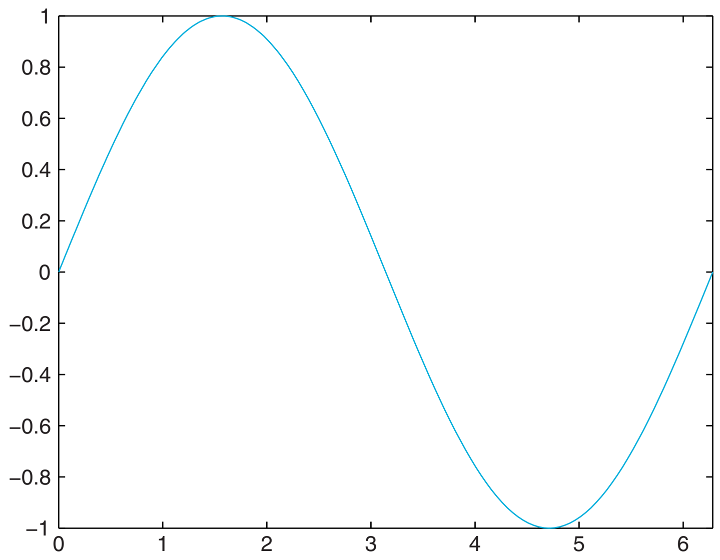


FIGURE 2.17 fplot in action.

```
>> quad(h,0,pi/2)

ans =

   1.0000
```

After visually inspecting the graph in Figure 2.17 and recalling high school calculus, you can appreciate that the **quad** function works on the function handle as intended.

In addition, you can not only tag pre-existing MATLAB functions, but also declare your own functions and tag them with a function handle, as follows:

```
>> q = @(x) x.^5 - 9.* x^4 + 8 .* x^3 - 2.* x.^2 + x + 500;
```

Now you have a rather imposing polynomial all wrapped up and neatly tucked away in the function handle *q*. You can do whatever you want with it. For example, you could plot it, as follows:

```
>> fplot(q,[0 10])
```

The result is shown in Figure 2.18.

Exercise 2.30: Try integrating a value of the polynomial. Does the result make sense?

Exercise 2.31: Do everything you just did, but using your own functions and function handles. Try declaring your own functions and evaluating them.

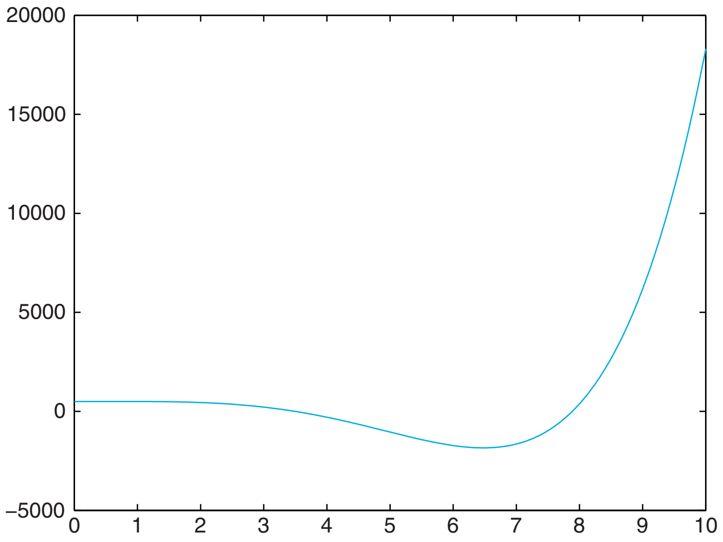


FIGURE 2.18 A polynomial in q, plotted from 0 to 10.

Suggestion for Exploration: Find another function that takes a function handle as input by using the MATLAB help function. See what it does.

Finally, you can save your function handles as a workspace. This way, you can build your own library of functions for specific purposes.

As usual, there are many ways to do the same thing in MATLAB. As should be clear by now, function handles are a convenient and robust way to pass functions to other functions that expect functions as an input.

2.7. THE FUNCTION BROWSER

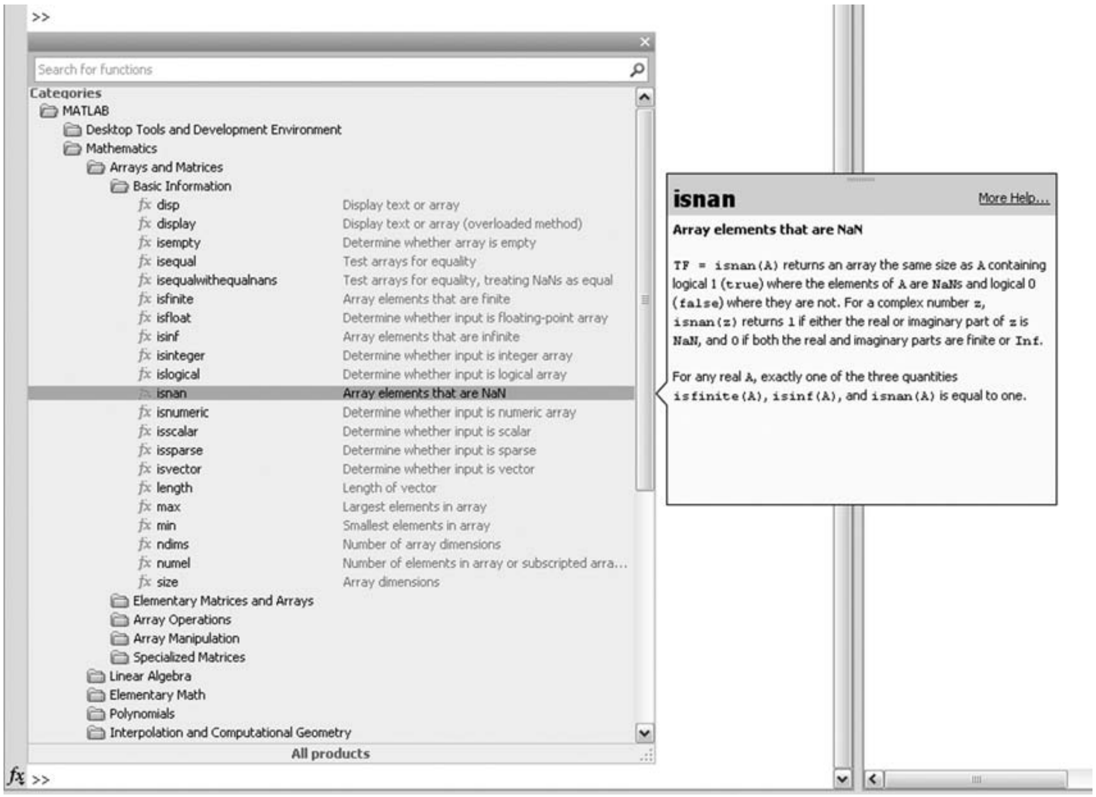
Since release 2008b (7.7), MATLAB contains a function browser. This helps the user to quickly find—and appropriately use—MATLAB functions. The introduction of this feature is very timely. MATLAB now contains thousands of functions, most of which are rarely used. Moreover, the number of functions is still growing at a rapid pace, particularly with the introduction of new toolboxes. Finally, the syntax and usage of any given function may change in subtle ways from one version to the next.

In other words—and to summarize—even experts can't be expected to be aware of all available MATLAB functions as well as their current usage and correct syntax. A crude but workable solution up to this date has been to constantly keep the MATLAB "Help Navigator" open at all times. This approach has several tangible drawbacks. First, it takes up valuable screen real estate. Second, it necessitates switching back and forth between what are essentially different programs, breaking up the workflow. Finally, the Help Navigator window requires lots of clicking, copy-and pasting and the like. It is not as well integrated in the MATLAB software as one would otherwise like.

The new "Function Browser" is designed to do away with these drawbacks. It is directly integrated into MATLAB. You can now see this in the form of a little *fx* that is hovering just left of the command prompt, at the far left edge of the command window. Clicking on it (or pressing Shift and F1 at the same time) opens up the Browser. Importantly, the functions are grouped in hierarchical categories, allowing you to find particular functions even if you are not aware of their name (such as plotting functions). The hierarchical trees can be rather deep, first distinguishing between MATLAB and its Toolboxes, then between different function types (e.g. Mathematics vs. Graphics) and then particular subfields thereof. Of course, the function browser also allows to search for functions by name. Type something in the search function field provides a quick list of functions that match the string that was inputted in the field. The list of functions also gives a very succinct but appropriate short description of what the function does. Hovering over a given entry with the cursor brings up a popup window with a more elaborate description of the function and its usage.

Finally, the function browser allows to drag and drop a given function from the browser into the command window.

To summarize, we expect the function browser to have a dramatic impact on the way people use MATLAB, essentially replacing the use of the Help Navigator for all but the



```
>>
Search for functions
Categories
MATLAB
  Desktop Tools and Development Environment
  Mathematics
    Arrays and Matrices
      Basic Information
        disp                    Display text or array
        display                 Display text or array (overloaded method)
        isempty                 Determine whether array is empty
        isequal                 Test arrays for equality
        isequalwithequalnans    Test arrays for equality, treating NaNs as equal
        isfinite                Array elements that are finite
        isfloat                 Determine whether input is floating-point array
        isinf                   Array elements that are infinite
        isinteger               Determine whether input is integer array
        islogical               Determine whether input is logical array
        isnan                   Array elements that are NaN
        isnumeric               Determine whether input is numeric array
        isscalar                Determine whether input is scalar
        issparse                Determine whether input is sparse
        isvector                Determine whether input is vector
        length                  Length of vector
        max                     Largest elements in array
        min                     Smallest elements in array
        ndims                   Number of array dimensions
        numel                   Number of elements in array or subscripted arra...
        size                    Array dimensions
      Elementary Matrices and Arrays
      Array Operations
      Array Manipulation
      Specialized Matrices
    Linear Algebra
    Elementary Math
    Polynomials
    Interpolation and Computational Geometry
All products
fx >>
```

isnan More Help...

Array elements that are NaN

`TF = isnan(A)` returns an array the same size as `A` containing logical 1 (`true`) where the elements of `A` are `NaN`s and logical 0 (`false`) where they are not. For a complex number `z`, `isnan(z)` returns `1` if either the real or imaginary part of `z` is `NaN`, and 0 if both the real and imaginary parts are finite or `Inf`.

For any real `A`, exactly one of the three quantities `isfinite(A)`, `isinf(A)`, and `isnan(A)` is equal to one.

FIGURE 2.19 The function browser.

most severe problems. It should allow for a quick integration of unknown or unfamiliar function in your code. We recommend to use it whenever necessary.

Figure 2.19. illustrates the use of the function browser for a function introduced in this chapter, isnan.

2.8. SUMMARY

This tutorial introduced you to the functionality and power of MATLAB. MATLAB contains a large number of diverse operators and functions, covering virtually all applied mathematics, with particularly powerful functions in calculus and linear algebra. If you would like to explore these functions, the MATLAB **help** function provides an excellent starting point. You can summon the help with the **help** command. Of course, you will encounter many useful functions in the sections to follow.

Try not to get too frustrated with MATLAB while learning the program and working on the exercises. If things get rough and the commands you entered don't produce the expected results, know that MATLAB is able to provide much needed humor and a succinct answer to why that is. Just type in the command **why**.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

help	**load**	**cos**
helpwin	**clear**	**close**
helpdesk	**length**	**title**
helpbrowser	**size**	**set**
+	**linspace**	**FaceColor**
-	**logspace**	**linewidth**
*****	**'**	**rref**
/	**./**	**loglog**
()	**.***	**semilogx**
^	**.^**	**semilogy**
log	**find**	**stairs**
exp	**==**	**pie**
sin	**~=**	**sound**
pi	**<**	**function**
format	**>**	**for**
e	**<=**	**while**
[ ]	**>=**	**end**
:	**&**	**%**
;	**|**	**if**
=	**~**	**else**
eye	**xor**	**pause**
ones	**any**	**subplot**
zeros	**all**	**surf**
rand	**plot**	**mesh**
randn	**bar**	**meshgrid**
who	**hist**	**shading**
whos	**figure**	**colormap**
save	**hold**	**xlim**

ylim	**isnan**	**@**
ginput	**histfit**	**quad**
markersize	**xlabel**	**why**
corrcoef	**feval**	
xlsread	**fplot**	

CHAPTER

3

Visual Search and Pop Out

3.1. GOALS OF THIS CHAPTER

The primary goal of this chapter is to collect and analyze reaction time data using the MATLAB® software. Reaction time measures to probe the mind have been the backbone of experimental psychology at least since the time of Donders (1868). The basic premise underlying the use of reaction times in cognitive psychology is the assumption that cognitive operations take a certain and measurable amount of time. In addition, it is assumed that additional mental processes add (more or less) linearly. If this is the case, increased reaction times reflect additional mental processes. Let us assume for the time being that this is a reasonable framework. Then, it is highly useful to have a program that allows you to quickly collect reaction time data.

3.2. BACKGROUND

Understanding how the mind/brain decomposes a sensory scene into features is one of the fundamental problems in experimental psychology and systems neuroscience. We take it for granted that the visual system, for example, appears to decompose objects into different edges, colors, textures, shapes, and motion features. However, it is not obvious a priori which features actually represent primitives that are encoded in the visual system. Many neurophysiological experiments have searched for neurons that are tuned to features that were chosen somewhat arbitrarily based on the intuitions of the experimentalists.

Psychologists, however, have developed behavioral experiments by which feature primitives can be revealed. In particular, a study by Treisman and Gelade (1980) has been particularly influential. This is probably due to the fact that it is extremely simple to grasp, yet the pattern of results suggests provocative hypotheses about the nature of perception (e.g., feature primitives, serial search, etc.).

So what is the visual search and pop-out paradigm that was used in the Treisman study?

Subjects are asked to report the presence or absence of a target stimulus (in this case, a colored lowercase letter “o”) among various numbers of distracter stimuli. If the distracter this delay for granted. Try it. In other words, write a program (M-file) that contains only the following lines and execute it:

```
tic
toc
```

In my case, **toc** reported 0.000004 seconds.

You can now create some code to test if MATLAB takes equal amounts of time to increment an index or if it depends on the magnitude of the index. So open a new M-file and enter the following:

```
format long; %We want to be able to see short differences in time
i = 1; %Initializing the index, i
t = [ ]; %Initializing the matrix in which we will store the times
while i < 11 %Starting loop
tic       %Starting stop-watch
i = i + 1; %Incrementing index
t(i,1) = toc ; %Ending stop-watch and putting respective time into the matrix
end       % End the loop
```

Try to run the program. It should execute rather quickly.

Now create a little plot by typing the following on the command line:

```
>> figure
>> plot (t)
```

The result should be fairly reproducible but the exact shape of the curve as well as the absolute magnitude of the values depends on the computer and its speed.

The result should look something like that shown in Figure 3.2.

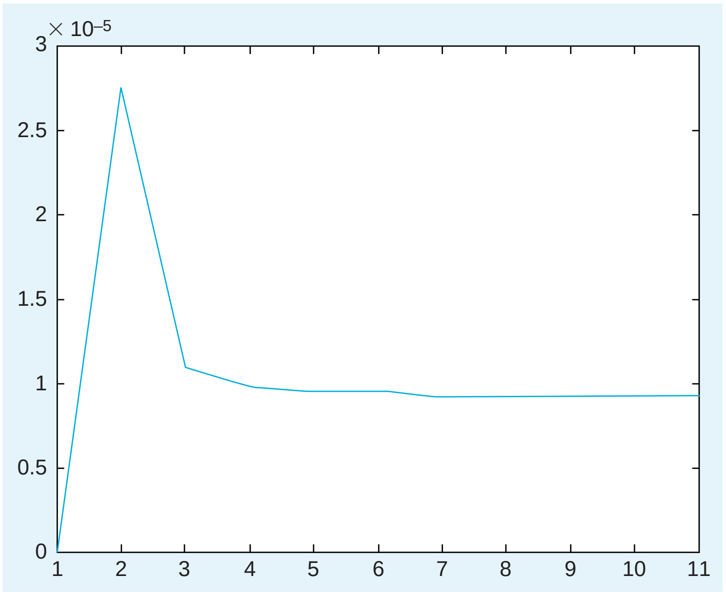


FIGURE 3.2 Task timing.

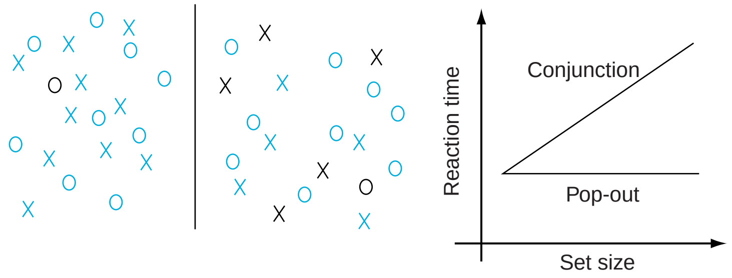


FIGURE 3.1 The pop-out task.

stimuli are just of a different color—that is, if they differ by a single feature—you usually find the "pop-out" effect: the reaction time to detect the target is independent of the number of distracters. Conversely, if more than one stimulus dimension has to be considered to distinguish targets and distracters (conjunction search), you typically find a linear relationship between reaction time and the number of distracters. See Figure 3.1.

As pointed out previously, this pattern of results immediately suggests the existence of "feature primitives," fundamental dimensions that organize and govern human perception as well as a serial scanner in the case of conjunction search, where one element of the stimulus set after the other is considered as a target (and confirmed or discarded). Often, the ratio of the slopes between conditions where the target is present versus where the target is absent suggests a search process that self-terminates once the target is found.

There are many, many potential confounds in this study (luminance, eye movements, spatial frequency, orientation, etc.). However, the results are extremely robust. Moreover, the study was rather influential. Hence, we will briefly replicate it here.

3.3. EXERCISES

In this section, we introduce and review some code that will help you to complete the project assignment in section 3.4. The first thing you need to be able to gather reaction time measures is a way to measure time. There are different ways to measure time in MATLAB. One of the most convenient (and, for our purposes, sufficient ones) comes in the form of the functions **tic** and **toc**. They work in conjunction and effectively implement a stopwatch. Try the following on the command line:

```
>> tic
>> toc
```

What is your elapsed time?

The time reported by MATLAB is the time between pressing Enter after the first statement and pressing Enter after the second statement.

Of course, operating in the real physical world, MATLAB also takes some time to execute the code itself. In most cases, this delay will be negligible. However, you should not take

As you can see, after an initial transient, the time does not depend on the actual value of the index, unlike most human mental processes (e.g., Shepard and Metzler, 1971). This could be taken as evidence for a different kind of information processing in man versus machine.

If you want to know the average time it took for the index to increment, type

>> mean(*t*)

If you want to know the maximum and minimum times, you can type **max(*t*)** or **min(*t*)**, respectively.

This example also illustrates several important points. First, when making an inductive claim about all cases all the time, you should sample a substantial range of the problem space (complete would be best). In this case, incrementing an index 10 times is not very impressive. What about incrementing it 100,000 times?

Exercise 3.1: Increment your index 100,000 times. What does the resulting graph look like? It should look something like the result shown in Figure 3.3.

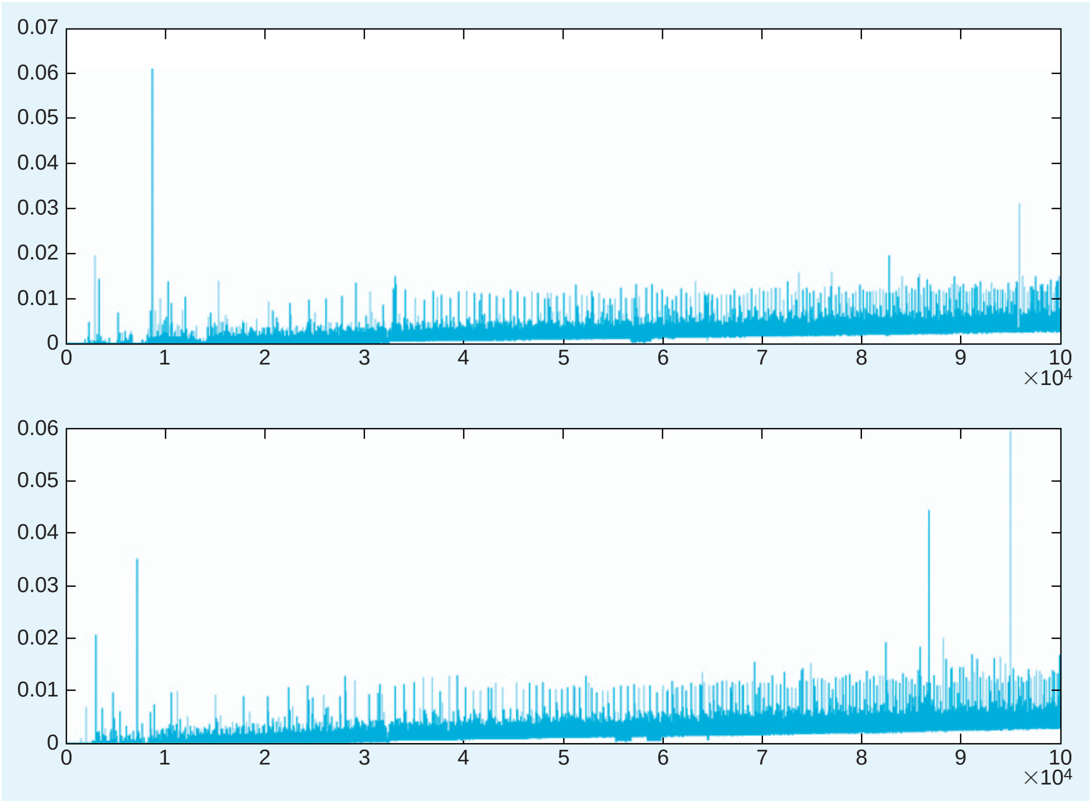


FIGURE 3.3 Task timing revisited.

Hence, we discourage premature conclusions on a scant database. Second, it shows that while MATLAB inherently takes care of "plumbing issues" such as memory management or the representation of variables, it cannot avoid the consequences of physical processes. In other words, they might very well impact the execution of your program. Therefore, you should always check that the program is doing what you think it is doing. The structure of the peaks and their robust nature (the subplots are different runs of the program) indicate that they are reliable and not just random fluctuations, probably induced by MATLAB having to change the internal representation of the variable *t*, which takes longer and longer as it gets larger. This makes sense because MATLAB is shuffling an increasingly large array around in memory, looking for larger and larger chunks thereof. Some of the observed spikes in time taken appear to be distributed largely at random, mostly due to other things going on with the operating system.

Finally, it is a lesson on how to avoid problems like this—namely by preallocating the size and representation of *t* in memory, if the final size is known in advance.

***Exercise* 3.2:** Replace the **line t = [];** with **t = zeros(100000,1);** to preallocate the size of the variable in memory. Then run it.

The result should look something like that shown in Figure 3.4.

There are still some issues left, but nowhere near as many as there were before. As you can see, the problem largely goes away (you should also close all programs other than MATLAB when running time-sensitive code). Note also the dramatic difference in the time needed to execute the program. The reason for this is the same—namely memory management. Therefore, if you can, always preallocate your variables, particularly when you know their size in advance and if their size is substantial.

If you'd like to, you can save this data (stored in the *t* variable) by clicking on the File menu from the MATLAB command window and then clicking on the Save workspace as entry. You can give your file any name you like. Later, you can import the data by clicking on Open in the File menu from the MATLAB command window (not the editor). Try this now. Save your workspace, clear it with **clear all**, and then open it again.

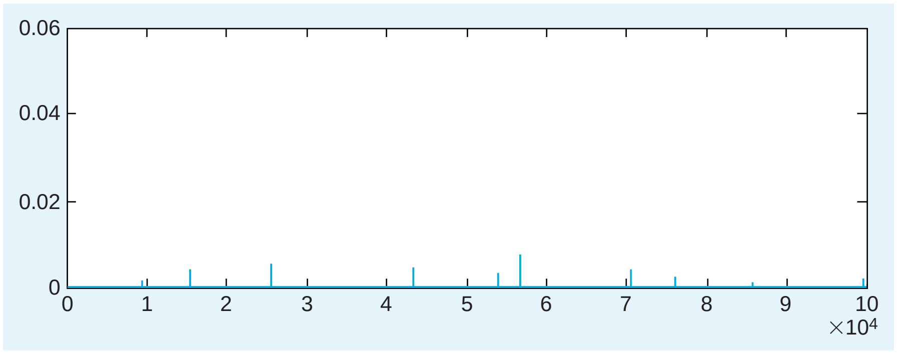


FIGURE 3.4 Problem solved. Mostly.

You can also use the **tic-toc** stopwatch to check on MATLAB. For example, the following code is supposed to check (use another M-file) whether MATLAB really takes a 0.5 second break:

```
tic %Start stopwatch
pause ( 0.5 ) %Take a 0.5(?) second break
toc %End stop-watch
```

By running this program several times, you can get a sense of accuracy and variance (precision) within internal timers in MATLAB. So much for time and timing.

What is still missing at this point is a way to handle random events. In the design of experiments, randomness is your friend. Ideally, you want everything that you don't vary systematically to behave randomly (effectively controlling all other variables, including unknown variables and unknown relations between them).

You encountered the random number generator earlier. Now you can utilize it more systematically. This time, it will be enough to generate random numbers with a uniform distribution. This is achieved by using the function **rand( )**. Remember that the function **randn( )** will generate random numbers drawn from a normal distribution. Conversely, **rand( )** draws from a uniform distribution between 0 and 1. This distinction is important to know, since you can use this knowledge to create two events that are equally likely. Now start a new M-file and add the following code:

```
a = rand(1,1) %Creates a random number and assigns it to the variable a
if a > 0.5 %Check if a is larger than "0.5"
b = 1 %We assign the value "1" to the variable b
else %If not,
b = 0 % We assign the value "0" to the variable b
end %End the condition check
```

Run this code a couple of times and see whether different random values are created every time. Note that this is a rather awkward—but viable—way to create integral random numbers. Recent versions of MATLAB include a new function **randi**, which draws from a uniform discrete distribution. For example, the command randi(2,30,1) yields a single column vector with 30 elements randomly drawn to be 1 or 2. This new function is a good example of how innovation in MATLAB versions makes previously accepted ways of doing things (such as generating discrete random numbers) obsolete. The old way still works, but the new one is much more elegant.

Next, we will introduce several functions and concepts that will come in handy when you are creating your program for the project in Section 3.4. The first is the concept of *handles*. A handle typically pertains to an object or a figure, for our purposes.

It is simply declared as follows:

```
h = figure
```

This creates a figure with the handle *h*. Of course, the name can be anything. Just be sure to remember which handle refers to which figure:

```
thiscanbeanything = figure
```

This creates the handles as variables in the workspace. You can check this by typing **whos** or simply by looking in the workspace window.

Handles are extremely useful. They literally give you a handle on things. They are the embodiment of empowerment.

Of course, you probably don't see that yet because handles are relatively useless without two functions that go hand in hand with the use handles. These functions are **get** and **set**.

The **get** function gives you information about the properties that the handle currently controls as well as the values of these properties. Try it. Type **get(h).**

You should get a long list of figure properties because you linked the handle *h* to a particular figure earlier. These are the properties of the figure that you can control. This capability is extremely helpful and implemented via the **set** function.

Let's say you don't like the fact that the pointer in your figure is an arrow. For whatever reasons you have, you would like it to be a cross-hair. Can you guess which figure property [revealed by **get(h)**] controls this? Try this:

```
set(h, 'Pointer', 'crosshair')
```

How do you know which property takes which values? That is something that you can find in the MATLAB help, under Figure properties. Don't be discouraged about this. It is always better to check. For example, Mathworks recently eliminated the former figure property "fullcrosshairs" and renamed it "fullcross".

Of course, some of the values can be guessed, such as the values taken by the property **visible—**namely **on** and **off**. This also illustrates that the control over a figure with pointers is tremendous.

Try **set (h,'visible','off')** and see what happens. Make sure to put character but not number values between ' '.

Of course, handles don't just pertain to figures; they also pertain to objects. Make the figure visible again and put some objects into it.

An object that you will need later is **text**. So try this:

```
g = text (0.5, 0.5, 'This is pretty cool')
```

If you did everything correct, text should have appeared in the middle of your figure. Text takes at least three properties: x-position, y-position, and the actual text.

But those are not all the properties of text. Try **get(g)** to figure out what you can do.

It turns out, you can do a lot. For example, you can change color and size of the text:

```
set(g,'color','r', 'fontsize', 20)
```

Also, note that this object now appears as a "child" of the figure **h**, which you can check with the usual method.

Now you should have enough control over your figures and objects to complete the project in Section 3.4.

Finally, you need a way for the user (i.e., the subject or participant of the experiment) to interact with the program. You can use the **pause** function, which waits for the user to press a key before continuing execution of the program. In addition, the program needs to identify the key press.

So type this:

```
pause %Waiting for single key press
h725 = get(h,'CurrentCharacter')
```

The variable *h725* should contain the character with which you overcame **pause**. Interestingly enough, it is a figure property that allows to retrieve the typed character in this case, but that is one of the idiosyncrasies of MATLAB. Be sure to do this within an m-file.

Another function you will need (to be able to analyze the collected data) is **corrcoef**. It returns the Pearson correlation between two variables, e.g.,

```
a = rand(100,2); %Creates 2 columns of 100 random values each, puts it in variable a
b = corrcoef(a(:,1),a(:,2)); %Calculates the Pearson correlation between the two columns
```

In my case, MATLAB returns a value close to 0, which is good because it shows that the random number calculator is doing a reasonable job only if we already assume that we know what the random generator is doing.

```
b =
   1.0000     0.0061
   0.0061     1.0000
```

Corrcoef as a function can also take several parameters:

```
[magnitude, p] = corrcoef(a(:,1),a(:,2)) %Same as before, but asking for significance

magnitude =

   1.0000     0.0061
   0.0061     1.0000

p =

   1.0000     0.9522
   0.9522     1.0000
```

According to MATLAB, there is a probability of 0.95 that the observed values were obtained by chance alone (which is, of course, the case). Hence, you can conclude that the correlation is not significant.

By convention, correlations with p values below 0.05 are called "significant."

The final function concerns checking of the equality of variables. You can check the equality of numbers simply by typing == (two equal signs in a row):

```
>> 5 == 6
ans =
  0

>> 5 == 5
ans =
  1
```

Since MATLAB 7, this technique is also valid for checking the equality of characters; Try this:

```
>> var1 = 'a'; var2 = 'b'; var1 == var2
>> var1 == var1
```

The more conventional way to check the equality of characters is to use the **strcmp** function:

```
>> strcmp(var1,var2)
>> strcmp(var2,var2)
```

Together with your knowledge from Chapter 2, "MATLAB Tutorial," you now have all the necessary tools to create a useful experimental program for data collection.

3.4. PROJECT

Your project is to implement the visual search paradigm described in the preceding sections in MATLAB. Specifically, you should perform the following:

- Show two conditions (pop-out search versus conjunction search) with four levels each (set size = 4, 8, 12, 16). These conditions can be blocked (first all pop-out searches, then all conjunction searches or something like that).
- It is imperative to randomly interleave trials with and without target. There should be an equal number of trials with and without targets.
- Make sure that the number of green and red stimuli (if you are red/green blind, use blue and red) is balanced in the conjunction search (it should be 50%/50%). Also, make sure that there is an equal number of *x* and *o* elements, if possible.
- Use only correct trials (subject indicated no target present when no target was presented or indicated target present when it was present) for the analysis.
- Try to be as quick as possible while making sure to be right. It would be suboptimal if you had a speed/accuracy trade-off in your data.
- The analysis should contain at least 20 correct trials per level and condition for a total of 160 trials. They should go quickly (about 1 second each).
- Pick two keys on the keyboard to indicate responses (one for target present, one for target absent).
- Report and graph the mean reaction times for correct trials as a function of pop-out search versus conjunction search and for trials where the target is present versus where the target is absent. Hence, you need between two and four figures. You can combine graphs for comparison (see below).
- Report the Pearson correlation coefficients between reaction time and set size and indicate whether it is significant or not (for each condition).
- Make a qualitative assessment of the slopes in the different conditions (we will talk about curve fitting in a later chapter.

Hints:

- Start writing one trial and make sure it works properly.
- Be aware that you effectively have an experimental design with three factors [Set size: 4 levels (4, 8, 12, 16), conjunction versus feature search: 2 levels, target present versus absent: 2 levels). It might make sense to block the first two factors and randomize the last one.

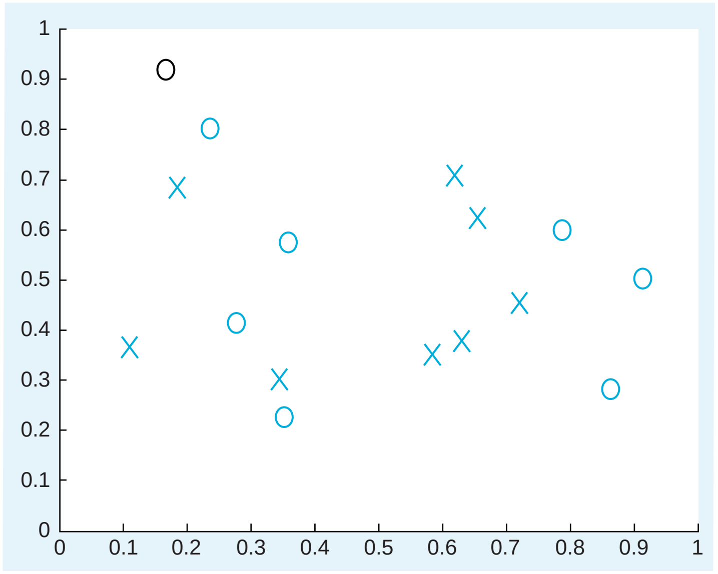


FIGURE 3.5 The display.

- Be sure to place the targets and distracters randomly.
- Start by creating a figure.
- Each trial will essentially consist of newly presented, randomly placed text.
- Be sure to make the figure big enough to see it clearly.
- Make sure to make the text vanish before the beginning of the next trial.
- Your display should look something like Figure 3.5.
- Determine reaction time by measuring time from appearance of target to user reaction.
- Elicit the key press and compare with the expected (correct) press to obtain a value for right and wrong answers.
- Put it into a matrix, depending on condition. It's probably best to have as many matrices as conditions.
- Plot it.
- Write a big loop that goes through trials. Do this at the very end, if individual trials work.
- You might want to have a start screen before the first trial, so as not to bias the times of the first few trials.
- If you can't do everything at once, focus on subgoals. Implement one function after the other. Start with two conditions.
- Figure 3.6 shows one of the exemplary result plots from a subject. Depicted is the relationship between mean reaction time and set size for trials where a target is present (only correct trials). Red: Conjunction search. Blue: Pop-out search. Pearson *r* for conjunction search in the data above: 0.97.

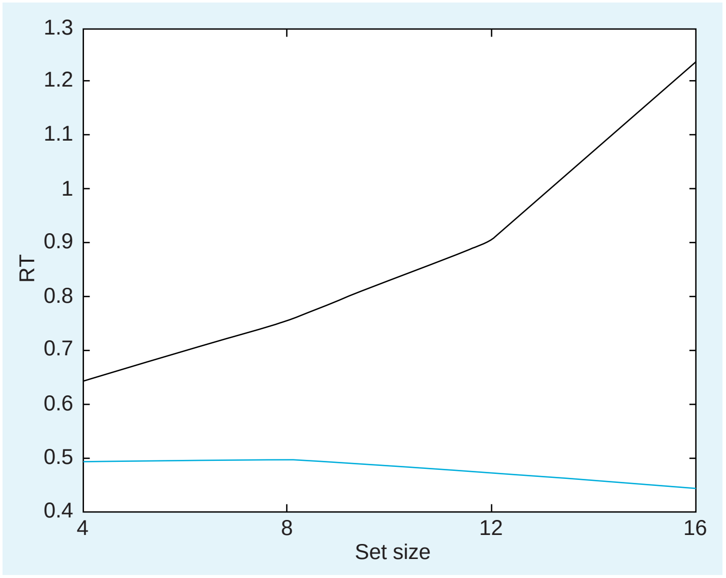


FIGURE 3.6 Typical results.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

clear all
tic
toc
mean
min
max
pause
rand
randi
==
strcmp
whos
get
set
text
corrcoef
pointer
fullcrosshair
visible
fontsize
CurrentCharacter

CHAPTER

4

Attention

4.1. GOALS OF THIS CHAPTER

The primary goals of this chapter are to consolidate and generalize what you learned in Chapter 3, "Visual Search and Pop Out," about data collection in the MATLAB® software. Moreover, we will elaborate on data analysis in MATLAB beyond simple correlations. You will also learn how reaction time data can be used to infer the mental process of spatial attention.

4.2. BACKGROUND

As the famous American psychologist James pointed out over 100 year ago, we all have a strong intuition what attention is:

> Everyone knows what attention is. It is the taking possession by the mind, in clear and vivid form, of one out of what seem several simultaneously possible objects or trains of thought. Focalization, concentration of consciousness are of its essence. It implies withdrawal from some things in order to deal effectively with others, and is a condition which has a real opposite in the confused, dazed, scatterbrained state. . . . (James, 1890, p. 403)

The idea of attention as a process by which mental resources can be concentrated or focused continues to pervade thinking in the scientific study of attention. Psychologists and neuroscientists have divided the concept into three different forms: space-based, object-based, and feature-based attention. In this chapter, we will focus on spatial attention. Helmholtz (1867) was one of the first experimentalists to demonstrate that one could covertly (i.e., by holding the eyes fixed) shift one's attention to one part of space prior to presentation of a long list of characters. He found that one could more effectively recollect the characters within the region of space to which the "attentional search light" was shifted.

In the modern study of attention, the Posner paradigm (Posner, 1980) has been particularly influential. This is likely owed to the fact that it is extremely simple to grasp, yet the pattern of results has potentially far-reaching implications for our understanding of spatial attention in mind and brain. In particular, this paradigm has been used to quantify the attentional deficits in patients with parietal-lobe damage (i.e., parietal hemi-neglect syndrome), leading to the theory that spatial attentional mechanisms may be localized in the parietal cortex.

4.2.1. So What Is the Posner Paradigm?

In the study of the Posner paradigm, subjects are asked to fixate in the center of the screen and not to break fixation for the duration of the trial. Then, a location on the screen is cued in some way (usually by highlighting or flashing something). After the cue, a target appears in either the cued location or in another location. Subjects are instructed to press a key as soon as they see the target. Figure 4.1 provides a schematic illustration of the paradigm:

Posner (1984) found that if the cue is valid, reaction time was substantially lower than if it was invalid. He interpreted this in terms of an "attentional spotlight" that is focused on a certain region in space and permanently shifting at a finite and measurable speed.

4.3. EXERCISES

Most of the functions needed to write software that allows you to gather reaction time data were already introduced in Chapter 3, "Visual Search and Pop Out." This time, we will introduce some functions that allow you to generalize the kinds of conditions in which such data are collected. To this end, we introduce another drawing function, **rectangle,** that will come in handy when creating your program in Section 4.4.

Try this code:

```
figure %Create a new figure
xlim([0 1]) %Set the range of values on the x-axis to (0 to 1)
ylim([0 1]) %Set the range of values on the y-axis to (0 to 1)
rectangle('Position', [0.2 0.6 0.5 0.2]) %Create a rectangle at the x-position 0.2, y-position
                                          %0.6 with an x-width of 0.5 and a y-height of 0.2
```

If you declare **rectangle** with a handle, you can change all properties of the rectangle. Try it. Rectangles have some interesting properties that can be changed.

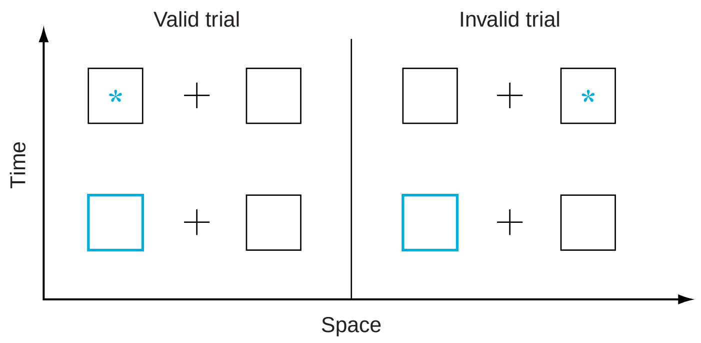


FIGURE 4.1 The Posner paradigm.

Regarding data analysis, the most important function we can introduce at this point is the t-test. MATLAB uses **ttest2** to test the hypothesis that there is a difference in the mean of two independent samples.

Consider this:

```
A = rand(100,1); %Create a matrix A with 100 random elements in one column
 B = rand(100,1); %Create a matrix B with 100 random elements in one column
[significant,p] = ttest2(A,B)
```

MATLAB should have returned:

```
      significant = 0
                                 p = 0.6689
```

This means that the null hypothesis was kept because you failed to reject it. You failed to reject it because the observed difference in means (given the null hypothesis is true) had a probability of about 0.67, which is far too high to reject the null hypothesis. This is what you should expect if the random number generator works. Now try this test:

```
B = B .* 2;
[significant,p] = ttest2(A,B)
```

```
significant = 1
p = 3.3467e-013
```

Now, the null hypothesis is rejected. As a matter of fact, the p-value is miniscule.

Note on seeding the random number generator: If you use the **rand()** function just as is, the SAME sequence of pseudorandom numbers will be generated each session. You can avoid this by seeding it first like this: **rand('state',number)**. It is important to note that the "random number generator" does no such thing. As a matter of fact, all numbers generated are perfectly deterministic, given the same seed number. We don't want to go on a tangent why this has to be the case or how to avoid this by relying on a genuinely random (at least as far as we can tell) natural process (such as radioactive decay). As long as you pick a different number as a seed each time, you should be fine, for all common intents and purposes. Hence, it is popular to make the number after the state argument dependent on the cpu-clock. In old versions of MATLAB (e.g. 7.04), this could be done as follows:

```
rand('state', sum(100*clock))
```

New versions of MATLAB (e.g. 7.7) rely on a very different system. Namely the notion of a random number stream that underlies rand, randn and randi. This random number stream is implemented as **randstream**.

To seed the generator in new versions of MATLAB, things are more complicated but also more versatile.

First type
RandStream.list
to get a list of available pseudorandom number generation methods. Mersenne twister with Mersenne prime 2^19937-1 sounds appealing.
Now type

```
s = Randstream('mt19937ar', 'seed', sum(100*clock))
```

Note the value of "Seed"
Now type

```
RandStream.setDefaultStream(s);
```

These changes are due to the fact that MATLAB is becoming increasingly object oriented. We are now handling Randstream "objects". Expect to see more of this in the future. For the project in Section 4.4, it might be useful to know at least one other comman data analysis function, namely **ANOVA** (analysis of variance). ANOVA generalizes the case of a two-sample t-test to many samples. For the purposes of this chapter, a one-way ANOVA will be sufficient:

```
A = rand(100,5); %Generating 5 levels with 100 repetitions each.
anova1(A); %Do a one-way balanced ANOVA.
```

In this case, there were no significant differences, as revealed by Table 4.1 and Figure 4.2.
Now try this:

```
B = meshgrid(1:100); %Generate a large meshgrid
B = B(:,1:5); %We only need the first five columns
```

TABLE 4.1 ANOVA Table 1

Source	SS	df	MS	F	Prob>F
Columns	0.288	4	0.072	0.81	0.5221
Error	44.2525	495	0.0894		
Total	44.5405	499			

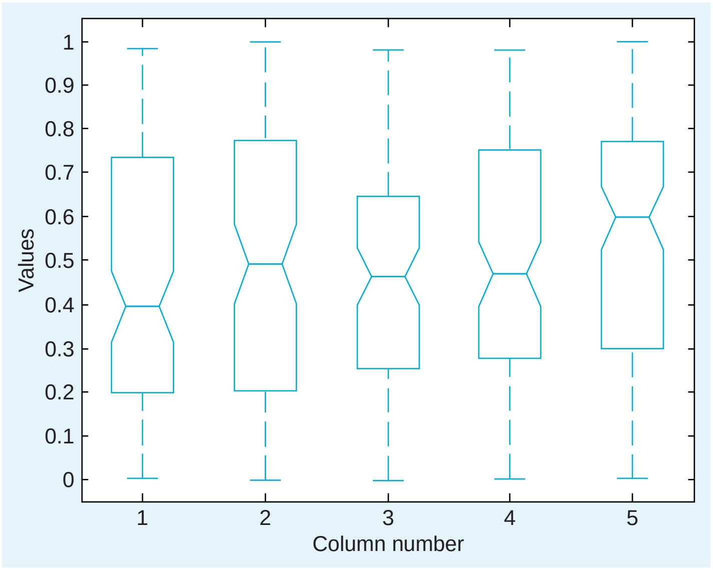


FIGURE 4.2 N.S.

```
A = A .* B; %Multiply!
anova1(A); %Doing the one-way balanced ANOVA again
```

This time, there can be no doubt that there is a positive trend, as you can see in Table 4.2 and Figure 4.3.

The **anova1** function assumes that different samples are stored in different columns and that different rows represent different observations in the same sample.

Note that **anova1** assumes that there is an equal number of observations in each sample. For more generalized ANOVAs or unequal samples, see **anova2** or **anovan**. Their syntax is very similar. This, however, should not be necessary for the following project.

4.4. PROJECT

For this project, your task is to replicate a generalized version of the Posner paradigm. In essence, you will measure the speed of the "attentional spotlight" in the vertical versus

TABLE 4.2 ANOVA Table 2

Source	SS	df	MS	F	Prob>F
Columns	207.519	4	51.8797	50.39	0
Error	509.66	495	1.0296		
Total	717.179	499			

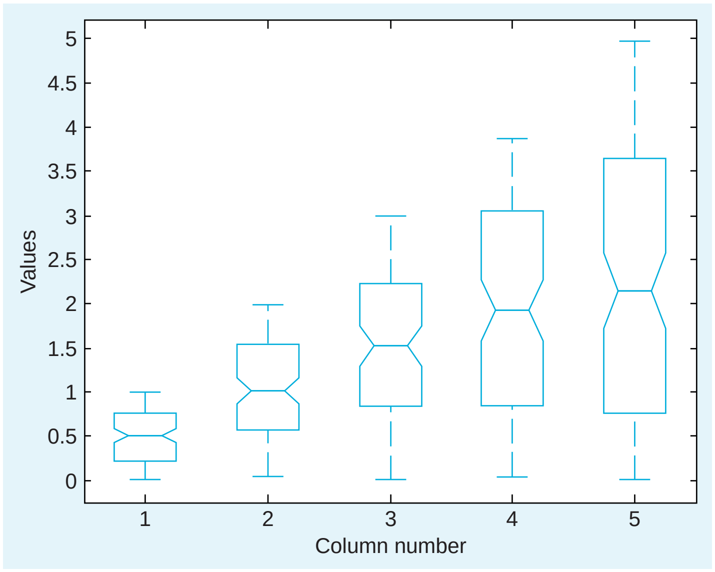


FIGURE 4.3 An effect.

horizontal directions. You need to create a program that allows you to gather data on reaction times in the Posner paradigm as described in the preceding sections. Most of the particular implementation is up to you (the nature of the cue, specific distances, etc.). However, be sure to implement the following:

- Cue and target must appear in one of 16 possible positions. See, for example, Figure 4.4.
- Make sure you have an equal number of valid and invalid trials. [If the trial is valid, the target should appear in the position of the cue. If the trial is invalid, the target position should be picked randomly (minus 1, the position of the cue).]
- Choose two temporal delays between cue and target: 100 ms and 300 ms. Make the delay an experimental condition.
- Collect data from 80 trials per spatial location of the cue (so that you have 20 for each combination of conditions: Valid/invalid, delay1/delay2). This makes for a total of 1280 trials. But they will go very, very quickly in this paradigm.
- Make sure that the picking of condition (valid/invalid, delay1/delay2, spatial location of cue) is random.

After collecting the data, answer the following questions:

1. Is there a difference in reaction times for valid versus invalid trials? (t-test)
2. Is there a difference in reaction times for different delays? (t-test)
3. Does the distance between target and cue matter? For this, use only invalid trials and plot reaction time as a function of
   a. Total distance of cue and target
   b. Horizontal distance of cue and target
   c. Vertical distance of cue and target
4. Related to this: Is there a qualitative difference in the slope of these lines? Is the scanner faster in one dimension than the other?
5. What is the speed of the attentional scanner? How many (unit of your choice, could be inches) does it shift per millisecond?

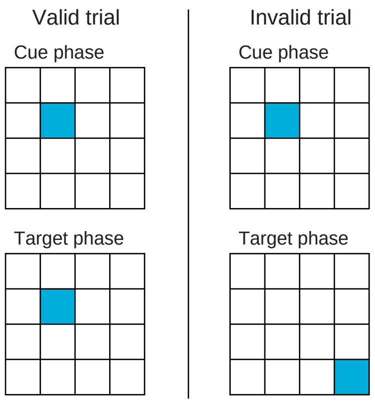


FIGURE 4.4 Valid and invalid trials.

Implement the project in MATLAB and answer the preceding questions. Illustrate with figures where appropriate.

***If you are adventurous**: Use **anova2** or **anovan** to look for interaction effects between type of trial (valid/invalid, delay and spatial location of cue).

Hints:

- Start writing one trial and make sure it works properly.
- Be aware that you effectively have an experimental design with three factors: Cue position (16 levels), trial type (2 levels), and temporal delay (2 levels). However, you can break it up into four factors: Horizontal cue position (4 levels), Vertical cue position (4 levels), trial type (2 levels), and temporal delay (2 levels), which will make it easier to assess the x- versus y-speed of the scanner.
- If you can't produce a proper cue, try reviewing object handles (in figures).
- Write a big loop that goes through trials. Do this at the very end, if individual trials work.
- If you can't do everything, focus on subgoals. Implement one function after the other. Start with two conditions. If you are not able to implement all eight conditions, try to get as far as you can.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

randstream
rectangle
ttest2
state
clock
anova1

CHAPTER

5

Psychophysics

5.1. GOALS OF THIS CHAPTER

In this chapter, you will learn how to use the MATLAB® software to do psychophysics. Once you master these fundamentals, you can use MATLAB to address any psychophysical question that might come to mind. While—in principle—all sensory modalities are open to psychophysical investigation, we will focus on visual psychophysics in this chapter.

5.2. BACKGROUND

Psychophysics deals with the nature of the quantitative relationship between physical and mental qualities. Today, the practice of psychophysics is ubiquitous in all fields of neuroscience that involve the study of behaving organisms, be they man or beast. Curiously enough, the origins of systematic psychophysics can be traced to a single individual: Gustav Theodor Fechner (1801–1887). Fechner's biography exhibits many telling idiosyncrasies. Born as the son of a pastor, he studied medicine at the University of Leipzig, but never practiced it after receiving his degree. Mostly by virtue of translating chemistry and physics textbooks from French into German, he was appointed professor of physics at the University of Leipzig. In the course of studying afterimages by gazing into the sun for extended periods of time—himself being the primary and sole subject—he almost lost his eyesight and went into deep depression in the early 1840s. This episode lasted for nearly a decade, a time which Fechner spent mostly within a darkened room. Emerging from this secluded state, he was overwhelmed by the sheer brilliant radiance of his surroundings, giving rise to his panpsychist worldview: he was now utterly certain that all things have souls, including inanimate objects such as plants and stones. Determined to share his insights with the rest of humanity, he soon started publishing on the topic, formulating an "identity theory" stating that the physical world and the spiritual world are not separate entities, but actually the same—the apparent differences resulting from different perspectives (first versus third person) onto the same object. In his view, this reconciles the incompatible dominant

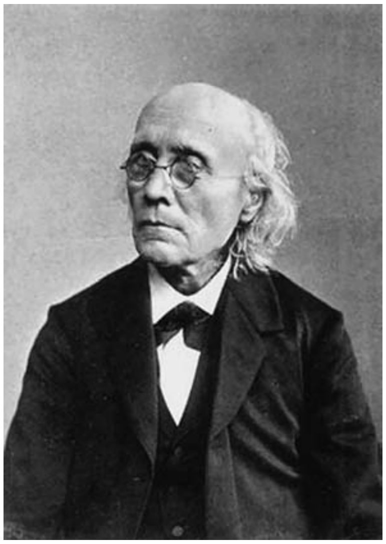


FIGURE 5.1 Gustav Theodor Fechner (1801–1887).

philosophical worldviews of the 19th century: idealism and materialism. However, his philosophical treatises on subjects such as the soul-life of plants or the transcendence of materialism were poorly received by the scientific community of the day. In order to convince his colleagues of the validity of his philosophical notions, he set out to devise methods that would allow him to empirically link physical and spiritual realms. His rationale being that if it can be shown that mental and physical qualities are in a clear functional relationship, this would lead credence to the notion that they are actually metaphysically identical.

Publishing the results of empirical studies on the topic in his *Foundations of Psychophysics* in 1860, he showed that this is the case for several mental domains, such as the relationship between physical mass and the perception of heaviness or weight. Fechner formulated several methods to arrive at these results that are still in use today. Importantly, he expressed the results of his investigations in mathematical, functional terms. This allowed the theoretical interpretation of his findings. Doing so, he introduced notions such as sensory thresholds quantitatively.

Ironically, inventing psychophysics did not help Fechner in convincing his philosophical adversaries of the merits of his identity theory. Few philosophers of the day renounced their idealistic or materialistic positions in favor of identity theory. Most of them simply chose to ignore Fechner, while the others mainly attacked him. Consequently, Fechner spent much of the remainder of his life fighting these real or imaginary adversaries, publishing two follow-up volumes in 1877 and 1882, chiefly focusing on the increasingly bitter struggle against the philosophical establishment of Imperial Germany. Ultimately, these efforts had little tangible or lasting impact. Meanwhile, the first experimental psychologists, particularly the group around Wundt, pragmatically used these very same methods to create a psychology that was both experimental and empirical. It is not to bold to claim that they never stopped and that contemporary psychophysics derives in an unbroken line from these very roots.

The key to visual psychophysics (and psychophysics more generally) is to elicit relatively simple mental phenomena that lend themselves to quantification by presenting physical stimuli that are easily described by just a few parameters such as luminance, contrast, or spatial frequency.

It is imperative that the experimenter has complete control over these parameters. In other words, the visual stimuli that he or she is presenting have to be precise. One way to create these stimuli is to use commercially available graphics editors, most prominently Photoshop®. While this practice is very common, it comes at a cost. For example, the images created by Photoshop have to be imported by the experimental control software. It is more elegant to create the stimuli in the same environment in which they are used. More importantly, the experimenter surrenders some degree of control over the created stimuli, when using commercial graphics editors, because the proprietary algorithms to perform certain image functions are not always completely documented or disclosed. This problem is equally avoided by creating the stimuli in a controlled way within MATLAB.

5.3. EXERCISES

We need to introduce methods by which to create and present visual stimuli of any kind on the screen. Fortunately, MATLAB includes a large library of adequate functions. We will introduce the most important ones here.

By default, MATLAB visualizes images by assuming triples in a 256-element RGB space. Each element of the triplet has to be an integral value between 0 and 255. This corresponds to a range of 8 bit. Hence, these elements can be represented by variables of the type uint8. These values correspond to the intensity of red, green, and blue at a particular location in the image. Depending on the physical output device (typically cathode ray tube or LCD displays), these values effectively regulate the voltage of individual ray guns or pixels. Of course, MATLAB also supports much higher bit-depths. For the purposes of our discussion, 8 bit will suffice.

One important caveat is that the relationship between the assigned voltage values of the three individual ray guns in the cathode ray tube (0 to 255) and the perceived luminance of the screen is not necessarily linear. Visual scientists generally calibrate their monitors by "gamma correcting" them. This linearizes the relationship between assigned voltage or intensity values and the perceived luminance.

To be able to linearize the relationship, you need a photometer. Because we assume that those are—due to their generally high price—not readily available, we will forgo this step for the purposes of this book. However, we urge budding visual scientists to properly calibrate their monitors before doing an experiment in which the veracity of the data is crucial—as is the case if they are intended for publication. For more information on the issue of monitor calibration, see for example Carpenter & Robson (1999).

To begin, create a simple matrix with the following command:

```
>> test_disp = uint8(zeros(3,3,3))
```

This command creates the matrix **test_disp**, which is a three-dimensional matrix with three elements in each dimension. Importantly, it is of the data type **uint8**, which MATLAB assumes by default for its imaging routines. Of course, the function image can also image other matrices, but this would require to specify additional parameters. Imaging only matrices of the type **uint8** is the most straightforward thing to do for the purposes of this chapter.

Now type the following:

```
>> figure
>> subplot(2,2,1)
>> image(test_disp)
```

The function image compels MATLAB to interpret the values in the matrix as commands for the ray guns of the monitor and to display them on the screen. You should now be looking at a completely and uniformly dark (black) subplot 1.

Now, type the following:

```
>> test_disp(2,2,:) = 255
>> subplot(2,2,2)
>> image(test_disp)
```

The structure of the matrix and the function of the **image** command now become apparent.

The 0 values are interpreted as turning off all ray guns. The 255 values are interpreted as full power. As you maximally engage all three guns (represented by the third dimension of the matrix), the result is an additive mix of spectral information that is interpreted by the visual system as white. This need not be the case. Now, try this:

```
>> subplot(2,2,3)
>> test_disp(2,2,1) = 0
>> image(test_disp)
```

The picture in subplot 2 is devoid of color from the red gun. It should appear cyan. Now try the following:

```
>> test_disp(2,2,:) = 0
>> test_disp(2,2,1) = 255
>> subplot(2,2,4)
>> image(test_disp)
```

This code has the opposite effect, yielding a red inner pixel. The result should look something like Figure 5.2.

Suggestion for Exploration: Can you create arbitrary other colors? Can you create arbitrary shapes?

While it is hard to surpass this example of using just 9 pixels in clarity, it is also somewhat pedestrian. The true power of this approach becomes clear when considering natural stimuli, which are also increasingly used in visual psychophysics. To do this, you need to import an image into MATLAB. For this example, use the **imread** function:

```
>> temp = imread('UofC.jpg')
```

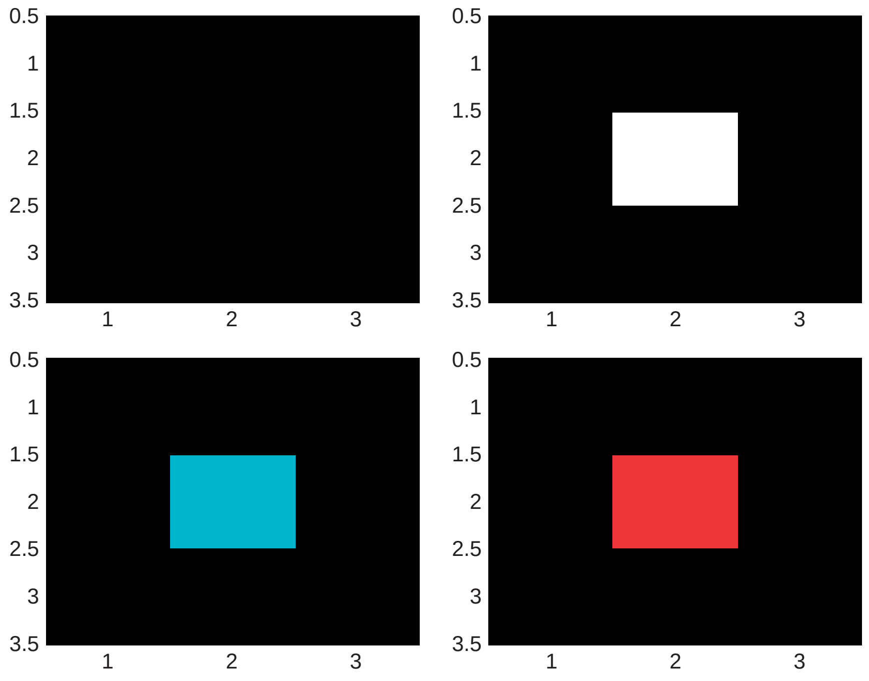


FIGURE 5.2 Testing the ray guns with matrices interpreted as images.

This command creates a large three-dimensional matrix of the **uint8** type (positive integral values from 0 to 255, as can be addressed by 8 bits).

Next, type the following to get a magnificent view of the Harper Library at the University of Chicago:

```
>> figure
>> subplot(2,2,1)
>> image(temp)
```

Now, you can manipulate this image in any way, shape, or form. Importantly, you will know exactly what you are doing, since you are the one doing it, which cannot be said for most of the opaque algorithms of image processing software. For example, you can separate the information in different color channels by typing the following:

```
>> for i = 1:3
>> bigmatrix(:,:,:,i) = zeros(size(temp,1),size(temp,2),3);
>> bigmatrix(:,:,i,i) = temp(:,:,i);
>> subplot(2,2,i+1)
>> upazila = uint8(bigmatrix(:,:,:,i));
>> image(upazila)
>> end
```

You get the picture shown in Figure 5.3 as a result.

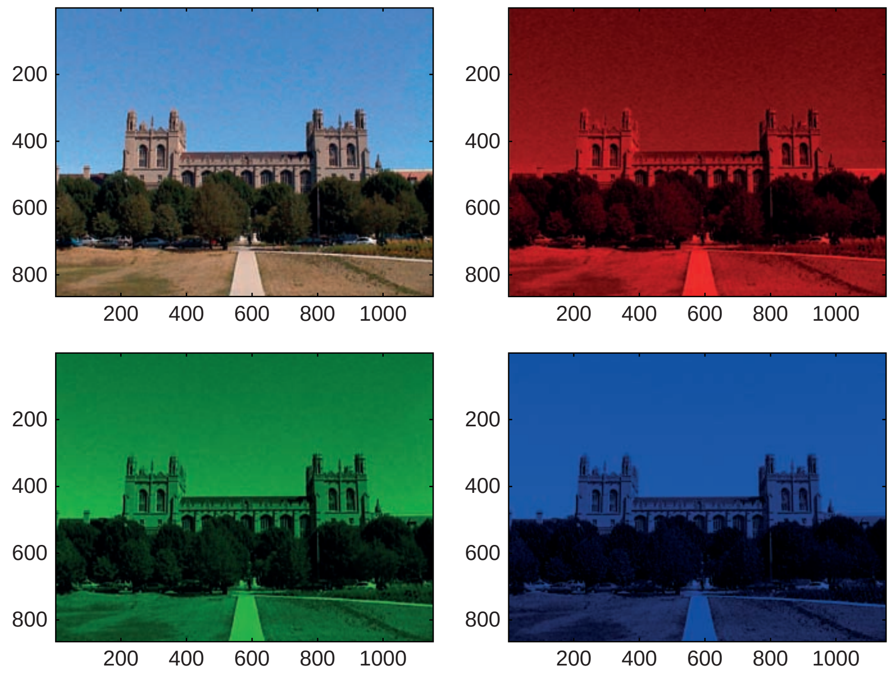


FIGURE 5.3 The University of Chicago Harper Library in red, green, and blue.

Suggestion for Exploration: Can you find the MATLAB "Easter eggs"? MATLAB once had a large number of those. Most of them have been removed by now. Some of them remain. One of them concerns the image function. If you type **image** without any arguments, the creepy "MATLAB Ghost" appears in your figure. Another remaining easter egg concerns the **spy** function, a function to visualize the structure of sparse matrices. Try it without arguments as well.

Note the use of the **upazila** helper variable. You have to use this because MATLAB interprets non-3D-**uint8** matrices differently when presenting images. Later you will learn how to make do without the **upazila** step. The different subplots illustrate the brightness values assigned to an individual ray gun (Upper left: all of them together. Upper right: red. Lower left: green. Lower right: blue). For technical reasons, we reproduce all images in grayscale in this chapter. For full color, see the insert with color panels. This allows you to assess the contribution of every single channel (red/green/blue) to the image. Another way of judging the impact of a particular channel is to leave it out. To do this, you add the other channels together, as follows:

```
>> bigmatrix2(:,:,:,1) = bigmatrix(:,:,:,1)+bigmatrix(:,:,:,2)+bigmatrix(:,:,:,3);
>> bigmatrix2(:,:,:,2) = bigmatrix(:,:,:,1)+bigmatrix(:,:,:,2);
>> bigmatrix2(:,:,:,3) = bigmatrix(:,:,:,1)+bigmatrix(:,:,:,3);
>> bigmatrix2(:,:,:,4) = bigmatrix(:,:,:,2)+bigmatrix(:,:,:,3);

>> figure
>> for i = 1:4
```

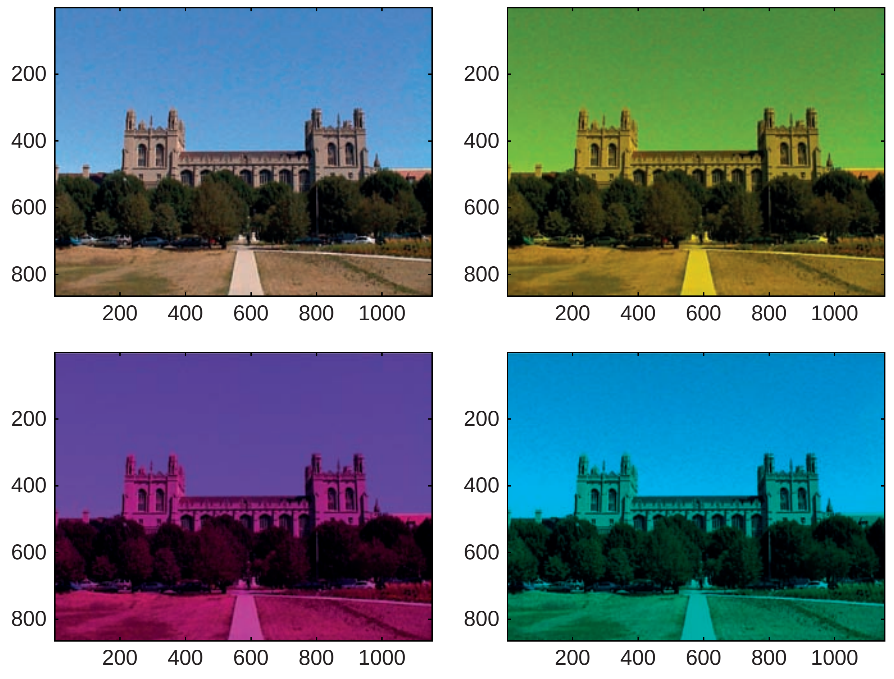


FIGURE 5.4 The University of Chicago Harper Library without red, green, and blue information.

```
>> subplot(2,2,i)
>> image(uint8(bigmatrix2(:,:,:,i)))
>> end
```

Doing so should yield the picture shown in Figure 5.4.

In this figure, the upper left is all channels. In the upper right, the blue channel is missing. In the lower left, the green channel is missing. In the lower right, the red channel is missing. Belaboring this point enhances an understanding of the relationship between the brightness values in the three-dimensional matrix and the appearance of the image.

Suggestion for Exploration: Find out what the Matlab **upazila** actually is.

You are now in a position to implement arbitrary changes to the image. For example, you can brighten it, increase the contrast, or selectively change the color balance. To explore this, start by changing the overall brightness by typing the following:

```
>> figure
>> subplot(2,2,1)
>> image(uint8(bigmatrix2(:,:,:,1)))
>> subplot(2,2,2)
>> image(uint8(bigmatrix2(:,:,:,1)+50))
>> subplot(2,2,3)
```

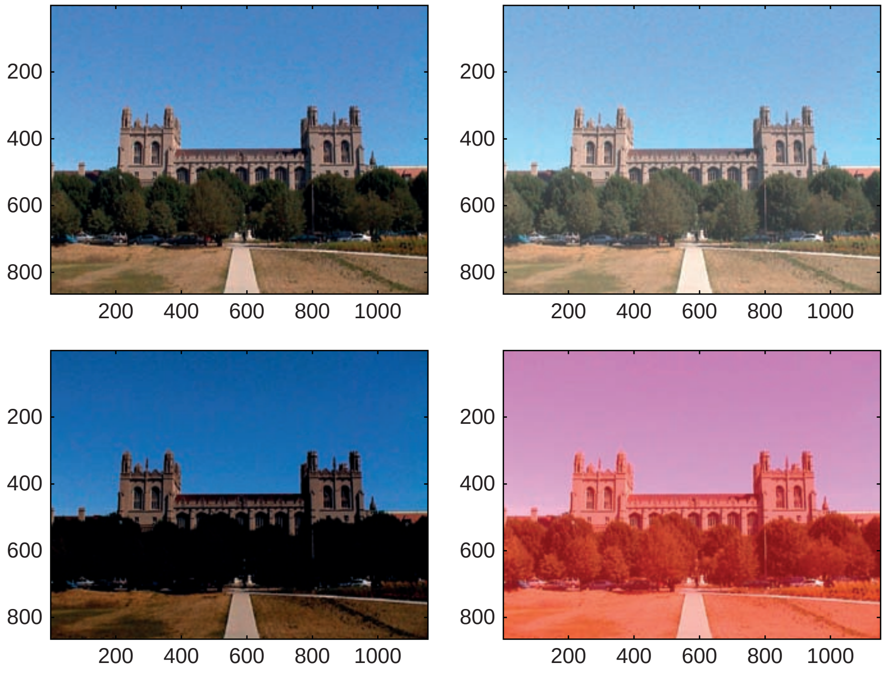


FIGURE 5.5 Brightening and darkening any or all ray guns has a profound effect.

```
>> image(uint8(bigmatrix2(:,:,:,1)-50))
>> subplot(2,2,4)
>> bigmatrix2(:,:,1,1) = bigmatrix2(:,:,1,1) + 100;
>> image(uint8(bigmatrix2(:,:,:,1)))
```

The result, shown in Figure 5.5, is a picture that has been somewhat brightened (upper right), darkened (lower left), and where the red channel has been turned (way) up in the lower right.

Suggestion for Exploration: Increase the contrast of the image. Also try to image matrices that are not of the type **uint8**.

One of the most common image manipulations using image editors is the smoothing or sharpening of the image. The former is often performed to get rid of random noise or granularities in the image. Scientists might do this to simulate and understand the output of the visual system of another species. Importantly, these ends are typically achieved by low- or high-pass filtering of the original image. Unfortunately, most users don't really understand what is happening behind the scenes when using a commercially available image editor. Of course, this is unacceptable for doing psychophysics in particular or science in general.

Hence, we will now discuss how to perform these operations in MATLAB. First, import another image by typing the following. This image is more suited to making the effects of your manipulations more readily apparent.

```
>> pic = imread('filtering.jpg')
>> figure
>> subplot(2,2,1)
```

Look at the image. So far, so good. Now, slightly blur the image. To do so, you will convolve the image with a filter. Refer to Chapter 10, "Convolution," to understand precisely the underlying mathematics of the operation. For purposes of this example, it is enough to understand that the convolution operation will allow you to blur the image by blending brightness values of adjacent pixels. To create a small 3×3 filter, you type

```
>> filter = ones(3,3)
```

then

```
>> lp3 = convn(pic,filter)
```

to perform the convolution of the image with the 3×3 filter. You might have noted that the values are no longer in the range between 0 and 255. This is due to the multiplying and adding brought about by the convolution. Next, divide by the block size (9) to rectify this situation:

```
>> lp3 = lp3./9;
```

This operation creates floating-point values, so you have to be careful when imaging this:

```
>> subplot(2,2,2)
>> image(uint8(lp3))
```

This code creates a very slightly low-pass filtered version of the image. This result is most readily apparent when you look at the texture of the hat or hair in the image. Now, try a more radical low-pass filtering:

```
>> filter = ones(25,25)
>> lp25 = convn(pic,filter)
>> lp25 = lp25./625;
>> subplot(2,2,3)
>> image(uint8(lp25))
```

Looking at the image reveals a significant blurring. This is the low-frequency component of the image. It is similar to what a typical nocturnal animal with a relatively poor spatial acuity might see (sans the color). You arrive at the image by blurring a substantial number of pixels together.

Suggestion for Exploration: What happens if you use ever larger filters?

Note that the matrices you created with the convolution operation are slightly larger than the one that represents the initial picture (which had a format of 600×800). This is due to the nature of the convolution operation. It creates an artificial black rim not present in the original picture. You will understand why this happens and why this is a hard problem when reading chapter 10. For now, and to (mostly) get rid of it and cut the image back to size, try the following:

```
>> lp25cor = lp25(13:612,13:812,:);
```

You might also have noted that the execution of the convolution operation took a significant amount of time. This might be important if you want to create your stimuli on the fly, as the subject does the experiment. To assess how your system stacks up against certain known benchmarks, type this command:

```
>> bench
```

Doing so makes MATLAB perform various typical operations and compare the speed of their execution to other benchmark systems. This is particularly crucial when you're running a time-sensitive program. Don't be surprised when receiving rather low benchmark values, particularly when running MATLAB over a network, despite basically fast hardware. To evaluate the reliability of the benchmark values, try running **bench** more than once.

Let's get back to the filtering problem. The image in the lower left corresponds to what psychophysicists would call the "low spatial frequency" channel. It contains the low spatial frequency information in the image. Notably, it is mostly devoid of sharp edges. This information about edges in the image is contained in the "high spatial frequency" channel. How do you get there? By subtracting the low spatial frequency information from the original image. Try this:

```
>> subplot(2,2,4)
>> hp = pic-uint8(lp25cor);
>> image(hp)
```

The image in the lower right now contains the high spatial frequency information. It represents most of the textures and sharp edges in the original image.

Unfortunately, it is rather dark (due to the subtraction). To appreciate the full high spatial frequency information, add a neutral brightness level back in:

```
>> hp = hp+127;
>> image(hp)
```

Much better. The final result should look something like that shown in Figure 5.6.

Exercise 5.1: Use the information in the high spatial frequency channel to sharpen (enhance the edges) of the original image.

We have discussed a variety of image manipulations with MATLAB, namely the manipulation of form, color, and spatial frequency. One remaining major issue is the creation of moving stimuli. There are many ways to do this in MATLAB. One of the most straightforward is to use the **circshift** function in combination with a frame capture function.

To use this function, type the following:

```
>> figure
>> pic3 = circshift(pic,[100 0 0]);
>> image(pic3)
```

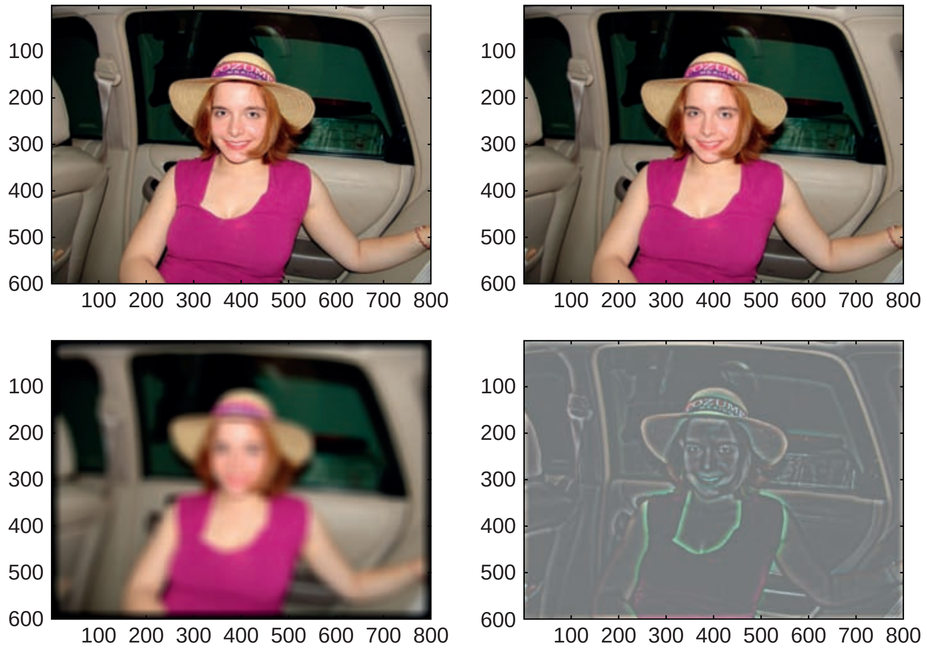


FIGURE 5.6 Information about texture is carried in different spatial frequency channels.

The **circshift** function shifts all matrix values by the stated amount in the second argument—in this case, 100 in the direction of the first dimension, nothing in the others.

You can use this to create a movie:

```
>> figure
>> pic4 = pic;
>> for i = 1:size(pic4,1)/10+1
>> image(pic4)
>> pic4 = circshift(pic4,[10 0 0]);
>> M(i) = getframe;
>> end
```

There are other ways to create movie frames, for example, using the **im2frame** function. However, this version lets you preview the movie as you create it. You can play it by typing

```
>> movie(M,3,24)
```

This command plays the movie in matrix *M* three times, at 24 frames per second, in the existing figure. One caveat for movies is size in memory. These frames take up a considerable amount of space. You might get an error message indicating that the frames could not be created if you go beyond the available memory. The available memory depends on the computer and operating system. To assess the memory situation on the machine you are using, type

```
memory
```

There are many caveats when making movies with MATLAB. For example, the choppiness will depend on many factors, including machine speed, available memory, step size of the **circshift**, as well as frame rate. On most systems, it will be hard to avoid trade-offs to create movies that are reasonably smooth.

Exercise 5.2: Use this knowledge to create a movie of a single white dot (pixel) that moves from the far left of the screen to the far right.

Suggestion for Exploration: Import two pictures with the same size. Create a movie in which one morphs into the other. *Hint:* Over time, the numerical values in the matrix that represents the image should gradually shift from one to the next while you capture this process in frames.

As the color information is represented in the third dimension of the matrix, you can also use **circshift** to elegantly swap colors, as in this example:

```
>> figure
>> for i = 1:3
>> subplot(1,3,i)
>> image(pic)
>> axis equal
>> axis off
>> pic = circshift(pic,[0 0 1]);
>> end
```

The result should look something like Figure 5.7.

Often, when creating large numbers of stimuli, you might want to save them on the hard disk to free up some space in available memory. You can easily do this by using the **imwrite** function. For example, you can save the image in which the RGB values were swapped for BRG by typing **imwrite(pic,'BRG.jpg','jpg')**. This should have created the file BRG.jpg as a .jpg file in your working directory. You can now open it with other image editing software, put it online, etc. Similarly, you can save the movie by typing

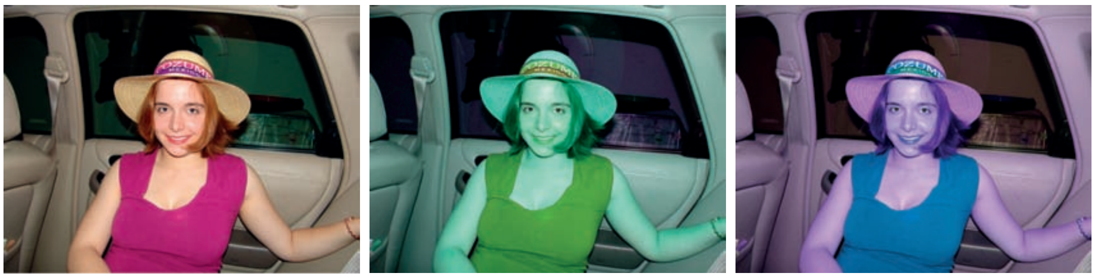


FIGURE 5.7 You can use **circshift** to shift colors.

```
movie2avi(M,'upazila.avi','quality',100,'fps',24)
```

which creates an .avi file named thismoves at a frame rate of 24 and a quality of 100 in your working directory. From now on, this file will behave like any other movie file you might have on your hard disk.

At this point, we have explored several important image manipulation routines that should really give you a deep appreciation of the way MATLAB represents and displays images. Of course, many more image manipulations are possible in MATLAB. We will leave those for you to discover and return to the task of collecting psychophysical data, using this newfound knowledge. Because MATLAB represents images as brightness values in a three-dimensional matrix, you can manipulate them at will with any number of matrix operations. In principle, you could write your own Photoshop toolbox in MATLAB.

Exercise 5. 3: Can you rotate an image by 90°? Can you rotate by an arbitrary number of degrees?

Exercise 5.4: Try adding different images together. For example, you can transmit secret information by embedding one image in another. Or create artificial stimuli. For example, in the attention community, it is popular to superimpose pictures of houses and faces.

Suggestion for Exploration: Implement your favorite Photoshop routine in MATLAB.

Let's get back to psychophysics. Fechner formalized three fundamental methods to elicit the relationship between mental and physical qualities and introduced them to a wider audience. These methods are still in use today. You should recognize the *method of limits* from visits to the ophthalmologist investigating your vision or the otologist investigating your hearing. Basically, the subject is presented with a series of stimuli in increasing (or decreasing) intensity and asked to judge whether or not the stimulus is present. This method is extremely efficient because only a few stimuli are necessary to establish fairly reliable thresholds. Unfortunately, the method suffers from hysteresis; the threshold is path dependent, as subjects exhibit a certain inertia (e.g., stating that the stimulus is still present even if they can't detect it, if coming from the direction of a stimulus being present). This problem can be overcome by counterbalancing (starting from different states). However, a better correction is the *method of constant stimuli*. In this method, the experimenter presents stimuli to be judged by the subject in random order, from a predetermined set of values. The advantage of this method is that it yields very reliable and mostly unbiased threshold measurements. The drawback is that one needs to sample a relatively large range of stimuli (as one doesn't a priori know where the threshold will lie) and a large number of repetitions per conditions to reduce error. Hence, this method is usually not used where time is at a premium (such as in a doctor's office), but rather in research, where the time of undergrad or grad student subjects is routinely sacrificed for increases in accuracy.

Finally, the *method of adjustment* lets the subject manipulate a test stimulus that is supposed to match a given control. This method is particularly popular in color psychophysics. It is relatively efficient, but suffers from its own set of biases.

5.4. PROJECT

In this project, you will use the method of constant stimuli to determine the absolute threshold of vision, a classic experiment in visual psychophysics (Hecht, Shlaer, and Pirenne, 1942). Obviously, you will be able to do only a crude mock-up of this experiment in the scope of this chapter. The actual experiment was extremely well controlled and took a long time to carry out (not to mention specialized equipment).

Since you are unconcerned with publishing the results (these are extremely well established), you can pull off a "naïve" version in order to highlight certain features and principles of the psychophysical method. If you want to increase experimental control, perform the experiment in a dark room and wait 15 minutes (or better 30 minutes) before data collection. Also, try to keep a fixed distance from the monitor (e.g., 50 cm) throughout the data collection phase of the experiment.

However, before you can collect data, you need to write a stimulus control program utilizing the skills from the previous two chapters and the image manipulation skills introduced in this chapter. Here is a simple program that will do what is needed (make this an M-file). Note the somewhat obsolete use of the modulus function to order the stimuli. We could also do this with randi in the latest versions of MATLAB. On the other hand, the use of the modulus function allows to have exactly the same number of trials per condition (as opposed to them having random frequencies).

```
clear all;  %Emptying workspace
close all; %Closing all figures

temp = uint8(zeros(400,400,3)); %Create a dark stimulus matrix
temp1 = cell(10,1); %Create a cell that can hold 10 matrices

for i = 1:10 %Filling temp1
  temp(200,200,:) = 255; %Inserting a fixation point
  temp(200,240,:) = (i-1)*10; %Inserting a test point 40 pixels right
                                          %of it. Brightness range 0 to 90.
  temp1{i} = temp; %Putting the respective modified matrix in cell
end %Done doing that

h = figure %Creating a figure with a handle h

stimulusorder = randperm(200); %Creating a random order from 1 to 200.
                                          %For the 200 trials. Allows to have
                                          %a precisely equal number per condition.

stimulusorder = mod(stimulusorder,10); %Using the modulus function to
                                          %create a range from 0 to 9. 20 each.
```

```
stimulusorder = stimulusorder + 1; %Now, the range is from 1 to 10, as
                                                     %desired.

score = zeros(10,1); %Keeping score. How many stimuli were reported seen

for i = 1:200  %200 trials, 20 per condition
image(temp1{stimulusorder(1,i)}) %Image the respective matrix. As
                          %designated by stimulusorder
i %Give subject feedback about which trial we are in. No other feedback.
pause; %Get the keypress
temp2 = get(h,'CurrentCharacter'); %Get the keypress. "." for present,
                          %"," for absent.
temp3 = strcmp('.', temp2); %Compare strings. If . (present), temp3 = 1,
                        %otherwise 0.
score(stimulusorder(1,i)) = score(stimulusorder(1,i)) + temp3; %Add up.
                                              % In the respective score sheet.
end %End the presentation of trials, after 200 have lapsed.
```

Note that these are relatively crude steps. In a real experiment, you might want to probe every luminance value and collect more samples per condition (50 or 100 instead of 20). Also, a time limit of exposure and decision time is usually used. But for now, this will do. When running this program yourself, make sure to focus on the central fixation dot. Don't get frustrated or bored. Psychophysical experiments are extremely intricate affairs, usually operating at the limits of the human sensory apparatus. Hence, they are rarely pleasant. So try to focus the firepower of your cortex on the task at hand. Also note that you will make plenty of errors. Don't get frustrated. That is the point of psychophysics. In a way, psychophysics amounts to a very sophisticated form of producing and analyzing errors. If you don't make any errors, there is no variance, and without variance, most of the psychophysical analysis methods fail—hence the large number of trials. Given enough trials, you can count on the statistical notion that truly random errors will average out, while retaining and strengthening the systematic trends in the data, revealing the properties of the system that produced it. As a matter of fact, you might want to throw in a couple of practice runs before deciding to analyze your data for real.

Given that you are likely to be what is technically called an untrained observer there will be various dynamics going on during the experiment. At first, practice effects will enhance the quality of your judgments; then fatigue will diminish it again. Also note that you are technically not a "naive" subject, as you are aware of the purpose of the experiment. Don't let this discourage you for now.

Doing so, we obtained the curve shown in Figure 5.8.

This figure shows a fairly decent psychometric curve. It is obvious that we did not see the dot on the left tail of the curve (the observed variation represents errors in judgment). Similarly, it is obvious that we did always see the dot on the right of the curve, yet there is some variation in the reported seen instances. In other words, the points on the left are below threshold, whereas the points on the right are already saturated. In a real experiment, we would resample the range between the brightness values 20 and 70 much more densely, as it is clear that the date points outside this range add no information. However, this neatly illustrates one problem of the method of constant stimuli. We didn't know where the threshold would lie. Hence, we had to

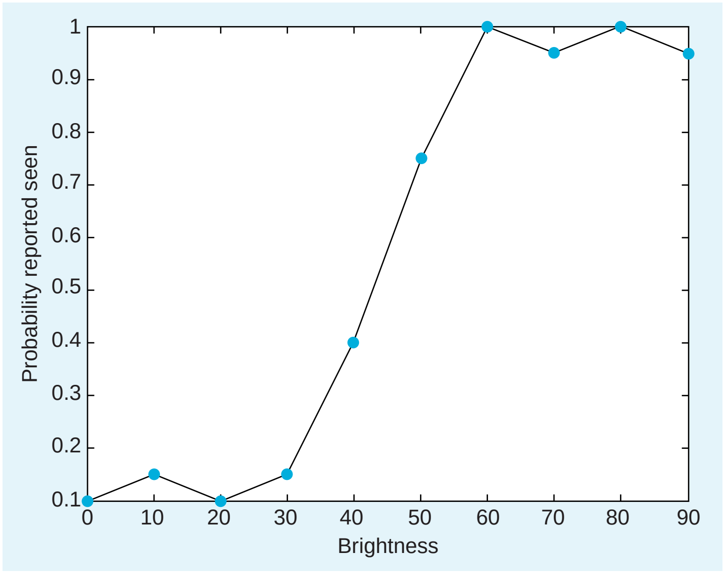


FIGURE 5.8 The psychometric curve reveals below-threshold regions, saturation regions, and a linear range.

sample a broad range—undersampling the crucial range and oversampling regions of no interest. Even limiting the range from 0 to 90 was an educated guess. Strictly speaking and without any previous knowledge, we would have had to sample the entire range of 0 to 255.

Psychophysicists like to boil down this entire dataset into one number: the *absolute threshold*. In this case, you can derive this value by interpolation. It is the x-value that corresponds to the intersection between the curve and the y-value of probability reported seen of 0.5, as shown in Figure 5.9.

In other words, this analysis indicates an absolute threshold of a brightness value of 43. If you want to get a precise threshold, you would have to resample the range between 30 and 60 (or even between 40 and 50) very densely.

Also note that this value of 43 is not inherently meaningful. Without having the monitor calibrated with a photometer, we don't know to how much physical light energy this corresponds. Hence, we can't relate it to the minimum number of light quanta that can be detected or such. However, this threshold is meaningful in the context of the behavioral task: a shifted threshold under different conditions can give rise to conjectures about the structure and function of the physiological system producing these thresholds, as you will see when doing the exercises. Moreover, the absolute threshold is a stochastic concept. It is not true that lights below it are never seen.

Of course, psychophysicists have very elaborate ways to analyze data like these. Most straightforwardly, they like to fit sigmoidal logistic curves to such data. We will go into the intricacies of psychophysical data analysis in the next chapter.

Finally, we chose luminance values that worked on our monitor, yielding a decent psychometric curve, allowing us to determine the threshold. You might have to use a different range when working within your setup.

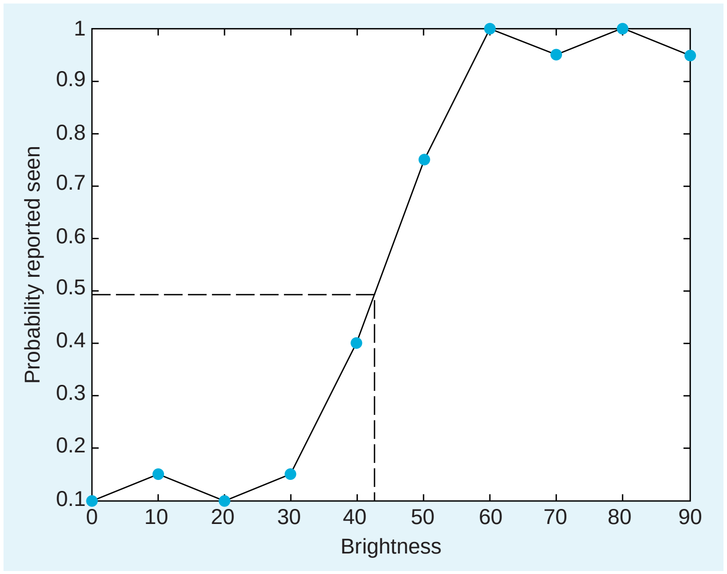


FIGURE 5.9 The psychometric curve allows you to establish the absolute detection threshold.

For more background on psychophysical methods, read the classic *Elements of Psychophysics* by Fechner (1860) or, for a modern treatment of the use of these methods in visual neuroscience, *The Psychophysical Measurement of Visual Function* by Norton, Corliss, and Bailey (2002).

In this project, you should specifically address the following issues:

- Compare thresholds in the periphery and center. You just did a parafoveal stimulus presentation (if you were honest and fixated the fixation point) or even a foveal presentation (if you looked at the stimulus directly). How does the threshold change in the periphery (putting the stimulus several hundred pixels away from the fixation point)?
- Determine the thresholds for brightness values of the red, green, and blue guns individually. Which gun has the lowest threshold (is perceived as brightest)? Which gun has the highest threshold (is perceived as dimmest)? Can you account for the white threshold (as we did above) by adding the individual thresholds?
- You just determined absolute thresholds. Another important concept in psychophysics is the relative threshold. To determine the relative threshold, put another test dot to the left of the fixation point. The task is now to indicate if the brightness of the right dot is higher (.) or lower (,). Does the relative threshold depend on the absolute brightness values of the dots? If so, can you characterize the relationship between relative threshold (difference in stimulus brightness values that gives a probability of 0.5) and absolute value of the stimuli?
- When determining the relative threshold, can you reason why it makes sense to ask which of the two is brighter, instead of asking if they are the same or different (which might be more intuitive)?

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

uint8
double
convn
circshift
image
bench
imread
imwrite
memory
movie
getframe
movie2avi
randperm
modspy

CHAPTER

6

Signal Detection Theory

6.1. GOALS OF THIS CHAPTER

This chapter will mostly concern the use of signal detection theory to analyze data generated in psychophysical—and hypothetical neurophysiological—experiments. As usual, we will do this in MATLAB.

6.2. BACKGROUND

At its core, signal detection theory (SDT) represents a way to optimally detect a signal in purely statistical terms without and explicit link to decision processes in particular or cognitive processes in general. However, in the context of our discussion, SDT provides a rich view of the problem of how to detect a given signal. In particular, it reframes the task as a decision process, adding a cognitive dimension to our understanding of this matter.

To illustrate the application of SDT in psychophysics, let us again consider the problem of reporting the presence or absence of a faint, barely visible dot of light, as in Chapter 5, "Psychophysics." In addition to the threshold, which is determined by the physical properties of the stimuli and the physiological properties of the biological substrate, there are cognitive considerations. In particular, observers have a *criterion* by which they judge (and report) whether or not a signal was present. Many factors can influence the criterion level and—hence—this report. You likely encountered some of those in the preceding chapter. For example, your criterion levels might have been influenced by doing a couple hundred trials, giving you an appreciation of what “present” and “absent” mean in the context of the given stimulus range (which is very dim overall). Moreover, motivational concerns might play a role when setting a criterion level. If subjects have an incentive to over- or under-report the presence of a signal—e.g., if they think that the experimenter expects this, they will in fact do so (Rosenthal, 1976). Interestingly, neuroeconomists utilize this effect by literally paying their subjects to prefer one alternative, in order to study the mechanisms of how the criterion level is set by their subjects.

Of course, one could question the real-world relevance of these considerations, given that they arose in very particular and arguably often rather contrived experimental settings. It is worth emphasizing that this first impression is extremely misleading. Today, signal detection theory constitutes a formal, stochastic way to estimate the probability by which some things are the case and others are not; by which some effects are real and others are not, and so on. As such, it has the broadest possible implications. Signal detection theory is used by pharmaceutical companies as well as oil prospectors, and it has even made its mark in public policy considerations. Of course, experimenters—psychophysicists in particular—also still use it.

The astonishing versatility and base utility of signal detection theory are likely owed to the fact that it goes to the very heart of what it means to be a cognitive organism or system, as we will now describe.

Consider the following situation. Let us assume you work for a company that builds and installs fire alarm systems. As these systems are ubiquitous in modern cities, business is good. However, you are confronted with a rather confounding problem: How sensitive should you make these alarms? The four possible cases are tabulated in a classical matrix, as shown in Table 6.1.

Let's peruse this matrix in detail as it is the foundation of the entire discussion to follow. The cell in the upper left represents the "desired" (as desired as it can be, given that there is a real fire in the building) case: there actually is a fire and the alarm does go off, urging the occupants of the building to leave and alerting the fire department to the situation. Ideally, you would like the probability of this event to be 1. In other words, you always want the alarm to go off when there is a fire present. This part should be fairly uncontroversial. The problem is that in order to reach a probability of 1 for this case, you need to set the criterion level for indicating "fire" by some parameters (usually smoke or heat or both) incredibly low. In fact, you need to set it so low that it will likely go off by levels of smoke or heat that can be reached without a fire being present. This puts you into the cell in the upper right. If this happens, you have a false alarm. From personal experience, you can probably confirm that the criterion levels of fire alarms are typically set in a hair-trigger fashion. Almost anything will set them off, and almost all alarms are therefore false alarms, given that the a priori probability of a real fire is very, very low. As is the case in modern cities. While this situation is better than having a real fire, false alarms are not trivial. Having them frequently is disruptive, can potentially have deleterious effects in case of a real fire (as the occupants of the building learn to stop taking action when the alarm goes off), and strains the resources of the firefighters (as a matter of fact, firefighters have been killed in traffic accidents on their way to false alarms). In other words, setting the sensitivity too high comes at a considerable cost. Hence, you want to lower the sensitivity enough to always be in a state that corresponds to one of the cells on the main diagonal of the matrix, either having a hit (if there is actually a fire present) or a "correct rejection," arriving in the lower right. The latter should be the most common case, indicating that there is no

TABLE 6.1 The Signal Detection Theory Payoff Matrix

	Real fire	**No fire (but possibly some smoke or heat)**
Alarm goes off	Hit	False alarm
Alarm does not go off	Miss	Correct rejection

fire and the fire alarm does not go off. Unfortunately, if you drop the sensitivity too low, you arrive in the worst cell of all in terms of potential for damage and fatalities: having a real fire, but the fire alarm does not alert you to this situation. This is called a “miss,” in the lower left.

In a way, this matrix illustrates what signal detection theory is all about: figuring out a way to set the criterion in a mathematically optimal fashion (in the applied version) and figuring out how and why people, organisms, and systems actually do set criteria when performing and solving cognitive tasks (in the pure research version).

If this description sounds familiar, it should. As contemporary science has largely adopted a stochastic view on epistemology, this fundamental situation of signal detection theory appears in many if not most experiments, disguised as the “p-value” problem.

You are probably well aware of this issue, so let us just briefly retrace it in terms of signal detection theory.

When performing an experiment, you observe a certain pattern of results. The basic question is always: How likely is this pattern, given there is no effect of experimental manipulation? In other words: How likely are the observed data to occur purely by chance? If they are too unlikely given chance alone, you reject the “null hypothesis” that the data came about by chance alone. That is—in a nutshell—the fundamental logic of testing for the statistical significance of most experimental data since Fisher introduced and popularized the concept in the 1920s (Fisher, 1925).

But how unlikely is too unlikely? Again, we face the fundamental signal detection dilemma, as illustrated in Table 6.2.

In science, the criterion level is conventionally set at 5%. This is called the *significance level*. If a certain pattern of data is less likely than 0.05 to have come about by chance, then you reject the null hypothesis and accept that the effect exists. Implicitly, you also accept that—at this level—5% of the published results will not hold up to replication (as they don’t actually exist). It is debatable how conservative this standard is or should be. For extraordinary claims, a significance level of 1% or even less is typically required. What should be apparent is that the significance level is a social convention. It can be set according to the perceived consequences of thinking there is an effect when there is none (*alpha error*) or failing to discover a genuine effect (*beta error*), particularly in the medical community. The failure to find the (side-) effect of certain medications has cost certain companies (and patients) dearly. For a dissenting view on why the business of significance testing is a bad idea in the first place, see for example Ziliak & McCloskey (2008).

Regardless of this controversy, one can argue that any organism is—curiously—in a quite similar position. You will learn more about this in Chapters 16, "Neural Decoding: Discrete Variables," and 17, "Neural Decoding: Continuous Variables." For now, let us discuss the fundamental situation as it pertains to the nervous system (particularly the brain) of the origanism. Interestingly, based on everything we currently assume to be true, the brain has no direct access to the status of the environment around it—as manifested in the values of physical parameters such as energy or matter. It learns about them solely by the pattern of activity

TABLE 6.2 Alpha and Beta Errors in Experimental Judgment

	Effect exists (H0 false)	**Effect does not exist (H0 true)**
We conclude it exists	Discovery of effect	Alpha error (false rejection of H0)
We conclude it doesn’t exist	Beta error (false retention of H0)	Failure to reject the null hypothesis

within the sensory apparatus itself. In other words, the brain deduces the structure of the external world by observing the structural regularities of its own activity in response to the conditions in the outside world. For example, the firing of a certain group of neurons might be associated with the presence of a specific object in the environment. This has profound philosophical implications. Among them is the notion that the brain decodes its own activity in meaningful ways, as they were established by interactions with the environment and represent meaningful associations between firing patterns and states in the environment. In other words, the brain makes actionable inferences about the state of the external world by cues that are provided by activity levels of its own neurons. Of course, these cues are rarely perfectly reliable. In addition, there is also a certain level of "internal noise", as the brain computes with components that are not perfectly reliable either. This discussion should make it clear how the considerations about stochastic decision making introduced previously directly apply to the epistemological situation in which the brain finds itself. We will elaborate on this theme in several subsequent chapters. It should already be readily apparent that this is not trivial for the organism, as it has to identify predators and prey, along with other biologically relevant hazards and opportunities in the environment. In this sense, errors can be quite costly.

Suggestion for Exploration: The basic signal detection situation seems to reappear in different guises over and over again. What we call the two *fundamental errors* varies from situation to situation. Try framing the results of a diagnostic test for an arbitrary disease in these terms (that here appear as *false positive* and *false negative*). Also, try to model a case in the criminal justice system in this way.

6.3. EXERCISES

With this background, it is now time to go back to the MATLAB® software. Let us discuss how you can use MATLAB to apply signal detection theory to the data generated in behavioral experiments.

Consider this situation. You run an experiment with 2000 trials. While running these trials, you record the firing rate from a single neuron in the visual cortex. In 1000 of the trials, you present a very faint dot. In the other 1000, you just present the dark background, without added visual stimulus.

Let's plot the (hypothetical) firing rates in this experiment. To do so, you use the **normpdf** function. It creates a normal distribution. Normal distributions occur in nature when a large number of independent factors combine to yield a certain parameter. A normal distribution is completely characterized by just two parameters: its mean and variance.

Now, let us create a plausible distribution of firing rates. There is evidence that the baseline firing rate of many neurons in visual cortex in the absence of visual stimulation hovers around 5 impulses/second (Adrians). Moreover, firing rates cannot be negative. This makes our choice of a normal distribution somewhat artificial, as it does—of course—yield negative values.

Hence, the following code will produce a plausible distribution of firing rates for background firing in the absence of a visual stimulus:

```
x = 0:0.01:10;
y = normpdf(x,5,1.5)
plot(x,y)
```

The third parameter of **normpdf** specifies the variance. In this case, we just pick an arbitrary, yet reasonable value—for neurons in many visual areas, the variance of the neural firing rate scales with and is close to the mean. Other values would also have been possible. Note that strictly speaking, it would make more sense to consider only integral firing rates, but for didactic reasons, we will illustrate the continuous case. This will not make a difference for the sake of our argument, and it is the more general case.

Now consider the distribution where the stimulus is, in fact, present. However, it is very faint. It is sensible to assume that this will change the firing rate of an individual neuron only very modestly (as the neuron needs the rest of the firing range to represent the remaining luminance range). A plausible distribution will be created by:

```
z = (normpdf(x,6,1.5));
plot(x,z)
```

In other words, we assume that adding the stimulus to the background adds only—on average—one spike per stimulus in this hypothetical example. For the sake of simplicity, we keep the variance of the distribution the same, in reality it would likely scale with the increased mean.

> ***Suggestion for Exploration:*** MATLAB has a large library of probability density functions. Try another one to model neural responses. A plausible starting point would be the poisson distribution, as it yields only discrete and positive values, as is the case with integral neural firing rates. Matlab offers the Poisson probability density function under the command **poisspdf**.

On a side note, this is a good point to introduce another class of Matlab functions, namely cumulative distribution functions. They integrate the probability density of a given distribution function (e.g., the normal distribution).

These are used for many calculations, as they provide an easy way to determine the integrated probability density of a given distribution at a certain cutoff point.

For example, **normcdf** is often used to determine IQ-percentiles.

As IQ in the general population is distributed with a mean of 100 and a standard deviation of 15, we can type:

```
normcdf(100,100,15)
```

to get the unsurprising answer:

```
ans =

   0.5000
```

If we want to find out the percentile of someone with an IQ of 127, we simply type:

```
>> normcdf(127,100,15)

ans =

   0.9641
```

In other words, the person has an IQ higher than 96.41% of the population.

Back to SDT. If you did everything right, you can now cast the problem in terms of signal detection theory. It should look something like Figure 6.1.

The plot in Figure 6.1 contrasts the case of stimulus absence versus stimulus presence; firing rate in impulses per second is plotted on the x-axis, whereas probability or frequency is plotted on the y-axis. The thick vertical black line represents the criterion level we chose.

The upper panel represents the case of an absent stimulus. For the cases to the right of the black line, the neuron concluded "stimulus present," even in the absence of a stimulus. Hence, they are false alarms. Cases to the left of the black line are correct rejections. As you can see, at a criterion level of 7.5 impulses per second, the majority of the cases are correct rejections.

The lower panel represents the case of a present stimulus. For the cases to the left of the black line, the neuron concluded "stimulus absent," even in the presence of a stimulus. Hence, they are misses. Cases to the right of the black line represent hits. At a criterion level of 7.5 impulses per second, the majority of the cases are misses.

We are now in a position to discuss and calculate the receiver operating characteristic (ROC) curve for this situation, defined by the difference in mean firing rate, variance, and

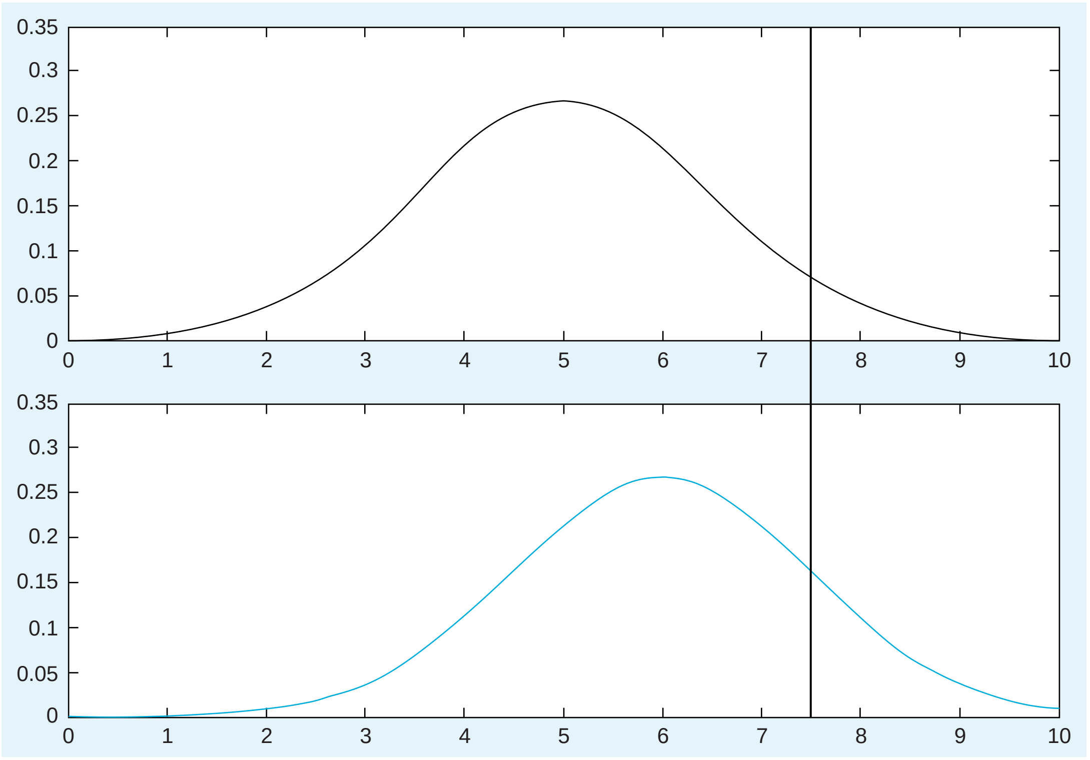


FIGURE 6.1 Signal absent versus signal present. Two different distributions.

shape of the distribution. The exotic-sounding term *receiver operating characteristic* originated in engineering, in particular the study of radar signals and their interpretation.

Generally speaking, an ROC curve is a plot of the false alarms (undesirable) against hits (desirable), for a range of criterion levels. "Area under the ROC curve" is a metric of how sensitive an observer is, as will be discussed later. Given the conditions that you can generally assume, ROC curves are always monotonically non-decreasing curves. In the context of tests, you plot the hit rate (or true positive rate or sensitivity) versus the false positive rate (or 1-specificity) to construct the ROC curve. Keep this in mind for future reference. It will be important.

First, try to plot the ROC curve:

```
figure
for i = 1:1:length(y) %Going through all elements of y
FA(i) = sum(y(1,i:length(y))); %Summing from ith element to rest → FA(i)
HIT(i) = sum(z(1,i:length(y))); %Summing from ith element to rest → Hit(i)
end
FA = FA./100; %Converting it to a rate
HIT = HIT./100; %Converting it to a rate
plot(FA,HIT) %Plot it
hold on
reference = 0:0.01:1; %reference needed to visualize
plot(reference,reference,'color','k') %Plot the reference
```

Note: This code could have been written in much more concise and elegant ways, but it is easier to figure out what is going on in this form.

To get the ROC curve, see Figure 6.2.

Note that the false alarms and hits are divided by 100 to get a false alarm and hit rate. The black line represents a situation in which hits and false alarms rise at the same rate – no sensitivity is gained at any point. As you can see, this neuron is slightly more sensitive than that, as evidenced by the deviation of its ROC curve from the black identity line. However, it rises rather gently. There is no obvious point where one should set the criterion to get substantially more hits than false alarms. This is largely due to the small difference in means between the distributions, which is smaller than the variance of the individual distribution. By experimenting with different mean differences, you can explore their effect on the ROC curves.

Exercise 6.1: Experiment with mean differences by yourself. The result should look something like Figure 6.3.

It becomes readily apparent that the ability to choose a criterion level that allows you to give a high hit rate without also getting a high false alarm rate is dependent on the difference between the means of the distributions. The larger the difference (relative to the variance) between the means, the easier it is to set a reasonable criterion level. For example, a mean difference of 5 allows you to get a hit rate of 0.9 virtually without any false alarms. This also gives a normative prescription to reduce false

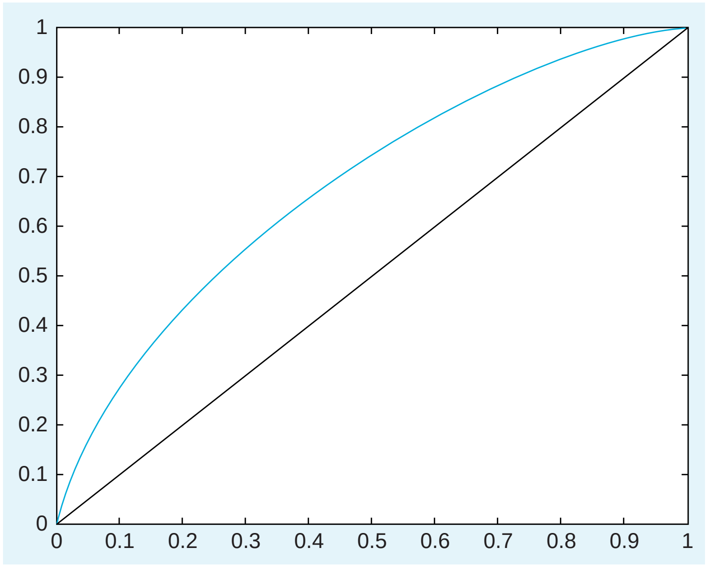


FIGURE 6.2 The ROC curve.

alarms, you should increase the difference in the means of the measured parameter between conditions of signal present (e.g., a fire) and signal not present (e.g., no fire). The clearer the parameters you choose differentiate between these two cases, the better off you will be. A similar case can be made for the variance of the signals. The less variance (often noise) there is in the signals, the better off you will be, when you are trying to distinguish between them. Hence, in order to create highly sensitive tests that discriminate between two situations, one needs to measure parameters that can be measured reliably without much noise but which exhibit a large difference in the mean parameter value, given the different situations in question.

Suggestion for Exploration: How are the ROC curves affected by increasing the variance of the distribution, while keeping the absolute mean difference the same?

The concept of difference (or distance) between means relative to the variance is of central importance to signal detection theory. Hence, it received its own name: Discriminability index (*d′* or *d prime)*. d′ is defined as the distance between the means of the two distributions normalized (divided) by the standard deviation common to the two distributions. It can be interpreted as a representation of signal strength relative to noise.

Importantly, d′ determines where an optimal criterion level should be set. For example, if d′ is very high, you can get 100% hits without any false alarms, by setting the criterion level properly. The situation is slightly more complicated when d′ is small, but there is a prescriptive solution for this case as well—it is discussed below.

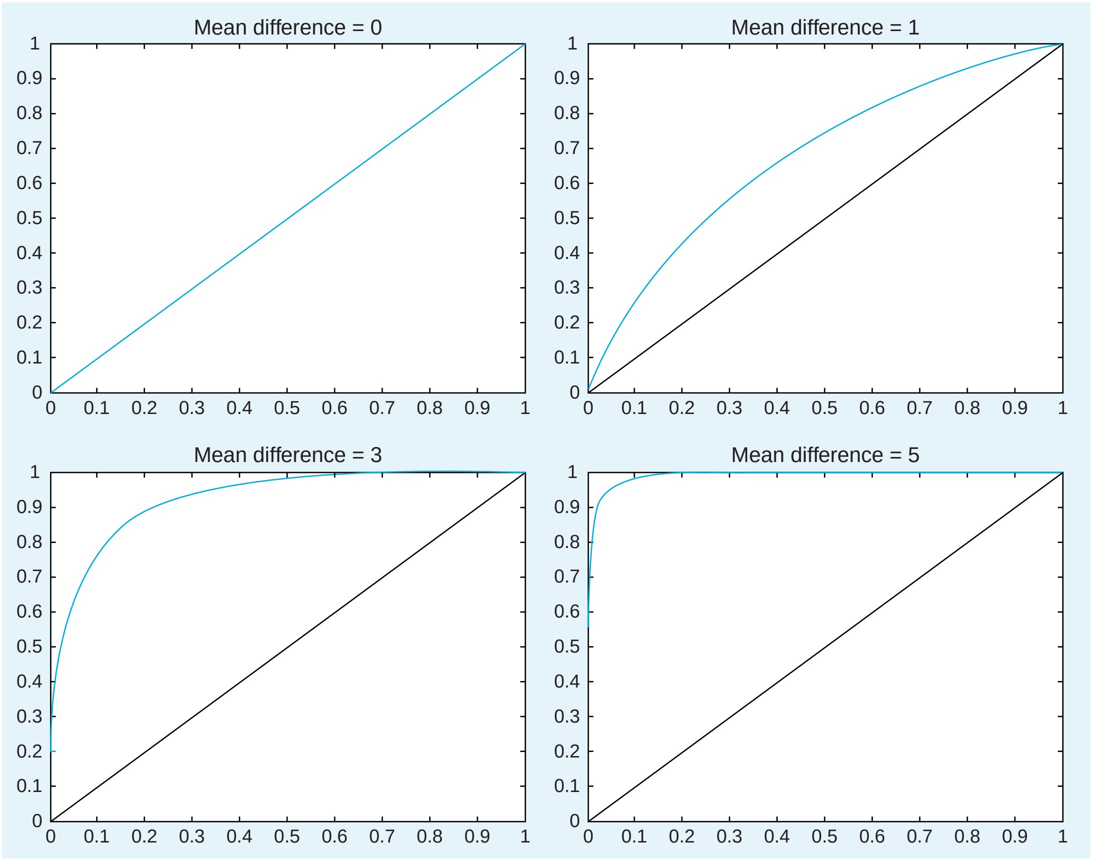


FIGURE 6.3 The shape of the ROC curve is dependent on the mean difference of the distributions..

This point makes intuitive sense. The errors derive from the fact that the "signal-present" and "signal-absent" distributions overlap. The more they overlap, the higher the potential for confusion. If the distributions don't overlap at all, you can easily draw a boundary without incurring errors or making mistakes.

Exercise 6.2: Consider Figure 6.4. It represents two distributions: one for "stimulus absent" on the left and one for "stimulus present" on the right. At which x-value would you put the criterion level? Can you plot the corresponding ROC curve (mean difference = 5, variance = 0.5)?

Suggestion for Exploration: Create a movie that shows the evolution of the ROC curve as a function of increasing d′ (for added insight, try various degrees of variance in the distribution). You can also download this movie from the website.

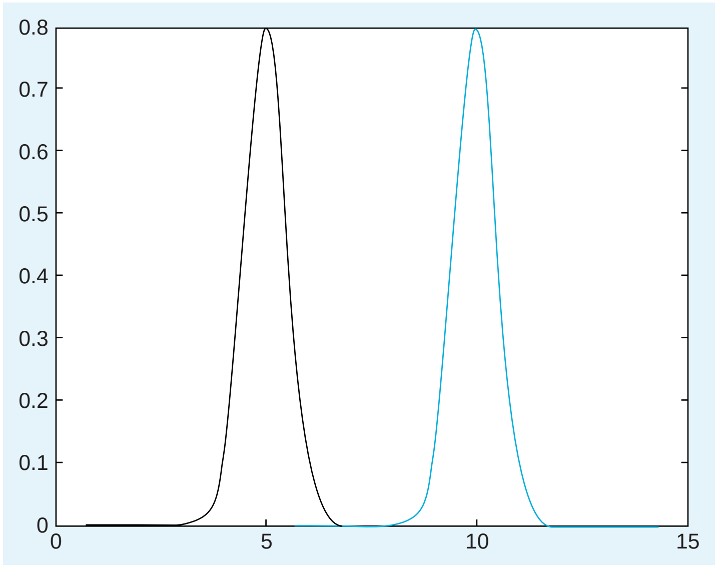


FIGURE 6.4 A case of high d′.

So far, so good. One central concept of signal detection theory that we are still missing is the notion of a likelihood ratio, or rather the use of likelihood ratios in signal detection theory.

While they sound rather intimidating, likelihood ratios are extremely useful because they are abstract enough to be powerful and flexible, yet specific enough to be of practical use. Hence, they are used in many fields, particularly diagnostics, but more generally in almost all of science. If you want to grasp the core of the concept, it is important to first strip off all these uses—some of which you might be already familiar with—and understand that it originally comes from statistics, or rather probability theory.

If you happen to appreciate analytical statistics, you might be appalled by the purely intuitive treatment of the likelihood ratio in this chapter. However, we deem this treatment appropriate for the purposes of our discussion.

Consider a situation in which you throw a fair and unbiased six-sided die. Each side has a probability of 1/6, which is about 0.1667. In other words, you expect the long-term frequency of a particular side to be 1 in 6. If you now want to know the probability that the die is showing one of the three lower numbers, you add the three individual probabilities and arrive at 0.5. Similarly, the probability that the die will show one of the three higher numbers is equally 0.5.

In other words, the ratio of the probabilities is $0.5/0.5 = 1$.

If you ask what the probability ratio of the upper 4 versus the lower 2 numbers is, you arrive at $(4{*}0.1667)/(2{*}0.1667) = 0.666/0.333 = 2/1 = 2$. In other words, the ratio of the probabilities is 2 and—in principle—you could call this a likelihood ratio.

In practice, however, the term *likelihood ratio* has a specific meaning, which we will briefly develop here.

To do so, we have to do some card counting. Let's say a deck of cards contains eight cards valued 2 to 9. In each round, the dealer draws two cards from this deck (without showing them to you). There is an additional, special deck that contains only two cards: one that is valued 1 and one that is valued 10. In the same round, the dealer draws one card from this special deck—again without showing it to you. However, the dealer does inform you of the total point value of all three cards on the table. Your task is to guess whether the card from the special deck is a 1 or a 10.

While this may sound like a rather complicated affair, the odds are actually hugely in favor of the player once you do an analysis of the likelihood ratios. So don't expect to see this game offered in Vegas any time soon.

Instead, let us analyze this game—we happen to call it Chittagong—for educational purposes. The highest possible point value in the game is 27, and it can happen only if you get the 10 in the special deck and the 8 and 9 in the normal deck. So there is only one way to arrive at this value. Similarly, the lowest possible point value is 6—by getting 1 in the special deck as well as 2 and 3 in the normal deck. This case is also unique. Everything else falls somewhere in between. So let us construct a table where we explore these possibilities (see Table 6.3).

Suggestion for Exploration: Can you re-create Table 6.3 with MATLAB using the permutation functions?

You can immediately see that vast regions of the table are not even in play. If the total value is below 15, you know that the special card had to be a 1. Moreover, if the total value is above 18, you know that the special card had to be a 10. Only four values are up to guessing, and even here, the odds are very good: As the player, you should guess 1 for 15 and 16, but 10 for 17 and 18. This state of affairs is due to the large difference between 1 and 10, relative to the possible range of normal values (5 to 17). In other words, d′ is very high in this game. This becomes immediately obvious when you plot the frequency distribution as histograms (10 = blue, 1 = black), as shown in Figure 6.5. This figure should look vaguely familiar (compare it to Figure 6.4).

Reducing the mean difference by 4 does change the distance between the distribution as well as the overall range. Suppose the cards in the special deck are replaced with two cards worth 0 and 5 points, respectively. What does the histogram of the frequency distributions look like now? (See Figure 6.6.)

The table of likelihood ratios, shown in Table 6.4, reflects this change.

As you can see, there is an intuitive and clear connection between likelihood ratio and d′. Of course, this relationship has been worked out formally. We will forgo the derivation here in the interest of getting back to neuroscience.

In this simple case, you can just set the criterion at the ratio between the probabilities. If this ratio is smaller than 1, guess 0. If it is larger, guess 5.

TABLE 6.3 Exploring the Likelihood Ratios in the Chittagong Game

Total points (TP)	Possible cases (=Probability) in which special deck card is 10	Possible cases (=Probability) in which special deck card is 1	Likelihood ratio (LR)
6	0	1	0
7	0	1	0
8	0	2	0
9	0	2	0
10	0	3	0
11	0	3	0
12	0	4	0
13	0	3	0
14	0	3	0
15	1	2	1/2 = 0.5
16	1	2	1/2 = 0.5
17	2	1	2/1 = 2
18	2	1	2/1 = 2
19	3	0	inf
20	3	0	inf
21	4	0	inf
22	3	0	inf
23	3	0	inf
24	2	0	inf
25	2	0	inf
26	1	0	inf
27	1	0	inf

In the technical literature, the likelihood ratio takes more factors into account: the prior probability as well as payoff consequences. Let us illustrate this case. Suppose there are not 2, but 10 cards in the special deck: You know that 9 have a value of 5, 1 has a value of 0. Hence, there is an a priori chance of 9/10 that the card will have a value of 5, and this does influence the likelihood ratio, as it should. Taking payoff consequences into account makes good sense because not all outcomes are equally good or bad (see the discussion at the beginning of the chapter). A casino could still make money off this game by adjusting the payoff matrix. For example, it could make the wins very small (as they are expected to happen often in a game like this), but the rare losses could be adjusted such that they are rather costly. A player has to take these considerations into account when playing the game and setting an optimal criterion value.

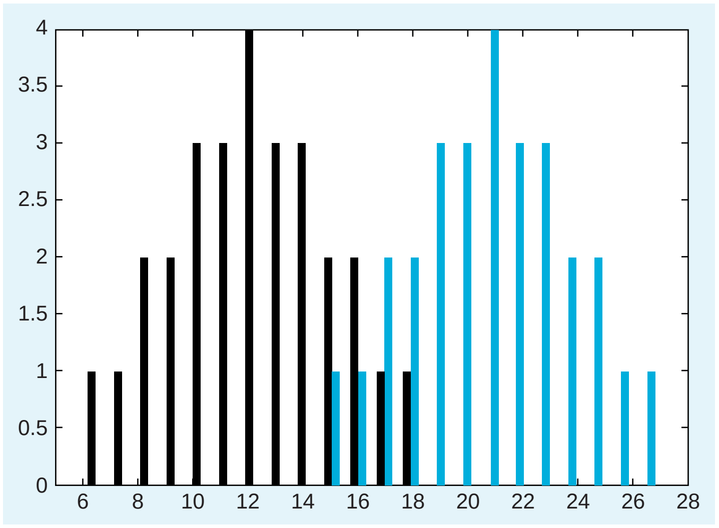


FIGURE 6.5 Histogram of frequency distributions.

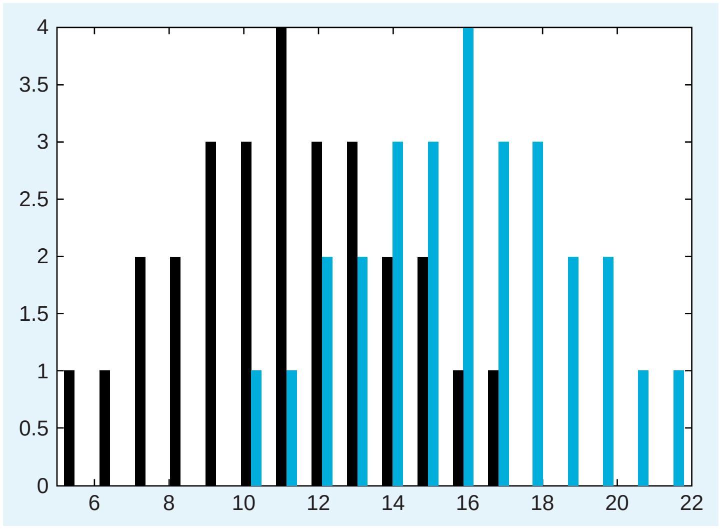


FIGURE 6.6 Histogram of frequency distributions with a smaller mean differences between the special cards (0 = black, 5 = blue).

TABLE 6.4 Revisiting Likelihood Ratios

Total points (TP)	Possible cases (=Probability) in which special deck card is 5	Possible cases (=Probability) in which special deck card is 0	Likelihood ratio (LR)
5	0	1	0
6	0	1	0
7	0	2	0
8	0	2	0
9	0	3	0
10	1	3	$1/3 = 0.33$
11	1	4	$1/4 = 0.25$
12	2	3	$2/3 = 0.66$
13	2	3	$2/3 = 0.66$
14	3	2	$3/2 = 1.33$
15	3	2	$3/2 = 1.33$
16	4	1	$4/1 = 4$
17	3	1	$3/1 = 3$
18	3	0	inf
19	2	0	inf
20	2	0	inf
21	1	0	inf
22	1	0	inf

To make this point more explicit, the likelihood ratio can be defined as follows:

$$l_{ij}(e) = \frac{p(e|s_i)}{p(e|s_j)} \tag{6.1}$$

So the likelihood ratio of an event e is the ratio of two conditional probabilities. One is the probability of the event given state s_i; the other, the probability of the event given state s_j. l_{ij} is always a single real number.

Moreover, we already discussed a more general situation where the likelihood ratio takes prior probabilities and payoffs into account:

$$l_{ij}(e) = \frac{stim\_frequency}{1 - stim\_frequency} * \frac{value\_of\_correct\_rejection - value\_of\_false\_alarm}{value\_of\_hit - value\_of\_miss} \tag{6.2}$$

This is particularly important for real life situations, where not all outcomes are equally valuable or costly.

As we alluded to before, the likelihood ratio is closely linked to ROC curves. Specifically, it is very important to characterize optimal behavior.

These considerations influence the likelihood ratio at which you should set your decision criterion. Importantly, there is a direct relationship between likelihood ratio and ROC curve: The slope of the ROC curve at a given point corresponds to the likelihood ratio criterion which generated the point (Green & Swets, 1966). In other words, an inspection of the slope can reveal where the criterion should optimally be set.

Let us illustrate these claims by revisiting the distributions introduced at the beginning of the chapter.

Use this code to plot the slope of the curves, analogous to Figure 6.2.

```
figure
x = 0:0.01:10;
%Note that x is ordered. If you start with empirical data, you will have to sort them first.
y = normpdf(x,5,1.5)
z = (normpdf(x,6.5,1.5));
subplot(2,1,1)
for i = 1:1:length(x)
FA(i) = sum(y(1,i:length(y)));
HIT(i) = sum(z(1,i:length(y)));
end
FA = FA./100;
HIT = HIT./100;
plot(FA,HIT)
hold on
baseline = 0:0.01:1;
plot(baseline,baseline,'color','k')

subplot(2,1,2)
for i = 1:length(x)-1
   m1(i) = FA(i)-FA(i+1); %This recalls the
   m2(i) = HIT(i)-HIT(i+1); %equation of a slope
end
m3 = m1./m2; %Dividing them
plot(m3)
```

The slope of the ROC curve is plotted in the lower panel, see Figure 6.7.

The philosophical implications of signal detection theory are deep. The message is that—due to the stochastic structure of the real world—infallibility is, in principle, impossible in most cases. In essence, in the presence of uncertainty (read: in all real life situations), errors are to be expected and cannot be avoided entirely. However, signal detection theory provides a precise analytical framework for optimal decision making in the face of uncertainty, while also being able to take into account subjective value judgments (such as preferring one kind of error over another).

As you might have noticed, we are really only scratching the surface here. Because situations in which a signal detection theory perspective is useful are truly ubiquitous—think of any kind of selection and quality control process, such as hiring decisions, admission decisions, marriage, dating, to say nothing of the myriad applications in materials science—signal detection theory has become a bottomless well. This should not be surprising, as it is arguably at the very heart of cognition itself. Yet, this led to a situation in which even specialists can be overwhelmed by

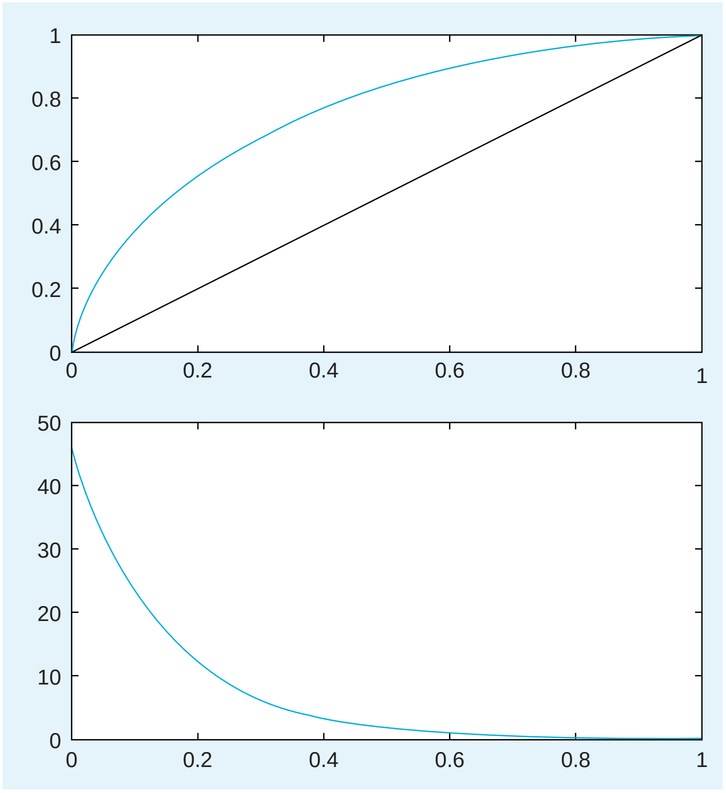


FIGURE 6.7 ROC curve with slope.

the intricacies in the field. Hence, the point of this brief treatment was to cover the conceptual essentials and their application. We are confident that it is enough to get you started in applying signal detection theory with MATLAB to problems in neuroscience.

For further reading, we highly recommend the classical and elaborate *Signal Detection Theory and Psychophysics* by Green and Swets (1966); the latest edition is still available in print. This book nicely highlights the role of signal detection theory in modern cognitive science in many colorful examples.

6.4. PROJECT

The project for this chapter is very straightforward. Many uses of signal detection in neuroscience involve the measurement of some "internal response" in addition to measuring a behavioral response (e.g., deciding whether a stimulus under the control of the experimenter is present or not). We assume that you do not currently have access to measure a "deep" internal response, such as firing rate of certain neurons that are presumably involved in the task. Instead, we ask you to redo the experiment in Chapter 5, but with a twist. Instead of just asking whether a faint stimulus is present or not, now elicit 2 judgments per trial: One whether the stimulus is present or not, the other how confident the subject is that it was present or not, on a scale from 1 (not certain at all) to 9 (very certain). Replot the data in terms of certainty. Get two distributions of certainty (one for situations where the stimulus was present, the other where it was not present) Doing so, please answer and explore the following questions:

- Where does the internal criterion of the subject lie?
- What is the d′ of the certainty distributions?
- Construct the ROC curve for the data (including slope).
- How sensitive is the subject as an observer? (Compare the area under the ROC curve with the area under the diagonal reference curve.)
- Can you increase the d′ by showing a different kind of stimulus?
- Can you shift the position of the criterion by biasing the payoff-matrix for your subject (e.g., rewarding the subject for hits)?

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

normpdf
normcdf
sum

CHAPTER

7

Frequency Analysis Part I: Fourier Decomposition

7.1. GOALS OF THIS CHAPTER

This chapter introduces the most common method of decomposing a time series into frequency components, Fourier analysis. You will learn about the Fourier transform and the associated amplitude and phase spectra. The MATLAB implementation of the Fast Fourier Transform (FFT), an efficient algorithm for calculating Fourier transformations, will be introduced and applied to the analysis of human speech sounds.

7.2. BACKGROUND

Figure 7.1 shows typical recordings of two human vowel sounds. How can you characterize these different sounds? Frequency analysis provides a way to examine the relative contributions of various frequencies to an overall signal. In the case of an auditory signal, a given frequency component would be termed *pitch.*

7.2.1. Real Fourier Series

Take some continuous function f. We can approximate such a function with a weighted series of sinusoids. Such a series is termed the real Fourier series:

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(nt) + \sum_{n=1}^{\infty} b_n \sin(nt) \tag{7.1}$$

Here, the coefficients a_n and b_n represent the relative strength of each frequency component. [a_0 represents whatever nonoscillatory component of $f(t)$.]

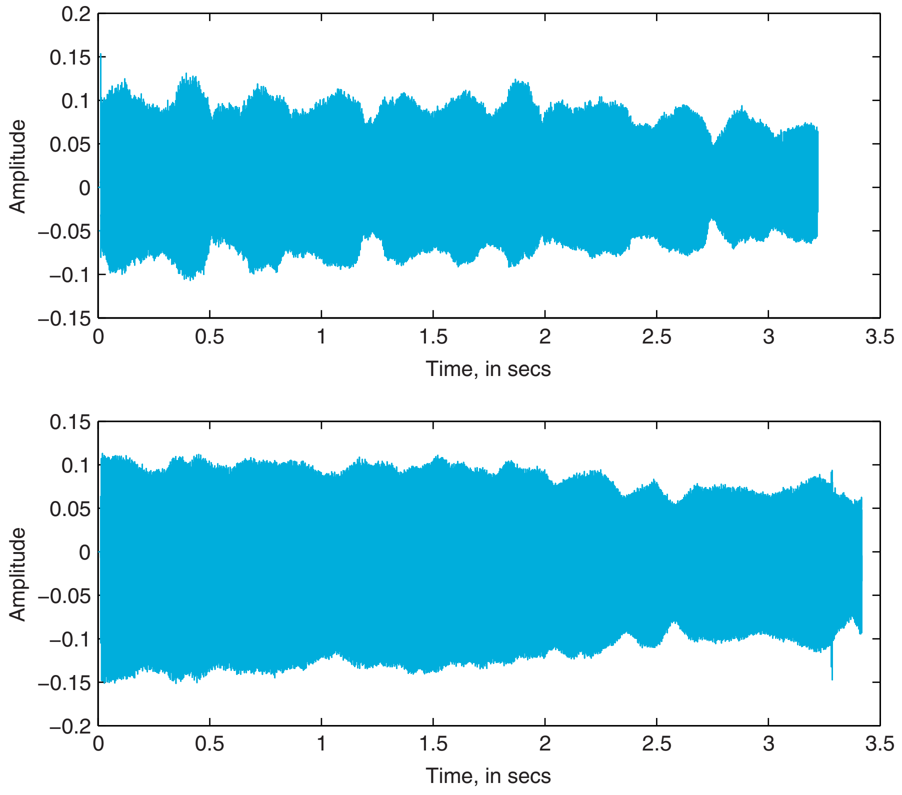


FIGURE 7.1 Acoustic time series representing two different human vowel sounds.

So, given *f(t)*, determining the coefficients a_n and b_n allows for the representation of *f(t)* as a series sum of sinusoids. Over the interval $-\pi$ to π, cosine and sine functions with differing frequencies have the special property of *orthogonality*. The integral of the product of two mutually orthogonal functions evaluates to zero. So, the integral of the product of cosine or sine functions with differing frequencies results in zero over this interval. Another interesting property of sine and cosine is that the integral of the square of a cosine or sine function over this integral is π. Both of these properties will be exploited to determine the Fourier series coefficients.

To find the strength, a_m, of a cosine component m, multiply by the corresponding cosine function and integrate:

$$\int_{-\pi}^{\pi} f(t)\cos(mt)dt = \int_{-\pi}^{\pi} \frac{a_0}{2}\cos(mt)dt + \sum_{n=1}^{\infty}\int_{-\pi}^{\pi}\cos(mt)\,a_n\cos(nt)dt + \sum_{n=1}^{\infty}\int_{-\pi}^{\pi}\cos(mt)b_n\sin(nt)dt \tag{7.2}$$

All terms on the right side except the cosine term where $m=n$ yield zero:

$$\int_{-\pi}^{\pi} f(t)\cos(mt)dt = a_m \int_{-\pi}^{\pi} \cos^2(mt)dt \tag{7.3}$$

The right side integral evaluates to one over the integration range, yielding an expression for the Fourier series term coefficient:

$$\int_{-\pi}^{\pi} f(t)\cos(mt)dt = \pi a_m \tag{7.4}$$

$$a_m = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\cos(mt)dt \tag{7.5}$$

In general, the interval of *f(t)* will not be $-\pi$ to π. For an interval centered on x with length $2L$, the expression becomes

$$a_m = \frac{1}{L}\int_{x-L}^{x+L} f(t)\cos\left(\frac{\pi}{L}mt\right)dt \tag{7.6}$$

A similar procedure using sine functions yields the coefficients for the sine terms of the Fourier series.

7.3. EXERCISES

Exercise 7.1: Write a MATLAB function to calculate coefficients for a real Fourier transform. *Hint:* The function will need to shift the interval so that the interval encompasses the entire time series. In other words, $x = 0$ and $L =$ half the range of t.

7.3.1. Complex Fourier Transform

Euler's identity,

$$e^{i\omega t} = \cos\omega t + i\sin\omega t \tag{7.7}$$

provides a straightforward way to formulate complex Fourier series representation for a given function, *f*(*t*):

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{\text{int}} \tag{7.8}$$

Similar to the real transform, coefficients for the complex Fourier transform can be found by

$$c_m = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(t)e^{-imt}dt \tag{7.9}$$

for a given coefficient m over the interval $-\pi$ to π. Over the interval $x-L$ to $x+L$, this becomes

$$C_m = \frac{1}{2L}\int_{x-L}^{x+L} f(t)e^{-(i\pi mt/_L)}dt \tag{7.10}$$

Exercise 7.2: Write a MATLAB function to calculate coefficients for a complex Fourier transform. This is essentially the discrete Fourier transform (DFT):

$$F_k = \frac{1}{N}\sum_{n=0}^{N-1} f_n e^{-i\,\frac{2\pi n}{N}\,k} \tag{7.11}$$

where k ranges from 0 to $N-1$, N is the number of points, and f_n is the value of the function at point n.

Let's look at how this method of the Fourier transform scales with N. Given a time series with N values, this method requires a multiplication of the series and the corresponding Fourier component and subsequent sum for each coefficient. Assuming a number of coefficients equivalent to N, then you have a process that scales with N^2. In other words, as N increases, the time required to compute the Fourier transform increases as N^2.

7.3.2. Fast Fourier Transform

With a few special tricks, a faster algorithm, the Fast Fourier transform (FFT), that scales in $N \log N$ time can be formulated. One of these tricks involves taking advantage of datasets exactly 2^N elements long. The increase in processing speed has made the FFT ubiquitous in signal processing. While a complete derivation of the algorithm is beyond the scope of this book, invoking the MATLAB implementation of the FFT will be discussed.

MATLAB provides an FFT function **fft(*X*)**, where X is a vector in time space. **fft** returns the frequency space representation of X.

To visualize the importance of the difference in scaling, execute the following code:

```
figure
hold on
N = 1:10 * 100;
plot(N, N.^2, 'b')
plot(N, N.*log(N), 'r')
```

Exercise 7.3: If N represents sample size, what can you observe about the benefits of scaling as N grows? Where does the efficiency of the FFT algorithm benefit most, for large N or small N?

7.3.3. The Inverse DFT

As you might imagine, there is an inverse to the DFT:

$$f_n = \sum_{k=0}^{N-1} F_k e^{i\frac{2\pi n}{N}} k \tag{7.12}$$

MATLAB provides **ifft()** to perform the inverse discrete Fourier transform.

Exercise 7.4: Generate a single sine wave. Use **fft()** to generate the discrete Fourier transform. Use **ifft()** to retrieve the original sine wave from the DFT.

7.3.4. Amplitude Spectrum

Often when you are using Fourier analysis, the amplitude spectrum is one of the first analyses performed. The amplitude spectrum graphs amplitude against frequency. In terms of the Fourier series representation, the amplitude spectrum depicts the magnitude of the coefficients at various frequencies. As such, it depicts the relative strengths of the various frequency components.

The following code generates a time series composed of 10 sine waves whose frequencies and amplitudes vary systematically.

```
L = 1000;
X = zeros(1,L);
sampling_interval=0.1;
t = (1:L) * sampling_interval;
for N = 1:10
        X = X + N * sin (N*pi*t);
end
plot(t, X);
Y = fft(X)/L;
```

Now, the variable Y contains the normalized FFT of X. Note the normalization factor L. Displaying the amplitude spectrum of X requires plotting the amplitudes at various frequencies. Note that **fft** returns only a single value, the transform coefficients. Now, how do you determine the frequency scale?

The return value of the FFT assumes that frequency is evenly spaced, from 0 to a theoretical result called the *Nyquist limit*. Nyquist demonstrated that a discrete sampling of a continuous process can capture frequencies no higher than half the sampling frequency. Since the code above has the sampling interval, this **Nyquist limit** is half the inverse of the sampling interval.

The following code calculates the Nyquist limit for the time series:

```
NyLimit = (1 / sampling_interval)/ 2;
```

When viewing the FFT, it is important to remember that the result is the complex transform. Thus, simply using the result of the FFT as a set of real coefficients can cause a number of problems. To display the amplitude spectrum, the absolute value of the complex coefficients will be used. The values returned by **fft** are the coefficients for frequencies from the negative Nyquist limit to the positive Nyquist limit. If the time series data are purely real, then the resultant transform will have even symmetry. That is, the transform will be symmetrical across the abscissa. So, in this very frequent case, only the first half of the result of **fft** is used. The following code employs **linspace** to generate frequency values and plots the amplitude spectrum. **linspace** generates a linearly spaced sequence of values given initial and final values. Here, the initial and final values are 0 and 1, with a value count of L/2. The resultant vector is scaled by the Nyquist limit to generate the frequency vector.

```
F = linspace(0,1,L/2)*NyLimit;
plot(F, abs(Y(1:L/2)));
```

7.3.5. Power

Power at a given frequency is defined as

$$\Phi(\omega) = |F(\omega)|^2 = F(\omega)F^*(\omega) \tag{7.13}$$

where F^* is the complex conjugate of F. To do this in MATLAB, use the function **conj** to return the complex conjugate of a series of complex values.

Here is a plot of the power spectrum of the time series generated for the amplitude spectrum:

```
plot(F, (Y(1:L/2).*conj(Y(1:L/2)));
```

7.3.6. Phase Analysis and Coherence

A power spectrum alone is not a complete representation of the information in the original signal. The various Fourier components can have various phases relative to one another, as illustrated in Figure 7.2

You can plot relative phase by frequency by plotting the inverse tangent of the ratio between the imaginary component and the real component. Why is this the case? Imagine the complex plane, with pure real values along the abscissa (x-axis) and pure imaginary values along the ordinate (y-axis). Any complex value in your 1D Fourier transform can be represented with a coordinate pair. The magnitude of the value is simply the distance from the origin to the coordinates, or the complex modulus. The phase is the angle formed by the abscissa and the line passing through the origin and the complex point. Thus, using basic trigonometry, the phase angle is $\tan^{-1}\left(\frac{imag}{real}\right)$.

How can you represent this in MATLAB?

```
L = 1000;
X = zeros(1,L);
```

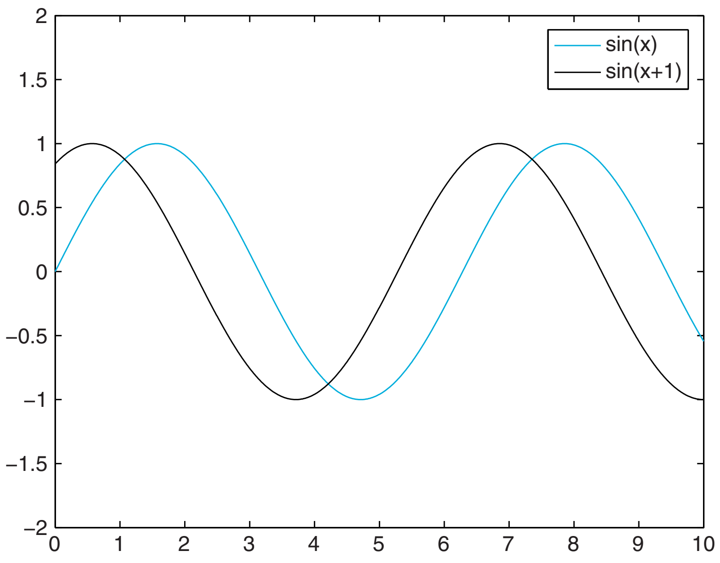


FIGURE 7.2 Phase difference between two sinusoids with the same frequency.

```
sampling_interval=0.1;
t = (1:L) * sampling_interval;
for N = 1:10
        X = X + N * sin (N*pi*t);
end
plot(t, X);
Y = fft(X)/L;
phi = atan(imag(Y)./real(Y));
F = linspace(0,1,L/2)*NyLimit;
plot(F, phi(1:L/2));
```

Exercise 7.5: Compare the phase spectrum generated in the preceding exercises with the phase spectrum of the corresponding cosine function. Compare their power spectra.

7.4. PROJECT

In this project, you will be asked to use Fourier decomposition to analyze vowel sounds produced by human speakers. On the companion website, you will find five examples of vowel sounds as produced by male American English speakers. Each sound corresponds to one of the vowel sounds in Table 7.1, below. The formant frequencies in Table 7.1 note the average formant frequencies as spoken by a male speaker of American English. You will use power spectra of these sounds to classify the recordings as one of these vowel sounds in the table.

To complete this project, you need to understand how formants relate to frequency analysis. The human vocal tract has multiple cavities in which speech sounds resonate. As such, most

TABLE 7.1 Average First and Second Formant Frequencies for Selected American English Vowels

Vowel Sound	First Formant	Second Formant
bit	342	2322
but	623	1200
bat	588	1952
boot	378	997

(Data from Hillenbrand et al., 1995.)

sounds have multiple strong frequency components. In classifying speech sounds, the lowest strong frequency band is termed the *first formant*. The next highest is termed the *second formant*, and so on.

Vowels lend themselves to a particularly simple characterization through their formants. Typically, vowel sounds have distinguishable first and second formants. Table 7.1 shows first and second formants for four vowel sounds in American English. Thus, the short 'i' sound would have strong frequency representation at 342 Hz and at 2322 Hz.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

fft
ifft
conj

CHAPTER

8

Frequency Analysis Part II: Nonstationary Signals and Spectrograms

8.1. GOAL OF THIS CHAPTER

The goal of this chapter is to extend Fourier analysis as covered in the previous chapter to nonstationary signals. The short-time Fourier transform will be introduced. Nonstationary examples will include applications to time-varying auditory signals and the EEG during sleep.

8.2. BACKGROUND

Figure 8.1 depicts the vocalizations of a zebra finch. How is this dissimilar from the sound signals you have examined thus far?

Note that different portions of the song have different envelopes with clearly defined breaks. If they are taken separately, you might imagine these subsections to have different Fourier spectra. In fact, they do. Figure 8.2 shows the Fourier spectrum for two subsections of song. The two subsections have very different distributions of power over frequency.

A Fourier transform of the full song returns the power distribution over the entire song. Any localization of frequency information to a time point or an interval is lost. Using the example of the bird song here, a Fourier transform of the entire song would eliminate any ability to associate frequency components with a given syllable. How, then, can you extend the techniques discussed earlier to such complex signals?

8.2.1. The Fourier Transform: Stationary and Ergodic

When applied to a signal, the term *stationary* indicates that certain statistical properties of the signal are uniform throughout. In other words, a subset of the signal is sufficient for

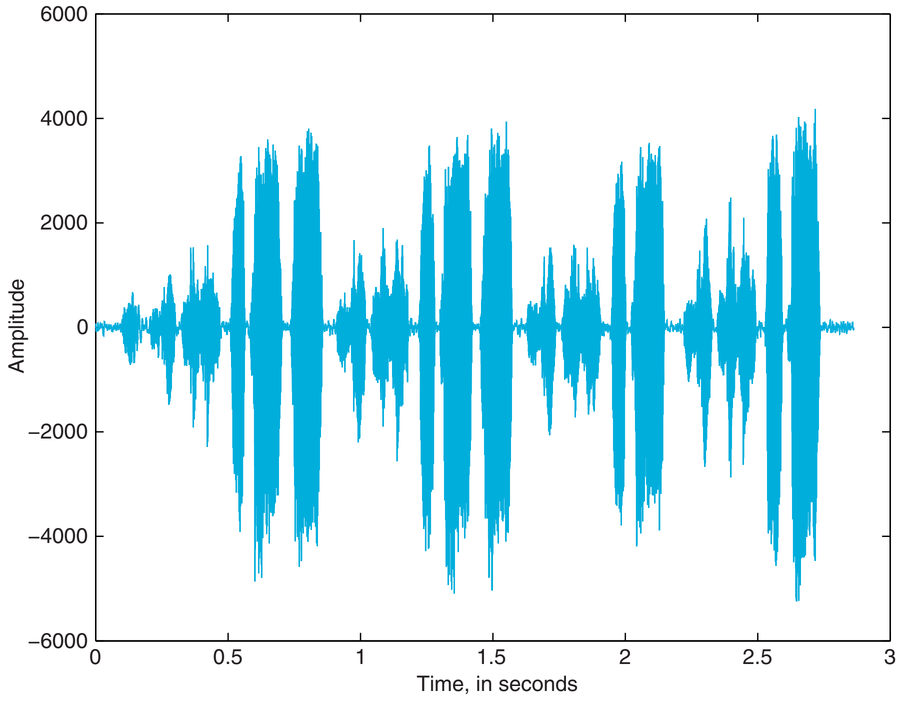


FIGURE 8.1 The sound amplitude of a zebra finch vocalization as a function of time.

analysis of the entire signal. The distribution of power over frequency remains the same over the whole signal.

A similar idea is the concept of *ergodicity*. Imagine an ensemble of related signals. Going with the example of zebra finch vocalizations, an appropriate ensemble would be the set of vocalizations from a set of birds. An ergodic ensemble is one in which each sample and the ensemble approach the same mean. In other words, analyzing one sample or a subset of the signals from the group can approximate the analysis of the ensemble. Ergodicity and stationarity are independent qualities. Neither implies the other.

The Fourier transform assumes a stationary signal. Unfortunately, many biological signals, including the birdsong in Figure 8.1, are nonstationary.

8.2.2. Windows

How can you employ the Fourier transform to a nonstationary signal? If you assume that the Fourier spectrum will change relatively little over a small interval of the signal, you could divide the overall signal into windows and calculate the Fourier transform for each window separately. If a signal is relatively stationary over short intervals, or quasistationary, this approach will often produce fruitful results. While many biological signals are not truly stationary, many are quasistationary and amenable to this approach.

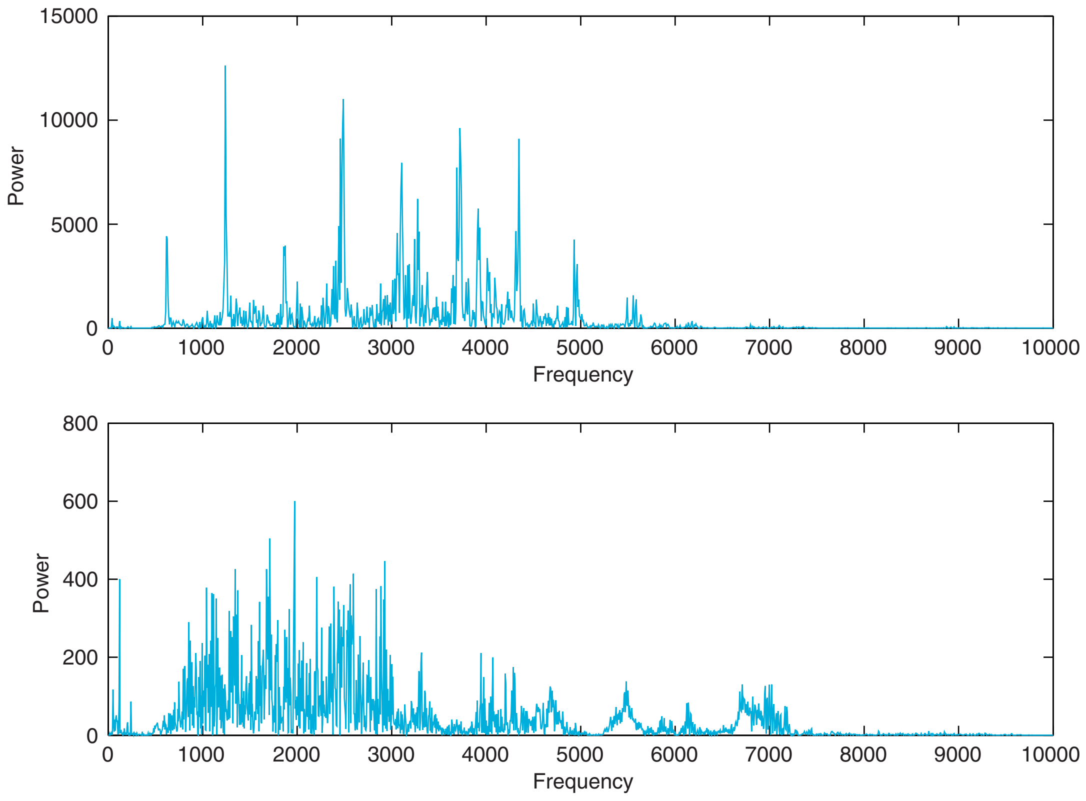


FIGURE 8.2 The power spectra of two portions of the zebra finch vocalizations depicted in Figure 8.1.

However, this approach breaks down somewhat at the interval boundaries, due to the stationary assumptions of the Fourier transform. Choosing overlapping intervals mitigates this somewhat. This is the basis for the short-time Fourier transform (STFT).

While a simple flat subset of the original time series might be the most straightforward window, an appropriate choice of window shape can amplify or minimize characteristics of the time series. For example, windows with tapered ends are used to minimize artifacts from the edges of the window. You can introduce the idea of a generalized window function, $w(t)$, which returns the value of the window at a given value of t. For values outside the window, $w(t)$ should return values equal to or close to 0.

Mathematically, the STFT is represented as:

$$X(\tau, \omega) = \int_{-\infty}^{\infty} x(t) w(t-\tau) e^{-j\omega t} dt \tag{8.1}$$

for a continuous signal. In this case, we are more interested in the discrete STFT,

$$X(m, \omega) = \sum_{n=-\infty}^{\infty} x(n) w(n-m) e^{-j\omega n} \tag{8.2}$$

As mentioned previously, there are many alternatives to the simple squared off window for calculating an STFT. We will briefly discuss three. The Hamming window function is commonly used:

$$w(n) = 0.53836 - 0.46164\cos\left(\frac{2\pi n}{N-1}\right) \tag{8.3}$$

where N is the number of points and n varies over the interval.

Signal Processing Toolbox of the MATLAB® software provides the function **hamming**, which returns a Hamming window of the desired length:

```
L = 100;
w = hamming(L);
plot(1:L, w)
```

Note that the Hamming window has a high amplitude at the center and low amplitude at the ends. This attenuation reduces the artifacts from the edge of the window interval. Another window function is the Hann window, whose functional form is similar to the Hamming window:

$$w(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{N-1}\right) \tag{8.4}$$

Like the Hamming window, the shape of the Hann window is used to reduce artifacts introduced at the edges of the finite windows from the signal. Gaussian window functions are often used as well:

$$w(n) = e^{-n^2} \tag{8.5}$$

A short-time Fourier transform using a Gaussian window function is sometimes denoted as a *Gabor transform*. A Gabor function is the product of a sinusoid and a Gaussian function. The Gaussian function causes the amplitude of the sinusoid to diminish away from the origin, but near the origin, the properties of the sinusoid dominate. By applying a Gaussian window and a Fourier transform to the time series, you are, in effect, applying a Gabor function filter to the data.

8.3. EXERCISES

As a part of Signal Processing Toolbox, MATLAB provides the function **spectrogram**, which calculates a short-time Fourier transform using a Hamming window. The data for Figure 8.1 is available on the companion website. Download the file song1.wav and load the file with **wavread**, as follows:

```
[amp, fs, nbits] = wavread('song1.wav');
```

The function **wavread** loads a sound file in WAVE format and returns the data as amplitude information ranging from –1 to +1. Here, you store the amplitude information in the variable *amp*. The sampling rate is returned in *fs*, and the number of bits per sample (resolution) is stored in *nbits*.

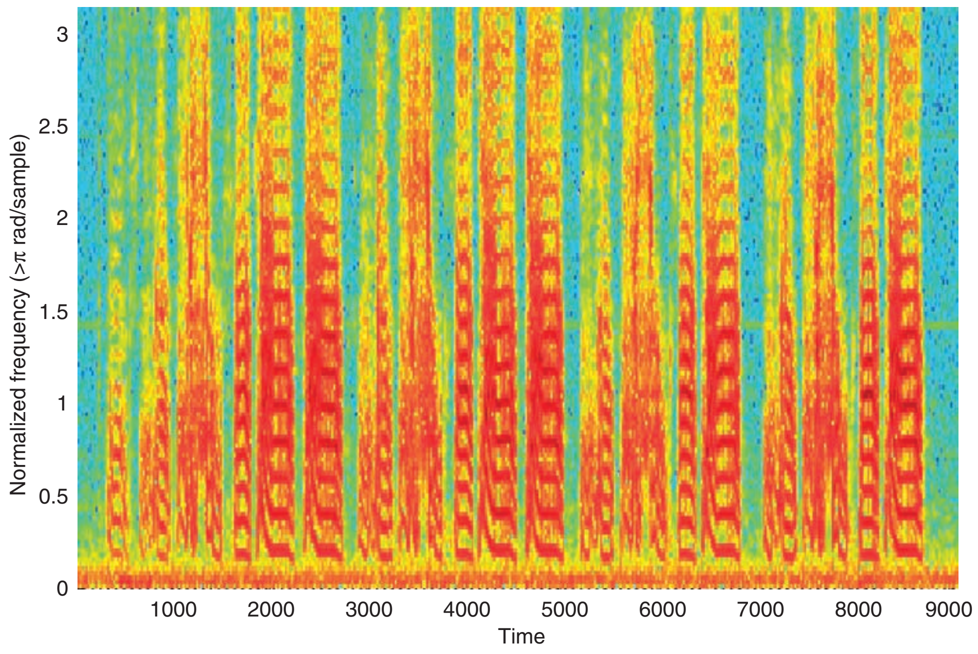


FIGURE 8.3 The spectrogram of the bird vocalization using the **spectrogram** function.

Now type

spectrogram(amp, 256, 'yaxis')

You should see something like Figure 8.3. The default operation of spectrogram calculates power of the signal by dividing the whole signal into eight portions with overlap and windowing the portion with a Hamming window. Here, the specified window size was 256. The optional parameter *'yaxis'* specifies that frequency should be on the y-axis rather than x-axis. If no return values are specified, the default operation renders the power spectral density over time using "hotter" colors (red, yellow, etc.) to designate frequency bands of greater energy.

If a sampling frequency is not specified, the time scale will not be correct. To show the correct time space for the loaded song, type

```
>> spectrogram(amp, 256, [ ], [ ], fs, 'yaxis')
```

Here, the empty brackets signify that the default settings for the window overlap, and FFT size should remain.

Also, **spectrogram** can return the power spectral density:

```
>> [S, F, T, P] = spectrogram(X);
>> mesh(P)
```

The preceding code generates a 3D plot of the spectrogram, where *z* magnitude, rather than color, represents power.

Exercise 8.1: In the bird song sample, try to determine where the sound changes using the time series data alone. Do the same with the STFT. Do your results agree?

Exercise 8.2: Examine the result of spectrogram with varying window sizes for the following time series:

```
>> t = 0:0.05:1;
>> X = [sin(5*t) sin(50*t) sin(100*t)];
```

Try values ranging from 16 to 1024 for the Hamming window width. How does the representation change with different Hamming window widths? Why might this occur?

8.3.1. *Limitations of the STFT*

The STFT is a fine resolution to the problem of determining the frequency spectrum of signals with time-varying frequency spectra. There are some limitations. Small frequency fluctuations are difficult to detect with the STFT because each subset of the signal is assumed to be stationary. Since the reported frequency distribution at a time point results from the analysis of a the entire window, choosing a smaller window does allow for better localization in time. However, a smaller window allows for fewer samples in each Fourier transform, which ultimately reduces frequency resolution, especially for lower frequencies. In other words, a trade-off exists between frequency and time localization.

The STFT is best employed when the fluctuations in frequency occur over a fairly uniform time scale. This allows selecting a single window size without substantial loss of information.

8.4. PROJECT

Typical sleep in human adults includes the well-known REM sleep as well as four well-characterized stages of non-REM sleep, or NREM sleep. During wakefulness, alpha waves dominate the EEG, in the frequency range 8 to 13 Hz. As the subject enters the first stage of non-REM sleep, the dominant wave type transitions from alpha waves to theta waves, in the range of 4 to 7 Hz. This is the first stage of non-REM sleep.

The second and third stages of non-REM sleep are characterized by sleep spindles, at 12 to 16 Hz, and the appearance of delta waves, ranging in frequency from 0.5 to 4 Hz. The fourth stage of sleep is characterized by a majority power distribution in the delta wave band. The third and fourth stages of NREM sleep are also termed slow wave sleep, to denote the prevalence of the low frequency delta waves in these two stages.

On the companion website, you can find three EEGs from patients falling asleep. Using spectrogram and any other frequency analysis tools learned thus far, try to determine when the subjects enter each of the NREM stages of sleep.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

hamming
spectrogram
wavread

CHAPTER

9

Wavelets

9.1. GOALS OF THIS CHAPTER

In this chapter, you will be introduced to the use of wavelets and wavelet transforms as an alternative method of spectral analysis. We will discuss a number of common wavelets and introduce Wavelet Toolbox of the MATLAB® software.

9.2. BACKGROUND

In Chapter 8, "Frequency Analysis Part II: Nonstationary Signals and Spectrograms," you used the short-time Fourier transform (STFT) to decompose the frequency composition of nonstationary signals. Under certain situations, though, the STFT results in a less-than optimal breakdown of frequency as a function of time. With increased precision in frequency distribution, localization in time becomes less precise. In other words, there is a time – frequency precision tradeoff. The reverse is also true: better temporal localization reduces the precision of the frequency distribution. This may bring to mind the well-known relationship of position and momentum of the Heisenberg uncertainty principle.

When using the STFT, you can adjust the transform window to enhance the desired characteristic. A larger window allows for better frequency resolution, and a smaller window allows for better temporal resolution. However, for the STFT, the window size is constant throughout the algorithm. This can pose a problem for some nonstationary signals. The wavelet transform provides an alternative to the STFT that often provides a better frequency/time representation of the signal.

9.2.1. What Is a Wavelet?

A *wavelet* is a function that satisfies at least the following two criteria:

1. The integral of the function $\psi(x)$ over all x is 0.

$$\int_{-\infty}^{\infty} \psi(x)dx = 0 \tag{9.1}$$

2. The square of $\psi(x)$ has integral 1. A function adhering to this property is called *square-integrable*.

$$\int_{-\infty}^{\infty} \psi^2(x)dx = 1 \tag{9.2}$$

Fulfilling the first criterion mandates that the wavelet function have an equal area above and below zero. Fulfilling the second criterion mandates that the function approach zero at positive and negative infinity. Because of this second criterion, the function decays away from the origin, unlike sinusoidal or other infinite waves (thus, wave*let*).

9.2.2. The Continuous Wavelet Transform

The continuous wavelet transform (CWT) is analogous to the continuous Fourier transform:

$$W(s,t) \equiv \int_{-\infty}^{\infty} x(u)\psi_{s,t}(u)du \tag{9.3}$$

Here, the parameter t is the typical t in the time series *x(t)*. The parameter s is called *scale* and is analogous to frequency for Fourier transforms. The wavelet function itself varies with both s and t:

$$\psi_{s,t}(x) \equiv \frac{1}{\sqrt{s}}\psi^*\left(\frac{x-t}{s}\right) \tag{9.4}$$

The inclusion of t and s allows the function to be scaled and translated (shifted) for different values of s and t. The original wavelet function (untranslated and unscaled) is often termed the *mother wavelet*, since the set of wavelet functions is generated from that initial function.

The scaling provides a significant benefit over the short-time Fourier transform. The multiple scales of the wavelet transform permit the equivalent of large- or small-scale transform windows in the same time series. The preceding transform can be approximated for a discrete time series.

9.2.3. Choosing a Wavelet

A number of wavelet functions are commonly used in data analysis. Here are two used primarily for spectral analysis.

Morlet wavelet (for large ω_0):

$$\psi(t) = \pi^{-\frac{1}{4}} e^{-\frac{1}{2}t^2} e^{-i\omega_0 t} \tag{9.5}$$

The Morlet wavelet was originally developed to analyze signals with short, high-frequency transients and long, low-frequency transients (see Figure 9.1).

Mexican hat wavelet:

$$\psi(t) = \frac{1}{\sqrt{2\pi}\sigma^3}\left(1 - \frac{t^2}{\sigma^2}\right) e^{\frac{-t^2}{2\sigma^2}} \tag{9.6}$$

The Mexican hat wavelet has poorer frequency resolution than the Morlet wavelet, but often better temporal resolution.

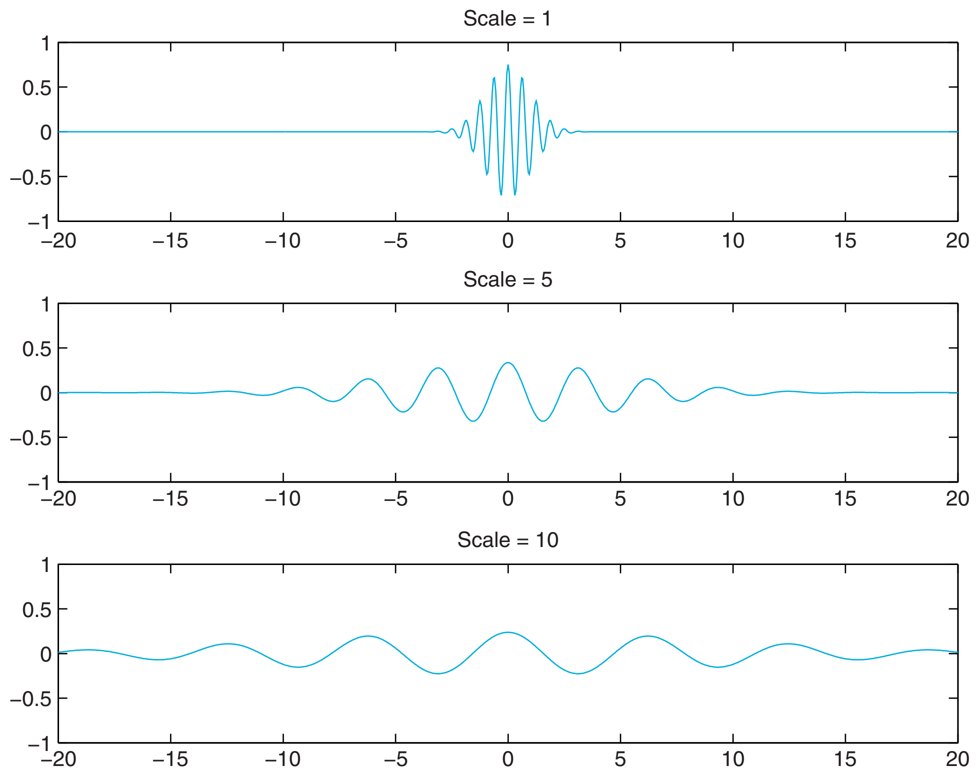


FIGURE 9.1 Morlet wavelet at various scales.

9.2.4. Scalograms

The *scalogram* depicts the strength of a particular wavelet transform coefficient at a point in time. As such, it is the wavelet analog of the spectrogram.

The scalogram in Figure 9.2 shows the continuous wavelet transform of the following signal with a Morlet wavelet (sigma = 10). This code generates a time series with three long blocks of time at 100, 500, and 1000 Hz. At every half second, a 0.05 transient at 1000 Hz is inserted.

```
Fs = 5000;
total_time = 5;
t = (1/Fs):(1/Fs):(total_time/3);
f = [100 500 1000];
x = [cos(f(1)*2*pi*t) cos(f(2)*2*pi*t) cos(f(3)*2*pi*t)];
t = (1/Fs):(1/Fs):total_time;

%add short transients
trans_time = 0:(1/Fs):0.05;
trans_f = 1000;
for secs = 0.5:0.5:4
    trans = cos(trans_f*2*pi*trans_time);
    x((secs*Fs):(secs*Fs+length(trans)-1)) = trans;
end
```

Be aware that the relationship between scale and frequency is an inverse one and that frequency increases with *decreasing* scale. Also, note how the frequency resolution improves for the higher frequency band in the later third of the series. This corresponds to the 1000 Hz section of the time series.

The code to generate and plot the CWT follows.

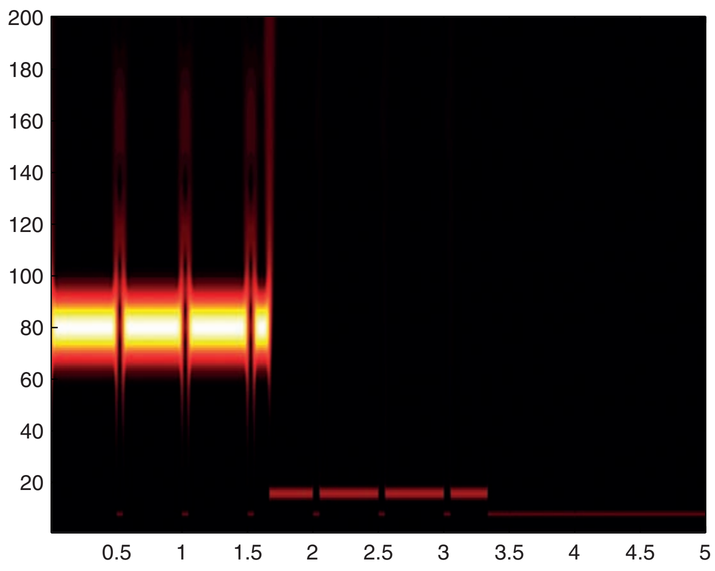


FIGURE 9.2 Scalogram for sinusoid + transient signal in text.

In **my_cwt.m:**

```
function coefs = simple_cwt(t, x, mother_wavelet, max_wavelet, scales, params)
% Generates coefs for a continuous wavelet transform
% t, x are time and data points for time series data
% mother_wavelet is a function, taking parameters (t, params),
%      where the value of params depends on the specific function used
% max_wavelet is the maximum range of the wavelet function (beyond which
%      the wavelet is essentially zero)
% scales is a vector of desired scales
% params is the parameter for the mother wavelet function

max_t = max(t);
dt = t(2)-t(1);
full_t = -(max_t/2):dt:(max_t/2);
coefs = zeros(length(scales), length(x));
points = length(x);
t_scale = linspace(-max_wavelet, max_wavelet, points);
dt = (max_wavelet*2)/(points-1);

mom_wavelet = feval(mother_wavelet, t_scale, params);
row = 1;

for scale = scales
      time_scale = [1+floor([0:scale*max_wavelet*2]/(scale*dt))];
      wavelet = mom_wavelet(time_scale);
      w = conv(x,wavelet)/sqrt(scale);
      mid_w = floor(length(w)/2);
      mid_x = floor(length(x)/2);
      w = w(((-mid_x:mid_x)+mid_w));
      scale % print scale to show progress
      coefs(row,:) = abs(w);
      row = row + 1;
end
```

In **my_morlet.m:**

```
function m=morlet(t, params)
sigma = params(1);
m = pi^-0.25*exp(-i*sigma.*t-0.5*t.^2);
```

In **plot_cwt.m:**

```
function plot_cwt(t, coefs, scales)
     imagesc(t, scales, coefs);
     colormap(hot);
     axis xy;
end
```

Here, **imagesc** generates an imagemap from two vectors of data. Given parameters *x*, *y*, and *c*, **imagesc** generates a colored area of *color(n,m)* centered at *x(n)* and *y(m)*. So, here in **plot_cwt**, at values of *t* and *coefs*, the corresponding scales value is used to assign a color.

To generate the scalogram, type:

```
scales = 1:200;
coefs = my_cwt(t, x, @my_morlet, 10, scales, [10]);
plot_cwt(t, coefs, scales);
```

9.2.5. The Discrete Wavelet Transform

In addition to the continuous wavelet transform, there is a transformation termed the *discrete wavelet transform* (DWT). However, the DWT is not merely a discretized continuous wavelet transform. Instead, the discrete wavelet transform calculates only a subset of the possible scales, usually dyadic values (successive values in 2^n, i.e., 1, 2, 4, 8, 16, 32, etc.). Moreover, the DWT is usually calculated using an algorithm called the *pyramid algorithm*, in which the data series is recursively split in two and reprocessed.

An exploration of the pyramid algorithm is beyond the scope of this chapter. For a thorough discussion, see Percival and Walden (2000). The DWT has been used to denoise signals and to cluster neural spikes for sorting (Quiroga, Nadasdy, and Ben-Shaul, 2004).

9.2.6. Wavelet Toolbox

Wavelet Toolbox provides an implementation of the DWT and a number of appropriate wavelets. Analyses using the discrete wavelet transform use different wavelets than analyses with the continuous transform. The Haar wavelet and the Daubechies wavelet are among the most widely used.

In the following commands, *'wname'* corresponds to the name of a specific wavelet included in Wavelet Toolbox. Possible choices are *'dbN'* for Daubechies N, *'haar'* for Haar, *'morl'* for Morlet, and *'mexh'* for the Mexican hat. To view all supported wavelets, use **help waveform.**

```
coefs = cwt(S, SCALES, 'wname')
```

The function **cwt** performs a continuous wavelet transform on the dataset *S*. The scales given as *SCALES* are used, and the wavelet is given by *'wname'*. The function **cwt** will also automatically plot the scalogram if given the parameter *'plot'* at the end:

```
coefs = cwt(x, 1:200, 'morl', 'plot')
```

```
[cA. cD] = dwt(X, 'wname')
X = idwt(cA, cD, 'wname')
```

The functions **dwt** and **idwt** perform a single level decomposition and synthesis given the wavelet name.

```
[C, L] = wavedec(X, N, 'wname')
X = waverec(C, L, 'wname')
```

The functions **wavedec** and **waverec** perform multilevel decomposition and synthesis given wavelet name and level *N*. Note that *N* cannot be greater than the exponent of the largest power of 2 less than the size of *X*. The *C* vector contains the transform, and the *L* vector contains bookkeeping information used by **wavedec** and **waverec** to find the position of the parts of the transform in *C*.

Here is an example plotting scales 2 through 7 for a Debauches 4 wavelet:

```
% here size(s) = 128
[C, L] = wavedec(s, 7, 'db4');
for scale = 2:7
        subplot(7,1,scale)
        c_sub = (2^(scale-1)):(2^scale);
        t_sub = linspace(1, time, time/size(c_sub));
        plot(t_sub, C(c_sub))
end
```

wavedemo

The **wavedemo** function opens an automated tour of Wavelet Toolbox, showing various transforms and functions provided by the toolbox.

9.3. EXERCISES

Exercise 9.1: Which of the following MATLAB functions can be wavelet functions? Why or why not?

```
function x = f_one(t)
        x = cos(t);
end

function x = f_two(t)
        if (x < 0 or x > pi/2)
                x = 0;
        else
                x = cos(t);
        end
end

function x = f_three(t)
        x = sqrt(2) * t * exp(-t^2/2) / pi^4;
end

function x = f_four(t)
        x = sqrt(2) * t^2 * exp(-t^2/2) / pi^4;
end

function x = f_five(t)
        x = (x > -1 && x < 0) * -1 + (x > 0 && x < 1);
end
```

Exercise 9.2: Generate the scalogram in Figure 9.2. Generate a spectrogram and compare. How clearly does each render the transients? The primary frequencies?

Exercise 9.3: Write a Mexican hat mother wavelet function compatible with the previous continuous wavelet transform code. Generate a scalogram of the sinusoid+transient signal used in Figure 9.2. Compare Mexican hat transform to the Morlet transform.

Exercise 9.4: Download the EEG signal wavelet, **eeg,** from the companion website. Generate scalograms using the Mexican hat and Morlet wavelet transforms. Compare to a spectrogram generated with **spectrogram().**

9.4. PROJECT

In Chapter 8, "Frequency Analysis Part II: Nonstationary Signals and Spectrograms," you used the short-time Fourier transform to look for sleep state transitions. Here, you will be asked to examine the same data files using the continuous wavelet transform and Morlet and Mexican hat wavelets. Compare and contrast your findings with what you found using only the STFT.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

cwt
dwt
idwt
wavedec
waverec

CHAPTER

10

Convolution

10.1. GOALS OF THIS CHAPTER

The purpose of this chapter is to familiarize you with the convolution operation. You will use this operation in the context of receptive fields in the early visual system as input response filters whose convolution with an input image approximates certain aspects of your perception. Specifically, you will reproduce the Mach band illusion and explore the Gabor filter as a model for the receptive field of a simple cell in the primary visual cortex.

10.2. BACKGROUND

A *convolution* is the mathematical operation used to find the output *y(t)* of a linear time-invariant system from some input *x(t)* using the impulse response function of the system *h(t)*, where *h(t)* is defined as the output of a system to a unit impulse input. It is defined as the following integral:

$$y(t) = h(t) * x(t) = \int_{-\infty}^{\infty} h(\tau)x(t-\tau)d\tau \tag{10.1}$$

This can be graphically interpreted as follows. The function *h(τ)* is plotted on the τ-axis, as is the flipped and shifted function *x(t-τ)*, where the shift *t* is fixed. These two signals are multiplied, and the signed area under the curve of the resulting function is found to obtain *y(t)*. This operation is then repeated for every value of *t* in the domain of *y*. It turns out that it doesn't matter which function is flipped and shifted since $h * x = x * h$.

You can also define a convolution for data in two dimensions:

$$y(k,t) = h(k,t) * x(k,t) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(\tau, K)x(k-K, t-\tau)dKd\tau \tag{10.2}$$

Basically, you take a convolution in one dimension to establish the k dependence of the result y and then use that output (which is a function of k, t and τ) to perform another convolution in the second dimension. This second convolution provides the t dependence of the result, y. It is important that you understand how to apply this to a two-dimensional data function because in this chapter you will be working with two-dimensional images. In the MATLAB® software, since you are working with discrete datasets, the integral becomes a summation, so the definition for convolution in 2D at every point becomes

$$y(n_1, n_2) = \sum_{k_1=-\infty}^{\infty} \sum_{k_2=-\infty}^{\infty} h(k_1, k_2) x(n_1 - k_1, n_2 - k_2) \tag{10.3}$$

Again, this is easier to understand pictorially. What you are doing in this algorithm is taking the dataset x, which is a matrix; rotating it by 180 degrees; overlaying it at each point in the matrix h that describes the response filter; multiplying each point with the underlying point; and summing these points to produce a new point at that position. You do this for every position to get a new matrix that will represent the convolution of h and x.

10.2.1. The Visual System and Receptive Fields

In this section we discuss in general the anatomy of the visual system and the input response functions that explain how different areas of the brain involved in this system might "perceive" a visual stimulus.

Light information from the outside world is carried by photons that enter the eyes and cause a series of biochemical cascades to occur in rods and cones of the retina. This biochemical cascade causes channels to close which leads to a decrease in the release of neurotransmitter onto bipolar cells. In general, there are two fundamental varieties of bipolar cells. On-bipolar cells become depolarized in response to light and off-bipolar cells become hyperpolarized in response to light. The bipolar cells then project to the ganglion cells which are the output cells of the retina. The response to light in this main pathway is also influenced by both the horizontal and amacrine cells in the retina. There are many types of retinal ganglion cells that respond to different visual stimuli.

A stimulus in the visual field will elicit a cell's response (above the background firing rate) only if it lies within a localized region of visual space, denoted by the cell's *classical receptive field*. In general, the ganglion cells have a center-surround receptive field due to the types of cells that interact to send information to these neurons. That is, the receptive field is essentially two concentric circles, with the center having an excitatory increase (+) in neuronal activity in response to light stimulus and the surround having an inhibitory decrease (−) in neuronal activity in response to light stimulus, or vice versa. The response function of the ganglion cells can then be modeled using a *Mexican hat* function, also sometimes called a *difference of Gaussians* function.

In the main visual pathway, the ganglion cells send their axons to the lateral geniculate nucleus (LGN) in the thalamus, which is in charge of regulating information flow to the cortex. These cells also are thought to have receptive fields with a center-surround architecture. LGN cells project to the primary visual cortex (V1). In V1, simple cells are thought to receive information from LGN neurons in such a way that they respond to bars of light at certain orientations and spatial frequencies. This can similarly be described as a Gabor function—a two-dimensional Gaussian filter whose amplitude is modulated by a sinusoidal function along an axis at a given orientation. Thus, different simple cells in V1 respond to bars of light at specific orientations with specific widths (this represents spatial frequency; see Dayan and Abbott, 2001). These and other cells from V1 project to many other areas in the cortex thought to represent motion, depth, face recognition, and other fascinating visual features and perceptions.

10.2.2. The Mach Band Illusion

Using your knowledge of the receptive fields or the response functions of the visual areas can help you understand why certain optical illusions work. The *Mach band illusion* is a perceptual illusion seen when viewing an image that ramps from black to white. Dark and light bands appear on the image where the brightness ramp meets the black and white plateau, respectively. These bands are named after Ernst Mach, a German physicist who first studied them in the 1860s. They can be explained with the center-surround receptive fields of the ganglion or LGN cells (Ratliff, 1965; Sekuler and Blake, 2002); we will use this model in this chapter although alternative explanations exist (for example, see Lotto, Williams, and Purves, 1999.)

The illusion is demonstrated in Figure 10.1. At the initiation of the stimulus brightness ramp, a dark band, darker than the dark plateau to the left, is usually perceived. At the termination of the brightness ramp, a light band is perceived brighter than the light plateau

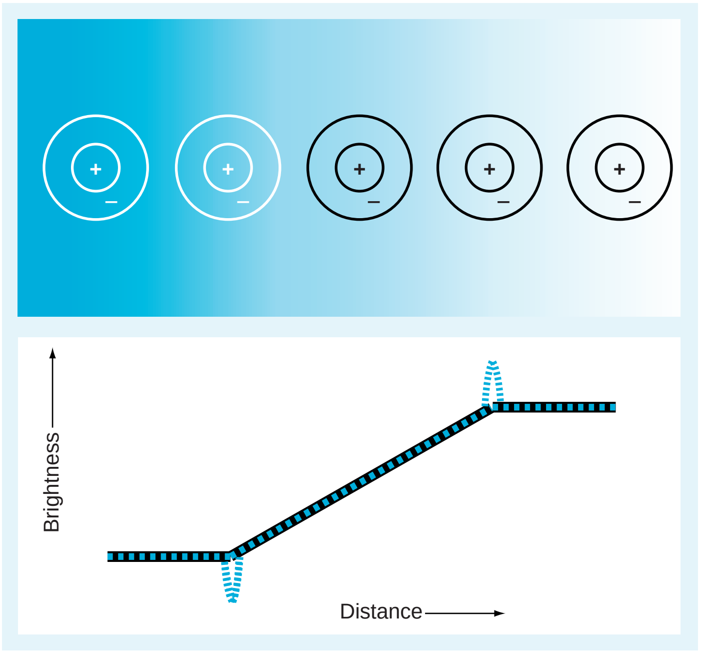


FIGURE 10.1 The Mach band illusion. Top of figure: the visual stimulus with various center-surround receptive fields superimposed. Bottom of figure: the actual brightness of the visual stimulus (black solid line) and the perceived brightness of the optical illusion (blue dotted line).

to its right. Figure 10.1 shows the center-surround receptive fields of sample neurons, represented by concentric circles, superimposed on the stimulus image. The center disk is *excitatory*, and the surrounding annulus is *inhibitory*, as indicated by the plus and minus signs. When the receptive field of a neuron is positioned completely within the areas of uniform brightness, the center receives nearly the same stimulation as the surround; thus, the excitation and inhibition are in balance. A receptive field aligned with the dark Mach band has more of its surround in a brighter area than the center, and the increased inhibition to the neuron results in the perception of that area as darker. Conversely, the excitation to a neuron whose receptive field is aligned with the bright Mach band is increased, since more of its center is in a brighter area than the surround. The decreased inhibition to such a neuron results in a stronger response than that of the neuron whose receptive field lies in the uniformly bright regime and thus the perception of the area as brighter.

10.3. EXERCISES

The goal for this chapter is to reproduce the Mach band optical illusion. First, you will create the visual stimulus. Then you will create a center-surround Mexican hat receptive field. Finally, you will convolve the stimulus with the receptive field filter to produce an approximation of the perceived brightness.

You begin by creating the M-file named ramp.m that will generate the visual input (see Figure 10.2). The input will be a 64×128 matrix whose values represent the intensity or brightness of the image. You want the brightness to begin dark, at a value of 10, for the first 32 columns. In the next 65 columns, the value will increase at a rate of one per column, and

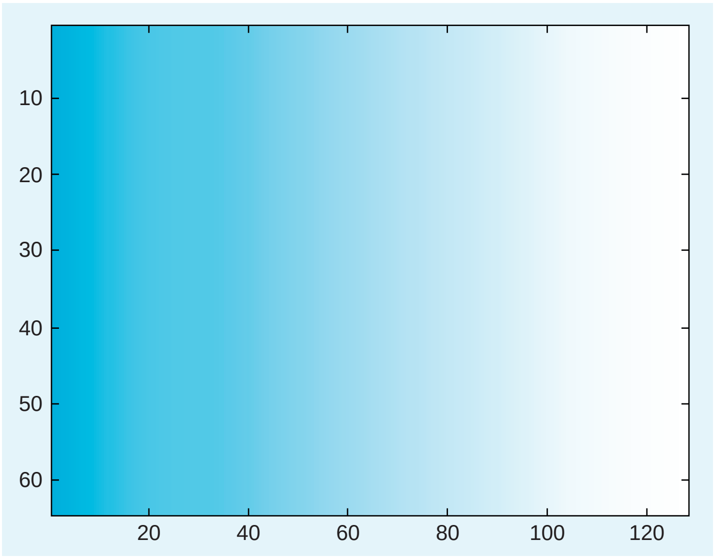


FIGURE 10.2 The brightness ramp stimulus used as visual input.

the brightness will stay at the constant value of 75 for the rest of the matrix. Open a new blank file and save it under the name ramp.m. In that file enter the following commands:

```
%ramp.m
% This script generates the image that creates the Mach band visual illusion.
In = 10*ones(64,128); %initiates the visual stimulus with a constant value of 10
for i = 1:65
  In(:,32 + i) = 10 + i;
  %ramps up the value for the middle matrix elements (column 33 to column 97)
end
In(:,98:end) = 75; %sets the last columns of the matrix to the final brightness value of 75
figure
imagesc(In); colormap(bone); set(gca, 'fontsize',20) %view the visual stimulus
```

Notice how the function **imagesc** creates an image whose pixel colors correspond to the values of the input matrix *In.* You can play with the color representation of the input data by changing the colormap. Here, you use the colormap *bone,* since it is the most appropriate one for creating the optical illusion, but there are many more interesting options available that you can explore by reading the help file for the function **colormap**.

You've just created an M-file titled ramp that will generate the visual stimulus. Note, however, that you use a **for** loop in ramping up the brightness values. Although it doesn't make much of a difference in this script, it is good practice to avoid using **for** loops when programming in MATLAB if possible, and to take advantage of its efficient matrix manipulation capabilities for faster run times (see Appendix A, "Thinking in MATLAB"). How might you eliminate the **for** loop in this case? One solution is to use the function **cumsum**. Let's see what it can do:

```
>> z = ones(3,4)
z =

  1  1  1  1
  1  1  1  1
  1  1  1  1

>> cumsum(z)
ans =

  1  1  1  1
  2  2  2  2
  3  3  3  3
```

The function will cumulatively add the elements of the matrix by row, unless you specify that dimension along which to sum should be the second dimension, or by column:

```
>> cumsum(z,2)
ans =

  1  2  3  4
  1  2  3  4
  1  2  3  4
```

You will want this cumulative sum by columns for this ramp function. Now rewrite the code in proper style for MATLAB without the **for** loop:

```
%ramp.m
% This script generates the image that creates the Mach band visual illusion.
In = 10*ones(64,128); %initiates the visual stimulus with a constant value of 10
% now ramp up the value for the middle matrix elements using cumsum
In(:,33:97) = 10 + cumsum(ones(64,65),2);
In(:,98:end) = 75; %sets the last columns of the matrix to the end value of 75
figure; imagesc(In); colormap(bone); set(gca, 'fontsize',20) %view the visual stimulus
```

You can look at how the values of the brightness increase from left to right by taking a slice of the matrix and plotting it, as shown in Figure 10.3. Look at the 32nd row in particular.

```
>> plot(In(32,:),'k','LineWidth',3); axis([0 128 0 85]); set(gca,'fontsize',20)
```

Next, you will create a script titled mexican_hat.m that will generate a matrix whose values are a difference of Gaussians. For this exercise, you will make this a 5×5 filter, as shown in Figure 10.4.

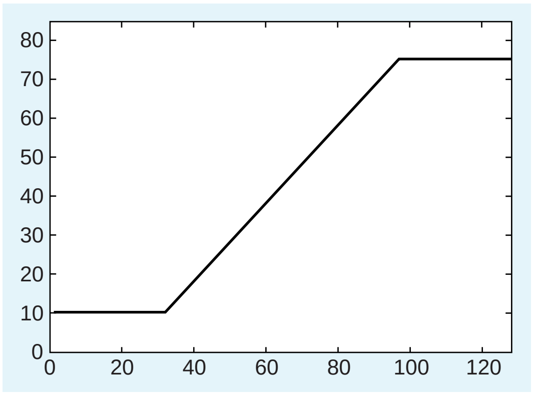


FIGURE 10.3 The brightness values in a slice through the ramp stimulus shown in Figure 10.2.

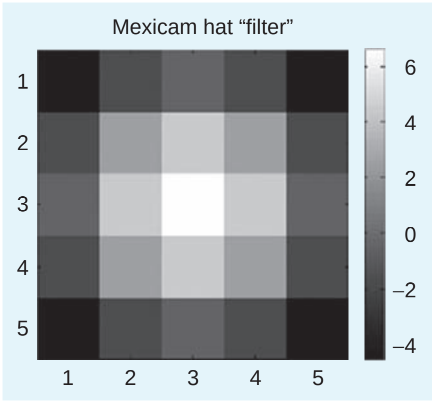


FIGURE 10.4 A 5×5 Mexican hat spatial filter.

```
% mexican_hat.m
% this script produces an N by N matrix whose values are
% a 2 dimensional mexican hat or difference of Gaussians
%
N = 5; %matrix size is NXN
IE=6; %ratio of inhibition to excitation
Se=2; %variance of the excitation Gaussian
Si=6; %variance of the inhibition Gaussian
S = 500; %overall strength of mexican hat connectivity
%
[X,Y]=meshgrid((1:N)-round(N/2));
% -floor(N/2) to floor(N/2) in the row or column positions (for N odd)
% -N/2+1 to N/2 in the row or column positions (for N even)
%
[THETA,R] = cart2pol(X,Y);
% Switch from Cartesian to polar coordinates
% R is an N*N grid of lattice distances from the center pixel
% i.e. R=sqrt((X).^2 + (Y).^2)+eps;
EGauss = 1/(2*pi*Se^2)*exp(-R.^2/(2*Se^2)); % create the excitatory Gaussian
IGauss = 1/(2*pi*Si^2)*exp(-R.^2/(2*Si^2)); % create the inhibitory Gaussian
%
MH = S*(EGauss-IE*IGauss); %create the Mexican hat filter

figure; imagesc(MH) %visualize the filter
title('mexican hat "filter"','fontsize',22)
colormap(bone); colorbar
axis square; set(gca,'fontsize',20)
```

Now take a second look at some of the components of this script. The function **meshgrid** is used to generate the X and Y matrices whose values contained the x and y Cartesian coordinate values for the Gaussians:

```
>> X

X =

  -2  -1   0   1   2
  -2  -1   0   1   2
  -2  -1   0   1   2
  -2  -1   0   1   2
  -2  -1   0   1   2

>> Y

Y =

  -2  -2  -2  -2  -2
  -1  -1  -1  -1  -1
   0   0   0   0   0
   1   1   1   1   1
   2   2   2   2   2
```

The function **cart2pol** converts the Cartesian coordinates X and Y into the polar coordinates R and $THETA$. You use this function to create the 5×5 matrix R whose values are the radial distance from the center pixel:

```
>> R

R =

  2.8284  2.2361  2.0000  2.2361  2.8284
  2.2361  1.4142  1.0000  1.4142  2.2361
  2.0000  1.0000       0  1.0000  2.0000
  2.2361  1.4142  1.0000  1.4142  2.2361
  2.8284  2.2361  2.0000  2.2361  2.8284
```

The $THETA$ variable is never used; however, it gives the polar angle in radians:

```
>> THETA

THETA =

  -2.3562  -2.0344  -1.5708  -1.1071  -0.7854
  -2.6779  -2.3562  -1.5708  -0.7854  -0.4636
   3.1416   3.1416        0        0        0
   2.6779   2.3562   1.5708   0.7854   0.4636
   2.3562   2.0344   1.5708   1.1071   0.7854
```

Finally, you're ready to generate the main script called mach_illusion.m to visualize how the Mexican hat function/center-surround receptive field of the neurons in the early visual system could affect your perception. In this simple model, the two-dimensional convolution of the input image matrix (generated by the ramp.m M-file) with the receptive field filter (generated by the mexican_hat.m M-file) gives an approximation to how the brightness of the image is perceived when filtered through the early visual system. This operation should result in a dip in the brightness perceived at the point where the brightness of the input just begins to increase and a peak in the brightness perceived at the point where the brightness of the input just stops increasing and returns to a steady value, consistent with the perception of Mach bands (see Figure 10.5). For a first pass, use the two-dimensional convolution function, **conv2**, that is built into MATLAB. As described in detail in the help section, this function will output a matrix whose size in each dimension is equal to the sum of the corresponding dimensions of the input matrices minus one. The edges of the output matrix are usually not considered valid because the value of those points have some terms contributing to the convolution sum which involved zeros padded to the edges of the input matrix. One way to deal with the problem of such edge effects is to reduce the size of the output image by trimming the invalid pixels off the border. You accomplish this by including the option **'valid'** when calling the **conv2** function:

```
%mach_illusion.m
clear all; close all
mexican_hat    %creates the mexican hat matrix, MH, & plots
```

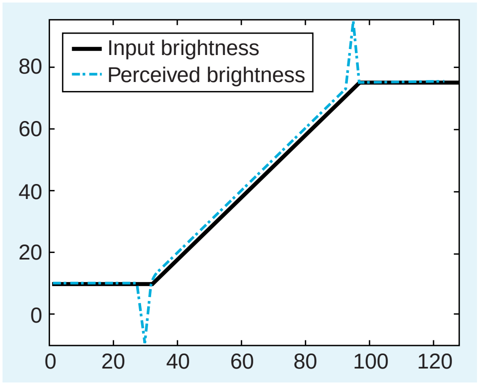


FIGURE 10.5 The Mach band illusion generated using the Mexican hat filter on the ramp input.

```
ramp            %creates image with ramp from dark to light, In, & plots
A=conv2(In,MH,'valid'); %convolve image and mexican hat
figure; imagesc(A); colormap(bone) %visualize the "perceived" brightness
%create plot showing the profile of both the input and the perceived brightness
figure; plot(In(32,:),'k','LineWidth',5); axis([0 128 -10 95])
hold on; plot(A(32,:),'b-.','LineWidth',2); set(gca,'fontsize',20)
lh=legend('input brightness','perceived brightness',2); set(lh,'fontsize',20)
```

Make sure that the mexican_hat.m and ramp.m M-files are in the same directory as the mach_illusion.m M-file. Note that the size of the output is indeed smaller than the input:

```
>> size(A)

ans =

  60  124
```

For fun, you can learn more about how the convolution works by changing the **'valid'** option in the **conv2** function call to either **'full'** or **'same'** and see how the output matrix *A* changes. One way to minimize the edge effects of convolution is to pad the input matrix with values that mirror the edges of the input matrix before performing the two-dimensional convolution and returning only the valid part of the output, which will now be the size of the original input matrix. The function **conv2mirrored.m** will do just this trick. It has been written in a generic form to accept matrices of any size:

```
%conv2_mirrored.m
function sp = conv2_mirrored(s,c)
% 2D convolution with mirrored edges to reduce edge effects
% output of convolution is same size as leading input matrix
[N,M]=size(s);
[n,m]=size(c); %% both n & m should be odd
%
% enlarge matrix s in preparation for convolution with matrix c
```

```
%via mirroring edges to reduce edge effects.
padn = round(n/2) - 1;
padm = round(m/2) - 1;
sp = [zeros(padn,M + (2*padm)); zeros(N,padm) s zeros(N,padm); zeros(padn,M + (2*padm))];
sp(1:padn,:) = flipud(sp(padn + 1:2*padn,:));
sp(padn + N + 1:N + 2*padn,:) = flipud(sp(N + 1:N + padn,:));
sp(:,1:padm) = fliplr(sp(:,padm + 1:2*padm));
sp(:,padm + M + 1:M + 2*padm) = fliplr(sp(:,M + 1:M + padm));
%
% perform 2D convolution
sp = conv2(sp,c,'valid');
```

Exercise 10.1: Put the figures generated by the mach_illusion.m script into a document, explain each figure, and give a short summary of the Mach band illusion as you understand it.

Exercise 10.2: Rather than **cumsum**, you could have also used the function **meshgrid** to efficiently ramp up the brightness values from dark to light when creating the matrix *In.* Read the help file for **meshgrid** and rewrite the ramp.m script using **meshgrid** rather than **cumsum.**

Exercise 10.3: Create the function **conv2_mirrored.m** using the code provided previously and place it in the same directory as your other files. Learn how the mirroring of the edges of the input matrix is accomplished by reviewing the help files on the functions **flipud** and **fliplr**. What determines the size of the mirrored-edge padding necessary and why? Rewrite your main script mach_illusion.m to use this convolution function rather than the **conv2** function. Check that your output matrix *A* is now the same size as the input matrix *In.*

Exercise 10.4: Change the slope of the ramp without changing the beginning or ending values of the input image. [*Hint:* The command **linspace** can be useful to find values of the ramp that will go from 10 to 75 in, say, 30 steps rather than 65: **linspace(10,75,30)**.] How does increasing or decreasing the slope affect the strength of the illusion?

Exercise 10.5: Convert the M-file named mexican_hat.m into a function where the inputs are the size of the matrix, the ratio of excitation to inhibition, the variance of excitatory and inhibitory Gaussians, and the overall strength of the filter. Also make the appropriate changes to the main script that calls this function, mach_illusion.m.

10.4. PROJECT

The receptive fields of simple cells in V1 reflect the orientation and spatial frequency preference of the neurons. One way to model this is to use the Gabor function, which is basically a two-dimensional Gaussian modulated by a sinusoid, as shown in Figure 10.6.

1. Observe how the receptive fields of simple cells in V1 modeled as Gabor functions with various spatial frequency and orientation preferences filter an image of a rose, which can be downloaded from the companion website. Create two files, the gabor_filter.m function and the gabor_conv.m script (using the following code), in the same directory as the conv2_mirrored.m file. Also, place the rose.jpg image file in the same directory. Now, run the gabor_conv.m script. It will take a convolution between the rose image with a Gabor function of a given orientation (OR) and spatial frequency (SF). The input parameters OR and SF will determine the orientation and spatial frequency of the filter. Thus, you will essentially "see" how simple cells in V1 with a given orientation and spatial frequency preference perceive an image. Try values of SF = 0.01, 0.05, and 0.1, and OR = 0, pi/4, and pi/2. Put the resulting figures into a document and explain the results. Try changing the Gabor filter from an odd filter to an even filter by using **cos** instead of **sin**. How does this affect the output?

```
% gabor_filter.m
function f = gabor_filter(OR, SF)
% Creates a Gabor filter for orientation and spatial frequency
% selectivity of orientation OR (in radians) and spatial frequency SF.
%
% set parameters
sigma_x=7;% standard deviation of 2D Gaussian along x-dir
sigma_y=17;% standard deviation of 2D Gaussian along y-dir
%
% create filter
```

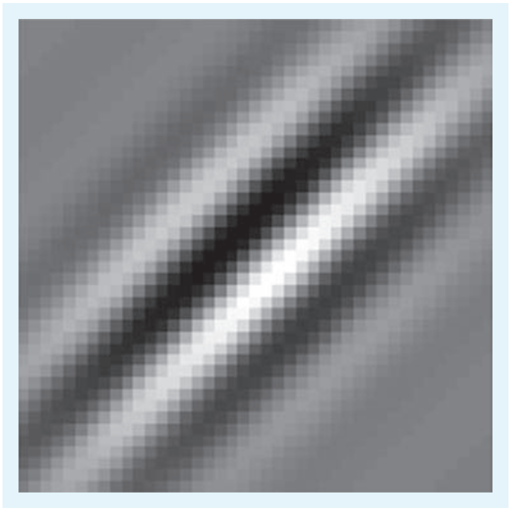


FIGURE 10.6 A Gabor function modeling the oriented receptive field of a V1 neuron.

```
[x,y] = meshgrid(-20:20);
X = x*cos(OR) + y*sin(OR); %rotate axes
Y = -x*sin(OR) + y*cos(OR);
f = (1/(2*pi*sigma_x*sigma_y)).*exp(-(1/2)*(((X/sigma_x).^2) + ...
    ((Y/sigma_y).^2))).*sin(2*pi*SF*X);
```

```
%gabor_conv.m
clear all; close all
I = imread('rose.jpg');
OR = 0; SF = .01;
G = gabor_filter(OR,SF);
figure
subplot(1,3,1); imagesc(G); axis square; colorbar; title ('Gabor function')
subplot(1,3,2); imagesc(I); title('original image')
subplot(1,3,3); imagesc(conv2_mirrored(double(I),G));
colormap(bone); title(['Convolved image OR = ',num2str(OR),' SF = ', num2str(SF)])
```

2. Now you can have some fun times with image processing and convolutions. Choose any image and convolve it with a function or filter of your choosing. To avoid edge problems, you can use the conv2_mirrored function provided, the **conv2** function with the **'valid'** option (as in the exercises), or you can use the function **imfilter** from the Image Processing Toolbox built into MATLAB, which does an operation similar to convolution. You can create your own filter or choose a predesigned filter in MATLAB using the **fspecial** function, also from Image Processing Toolbox. You can learn more about these functions through the online help. Hand in your code and picture of before and after the filtering, along with an image of the filter used in the convolution.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

imagesc
colormap
cumsum
meshgrid
cart2pol
conv2
flipud
fliplr

CHAPTER

11

Introduction to Phase Plane Analysis

11.1. GOAL OF THIS CHAPTER

The goal of this chapter is to examine the cone and horizontal cell system using a qualitative visualization technique called *phase plane analysis*; this system will be discussed further in Chapter 21, "Models of the Retina." The techniques presented here will be used again in Chapter 12, "Exploring the Fitzhugh-Nagumo Model."

11.2. BACKGROUND

In this chapter you will be studying a retinal feedback model; this model is described further in Chapter 21, "Models of the Retina." The system is represented as follows:

$$\frac{d\tilde{C}}{dt} = \frac{1}{\tau_C}(-\tilde{C} - k\tilde{H}) \tag{11.1}$$

$$\frac{d\tilde{H}}{dt} = \frac{1}{\tau_H}(-\tilde{H} + \tilde{C}) \tag{11.2}$$

Typical values for these parameters are $\tau_C = 0.025\,\text{sec}$, $\tau_H = 0.08\,\text{sec}$, and $k = 4$. Now assume that the light intensity is $L = 10$ (i.e., daylight). For your initial conditions, choose that $C(0) = H(0) = 0$. Finally, be aware that:

$$\tilde{C} = C - \frac{L}{k+1} \quad \text{and} \quad \tilde{H} = H - \frac{L}{k+1} \tag{11.3}$$

For further details of the basic biology of this system, see Chapter 21, "Models of the Retina." In that chapter, we will examine in more detail the model of retinal feedback between cone cells and horizontal cells of the retina, shown in Equations 11.1 and 11.2. Although the explicit solutions determined in that chapter are more informative, many more complicated systems (such as the Fitzhugh-Nagumo system presented in Chapter 12, "Exploring the Fitzhugh-Nagumo Model") can only be qualitatively described. When we describe a system qualitatively, we look for steady-state values of the solutions (often called *fixed points*) and try to classify the dynamics of the solution that led to these steady-state values. In Chapter 21, "Models of the Retina," we will consider the following system in more detail:

$$\frac{dx}{dt} = x + y \tag{11.4}$$

$$\frac{dy}{dt} = 4x + y \tag{11.5}$$

which has eigenvalues -1 and 3 and has the solution:

$$x(t) = C_1 e^{3t} + C_2 e^{-t} \tag{11.6}$$

$$y(t) = 2C_1 e^{3t} - 2C_2 e^{-t} \tag{11.7}$$

This solution can be described qualitatively. If you wait long enough, then this system will approach one of two states. If $C_1 = 0$, then:

$$\lim_{t\to\infty} x(t) = \lim_{t\to\infty} y(t) = 0 \tag{11.8}$$

Therefore, one says that $(x, y) = (0, 0)$ is a steady-state or fixed point of the system. For $C_1 \neq 0$, then:

$$\lim_{t\to\infty} x(t) = \lim_{t\to\infty} y(t) = \infty \tag{11.9}$$

Therefore, the only finite steady-state solution to this system is $(x, y) = (0, 0)$. Regardless of how you choose C_1 and C_2, there are no other steady-state values for this system. Since the initial conditions determine C_1 and C_2, then those initial conditions that lead to $C_1 = 0$ will have solutions that steadily tend toward the fixed point (0, 0), while all others will steadily tend toward infinity (i.e., away from the fixed point at the origin). A fixed point with this property—that is, with some initial conditions leading to the fixed point and others leading away from it—is called a *saddle point*. This simple qualitative description of identifying the steady state(s) of the solution, the dynamics of what initial conditions lead to the steady state(s), and how it is reached steadily or in an oscillatory fashion can all be determined from a phase plane analysis of the system.

The first step in phase plane analysis is to set up a phase plane. The axes for the plane represent the state variables characterizing the system. In the preceding example, the phase plane is constructed with y as the ordinate and x as the abscissa. Next, the -x- and y-nullclines are plotted. The x-nullcline is the curve in the x-y plane, where:

$$\frac{dx}{dt} = 0$$

A similar definition applies for the y-nullcline. Intersections of these nullclines represent points where:

$$\frac{dx}{dt} = \frac{dy}{dt} = 0$$

so x and y are no longer changing with time. In other words, these intersections represent steady-state values or fixed points of the system. Next, a vector field is constructed by assigning the following vector to every point on the x-y plane:

$$\begin{bmatrix} \frac{dx}{dt} & \frac{dy}{dt} \end{bmatrix}^T$$

Notice that this vector field can be determined without knowing the solution to the system. Since the slope of these vectors is:

$$m = \frac{dy}{dt} \Big/ \frac{dx}{dt} = \frac{dy}{dx} \tag{11.10}$$

by the chain rule, the vector field must be tangent to any solution (x, y) of the system. This allows you to use the vector field to calculate the solution of the system for any initial condition (x_o, y_o). Such a solution when plotted on the phase plane is called a *trajectory*. The phase plane, nullclines, vector field, and several trajectories are shown in Figure 11.1 for the system in Equations 11.4 and 11.5.

In the figure, the nullclines are plotted as dashed lines. Notice that these nullclines intersect at the point $(x, y) = (0, 0)$ indicating that this is the steady-state of the system in agreement with what was predicted by considering the explicit solutions (Equations 11.6 and 11.7). Any linear system of ordinary differential equations described by a matrix with real eigenvalues of opposite sign (recall that the eigenvalues for this system are –1 and 3) will have a saddle point at the intersection of its nullclines.

If the matrix describing the linear system has real eigenvalues that are both negative, then the fixed point is called a *nodal sink*. The classic phase portrait of a nodal sink is shown in Figure 11.2.

If the matrix describing the linear system has real eigenvalues that are both positive, then the fixed point is called a *nodal source*. The classic phase portrait of a nodal source is shown in Figure 11.3. Notice the difference in the direction of the arrows in the vector field in this figure.

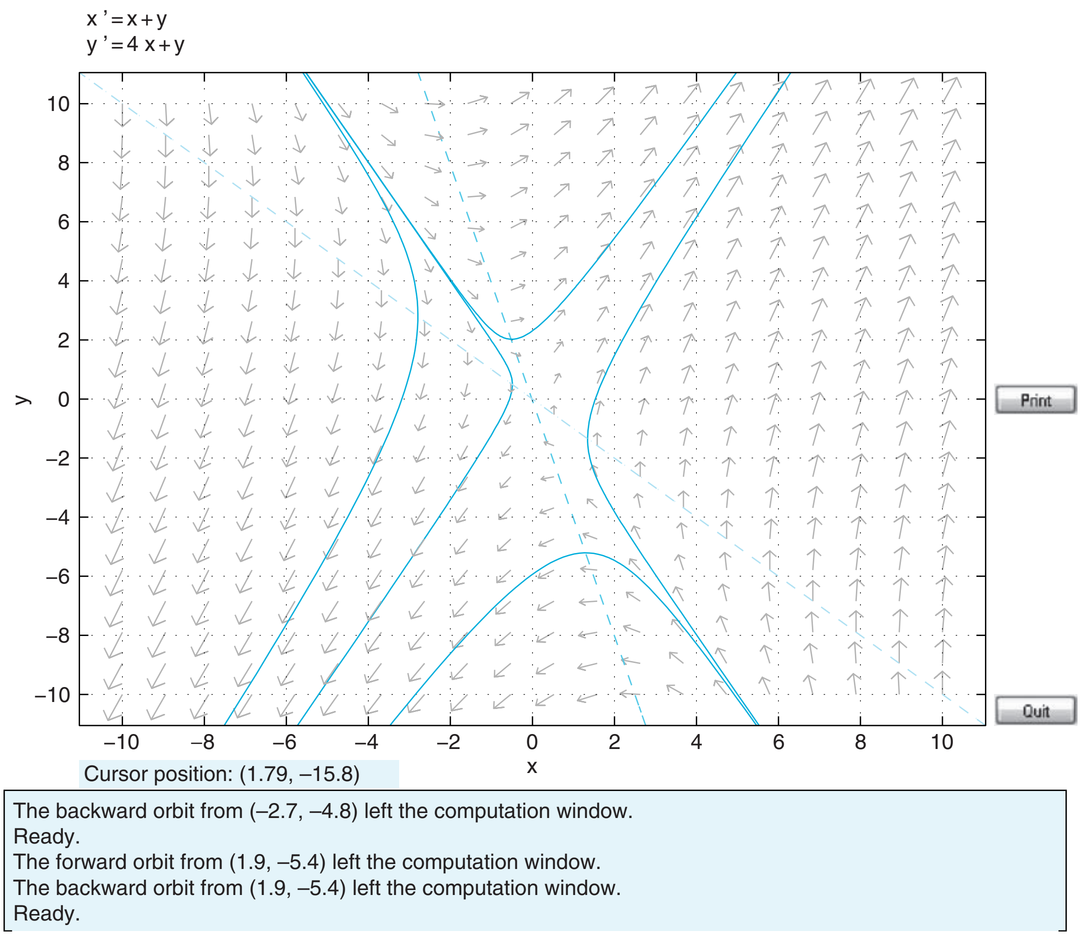
If the matrix describing the linear system has imaginary eigenvalues that have negative real parts, then the fixed point is called a *spiral sink*. The classic phase portrait of a spiral sink is shown in Figure 11.4.

If the matrix describing the linear system has imaginary eigenvalues that have positive real parts, then the fixed point is called a *spiral source*. The classic phase portrait of a spiral source is shown in Figure 11.5.

These five types of equilibria are collectively known as the *generic equilibria*. There are also five nongeneric equilibria. The most important nongeneric equilibrium is called a *center*. It occurs when the eigenvalues of the matrix are purely imaginary. The classic phase portrait of a center is shown in Figure 11.6.

11.3. EXERCISES

The phase portraits in the preceding section were drawn using a downloadable M-file called pplane7.m. The phase plane consists of three basic features: the nullclines intersecting at the fixed point of the system, the vector field showing how the solutions change over



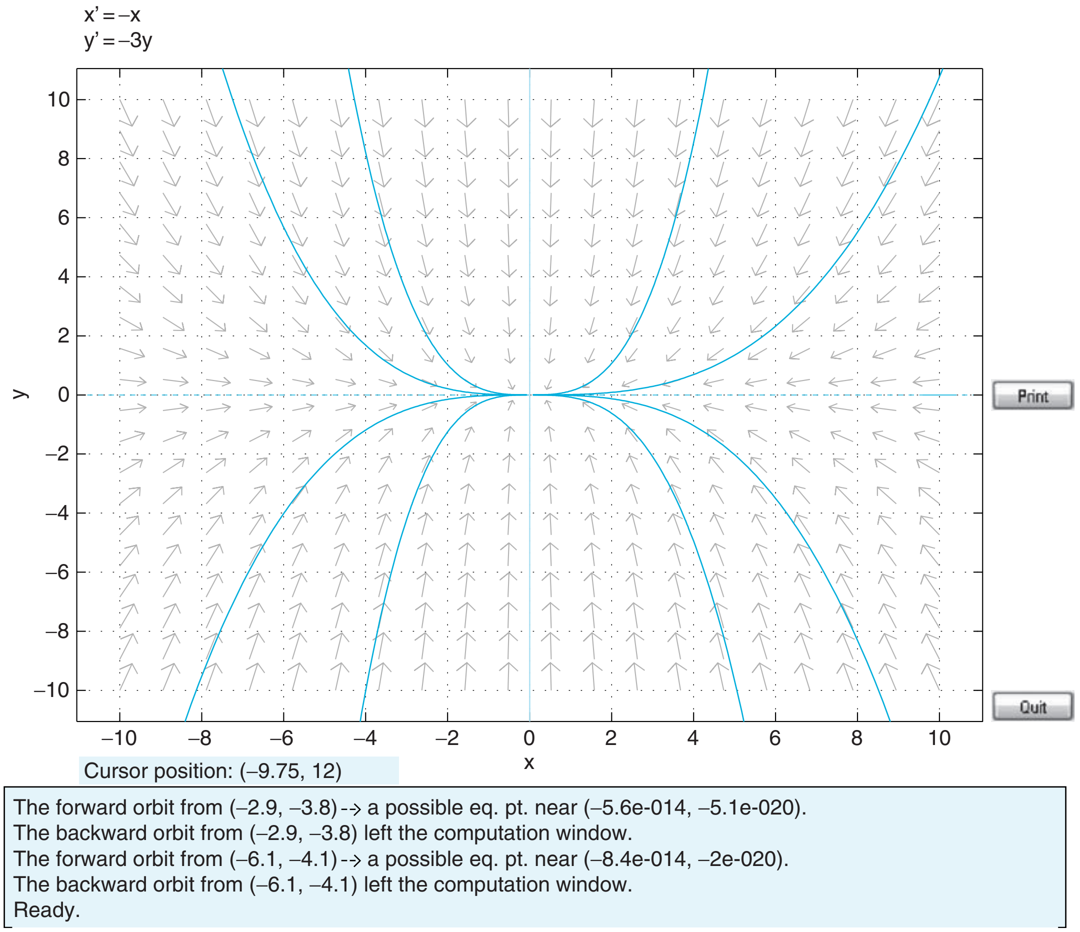
Cursor position: (1.79, −15.8)

```
The backward orbit from (−2.7, −4.8) left the computation window.
Ready.
The forward orbit from (1.9, −5.4) left the computation window.
The backward orbit from (1.9, −5.4) left the computation window.
Ready.
```

FIGURE 11.1 Phase plane of a linear system showing saddle node stability.

time, and trajectories showing how the solution approaches its steady-state from a given initial condition. The first exercise of this chapter will involve writing a simple version of pplane7. Several functions built into MATLAB will aid in coding each of the basic features of the phase plane mentioned previously.

Plotting the nullclines of the system requires no more than the basic plotting commands used throughout previous chapters. Plotting the vector fields can be greatly aided by the functions **meshgrid()** and **quiver()**. The function **meshgrid** takes two vector arguments *x* and *y*, and returns two square matrices *X* and *Y* such that each row of *X* is a copy of the vector *x*, and each column of *Y* is a copy of the vector *y*. This function is useful for evaluating functions of two variables. For example, suppose you wanted to evaluate the function $f(x,y) = x+y$. You could do this using **for** loops; for example, you could type



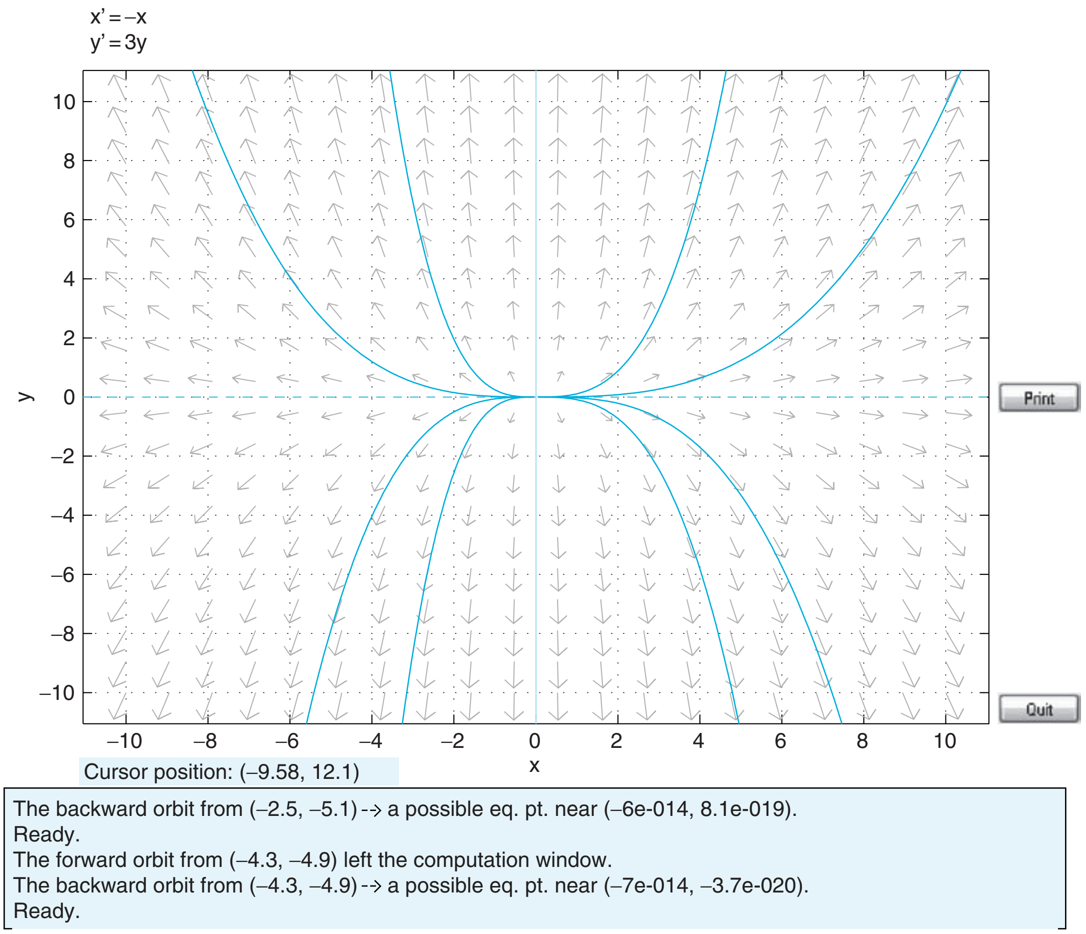
```
The forward orbit from (–2.9, –3.8)--> a possible eq. pt. near (–5.6e-014, –5.1e-020).
The backward orbit from (–2.9, –3.8) left the computation window.
The forward orbit from (–6.1, –4.1)--> a possible eq. pt. near (–8.4e-014, –2e-020).
The backward orbit from (–6.1, –4.1) left the computation window.
Ready.
```

FIGURE 11.2 Phase plane of a linear system showing nodal sink stability.

```
>> x=[0:0.1:10];
>> y=x;
>> for i=1:length(x)
>>     for j=1:length(y)
>>         f(i,j)=x(i)+y(j);
>>     end;
>>end;
```

which produces the same results as the commands

```
>> x=-10:10;
>> y=x;
>> [X, Y]=meshgrid(x,y);
>> f=X+Y;
```



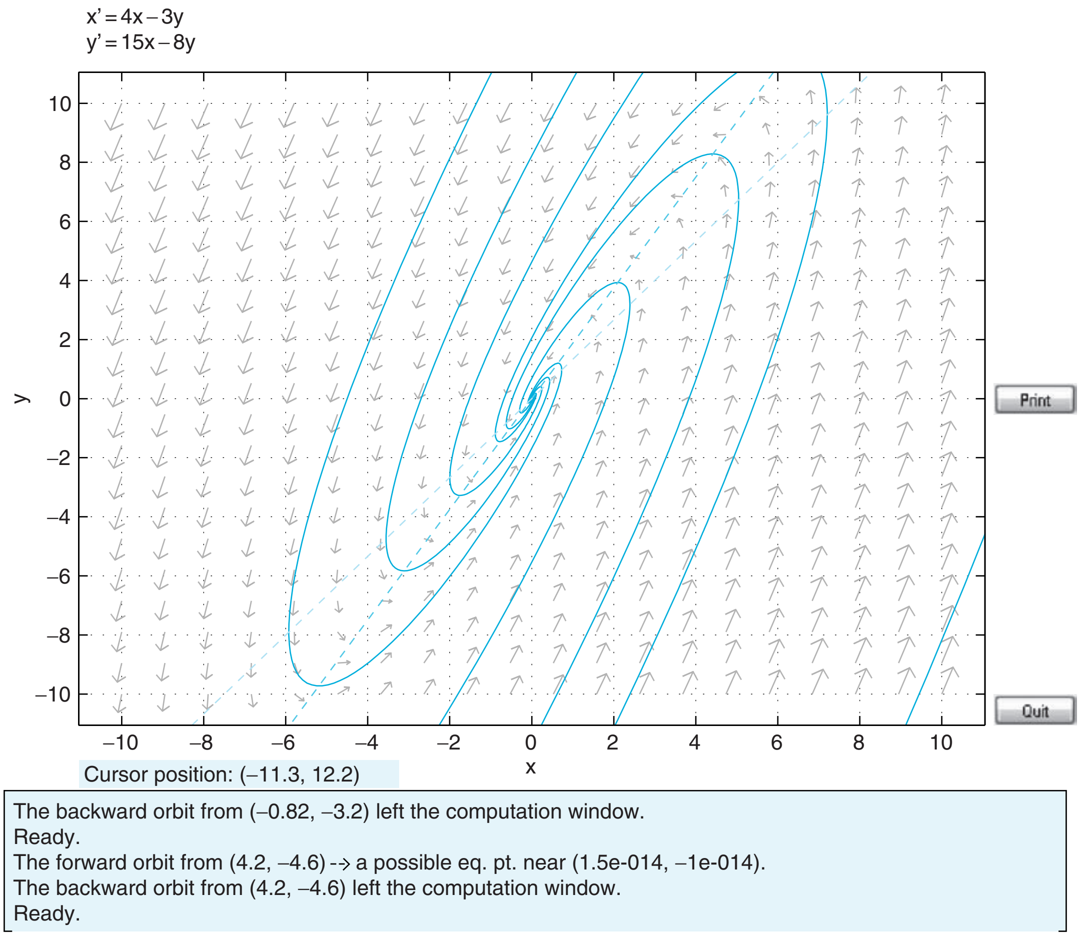
```
The backward orbit from (–2.5, –5.1) --> a possible eq. pt. near (–6e-014, 8.1e-019).
Ready.
The forward orbit from (–4.3, –4.9) left the computation window.
The backward orbit from (–4.3, –4.9) --> a possible eq. pt. near (–7e-014, –3.7e-020).
Ready.
```

FIGURE 11.3 Phase plane of a linear system showing nodal source stability.

Evaluating functions of two variables is important in this chapter because the model you wish to study (Equations 11.1 and 11.2) expresses the derivatives as functions of the two variables: $\tilde{C}$ and $\tilde{H}$. By comparing the matrix f as defined in the preceding code to Equation 11.1, you see that f holds the values of the derivative $\frac{dx}{dt}$ for several values of x and y. You could define a matrix g that holds the y-derivative by using the following command:

>>g=4X+Y;

Once you have evaluated these derivatives using **meshgrid,** we can plot a vector field using the **quiver** command. If you have the matrices X, Y, f, and g defined as shown here, then type this command:



```
The backward orbit from (–0.82, –3.2) left the computation window.
Ready.
The forward orbit from (4.2, –4.6) --> a possible eq. pt. near (1.5e-014, –1e-014).
The backward orbit from (4.2, –4.6) left the computation window.
Ready.
```

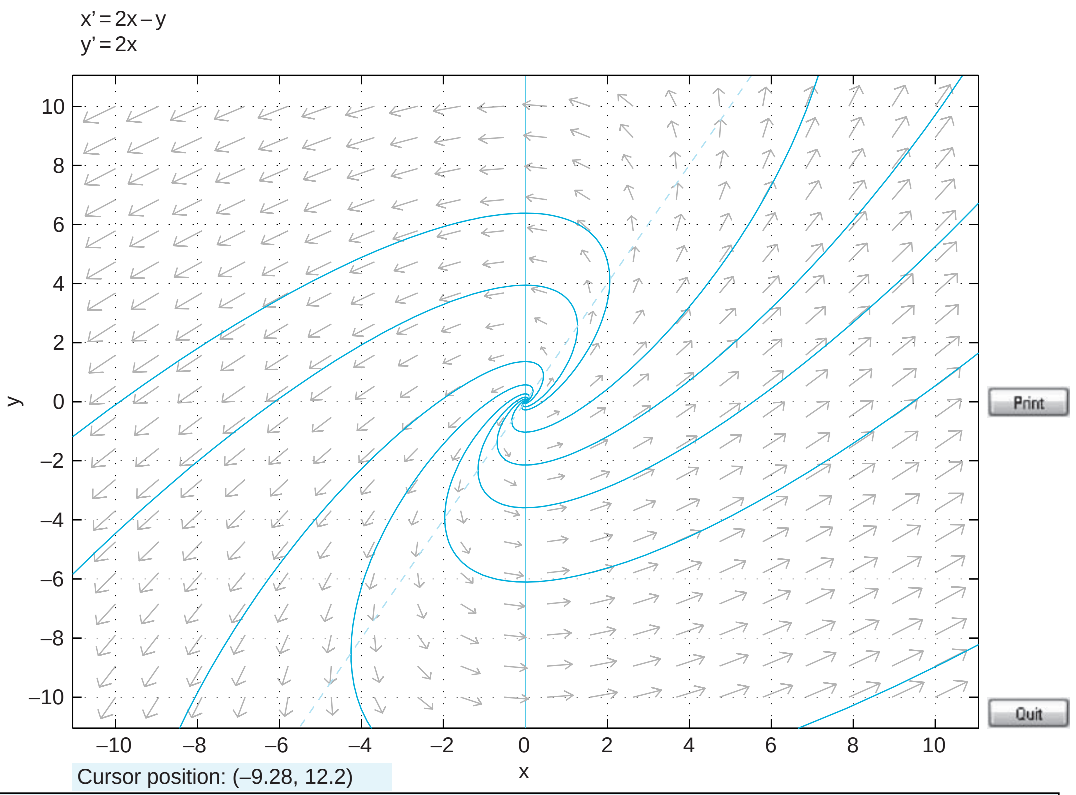
FIGURE 11.4 Phase plane of a linear system showing spiral sink stability.

>>quiver(X,Y,f,g);

You should get the result shown in Figure 11.7.

The function **quiver** works by plotting a vector on the plane at points (x, y) with components (f, g).

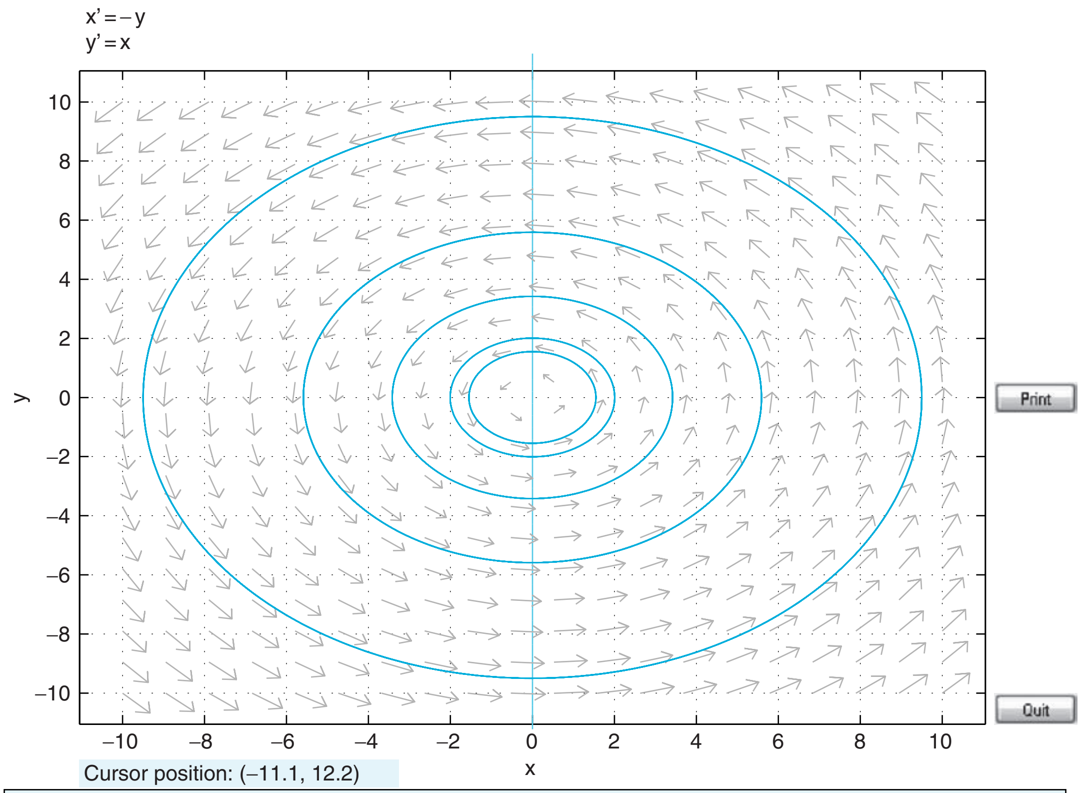
You can plot the trajectory of a system given an initial condition in several ways. One method is to use a numerical solver such as the **ode_euler()** or **RK4()** functions you will write in Chapter 19, "Voltage-Gated Ion Channels," to solve for x and y given some initial condition and then plot x versus y. Another method would be to calculate the derivatives of x and y at the initial condition, move the system a short distance in the direction indicated by the derivative, and then repeat over many time steps. Either method will work, and both can be done with no more than the basic functions introduced in Chapter 2, "MATLAB Tutorial."



```
The backward orbit from (−1.5, 6.1) --> a possible eq. pt. near (−6.6e-016, −1.7e-014).
Ready.
The forward orbit from (4.7, 4.5) left the computation window.
The backward orbit from (4.7, 4.5) --> a possible eq. pt. near (−6.9e-015, −7.1e-015).
Ready.
```

FIGURE 11.5 Phase plane of a linear system showing spiral source stability.

Exercise 11.3.1: Write a function **phase_plane(A, init)** that takes a matrix and performs a phase plane analysis for the linear system, $\mathbf{u}' = \mathbf{A}\mathbf{u}$. The function should plot a phase plane with axes x and y, plot the nullclines, create the vector field, and plot the phase plane trajectory that passes through the initial condition. Finally, the program should output the type of equilibrium point, saddle point, spiral sink, etc. If the equilibrium point is nongeneric, then the program can just output nongeneric as the class type. This is quite a complicated program, so you may want to write several smaller functions that can be called within **phase_plane**—for example, a separate function that will simply classify the fixed point and then another that will create a vector field, etc. *Hint:* The Boolean function **isreal()** will return 0 if the argument is not a real number and 1 if it is. This function might be useful for deciding whether or not the eigenvalues of A are real, so that the fixed point of the system can be classified.



```
The backward orbit from (3, –4.7) --> a nearly closed orbit.
Ready.
The forward orbit from (4.1, –8.6) --> a nearly closed orbit.
The backward orbit from (4.1, –8.6) --> a nearly closed orbit.
Ready.
```

FIGURE 11.6 Phase plane of a linear system showing center stability.

11.4. PROJECT

Use your **phase_plane** program to analyze the retinal model described at the beginning of this chapter. The matrix describing this linear system is:

$$\begin{bmatrix} \frac{-1}{\tau_C} & \frac{-k}{\tau_C} \\ \frac{1}{\tau_H} & \frac{-1}{\tau_H} \end{bmatrix}$$

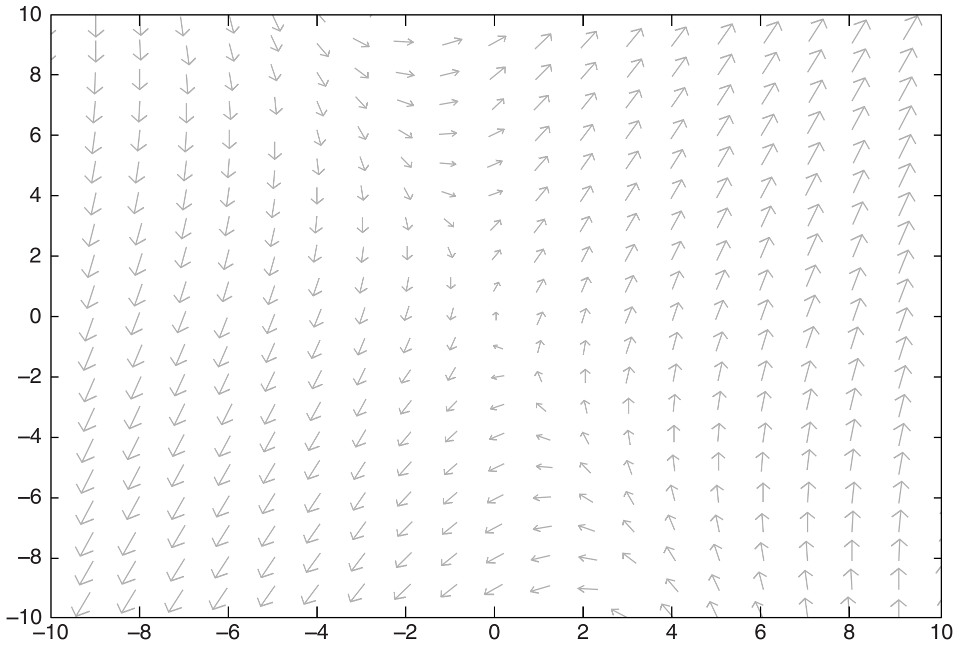


FIGURE 11.7 Vector field created by the **quiver()** command.

Identify what kind of behavior the fixed point exhibits. Repeat using the parameters for dim light:

$$\tau_C = 0.1 \text{ sec}, \ \tau_H = 0.5 \text{ sec, and } \ k = 0.5.$$

What is the behavior of the fixed point now?

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

isreal
quiver
eig

CHAPTER

12

Exploring the Fitzhugh-Nagumo Model

12.1. THE GOAL OF THIS CHAPTER

In this chapter we will use the techniques of phase plane analysis to analyze a simplified model of action potential generation in neurons known as the Fitzhugh-Nagumo (FN) model. Unlike the Hodgkin-Huxley model, which has four dynamical variables, the FN model has only two, so the full dynamics of the FN model can be explored using phase plane methods.

12.2. BACKGROUND

The FN model can be created from the Hodgkin-Huxley model by combining the variables V and m into a single variable v and combining the variables n and h into a single variable r. The four equations of the Hodgkin-Huxley model then become the two-equation system (Fitzhugh, 1961)

$$\frac{dv}{dt} = c(v - \frac{1}{3}v^3 + r + I) \tag{12.1}$$

$$\frac{dr}{dt} = -\frac{1}{c}(v - a + br) \tag{12.2}$$

where a, b, c, and I are parameters of the model.

In Chapter 11, "Introduction to Phase Plane Analysis," we analyzed a system of linear differential equations that had the following general form:

$$\frac{dx}{dt} = ax + by \tag{12.3}$$

$$\frac{dy}{dt} = cx + dy \tag{12.4}$$

In the current chapter we would like you to consider more complicated differential equations (such as those of the FN model). Suppose that you have a system of differential equations of the form:

$$\frac{dx}{dt} = f(x,y) \tag{12.5}$$

$$\frac{dy}{dt} = g(x,y), \tag{12.6}$$

where f and g are more complicated functions of x and y. You begin by plotting the x- and y-nullclines, which are given by $f(x, y) = 0$ and $g(x, y) = 0$, respectively. These nullclines may intersect never, once, or more than once. If the nullclines never intersect, then the system has no finite steady-state solutions. If there is one point of intersection, then there is only one steady-state solution. Linear systems have at most one steady-state solution (unless they are degenerate). Nonlinear systems, however, can have any number of steady-state values. This will be important in your understanding the trajectories, which may be seen in nonlinear systems. A vector field and trajectories given initial conditions can be calculated for nonlinear systems in the exact same manner as calculated for linear systems. Lastly, you can classify the fixed points (steady-state values) as you did in Chapter 11, "Introduction to Phase Plane Analysis." You perform this by linearizing the functions f and g about each fixed point. You assume that the functions f and g have Taylor expansions, so:

$$f(x,y) = f(x_{ss},y_{ss}) + \frac{\partial f(x_{ss},y_{ss})}{\partial x}(x - x_{ss}) + \frac{\partial f(x_{ss},y_{ss})}{\partial y}(y - y_{ss}) + \textit{ higher order terms}, \tag{12.7}$$

and

$$g(x,y) = g(x_{ss},y_{ss}) + \frac{\partial g(x_{ss},y_{ss})}{\partial x}(x - x_{ss}) + \frac{\partial g(x_{ss},y_{ss})}{\partial y}(y - y_{ss}) + \textit{ higher order terms}. \tag{12.8}$$

As you approach the fixed points, the higher order terms tend to zero since $x - x_{ss}$, $y - y_{ss} << 1$. Additionally, $f(x_{ss}, y_{ss}) = g(x_{ss}, y_{ss}) = 0$, so:

$$f(x,y) \approx \frac{\partial f(x_{ss},y_{ss})}{\partial x}(x - x_{ss}) + \frac{\partial f(x_{ss},y_{ss})}{\partial y}(y - y_{ss}) \quad \text{and} \tag{12.9}$$

$$g(x,y) \approx \frac{\partial g(x_{ss},y_{ss})}{\partial x}(x - x_{ss}) + \frac{\partial g(x_{ss},y_{ss})}{\partial y}(y - y_{ss}). \tag{12.10}$$

Substituting these equations into Equations 12.1 and 12.2 yields:

$$\frac{dx}{dt} = \frac{d(x - x_{ss})}{dt} = \frac{\partial f(x_{ss},y_{ss})}{\partial x}(x - x_{ss}) + \frac{\partial f(x_{ss},y_{ss})}{\partial y}(y - y_{ss}) \tag{12.11}$$

$$\frac{dy}{dt} = \frac{d(y - y_{ss})}{dt} = \frac{\partial g(x_{ss},y_{ss})}{\partial x}(x - x_{ss}) + \frac{\partial g(x_{ss},y_{ss})}{\partial y}(y - y_{ss}). \tag{12.12}$$

Expressing this system as a matrix equation gives:

$$\begin{bmatrix} (x - x_{ss})' \\ (y - y_{ss})' \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f(x_{ss}, y_{ss})}{\partial x} & \dfrac{\partial f(x_{ss}, y_{ss})}{\partial y} \\ \dfrac{\partial g(x_{ss}, y_{ss})}{\partial x} & \dfrac{\partial g(x_{ss}, y_{ss})}{\partial x} \end{bmatrix} * \begin{bmatrix} (x - x_{ss}) \\ (y - y_{ss}) \end{bmatrix}. \tag{12.13}$$

If you let:

$$u = \begin{bmatrix} (x - x_{ss}) \\ (y - y_{ss}) \end{bmatrix} \quad \text{and} \quad J = \begin{bmatrix} \dfrac{\partial f}{\partial x} & \dfrac{\partial f}{\partial y} \\ \dfrac{\partial g}{\partial x} & \dfrac{\partial g}{\partial y} \end{bmatrix} \tag{12.14}$$

then you can write Equation 12.13 as:

$$u' = J|_{(x_{ss}, y_{ss})} * u. \tag{12.15}$$

The matrix J is called the *Jacobian matrix*. It is a very important matrix in the mathematics of multivariable calculus. Equation 12.15 tells you that to a first-order approximation the nonlinear system in Equations 12.5 and 12.6 can be approximated by the linear system of Equation 12.15. The eigenvalues of the Jacobian matrix (evaluated at the fixed point) allow you to classify the fixed point as a saddle point, spiral sink, etc. Equation 12.15 is an approximation to the nonlinear system. You might wonder at what point the approximation breaks down. There is a theorem that we will state without proof which says that when the dynamics of the fixed point of the linear system in Equation 12.12 is a generic fixed point, then the fixed point of the nonlinear system in Equations 12.1 and 12.2 has the same dynamics. If the linear system has a nongeneric fixed point such as a center, then no conclusion can be drawn about the dynamics of the fixed point of the nonlinear system. See Chapter 11, "Introduction to Phase Plane Analysis," for a review of generic and nongeneric equilibria.

Note that information about the dynamics of the fixed point applies only to a limited neighborhood centered about the fixed point. A spiral source, for example, can spiral out to infinity or spiral out and approach a circular orbit. The latter case is called a *limit cycle*. Nonlinear systems in higher dimensions (three or more) can have even more complicated dynamics, not all of which have currently been discovered. The best studied dynamics of higher order nonlinear systems include Lorenz attractors and chaos.

12.3. EXERCISES

In this chapter we will explore the **pplane7** program written by Dr. John C. Polking of Rice University. This program was used to make the figures in the Background section of Chapter 11, "Introduction to Phase Plane Analysis," and can be downloaded free at `http://math.rice.edu/~dfield/`. After downloading the script, you can run it by typing the following:

```
>> pplane7;
```

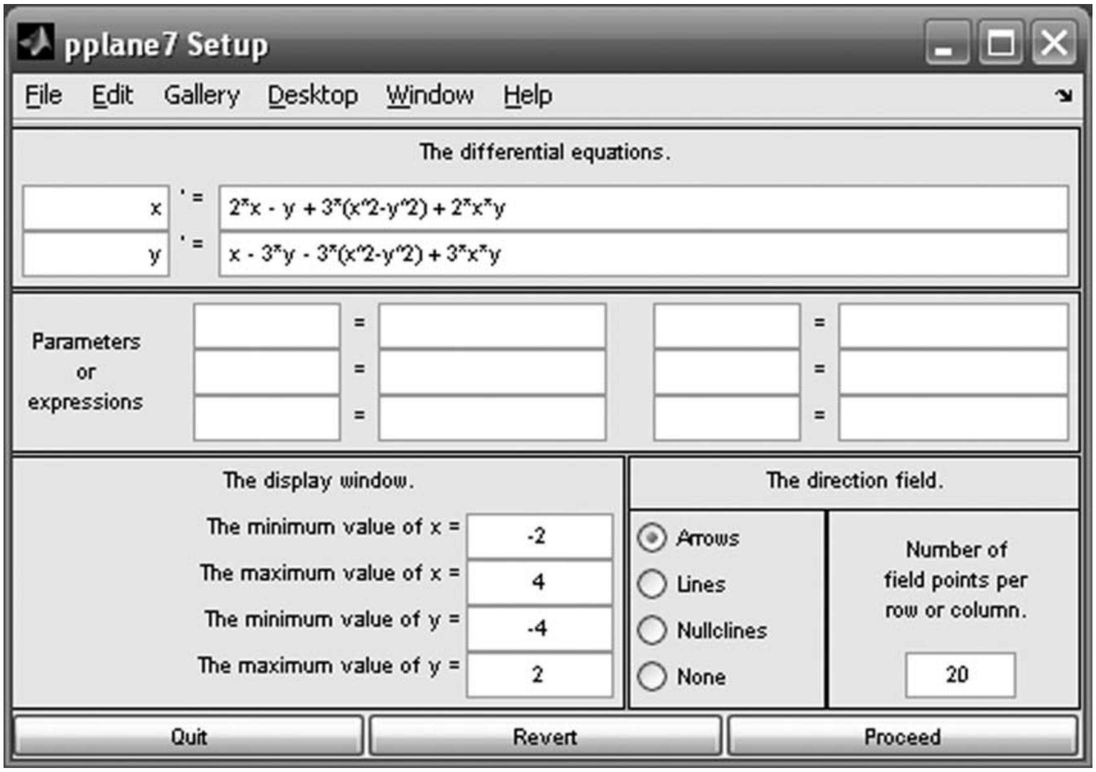
Entering this command will open the pplane7 Setup window shown in Figure 12.1.

Set up the FN model by changing the variables x and y to v and r according to Equations 12.1 and 12.2. The parameter values you can use for now are $a = 0.7$, $b = 0.8$, $c = 3$, and $I = 0$. Set the display window such that v ranges from –3 to 3 and r ranges from –2 to 4. Leave the other settings the same. If you have done this correctly, the Setup Window will look like the one in Figure 12.2.

Now click the Proceed button, and a pplane7 Display window will come up, as shown in Figure 12.3.

Next, open the Solutions menu and click Show nullclines. This will display the v-nullcline in magenta and the r-nullcline in red. The phase plane will now look like the one shown in Figure 12.4.

Next, open the Option menu, select Solution Direction, and then select Forward. This will ensure that when an initial condition is provided to the system, the trajectory will



The differential equations.

x ' = 2*x - y + 3*(x^2-y^2) + 2*x*y

y ' = x - 3*y - 3*(x^2-y^2) + 3*x*y

Parameters or expressions

The display window.

The minimum value of x = -2

The maximum value of x = 4

The minimum value of y = -4

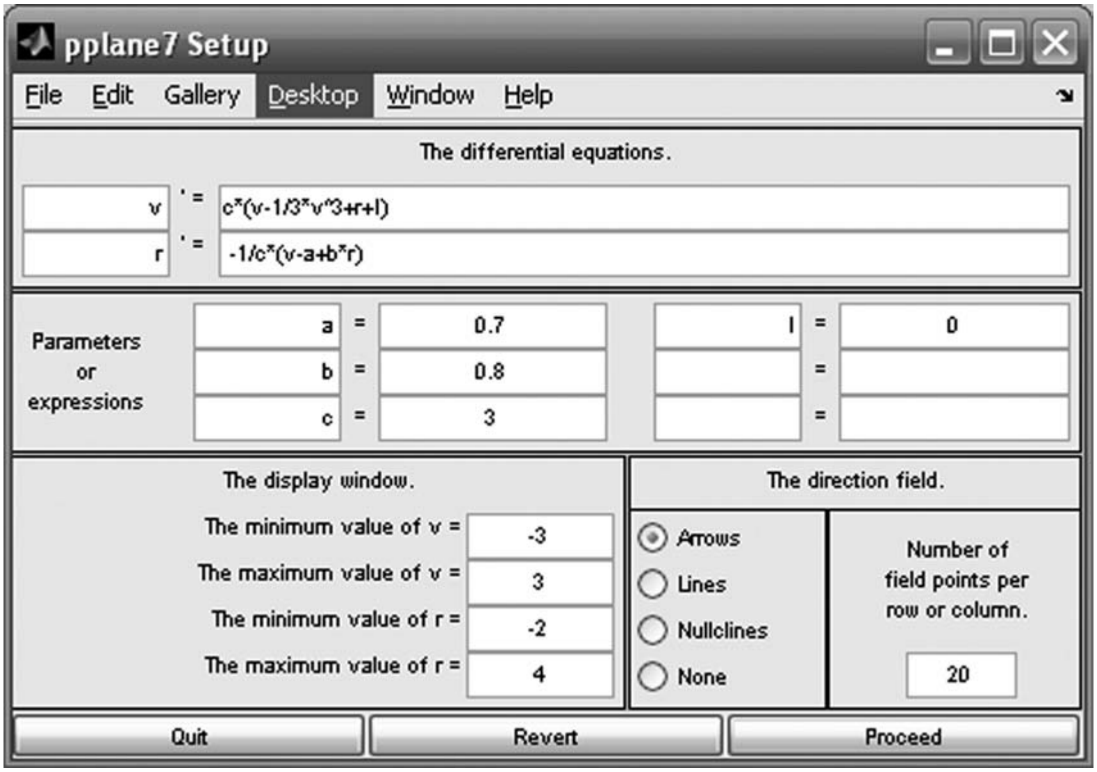
The maximum value of y = 2

The direction field.

[✓ Arrows] Lines Nullclines None

Number of field points per row or column. 20

FIGURE 12.1 pplane7 Setup window.



The differential equations.

v ' = c*(v-1/3*v^3+r+I)

r ' = -1/c*(v-a+b*r)

Parameters or expressions

a	0.7	I	0
b	0.8		
c	3		

The display window.

The minimum value of v = -3

The maximum value of v = 3

The minimum value of r = -2

The maximum value of r = 4

The direction field.

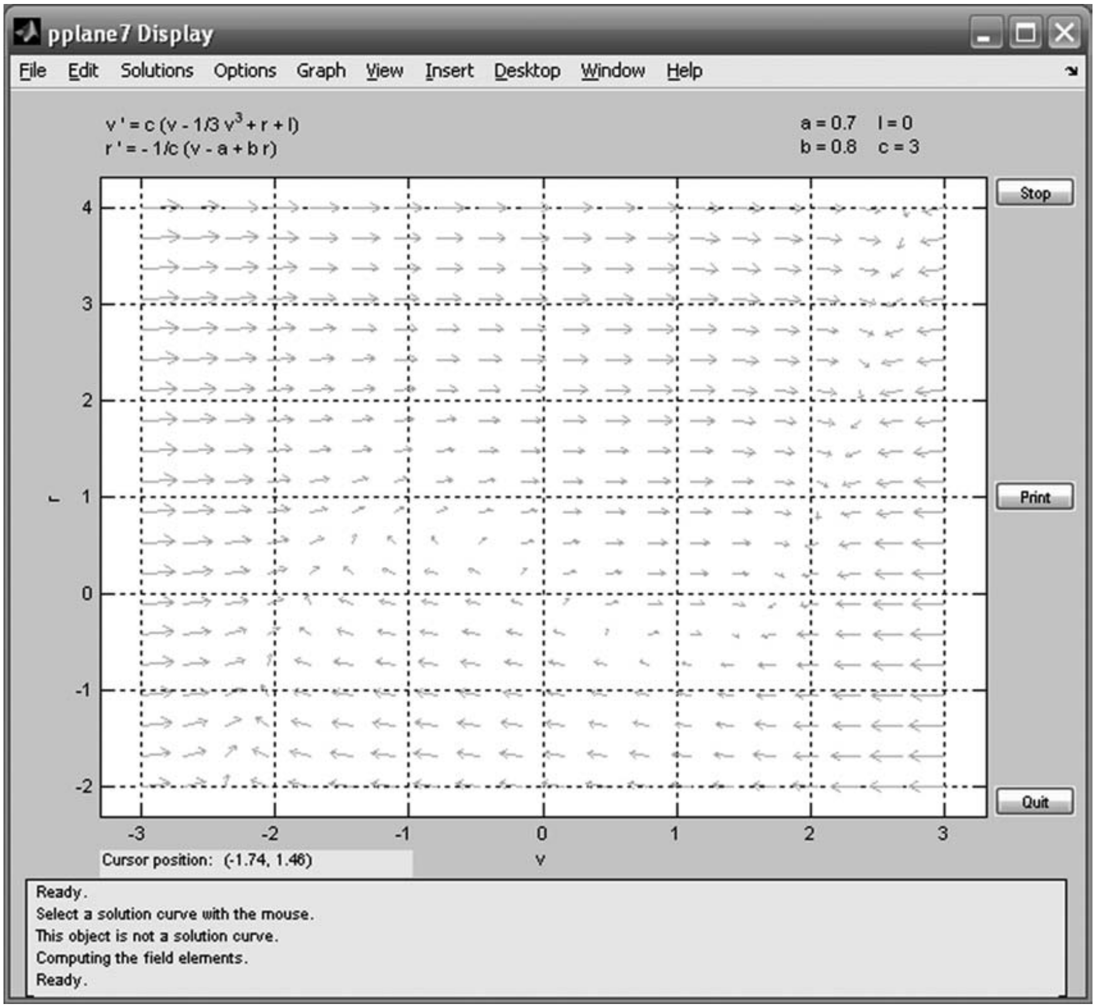
[✓ Arrows] Lines Nullclines None

Number of field points per row or column. 20

FIGURE 12.2 pplane7 Setup window with FN model.

be plotted only as time moves forward. Finally, open the Solutions menu and select Find an Equilibrium Point. This will turn the mouse pointer into a crosshair. Place the crosshair near the intersection of the nullclines and click. An Equilibrium point data window will open, revealing that the equilibrium is located at $(v, r) = (1.1994, -0.62426)$. If you would like to enter in an initial condition to see a trajectory in the phase plane, you have two options. First, you can open the Solutions menu and then click Keyboard Input. This will allow you to enter the initial conditions. After you click Compute, a trajectory in blue is depicted on the phase plane in the pplane7 Display window. Alternatively, you can click Solutions and then select Plot several solutions. Again, the mouse pointer is converted to a crosshair. You can now click on the phase plane at the point representing the initial condition and press Enter. Several trajectories are shown in the phase plane in Figure 12.5.

Finally, you can obtain the voltage trace from the phase plane by opening the Graph menu and selecting *v vs t*. This will again convert the mouse pointer into a crosshair. Use the crosshair to select any trajectory on the phase plane. A pplane7 *t*-plot such as the one in Figure 12.6 will appear.

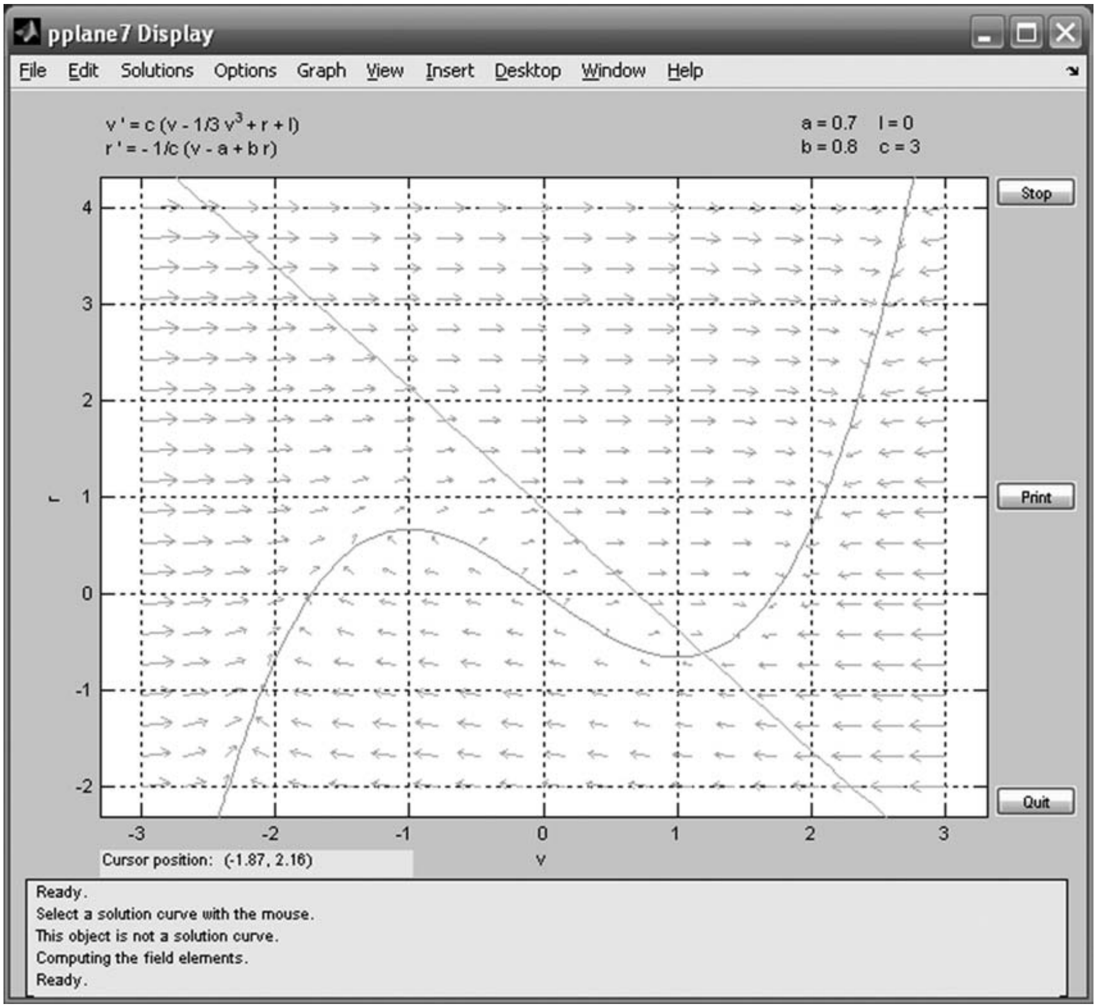

pplane7 Display
File Edit Solutions Options Graph View Insert Desktop Window Help
v ' = c (v - 1/3 v^3 + r + I)
r ' = - 1/c (v - a + b r)
a = 0.7 I = 0
b = 0.8 c = 3
Stop
Print
Quit
r
v
Cursor position: (-1.74, 1.46)

```
Ready.
Select a solution curve with the mouse.
This object is not a solution curve.
Computing the field elements.
Ready.
```

FIGURE 12.3 pplane7 Display window.

The plot in Figure 12.6 shows that if you change the membrane potential of the neuron to 2.4, it decays back down to the equilibrium value 1.1994 as previously determined. This is analogous to giving a neuron a subthreshold depolarizing stimulus. After the brief depolarizing stimulus, the neuron's membrane potential will exponentially relax back down to its equilibrium resting potential.

Exercise 12.1: Is the equilibrium point in the preceding model system stable (i.e., are trajectories attracted to this point or repelled from it)?


pplane7 Display
File Edit Solutions Options Graph View Insert Desktop Window Help
v ' = c (v - 1/3 v^3 + r + I)
r ' = - 1/c (v - a + b r)
a = 0.7 I = 0
b = 0.8 c = 3
Stop
Print
Quit
r
v
Cursor position: (-1.87, 2.16)

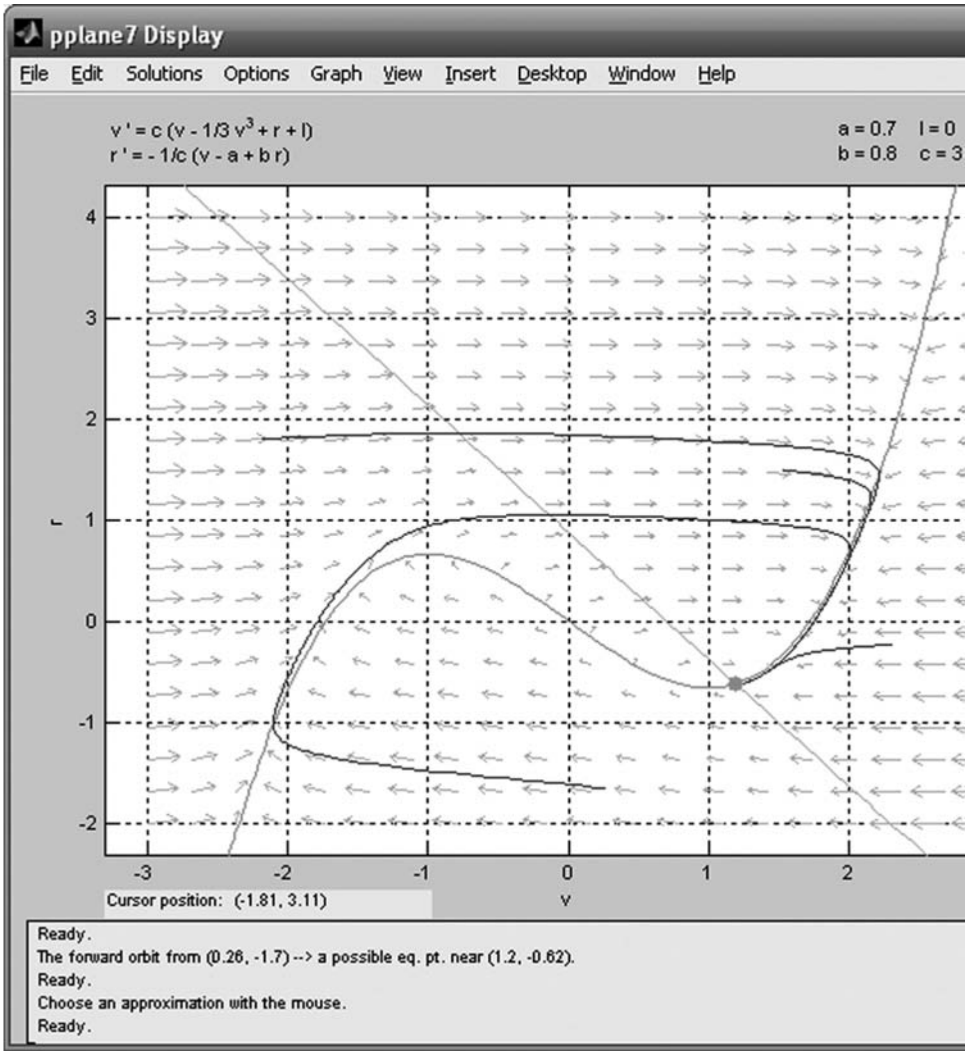
```
Ready.
Select a solution curve with the mouse.
This object is not a solution curve.
Computing the field elements.
Ready.
```

FIGURE 12.4 Phase plane with v- and r-nullclines depicted.

12.4. PROJECT

In this project, you will explore the Fitzhugh-Nagumo model that you set-up with **pplane7** by injecting different levels of current and examining how the behavior of the model neuron mimics that of a real neuron. Specifically, you should do the following:

- Change the injected current value to $I = -0.2$ in the Setup window and click Proceed. Follow the previous instructions to display the nullclines. Calculate a trajectory in the Forward direction with the initial condition $(v, r) = (1.1994, -0.62426)$. Is this point still stable?


pplane7 Display
File Edit Solutions Options Graph View Insert Desktop Window Help
v ' = c (v - 1/3 v^3 + r + I)
r ' = - 1/c (v - a + b r)
a = 0.7 I = 0
b = 0.8 c = 3
4
3
2
1
0
-1
-2
r
-3
-2
-1
0
1
2
v
Cursor position: (-1.81, 3.11)

```
Ready.
The forward orbit from (0.26, -1.7) --> a possible eq. pt. near (1.2, -0.62).
Ready.
Choose an approximation with the mouse.
Ready.
```

FIGURE 12.5 Phase plane with sample trajectories.

- Determine what v versus t looks like for a trajectory on this phase plane. Would you classify the injected input of -0.2 as a superthreshold or subthreshold stimulus? Does this neuron exhibit subthreshold oscillations for this value of injected current?
- Change the injected current value to $I = -0.4$ in the Setup window and click Proceed. Follow the previous instructions to display the nullclines. Calculate a trajectory in the Forward direction with the initial condition $(v, r) = (1.1994, -0.62426)$. Is this point still stable? Plot several trajectories on this phase plane. Since the nullclines intersect at only a single point, there are no other equilibrium points for this system, but trajectories may be

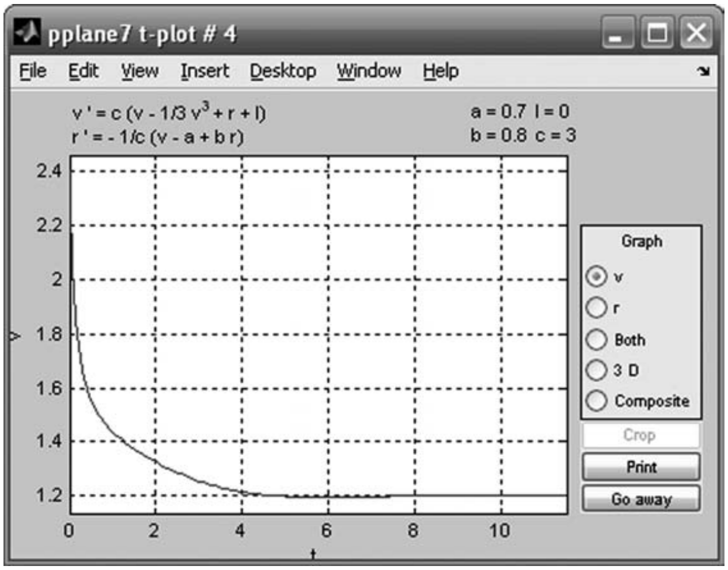

pplane7 t-plot # 4
File Edit View Insert Desktop Window Help
v ' = c (v - 1/3 v^3 + r + I)
r ' = - 1/c (v - a + b r)
a = 0.7 I = 0
b = 0.8 c = 3
Graph
v
r
Both
3 D
Composite
Crop
Print
Go away
t

FIGURE 12.6 pplane7 *t*-plot showing voltage over time.

attracted to some other closed orbit—for example, a circular orbit. Are these trajectories attracted to a closed orbit?

- Determine what v versus t looks like for a trajectory that is attracted to a closed orbit, also called a *limit cycle*? Would you classify this injected stimulus as a superthreshold or subthreshold stimulus?
- Finally, repeat the analysis for $I = -1.6$ and examine v versus t. Does this neuron spike continuously as it did before? Neurons are known to exhibit a phenomenon called *excitation block*, whereby increasing the current injection can often repress repetitive firing behavior.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

pplane7 (free script by John C. Polking available at `http://math.rice.edu/~dfield/`)

CHAPTER

13

Neural Data Analysis: Encoding

13.1. GOALS OF THIS CHAPTER

The primary goal of this chapter is to introduce you to fundamental methods of analyzing spike trains of single neurons used to characterize their encoding properties: raster plots, peri-stimulus time histograms, and tuning curves. While there are prepackaged tools available for these methods, in this chapter you will program these tools yourself and use them to analyze behavioral data recorded from motor areas of a macaque monkey.

13.2. BACKGROUND

In general, neuroscientists are interesting in knowing what neurons are doing. More specifically, neuroscientists are often interested in neural *encoding*—how neurons represent stimuli from the outside world with changes in their firing properties. Let's say you are studying a neuron from a visual area. You would first present a subject with controlled visual stimuli with a number of known properties—orientation, luminance, contrast, etc. If you are studying a neuron from a motor area, your stimuli might be a set of stereotyped movements which the subject performs. Using standard electrophysiological techniques, you then record the response of the neuron to each stimulus. You can repeat the presentation of a given stimulus and then see how similar (or different) the neuronal responses are. A *raster plot* is a simple method to visually examine the trial-by-trial variability of these responses. You can examine what features these responses have in common by averaging over all responses to create a *peri-stimulus time histogram*. Finally, to capture how the average response of the neuron varies with some sensory or motor feature, you can generate a *tuning curve* that maps the feature value onto the average response of the neuron.

13.3. EXERCISES

13.3.1. Raster Plot

Because action potentials are stereotyped events, the most important information they carry is in their timing. A raster plot replaces each action potential with a tick mark that corresponds to the point where the raw voltage trace crosses some threshold.

Load the dataset for this chapter from the companion website. Contained within that dataset is a variable *spike*, which contains the firing times (in seconds) of a single neuron for 47 trials of the same behavioral task. Here, you are recording from a cell in the motor cortex, and the task involves moving the hand from the same starting position to the same ending position. For each trial, the spike times are centered so that the start of movement coincides with a timestamp of 0 seconds. Because the neuron did not fire the same number of times for each trial, the data are stored in a struct, which is a data structure that can bundle vectors (or matrices) of different lengths. To access the spike times for the first and second trials, type:

```
t1 = spike(1).times;
t2 = spike(2).times;
```

If you look at the workspace, you can verify that that the vectors *t1* and *t2* are not the same length. Now plot the first trial as a raster:

```
figure                          %Create a new figure
hold on                         %Allow multiple plots on the same graph
for i = 1:length(t1)            %Loop through each spike time
        line([t1(i) t1(i)], [0 1])  %Create a tick mark at x = t1(i) with a height of 1
end
ylim([0 5])                     %Reformat y-axis for legibility
xlabel('Time (sec)')            %Label x-axis
```

Even when you're looking at one trial, it appears that the neuron fires sparsely at first but then ramps up its firing rate a few hundred milliseconds before the start of movement. Now plot the next trial:

```
for i = 1:length(t2)
        line([t2(i) t2(i)], [1 2])
end
```

Your result should look like those in Figure 13.1.

The relationship between the firing rate and start of movement is not nearly as clear in the second trial as in the first trial. To resolve this discrepancy, you can write a loop to plot rasters for all trials. You will do this as part of this chapter's project. However, it would also be nice to know what the response of the "average trial" looks like.

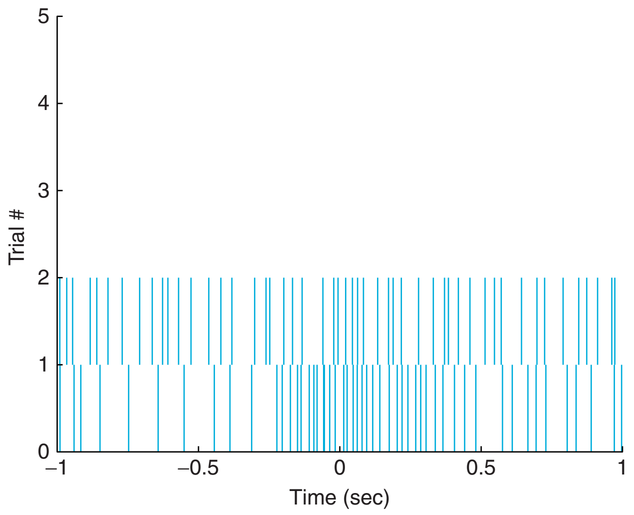


FIGURE 13.1 A raster plot of spike times of the sample neuron for the first two trials.

13.3.2. Peri-Stimulus Time Histogram

The average neural response is captured by the *peri-stimulus time histogram*, which is abbreviated PSTH and sometimes referred to as the *peri-event time histogram*. *Peri-stimulus* means that all the trials are centered relative to some relevant stimulus or event—in this case, the start of movement. *Time histogram* means you divide the time period into a series of bins (0 to 100 ms, 100 to 200 ms, etc.) and count how many spikes fall in each bin for all trials. Luckily, the MATLAB® software has a function that makes this task easy: **histc**. To look at all trials, you will initialize the PSTH with zeros and then sequentially add each trial's results. Try the following:

```
edges = [–1:0.1:1];     %Define the edges of the histogram
psth = zeros(21,1);     %Initialize the PSTH with zeros
for j=1:47              %Loop over all trials
                        %Add current trial's spike times
  psth = psth+histc(spike(j).times,edges);
end
bar(edges,psth);        %Plot PSTH as a bar graph
xlim([-1.1 1])          %Set limits of X-axis
xlabel('Time (sec)')    %Label x-axis
ylabel('# of spikes')   %Label y-axis
```

Your results should look like those in Figure 13.2.

Now the pattern in neuronal activity is clear: the firing rate begins to increase about half a second before movement start and returns to baseline by half a second after movement start. Of course, for the y-axis to indicate firing rate in spikes per second, you need to divide each bin's spike count by both the bin width and the number of trials.

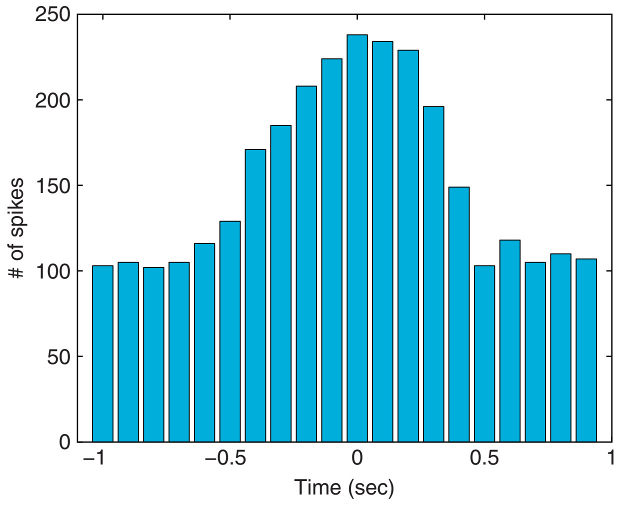


FIGURE 13.2 A peri-stimulus time histogram centered on the start of movement.

13.3.3. Tuning Curves

Many neurons respond preferentially to particular values of the stimulus. Typically, this activity gradually falls off from a maximum (corresponding to the preferred stimulus) along some stimulus dimension (e.g., orientation, direction). By plotting the stimulus dimension on the x-axis and the neural activity (typically a firing rate) on the y-axis, you can determine the preferred stimulus of a neuron. Figure 13.3 shows a tuning curve of a

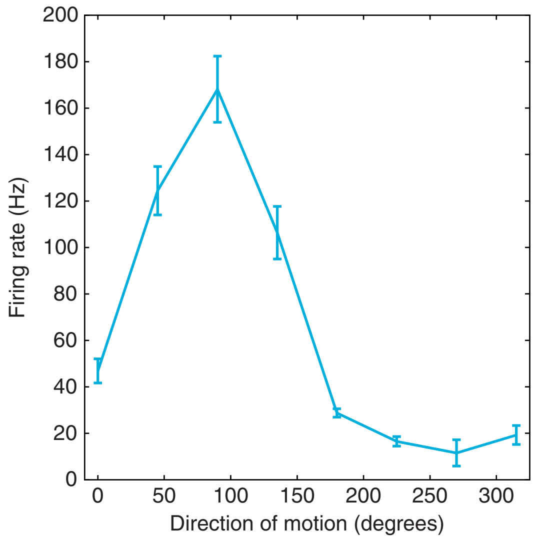


FIGURE 13.3 A tuning curve for a neuron from area MT.

neuron from area MT, which is a part of the visual cortex that aids in the perception of motion. As you can tell, the neuron prefers upward motion (motion toward 90°).

13.3.4. Curve Fitting

Typically, tuning curves like this are fit to a function such as a Gaussian curve or a cosine function. Because all measurements made in the real world come with errors, it is usually impossible to describe empirical data with a perfect functional relationship. Instead, you fit data with a curve that represents a model of the proposed relationship. If this curve fits the data well, then you conclude that your model is a good one.

The simplest relationship you will typically look for is a linear one. Many neurons are thought to encode stimuli linearly. For example, ganglion cells in the limulus (horseshoe crab) increase their firing rate linearly with luminance of a visual stimulus (Hartline, 1940). You can simulate this relationship as follows:

```
x = 1:20;              %Create a vector with 20 elements
y = x;                 %Make y the same as x
z = randn(1,20);       %Create a vector of random numbers with same dimensions as x
y = y + z ;            %Add z to y, introducing random variation
plot(x,y, '.' )        %Plot the data as a scatter plot
xlabel('Luminance')
ylabel('Firing rate')
```

MATLAB contains prepackaged tools for fitting linear relationships. Just click on the figure, select Tools, and then select Basic Fitting. Check the boxes for Linear and Show equations, and you will see the line and equation that best fit your data. However, you might also like to be able to do this yourself. The command in MATLAB to fit data to a polynomial is **polyfit**. For example:

```
p=polyfit(x,y,1) %fits data to a linear, 1st degree polynomial
```

The first value in *p* is the slope and the second value is the y-intercept. If you plot this fitted line, your result should be similar to Figure 13.4.

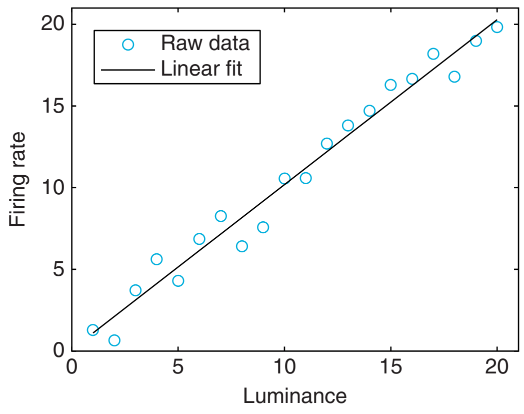


FIGURE 13.4 A linear fit of the relationship between the firing rate of a simulated ganglion cell and the luminance of the stimulus.

```
hold on                 %Allow 2 plots of the same graph
yFit = x*p(1)+p(2);    %Calculate fitted regression line
plot(x,yFit)            %Plot regression
```

Now you will fit data to a more complicated function—a cosine. First, generate some new simulated data:

```
x = 0 : 0.1 : 30;       %Create a vector from 0 to 10 in steps of 0.1
y = cos (x);            %Take the cosine of x, put it into y
z = randn(1,301);       %Create random numbers, put it into 301 columns
y = y + z;              %Add the noise in z to y
figure                  %Create a new figure
plot (x,y)              %Plot it
```

MATLAB does not have a built-in function for fitting this to a cosine, but it does have a nonlinear curve-fitting function: **nlinfit**. You will need to specify the details of the fit. Here, you will use a cosine function with the y-offset, amplitude, and phase as free parameters. You can define this function "inline," which means it can be used by other functions in MATLAB in the same session or M-file.

Type this command to define a generic cosine function:

```
mystring = 'p(1) + p(2) * cos ( theta - p(3) )'; %Cosine function in string form
```

Here, *p(1)* represents the y-offset; *p(2)*, the amplitude; and *p(3)*, the phase. You can assume the frequency is 1. Now enter the following:

```
myfun = inline ( mystring, 'p', 'theta' ); %Converts string to a function
```

This function accepts angles *theta* and parameter vector *p* and transforms them using the relationship stored in *mystring*.

```
p = nlinfit(x, y, myfun, [1 1 0] ); %Least squares curve fit to inline function "myfun"
```

The first parameter of **nlinfit** is a vector of the x-values (the angle *theta* in radians). The second parameter is the observed y-values. The third parameter is the name of the function to fit, and the last parameter is a vector with initial guesses for the three free parameters of the cosine function. If the function doesn't converge, use a different initial guess. The **nlinfit** function returns the optimal values of the free parameters (stored in *p*) that fit the data with the cosine function, as determined by a least squares algorithm.

Optionally, instead of defining a function inline, you can also save a function in an M-file. In that case, you will need to include an @ (at) symbol before the function name, which will allow MATLAB to access the function as if it were defined inline:

```
p = nlinfit(x, y, @myfun, [1 1 0] ); %Least squares curve fit to function "myfun.m"
```

You can use the inline function to convert the optimized parameters into the fitted curve. After plotting this, your result should look similar to Figure 13.5.

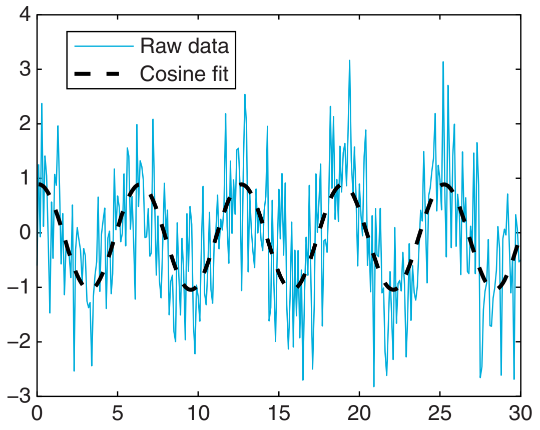


FIGURE 13.5 A nonlinear fit of a simulated, noisy cosine relationship.

```
hold on              %allows 2 plots of the same graph
yFit = myfun(p,x);   %calculates fitted regression line
plot(x,yFit,'k')     %plots regression
```

13.4. PROJECT

The data that you will use for your project were recorded from the primary motor cortex (abbreviated MI) of a macaque monkey (data courtesy of the Hatsopoulos laboratory). MI is so named because movements can be elicited by stimulating this area with a small amount of electricity. It has also been shown that MI has direct connections to the spinal cord. MI plays an important role in the control of voluntary movement (as opposed to reflexive movements). This doesn’t mean that MI directly controls movement, because other areas in the basal ganglia and the brainstem are important as well. Animals with a lesioned MI can still make voluntary movements, albeit less dexterously than before. However, it is clear that MI contains information about voluntary movement, usually a few hundred milliseconds before it actually happens. Thus, the usual terminology of “stimulus” and “response” is awkward here because the neural “response” usually precedes the experimental “stimulus.” There is also a somatotopic map in MI, meaning that there are separate areas corresponding to face, arm, leg, or hand movements. These data are recorded from the arm area.

The behavioral data were collected using a manipulandum, which is an exoskeleton that fits over the arm and constrains movement to a 2-D plane. Think of the manipulandum as a joystick controlled with the whole arm. The behavioral task was the center-out paradigm pioneered by Georgopoulos and colleagues (1982). The subject first holds the cursor over the center target for 500 ms. Then a peripheral target appears at one of eight locations arranged in a circle around the center target. In this task there is an instructed delay, which means that after the peripheral target appears, the subject must wait 1000–1500 ms for a go cue. After the go cue, the subject moves to and holds on the peripheral target for 500 ms, and the trial is completed.

There are two interesting time windows here. Obviously, MI neurons should respond during a time window centered around the go cue, since this is when voluntary movement begins. However, MI neurons also respond during the instructed delay. This result is somewhat surprising because the subject is holding still during this time. The usual interpretation is that the subject is imagining or preparing for movement to the upcoming target. This means that MI is involved in planning as well as executing movement.

If you treat the direction to the peripheral target as the stimulus, you can arrange the neuronal responses in a tuning curve. These can be described with the same cosine curve used before, where the phase of the fitted cosine corresponds to the preferred direction of the neuron.

In this dataset, the neuronal spiking is stored in a struct called *unit*. Information for unit #1 is accessed with *unit(1)*. Spike times are stored in *unit(1).times*. There are three more important variables: the instruction cue times are stored in *instruction*, the go cue times are stored in *go*, and the direction of peripheral target is stored in *direction* (1 corresponds to 0 degrees, 2 corresponds to 45 degrees, etc.).

In this project, you are asked to do the following:

1. Make raster plots and PSTHs for all the neurons for both time periods: instruction cue to 1 second afterward, and 500 ms before the movement onset to 500 ms afterward. Which neurons are the most responsive? Print out a few examples. Do you think the PSTHs are a good summary of the raster plots? How does the time course of the responses differ between the two time periods?
2. Create tuning curves and fit a cosine tuning curve to the firing rates of all neurons for each time period. Report the parameters of the fit for each neuron and save this information for later chapters. How good of a description do you think the cosine curve is? Do the tuning curves differ between the two time periods? If so, why do you think this is?

Figures 13.6 and 13.7 show examples of what your results might look like. The locations of the smaller plots correspond to the locations of their associated peripheral targets. Here,

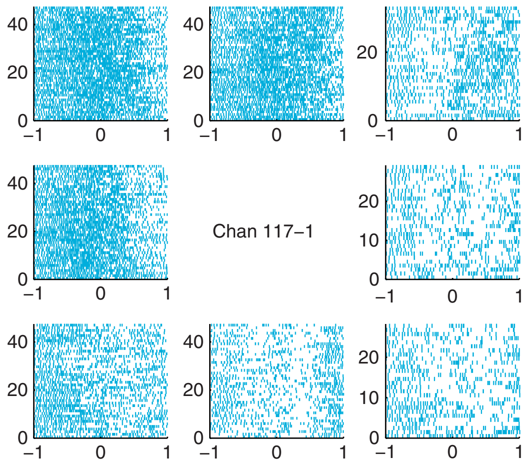


FIGURE 13.6 An example of a full raster plot for the first neuronal unit recorded from electrode #117.

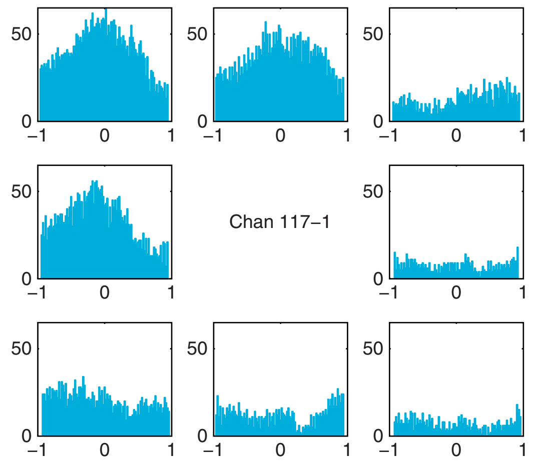


FIGURE 13.7 An example of a full peri-stimulus time histogram for the first neuronal unit recorded from electrode #117.

a timestamp of 0 seconds corresponds to the start of movement. You can use the command **subplot** to subdivide the plotting area. For example, the command **subplot(3,3,i)** makes the i^{th} square in a 3×3 grid the active plotting area.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

histc
randn
bar
polyfit
nlinfit
subplot

CHAPTER

14

Principal Components Analysis

14.1. GOALS OF THIS CHAPTER

Previously, we explored how the MATLAB® software can be used to visualize neural data. This is a powerful tool. For example, a simple 2D tuning curve can demonstrate how a single neuron encodes a stimulus parameter in terms of a firing rate. However, it is not clear how this applies to multidimensional data. How do you represent stimulus encoding of a population of neurons or of a time-varying firing rate?

One solution is to try to compress data to make them easier to work with. If you can reduce the dimensionality to two or three dimensions, you can then use your visualization tools. In this chapter you will see how principal components analysis can be used to perform dimensionality reduction. You will also explore an application of this technique to spike sort neuronal waveforms. This will prepare you for the next chapter, where you will use principal components to capture the temporal aspects of a peri-stimulus time histogram.

14.2. BACKGROUND

Principal components analysis (PCA) performs a linear transformation on data and can be used to reduce multidimensional data down to a few dimensions for easier analysis. The idea is that many large datasets contain correlations between the dimensions, so that a portion of the data is redundant. PCA will transform the data so that as much variation as possible will be crammed into the fewest possible dimensions. This allows you to compress your data by ignoring other dimensions. To apply PCA, you first need to understand how the correlations between dimensions can be described by a covariance matrix.

14.2.1. Covariance Matrices

You will start by analyzing some simulated data. You will look at zero-mean, Gaussian noise ("white noise") in two dimensions. You can generate this in MATLAB using the function **normrnd**. In the first variable (*a*), the two dimensions will be uncorrelated, but in the second (*b*) there will be a significant correlation between the two dimensions. Use the following code to generate *a* and *b*:

```
n=500;                              %n = number of datapoints
a(:,1)=normrnd(0,1,n,1);            %n random Gaussian values with mean 0, std.
                                    %deviation 1
a(:,2)=normrnd(0,1,n,1);            %Repeat for the 2nd dimension.
b(:,1)=normrnd(0,1,n,1);            %n random Gaussian values with mean 0, std.
                                    %deviation 1
b(:,2)=b(:,1)*0.5+0.5*normrnd(0,1,n,1); %For b, the 2nd dimension is correlated with the 1st
```

If you plot the columns of these variables against one another, the data should look something like Figure 14.1.

You should already be familiar with the concepts of *mean*, *variance*, and *standard deviation*. If the sample data consist of n observations stored in a vector x, the sample mean is defined as follows:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{14.1}$$

The variance (σ^2) of the sample is simply the expected value (mean) of the squared deviations from the sample mean:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{14.2}$$

The standard deviation (σ) is just the square root of the variance. Unfortunately, using the preceding expression as an estimate of the sample variance is a bad idea because this estimate is biased: it systematically underestimates the variance. However, it can be shown that the unbiased estimator is formed by replacing n by $n - 1$ in the first term. Thus, the *sample variance* (s^2) is usually defined as follows:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{14.3}$$

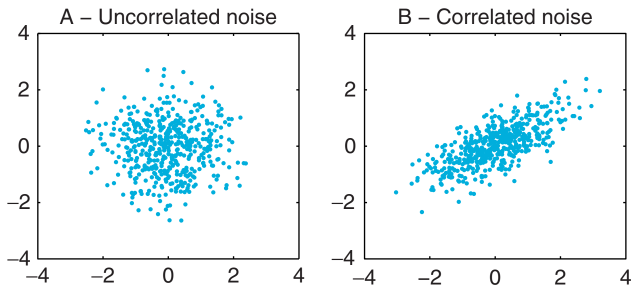


FIGURE 14.1 Samples from a two-dimensional Gaussian distribution where the dimensions are uncorrelated (A) and correlated (B).

This is how the function **var** in MATLAB is defined. If you subtract the mean from your data, then you can more compactly express the sample variance as follows:

$$s^2 = \frac{1}{n-1}(x-\bar{x})^T(x-\bar{x}) \tag{14.4}$$

The superscript T signifies a transpose, whereby matrix columns are changed to rows and vice versa: an m by n matrix becomes an n by m matrix. In MATLAB, a transpose is designated with an apostrophe placed after the variable.

Let's compare the preceding formula with the function **var**. In the following code, do the two expressions give the same result?

```
var(a(:,1))                %Compute sample variance of 1st dim of "a"
c=a(:,1)-mean(a(:,1));     %Subtract mean from 1st dim of "a"
c'*c/(n-1)                 %Compute sample variance of 1st dim of "a"
                           %Note the apostrophe denoting transpose(c)
```

The covariance is analogous to the variance, except that it is computed between two vectors, not a vector and itself. If you have a second data vector y with n independent values, then the *sample covariance* is expressed as follows:

$$\text{cov}(x,y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y}) \tag{14.5}$$

You can see that if x and y are the same, the sample covariance is the same as the sample variance. Also, if x and y are uncorrelated, the covariance should be zero. A positive covariance means that when x is large, so is y; while a negative covariance means that when x is large, y is small. The last thing you need to define is the *covariance matrix*. If the data have m dimensions, then the covariance matrix is an m by m matrix where the diagonal terms are the variance of each dimension and the off-diagonal terms are the covariances between dimensions. If the additional dimensions are stored as extra columns in variable x (so x becomes an n by m matrix), then the sample covariance can be computed the same way as the sample variance:

$$\text{cov}(x) = \frac{1}{n-1}(x-\bar{x})^T(x-\bar{x}) \tag{14.6}$$

This is computed by the function **cov** in MATLAB. Compare the two methods by using the following code (the function **repmat** is used to create multiple copies of a vector):

```
cov(a)                       %Compute the covariance matrix for "a"
c=a-repmat(mean(a),n,1);     %Subtract the mean from "a"
c'*c/(n-1)                   %Compute the covariance matrix for "a"
```

The covariance of the correlated noise should have large off-diagonal terms. One reason to compute the covariance is that it plays the same role in the multivariate Gaussian distribution as the variance plays in the univariate Gaussian. You can use the covariance and the function **mvnrand** (*mvn* stands for *multivariate normal*) in MATLAB to generate new multivariate correlated noise (*b2*). Plot *b2* on top of the correlated noise generated earlier (*b*). Did the covariance matrix adequately capture the structure of the data?

```
sigma = cov(b)                %Compute the covariance matrix of b
b2=mvnrnd([0 0],sigma,n);     %Generate new zero-mean noise with the same covariance matrix
```

14.2.2. Principal Components

Principal components analysis is essentially just a coordinate transformation. The original data are plotted on an *X*-axis and a *Y*-axis. PCA seeks to rotate these two axes so that the new axis X' lies along the direction of maximum variation in the data. PCA requires that the axes be perpendicular, so in two dimensions the choice of X' will determine Y'. You obtain the transformed data by reading the *x* and *y* values off this new set of axes, X' and Y'. For more than two dimensions, the first axis is in the direction of most variation; the second, in direction of the next-most variation; and so on.

How do you get your new set of axes? It turns out they are related to the eigenvalues and eigenvectors of the covariance matrix you just calculated. We previously used eigenvalues and eigenvectors to describe the behavior of a linear system of equations. In PCA, each eigenvector is a unit vector pointing in the direction of a new coordinate axis, and the axis with the highest eigenvalue is the axis that explains the most variation.

This concept may seem confusing, so start by looking at the correlated noise data (*b*). You could make a decent guess at the principal components just by looking at the data: the first principal component line should fall on the long axis of the ellipse-shaped cluster. You can use the function **eig** in MATLAB to compute the eigenvectors and eigenvalues of the covariance matrix **sigma** you computed previously:

```
[V, D] = eig(sigma)   %V = eigenvectors, D = eigenvalues for covariance matrix sigma
```

This will output something like the following (because the noise was generated randomly, the exact values will vary):

```
V =
   0.5387      -0.8425
  -0.8425      -0.5387
D =
  0.2048             0
  0             1.3341
```

The eigenvalues are stored on the diagonal of *D*, while the corresponding eigenvectors are the rows stored in *V*. Because the second eigenvalue is bigger, the second eigenvector is the first principal component. This means that a vector pointing from the origin to (–0.8445, –0.5387) lies along the axis of maximum variation in the data. Type the following to plot the new coordinate axes on the original data:

```
plot(b(:,1),b(:,2),'b.');                       %Plot correlated noise
hold on
plot(3*[-V(1,1) V(1,1)],3*[-V(1,2) V(1,2)],'k')  %Plot axis in direction of 1st eigenvector
plot(3*[-V(2,1) V(2,1)],3*[-V(2,2) V(2,2)],'k')  %Plot axis in direction of 2nd eigenvector
```

This will produce a graph like the one in Figure 14.2.

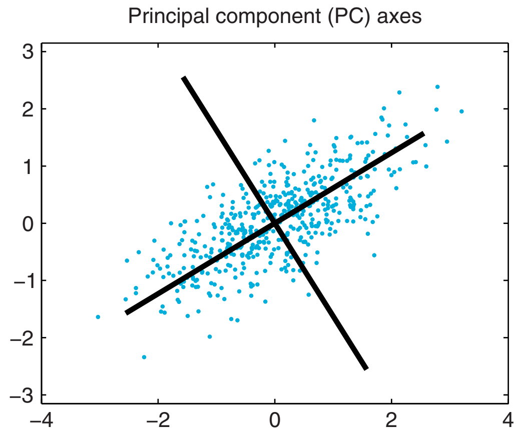


FIGURE 14.2 The first two principal component axes plotted with the original correlated data *b*.

Now you use these new coordinate axes to reassign the (*X*,*Y*) values to all your datapoints. First, you want to reorder the eigenvectors so that the first principal component is in the first row. Then you can simply multiply the data by this reordered matrix to obtain the new, transformed data. For example:

```
V2(:,1) = V(:,2);    %Place the 1st principal component in the 1st row
V2(:,2) = V(:,1);    %Place the 2nd principal component in the 2nd row
newB = b*V2;         %Project data on PC coordinates
```

If you plot these transformed data, it is clear that you just rotated the data so that most of the variation lies along the X-axis, as shown in Figure 14.3.

If you stop here, you haven't gained much, since the transformed data have just as many dimensions as the original data. However, if you wanted to compress the data, you can now

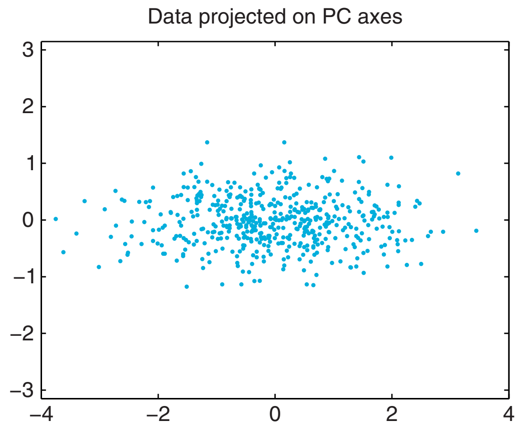


FIGURE 14.3 The correlated data *b* projected on the first two principal components.

just throw away the second column (the data plotted on the *Y*-axis in Figure 14.3). The whole point of PCA is that if you force as much of the variation as possible into a few dimensions, you can throw away the rest without losing much information.

How much variation can you capture by doing this? It turns out that the fraction of variation captured by each principal component is the ratio of its eigenvalue to the sum of all the eigenvalues. For example, the first principal component has an eigenvalue of around 1.33, and the second principal component has an eigenvalue of around 0.20. That means if you keep just the first column of the transformed data, you still keep 87% of the data (0.87=1.33/[1.33+0.20]). That is, you can compress the size of the data by 50% but lose only 13% of the variation.

Conveniently, MATLAB already has a function that performs all these calculations in one fell swoop: **princomp**. Type the following to compute principal components for the correlated data:

```
[coeff,score,latent]=princomp(b); %Compute principal components of data in b
```

The eigenvectors are stored in the variable **coeff**, the eigenvalues are stored in **latent**, and the transformed data (the old data projected onto the new PC axes) are stored in **score**. MATLAB even orders the eigenvectors so that the one with highest eigenvalue is first.

14.2.3. Spike Sorting

One common application of PCA is the spike sorting of neural data. Typically, a data acquisition system monitors a raw voltage trace. Every time the voltage crosses some threshold, the raw voltage is sampled during a time window surrounding this crossing to produce the recorded spike waveform. For example, in one commercially available data acquisition system (Cerebus system, Cyberkinetics Neurotechnology Systems, Inc.), each spike waveform consists of a 1600 μs section of the voltage trace sampled 30 times per millisecond for a total of 48 data points.

Because any experimental system contains noise, the threshold crossing is often triggered by a chance deviation from the mean and not an action potential. Thus, after these recordings are made, the noise must be differentiated from the real spikes. You must also determine if the real spikes came from one or many neurons and then sort them accordingly. How do you compare waveforms? You can start by plotting them all on the same graph. For example, Figure 14.4 contains the first 200 waveforms from one electrode of a multielectrode array recording from the primary motor cortex in a macaque monkey.

First, remember that since this is an extracellular recording, the sign of the voltage trace of the action potential is reversed, and the amplitude is much smaller than for intracellular recordings (microvolts instead of millivolts). You can immediately see that there is a large amplitude unit on the electrode. There may also be a smaller amplitude unit with a larger trough-peak spike width. There is also some noise. It is not immediately clear how to separate these categories, and you are looking at only 200 spikes. How do you deal with all 80,000 spikes that were recorded during an hour-long session? You could represent each waveform as a single point, but then each point would be in a 48-dimensional space. How do you make this analysis easier?

The solution (as you may have guessed) is to compress the data using principal components. Then you can plot the first versus the second principal component and see whether

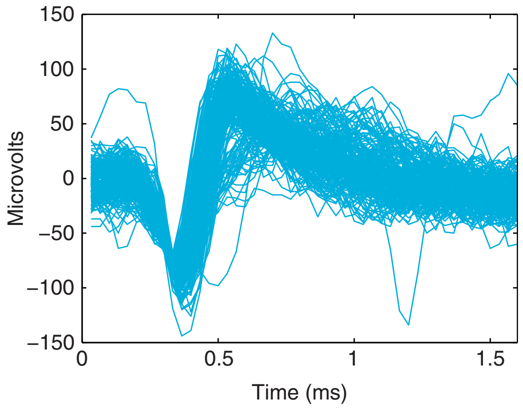


FIGURE 14.4 A plot of 200 extracellular action potential waveforms recorded from a microelectrode array implanted in the primary motor cortex of a macaque monkey.

the data fall into clusters. When you visualize all the spikes in a graph like the one in Figure 14.5, it becomes clear that there are two major clusters.

Unfortunately, with so many spikes, it's not clear how densely packed the clusters are. You can create a 3D histogram using the function **hist3** and then visualize the histogram using the function **surface**. After loading the data from the companion website, use the following code to reproduce Figure 14.6:

```
wf=session(1).wf;                  %Load waveforms
[coeff,score]=princomp(wf);        %Compute principal components
edges{1}=[-300:25:300];            %Bin for the X-axis
edges{2}=[-250:25:250];            %Bin for the Y-axis
h=hist3(score(:,1:2),edges);       %Compute a 2-D histogram
s=surface(h');                     %Visualize the histogram as a surface (note the apostrophe)
set(s,'XData',edges{1})            %Label the X-axis
set(s,'YData',edges{2})            %Label the Y-axis
```

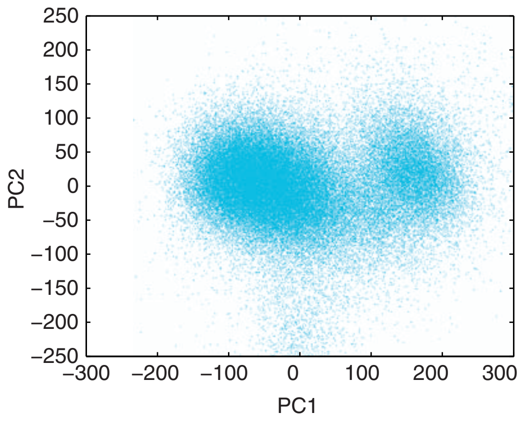


FIGURE 14.5 A scatterplot of the motor cortical spike waveforms projected on the first and second principal components.

The default view is looking straight down on the surface. To make the figure prettier, click the Rotate 3D button (its icon is a counterclockwise arrow encircling a cube). Then click and drag on the figure to rotate it. Play with the function **colormap** to change the color scheme and type **help graph3D** for the list of color maps. Figure 14.6 uses **colormap(white)**.

Unfortunately, it is difficult to quickly spike sort these waveforms without a good graphical user interface. For example, you would like to be able to select a point in PC space and see what the corresponding waveform looks like. You want to be able to circle a group of points and then see both the average waveform and the interspike-interval histogram. This is the sort of functionality provided by commercial spike sorting packages, such as Offline Sorter by Plexon, Inc. You can implement something similar in MATLAB, but doing so is beyond the scope of this book.

In the project you will perform in this chapter, instead of using a graphical selection tool, you will select waveforms by looking at distances in PC space. Pick a point in PC 1 versus PC 2 space that you think is at the center of a cluster. Then calculate the Euclidean distance of every other point from this template point and pick all those that fall below a certain threshold. This is equivalent to drawing a circle on the PC graph and picking all the points that fall within the circle. Now you can calculate any statistic you want of the sorted waveforms, such as the average waveform or the interspike interval histogram. The function **find** may be useful. For example, if you store your Euclidean distances in *dist* and your distance threshold in *threshold*, you can find the indices of all the waveforms meeting this threshold with **ind=find(dist<threshold);**.

You can use this average waveform as the basis of a template sort, a common strategy in spike sorting. The average waveform is a template to which you compare all other waveforms. Calculate the mean-squared error for each waveform from this template waveform. Then keep all waveforms whose mean squared error falls below a certain threshold.

While this procedure would be easier with a commercial spike sorter, sometimes custom procedures in MATLAB can be useful. In the project for this chapter, you will also consider the problem of comparing waveforms from one day to the next. Because the principal components change depending on the data, instead of calculating the PCs of the second day, you will project the second day's data onto the first day's PCs. This isn't something that is usually possible with spike sorting software, so understanding how to implement this in MATLAB expands your analytical possibilities.

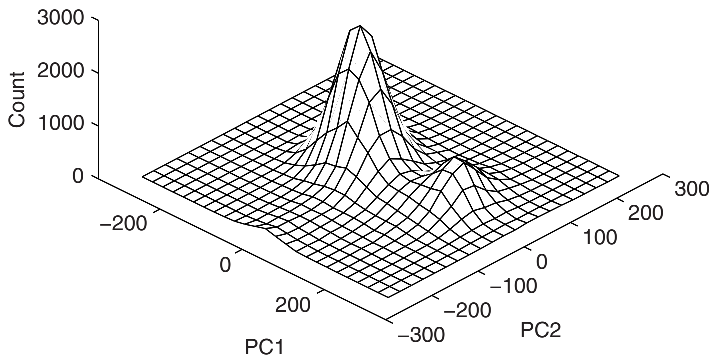


FIGURE 14.6 A three-dimensional histogram showing the frequency of motor cortical spike waveforms projected on the first (PC1) and second (PC2) principal components.

14.3. EXERCISES

Exercise 14.1: When you computed the covariance matrix of the uncorrelated data *a*, why are the off-diagonal terms nonzero? Generate several new examples of uncorrelated noise. What do you think the average covariance matrix should be?

Exercise 14.2: Use **princomp** to compute the principal components of the correlated noise you generated in *b*. Are they different from what you computed using the covariance matrix method? If they are, how would this affect the transformed data?

Exercise 14.3: Use **princomp** to compute the principal components of uncorrelated noise. What are the PCs? What would you expect them to be?

14.4. PROJECT

In this project, you will build your own primitive spike sorter using principal components analysis to analyze extracellular data from recordings in the primary motor cortex of a nonhuman primate (data courtesy of the Hatsopoulos laboratory). The spike waveform and spike times data for this project are stored in a struct called *session*. You can access the data using the following code:

```
wf1 = session(1).wf;           %waveforms from the 1st day
wf2 = session(2).wf;           %waveforms from the 2nd day
stamps1 = session(1).stamps;   %time stamps from the 1st day
stamps2 = session(2).stamps;   %time stamps from the 2nd day
```

Specifically, you are asked to do the following:

1. Apply PCA to the first day's waveforms. What percent of variation is captured by the first two dimensions?
2. Spike sort the first day's waveforms using a template sort. First, select a region of interest in 2D PC space (a circle at the heart of a cluster) by finding all points in PC space within a certain Euclidean distance from a given point. Calculate the average waveform of the waveforms in this region. Use this average waveform as the template

in a template sort. Plot the template and all the sorted waveforms for each neuron you think is present. Also plot the interspike interval histograms, which are just histograms of the times between sorted spikes. The function **diff** may be useful for this task.

3. Project the second day's data onto the first day's principal component's axes. How is this different from the second day's data projected on its own principal components? Repeat the sort you used for the first day's data. How do the neurons compare? Do you think they are the same neurons?

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

cov
eig
hist3
mvnrand
normrnd
princomp
surface
transpose
repmat
find
diff

CHAPTER

15

Information Theory

15.1. GOALS OF THIS CHAPTER

Thus far, we have assumed that a neuron encodes any relevant stimulus parameters by modifying its firing rate. We used this assumption to construct tuning curves describing this stimulus encoding. But a neuron could also encode a stimulus by changing the relative timing of its spikes. In this chapter we will introduce the methodology used in a series of papers by Richmond and Optican exploring temporal encoding in a primate visual area. They used principal components analysis and information theory to argue that a temporal code provided more information about the stimulus than a rate code did. You will apply similar methodology to data recorded from the primate motor cortex. Note that this chapter assumes familiarity with principal components analysis introduced in Chapter 14.

15.2. BACKGROUND

Richmond and Optican studied pattern discrimination in a primate visual area, the inferior temporal (IT) cortex. They addressed the question of temporal coding in IT in a well-known series of papers in the *Journal of Neurophysiology* (Optican and Richmond, 1987; Richmond and Optican, 1987; Richmond et al., 1987). They found that the firing rate of IT neurons modulated in response to the presentation of one of 64 two-dimensional visual stimuli. They also saw evidence of temporal modulation that was not captured by the firing rate.

To quantify the relevance of this temporal modulation, Richmond and Optican converted the raster plot for each trial into a spike density function (defined later in Section 15.2.2). They then computed principal components (PCs) of these functions. They computed the mutual information between the stimulus and either the firing rates (rate code) or the first three PCs (temporal code). Their results indicated that the temporal code carried on average twice the information as the rate code.

15.2.1. Motor Cortical Data

In this chapter you will use a similar approach to examine encoding of movement direction in the primate motor cortex. You have already seen that motor neurons modulate their firing rate systematically depending on the direction of motion during a center-out task and that this modulation can be fit to a cosine-tuning curve. However, this analysis computed the firing rate over a coarse time bin (1 second). A tuning curve might predict a firing rate of 30 Hz for a preferred stimulus, but there are a lot of ways to arrange 30 spikes (each lasting 1–2 ms) over a 1-second time period.

When you use a rate code, you are implicitly assuming that there is no additional information contained in the relative timing of the spikes. This does not mean you assume there is no temporal variation. Instead, a rate code assumes this temporal variation is uninformative about the stimulus. For example, when you calculated a peri-stimulus time histogram (PSTH) centered on movement time in the preceding chapter, you saw the firing rate ramp up slowly 500 ms before movement initiation and ramp back down to baseline by 500 ms later. If each direction elicits this same temporal response scaled up or down, then the coarse rate code is appropriate. However, if the temporal response varies systematically across movement directions, then the rate code will ignore potentially useful information.

Figure 15.1 contains the raster plots (left) and PSTHs (right) for a unit from electrode #19 of the motor cortical dataset. The spike times are relative to the beginning of the instructed delay, where the peripheral target is visible but the subject is still holding on the center target. The responses to a preferred stimulus (135° or 180°) are similar; both show an increase in the firing rate about 500 ms after the target appears. However, the responses to an anti-preferred stimulus (0° or 315°) are different; both show a transient increase in the firing rate

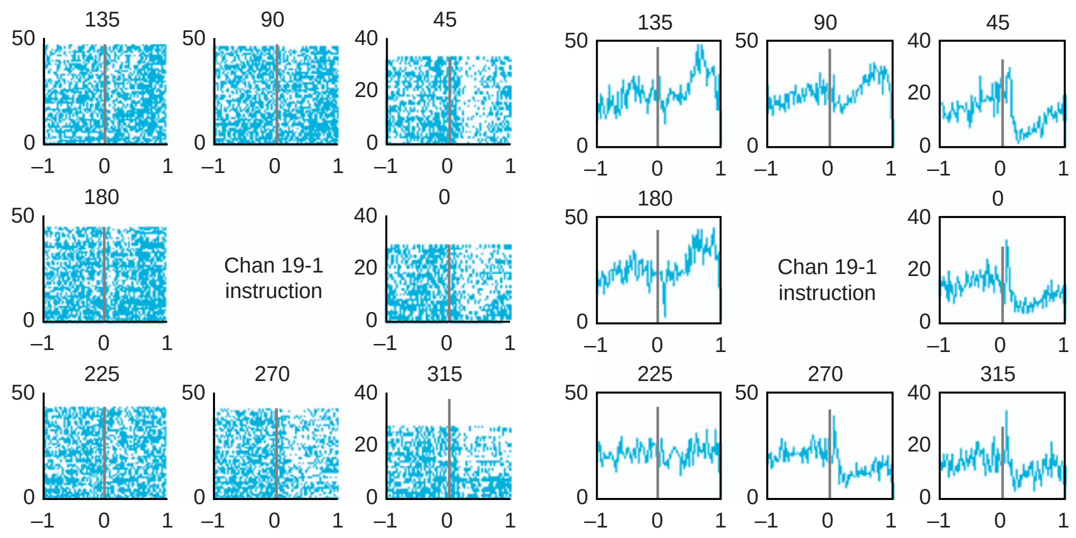


FIGURE 15.1 *Left*: A raster plot of the sample neuron. The x-axis is time in seconds, the y-axis is the trial number, and the title reflects the movement direction in degrees. *Right:* A peri-stimulus time histogram for the same neuron. Here, the y-axis reflects the number of spikes in a 10 ms bin across all trials.

in the first 100 ms followed by a marked depression for the following 900 ms. A rate code is unlikely to capture all the information in these responses.

How do you quantify such temporal information? Principal components analysis might work. However, you first need to think about how to format the spike times. Binning the data seems natural, but choosing the proper bin size can be tricky. If the bin is too small (1 ms), then you may make your potential response space huge (2^{1000} possible responses) compared to the number of trials. If the bin is too large (1 second), then you lose potentially useful temporal variations within the time bin. The best bin size is close to the order of the temporal dynamics you are interested in. If you observed consistent variations on a 50–100 ms time scale, a 50–100 ms time bin would probably work.

15.2.2. Spike Density Functions

Another problem with binning spike times is that binning can introduce artifacts into the data. Suppose a response to a stimulus always consists of a single spike around time *t*, and the variability from trial to trial follows a Gaussian distribution around *t*. If *t* sits right on an edge between two bins, then sometimes it will be counted in the first bin, and other times it will be counted in the second bin. This produces the illusion of a bimodal response when, in fact, there is only a single response.

An elegant solution to this problem is to convert each raster plot into a continuous spike density function. You first bin the spike times at a fine time resolution (1 ms) so that each bin has a 0 or 1. You then convolve this data with another function, called the *kernel*. The kernel captures how precise you think the spike times are: a wide kernel implies high variability, whereas a narrow kernel implies high precision.

You explored 2D convolution in Chapter 10; the 1D convolution function in the MATLAB® software is **conv**. Pay attention to the length of the resulting vector because **conv(*a*,*b*)** results in a function whose length is **length(*a*)+length(*b*)-1**. Suppose the kernel is a Gaussian function with a standard deviation of 15 ms. This is equivalent to putting a confidence interval on the spike times. This kernel means you believe that a neuron that wants to fire at 0 ms will actually fire at between –30 and 30 ms 95% of the time. To use this kernel in MATLAB, you need to evaluate the Gaussian over a range of time values (every 1 ms from –45 ms to 45 ms). If you convolve this function with 1 second of spike data (binned every 1 ms), the resulting vector will contain 1090 values. The corresponding time axis is –45 ms to 1045 ms. If you don't want values outside the 1-second time period of interest, you just select the middle section. This is shown in the following code, where the vector *binned* contains the binned spike data and the spike density function is stored in vector *s*:

```
sigma = .015;                      %Standard deviation of the kernel = 15 ms
edges=[-3*sigma:.001:3*sigma];     %Time ranges form -3*st. dev. to 3*st. dev.
kernel = normpdf(edges,0,sigma);   %Evaluate the Gaussian kernel
kernel = kernel*.001;              %Multiply by bin width so the probabilities sum to 1
s=conv(binned,kernel);             %Convolve spike data with the kernel
center = ceil(length(edges)/2);    %Find the index of the kernel center
s=s(center:1000+center-1);         %Trim out the relevant portion of the spike density
```

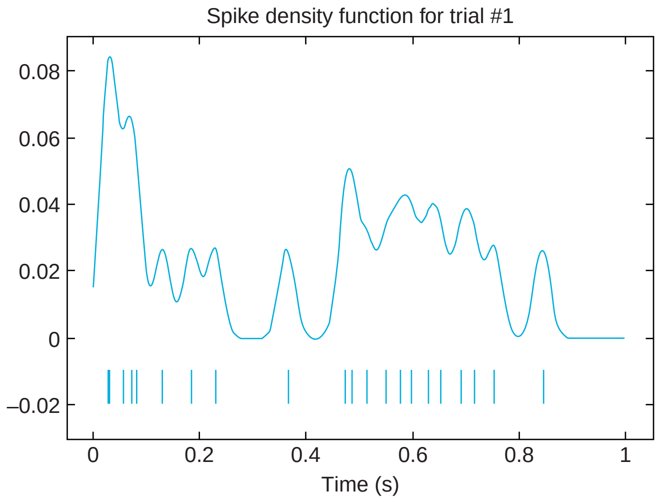


FIGURE 15.2 *Top:* An example of a spike density function using a Gaussian kernel with a standard deviation of 15 ms. *Bottom:* the original raster plot.

An example of a spike density function is shown in Figure 15.2, along with the original raster plot.

Once you compute spike density functions for each neuron, you can compute the principal components of the spike density functions. However, to avoid computing a 1000×1000 dimensional covariance matrix, you should first sample the spike density functions every 10 or 20 ms and then apply principal component analysis.

15.2.3. Joint, Marginal, and Conditional Distributions

The goal here is to compute the amount of information contained in a firing rate code compared to a temporal code (as captured by principal components). However, before defining "information" precisely, we need to review the concept of joint, marginal, and conditional probability distributions. For a discrete variable (which is all we will consider here), a *probability distribution* is simply a function that assigns a probability between 0 and 1 to all possible outcomes such that all the probabilities sum to 1. A *joint probability distribution* is the same, except it involves more than one variable and thus assigns probabilities to combinations of variables.

In this case, you have a stimulus S, which can take on one of eight discrete values. Suppose you divide the firing rates R of the sample unit into three bins (low, medium, and high firing rate). If you count how many times each of 24 possible combinations shows up in the data, you end up with something like Table 15.1.

From Table 15.1, it is clear that there is a relationship between the firing rate and stimulus direction. A high firing rate corresponds to a stimulus of 3, 4, or 5, while a low firing rate corresponds to a stimulus of 7, 8, 1, or 2. Remember that $S = 1$ corresponds to a movement direction of 0°, $S = 2$ to 45°, and so on. You can determine the observed joint

TABLE 15.1 Observed Counts of Stimulus-Response Pairs for the Sample Neuron

	$S = 1$	**$S = 2$**	**$S = 3$**	**$S = 4$**	**$S = 5$**	**$S = 6$**	**$S = 7$**	**$S = 8$**
$R<20$	21	28	2	1	1	8	33	17
$20\leq R<30$	7	5	23	20	17	29	9	6
$R\geq 30$	1	0	21	26	27	7	1	5

probability distribution $P(S,R)$ simply by dividing each count by the total number of counts (here, this is 315). In addition, you can compute the marginal probability distributions, which are the probability distributions of one variable computed by summing the joint probability distribution over the other variable:

$$P(S) = \sum_R P(s,r) \quad \text{and} \quad P(R) = \sum_S P(s,r) \tag{15.1}$$

Note that in these equations, S and R refer to all possible stimuli or responses, respectively, and that s and r refer to a particular stimulus or response.

Table 15.2 shows the values of $P(S,R)$. The marginal distributions are listed in the last row and far right column because they are the sum taken across rows and across columns of the joint distribution.

The last concept we need to address is the *conditional distribution*. The distribution of the response R given knowledge of the stimulus S is written $P(R \mid S)$ and is defined as $P(R \mid S) = P(S,R)/P(S)$. Likewise, the conditional distribution of S given R is $P(S \mid R) = P(S,R)/P(R)$. As an example, what is the probability distribution of firing rates given a movement direction of 0° ($S = 1$)? To find the answer, simply take the first column of values in Table 15.2, $P(S = 1, R)$, and divide by $P(S = 1)$ to get $P(R \mid S = 1)$. Hence, the probably of a low firing rate given $S = 1$ is $0.067/0.092 = 0.73$ and so on.

15.2.4. Information Theory

The foundation of information theory was laid in a 1948 paper by Shannon titled, "A Mathematical Theory of Communication." Shannon was interested in how much

TABLE 15.2 The Joint Probability Distribution $P(S,R)$ and the Marginal Distributions $P(R)$ and $P(S)$ for the Sample Neuron

$P(S,R)$	**$S = 1$**	**$S = 2$**	**$S = 3$**	**$S = 4$**	**$S = 5$**	**$S = 6$**	**$S = 7$**	**$S = 8$**	**$P(R)$**
$R<20$	0.067	0.089	0.006	0.003	0.003	0.025	0.105	0.054	0.352
$20\leq R<30$	0.022	0.016	0.073	0.063	0.054	0.092	0.029	0.019	0.368
$R\geq 30$	0.003	0.000	0.067	0.083	0.086	0.022	0.003	0.016	0.279
$P(S)$	0.092	0.105	0.146	0.149	0.143	0.140	0.137	0.089	

information a given communication channel could transmit. In neuroscience, you are interested in how much information the neuron's response can communicate about the experimental stimulus.

Information theory is based on a measure of uncertainty known as *entropy* (designated "H"). For example, the entropy of the stimulus S is written $H(S)$ and is defined as follows:

$$H(S) = -\sum_S P(s)\log_2 P(s). \tag{15.2}$$

The subscript S underneath the summation simply means to sum over all possible stimuli $S = [1, 2 \ldots 8]$. This expression is called "entropy" because it is similar to the definition of entropy in thermodynamics. Thus, the preceding expression is sometimes referred to as "Shannon entropy." The entropy of the stimulus can be intuitively understood as "how long of a message (in bits) do I need to convey the value of the stimulus?" For example, suppose the center-out task had only two peripheral targets ("left" and "right"), which appeared with an equal probability. It would take only one bit (a 0 or a 1) to convey which target appeared; hence, you would expect the entropy of this stimulus to be 1 bit. That is what the preceding expression gives you, as $P(S) = 0.5$ and $\log_2(0.5) = -1$. The center-out stimulus in the dataset can take on eight possible values with equal probability, so you expect its entropy to be 3 bits. However, the entropy of the observed stimuli will actually be slightly less than 3 bits because the observed probabilities are not exactly the same.

Next, you want to measure the entropy of the stimulus given the response, $H(S \mid R)$. For one particular stimulus, the entropy is defined similarly to the previous equation:

$$H(S|r) = -\sum_S P(s|r)\log_2 P(s|r) \tag{15.3}$$

To get the entropy $H(S \mid R)$, you just average over all possible responses:

$$H(S|R) = -\sum_R \sum_S P(r)P(s|r)\log_2 P(s|r) \tag{15.4}$$

Now you can define the information that the response contains about the stimulus. This is known as *mutual information* (designated I), and it is the difference between the two entropy values just defined:

$$I(R;S) = H(S) - H(S|R) = \sum_R \sum_S P(r)P(s|r)\log_2\left(\frac{P(s|r)}{P(s)}\right) \tag{15.5}$$

Why does this make sense? Imagine you divide the response into eight bins and that each stimulus is perfectly paired with one response. In this case, the entropy $H(S \mid R)$ would be 0 bits, because given the response, there is no uncertainty about what the stimulus was. You already decided the $H(S)$ was theoretically 3 bits, so the mutual information $I(R;S)$ would be 3 bits – 0 bits = 3 bits. This confirms that the response has perfect information about the stimulus.

Suppose instead that you divide the response into two bins, and that one bin corresponds to stimuli 1–4 and the other bin corresponds to stimuli 5–8. Each bin has four equally likely choices, so the entropy $H(R \mid S)$ will be 2 bits. Now the mutual information is $I(R;S)$ = 3 bits – 2 bits = 1 bit. This means that response allows you to reduce the uncertainty about the stimulus by a factor of 2, which makes sense because the response divides the stimuli into two equally likely groups. This also emphasizes that the choice of bins affects the value of the mutual information.

Note that you can use the definition of conditional probability to rearrange the expression for mutual information. The following version is easier to use with the table of joint and marginal probabilities computed earlier. Mutual information can also be defined as follows:

$$I(R;S) = \sum_{R}\sum_{S} P(s,r)\log_2\left(\frac{P(s,r)}{P(s)P(r)}\right) \tag{15.6}$$

Applying this equation to the joint distribution of the sample neuron gives a mutual information of 0.50 bits for a rate code.

15.2.5. Understanding Bias

Now try estimating the mutual information of an uninformative neuron. "Uninformative" means that the firing rate probabilities are independent of the stimulus probabilities. By the definition of independence, the joint probability distribution of two independent variables is the product of their marginal distributions $P(R,S) = P(S)P(R)$. If you substitute this into the previous expression for mutual information, you will see the quantity inside the logarithm is always 1. Since $\log_2(1)=0$, this means the mutual information of two independent variables is also 0.

To make things easy, assume that each of the three responses is equally likely and that each of the eight stimuli is equally likely. Thus, $P(R) = 1/3$ and $P(S) = 1/8$, and each value of the joint probability distribution is the same: $P(S,R) = 1/24$. The mutual information of this distribution is still 0 bits.

However, even if this is the true probability distribution, the observed probabilities would likely be different. You can simulate the values of observed probabilities with the following code. Here, you will simulate 24 random trials, so the expected count for each cell is 1:

```
edges=[0:24]/24;            %Bin edges for each table entry
data=rand(24,1);            %Generate 24 random values between 0 and 1
count = histc(data,edges);  %Count how many fall in the bin edges
count=count(1:24);          %Ignore the last value (counts values equal to 1).
count=reshape(count,3,8);   %Reformat the table.
```

This might lead to the counts shown in Table 15.3.

TABLE 15.3 The Observed Counts of the Stimulus-Response Pairs for 24 Random Trials of an Uninformative Neuron

	S = 1	*S* = 2	*S* = 3	*S* = 4	*S* = 5	*S* = 6	*S* = 7	*S* = 8
$R<20$	0	2	1	1	1	2	1	0
$20\leq R<30$	1	0	1	0	0	1	0	2
$R\geq 30$	1	2	1	2	0	2	3	0

If you calculate the mutual information of the associated joint distribution (divide each count by 24 for the joint distribution), you get 0.50 bits, which is much higher than the 0.00 bits you expect. In fact, this is the same information as the sample neuron found previously. How can you now trust this earlier calculation?

Calculating mutual information directly from the observed probability distribution (as done here) leads to a biased estimate. A *biased* estimate is one that will not equal the true value, even if the estimate is averaged over many repetitions. The estimate of mutual information becomes unbiased only when you have infinite data. Such datasets are hard to come by.

However, all hope is not lost, because you can reduce the size of this bias. To start with, note that the number of trials is the same as the number of bins in the joint distribution. The sample data had 315 trials, which should reduce the chance of spurious counts. Repeating the previous exercise with 315 random trials might give the counts shown in Table 15.4.

The mutual information calculating from this table's joint distribution is now just 0.03 bits, which is much closer to the expectation (0 bits). This gives you more confidence in the 0.50 bits you estimated for the sample neuron. However, these numbers are each generated from a single random experiment. In the exercises you will see that repeated experiments confirm this trend: increasing the number of trials does decrease of the bias of the estimate of mutual information.

Note that the relevant parameter is actually the number of trials per bin. This means that if you have 315 trials but you also have 315 bins, you will still have a significant bias. Therefore, you must choose the number of bins carefully. It seems that large bins throw away information (shouldn't a spike count of 4 be treated differently than 16?), but smaller bins introduce a larger bias.

15.2.6. Shuffle Correction

These simulations are similar to a simple method (called *shuffle correction*) that corrects for the bias in the mutual information estimate. Suppose you store the stimulus values in a vector **dir**. You are interested in determining what the estimated mutual information would be if the firing rates were independent of the stimulus values. To do this, you randomly rearrange (or "shuffle") the stimulus vector and then compute a new joint distribution and estimate mutual information from that. If you repeat this operation many times, you can derive a "null distribution" of mutual information estimates of a firing rate which carries no information. Thus, you can conclude that any neuron whose mutual information value that is significantly different from this null distribution is informative. You can also calculate a "shuffle corrected"

TABLE 15.4 The Observed Counts of the Stimulus-Response Pairs for 315 Random Trials of an Uninformative Neuron

	$S = 1$	$S = 2$	$S = 3$	$S = 4$	$S = 5$	$S = 6$	$S = 7$	$S = 8$
$R<20$	14	25	11	16	15	21	15	14
$20\leq R<30$	12	11	11	16	12	14	9	12
$R\geq 30$	11	11	16	6	8	14	7	14

mutual information estimate by subtracting the mean of this null distribution. The following code shows how you can shuffle the stimulus values in MATLAB:

```
ind=randperm(315);  %Randomly arrange numbers 1 to 315 in vector "ind"
dirSh=dir(ind);     %Use "ind" to randomly shuffle the vector of stimuli "dir"
```

Now you can compute a table counting combinations of the original firing rate and the shuffled stimulus. One example might be the one shown in Table 15.5.

The mutual information of the associated joint distribution is 0.03 bits. Repeating the shuffling 30 times gives a mean information of 0.03 bits and a standard deviation of 0.01 bits for the null distribution. Hence, the "shuffle corrected" mutual information of the original neuron is 0.50 bits − 0.03 bits = 0.47 bits. Thus, you can be confident that a rate code contains significant information about the stimulus direction.

Bias correction is particularly important when you are comparing mutual information of joint distributions that have different numbers of bins. In the project for this chapter, you will compare a rate code to a temporal code. Comparing a rate code to the first principal component is straightforward because you could use the same number of bins for each variable. However, if you use the n bins to look at the first principal components, you would have to use n^2 bins to look at the first two principal components together using the same bin widths. As you know, increasing the number of bins increases the bias. If you compared uncorrected estimates, it would be easy to assume the second principal component provides additional information when it actually doesn't.

It should also be noted that accurate estimation of information measures is an active research field and that shuffle correction is perhaps the simplest of available techniques. Refer to Panzeri et al. (2007) for a review of current bias correction techniques and to Hatsopoulos et al. (1998) for another example of the use of shuffle correction.

TABLE 15.5 The Observed Counts of the Stimulus-Response Pairs for the "Shuffled" Version of the Sample Neuron

	$S = 1$	$S = 2$	$S = 3$	$S = 4$	$S = 5$	$S = 6$	$S = 7$	$S = 8$
$R<20$	8	10	12	20	18	16	16	11
$20\leq R<30$	13	12	22	14	17	15	11	12
$R\geq 30$	8	11	12	13	10	13	16	5

15.3. EXERCISES

Exercise 15.1: Compute the entropy of the observed values of the stimulus.

Exercise 15.2: Run 30 simulations each of the observed joint distribution of the uninformative neuron (8 stimuli, 3 responses) with n trials, where $n = 25, 50, 100, 200, 400, 800$. Plot the mean and standard deviation of the bias as a function of the number of trials.

Exercise 15.3: Repeat Exercise 15.2 but with 6 and 12 response bins. Plot the mean and standard deviation of the bias as a function of the number of trials.

Exercise 15.4: Combine data from Exercises 15.2 and 15.3 and plot the mean bias as a function of the number of trials per bin.

15.4. PROJECT

Choose at least five active neurons from the dataset (including the unit from electrode #19) available at the companion website to analyze. Convert each raster plot into a spike density function. Report the details of the kernel you used. Calculate principal components (PCs) of the spike density functions. Calculate the shuffle-correction mutual information between the stimuli and, in turn, the firing rate, first PC score, and first and second PC score considered together.

In addition, answer the following questions:

1. Is there evidence for a temporal code?
2. How similar are the first few PCs (not the scores) calculated for the different neurons? What do you think they represent?
3. How does a temporal code that depends only on the first PC differ from one which depends on two or more PCs?

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

conv
reshape

CHAPTER

16

Neural Decoding Part I: Discrete Variables

16.1. GOALS OF THIS CHAPTER

Thus far, you have seen how the MATLAB® software can be used to solve frequently encountered problems in neuroscience. However, many other software packages also address these problems. The advantage of learning MATLAB (and computer programming in general) is flexibility. MATLAB allows you to attack new problems that do not have prepackaged solutions. With that in mind, this chapter will introduce an open-ended approach toward solving the problem of neural decoding. Specifically, this chapter will address how to predict the upcoming direction of movement from a population of neuronal signals recorded from motor areas of a macaque monkey.

Note that this chapter assumes completion of Chapter 13, “Neural Data Analysis: Encoding,” as it makes use of the preferred directions calculated in the exercises.

16.2. BACKGROUND

What is *neural decoding*? Simply put, it is a mathematical mapping from the brain activity to the outside world. In the sensory domain, the outside world consists of the received visual, auditory, or other sensory information. In the motor domain, the outside world consists of the state of the skeletomuscular system. This is the inverse of *neural encoding*, which maps the outside world to brain activity. For example, in Chapter 13, you looked at how a cosine tuning curve specifies how a neuron modulates its firing rate depending on the upcoming direction of movement. In contrast, estimating this movement direction from one (or many) observed firing rate(s) is an example of neural decoding. Because signals about motor intention precede movement, decoding can be thought of as “mind reading.” Neuroscientists seek to predict an action as soon as it is intended, before it ever takes place.

Neural decoding can also be thought of as pattern recognition. A set of neuronal spike times represents a pattern, and the goal of the decoder is to figure out which stimuli or movements are associated with which patterns. This is a common problem in science. Doctors perform pattern recognition when they produce a diagnosis from a collection of physical and physiological findings. For example, an electrocardiogram (EKG) trace contains a repeated, stereotypical pattern that corresponds to a single heart beat. Each part of the trace corresponds to de- and repolarization of a different part of the heart. Therefore, doctors can use deviations from the normal EKG as clues about underlying abnormalities. An elevation in one part of the trace (ST elevation) is used to help diagnose heart attacks. This is pattern recognition.

Interpreting a raster plot is not quite that straightforward (neither is interpreting EKGs, but that's another matter). Figure 16.1 shows 10 raster plots and a peri-stimulus time histogram for a motor neuron. Each neuron's spike times is centered on the start time of repeated movements in the same direction. There is a clear pattern. On every trial, there is a transient increase in the neuron firing rate starting a few hundred milliseconds before the movement starts. However, this is only an approximate relationship. If you look at each raster individually, it is not clear exactly when the movement begins. In addition, each raster plot is different. This means you can't simply perform pattern recognition by using a "look-up table" because it is unlikely that a neuron's response will ever exactly repeat itself.

In this chapter we will implement different strategies for predicting the direction of an upcoming movement based on the firing rates of a population of neurons. This is relevant to two distinct goals of neuroscience. First, neuroscientists would like to understand the brain on a functional level. Neuroscientists ask, "What is the brain doing and how is it doing that?" Second, neuroscientists are interested in using neuronal signals to do something useful. Neural prosthetics seek to do this. For example, cochlear implants convert

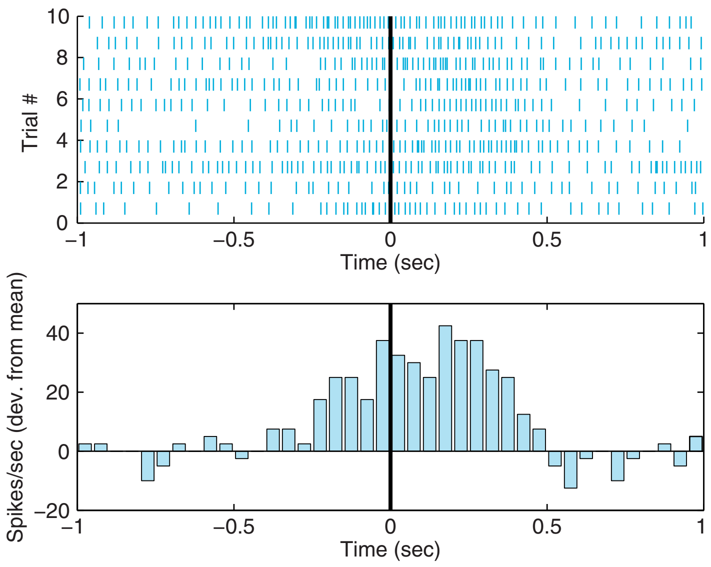


FIGURE 16.1 *Top:* A raster plot of 10 trials of a center-out task. *Bottom:* A peri-stimulus time histogram of the same data. The start of movement occurs at *time* = 0 seconds.

sound into digital signals used to stimulate the auditory nerve, thus restoring speech perception in the deaf (Papsin and Gordon, 2007). A decoding strategy introduced in the next chapter (the linear filter) was used in a neuroprosthetic that allowed a human with tetraplegia to control a computer cursor and other devices (Hochberg et al., 2006).

It is important to note, however, that decoding to understand how the brain works is different from decoding for control. You have seen that neuronal activity in the motor cortex is directionally tuned, but that is not the same as saying these neurons encode direction. Properly interpreting what is being encoded requires more experiments than what we have described thus far. In the canonical center-out experiment, the posture is the same for all eight directions. Thus, instead of encoding direction, the neuron might be encoding the specific sequence of muscle activations, the desired end posture, or the spatial location of the target. The adage "correlation does not imply causation" applies here.

One area of debate in neuroscience is, "At what time scale should we look for information?" *Rate encoding* holds that the firing rate calculated over some broad time span (usually 100s of milliseconds) contains all the necessary information. *Temporal coding* holds that additional information is available at smaller time scales. At the extreme, you could use a 1 ms time bin where each bin either has a 0 (no spike) or a 1 (spike). This approach likely contains more information about the stimulus than a coarse firing rate. However, it also greatly increases the dimensionality of the potential responses. Instead of one discrete variable (the firing rate over 1 second), which might reasonably take a few dozen values, a 1 ms time bin gives a 1000×1 vector of binary variables with as many as 2^{1000} possible values, which is a number larger than the estimated number of atoms in the universe. Thus, for computational simplicity, you can start by assuming a coarse rate code.

16.2.1. Population Vector

In Chapter 13, "Neural Data Analysis: Encoding," we introduced the concept of directional tuning of motor cortical neurons. This is an encoding model that translates an upcoming direction of movement into a neuronal firing rate. If you now want to decode direction from a firing rate, you are faced with two problems. First, since a cosine-tuning curve is symmetric, most firing rates are ambiguous because they could be associated with two movement directions. Second, the firing rate signal is very noisy. This is due to both intrinsic neuronal noise as well as measurement noise introduced by the equipment. How, then, can you decode movement direction?

The solution is averaging over a population of neurons, which decreases the effects of noise and allows disambiguation of the movement direction. The "population vector" algorithm introduced by Georgopoulos and colleagues is an intuitive way to use cosine-tuning information from a population of neurons to decode movement direction (1986). In Chapter 13, you determined the preferred direction of each neuron using information from all trials.

Having done this, proceed as follows:

1. Assume that each neuron "votes" for its preferred movement direction. Specifically, each neuron is going to contribute a "response vector" that is aligned with its preferred direction.

2. The magnitude (or length) of each neuron's response vector is determined by the neural activity of the neuron during each trial. This is the weight given to each neuron's vote. For now, assume the weight is simply the firing rate during the hold period.
3. Sum all the response vectors from all neurons to arrive at the population vector for this trial. The direction of this population vector corresponds to the predicted direction.

This can be expressed as a formula,

$$\vec{P} = \sum_{i=1}^{n} w_i \vec{C_i} \tag{16.1}$$

where for n neurons, P is the population vector, w_i is the weight given to each vote, and C_i is a unit vector pointing in the i^{th} neuron's preferred direction. The arrows above P and C_i indicate that these quantities are vectors. Recall that if you represent vectors with Cartesian coordinates, you can sum the X and Y components separately. Thus, if θ_i is the i^{th} neuron's preferred direction in radians, then the population vector can be broken down into its X and Y components:

$$P_X = \sum_{i=1}^{n} w_i \cos(\theta_i) \quad P_Y = \sum_{i=1}^{n} w_i \sin(\theta_i) \tag{16.2}$$

The perspective has changed. Previously, you considered all trials to determine the preferred direction of a given neuron. Now, you consider the neural activity of all cells in a given trial to determine the combined response of the population of neurons: the population vector. The population vector points toward the upcoming movement direction.

The population vector is useful because it is easy to implement and intuitive to understand. It is based on the theory that motor cortex neurons fire to produce muscle forces, which, given a certain posture, act in the neuron's preferred direction. However, the population vector is limited because a number of conditions must be met for the method to perform well (Georgopoulos, 1986). For example, the neuron's tuning curves must actually follow a cosine or at least be radially symmetric around the preferred direction. Also, the preferred directions must be uniformly distributed.

16.2.2. Maximum Likelihood

You can develop a more general decoding algorithm by relaxing some of the assumptions made by the population vector. One way to do this is to use statistical methods. You assume that a neuron modulates its firing rate in response to the upcoming movement direction. However, you do not assume exactly how (which a cosine tuning curve does). You assume that the neuron's target firing rate will be corrupted by noise, so that for a given direction you will observe a distribution of firing rates rather than a single firing rate. If you assume the form of this distribution, you can use standard statistical methods to estimate its parameters. For example, if you thought the distribution of firing rates was

Gaussian, you could characterize it fully by computing the mean and standard deviation of all the firing rates for trials moving in one direction.

Once you have estimated the parameters, you can calculate the probability of any firing rate giving a certain direction. This is a conditional probability. We reviewed this in Chapter 15, "Information Theory." Briefly, the conditional probability of event A given event B is denoted as $P(A \mid B)$. It is defined as the joint probability A and B divided by the probability of B:

$$P(A|B) = \frac{P(A,B)}{P(B)} \tag{16.3}$$

Intuitively, this can be thought of as the probability of A taking into account some piece of information (that B has happened). Here, you are interested in the conditional probability of a firing rate R given the direction d: $P(R \mid d)$. For one neuron, the maximum likelihood approach is to look at the firing rate and select the direction for which this firing rate is most likely. As an example, consider a simplified center-out experiment in which targets are presented at equal probabilities to the left or right, and you are recording from a neuron that prefers movement to the right. Say this neuron fires at 25 Hz (+/− 5 Hz) before right targets and fires at 10 Hz (+/− 3 Hz) before left targets. If you assume these firing rates follow Gaussian distributions, then the maximum likelihood algorithm would predict a right target for a firing rate $\geq$ 17 Hz and a left target for firing rates $\leq$ 16 Hz (see Figure 16.2).

If there is more than one neuron, the situation is more complicated. You make the simplifying assumption that the neuron's firing rates are independent. At first, this seems at odds with our understanding of the brain: aren't connections between neurons the whole point? However, most neurons in the sample are separated by large distances (for neurons), and you may attempt to relax this assumption in the exercises. Independence means you can express the probability of a set of firing rates as the product of the probabilities for each individual firing rate. This calculation is performed when you say the chance of

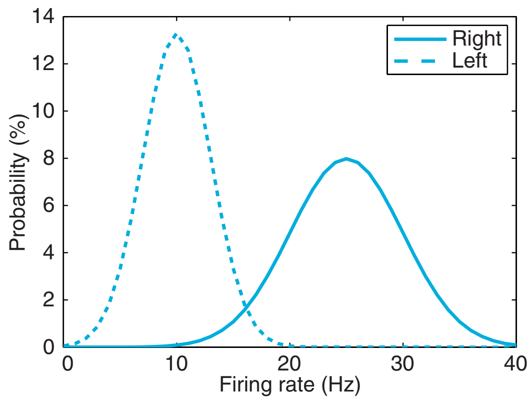


FIGURE 16.2 Firing rate distributions of a hypothetical, rightward-preferring neuron.

flipping four heads in a row is $1/16 = (1/2)^4$. The probability of a set of firing rates $R = [r_1, r_2, \ldots, r_n]$ for a given direction d can be expressed as follows:

$$P(R|d) = \prod_{i=1}^{n} P(r_i|d) \tag{16.4}$$

The maximum likelihood approach predicts the direction associated with the highest likelihood $P(R \mid d)$. However, when you're implementing this in MATLAB, it is more convenient to pick the direction that maximizes the log-likelihood: $\log[P(R \mid d)]$. Because the natural logarithm is a monotonically increasing function, the choice of direction that maximizes the log-likelihood will also maximize the likelihood. This approach avoids calculating the product of very small numbers in MATLAB, which, due to numerical precision constraints, quickly becomes inaccurate. Instead, you sum negative numbers (since the probabilities are less than 1). The log-likelihood can be expressed as follows:

$$\log[P(R|d)] = \sum_{i=1}^{n} \log[P(R_i|d)] \tag{16.5}$$

One point to note is that the maximum likelihood approach relies on an experimental trick: you set the prior probabilities of all directions to be equal. This is important, because if you are decoding direction, you want to maximize the conditional probability of the direction given the firing rates, $P(d \mid r)$. This algorithm is maximizing the reverse: the conditional probability of the firing rates given the direction, $P(r \mid d)$. This simplification is valid only if all directions are equally probable. If they are not, you will need to calculate $P(d \mid r)$ using Bayes' rule (refer to any statistics textbook for information on Bayes' rule).

16.2.3. Data

Here, you will use the dataset from Chapter 13, "Neural Data Analysis: Encoding." However, there is a second dataset for this chapter, available on the companion website. The first dataset will be used to train the decoding algorithms, meaning these data are used to estimate the parameters of the model (such as preferred direction). The second dataset will be used to test the algorithm built using the first dataset. It is important not to test a prediction algorithm using the same data you trained with. Otherwise, the optimal prediction would be "the exact same thing is going to happen." Testing on novel data helps ensures that the model does not overfit the data.

To compare the population vector method to the maximum likelihood method, you will need to bin the population vector to force to assume one of the eight discrete directions. You can use the function **histc** in MATLAB to do this, though you should remember that the 0-degree direction will correspond to two bins: 0 to 22.5 degrees and 337.5 to 360 degrees.

You also need to define an accuracy metric for the predictions made. The percentage correct is the easiest to compute. However, the mean squared error may be more appropriate, as it penalizes large errors more than small errors.

16.3. EXERCISES

Exercise 16.1: Implementation

1. Implement a population vector decoder. Use the preferred directions determined for the training dataset. Predict the movement direction of the test dataset by using the firing rates during the hold period as weights. The hold period start times are stored in the variable *instruction.* The hold period ends 1 second after the start time. Bin the predicted direction to convert it to one of the eight discrete movement directions.
2. Implement a maximum likelihood decoder. Assume a Poisson firing rate model and independent firing rates. Determine the mean firing rate for each neuron and each direction for the training dataset. Use the function **poisspdf** to determine the likelihood of each direction. Pick the direction that maximizes the log-likelihood of all firing rates for a given trial.
3. Compare the accuracy of these two decoding methods, using a percent accuracy or mean squared error.

Exercise 16.2: Variations

1. The population vector methods make assumptions about the data: that neurons were cosine tuned and that preferred directions are uniformly distributed. Are these assumptions valid? Provide evidence for your answer.
2. Instead of weighting the response vectors using the firing rate, weight using the change in firing rate from baseline. Use the mean firing rate across all directions to determine the baseline firing rate. Does this affect the decoding accuracy? Why might this be?
3. Use a Gaussian firing rate model for the maximum likelihood decoder. The likelihood can be determined with the function **normpdf** and the mean and standard deviation for each neuron and each direction. Does this affect the decoding accuracy? Why might this be?

16.4. PROJECT

Create a new decoder by either modifying the algorithms introduced here or by developing your own ideas. Report the accuracy of the new decoder as compared to population vector or maximum likelihood methods. Here are some suggested approaches:

- *Easy*: Change the specific implementation of one of the decoders. For the population vector, create a new function to determine the weights from the firing rates. For maximum likelihood, use a different probability distribution as a firing rate model. Alternatively, try to include temporal coding by using principal component analysis or using smaller time bins.

- *Medium*: Transform the data to make it conform to the assumptions made by decoders. For the population vector, change the weighting scheme of the population vector algorithm to compensate for a nonuniform distribution of preferred directions. For maximum likelihood, try to compensate for correlations between neurons, or between a neuron's current firing rate and its past firing rates.
- *Hard*: Create a new decoding technique. For example, the algorithms introduced treat firing rates as independent. More information might be available if firing rates are pooled across neurons. You could then use a distance metric to classify a novel vector of firing rates that falls into one of eight clusters (for each direction of movement).

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

poisspdf
normpdf

CHAPTER

17

Neural Decoding Part II: Continuous Variables

17.1. GOALS OF THIS CHAPTER

The preceding chapter explored methods of decoding movement direction from neuronal signals. The movement direction was a discrete stimulus, taking on one of eight possible values. In this chapter you will look at how to decode a continuous, time-varying stimulus from neuronal signals. Specifically, this chapter will address how to decode the instantaneous hand position from a population of motor cortical neurons recorded from a macaque monkey. You will also see how information about how the hand position changes over time can be used to improve your decoding.

Note that this chapter assumes familiarity with Chapter 13, "Neural Data Analysis: Encoding," as well as Chapter 16, "Neural Decoding Part I: Discrete Variables."

17.2. BACKGROUND

In the preceding chapter, we discussed how neurons in the motor cortex carry information about upcoming movements. You were able to use this information to decode the direction of a movement made to one of eight targets. But what do you do if movement isn't so constrained as it is in the center-out task? Another common experimental paradigm in motor control literature can be described as the "pinball" task: a target appears somewhere in a 2D playing field, and as soon as the subject moves the cursor within this target, a new target appears at a different, randomly selected position. There are no hold times in this task, unlike the center-out task, so the hand is constantly moving. You are interested in decoding the trajectory of the hand, meaning you want to know *X*- and *Y*-positions of the hand at each time point. *X*- and *Y*-positions are examples of kinematic variables, meaning they describe the motion of the object (the hand), but not with the forces that generated the motion. Considering just the limb kinematics is easier than considering the full limb dynamics, which requires a model of how muscle forces and the physical properties of the limb interact to produce this motion.

What method can you use to decode these kinematic variables? You could just modify the algorithms you have already seen. Recall that the population vector has a magnitude as well as direction. If you assume this magnitude is proportional to the instantaneous speed, you could simply compute a population vector for each time bin and then add them tail to tip to create an estimate of the trajectory. This was, in fact, an early approach to the problem (Georgopoulos et al., 1988). Or you could divide the *X*- and *Y*-axes into bins and then use the maximum likelihood method to determine which grid square the hand is located in.

In this chapter we will take a different approach. We mentioned previously that a simple neuronal encoding model assumes the firing rate varies linearly with stimulus intensity. You can apply this to motor cortical neurons and assume they fire linearly with the instantaneous hand position. Since you know there is a time lag between motor cortical activation and hand movement at the periphery, you will assume that the relationship between firing rate is a function of the movement at some fixed time in the future.

Load the dataset for this chapter, available on the companion website (data courtesy of the Hatsopoulos laboratory). There are two main variables: **kin** stores the *X*- and *Y*-positions (sampled every 70 ms), and **rate** stores the number of spikes in each 70 ms bin. Now look at the relationship between just one kinematic variable (*Y*-position) and one neuron's firing rate. Notice how the indices are offset to create a vector of spike counts that lead the kinematics by two time bins:

```
>> y=kin(3:1002,2);     %except of y-position
>> x=rate(1:1000,27);   %neuron #27's spike count, lagged 140 ms
```

Plot the spike count of the *X*-axis and the kinematics on the *Y*-axis. Fit the data using **polyfit**. You should end up with something like Figure 17.1.

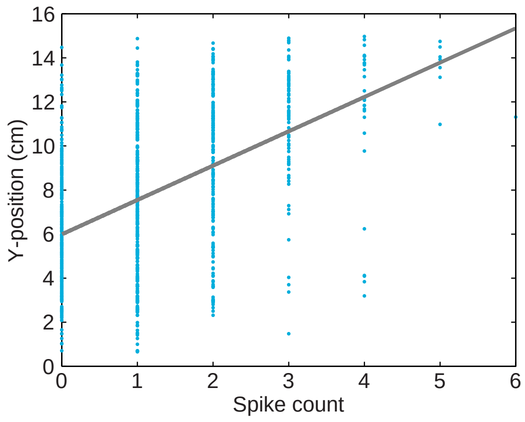


FIGURE 17.1 A linear fit of the relationship between *Y*-position and the lagged spike count.

17.2.1 Linear Filter

You now have a decoder relating Neuron #27's spike counts X to the instantaneous y-position Y, which takes the familiar linear form, $Y=mX+b$, where m and b are the coefficients returned by **polyfit**. But what exactly is the MATLAB® software doing when you run **polyfit**? It is determining the optimal linear regression that minimizes the squared residuals (which are the differences between the fitted data and the actual data). The optimal linear regression can be expressed analytically. If you express the linear relationship in matrix form as $Y = Xf$, where Y is the kinematics, X is the spike counts, and f is the decoding filter (the y-intercept b has disappeared, but you will add a column of 1s to X to account for this), then minimizing the squared residuals is the same as solving the following equation:

$$(X^T X)f = X^T Y. \tag{17.1}$$

You can use this to solve for the desired decoder:

$$f = (X^T X)^{-1} X^T Y \tag{17.2}$$

A matrix inverse can be computed in MATLAB with the function **inv**, and a transpose is denoted with an apostrophe. Compare this solution to the values you derived with **polyfit**:

```
p=polyfit(x,y,1)           %MATLAB linear regression
x=[x ones(length(s),1)];   %Add a row of 1's to allow y-intercept
f=inv(x'*x)*x'*y           %Analytical linear regression
```

The advantage of this approach is that you can easily add more neurons or more kinematics to the model: you just add more columns to X and Y. The following code shows an example of decoding using all of kinematics and all the neuronal firing rates lagged two time bins (140 ms):

```
yTrain=kin(3:3002,:);                    %Kinematic training data
yTest=kin(3003:3102,:);                  %Kinematic test data
xTrain=[rate(1:3000,:) ones(3000,1)];    %Training firing rates
xTest =[rate(3001:3100,:) ones(100,1)];  %Test firing rates
f=inv(xTrain'*xTrain)*xTrain'*yTrain;    %Create linear filter
yFit = xTest*f;                          %Reconstruct the test kinematics
```

The results, shown in Figure 17.2, look pretty good, especially for instantaneous Y-position.

This approach is known as the *linear filter*. The linear filter was used in 2002 to show that a macaque monkey could successfully control a computer cursor with neuronal signals (Serruya et al., 2002) and again in 2006 to show that a human with tetraplegia could control a variety of neural prosthetics with neuronal signals (Hochberg et al., 2006). Of course, the linear filter can be used on data recorded outside the motor cortex. For example, it has been used to decode visual information from retinal ganglion cells (Warland et al., 1997).

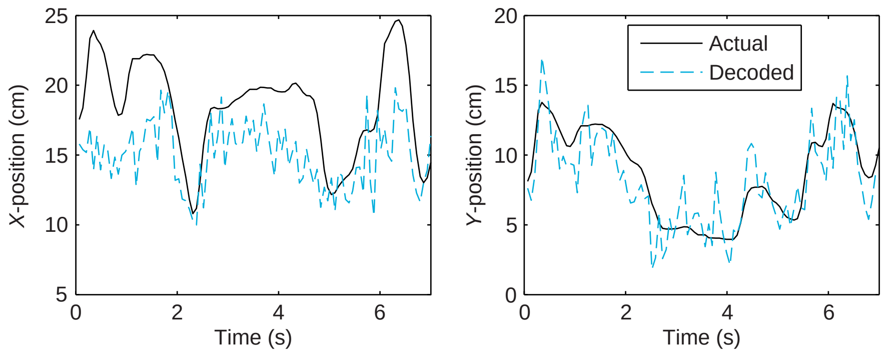


FIGURE 17.2 The actual raw and reconstructed (decoded) X- and Y-positions. Decoding was performed using a linear filter.

17.2.2 Particle Filter

The main advantages of the linear filter are that it is very fast and it is easy to calculate. However, it does have some limitations. For one, a linear encoding model can predict a negative spike count, which is impossible. We won't address that issue here. Another is that the linear filter treats each kinematic datapoint as independent. That is, it says that the hand position now is uncorrelated with its position 70 ms ago. This is not true. The hand is a physical object whose acceleration is a result of the forces acting on it. Building a realistic musculoskeletal model to capture these dynamics is difficult and computationally intensive. However, you can approximate such a model by adding a continuity constraint to your kinematics. This means that you assume the hand position at the next time point will be close to the position at the current time point. This prevents the position estimate from taking impossible values, such as predicting the hand can traverse the whole playing field in 70 ms.

You will formalize this continuity constraint in a *state equation*. Here, the "state" is simply the set of kinematic variables you are interested in, Y. The state equation contains a model of how you think the state evolves over time. One model would be a simple Gaussian random walk, meaning that the next position is equal to the current position plus some Gaussian noise. You write this as follows:

$$Y_{k+1} = Y_k + w, w \sim N(0, W) \tag{17.3}$$

The $\sim$ (tilde) just indicates that the w is a zero-mean, normally distributed random variable with covariance matrix W. A better model would be a linear model with Gaussian noise, expressed as

$$Y_{k+1} = Y_k a + w, w \sim N(0, W) \tag{17.4}$$

where a is a matrix describing the linear transformation. Suppose the state contains Y-position and its derivative, Y-velocity. The linear transformation might predict that the next position is the sum of the previous position and the previous velocity. The following code provides an illustration:

```
y1=[1 0.1]     %State vector: position = 1, velocity = 0.1
a=[1 0;1 1]    %Next position = prev. position + prev. velocity
               %Next velocity = prev. velocity
y2=y1*a        %Apply linear transform to prev. state
```

In addition to the state equation, you also need an *observation equation*. This is the same as an encoding model: it tells you what spike counts you are likely to observe given the current state. You can start by assuming linear encoding with zero-mean, Gaussian noise, such as

$$X_k = Y_k h + q_k, q \sim N(0, Q) \tag{17.5}$$

where X is the spike counts, Y is the kinematics, h is the linear encoding model, and q is a random noise term. Because these are linear relationships, you can solve for a and h just as you solved for the linear filter. However, X and Y will be interchanged because you are going in the opposite direction (encoding rather than decoding). The encoding matrix h can be expressed as

$$h = (Y^T Y)^{-1} Y^T X \tag{17.6}$$

The residuals of this fit are defined as the difference between the actual values and the fitted values: $X\text{-}Yh$. The covariance matrix is just the expected value (mean) of the squared residuals. Thus, the covariance matrix Q can be expressed as

$$Q = E[(X - Yh)^T (X - Yh)] \tag{17.7}$$

As shown in the following code sample, these expressions are easy to code in MATLAB. You can use the same approach to derive the state transformation matrix a and the state noise covariance W, though here you will be working with the kinematics (Y_{k+1}) and the kinematics lagged one time step (Y_k):

```
h=inv(yTrain'*yTrain)*yTrain'*xTrain;                %h = linear encoding matrix
Q=(xTrain-yTrain*h)'*(xTrain-yTrain*h)/length(xTrain);  %Q = noise covariance
```

The idea is that when you estimate the position of the next time point, you have two sources of information. First, you have an observation equation that specifies how the position is encoded in the neuronal spike counts. This is all the information you used in the linear filter. Second, you also have a state equation, which explains how likely a position is given the previous position. You want to combine these two sources of information in a principled way. If you are familiar with Bayesian statistics, the state equation is providing the prior distribution for the state, and the observation equation is used to refine this to give a posterior distribution of the state given neuronal spike counts. For the assumptions made previously (linear observation and state equations with Gaussian noise), there exists an analytical solution for the optimal decoder. This is the well-known Kalman filter, which has been shown to be more accurate than the linear filter for online control of a neural prosthetic (Wu et al., 2004). A similar filter has also been derived using a Poisson encoding model instead of Gaussian encoding (Brown et al., 1998).

Because Bayesian statistics are not particularly intuitive and the focus here is on programming in MATLAB, you will numerically approximate this optimal decoder rather than use the analytical solution. The disadvantage of numerical methods is they take much longer to compute than analytical solutions like the Kalman or linear filter. This means that the numerical approach might not be practical for online control. However, since you are working offline, time is not as much of an issue. The advantage of numerical methods is that they are more flexible than the Kalman filter. The Kalman filter is optimal only for a certain set of assumptions. If you want to modify the state or observation equation, an analytical solution is unlikely to exist. But you can numerically approximate the solution for any arbitrary state and observation equation (given enough time).

This method of approximation is known as the *particle filter*, which has been applied to the problem of decoding position from motor cortical signals (Brockwell et al., 2004). The idea is that you create a collection of "particles" that are guesses about the current position. This is simulating the prior distribution, the distribution of positions ignoring the spike counts. You then assign weights to each particle based on how likely it is given the lagged spike counts. You then draw a new set of particles from the old set according to these weights. This new set is simulating the posterior distribution of positions given the spike counts. The estimate of the position is just the mean of this new set of particles. Last, you move each particle forward one time step in the state model and start the process over for the next time step.

Let's look at the steps in more detail. First, you need to fit the state and observation equations to the training data. This means computing a and h as well as the noise covariance matrices W and Q. You will also need to specify an initial distribution. Use the function **cov** to calculate the covariance matrix of the kinematics training data. For the test data, just follow these steps (as suggested in Brockwell et al., 2004):

Step 1: For the first time step ($t = 1$), create a population of particles, $y_t^{(j)}$, $j = 1, \ldots, M$ from the initial distribution of your state model. M is just the number of particles; start by setting $M = 1000$. Use the function **mvnrnd** and the mean and covariance of your training data to create a set of random particles.

Step 2: Compute weights for each particle, where the weight is the probability of the particle given the lagged firing rates:

$$w_t^{(j)} = p\left(x_t | y_t = y_t^{(j)}\right) \tag{17.8}$$

In MATLAB, use your encoding model to create a vector of predicted firing rates given the particle location: **xHat = x*h;**. Then compute the probability of each predicted firing rate given the actual firing rate and the covariance matrix Q computed earlier: **p=mvnpdf(xHat,x,Q);**.

Step 3: Normalize all the weights w to sum to 1 and draw a new set of particles where each particle is chosen with a probability equal to its weight. To figure out how many times each particle should be picked, try using the following code:

```
n=rand(M,1);                   %pick M random values from 0 to 1
edges = [0; cumsum(w)];        %create edges where each bin corresponds to one particle
numPicked = histc(y,edges);    %number of times each particle is picked
```

Create a new vector of particles corresponding to the number of picks stored in *numPicked*. Your position estimate is simply the mean of this new set of particles.

Step 4: Move each particle one time step forward in the state equation. That is, apply the linear transformation matrix a and add zero-mean Gaussian noise using the function **normrnd** and the covariance matrix W. Set $t = t + 1$ and then return to step 2.

17.3. EXERCISES

Exercise 17.1: Linear Filter

1. Train a linear filter on the first dataset for this chapter using a two-time-bin (140 ms) lag and test it on the second dataset. Report both the mean-squared error and the correlation coefficients of your fit to the test data.
2. Test a variety of lags. Is two time bins the optimal lag?
3. You can use more than one lag by appending the lagged rates as extra columns of X. For example, if you used lags of one and two time bins, you would have 84 columns instead of 42 (not including the column of 1s). This corresponds to a filter length (which is the total number of lags) of 2. Test a variety of filter lengths. What do you think the best filter length is?
4. Try adding higher-order derivatives to the kinematics variable. You can approximate the derivative using the function **diff** in MATLAB. The resulting vector will have one value less than the original, so pad this result with zeros to make the lengths match. Does this improve decoding of the X- and Y-positions?

Exercise 17.2: Particle Filter Implement a particle filter. Train on the first dataset using a two-time-bin (140 ms) lag and test it on the second dataset. Add at least X and Y velocity to the state variable. Report both the mean-squared error and the correlation coefficients of your fit for the X- and Y-positions. How does this compare to the linear filter with a lag of two time bins?

17.4. PROJECT

Improve upon one of the two filters introduced in this chapter or develop your own decoder. For the linear filter approach, try using a nonlinear decoding model, though you would need to use different techniques to fit the decoding model (the function **nlinfit** might be useful). For the particle filter, try using something other than the linear plus Gaussian noise used in the state and observation equations. Comment on the assumptions that you are changing and why you think this is an improvement. Report both the mean-squared error and the correlation coefficients of your fit for the X- and Y-positions of the test data.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

inv

CHAPTER

18

Functional Magnetic Imaging

18.1. GOALS OF THIS CHAPTER

This chapter will introduce you to functional magnetic imaging (fMRI) as a fundamental noninvasive tool in understanding brain functioning in humans. We will describe the basic physics behind structural and functional magnetic resonance imaging. We will then describe the major experimental paradigms used in fMRI research and the kinds of data that are collected in an fMRI experiment. Finally, using existing fMRI data from a simple finger-tapping task, we will show you how to analyze and visualize the data to come up with a statistical parametric map of activation in the brain. After completing this project, you should expect to understand how researchers take fMRI data to infer activation associated with a behavioral task in different parts of the brain.

18.2. BACKGROUND

Functional magnetic resonance imaging has emerged as the dominant form of noninvasive functional imaging in humans. Although it is a relatively young technology that began in the early 1990s, it now plays a major role in many subfields of psychology, cognitive science, and neuroscience. It is even creeping up in other disciplines such as sociology and economics. As of the beginning of 2008, a quick online search of articles on PubMed revealed over 180,000 papers that reference the use of fMRI. Some have criticized fMRI as a scientific tool, claiming that it is little more than modern phrenology. However, we believe that it is extremely useful in localizing parts of the brain that may be involved in a particular sensory, cognitive, or motor behavior. Therefore, fMRI can be used together with psychophysical and other physiological measures to understand information processing in the brain.

18.2.1 Basic Physics of the MRI Signal

We will describe the basic physical principles that create the MRI signal. Although the physics behind MRI is inherently quantum mechanical, most of the ideas can be expressed in classical terms that you would learn in a high school physics course. There are two phenomena that must be understood: precession and relaxation. Atoms with an odd number of protons (or neutrons) in their nucleus such as hydrogen act like tiny magnetic dipoles because they possess a quantum mechanical spin. As you may remember from your high school physics, an electrical charge that is rotating will generate a magnetic field perpendicular to its rotational plane according to the right-hand rule. In the presence of an external magnetic field, B_o, these proton magnetic dipoles will tend to align with it by precessing around the B_o axis like a top precesses about the gravitational field. The precession frequency, F_o, of the protons in the nucleus is proportional to the strength of B_o with a proportionality constant that depends on the type of atomic nucleus:

$$F_o = \gamma B_o \tag{18.1}$$

This is called the *Lamour frequency* and characterizes the resonant frequency of the atomic material that is being imaged. In the presence of the static magnetic field, B_o, the spinning protons will eventually settle and align their spins with the external magnetic field and by doing so will create their own internal magnetization, M_o. The time constant that characterizes this settling or relaxation time is called the *T_1 time*.

To create an MRI signal, an external oscillating magnetic pulse, B_1, is applied in the transverse direction perpendicular to B_o. This pulse is called an *RF pulse* because the magnetic field frequency is in the radio frequency range (i.e., megahertz range) and typically lasts for a millisecond or so. This pulse is generated by a wire coil that lies in a plane parallel to B_o. If the oscillating frequency of B_1 is close to the resonant frequency (i.e., the precession frequency of the protons), the internal magnetization will be perturbed and shift its orientation toward the transverse direction. This is very much like a forced harmonic oscillator that will begin to oscillate with a very large amplitude if a forcing frequency matches the resonant frequency. The shifted internal magnetization of the protons will precess at its resonant frequency and will inductively generate an electrical signal in the same coil that generated the RF pulse. Again from high school physics, you know that a changing magnetic field generates an electrical field and will create an electric current if a wire is nearby. This inductive current will decay in time with a relaxation time constant T_2 after the RF pulse is turned off because the precessing protons that initially were in phase with each other will no longer be phase locked with each other. The time between pulses is referred to as the repetition time, *TR*.

To create an image from the MRI signal, additional gradient coils create a gradient in the static magnetic field, B_o, such that the strength of the static field varies linearly along different spatial axes. According to the Lamour frequency equation, the resonant frequency is proportional to the magnitude of the static field. In one dimension, a gradient will linearly shift the resonant frequency of the atomic material along that dimension. If an RF pulse is applied at a particular frequency, this will predominantly excite only one point along the spatial dimension. Imagine a set of harmonic oscillators with linearly varying resonant frequencies placed along one axis. A forcing oscillation at a particular frequency (the RF pulse in MRI) will excite those harmonic oscillators whose resonant frequencies are close to the forcing frequency.

More importantly, the relative phase between the oscillator and the forcing oscillation will vary linearly along the axis. This is the essence of MRI imaging.

18.2.2. BOLD Signal (fMRI)

The oxygenation concentration of blood was discovered to alter the MRI signal (Ogawa et al., 1992). In particular, as the ratio of oxygenated to deoxygenated hemoglobin increased, the MRI signal increased. It was soon found that brain activation in the human also affected the MRI signal, presumably due to changes in blood oxygenation levels surrounding the brain tissue (Kwong et al., 1992). The time course of this blood oxygenation level dependent (BOLD) signal initially shows a weak decrease followed by a much stronger increase that peaks several seconds (~5 seconds) after a stimulus is presented to the subject. Vascular physiology suggests that the source of this BOLD signal is primarily the veinous and capillary blood as opposed to arterial blood. The early weak decrease in BOLD is due to increased metabolism, resulting in an increase in relative deoxyhemoglobin concentration followed by an increase in BOLD due to an increase in blood flow, resulting in a relative decrease in deoxyhemoglobin concentration. Most studies have focused on the later, robust increase in the MRI signal. You should keep the relatively slow dynamics of the signal in mind when interpreting fMRI data because this places a limit to the temporal resolution of tracking neural activity.

18.2.3 Preprocessing of Signal

The data that are acquired in an fMRI experiment require a number of preprocessing steps before formal data analysis is performed. The data we will give you, available on the companion website, have already been preprocessed. However, it is helpful to understand what has been done to the data before you work with it (Smith, 2004). The signal from each voxel is initially represented in the Fourier domain as a set of complex numbers (k-space representation). This needs to be transformed into a set of real numbers in the time domain. Because each image is acquired in two-dimensional slices, each slice is acquired at a slightly different time. Therefore, slice-timing correction is performed. Motion correction is then performed such that each brain image is spatially adjusted so that they are all co-localized. Finally, the data are usually spatially and temporally filtered.

18.2.4 Experimental Designs

There are two major types of experimental design used in fMRI: (1) the block design and (2) the event-related design. The block design was the first approach used in early fMRI experiments and is quite easy to implement. Blocks of time (typically tens of seconds long) are defined in which subjects are either presented with multiple stimuli or perform a task repeatedly (experimental block) or are presented with nothing or asked to rest (control block). These blocks are typically presented in an alternating fashion. The BOLD signal is then compared between the experimental and control blocks. The event-related design is used when you want to examine the relationship between a behavioral event and the dynamics of the BOLD signal. In the exercises and project, you will use fMRI data that were collected using the block design, so we won't discuss the event-related design any further.

18.3. EXERCISES

These exercises are presented to introduce some of the techniques used to analyze fMRI data. Download the file P05120.7-MC.mat from the companion website. Type **load P05120.7-MC.mat** to load the data from an fMRI experiment involving a simple finger-tapping task from one human subject (data courtesy of Kristine Mosier). The experiment follows a simple block design alternating between finger-tapping (active block) and rest (control block). Each block lasts for 30 seconds. A complete image of the brain was acquired every 3 s (i.e., $TR = 3$ seconds), and there are 90 time samples for the whole experiment. The data are stored in the variable **Vfunc**, which is a four-dimensional matrix corresponding to the x (medial-lateral), y (anterior-posterior), z (inferior-superior) spatial dimensions and time. Each element of the variable **Vfunc** represents the time course of the MR signal from one voxel or volume element from the brain. One way researchers examine activation at a particular voxel is to cross-correlate the signal with the expected hemodynamic response. Begin with a simple box car hemodynamic response that is on (i.e., 1) during the active block and off (i.e., 0) during the control blocks (i.e., when the subject is at rest). The experiment begins with a rest (control) block of 10 images:

```
>> hemo=[repmat(0,10,1);repmat(1,10,1);repmat(0,10,1);repmat(1,10,1);repmat(0,10,1);...
repmat(1,10,1);repmat(0,10,1);repmat(1,10,1);repmat(0,10,1);];
```

Begin by looking at one voxel. Plot **Vfunc(27,37,9,:)** along with the expected hemodynamic response in two subplots in the same figure. You will first need to use the **reshape** function to convert **Vfunc(27,37,9,:)** into a column vector**.** The result should look like Figure 18.1.

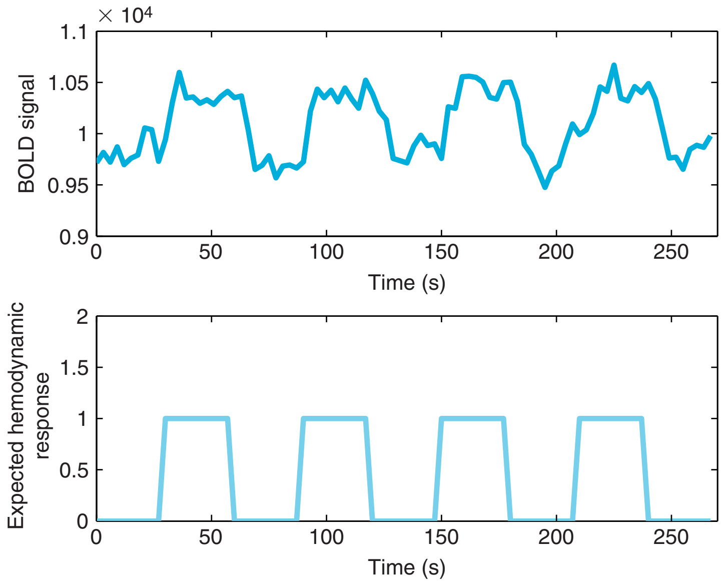


FIGURE 18.1 The BOLD signal from one sample voxel along with the expected hemodynamic response.

Notice how the BOLD signal oscillates at the same frequency as the expected hemodynamic response.

Exercise 18.1:
Compute a power spectrum of that voxel using **pwelch**.

Exercise 18.2:
Compute the cross-covariance between the voxel activation and the expected hemodynamic response using **xcov**:

```
>> [b a]=xcov(voxel,hemo,'coeff');
```

The result is shown in Figure 18.2.

The **xcov** function generates the cross-covariance instead of the cross-correlation because you want to examine how the two signals co-vary with respect to their respective means. The function **xcorr** would consider their co-variation ignoring their mean values despite the fact the two signals have completely different units and magnitudes. The **'coeff'** flag makes sure that the output represents the normalized correlation coefficient ranging from –1 to 1. If you zoom in the figure, you will notice that the peak in the cross-covariance occurs at a lag time. This is the biophysical delay between the performance-based neural activation and the hemodynamic response.

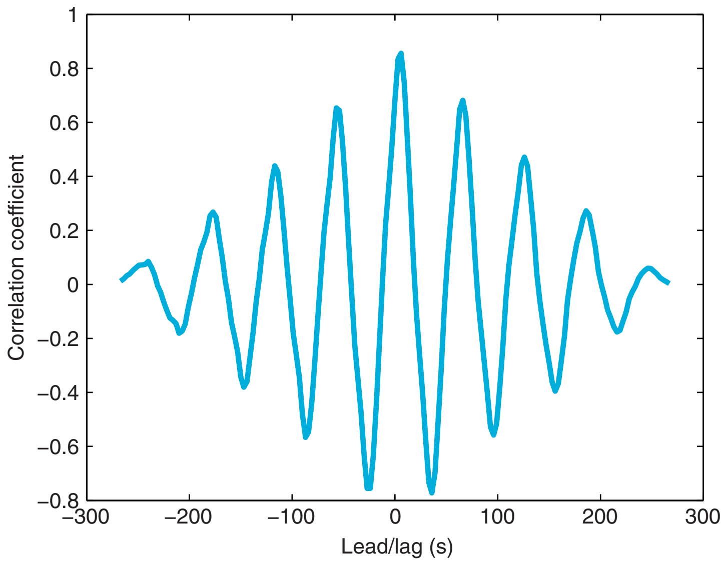


FIGURE 18.2 The cross-covariance between the sample voxel and the expected hemodynamic response.

Continued

Exercise 18.2: *(continued)*

To quantitatively determine which voxels are significantly activated, you will apply a regression model or general linear model (glm) to find a linear relationship between the expected hemodynamic response and the actual BOLD signals. You will use the biophysical delay, computed with the cross-covariance function, in the model. Specifically, you will find the optimal (in the least-squares sense) offset and gain parameters that relate the expected hemodynamic response and the voxel's BOLD signal such that

$$voxel = offset + gain \times hemo + \varepsilon \tag{18.2}$$

where ε is normally distributed noise. You will use the function **glmfit** in the MATLAB® software to do this. This function allows you to apply a whole variety of general linear models, including a simple linear regression model, by using the identity link function and normally distributed noise. However, before using this function, you need to introduce the biophysical delay into the expected hemodynamic response.

Exercise 18.3:

Run **glmfit** on the sample voxel using the delayed hemodynamic response. What are the offset and gain parameters that are computed and their p-values:

```
>> [param, dev, stats] = glmfit(hemo_delay, voxel,'normal', 'identity');
```

param stores the offset and gain parameters, **dev** represents the deviance, and **stats** stores other statistical values including p-values for the parameter (i.e., **stats.p**).

18.4. PROJECT

This project involves analyzing the entire fMRI dataset using the general linear model and then plotting a grayscale figure showing significant activation across the brain. Specifically, you should do the following:

- Apply the **glmfit** function on each voxel separately to determine significant activation due to the task. Use the expected hemodynamic response as your covariate in the model with a delay determined from the cross-correlation peak you found in Exercise 18.2.
- Use a p-value of .001 for the gain parameter as a threshold for activation.
- Generate a grayscale surface plot for each *x*-*y* slice of the brain using white to represent significant activation and black to represent nonsignificant activation. Your result should look something like Figure 18.3.

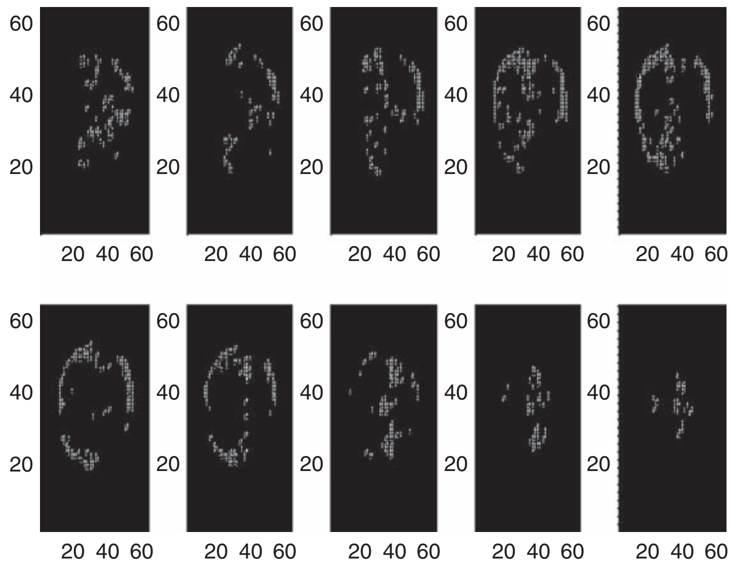


FIGURE 18.3 Statistical parametric maps of activation ($p < .001$) across all horizontal brain slices.

18.4.1 Methods Used to Collect fMRI Data

Subject and Paradigm: One female subject, age 39, was used. A block design was used, alternating 30 seconds "ACTIVE," 30 seconds "REST" for 135 seconds. The active block required the subject to touch the thumb to the four fingers bilaterally and repeat this in a self-paced manner.

BOLD parameters: Reverse spiral pulse sequence, single echo, TR/TE = 3000/50 ms, flip angle = 90, 64×64 matrix, FOV = 24 cm, slice thickness = 7.0 mm/skip 1.0, 10 axial slices, bandwidth = 100 kHz. The first 12 acquired brain images were eliminated to account for equilibration.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

pwelch
repmat
reshape
xcorr
xcov
glmfit

CHAPTER

19

Voltage-Gated Ion Channels

19.1. GOAL OF THIS CHAPTER

This chapter will explore the dynamics of ion channels using methods similar to those introduced by Hodgkin and Huxley in 1952. You will derive ordinary differential equations that approximate real ion channel behavior and solve these equations using a numerical integrator written in the MATLAB® software. Finally, you will visualize the dynamics by predicting current responses to single channel voltage-clamp experiments.

19.2. BACKGROUND

Ion channels are a class of multimeric transmembrane proteins with a hydrophilic pore that facilitates transport of ions across the cell membrane. The size of the ion channel pore and the charge of amino acids near the opening of the pore help exclude entry of some ions while promoting the entry of others. This confers upon the ion channel a selective permeability to different ions. Several factors can induce conformational changes in the ion channel, altering its quaternary structure and therefore its permeability. These factors are referred to as *gating variables* because they function as a gate between the ion channels' different conformational states.

Most ion channels are classified according to the nature of their gating and their selectivity of ions. The largest subclasses of ion channels classified by gating are the *ligand-gated* and *voltage-gated* ion channels. Ligand-gated ion channels change conformation when a ligand binds to them. The most common ligand-gated ion channels found at the post-synaptic membrane of neurons include NMDA, kainate, AMPA, and $GABA_A$ receptors.

The second class of ion channels, voltage-gated ion channels, undergo conformational changes corresponding to alterations in membrane potential. Voltage-gated ion channels can show selectivity for sodium, potassium, or calcium. In this chapter you will model voltage-gated potassium channels (K_v channels) and voltage-gated sodium channels (Na_v channels). K_v channels generally gate between two conformations—an open conformation permeable to potassium and a closed conformation impermeable to potassium—while Na_v channels often gate between three stable conformations—an open conformation permeable to sodium, a closed conformation impermeable to sodium, and an inactive conformation also impermeable to sodium. Although both the inactive and closed states of Na_v channels are impermeable to sodium, they represent different conformational states of the channel and have different kinetics.

19.2.1. The Model

Suppose you are interested in building a model to predict the current response to a single ion channel voltage-clamp experiment. Since ionic currents pass through the ion channel to enter the cell, you can consider the ion channel as an electrical resistor whose resistance depends on the conformational state of the ion channel. The equation that relates the resistance of a resistor, R, to the current it passes, I, and the voltage drop across the resistor, V, is Ohm's law:

$$V = IR \tag{19.1}$$

If you divide this equation through by the resistance, then you can write this equation as

$$I = gV \tag{19.2}$$

where g is the conductance of the resistor (note $g = 1/R$). Finally, if you suppose that the conductance is directly proportional to the probability that the channel is in the open conformation, then Equation 19.1 becomes

$$I = g_{\max} * P_o * V, \tag{19.3}$$

where g_{max} is the maximum conductance of the channel, and P_o is the probability that the channel is in the open conformation. Therefore, determining the conductance of the channel is equivalent to determining the probability that the channel is open.

19.2.2. K_v Channel

Let's begin with the simplest case, the K_v channel. For the K_v channel, Equation 19.3 will take the form

$$I_K = \overline{g}_K * n * V \tag{19.4}$$

where $\overline{g}_K = 36\,\mu S/cm^2$ is the maximum conductance of the K_v channel, and n is the probability that the channel is in the open conformation. Next, suppose that the ion channel can exist only in an open or closed conformation as depicted by the reversible reaction in the equation

$$(K_v)_{closed} \underset{k_{-1}}{\overset{k_1}{\leftrightarrow}} (K_v)_{open} \tag{19.5}$$

where k_1 is the rate the channel goes from closed to open, and k_{-1} is the rate the channel goes from open to closed. Next, assume that the change in the probability of open channels over time is equal to the probability of the channel being closed, and then going from closed to open (at rate k_1), minus the probability of its being open, and then closing (at rate k_{-1}). This can be represented by the equation

$$\frac{dn}{dt} = (1-n)k_1 - nk_{-1} = k_1 - (k_1 + k_{-1})n, \tag{19.6}$$

since all channels are either open or closed. Now further assume that at time $t = 0$ all the channels are closed so that $n(0) = 0$. Finally, recall that for a voltage-gated ion channel the gating between conformational states depends on the membrane potential, so the rates k_1 and k_{-1} are both functions of voltage. If you were modeling ligand-gated ion channels, then these rates would depend on the concentration of the ligand. Hodgkin and Huxley used voltage clamp experiments to help determine these rates. In this chapter assume the following functional forms (see Hodgkin and Huxley, 1952) for the transition rates between conformational states of the K_v channel:

$$\begin{aligned} k_1 &= \frac{0.01 * (V+10)}{\exp\left(\dfrac{V+10}{10}\right) - 1} \\ k_{-1} &= 0.125 * \exp\left(\frac{V}{80}\right). \end{aligned} \tag{19.7}$$

19.2.3. The Na_v Channel

The Na_v channel is slightly more complicated, since it has three stable confirmations and therefore a greater number of possible transitions between conformational states. For simplicity, we will ignore transitions between inactive and closed conformational states, and assume that the channel is governed by the following reversible reactions that act independently of each other:

$$\begin{aligned} &(Na_v)_{closed} \underset{k_{-1}}{\overset{k_1}{\leftrightarrow}} (Na_v)_{open} \\ &(Na_v)_{inactive} \underset{k_{-1}}{\overset{k_1}{\leftrightarrow}} (Na_v)_{open}. \end{aligned} \tag{19.8}$$

If you let m represent the probability of the channel being open given that it was closed previously, and you let h represent the probability of the channel being open given that it was inactive previously, then Equation 19.3 takes on the form:

$$I_{Na} = \overline{g}_{Na} * m * h * V, \tag{19.9}$$

where $\overline{g}_{Na} = 120\ \mu S/cm^2$. The previous reversible reactions lead to the following differential equations:

$$\begin{aligned}\frac{dm}{dt} &= (1-m)k_1 - mk_{-1} = k_1 - (k_1 + k_{-1})m \\ \frac{dh}{dt} &= (1-h)k_1 - hk_{-1} = k_1 - (k_1 + k_{-1})h.\end{aligned} \tag{19.10}$$

For the $\mathrm{Na_v}$ open-close kinetics, assume:

$$\begin{aligned}k_1 &= \frac{0.1 * (V+25)}{\exp\left(\frac{V+25}{10}\right) - 1} \\ k_{-1} &= 4 * \exp\left(\frac{V}{18}\right),\end{aligned} \tag{19.11}$$

and for $\mathrm{Na_v}$ inactivation kinetics, assume:

$$\begin{aligned}k_1 &= 0.07 * \exp\left(\frac{V}{20}\right) \\ k_{-1} &= \frac{1}{\exp\left(\frac{V+30}{10}\right) + 1}.\end{aligned} \tag{19.12}$$

Simple intuition has led naturally to a model that expresses the current you expect to flow through a channel in terms of a differential equation for the probability of the channel being open.

Next, we will discuss a simple algorithm to numerically solve differential equations such as Equation 19.6.

19.2.4. Solving Differential Equations Numerically

To understand the current-voltage properties of an ion channel, you can solve an ordinary differential equation describing the probability of the channel being open given the initial condition that $P_o(0) = 0$. In general, solving differential equations can be very tricky (if not impossible), but some simple techniques for approximating solutions do exist. Perhaps the simplest method for approximating the solution to a differential equation is Euler's method. It is based on the definition of the derivative:

$$\frac{df}{dx} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}, \tag{19.13}$$

which for small nonzero values of Δx implies that:

$$\frac{df}{dx} \approx \frac{f(x+\Delta x)-f(x)}{\Delta x} \Rightarrow f(x+\Delta x) \approx f(x) + \Delta x * \frac{df}{dx}. \tag{19.14}$$

Equation 19.14 lets you determine an approximation for the value of a function f at a point $(x+\Delta x)$ given information about the value of the function of f at x and the derivative of f. You can examine Euler's method by choosing a differential equation whose solution is known and compare it to the approximation obtained from Equation 19.14. Now try to apply Equation 19.14 to a simple differential equation:

$$\frac{df}{dx} = 2x, \text{ where } f(0) = 1. \tag{19.15}$$

This differential equation has the obvious solution $f(x) = x^2 + 1$, since it satisfies Equation 19.15 with the condition that $f(0) = 1$. To approximate the solution to this equation using Euler's method, you proceed by first plugging Equation 19.15 into Equation 19.14 to get:

$$f(x+\Delta x) \approx f(x) + \Delta x * 2x. \tag{19.16}$$

Next, you choose $\Delta x = 0.1$ (i.e., something small, since the approximation is most valid for small Δx) and $x = 0$ and plug into Equation 19.16 to obtain:

$$f(0+0.1) \approx f(0) + 0.1 * 2 * 0 \Rightarrow f(0.1) \approx 1. \tag{19.17}$$

This equation can be repeated to give an estimate of $f(0.2)$ such that:

$$f(0.1+0.1) \approx f(0.1) + 0.1 * 2 * 0.1 \Rightarrow f(0.2) \approx 1.02, \tag{19.18}$$

where the previous value $f(0.1)$ has been substituted into Equation 19.16 instead of the initial condition. Although you might be tempted to approximate $f(0.2)$ directly from the initial condition by letting $\Delta x = 0.2$, recall that the approximation is best when Δx is as close to 0 as possible. This procedure can be repeated to approximate $f(x)$ over any range of x desired. Of course, without the help of modern computers, this method would be too laborious to be practical for any moderately large range of x.

Another method for numerically solving differential equations is known as the Runge-Kutta method (RK method). We shall derive the second-order RK method for the general differential equation:

$$\frac{dy}{dx} = f(x,y), \tag{19.19}$$

with the initial condition $y(x_o) = y_o$. Note that the ion-channel gating Equations 19.6 and 19.10 are of this general form. Our first step in deriving the formulae for the RK method is to assume that the numerical solution $y(x)$ that we are looking for has a Taylor series expansion that converges on the interval I for which we want to find the numerical solution. If it does, then:

$$y(x) = y(x_o) + \frac{y'(x_o)}{1!} * (x-x_o) + \frac{y''(x_o)}{2!} * (x-x_o)^2 + \ldots + \frac{y^{(n)}(x_o)}{n!}(x-x_o)^n \tag{19.20}$$

is the Taylor series expansion. Let us define $\Delta x = (x - x_o)$, and substitute this into Eq. 9.20, which gives:

$$y(x_o + \Delta x) = y(x_o) + \frac{y'(x_o)}{1!} * (\Delta x) + \frac{y''(x_o)}{2!} * (\Delta x)^2 + \ldots + \frac{y^{(n)}(x_o)}{n!}(\Delta x)^n. \tag{19.21}$$

We can make a polynomial approximation to the series in Eq. 19.21. If we make a 2nd-order approximation, then:

$$y(x_o + \Delta x) \approx y(x_o) + \frac{y'(x_o)}{1!} * (\Delta x) + \frac{y''(x_o)}{2!} * (\Delta x)^2. \tag{19.22}$$

Notice that the first-order approximation reduces to the key equation in iterating Euler's method (see Equation 19.14). Next, we realize that Eq. 19.19 gives us a substitution for $y'(x_o) = f(x_o, y_o)$. We can take the derivative of Eq. 19.19 to obtain a substitution for $y''(x_o)$ as follows:

$$y''(x_o) = \frac{\partial f(x_o, y_o)}{\partial x} + \frac{\partial f(x_o, y_o)}{\partial y} y'(x_o) = \frac{\partial f(x_o, y_o)}{\partial x} + \frac{\partial f(x_o, y_o)}{\partial y} f(x_o, y_o). \tag{19.23}$$

Making these two substitutions into Equation 19.22 gives:

$$y(x_o + \Delta x) \approx y(x_o) + f(x_o, y_o) * (\Delta x) + \left[\frac{\partial f(x_o, y_o)}{\partial x} + \frac{\partial f(x_o, y_o)}{\partial y} * f(x_o, y_o)\right] * \frac{(\Delta x)^2}{2}. \tag{19.24}$$

If f were a function whose derivatives could easily be calculated, then we could simply end here and use Equation 19.24 in much the same way as we used Eq. 19.14 to iterate Euler's method. This does not happen often, however, so we will try to find a simplification for Eq. 19.24 that does not involve partial derivatives of f. Since y was assumed to have a Taylor series expansion, then its derivative does too, so by Equation 19.19 f must also have a Taylor series expansion. The Taylor series expansion of f is slightly complicated, however, since f is a multivariable function. The Taylor expansion is as follows:

$$f(x_o + a, y_o + b) = f(x_o, y_o) + \frac{\partial f(x_o, y_o)}{\partial x} * a + \frac{\partial f(x_o, y_o)}{\partial y} * b + \ldots \tag{19.25}$$

The terms shown in Equation 19.25 are through first order. If we let $a = \Delta x$ and $b = \Delta x * f(x_o, y_o)$, and we substitute these into Equation 19.25 keeping only up to the first order terms of the series, then we get:

$$f[x_o + \Delta x, y_o + \Delta x * f(x_o, y_o)] = f(x_o, y_o) + \left[\frac{\partial f(x_o, y_o)}{\partial x} + \frac{\partial f(x_o, y_o)}{\partial y} * f(x_o, y_o)\right] * \Delta x. \tag{19.26}$$

If we subtract $f(x_o, y_o)$ from the right hand side of Eq. 13 and multiply through by $\Delta x/2$ we obtain:

$$\frac{\Delta x}{2}\{f[x_o + \Delta x, y_o + \Delta x * f(x_o, y_o)] - f(x_o, y_o)\} = \left[\frac{\partial f(x_o, y_o)}{\partial x} + \frac{\partial f(x_o, y_o)}{\partial y} * f(x_o, y_o)\right] * \frac{(\Delta x)^2}{2}. \tag{19.27}$$

The right-hand side of Equation 19.27 is the last term of Eq. 19.24. If we now substitute Eq. 19.27 into Eq. 19.24, then we will have removed the partial derivatives of *f*. After slight simplification we achieve:

$$y(x_o + \Delta x) \approx y(x_o) + \{f(x_o, y_o) + f[x_o + \Delta x, y_o + \Delta x * f(x_o, y_o)]\} * \frac{(\Delta x)}{2}. \tag{19.28}$$

We now have an equation to give us the next y value given the previous value that requires only the initial condition $y(x_o) = y_o$, the step size Δx, and the differential equation, which gives us *f*. Since Equation 19.28 is quite cumbersome looking, it is often presented as a set of equations in the following way:

$$\begin{aligned} y(x_o + \Delta x) &= y(x_o) + \frac{1}{2}(u_1 + u_2) \quad \text{where} \\ u_1 &= \Delta x * f(x_o, y_o) \text{ and} \\ u_2 &= \Delta x * f(x_o + \Delta x, y_o + u_1) \end{aligned} \tag{19.29}$$

Make the substitutions and convince yourself that this system is equivalent to Equation 19.28. In Equation 19.22 above we truncated the Taylor expansion of y to second-order. For that reason, the set of equations shown above are called the second-order RK equations. We can build higher order RK equations by keeping higher-orders of the series expansion of y and using higher-order expansions of the function f to remove partial derivatives. The following set of equations is the result of truncating y to fourth order.

$$\begin{aligned} y(x_o + \Delta x) &= y(x_o) + \frac{1}{6}(v_1 + 2v_2 + 2v_3 + v_4) \quad \text{where} \\ v_1 &= \Delta x * f(x_o, y_o) \\ v_2 &= \Delta x * f\left(x_o + \frac{\Delta x}{2}, y_o + \frac{v_1}{2}\right) \\ v_3 &= \Delta x * f\left(x_o + \frac{\Delta x}{2}, y_o + \frac{v_2}{2}\right) \\ v_4 &= \Delta x * f(x_o + \Delta x, y_o + v_3) \end{aligned} \tag{19.30}$$

The fourth-order RK equations are the most popular numerical method for solving differential equations used today. Although a higher-order expansion in y would give a more accurate solution, the increased processing time required by a computer to achieve the solution is often not worth the minor improvement.

19.3. EXERCISES

For these exercises, begin by writing a function called **ode_euler**, which will implement Euler's method to solve the sample differential equation of Equation 19.14 for x = 0:0.1:10:

```
function f = ode_euler(x, f_o)
%This function takes two arguments, x and f_o.
%x is a vector that specifies the time points that the function f should be
%approximated for.
%f_o is the initial condition.
%The function returns a vector, f, representing the approximate solution to the
%differential equation, df/dx=2x with f(0)=f_o.

%Set delta_x as the difference between successive x values.
delta_x=x(2)-x(1);

%Determine how many points we need to approximate by finding the length of
%vector x.
l_x=length(x);

%Initialize f by creating a vector of the right length. We will reset the elements to
%the correct values in the for loop below.
f=zeros(1, l_x);

%Set the initial value of f to f_o.
f(1)=f_o;

%Use a for-loop to implement Eq. 19.14
for i=1:(l_x-1)
   f(i+1)=f(i) + delta_x*2*x(i);      % line 24
end;
```

Now visualize the solution by plotting this approximation for f alongside the exact solution, $f(x) = x^2 + 1$. See whether your solution looks like the one shown in Figure 19.1. Before proceeding, you should explore the relationship between the value of Δx and the validity of the approximation of Euler's method. For example, if you plot the exact solution alongside several approximations, each with a different Δx, how quickly does the approximation cease to be reasonable? Similarly, at what point does decreasing Δx fail to provide significant improvement in the approximation despite increased run time? Basic questions such as these are important to consider whenever a numerical method is employed to approximate the solution to a differential equation.

The function **ode_euler** can solve only the sample differential equation of Equation 19.14 because the derivative was plugged into Euler's method explicitly in line 24. You can generalize this function to solve any differential equation by introducing the **feval()** function. The **feval()** function evaluates functions by taking a functional handle that references the function to be evaluated and a variable number of arguments depending on the number of input arguments required by the function referenced by the function handle. As an example, consider the following command:

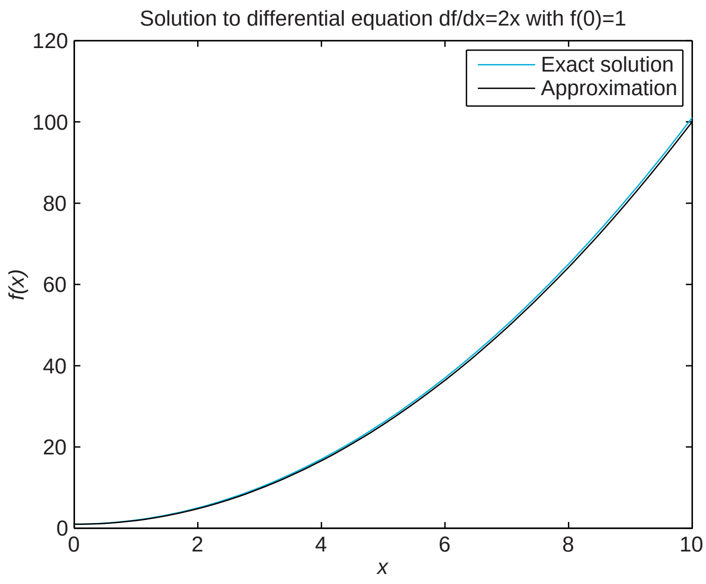


FIGURE 19.1 Exact and approximate solution to differential equation.

```
>>f=feval(@ode_euler, 0:0.1:10, 1);
```

This line is equivalent to typing

```
>>f=ode_euler(0:0.1:10, 1);
```

except with the **feval()** function the name of the function is a variable argument, so it can be changed. To see how this can be applied to generalize the code for Euler's method, first create another function called **f_prime():**

```
function df = f_prime(x)
%This function takes a point x and calculates the derivative of f at the point x.

df=2*x;
```

Now modify the first line of **ode_euler** so that it takes an additional argument, a function handle to a differential equation, and modify line 24 to use **feval()** to determine the value of the derivative according to the function referred to by the function handle. If everything is done correctly, then the command

```
>>f=ode_euler(@f_prime, 0:0.1:10, 1);
```

should produce the same results as before **feval()** was introduced into the code, but now **ode_euler** makes no explicit reference to any particular differential equation.

Exercise 19.1: Write a function **n_prime(*t*, *V*)** that calculates the derivative of n at the point t given that the membrane potential is V. *Hint:* You will need to use Equations 19.6 and 19.7.

Exercise 19.2: Modify the first line of **ode_euler()** so that it takes an additional input argument, V, and line 24, so that **feval()** takes three arguments, a function handle such as **@n_prime** and two additional input arguments, t and V, to the function referred to by the function handle.

Exercise 19.3: Write a function **RK4(fhandle, x, f_o)** that uses the fourth order Runge-Kutta method to numerically solve the differential equation referenced by the function handle fhandle. Use RK4 to solve the differential equation 19.14 for $\Delta x = 0.1$. Compare this to the exact solution and the solution using Euler's method (see Figure 19.1).

19.4. PROJECT

In this project, you will use the Euler method to derive the current kinetics for the voltage-gated potassium and sodium channels:

1. Write a function called **K_v(*t*, *V*)** that takes a time interval t and a holding potential V and returns the current response of a K_v channel over the time range specified by t. *Hint:* **K_v** should call **ode_euler** or RK4 with the inputs **@n_prime**, t, n_o, and V, and use the result along with Equation 19.4 to determine I_K.
2. Use **K_v** to plot the current response of a K_v channel when the membrane potential is clamped to –30 mV. Repeat this for holding potentials from –30 mV to 50 mV in 10 mV increments, and plot the solutions on the same graph. *Hint:* See **hold on** command.
3. Write functions **m_prime(*t*, *V*)** and **h_prime(*t*, *V*)** that calculate the derivative of m and h at the point t given that the membrane potential is V. This will be completely analogous to the code for **n_prime**.
4. Write a function called **Na_v(*t*, *V*)** that takes a time interval t and a holding potential V, and returns the current response of a Na_v channel over the time range specified by t. *Hint:* **Na_v** should call **ode_euler** or RK4 twice, once with the inputs **@m_prime**, t, m_o, and V and another with the inputs **@h_prime**, t, h_o, and V and use these results along with Equation 19.9 to determine I_{Na}.
5. Use **Na_v** to plot the current response of an Na_v channel when the membrane potential is clamped to –30 mV. Repeat this for holding potentials from –30 mV to 50 mV in 10 mV increments, and plot the solutions on the same graph.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

feval
hold on

CHAPTER

20

Models of a Single Neuron

20.1. GOAL OF THIS CHAPTER

The goal of this chapter is to incorporate previous models of voltage-gated ion channels into a model of single neuron dynamics. This chapter will continue to follow work done by Hodgkin and Huxley (1952) resulting in a system of four ordinary differential equations that model action potential generation in neurons.

20.2. BACKGROUND

Neurons communicate with each other by transmitting and receiving electrochemical signals called *action potentials*. These action potentials are transient fluctuations in the cell's membrane potential, which propagate down a cell's axon without attenuation. In the central nervous system, action potentials have a duration on the order of milliseconds (1–2 msec usually) and can often be divided into three phases. The first phase of the action potential is a rapid depolarization of the membrane called the *rising phase* or *upstroke* of the action potential. This is followed by a repolarization of the membrane called the *falling phase* or *downstroke* of the action potential. The last phase follows a hyperpolarization of the membrane and is called the *undershoot*. A depiction of the action potential is shown in Figure 20.1.

Some of the earliest experiments to elucidate the mechanism underlying action potentials were performed by Hodgkin and Katz (1949), who showed that reducing the extracellular concentration of sodium led to a shorter upstroke phase of the action potential in giant squid axon. They inferred from this that the upstroke of the action potential depends on the cell increasing its permeability to sodium. They also suggested that the falling phase was due to an increase in potassium permeability. Therefore, they concluded that the action potential was generated by selective changes in membrane permeability to sodium and potassium. We now know that ion channels are responsible for this selective permeability. These experiments were later followed up by Hodgkin and Huxley (1952), who performed voltage-clamp experiments to characterize the dynamics of these changes in permeability and then proposed the mathematical model of action potential generation outlined in the following section.

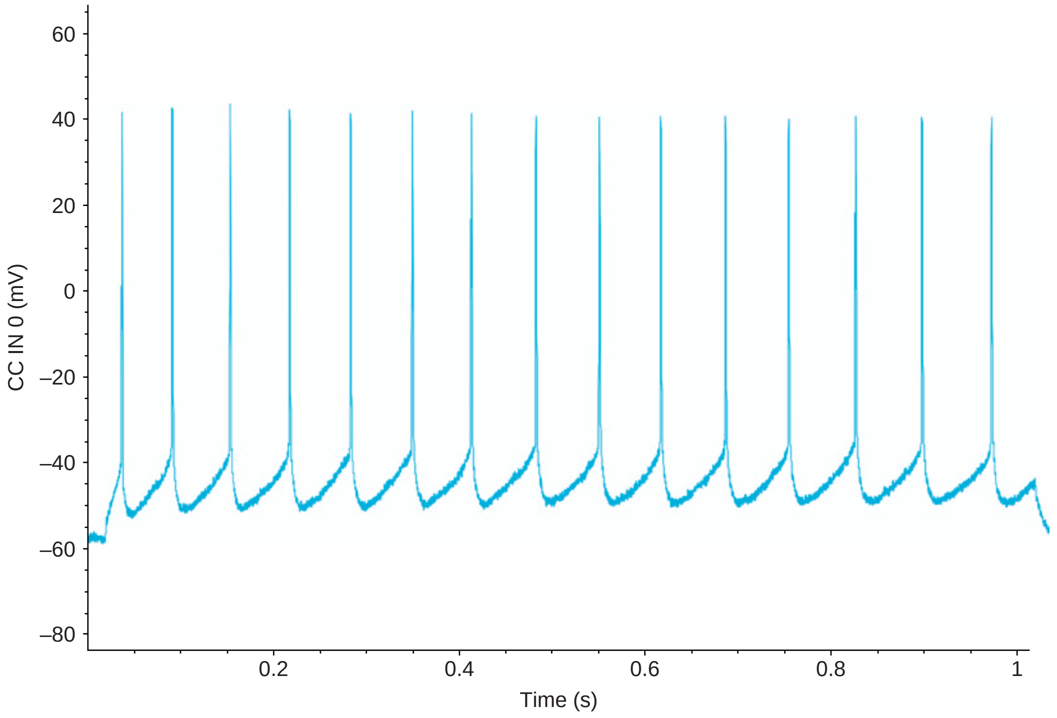


FIGURE 20.1 Intracellular action potential spike train from a deep pyramidal neuron recorded from the frontal cortex of a mouse. (Courtesy of Amber Martell)

20.2.1. The Model

Neurons are incredibly complex. Like all eukaryotic cells, they are composed of many organelles, including a nucleus, mitochondria, an endoplasmic reticulum, etc. Each of these organelles has a role that enables the cell as a whole to perform its functions, including generating action potentials. Trying to capture all the complexity of a real neuron in a single model is impossible. Fortunately, it is also unnecessary, since, for purposes of this chapter, you are interested only in understanding action potential generation in neurons, and not any of the other complex processes that neurons undergo. Therefore, you should restrict your neuron model to include only those elements that contribute most directly to generating action potentials and ignore elements of a neuron that contribute less to action potential generation. In general, it is often not clear what elements of a complex biological system are most directly related to a behavior of interest, and the choices you make in constructing a model are often not validated until the results of the model can be compared to experiments.

In this model, assume that action potential generation in neurons is mainly carried out by the electrical properties of the cell membrane. Several factors contribute to the electrical properties of the cell membrane. For instance, ion channels such as Na_v, K_v, and leak channels span the membrane and selectively pass ions across it. The voltage-gated channels are represented in Figure 20.2 as variable resistors (the resistors with an arrow going through

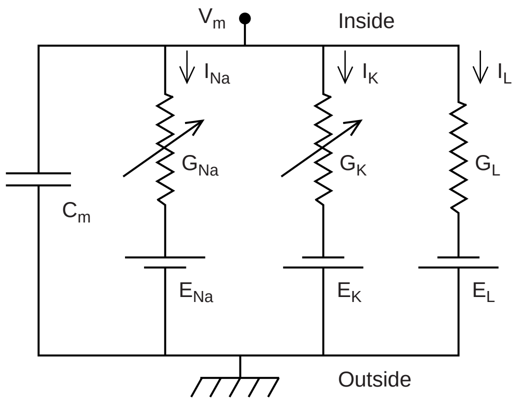


FIGURE 20.2 An electrical circuit diagram of a single axonal compartment of a neuron. (Bower JM, Beeman D. The Book of Genesis: Exploring Realistic Neural Models with the GEneral NEural SImulation System, 2003)

them) because the amount of resistance to flow depends on the membrane potential, whereas the leak channel, which has a constant resistance to ion flow, is represented by an ordinary resistor. The phospholipids that comprise the membrane, which do not conduct electric charges, allow for most of the cell membrane to function as a dielectric, an insulative material that separates ions in the cytoplasm from those in the extracellular milieu. Although ions cannot flow through the phospholipid bilayer of the cell directly, charge can accumulate on one side of the cell membrane, inducing an opposed charge buildup on the opposite side of the membrane just as a capacitor does. This charge buildup involves charges moving toward the membrane and represents a capacitive current. Finally, the cell membrane contains many other transmembrane proteins such as the Na^+/K^+-ATPase that helps maintain ion concentration gradients across the cell membrane. The presence of a sodium concentration gradient, for example, ensures that when sodium ions have equilibrated across the Na_v channels of the neuron, there will be a nonzero potential across the membrane. This potential is called the *sodium reversal potential*. Similarly, there will be a reversal potential for the K_v channel. The electrical properties mentioned so far can be summarized by creating an electric circuit equivalent to the neuronal model. The circuit is shown in Figure 20.2. Notice that current flows from the inside of the cell (at the top of the electrical circuit) to the outside of the cell by either inducing a charge buildup at the membrane (represented by the capacitor) or by flowing through one of the three ion channels present in the membrane. From simple electrical circuit theory, you can represent the following circuit using a set of equations. In the next section, we will review the important concepts of electrical circuits needed to understand the circuit in Figure 20.2.

To express the circuit as a set of equations, you need to know four fundamental laws of electronics. The first is Ohm's law, which states that for some resistors (called *Ohmic resistors*) the voltage drop across the resistor, V_R, is related to the current flowing through the resistor, I, and the resistance of the resistor, R, by the equation:

$$V_R = IR \tag{20.1}$$

which can also be written as:

$$I = gV_R \tag{20.2}$$

where g is the conductance of the resistor (note $g = 1/R$). The second law you will need states that the voltage drop across a capacitor, V_C, is related to the current induced by the capacitor, I, and the capacitance of the capacitor, C, by the equation

$$V_C = \frac{1}{C}\int I(t)dt \tag{20.3}$$

The last two laws that you will need are collectively known as *Kirchhoff's Loop Rules*. The first rule, Kirchhoff's Current Rule, states that the sum of current entering a circuit junction equals the sum of current exiting it, and a circuit junction is any intersection of wire where current has more than one path to flow down. The equation for this rule is given by:

$$\sum I_{in} = \sum I_{out} \tag{20.4}$$

The second rule, Kirchoff's Voltage Rule, states that the potential drop between any two points on a circuit is independent of what path was taken to arrive there. If you assume that the start and end point are the same, then this rule implies that the voltage drop across any closed loop is zero, and can be written as:

$$\sum_{loop} V = 0 \tag{20.5}$$

Now you can use these simple rules to calculate the membrane potential of the circuit in Figure 20.2. The membrane potential is defined as the potential difference between the inside and the outside of the cell. Therefore, in Figure 20.2 the membrane potential is the potential drop across any path from the inside of the cell to the outside. Beginning with the path that includes the capacitor, you see that the voltage drop across the capacitor is just the membrane potential, so Equation 20.3 becomes:

$$V_M = \frac{1}{C_M}\int I(t)dt \tag{20.6}$$

which can be rearranged to give:

$$I = C_M\frac{dV_M}{dt} \tag{20.7}$$

Now examine the potential drop across the second path (the sodium channel), which consists of two elements, a resistor and a battery. The total drop across both these elements is just the potential difference between the inside and outside of the cell, V_M, so:

$$V_M = V_R + E_{Na} \Rightarrow V_R = V_M - E_{Na} \tag{20.8}$$

Upon substitution into Equation 20.2, you have:

$$I_{Na} = g_{Na} * (V_M - E_{Na}) \tag{20.9}$$

Following the same process across the last two paths produces equations nearly identical to Equation 20.9 for I_K and I_L,

$$I_K = g_K * (V_M - E_K) \tag{20.10}$$

$$I_L = g_L * (V_M - E_L) \tag{20.11}$$

Finally, use Kirchhoff's Current Rule to see that if you inject a current into the cell of I_{inj}, then:

$$I_{inj} = I + I_{Na} + I_K + I_L \tag{20.12}$$

Rearranging Equation 20.12 and substituting in Equations 20.7 and 20.9 through 20.11 gives:

$$C_M \frac{dV_M}{dt} = -g_{Na} * (V_M - E_{Na}) - g_K * (V_M - E_K) - g_L * (V_M - E_L) + I_{inj} \tag{20.13}$$

Recall that the sodium and potassium channels are voltage-gated, so their conductances are functions of voltage. In Chapter 19, "Voltage-Gated Ion Channels," you modeled the potassium conductance as:

$$g_K = \bar{g}_K * n \tag{20.14}$$

where

$$\frac{dn}{dt} = k_{1n} - (k_{1n} + k_{-1n})n \tag{20.15}$$

and the sodium conductance by:

$$g_{Na} = \bar{g}_{Na} * m * h \tag{20.16}$$

where

$$\begin{aligned} \frac{dm}{dt} &= k_{1m} - (k_{1m} + k_{-1m})m \\ \frac{dh}{dt} &= k_{1h} - (k_{1h} + k_{-1h})h \end{aligned} \tag{20.17}$$

If you substitute Equation 20.14 and 20.16 into Equation 20.13 and collect Equations 20.15 and 20.17, you get the following system of equations:

$$\begin{aligned} C_m \frac{dV_M}{dt} &= -\bar{g}_{Na} mh(V_M - E_{Na}) - \bar{g}_K n(V_M - E_K) - g_L(V_M - E_L) + I_{inj} \\ \frac{dn}{dt} &= k_{1n} - (k_{1n} + k_{-1n})n \\ \frac{dm}{dt} &= k_{1m} - (k_{1m} + k_{-1m})m \\ \frac{dh}{dt} &= k_{1h} - (k_{1h} + k_{-1h})h \end{aligned} \tag{20.18}$$

In the original Hodgkin-Huxley model, the final equations proposed were as follows:

$$\begin{aligned}
C_m \frac{dV_M}{dt} &= -\bar{g}_{Na} m^3 h (V_M - E_{Na}) - \bar{g}_K n^4 (V_M - E_K) - g_L (V_M - E_L) + I_{inj} \\
\frac{dn}{dt} &= k_{1n} - (k_{1n} + k_{-1n}) n \\
\frac{dm}{dt} &= k_{1m} - (k_{1m} + k_{-1m}) m \\
\frac{dh}{dt} &= k_{1h} - (k_{1h} + k_{-1h}) h
\end{aligned} \tag{20.19}$$

The changes to the first equation were made so that the model would better fit with the experimental data, although some explanation of the addition of these exponents has since been made from first principles.

Many of the parameter values needed to evaluate the system of Equation 20.19 are mentioned in Chapter 19, "Voltage-Gated Ion Channels." Table 20.1 identifies these parameter values along with some additional parameter values for the leak channel and capacitance of the membrane.

The functional forms for the transition rates between conformational states of the sodium and potassium channels are given in Equations 20.20–20.22. These rates were discussed in more detail in Chapter 19, "Voltage-Gated Ion Channels".

$$\begin{aligned}
k_{1n} &= \frac{0.01 * (10 - V_M)}{\exp\left(\dfrac{10 - V_M}{10}\right) - 1} \\
k_{-1n} &= 0.125 * \exp\left(\frac{-V_M}{80}\right)
\end{aligned} \tag{20.20}$$

TABLE 20.1 Parameter Values for Hodgkin-Huxley Model

Parameter	Value
C_M	1 μF/cm^2
$\bar{g}_K$	36 μS/cm^2
$\bar{g}_{Na}$	120 μS/cm^2
g_L	0.3 μS/cm^2
E_K	–12 mV
E_{Na}	115 mV
E_L	10.613 mV

$$k_{1m} = \frac{0.1 * (25 - V_M)}{\exp\left(\frac{25 - V_M}{10}\right) - 1} \tag{20.21}$$

$$k_{-1m} = 4 * \exp\left(\frac{-V_M}{18}\right)$$

$$k_{1h} = 0.07 * \exp\left(\frac{-V_M}{20}\right)$$

$$k_{-1h} = \frac{1}{\exp\left(\frac{30 - V_M}{10}\right) + 1} \tag{20.22}$$

20.3. EXERCISES

Trying to write code to implement a set of equations such as Equation 20.19 while keeping track of all the rate functions and necessary parameters can seem daunting. The key to keeping larger coding projects manageable is to write many smaller functions first and then put them together to create larger functions until eventually the project is complete. For example, the following code is for a function **n_prime** that takes the current value of n and the current membrane potential V_m and returns the derivative of n according to the second equation of the Hodgkin-Huxley model:

```
function dn=n_prime(V_m, n)
%This function takes two arguments the membrane potential and the current
%value of the state variable n, and returns the value of the derivative of n for
%these values.
%First calculate the values of the forward and backward rate constants, k_1n and k_2n.

k_1n+0.01*(10-V_m)/(exp((10-V_m)/10)-1);
k_2n+0.125*exp(-V_m/80);

%Next calculate the value of the derivative.
dn+k_1n – (k_1n+k_2n)*n;
```

Exercise 20.1: Write a function **m_prime(*V_m, m*)** similar to the one in the preceding example that calculates the derivative of m given its current value and membrane potential.

Exercise 20.2: Write a function **h_prime(*V_m, h*)** similar to the one in Exercise 20.1 that calculates the derivative of *h* given its current value and membrane potential.

Exercise 20.3: Write a function **V_prime(*V_m, n, m, h, I_inj*)** that calculates the derivative of *V_m* given its current value, the values of the other state variables, and the injected current. *Hint:* Just repeat what you've done so far using the first equation of the Hodgkin-Huxley model.

20.4. PROJECT

In this project, you will model the voltage dynamics of a Hodgkin-Huxley neuron. You should perform the following:

1. Write a function **hodgkin_huxley(*t, I_inj*)** that takes a time series *t* and a constant representing injected current and returns the value of *V* at every point in *t*. Assume that the initial value for *V* is 10 mV. *Hint:* See Chapter 19, "Voltage-Gated Ion Channels," for a similar example.
2. Plot *V* versus *t* for injected currents of 5, 10, and 15 A/cm^2.
3. Determine what happens to the frequency of firing as the injected current increases.
4. Indicate how the action potential generated by this model compares to the result in Figure 20.1.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

length
for-loop
plot
hold on

CHAPTER

21 Models of the Retina

21.1. GOAL OF THIS CHAPTER

The goal of this chapter is to understand the basic structure of the retina and to see how to create simple models of neuronal interactions. In this chapter you will build a simple model describing the interaction between cone cells and horizontal cells of the retina and solve it exactly by taking advantage of the capability of the MATLAB® software to easily manipulate matrices.

21.2. BACKGROUND

21.2.1. Neurobiological Background

The retina is the part of the eye that transforms light into an electrochemical message sent to the brain for processing. The mechanism is quite complicated, but we will give a brief overview. Light first contacts the cornea, a transparent tissue covering the pupil and the iris. The cornea helps converge light through the pupil. Next, light passes through the lens, where it is further focused onto the retina in the back of the eye. The retina has five distinct classes of neurons arranged into cell layers. Light first contacts the innermost layers of the retina, but it is the outermost layer that first processes the incoming light signal. The layer responsible for processing the incoming light is composed mainly of two different cell types: cones and rods. Rods are mainly responsible for sensing brightness, and cones are responsible for detecting color.

This chapter pertains to cones, which we will discuss exclusively from now on. Mammals such as humans have three types of cones. Each type is adept at "seeing" a certain color: red, green, or blue. Cones have a G-protein coupled receptor on their cell surface called *rhodopsin*. This receptor is closely associated with a chromophore called *11-cis-retinal*. When a photon of light hits the chromophore, it isomerizes to *11-trans-retinal*. This conformational change is detected by the rhodopsin molecule, and a

G-protein is activated, which eventually closes ion channels on the cell membrane. The result is that ions (i.e., current) can no longer enter the cell, and the cell hyperpolarizes. Hyperpolarization decreases the cone's release of glutamate, a neurotransmitter that often has excitatory postsynaptic effects, which in turn decreases activity in postsynaptic cells in the next retinal layer. These postsynaptic cells are called *horizontal cells* (H-cells). H-cells normally maintain reciprocal synaptic connections with the cones that synapse onto them. H-cells release GABA, a neurotransmitter that has inhibitory postsynaptic effects. When the cones hyperpolarize in response to light and the H-cells decrease activity, the cones become disinhibited and begin to depolarize. This process is referred to as *negative feedback* because the initial light that induced hyperpolarization causes H-cells to feed back upon the cones in a way that counteracts the initial hyperpolarization. It is believed that this negative feedback is a regulatory mechanism to control color contrast. When the level of negative feedback of horizontal cells to cones is changed, the cones' response can be altered. Slight changes in feedback might be responsible for helping to determine changes in color. After all, although we have only three types of cones, humans can distinguish between millions of different colors! In addition to feeding back onto cones, horizontal cells also send signals to bipolar cells, which signal to amacrine cells, which finally signal to ganglion cells. The axons of these ganglion cells make up what anatomists call the *optic nerve*, the large nerve that connects the eye to the brain. The brain is then responsible for decoding the information sent from the retina to create what you "see" when you look at an object.

21.2.2. The Model

The model used in this chapter will be a system of two linear differential equations. The first will describe changes in the current leaving the cone of the retina, *C(t)*, and the second will describe the current leaving the horizontal cell, *H(t)*. We could build a larger system to account for the bipolar cells, amacrine cells, and ganglion cells, but we will keep it simple for now. The system is represented as follows:

$$\frac{dC}{dt} = \frac{1}{\tau_C}(-C - kH + L) \tag{21.1}$$

$$\frac{dH}{dt} = \frac{1}{\tau_H}(-H + C). \tag{21.2}$$

The first equation has three terms. The first indicates that the change in current is negatively proportional to the amount of current inside the cone, C. The second term represents the fact that the change in current is proportional to the current inside the horizontal cell, H, which negatively feeds back on the cell, and the third term indicates that the change in current into the cone is dependent on the light level, L. If the light level is high, then many photons will pass through the pupil, land on the retina, and activate the cones, resulting in a large change in current. The second equation states that the change in current in the horizontal cells depends negatively on the amount of current in the horizontal cells and the current of the cone cell that synapses onto the horizontal cell. Recall that the horizontal cells do not respond directly to light stimuli, so there is no term for the light intensity in the second equation. All other symbols in the preceding equations represent parameters (i.e., constants). Typical values for these parameters are $\tau_C = 0.025$ sec, $\tau_H = 0.08$ sec, and $k = 4$. Now also assume that the light intensity, L, is a constant, particularly $L = 10$. Finally, for the initial conditions, choose that $C(0) = H(0) = 0$. *There is no current moving through either cell at* t $= 0$.

The model equations as they are currently written can be simplified by a clever substitution. If you let:

$$\tilde{C} = C - \frac{L}{k+1} \quad \text{and} \quad \tilde{H} = H - \frac{L}{k+1}, \tag{21.3}$$

and substitute these equations into the previous equations, then you get:

$$\frac{d\tilde{C}}{dt} = \frac{1}{\tau_C}(-\tilde{C} - k\tilde{H}) \tag{21.4}$$

$$\frac{d\tilde{H}}{dt} = \frac{1}{\tau_H}(-\tilde{H} + \tilde{C}). \tag{21.5}$$

This is the model that you will study in its final form in this chapter. Note that the initial conditions now give:

$$\tilde{C}(0) = \tilde{H}(0) = \frac{L}{k+1} \tag{21.6}$$

21.2.3. Mathematical Background

Systems like the one in Equations 21.4 and 21.5 are especially suitable for study in MATLAB because they can be readily solved using simple matrix manipulations, as illustrated in the following simple example. Suppose you wanted to solve the system shown in Equations 21.7 and 21.8:

$$\frac{dx}{dt} = x + y \tag{21.7}$$

$$\frac{dy}{dt} = 4x + y \tag{21.8}$$

You begin by writing this system in matrix form to get:

$$\begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \tag{21.9}$$

If you let the vector

$$\begin{bmatrix} x \\ y \end{bmatrix} = \vec{v} \quad \text{and} \quad A = \begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix}, \tag{21.10}$$

then the system in Equations 21.7 and 21.8 becomes:

$$\frac{d\vec{v}}{dt} = A * \vec{v}. \tag{21.11}$$

Based on the Eigendecomposition Theorem (see Appendix B, "Linear Algebra Review"), you can substitute in for A to get the following equation:

$$\frac{d\vec{v}}{dt} = VDV^{-1} * \vec{v}. \tag{21.12}$$

Next, you multiply across the left by V^{-1} to get:

$$V^{-1}\frac{d\vec{v}}{dt} = V^{-1}VDV^{-1} * \vec{v} = DV^{-1} * \vec{v}. \tag{21.13}$$

If you let $V^{-1} * \vec{v} = \vec{u}$, then Equation 21.11 becomes:

$$\frac{d\vec{u}}{dt} = D * \vec{u} \tag{21.14}$$

This equation is similar to Equation 21.11 except for one very important exception: D is diagonal. The eigendecomposition of:

$$A = \begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix}$$

gives the eigenvalue matrix

$$D = \begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix}$$

and the eigenvector matrix

$$V = \begin{bmatrix} 1 & 1 \\ 2 & -2 \end{bmatrix}.$$

If you substitute in for D and convert Equation 21.14 into a system of equations, you get:

$$\frac{d\vec{u}}{dt} = \begin{bmatrix} \frac{du_1}{dt} \\ \frac{du_2}{dt} \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix} * \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 3u_1 \\ -u_2 \end{bmatrix} \Rightarrow \tag{21.15}$$

$$\frac{du_1}{dt} = 3u_1$$

$$\frac{du_2}{dt} = -u_2.$$

This system is also a system of differential equations. However, each equation can be solved independently of one another to yield the solution:

$$\vec{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} C_1 e^{3t} \\ C_2 e^{-t} \end{bmatrix}. \tag{21.16}$$

Finally, recall that you let $V^{-1}\,\vec{v} = \vec{u}$, so that $V\vec{u} = \vec{v}$, and:

$$\vec{v} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & -2 \end{bmatrix} * \begin{bmatrix} C_1 e^{3t} \\ C_2 e^{-t} \end{bmatrix} = \begin{bmatrix} C_1 e^{3t} + C_2 e^{-t} \\ 2C_1 e^{3t} - 2C_2 e^{-t} \end{bmatrix} = C_1 \begin{bmatrix} 1 \\ 2 \end{bmatrix} e^{3t} + C_2 \begin{bmatrix} 1 \\ -2 \end{bmatrix} e^{-t} \Rightarrow \tag{21.17}$$

$$x(t) = C_1 e^{3t} + C_2 e^{-t}$$

$$y(t) = 2C_1 e^{3t} - 2C_2 e^{-t}.$$

Notice where the eigenvalues and eigenvectors appear in the preceding solution. The eigenvalues, –1 and 3, appear in the exponents, and the eigenvectors appear as constant vectors multiplying the exponential with the corresponding eigenvalue. In general, the solution to any system of the form given in Equation 21.11 is:

$$\vec{v} = \begin{bmatrix} x \\ y \end{bmatrix} = C_1 * EV_1 * e^{\lambda_1 t} + C_2 * EV_2 * e^{\lambda_2 t} \tag{21.18}$$

where $\lambda_1$and λ_2 are distinct (not equal) eigenvalues of the matrix A, and EV_1 and EV_2 are the corresponding eigenvectors. If A has eigenvalues that are the same, then Equation 21.18 does not apply.

21.3. EXERCISES

You can see from Equation 21.18 that any set of equations that can be made into the form of Equation 21.11 can be solved by finding the eigenvalues and eigenvectors of the matrix A, as long as A has distinct eigenvalues. You can do this easily in MATLAB with the following command:

```
>>[V, D] = eig(A);
```

For example, typing the command

```
>>[V, D] = eig([1 1;4 1])
```

produces the response:

```
V =

   0.4472   -0.4472
   0.8944    0.8944
D =

   3.0000    0
      0    -1.0000.
```

The matrix D is a diagonal matrix with diagonal elements given by the eigenvalues of A. The columns of the matrix V correspond to the eigenvectors of A. The matrix D is the same as the one given earlier used to derive Equation 21.15, but the matrix V produced by MATLAB is different from the one used to derive Equation 21.17. Nonetheless, you still have the relationship that $A = VDV^{-1}$. You can check this by typing the command

```
>> V*D*inv(V)

ans =

   1.0000   1.0000
   4.0000   1.0000.
```

The **inv()** function determines the inverse of a matrix.

Reexamining Equation 21.18 reveals that in order to find $x(t)$ and $y(t)$, you still need to solve for C_1 and C_2. It can be shown that if you have the initial conditions that $x(0) = x_o$ and $y(0) = y_o$, then:

$$\begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = V^{-1} \begin{bmatrix} x_o \\ y_o \end{bmatrix} \tag{21.19}$$

Therefore, you can find these constants in MATLAB by typing the command

```
>>[c_1; c_2] = inv(V)*[x_o; y_o];
```

Finally, you can create v according to Equation 21.18 by typing:

```
>>v = c_1*V(:, 1)*exp(D(1,1)*t)+c_2*V(:, 2)*exp(D(2,2)*t);
```

for some previously defined vector t. You can access the separate solutions x and y with $v(1, :)$ and $v(2, :)$, respectively.

Exercise 21.1: Use MATLAB to check that if you let $A = [1\ 1;\ 4\ 1]$, $V = [1\ 1;\ 2\ {-2}]$, and $D = [3\ 0;\ 0\ {-1}]$, then $A = VDV^{-1}$ also holds.

Exercise 21.2: Put the system of equations in Equations 21.4 and 21.5 into matrix form. What is the matrix A?

21.4. PROJECT

In this project, you will solve the retinal feedback system described previously. Specifically, you should do the following:

1. Write a function **solution*(A, init)*** that takes a matrix with distinct eigenvalues and a set of initial conditions, and plots the solutions *x(t)* and *y(t)* on the same graph. The function should return an output message if the input matrix does not have distinct eigenvalues. *Hint:* See the **error()** function help for producing output messages for functions that you want to throw an error message under certain conditions.
2. Use the function **solution*(A, init)*** to plot the solution of the retinal feedback system. In low light levels ($L = 3$), it takes longer for cells to respond. The measured parameters under low light level conditions are $\tau_C = 0.1$ sec, $\tau_H = 0.5$ sec, and $k = 0.5$. Plot the solution to the system with these parameters and compare with the previous plots. Under which conditions do the cones respond more strongly? Does this make sense?

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

eig
inv
error

CHAPTER

22

Simplified Model of Spiking Neurons

22.1. GOAL OF THIS CHAPTER

The goal of this chapter is to study a computationally efficient spiking cortical neuron model first introduced by Izhikevich (2003), and to generalize this model to a network of neurons. Ultimately, you will obtain and examine a raster plot of modeled network activity.

22.2. BACKGROUND

The task of understanding how different areas of the brain interact with each other to perform higher level functions such as motor coordination and speech is a major interest of modern neuroscience but also an extremely difficult one. Many factors contribute to the global dynamics of neural networks. First, neurons isolated from a network exhibit a variety of patterns and behaviors. Some examples include regular spiking neurons, fast spiking neurons, intrinsic bursting neurons, and subthreshold membrane oscillations, as shown in Figure 22.1. Some of these behaviors are more common than others. For example, under normal conditions there are more regular spiking neurons in the cortex than intrinsic bursting ones. How these different dynamics are manifested at the network level remains an open area of current research. Second, the synaptic coupling between neurons can have a large impact on the network's dynamics leading to synchronization among the neurons and network oscillations.

Network oscillations in the brain are often categorized by their frequency. Oscillations with a frequency less than 4 Hz are called *delta rhythms*. Oscillations between 4 and 8 Hz are called *theta rhythms*. Rhythms from 8 to 12 Hz are called *alpha rhythms*, and rhythms from 12 to 30 Hz are called *beta rhythms*. Rhythms above 30 Hz are called *gamma rhythms*.

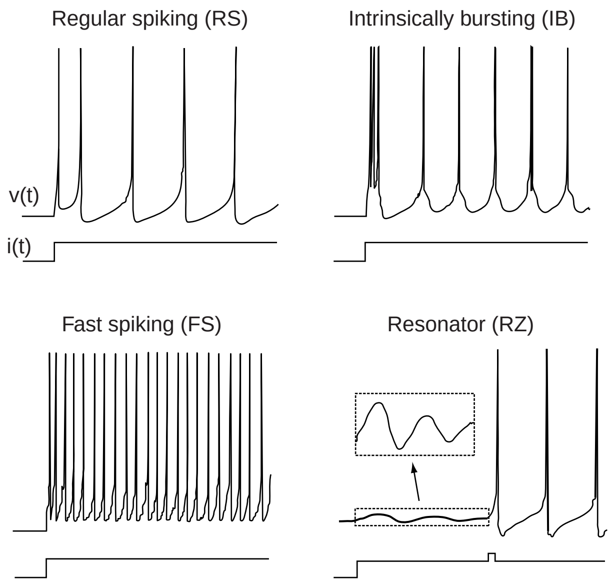


FIGURE 22.1 Known types of neurons. (An electronic version of the figure and reproduction permissions are freely available at www.izhikevich.com.)

22.2.1. The Model

The model for this chapter is a two-dimensional system of ordinary differential equations with a reset condition, as shown in Equations 22.1–22.3:

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \tag{22.1}$$

$$\frac{du}{dt} = a(bv - u) \tag{22.2}$$

The reset condition is:

$$\text{if}: v \geq 30 \quad \text{then:} \quad \begin{cases} v \leftarrow c \\ u \leftarrow u + d. \end{cases} \tag{22.3}$$

The variable v represents the membrane potential of the neuron while u represents a generic recovery variable that feeds back negatively onto v. There are five additional parameters in the model: I, a, b, c, and d. The parameter I represents external input to the neuron. This input could be thought of as external input to the neuron from outside the network or even synaptic input from a neuron within the network. The parameter a controls the rate of recovery of u, and b controls the sensitivity of recovery to subthreshold fluctuations of the membrane potential. The parameters c and d control the after-spike reset values for v and u, respectively. If you choose certain parameter combinations, this simple model can exhibit all the firing patterns and behaviors shown in Figure 22.1.

To model a whole network of neurons, you will have to couple many neurons together where each neuron behaves according to Equations 22.1–22.3. This means that you will have to select a value for each parameter for every neuron you model. Additionally, since the network is coupled (i.e., the neurons are connected to each other), the input *I* to a particular neuron will depend on other neurons in the network that synapse onto it. Therefore, you will need to model the connectivity of the network. You could choose to make every neuron in the network be connected to every other neuron in the network, or possibly make neurons connect only to neurons that are close. Additionally, you need to choose whether a connection between neurons will be excitatory or inhibitory and how strong the connection will be.

22.3. EXERCISES

The code for implementing Equations 22.1–22.3 are not very complicated. You can solve them using Euler's method, as introduced in Chapter 19, "Voltage-Gated Ion Channels." The script for implementing a single neuron is as follows (adapted from Izhikevich, 2003):

```
%These are some default parameter values
I=10;
a=0.02;
b=0.2;
c=-65;
d=8;
%The initial values for v and u
v=-65;
u=b*v;
%Initialize the vector that will contain the membrane potential time series.
v_tot=zeros(1000, 1);

for t=1:1000
  %set v_tot at this time point to the current value of v
  v_tot(t)=v;
  %Reset v and u if v has crossed threshold. See Eq. 3 above.
  if (v >= 30)
    v=c;
    u=u+d;
  end;
  %Use Euler's method to integrate Eqs. 1 and 2 from above. Here v is
  %calculated in 2 steps in order to keep the time step small (0.5 ms step in the
  %line below).
  v=v+0.5*(0.04*v^2+5*v+140-u+I);
  v=v+0.5*(0.04*v^2+5*v+140-u+I);
  u=u+a*(b*v-u);
end;
```

```
%This line uses the function find to locate the indices of v_tot that hold elements
%with values greater than or equal to 30 and then sets these elements to 30.
%This normalizes to heights of the action potential peaks to 30.
v_tot(find(v_tot >= 30))=30;
%Plot the neuron's membrane potential.
plot(v_tot);
```

Exercise 22.1: Before going on to generalize this model to a network of neurons, you should explore this model thoroughly. See if you can discover what parameter sets lead to regular spiking, fast spiking, or intrinsically bursting behavior. What kinds of behaviors do the following parameter sets produce?

a. $[a, b, c, d] = [0.02, 0.2, -65, 8]$
b. $[a, b, c, d] = [0.02, 0.2, -55, 4]$
c. $[a, b, c, d] = [0.1, 0.2, -65, 2]$
d. $[a, b, c, d] = [0.1, 0.25, -65, 2]$

Now consider how to modify this script to model a network of neurons where each neuron is described by the dynamics of Equations 22.1–22.3. First, convert the parameters from numbers to vectors. The vectors will hold the value of each parameter for each neuron in the network. Since you want some neurons in the network to be regular spiking and others to be intrinsically bursting, you will need to have different values for the elements of the vectors. The modified code should begin as follows:

```
% The number of excitatory neurons in the network. The mammalian cortex has
% about 4 times as many excitatory neurons as inhibitory ones.
Ne=800;
%The number of inhibitory neurons in the network.
Ni=200;
%Random numbers
re=rand(Ne, 1);
ri=rand(Ni, 1);
%This will set the value of a for all excitatory neurons to 0.02 and the value of a
%for inhibitory neurons to a random number between 0.02 and 0.1
a=[0.02*ones(Ne, 1); 0.02+0.08*ri];
%This will allow b to range from 0.2–0.25
b=[0.2*ones(Ne, 1); 0.25–0.05*ri];
%This will allow the spike reset membrane potential to range between -65 and -%50
c=[-65+15*re.^2; -65*ones(Ni,1)];
%This will allow the recovery reset value to range between 2 and 8
d=[8–6*re.^2; 2*ones(Ni, 1)];
⋮
```

Before you continue with the code, it is worthwhile to consider how these definitions of the parameters impact the composition of the network.

Exercise 22.2: What parameter sets are most neurons likely to possess? Is there any correlation between excitatory neurons as represented in this model and regular spiking neurons, for example?

The next line of code should create a weight matrix that holds the strength of the connectivity between every pair of neurons in the network. Since the network has $Ne + Ni$ neurons, then the weight matrix will be a square matrix with these dimensions. The code to implement this is:

```
S=[0.5*rand(Ne+Ni, Ne), -rand(Ne+Ni, Ni)];
```

Notice that this definition allows the strength of connections of excitatory neurons onto other neurons to range from 0 to 0.5, whereas inhibitory neurons have a synaptic strength between 0 and –1. According to this definition, a single inhibitory neuron can have, in general, a stronger effect on the neurons it contacts than a single excitatory neuron, which is supported by current experimental research. Also notice that very few elements of S will be exactly 0, so in this model almost every neuron has synaptic contacts with all other neurons in the network. The rest of the code for the network model is:

```
%The initial values for v and u
v=-65*ones(Ne+Ni,1);
u=b.*v;
%Firings will be a two-column matrix. The first column will indicate the time that a
%neuron's membrane potential crossed 30, and the second column will be a number
%between 1 and Ne+Ni that identifies which neuron fired at that time.
%firings=[];

for t=1:1000
   %Create some random input external to the network
   I=[5*randn(Ne, 1); 2*randn(Ni,1)];
   %Determine which neurons crossed threshold at the current time step t.
   fired=find(v > =30);
   %Add the times of firing and the neuron number to firings.
   firings=[firings; t*ones(1, length(fired)), fired];
   %Reset the neurons that fired to the spike reset membrane potential and
   %recovery variable.
   v(fired)=c(fired);
   u(fired)=u(fired)+d(fired);
   %Add to the input, I, for each neuron a value equal to the sum of the synaptic
   %strengths of all other neurons that fired in the last time step connected to that
   %neuron.
   I=I+;sum(S(:,fired), 2);
```

```
    %Move the simulation forward using Euler's method.
    v=v+0.5*(0.04*v^2+5*v+140-u+I);
    v=v+0.5*(0.04*v^2+5*v+140-u+I);
    u=u+a*(b*v-u);
end;
%Plot the raster plot of the network activity.
plot(firings(:,1), firings(:,2),'.');
```

22.4. PROJECT

In this project, you will examine the behavior of a cortical network of spiking neurons. Specifically, you are asked to do the following:

- According to the definition of the parameter values for *c* in the network model, determine whether an inhibitory neuron can be an intrinsically bursting neuron. Will there be more regular spiking neurons in the network or intrinsically bursting neurons?
- Examine the raster plot produced by the preceding code. Are there any oscillations present in the network? If so, are they delta rhythms, theta rhythms, alpha rhythms, etc.?
- Modify the code by redefining *c* and *d* to allow for more bursting neurons to be present in the network. What effect, if any, does this have on the presence of network oscillations?
- Alter the weight matrix so that there are fewer connections between neurons of the network. What effect does this have on the network dynamics?

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

rand
randn
plot
find

CHAPTER

23

Fitzhugh-Nagumo Model: Traveling Waves

23.1. GOALS OF THIS CHAPTER

The purpose of this chapter is to learn how to model traveling waves in an excitable media. This entails the solution of a partial differential equation involving a first derivative in time coordinates and a second derivative in spatial coordinates. You will learn how to compute a second derivative in the MATLAB® software and use a modification of the Fitzhugh-Nagumo model introduced in Chapter 12, "Exploring the Fitzhugh-Nagumo Model," to generate traveling waves in both one and two dimensions.

23.2. BACKGROUND

The Fitzhugh-Nagumo model is often used as a generic model for excitable media because it is analytically tractable. You will use it as a simple model to generate traveling waves by the addition of a diffusion term: a second derivative in spatial coordinates. In this chapter you will modify the Fitzhugh-Nagumo model introduced in Chapter 12 in this way and study its behavior in one and two dimensions. In this way you can simulate action potential wave propagation along the axon of a single neuron or the spreading of electrical potential waves in a network of cortical neurons.

There are many forms of the equations for the voltage, v, and recovery, r, variables in the Fitzhugh-Nagumo model. In general they are given by:

$$\frac{\partial v}{\partial t} = f(v) - r + I + \frac{\partial^2 v}{\partial x^2} \tag{23.1}$$

$$\frac{\partial r}{\partial t} = av - br \tag{23.2}$$

The function $f(v)$ is a third order polynomial that provides positive feedback, whereas the slower recovery variable r provides negative feedback. By making the voltage and recovery variables functions of spatial coordinates as well as time, you can model dynamics in a spatially extended regime. The final term in the first equation introduces diffusion into the system, and thus the first equation is known as a *reaction-diffusion equation*.

A note of caution: As with ordinary differential equations, whenever you attempt to solve partial differential equations computationally, you must be careful that the various errors that can be introduced, such as truncation errors and roundoff errors, are not significant and that the necessary conditions for stability are met. See Strauss (1992) for a more in-depth discussion of such matters. If you are not careful, then the solutions produced by your code may stray quite significantly from the true solutions you seek.

23.3. EXERCISES

23.3.1. Second Derivative Operator

How do you model a second derivative computationally in MATLAB? There are a few approaches to this, but the one you will use here is the simplest computational approximation known as the *centered second difference:*

$$\frac{d^2v(x)}{dx^2} \sim \frac{v(x+\Delta x) - 2v(x) + v(x-\Delta x)}{(\Delta x)^2} \tag{23.3}$$

This approach can be justified by combining the Taylor expansions for $v(x+\Delta x)$ and $v(x-\Delta x)$ (Strauss, 1992). If the mesh size of the spatial variable is represented by Δx, then the j^{th} element of the array v, v_j, is the value of v for $x = j\,\Delta x$, so you have:

$$\frac{d^2v_j}{dx^2} \sim \frac{v_{j+1} - 2v_j + v_{j-1}}{(\Delta x)^2} \tag{23.4}$$

The second derivative can thus be computed by convolving the array v with the second derivative operator filter $F = [1\ -2\ 1]/(\Delta x)^2$. This can be extended to two dimensions as well. Assuming equal mesh spacing along both directions ($\Delta x = \Delta y$), then the two-dimensional second derivative operator filter is given by $F = [0\ 1\ 0;\ 1\ -4\ 1;\ 0\ 1\ 0]/(\Delta x)^2$.

Now create a function in MATLAB for the second derivative operator in one dimension and name it **secDer.m**. Its input will be the one-dimensional array *v(x)*, the spatial mesh size, *dx*, and the output will be the second derivative, $v''(x)$. This function will use the convolution function **conv**, which, by default, introduces undesirable edge effects. Also you will then include an option to improve the edge effects by making the boundary conditions periodic. You'll do this by adding a third input to the function, *BC*, which, if set to 1, will return the default **conv** output found by padding the input matrix with zeros, also known as *free boundary conditions*, and if set to 2, then you will have *periodic boundary conditions*. Periodic boundary conditions means that the boundaries of the input array (i.e., the first and last elements) are considered neighboring points. You could use the **if, elseif** control structure to carry out the options for the boundary conditions, but instead we will introduce you to another useful control structure that MATLAB offers: **switch**.

```
function V=secDer(v,dx,BC)
%
%F is the discrete 2nd derivative filter in 1D
F = [1 -2 1 ]/dx^2;
%
%BC determines your boundary conditions
switch BC
  case 1 %free bc's
    Vconv(v,F);
    V=V(2:end-1); %return an array the same size as the input array
  case 2 %periodic bc's
    %since the convolution filter is of length 3 then we only have to
    %pad the input array v by 1 element on either side
    pv=zeros(1,length(v)+2); %extend the input array by 2
    pv(2:end-1) = v;
    %now we fill in these two padded points with the values that the extended
    %input array would have if the first element of v and the last element of v
    %were neighbors
    pv(1)=v(end);
    pv(end)=v(1);
    V=conv(pv,F);
    V=V(3:end-2); %return the valid portion of the convolution
end
```

Give this a try and see how it works by testing it on a function whose second derivative is well known: cosine. If $f(x) = \cos(x)$, then $f''(x) = -\cos(x)$. You can compare the output of the second derivative function, **secDer,** with the analytic solution by running the following script, whose output is shown in Figure 23.1.

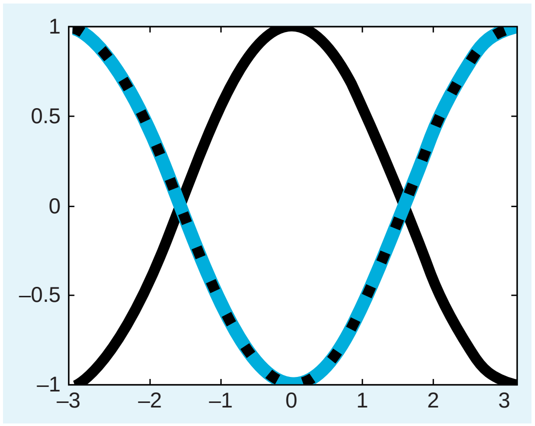


FIGURE 23.1 Testing the second derivative function **secDer.m**. The solid black line is the input, $\cos(x)$, and the blue line is the output of the **secDer** function with periodic boundary conditions. This matches exactly with the analytic solution to the second derivative, $-\cos(x)$, shown as the dotted black line.

```
x=linspace(-pi,pi,100); %forces even spacing in array of 100 pts from –pi to pi
dx=x(2)-x(1); %determines this spacing, the spatial mesh size
x=x(2:end); % for periodicity knock of 1st term in x array so that we don't have repeat
            % value of cos(x) at the endpoints of x (since cos(-pi)=cos(pi))
f=cos(x); %input array
d2f=secDer(f,dx,2); %computational solution to the second derivative of f with
periodic BC
d2fA=-cos(x); %analytic solution to the second derivative of f
plot(x,f,'k','LineWidth',3) %plot input f
hold on
plot(x,d2f,'b','LineWidth',5) %plot computational result of f"
plot(x,d2fA,'k:','LineWidth',3) %plot analytic result of f"
axis([-pi pi -1 1]); set(gca,'fontsize',20)
```

Exercise 23.1: Compare this with the result you get if you use free boundary conditions. You can do this by setting *BC* to 1 when calling the **secDer** function. Remove or comment out the last line of the script above, which sets the limits for the axes in the plot, so that you can see the effect of changing the boundary conditions.

Exercise 23.2: Rewrite the **secDer.m** function using the **if, elseif** control structure rather than the **switch** control structure. Test your function by using ***f*** **= sin(*x*)** and compare your result with the known solution ***f*** **″= -sin(*x*)**.

With this second derivative operator, you can now model a traveling wave in 1D. For the one-dimensional problem, you will use the following form of the Fitzhugh-Nagumo equations (Wilson, 1999):

$$\frac{\partial v}{\partial t} = 10\left[v - \frac{1}{3}v^3 - r + D\frac{\partial^2 v}{\partial x^2}\right] + I \tag{23.5}$$

$$\frac{\partial r}{\partial t} = p[a + 1.25v - br] \tag{23.6}$$

The variables $v(x,t)$ and $r(x,t)$ are the voltage and the recovery variables at position x at time t. If modeling a pulse traveling along a nerve fiber, you can think of x as the position along the nerve fiber. Similarly, if you want to model a traveling wave of activity across a one-dimensional network of neurons, then x indicates the neuron location in the one-dimensional population. Now consider the latter case. You will use the following parameter values: $D = 1$, $a = 1.5$, $b = 1$, and $p = 0.8$. For this set of parameters, the steady state values are $v0 = -1.5$ and $r0 = -3/8$, which you will use as your initial conditions for these variables. The driving stimulus is given by I. To solve this system, you are going to need an ODE solver.

23.3.2. Built-in ODE Solvers

In previous chapters you solved differential equations through manually written ODE solvers using the Euler method and the Runge-Kutta method, which had the advantage of complete transparency in the mechanisms behind the operations. In this chapter we will introduce you to the most practical and commonly used of the built-in ODE solvers in MATLAB: the function **ode45**. This solver is based on an explicit Runge-Kutta formula and has been optimized to adaptively find the most efficient time steps to produce a solution within a certain allowed relative error tolerance (10^{-3} by default) and absolute error tolerance (10^{-6} by default). Look at the help section for **ode45** for more information on how to adjust these options as well as to learn about the other built-in ODE solvers offered by MATLAB and the conditions under which to use them.

To familiarize yourself with the proper syntax for using the **ode45** ODE solver, first consider the simpler case of solving this system of equations for one point in space (i.e., for one neuron.) First, you must code the system of first-order ODEs as a function that the solver can use. You will represent the Fitzhugh-Nagumo system in a function called **F_N1**. The **F_N1** function assumes that v and r become elements $V(1)$ and $V(2)$ of the two-element input vector V. Although t and V must be the function's first two arguments, the function does not need to use them. The output *vdot*, the derivative of V, must be a column vector, as shown in the following code:

```
function vdot = F_N1(t,V)
%
%set parameters of the model
a=1.5; b=1; p=.08; I=1.5;
%dv/dt:
vdot(1) = 10*(V(1) - (V(1).^3)/3 - V(2) + I);
%dr/dt
vdot(2) = p*(1.25*V(1) + a - b*V(2));
vdot=vdot'; %make correct dimensions for ODE solver: must be a column vector
```

Note that the diffusion term is left out, since there is only one point in space and a spatial derivative makes no sense in this case. For this one neuron system, you set the input to 1.5 so that the model will initiate a series of action potentials. You can then generate and plot the solution as follows:

```
v0=[-1.5;-3/8];  %initial conditions for V variable
tspan=[0 100]; %beginning and end values of time
[t,v] = ode45('F_N1', tspan, v0);
plot(t,v(:,1),'k*','LineWidth',5);
```

Now look at what was produced by running the preceding script:

```
>> whos
Name        Size     Bytes    Class

t         2597x1     20776    double
tspan        1x2        16    double
v         2597x2     41552    double
v0           2x1        16    double
```

Note that the time, which goes from 0 to 100, is a column vector of 2597 points. These time points are not evenly distributed between 0 and 100; rather, the mesh size varies and has been selected by the solver to most efficiently compute the differential equation within the tolerated error. For example, consider how the spacing between time points varies in just the first 10 time points:

```
>> t(1:10)

ans =

         0
    0.0015
    0.0031
    0.0046
    0.0061
    0.0138
    0.0214
    0.0291
    0.0367
    0.0451

>> diff(t(1:10))

ans =

    0.0015
    0.0015
    0.0015
    0.0015
    0.0077
    0.0077
    0.0077
    0.0077
    0.0084
```

The first column of the output vector, *v(:,1)*, represents the voltage values at the corresponding times starting with *v0(1)* at the first time point. Similarly, the second column vector of the output, *v(:,2)*, represents the recovery variable values for the corresponding times, starting with *v0(2)*.

When you specify specific time points in **tspan,** the ODE solver will still use its most efficient time mesh to solve the differential equation; however, it will now return the values of the outputs at the specific times indicated in **tspan**. Compare the previous result with that found by specifying the time to be from 0 to 100 at intervals of 4:

```
hold on
tspan=[0:4:100];
[t,v] = ode45('F_N1',tspan,v0);
plot(t,v(:,1),'b*','LineWidth',5)
```

This result is shown in Figure 23.2.

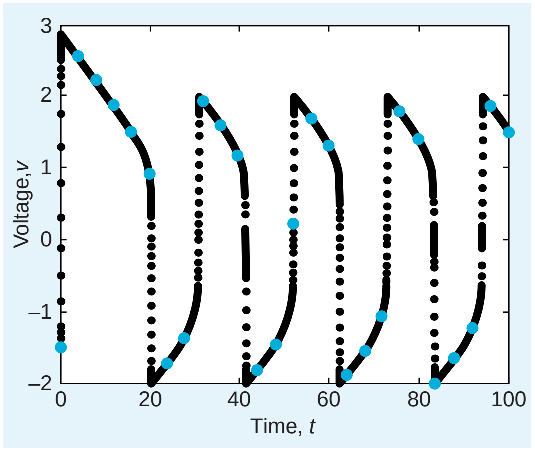


FIGURE 23.2 Voltage, v, versus time, t, output of Fitzhugh-Nagumo system of equations for one point in space found using the **ode45** solver with **tspan**=**[1 100]** (black dots) and with **tspan**=**[1:4:100]** (blue dots.)

23.3.3. Fitzhugh-Nagumo Traveling Wave

You are now ready to tackle the full problem of simulating a propagating wave along a line of neurons. Let the number of neurons be given by N. A naïve approach might be to allow the initial conditions to be a $2 \times N$ matrix with the first row representing the N initial voltages and the second row the N values of the initial recovery variables. However, recall that the **ode45** solver will accept only a single column array for the initial conditions. What you will do to satisfy this requirement is let the initial value column vector be of length $2N$ and let the first N elements represent the initial voltages of the N neurons and the last N elements represent the initial recovery values. The ODE solver will produce as its output a $t \times 1$ time vector, and a $t \times 2N$ matrix, whose first N columns represent the evolution of the voltage of the population of neurons as time progresses and whose second N columns represent the evolution of the recovery variables.

The stimulus to initiate the wave that you will use is $I = 6$ for the first 0.5s; then the stimulus will be off, $I = 0$, for the rest of the time. You can choose where along the line of neurons to initiate the wave. In the following example, you stimulate the center cells. The following script, **FNmain.m**, will produce a traveling wave of activity along a one-dimensional population of N neurons whose dynamics are governed by the Fitzhugh-Nagumo equations, as shown in Figure 23.3.

```
%FNmain.m
clear all; close all
%
global N I BC %by making these variables global they can exist within
%the workspace of functions without explicitly being input to the functions
N=128; %number of neurons
v0(1:N)=-1.5; %initial conditions for V variable
v0(N+1:2*N)=-3/8; %initial conditions for R variable
```

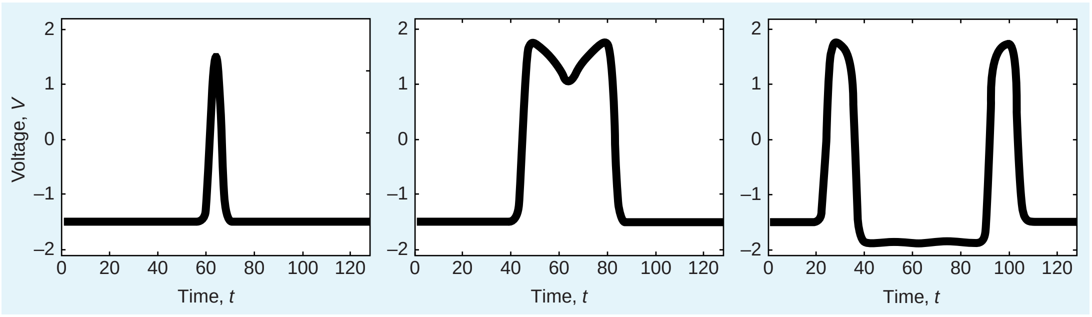


FIGURE 23.3 Traveling Fitzhugh-Nagumo wave in one dimension for $t = 1$ (left), $t = 5$ (center), and $t = 10$ (right).

```
I=6; %the input stimulus value
BC = 2; %set to 1 (free) or 2 (periodic boundary conditions)
%
tspan=[0:.1:.5]; %time with stimulus
[t1,v1] = ode45('F_N',tspan,v0);
tspan=[.5:.1:25]; % time without stimulus
I=0;%turn off stimulus
[t2,v2] = ode45('F_N',tspan,v1(end,:)'); %note: initial cond are final v1 values
%piece together (concatenate) time (t1 and t2) and solution (v1 and v2)
%variables without double counting the seam values
t=[t1; t2(2:end)]; v=[v1; v2(2:end,:)];
%spacetime plot of v variable w/ neurons along y axis, time along x-axis
figure(1); imagesc(v(:,1:N)') ; colorbar
%spacetime plot of r variable w/ neurons along y axis, time along x-axis
figure(2); imagesc(v(:,N+1:end)'); colorbar
%create a movie of the traveling wave
figure(3)
for i=1:length(t)
   plot(v(i,1:N))
   axis([0 N -2.1 2.2])
   pause(.05)
end
```

When you run the preceding **FNmain.m** script, make sure that this M-file is in the same directory as the **secDer** function previously created as well as the following function **F_N**, since the main script calls these functions. The function **F_N** describes the coupled system of differential equations used to model the dynamics.

```
function vdot = F_N(t,v)
global N I BC
%
%set parameters of the model
```

```
D=10; a=1.5; b=1; p=.08;
%dv/dt:
vdot(1:N) = 10*(v(1:N) - (v(1:N).^3)/3 - v(N+1:end)) + D*secDer(v(1:N),1,BC)';
%add input I to the center five cells
vdot(round(N/2)-2:round(N/2)+2) = vdot(round(N/2)-2:round(N/2)+2) + I;
%dr/dt:
vdot(N+1:2*N) = p*(1.25*v(1:N) + a - b*v(N+1:end));
%
vdot=vdot'; %make correct dimensions for ODE solver
```

Exercise 23.3: Use periodic boundary conditions, as in the preceding example, but change the location of the wave initiation to some point off center. For example, rather than stimulate the center cells, stimulate cells a fourth of the way from the edge. Watch the two waves initiated annihilate each other.

Exercise 23.4: Change the boundary conditions to make them free boundary conditions and start the wave at the left end of the array to create a traveling wave that goes from left to right. Play with the parameters to see how they affect the dynamics.

23.4. PROJECT

In this project, you will simulate a traveling wave of activity in a two-dimensional $N\times N$ array of cortical neurons. This time you will use these versions of the Fitzhugh-Nagumo equations (Murray, 2002):

$$\frac{\partial v}{\partial t} = -v(a-v)(1-v) - r + D\frac{\partial^2 v}{\partial x^2} \tag{23.7}$$

$$\frac{\partial r}{\partial t} = bv - gr \tag{23.8}$$

with the parameters taking on the values $a = 0.25$, $b = 0.001$, $g = 0.003$, and $D = 0.05$. You will compute the solution using a similar method as that used for the one-dimensional problem. This time the initial value column vector will be of length $2N^2$, where the first N^2 elements will represent the initial voltages of the $N\times N$ neuron array and the last N^2 elements will represent the initial recovery values. The ODE solver will produce as its output a $t\times 1$ time vector, and a $t\times 2N^2$ matrix, whose first N^2 columns represent the evolution of the voltage of the population of neurons as time progresses and whose second N^2 columns represent the evolution of the recovery variables. For a given row of this output matrix, i.e., a particular time point, you can reconstruct the $N\times N$ array of voltage variables from the $1\times N^2$ array using the *reshape* function, whose input is the matrix to be reshaped as well as the number of rows and columns desired in the output.

```
>> A=[1 2 3 4 5 6 7 8 9]

A =

    1   2   3   4   5   6   7   8   9

>> a=reshape(A,3,3)

a =

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

You will need to do this when you call the two-dimensional second derivative function as well as when you wish to visualize the results. There are more elegant and efficient ways to handle the issue of programming the two-dimensional partial differential equation, but here we will present a way to solve the problem in MATLAB that is conceptually simple, allowing you to use the tools previously used while not introducing any more complicated commands.

In the following script, **FN2main.m**, you will create an .avi file of the simulation and name it **TravelingWave.avi**:

```
%FN2main.m
clear all; close all
% next 4 lines are necessary to create the movie file of the wave
fig=figure;
set(fig,'DoubleBuffer','on');
set(gca,'NextPlot','replace','Visible','off')
mov = avifile('TravelingWave.avi')
%
tspan=[0:5:800]; %simulation time
global N BC
BC=1; %boundary conditions: 1 free, 2 periodic
N=32; %number of neurons
v0(1:N^2)=0; %initial conditions for V variable
v0(N^2+1:2*N^2)=0; %initial conditions for R variable
v0(1:N) =.6; %initially stimulate all cells along left edge of population
[t,v] = ode45('F_N2',tspan,v0);
%obtain min and max values of output
clims = [min(min(v(:,1:N^2))) max(max(v(:,1:N^2)))];
%generate the movie of the voltage
for i=1:size(v,1)
   figure(1)
   im = imagesc(reshape(v(i,1:N^2),N,N), clims);
   axis square
   set(im,'EraseMode','none');
   Frame=getframe(gca);
   mov = addframe(mov,Frame);
```

```
end
mov = close(mov);
```

Specifying the same *clims* in the **imagesc** option for each time frame of the imaged voltage array ensures that the same colormap is used throughout the movie, which is analogous to using consistent z-axis limits. The preceding main script calls on a number of functions that need to be created and stored in the same directory as the main script. These functions, **F_N2, SecDer2**, and **conv2periodic**, follow. Note that you will need to complete the **F_N2** function. The **SecDer2** function is just an extension of the **SecDer** function to two dimensions and employs the reshape function to carry out the 2D convolutions. The **conv2_ periodic** function has been written in a generic form so that it can handle input matrices of various sizes for future applications.

```
function vdot = F_N2(t,v)
global N BC
%
D=.05; a=0.25; b=.001; g=.003; %set the parameters
%
%dV/dt
vdot(1:N^2)=-(v(1:N^2)).*(a-(v(1:N^2))).*(1-(v(1:N^2)))-...
   v(N^2+1:end)+D.*secDer2(v(1:N^2),1,BC);
%dR/dt
vdot(N^2+1:2*N^2)=???
%
vdot=vdot';
```

```
function V=secDer2(v,dx,BC)
global N
%
F = [0 1 0; 1 -41; 0 1 0]/dx^2;
%determines your boundary conditions
switch BC
   case 1 %free bc's
     V=conv2(reshape(v,N,N)',F,'same');
     V=reshape(V',N*N,1);
   case 2 %periodic bc’s
     V=conv2_periodic(reshape(v,N,N)',F);
     V=reshape(V',N*N,1);
end
```

```
function sp = conv2_periodic(s,c)
% 2D convolution for periodic boundary conditions.
% Output of convolution is same size as leading input matrix
[NN,M]=size(s);
[n,m]=size(c); %% both n & m should be odd
%enlarge matrix s in preparation convolution with matrix c via periodic edges
padn = round(n/2) - 1;
```

```
padm = round(m/2) - 1;
sp=[zeros(padn,M+(2*padm)); ...
        zeros(NN,padm) s zeros(NN,padm); zeros(padn,M+(2*padm))];
%fill in zero padding with the periodic values
sp(1:padn,padm+1:padm+M)=s(NN+1-padn:NN,:);
sp(padn+1+NN:2*padn+NN, padm+1:padm+M)= s(1:padn,:);
sp(padn+1:padn+NN,1:padm)= s(:,M+1-padm:M);
sp(padn+1:padn+NN,padm+M+1:2*padm+M)= s(:,1:padm);
sp(1:padn,1:padm)= s(NN+1-padn:NN,M+1-padm:M);
sp(padn+NN+1:2*padn+NN,1:padm)= s(1:padn,M+1-padm:M);
sp(1:padn,padm+M+1:2*padm+M)=s(NN+1-padn:NN,1:padm);
sp(padn+NN+1:2*padn+NN,padm+M+1:2*padm+M)=s(1:padn,1:padm);
%
%perform 2D convolution
sp = conv2(sp,c,'same');
% reduce matrix back to its original size
sp = sp(padn+1:padn+NN,padm+1:padm+M);
```

In this project, you should do the following:

- Create the main script and the functions given here in the same working directory. Complete the **F_N2** function by replacing the symbols **???** with the proper quantities so that **FN_2** implements the Fitzhugh-Nagumo model equations as given in this section. Run the main script to generate a plane wave from the left, as shown in Figure 23.4.
- Alter the code so that the wave is initiated from one of the corners or in the center of the array and forms a propagating ring. Also try running the simulation with periodic boundary conditions. Play with the various parameters to see how they affect the wave dynamics.
- Now use the model to generate a spiral by resetting the upper half of the voltage and recovery variables to 0 when the traveling plane wave is approximately halfway across the neural network. Start with the original code, as in the first part of the project so that a plane wave is initiated along the entire left edge of the array. In the main script, change the name of the .avi file that will be created to **SpiralWave.avi**. Let N = 60 and

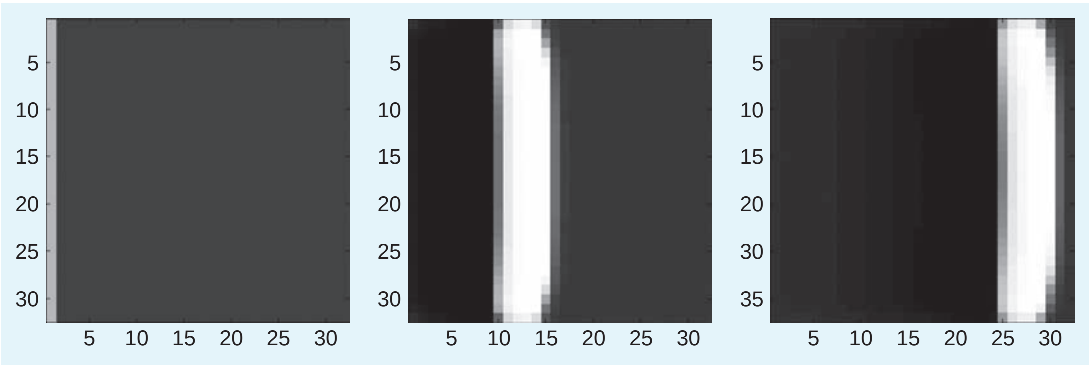


FIGURE 23.4 Two-dimensional traveling wave produced by the Fitzhugh-Nagumo equations for $t = 0$ (left), $t = 300$ (center), and $t = 600$ (right).

tspan =[0:5:1800]. This will cause the simulation to take more time to run but will allow you to view the spiral well.

- In the main script, create a variable *Reset* that can take on the value 0 or 1. Use the **switch** control structure so that when *Reset* is equal to 0, it runs the original script, and when it is equal to 1, the line
[t,v] = ode45('F_N2',tspan,v0);
will not be executed, and in its place the following lines will be executed:
[t1,v1] = ode45('F_N2',tspan(1:round(length(tspan)/3)),v0);
vR=Vreset(v1(end,:));
[t2,v2] = ode45('F_N2',tspan(round(length(tspan)/3):end),vR);
v=[v1; v2(2:end,:)];
t=[t1; t2(2:end,:)];
Create the following **Vreset** function in the same directory as the main file:

```
function vR=Vreset(v)
global N
%
%reset half of voltage variables to zero
VR = reshape(v(1:N^2),N,N)';
VR(:,1:round(N/2))=0;
%reset half of recovery variables to zero
RR = reshape(v(N^2+1:end),N,N)';
RR(:,1:round(N/2))=0;
%
vR=[reshape(VR',N*N,1);reshape(RR',N*N,1)];
```

Explain how the **Vreset** function works. Submit the main script and set of functions that give the option to generate spiral waves. Plot screenshots of the spiral wave generated at various timesteps *t(i)* by using the command:
imagesc(reshape(v(i,1:N^2),N,N), clims); axis square
for various values of *i*, as shown in Figure 23.5. Again, you can play with the various parameters to see how they affect the wave dynamics.

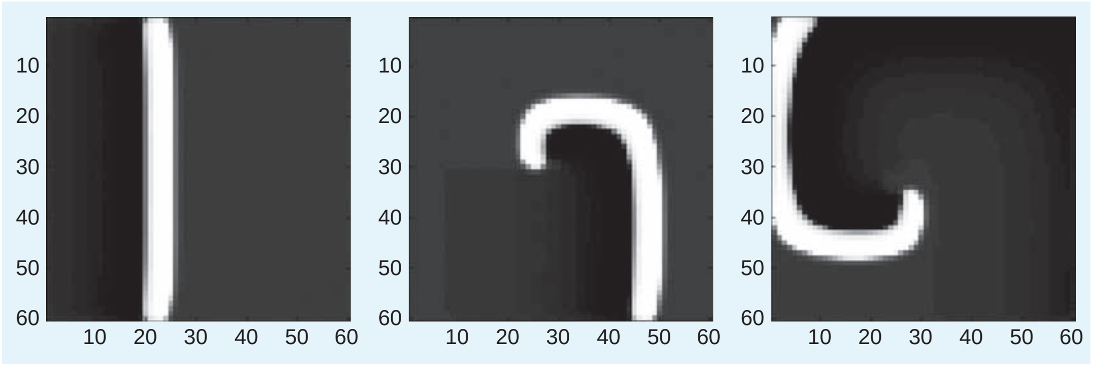


FIGURE 23.5 Spiral wave produced by the Fitzhugh-Nagumo equations for $t = 500$ (left), $t = 1000$ (center), and $t = 1500$ (right).

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

switch
ode45
global
imagesc
reshape

CHAPTER

24

Decision Theory

24.1. GOALS OF THIS CHAPTER

In this chapter you will learn how to implement certain basic mathematical models of decision making using the MATLAB® software. The exploration of decision models will introduce finite difference equations in the context of the diffusion equation, as well as simple 3D visualization. A simple model accounting for perceptual decisions and corresponding activity in cortical areas LIP and MIT will be discussed.

24.2. BACKGROUND

Explorations of reaction time have demonstrated a trade-off between accuracy and speed (e.g. Swensson 1972). That is, subjects asked to make decisions in shorter time make more errors. Any serious model for reaction time must account for this phenomenon.

However, given larger and larger time windows, subjects still occasionally err. In other words, a model must also allow for the possibility of error. A fully deterministic account of the decision would not allow occasional incorrect responses.

These two points provide some guidance on how to treat incoming stimuli with respect to time. With increasing exposure to a stimulus, a subject gains more and more evidence of a particular perception. Thus, one can speak of *evidence accumulation*.

At some point, the subject must make a decision. An evidence threshold to denote the evidence necessary for a choice is a fairly straightforward way to incorporate the decision process.

24.3. EXERCISES

To begin, write a naive model in which evidence is accumulated at a constant rate:

```
function evid = model1(rate, threshold)
        blocksize=100;
        evid = [0];
        last = 1;
        while evid(last) < threshold
                evid = [evid; evid(last)+rate];
                last = last + 1;
        end
end
```

Exercise 24.1: Simulate 500 choice experiments. What is the distribution of results?

However, this approach does not account for the possibility of error for longer presentations. The model must be altered to account for occasional failure. To allow for reported variability, the accumulation of evidence can be made probabilistic.

Now revise the original model to incorporate a probabilistic evidence accumulation:

```
function evid = model2(prob, threshold)
        blocksize=100;
        evid = [0];
        last = 1;
        while evid(last) < threshold
                evid = [evid; evid(last)+(rand() < prob)];
                last = last + 1;
        end
end
```

Exercise 24.2: Choose a threshold and rate and graph five simulations of evidence accumulation.

24.3.1. The Race Model

A classical psychological testing paradigm is the alternative forced choice test, usually with two choices (2 AFC). The race model provides a simple way to account for evidence accumulation for two (or more) choices in an alternative forced choice experiment. The race model consists of a stochastic evidence accumulation process, much like you have already explored, and a decision threshold. Once the evidence accumulation process for a given choice achieves its corresponding threshold, that choice is selected.

Exercise 24.3: Write code to simulate the race model and to graph evidence accumulation for each choice on the same figure. Simulate a two-choice experiment with probabilities (0.4, 0.6) of success and failure, respectively, and generate three trials.

Hint: Use **model2()** twice—once to accumulate evidence for each choice.

Exercise 24.4: Modify your code to generate experiments given a fixed time interval. In other words, write a function:

function c = forced_choice(probs, thresholds, t)

where *probs* is a vector of probabilities for evidence of a given outcome over a period of time, *thresholds* is a vector of thresholds for each option, and *t* is a set number of time intervals for the choice experiment. Here, *c* returns the choice made, which would be the process closest to its threshold.

Exercise 24.5: How does this model address the two-model criteria outlined in the introduction (i.e., a higher error with less time and occasional error even with long intervals)? Demonstrate this with **forced_choice()**.

24.3.2. Problems with the Race Model

The race model does meet some of the original criteria for a decision model, but it fails to account for interactions between choices. For example, if the two choices are mutually exclusive, evidence for one alternative may be counterevidence for the other.

24.3.3. Alternatives to the Race Model

One way to model this scenario is to compare the difference between the evidence counts against a threshold instead of having separate evidence counts and thresholds for each choice.

Exercise 24.6: Create a function **model3()**, which returns the accumulation over time of evidence for the difference between two choices.

Exercise 24.7: How well does this model correspond to actual performance?

24.3.4. Random Walks

Instead of a single threshold for the difference in evidence between the two choices, the decision model could employ two opposed thresholds, one for each choice option. In this scenario, the decision process would track evidence accumulated for each option as a total. Evidence for one choice would accumulate positively, and evidence for the opposing choice would accumulate negatively. The current total would represent the decision process over time, and a total between the two thresholds would represent the undecided state.

As described here, the future behavior of the system depends only on the current state in the decision process. This criterion, as applied to a discrete system here, is called the *Markov property* for a Markov process of order 1. Formally, for a system given states X_1, X_2, X_3,... X_t, and a number of steps t,

$$p(X_t|X_1, X_2, X_3, \ldots X_{t-1}) = p(X_t|X_{t-1})$$

In other words, the probability of a given state depends only on the immediately previous state. The scenario here is a special type of Markovian process, called a *random walk*. At any time step t, there is some probability, p, for accumulating evidence of one alternative or the other $(1 - p)$. In essence, the total evidence value walks randomly along the set of values of total evidence as time progresses.

24.3.5. Diffusion

Currently, the model assumes a discrete evidence count. The analogous process for continuous evidence is the diffusion equation

$$\frac{\partial \phi}{\partial t} = D \nabla^2 \phi \tag{24.1}$$

or in one dimension

$$\frac{\partial \phi}{\partial t} = D \frac{\partial^2 \phi}{\partial x^2} \tag{24.2}$$

where D is the constant of diffusion and phi is the density of the diffusing material. Here, phi represents the probability of evidence over time.

To explore the diffusion equation, you will use a finite difference approach to estimate the change in the probability distribution over time. The finite difference method estimates the infinitely small derivatives by finite differences. As such, finite difference methods can be unstable for large time steps.

To approximate phi over time and space, assume both are discrete, yet partitioned into very small steps. Use the following notation to denote an element of phi in space and time:

$$\phi_{space}^{time} \tag{24.3}$$

Thus, a superscript denotes a time index, and a subscript denotes a space index. Naturally, this lends itself to representation in MATLAB as a two-dimensional array.

Next, replace the derivatives with differences:

$$\frac{\phi_x^{t+1} - \phi_x^t}{\Delta t} = D\frac{\left((\phi_{x+1}^t - \phi_x^t) - (\phi_x^t - \phi_{x-1}^t)\right)}{(\Delta x)^2} \tag{24.4}$$

The derivative of phi over time becomes the difference over one time step at the same location in space, divided by the size of time step. The second partial derivative of phi with respect to space becomes the difference between approximations of the first derivative over space, with time held constant, all divided by the space step squared. When you combine terms, this can be simplified to yield an expression that calculates a value for phi at a given time step using only values from the previous time step:

$$\phi_x^{t+1} = \phi_x^t + D\frac{\Delta t}{(\Delta x)^2}(\phi_{x+1}^t - 2\phi_x^t + \phi_{x-1}^t) \tag{24.5}$$

The following code implements a finite difference approximation of a scenario governed by the diffusion equation. Here, an initial pulse occurs at $x = 20$. Since phi is initialized to all zeros, the iteration bounds of x in the inner loop implicitly enforce boundary conditions of 0 on either end of the 1D space:

```
phi = zeros(1000, 1000);
dt = 0.001; % time from 0 to 1 sec
dx = 0.05; % space from 0 to 50
D=0.5;
phi(1,400) = 1; % initial condition
for t = 1:999
        for x = 2:999
                phi(t+1,x)=phi(t,x) + D*dt/dx^2* ...
                        (phi(t,x+1)-2*phi(t,x)+phi(t,x-1));
        end
end
```

A graph of this in three dimensions using **mesh()** shows how the probability flows with time:

```
mesh(phi)
```

Exercise 24.8: Modify the preceding code for initial conditions of phi = 0.5 at 15 and 35 and graph using **mesh()**.

The diffusion equation as presented here is the continuous limit of an unbiased random walk, one for which the transition probabilities are equivalent. In this case, this does not hold. To account for bias, you need an additional term:

$$\frac{\partial \phi}{\partial t} = B\frac{\partial \phi}{\partial x} + D\frac{\partial^2 \phi}{\partial x^2} \tag{24.6}$$

Here, B is a bias constant, roughly equal to the difference in transition probabilities.

Exercise 24.9: Transform the biased diffusion equation into a finite difference equation and calculate a numerical solution. Choose constant values $B = 0.05$, $D = 0.5$, and initial conditions of a single impulse at the midpoint in space.

Exercise 24.10: Add code to vary the time interval and to determine the time course of the mean value. (*Hint:* Sum all values at a fixed time.) Finally, add code to implement rendering a choice by comparing the mean value to thresholds. Run 10 trials and compare trial performance to the results of **model3()** in Exercise 24.6.

24.3.6. Cortical Models

Any neurobiological model should account for the perceived successes of the diffusion model in describing the characteristics of reaction time in decision processes under psychometric testing. One such model was proposed by Shadlen and Newsome (2001) to account for interactions between visual areas MT and LIP. Area MT is a cortical area sensitive to visual motion, and area LIP is a cortical area implicated in decision processes. Under this model, neurons in area LIP function as integrators, accumulating rate information from neurons in area MT over time. Neurons in area MT, whose preferred directions oppose each other, inhibit the corresponding LIP neuron of the opposing neuron. In other words, LIP neurons integrate the difference between sensory cells with opposing preferred direction sensitivities.

For this simulation, you will draw firing rates from a normal distribution of typical rates for an MT neuron, varying with orientation:

```
function rate = mtneurons(preferred, stimulus, neurons)
        % assuming index 1 is preferred
        typical_rate_mean = [30 20 15 10 5 5 5 5 5 5 5 5 5 10 15 20];
        typical_rate_stdev = sqrt(typical_rate_mean);
        mean = typical_rate_mean(1+mod(stimulus-preferred));
        stdev = typical_rate_stdev(1+mod(stimulus-preferred));
        rate = normpdf(mean*ones(1,neurons), stdev*ones(1,neurons));
end
```

The preceding function returns a vector of rates for a pool of MT neurons with a common direction preference and common stimulus.

24.4. PROJECT

In this project, you will write a simulation of MT and LIP neurons using the Shadlen-Newsome model. Specifically, you are asked to do the following:

- Write code to generate rate values over time for two pools of MT neurons with opposing direction preferences. (*Hint:* The code will need to iterate over time, invoking **mtneurons()** once for each pool at every time step.)
- Under the Shadlen-Newsome model, the firing rate of LIP reflects the running total of the difference in firing rates over time. Write code to sum the rates in each of the pools and integrate the difference over time. Add code to generate a firing rate for an LIP neuron at each time step that varies with the integral of the MT firing rate difference.
- Compare the MT-LIP model to the psychological model. Add code that determines a choice after a time interval by comparing the LIP rate to thresholds. Add code to run multiple trials varying time. Compare performance.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

mesh

CHAPTER

25

Markov Models

25.1. GOAL OF THIS CHAPTER

In this chapter you will learn about Markov processes and how you can use them to model many phenomena, including human behavior. You will also learn how to build a hidden Markov model to characterize the spiking behavior of a neuron.

25.2. BACKGROUND

Note the neural signal in Figure 25.1. There are periods of increased and depressed firing against a steady, tonic rate of firing. If the different firing rates are caused by different states in the system, is there a straightforward way to model this?

A *Markov model* describes a system as a set of discrete states and transition probabilities of moving from any one state to any other state of the model. Beyond a set of states and probabilistic transitions, a Markov model also adheres to the *Markov property*, which states that the transition probability from any state in the network depends only on some finite set of prior states. Thus, given a current state, knowing the probabilities for the next state of the system is merely a question of knowing some n previous states and not the entire state trajectory of the system. Formally, the Markov property states for a system given states X_1, X_2, X_3,.... X_t, and a number of steps t denoting the time progression of the model,

$$P(X_t|X_1, X_2, X_3, \ldots X_{t-1}) = P(X_t|X_{t-1-n,..}X_{t-1})$$

In other words, the states at time steps n before the current state are irrelevant to the probability of the next state.

Given some finite number n of prior state history for determining state transitions, a Markov model with an n prior state history is termed an n^{th} order Markov model. The most common Markov models are first order Markov models, in which transition probabilities depend only on the current state. We will focus on first order models in this chapter.

Formally, a first order model consists of {S, T, s, O, E}, where

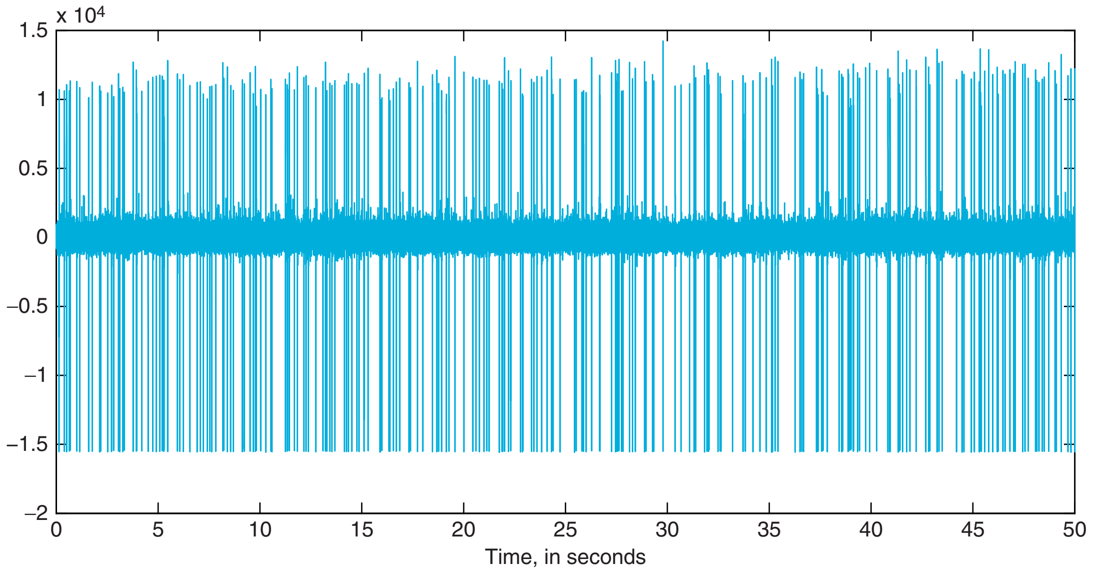


FIGURE 25.1 An extracellularly recorded spike train.

S is a set of states;
T is a matrix of probabilities for transition between pairs of states;
s is the initial state;
O is a set of outputs;
E is a matrix of probabilities to emit each of the possible outputs, given a state.

In many simpler first order models, states are assigned outputs with probability 1.0. Thus, a certain output denotes the model having been in the corresponding state.

As a first example, let's examine syllable order in birdsong. The vocalizations of song birds (passerine birds) have many parallels to human language, including a hierarchically organized vocal structure and finite acquisition periods for the learning of song during development. A sample song from a zebra finch is shown in Figure 25.2. The lower graph shows amplitude variation over time. The upper graph shows a spectrogram of the data, which shows the frequency content of the amplitude signal over time.

Within the structure of the song, there is a clear substructure of elements separated by relatively quiet intervals. These larger groupings are termed *motifs*. Within the motifs are smaller discrete elements, termed *syllables*. The division of the song into these parts might be clearer in the spectrogram. Note that the syllable order within a motif is fairly regular from motif to motif.

During the analysis of song, noting the sequence of syllables is often of interest. A possible annotation is shown in Figure 25.3. Note that syllable 2 repeats and syllables 1 and 8 do not always occur. From this marking, you can attempt to generate a Markov model for a single motif. Figure 25.4 illustrates the state transitions for such a model.

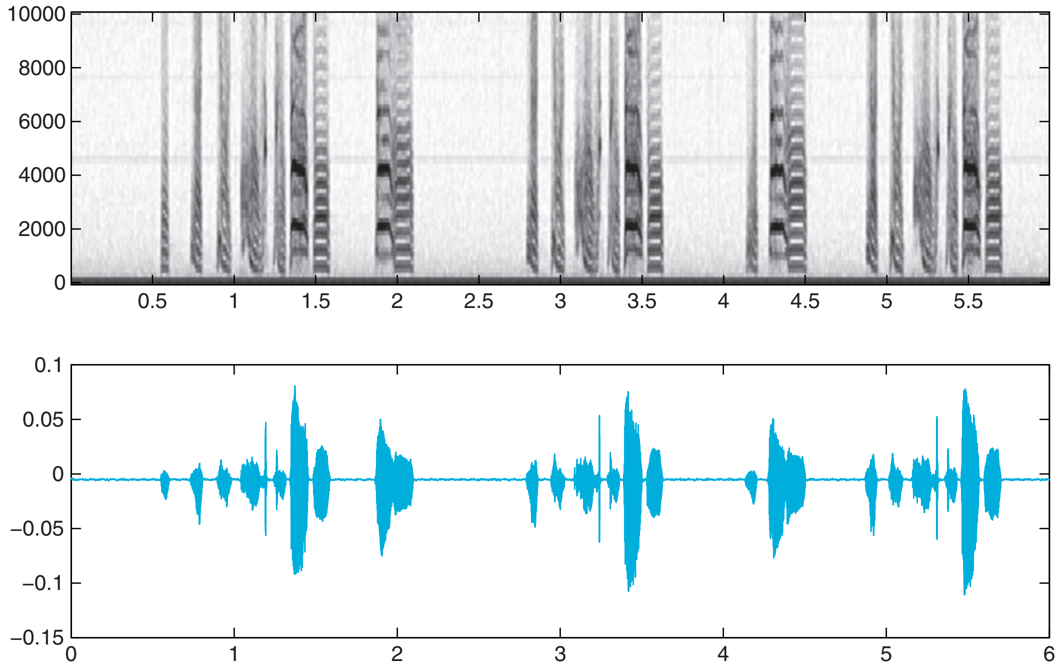


FIGURE 25.2 Sonogram (frequency components over time) and sound amplitude of three motifs from a zebra finch adult.

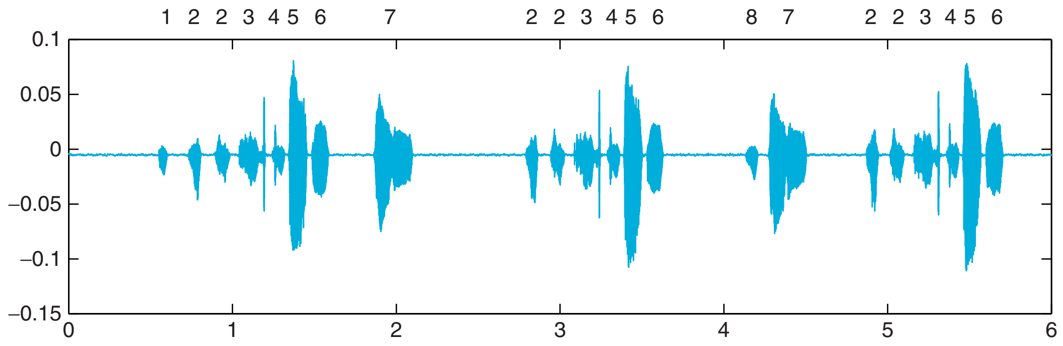
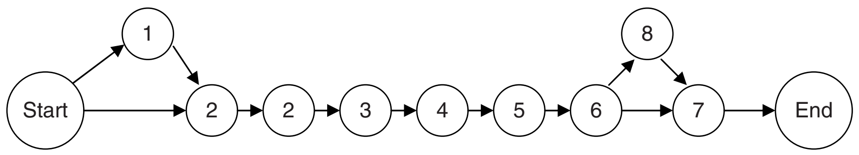


FIGURE 25.3 Annotated song.


$$\begin{tikzcd}
 & 1 \arrow[dr] & & & & & & 8 \arrow[dr] & & \\
\text{Start} \arrow[ur] \arrow[rr] & & 2 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 4 \arrow[r] & 5 \arrow[r] & 6 \arrow[u] \arrow[r] & 7 \arrow[r] & \text{End}
\end{tikzcd}$$
FIGURE 25.4 State transitions for the annotated song.

In Figure 25.4, each circle represents a separate state of the model. The number within the circle denotes the syllable number output by the state. Note the two separate states that output syllable 2. This accounts for the repetition in the natural sequence within the song. Also note the absence of transition probabilities. Ideally, estimating transition probabilities accurately would require examining transitions throughout a large sampling of song. Limiting the sample set to the three motifs in the annotated song in Figure 25.2, the transition probabilities all evaluate to 1.0, excepting the transitions from start to 1 (0.33), start to the first 2 (0.66), 6 to 8 (0.33), and 6 to 7 (0.66).

The state transition diagram in Figure 25.4 has 11 separate states, including the start and end states for the beginning and end of song. With the set of states and the transition probabilities estimated from the song sample, a transition matrix can be generated.

```
T = [ ...
        zeros(1, 11); ...
        0.33 zeros(1, 10); ...
        0.66 1.0 zeros(1, 9); ...
        0.0 0.0 1.0 zeros(1, 8); ... % to second two
        zeros(1,3) 1.0 zeros(1, 7); ... % to three
        zeros(1,4) 1.0 zeros(1, 6); ...
        zeros(1,5) 1.0 zeros(1, 5); ... % to five
        zeros(1,6) 1.0 zeros(1, 4); ... % to six
        zeros(1,7) 0.66 zeros(1, 3); ... % to seven
        zeros(1,7) 0.33 zeros(1, 3); ... % to eight
        zeros(1,8) 1.0 1.0 zeros(1, 1); ... % to end
        ];

s = 1;
f = 11;
```

Now, generate the output vector and emission matrix. Each state produces a single output with 100% certainty. Values of −1 are produced by the start and end states. Now assume that each state emits an output representative of the state with 100% certainty:

```
O = [1 2 3 4 5 6 7 8 −1];

E = zeros(11, 9);
E(4:11,2:9) = eye(8);
E(2:3,1:2) = eye(2);
E(1,9) = 1;
```

You can then generate a possible sequence of states and outputs with the following code:

```
function [seq, emit]= markov_sequence(T, s, E, O, f, max_seq)
% T is M by M matrix containing probability of moving
%       from row state to column state. T(2, 3) = probability
%       of state 2 to state 3 transition.
% s is a scalar specifying the start state
% E is M by N matrix containing probability of emitting
%       output N at state M.
% O is a vector of output values
% f is a scalar representing a termination state.
% max_seq is a scalar representing the maximum sequence size.

      seq = [s];

      % initial emission
      emission = min(find(rand < cumsum(E(s,:))));
      emit = [O(emission)];

      while s ~= f && length(seq) < max_seq
         % transition to new state
         state_vector = T(:,s)';
         p = rand;
         s = min(find(p < cumsum(state_vector)));

         seq = [seq; s];

         % find emission
         emission = min(find(rand < cumsum(E(s,:))));
         emit = [emit; O(emission)];

   end
```

25.3. EXERCISES

Exercise 25.1: Generate a number of sequences for the preceding Markov model. Do the output syllables match what you expect? How about the state sequence?

Exercise 25.2: Load a larger recording of the same bird and create a Markov model for multiple motifs. You can find a longer recording in file **zf_y89.wav** on the companion website. Load the file using **wavread()**.

25.3.1. Hidden Markov Models

A Markov model works quite well if the states and transitions are easily measured. Often, however, only the outputs are observable, and there is not necessarily a one-to-one correspondence between output and internal state. Such a scenario is termed a *Hidden Markov Model*, or *HMM*, because the current state of the system generating the output is unknown.

The brain of songbirds (specifically oscine passerine birds) has discrete nuclei involved in the learning and production of song. Closely related birds lack these nuclei. In the song system, efferent connections from auditory nuclei connect to a nucleus termed HVC, which has been implicated in generating as well as perceiving song. HVC neurons make two primary efferent connections: one to the nucleus RA and one to a large forebrain nucleus termed Area X. Area X is part of a three-nucleus thalamic loop that also connects to RA. The forebrain nucleus loop containing Area X has been compared with similar circuits in mammals that involve the basal ganglia.

Here, however, the focus is on nucleus RA. The primary efferent projection of RA is to a brain stem area nXII, which innervates the muscles of the syrinx, the sound production organ of songbirds. RA has distinctive neural activity patterns, particularly during sleep. Activity is strongly tonic, punctuated by strong bursts of activity. The bursts of activity are believed to originate afferently from HVC. Following bursts of activity, firing rate is attenuated. It is these three states — tonic, driven, and post-burst — that will constitute the hidden states of our Hidden Markov Model.

25.3.2. Modeling Tonic Firing

Here, you will model the tonic firing state fairly simply. The tonic firing model will have only one state, with a fixed probability of firing (see Figure 25.5). The primary transition will be from the tonic firing state back to itself, to model the consistent tonic firing observed in RA.

25.3.3. Modeling Bursting

You will model the drive from HVC as a two-state network, with the drive-generated increase in firing and the post-burst depression as separate, recursive states (see Figure 25.6).

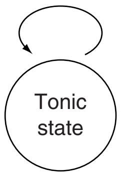


FIGURE 25.5 The tonic firing state.

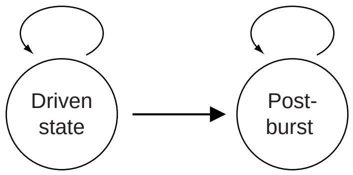


FIGURE 25.6 The two-state network.

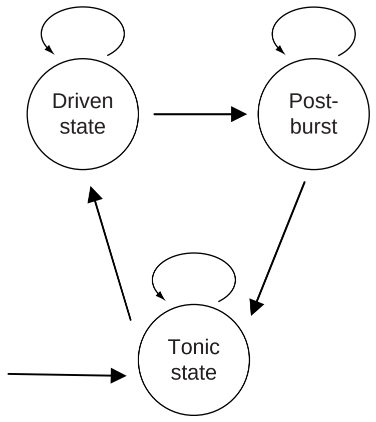


FIGURE 25.7 The full model.

25.3.4. Full Model

Now you will connect the two models, as shown in Figure 25.7.

The model in this figure shows the three states interconnected. Note that the drive and post-burst depression states are in sequence.

25.3.5. Classical Questions with Hidden Markov Models

Work with Hidden Markov Models has provided well-tested algorithmic means of exploring the following three questions.

1. Given a sequence of observations and a model, what is the most probable sequence of internal states that produced the observation?
2. Given a series of observations, what is the most probable HMM?
3. Given a model, what is the most probable output?

The statistics toolbox in the MATLAB® software provides a number of routines that implement the algorithms that solve these questions.

Hmmgenerate generates an example sequence given emission and state transition matrices:

seq = hmmgenerate(sequence_length, trans, emissions)

Hmmestimate estimates transition and emission probabilities given a sequence and the corresponding states. In other words, **hmmestimate** is appropriate if the set of hidden states for a given sequence of outputs is known.

[est_trans, est_emis] = hmmestimate(seqs, states)

Hmmtrain uses the Baum-Welch algorithm to estimate transition and emission probabilities using maximum likelihood. Note that **hmmtrain** expects initial guesses for transition and emission probability matrices.

[esttr, estemit] = hmmtrain(seqs, trguess, emitguess)

Hmmdecode calculates probabilities for a sequence or for a set of sequences given transition and emission probabilities:

pstates = hmmdecode(seqs, tr, emit)

Hmmviterbi calculates the most probable sequence of states given the series of observations:

states = hmmviterbi(seqs, tr, emit)

25.4. PROJECT

For this project, you will create a Markov model to approximate spiking in an electrophysiological recording. We will estimate the sequence of internal states and use this to estimate the transition and emission matrices. On the companion website, find the file RA-spike-times.mat. This file contains a series of spike times for an electrophysiological recording of area RA. Transform the spike times into an instantaneous rate (as the reciprocal of interspike interval) and determine appropriate thresholds for the three states (driven, tonic, and post-burst).

Using the thresholds for the three states, you can determine the state for each moment in time. In other words, at each 1 ms slice of time, you can estimate the state of the system by comparing the instantaneous rate with the rate thresholds set for each state. After calculating the state at each point in time, you will have a sequence of states, **states**.

Transform the spike data into point process data. This will form the sequence of outputs. Take the raw recording and find the time location of spikes. Next, divide the entire interval into an array of 1 ms time slices. Whenever a time slice contains a slice, set the value corresponding to that time slice to 1. This array is the sequence of outputs, **outputs**.

Use **hmmestimate(outputs+1, states)** to calculate transition and emission matrices. (Regarding the addition of 1 to **outputs**, note that hmmestimate expects not the actual outputs but the index in the set of output values. Thus, since the outputs are 0 and 1, 1 and 2 can be used for hmmestimate.)

Use either **markov_sequence** or **hmmgenerate** (use the help function in MATLAB!) to generate sample outputs from the HMM. Do these resemble the sparse time data? Generate a transition diagram from the transition matrices. How well does the diagram adhere to the original model?

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

hmmestimate
hmmgenerate
hmmtrain
hmmdecode
hmmviterbi

CHAPTER

26

Modeling Spike Trains as a Poisson Process

26.1. GOALS OF THIS CHAPTER

This chapter introduces the Poisson random process as a model to characterize trains of action potentials generated by neurons. You will learn how to generate a homogeneous and inhomogeneous Poisson process using the MATLAB® software and determine how well it captures the statistical properties of real neuronal spike trains. In the project at the end of this chapter, you will learn how to modify a Poisson model to account for history effects such as refractoriness and burstiness.

26.2. BACKGROUND

What is an appropriate model for simulating spike trains?

Consider the spike train shown in Figure 26.1. Some spike train statistics can be calculated easily. For example, the mean spike rate is simply the number of spikes divided by the time interval. The inverse of the interval between adjacent spikes approximates the instantaneous rate.

The refractory period for firing neurons limits activity to one per refractory period. Taking a 1 ms refractory period, the time interval can be divided into a sequence of 1 ms periods. Each 1 ms period can be examined as a potential firing event for the neuron. This characterizes the larger measurement interval as a series of event opportunities or trials, some of which result in an actual event. When they are viewed as a series of trials, it becomes clear that the number of opportunities greatly outweighs the number of actual events. As such, whatever probabilistic model used should effectively simulate low success rates over many trials.

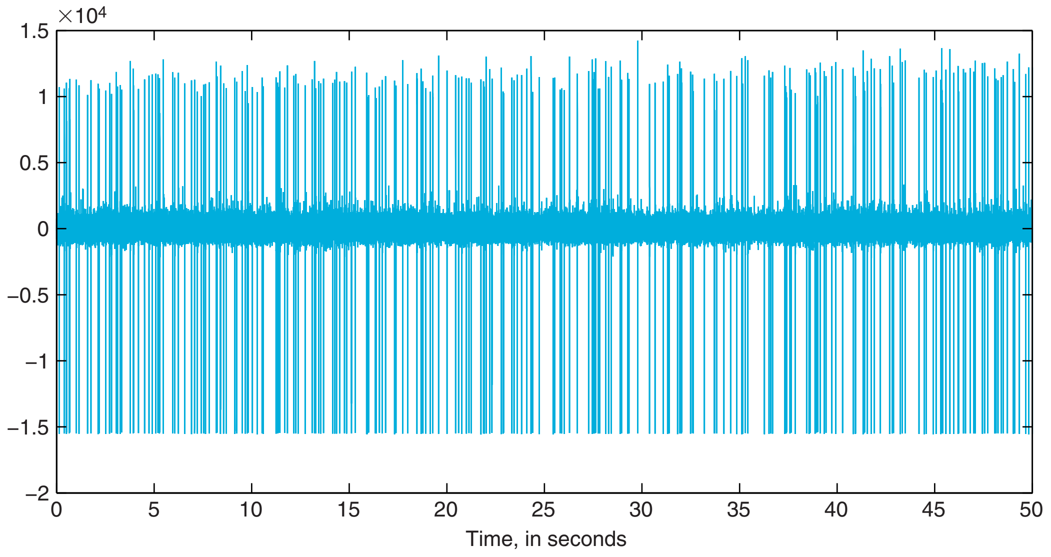


FIGURE 26.1 An extracellularly recorded spike train.

26.2.1. Poisson Processes

A Poisson process is a random process in which events occur independently with low probability over continuous time. While the Poisson process does meet most criteria for a neural spike train, the refractory period and other features prevent statistical independence. However, as a first approximation, a Poisson process provides a good tool for exploring neural spike trains.

26.2.2. The Poisson Distribution

One of the most important properties of the Poisson process is that the distribution of events within an interval follows a Poisson distribution. A Poisson distribution is *parameterized* by a single value, *lambda*. *Lambda* is often interpreted as the rate of success. The reason is that the expectation value (or mean) of a Poisson random variable is *lambda*. In other words, for a random variable describing the number of events per interval, the mean value is *lambda*.

The following equation shows the *probability mass function* for a Poisson distribution for a value of *lambda*:

$$P(x = k) \equiv \frac{\lambda^k}{k!} e^{-\lambda} \tag{26.1}$$

A *probability mass function* maps success rates to probabilities.

A Poisson random variable with parameter *lambda* has *expectation value lambda*. This means that over many trials, the mean number of successes will approach *lambda*. A Poisson process has the unusual property that the variance is also equal to *lambda*. One last interesting property of the Poisson distribution is that the sum of two Poisson processes distributes as a Poisson process with *lambda* equal to the sum of the *lambda* values of the individual processes.

26.2.3. Intervals between Poisson-Distributed Events

The intervals between Poisson-distributed events have a distinct distribution as well. Spacing between subsequent events follows an exponential distribution:

$$P(x \leq k) \equiv 1 - e^{-\lambda k} \tag{26.2}$$

Note that the exponential distribution is a continuous probability distribution. As such, there is no explicit expression for the probability of a discrete value. Instead, the preceding cumulative probability distribution yields a probability that a random value exponentially distributed will fall within the range 0 . . . *k*. It is important to note that the refractory period of real neurons causes the inter-spike interval distribution to diverge from an exponential distribution for small intervals.

26.3. EXERCISES

26.3.1. Generating a Poisson Random Variable

Normally, the easiest way to sample from a Poisson distribution is to use the Statistics Toolbox for MATLAB. The Statistics Toolbox provides the function **poissrnd(*lambda*)**, which returns a random number that varies according to a Poisson distribution with parameter *lambda*. **poissrnd** can also return a vector or a matrix of random Poisson values:

```
>> x = poissrnd(5);
>> y = poissrnd(5, 3, 1); % a vector
>> z = poissrnd(5, 5); % a 5 x 5 matrix
```

Exercise 26.1: Generate a probability mass function for the Poisson distribution for *lambda* = 5. Calculate the expectation value

$$\left(E[x] = \sum_{x=0}^{\infty} x \frac{\lambda^x}{x!} e^{-\lambda} \right) \tag{26.3}$$

and the variance

$$\left(Var[x] = E[x^2] - (E[x])^2 \;\; \text{where} \;\; E[x^2] = \sum_{x=0}^{\infty} x^2 \frac{\lambda^x}{x} e^{-\lambda} \right). \tag{26.4}$$

Do the same for *lambda* = 7. Generate histograms for many trials drawn from the same Poisson distributions. Calculate estimated means and variances for the random data. Do the histograms make sense? How do the means and variances compare to *lambda*?

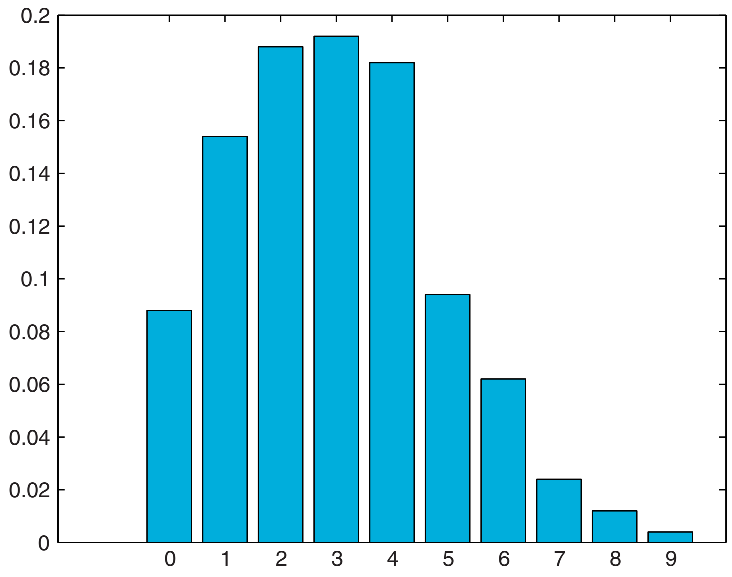


FIGURE 26.2 An estimate of a probability mass function generated with the **poissrnd** function in MATLAB.

You can approximate a probability mass function by calculating a large number of Poisson variables and generating a histogram of the results, as follows:

```
>> hist(poissrnd(4, 500, 1))
```

To generate a true estimate of a probability mass function, you would need to capture the counts in each bin and normalize by the total trial count (see Figure 26.2):

```
>> h = hist(poissrnd(4, 500, 1));
>> bins = 0:(length(h)–1); % zero is the first bin
>> count = sum(h);
>> bar(bins, h/count)
```

26.3.2. Generating Poisson Variables without the Statistics Toolkit

As mentioned previously, the easiest method for a Poisson variate is the function in the Statistics Toolbox. However, if the Statistics Toolbox is not available, you can use the PDF to calculate probabilities for each successive outcome and use a flat random distribution to determine the outcome.

The algorithm would look something like this:

```
set the initial outcome to 0
pick a uniformly distributed probability from 0 to 1, p
iterate over outcomes
```

```
    calculate the probability of the current outcome
    if $p <$ the calculated probability, then this is the outcome
    otherwise
        subtract the calculated probability from $p$
        increment the current outcome
```

The preceding algorithm works for any discrete probability distribution.

Exercise 26.2: Turn the preceding algorithm into a function **my_poisson()** that accepts a single parameter *lambda* and returns a Poisson random variable. Can you explain the previous steps? Why is the probability of the current outcome subtracted from the uniformly distributed value?

Knuth proposes an alternative algorithm:

```
set $T$ to exp(–lambda), $k$ to 0, and $p$ to 1
        increment $k$
        choose a random number $u$ between 0 and 1, uniformly distributed
        multiply $p$ by $u$
        continue until $p$ is less than $T$
$k - 1$, the number of iterations such that $p$ is just above the threshold $T$, represents the number of
successes in a Poisson trial with parameter lambda
```

Exercise 26.3: Implement the preceding algorithm as **my_poisson2()**. Does this return results similar to the algorithm shown earlier in this section? This algorithm has the efficiency advantage of calculating only a single exponential per random number.

26.3.3. Generating Interspike Intervals

Much as you can do with the Poisson distribution, you can generate random numbers that are exponentially distributed by using a command from the Statistics Toolbox in MATLAB: **exprnd(*lambda*).**

However, exponentially distributed random numbers can easily be simulated with a uniform random number generator. Given a uniform random variable *U*, then *–ln(U)* divided by *lambda* yields a number exponentially distributed, with mean *lambda*.

26.3.4. Generating a Spike Train

As a first attempt, you can generate a spike train from an average spike rate per small time slice.

Exercise 26.4: Given an average rate of 25 spikes per second, estimate the distribution of spikes over 10 seconds two distinct ways. First, generate a histogram of samples from a Poisson distribution representing the spike count over the 10 second interval. Second, divide the 10 second interval into 1 ms pieces and calculate an appropriate probability of the single event for each 1 ms interval (hint: lambda * dT). Sum the overall event count for a spike count. How does this compare with the original data on the companion website (sample-RA.mat)?

Exercise 26.5: You can also use the exponential distribution of the interspike intervals to generate a simulated spike train. To do so, continue to draw intervals from an exponential distribution with the desired *lambda* while the total of the intervals remains below the overall spike train duration. The intervals represent the interspike intervals between consecutive events. Generate a 10-second spike train using the exponential distribution of interspike intervals with an average rate of 25 spikes per second. How does the spike train compare to that generated with an explicit Poisson variable?

26.3.5. Nonhomogeneous Poisson Processes

In the formulation of the Poisson process, we have discussed only a constant *lambda*. Such Poisson processes are termed *homogeneous*. An important generalization of the Poisson process is the addition of a time-dependent *lambda*. Processes in which *lambda* varies with time are called *nonhomogeneous*. Nonhomogeneous Poisson processes are often used to simulate spike trains in the presence of an external stimulus with known characteristics, such as experimental conditions testing a neuron sensitive to sensory stimuli.

Given a function $\lambda(t)$, you can calculate the mean *lambda* for the interval over which you are selecting Poisson values. If you use a time interval t, $t + \Delta t$, then you can calculate the mean over the interval:

$$\frac{1}{\Delta t}\int_{t}^{t+\Delta t} \lambda(\tau)d\tau \qquad (26.5)$$

By doing this, you can generate spike trains as before, by dividing the overall spike train duration into 1 ms subintervals and calculating the probability of an event in each subinterval.

Exercise 26.6: On the companion website, you can find the file MT.mat, which represents the responses of a neuron in visual area MT. Cells were exposed to a stimulus paradigm that incorporated 4 seconds of idle activity, after which a visually relevant stimulus was presented for 500 ms. Formulate a plausible *lambda* function and simulate additional spike trains. Compare to the spike train in MT.mat.

26.4. PROJECT

In reality, spike trains can rarely be treated as a pure Poisson process or even as a non-homogeneous Poisson process because of history effects such as the absolute and relative refractory period and burstiness. The neural refractory period eliminates extremely low duration intervals, whereas burstiness increases the likelihood of low duration interspike intervals. In either case, the shape of the interspike interval distribution is altered from a simple exponential function.

However, Brown, et al. propose an application of the time-rescaling theorem to generate point process data from a probability density that varies with the positions of previous spikes. To employ this method, a conditional intensity function is needed. A conditional intensity function relates time to spike rate given times for some set of prior spikes. As such, details such as the refractory period and burstiness can be accounted for.

First, a conditional intensity function must be defined. A conditional intensity function yields instantaneous firing rate given the spiking history of the cell. In other words, the conditional intensity function is a conditional probability. Formally, it is written as:

$$\lambda(t|H) \tag{26.6}$$

where H is the spiking history and t is the lag time from the last spike.

Brown et al. provide a means for calculating the conditional intensity function:

$$\lambda(t|H) = \frac{f(t|H)}{1 - \int_{u'}^{t} f(u|H)du} \tag{26.7}$$

where f() is the conditional ISI distribution, given spiking history and u′ is the time of the spike immediately preceding the time t.

To estimate a conditional intensity function for a neuron from measured spike data, choose some fixed number of spikes, n, of history to track. At every event, determine the spike time history for the previous n spikes. For each spike time history, capture a histogram of times until the next spike. This histogram, after normalization, estimates f().

To estimate the conditional intensity function at time t, determine the spiking history for the n spikes previous to t. Given this history, determine the firing probability for interval t from the most recent spike (this is f()). Divide this firing probability by 1 minus the sum of the probabilities for all lag times up to t.

To generate a spike train consistent with the conditional intensity function, use 1 ms intervals and calculate the conditional intensity function at each 1 ms interval (each value t) in sequence. To determine whether a spike occurs in a given 1 ms interval, multiply the conditional intensity function at t by the interval size. Use this product as the probability of an event occurring in the given interval. In other words, pick a uniformly distributed number from 0 to 1. If the number exceeds the event probability, no event occurs.

On the companion website are a number of sample spike trains. Pick one train and use the above algorithm to generate a similar event train.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

poissrnd
exprnd

CHAPTER

27

Synaptic Transmission

27.1. GOALS OF THIS CHAPTER

This chapter will use a number of methods to characterize the processes surrounding synaptic transmission, in particular the release and diffusion of neurotransmitters. This chapter will also introduce handles for graphic objects to update images dynamically. By the end of this chapter, you should have an understanding of different random variables, discrete distributions, finite difference approximations to partial differential equations, and the use of graphic handles for rudimentary animation.

27.2. BACKGROUND

Chemical synapses use the release of chemical neurotransmitters to propagate signals from one neuron (presynaptic) to another (postsynaptic). The two cells are separated by the *synaptic cleft*, a gap of approximately 40 nm between the presynaptic and postsynaptic membranes.

On the presynaptic side of the cleft, depolarization from the arrival of the action potential triggers the opening of voltage-sensitive Ca^{+2} channels. The subsequent influx of calcium ions causes the fusion of neurotransmitter-containing vesicles with the presynaptic cell membrane, releasing neurotransmitters into the synaptic cleft.

On the postsynaptic side, neurotransmitters diffuse across the cleft and bind to neurotransmitter-specific sites on channels on the postsynaptic terminal. These receptors selectively allow ions to enter the postsynaptic terminal. The influx of positive or negative charge changes the voltage across the postsynaptic membrane, creating a small hyperpolarization or depolarization. Over time, the concentration of the neurotransmitters decreases to a level insufficient to activate the postsynaptic receptors.

Specifically, in this chapter we will focus on the neuromuscular junction, the site of innervation of skeletal muscle. We will also focus on the steps of neurotransmitter release and neurotransmitter diffusion.

27.3. EXERCISES

27.3.1. Modeling Neurotransmitter Release

In a classic experiment, Fatt and Katz demonstrated at the neuromuscular junction that spontaneous postsynaptic potentials occurred at voltages of 0.5 mV in the absence of presynaptic stimulation. Later work (del Castillo and Katz) further demonstrated that single acetylcholine channels produced much smaller responses than the 0.5 mV measured by Fatt and Katz. This result implied that the spontaneous results observed by Fatt and Katz not only recruited multiple acetylcholine channels, but also recruited roughly the same number of channels during each spontaneous postsynaptic potential del Castillo and Katz posited that synaptic transmission occurred in discrete units, termed quanta. Additional work (del Castillo and Katz) established that increasing Ca^{+2} at the postsynaptic terminal produced postsynaptic responses in increments of the original spontaneous postsynaptic responses. The step-like response implicated neurotransmitter release as a quantized process. In other words, neurotransmitter release occurs in discrete steps rather than a continuous concentration in response to increasing calcium concentration. Synaptic vesicles contain a relatively constant number of neurotransmitter molecules. This fixed number of molecules per vesicle provides for the observed quantization of postsynaptic responses. At the neuromuscular junction, each vesicle contains approximately 5000 acetylcholine (ACh) molecules.

For purposes of this chapter, assume that the release of individual vesicles occurs independently with some probability p. In the presence of low calcium concentration, p is low. After an action potential and subsequent calcium influx, p increases. You can model the release of a single vesicle as a Bernoulli random variable.

27.3.2. Modeling Random Variables

A Bernoulli random variable X takes on a value of 1 with probability p or a value of 0 with probability $1 - p$. A coin toss is an excellent example of a Bernoulli process. Take a coin that lands on heads with probability 1/2. A Bernoulli random variable models a single coin flip, or a single trial.

Thus, in the case of the release of a single vesicle, the vesicle is released with probability p or not with probability $1 - p$.

Exercise 27.1: Using the **rand** function in the MATLAB® software, write a function titled **my_bernoulli_rnd()** to return the result of a Bernoulli trial, given p. You should be able to invoke the function like this:

```
>> p = 0.5;
>> my_bernoulli_rnd(p)
ans =

1
```

You can model the process of multiple vesicles as the sum of multiple Bernoulli variables with probability p. The sum of n Bernoulli trials with probability p of success is termed a *binomial random variable* with parameters n and p. Such a variable is called *binomial* because the probability of a certain number of successes in a trial can be calculated with the binomial coefficient:

$$f(k;n,p) = C(n,k)p^k(1-p)^k \tag{27.1}$$

where
$$C(n,k) = \binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{27.2}$$

The function **f()**, called a *probability mass function*, yields the probability of a certain number of successes, k, in a single binomial trial with parameters n and p. A single binomial trial with parameters n and p would model the number of successes for an experiment in which each trial has n coin tosses with probability p of success.

Exercise 27.2: Generate a graph of the probability mass function for a binomial random variable with parameters $n = 10$ and $p = 0.5$.

Under conditions of calcium influx, approximately 150 quanta can be released at the neuromuscular junction in 1–2 milliseconds. In the absence of action potentials, only 1 quantum per second is released by the presynaptic terminal. For this chapter, model the release of multiple vesicles as a binomial random variable in which p is either 0.001 or 150, depending on the state of calcium concentration. Then, the number of successes will be the number of vesicles released into the synaptic cleft.

The following code calculates p for 1 second of time, choosing 10 random 1 ms intervals during which p is high, representing the calcium influx following an action potential:

```
>> t=0:999;
>> t_slices=length(t);
>> p=ones(t_slices,1)*0.001;
>> x=rand(10,1)*t_slices;
>> p(floor(x)+1)=150;
```

Note that x holds the set of random intervals. When you evaluate **p(floor(x)+1) = 150**, each element of x corresponding to one of the random intervals is set to 150. The use of **floor()** forces the values of x to integer values, the proper type for indices. (The **floor()** function truncates the decimal portion of a value.) You can see which indices contain large p values with the following command:

```
>> find(p>1)
```

Exercise 27.3: Evaluate the preceding code and graph **p(*t*)** against time. Write code to generate a binomial random variable at each time slice (use **binornd**). Graph the number of successes, or released vesicles, as a function of time.

27.3.3. Modeling the Motion of a Single Molecule

Upon release, the neurotransmitter molecules enter the synaptic cleft and diffuse across fairly quickly. On a microscopic level, diffusion is the aggregate effect of many particles moving randomly. As a first step, examine the motion of a single molecule:

```
xbounds = [0 10];
ybounds = [0 4];
xdata = [mean(xbounds)];
ydata = [0];
xgrid = 0.01;
ygrid = 0.01;
figure
handle = scatter(xdata, ydata, 'filled');
xlim(xbounds);
ylim(ybounds);

for t = 1:10000
   p = 0.5;
   dx = ((rand > p) - 0.5) * 2;
   dy = ((rand > p) - 0.5) * 2;
   xdata = xdata + dx*xgrid;
   % these two lines assure the molecule stays in x bounds
   xdata(find(xdata < xbounds(1))) = xbounds(1);
   xdata(find(xdata > xbounds(2))) = xbounds(2);
   ydata = ydata + dy*ygrid;
   % these two lines assure the molecule stays within y bounds
   ydata(find(ydata < ybounds(1))) = ybounds(1);
   ydata(find(ydata > ybounds(2))) = ybounds(2);
   set(handle, 'xdata', xdata, 'ydata', ydata);
   drawnow;
end
```

Some sample screenshots of the resulting animation are shown in Figure 27.1. In the preceding code, the handle of the scatterplot is stored in a variable. Much like matrices or scalar numbers, variables can store other types of information. In this case, you are storing a handle. Most of the graphics functions in MATLAB return a handle when invoked. You can use the handle to modify properties at a later time, as done previously with **set**.

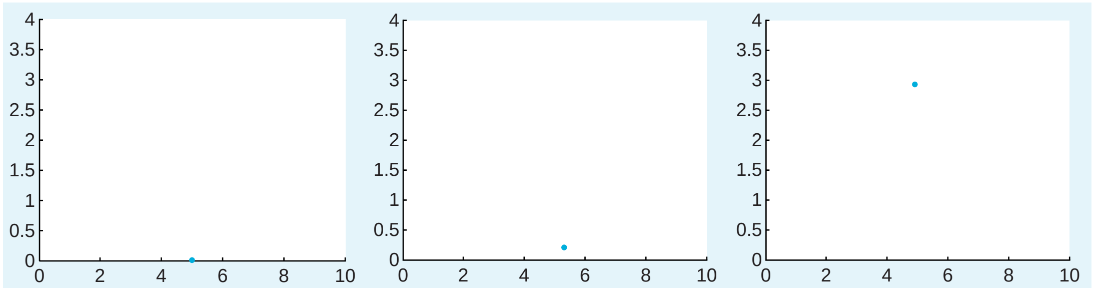


FIGURE 27.1 Screenshots of the animation for the diffusion of a single neurotransmitter molecule across the synaptic cleft for $t = 1$ (left), $t = 500$ (middle), and $t = 10000$ (right).

The function **set** allows specifying properties of a graphics object. The first argument should be the handle. Following this should be a series of property, value pairs, consisting of the name of a property in single quotation marks and the desired value of the property. Here, the x and y data for the scatterplot are changed. Each time the variables *xdata* and *ydata* are modified, **set** is invoked to update the coordinates of the point in the scatterplot.

After **set** comes the function **drawnow**. Even though **set** updates the coordinates of the point in the scatterplot, MATLAB will not redraw the figure without notification. **drawnow** provides notification that the figure has changed and forces a redraw. Try running the preceding code with **drawnow** commented out.

Exercise 27.4: Modify the preceding code to model multiple particles (try 100).

As the number of particles increases, modeling the motion of individual particles becomes computationally limiting. Instead, you can model concentrations rather than particle counts.

27.3.4. Modeling Diffusion

To model the process as a concentration, you need to use the diffusion equation:

$$\frac{\partial \phi}{\partial t} = D \nabla^2 \phi \tag{27.3}$$

Or in two dimensions, you can use:

$$\frac{\partial \phi}{\partial t} = D \frac{\partial^2 \phi}{\partial x^2} + D \frac{\partial^2 \phi}{\partial y^2} \tag{27.4}$$

where D is the constant of diffusion and ϕ is the concentration of acetylcholine, ACh.

In this chapter you will use a *finite difference* approach to estimate the change in the probability distribution over time. The finite difference method estimates the infinitely small derivatives by finite differences. Because this approximates the continuous equation with discrete differences, we should mention that large time steps can produce unstable results.

To approximate ϕ over time and space, assume both are discrete yet partitioned into very small steps. Use the following notation to denote an element of phi in space and time:

$$\phi_{space}^{time} \tag{27.5}$$

Thus, a superscript denotes a *time* index, and a subscript denotes a *space* index.

To generate the finite difference equation, replace the derivatives with differences:

$$\frac{\phi_{x,y}^{t+1} - \phi_{x,y}^{t}}{\Delta t} = D\left[\frac{\left(\left(\phi_{x+1,y}^{t} - \phi_{x,y}^{t}\right) - \left(\phi_{x,y}^{t} - \phi_{x-1,y}^{t}\right)\right)}{(\Delta x)^2} + \frac{\left(\left(\phi_{x,y+1}^{t} - \phi_{x,y}^{t}\right) - \left(\phi_{x,y}^{t} - \phi_{x,y-1}^{t}\right)\right)}{(\Delta y)^2}\right] \tag{27.6}$$

With some rearrangement, you get:

$$\phi_{x,y}^{t+1} = \phi_{x,y}^{t} + D\Delta t\left[\frac{\left(\phi_{x+1,y}^{t} - 2\phi_{x,y}^{t} + \phi_{x-1,y}^{t}\right)}{(\Delta x)^2} + \frac{\left(\phi_{x,y+1}^{t} - 2\phi_{x,y}^{t} + \phi_{x,y-1}^{t}\right)}{(\Delta y)^2}\right] \tag{27.7}$$

which provides an expression for a concentration at a given spatial location and time in terms only of concentrations at previous time steps. If you use the same grid spacing for x and y, you can simplify even further:

$$\phi_{x,y}^{t+1} = \phi_{x,y}^{t} + D\Delta t\left[\frac{\left(\phi_{x+1,y}^{t} + \phi_{x,y+1}^{t} + \phi_{x-1,y}^{t} + \phi_{x,y-1}^{t} - 4\phi_{x,y}^{t}\right)}{(\Delta x)^2}\right] \tag{27.8}$$

You already saw an efficient way to compute this spatial second derivative in MATLAB. As discussed in Chapter 23, "Fitzhugh-Nagumo Model: Traveling Waves," this is found via a two-dimensional convolution of ϕ with the filter **[0 1 0; 1 4 1; 0 1 0]/dx^2**. To encode the dynamics of the ACh concentration diffusion in MATLAB, use a three-dimensional array: two spatial dimensions and one temporal dimension.

The following code implements iterations of the previous equation, using a three-dimensional array to track change in concentration. The time steps are in 10 ns increments, and the spatial steps are in 1 nm increments. The diffusion constant here is 4×10^{-6} cm^2/sec, and free boundary conditions are used:

```
% phi : 100 x steps (100 nm), 40 y steps (40 nm)
%       100 t steps (1 t = 10 us)
clear all
close all
phi = zeros(100, 40, 100);
```

```
dt = 1e-10; % time in steps of 10 ns
dx = 1e-9; % space from 0 to 50
D=4e-6 * (1/100)^2;
phi0=5000/(dx^2);
phi(50,1,1) = phi0; % initial condition
F = [0 1 0; 1 -4 1; 0 1 0]/dx^2;
for t = 1:99
   phi(:,:,t+1) = phi(:,:,t) + ...
      D*dt*conv2(phi(:,:,t),F,'same');
   t
end
```

You can plot a time-slice of the concentration using **surf(phi(:, :, t))**. Sample screenshots of the concentration diffusing in the synaptic cleft using the **surf** function are shown in Figure 27.2. Similarly, you can visualize this using the command **imagesc(phi(:,:,t), [0 phi0])**, where the **[0 phi0]** input to the function ensures that the color scale is the same for all t.

Exercise 27.5: Generate an animated display to show the evolution of the ACh concentration diffusion using **surf()**. Capture the handle and use **set()** with the property **'zdata'**.

Exercise 27.6: The algorithm could be changed to save only the current and previous time steps, allowing an unlimited number of time steps to be calculated if so desired without exceeding the memory of MATLAB. Obviously, this approach is less than optimal if all the time step calculations are needed, but, for an animation, only the current time step is necessary. Modify the algorithm to store only the current and previous time step and animate the diffusion.

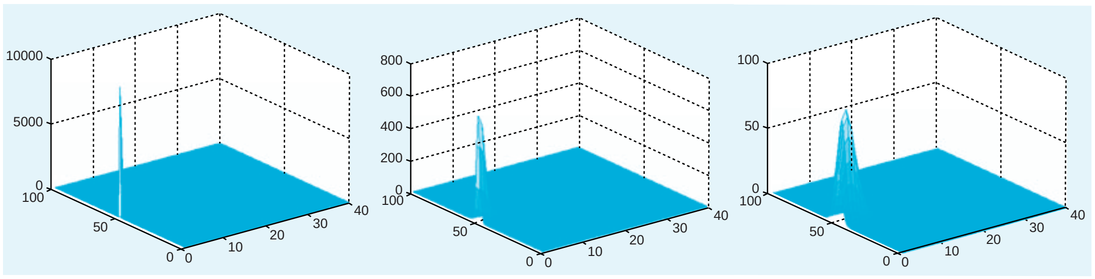


FIGURE 27.2 Screenshots of the animation for the diffusion of the concentration of acetylcholine, ACh, in the synaptic cleft for $t = 10$ µs (left), $t = 250$ µs (middle), and $t = 1000$ µs (right).

27.4. PROJECT: COMBINING VESICULAR RELEASE WITH DIFFUSION

At this point you need to combine the release of neurotransmitters with the diffusion of the neurotransmitters across the synaptic cleft. Combine the neurotransmitter release code with the code for diffusion.

To do so, use a single time loop for both the vesicular release and the diffusion code. At any given time slice, the code will need to determine the number of vesicles released. If vesicles are released during a given time slice, then (1) a location along the presynaptic edge of the diffusion grid needs to be selected, and (2) the concentration of ACh in that square of the finite difference grid needs to be increased.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

set
drawnow
surf
poissrnd

CHAPTER

28

Neural Networks Part I: Unsupervised Learning

28.1. GOALS OF THIS CHAPTER

This chapter has two goals that are of equal importance. The first goal is to become familiar with the general concept of unsupervised neural networks and how they may relate to certain forms of synaptic plasticity in the nervous system. The second goal is to learn how to apply neural networks using Neural Networks Toolbox built into the MATLAB® software to address a particular problem.

28.2. BACKGROUND

Neural networks have assumed a central role in a variety of fields. The nature of this role is fundamentally dualistic. On the one hand, neural networks can provide powerful models of elementary processes in the brain, including processes of plasticity and learning. On the other hand, they provide solutions to a broad range of specific problems in applied engineering, such as speech recognition, financial forecasting, or object classification.

28.2.1. But What Is a Neural Network?

Despite the "biological" sounding name, neural networks are actually quite abstract computing structures. In fact, they are sometimes referred to as *artificial neural networks*. Essentially, they consist of rather simple computational elements that are connected to each other in various ways to serve a certain function. The architecture of these networks was inspired by mid- to late-20th century notion of brain function, hence, the term *neural*.

At its conceptual core, a unit in a neural network consists of three things (see Figure 28.1):

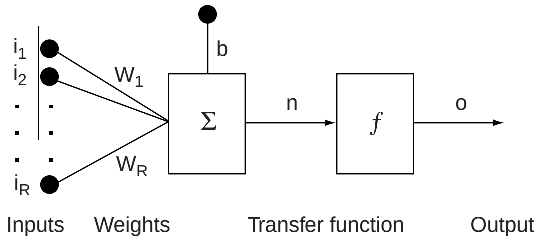


FIGURE 28.1 The concept of a neural network.

- A set of inputs that can vary in magnitude and sign coming from the outside world or from other neurons in the network.
- A set of weights operating on these inputs that can vary in magnitude and sign (implementing synaptic efficiency and type of synapse on a neuron). There is also a bias weight, *b*, that operates on an input that is fixed to a value of 1.
- A transfer function that converts the net input to an output (e.g., summation—or whatever it is that the neuron is doing).

The output of a unit can then become the input to another unit. Individual units are typically not very functional or powerful. Neural networks derive their power from connecting large numbers of neurons in certain configurations (typically called *layers*) and from learning (i.e., setting the weights of these connections).

Neural networks are extremely good at learning a particular function (such as classifying objects). There are several different ways to train a neural network, and you will become acquainted with the most common ones in this and the next chapter.

Such trained multilayer networks are extremely powerful. It has been shown that a suitable two-layer network can approximate any computable function arbitrarily well. In other words, neural networks that are set up properly can do anything that can be done computationally. This is what makes them so appealing for applied engineering problems because the problem solver might not always be able to explicitly formulate a solution to a problem, but one might be able to create and train a neural network that can solve the problem, even if one doesn't understand how. For example, a neural network would be particularly useful to control the output of a sugar factory, given known inputs—a task at which humans have been shown not to be particularly good.

28.2.2. Unsupervised Learning and the Hebbian Learning Rule

Despite the fact that neural networks are very far from real biological neural networks, the learning rules that have been developed to modify the connections between computing elements in neural networks resemble properties of synaptic plasticity in the nervous system. In this chapter we will focus on *unsupervised learning rules* (in contrast to supervised or error-correcting learning rules), which turn out to be very similar to Hebbian plasticity rules that have been discovered in the nervous system. Unsupervised learning tries to capture the statistical structure of patterned inputs to the network without an explicit teaching signal. As will be clear in a moment, these learning rules are sensitive to correlations between components of patterned inputs and strengthen connections between components that are correlated and weaken connections that are uncorrelated. These learning rules serve at least three computational functions: (1) to form associations between two sets of patterns, (2) to group patterned inputs that are similar into particular categories, and (3) to form content-addressable memories such that partial patterns that are fed to the network can be completed.

Donald Hebb was one of the first to propose that the substrate for learning in the nervous system was synaptic plasticity. In his book *The Organization of Behavior*, Hebb stated, "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased" (1949, p. 62). In fact, William James, the father of American psychology, formulated the same idea almost 60 years earlier in his book *The Principles of Psychology* when he stated, "When two elementary brain-processes have been active together or in immediate succession, one of the them, tends to propagate its excitement into the other" (1890, p. 566). Nevertheless, the concept of synaptic plasticity between two neurons that are co-active is usually attributed to Donald Hebb. Mathematically, Hebbian plasticity can be described as:

$$\Delta w_{ij} = \varepsilon \cdot pre_i \cdot post_j \tag{28.1}$$

where Δw_{ij} denotes the change in synaptic weight between a presynaptic neuron i and a postsynaptic neuron j, ε is a learning constant that determines the rate of plasticity, prei is the activity of the presynaptic neuron i, and post$_j$ is the activity of the postsynaptic neuron j. The Hebbian learning rule states that a synapse will be strengthened when the presynaptic and postsynaptic neurons are active. Neurophysiologically, this means that the synapse will be potentiated when the presynaptic neuron is firing and the postsynaptic neuron is depolarized. Bliss and Lomo (1973) first showed that the synapses between the perforant pathway and the granule cells in the dentate gyrus of the hippocampus could be artificially potentiated using a stimulation protocol that followed the Hebbian learning rule. The effects of the stimulation protocol they used has been termed *Long-Term Potentiation* because the synapses appear to be potentiated indefinitely. Since that time, many other experiments have shown that Long-Term Potentiation could be implemented in many parts of the brain, including the neocortex. *Long-Term Potentiation* with initial capital letters, which refers to an artificial stimulation protocol, should be distinguished from the lowercase term *long-term potentiation*, which refers to the concept that synapses may be potentiated naturally when some form of associative learning takes place.

From a computational perspective, the simple Hebbian rule can be used to form associations between two sets of activation patterns (Anderson et al., 1977). Imagine a feedforward network consisting of an input (presynaptic) and output (postsynaptic) set of neurons, f and g, respectively, that are fully connected as shown in Figure 28.2. The transfer function of the f neurons is assumed to be linear, and thus, this network is referred to as a *linear associator* (Anderson et al., 1977). The activation of each set of neurons can be viewed as column vectors, $\overrightarrow{f}$ and $\overrightarrow{g}$. Assume you want to associate a green traffic light with "go," a red traffic light with "stop," and a yellow traffic light with "slow." Furthermore, assume that the green, red, and yellow traffic lights are coded as mutually orthogonal and normal (i.e., unit length) $\overrightarrow{f}$ vectors.

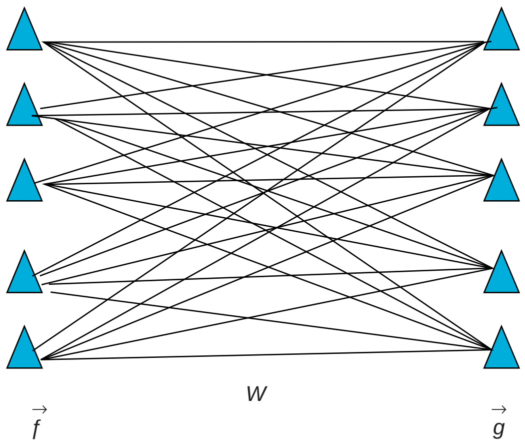


FIGURE 28.2 A simple linear associator network composed of an input and output set of neurons that are fully connected.

The Hebbian learning rule can be used to form these associations. Mathematically, the learning rule generates a weight matrix as an outer-product between the f and g vectors:

$$W = \vec{g}\,\vec{f}^{\,T} \tag{28.2}$$

$$W = \vec{g}_{go}\vec{f}^{\,T}{}_{green} + \vec{g}_{stop}\vec{f}^{\,T}{}_{red} + \vec{g}_{slow}\vec{f}^{\,T}{}_{yellow} + \ldots \tag{28.3}$$

Now, if you probe the network with the green presynaptic pattern, you get the following:

$$W\vec{f}_{green} = \vec{g}_{go}\vec{f}^{\,T}{}_{green}\vec{f}_{green} + \vec{g}_{stop}\vec{f}^{\,T}{}_{red}\vec{f}_{green} + \vec{g}_{slow}\vec{f}^{\,T}{}_{yellow}\vec{f}_{green} + \ldots \tag{28.4}$$

Given that the input patterns are orthogonal to each other and normal, the output of the network is "go":

$$W\vec{f}_{green} = \vec{g}_{go} \cdot 1 + 0 + 0 + \ldots \tag{28.5}$$

28.2.3. Competitive Learning and Long-Term Depression

Despite its elegant simplicity, the Hebbian learning rule as formulated in Equation 28.1 is problematic because it only allows for potentiation, which means that the synapse will only grow stronger and eventually saturate. Neurophysiologically, it is known that synapses can also depress using a slightly different stimulation protocol. Fortunately, there is a neural network learning rule that can either potentiate or depress. It was proposed by Rumelhart and Zipser (1985) and is referred to as the competitive learning rule:

$$\Delta w_{ij} = \varepsilon \cdot pre_i \cdot post_j - \varepsilon \cdot post_j \cdot w_{ij} \tag{28.6}$$

The first term on the right side of Equation 28.6 is exactly the Hebbian learning rule. The second term, however, will depress the synapse when the postsynaptic neuron is active, regardless of the state of the presynaptic neuron. Therefore, if the presynaptic neuron is not active, the first term goes to 0 and the synapse will depress. Also, notice that depression is proportional to the magnitude of the synaptic weight. This means that if the weight is very large (and positive), depression will be stronger. Conversely, if the weight is small, depression will be weaker. The competitive learning rule can be described equivalently as follows:

$$\Delta w_{ij} = \varepsilon \cdot post_j \cdot (pre_i - w_{ij}) \tag{28.7}$$

The formulation in Equation 28.7 makes it clear what the learning rule is trying to do: learning will equilibrate (i.e., terminate), when the synaptic weight matches the activity of the presynaptic neuron. On a global scale, this means that the learning rule is trying to develop a matched filter to the input and can be used to group or categorize inputs. To make this clearer, consider two kinds of patterned inputs corresponding to apples and oranges. Each example of an apple or orange is described by a vector of three numbers that describes features of the object such as its color, shape, and size. Consider the problem of developing a neural network to categorize the apples and oranges (see Figure 28.3).

Imagine the apple and orange vectors clustered in a three-dimensional space. The weights feeding into either the apple unit or orange unit can also be viewed as vectors with the same dimensionality as the input vectors. Before learning, the apple and orange weight vectors are pointing in random directions (see Figure 28.4). However, after learning, the weight vectors will be pointing toward the center of the apple and orange input vector clusters because the competitive learning rule will try to move the weight vectors to match the inputs (see Figure 28.5). Finally, notice how the two category units are mutually inhibiting each other (the black circles indicate fixed inhibitory synapses that do not undergo plasticity). This mutual inhibition allows only one unit to be active at one time so that only one weight vector is adjusted for a given input vector.

28.2.4. Neural Network Architectures: Feedforward Versus Recurrent

As with real neural circuits in the brain, artificial neural network architectures are often described as being *feedforward* or *recurrent*. Feedforward neural networks process signals in a one-way direction and have no inherent temporal dynamics. Thus, they are often described as being *static*. In contrast, recurrent networks have loops and can be viewed as

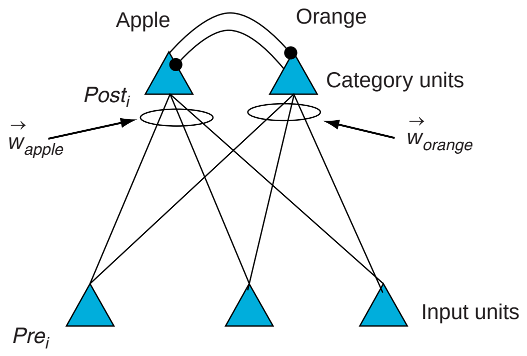


FIGURE 28.3 A two-layer neural network that takes input vectors corresponding to apples and oranges and categorizes them by activating one of the two category units.

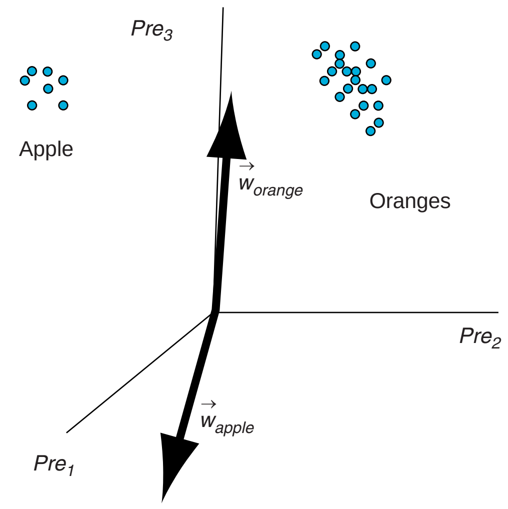


FIGURE 28.4 The weight vectors in an untrained competitive learning neural network.

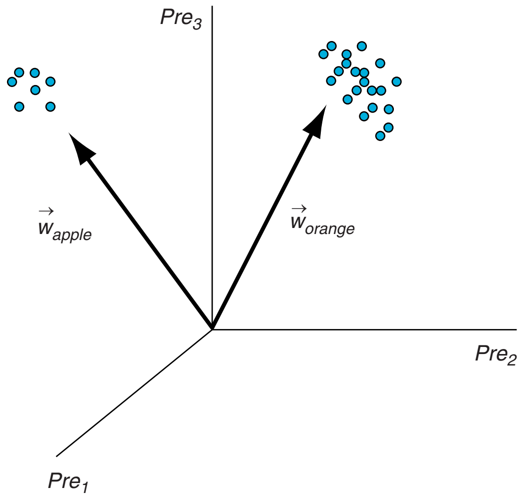


FIGURE 28.5 The weight vectors in a competitive learning neural network that has been trained.

a dynamic system whose state traverses a state space and possesses stable and unstable equilibria. The linear associator described previously is an example of a feedforward network. The competitive learning network is a sort of hybrid network because it has a feedforward component leading from the inputs to the outputs. However, the output neurons are mutually connected and, thus, are recurrently connected.

An example of purely recurrent neural network is the Hopfield network. A Hopfield network uses a Hebbian-like learning rule to generate stable equilibria corresponding to patterns that are desired to be stored. When partial patterns of stored patterns are provided to a Hopfield network, the network's state will tend to progress toward its stable equilibrium corresponding to the stored pattern and, thus, complete the pattern. This is an example of a content-addressable memory.

28.3. TRYING OUT A NEURAL NETWORK

28.3.1. Neural Network Toolbox in MATLAB

MATLAB itself does not ship with specialized functions to make neural network applications easier. Of course, this is not strictly necessary. Everything that constitutes a neural network (inputs, weights, transfer function, and outputs) can be implemented using matrices and matrix operations, which do ship with MATLAB. Of course, implementing them would be extremely time-consuming. You probably wouldn't want to do that. Particularly the functions for training the network can be rather intricate. Fortunately, The MathWorks has developed a specialized toolbox for neural networks: Neural Network Toolbox.

28.3.2. Competitive Learning and Multilayer Networks

You can create a competitive learning neural network using Neural Network Toolbox with the function **newc**. Suppose you want to categorize the following six two-dimensional vectors into two classes:

```
>> inp = [0.1 0.8 0.1 0.9 0.3 0.7; 0.2 0.9 0.1 0.8 0.2 0.9];
```

Each column of this matrix represents one two-dimensional input vector.

There are three vectors near the origin and three vectors near (1,1). What now?

First, create a two-neuron layer (the minimum for a competitive network) with two input elements ranging from 0 to 1 by typing:

```
>> mynet = newc([0 1; 0 1], 2);
```

Your two-neuron competitive learning neural network is then created. The preceding syntax means that you created a new neural network **mynet** as a competitive learning network with the limits for the two inputs of 0 and 1, and your network consists of two output neurons.

Typing **mynet** again shows you the properties of your neural network. You could have given it any other name. If you look in your workspace, you will see that a "network" structure of this name has been created. A structure has many properties. For example, you can see that the input weights are set to 0.5 by typing **mynet.IW{1,1}**. The weights are initialized to the centers of the input ranges with the function **midpoint**. This is automatically called by **newc** and applied to the weights. You will notice that **mynet.IW** is a cell array with one element. That element contains a 2×2 matrix where each row corresponds to all the weights feeding one of the two neurons. Typing **mynet.b{1,1}** will show the initial biases of the two neurons. The biases are fairly set by **initcon**, which is also automatically called by **newc** and yields the biases 5.4366 and 5.4366.

As mentioned previously, weights of biases connect inputs that are always set to 1. Therefore, the bias on each neuron can be viewed as the baseline activity of that neuron without any modulatory inputs. Most of the property values in the network structure are contained in "cells." In general, you can access properties of structures by using the syntax **structurename.propertyname** (do not include spaces between the names). This is very important because all networks are structures.

This network needs to be trained to classify properly. In this network, the neurons compete to respond to the input. You can train for 1000 epochs by setting:

```
>> net.trainParam.epochs = 1000
```

Then start the training by typing:

```
>> mynet = train(mynet,inp);
```

Look at your final weights and biases: **mynet.IW{1,1} ; mynet.b{1,1}**

Use this net to classify your original input:

```
>> output = sim(mynet,inp)
```

The response will be in terms of sparse vectors, so you want to convert that to indices:

```
>> outputindex = vec2ind(output)
```

It should produce something like this:

```
>> outputindex = 1  2  1  2  1  2
```

This result indicates that the first, third, and fifth input vectors activated neuron 1 and the second, fourth, and sixth input vectors activated neuron 2.

28.3.3. Recurrent Network

For this example, use the Hopfield neural network command **newhop** to store two four-dimensional patterns. The command **newhop** is actually an adapted version of the original Hopfield network and stores patterns at the corners of a hypercube with values of 1 or –1 at the corners. The two column vectors you will store are:
[1 1 −1 −1]′ and [−1 −1 1 1]′

Create a matrix, *T*, composed of these two column vectors and train the network as follows:

```
>> net = newhop(T)
```

Now test to see whether those two patterns were stored in the network:

```
>> [Y,Pf,Af] = sim(net,2,[ ],T);
```

The variable *Y* gives you the states to which the network equilibrated using the inputs *T*.

Now create partial input patterns, *T2*, and see where the network equilibrates:

```
>> T2=[1 1 0 0; 0 0 −1 −1]';
```

28.4. PROJECT

Greebles live in dangerous times (Gauthier and Tarr, 1997). Recent events led to the creation of the "Department for Greeble Security." For this project, you are a programmer for this recently established ministry and your job is to write software that distinguishes the "good" Greebles from the "bad" Greebles (see Figure 28.6). Researchers in another section of the department have shown that three parameters correlate with the tendency that a Greeble is good or bad. These parameters, identified in Figure 28.7, are "boges" length, "quiff" width, and "dunth" height (Gauthier, Behrmann, and Tarr, 2004). Specifically, it has been shown that good Greebles have long boges, thin quiffs, and high dunths, whereas the bad Greebles tend to have short boges, thick quiffs, and low dunths. Of course, this relationship is far from perfect.

A given individual Greeble might have any number of variations of these parameters. In other words, this classification is not as clearcut and easy as your superiors might want it to be. That's where you come in. You decide to solve this problem with a neural network, since you know that neural networks are well suited for this kind of problem.

In this project you will be asked to create two neural networks.

1. The first neural network will be a competitive learning network that distinguishes good from bad Greebles. Specifically, you should do the following:
   a. Train the network with the training set on the companion website (it contains data on Greebles who have been shown to be good or evil in the past, along with their parameters for boges length, quaff width, and dunth height).

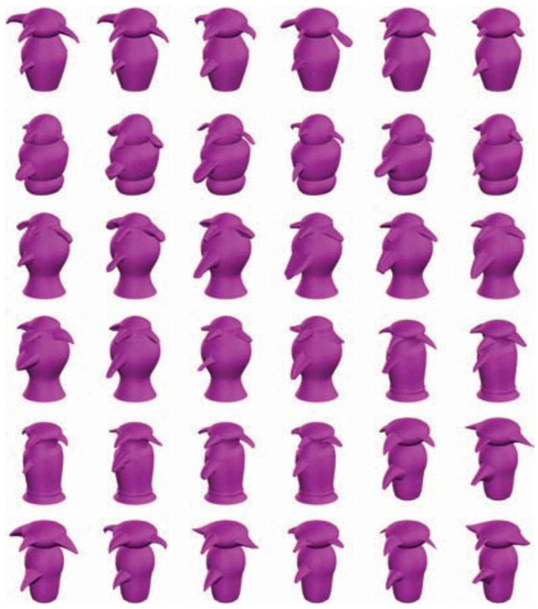


FIGURE 28.6 Meet the Greebles. Image courtesy of Michael J. Tarr, Brown University, http://www.tarrlab.org/

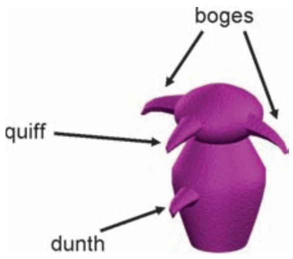


FIGURE 28.7 The anatomy of a Greeble. Image courtesy of Michael J. Tarr, Brown University, http://www.tarrlab.org/

b. Test the network with the test set on the companion website (it contains parameters on Greebles that were recently captured by the department and suspected of being bad). Use your network to determine if they are (more likely to be) good or bad.
c. Document these steps, but make sure to include a final report on the test set. Which Greebles do you (your network) recognize as being bad, and which do you recognize as being good?
d. Qualitatively evaluate the confidence that you have in this classification. Include graphs and figures to this end.
Good luck! The future and welfare of the Greebles rest in your hands.

Hints:

- Load the two training populations from the companion website using the command **xlsread('*filename*')**. Each file contains measurements of three parameters (in inches): boges length, quaff width, and dunth height. Each row represents an individual Greeble.
- Before you do anything else, you might want to plot your populations in a three-dimensional space (you have three parameters per individual). You can do this by using **plot3(*param1,param2,param3*)**. In other respects, **plot3** works just like **plot**.
- Merge the data into a big training vector.
- Create the competitive network.
- Train the competitive network.
- Download the test files and test the population with your trained network.
- Your program should produce a final list indicating which Greebles in the test population are good and which are bad. Also, graph input weights before and after training.
- *Disclaimer: No actual Greebles were hurt when preparing this project.*

2. The second neural network that you will create is a Hopfield network that will store the prototypical good and bad Greebles. Specifically, you should do the following:
   a. Create the prototypical good and bad Greebles by taking the average features of the good and bad Greebles.
   b. Normalize the prototypical features so that the largest feature value is 1 and the lowest feature value is –1.

c. Build a Hopfield network to store the good and bad prototypes (i.e., two feature vectors).
d. Use the test set from the companion website to see whether the Hopfield network can categorize the suspected Greebles as prototypical good or bad Greebles. Compare these results with the results using the competitive learning network.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

rand ('state', number)
.
{ }
sim
newc
train
cat
midpoint
initcon
vec2ind
plot3
xlsread
newhop

CHAPTER

29

Neural Network Part II: Supervised Learning

29.1. GOALS OF THIS CHAPTER

This chapter has two primary goals. The first goal is to introduce to you the concept of supervised learning and how it may relate to synaptic plasticity in the nervous system, particularly in the cerebellum. The second goal is for you to learn to implement single-layer and multilayer neural network architectures using supervised learning rules to solve particular problems.

29.2. BACKGROUND

29.2.1. Single-Layer Supervised Networks

Historically, perceptrons were the first neural networks to be developed and happened to employ a supervised learning rule. Inspired by the latest neuroscience research of the day, McCulloch and Pitts (1943) suggested that neurons might be able to implement logical operations. Specifically, they proposed a neuron with two binary inputs (0 or 1), a threshold that can be met or not, and a binary output (0 or 1). In this way, logical operators like AND or OR can be implemented by such a neuron (by either firing or not firing if the threshold is met or not). See Table 29.1 for an example of implementing the logical AND operator with a threshold value of 2.

Later, Frank Rosenblatt (1958) used the McCulloch and Pitts Model and recent theoretical developments by Hebb to create the first perceptron. It is a modified McCulloch and Pitts neuron, with an arbitrary number of weighted inputs. Moreover, the inputs can have any magnitude (not just binary), but the output of the neuron is 1 or 0. The weight can be different for each input. Setting the weights differently allows for more powerful computations.

TABLE 29.1 Perceptron Implementing AND with a Threshold of 2

Input 1	Input 2	Sum	Comparing with Threshold	Output
0	0	0	<2	0
0	1	1	<2	0
1	0	1	<2	0
1	1	2	=2	1

Perceptrons are good at separating an input space into two parts (the output). Training a perceptron amounts to adjusting the weights and biases such that it rotates and shifts a line until the input space is properly partitioned. If the input space is higher than two dimensions, the perceptron implements a hyperplane (with one dimension less than the input space) to partition it into two regions. If the network consists of multiple perceptrons, each can achieve one partition. The weights and biases are adjusted according to the *perceptron learning rule*:

1. If the output is correct, the weight vector associated with the neuron is not changed.
2. If the output is 0 and should have been 1, the input vector is added to the weight vector.
3. If the output is 1 and should have been 0, the input vector is subtracted from the weight vector.

It works by changing the weight vector to point more toward input vectors categorized as 1 and away from vectors categorized as 0.

Although a real neuron either fires an action potential or not (i.e., generates a 1 or 0), the response of a neuron is often described by its firing rate (i.e., the number of spikes per unit time), resulting in a graded response (Adrian and Matthews, 1927). To model these responses, you will use a different neural network model, called a *linear network*. The main difference between linear networks and perceptrons lies in the nature of the transfer function. Where the perceptron had a step function to map inputs to (binary) outputs, a linear network has a linear transfer function. The learning rule for the linear network, called the *Widrow-Hoff learning rule*, is essentially the same as that for the perceptron (Widrow and Hoff, 1960). This rule compares targets and outputs (specifically, it subtracts them) and squares the difference. It then sums all these differences and takes the mean to arrive at the mean squared error [MSE = mean (targets – outputs) ^ 2]. It then sets the weights and biases such that the mean square error decreases from epoch to epoch. It does this by taking the derivative of the MSE with respect to each weight, thereby following the gradient of the MSE in weight space. Since the preceding MSE equation is quadratic, this error function will have one global minimum (if it has any). Hence, you can be assured that the Widrow-Hoff rule will find this minimum by gradually descending into it from the starting point (given by the initial input weights). In other words, you are moving into the minimum of an error surface. The MATLAB® software includes a visual demonstration of that: **demolin1.** The Widrow-Hoff learning rule is expressed mathematically as:

$$\Delta w_{ij} = \epsilon\left(t_j - post_j\right) pre_i \tag{29.1}$$

where Δw_{ij} is the weight change between input i and neuron j, ε is the learning rate constant, t_j is the target on neuron j, $post_j$ is the output of postsynaptic neuron j, and pre_i is the presynaptic input i.

29.2.2. Multilayer Supervised Networks

Since perceptrons are vaunted for their ability to implement and solve logical functions, it came as quite a shock when Minsky and Papert (1959) showed that a single layer perceptron can't solve a rather elementary logical function: XOR (exclusive or) (see Figure 29.1). This finding also implies that all similar networks (linear networks, etc.) can solve only linearly separable problems. These events caused a sharp decrease in the interest in neural networks until it resurged in the 1980s.

The creation of non-linear multilayer networks with hidden units and learning rules to train them such as the backpropagation network was one important reason for the resurgence of neural networks. These networks can implement nonlinear (yet differentiable) transfer functions. As a matter of fact, properly set up, these networks can approximate any function.

29.2.3. Supervised Learning in Neurobiology

Although there is no definitive evidence for neural plasticity that is guided by a "teaching" signal as supervised learning requires, there are some intriguing experimental findings which suggest the possibility that the physiology of the cerebellum may support a form of supervised plasticity. David Marr and James Albus independently proposed that the unique and regular anatomical architecture of the cerebellum could instantiate error-based plasticity that might underlie motor learning. In particular, they proposed

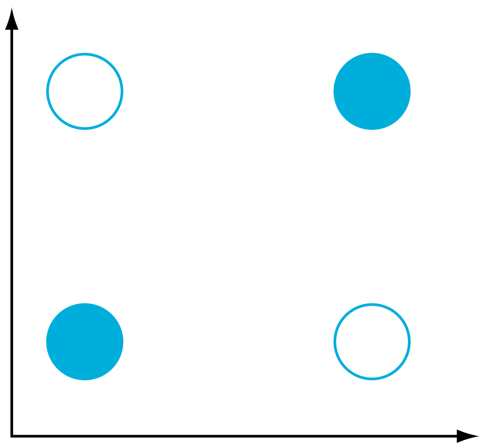


FIGURE 29.1 The XOR problem that a single layer network cannot solve. The XOR problem requires that the neuron respond (i.e., white circles) when only one of the inputs is on but not both. This is not solvable by a single-layer perceptron or linear network because it is not linearly separable.

that the climbing fibers acting on Purkinje cells from the inferior olive could provide an error signal to modify the synapses between the parallel fibers and the Purkinje cells. Experimental support for this theory first came from Ito and Kano (1982), who discovered that long-term depression could be induced in the synapse between the parallel fibers and the Purkinje cells. Through electrical stimulation of the parallel fibers and at the same time stimulation of the climbing fibers, each of which forms strong synaptic contacts with a particular Purkinje cell, the parallel fiber synapse could be depressed. Because the Purkinje cells are inhibitory on the deep cerebellar nuclei, this synaptic depression would disinhibit the deep cerebellar nuclei.

A number of motor learning experiments have provided additional support for the idea that the cerebellum supports supervised learning via the climbing fibers. Sensory-motor adaptation experiments in which the gain between the movement and its sensory consequences are altered have shown transient increases in climbing fiber input (as measured by the complex spike rate) during learning (Ojakangas and Ebner, 1992, 1994). In addition, classical conditional experiments have suggested that the cerebellum plays a role in learning associations between unconditioned stimuli (US) and conditioned stimuli (CS) (Medina et al., 2000). For example, learning the association between an air puff (US) generating an eye-blink response and a tone (CS) is disrupted by reversible inactivation to parts of the cerebellum (Krupa, Thompson, and Thompson, 1993). Moreover, it has been shown that the tone enters the cerebellar cortex via the parallel fiber pathway, whereas the air puff, acting like a teacher, enters through the climbing fiber input. In fact, a mathematical formulation of classical conditioning proposed by Rescorla and Wagner (1972) quite closely resembles the Widrow-Hoff error correction learning rule used in supervised neural networks.

29.3. EXERCISES

29.3.1. Perceptrons

For this example, start by creating a single perceptron using the following command:

```
>> mynet = newp([-5 +5] , 1)
```

Your one neuron perceptron neural net is then created. The preceding syntax means that you create a new neural network named **mynet** as a perceptron with the limits for the single input of –5 and +5, and your network consists of only one neuron. Typing **mynet** again shows you the properties of your neural network. You could give it any other name. If you look in your workspace, you will see that a "network" structure of this name has been created. A structure has many properties. For example, you can see that the input weight is set to 0 by typing **mynet.IW**. Typing **mynet.b** will show the initial bias, and it will also be 0. Of course, you can change the bias and the initial weights. We already discussed the significance of weights. Biases basically add a threshold to the neuron. If the net sum of a perceptron is larger than 0, the output will be 1; otherwise, it will be 0. Adding a bias changes the point where this decision is made. For example, if the bias is –5, the net sum has to exceed 5 for the output to yield 1 (the bias of –5 is subtracted from the net sum; if it is less than 5, it will fall below 0).

An important point to note is that most of the property values in this network structure are contained in "cells." A cell is basically a meta-matrix—in other words, a variable that contains matrices. The syntax for accessing cells is different than that for accessing matrices themselves.

You access a cell by using wavy or curly brackets: { }. For example, set the initial weight to 1 by typing:

```
>> mynet.IW{1,1} = 1
```

You can make sure that this change took effect by typing **mynet.IW**.

> ***Exercise 29.1:*** Change the bias to 5.

So far, you haven't done very much with neural networks. The network is just kind of sitting there. You can change that now.

Create a new network with an arbitrary name. In this case, call it **Hans**. It will have two inputs that can range from –5 to 5. But it will be a single neuron.

```
>> Hans = newp([-5 5; -5 5],1)
```

Now you need to set the input weights. Currently, they are set to 0. Hence, every input will yield 0. Set the input weights to 1 and –1, respectively:

```
>> Hans.IW{1,1} = [1 -1]
```

If the sum of the input times the weights meets or exceeds 0, the network should return 1; otherwise, it will return 0. Try it by feeding it some input:

```
inp1 = [1; 0.5]       %Note: The first, positive weighted input larger than the second
inp2 = [0.5; 1]     %Note: The second, negative weighted input is larger than the first
```

Type the following command to see what the output of your network is:

```
>> sim(Hans,inp1)

ans =
  1

>> sim(Hans,inp2)

ans =
  0
```

And, yes, the network classified these inputs correctly. You can now feed the network a large number of random numbers. Look to see if they are classified correctly, like this:

```
>> a = rand(2,10)       %Create 20 random numbers, arranged as 2 rows, 10 columns
>>
>> output = sim(Hans,a)
```

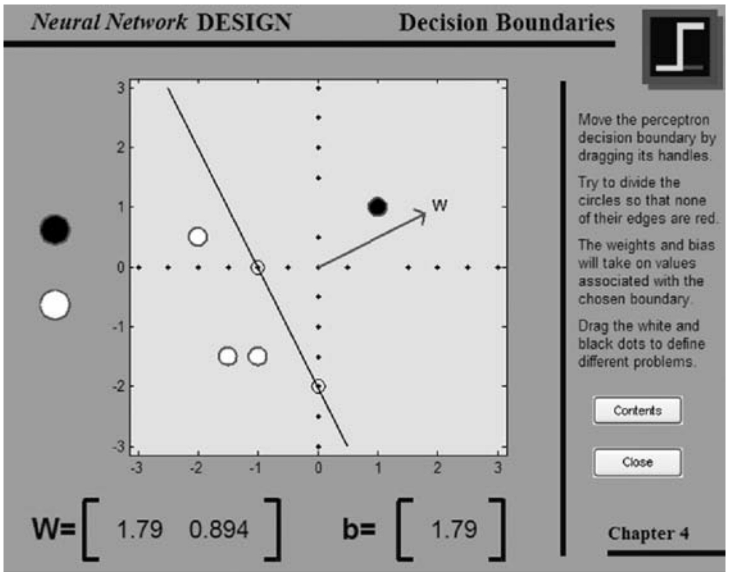
Compare the logical values in **output** with the values in the input. If the value in the first input row is larger than the value in the second input row (for a given column), the output value (for that column) should be 1; otherwise, it should be 0.

Exercise 29.2: Adjust the bias to some arbitrary value and see how the input/output mapping changes.

When you adjust the weights and the bias, it can be shown that any linearly separable problem (a problem space that can be separated by a line—or more generally by a hyperplane) can be solved by a perceptron. To verify whether this is the case, play around with the interactive perceptron, where you can arbitrarily set the decision boundary yourself. Type **nnd4db** (see Figure 29.2). Try to separate the white and black circles; they represent the inputs. The output is represented by the black line, creating two regions, one region corresponding to the number 0 and the other region corresponding to the number 1.

Exercise 29.3: Re-create this perceptron (its weights and biases) on the command line and see whether the demo is accurate.

What if you don't know the weights or don't want to find the weights? What if you only know the problem? Luckily, one of the strongest functions of neural networks is their ability to learn—to solve problems like this on their own. You will do this now. The first thing you need is a learning rule, a rule that tells you how to update the weights (and biases), given a certain existing relationship between input and output. The perceptron learning rule (**learnp**) inherent to Neural Network Toolbox in MATLAB is an instance of supervised learning, giving the network pairs of inputs and desired (correct) output. To test this, set the input weights of your neuron to a different level, one that will produce "wrong outputs," given how the inputs were classified previously. Use those known input/output



Move the perceptron decision boundary by dragging its handles.

Try to divide the circles so that none of their edges are red.

The weights and bias will take on values associated with the chosen boundary.

Drag the white and black dots to define different problems.

W= [1.79 0.894] b= [1.79]

FIGURE 29.2 An interactive display of the decision boundary of a perceptron provided in Neural Network Toolbox in MATLAB. The display shows how a perceptron creates a linear decision boundary whose slope and intercept can be modified by adjusting the weight vector and bias.

mappings as a training set. Make sure to reset the bias of the "Hans" network back to 0, so as not to complicate things (technically, a bias is a weight with an input that is always 1).

For example:

```
>> w = [0.6 -0.9]             %New weights. Before: [1 -1]
>> Hans.IW{1,1} = w;          %Setting the input weights function to w
>> inp1 = [0.846; 0.723];     %This was my first random number pair
>> Output = 1;                %The output was positive, as it should be
```

Typing something like **Newoutput = sim(Hans,inp1)** will now yield 0.

```
>> Error = Output – Newoutput;
```

Change in weights:

```
>> dw = learnp(w,inp1,[ ], [ ], [ ], [ ], Error, [ ], [ ], [ ]) %learnp takes many arguments.
                                                                %Don't worry about most of them now.
```

New weights:

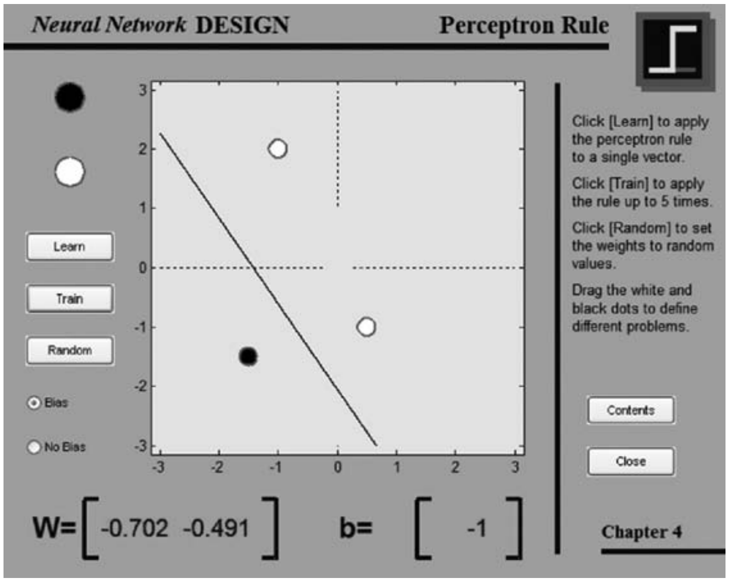
```
>> w = w + dw
```

You can try this dynamically, in an interactive demo, by typing **nnd4pr** (see Figure 29.3).

29.3.2. Linear Networks

Now create a linear network. The syntax is analogous to the syntax that created the perceptron network:

```
>> newnet = newlin( [-10 10; -10 10],1);
```



Click [Learn] to apply the perceptron rule to a single vector.

Click [Train] to apply the rule up to 5 times.

Click [Random] to set the weights to random values.

Drag the white and black dots to define different problems.

W=[-0.702 -0.491] b=[-1]

FIGURE 29.3 An interactive display of the perceptron learning rule provided in Neural Network Toolbox.

The first argument is a matrix that specifies the range of the two inputs. The second argument is the number of neurons in the network. More precisely, it is the number of outputs of the network. For now, set both input weights to 1:

```
>> newnet.IW{1,1} = [1 1];
```

See if the thing works properly. It should take the dot product of the inputs:

```
>> inp = [5; 7];
>> output = sim(newnet,inp)

output =
  12
```

Since the weights were both 1, the dot product amounts to simple addition in this case.

Change the weights to 3 and 4 and see what happens:

```
>> newnet.IW{1,1} = [3 4];
>> output = sim(newnet,inp)

output =
  43
```

The same can be gained by typing:

```
>> [3 4] * [5;7]
```

In other words, this neural network implements an inner product (dot product).

Exercise 29.4: See if this example generalizes. Try negative numbers. What if the input goes beyond the range with which the network was initialized?

Exercise 29.5: Change the bias. How does that influence the calculation of the dot product?

Of course, you want a more useful neural net than just one that can take the dot product; you can take the dot product without neural nets. A classical function of linear neural networks is the automatic classification of input objects into different categories. To achieve that, you need to train the network. To do this, you use an automated version of the perceptron training rule **learnp**; it's called **train**. It goes through all the inputs, updates the weights and biases according to **learnp**, and then goes through all the inputs again. It then calculates a mean square error by comparing the outputs of the weight-adjusted net with the target outputs (i.e., the Widrow-Hoff learning rule). It uses a least mean square algorithm to minimize the mean square error (by changing the weights) until either:

a. A given target mean square value is reached
b. The maximum number of training runs (epochs) is reached

Why don't you give it a try? Say you have six target outputs from two sets, **set1** and **set2**:

```
>> set1 = [2, 2; -2, 2 ; 0.5, 1.5];
>> set2 = [1, -2; -1, 1; -0.5 -0.5];
```

Now plot them to see what is going on:

```
>> figure %Opening a new figure
>> plot(set1(:,1),set1(:,2),'*') %Plotting the first set
>> hold on; %Holding on
>> plot(set2(:,1),set2(:,2),'*', 'color', 'k') %Plotting the second set in black
```

Looking at the graph in Figure 29.4, you can see what is going on. You can also see that the problem is, in principle, solvable: you can draw a line that separates the blue and black stars. Now create a network that will find this solution. Starting at 0 inputs weights and 0 bias.

First, put these sets in the proper input form. Remember that for neural networks, rows are different input dimensions, and columns are different examples in those dimensions (your points). To achieve this, you have to concatenate and transpose the matrices. You concatenate them along the column dimension by using **cat**:

```
>> inputset = cat(2, set1', set2')
```

Then you assign the corresponding targets. Arbitrarily assign 1 to the blue set and 0 to the black set:

```
>> targets = [1 1 1 0 0 0]
```

Now create a new network that you will train; call it **netz**. Set the proper range (nothing in the input set goes beyond that):

```
>> netz = newlin( [-2 2; -2 2],1);
```

Finally, you have to set a training goal. Say you're happy if the mean squares error falls below 0.08:

```
>> netz.trainParam.goal= 0.08; %That's low. Let's be picky.
```

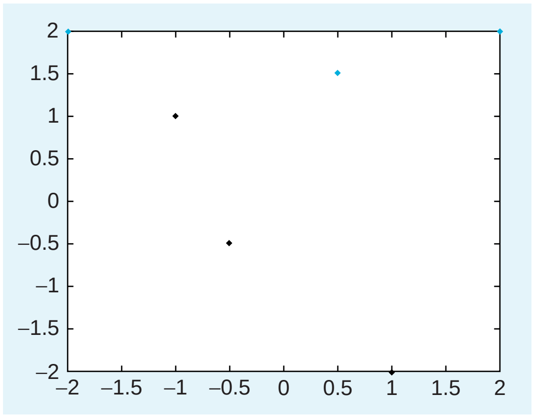


FIGURE 29.4 A plot of six target outputs (blue and black stars) that are to be classified.

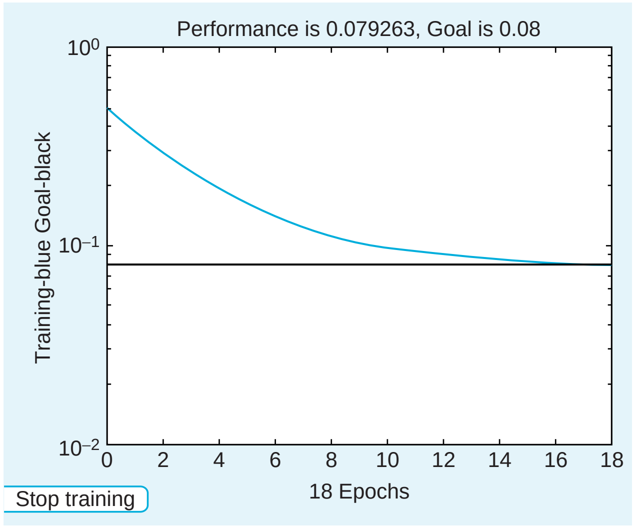


FIGURE 29.5 The mean square error as a function of training epochs.

A final train command gets you going:

```
>> [netz, training] = train(netz,inputset,targets); %Specifying two outputs on the left hand
```

What happens? If you did everything right, something like this should happen: the sample net achieved the goal in 18 training epochs (see Figure 29.5).

These training data are stored in the training matrix. You also have a new **netz**. Challenge the new **netz** with some input and see whether it correctly classifies the input. Pick something in the middle of the black range, like –1, 0:

```
>> inp4 = [-1; 0]
>> sim(netz,inp4)
ans =
   0.1453
```

Close but not quite. Of course, the result should be 0, but this might be the best you can do, given the sparse input.

Exercises 29.6: Explore some points in the space and see what their output is. Is it about what you expect? Does the network make gross errors? What about the initial input vector?

Exercise 29.7: Run with more ambitious training goals, like 0.01. Can you reach it, given the sparse input (six input data points)?

Exercise 29.8: **Netz.trainParam** stores the training parameters. Type **netz.trainParam** and see how many epochs it runs. Then change it to a higher value and see if it converges later. What about 1000 epochs? What about 10000?

A linear classifier allows you to roughly categorize and classify inputs if they are linearly separable. Adjusting the weights and biases amounts to creating a linear transfer function that separates the desired outputs maximally and optimally. Of course, there is only so much that a linear function can do. But it's not too bad.

A supervised linear network can be applied to the Greeble problem used in Chapter 28, "Neural Networks Part I: Unsupervised Learning". Note that there is some initial classification just by setting all the initial weights to 1 (see Figure 29.6A). But it is not very good, and the absolute output values are all over the place (4000?). The final classification by the linear network is pretty good (see Figure 29.6B). Values cluster around 1 and 0, although there is quite a bit of variance. The linear network couldn't separate the clusters any better than this, given the variance in the input and the amount of training. You should always check your network by visualizing the outputs of trained and untrained networks with simple plots like this.

There is one major caveat regarding this kind of learning: The default learning method **learnb** (batch learning) updates all the weights and inputs at once, taking all the inputs into account at once. Hence, the network might not learn at all if there are too many inputs. Here, you never come close to the minimum of the mean square error. This is characterized by a runaway mean square error (see Figure 29.7). This behavior is largely due to the **trainb** learning rule that is used by default. For more challenging problems like the one here, you can change that to **trainr** (incremental learning):

```
>> net.trainFcn = 'trainr'
```

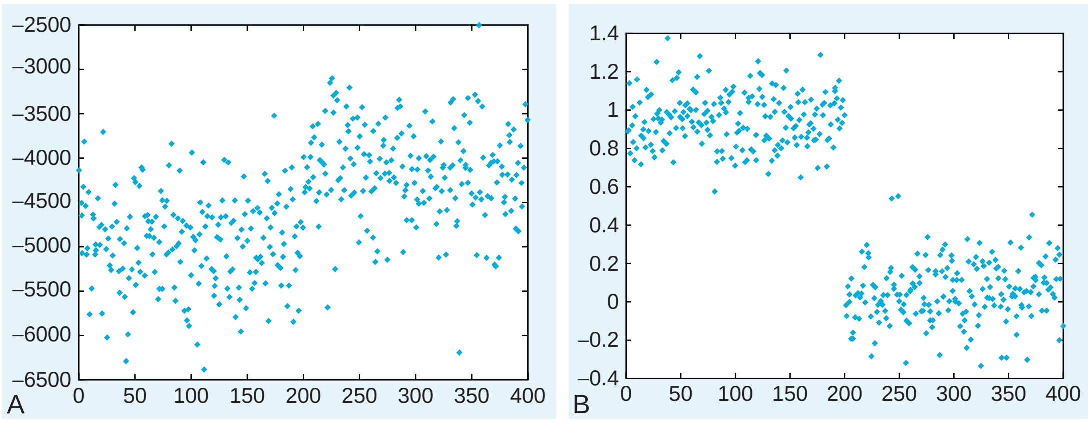


FIGURE 29.6 The problem of categorizing "good" from "bad" Greebles using a supervised linear network. A. Classification when all the weights are set to 1 (Before training). B. Classification after application of the supervised learning rule (After training).

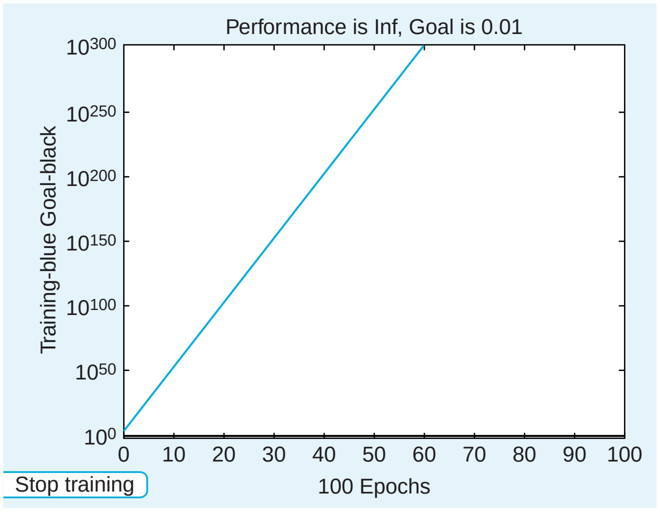


FIGURE 29.7 The potential problem of runaway mean square error when using batch learning.

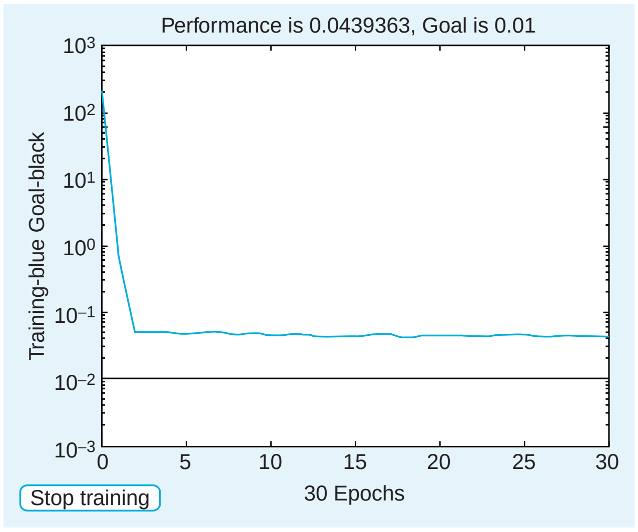


FIGURE 29.8 The use of incremental learning can sometimes solve the problem of runaway mean square error.

This command changes the actual training function of the network (named **net** here) to incremental training. The effect of this incremental learning on the mean square error is shown in Figure 29.8.

Problem solved.

29.3.3. Backpropagation

Most of the principles of backpropagation are the same as in the other networks, but you now have to specify the number of layers and transfer functions between the layers. For example, create a two-layer, feedforward network (one layer feeds into the next). This is achieved by using the function **newff**:

```
>> net =  newff([0 1;  0 1], [4,2], {'tansig','purelin'}, 'traingd');
```

This command specifies a network with two layers and two inputs. The arguments, in order, are as follows:

1. Input range (both inputs range from 0 to 1).
2. Number of neurons in each layer. Here, 4 is in the first, 2 in the second.
3. Transfer functions between inputs and outputs in the respective layer. First, you have a sigmoidal transfer function for the first layer (actually, Tan-sigmoid, abbreviated **tansig**, which creates outputs from –1 to 1; alternatively, **logsig** creates outputs from 0 to 1) and linear transfer function (**purelin**) for the second layer.
4. The training function, **traingd**. Backpropagation shifts the weights along a negative gradient. You are descending along the steepest gradient, hence **gd**.

Exercise 29.9: Create another network with different transfer functions, neurons, and inputs.

Exercise 29.10: Create another network with a larger number of layers—say three layers.

You should initialize your input weights by typing:

```
>> net = init(net)
```

This command will randomize the input weights. As in the other networks, you can simulate input-output mapping with these initial weights with the function **sim (net, inputs)**. Now you can train your network to do whatever you want. All you need is input matrices with appropriate inputs and outputs—for example, **[net, perf] = train (net, inputs, targets);**.

Note: Make sure to set the right training goals, e.g., a certain MSE or a certain number of epochs. Also, **traindg** is batch training, analogous to **trainb**. If things don't work as expected, use **net.trainFcn** and set it to **traingdm** (analogous to **trainr**).

Exercise 29.11: Solve the Greeble problem with a multilayer feedforward network, using the backpropagation learning rule. What are the results?

Suggestion for Exploration: Try this exercise using different transfer functions between the layers. See what can be done. Or wait for the final project to do this.

29.3.4. Sound Manipulation in MATLAB

As you saw in earlier chapters, MATLAB is useful not just for analyzing data, but also for experimental control and data gathering. Here, you will see that you can use MATLAB to design the stimulus material itself. Before you start, check the volume control to make sure that the loudspeaker of your PC is not muted and that the volume is turned up. There are many MATLAB functions dealing with auditory information; as a matter of fact, a whole toolbox is devoted to it. Here, we will handle sound only in very fundamental ways.

The first thing you might want to do is create your sound stimuli. To do so, try this:

```
>> x = 0:0.1:100;
>> y = sin(x);
>> sound(y);
```

Did you hear anything? What about this:

```
>> y = sin(2 .* x);
>> sound(y);
>> y = sin(4 .* x);
>> sound(y);
```

You know that your code created sine functions of increasing frequency. What you are listening to is the auditory representation of these sine functions, pure sine waves. Of course, most acoustic signals are not that pure. Try this example to listen to the sound of randomness—white noise:

```
>> x = randn(1,10001);
>> sound(x)
```

The **sound** function interprets the entries in an array (here the array **x**) as amplitude values and plays them as sound via your speakers. That means you can manipulate psychological qualities of the sound by performing operations in MATLAB. You already saw how to manipulate pitch (by manipulating the frequency). You can manipulate loudness by changing the magnitude of the values in the array. The **sound** function expects values in the range 1 to –1. So how does this example sound?

```
>> x = x ./ 5;
>> sound(x)
```

This should sound much less violent.

Of course, you can manipulate the sound in any way you want; for example, you can mix two signals:

```
>> z = x + y;
>> sound(z)
```

This example should sound like the sine wave from before, plus noise.

Exercise 29.12: What happens if you add two different frequencies of sine waves and play it?

In short, the sound you hear should be more complex, and rightfully so. Any arbitrarily complex sound pattern (or any signal, really) can be constructed by appropriately adding sine waves. You will use this property later. Speaking of complex sounds, most practical applications will require you to deal with sounds that are much more complex than pure sine waves. So let's look at one. Luckily, MATLAB comes with a complex sound bite:

```
>> load handel
>> sound(y, Fs);
```

You might be confused by the second parameter, *Fs*. It is the sampling rate at which the signal was sampled. It specifies how many amplitude values are played per second. If your array has 10,000 elements and the sampling rate is 10,000, it takes 1 second to play it as sound.

Exercise 29.13: Play your new sound again at a higher/lower sampling rate. What is happening?

Now look at the structure of the amplitude values in the *y* matrix to help you understand how **sound** works:

```
>> figure
>> plot(y)
```

The result should look something like Figure 29.9.

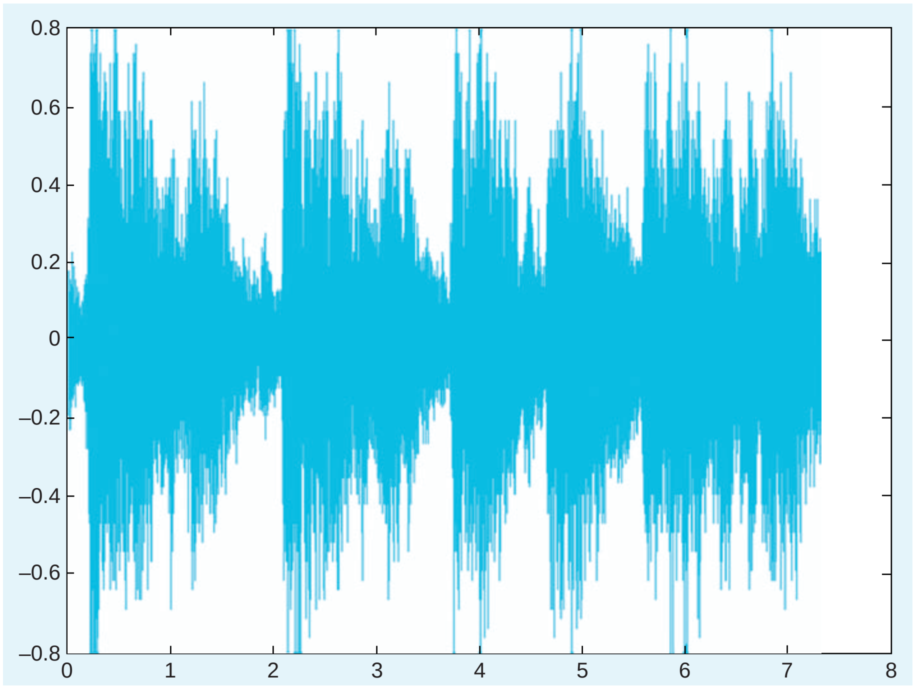


FIGURE 29.9 The raw acoustic pressure amplitude of the handel sound bite.

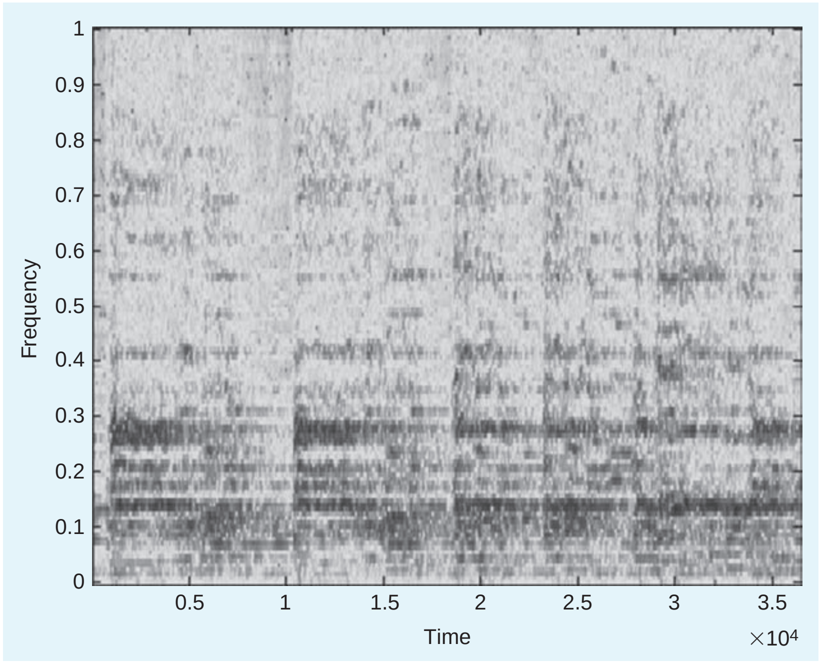


FIGURE 29.10 The spectrogram representation of the handel sound bite.

If you already listened to the sound bite, this result will probably not surprise you. As a matter of fact, this information about sound amplitudes alone is not very powerful in itself. It is much better and more useful to look at the spectral power of a signal over time. To do this, use a function called **spectrogram**. It comes with Signal Processing Toolbox in MATLAB.

The operation of the **spectrogram** function is rather complex and beyond the scope of this chapter. In principle, say that it decomposes the signal into sine waves and plots the power (how much of that frequency if it were composed out of sine waves is in the signal) over time:

```
>> spectrogram (y)
```

A spectrogram of the handel sounds looks like Figure 29.10.

Of course, you probably will want to import your own sounds into MATLAB. You can do this by using the **wavread** function. Download the file **speech_sample.wav** from the companion website and then type:

```
>> y = wavread('speech_sample.wav');
```

Exercise 29.14: What is the person in the sample file saying? *Hint:* The signal was sampled at 22050 Hz. You might want to take that fact into account when playing it. Look at the spectrogram, too.

Suggestion for Exploration: Listen to your data, literally. Listen to some of the stuff you did in previous chapters. How does it sound?

In this section, you saw how MATLAB lets you create, manipulate, and analyze acoustic stimuli. You can use it to design sound stimuli that are precisely timed and have very specific properties. Obviously, this is extremely useful for acoustic experiments.

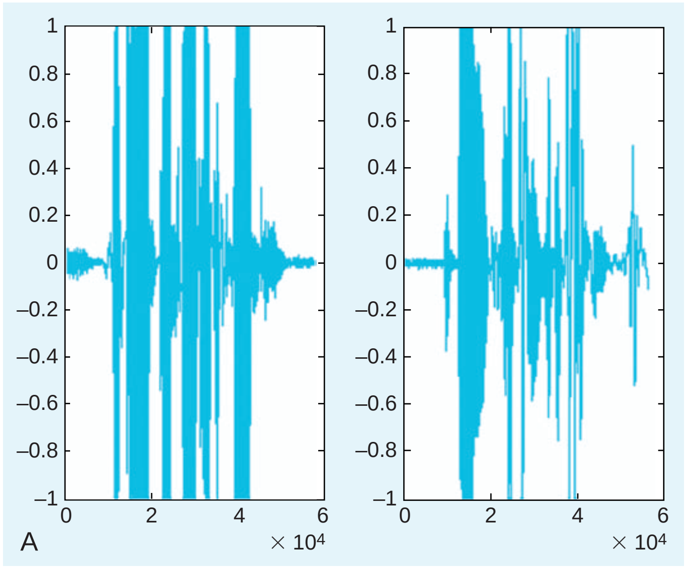
29.4. PROJECT

In this project, you will create a network that correctly classifies the gender of two target speech bites. To do this, train the network with a total of six speech bites of both genders. Load these files (called Female_Training1 to 3 and Male_Training1 to 3) from the companion website. Of course, the two speakers differ in all kinds of ways other than just gender (age, race, English as a first/second language, idiosyncratic speech characteristics, life style, etc.). If you were to face this problem in real life, you would have to train the network with a large number of speakers from both genders so that the network can extract gender features from all these irrelevant dimensions. But for purposes of this chapter, this approach will be fine.

Specifically, you should create a network that reliably distinguishes the gender of the two target speech bites. Provide some evidence that this is the case and submit the source code.

Hints:

- You might have to implement a network using the backpropagation learning rule.
- Take the spectrogram of the sound files. Use those as the inputs to your neural network. Figure 29.11 shows the amplitude and spectrogram of this sentence: "The dog jumped over the fence." The left subplot is a female speaker, and the right subplot is a male speaker.
- Using the command **a = spectrogram(b)** will return an array of complex numbers in *a*. They have an imaginary and a real part. For purposes of this chapter, the real part will do. For this, type **c = real(a)**. In this case, *c* will now contain the real part of the imaginary numbers in *a*.
- This project is deliberately underconstrained. Basically, you can try whatever you want to solve the problem. As a matter of fact, we encourage this, since it will help you understand neural networks that much better. Don't be frustrated and don't panic if you can't figure out the solution right away.
- Neural networks expect inputs. These inputs come in the form of rows. Match the number of inputs to a neural network (say two) to the number of rows in the input matrix that you feed it (has to have two rows). The repetitions of the input go in the columns (say 10 columns).
- If you take a spectrogram of the speech patterns with which you are supposed to train the network, it will return 129 rows and several thousand columns. A neural network that takes the entire information from this matrix has to have 129 inputs.
- This result is obviously rather excessive. If you closely observe the spectrogram (see Figure 29.11B), you might be able to get away with fewer rows and columns. The rows tessellate the frequency spectrum (y-axis below). Power in a particular frequency band is



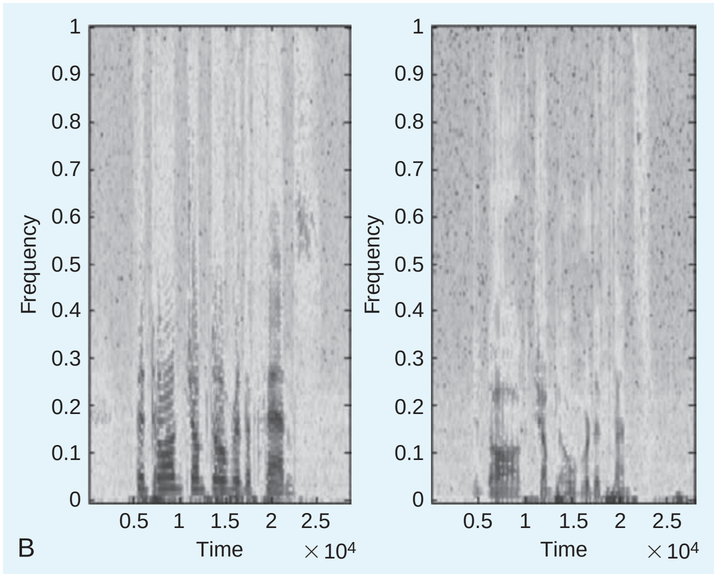


FIGURE 29.11 The amplitudes and spectrograms of a female and male speaker uttering "The dog jumped over the fence".

at a particular row. Power in a particular frequency band at a particular time is in a particular combination of row and column.

- The point is that the left spectrogram has much more power in the upper frequencies than the one on the right. If you properly combine frequency bands (or sample them), you might be able to get away with a neural network that has only 5 or 10 inputs.
- The same applies for time. Your spectrogram will have several thousand columns. This kind of resolution is not necessary to get the job done. Try combining (averaging) the values in 100 or so columns into one. Then feed that to the neural network for training.
- It is highly recommended to use a backprop network.
- Try using at least three layers.
- Try using nonlinear transfer functions between the layers.
- Try having a large number of hidden units (definitely more than 1).
- You will probably be using a supervised learning rule. The target is defined by which file the data came from (Pascal or Kira). Create an artificial index, assigning 1 and 0 (natural) or 1 and 2 (politically correct) to each.
- Always remember what the rows and columns of the variables you are using represent. Be aware of transformations in dimensions.

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

cat
demolin1
init
learnp
logsig
newff
newlin
newp
nnd4db
purelin
real
sim
sound
spectrogram
tansig
train
trainb
traingd
traingdm
trainr
wavread

APPENDIX

A

Thinking in MATLAB

A.1. ALTERNATIVES TO MATLAB

For a multitude of reasons, this book uses the MATLAB software package as its computational language. Primary among these reasons is the ubiquity of the software in the field of neuroscience. Even those who use an alternative are likely to have some familiarity with MATLAB and many neuroscience-specific packages are only available for MATLAB.

That said, many alternative computation suites exist. No package can meet every need, and you may decide that an alternative suits your needs better than MATLAB. In this appendix, alternatives to MATLAB are briefly discussed. Hopefully, this discussion will provide some understanding why these alternatives are chosen.

A.1.1. Octave

Of all alternatives available for the MATLAB® software, Octave is probably the one most similar to MATLAB. Octave is an open source numerical package designed to be highly syntactically compatible with MATLAB. There are minor syntactical differences between Octave and MATLAB, but the most significant differences involve portions of MATLAB outside the language core. For example, equivalent code for most MATLAB toolboxes does not exist for Octave. Also, Octave lacks a profiler and compiler. An example follows:

```
function x = eulerdfn, dt, steps, x0)
      t = 0:dt:(steps* dt);
      x = zeros(1, steps);
      x(1) = x0;
      for index = 2:steps
            x(index) = feval(dfn, x(index-1)) * dt + x(index-1);
      end
end
```

A huge advantage of Octave over MATLAB is cost. As open source software, Octave is entirely free. Moreover, the source code for Octave is freely available, allowing examination of the algorithms behind the software. However, Octave is not MATLAB, and it is important to note that the same level of support, commercial and otherwise, is not present for Octave.

A.1.2. Python

Python is a popular programming language for which substantial quantitative tools exist. Syntax differs quite substantially from MATLAB. As a general-purpose programming language, Python does not have inherent visualization and quantitative capabilities. The most actively developed quantitative package for Python is **NumPy**, which provides fast array-handling primitives:

```
import numpy;

def euler(dfn, dt, steps, x0):
    x = array()
    for index in xrange(steps):
        x[index] = dfn(x[index-1]) * dt + x[index -1]
    return x
```

One of the most popular 2D visualization packages for Python is matplotlib. For users familiar with MATLAB, matplotlib includes an interactive interface that syntactically resembles the plotting commands of MATLAB, as shown in this example:

```
import pylib

t = arange(0, 10, 0.01)
plot(t, euler(lambda x: x* 0.1, dt=0.01, steps=1000, 1))
```

As a general-purpose programming language, Python excels in solving problems often ill-suited to MATLAB. MATLAB has poor support for most varieties of nonquantitative data, especially character strings or highly structured data. Moreover, many problems lend themselves to programmatic abstractions that are often more difficult to encode in MATLAB.

Like other open source software, Python (and associated packages) is freely available. Together, NumPy and matplotlib are included in SciPy, a comprehensive scientific computing package for Python. With the choice of adjunct packages comes some installation and configuration effort on the part of the user beyond what is required for MATLAB.

A.1.3. Mathematica

Mathematica is commercial software published by Wolfram Research. While Mathematica does have extensive support for quantitative processing, it also has extensive support for symbolic processing.

A.1.4. C

The primary advantages of C or C++ are speed and flexibility. The interactive exploration often inherent in data analysis in MATLAB has no counterpart in the C compile-run-debug cycle. There is no standard graphical library for C. Additionally, the standard C and C++ libraries do not include quantitative functions equivalent to those in MATLAB. However, where access to the underlying operating system is necessary or speed is extraordinarily important, C is often the best tool.

For programmers coming from C, it is important to note that multidimensional arrays in MATLAB are stored in column-major order, belying the original FORTRAN parentage of MATLAB. Column-major order indicates that members of the same column are stored sequentially in memory. Most programming languages use row-major order, in which elements sharing a row are stored sequentially in memory. This becomes important when using a single index to dereference a multidimensional array. The following snippet of MATLAB code demonstrates this:

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

A =

  1  2  3
  4  5  6
  7  8  9

>> A(3)

ans =

  7
```

The alternative code in C evaluates to 3:

```
#include <stdio.h>

int main() {
  int A[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
  int *B = (int*)A;
  printf("%d\n", B[2]);
}
```

A.1.5. FORTRAN

Like C, FORTRAN is a general-purpose programming language, without any default graphics package. The original goal of FORTRAN was the simplification of mathematical programming, and the language standard has always focused on a fairly simple core suitable for quantitative expressions.

The primary advantage of FORTRAN over MATLAB is speed. Not only is FORTRAN compiled, but the simplicity of the language allows for compiler optimizations not usually possible with many other compiled languages, such as C. Even though FORTRAN is fairly straightforward for a general-purpose programming language, it requires more effort to learn than MATLAB and has no provision for interactivity.

A.2. A FEW WORDS ABOUT PRECISION

Like most quantitative software, MATLAB does not represent most floating-point numbers exactly. When looking at the workspace, you may have noticed that nearly all variables have the label **double**. This label refers to the internal representation of the floating-point number. This representation is the default representation of values in MATLAB.

The **double** representation corresponds to a standard floating-point representation used throughout most quantitative software (see Figure A.1). (**Double** here is in deference to single-precision floating-point, a lesser used floating-point representation that consumes half the memory.) Since the Institute of Electrical and Electronics Engineers (IEEE) standards body oversees the specification of this format, it is commonly known as IEEE 754 or 64-bit IEEE floating-point.

The sign bit denotes whether the number as a whole is positive or negative. The exponent is a base 2 number biased by $2^{10} - 1$, or 1023. The representation of exponents is 1023 plus the exponent's value. This system allows for exponents in the range –1023 to 1023.

The representation of the mantissa is the most complex portion of this standard. The digits in the mantissa represent a binary fraction, where each successive digit represents a successive fractional power of 2. Additionally, the mantissa is the fractional part of the number; there is a 1 implicit in the number not represented in the format.

The following example illustrates how a decimal floating-point number is represented internally by MATLAB.

Consider 15.1875.

In base 2, 15 is 1111_2. As a binary fraction, 0.1875 is 0.0011_2. (0.1875 is 3/16, or $1/8 + 1/16$, or $0 * 1/2 + 0 * 1/4 + 1 * 1/8 + 1 * 1/16$.) In total, 15.1875 is 1111.0011_2. To use exponential notation, this is $1.1110011_2 \times 2^3$. To store this as in double-precision format, you need to discard the initial 1 from the mantissa and bias the exponent, as shown in Figure A.2.

Why is this important? Double precision floating point and, by extension, MATLAB, can represent only a small subset of floating-point numbers with absolute precision. These numbers are those whose fractional part is a sum of fractional powers of 2. For example, 7/16 can be perfectly represented ($1/4 + 1/8 + 1/16$), but 7/17 cannot.

Precision issues are particularly significant when testing for 0. When you are dealing with floating-point numbers, operations that you might expect to produce 0 often result in very small fractional numbers. A more viable test is a range around 0. For example, instead of

```
if x == 0
```

use

```
if abs(x) < 1.0e-6
```

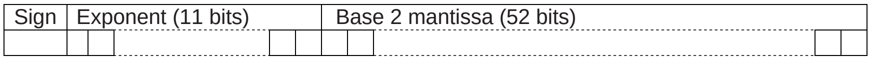


FIGURE A.1 Representation of floating-point numbers.

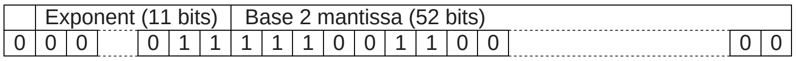


FIGURE A.2 Representation of the number 15.1875.

A.2.1. Taking Advantage of Matrix Operations

When you are writing code that processes large datasets, efficiency begins to become an important consideration. Serious optimization of an algorithm may require significant thought and effort. However, taking advantage of matrix operations whenever possible can often speed up code immensely without retooling algorithms. For example, consider the following two code snippets. Both add two matrices together:

```
A = ones(4, 4); * 3; % matrix of threes
B = ones(4, 4); * 6; % matrix of sixes
C = zeros(4, 4);
for i = 1:4
        for j = 1:4
                C(i, j) = A(i, j) + B(i, j);
        end
end
```

or

```
A = ones(4, 4); * 3; % matrix of threes
B = ones(4, 4); * 6; % matrix of sixes
C = zeros(4, 4);
C = A + B;
```

While both pieces of code accomplish the same task, the second executes measurably faster. Note that the second snippet avoids the nested **for** loops.

Understanding why these two bits of code execute so differently requires a brief explanation of how MATLAB evaluates code. Individual operations in MATLAB execute as compiled machine code, at high speed. For example, the matrix addition in the second code section executes in this manner. Each invocation, however, requires processing within MATLAB's interpreter before the operation's underlying code can execute.

So, in the case of the first example, evaluation of the inner statement alone requires evaluating each of the two index variables, three matrix lookups, and a scalar addition, and then storing the scalar result. In between operations, the interpreter must be constantly consulted to determine the next step. In the second case, all of these sub operations run within the internal MATLAB code for a matrix add, avoiding the overhead of the interpreter between each step.

A.2.2. Conditional Expressions

Relational operations can often function as an alternative to an **if** statement nested within a **for** loop. A relational operator acting on a matrix returns a matrix of the same shape with values of 1 for **true** and 0 for **false**:

```
A = ones(4, 4);
B = rand(4, 4);
for i = 1:4
        for j = 1:4
```

```
            if (B(i, j) > 0.5)
                A(i, j) = A(i, j) + B(i, j);
            end
        end
end
```

Compare the preceding example with the following:

```
A = ones(4, 4);
B = rand(4, 4);
A = A + (B .* (B > 0.5));
```

In the latter example, the single expression takes the place of the nested **for** loops and **if** statement. The inner relational expression evaluates to a 4×4 matrix whose elements are 1 if the corresponding element of *B* is greater than 0.5. Thus, the element-wise multiplication of this matrix with *B* generates a matrix whose elements are either the corresponding element of *B*, if *B* is greater than 0.5, or 0, if that element of *B* is less than or equal to 0.5.

A.2.3. Extracting Subsets from Arrays

Many times, an **if** statement nested within a **for** loop is used to extract some subset of values from a matrix. The use of matrix relational operations and **find** can eliminate the need for the iteration. The function **find** returns all the indices of the input for which the input is nonzero. For example:

```
>> A = [1 2 3 4];
>> find(A < 3)

ans =

  1   2
```

Specifying a set of values for the index of a matrix will return a subset of the matrix values. This can apply to the results of **find:**

```
>> A = [8 9 10 11];
>> find(mod(A,2) == 0)

ans =

  1   3

>> A(find(mod(A,2) == 0))

ans =

  8   10
```

APPENDIX

B

Linear Algebra Review

Informally, the terms *matrix* and *array* are often used interchangeably. More precisely, a matrix is a two-dimensional numeric array that represents a linear transformation. The mathematical operations defined on matrices are the subject of linear algebra. This appendix will describe some of the basic concepts of linear algebra that you'll need to get started.

B.1. MATRIX DIMENSIONS

An $m \times n$ matrix has m rows and n columns. Here is an example of a 2×3 matrix, A:

$$A = \begin{pmatrix} 2 & 4 & 8 \\ 1 & 7 & 3 \end{pmatrix}.$$

You can refer to a particular element in the matrix by its row and column placement. So, for the matrix A, the element in the first row and third column is the number 8. The syntax to identify this element in the MATLAB® software is **A(1,3)**.

B.2. MULTIPLICATION

When a matrix is multiplied by a number, *every* element of the matrix is multiplied by the number. Here is an example of a matrix, F, that is equal to 5 times the matrix A:

$$F = 5 * A = 5 * \begin{pmatrix} 2 & 4 & 8 \\ 1 & 7 & 3 \end{pmatrix} = \begin{pmatrix} 10 & 20 & 40 \\ 5 & 35 & 15 \end{pmatrix}$$

When you multiply two matrices together, $A * B$, each of the elements of the resulting matrix, C, is the sum of the corresponding row elements of A times the corresponding column elements of B. In other words, all elements of C may be obtained by using the following simple rule:

The element in row i and column j of the product matrix $A * B$ is equal to row i of A times column j of B.

Note that if A is $m \times n$, and B is $n \times l$ (the number of columns in A and the number of rows in B MUST match; otherwise, the product is undefined), then the dimensions of C are $m \times l$. For example:

$$A * B = \begin{pmatrix} 2 & 4 & 8 \\ 1 & 7 & 3 \end{pmatrix} \begin{pmatrix} 5 \\ 6 \\ 3 \end{pmatrix} = \begin{pmatrix} 2*5+4*6+8*3 \\ 1*5+7*6+3*3 \end{pmatrix} = \begin{pmatrix} 58 \\ 56 \end{pmatrix} = C$$

Observe that A * B is *not* the same as B * A. In this case, matrix multiplication is not *commutative*, or in general, matrices do not *commute* under multiplication. We will return to matrix multiplication and present its geometric interpretation in Section B.5.

B.3. ADDITION

When you add matrices algebraically, you add them component by component, adding corresponding elements. For example,

$$A + F = \begin{pmatrix} 2 & 4 & 8 \\ 1 & 7 & 3 \end{pmatrix} + \begin{pmatrix} 10 & 20 & 40 \\ 5 & 35 & 15 \end{pmatrix} = \begin{pmatrix} 12 & 24 & 48 \\ 6 & 42 & 18 \end{pmatrix}$$

Note that if you add $A + A$, you would get $2A$ in accord with the definition of scalar-matrix multiplication presented in the preceding section. For the sum to exist, the dimensions of the matrices being added must be exactly the same. So, for example, you cannot add together the matrices A and B as defined previously; instead, you say that the sum is *undefined* or *meaningless*.

B.4. TRANSPOSE

To transpose a matrix, you simply write the rows as columns; that is, you interchange rows and columns. So

$$A^T = \begin{pmatrix} 2 & 1 \\ 4 & 7 \\ 8 & 3 \end{pmatrix}.$$

You carry out this operation in MATLAB by entering the command **A'**.

B.5. GEOMETRICAL INTERPRETATION OF MATRIX MULTIPLICATION

There is also a geometrical interpretation of matrix-vector multiplication that can be extremely useful. First, see what happens when a vector is multiplied by a scalar. Suppose that

$$B = \begin{pmatrix} 3 \\ 4 \end{pmatrix}.$$

You can plot the vector B on the Cartesian plane if you assume that the x-component of the vector is the element in the first row and the y-component of the vector is the element in the second row. Therefore, the vector B written as:

$$\begin{pmatrix} 3 \\ 4 \end{pmatrix}$$

is equivalent to the vector $B = (3\hat{x} + 4\hat{y})$; you may be more familiar with this notation from physics. This results in the graph shown in Figure B.1.

Next, you multiply the vector B by a scalar, 2, to get:

$$2 * B = \begin{pmatrix} 6 \\ 8 \end{pmatrix}$$

If you plot this new vector alongside B, then you get the graph shown in Figure B.2.

Notice that multiplying a vector by a scalar changes only its length. It does not change the direction of the vector. Now see what happens when a vector is multiplied by a matrix. Let:

$$A = \begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix},$$

so that

$$A * B = \begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix} * \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 7 \\ 16 \end{bmatrix}.$$

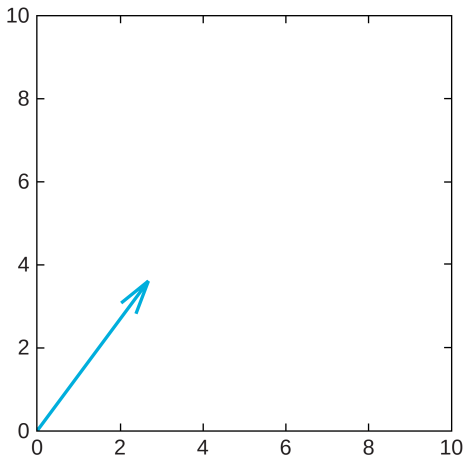


FIGURE B.1 The vector B graphed.

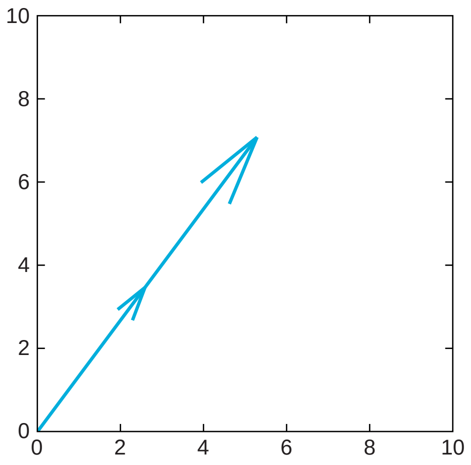


FIGURE B.2 The vector B multiplied by a scalar (in this case 2).

Since the matrix A is square, the product of A and B has the same dimensions as the vector B (in this case both are 2×1). Therefore, you can plot the vectors $A * B$ and B on the same graph to obtain the result shown in Figure B.3.

Here, you can see that multiplication of vector B by the matrix A has resulted in rotating B counterclockwise and stretching it out. Now try another example, where A is the same, but:

$$B = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

so

$$A * B = \begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \end{bmatrix} = 3 * \begin{bmatrix} 1 \\ 2 \end{bmatrix} = 3B.$$

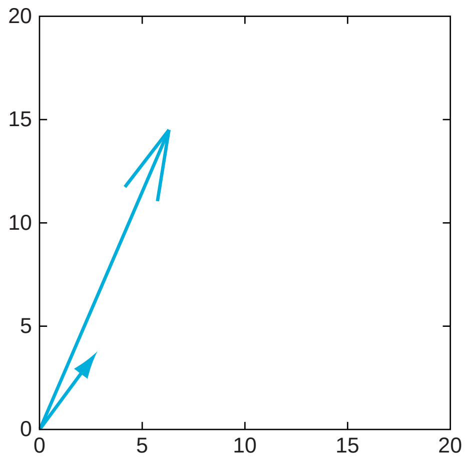


FIGURE B.3 The vector B as well as the vector $A * B$.

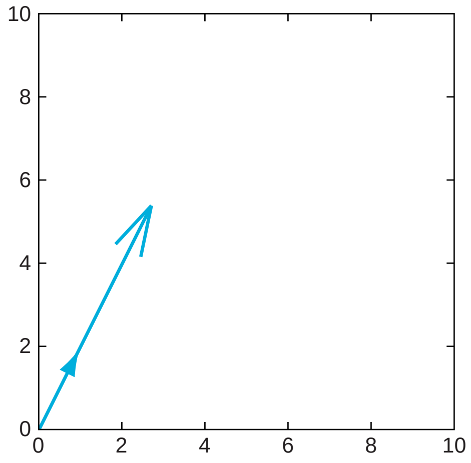


FIGURE B.4 The vector A * B (long) with eigenvector B (short).

If you plot B and A * B together, then you get the result shown in Figure B.4.

In this case, multiplication of the vector B by the matrix A is equivalent to multiplication of B by a scalar, in this case 3. It is possible, then, that for a square matrix A, there may exist a vector B and a lambda such that:

$$A * B = \lambda B$$

for some scalar λ. Geometrically, this means that for a given matrix A, there is a vector B that does not rotate when multiplied by A. The scalar λ is called an *eigenvalue* of the matrix A, and B is called an *eigenvector* of the matrix A corresponding to the eigenvalue λ. Before we have the necessary tools to determine the eigenvalues and eigenvectors of a matrix systematically, we must first discuss the determinant and inverse of a square matrix.

B.6. DETERMINANT

The determinant exists only for square matrices. For a 2×2 matrix:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix},$$

the determinant of A is written as follows: $\det(A) = ad - bc$.

For larger square matrices, the determinant is still defined and can be calculated using the command **det(A)** in MATLAB. From now on, however, we will focus on 2×2 square matrices, so the preceding definition of the determinant will suffice.

B.7. INVERSE

The inverse of a matrix D, D^{-1}, is the matrix that, when multiplied with the original matrix, equals the identity matrix. Note that to take the inverse of a matrix, the matrix must be square ($m \times m$). If:

$$D = \begin{pmatrix} 2 & 3 \\ 5 & 7 \end{pmatrix},$$

then its inverse is:

$$D^{-1} = \begin{pmatrix} -7 & 3 \\ 5 & -2 \end{pmatrix}.$$

So you have:

$$D * D^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

The function in MATLAB for the identity matrix is **eye(*n*)**, where the value ***n*** determines the dimension of the identity matrix. MATLAB will solve for the inverse of a matrix, if it exists, with the function **inv(*A*)**, where A is the matrix to be inverted.

For a 2×2 square matrix:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix},$$

the inverse is given by:

$$A^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix},$$

if **det(A)** $\neq$ **0**.

Note that:

$$\begin{aligned} A * A^{-1} &= \begin{bmatrix} a & b \\ c & d \end{bmatrix} * \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \\ &= \frac{1}{ad - bc} \begin{bmatrix} a & b \\ c & d \end{bmatrix} * \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \\ &= \frac{1}{ad - bc} \begin{bmatrix} ad - bc & 0 \\ 0 & ad - bc \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I. \end{aligned}$$

Note also from this example, then, if **det(*A*)** $=$ **0**, then A^{-1} is undefined, since the inverse is proportional to:

$$\frac{1}{\det(A)}.$$

In general, a square matrix A has an inverse if and only if $\mathbf{det(A)} \neq \mathbf{0}$.

B.8. EIGENVALUES AND EIGENVECTORS

Recall that finding the eigenvalues and corresponding eigenvectors of a square matrix A is equivalent to solving for λ and B such that:

$$A * B = \lambda B \tag{B.1}$$

Note that:

$$B = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

the zero-vector, is a solution of Equation B.1 regardless of the matrix A, as long as A is 2×2. This solution is called the *trivial-solution*, and will not be of interest here, so you will also require that B is not the zero-vector when solving Equation B.1. Equation B.1 is equivalent to:

$$A * B = \lambda I * B, \tag{B.2}$$

where I is the identity matrix. Moving the right side to the left gives:

$$A * B - \lambda I * B = (A - \lambda I) * B = 0 \tag{B.3}$$

If the matrix $(A - \lambda I)$ has an inverse, then multiplying through Equation B.3 by the inverse gives:

$$(A - \lambda I)^{-1}(A - \lambda I) * B = (A - \lambda I)^{-1} * 0 = 0 \tag{B.4}$$

but a matrix multiplied by its inverse equals the identity matrix so:

$$I * B = B = 0. \tag{B.5}$$

This is exactly the solution that you do NOT want, which means that the only way for $B \neq 0$ is if $(A - \lambda I)$ does not have an inverse. Recall that the matrix $(A - \lambda I)$ does not have an inverse if and only if:

$$\det(A - \lambda I) = 0. \tag{B.6}$$

Equation B.6 is called the *characteristic equation* of the matrix A. It is the only equation you need to calculate the eigenvalues and eigenvectors of a matrix. Now, use it in an example to solve for the eigenvalues and eigenvectors of the matrix

$$A = \begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix}$$

shown in Chapter 11, "Introduction to Phase Plane Analysis." Plugging A into Equation B.6 gives:

$$\det\left(\begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) = 0 \Rightarrow$$

$$\det\left(\begin{bmatrix} 1-\lambda & 1 \\ 4 & 1-\lambda \end{bmatrix}\right) = (1-\lambda)^2 - 4 = \lambda^2 - 2\lambda - 3 = 0.$$

You can solve the quadratic equation for λ to get $\lambda = \{-1, 3\}$. These are the eigenvalues of the matrix A. You can solve for the corresponding eigenvectors as follows:
For $\lambda = 3$, Equation B.1 becomes:

$$A * B = 3B.$$

Substitute A into the preceding equation and let:

$$B = \begin{bmatrix} x \\ y \end{bmatrix}.$$

The preceding equation becomes:

$$\begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = 3\begin{bmatrix} x \\ y \end{bmatrix} \Rightarrow \begin{bmatrix} x & +y \\ 4x & +y \end{bmatrix} = \begin{bmatrix} 3x \\ 3y \end{bmatrix}$$

Solving the system of equations gives $y = 2x$. If you choose $x = 1$, then $y = 2$, and

$$B = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

is a corresponding eigenvector to $\lambda = 3$. Notice how this agrees with what was discovered earlier. Feel free to prove that the eigenvector for $\lambda = -1$ is:

$$B = \begin{bmatrix} 1 \\ -2 \end{bmatrix}.$$

Knowing the eigenvalues and eigenvectors of a given matrix is useful for many reasons. For one thing, if λ is an eigenvalue of the matrix A and v is an associated eigenvector, then $x = e^{\lambda t}v$ is a solution to the system of differential equations described by $dx/dt = Ax$ and satisfies the initial condition $x(t = 0) = v$.

In MATLAB, the command **[*V*,*D*]** = **eig(*A*)** will return two matrices: D and V. The elements of the diagonal matrix D are the eigenvalues of the square matrix A. The columns of the matrix V are the corresponding eigenvectors. Therefore, $A * V = V * D$.

B.9. EIGENDECOMPOSITION OF A MATRIX

Next, we will describe a powerful theorem called the *Eigendecomposition Theorem*. This theorem follows:

For an n×n *matrix* A *with distinct eigenvalues you can write:*

$$A = V * D * V^{-1},$$

where V *is the square matrix whose columns are the eigenvectors of* A *and* D *is the square diagonal matrix formed by placing the eigenvalues of* A *along the primary diagonal of* D *and letting all other elements of* D *equal 0.*

You can apply this theorem to the matrix

$$A = \begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix}.$$

First, you form the matrix V from the eigenvectors of A:

$$V = \begin{bmatrix} 1 & 1 \\ 2 & -2 \end{bmatrix}.$$

Using Equation B.4 gives you the inverse of V:

$$V^{-1} = \frac{1}{-4}\begin{bmatrix} -2 & -1 \\ -2 & 1 \end{bmatrix}.$$

Next, you define D. Note that it is important to put the eigenvalues in the right order. Since the first column of V is the eigenvector corresponding to $\lambda = 3$, this is the eigenvalue that must appear in the first row of D. As long as you are consistent, it does not matter where the eigenvalues or eigenvectors are placed within their respective matrices. Placing the eigenvalues gives:

$$D = \begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix}.$$

Finally, you can test the theorem by calculating $V * D * V^{-1}$:

$$\begin{aligned} VDV^{-1} &= \begin{bmatrix} 1 & 1 \\ 2 & -2 \end{bmatrix}\begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix}\left(\frac{1}{-4}\right) * \begin{bmatrix} -2 & -1 \\ -2 & 1 \end{bmatrix} \\ &= \left(\frac{1}{-4}\right) * \left(\begin{bmatrix} 1 & 1 \\ 2 & -2 \end{bmatrix} * \begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix}\right) * \begin{bmatrix} -2 & -1 \\ -2 & 1 \end{bmatrix} \\ &= \frac{1}{-4} * \begin{bmatrix} 3 & -1 \\ 6 & 2 \end{bmatrix} * \begin{bmatrix} -2 & -1 \\ -2 & 1 \end{bmatrix} = \frac{1}{-4} * \begin{bmatrix} -4 & -4 \\ -16 & -4 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix} = A. \end{aligned}$$

Note that you can rearrange the Eigendecomposition Theorem to arrive at the equation $D = V^{-1} * A * V$, which is sometimes also useful.

APPENDIX

C

Master Equation List

CHAPTER 6

$$l_{ij}(e) = \frac{p(e|s_i)}{p(e|s_j)} \tag{6.1}$$

$$l_{ij}(e) = \frac{stim\_frequency}{1 - stim\_frequency} * \frac{value\_of\_correct\_rejection - value\_of\_false\_alarm}{value\_of\_hit - value\_of\_miss} \tag{6.2}$$

CHAPTER 7

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(nt) + \sum_{n=1}^{\infty} b_n \sin(nt) \tag{7.1}$$

$$\int_{-\pi}^{\pi} f(t)\cos(mt)dt = \int_{-\pi}^{\pi} \frac{a_0}{2}\cos(mt)dt + \sum_{n=1}^{\infty}\int_{-\pi}^{\pi} \cos(mt)a_n\cos(nt)dt + \sum_{n=1}^{\infty}\int_{-\pi}^{\pi} \cos(mt)b_n\sin(nt)dt \tag{7.2}$$

$$\int_{-\pi}^{\pi} f(t)\cos(mt)dt = a_m \int_{-\pi}^{\pi} \cos^2(mt)dt \tag{7.3}$$

$$\int_{-\pi}^{\pi} f(t)\cos(mt)dt = \pi a_m \tag{7.4}$$

$$a_m = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\cos(mt)dt \tag{7.5}$$

$$a_m = \frac{1}{L}\int\limits_{x-L}^{x+L} f(t)\cos\left(\frac{\pi}{L}mt\right)dt \tag{7.6}$$

$$e^{i\omega t} = \cos\omega t + i\sin\omega t \tag{7.7}$$

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{\text{int}} \tag{7.8}$$

$$c_m = \frac{1}{2\pi}\int\limits_{-\pi}^{\pi} f(t)e^{-imt}dt \tag{7.9}$$

$$c_m = \frac{1}{2L}\int\limits_{x-L}^{x+L} f(t)e^{-(i\pi mt/L)}dt \tag{7.10}$$

$$F_k = \frac{1}{N}\sum_{n=0}^{N-1} f_n e^{-i\frac{2\pi n}{N}k} \tag{7.11}$$

$$f_n = \sum_{k=0}^{N-1} F_k e^{i\frac{2\pi n}{N}k} \tag{7.12}$$

$$\Phi(\omega) = |F(\omega)|^2 = F(\omega)\mathrm{F}*(\omega) \tag{7.13}$$

CHAPTER 8

$$X(\tau,\omega) = \int_{-\infty}^{\infty} x(t)w(t-\tau)e^{-j\omega t}dt \tag{8.1}$$

$$X(m,\omega) = \sum_{n=-\infty}^{\infty} x(n)w(n-m)e^{-j\omega n} \tag{8.2}$$

$$w(n) = 0.53836 - 0.46164\cos\left(\frac{2\pi n}{N-1}\right) \tag{8.3}$$

$$w(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{N-1}\right) \tag{8.4}$$

$$w(n) = e^{-n^2} \tag{8.5}$$

CHAPTER 9

$$\int\limits_{-\infty}^{\infty} \psi(x)dx = 0 \tag{9.1}$$

$$\int_{-\infty}^{\infty} \psi^2(x)dx = 1 \tag{9.2}$$

$$W(s,t) \equiv \int_{-\infty}^{\infty} x(u)\psi_{s,t}(u)du \tag{9.3}$$

$$\psi_{s,t}(x) \equiv \frac{1}{\sqrt{s}}\psi\left(\frac{x-t}{s}\right) \tag{9.4}$$

$$\psi(t) = \pi^{-\frac{1}{4}}e^{-\frac{1}{2}t^2}e^{-i\omega_0 t} \tag{9.5}$$

$$\psi(t) = \frac{1}{\sqrt{2\pi\sigma^3}}\left(1-\frac{t^2}{\sigma^2}\right)e^{\frac{-t^2}{2\sigma^2}} \tag{9.6}$$

CHAPTER 10

$$y(t) = h(t) * x(t) = \int_{-\infty}^{\infty} h(\tau)x(t-\tau)d\tau \tag{10.1}$$

$$y(k,t) = h(k,t) * x(k,t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} h(\tau,K)x(k-K,t-\tau)dKd\tau \tag{10.2}$$

$$y(n_1,n_2) = \sum_{k_1=-\infty}^{\infty}\sum_{k_2=-\infty}^{\infty} h(k_1,k_2)x(n_1-k_1,n_2-k_2) \tag{10.3}$$

CHAPTER 11

$$\frac{d\tilde{C}}{dt} = \frac{1}{\tau_C}(-\tilde{C} - k\tilde{H}) \tag{11.1}$$

$$\frac{d\tilde{H}}{dt} = \frac{1}{\tau_H}(-\tilde{H} + \tilde{C}) \tag{11.2}$$

$$\tilde{C} = C - \frac{L}{k+1}, \text{ and } \tilde{H} = H - \frac{L}{k+1} \tag{11.3}$$

$$\frac{dx}{dt} = x + y \tag{11.4}$$

$$\frac{dy}{dt} = 4x + y \tag{11.5}$$

$$x(t) = C_1 e^{3t} + C_2 e^{-t} \tag{11.6}$$

$$y(t) = 2C_1 e^{3t} - 2C_2 e^{-t} \tag{11.7}$$

$$\lim_{t\to\infty} x(t) = \lim_{t\to\infty} y(t) = 0 \tag{11.8}$$

$$\lim_{t\to\infty} x(t) = \lim_{t\to\infty} y(t) = \infty \tag{11.9}$$

$$m = \frac{dy}{dt} \Big/ \frac{dx}{dt} = \frac{dy}{dx} \tag{11.10}$$

CHAPTER 12

$$\frac{dv}{dt} = c(v - \frac{1}{3}v^3 + r + I) \tag{12.1}$$

$$\frac{dr}{dt} = -\frac{1}{c}(v - a + br) \tag{12.2}$$

$$\frac{dx}{dt} = ax + by \tag{12.3}$$

$$\frac{dy}{dt} = cx + dy \tag{12.4}$$

$$\frac{dx}{dt} = f(x,y) \tag{12.5}$$

$$\frac{dy}{dt} = g(x,y) \tag{12.6}$$

$$f(x,y) = f(x_{ss}, y_{ss}) + \frac{\partial f(x_{ss}, y_{ss})}{\partial x}(x - x_{ss}) + \frac{\partial f(x_{ss}, y_{ss})}{\partial y}(y - y_{ss}) + \ldots \tag{12.7}$$

$$g(x,y) = g(x_{ss}, y_{ss}) + \frac{\partial g(x_{ss}, y_{ss})}{\partial x}(x - x_{ss}) + \frac{\partial g(x_{ss}, y_{ss})}{\partial y}(y - y_{ss}) + \ldots \tag{12.8}$$

$$f(x,y) \approx \frac{\partial f(x_{ss}, y_{ss})}{\partial x}(x - x_{ss}) + \frac{\partial f(x_{ss}, y_{ss})}{\partial y}(y - y_{ss}) \tag{12.9}$$

$$g(x,y) \approx \frac{\partial g(x_{ss}, y_{ss})}{\partial x}(x - x_{ss}) + \frac{\partial g(x_{ss}, y_{ss})}{\partial y}(y - y_{ss}) \tag{12.10}$$

$$\frac{dx}{dt} = \frac{d(x - x_{ss})}{dt} = \frac{\partial f(x_{ss}, y_{ss})}{\partial x}(x - x_{ss}) + \frac{\partial f(x_{ss}, y_{ss})}{\partial y}(y - y_{ss}) \tag{12.11}$$

$$\frac{dy}{dt} = \frac{d(y - y_{ss})}{dt} = \frac{\partial g(x_{ss}, y_{ss})}{\partial x}(x - x_{ss}) + \frac{\partial g(x_{ss}, y_{ss})}{\partial y}(y - y_{ss}) \tag{12.12}$$

$$\begin{bmatrix}(x-x_{ss})' \\ (y-y_{ss})'\end{bmatrix} = \begin{bmatrix}\dfrac{\partial f(x_{ss},y_{ss})}{\partial x} & \dfrac{\partial f(x_{ss},y_{ss})}{\partial y} \\ \dfrac{\partial g(x_{ss},y_{ss})}{\partial x} & \dfrac{\partial g(x_{ss},y_{ss})}{\partial x}\end{bmatrix} * \begin{bmatrix}(x-x_{ss}) \\ (y-y_{ss})\end{bmatrix} \tag{12.13}$$

$$u = \begin{bmatrix}(x-x_{ss}) \\ (y-y_{ss})\end{bmatrix} \text{ and } J = \begin{bmatrix}\dfrac{\partial f}{\partial x} & \dfrac{\partial f}{\partial y} \\ \dfrac{\partial g}{\partial x} & \dfrac{\partial g}{\partial y}\end{bmatrix} \tag{12.14}$$

$$u' = J|_{(x_{ss},\ y_{ss})} * u \tag{12.15}$$

CHAPTER 14

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{14.1}$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2 \tag{14.2}$$

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})^2 \tag{14.3}$$

$$s^2 = \frac{1}{n-1}(x-\bar{x})^T(x-\bar{x}) \tag{14.4}$$

$$\mathrm{cov}(x,y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y}) \tag{14.5}$$

$$\mathrm{cov}(x) = \frac{1}{n-1}(x-\bar{x})^T(x-\bar{x}) \tag{14.6}$$

CHAPTER 15

$$P(S) = \sum_R P(s,r) \text{ and } P(R) = \sum_S P(s,r) \tag{15.1}$$

$$H(S) = -\sum_S P(s)\log_2 P(s) \tag{15.2}$$

$$H(S|r) = -\sum_S P(s|r)\log_2 P(s|r) \tag{15.3}$$

$$H(S|R) = -\sum_R \sum_S P(r)P(s|r)\log_2 P(s|r) \tag{15.4}$$

$$I(R;S) = H(S) - H(S|R) = \sum_R \sum_S P(r)P(s|r)\log_2\left(\frac{P(s|r)}{P(s)}\right) \tag{15.5}$$

$$I(R;S) = \sum_R \sum_S P(s,r)\log_2\left(\frac{P(s,r)}{P(s)P(r)}\right) \tag{15.6}$$

CHAPTER 16

$$\vec{P} = \sum_{i=1}^{n} w_i \vec{C_i} \tag{16.1}$$

$$P_X = \sum_{i=1}^{n} w_i \cos(\theta_i) \text{ and } P_Y = \sum_{i=1}^{n} w_i \sin(\theta_i) \tag{16.2}$$

$$P(A|B) = \frac{P(A,B)}{P(B)} \tag{16.3}$$

$$P(R|d) = \prod_{i=1}^{n} P(r_i|d) \tag{16.4}$$

$$\log[P(R|d)] = \sum_{i=1}^{n} \log[P(R_i|d)] \tag{16.5}$$

CHAPTER 17

$$(X^T X)f = X^T Y \tag{17.1}$$

$$f = (X^T X)^{-1} X^T Y \tag{17.2}$$

$$Y_{k+1} = Y_k + w, w \sim N(0, W) \tag{17.3}$$

$$Y_{k+1} = Y_k a + w, w \sim N(0, W) \tag{17.4}$$

$$X_k = Y_k h + q_k, q \sim N(0, Q) \tag{17.5}$$

$$h = (Y^T Y)^{-1} Y^T X \tag{17.6}$$

$$Q = E[(X - Yh)^T (X - Yh)] \tag{17.7}$$

$$w_t^{(j)} = p(x_t | y_t = y_t^{(j)}) \tag{17.8}$$

CHAPTER 18

$$F_o = \gamma B_o \tag{18.1}$$

$$voxel = offset + gain \times hemo + \varepsilon \tag{18.2}$$

CHAPTER 19

$$V = IR \tag{19.1}$$

$$I = gV \tag{19.2}$$

$$I = g_{\max} * P_o * V \tag{19.3}$$

$$I_K = \overline{g}_K * n * V \tag{19.4}$$

$$(K_v)_{closed} \underset{k_{-1}}{\overset{k_1}{\longleftrightarrow}} (K_v)_{open} \tag{19.5}$$

$$\frac{dn}{dt} = (1-n)k_1 - nk_{-1} = k_1 - (k_1 + k_{-1})n \tag{19.6}$$

$$\begin{gathered} k_1 = \frac{0.01 * (V+10)}{\exp\left(\frac{V+10}{10}\right) - 1} \\ k_{-1} = 0.125 * \exp\left(\frac{V}{80}\right) \end{gathered} \tag{19.7}$$

$$\begin{gathered} (Na_v)_{closed} \underset{k_{-1}}{\overset{k_1}{\longleftrightarrow}} (Na_v)_{open} \\ (Na_v)_{inactive} \underset{k_{-1}}{\overset{k_1}{\longleftrightarrow}} (Na_v)_{open} \end{gathered} \tag{19.8}$$

$$I_{Na} = \overline{g}_{Na} * m * h * V \tag{19.9}$$

$$\begin{gathered} \frac{dm}{dt} = (1-m)k_1 - mk_{-1} = k_1 - (k_1 + k_{-1})m \\ \frac{dh}{dt} = (1-h)k_1 - hk_{-1} = k_1 - (k_1 + k_{-1})h \end{gathered} \tag{19.10}$$

$$\begin{gathered} k_1 = \frac{0.1 * (V+25)}{\exp\left(\frac{V+25}{10}\right) - 1} \\ k_{-1} = 4 * \exp\left(\frac{V}{18}\right) \end{gathered} \tag{19.11}$$

$$k_1 = 0.07 * \exp\left(\frac{V}{20}\right)$$
$$k_{-1} = \frac{1}{\exp\left(\frac{V+30}{10}\right)+1} \tag{19.12}$$

$$\frac{df}{dx} = \lim_{\Delta x \to 0}\frac{f(x+\Delta x)-f(x)}{\Delta x} \tag{19.13}$$

$$\frac{df}{dx} \approx \frac{f(x+\Delta x)-f(x)}{\Delta x} \Rightarrow f(x+\Delta x) \approx f(x) + \Delta x * \frac{df}{dx} \tag{19.14}$$

$$\frac{df}{dx} = 2x, \text{ where } f(0) = 1 \tag{19.15}$$

$$f(x+\Delta x) \approx f(x) + \Delta x * 2x \tag{19.16}$$

$$f(0+0.1) \approx f(0) + 0.1 * 2 * 0 \Rightarrow f(0.1) \approx 1 \tag{19.17}$$

$$f(0.1+0.1) \approx f(0.1) + 0.1 * 2 * 0.1 \Rightarrow f(0.2) \approx 1.02 \tag{19.18}$$

$$\frac{dy}{dx} = f(x,y) \tag{19.19}$$

$$y(x) = y(x_o) + \frac{y'(x_o)}{1!} * (x-x_o) + \frac{y''(x_o)}{2!} * (x-x_o)^2 + \ldots + \frac{y^{(n)}(x_o)}{n!}(x-x_o)^n \tag{19.20}$$

$$y(x_o+\Delta x) = y(x_o) + \frac{y'(x_o)}{1!} * (\Delta x) + \frac{y''(x_o)}{2!} * (\Delta x)^2 + \ldots + \frac{y^{(n)}(x_o)}{n!}(\Delta x)^n \tag{19.21}$$

$$y(x_o+\Delta x) \approx y(x_o) + \frac{y'(x_o)}{1!} * (\Delta x) + \frac{y''(x_o)}{2!} * (\Delta x)^2 \tag{19.22}$$

$$y''(x_o) = \frac{\partial f(x_o,y_o)}{\partial x} + \frac{\partial f(x_o,y_o)}{\partial y}y'(x_o) = \frac{\partial f(x_o,y_o)}{\partial x} + \frac{\partial f(x_o,y_o)}{\partial y}f(x_o,y_o) \tag{19.23}$$

$$y(x_o+\Delta x) \approx y(x_o) + f(x_o,y_o) * (\Delta x) + \left[\frac{\partial f(x_o,y_o)}{\partial x} + \frac{\partial f(x_o,y_o)}{\partial y} * f(x_o,y_o)\right] * \frac{(\Delta x)^2}{2} \tag{19.24}$$

$$f(x_o+a, y_o+b) = f(x_o,y_o) + \frac{\partial f(x_o,y_o)}{\partial x} * a + \frac{\partial f(x_o,y_o)}{\partial y} * b + \ldots \tag{19.25}$$

$$f[x_o+\Delta x, y_o+\Delta x * f(x_o,y_o)] = f(x_o,y_o) + \left[\frac{\partial f(x_o,y_o)}{\partial x} + \frac{\partial f(x_o,y_o)}{\partial y} * f(x_o,y_o)\right] * \Delta x \tag{19.26}$$

$$\frac{\Delta x}{2}\{f[x_o+\Delta x, y_o+\Delta x * f(x_o,y_o)] - f(x_o,y_o)\} = \left[\frac{\partial f(x_o,y_o)}{\partial x} + \frac{\partial f(x_o,y_o)}{\partial y} * f(x_o,y_o)\right] * \frac{(\Delta x)^2}{2} \tag{19.27}$$

$$y(x_o + \Delta x) \approx y(x_o) + \{f(x_o, y_o) + f[x_o + \Delta x, y_o + \Delta x * f(x_o, y_o)]\} * \frac{(\Delta x)}{2} \tag{19.28}$$

$$\begin{aligned} y(x_o + \Delta x) &= y(x_o) + \frac{1}{2}(u_1 + u_2) \quad \text{where} \\ u_1 &= \Delta x * f(x_o, y_o) \text{ and} \\ u_2 &= \Delta x * f(x_o + \Delta x, y_o + u_1) \end{aligned} \tag{19.29}$$

$$\begin{aligned} y(x_o + \Delta x) &= y(x_o) + \frac{1}{6}(v_1 + 2v_2 + 2v_3 + v_4) \quad \text{where} \\ v_1 &= \Delta x * f(x_o, y_o) \\ v_2 &= \Delta x * f\left(x_o + \frac{\Delta x}{2}, y_o + \frac{v_1}{2}\right) \\ v_3 &= \Delta x * f\left(x_o + \frac{\Delta x}{2}, y_o + \frac{v_2}{2}\right) \\ v_4 &= \Delta x * f(x_o + \Delta x, y_o + v_3) \end{aligned} \tag{19.30}$$

CHAPTER 20

$$V_R = IR \tag{20.1}$$

$$I = gV_R \tag{20.2}$$

$$V_C = \frac{1}{C}\int I(t)dt \tag{20.3}$$

$$\sum I_{in} = \sum I_{out} \tag{20.4}$$

$$\sum_{loop} V = 0 \tag{20.5}$$

$$V_M = \frac{1}{C_M}\int I(t)dt \tag{20.6}$$

$$I = C_M \frac{dV_M}{dt} \tag{20.7}$$

$$V_M = V_R + E_{Na} \Rightarrow V_R = V_M - E_{Na} \tag{20.8}$$

$$I_{Na} = g_{Na} * (V_M - E_{Na}) \tag{20.9}$$

$$I_K = g_K * (V_M - E_K) \tag{20.10}$$

$$I_L = g_L * (V_M - E_L) \tag{20.11}$$

$$I_{inj} = I + I_{Na} + I_K + I_L \tag{20.12}$$

$$C_M \frac{dV_M}{dt} = -g_{Na} * (V_M - E_{Na}) - g_K * (V_M - E_K) - g_L * (V_M - E_L) + I_{inj} \tag{20.13}$$

$$g_K = \overline{g}_K * n \tag{20.14}$$

$$\frac{dn}{dt} = k_{1n} - (k_{1n} + k_{-1n})n \tag{20.15}$$

$$g_{Na} = \overline{g}_{Na} * m * h \tag{20.16}$$

$$\begin{aligned} \frac{dm}{dt} &= k_{1m} - (k_{1m} + k_{-1m})m \\ \frac{dh}{dt} &= k_{1h} - (k_{1h} + k_{-1h})h \end{aligned} \tag{20.17}$$

$$\begin{aligned} C_m \frac{dV_M}{dt} &= -\overline{g}_{Na} mh(V_M - E_{Na}) - \overline{g}_K n(V_M - E_K) - g_L(V_M - E_L) + I_{inj} \\ \frac{dn}{dt} &= k_{1n} - (k_{1n} + k_{-1n})n \\ \frac{dm}{dt} &= k_{1m} - (k_{1m} + k_{-1m})m \\ \frac{dh}{dt} &= k_{1h} - (k_{1h} + k_{-1h})h. \end{aligned} \tag{20.18}$$

$$\begin{aligned} C_m \frac{dV_M}{dt} &= -\overline{g}_{Na} m^3 h(V_M - E_{Na}) - \overline{g}_K n^4(V_M - E_K) - g_L(V_M - E_L) + I_{inj} \\ \frac{dn}{dt} &= k_{1n} - (k_{1n} + k_{-1n})n \\ \frac{dm}{dt} &= k_{1m} - (k_{1m} + k_{-1m})m \\ \frac{dh}{dt} &= k_{1h} - (k_{1h} + k_{-1h})h. \end{aligned} \tag{20.19}$$

$$\begin{aligned} k_{1n} &= \frac{0.01 * (10 - V_M)}{\exp\left(\dfrac{10 - V_M}{10}\right) - 1} \\ k_{-1n} &= 0.125 * \exp\left(\frac{-V_M}{80}\right) \end{aligned} \tag{20.20}$$

$$k_{1m} = \frac{0.1 * (25 - V_M)}{\exp\left(\frac{25 - V_M}{10}\right) - 1}$$
$$k_{-1m} = 4 * \exp\left(\frac{-V_M}{18}\right) \tag{20.21}$$

$$k_{1h} = 0.07 * \exp\left(\frac{-V_M}{20}\right)$$
$$k_{-1h} = \frac{1}{\exp\left(\frac{30 - V_M}{10}\right) + 1} \tag{20.22}$$

CHAPTER 21

$$\frac{dC}{dt} = \frac{1}{\tau_c}(-C - kH + L) \tag{21.1}$$

$$\frac{dH}{dt} = \frac{1}{\tau_H}(-H + C) \tag{21.2}$$

$$\tilde{C} = C - \frac{L}{k+1} \text{ and } \tilde{H} = H - \frac{L}{k+1} \tag{21.3}$$

$$\frac{d\tilde{C}}{dt} = \frac{1}{\tau_C}(-\tilde{C} - k\tilde{H}) \tag{21.4}$$

$$\frac{d\tilde{H}}{dt} = \frac{1}{\tau_H}(-\tilde{H} + \tilde{C}) \tag{21.5}$$

$$\tilde{C}(0) = \tilde{H}(0) = \frac{L}{k+1} \tag{21.6}$$

$$\frac{dx}{dt} = x + y \tag{21.7}$$

$$\frac{dx}{dt} = 4x + y \tag{21.8}$$

$$\begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{21.9}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \vec{v} \quad \text{and} \quad A = \begin{bmatrix} 1 & 1 \\ 4 & 1 \end{bmatrix} \tag{21.10}$$

$$\frac{d\vec{v}}{dt} = A * \vec{v} \tag{21.11}$$

$$\frac{d\vec{v}}{dt} = VDV^{-1} * \vec{v} \tag{21.12}$$

$$V^{-1}\frac{d\vec{v}}{dt} = V^{-1}VDV^{-1} * \vec{v} = DV^{-1} * \vec{v} \tag{21.13}$$

$$\frac{d\vec{u}}{dt} = D * \vec{u} \tag{21.14}$$

$$\frac{d\vec{u}}{dt} = \begin{bmatrix} \frac{du_1}{dt} \\ \frac{du_2}{dt} \end{bmatrix} = \begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix} * \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 3u_1 \\ -u_2 \end{bmatrix} \Rightarrow \tag{21.15}$$

$$\frac{du_1}{dt} = 3u_1$$

$$\frac{du_2}{dt} = -u_2$$

$$\vec{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} C_1e^{3t} \\ C_2e^{-t} \end{bmatrix} \tag{21.16}$$

$$\vec{v} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & -2 \end{bmatrix} * \begin{bmatrix} C_1e^{3t} \\ C_2e^{-t} \end{bmatrix} = \begin{bmatrix} C_1e^{3t} + C_2e^{-t} \\ 2C_1e^{3t} - 2C_2e^{-t} \end{bmatrix} = C_1\begin{bmatrix} 1 \\ 2 \end{bmatrix}e^{3t} + C_2\begin{bmatrix} 1 \\ -2 \end{bmatrix}e^{-t} \Rightarrow \tag{21.17}$$

$$x(t) = C_1e^{3t} + C_2e^{-t}$$

$$y(t) = 2C_1e^{3t} - 2C_2e^{-t}$$

$$\vec{v} = \begin{bmatrix} x \\ y \end{bmatrix} = C_1 * EV_1 * e^{\lambda_1 t} + C_2 * EV_2 * e^{\lambda_2 t} \tag{21.18}$$

$$\begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = V^{-1}\begin{bmatrix} x_o \\ y_o \end{bmatrix} \tag{21.19}$$

CHAPTER 22

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \tag{22.1}$$

$$\frac{du}{dt} = a(bv - u) \tag{22.2}$$

$$\text{if } v \geq 30, \text{ then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{22.3}$$

CHAPTER 23

$$\frac{\partial v}{\partial t} = f(v) - r + I + \frac{\partial^2 v}{\partial x^2} \tag{23.1}$$

$$\frac{\partial r}{\partial t} = av - br \tag{23.2}$$

$$\frac{d^2 v(x)}{dx^2} \sim \frac{v(x + \Delta x) - 2v(x) + v(x - \Delta x)}{(\Delta x)^2} \tag{23.3}$$

$$\frac{d^2 v_j}{dx^2} \sim \frac{v_{j+1} - 2v_j + v_{j-1}}{(\Delta x)^2} \tag{23.4}$$

$$\frac{\partial v}{\partial t} = 10\left[v - \frac{1}{3}v^3 - r + D\frac{\partial^2 v}{\partial x^2}\right] + I \tag{23.5}$$

$$\frac{\partial r}{\partial t} = p[a + 1.25v - br] \tag{23.6}$$

$$\frac{\partial v}{\partial t} = -v(a - v)(1 - v) - r + D\frac{\partial^2 v}{\partial x^2} \tag{23.7}$$

$$\frac{\partial r}{\partial t} = bv - gr \tag{23.8}$$

CHAPTER 24

$$\frac{\partial \phi}{\partial t} = D\nabla^2 \phi \tag{24.1}$$

$$\frac{\partial \phi}{\partial t} = D\frac{\partial^2 \phi}{\partial x^2} \tag{24.2}$$

$$\phi_{space}^{time} \tag{24.3}$$

$$\frac{\phi_x^{t+1} - \phi_x^t}{\Delta t} = D\frac{\left((\phi_{x+1}^t - \phi_x^t) - (\phi_x^t - \phi_{x-1}^t)\right)}{(\Delta x)^2} \tag{24.4}$$

$$\phi_x^{t+1} = \phi_x^t + D\frac{\Delta t}{(\Delta x)^2}(\phi_{x+1}^t - 2\phi_x^t + \phi_{x-1}^t) \tag{24.5}$$

$$\frac{\partial \phi}{\partial t} = B\frac{\partial \phi}{\partial x} + D\frac{\partial^2 \phi}{\partial x^2} \tag{24.6}$$

CHAPTER 26

$$P(x = k) \equiv \frac{\lambda^k}{k!}e^{-\lambda} \tag{26.1}$$

$$P(x \le k) \equiv 1 - e^{-\lambda k} \tag{26.2}$$

$$E[x] = \sum_{x=0}^{\infty} x\frac{\lambda^x}{x!}e^{-\lambda} \tag{26.3}$$

$$Var[x] = E[x^2] - (E[x])^2, \text{ where } E[x^2] = \sum_{x=0}^{\infty} x^2\frac{\lambda^x}{x!}e^{-\lambda} \tag{26.4}$$

$$\frac{1}{\Delta t}\int_t^{t+\Delta t} \lambda(\tau)d\tau \tag{26.5}$$

$$\lambda(t|H) \tag{26.6}$$

$$\lambda(t|H) = \frac{f(t|H)}{1 - \int_{u'}^{t} f(u|H)du} \tag{26.7}$$

CHAPTER 27

$$f(k; n, p) = C(n, k)p^k(1 - p)^k \tag{27.1}$$

$$\text{where } C(n, k) = \binom{n}{k} = \frac{n!}{k!(n - k)!} \tag{27.2}$$

$$\frac{\partial \phi}{\partial t} = D\nabla^2\phi \tag{27.3}$$

$$\frac{\partial \phi}{\partial t} = D\frac{\partial^2 \phi}{\partial x^2} + D\frac{\partial^2 \phi}{\partial y^2} \tag{27.4}$$

$$\phi_{space}^{time} \tag{27.5}$$

$$\frac{\phi_{x,y}^{t+1} - \phi_{x,y}^{t}}{\Delta t} = D\left[\frac{\left((\phi_{x+1,y}^t - \phi_{x,y}^t) - (\phi_{x,y}^t - \phi_{x-1,y}^t)\right)}{(\Delta x)^2} + \frac{\left((\phi_{x,y+1}^t - \phi_{x,y}^t) - (\phi_{x,y}^t - \phi_{x,y-1}^t)\right)}{(\Delta y)^2}\right] \tag{27.6}$$

$$\phi_{x,y}^{t+1} = \phi_{x,y}^{t} + D\Delta t\left[\frac{(\phi_{x+1,y}^{t} - 2\phi_{x,y}^{t} + \phi_{x-1,y}^{t})}{(\Delta x)^2} + \frac{(\phi_{x,y+1}^{t} - 2\phi_{x,y}^{t} + \phi_{x,y-1}^{t})}{(\Delta y)^2}\right] \tag{27.7}$$

$$\phi_{x,y}^{t+1} = \phi_{x,y}^{t} + D\Delta t\left[\frac{(\phi_{x+1,y}^{t} + \phi_{x,y+1}^{t} + \phi_{x-1,y}^{t} + \phi_{x,y-1}^{t} - 4\phi_{x,y}^{t})}{(\Delta x)^2}\right] \tag{27.8}$$

CHAPTER 28

$$\Delta w_{ij} = \varepsilon \cdot pre_i \cdot post_j \tag{28.1}$$

$$W = \vec{g}\vec{f}^{T} \tag{28.2}$$

$$W = \vec{g}_{go}\vec{f}^{T}_{green} + \vec{g}_{stop}\vec{f}^{T}_{red} + \vec{g}_{slow}\vec{f}^{T}_{yellow} + \ldots \tag{28.3}$$

$$W\vec{f}_{green} = \vec{g}_{go}\vec{f}^{T}_{green}\vec{f}_{green} + \vec{g}_{stop}\vec{f}^{T}_{red}\vec{f}_{green} + \vec{g}_{slow}\vec{f}^{T}_{yellow}\vec{f}_{green} + \ldots \tag{28.4}$$

$$W\vec{f}_{green} = \rightarrow g_{go} \cdot 1 + 0 + 0 + \ldots \tag{28.5}$$

$$\Delta w_{ij} = \varepsilon \cdot pre_i \cdot post_j - \varepsilon \cdot post_j \cdot w_{ij} \tag{28.6}$$

$$\Delta w_{ij} = \varepsilon \cdot post\mathrm{j} \cdot (pre_i - w_{ij}) \tag{28.7}$$

CHAPTER 29

$$\Delta w_{ij} = \varepsilon(t_j - post_j)pre_i \tag{29.1}$$

References

PREFACE REFERENCES

Karpicke, J.D. and Roediger, H.L. (2008). The critical importance of retrieval for learning. *Science*, *319*, 966–968.

CHAPTER 1 REFERENCES

Genesee, F. (1985). Second language learning through immersion: A review of U.S. programs. *Review of Educational Research*, *55*(4, Winter), 541–561.

Hubel, D.H. and Wiesel, T.N. (2004). *Brain and visual perception: The story of a 25-year collaboration*. New York: Oxford University Press, 707.

Marr, D. (1982). *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. New York: W.H. Freeman and Company.

CHAPTER 2 REFERENCES

Berry, D.C. and Broadbent, D.E. (1984). On the relationship between task performance and associated verbalized knowledge. *The Quarterly Journal of Experimental Psychology*, *36A*, 209–231.

CHAPTER 3 REFERENCES

Donders, F.C. (1868). Over de snelheid van psychische processen. Onderzoekingen gedaan in het Physiologisch Laboratorium der Utrechtsche Hoogeschool, Tweede reeks, II, 92–120. Reprinted in and translated as Donders, F.C. (1969). On the speed of mental processes. *Acta Psychologica*, *30*, Attention and Performance II, 412–431.

Shepard, R. and Metzler., J. (1971). Mental rotation of three dimensional objects. *Science, 171*(972), 701–703.

Treisman, A. and Gelade, G. (1980). A feature integration theory of attention. *Cognitive Psychology, 12*, 97–136.

CHAPTER 4 REFERENCES

Helmholtz, H. (1867). *Handbuch der Physiologischen Optik*. Hamburg: Voss.

James, W. (1890). *The principles of psychology*. Vol. 1. New York: Henry Holt.

Posner, M.I. (1980). Orienting of attention. *Quarterly Journal of Experimental Psychology*, *32*, 3–25.

CHAPTER 5 REFERENCES

Carpenter, R. and Robson, J. (1999). *Vision Research: A Practical Guide to Laboratory Methods*. New York: Oxford University Press.

Fechner, G.T. (1860). *Elemente der Psychophysik*. Leipzig: Breitkopf und Härtel.

Hecht, Shlaer and Pirenne (1942). Energy, quanta, and vision. *J. Gen. Physiol*. *25*, 819–840.

Norton, T.T., Corliss, D.A., and Bailey, J.E. (2002). *The Psychophysical Measurement of Visual Function*. Woburn, MA: Butterworth-Heinemann.

CHAPTER 6 REFERENCES

Green, D.M. and Swets, J.A. (1966). *Signal Detection Theory and Psychophysics*. New York: John Wiley & Sons, Inc.

Fisher, R.A. (1925). *Statistical Methods for Research Workers*. Edinburgh: Oliver and Boyd.

Rosenthal, R. (1976). *Experimenter Effects in Behavioral Research*. New York: Irvington.

Ziliak, S.T. and McCloskey, D.N. (2008). *The Cult of Statistical Significance. How the Standard Error Costs Us Jobs, Justice, and Lives*. University of Michigan Press.

CHAPTER 7 REFERENCES

Van Drongelen, W. (2006). *Signal Processing for Neuroscientists: An Introduction to the Analysis of Physiological Signals.*

Hillenbrand, J., Getty, L.A., Clark, M.J., and Wheeler, K. (1995). Acoustic characteristics of American English vowels. *J. Acoust. Soc. Am*., *97*, 3099–3111.

Peterson, G.E. and Barney, H.L. (1952). Control methods used in a study of the vowels, *J. Acoust. Soc. Am. 24,* 175–184.

CHAPTER 8 REFERENCES

Van Drongelen, W. (2006). *Signal Processing for Neuroscientists: An Introduction to the Analysis of Physiological Signals*. Burlington, MA: Academic Press.

CHAPTER 9 REFERENCES

Percival, D. and Walden, A. (2000). *Wavelet Methods for Time Series Analysis*. Cambridge: Cambridge University Press.

Quiroga, R., Nadasdy, Z., and Ben-Shaul, Y. (2004). Unsupervised spike detection and sorting with wavelets and superparamagnetic clustering. *Neural Computation*, *16*, 1661–1687.

CHAPTER 10 REFERENCES

Dayan, P. and Abbott, L.F. (2001). *Theoretical neuroscience*. Cambridge, MA: MIT Press.

Lotto, R.B., Williams, S.M., and Purves, D. (1999). An empirical basis for Mach bands. *Proceedings of the National Academy of Sciences USA*, *96*, 5239–5244.

Ratliff, F. (1965). *Mach bands: Quantitative studies on neural networks in the retina*. San Francisco, CA: Holden-Day.

Sekular, R. and Blake, R. (2002). *Perception* 4th Ed. New York: McGraw-Hill.

CHAPTER 11 REFERENCES

(no references)

CHAPTER 12 REFERENCES

Fitzhugh, R. (1961). Impulses and physiological states in theoretical models of nerve membrane. *Biophysical Journal*, *1*, 445–466.

CHAPTER 13 REFERENCES

Georgopoulos, A.P., Kalaska, J.F., Caminiti, R., and Massey, J.T. (1982). On the relations between the direction of two-dimensional arm movements and cell discharge in primate motor cortex. *The Journal of Neuroscience*, *2*(11), 1527–1537.

Hartline, H.K. (1940). The receptive fields of optic nerve fibers. *American Journal of Physiology*, *130*, 690–699.

CHAPTER 14 REFERENCES

(no references)

CHAPTER 15 REFERENCES

Hatsopoulos, N.G., Ojakangas, C.L., Paninski, L., and Donoghue, J.P. (1998). Information about movement direction obtained from synchronous activity of motor cortical neurons. *Proceedings of the National Academy of Sciences USA*, *95*(26), 15706–15711.

Optican, L.M. and Richmond, B.J. (1987). Temporal encoding of two-dimensional patterns by single units in primate inferior temporal cortex. III. Information theoretic analysis. *Journal of Neurophysiology*, *57*(1), 162–178.

Panzeri, S., Senatore, R., Montemurro, M.A., and Petersen, R.S. (2007). Correcting for the sampling bias problem in spike train information measures. *Journal of Neurophysiology*, *98*(3), 1064–1072.

Richmond, B.J. and Optican, L.M. (1987). Temporal encoding of two-dimensional patterns by single units in primate inferior temporal cortex. II. Quantification of response waveform. *Journal of Neurophysiology*, *57*(1), 147–161.

Richmond, B.J., Optican, L.M., Podell, M., and Spitzer, H. (1987). Temporal encoding of two-dimensional patterns by single units in primate inferior temporal cortex. I. Response characteristics. *Journal of Neurophysiology*, *57*(1), 132–146.

Shannon, C.E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, *27*, 379–423, 623–656.

CHAPTER 16 REFERENCES

Georgopoulos, A.P., Schwartz, A.B., and Kettner, R.E. (1986). Neuronal population coding of movement direction. *Science*, *233*(4771), 1416–1419.

Hochberg, L.R., Serruya, M.D., Friehs, G.M., Mukand, J.A., Saleh, M., Caplan, A.H., et al. (2006). Neuronal ensemble control of prosthetic devices by a human with tetraplegia. *Nature*, *442*(7099), 164–171.

Papsin, B.C. and Gordon, K.A. (2007). Cochlear implants for children with severe-to-profound hearing loss. *New England Journal of Medicine*, *357*(23), 2380–2387.

CHAPTER 17 REFERENCES

Brockwell, A.E., Rojas, A.L., and Kass, R.E. (2004). Recursive Bayesian decoding of motor cortical signals by particle filtering. *Journal of Neurophysiology*, *91*(4), 1899–1907.

Brown, E.N., Frank, L.M., Tang, D., Quirk, M.C., and Wilson, M.A. (1998). A statistical paradigm for neural spike train decoding applied to position prediction from ensemble firing patterns of rat hippocampal place cells. *The Journal of Neuroscience: The Official Journal of the Society for Neuroscience*, *18*(18), 7411–7425.

Georgopoulos, A.P., Kettner, R.E., and Schwartz, A.B. (1988). Primate motor cortex and free arm movements to visual targets in three-dimensional space. II. Coding of the direction of movement by a neuronal population. *The Journal of Neuroscience: The Official Journal of the Society for Neuroscience*, *8*(8), 2928–2937.

Hochberg, L.R., Serruya, M.D., Friehs, G.M., Mukand, J.A., Saleh, M., Caplan, A.H., et al. (2006). Neuronal ensemble control of prosthetic devices by a human with tetraplegia. *Nature*, *442*(7099), 164–171.

Serruya, M.D., Hatsopoulos, N.G., Paninski, L., Fellows, M.R., and Donoghue, J.P. (2002). Instant neural control of a movement signal. *Nature*, *416*(6877), 141–142.

Warland, D.K., Reinagel, P., and Meister, M. (1997). Decoding visual information from a population of retinal ganglion cells. *Journal of Neurophysiology*, *78*(5), 2336–2350.

Wu, W., Shaikhouni, A., Donoghue, J.P., and Black, M.J. (2004). Closed-loop neural control of cursor motion using a Kalman filter. *Conference Proceedings: Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, *6*, 4126–4129.

CHAPTER 18 REFERENCES

Kwong, K.K., et al. (1992). Dynamic magnetic resonance imaging of human brain activity during primary sensory stimulation. *Proceedings of the National Academy of Science USA*, *89*(12), 5675–5679.

Ogawa, S., et al. (1992). Intrinsic signal changes accompanying sensory stimulation: Functional brain mapping with magnetic resonance imaging. *Proceedings of the National Academy of Science USA*, *89*(13), 5951–5955.

Smith, S.M. (2004). Overview of fMRI analysis. *British Journal of Radiology*, *77*(Spec. No. 2), S167–175.

CHAPTER 19 REFERENCES

Hodgkin, A.L. and Huxley A.F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology, 116*, 500–544.

CHAPTER 20 REFERENCES

Hodgkin, A.L. and Huxley, A.F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, *117*, 500–544.

Hodgkin, A.L. and Katz, B. (1949). The effect of sodium ions on the electrical activity of the gaint axon of the squid. *Journal of Physiology, 108*(1), 37–77.

CHAPTER 21 REFERENCES

(no references)

CHAPTER 22 REFERENCES

Izhikevich, E.M. (2003). Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, *14*(6), 1569–1572.

CHAPTER 23 REFERENCES

Murray, J.D. (2002). *Mathematical biology I: An introduction*. New York: Springer-Verlag.

Strauss, W.A. (1992). *Partial differential equations: An introduction*. New York: John Wiley & Sons, Inc.

Wilson, H.R. (1999). *Spikes, decisions, and actions: Dynamical foundations of neuroscience*. Oxford: Oxford University Press.

CHAPTER 24 REFERENCES

Shadlen, M.N. and Newsome, W.T. (2001). Neural basis of a perceptual decision in the parietal cortex (area LIP) of the rhesus monkey. *Journal of Neurophysiology*, *86*(4), 1916–1936.

Swensson, R. (1972). The elusive tradeoff: Speed vs accuracy in visual discrimination tasks. *Perception & Psychophysics.* Jul Vol 12(1-A), 16–32.

CHAPTER 25 REFERENCES

(no references)

CHAPTER 26 REFERENCES

Brown, E.N., Barbieri, R., Ventura, V., Kass, R.E., and Frank, L.M. (2002). The time-rescaling theorem and its application to neural spike train data analysis. *Neural. Comput*. *14*, 325–346.

Donald E. Knuth (1969). *Seminumerical Algorithms*. The Art of Computer Programming, Volume 2. Addison Wesley.

CHAPTER 27 REFERENCES

del Castillo, J. and Katz, B. (1954). The effect of magnesium on the activity of motor nerve endings. *J. Physiol (Lond), 124*, 553–559.

del Castillo, J. and Katz, B. (1954). Quantal components of the end-plate potential. *J Physiol*, *124*, 560–573.

Fatt, P. and Katz. B. (1952). Spontaneous sunthershold activity at motor nerve endings. *J Physiol (Lond), 117*, 109–128.

CHAPTER 28 REFERENCES

Anderson, J.A., et al. (1977). Distinctive features, categorical perception, and probability learning: Some applications of a neural model. *Psychological Review*, *84*, 413–451.

Bliss, T.V. and Lomo, T. (1973). Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path. *Journal of Physiology*, *232*(2), 331–356.

Gauthier, I., Behrmann, M., and Tarr, M.J. (2004). Are Greebles like faces? Using the neuropsychological exception to test the rule. *Neuropsychologia*, *42*(14), 1961–1970.

Gauthier, I. and Tarr, M.J. (1997). Becoming a "Greeble" expert: Exploring mechanisms for face recognition. *Vision Research*, *37*(12), 1673–1682.

Hebb, D.O. (1949). *Organization of behavior*. New York: John Wiley & Sons.

James, W. (1890). *The principles of psychology*. New York: Henry Holt & Sons, Inc.

Rumelhart, D.E. and Zipser, D. (1985). Feature discovery by competitive learning. *Cognitive Science*, *9*, 75–112.

CHAPTER 29 REFERENCES

Adrian, E.D. and Matthews, R. (1927). The action of light on the eye: Part I. The discharge of impulses in the optic nerve and its relation to the electric changes in the retina. *Journal of Physiology*, *63*(4), 378–414.

Albus, J.S. (1971). A theory of cerebellar function. *Math. Biosci*. *10*, 25–61.

Hebb, D.O. (1949). *Organization of behavior*. New York: John Wiley & Sons.

Ito, M. and Kano, M. (1982). Long-lasting depression of parallel fiber-Purkinje cell transmission induced by conjunctive stimulation of parallel fibers and climbing fibers in the cerebellar cortex. *Neuroscience Letters*, *33*(3), 253–258.

Krupa, D.J., Thompson, J.K., and Thompson, R.F. (1993). Localization of a memory trace in the mammalian brain. *Science*, *260*(5110), 989–991.

Marr, D. (1969). A theory of cerebellar cortex. *J Physiol*. Jun:*202*(2), 437–470.

McCulloch, W.S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *5*, 115–133.

Medina, J.F., et al. (2000). Mechanisms of cerebellar learning suggested by eyelid conditioning. *Current Opinions in Neurobiology*, *10*(6), 717–724.

Minsky, M. and Papert, S. (1969). *Perceptrons: An introduction to computational geometry*. Cambridge, MA: MIT Press.

Ojakangas, C.L. and Ebner, T.J. (1992). Purkinje cell complex and simple spike changes during a voluntary arm movement learning task in the monkey. *Journal of Neurophysiology*, *68*(6), 2222–2236.

Ojakangas, C.L. and Ebner, T.J. (1994). Purkinje cell complex spike activity during voluntary motor learning: Relationship to kinematics. *Journal of Neurophysiology*, *72*(6), 2617–2630.

Rescorla, R.A. and Wagner, A.R. (1972). A theory of Pavlovian conditioning: Variations in the effectiveness of reinforcement and nonreinforcement. In Black, A. and Prokasy, W.F. (Eds.), *Classical conditioning II* (64–69). New York: Appleton-Century-Crofts.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*, 386–408.

Widrow, B. and Hoff, M.E. (1960). Adaptive switching circuits. In *IRE WESCON Convention Record*. New York: IRE, 96–104.