MORGAN & CLAYPOOL PUBLISHERS

MATLAB for Engineering and the Life Sciences

Joseph V. Tranquillo

SYNTHESIS LECTURES ON ENGINEERING

MATLAB for Engineering and the Life Sciences

Synthesis Lectures on Engineering

MATLAB for Engineering and the Life Sciences
Joseph V. Tranquillo
2011

Systems Engineering: Building Successful Systems
Howard Eisner
2011

Fin Shape Thermal Optimization Using Bejan's Constructal Theory
Giulio Lorenzini, Simone Moretti, and Alessandra Conti
2011

Geometric Programming for Design and Cost Optimization (with illustrative case study problems and solutions), Second Edition
Robert C. Creese
2010

Survive and Thrive: A Guide for Untenured Faculty
Wendy C. Crone
2010

Geometric Programming for Design and Cost Optimization (with Illustrative Case Study Problems and Solutions)
Robert C. Creese
2009

Style and Ethics of Communication in Science and Engineering
Jay D. Humphrey and Jeffrey W. Holmes
2008

Introduction to Engineering: A Starter's Guide with Hands-On Analog Multimedia Explorations
Lina J. Karam and Naji Mounsef
2008

Introduction to Engineering: A Starter's Guide with Hands-On Digital Multimedia and Robotics Explorations
Lina J. Karam and Naji Mounsef
2008

CAD/CAM of Sculptured Surfaces on Multi-Axis NC Machine: The DG/K-Based Approach
Stephen P. Radzevich
2008

Tensor Properties of Solids, Part Two: Transport Properties of Solids
Richard F. Tinder
2007

Tensor Properties of Solids, Part One: Equilibrium Tensor Properties of Solids
Richard F. Tinder
2007

Essentials of Applied Mathematics for Scientists and Engineers
Robert G. Watts
2007

Project Management for Engineering Design
Charles Lessard and Joseph Lessard
2007

Relativistic Flight Mechanics and Space Travel
Richard F. Tinder
2006

MATLAB for Engineering and the Life Sciences

Joseph V. Tranquillo

www.morganclaypool.com

ISBN: 9781608457106 paperback
ISBN: 9781608457113 ebook

DOI 10.2200/S00375ED1V01Y201107ENG015

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON ENGINEERING

Lecture #15
Series ISSN
Synthesis Lectures on Engineering
Print 1939-5221 Electronic 1939-523X

MATLAB for Engineering and the Life Sciences

Joseph V. Tranquillo
Bucknell University

SYNTHESIS LECTURES ON ENGINEERING #15

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

In recent years, the life sciences have embraced simulation as an important tool in biomedical research. Engineers are also using simulation as a powerful step in the design process. In both arenas, Matlab has become the gold standard. It is easy to learn, flexible, and has a large and growing userbase. *MATLAB for Engineering and the Life Sciences* is a self-guided tour of the basic functionality of Matlab along with the functions that are most commonly used in biomedical engineering and other life sciences. Although the text is written for undergraduates, graduate students and academics, those in industry may also find value in learning Matlab through biologically inspired examples. For instructors, the book is intended to take the emphasis off of learning syntax so that the course can focus more on algorithmic thinking. Although it is not assumed that the reader has taken differential equations or a linear algebra class, there are short introductions to many of these concepts. Following a short history of computing, the Matlab environment is introduced. Next, vectors and matrices are discussed, followed by matrix-vector operations. The core programming elements of Matlab are introduced in three successive chapters on scripts, loops, and conditional logic. The last three chapters outline how to manage the input and output of data, create professional quality graphics and find and use Matlab toolboxes. Throughout, biomedical examples are used to illustrate Matlab's capabilities.

KEYWORDS

computing, MATLAB, matrix, vector, loops, scripting, conditional logic, biological computing, programming, simulation

Contents

Preface

In 2004, Joel Cohen published a paper in the Public Library of Science (PLoS) Biology, titled "Mathematics is Biology's Next Microscope, only Better; Biology is Mathematics' Next Physics, Only Better". The premise of the article was that in the near future there will be an explosion in both math and biology as the two develop a similar synergistic relationship to the one that exists between math and physics. The article goes on to hint that the computer will play a very large role in this revolution, pushing mathematicians to confront the complexity and unpredictable nature of biology, and pushing biologists to become more comfortable with the rigor of mathematics. To quote directly,

> *The four main points of the applied mathematical landscape are data structures, algorithms, theories and models (including all pure mathematics), and computers and software. Data structures are ways to organize data, such as the matrix used above to describe the biological landscape. Algorithms are procedures for manipulating symbols. Some algorithms are used to analyze data, others to analyze models. Theories and models, including the theories of pure mathematics, are used to analyze both data and ideas. Mathematics and mathematical theories provide a testing ground for ideas in which the strength of competing theories can be measured. Computers and software are an important, and frequently the most visible, vertex of the applied mathematical landscape.*

If you are going to work in the life sciences in the coming decades, it will be necessary for you to master the rigor of algorithmic thinking, ways of storing and manipulating data, and the simulation of biological models.

Engineers have been using simulation as a tool for many decades. It has been incorporated into nearly every phase of the design process and is a core tool of engineering science. As such, an amazing array of specialized computing languages have cropped up, each tailored to particular needs. As learning to program can be a significant investment, you many wonder which tool you should learn. It should be a tool that is easy to learn and useful right away. A good first language should introduce you to the main elements of *every* programming language so that you can easily learn a more specific language later. It should also be a language with a large enough userbase that you can share code and algorithms. As evidenced by the number of courses taught at the undergraduate level, Matlab fits the bill on all counts.

No one source could possibly do justice to the enormous capabilities of Matlab. You can think of this text as the survival version of Matlab. As such, it is written to be a breadth-first

approach, from which the reader can jump off to other sources to go into more depth. Given the outstanding world-wide support for Matlab, after reading this text, you should be able to find what you need online.

For the student, it is important to understand that coding is like learning to ride a bike - you can only learn through concentrated, hands-on experience. And, like learning to ride one bike will make learning to ride another bike easier, once you learn how to program in Matlab, it will be much easier to pick up another programming language. A common problem with learning a language from a programming manual is that the language is divorced from actual problems. In this text, an effort was made to tie programming techniques and concepts to biomedical or biological applications. As such, there is a bias toward simulating classic models from theoretical biology and biophysics. At the end of most chapters are a series of exercises. It is important to complete them because it is here that additional Matlab commands and concepts are introduced. You will also find that you will be typing in much of the code that appears in the text by hand. There is a reason why it is important to type in the code yourself - in doing so, you will have time to question the purpose of the line.

For the instructor, the intent of this text is to take the emphasis off of learning the syntax of Matlab so that your class and lab time can focus on algorithmic thinking, mathematical routines, and other higher-level topics that are difficult to learn from a text. A command line approach is used rather than to rely on Matlab's many built-in graphical user interfaces because the command line is backward compatible and will work on different computing architectures. The author has written a conference proceeding for the American Society of Engineering Education (2011), "A Semester-Long Student-Driven Computational Project" (download from www.asee.org or contact the author at jvt002@bucknell.edu), that details how the text was incorporated into a course. In particular, the paper advocates the idea of "Coding to Think", the engineering science equivalent of "Writing to Think". The article also contains a template for a semester-long project, ideas for games that can teach algorithmic thinking as well as a number of references to other computing education papers.

This text would not have been possible without the support of several important groups. First, I would like to thank the Biomedical Engineering Department at Bucknell University, most especially the Class of 2012 who used the first version of this text. Second, I would like to thank a number of faculty colleagues, most especially Jim Maneval, Ryan Snyder, Jim Baish, Donna Ebenstein and Josh Steinhurst, for their helpful conversation and comments. Last, I wish to thank my family for their patience and for keeping me focused on what is really important in life.

Joseph V. Tranquillo
Lewisburg, Pennsylvania

CHAPTER 1

Introduction

1.1 INTRODUCTION

Learning to program can be compared to learning to ride a bike - you can't really learn it from a book, but once you do learn you will never forget how. The reason is that learning to program is really learning a thought process.

This text is not meant to be a supplement for a rigorous approach to Matlab. It is meant to explain why Matlab is an important tool to learn as an undergraduate and to highlight the portions of Matlab that are used on a daily basis. Unfortunately, you will not find the coolest, fanciest or even the best parts of Matlab here, but rather a biased view of the most useful parts. You can think of this as survival Matlab.

1.2 A SHORT HISTORY OF COMPUTING

Matlab is in some sense a blip in the overall history of computing. To provide some context, below is an abbreviated history of computing.

1.2.1 THE PRE-HISTORY OF COMPUTING

Any history of computing must start with the logic of Aristotle. He was responsible for what in computing has become known as conditional logic (what Aristotle called a Syllogism and later was called deductive reasoning). For example, "If all men are mortal and Socrates is a man, then Socrates is mortal". Aristotle went on to categorize various types of conditionals, including the logical ideas of AND, OR and NOT.

The next great computational hurdle occurred with the publication of *An Investigation of the Laws of Thought, on Which are Founded the Mathematical Theories of Logic and Probabilities* in 1854 by George Boole. In that work, Boole laid out a method of transforming Aristotle's logical statements into formal mathematical calculus. The key insight was that just as there are formal operations that work on numbers, *e.g.,* addition and division, there also exist formal operations that work on logical statements. More work by Boole and Augustus De Morgan advocated the position that logical human thought was simply computation following mathematical rules, and that a machine could in principle perform the same functions.

An attempt to build such a machine was carried out by Charles Babbage, an English mathematician and mechanical engineer, even before Boole had published his work. Babbage laid out plans for what was called a *analytical engine*, a mechanical device that would realize the type of general computing outlined by Boole. As in most practical applications of theory, there were a number of technical obstacles to overcome. Perhaps the greatest was that Babbage could not secure funding to build his device. It was in fact never built until the London Science Museum used Babbage's original plans to make a perfectly working analytical engine in 1991. What is amazing is that Babbage foresaw the type of architecture that would be necessary to make a working computer, including a CPU with some sort of programming language, a data bus, memory and even a type of printer and screen to visualize outputs.

As luck would have it the daughter of Lord Byron, Ada Lovelace, helped to translate Babbage's work into French. In her translation, she added in many of her own ideas which came to the attention of Babbage. As a result of those notes, Babbage realized that for his general computing device (hardware) to perform specific functions, a programming language (software) would be necessary. While Babbage focused on the mechanics of the device, Lovelace began to write the first computer algorithms. The first computer program ever written was by Ada and computed Bernoulli numbers. It is amazing that she was writing algorithms for a machine that didn't even exist! One of Lovelace's major advancements was to show how data and the computer instructions for operating on that data could be saved in the same memory. She also was the first to recognize that a computer could do more than act as a fancy calculator.

While computing is now often regarded as a practical pursuit, there are some who have gone down the more philosophical road outlined by Aristotle and Boole. For example, Steven Wolfram, the creator of Mathematica, published a book called *A New Kind of Science* in 2002 that advocated the idea that all of reality (including time and space) are the result of a giant algorithm. Others in the artificial intelligence and cognitive science fields have used the computer as a metaphor (either for or against) the human mind as nothing but a very good computer. There is now even a field called *experimental mathematics* which attempts to prove true mathematical statements with a new criteria for proof that uses numerical approximations on a computer.

1.2.2 THE EARLY HISTORY OF DIGITAL COMPUTING

The history of digital computing in the form we know it today began with a series of seminal papers in the 1930s by Kurt Godel, Alonzo Church, Emil Post and Alan Turing. All helped to put down a mathematical formulation for what it means to compute something. With an eye toward practical applications, they also outlined how it might be possible to build a device that could perform automatic computations. In a convergence of ideas, all of their proposals were found to be equivalent ways of computing.

The difference between the 20th century and 19th century approaches to computing was that the 20th century approach was based upon electricity, not mechanics. As such, switching between states was faster and more functions could be performed in a given amount of time. With the advent of World War II, both the Americans and English attempted to use computers to perform tasks such as computing navigation instructions for ships and trajectories of ballistics. In one of the most celebrated moments in computing history, an algorithm developed by Alan Turing cracked the German Enigma cipher, enabling the English to have detailed knowledge of German plans.

These first computers used vacuum tubes as the way to transition between states. In 1947, Bell Labs created the first transistor, a tiny electrical switch that relied on quantum mechanics. The development, and subsequent miniaturization of the transistor, decreased the power and size of the computer hardware, while at the same time increasing switching speed. The trend to make transistors smaller and more efficient has continued to push the entire electronics industry to greater and greater feats of engineering.

Following the hardware/software split of Babbage and Lovelace, some worked on computing hardware while others worked on developing algorithms. It was during 1940-1960 that many of the most important computer algorithms were developed, including the Monte Carlo method (John von Neumann, Stan Ulam and Nick Metropolis), the simplex method of linear programming (George Dantzig), the Householder matrix decomposition (Alston Householder), the quicksort method of sorting (Tony Hoare) and Fast Fourier Transform (James Cooley and John Tukey).

In the early history of computing, programmers were forced to speak the same language as the computers (1s and 0s). This was very tedious and soon was replaced with a somewhat more convenient type of programming language called *assembly*. Assembly languages allow programmers to avoid using 1s and 0s but programming was still very cryptic. In 1957, John Backus led a team at IBM to create FORTRAN, the first high-level programming language. It introduced plain English keywords into programming, *e.g., if*, *while*, *for* and *read*, that made code easier to write, read and share. Other high-level languages followed, such as LISP, COBOL, ALGOL and C. The major advantage of these high level languages was that they were not specific to particular computer hardware. As such, users could share programs more easily. The breakthrough made by Backus and his team was to create a *compiler*. The purpose of a *compiler* is to make the translation from the human text to an *executable*, the particular language of 1s and 0s that can be read by that particular computer. It is important to realize that while the text of the program is not unique, the compiler and the executable is unique to a computer's type of hardware. For example, if C code is compiled on a PC, it will run on the PC but not on a MAC. The actual code, however, could be written and tested on either machine.

1.2.3 MODERN COMPUTING

While there was a clear dividing line between the pre-history of computing and early computing, no one event signaled the modern era of computing. In fact, there were several events, most of which enabled non-computer users to begin using a computer. Below we discuss a few of the more recent advances which are part of modern computing.

In the early days of computing, only one program was allowed to run on the hardware at a time. With the advent of *operating systems*, computers gained the ability to multitask. An operating system is a master program that directs which programs can run and for how long. To be clear, the computer is still doing one thing at a time, but now it can switch back and forth between tasks. If the switching is fast enough, it can give the user the illusion that the computer is multitasking. This is the reason why you can surf the web, listen to music and work on your Matlab homework all at the same time. The three most common operating systems are Microsoft Windows, Mac OS and various favors of UNIX (including Linux).

The machine language of 1s and 0s is sometimes called the first generation of computing languages. Assembly and high-level languages are the second and third generation. The theme is that each generation built upon the generations that came before. As such, there are two new types of languages that can be called the fourth generation of programming. They are the *interpreted* and *object-oriented* languages. Interpreted languages can be contrasted with compiled languages. In a compiled language, an executable is created by the user that will only run on a particular computer. If the user wants to modify their code or move it to another type of computer, they must create a new executable. To overcome these limitations, the user could use an interpreted language. Here the computer has a program called an *interpreter* which will read the lines of text one by one and transform them into 1s and 0s on-the-fly. This has several advantages. First, the user now can modify code and run it without needing to compile the code. Second, it is possible to move code from one type of machine to another as long as both have the right interpreter. The disadvantage of an interpreter is that it has more overhead in terms of memory and operations, and so an interpreted program will run much slower than a compiled program. Examples of interpreted languages are Perl, Python and Matlab. Object-oriented languages focus on allowing data to be handled in a more natural way. It is often the case that what we call data is not simply one number, but it is a series of *objects*. These objects may be numbers, text or other values. As a medical example, imagine that we wish to create a database of patient records. In an object oriented language we would create a data structure (called a *class*) which contains all of the information for a generic patient. Then for each individual patient we would create a particular *instance* of the class with the appropriate *attributes*. Although object-oriented programming can be traced back to the 1960s, the ideas did not propagate out of the database and artificial intelligence communities until the 1980s. Now, nearly every programming language has some support for object oriented programming.

The advent of the personal computer occurred in the 1980s, followed quickly afterward by the laptop revolution. Prior to that time, most computer users required access to a shared computer mainframe. You would submit your job into a queue and then wait until it was your turn. Unfortunately, you had no way to test your program on your own. It might be the case that you made a very small error in your program and had to wait for the output to find out where you went wrong. And then after you had corrected your error, you would still need to resubmit your job to the queue. The personal computer changed all of that, but it also had another side effect. Personal computing opened up an enormous business niche because, where before there were only a few computer users at university and government labs, now nearly everyone owned a computer.

The last modern change to computing has been the creation of networks of computers. The first and widest reaching is also the most obvious - the internet. One way to think of the internet is as a giant computer that connects people through the computers they use. It has given rise to ideas such as *cloud computing* and *dynamic web pages*. Another version of network computing is *parallel computing* and *distributed computing*. Both are used to solve very big or very hard problems with many computers at once. Although there is not a clear dividing line between the two, usually parallel computing is when processors communicate on some dedicated network, while distributed processing uses a non-dedicated network, such as the internet.

1.3 A HISTORY OF MATLAB

In the previous section, high-level computer languages and algorithm development were discussed. One problem was that many algorithms were written but all in different languages and for different computers. The US government recognized this problem and so Argonne National Labs took up the task of writing a standard set of numerical algorithms that would be freely available (and still are at www.netlib.org). They were the BLAS (Basic Linear Algebra Subroutines), LINPACK (Linear Algebra Subroutines for Vector-Matrix Operations), EISPACK (To compute eigenvalues and eigenvectors) and other general purpose libraries of algorithms. Cleve Moler was one of the programmers who helped write the LINPACK and EISPACK libraries. He later went on to teach at the University of New Mexico. While there, he realized that the Argonne libraries were cumbersome for those not already firmly grounded in scientific computing. Moler recognized in 1983 that the libraries could reach a much wider audience if a nicer interface was written. And so he created Matlab, the "MATrix LABoratory" in 1983 using FORTRAN. The first version was in fact used by Moler to teach undergraduates the basics of programming. Over the next two years, Moler visited a number of universities. At each university he would give a talk and leave a copy of Matlab installed on the university computer system. The new program quickly became a sort of "cult" phenomenon on campuses around the US.

In 1984, Jack Little and Steve Bangert were so impressed with the new programming environment that they rewrote many of the Matlab functions in C allowing it to run on the new

IBM PC. They also added some built in functions that allowed Matlab to have its own native programming language, better graphics and a Matlab interpreter. Initially, this version was free, but Moler, Little and Bangert soon banded together to formed Mathworks. At the end of 1984 they sold their first product to Professor Nick Trefethen at MIT.

In the intervening years, Matlab has become the standard for engineering computing, with millions of users around the world. While the original version of Matlab had only 80 built-in functions, it now has thousands, written not only by Mathworks employees, but by users. Mathworks itself has undergone a transformation from selling the standard Matlab distribution, to a variety of toolboxes that are of interest to particular disciplines. It also has the capability (like Maple and Mathematica) to perform symbolic logic and contains a system simulation program called Simulink.

1.4 WHY MATLAB?

At various times during my career I have been asked why I have chosen Matlab as a programming language. And throughout my career I have given various answers ranging from "it is what I know best" to "it does what I want". What is important is that both answers are probably not your answers right now, and they may never be. Below are some reasons why you should take learning Matlab seriously. First, engineers often need to perform tedious calculation tasks over and over again. The calculations might range from something simple, *e.g.,* taking the average of a list of numbers, to something more complex, *e.g.,* simulating how an ecosystem will react to the introduction of a non-native species. Here most any type of computing language can help. But, some languages are easier to learn and others are more flexible. Unfortunately it seems to be the trend that the most powerful languages are also the most difficult to learn. Matlab strikes a good balance between being easily learned and flexible. For example, Matlab is an interpreted language (see the advantages above) but also can be compiled (see the advantages above). Second, engineers often make figures to represent large quantities of data. Matlab is one of the few programming languages that has graphics capabilities built-in. It is important to say a word here about Excel. It may be very tempting to default to Excel. After all, it can perform calculations and does have graphics capabilities. The problem is that unless you are willing to learn Visual Basic and write macros, Excel is very limited in its computational abilities. Furthermore, Visual Basic tends to be more difficult to learn than Matlab. Matlab also wins out in that it was designed for engineers, whereas Excel was not. As such, Matlab is a common programming language spoken by nearly all engineers, regardless of their training. Third, because there is a large community of Matlab users, there are many great Matlab books and online resources. It is often the case that a problem you must solve has already been solved by another Matlab user and they have posted their code online at Matlab Central (http://www.mathworks.com/matlabcentral/).

There is no claim that Matlab is the "best". If you are looking to write a database, perform high-end scientific computing on huge supercomputers, or do any type of natural language

processing, some other language may be what you want. But as a language that is easy to learn, designed for engineers, and the common computing language of engineering, you can't do better than Matlab.

CHAPTER 2

Matlab Programming Environment

2.1 INTRODUCTION

The purpose of this chapter is to introduce you to the basic functions and environment of Matlab. Most every other capability within Matlab is built from these basic concepts, so it is important to have a good understanding of what follows.

2.2 THE MATLAB ENVIRONMENT

Although Matlab goes by one name, it is in reality a full service computational engine composed of many integrated parts. There is the core of the program that contains the most basic operations. These functions are often very deeply compiled and cannot be changed. There are graphical libraries that allow users to create plots and images as well as graphical user interfaces for interacting with programs. Users can create their own functions or take advantage of the additional libraries contained in toolboxes. Luckily, Matlab provides an environment, shown in Figure 2.1, that ties everything together.

At the very top of the window there are a series of toolbars. It is here that you may obtain useful information, such as the built-in help. Below the toolbar is a large window that is mostly blank. It contains what is known as a command prompt, designated by >>. It is to the command prompt that you will type commands that will be executed by Matlab. In the top right there is a Workspace window that lists all of the variables that you currently have in memory. In the bottom right there is a Command History window that lists commands that have been issued to the command line. On the left side is a list of all of the files in your current working directory.

Matlab has many tricks that involve the graphical interface. Versions of Matlab, however, do vary, and Matlab is not the same on every computing architecture. To allow the most flexibility, we will use the common element of all Matlab environments; the command line.

2.3 THE DIARY COMMAND

In the following sections, you will be asked to enter commands on the command line and then observe the results. As a portion of your assignment (and other assignments to follow), you will

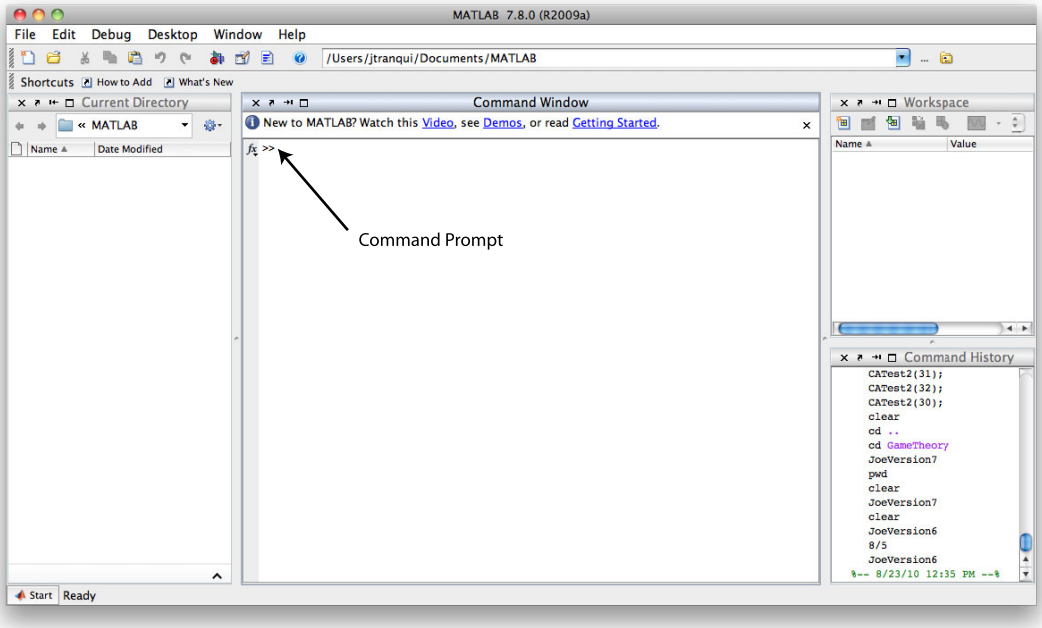

Figure 2.1: The Matlab Command Window

demonstrate that you have followed the tutorial below by keeping a record of your commands. Matlab has a command called *diary* that will create a text file containing a record of your commands. Throughout this text, italics are used to indicate a built-in Matlab function.

To begin, create a folder on your desktop and then navigate your current path (at the top of the toolbar) to the folder. Next, enter the following at the command prompt

```
>> diary Problem1.diary
```

This command will record all subsequent commands into a file called "Problem1.diary". As outlined in a separate document, all diary files should end with ".diary". After you issue the command, you should see the file in the Current Directory window. When you wish to turn off the diary command you can enter

```
>> diary off
```

You should wait to turn off the diary until after you have finished the exercises below.

2.4 AN INTRODUCTION TO SCALARS

In the following section, we will go on a self-guided tour of some basic Matlab functions. To begin, type the following at the command prompt.

```
>> r = 5;
```

Notice that the variable *r* now appears in the workspace window. Now type in the following command which is similar to the above but without the semicolon

```
>> a = 25
```

You should see that the Matlab command line echos back what you typed, verifying that the variable *a* is set to a value of 25. This is your first introduction to a feature of Matlab. Adding a semicolon to a line will suppress any output to the command line, while no semicolon will print out the value of the variable. Next type the following on the command line

```
>> whos
```

The *whos* command will print out all of the variables in your workspace to the command line. You should notice that there is additional information given. For example, you will see in the row corresponding to the variable *a*

```
Name        Size              Bytes  Class     Attributes
a           1x1                   8  double
```

Here we can see that *a* is of size 1×1, meaning that it is a scalar. Later we will discuss variables of other sizes, *e.g.,* vectors and matrices. We also can see that *a* is stored in 8 bytes as a double class. There are in fact a number of different types of data which can be stored in Matlab. For example, to learn more about the data type *single* you can type the following on the command line.

```
>> help single
```

First, you should note that we used a new command, *help*. This is very useful when you know the name of a command but do not know how to use it. By typing "help single", you learn that the command *single* will convert a variable to type "single". You will also see in the second line how to use the command, *e.g.,* Y = SINGLE(X), along with what options are available. Try typing

```
>> b = single(a)
```

You should also note that Matlab is sensitive to case

```
>> r = 3;
>> R = 11;
```

will create two different variables, *r* and *R*.

There are a few things to note about this command. First, Matlab's *help* uses all capitals to designate functions, *e.g.,* Y=SINGLE(X), but they are not entered this way on the command line. Second, anything to the right of the equal sign in parentheses is consider an input. Anything to the left of the equal sign is considered an output. There are some functions that take more than one

input and can send out multiple outputs. Lastly, if you now type 'whos' on the command line, you will see that 'a' is a double (8 bytes), but 'b' is a single (4 bytes).

As an exercise, investigate the *sin* and *log* commands. Does sin accept inputs in degrees or radians? Does *log* take the natural log or the log base ten? Note that both commands have some related functions listed under "See also".

2.5 BASIC ARITHMETIC

Matlab has all of the functions of your calculator (and many more as you will see in future chapters). In this section, we will investigate some of the basic functions that are most likely present on your calculator. We can add two numbers

```
>> 4+5
```

or multiply numbers

```
>> 5*6
```

and Matlab supports subtraction

```
>> 4-5
```

and division

```
>> 5/6
```

2.5.1 PRIORITY OF COMMANDS

One problem with basic arithmetic is that it is not always clear what order the operations should be performed in. For example, try entering

```
>> 4+5*6
```

It should be clear from the result that the multiplication was performed first and then the addition. This is because in Matlab multiplication has a higher precedence than addition. But what if we wanted to perform the addition first and then the multiplication? In this case, we can use parentheses to make the order clear

```
>> (4+5)*6
```

Although Matlab has a built in ordering of preference, it is generally helpful for debugging to use parenthesis to make the intended ordering more clear.

2.5.2 REISSUING PREVIOUS COMMANDS

There are many times in Matlab when you may wish to either repeat a command over again, or enter a command that is similar to one previously issued. Matlab has three nice features that can save you some typing time. First, in the example above, you may wish to reissue the command $4 + 5 * 6$. If you double click on this command in the Command History window (lower right), it will copy and

paste the command and execute it for you on the command line. Although this may not seem like a great savings in time, in the future you may have issued so many commands that the one you want is not easy to find. The second helpful feature is the up and down arrows. Try using the up arrow and you will find that Matlab will move through previous commands. If you hit the enter key, the current command will be executed again. Likewise, you can use the down arrow key if you have moved past a command. The third feature is known as completion. If you start typing a command and then hit the up arrow, it will find the last time you typed a command that began that way. As an example, in an earlier exercise you set the variable r to a value of 5. If you wish to change that value, you could use the arrow key. But, a faster way is to begin entering the command

```
>> r =
```

and then hit the up arrow. You could then change r to 4 and hit enter to update the value.

2.5.3 BUILT-IN CONSTANTS

Let us assume for the moment that r is actually the radius of a circle and we would like to find the circumference, *e.g.,* $2\pi r$. In Matlab, we could define a constant with the value of π and then perform the multiplication. Matlab, however, already has built in a number of predefined constants, such as *pi* for π. On the command line enter

```
>> Circ = 2*pi*r;
```

Because the semicolon was used, Matlab did not print out the result to the command line. It did, however, store the value in the variable $Circ$. To see the value, simply type 'Circ' on the command line and press enter. If you wanted to find the area, *e.g.,* πr^2, try typing

```
>> Area = pi*r ^ 2
```

Note that the symbol $\wedge$ is for any root. So if you wanted to find the value of $\sqrt[3]{6.4}$, you would enter

```
>> 6.4 ^ (1/3)
```

Note that parentheses were used to indicate the desired order of precedence. The symbol $\wedge$ can also be useful for specifying large and small numbers, for example

```
>> 10 ^ -6
>> 10 ^ 8
```

A second very common constant is the imaginary number i.

```
>> c = 4 + 3i
```

In this example, the variable c is a complex number. You can verify this by issuing the *whos* command. You can also view the help for the following constants within Matlab, *inf*, *nan*.

2.5.4 FINDING UNKNOWN COMMANDS

Because Matlab commands are not on buttons you can see (like on your calculator), it can sometimes be difficult to know the name of what you want. To this purpose, Matlab has a function called *lookfor* that will search all of the help files for your entry. For example, if you know that you would like to find the hyperbolic tangent function but don't know its name in Matlab, try typing

```
>> lookfor tangent
```

You will get a list of the functions that use the word "tangent" in their help files. Although the list may be long, you can scan it and find that *tanh* is the function you want.

2.6 THE LOGISTIC EQUATION

One of the more famous equations in mathematics is the logistic equation. The connection to biomedical research is that the logistic equation was created to study the growth of a population of reproducing species. The catch is that most environments can only sustain so many individuals (usually because of some finite resource like food). So, after an initial explosion in growth, the population size will settle down to some reasonable number. This equation is sometimes used as a simple model of bacterial infection. The logistic equation can be written in the form of a difference equation

$$z_{n+1} = r z_n [1 - z_n] \tag{2.1}$$

where z_n is the current value of z, z_{n+1} is the next value of z and r is a constant. It is important to realize that the logistic equation is scaled, *e.g.,* normalized, so that the maximum population is 1. Using what you already know, set $r = 1.5$ and the initial population value to $z = 0.5$, *e.g.,* half of the maximum. Then issue the following command

```
>> z = r*z*(1-z)
```

Because the variable z appears on the left and right side of the equation, it is important to understand how Matlab processes a command of this type. Remember that anything to the right is an input and anything to the left is an output. So, the old value of z is used on the right side, *e.g.,* z_n, but then that value of z is overwritten in memory to form the output, *e.g.,* z_{n+1}. The reason for writing out commands in this way is that we can issue it once and then use the up arrow to keep issuing it again and again. What is happening here is that you are using the current value to find the next value. You should try this to see what happens to the value of z (you should see it decrease, headed for 0.0). We will return to this example in future chapters to show how using some simple programming we can make this type of iterative operation much more efficient.

To give a bit of a flavor for some strange things that can happen using the Logistic equation, try repeating the above (remember to reset the initial value of z to 0.5 for each trial) but with the following values for r

r = 3.2
r = 3.45
r = 3.6

At this point, you should turn off the diary command ("diary off"). Then using a text editor, you can open the diary file. At the end of the text file, you must write a brief description of what you found for the different values of r. What would this mean for a real population of bacteria?

2.7 CLEARING VARIABLES AND QUITTING MATLAB

A useful (but also very dangerous) command is *clear*. If you issue the following to the command line

```
>> clear
```

everything in Matlab's memory will be erased. There are times when this is very useful, but it should be used carefully. An alternative is to only clear specific variables, for example

```
>> a = 34.5;
>> b = 27.4;
>> whos
>> clear a
>> whos
```

Note that in this example we only cleared the variable a.

Matlab is very simple to quit. Simply type *quit* on the command line. It is important to know that when Matlab is quit, all variables that were created will be lost.

2.8 EXAMPLES

Each chapter will contain a series of problems that will help you strengthen your understanding of the material presented. For all problems below, you should begin a diary entry, complete the problem and then end the diary entry.

1. As problem 1 was following along with the chapter, your first exercise is simply to rename your diary file "Problem1.diary".

2. Use the *lookfor* command to find the name of Matlab's function for taking the exponential of a number. Then use the *help* function to find out how it works. Then demonstrate that you can find the value of $e^{3.4}$. Do the same for the operation of a factorial and then report the value for 18!

3. Create two complex variables $g = 25 + 2i$ and $h = 1 - 4i$ and then multiply them in Matlab and store the result in a variable k. Demonstrate that you can use the *real* and *imag* commands in Matlab to find the real and complex parts of k.

4. When a fluid is placed in a thin tube, it will rise up that tube, against gravity, due to capillary forces (what causes dyed water to be sucked up the xylem of a celery stalk). It can be calculated

analytically how high a liquid would rise (h) in a tube due to the capillary effect

$$h = \frac{2\sigma cos(\phi)}{r\rho g} \tag{2.2}$$

where $\sigma = 0.0728 J/m^2$ is the surface tension, $\phi = 0.35$ radians is the contact angle, $r = 0.001m$ is the tube radius, $\rho = 1000 kg/m^3$ is the fluid density and $g = 9.8 m/s^2$ is the force of gravity. Using these values, compute the rise of the fluid. First, declare all of the variables and then find the answer using an equation with only variable names and the constant 2. Then change $r = 0.002m$ and recompute the rise.

CHAPTER 3

Vectors

3.1 INTRODUCTION

In this chapter and the next, we will discuss the core of what makes Matlab special - the ability to efficiently operate on vectors and matrices. Although Matlab has come a long way since its humble beginnings, vectors and matrices remain the core of nearly everything that happens in Matlab.

You will want to turn on the *diary*, as your first exercise is to follow along with the tutorial below.

3.2 VECTORS IN MATLAB

Whereas previous chapters considered single numbers, called scalars, vectors are a fancy name for a list. It could be a to-do list, a list of fruits, or for most scientific and engineering applications, some sort of numerical data.

3.2.1 CREATING VECTORS IN MATLAB

```
>> a = [1 1 2 3 5 8 13];
>> whos
```

This command will simply list the number of elements in *a* and their type.

We can perform operations on the vector *a*. Matlab has many built-in functions. Some work on scalars, as in the previous chapter, but others work on vectors. For example,

```
>> b = sum(a);
```

will sum up all of the values of *a* and report the outcome in *b*. In this example, we turned a vector, *a*, into a scalar, *b*. A similar command, *prod*, will return the product of all values in a vector. Another useful function is the *length* command which gives the number of elements in a vector

```
>> NumElements = length(a)
```

Matlab also can perform operations that transform one vector into another vector. For example, in the previous chapter, you learned that the hyperbolic tangent function is *tanh*. If you enter on the command line

```
>> b = tanh(a)
```

b will be a vector that contains the hyperbolic tangent of each entry of *a*.

3.2.2 CREATING REGULAR VECTORS

You could simply enter in each value of a vector, as shown above. Sometimes, however, you may want to create a vector with a more regular pattern. For example,

```
>> c = 1:21;
```

will create a vector, *c*, that has 21 entries, where each entry is one bigger than the last, *e.g.,* a list from 1 to 21. You can verify this by typing *c* on the command line and pressing enter (without a semi-colon). What is more, you can use another feature of the colon notion to skip values. For example,

```
>> d = 1:2:21
```

will generate a vector, *d*, that starts at 1 and then skips every other number until reaching 21. You should verify this by entering *d* on the command line. The colon notion is always *start*:*skip*:*stop*, and can be very useful for creating patterned vectors. For example, you might want to have all numbers that are multiples of 10 (skip) between 50 (start) and 200 (stop)

```
>> e = 50:10:200;
```

3.2.3 SPECIAL VECTORS AND MEMORY ALLOCATION

Many other programming languages, especially those that are compiled, require that the user specify up front exactly what variables will be used throughout the program. What is required is memory to be reserved for storage. The big advantage of allocating memory up front is that it is easier to write (and read) to a vector if it is all in the same place in the computer's memory. As an interpreted language, Matlab does not require you to specify variables up front. This can be a very nice feature, but it can also cause Matlab to operate slowly at times. To overcome this limitation, Matlab will allow the user to allocate space for a vector. The most usual way to do so is using the *zeros* command

```
>> f = zeros(25,1);
```

that will create a vector, *f*, with zeros in all 25 locations. A second useful feature of Matlab is the *ones* function

```
>> g = ones(45,1);
```

that will create a vector of length 45 where every entry is a 1. The *ones* command is useful for the following

```
>> h = 0.255*ones(18,1)
```

In your diary, explain the output of the command above.

3.3 VECTOR INDICES

You can think of a vector as a series of bins, one for each element. What makes vectors powerful is that it is not only the value that is saved but also a type of address, called an index. For example, if we would like to get the fourth entry of *a*, *e.g.,* the value 3, we can get only that value by typing

```
>> a(4)
```

What is more, Matlab uses the same colon notion explained above to index vectors. For example,

```
>> a(3:5)
```

will return the values of *a* at indices 3,4 and 5. So the output will be a subset of the entire array, here the values 2, 3 and 5. Also as above, the colon notion can be used to skip values. For example,

```
>> a(1:2:7)
```

will start with the first value and then skip every other value until getting to the 7th value. So the output will be 1, 2, 5, and 13. This type of regular skipping can be very useful in large vectors, for example, if you wanted only every 10th value.

There is an additional function in Matlab that can be very useful.

```
>> a(end)
```

will return the very last value of the vector. The advantage here is that you can use *end* with the colon notion

```
>> a(1:2:end)
>> a(1:2:end-1)
```

Explain in your diary why the two commands above give different answers. A hint is that *end-1* is the second to last entry of the vector *a*.

When dealing with very large vectors, it can be helpful to use the *find* command in Matlab. For example,

```
>> j = find(a==8)
```

In this statement, we are using *find* to search the vector *a* for the value 8. If a value of 8 is found, *find* will return the index where the value was found. In this case, j=6, because the value 8 is in the 6th location of the vector *a*. In the future, you will learn about conditional statements which will allow you to find entries that are, for example, greater than some number.

One last example will demonstrate the power of indexing in Matlab. Imagine that you would like to enter some values into the vector f, which was created above. More specifically you would like to place the values 3, 25.7 and 91.6 in the 4th, 9th and 25th locations of f. You could enter the following commands

```
>> indices = [4 9 25];
>> values = [3 25.7 91.6];
>> f(indices) = values;
>> f
```

3.4 STRINGS AS VECTORS

In Matlab, a vector can be composed of any type of data. For example, vectors could have complex values, *e.g.,* contain real and imaginary parts, integers or even characters. In Matlab, a vector of characters is called a *string*.

```
>> s = 'Matlab is Great!';
```

s is a vector of characters, or a string. Note that strings in Matlab are enclosed in single quotes. You can verify that this is indeed stored in the same way as a usual vector by entering

```
>> s(3)
>> s(10:end)
```

Strings stored as a vector of characters can be very important in cases such as creating file names. Let us assume that you have a series of 100 files that contain data on patients 1 through 200. A common operation might be to open up each file to search for some important characteristic, *e.g.,* blood pressure. Although you will learn how to perform the details of a this task in later chapters, a first step is generating each unique filename. Rather than entering all 200 in by hand, we can specify a variable

```
>> x = 1:2:200;
>> ThisPatient = x(29);
>> PatientFilename = ['Patient' num2str(ThisPatient) 'Data.dat'];
```

You will want to begin by understanding the first two lines. First, a vector is created that ranges from 1 to 200, skipping every other number. Then we will pick off the number in the 29th spot and store it in a scalar called $ThisPatient$. In the last line, we build up a vector of characters by starting with the string "Patient". The *num2str* command is used because we can not mix different data types within a single vector. So, the numerical value of $ThisPatient$ must be converted to a string. You should see the help file for *num2str* for more information. Lastly, we add "Data.dat" onto the end of the file. You should verify that your commands have worked by typing in

```
>> PatientFilename
```

3.5 SAVING YOUR WORKSPACE

There are times when it is helpful to save all of the data in Matlab's memory. For example, there are some large projects where you may want to quickly be able to pick up where you left off.

```
>> save MyFirstMatlabWorkspace
```

The *save* command will save all of the variables in your workspace to a file called "MyFirstMatlab-Workspace.mat" in your current working directory. The ".mat" file extension indicates a Matlab data file. The advantage of a Matlab data file is that it will work on any version of Matlab - PC, MAC or UNIX. To return the variable to the Matlab memory

```
>> clear
>> whos
>> load MyFirstMatlabWorkspace
```

Note that the *load* command would work even if you had quit Matlab and then wanted to reload the variables in your workspace. You may have noticed that the *save* command saves your entire workspace. If you are only interested in the two vectors a and f, you can enter

```
>> save MySecondMatlabWorkspace a f
```

You will be using the *save* in some of your homework exercises.

3.6 GRAPHICAL REPRESENTATION OF VECTORS

With short vectors, such as those created so far, it is possible to view our data on the command line (by simply leaving off a semi-colon). But when vectors become long, it can be very useful to take advantage of the graphic capabilities of Matlab. Consider the following commands.

```
>> x = 0:0.01:10;
>> y = sin(x);
>> length(y)
```

The first command creates a vector, x that starts at 0 and ends at 10, but in increments of 0.01, creating a vector with 1001 entries. The second line creates a vector, y that applies the sine function to the vector x. Therefore, y will also have 1001 entries. To plot the two vectors against one another

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

It is also possible to plot a vector with specifying an x-axis vector

```
>> plot(y)
```

where it is assumed that the x-axis is incremented by one for each value of y. In future chapters, we will focus more on how to fine tune Matlab's graphics.

3.6.1 POLYNOMIALS

Matlab stores a number of useful structures as vectors. For example, polynomials can be manipulated as a vector of coefficients. The polynomial

$$x^3 + 2x^2 - 4x + 8 \tag{3.1}$$

can be represented in Matlab as

```
>> u = poly([1 2 -4 8]);
```

which will give the coefficients of the polynomial with the roots 1, 2, -4, and 8. Note that a third order polynomial must contain four entries. A zero entry may also be used as a place holder. So

```
>> w = poly([-1 0 -1 4]);
```

represents

$$-x^3 - x + 4 \tag{3.2}$$

Finding the zeros crossings, *e.g.*, *roots*, of a polynomial is useful in a number of situations. For second order equations, it is simple to use the quadratic equation. For higher order polynomials, it becomes practical to use a numerical method, *e.g.*, Newton's Method. Matlab has a built-in root finder called *roots*.

```
>> roots(u)
>> roots(w)
```

Note how, in the last command, Matlab will report complex numbers.

3.7 EXERCISES

1. Turn in the diary of your commands for this chapter. You must also include the two ".mat" files you created in Section 3.5 and answer the questions in Sections 3.2.3 and 3.3.

2. Sine waves are used in many life science and engineering applications to drive everything from the varying magnetic fields of MRI machines to the supply of nutrients to tissue being grown in a bioreactor. For this problem, you must show your commands in a diary. The general equation for a sine wave is

$$y = A \cdot sin\,[2\pi\omega t + \phi] \tag{3.3}$$

   where A is the amplitude, ω is the frequency in Hertz (1/seconds) and ϕ is an angle in radians. First create a vector for time (t) that starts at 0, ends at 5 and skips in increments of 0.001. Next, define $A = 1$, $\omega = 2Hz$ and $\phi = \pi$. Using the parameters and time vector, create a vector y and then plot t against y.

   In the next part of this problem, you will create a second sine wave (in a variable z)

$$z = A \cdot sin\,[2\pi\omega t + \phi] \tag{3.4}$$

   with $A = 0.5$, $\omega = 1Hz$ and $\phi = \pi/2$. Note that you can use the same time vector to create z. You will next plot z against t, but on the same figure as your previous sine wave. To plot two data sets on the same figure, you must use the *hold on* command. In general, plotting two data sets will be achieved by

```
>> plot(t,y)
>> hold on
>> plot(t,z)
```

3. One important use for computing is to test out algorithms that will find some important characteristic of a data set, *e.g.,* frequency spectrum, time to reach a maximum. The problem with biological data is that it often contains noise. Therefore, many algorithms designed to be used on biological data must work even in the presence of noise. Matlab has a built in command, *rand*, that can be used to generate random numbers. For example,

```
>> r = rand(1000,1);
```

will create 1000 random numbers between 0 and 1. Use the help function (hint: look at the examples in the help file) to create 2000 random numbers that range between -1 and 2 and store them in a variable, *RandomNumbers*. It may be helpful to plot *RandomNumbers* to check that you have created what you intended. Use the *save* command to save the numbers into a file, "RandomNumbers.mat".

4. A Holter Monitor is a device that is used to continuously record ECG, even when the patient may be at home. It is often the case that a patient will allow the device to record for some period of time and then upload the data (using the internet or a phone line) back to the physician. Create a character string, called *FirstName*, with your first name. Create a character string, called *LastName*, with your last name. Then create a filename (another string) with the following format

   `LastName,FirstName-HolterMonitor6.2.2011.dat`

   In this way, the physician will know exactly what data he/she is looking at. Enter your commands for creating the filename in a diary file.

CHAPTER 4

Matrices

4.1 INTRODUCTION

While vectors are useful for storing lists, matrices are good for storing lists of lists. As such, you can think of a matrix as a group of vectors. Although the data can be of any type, *e.g.*, doubles, characters, integers, all of the vectors that make up a matrix must have the same length.

4.2 CREATING A MATRIX AND INDEXING

Creating a matrix in Matlab is not all that different from creating a vector. For example, to create the matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 4 & 6 & 8 & 6 \\ 7 & 0 & 3 & 12 & 7 \\ 2 & 5 & 0 & 4 & 8 \end{bmatrix}$$

you would enter

```
>> A = [1 2 3 4 5; 0 4 6 8 6; 7 0 3 12 7; 2 5 0 4 8];
>> whos
```

You should see that matrix *A* has the dimensions 4×5 and consists of doubles (Matlab's default format). Rows are separated by semi-colons. It is also convention that matrices in Matlab use capital letters, while scalars and vectors are lower case. This is not necessary but can help distinguish between the two types of data.

There are some occasions when you may want to know the dimension of a matrix. Similar to the *length* command for vectors, you can use the *size* command for matrices

```
>> [NumRows NumCol] = size(A);
```

where the variables *NumRows* and *NumCol* will contain the number of rows and columns of the matrix.

4.2.1 SIMPLIFIED METHODS OF CREATING MATRICES

There are several additional ways that a matrix could be created.

```
>> B = zeros(5,6);
```

will create a matrix with 5 rows and 6 columns where all the entries are zeros. This is a generalization of the *zeros* command and is a good way to allocate memory for a matrix. Similarly,

```
>> C = ones(4,3);
```

will create a 4×3 matrix of ones. Again, as in the vector version of the *ones* command, we can scale the entire matrix.

```
>> C = 23*ones(4,3);
```

A very common matrix to create is one that has values only along a diagonal, and zeros everywhere else. For example,

```
>> diagvector = ones(7,1);
>> D = diag(diagvector)
```

The first command creates a vector of ones that is of length 7. Then this vector is placed on the diagonal of the matrix *D*. In using the *diag* command, Matlab automatically fills in the necessary zeros. In fact, the example above is a matrix that is used very often in linear algebra called the Identity Matrix. Matlab has a command, *eye*, which creates any N ×N identity matrix. The *diag* function, however, can be used to do much more. Try to interpret the following sequence of commands

```
>> DiagVector = [1 2 3 4 5 6];
>> UpDiagVector = [7 8 9 10 11];
>> DownDiagVector = [12 13 14 15 16];
>> E = diag(DiagVector) + diag(UpDiagVector,1) + diag(DownDiagVector,-1)
```

Notice how the *diag* command can be used to put values above (indicated with a 1) or below (indicated with a -1) the diagonal, but that the vector used must be the proper length. For example, in a 6×6 matrix, you would place a vector of length 5 above or below the diagonal. If at some later time you wanted to add another diagonal 2 off the main diagonal,

```
>> E = E + diag([17 18 19 20], 2)
```

There are also some occasions where you may want to create a random matrix.

```
>> F = rand(5,6);
```

creates a 5×6 matrix with random values that range between 0 and 1.

4.2.2 SPARSE MATRICES

So far we have created matrices that have specified every value in the matrix, even if they have a value of zero. But zeros take up space in memory. Matlab has a way to compress a matrix in such a way that any zeros are not stored. This is especially useful for very large matrices that contain mostly zeros. In scientific computing these type of matrices are known as *sparse*.

```
>> G = sparse(E);
>> whos
```

You should notice that the matrix *G* is now a sparse matrix and takes up fewer bytes in memory than the original matrix *E*. In small matrices the savings is usually not worth it. But, try issuing the following commands and observe the difference in memory used by matrices *H* and *J*.

```
>> H = diag(ones(500,1));
>> J = sparse(H);
>> whos
```

4.3 INDEXING A MATRIX

Values can be read from a matrix in much the same way as they are read from a vector. For example,

```
>> E(1,2)
```

will retrieve the value of 7 in the first row and second column. Try getting the value of 15 in the fifth row and fifth column. You can also get entire columns or rows from a matrix using the colon notation.

```
>> E(2,:)
```

will get the entire second row of E.

```
>> ColThree = E(:,3)
```

will get the entire third column of E and store it in a vector called $ColThree$. What is more, we can get any submatrix of a larger matrix.

```
>> SubMatrixOfE = E(2:5,3:6)
```

will get from row 2 to row 6 between columns 3 and 6. Note that this submatrix will be a 4×4 matrix of its own. How might you get the bottom right 3×3 submatrix? Note that you can even use the *end* command introduced in the previous chapter.

4.3.1 HIGHER DIMENSIONAL MATRICES

Although Matlab is designed to handle two-dimensional matrices, there is some support for higher dimensional matrices.

```
>> P = zeros(2,3,5);
>> Q = rand(3,4,5);
>> whos
```

You can read from higher dimensional matrices in the same way as other matrices

```
>> R = Q(1,2:3,2:4)
>> whos
```

4.4 SIMPLE MATRIX ROUTINES

There are a number of matrix routines that can be useful in real problems. First, there are occasions where we may want to transform a matrix into a vector using the *reshape* command

```
>> k = reshape(E,36,1)
```

Here the vector k contains the same values as the matrix E, where each of the columns have been stacked on top of one another. But we can use reshape to do some other interesting things as well.

```
>> L = reshape(E,9,4)
```

changes the 6×6 E matrix into a 9×4 matrix, again wrapping through the columns.

Another command that can be useful is the *squeeze* command. You may have noticed that the R matrix above is a 1×2×3 which is really a 2×3 matrix. To have Matlab compress any dimensions of 1

```
>> S = squeeze(R)
>> whos
```

4.5 VISUALIZING A MATRIX

4.5.1 SPY

It is often useful to be able to visualize the entries of a matrix. For example,

```
>> A = diag(ones(100,1))+diag(-3*ones(99,1),1)+diag(25*ones(99,1),-1);
```

First make sure that you understand how this matrix was built up using *diag* and *ones* commands. In some applications it is useful to know where any non-zero entries exist in the matrix.

```
>> spy(A)
```

will bring up a graph showing the location of all non-zero entries. At the bottom of the graph you will also find the number of non-zero entries in the matrix. In the toolbar (at the top of the graph) you will see a small magnifying glass with a '+' sign. Click on this symbol and select an area that contains some dots. You can use this zoom function on any type of plot within Matlab. To zoom back to the original view simply double click anywhere on the plot. Note that next to the positive magnifying glass is a negative magnifying class which can be used to zoom out.

4.5.2 IMAGESC AND PRINT

There are many situations where a matrix will be used to store data. In these situations, we might want to view all of the data at once.

```
>> B = rand(100,100);
>> B(25:50,50:75) = 1.5;
>> imagesc(B)
>> colorbar
```

creates a random 100×100 matrix (with values ranging from 0 to 1, representing some data we have collected from an experiment). The second command fills a block of the matrix with the value 1.5. The next two commands create a color plot of the values in B and then adds a color bar.

There are times when it is helpful to create a jpeg of an image so it can be imported into other applications, *e.g.*, PowerPoint, Word. The Matlab *print* command can be used to create an external file

```
>> print('-djpeg','MyFirstImageFile.jpeg');
```

If you look at the help file for *print* you will notice that the first entry specifies the file type (in this case a jpeg). You should notice that there are many other types of image files that you could create. The second entry specifies the name of the file. You should try to open "MyFirstImageFile.jpeg" to be sure it has been created properly.

4.6 MORE COMPLEX DATA STRUCTURES

Standard matrices and vectors must contain all of the same type of data. For example, we may have an entire vector of characters or an entire matrix of integers. Although there are cases where we want to mix types of data, matrices and vectors will not help us. But what if we want to have data for a patient that contains a combination of names and medications (strings), heart rate and blood pressure (doubles) and number of times a nurse has checked in today (integer). Matlab does have two ways to handle this sort of problem.

4.6.1 STRUCTURES

The idea of a structure is that data types can be mixed, but then referenced easily. Enter the following commands

```
>> P(1).Name = 'John Doe';
>> P(1).HeartRate = 70.5;
>> P(1).Bloodpressure = [120 80];
>> P(1).Medication = 'Penicillin';
>> P(1).TimesChecked = int16(4);
>> who
>> P
```

Above we have created a data structure, P, for the patient "John Doe". You will note that P is now of type *struct* and that when you type P on the command line it will tell you which data are in P. To retrieve a value from P

```
>> P(1).Name
>> P(1).Bloodpressure
```

You may wonder why we needed to reference the first value of P, *e.g.*, P(1). The reason is that now we can easily enter another patient

```
>> P(2).Name = 'Jane Doe';
>> P(2).HeartRate = 91.3;
>> P(2).Bloodpressure = [150 100];
>> P(2).Medication = 'Coffee';
>> P(2).TimesChecked = int16(2);
>> who
>> P
```

It should be noted that you can have fields within fields too

```
>> P(2).Family.Father = 'Jerry Doe';
>> P(2).Family.Mother = 'Julia Doe';
>> P
>> P.Family
```

4.6.2 CELL ARRAYS

While a structure must be referenced by a known name, a cell is simply a matrix of other data structures, where the data structures do not need to match.

```
>> T = {rand(3,3), 5.3,'BMEG 220'; int16(27),~[24.5 37.8],'Matlab'};
>> who
>> T
```

Here T is a matrix, but each element can take the form of any other data structure. For example,

```
>> T{1,3}
```

will retrieve the text string in the first column and third row. Notice that for *cells* you index using brackets, not parentheses. How can you retrieve the 3×3 random matrix? How about the value 27?

One last example will demonstrate the power of cells.

```
>> U = {T 5; 'Computing' [39.2 47]};
>> U{1,1}
>> U{1,1}{2,3}
```

Here the cell array U has been embedded within it another cell array T. The second command retrieves the T cell and the third command will retrieve a specific entry within T. You should save only U and P in a Matlab file called "MatrixStructures.mat". For help on saving a Matlab data file, see the previous chapter.

4.7 EXERCISES

1. Turn in the diary for this chapter along with the image file created in Section 4.5.2 and .mat file in Section 4.6.2.

2. Matrices can be a very good way to visualize distributions in space. For example, in modeling the spread of a disease throughout a population, it is important to know who is infected, who is immune and who is susceptible to infection. Let us assign anyone infected a value of 1, anyone immune a value of 2 and anyone susceptible a value of 3. We can visualize what is happening at a particular time with a 2D plot, where each coordinate represents a portion of a college campus.

   Start by creating a 40×40 matrix that contains all 3's (hint: use the *ones* command). Next, create two small regions of infections, one bounded by x=3, x=7, y=8, y=11, and another

bounded by x=35, x=39, y=27, y=32. Next, create a line of immunization (students vaccinated) that ranges from x=2 to x=38 and y=20. Use the *imagesc* and *print* commands to visualize the distribution of infected and not infected students and then print the image to a jpeg. You should include a colorbar.

3. A matrix can be very useful for storing the structure of a network. In biological applications, a network may be the connections between cardiac or neural cells, the interactions between different genes or even the relationships between species in an ecosystem. Typically, each "player" (also called an "agent" or "unit") is assigned an integer that is in the range 1 to N (where N is the total number of players in the network). Each row of our matrix will correspond to a player. For example, row one is dedicated to player 1, row two is dedicated to player 2 and so on. In each row (corresponding to the current player) we place a 1 in the location of any player with whom it can interact. For example, if row 8 is

$$[001000100001] \tag{4.1}$$

it means that player 8 has some direct relationship with players 3, 7 and 12. This type of matrix is called an adjacency matrix.

A very common and simple type of network structure is a one-dimensional ring. In a ring, each player is connected to the player before and after. For example, player 4 would be connected to players 3 and 5. Because of the ring structure, player 1 is also connected to the very last player, completing the ring.

Create an adjacency matrix, A, that describes a ring with 10 players. Remember that player 1 is connected to player 10 (and vice-versa), completing the ring. You should first write out the matrix by hand so you understand the basic structure. One method of creating the matrix would then be to simply program in the entire matrix by hand. For this exercise you must use the *diag* command to create the matrix. Note that you may need to add in a few extra commands (using what you know about indexing a matrix) to complete the loop. You should be able to create the entire matrix in 3 commands which you show in a diary file.

Because the matrix is mostly made of zeros, create a new sparse matrix, B. Then save A and B into a ".mat" file. Lastly, use the *spy* command on matrix B (note that *spy* works on sparse matrices too). Save the figure in a jpeg file.

4. Bacteria, and other micro-organisms, navigate throughout space using a variety of sensors. For example, they may simultaneously detect pH, glucose concentration gradient, light direction and temperature. To move in a particular direction a bacterium must decide which stimulus to move toward (or away from), but it may sometimes have conflicting "motivations", *e.g.*, moving

toward warmth may move it away from glucose. To model a large population of bacteria we would need to keep track of the state of each bacteria. Let's assume that pH is a number, glucose concentration gradient is a 2×2 matrix (defines 2 unit vectors), light direction is three numbers (angles in degrees) and temperature is a qualitative string (hot, just right and cold). In this exercise, you will create a Matlab structure, *Bac* to store the data for 3 bacteria. The names of the structures should be, pH, $Glucose$, $Light$ and $Temp$. You should be able to index data using $Bac(1)$, $Bac(2)$ and $Bac(3)$. For example, $Bac(1).pH$ should return a single number, $Bac(2).Temp$ should return a string, and $Bac(3).Glucose$ should return a matrix. You can make up any values you want for the entries for each bacteria. What is important is the makeup of the structure. When you are finished, save only the Bacteria Structure, *Bac*, into a ".mat" file.

CHAPTER 5

Matrix – Vector Operations

5.1 INTRODUCTION

The original intent of Matlab was to provide an all purpose platform for performing operations on matrices and vectors. Although Matlab has progressed much farther since these early ambitions, matrix-vector operations are still the core of the environment. In this chapter, we explore the basic functions Matlab has for performing operations on matrices and vectors.

This text is not meant to teach you about linear algebra. But there is a small amount of linear algebra that is necessary to understand before we move on to Matlab's matrix-vector operations. Perhaps the most important concept, is multiplication of a matrix and a vector. To demonstrate the concept, we will work with a 3×3 matrix and a 3×1 vector.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ ? \end{bmatrix}$$

To perform this multiplication we first multiply the values across the top row of the matrix by the values down the vector. The result will be

$$a * A + b * B + c * C \tag{5.1}$$

This term will form the first value in the resulting vector

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} a*A + b*B + c*C \\ ? \\ ? \end{bmatrix}$$

We then move down one row of the matrix and multiply again by the elements in the vector

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} a*A + b*B + c*C \\ d*A + e*B + f*C \\ ? \end{bmatrix}$$

and again for the last row of the matrix

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} A \\ B \\ C \end{bmatrix} = \begin{bmatrix} a*A + b*B + c*C \\ d*A + e*B + f*C \\ g*A + h*B + i*C \end{bmatrix}$$

Note that the mechanics of matrix-vector multiplication is to go across the rows of the matrix and down the column of the vector. To make the above more concrete, try to follow the example below

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} 10 \\ 11 \\ 12 \end{bmatrix} = \begin{bmatrix} 1*10 + 2*11 + 3*12 \\ 4*10 + 5*11 + 6*12 \\ 7*10 + 8*11 + 9*12 \end{bmatrix} = \begin{bmatrix} 68 \\ 167 \\ 266 \end{bmatrix}$$

Two points are important about matrix-vector multiplication. First, the number of columns in the matrix must be the same as the number of rows in the vector. Otherwise, the multiplication of matrix rows against the vector column will not work out right. The implication is that we need a matrix that is N × M and a vector that is Mx1 where M and N are any integers. Second, it is often a convention to use bold capital letters to denote matrices and bold lower case letters for vectors. We can therefore rewrite a matrix-vector multiplication as

$$\mathbf{Ax} = \mathbf{b} \tag{5.2}$$

where **A** is the matrix, **x** is the vector and **b** is the result of the multiplication.

5.2 BASIC VECTOR OPERATIONS

We have already explored a few operations on vectors. For example, in Chapter 3 we use the *sum* command. There are a number of additional commands that work on vectors. For a small sample, try issuing the commands below. You may also want to view the help files for these commands to better understand how they work.

```
>> v = [8 -1 3 -12 37 54.3];
>> mean(v)
>> std(v)
>> sort(v)
>> sin(v)
>> log(v)
>> max(v)
>> min(v)
>> i = find(v==-12)
```

It is important to note that some of the above commands will output only a single number, while others will apply a function to each element of the vector.

5.2.1 VECTOR ARITHMETIC

There are a number of very simple arithmetic operations which work on vectors. The most simple is to multiply all values by some constant value. For example,

```
>> y = [1 3 5 7];
>> x = 2*y
```

Note that the * in Matlab denotes multiplication. You can also add a constant value to all elements of a vector.

```
>> z = y + 2
```

You should try subtracting (use "-" symbol) a constant or dividing (use "/" symbol) by a constant.

5.2.2 VECTOR TRANSPOSE

Above we created a vector y that has the dimensions 1×4. The problem is that we might eventually want to multiply a matrix by this vector and it is in the wrong format. To transform y into a 4×1 vector we can simply use the *transpose* command.

```
>> w = y'
>> whos
```

5.2.3 VECTOR - VECTOR OPERATIONS

Matlab can also perform operations on two vectors. Try the four operations below and do not worry if you get errors.

```
>> l = [1 4 9 10];
>> m = [-1 35 2 0.5];
>> m-l
>> m+l
>> m/l
>> m*l
```

You will notice that the operations of addition and subtraction act the way you would expect. The division and multiplication, however, do not do what we expect. For the division we would expect to simply divide each element of vector m by each element of vector l and return a vector of the results. Instead, a single number was returned. In the multiplication example, Matlab would not even perform the operation. To understand the problem we will present two different scenarios. First,

$$\begin{bmatrix} 1 & 4 & 9 & 10 \end{bmatrix} \begin{bmatrix} -1 \\ 35 \\ 2 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 1\text{*-1} + 4\text{*35} + 9\text{*2} + 10\text{*0.5} \end{bmatrix}$$

Here we are simply using the type of multiplication used in the first section of this chapter for matrices and vectors. The only exception is that we are multiplying a 1×4 matrix by a 4×1 vector. The result is a 1X1 scalar. To perform this operation in Matlab

```
>> m*l'
```

Here the *transpose* command was used to turn the 1×4 *l* vector into a 4×1 vector. In fact, this type of vector-vector multiplication has the name of "dot product" or "inner product" in linear algebra. Matlab has a command, *dot* for taking dot products between two vectors, regardless of their dimensions.

The dot product, however, was not what we originally intended. What we expected was to multiply each of the elements of *m* by each of the elements of *l* and then place them in a new vector. To make this operation distinct from the dot product, Matlab has a special command.

```
>> m.*l
```

will perform the desired operation. The ".*" command is shorthand for an operation that is performed on each element. Try the following to see how element division works.

```
>> m./l
```

This still leaves us with why the division command in the original example yielded any answer at all. The answer is that "/" means something in Matlab that relates to matrix-vector division. It will be discussed below.

5.3 BASIC MATRIX OPERATIONS

Many of the basic commands we have already discussed work on matrices. First, we will use a built-in command to generate a matrix

```
>> B = magic(4)      %You may want to look at the help for 'magic'
```

First, you should notice that % is a special symbol that denotes a comment in Matlab. So, any text that appears after % will not be sent to Matlab. Below, comments will be used to help you understand the commands. You do not need to enter comments on your command line.

You may also have noticed in some of the help files that many commands that work on scalars also work on vectors and matrices. For example,

```
>> sum(B)                % sum across the rows
>> sum(B,2)              % sum down the columns
>> mean(B)               % mean down the rows
>> min(B)                % minimum down the rows
>> max(B)                % maximum down the rows
>> min(min(B))           % find the absolute minimum of B
>> exp(B)                % take the exponential of each element of B
>> [i j] = find(B==10)   % find row (i) and column (j) of value 10
```

This is a strength of Matlab - most operations are defined for scalars, vectors and matrices. By default, Matlab assumes that all operations on a matrix will be performed on the rows. The "sum(B)"

command thus returns a vector that is the sum of each row. But the *sum* command can total up columns instead by "sum(B,2)". You may also have noticed that you can embed commands within one another. For example, "min(min(B))" will first create a vector with the minimum value for each row (with the inner *min* command), but then take the minimum of that vector (with the outer *min* commands). The result will be the minimum value in the entire matrix.

5.3.1 SIMPLE MATRIX FUNCTIONS

One of the simplest matrix functions is the transpose

```
>> B'
```

The meaning of a matrix transpose is that the first row becomes the first column. The second row becomes the second column and so on. Although the example above is a 4×4 (or *square* matrix) it should be easy to check that the transpose of a N × M matrix is a M × N matrix.

Like vectors, we can perform basic arithmetic operations on matrices.

```
>> C = [1 2 3; 4 5 6; 7 8 9];
>> D = [-3 -2 -1; -6 -5 -4; -9 -8 -7];
>> C-D          % subtract the elements of D from C
>> C+D          % add the elements of C and D
>> C.*D         % multiply the elements of C and D
>> C./D         % divide the elements of C by the elements of D
>> C.^3         % cube each element in C
```

Each of the operations above is performed element-wise (because + and - are that way naturally, and we used ".*", "./" and ".∧"). It is possible, however, to multiply one matrix by another matrix.

```
>> C*D          % perform a full matrix multiply
```

where the pattern used for matrix-vector multiplication is used to create the entries of a new matrix. For example,

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix} \begin{bmatrix} D_{11} & D_{12} & D_{13} \\ D_{21} & D_{22} & D_{23} \\ D_{31} & D_{32} & D_{33} \end{bmatrix} =$$

$$\begin{bmatrix} C_{11}D_{11} + C_{12}D_{21} + C_{13}D_{31} & C_{11}D_{12} + C_{12}D_{22} + C_{13}D_{32} & C_{11}D_{13} + C_{12}D_{23} + C_{13}D_{33} \\ C_{21}D_{11} + C_{22}D_{21} + C_{23}D_{31} & C_{21}D_{12} + C_{22}D_{22} + C_{23}D_{32} & C_{21}D_{13} + C_{22}D_{23} + C_{23}D_{33} \\ C_{31}D_{11} + C_{32}D_{21} + C_{33}D_{31} & C_{31}D_{12} + C_{32}D_{22} + C_{33}D_{32} & C_{31}D_{13} + C_{32}D_{23} + C_{33}D_{33} \end{bmatrix}$$

Again, multiplications is carried out by moving across the rows of the first matrix and down the columns of the second matrix. Another way to think about matrix-matrix multiplication is that if we want to know the entry at any row and column of the resulting matrix, we simply take the dot product of that row of the first matrix (**C**) with the column of the second matrix (**D**).

Using the idea of matrix-matrix multiplication we can also perform matrix exponentiation

```
>> C^4
```

which is the same as the matrix multiplication C*C*C*C.

5.4 MATRIX-VECTOR OPERATIONS

As the core of Matlab is vector and matrix operations, it is not surprising that there are many functions for such operations. Below we explore only a few of the most important.

5.4.1 OUTER PRODUCTS

The inner product is an operation on two vectors that results in a scalar. The outer product is an operation on two vectors that results in a matrix. The key to understanding the difference is in the dimensions of the two vectors.

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{bmatrix} s & t \end{bmatrix} = \begin{bmatrix} a*s & a*t \\ b*s & b*t \\ c*s & c*t \\ d*s & d*t \end{bmatrix}$$

You should note that this operation is the same as the matrix-matrix multiplication but now with two vectors. In general, the outer product of a Nx1 vector and 1×M vector is a N ×M matrix.

5.4.2 MATRIX INVERSE

Earlier we introduced the matrix-vector equation

$$\mathbf{Ax} = \mathbf{b} \tag{5.3}$$

Many problems in engineering can be put in these terms, where we know **A** and **b** and would like to find **x**. Using some ideas from linear algebra, we could rearrange our equation.

$$\begin{aligned} \mathbf{Ax} &= \mathbf{b} && (5.4)\\ \mathbf{A^{-1}(Ax)} &= \mathbf{A^{-1}b} && (5.5)\\ \mathbf{(A^{-1}A)x} &= \mathbf{A^{-1}b} && (5.6)\\ \mathbf{(I)x} &= \mathbf{A^{-1}b} && (5.7)\\ \mathbf{x} &= \mathbf{A^{-1}b} && (5.8) \end{aligned}$$

where $\mathbf{A^{-1}}$ is called the *inverse* of **A**. By definition

$$\mathbf{A^{-1}A} = I \tag{5.9}$$

where **I** is the identity matrix, *e.g.*, ones along the diagonal and zeros elsewhere.

There are two problems with inverses. The first is mathematical and that is that not every matrix has an inverse. The criterion to have an inverse is that the matrix must be square, *e.g.*, same number of rows and columns, and must have a determinate (*det* command in Matlab) that is non-zero. A matrix with these properties is called *invertible* or *non-singular*. A matrix that doesn't have an inverse is called *singular*. Second is a more practical concern. Even for a matrix that is invertible, there are a number of numerical methods for finding $\mathbf{A^{-1}}$ given **A**. Matlab has a command (*inv*) for just this sort of operation, but due to the limits of numerical accuracy, it sometimes will not be able to return an inverse.

```
>> A = [3 2 -5; 1 -3 2; 5 -1 4];
>> det(A);                %Check to make sure an inverse will work
>> Ainverse = inv(A);
```

Now if we define a vector **b** we can try to solve for **x**

```
>> b = [12; -13; 10];
```

Note that **b** is a 3×1 vector. This situation corresponds to the following set of linear equations.

$$\begin{bmatrix} 3 & 2 & -5 \\ 1 & -3 & 2 \\ 5 & -1 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3x_1 + 2x_2 - 5x_3 \\ 1x_1 - 3x_2 + 2x_3 \\ 5x_1 - 1x_2 + 4x_3 \end{bmatrix} = \begin{bmatrix} 12 \\ -13 \\ 10 \end{bmatrix}$$

Where the vector **x** is what we want to solve for. In Matlab there are two ways to solve for **x**. The first is to compute the inverse and then solve.

```
>> x = Ainverse*b
```

The second reveals the special role played by the \ symbol in Matlab

```
>> x = A\b
```

And you can verify that **x** really is the solution by

```
>> A*x
```

It is important to note that Matlab uses different numerical methods when the inverse is taken directly and when the \ symbol is used. The \ symbol is nearly always better for both accuracy and time.

5.5 OTHER LINEAR ALGEBRA FUNCTIONS

Some other commands that have a direct tie to linear algebra are *rank*, *trace*, *det* (determinate) and *eig* (eigenvalues and eigenvectors).

```
>> Rand = rand(5,5);
>> rank(Rand)
>> trace (Rand)
>> [V,D] = eig(Rand)
>> det(Rand)
```

You may wish to view the help for these functions or consult a linear algebra text book.

5.6 MATRIX CONDITION

There are some instances when a set of equations are encountered, which in theory have a solution, but it is difficult to compute an inverse numerically. Consider the following system of equations:

$$a + b = 2 \tag{5.10}$$
$$a + (1 + \epsilon)b = 3 \tag{5.11}$$

where ϵ is some small number. We can solve this equation by subtracting Equation 5.11 from Equation 5.10. The result is

$$b = \frac{3}{\epsilon} \tag{5.12}$$

so the smaller ϵ the larger b will be, and with it, a will become large and negative. Matlab typically does not do well when numbers become too small or too large.

We can recast our problem in **Ax** = **b** form

$$\begin{bmatrix} 1 & 1 \\ 1 & 1+\epsilon \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

Note that there is nothing special about **x** or **b**, but **A** is where the problem lies. The idea of matrix condition is a way to quantify how much numerical error will be associated with inverting a particular matrix. The Matlab condition command is *cond*. To expose this problem in Matlab, try the following commands, where $\epsilon = 1^{-20}$

```
>> A = [1 1; 1 1+1e-20];
>> inv(A)
>> cond(A)
```

Although in theory, you should be able to take the inverse of **A**, Matlab will give an error that the matrix is singular to working precision.

5.7 EXERCISES

1. Turn in the diary for this chapter.

2. Create a random 25×1 vector, r, using the *rand* command. Sort this vector using the *sort* command in DESCENDING order. Turn in your sorted vector in a ".mat" file.

3. Nearly every ECG device has an automated heart beat detector (detects the peak of the R-wave), and can count the number of beats since turning on the device. The problem is that it requires these counts to be turned into heart rate changes over time. The data is stored in a vector, with each entry as the number of beats per minute, but where the counting of beats from the beginning did not stop at the end of each minute. Below is data from the first 7 minutes of recording.

$$a = \begin{bmatrix} 0 & 64 & 137 & 188 & 260 & 328 & 397 & 464 \end{bmatrix}$$

   Use the *diff* command to take the difference between each value. Note that the length of the difference is one less than the length of the original vector. Find the average heart rate over these 7 minutes. Show your commands in a diary file.

4. The Nernst Equation is used to compute the voltage across a cellular membrane given a difference in ionic concentration

$$E_k = \frac{RT}{F} ln \left[\frac{[K]_e}{[K]_i} \right] \tag{5.13}$$

   where E_k is the Nernst Potential in mV, R is the Ideal Gas constant ($1.98 \frac{cal}{K \cdot mol}$), F is Faraday's constant ($96480 \frac{C}{mol}$) and $[K]_i$ and $[K]_e$ are the concentrations of intracellular and extracellular Potassium in mM respectively. While working in a lab you recognize that in performing repeated cellular experiments, you must know the value of the Potassium Nernst potent. In the experiment, you can control the temperature and the extracellular concentration of Potassium. To avoid needing to compute the Nernst Potential each time, you decide to create a lookup table, *i.e.,* a matrix, containing the Nernst Potentials, thus avoiding the need to perform a new calculation for every experiment.

   Begin by typing in the following commands

```
>> Temp = [280:5:320];
>> Ke = [0.5:1:29.5];
>> R = 1.98;
>> F = 96480;
>> Ki = 140;
```

You then recognize that you can form E_k from two vectors

$$A = \frac{RT}{F}B = ln\left[\frac{[K]_e}{[K]_i}\right] E_k = A * B$$

where A*B is an outer product. Note that you will need to use the *log* command in Matlab and make sure that your vectors are the appropriate dimensions for an outer product. Store your result in the matrix Ek as a ".mat" file

5. As a biomedical engineer you may be interested in designing a bioreactor for cell culture. A typical chemical formula for aerobic growth of a microorganisms is

$$C_2H_5OH + aO_2 + bNH_3 \rightarrow cCH_{1.7}H_{0.15}O_{0.4} + dH_2O + eCO_2$$

where term $CH_{1.7}H_{0.15}O_{0.4}$ represents metabolism of the microorganism. The ratio of moles of CO_2 produced per mole of O_2 consumed is called the respiratory quotient, RQ, which can be determined experimentally. Given this ratio we have 4 constants, a-d that are unknown. We can perform a mass balance on each of the four key elements

$$\begin{aligned} Carbon &: 2 &&= c + (RQ)a \\ Hydrogen &: 6 + 3b &&= 1.7c + 2d \\ Oxygen &: 1 + 2a &&= 0.4c + d + 2(RQ)a \\ Nitrogen &: b &&= 0.15c \end{aligned}$$

in matrix notation

$$\begin{bmatrix} RQ & 0 & 1 & 0 \\ 0 & -3 & 1.7 & 2 \\ -2 & 0 & 0.4 & 1 \\ 0-1 & 0.15 & 0 & \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \\ 1-2RQ \\ 0 \end{bmatrix}$$

Let $RQ = 0.8$ and then find the vector [a b c d] using Matlab's matrix solve. Turn in a diary and ".mat" file with the final vector.

CHAPTER 6

Scripts and Functions

6.1 INTRODUCTION

In this chapter, we will learn about two related ideas that allow Matlab to be a very powerful all-purpose engineering application. The first is the idea of a *script* and the second is the concept of a *function*. It will often be the case, that future homework problems will require you to write either a script or a function.

6.2 SCRIPTS

You may have found in previous sections that you would often type in a number of statements on the command line in some sequence. For example, imagine trying to set up the following matrix.

$$\begin{bmatrix} 1 & 2 & 0 & 0 & 0 \\ 0 & 4 & 6 & 8 & 0 \\ 0 & 0 & 3 & 12 & 0 \\ 0 & 0 & 0 & 4 & 8 \\ 0 & 0 & 0 & 0 & 10 \end{bmatrix}$$

One strategy would be to enter the entire matrix on one line. A second, more reasonable strategy may be to recognize that you might create a few diagonal vectors using the *diag* command. The problem is that you will need to issue a sequence of statements to the command line in the right order. For a 5×5 matrix, this may not be all that bad, but imagine if this were a 200×200 matrix. To make matters even more difficult, there are many computer algorithms that would require thousands of lines to be typed in one by one. One mistake might mean reentering all commands from the beginning.

To overcome this problem, Matlab makes it possible to create a script - a text file that contains all of the commands to be entered in sequence. We will use the matrix above to motivate the creation of our first script.

```
>> edit MyFirstScript
```

The edit command in Matlab opens up the Matlab text editor. The second argument is the name of a file where the script will be saved. In fact, the file will be called 'MyFirstScript.m' where the '.m' designates that the file is a Matlab script. It is important to know that any text editor will do (a script file is simply a text file), but Matlab's editor has some built in functions that can help you check for errors. Enter the following commands into the script file

```
DiagVec = [1 4 3 4 10];                 %Create Diagonal Vector
UpDiagVec = [2 6 12 8];                 %Create Upper Diagonal Vector
A = diag(DiagVec) + diag(UpDiagVec,1);  %Create Matrix
A(2,4) = 8;             %Add in one more value in the second row, fourth column
```

Note that you can include % followed by comments. Matlab will not execute anything on the line that follows a %. Now save the file and close the editor. You should see the file MyFirstScript.m in your folder directory window. To issue the commands in your script to the Matlab command line, you simply enter

```
>> MyFirstScript
>> whos
>> A
```

Matlab will literally enter the commands one at a time in sequence. This method of computing is called *interpreted programming* or *scripting* and is the default in Matlab. You should notice that from this point onward, any command that contains >> is meant to be entered on the command line. If no >> is included, it means you should include the command in a script.

6.3 GOOD PROGRAMMING HABITS

It was mentioned above that some programs could contain thousands (maybe even millions) of lines of code. In such a long program, it would be very easy to become lost. Below we will discuss some ways to make your code more easy to read.

6.3.1 COMMENTS AND VARIABLES

The first script created above was relatively simple and an experienced Matlab programmer could easily decode its intent. As you become more versed in Matlab, your scripts will become more complex and much more difficult to understand. Comments are the key to explaining complex code. Imagine two scenarios when you are including comments: 1) If you handed your code to another Matlab programmer, could they follow what you were trying to do? 2) If you came across your own code five years from now, would you remember what you were doing and why?

There are some standard ways of commenting that can be very helpful. First is to include a purpose at the top of your script. It is helpful to include in this statement a date or version number. In this way, if you create a newer version you will not confuse the two. If you are writing code which may someday be shared with another programmer, you should include your name as the author. Second, it is very helpful to add comments to any variables you define, along with units and a brief note on the relationship to your problem. For example, if you are solving a circuit problem, you may initialize a vector, v, at the top of your script that will contain a vector of voltages at each node

```
>> v = zeros(4,1);  %Voltage vector at nodes in mV
```

Third, there are often blocks of code (ranging from a few lines to hundreds) which together perform some subprocess of the entire script. In these cases, it is helpful to have a header at the beginning of each block of code that explains the lines below. For example, the matrix, *A* above may have been created as the resistance between nodes, *i.e.,* the coefficients of simultaneous equations. Furthermore, you should include the following line to "MyFirstScript.m".

```
%%--------------Create Coefficient Matrix for Circuit---------------
```

The double percent sign is a feature of Matlab 7 or later and allows what is know as Cell Mode. In Cell Mode, regions of code will be blocked together and easy to identify as a functional unit in the code.

Fourth, there may be Matlab commands which require some explanation (you do not need to enter the command below)

```
A(g(l),h(m)) = sum(min(B(g(i)-10:g(i)+10,B(h(i)-10:h(i)+10))));
```

Although a command such as the one above may make sense at the time, it has so many embedded functions and indices that it will be difficult to understand what is going on. A comment can help in these cases.

Lastly, it is often helpful to use indentation to make the beginning and end of a functional block of code. This concept will become more clear in Chapter 7.

6.3.2 CATCHING ERRORS AND DISPLAYING TEXT

There are two commands in Matlab that can allow you to be alerted to errors and display information. Open up the file "MyFirstScript.m" and add the following line between the third and fourth line

```
disp('Diagonals created, now entering off diagonal entries');
```

In more advanced programs you may also want to be able to report out any errors. To view how this will work, enter the following on the last line of your first script

```
error('Check to make sure matrix is non-singular first!');
```

Try running the script with these two additions and observe the results. You may want to view the help files to understand more about the features of *disp* and *error*. Do not be alarmed when the *error* command sends a text to your screen in red.

The difference between *disp* and *error* is that *disp* will simply send text to the command line and continue on with your script. On the other hand, *error* will display text in red and then terminate your program.

6.4 SCRIPT EXAMPLE - THE RANDOM WALK

A bacteria placed in a uniform concentration of nutrients will follow what is known as a random walk. There has been much written on random walks, in fields as diverse as thermodynamics,

economics and history. In a biological context, a random walk can be thought of as a search process, *e.g.,* for food, light or a potential mate, when there are no outside cues.

To program a random walk, open a script file called "RandomWalk.m" and enter the initial $x - y$ position on the first line.

```
C = [1,1];          %Initial Coordinate
```

Next, we define a random distance to move in the x and y directions

```
Move = randn(1,2);  % get next move
```

and, lastly, move to a new spot

```
C = [C; C+Move];    % make next move relative to current position
```

The line above requires some explanation. In the first line of our script we created a vector [1 1]. In the second line we created a direction vector, [NextCx NextCy]. In the third line, we are turning C into a matrix. The first row of the matrix will contain the first x-y coordinate and the second row will contain the second x-y coordinate (C+Move). To make another move

```
Move = randn(1,2);      % get next move
C = [C; C(end,:)+Move]; % make next move relative to current position
```

Here the general idea is the same as above, we are adding a third location to the bottom of the matrix C, which is now 3×2. The seemingly strange indexing ($C(end,:)$) is to keep the dimensions uniform. Remember that $Move$ is a 1×2 vector and can only be added to another 1×2 vector, in this case the current last row of C. Try running your script to be sure that it works. Then try adding 8 more lines of the $Move$ and C commands to your script. At the bottom of your script include the following line

```
plot(C(:,1),C(:,2))     %Plot the random path traversed
```

You should see that the x-y position of your bacteria has been printed out. An enormous advantage of a script is that you can run your script again very easily without needing to reenter each command one at a time. And due to the nature of a random walk, you should get a different result each time.

6.5 FUNCTIONS

In 1962, Herbert Simon, a pioneer in the field of artificial intelligence and winner of the Nobel Prize in Economics, wrote an article called, "The Architecture of Complexity". In it was a parable of a watchmaker who is interrupted every few minutes to answer the phone. In frustration the watchmaker develops a solution - make a number of very simple parts that take less time to create, then assemble these simpler parts together into the watch. In this way, if interrupted, the watchmaker will know where to pick up after an interruption. Simon's conclusion was that any system that is sufficiently complex will only become so, and continue to function, if it is arranged in some sort of hierarchy. Simple operations are put together to make more complex functions and so on up the hierarchy.

The computational equivalent of building a hierarchy is to split up a long and complex programming assignment into smaller *functions* (also sometimes called subroutines). The simpler functions can then be combined together to perform the more complex task.

The reason for programming this way is four-fold. First, and already mentioned above, is clarity. Imagine that you write a long section of code, leave it for a year and then come back to it. Will you know where you left off? This has echos of the watchmaker parable above. A function can help make clear what has and has not already been finished. Second, it is much easier to fix small sections of code as you go. This is the idea of debugging, a topic that will be discussed at the end of this chapter. Third is reuse of code. There are often times when a particular function will be used more than once in a single programming assignment. And there are many instances where a particular function may be used again in a completely separate assignment. For example, a function that builds an adjacency matrix is a task that might be reused by programs for cardiac, neural, gene, social and ecological networks. Fourth is collaboration. Functions allow teams of programmers to coordinate their work to solve very large problems.

6.5.1 INPUT-OUTPUT

The coordination that would be required to collaborate is contained largely in how a function will transform inputs into outputs. Below is the general form that would allow a function to be called from the Matlab command line

```
OUTPUTS = FunctionName(INPUTS);
```

It may come as no surprise that many of the operations you have been using in Matlab are really just functions that are part of the Matlab distribution. To view how a built-in function is written in Matlab, use the *type* command

```
>> type mean
```

which will display the actual Matlab code that is executed when you use the *mean* command. You should first notice that the first line of *mean* contains a function declaration with the following form

```
function y = mean(x,dim)
```

The key word *function* tells Matlab the following file will be a function with the form OUTPUTS = FunctionName(INPUTS). It is important that your "FunctionName" is also the name of the script file you created. In our example, the text file that contains the mean function is called "mean.m". Therefore, if you create a function called "SquareTwoNumbers", your text file would have the name "SquareTwoNumbers.m" and would have a first line of [xsquared,ysquared] = SquareTwoNumbers(x,y).

Following the function declaration for *mean* is a long section of code that is commented (%) out. This is in fact the help for the *mean* command. One very nice feature of Matlab is that

the code and help information are all in the same file. When you write a function, it is good programming form to always include a help section. And you should follow the convention used in Matlab of explaining the format for the inputs (are they scalars, vectors or matrices?) and outputs. It is also helpful to have a few brief words on how the function works.

You should note that not all built-in Matlab functions are ".m" files.

```
>> type sum
```

will return "sum is a built-in function". There are a number of core Matlab functions that have been compiled. The reason to compile a function (as opposed to running it as an interpreted script) is that it will run much faster. Although we will not discuss them here, Matlab has a way of deeply compiling your own functions. You can find out more about this by typing "help mex".

6.5.2 INLINE FUNCTIONS

Matlab does allow users to create what are called *inline* functions. These are functions that appear in the same text file. Although they can be very useful in some situations, we will not discuss them here.

6.5.3 THE MATLAB PATH

You have already seen that Matlab has built-in functions, some compiled and some as scripts. But all of these scripts are stored in directories and chances are very good that you are not in those directories. Matlab needs to know how to find "mean.m" when you use that particular function. For this reason, Matlab contains a list of directories to search for "mean.m". You can see this list by typing in

```
>> path
```

The advantage of the path is that you do not need to tell Matlab where to look for built-in functions. The disadvantage is that if you write a new function, and are not in the directory that contains your new function, Matlab will not know where to look for it. There are two potential solutions. First, you could simply move into the directory that contains your function. For simple programming exercises this is your easiest solution. In a large programming assignment, however, you may wish to have several directories to help organize your functions. In this case, you will need to add a search path to Matlab. You can view the help for *path* to see examples for your particular operating system. You may also use the *addpath* command.

One additional consideration when naming functions is that Matlab already has a number of functions. If you name your new function the same as a built-in function, Matlab may become confused as to which function to use. It is a good idea to use unique names for your functions.

6.5.4 FUNCTION SIZE

So how big should your function be? The answer will depend upon the nature of your problem. In general, a function is meant to achieve one task. It should be something that a user could pick up and have a good idea of how it will work. On the other hand, the task should not be trivial. The balance of these two will change as you become a better programmer. For now, you should focus on writing short, simple functions and nest them in a hierarchy - *i.e.,* use simple functions to build functions of medium complexity and then use the medium complexity functions to create more complex functions, and so on.

6.6 DEBUGGING

In writing both scripts and functions it is very helpful to have some techniques for debugging your code. First, Matlab does have a built in debugging tool. Although we will not discuss the debugging tool here, you can find out more information by viewing the help for the *debug* command. Second, trying to write all of your code at once may help you organize your ideas, but you should not expect it to run properly. Rather it is good programming practice to write your code in small sections, testing them out as you go. In this way, you can add a line or two to your program and then test them out to be sure they work. If there is an error, you will know that it has occurred in those lines. Third, you can take the semi-colon off of any line in a Matlab script and the output will be sent to the command-line as the script runs. The advantage is that you can check to make sure that scalars, vectors or matrices look right. The one danger in this method of debugging is that you might overload the command line, *e.g.,* sending a 1000×1000 matrix to the command line is not a good idea. Rather, you may create a new variable that will give you the information you need to know. For example, to perform a quick check on a 1000×1000 matrix you might simply include the following line

```
[N, M] = size(A)
```

which will allow you to see the size of your matrix on the command line. Alternatively, it might be the *sum* or *spy* command that you may want to use to help you debug.

When it comes to writing functions, it is often a good idea to write your desired function as a script first. In this way, you can simply put the inputs and outputs at the top of the file and change them by hand. Once you become confident that your code is working the way you intended, you can turn it into a function.

6.7 USER INPUT

Earlier it was mentioned that a good Matlab function has defined inputs and outputs. In a large program, one function may call another, which called another, and so the output of one function may become the input to another function. But often, we also want our program to be interactive, meaning that the user can change inputs that impact the flow of the program.

6.7.1 INPUT

There are times when you may wish to allow the user to enter some input to a script or function. The input may be to make a decision about how to proceed, *e.g.,* perform algorithm 1 or algorithm 2, change a parameter value, *e.g.,* coefficient or an initial condition, or even terminate the program. The command in Matlab to use in these instances is *input*.

```
result = input('The rate constant is-->');
```

When placed in a script (or function), the text "The rate constant is –>" will be displayed on the command line. Matlab will then wait for the user to enter a command. Once the user enters a number and presses enter, that number will be stored in the variable *result* and will then be available to the rest of the script. The *input* command can be used to return any Matlab data type.

6.7.2 GINPUT

You will create a new script that contain the following commands. Name your script file "MySecondScript". First, create the matrix **A** in Matlab and then use the *imagesc* command to visualize it in a figure.

$$A = \begin{bmatrix} 1 & 2 & 0 & 6 & 1 \\ 0 & 4 & 6 & 8 & 0 \\ 2 & 0 & 3 & 12 & 2 \\ 8 & 2 & 0 & 4 & 8 \\ 7 & 0 & 5 & 0 & 10 \end{bmatrix}$$

Next, you should also add a color bar using the *colorbar* command. On the next line you should include the following line

```
[x, y] = ginput(2);
```

The meaning of the line above is to bring up a cursor which you can move around with your mouse. When you click your right mouse button, the x-y, coordinate of that first click will be stored in the value x and y. But the argument to *ginput* is a 2, meaning that the command will wait for 2 clicks of the mouse. Save and run your script and then try clicking two different points on your figure. View the 2×1 vectors x and y to verify that they have selected the coordinates of your mouse clicks.

6.8 FUNCTION EXAMPLE

The random walk script created above was able to simulate the random movement of a bacterium. The script was limited in at least two ways. First, there was no way for the user to specify parameters from the outside. Second, two lines were repeated over and over again. In a preview of the next chapter, we will begin by tackling this second problem. Open up a script called "RandomWalkReprise.m" and enter

```
figure(1)
hold on
C = [1,1];                  %Initial Coordinate
for i=1:10
    Move = randn(1,2);      % get next move
    C = [C; C(end,:)+Move]; % make next move relative to current position
    plot(C(end-1:end,1),C(end-1:end,2));
    pause(1)
end
plot(C(:,1),C(:,2))
```

The first two lines create a blank figure and then make sure that every future plot command will be sent to Figure 1. The next line creates the initial coordinate. The following line contains what is known as a *for* loop. You will learn much more about loops in the next chapter. For now, all you need to know is that Matlab will execute the commands between the *for* and *end* ten times, *e.g.*, 1:10. The first two lines inside the for loop are familiar. The plot command should also be familiar, but you should make sure that you understand the indexing (hint: we are drawing a line from the previous point to the current point each time through the loop). The last line in the for loop will pause for one second to give you a chance to see what is happening. Run the script and observe the results.

Now that the script is working, we can begin to make it more general. Copy RandomWalkReprise.m to RandomWalkReprise2.m and modify the new script to match the code below

```
Initialx = [2,1];   %Initial coordiate
NumLoops = 50;      %Number of steps to take
pausetime = 0.2;    %Duration of pause

figure(2)           %Change to not confuse with previous script
hold on
C = Initialx;       %Initial Coordinate
for i=1:NumLoops
    Move = randn(1,2);      % get next move
    C = [C; C(end,:)+Move]; % make next move relative to current position
    plot(C(end-1:end,1),C(end-1:end,2));
    pause(pausetime)
end
plot(C(:,1),C(:,2))
```

In this version of the script, all of the values which we want to be user defined have been placed at the top of the script. The last step is to turn $Initialx$, $NumLoops$ and $pausetime$ into inputs to a function, called "RandomWalk", with the matrix C as the output

```
function C = RandomWalker(Initialx, NumLoops, pausetime)
figure(2)           %Change to not confuse with previous script
hold on
C = Initialx;       %Initial Coordinate
for i=1:NumLoops
    Move = randn(1,2);      % get next move
    C = [C; C(end,:)+Move]; % make next move relative to current position
    plot(C(end-1:end,1),C(end-1:end,2));
    pause(pausetime)
end
```

Then on the command line try

```
>> clear
>> C = RandomWalker([3,4], 400, 0.01);
>> whos
```

The last important note about functions is that when they are completed, Matlab's memory will contain only the inputs and outputs. For example, *Move* is a variable that is used only internal to the function and therefore does not show up in Matlab's memory after the function has completed.

6.9 EXERCISES

1. Turn in the ".m" files MyFirstScript.m, MySecondScript.m, RandomWalk.m, RandomWalkReprise.m, RandomWalkReprise2.m and RandomWalker.m.

2. Epidemics occur when an infection appears seemingly out of nowhere, spreads throughout a population, sometimes disappearing just as suddenly. The Kermack-McKendrick model describes the number of susceptible, x_n, the number of infected, y_n, and the number of removals (quarantined, immunized), z_n. For a constant population we can assume $x_n + y_n + z_n = N$. Once N is set, we therefore only need to know two of the variables to know the remaining variable. The dynamics of Kermack-McKendrick model are

$$x_{n+1} = x_n e^{-ay_n} \tag{6.1}$$
$$y_{n+1} = (1 - e^{-ay_n})x_n + by_n \tag{6.2}$$

   where e can be represented by the *exp* function in Matlab, a and b are constants. First, create a script called 'KMmodelScript' that has the following lines at the top

```
a = 0.02;
b = 0.0;
xn = 25;
yn = 5;
```

Then add two more lines

```
xn1 = ?
yn1 = ?
```

where you fill in ? with the Kermack-McKendrick model. You must turn in this script. Next, create a function called "KMmodel" that will take as inputs a, b, x_n and y_n and output x_{n+1} and y_{n+1}. You then must create a script called 'KMmodelFunctionTest' that contains the following lines

```
a = 0.02;
b = 0.0;
xn = 25;
yn = 5;
[xn1,yn1] = KMmodel(a,b,xn,yn);
```

Turn in KMmodel.m and KMmodelFunctionTest.m.

3. One way to model the dynamics of a population of a particular species is to think of it as a series of generational groups, for example, infants, juveniles, mature and post-reproduction. In all cases, as the years go by, some animals will move from one category to another, some will die, others will be born as infants. We can define a vector

$$\mathbf{x^n} = \begin{bmatrix} x^n_{infant} \\ x^n_{juvenile} \\ x^n_{mature} \\ x^n_{old} \end{bmatrix}$$

Then we can hypothesize that there is a matrix, $\mathbf{A}$, which will transform $\mathbf{x}^n$ to $\mathbf{x}^{n+1}$

$$\mathbf{x}^{n+1} = \mathbf{A}\mathbf{x}^n \tag{6.3}$$

Such a matrix is called a stage structure matrix. From 1973 to 1987 Brault and Caswell studied killer whale populations and developed the following $\mathbf{A}$ matrix

$$A = \begin{bmatrix} 0 & 0.0043 & 0.1132 & 0 \\ 0.9775 & 0.9111 & 0 & 0 \\ 0 & 0.0736 & 0.9534 & 0 \\ 0 & 0 & 0.0452 & 0.9804 \end{bmatrix}$$

Moving the current population forward one year therefore requires multiplication by $\mathbf{A}$. Moving the population forward by two years requires two multiplications by $\mathbf{A}$

$$\mathbf{x}^{n+1} = \mathbf{A}\mathbf{x}^{n} \tag{6.4}$$
$$\mathbf{x}^{n+2} = \mathbf{A}\mathbf{x}^{n+1} \tag{6.5}$$

or

$$\mathbf{x}^{n+2} = \mathbf{A}\mathbf{A}\mathbf{x}^{n} \tag{6.6}$$
$$\mathbf{x}^{n+2} = \mathbf{A}^{2}\mathbf{x}^{n} \tag{6.7}$$
$$\tag{6.8}$$

Therefore, if we want to predict the population at any year in the future we can use the idea of matrix exponentiation

$$\mathbf{x}^{n+2} = \mathbf{A}\mathbf{A}\mathbf{x}^{n} \tag{6.9}$$
$$\mathbf{x}^{n+y} = \mathbf{A}^{y}\mathbf{x}^{n} \tag{6.10}$$
$$\tag{6.11}$$

where the exponent y is the years in the future.

Write a function ("PopulationStage") that will take in any current population vector $\mathbf{x}$, a stage structure matrix $\mathbf{A}$ and predict the population at any given year y. Write a script to test your new function with the $\mathbf{A}$ as defined above and a current population vector $x = [10, 60, 110, 70]$. Show that you can predict the population 50 years into the future.

CHAPTER 7

Loops

7.1 INTRODUCTION

One of the prime purposes of using a computer is to automate a task that would be very tedious to perform by hand (either with pencil and paper or on a calculator). The usual implication is that some task is to be performed over and over again in some systematic way. This chapter will be concerned with the programming concept of a loop, a feature that is at the heart of nearly every computer algorithm. Perhaps the most important concept to understand about loops is the least intuitive; how to get them to stop. In fact, the method of stopping a loop is often how they are classified. In this chapter, we will explore the *for* loop in detail and wait until the next chapter to explore the *while* loop.

7.2 THE FOR LOOP

There are often algorithms where you know ahead of time exactly how many times an operation must be performed before stopping the loop. For example, if we must perform some data analysis and know that there are 1037 points, there should be some way to move from point 1 (perform the analysis) to point 2 (perform the analysis) to point 3, and so on. In Matlab, and nearly all other programming languages, this type of operation is performed by a *for* loop.

In Matlab, the most basic type of *for* loop is the following (do not enter the following commands)

```
for i = 1:200
    COMMANDS HERE
end
```

There are four parts to any *for* loop. The first is a variable (in this case, i) that will be used to keep track of the number of times through the loop. The second is the values the variable i will take on as it repeats. In our example, i will start by taking the value 1, then 2, then 3 and so on until reaching 200. Third is to place a bound on what will be looped over, denoted in Matlab by the keyword *end*. Fourth are the commands that will be executed (there can be as many as needed) between the *for* and the *end* lines. Note that this is a good example of using indentation to make clear where the loop starts and ends, as well as the commands to be executed.

To better understand how loops work, analyze the following code.

```
x = 1;
for i = 1:5
    x = x + 1;
end
```

In this code, the variable x will start at a value of 1. Then the loop will begin with $i = 1$. On the first pass through the loop, x will be increased by 1 (*e.g.*, $x = 2$). On the second pass through the loop, ($i = 2$) x will be increased again by 1, *e.g.*, $x = 3$, and so on until $i = 5$.

It should be noted that there is no reason why you must start a loop at 1. The following will all yield valid loops (do not enter on the command line, simply view the lines below):

```
for i = 89:108
for i = -27:38
for i = 234:20
for i = 23.5:57.5
```

You should see that you can start at any number (positive, negative or even a non-integer) and then loop up or down.

7.2.1 FOR LOOPS OVER NON-INTEGERS

You probably noticed that the colon notation was used to specify that Matlab should loop from some number to another number in increments of 1. There are often cases where we would like to increment by some other value. For example, imagine that you would like to assign a value at points along a line of length $1cm$ every $0.01cm$.

```
dx = 0.01
for x = 0:dx:1
    v = x^3 + x^2 + 3
end
```

Upon executing the commands, you should see the values of v on the command line.

7.2.2 VARIABLE CODING

Above we defined a variable dx to specify how much to increment each time through the loop. There is no need to do this, *e.g.*, for x = 0:0.01:1 would have worked, but it demonstrates the good programming practice of variable coding as opposed to hard coding. In hard coding, we simply write out all of the numbers in our program, *e.g.*, type in 0.01 for dx everywhere in the code. In soft coding, we define a variable, *e.g.*, dx, and then use that variable throughout the program. There are two reasons why it is good practice to use variable coding. First, by defining a variable dx, we can signal that there is some actual meaning (here a spatial size) to the variable. Second, if at any point in time you would like to change dx, you can make the change in one place, eliminating the need to search for all places where a step size is needed.

7.2.3 FOR LOOPS OVER AN ARRAY

A second trick that is often helpful is to loop over some array that has already been created

```
>> a = [2 50 34.5 27 91];
>> for i = 1:length(a)
>>     2*a(i)+100          %Any command here that requires a(i)
>>end
```

Here a vector *a* is created before the loop. The goal is to progress through the entire vector one element at a time, *e.g.,* a(i), performing operations on that element. Note that, in this case, the variable *i* is playing two roles. It is the loop variable, but it is also the index to the vector, *s*. This is one of the most powerful aspects of having a loop variable.

7.2.4 STORING RESULTS IN A VECTOR

There are two problems with the above script. First, sending the output to the command line makes it difficult to interpret trends in the results. Second, we do not have any record of the resulting calculation. To fix both problems, we can use the loop variable *i*. Enter the following into a script.

```
dx = 0.01;              % spatial step in cm
space = 0:dx:2;         % create space vector

%Allocate memory for a Concentration vector
Concentration = zeros(length(space),1);

for i = 1:length(space)
    %Any command here that requires a(i)
    Concentration(i) = space(i)^2;
end
plot(space,Concentration)
```

In this example, a *space* vector is created. Then the a vector *Concentration* is built up one element at a time by looping over the entire *space* vector. The catch is that the *space* vector is also used to create the *Concentration* vector. As shown in the plot, the above commands create a concentration gradient in space.

A careful reader may realize that the above section of code could have been created much more efficiently without any loops, by using a simple element-wise vector multiply

```
dx = 0.01;                  % spatial step in cm
space = 0:dx:2;             % create space vector
Concentration= space.^2;    % simple element-wise operation
plot(space,Concentration)
```

In fact, if faced with this type of problem you should choose the above solution, as loops are generally much slower than matrix-vector operations. There are, however, many instances where a problem can not be decomposed into matrix-vector operations. One such case is when you must know one result to compute another result, as in the Kermack-McKendrick difference equation in the previous chapter. A second example is the numerical integration of a differential equation.

7.3 EULER INTEGRATION METHOD

Great emphasis is often placed on *solving* differential equations, meaning that a closed form analytical solution can be written down. The vast majority of differential equations, however, can not be solved analytically. This is especially true for biological systems, where the equations that govern how quantities change are almost always non-linear. In cases where a differential equation can not be solved analytically, we can approximate a solution numerically. There are a number of numerical integration methods, but the simplest to understand is the Euler Method.

At the heart of the Euler Method is knowing where you are (current state) and where you are headed (slope). In Figure 7.1, we define the current state as V^t and the current slope as $\dot{V}^t$. We can then approximate the slope as

$$\frac{dV}{dt} = \dot{V}^t = \frac{\Delta V^t}{\Delta t} \tag{7.1}$$

If we now pick a Δt we can predict where V will be at time $= t + \Delta t$

$$V^{t+\Delta t} = V^t + \Delta V \tag{7.2}$$
$$V^{t+\Delta t} = V^t + \Delta t \cdot \dot{V}^t \tag{7.3}$$

Note that on the left-hand side is the prediction at time $= t + \Delta t$ and on the right-hand side are all terms at time $= t$. In the figure, the predicted $V^{t+\Delta t}$ is not the exact solution, *i.e.*, it does not fall on the solid line. But, as Δt is made small, the approximation will become better and better.

The generic approach of moving step-by-step through a differential equation is called numerical integration. The ability to integrate a differential equation (even a non-linear one) is a very important role for computer programming. And the Euler Method will work even if the independent variable is something other than time. You should also note that there are many more numerical integration methods that we will touch upon in future chapters.

7.3.1 NUMERICAL INTEGRATION OF PROTEIN EXPRESSION

To demonstrate how the Euler Method works in a real situation, we will integrate the differential equation for protein expression.

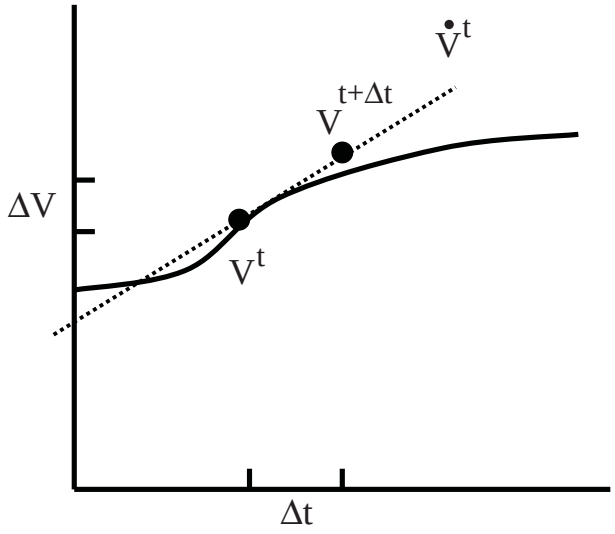

Figure 7.1: Demonstration of Euler Method. The dotted line indicates the slope of V at time t. The solid line indicates an analytical solution.

$$\frac{dY}{dt} = f(x) - \alpha Y \tag{7.4}$$

$$f(X) = \frac{\beta X}{K^n + X^n} \tag{7.5}$$

The differential equation above explains how a protein X can promote the upregulation of a gene that produces protein Y. The term $-\alpha Y$ expresses how over time the concentration of Y will fall due to protein degradation and cell division. The constant term α describes the rate at which this loss of Y will occur. The term $f(X)$ governs the upregulation of Y, *e.g.,* increase in concentration of Y, and is dependent upon the concentration of X. The form of $f(X)$ can vary, but we have chosen to use the *Hill* equation for a promotor protein. The Hill equation has three variables. K is the activation coefficient and governs how much of X must be present to cause Y to be expressed half as much as the maximum expression rate. β is the maximum expression, and the exponent n governs how quickly a promotor can switch from on to off as a function of X concentration.

Using some fancy analytical technique, we may be able to solve the equation analytically, but the Euler method and some basic programming skills will allow us to investigate the meaning of the parameters much more easily.

```
alpha = 1.0;      %Degradation of Y
```

```
beta = 1000.0;    %Hill Maximal Expression Level
K = 20;           %Hill Half-Maximal Activation
n = 3;            %Hill Exponent

%Create time vector
dt = 0.01;
EndTime = 20;
time = 0:dt:EndTime;

%Find Rate of Y production given concentration of X
x = 10;                %Concentration of Promotor x
fx = beta*x/(K^n+x^n); %rate of Y production

%Initialize Y concentration vector
Y = zeros(length(time),1);
Y(1) = 0.0;               %set to 0 but could be changed

%Loop over time using Euler Method
for i = 2:length(time)
    Y(i) = Y(i-1) + dt*(fx-alpha*Y(i-1));
end

%Plot out expression of Y over time
plot(time,Y);
```

Running the script above should result in a plot of the time course of Y expression. You will revisit this example in an exercise below, so save your script as "ProteinExpressionScript.m".

7.4 THE LOGISTIC EQUATION REVISITED

In Section 2.6, the logistic equation was described as

$$z_{n+1} = r z_n [1 - z_n] \tag{7.6}$$

where z_n is the current value of z, z_{n+1} is the next value of z and r is a constant. For example, we will set $r = 3.2$ and the initial population value to $z = 0.5$, *e.g.*, half of the maximum. In our previous use of the logistic equation, we simply reentered the command over and over again. Now with scripting and loops, we can automate the process. Enter the following commands into a script called "Logistic.m".

```
r = 3.2;                  %constant for logistic equation
Initialz = 0.5;           %initial condition
```

```
N = 100;                        %number of iterations to perform
z = zeros(N,1);                 %Create vector to store z
z(1) = Initialz;                %Set first value to initial

for i=2:N                       %Start at 2 because we already z(1)
    z(i) = r*z(i-1)*(1-z(i-1));    %calculate next value
end

plot(z)
```

The script above will iterate through the logistic equation 100 times and then plot the results.

7.5 THE WHILE LOOP

There are times when it is not possible to know exactly how many times a loop should be performed. It is still very important, however, to have some way to stop the loop from iterating forever. In these types of situations, you may use a *while* loop. Because *while* loops require checking a logical condition, *e.g.*, is some variable larger than some other variable, we will not discuss *while* loops until the next chapter.

7.6 NESTED LOOPS

Loops are assigned a variable for two purposes. The first is so that the variable can be used to perform some useful function, *e.g.*, as an index to an array or matrix. The second is because loops may exist within other loops, and we need variables to make it clear where in the iteration sequence we are. Do not enter the commands below.

```
for i=1:N
    for j=1:M
        COMMANDS HERE
    end
end
```

In the template code above i is set to 1, then j is looped from 1 to M. Then i is set to 2 and j is again looped from 1 to M and so on until $i = N$. You are certainly not limited to two nested loops, and you can mix and match *for* and *while* loops as needed. Below we discuss two situations where this type of nested loop structure can be very useful.

7.6.1 LOOPING OVER MATRICES

There are situations where it may be helpful to move systematically through a matrix. For example, consider that a bacterium is capable of swimming up a sucrose gradient to reach a plentiful supply of nutrients. If we were to simulate a bacterium swimming in such an environment, one of our first

steps would be to create a concentration gradient. Let us assume that we will create a gradient that is low in the middle of a 200×200 grid and increases as we radiate outward from the center. The problem is that we can not simply issue a few *diag* (or other Matlab) commands to create our matrix. We must move through each point on the grid and compute its distance from the center.

```
C = zeros(200,200);
CenterPoint = [100 100];  %[xcoordinate ycoordinate]
dx = 0.01;                %spatial step size

%compute real physical location of center
CenterLocation = CenterPoint*dx;
a = 2.5;    %scale factor for distance-concentration

for i = 1:200       %x loop
  for j = 1:200     %y loop
      XL = dx*i;    %x location
      YL = dx*j;    %y location
      DistanceFromCenter = sqrt((XL-CenterLocation(1))^2+ ...
                  (YL-CenterLocation(2))^2);
      C(i,j) = a*DistanceFromCenter;
  end
end

imagesc(C);
colorbar
```

In the line that created *DistanceFromCenter* we used another feature of Matlab. There are some cases where a line must be very long, and therefore can be difficult to read. The "..." means to continue the command onto the next line.

It is important to note that we could have easily created a more general script by not assuming the grid would be 200 x 200. We also have the flexibility to make the center point anywhere on the grid we would like. Set *CenterPoint* to [50, 75] and create a figure called "SucroseGradient.jpeg" showing the result. Can you see how you might turn a script such as the one above into a general function? Can you also see how you could create three nested loops to create a 3D gradient? There are other situations where you may need to use nested loops to move through a data structure, but you should use it only as a last resort. If you can find a way to perform an operation without loops, it will generally execute much faster.

7.6.2 PARAMETER VARIATION

In Section 7.4, a function was created to iterate through the logistic equation. In this section we will explore how the dynamics of the logistic equation change as the parameter r is varied. Copy the file "Logistic.m" to a new script called "LogisticBifurcation.m".

```
r = [2:0.001:4];              % create vector with r values

figure(1)
hold on

for j = 1:length(r)           % loop over the r vector
    Initialz = 0.5;
    N = 1000;                 %CHANGED TO 1000
    z = zeros(N,1);
    z(1) = Initialz;
    for i=2:N
        %calculate next value of x using current r value
        z(i) = r(j)*z(i-1)*(1-z(i-1));
    end


    %create a vector with all values of r(j)
    rvec = r(j)*ones(500,1);

    %only use the last 500 values of z
    truncatedz = z(501:end);

    %plot points, but do not connect lines
    plot(rvec,truncatedz,'.');
end   %End r loop
```

In the script above, one loop is used to systematically change r and a second loop is used to perform the iteration of the logistic equation. You should notice that some of the skills learned in the sections above have been used here, for example, creating the r vector first and then looping through it with the loop variable j. You should also notice that we created two temporary variables, $rvec$ and $truncatedx$, for the purposes of removing any of the initial dynamics (transients) of the logistic iteration. Here we are only interested in the long-term behavior. Save your bifurcation plot in a figure called "LogisticBif.jpeg".

We can now turn to the meaning of the plot. On the x-axis, is the parameter r that we varied from experiment to experiment. On the y-axis we have done something a bit unusual -

we have plotted the entire time series on top of one another. To gain insight into the logistic behavior at a particular value of r, we can simply draw a vertical line upward to see where it intersects values for z. If z has reached a steady-state, it will appear as only one point because every point is the same. On the other hand, if z oscillates between two values, our vertical line will intersect at 2 points. You should notice in your plot that as r is increased, we move from steady-state to oscillating between 2 z values, to oscillating between 4 z values, to 8 z values and so on.

There are two points about the plot you have created that are important in many biological systems. The first is that as r is changed, the system moves from one type of behavior to another, *e.g.*, from steady-state to oscillating. What is most striking is that the transition is abrupt and occurs at a particular value of r. These types of abrupt transitions are called **bifurcations** and are found in many biological systems. The parameter r is called the bifurcation parameter. In the logistic equation, there is only one parameter, r, that needs to be varied to drive the system through a bifurcation. In many biomedical problems, however, it is some combination of variables that will move the system through a bifurcation. Systems with bifurcations are excellent candidates for parametric studies and highlight one of the strengths of using computation to study biological systems.

The second important point is that for some values of r, the behavior of the logistic equation becomes chaotic. It is easy to see from the plot that as r is increased, the period of oscillation doubles, then doubles again, then again, and so on. The meaning is that the time series will visit first 1, then 2, then 4, then 8, and then 16 unique values of z. This doubling continues until the period is infinite. The meaning of an infinite number of possible values for z is that there is no longer any defined period, *e.g.,* the behavior never repeats. The logistic equation demonstrates one of the most studied pathways to a chaotic system - a series of period doubling bifurcations.

Now that you have a bifurcation map and understand its meaning, you should copy "Logistic.m" to "LogisticTest.m" and then try different values of r. You should be able to predict when r will result in steady-state, periodic or chaotic behavior from the bifurcation plot. Create a plot of a chaotic time series of the logistic equation and save it in a figure called "LogisticChaos.jpeg".

7.7 EXERCISES

1. Turn in "Logistic.m", "LogisticBifurcation", "ProteinExpressionScript.m", "LogisticBif.jpeg", "SucroseGradient.jpeg","LogisticChaos.jpeg".

2. In Section 7.3.1, you created a script to compute expression of protein Y over time as a function of a promotor protein X. In this exercise, you will modify you script to evaluate how Y expression changes as X is changed. To do so, you will treat X as a parameter that will vary from 0 to 50 in increments of 1. The goal is to record the steady-state, *e.g.*, $t \rightarrow \inf$, value of Y. Below is a portion of a script that should help you organize your script, "Assignment6-Problem2.m".

```
figure(1)
hold on

%Find Rate of Y production given concentration of X
x = [0:1:50];
SteadyStateY = zeros(length(x),1);

for j = 1:length(x)
    fx = beta*x(j)/(K^n+x(j)^n); %rate of Y production

    %PLACE APPROPRIATE COMMANDS HERE

    %Plot out expression of Y over time
    plot(time,Y);
    pause(0.2)
    SteadyStateY(j) = Y(end);
end

figure(2)
plot(x,SteadyStateY);
```

In Figure 1, we will plot the time courses of Y for each value of x. The *pause* command is used to clarify the trends as x is increased. In Figure 2, we are plotting only the very last (steady-state) value for Y as a function of x.

3. Many biological systems can not be characterized by a single differential equation. In these cases, we can think of the system as a set of coupled differential equations. For example,

$$\frac{dx}{dt} = f(x, y, z) \tag{7.7}$$
$$\frac{dy}{dt} = g(x, y, z) \tag{7.8}$$
$$\frac{dz}{dt} = h(x, y, z) \tag{7.9}$$
$$\tag{7.10}$$

notice that functions f, g and h are functions of all three variables, *i.e.,* the equations are coupled. Also notice that f, g and h could take any form and will mostly likely not be linear. For example, the FitzHugh-Nagumo model of a neuron is

$$\frac{dV}{dt} = V - \frac{V^3}{3} - W + I \tag{7.11}$$
$$\frac{dW}{dt} = a * (V + b - cW) \tag{7.12}$$

where V is the cell membrane potential, W is a recovery variable and I is a stimulus current. We will assume the constants are $a = 0.08$, $b = 0.7$ and $c = 0.8$. Because these equations are non-linear, we cannot transform them to the form

$$\frac{d}{dt}\mathbf{x} = \mathbf{A}\mathbf{x} \tag{7.13}$$

and therefore need to use a numerical integration technique. You can assume that the initial values for V and W are both 0. Integrate the equations using the Euler Method with a $\Delta t = 0.01$ from $time = 0$ to $time = 500$. You should create a script called, "FHN.m" that will allow you to change parameters. At the end of the script, you should plot the membrane voltage, V, versus time. The parameter to vary is the stimulus current I. First try, $I = 0$. Next try $I = 1.0$. Hint: You should see two very different behaviors. The stimulus current is in fact a bifurcation parameter of the system. Find the value of I at which the bifurcation occurs. Place this value in a comment at the bottom of your script "FHN.m".

CHAPTER 8

Conditional Logic

8.1 INTRODUCTION

Conditional logic is the use of true and false statements in programming. When given a statement it will either definitely be true or definitely be false. In computing terms, we can assign a "1" to any statement that is true and a "0" to any statement which is false. The two types of states, true or false, also goes by the name of Boolean logic.

The purpose of using Boolean logic is that it can be used to alter the flow of a program. Here we introduce idea of the *state* of the program, which is simply the values of all of the variables up to that point that are available in memory. Using this state, we can send the program down different pathways. Therefore, conditional logic allows a programmer to write a much more sophisticated and flexible program.

8.2 LOGICAL OPERATORS

To begin understanding how logical operators work, enter the following commands.

```
>> a = 5;
>> b = -1;
>> c = 0;
>> d = 3;
>> e = 3;
```

You can now think of the state as the variables $a - e$ in memory. Next enter the following commands one at a time to the command line. For more help, you may wish to type "help relop".

```
>> logical(a)   %true (1) because a has a value other than 0
>> logical(c)   %false (0) because c has a numerical value of 0
>> a==b         %false because a is not equal to b
>> b~=c         %true because b is not equal to c
>> d>e          %false because d is equal to d
>> d>=e         %true because d is equal to e
>> d!=b         %true because d is not equal to b
>> d<a          %true vecause d is less than a
```

And, logical statements can be combined together using && (logical AND) and || (logical OR)

```
>> (a==b)||(d==e)       %True because d is equal to e
>> (a==b)&&(d==e)       %False because a is not equal to b
>> (a >= d)&&(c)        %false because c is logically 0
```

8.2.1 RANDOM BOOLEANS

Logical operations can be used to create a number of interesting data structures. One that has been studied extensively by mathematicians, and has applications in systems biology, is a description of a randomly connected network. In Chapter 4, we created an adjacency matrix for a ring. Using logical operations, we can create the adjacency matrix for randomly connected nodes.

```
>>N = 100;           %Number of Nodes
>>Temp = rand(N,N); %Temp filled with numbers between 0 and 1
>>A = Temp>0.5;     %Any value >0.5 becomes 1, any value < 0.5 becomes 0
>>spy(A)
```

Because of the random distribution between 0 and 1, and the value 0.5, the matrix will be half filled of 0s, with the other half filled with 1s. Remembering that a 1 indicates a connection; this means that each node is connected to, on average, half of the other nodes. To change the number of connections, all that is necessary is to change the line "A = Temp>0.5".

8.2.2 LOGICAL OPERATIONS ON STRINGS

A second situation where logical operations can be helpful is in checking if two character strings are the same. For example, such an operation may be very useful in searching or sorting the patient database created in Section 4.6.1. Matlab has a number of commands specifically for

```
>>PatientName1 = 'John Doe';
>>PatientName2 = 'Jane Doe';
>>strcmp(PatientName1,PatientName2)
>>strcmp(PatientName1,'John Doe');
```

For more logical operations on strings, view the help for *strcmp*.

8.2.3 LOGIC AND THE FIND COMMAND

In an exercise in Chapter 5, you created a vector that contained the first 7 minutes of heart rate recordings.

```
>>HeartRateData = [0  64  137  188  260  328  397  464];
```

Now you would like to find the minutes when the heart rate went above 70.

```
>> highHRminute = find(diff(HeartRateData)>70)
```

In one command, using some logic and the *diff* and *find* commands, we can identify the minutes where the heart rate when above 70.

8.3 IF, ELSEIF AND ELSE

Logical commands can be used to control the flow of a program using the *if*, *elseif* and *else* structures in Matlab. Do not enter the commands below.

```
x = 2;

if (x>1)
    COMMANDS HERE FOR x GREATER THAN 1
elseif (x==1)
    COMMANDS HERE FOR x EQUAL TO 1
else
    COMMANDS HERE FOR x LESS THAN 1
end
```

In this template, code we would issue the commands in "COMMANDS HERE FOR x GREATER THAN 1", because $x = 2$. It is important to note that there can be any number of commands.

8.3.1 THE INTEGRATE AND FIRE NEURON

One of the most simplistic models of an excitable cell, *e.g.*, neuron and muscle, is known as the integrate and fire model. The model has two phases: 1) a period where it will integrate any electrical input and charge up the cell membrane, and 2) a period when the cell produces a spike in membrane potential and then resets back to rest. During the charging phase, we can have the cell obey a simple differential equation for an RC circuit.

$$\frac{dV}{dt} = I - \frac{V}{R \cdot C} \tag{8.1}$$

where R and C are the membrane resistance and capacitance. I is the current entering the cell. By looking at the equation, if I is a constant current input, $V(t)$ will rise up to some steady-state value (exactly like the charging of an RC circuit). To switch to the second phase, we must define a threshold voltage, V_t. When the cell membrane voltage reaches V_t, the program will stop using the differential equation and then do two things. First, a "spike" will be issued. The meaning of the spike is to set V to some constant value V_{peak} for only that time step. Second, on the following time step, V will be reset back to an initial value of 0. After the reset, the cell membrane is ready to be charged again. These ideas can be captured in the following code called, "IAF.m".

```
dt = 0.01;
EndTime = 50.0;
time = 0:dt:EndTime;

Vt = 5;          % Threshold voltage
R = 1;           % Set to 1 for simplicity
C = 2;           % set to 2 for simplicity
```

```
I = 3;
Vpeak = 100;    % set membrane voltage of spikes

V = zeros(length(time),1);
V(1) = 0;

for i=2:length(time)
    if (V(i-1)>Vt)
        V(i-1) = Vpeak;
        V(i) = 0;
    else
        V(i) = V(i-1) + dt*(I-V(i-1)/(R*C));
    end
end

plot(time,V);
```

You should notice in your plot that the membrane charges to $V = 5$, generates a spike and then returns to 0. Upon returning to zero, it will begin charging again. The lines

```
if (V(i-1)>Vt)
    V(i-1) = Vpeak;
    V(i) = 0;
else
```

require some explanation because a trick was used here. At a particular iteration through the loop, the loop variable has the value of i. When we perform the logical statement $V(i-1) > V_t$, we are really checking if the previous value was above threshold. If the statement is true we want the previous value to instead be replaced with a spike, thus the statement $V(i-1) = Vpeak$. Then we want the current value of V to be reset (V(i)=0). This appears strange because we are going back to a previous value of V and then changing it. There are other ways to write this section of code, but this is much more efficient.

As a test of your script, you should slowly increase I, observing how the pattern of spikes changes. You should notice that as I increases, the rate of firing increases. This is exactly what occurs in most neurons in the brain. If you then decrease I below some level, no firing will occur at all. This behavior is also observed in neurons.

8.3.2 CATCHING ERRORS

In Section 6.3.2, we learned how to create an *error* in Matlab, which will display red text and terminate the script. The combination of *error* and *if-else* logic can be used to catch problems in a function and report the problem to the user. For example,

```
x = 1;
a = 5;
if (a>0)
    b=a/x;
else
    error('Divide by Negative Number Not Allowed');
end
```

8.3.3 FUNCTION FLEXIBILITY

You may have noticed that some Matlab functions can take a number of different arguments. For example, you can call the function *mean* in two different ways.

```
>> mean(A);
>> mean(A,2);
```

Inside the function (mean.m), the program must first determine if there are 1 or 2 input arguments. At that point, it will determine how to proceed. In every function, there is a built-in variable that is defined *nargin* (look at the help for *nargin* for other useful function commands) that contains the number of input arguments. You should try *type mean.m* to see how the function was written to take into account the two different ways of called *mean*.

8.3.4 WHILE LOOPS

In the previous chapter, the *while* loop was introduced as a way to begin an iteration without knowing exactly how many times it should cycle through. As pointed out, however, every loop must have some way to terminate. In the case of the while loop, we will continue to iterate until some logical condition is met

```
counter = 1;
while (counter<=25)
    counter = counter+1
end
```

In the above code, we set *counter* equal to 1. On the first time through the while loop, we check if *counter* is less than or equal to 25. If it is, then we proceed through the next iteration. But, within each iteration, *counter* is being incremented by 1. Again, at the beginning of each iteration, the condition *counter* $<=$ 25 is checked. When *counter* becomes 26, the condition is not met and the *while* loop terminates.

8.3.5 STEADY-STATE OF DIFFERENTIAL EQUATIONS

The above section demonstrated how a *while* loop uses logical statements to terminate a loop. It should be noted that any logical statement can be used, and that the loop will terminate when the statement becomes false. To demonstrate a more practical reason to use a *while* loop we will

consider iterating through the Kermack and KcKendrick differential equation model for the spread of a disease.

$$\frac{dS}{dt} = -\beta SI \tag{8.2}$$

$$\frac{dI}{dt} = \beta SI - \gamma I \tag{8.3}$$

$$\frac{dR}{dt} = \gamma I \tag{8.4}$$

where S is the population of Susceptible, I is the population of infected and R is the population of recovered. β is a constant that reflects the number of susceptible that become infected each time step. γ is a constant that reflects the number of infected which recover each time step. The script, "SIR.m" below, uses a *while* loop to check when the variable S does not change.

```
dt = 0.01;
change = 0.001; %how to define stop condition
beta = 0.003;
gamma = 0.01;

%initial values
S = [0 100];
I = [0 2];
R = [0 0];

i = 2;

while (abs(S(i-1)-S(i))>change)
    i = i+1;
    S(i) = S(i-1) + dt*(-beta*S(i-1)*I(i-1));
    I(i) = I(i-1) + dt*(beta*S(i-1)*I(i-1)-gamma*I(i-1));
    R(i) = R(i-1) + dt*(gamma*I(i-1));
end

figure(1)
hold on
time = dt*[1:length(S)];
plot(time,S)
plot(time,I,'k');
plot(time,R,'r');
```

The reason for the *while* loop only becomes apparent as the values for β and α are changed. First, change "figure(1)" to "figure(2)" so that you can view both figures side-by-side. Then change $\beta = 0.3$

and rerun "SIR.m". You should notice that the time on the y-axis is drastically different in the two figures. If we used a *for* loop, we would not know ahead of time how many iterations to execute before S reached a constant value. But, with a *while* loop, we can explicitly check when S reaches a steady-state.

8.3.6 BREAKING A LOOP

A for loop should be used when it is known how many times an operation should be performed. A while loop should be used when an operation should be repeated until some condition is met. There are times, however, when a loop is necessary that does not fit easily into either of these predetermined types. To help gain more flexibility, Matlab has a *break* command. Note that the break command typically would be placed inside of an *if* statement within a loop.

```
counter = 1;
for i=1:100
    counter = counter+1;

    if(counter>=76)
        break;
    end
end
counter
```

The code above will terminate the loop early when *counter* reaches 76.

8.3.7 KILLING RUNAWAY JOBS

Matlab can sometimes get carried away with an operation, usually if a command in the loop uses too much memory or a loop becomes infinite. In these instances, it is helpful to be able to kill whatever Matlab is doing. Although any work that has been performed will be destroyed, you may find that it is useful in some cases. To suspend a process, press *Control C*.

8.4 SWITCH STATEMENTS

The if, elseif, else format is very good to use if you need to direct your program in only a few directions. Here you can imagine the flow of your program proceeding down a main trunk and then reaching a branch point. If there are only two or three branches, you can easily use if-else statements. If the branch that occurs in the code must go in more than three directions, there is another type of logical structure that can be very helpful - the *switch* structure.

Let us assume that you wish to study how neurons might synchronize to one another, and how hypersynchronization may lead to epilepsy. You have the dynamics of the individual neurons already coded, probably in the form of a difference or differential equation. Now you want to test out

how different networks of neurons might synchronize. The task of the study is to determine if some networks are easier (or harder) to synchronize. For this study, you will need to create a variety of different networks. You would like to try a one-dimensional ring (see Chapter 4), a two-dimensional grid with Von Neumann connections, randomly connected neurons (see Section 8.2.1) and what is known as a small world network. Here you could define a variable *NetworkType* that contains a character string. Then you could include a switch statement that will send the code in different directions, depending upon the variable *NetworkType*. You do not need to enter the commands below.

```
NetworkType = '2DGrid';

switch(NetworkType)
case {'1DRing'}
    N = input('Enter Size of Ring:');
    Network = %Create Ring Network Adjacency Matrix Here
case {'2DGrid'}
    [Nx,Ny] = input('Enter number of rows and columns');
    Network = %Create 2D Diffusion-type Adjacency Matrix Here
case {'Random'}
    N = input('Enter number of random points:');
    Network = %Create Random 2D Adjacency Matrix Here
case{'SmallWorld'}
    N = input('Enter number of nodes in small world');
    Network = %Create Small World Network Here
otherwise
    disp('Please Enter a Valid Network Type');
end
spy(Network)
```

In this instance, you used a character string as the variable that controls switching. But, you could use any data type in Matlab. You could also go back to this code at some later time and easily add another type of network, *e.g.,* A Moore Neighborhood for the 2D grid.

8.5 EXERCISES

1. Turn in "IAF.m", "SIR.m".

2. A great deal of study has gone into how patterns are generated in nature, for example, the spots on a leopard, stripes on a Zebra or more intricate patterns on some sea shells. It would be tempting to attribute these patterns to genetics, but experiments have shown that the patterns are very sensitive to environmental conditions present at specific stages of development. For example, a mollusk that is placed in a colder environment may display a very different pattern.

As such, theoretical biologists proposed that the genetics lay down some very simple rules that in slightly different situations can lead to drastically different patterns.

One of the simplest pattern generating mechanisms was discovered in the early 1980s by Steven Wolfram. It is a subclass of mathematical systems called cellular automata. Imagine a one-dimensional line composed of a number of individual elements. Each element can be either on (logical 1) or off (logical 0). We can then make a second one-dimensional line below the first line. But, we can make the on and off pattern of this second line dependent upon what is on or off in the first line. Then we can add a third line, with values dependent upon the second line. We can continue this pattern indefinitely. In this case, the first line is like a boundary condition, and then we simply follow some rules to continue adding lines.

Figure 8.1 shows the Wolfram rules that look to only three elements of the previous line: the element directly above you, the element above and to the left, and the element above and to the right. You should note that, in the figure, the 8 combinations in the top row are the only possible combinations of three elements. These combinations are always given in the same order. The rule shown is [0 1 1 0 1 1 1 0], which completely specifies the result for all possible combinations of the three values in the previous one-dimensional line. A bit of calculation will show that there are 256 (2^8) possible rules. Wolfram explored all 256 and showed that some are boring, some interesting, and others are very unexpected. Below is code, which you should enter into a script called, "WolframCA.m", that implements the [0 1 1 0 1 1 1 0] rule.

```
L = 300;  %length of one dimensional line
T = 300;  %number of one dimensional lines to add

A = zeros(T,L);   %Initialize A matrix to store CA
A(1,L) = 1;       %Initial Condition

for i = 2:T
    for j = 2:L-1      %Don't loop over the two end points

        l=A(i-1,j-1);   %left at previous time
        m=A(i-1,j);     %this node at previous time
        r=A(i-1,j+1);   %right at previous time

        % Use logic to go through all 8 cases
        if (l && m && r)
            A(i,j) = 0;
        end
```

```
            if (l && m && ~r)
                A(i,j) = 1;
            end

            if (l && ~m && r)
                A(i,j) = 1;
            end

            if (l && ~m && ~r)
                A(i,j) = 0;
            end

            if (~l && m && r)
                A(i,j) = 1;
            end

            if (~l && m && ~r)
                A(i,j) = 1;
            end

            if (~l && ~m && r)
                A(i,j) = 1;
            end

            if(~l && ~m && ~r)
                A(i,j) = 0;
            end

        end
    end

    colormap(gray(2));
    image(2-A);
```

Explore different rules by changing the values for $A(i, j) =$ terms. You should note that even one change can completely alter the patterns generated. At the bottom of your script, you must report (in comments) the types of behavior you observed.

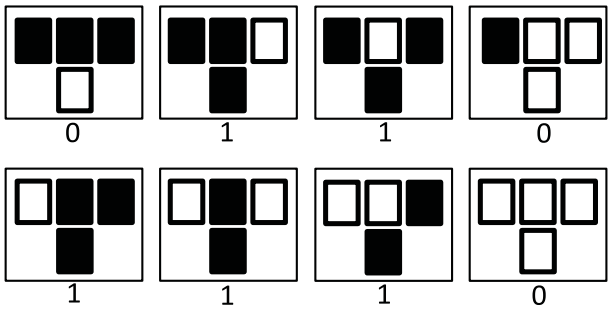

Figure 8.1: Example of Wolfram Cellular Automaton Rules.

CHAPTER 9

Data In, Data Out

9.1 INTRODUCTION

In previous sections, it was shown how, using the *save* and *load* commands, we could easily store and then restore anything in Matlab's memory. But, most powerful programming languages have some mechanism for reading in data from other applications, as well as some way to write data out that can be read by other applications. For example, you may wish to create geometries (adjacency matrices) or initial conditions in Matlab, but then send these files to another program that will run on a supercomputer, *i.e.,* on many computers in parallel. In this way, you could run an enormous biomedical simulation with millions (or maybe even billions of points) in a compiled, *i.e.,* faster than matlab, program. Alternatively, you may receive data from a large simulation and need to read data into matlab to analyze it. In this chapter, you will learn how to read in and write out data to and from matlab.

9.2 BUILT IN READERS AND WRITERS

The problem with the ".mat" file format is that you cannot open the file in anything other than matlab. Matlab has a number of functions that allow for files to be read and written in other formats. For example, there are two commands, *xlsread* and *xlswrite*, that allow matlab to share files with Excel.

```
A = rand(20,30);
xlswrite('FirstExcelExample.xls',A);
```

There are many options to use with *xlswrite*, and you should view them if you need to perform a more sophisticated function. Matlab also has a method of reading from an excel file.

```
[Numeric,Txt,Raw]=xlsread('FirstExcelExample.xls');
```

It should be noted that for communication with external programs, it is important to have the proper version of matlab. It is possible that your version may not support all of the input and output formats explained in this chapter.

An even more basic file is known as a flat text file. Here the data is simply arranged in a text file in columns and rows. Enter the following numbers into a file called "MyFirstTextFile.txt".

```
1   2   3   4   5
6   7   8   9   10
11  12  13  14  15
```

To load the following into matlab you would type

```
[B] = load('MyFirstTextFile.txt');
```

In the file above, matlab used spaces (or tabs) to make clear where the breaks are between numbers. Here the spaces (or tabs) were used as deliminators. A deliminator is simply any character that separates other characters. For many reasons, it may have been more clear for the file to have had the format

```
1:2:3:4:5
6:7:8:9:10
11:12:13:14:15
```

where ":" is being used as a deliminator. Matlab has a variety of commands for reading (*dlmread* and *textscan*) and writing (*dlmwrite*) these types of files.

For a more complete list of all the types of file formats supported in matlab, view the help for *fileformats*.

9.3 WRITING ARRAYS AND VECTORS

In the above example, we created a relatively small matrix (A) that could easily be stored in memory. Then we could simply send the entire matrix to a file. In the context of a simulation, however, we might be generating the values as we progress throughout a simulation, and we generally do not need to know all values at all times. In these situations, there is no need to store all of the data in memory. Rather, we can send the data to a file on the hard drive as it is generated. We will now take a bit of a detour to create a simulation that generates more data than can be stored in memory, and therefore requires the creation of an external data file.

9.3.1 DIFFUSION MATRICES

To demonstrate a very generic situation where writing to a file is very helpful, we will examine an important concept in the movement of any group of particles that is conserved as it flows through a two-dimensional grid. The particles could be ions, animals of a species, a volume of fluid, or even something less tangible such as heat energy. The key is that given a type of particle, q, we can define a flow rate, $\frac{dq}{dt}$. In electrical circuits the conserved quantity is charge (q) and the flow is current, I.

$$I = \frac{dq}{dt} \tag{9.1}$$

Because the particles (charge ions in this case) must be conserved, we can total up all of the charge entering a point and all the charge leaving a point, and they must be equal.

$$I_{in} = I_{out} \tag{9.2}$$
$$0 = I_{in} - I_{out} \tag{9.3}$$

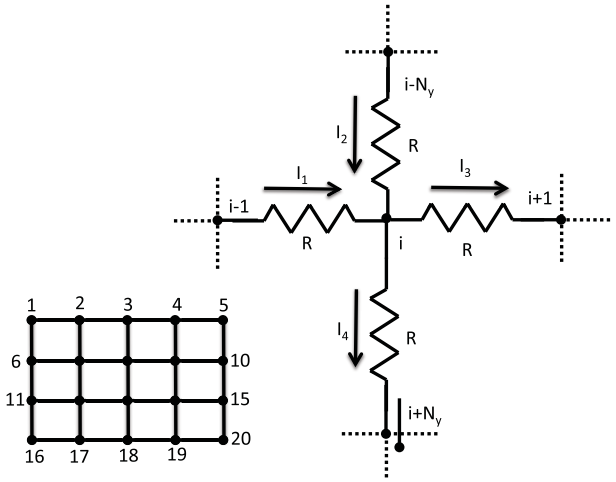

Figure 9.1: Two dimensional resistor grid, showing the neighbors of a generic node i

Figure 9.1 shows a small portion of a large two-dimensional grid. Each point in the grid is connected to its left, right, up and down neighbors and may share charge with only those nodes. Therefore, current may flow left, right, up or down through some resistance, and if we total up the currents entering and leaving any node, they should sum to zero. Given the directions of our arrows, which are arbitrary but should be consistent for every node, we can derive

$$I_1 + I_2 - I_3 - I_4 = 0 \tag{9.4}$$

Using Ohm's Law, we can reexpress the above equation in terms of the voltages at the neighboring nodes (V) and resistances between nodes (R)

$$\frac{V_{i-1} - V_i}{R} + \frac{V_{i-N_y} - V_i}{R} - \frac{V_i - V_{i+1}}{R} - \frac{V_i - V_{i+N_y}}{R} = 0 \tag{9.5}$$

$$\frac{V_{i-1} - 2V_i + V_{i+1}}{R} + \frac{V_{i-N_y} - 2V_i + V_{i+N_y}}{R} = 0 \tag{9.6}$$

$$\frac{V_{i-N_y} + V_{i-1} - 4V_i + V_{i+1} + V_{i+N_y}}{R} = 0 \tag{9.7}$$

where subscripts denote the node numbers. We have used i as the node number to indicate that our analysis will work for any node in the grid. The left and right neighbors therefore have subscripts $i - 1$ and $i + 1$. The node numbers for the up and down neighbors are offset by the number of nodes in each row of the grid.

If we were to write out an equation for each node in the grid, we would have a very similar form to the equation above. The node itself would have a coefficient of $-4/R$, and the neighbors

would each have coefficients of $1/R$. The nodes at the corners and at edges will have slightly different equations, but they could easily be derived by the same conservation of current laws. In fact, the voltages and coefficients can be decoupled and take the form of

$$\mathbf{Dv} \tag{9.8}$$

where D contains the coefficients and v contains a vector of the voltages. Nearly all physical quantities have some sort of conservation law, and therefore our analysis applies to much more than electronic circuits.

The code below creates a diffusion matrix, T, for a grid that is $M \times N$. You should save the file as "FHNPropagate.m". The code is admittedly long. You may wish to read the text after the code to gain some overall understanding of how it works. Then, as you write each line, you should be thinking about how it fits into the overall context of the code.

```
M = 5;  %Number of row
N = 5;  %Number of columns

T = zeros(N*M,N*M);

for i = 1:M     %Loop over rows
    for j=1:N   %Loop over columns
        %Computer node number assumed numbering
        %starts going across rows
        Node = (i-1)*N+j;

       %All Interior Nodes
       if ((i>1) && (j>1) &&(i<M) &&(j<N))
            T(Node,Node-1) = 1;
            T(Node,Node+1) = 1;
            T(Node,Node+N) = 1;
            T(Node,Node-N) = 1;
            T(Node,Node) = -1.0*sum(T(Node,:));
       end

       %Top Boundary
       if(i==1)
           if (j==1)    %Upper left corner
                T(Node,Node+1) = 2;
                T(Node,Node+N) = 2;
                T(Node,Node) = -1.0*sum(T(Node,:));
```

```
    elseif(j==N) %Upper right corner
         T(Node,Node-1) = 2;
         T(Node,Node+N) = 2;
         T(Node,Node) = -1.0*sum(T(Node,:));
    else         %Top edge
        T(Node,Node+1) = 2;
        T(Node,Node-1) = 1;
        T(Node,Node+N) = 1;
        T(Node,Node) = -1.0*sum(T(Node,:));
    end

end

%Bottom Boundary
if (i==M)
 if(j==1)    %Lower left corner
         T(Node,Node+1) = 2;
         T(Node,Node-N) = 2;
         T(Node,Node) = -1.0*sum(T(Node,:));
 elseif(j==N)    %Lower right corner
         T(Node,Node-1) = 2;
         T(Node,Node-N) = 2;
         T(Node,Node) = -1.0*sum(T(Node,:));
 else            %bottom edge
         T(Node,Node-1) = 1;
         T(Node,Node+1) = 1;
         T(Node,Node-N) = 2;
         T(Node,Node) = -1.0*sum(T(Node,:));
 end
end

%Left Boundary
if((j==1)&&(i~=1)&&(i~=M))
         T(Node,Node+1) = 2;
         T(Node,Node-N) = 1;
         T(Node,Node+N) = 1;
         T(Node,Node) = -1.0*sum(T(Node,:));
end
```

```
            %RightBoundary
            if((j==N)&&(i~=1)&&(i~=M))
                      T(Node,Node-1) = 2;
                      T(Node,Node+N) = 1;
                      T(Node,Node-N) = 1;
                      T(Node,Node) = -1.0*sum(T(Node,:));
            end

        end
    end

    spy(T)
```

There are a few points that are important to note. First, we have created a 5×5 grid in this example but could easily change M and N. Second, the numbering of nodes is to start with 1 and move across a row until reaching $Node = 5$. Then the numbering starts up again on the next row and continues until $Node = 10$. Although there are nested loops (loop over rows and then columns), we can, for any i and j, compute the node number (stored in $Node$). It then makes it much easier to reference any other node to the current node. For example, the node to the left will be $Node - 1$, the node to the right will be $Node + 1$, the node above would be $Node + N$ and the node below will be $Node - N$. Inside the loops are a series of *if* statements that handle interior, edge and corner nodes. Although we will not go into the theoretical rationale, it is convention for the coefficient to be "2" when a neighbor does not have a *pair*. What we mean by a pair is best illustrated by an example. For nodes on the left edge, there is no connection between $Node$ and $Node - 1$ (because it is on the other side of the grid). In this case, we then make $Node + 1$ count twice. You may notice that the first *if* statement handles all of the interior nodes, while all the other *if* statements handle the edge and corner nodes. The last command ($spy(T)$) will allow you to visualize the entries. You should note that a diffusion matrix is a special type of adjacency matrix.

The last point is that from the *spy* command, T is sparse and all entries are along 5 diagonal lines. For this reason, the above code could have been greatly condensed, *i.e.,* optimized, by removing the *if* commands and replacing them with the appropriate *diag* commands. The code was created as above for illustration purposes. A further reduction in memory could be achieved by using the *sparse* command, as introduced in Section 4.2.2.

9.3.2 EXCITABLE MEMBRANE PROPAGATION

The heart is often thought of as a mechanical pump. But the control and coordination of the pump is largely achieved through electrical communication between neighboring cells. In fact, we can use a diffusion matrix to explain how ionic currents are shared between cells. In this section, we will combine the diffusion matrix above and the excitable FitzHugh-Nagumo model introduced in Chapter 7.

You should add the following lines to "FHNPropagate.m".

```
dt = 0.01;
EndTime = 500;
time = 0:dt:EndTime;

a = 0.08;
b = 0.7;
c = 0.8;
I = zeros(M*N,1);       %Simulus Current Vector
I(1) = 0.6;             %Stimulate only Node 1

V = zeros(M*N,1);       %Set up Vector to hold V
W = zeros(M*N,1);       %Set up Vector to hold W
VOld = zeros(M*N,1);    %Set up Vector to hold V
WOld = zeros(M*N,1);    %Set up Vector to hold W

fid = fopen('FHNProp.txt','w');

for i=2:length(time)
    V = VOld + dt.*(VOld-(VOld.^3)/3-WOld+I - T*V);
    W = WOld + dt.*(a.*(VOld+b-c.*WOld));

    fprintf(fid,'%f\t',time(i));  %Print out time
    for k = 1:M*N                 %loop over V values
        fprintf(fid,'%f\t',V(k));
    end
    fprintf(fid,'\n');            %Go to next line

    VOld = V;
    WOld = W;

end
fclose(fid);
```

You should run the code above. After completion, you should see a file "FHNProp.txt" in your directory. We will first discuss how this implementation of the FitzHugh-Nagomo model is different from the one in Chapter 7. The major difference is that rather than simulate one FitzHugh-Nagomo cell we are simulating one at every grid point ,*e.g.,* N $\times$ M = 25. For this reason, we have assigned each cell a number, corresponding to the *Node* number discussed above.

In numbering each cell, we can create a vector V and vector W that hold the values at all 25 cells at any one time. You will also note that we created two additional vectors, $VOld$ and $WOld$, which are temporary vectors that will be used in calculations. Within the time loop, we now perform an update on the differential equations for V and W for all 25 nodes in one step. This is an excellent example of how Matlab's matrix and vector operations can help make code easier to write, read and run more efficiently. You should note that the left side contains all V-terms and the right side contains all $VOld$-terms. This is done so that we do not mix up previous and current values. For example, we want to use the same $VOld$ in the updating of V as we use in the updating of W. You will also note that a new term, $-T * V$ has appeared in the update for V. This is a vector that describes the current entering (or leaving) nodes from their neighbors.

You should note that the methods used above could be used to describe nearly any interacting nodes, described by any adjacency matrix. In this way, we can decouple the dynamics at any node from the way nodes are connected together in a network.

We are not saving the value of V in memory at every node at every time step. Rather, we are generating a new value of V and then sending it to a file using the *fprintf* command. First, before the time loop begins, we open a file using the *open* command. The input to the command is the filename with the option "w", meaning that this files is being opened for writing. Within the loop, the *fprintf* command is used to send data to the file, and then after the loop is over, the file is closed using the *fclose* command. The variable fid is called a file identifier. This file identifier can become very important if more than one file is open at any one time, as it will make clear which data is going to which file.

The *fprintf* command has a very particular format. The first input is the file identifier. The second input is the format of the text that will be sent to the data file. The third input is the value of any variables. You should view the help for *fprintf* for details. In the example above, before the k-loop, the value for $time$ is printed. The format is "%f\t", meaning that a numerical double, %f will be printed, followed by a tab (\t). The third option is what places the value of $time$ into the place of %f. The loop moves though the V vector, printing a particular value of V followed by a tab. After the loop is printed, a return character,\n, is printed Therefore, on each iteration through the time loop, a new line is created in the open file that contains all of the values of V, separated by tabs.

The command *fprintf* is the most general and powerful way to print out data, but there are other commands, *e.g., sprintf*, that can be useful and more efficient.

9.4 READING IN ARRAYS AND VECTORS

The *load* command introduced earlier can be used to read in saved matlab work spaces. It can also be used to read in text files that were created in text format.

```
Y = load('FHNProp.txt');
```

will load the entire data file "FHNProp.txt" into the matrix Y. We could then plot values against one another. For example,

```
plot(Y(:,1),Y(:,2))
```

will plot the time vector, Y(:,1), against the first node, Y(:,2). The *load* command is very limited and if a more complex text file must be loaded in, *e.g.*, a delimitated text file, the *textscan* is the command to use.

9.4.1 IRREGULAR TEXT FILES

In some situations, a file does not have a regular layout. Enter the following into a text file, called "IrregularText.txt", using a text editor or Matlab's editor.

```
FHN Parameters
0.01 500
0.08 0.7 0.8
0.6
```

To read in the text above, enter the following code into a script called "IrregularTextReader.m"

```
fid = fopen('IrregularText.txt','r');
FirstLine = fscanf(fid,'%s %s',2);

Temp = fscanf(fid,'%f %d',2);
dt = Temp(1);
TimeSteps = Temp(2);

Temp = fscanf(fid,'%f %f %f',3);
a = Temp(1);
b = Temp(2);
c = Temp(3);

I = fscanf(fid,'%f',1);

fclose(fid)
```

The *fscanf* command will read one line at a time in the format specified. You should view the help file for more details.

9.5 READING AND WRITING MOVIES AND SOUNDS

Movies are simply a series of images played in sequence. In previous chapters, we have created a simple type of movie using a loop and the *pause* command. But, matlab also has support for creating

standalone movies that can be run outside of matlab, *i.e.,* in a presentation. To demonstrate the ability to create movies, we will create a script that simulates John Conway's Game of Life. The game of life is played on a grid where each element in the grid has eight neighbors, the usual up, down, left and right, but also the four diagonals. Each element can be on (1) or off (0) at any one time. We begin with an initial condition (pattern of 1s and 0s on the grid). Then to get to the next time step, we apply two simple rules to each element. If an element is on and either 2 or 3 of the neighbors are also on, that element will be on the next step. Otherwise, the node is turned off. If an element is off but exactly 3 of its neighbors are on, it will be on the next step. Otherwise, the node remains off on the next step.

Conway's rules are called the game of life because we can think of the rules as corresponding to some real situations. If a node is "alive", it could die by two different mechanisms. First, it could die of "loneliness" if not enough of its neighbors are on. Second, it could die if it is crowded out. It will only survive if 2 or 3 of its neighbors are on (not lonely but not overcrowded either). On the other hand, 3 neighbors can "reproduce" and give rise to a new alive element. Below is code which will play the Game of Life on a 50×50 grid for 100 iterations. The initial condition is set up such that activity will not simply die out. There are many other patterns that could be created, and you could experiment with different initial conditions as an exercise. Below is a script called "GameOfLife.m".

```
% GameOfLife.m - Conway's Game of Life

%Set up Parameters of Simulation
Dim=50;
T = 100;
Delay = 0.1;

%Initial Conditions
GRID = zeros(Dim,Dim);
GRID(10,10:11)=1;
GRID(11,11)=1;
GRID(11,15:17)=1;
GRID(9,16)=1;


%the world is round
up=[2:Dim 1];               %Establish pattern 1 counting up
down=[Dim 1:Dim-1];         %Establish pattern 2 counting down

for i=1:T
    neighbours=GRID(up,:)+GRID(down,:)+GRID(:,up)+GRID(:,down)+...
```

```
        GRID(up,up)+GRID(up,down)+GRID(down,up)+GRID(down,down);
    GRID = (neighbours==3) | (GRID & neighbours==2);

    imagesc(GRID);
    M(i,:) = getframe;
    %M(i) = getframe;       %May be need for some Matlab versions
    pause(Delay);
end
```

The purpose of the script above is to demonstrate how to capture the dynamics in a movie. You will notice that after the *imagesc* command displays $GRID$ there is a command $M(i) = getframe$. The *getframe* command will store the current figure in a special movie frame structure. If you run the script and type *whos*, you will notice that a very large structure, M, has been created. To play the movie in matlab

```
>> movie(M)
```

You can view the *movie* command to see other options for how you can play back the movie you have created. The *movie* command, however, will only work within matlab. To export the movie so it can be played outside of matlab

```
>> movie2avi(M,'CATestMovie.avi');
```

which will create an "avi" file that can be played with many external movie players, including inside a power point presentation. Depending upon how matlab was set up, you may get a warning when you issue this command. You should still see the file "CATestMovie.avi" in your directory. Try to play it using RealPlayer or Windows Media Player. You should view the help for the *movie2avi* command to learn how to change options such as frame rate, compression and quality. There is also an "mpeg" writer that can be downloaded from the Mathworks website.

9.5.1 SOUNDS

Matlab also has support for recording, *e.g.*, *wavwrite*, *wavrecord*, and playing, *e.g.*, *sound*, *waveplay*, *wavered*, and wave. To test your system, try entering

```
>> load handel
>> sound(y,Fs);
```

To learn more about the support for movies and other audio/visual commands, view the help for *audiovideo*.

9.5.2 READING IN IMAGES

You have already learned how to write images out using the *print* command. Matlab also has commands for reading in images. First, we will create a "jpeg" image from one of Matlab's built in images.

```
>> load cape
>> imagesc(X)
>> colormap(map)
>> print('-djpeg','CapeCod.jpeg');
>> close all
```

You should note that the command *close all* will close all open figures. To load in the jpeg you created

```
>> A = imread('CapeCod.jpeg','jpeg');
>> imagesc(A)
```

The *imread* command can be used to load many different types of image files into a matrix. Once in Matlab's memory, the image can be displayed in the same way as any other matrix.

9.6 BINARY FILES

It is very simple to write out data to a text file using *fprintf* or other such commands. The problem is that the size of the text file can become very large because when a number, for example "0.0012345" is sent to the file, it is typically stored as a character string. Then when it is read back in, it is converted back to a numerical value. The problem should become apparent if you issue the following commands

```
>> a = ' 0.00123456789';    %character string
>> b = 0.00123456789;                  %numerical value
>> whos
```

Storing as a character requires 26 Bytes where as a floating point number requires only 8. So, there can be a very large savings in hard drive space if numbers are stored as numbers. This format is typically called *binary* because the numbers are stored in machine language (1s and 0s).

9.6.1 WRITING BINARY FILES

To read and write in binary requires a few different commands in matlab, for example *fwrite*. Copy "FHNPropagate.m" to "FHNPropagateBinaryOutput.m". Then change the line fid = fopen('FHNProp.txt','w'); to

```
fidb = fopen('FHNProp.bin','wb');
```

The option "wb" specifies that the file is writable and binary. Next replace

```
fprintf(fid,'%f\t',time(i));    %Print out time
for k = 1:M*N                  %loop over V values
    fprintf(fid,'%f\t',V(k));
end
fprintf(fid,'\n');             %Go to next line
```

with

```
fwrite(fidb,V,'double');
```

Lastly, change fclose(fid); to

```
fclose(fidb);
```

If you run "FHNPropagateBinaryOutput.m" a file, "FHNProp.bin" will be created. In addition to the file being somewhat smaller, you will note that there is the added benefit to writing out the data in binary - you no longer need the k loop. Rather, you can simply have matlab send the entire vector V to the file.

9.6.2 READING BINARY FILES

load and other commands will read in files saved in text format. Matlab also has commands to read in binary files. We will not expand upon these commands here, but you can analyze the following commands that read in a particular time step of "FHNProp.bin".

```
TimeStep = 200;
fid = fopen('FHNProp.bin','rb');

%Move from the beginning of the file to the timestep
fseek(fid,TimeStep*8*M*N,-1);

%Read in one time step worth of data in double format
Data = fread(fid,M*N,'double');

%Reshape Data to be an MxN matrix
Data = reshape(Data,M,N);

%display the image at timestep 200
imagesc(Data)
fclose(fid)
```

Of course, it is possible to read in the entire data file, by placing the commands above in a loop that iterates through $TimeStep$.

9.6.3 HEADERS

Nearly all files will be stored in one of two ways - text or binary. The key is to understand the format so that you can write a script to read the data. The format should be contained within the documentation for the program. A good file format will contain what is known as a header. In the header is important information, which would be helpful for reading in data. For example, in our previous example of storing binary data, it would be helpful to know the dimensions M and N, number of timesteps and the type of data. It would therefore have been helpful to first print out a text line at the top of the binary data file.

```
1000 500 200 double
BINARY DATA
```

The above file header would let a user know that the following binary data is 200, 1000×500 blocks of data that have the double format. In read or writing a header, you will mix Matlab's text and binary reading capabilities.

9.7 EXERCISES

1. Turn in "FirstExcelExample.xls", "FHNPropagate.m", "IrregularText.txt", "IrregularTextReader.m", "GameOfLife.m", "CATestMovie.avi", "FHNPropagateBinaryOutput.m".

2. Create a function that will compute the Fibonacci series starting with any two numbers. Remember that the series is defined by

$$F_n = F_{n-1} + F_{n-2} \tag{9.9}$$

   Your function should be of the form

```
function [Series]=FSeries(Num1,Num2,NumIterations);
```

3. Create a script,"WriteOutFibonacci.m" that generates a file with the following format

```
Fibonacci Series Starting with Num1 and ending with Num2
NumIterations
BINARY DATA HERE
```

   Num1, Num2 and NumIterations should be numerical values. Note that to create the text strings you may need to use the *num2str* command learned in Section 3.4. Note that in your script you must pick values for Num1, Num2 and NumIterations, then call the *FSeries* function to generate the vector *Series*. Then you can generate the header and write *Series* as binary data. Note that you should have Num1 and Num2 be some value other than 1, and NumIterations should be at least 100.

4. Create a function, "ReadInFibonacci" that will read in any file generated by "WriteOutFibonacci.m". The function should be of the form

```
function [Series]=ReadInFibonacci(filename);
```

CHAPTER 10

Graphics

10.1 INTRODUCTION

Built-in graphics is one of the key features of Matlab. In previous chapters, the *plot* and *imagesc* commands were introduced as ways of graphically displaying data. In this chapter, we will introduce more graphical options as well as explain some of the tools in Matlab for fine tuning graphics.

10.2 DISPLAYING 2D DATA

Two dimensional data is very often a simple line plot of an independent variable versus a dependent variable. For example, Schnakenberger (1979) gives a set of differential equations that describes an oscillating chemical reaction

$$\frac{dx}{dt} = x^2y - x \tag{10.1}$$

$$\frac{dy}{dt} = a - x^2y \tag{10.2}$$

where x and y are the chemicals and a is a constant parameter. The two equations above are non-linear, and so we could use the Euler Method to solve them. An alternative is to use a graphical non-linear dynamics technique where we plot all of the values for x and y that cause $\frac{dx}{dt} = 0$. In other words, we wish to plot the function

$$0 = x^2y - x \tag{10.3}$$

on an x-y axis for the first equation. We can do the same for the second differential equation. After a bit of algebra, we find that we must plot the following two functions

$$y = \frac{1}{x} \tag{10.4}$$

$$y = \frac{a}{x^2} \tag{10.5}$$

In non-linear dynamics, these two curves are called *nullclines*. $y = \frac{1}{x}$ is the nullcline of x, *e.g.*, the set of points where $\frac{dx}{dt} = 0$, and $y = \frac{a}{x^2}$ is the nullcline of y, *e.g.*, the set of points where $\frac{dx}{dt} = 0$. To make these plots, we can issue the following commands

```
a = 1;
x = -0.5:0.01:0.5;      %define a range for x
```

```
yxnull = 1./x;          %create x nullcline
yynull = a./(x.^2);     %create y nullcline
plot(x,yxnull,'r');     %plot x null in red
hold on
plot(x,yynull,'g');     %plot y null in green
```

You should first notice that the colors of the plots can be changed using an option to the *plot* command (view the help for *plot* for more color options). There are a few issues with the plot that is generated. First, there is a problem with the scales of the plots because some parts go to infinity. To rescale the axes, the following commands can be used

```
>> axis([-0.5 0.25 -10 100])
```

where the *axis* takes a vector that has the form $[x_{min}, x_{max}, y_{min}, y_{max}]$. You can therefore zoom in to any portion of your data to take a closer look.

There are times when you must distinguish your plot in some way other than to use colors. The plot command has a number of options for that as well.

```
a = 1;
x = -0.5:0.01:0.5;
yxnull = 1./x;
yynull = a./(x.^2);
plot(x,yxnull,'k');       %Make solid line
hold on
plot(x,yynull,'k-.');     %Make dash-dot line
axis([-0.5 0.5 -10 100])
```

Now both plots are in black, but the y nullcline is a solid line, and the x nullcline is a dash-dot line. You should view the help for *plot* to see the other options.

It is also important to add appropriate axis labels and a title for your graph by issuing the following commands

```
xlabel('x variable');
ylabel('y variable');
title('X and Y Nullclines for Schnakenberger (1979) Model');
```

There are times when it is useful to add grid lines

```
grid on
```

We will discuss in a future section how to have more control over the scale of the grid. To turn off a grid you can simply execute

```
grid off
```

You can do the same with the upper and right bounding boxes on the plots using the command *box*. By default Matlab has the box on, so

```
box off
```

will create a plot with the usual x-y axes. You may also want to adjust the aspect ratio of your plot, *e.g.,* relative lengths of the x-y axes. For example,

```
axis square
```

will create equal equal sizes for the x and y axes. See the *axis* command for more options.

10.2.1 FIGURE NUMBERS AND SAVING FIGURES

You may have noticed that if you simply issue a *plot* command, Matlab will automatically start with "Figure 1". If another plot command is issued, it will simply write over Figure 1. To start a new figure

```
>> figure(2)
```

A second problem is that we may want to create a figure but then add to it later. To illustrate how this can be accomplished, we will the examine the Hindmarsh Rose model for cortical neurons.

$$\frac{dx}{dt} = -ax^3 + bx^2 + y - z + I_0 \tag{10.6}$$

$$\frac{dy}{dt} = -dx^2 - y + c \tag{10.7}$$

$$\frac{dz}{dt} = rsx - rz - rsx_1 \tag{10.8}$$

where the values of a, b, c, d, r, s and x_1 are constants and I_0 is an externally applied current. We will assume that r is small and so z adapts slowly enough that we can treat it as a constant, *e.g.,* $z = 0$. Furthermore, we can assume there is no external stimulus, *e.g.,* $I_0 = 0$. Therefore, our set of equations becomes

$$\frac{dx}{dt} = -ax^3 + bx^2 + y \tag{10.9}$$

$$\frac{dy}{dt} = -dx^2 - y + c \tag{10.10}$$

$$\tag{10.11}$$

and we can then plot x and y nullclines using the following equations

$$0 = -ax^3 + bx^2 + y \tag{10.12}$$

$$0 = -dx^2 - y + c \tag{10.13}$$

$$\tag{10.14}$$

and therefore we must plot the functions

$$y = ax^3 - bx^2 \tag{10.15}$$

$$y = -dx^2 + c \tag{10.16}$$

$$\tag{10.17}$$

So we do not conflict with what has already been plotted in Figure 1; type the following into a script called "HMRModel.m".

```
figure(2)
```

which will open up a new (and blank) figure. Then type the following commands into the script

```
a = 0.3;
b = 2.5;
c = -1.;
d = 1.0;
x = -5:0.01:10;                 %define a range for x
yxnull = a.*x.^3 - b.*x.^2;     %create x nullcline
yynull = -d.*x.^2 + c;          %create y nullcline
plot(x,yxnull,'k');
hold on
plot(x,yynull,'k-.');
xlabel('x variable')
ylabel('y variable');
title('X and Y Nullclines for Hindmarsh-Rose Model')
axis([-5 10 -70 10])
```

You should note that, unlike the previous model, in the Hindmarsh-Rose model, there are three points where the nullclines intersect. The meaning of these intersections is that both $\frac{dx}{dt} = 0$ and $\frac{dy}{dt} = 0$, and therefore the $x - y$ values for these intersections are equilibrium points. We do not know from the plot if they are stable or unstable, but from the plot, we do know that they exist even without solving the equations numerically.

To finish off our plot, it would be good to add a legend. Add the following command to the end of your script.

```
legend('x nullcline','y nullcline');
```

The last step is to save your plot in a file on your hard drive. In previous sections, you created a "jpeg" file using the *print* command. You could easily change the options to create other file formats. But, Matlab has its own special file format (".fig") for images.

```
saveas(gcf,'HMR.fig');
```

The *saveas* command will save the current figure (specified by *gcf*, meaning "get current figure") into a file "HMR.fig". You should then close your figure. But you have saved all of the information in "HMR.fig". To bring the figure back into the Matlab environment

```
open 'HMR.fig'
```

will reopen the figure. You will see in later sections that you can continue to modify the figure. In this way, you can save work on any graphic and then reopen it at a later time.

10.2.2 VELOCITY MAPS

Non-linear differential equations, such at the Hindmarsh-Rose model may be solved numerically using integration methods such as the Euler Method. An alternative was provided by plotting the nullclines in state space. But, there is another way to gain a more intuitive understanding of the dynamics of differential equations.

```
a = 0.3;
b = 2.5;
c = -1.;
d = 1.0;
[X,Y] = meshgrid(-5:10,-70:10:10);  %Make a grid over x and y
xvec = -a.*X.^3 + b.*X.^2 + Y;      %create x vector
yvec = -d.*X.^2 -Y + c;             %create y vector
quiver(X,Y,xvec,yvec,0.3);          %Plot Velocity Vector Field
```

The command *quiver* generates an arrow at each point in the grid specified by X and Y that has a length of $xvec$ in the x direction and $yvec$ in the y direction. The extra value of 0.3 simply scales the vectors. The command *meshgrid* simply creates the X and Y matrices.

The meaning of a velocity map is that if we start the system, *i.e.,* initial condition, at any x-y point, the system will follow the velocity vectors. In fact, the length of the vectors even gives us some indication of how fast the system will move from one point to another. For this reason, velocity maps are sometimes called flow maps - we can think of an initial condition flowing around the state space.

10.2.3 LOG AND SEMI-LOG PLOTS

In 1935, George Zipf, a linguistics professor at Harvard, made an incredible claim. If you analyze a large volume written in English and keep track of the words used, a pattern emerges. We can rank the word used most often as 1, second as 2, third as 3 and so on. We can then count the number of times each word is used. If we plot the word rank, x, against the number of times used, $f(x)$, the data surprisingly fits

$$f(x) = ax^{-k} \tag{10.18}$$

where a and k are constants that can be determined from experimental data. What is more, Zipf and others found that it was not only English that followed this pattern, but nearly all other languages. US City populations were found to follow the same equation. So was the distribution of wealth in many different economies. The size of rivers, newspaper distributions, and even computer memory were found to follow the same trend. Another name for Zipf's Law is a Power Law. It was only a matter of time before biologists began to test if Zipf's Law applied to living systems.

Let us assume that we would like to plot a function that represents the ranking of the size of bloods vessel (*cm*) in the brain

```
a = 2;
k = 1.4;
x = logspace(1,3,100);
fx = a.*x.^(-k);
figure(1)
plot(x,fx)
figure(2)
loglog(x,fx)
```

The command *logspace* generates 100 points between 10^1 and 10^3. The command *loglog* plots $log(fx)$ against $log(x)$. You can verify that the log-log plot should be a straight line with a bit of analysis. Matlab also has support for semi-log plots in the commands *semiology* and *semiology*.

10.2.4 IMAGES

In an earlier section, the commands *imagesc* and *colorbar* were introduced. There are some additional commands that can allow for more flexibility.

```
minc = -2;      %minimum value
maxc = 5;       %maximum value

%random numbers between minc and maxc
X = minc + (maxc-minc).*rand(100,100);
X(50,50) = 100; %make one value very large
imagesc(X)
colorbar
```

The problem with these commands is that all of the values are between -2 and 5, except for the one value in the center. When *imagesc* is used, it automatically scales all of the values. To fix this problem, add the following command to the end of the script above

```
caxis([-2 5])
```

The last command, *caxis*, will rescale the color map from -2 to 5. The value of 100 will be treated as a value of 5, *i.e.*, it will appear red.

In our examples so far, we have used the default *colormap*. You should view the help for *colormap* and then issue the commands below to gain some idea of the types of colormaps that are available.

```
load flujet
imagesc(X)
colormap(winter)
```

and

```
load spine
image(X)
colormap bone
```

10.2.5 OTHER 2D PLOTS

Matlab has support for a wide range of other two-dimensional plots. Below we explore only a few. Enter the commands below to view each type of graphic. You can always view the help to learn more about any of the functions introduced below.

```
>> pie([2 4 9 10],{'Mutation 1','Mutation 2','Mutation 3','Mutation 4'});
>> pie3([2 4 9 10],{'Mutation 1','Mutation 2','Mutation 3','Mutation 4'});
>> load carsmall          %Loads in premade data on cars
>> boxplot(MPG, Origin)  % Note the error bars
```

Sometimes we also wish to display discrete data, *i.e.*, points are not connected together with lines.

```
    >> x = 0:0.01:1;    %Generate x vector
%Generate sinusoid with random noise
    >> y = 5*sin(2*pi.*x) + rand(length(x),1);
    >> stairs(y)
    >> stem(y)
```

At other times, you may wish to create a histogram that shows the frequencies of different types of events. Below is code to generate a random 1000 x 1000 adjacency matrix where each node is connected to 10% of the network.

```
>> A = rand(1000,1000);
>> A = A<0.1;
>> NumConnections = sum(A,2);   %CHECK THIS
>> hist(NumConnections,20)
```

The histogram shows how most nodes are connected to 10% of nodes (*100* other nodes), but because it is random, there is a distribution.

Lastly, Matlab will allow you to create *bar* charts that graphically compare many different types of data.

```
B = rand(10,5);
bar(B)
xlabel('Gene Mutation')
ylabel('Frequency')
legend('American','Canadian','Mexican','English','German')
```

The chart above may have been a comparison of the frequency of 10 different gene mutations in 5 different populations.

10.2.6 SUBPLOTS

There are occasions when it is convenient to display several plots side-by-side. In these instances, you will want to use the *subplot* command.

```
x = 0:0.01:1;    %Generate x vector
%Generate 4 noisy sinusoids
y1 = 5*sin(2*pi.*x) + rand(1,length(x));
y2 = 5*sin(2*2*pi.*x) + rand(1,length(x));
y3 = 5*sin(3*2*pi.*x) + rand(1,length(x));
y4 = 5*sin(4*2*pi.*x) + rand(1,length(x));

figure(1)
subplot(2,2,1)
plot(x,y1)
subplot(2,2,2)
plot(x,y2)
subplot(2,2,3)
plot(x,y3)
subplot(2,2,4)
plot(x,y4)
```

The help for the *subplot* command has more information about how to create more complex multi-panel figures.

10.3 FIGURE HANDLES

In previous sections, we generated a number of plots and learned to alter some aspects of the figure. To gain more flexibility over the appearance of figures, Matlab allows the user to output a figure handle. First, we must clear everything in memory and close all figures.

```
clear all       %clears everything in matlab's memory
close all       %closes all open figures
```

Next, we will create a figure that contains a sinusoid

```
x = 0:0.001:1;
h = plot(x,sin(2*pi*x));
```

where *h* is the figure handle.

```
get(h)
```

The command *get* will display all of the options in the figure handle. To get a particular option, you can issue

```
get(h,'LineStyle')
```

We will not discuss all of the options, but there are some that are useful to know how to change. To change an option, you can use the *set* command

```
set(h,'LineWidth',3);
```

which will change the line width from the default of 0.5 to 3 (make the line thicker).

10.3.1 THE HIERARCHY OF FIGURE HANDLES

In the above section, we only showed some of the basics of getting and setting figure options. But the figure you created contains much more information. The highest level handle is *gcf* which is equal to 1.

```
gcf
get(gcf)
```

is a command that will show all of the main figure options. But in these figure options are *Children*, other handles to various parts of the figure. For example, if we want the handle to the axis

```
AxisHandle = get(gcf,'Children')
```

The axis is given a special handle in Matlab called *gca*. If you type

```
gca
```

on the command line if should give you the same answer. The axis handle has its own Children that contain the plot you created.

```
PlotHandle = get(AxisHandle,'Children')
h
get(gca,'Children')
```

and you can verify that all three commands should give the same figure handle.

This may all be a bit confusing, but it allows the user to keep track of the various parts of complex plots, *e.g.,* subplots. For example, we can now set the line width of out plot with

```
set(h,'LineWidth',3);
```

Alternatively, we could have issued

```
set(PlotHandel,'LineWidth',3);
```

Now we can add a second plot

```
hold on
g = plot(x,sin(1.5*2*pi*x),'r');
```

We have explicitly created a figure handle (g) to make it easier to change this plot. But try the following command

```
l = get(gca,'Children');
```

You should notice that now there are two file handles, and we could treat each differently. In fact, $l(1)$ is our most recently created figure handle and $l(2)$ is our previous handle. So, if we wanted to view the options for our recently created figure, we could type

```
get(g)
```

or

```
get(l(1))
```

Now let's suppose we wish to change the line thickness and the line type from solid to dash-dot.

```
set(l(1),'LineWidth',3)
set(l(1),'LineStyle','-.');
```

Figure handles can become very complex, but you should remember that they are nested in hierarchies with *gcf* at the top and *gca* one step down.

10.3.2 GENERATING PUBLICATION QUALITY FIGURES

Given the flexibility in how aspects of figures can be changed, it should not come as a surprise that many engineers and scientists create images for their journals and other technical writings in Matlab. Below is a template that demonstrates how to generate a publication quality figure.

Brain waves are often thought of as being composed of multiple frequency bands. For example, Delta waves range from 0-4Hz, Theta waves range from 4-7Hz, Alpha waves range from 8-12Hz, Beta waves range from 12-30Hz and Gamma waves are about 30Hz. The Electroencephalogram (EEG) can be scored by a clinical neurologist based upon the strength, *i.e.,* amplitude, of the frequencies present. In fact, they can get an accurate picture of the state of the patient (awake, sleeping, dreaming, thinking), just by looking at the EEG. Below is a section of code that will create a theoretical EEG. Following the creation of the code are a series of commands that change shapes, sizes, colors and even the aspect ratio of the figure. The last line creates an encapsulated postscript file with high resolution (600 dots per square inch).

```
x = 0:0.001:2;
%Create EEG signal from various sinusoids
%Amplitudes reflect signal contribution
EEG = 1*sin(2*2*pi*x);          %2Hz Delta
EEG = EEG + 1*sin(6*2*pi*x);    %6Hz Theta
EEG = EEG + 1*sin(10*2*pi*x);   %10Hz Alpha
EEG = EEG + 4*sin(20*2*pi*x);   %20Hz Beta
EEG = EEG + 2*sin(50*2*pi*x);   %50Hz Gamma

h = plot(x,EEG,'k');

xlabel('Time (s)','FontSize',20)
ylabel('EEG (microV)','FontSize',20,'Rotation',90)
axis([0 2 -8 8])
set(gca,'Box','off')
```

```
set(gca,'TickDir','Out')
set(gca,'XTick',[0:0.25:2])
set(gca,'YTick',[-8:4:8])
set(gca,'FontSize',20)
set(gca,'LineWidth',2)

set(gcf,'Color','white')
set(gcf,'Position',[400 400 800 400])
set(gcf,'PaperPosition',[4 4 8 4])
print('-depsc','FrequencyAnalysis.eps','-r600');
```

You should also note that Matlab allows for inline text, arrows and other simple figures to be added to the plot. These functions, however, are often better to include in power point or another graphics package.

10.4 DISPLAYING 3D DATA

There are occasions where it is helpful to visualize data in three dimensions. Matlab has a number of commands designed for 3D plots. Below are some lines which demonstrate some of Matlab's capabilities

```
figure(1)
t = 0:pi/50:10*pi;
plot3(sin(t),cos(t),t)
xlabel('sin(t)')
ylabel('cos(t)')
zlabel('t')
```

You can actively move around these data by clicking the rotation tool on the toolbar (next to the hand tool). Try holding down the left mouse button and then dragging. This should rotate the figure.

```
figure(2)
[X,Y,Z] = peaks(30);
surfc(X,Y,Z)

figure(3)
contour(Z)

figure(4)
mesh(X,Y,Z)
```

You should view the help for these functions to learn about additional 3D plotting functions. You may also wish to explore the help for *view*, a command that allows the user to set the three-dimensional view point.

10.5 EXERCISES

1. Turn in "HMRModel.m", "HMR.fig", "FrequencyAnalysis.eps".

2. A common device used to grow cells and bacteria is a chemostat. The idea is that as cells metabolize they require a constant supply of nutrients but also need some way to eliminate waste products. A Chemostat is a chamber that has a constant influx of nutrients, while at the same time having a constant efflux of solution in the chamber. In such a situation, we can imagine keeping track of both the cell population (N) and the concentration of the nutrient (P). Below are two differential equations that describe this situation.

$$\frac{dN}{dt} = N\left(\frac{K_{max}C}{K_n + C}\right) - \frac{FN}{V} \tag{10.19}$$

$$\frac{dC}{dt} = \frac{F}{V}(C_o - C) - \alpha N\left(\frac{K_{max}C}{K_n + C}\right) \tag{10.20}$$

where C_o is the concentration of the supply, V is the volume of the growth chamber, F is the input and output flow rate, α is a yield constant, K_{max} is the maximum growth rate of cells and K_n is a half maximum growth rate relative to concentration.

The two equations above can be rearranged using some algebra to find equations for the N and C nullclines.

$$C = \frac{\frac{F}{V}K_n}{K_{max} - \frac{F}{V}} \tag{10.21}$$

$$N = \frac{F(C_o - C)(K_n + C)}{V\alpha K_{max}} \tag{10.22}$$

Therefore, the N nullcline is a quadratic and the C nullcline is a simple line. Create a script that will plot both nullclines in a N-C phase space. The following code will create the nullclines for specific values of the constants

```
alpha = 3.0;          %Yield Constant
F = 1.0;              %in and out flow of Chemostat
V = 1.0;              %Chemostat Volume
Kmax = 6;             %Maximum Growth Rate
Kn = 50.0;            %Half Max of Growth Rate
Co = 200.0;           %Concentration of Supply


N = 0:0.1:1000;
C = 0:0.1:200;
```

```
Cvec = (F/V).*Kn./(Kmax-(F/V));            %For N Nullcline
Nvec = F.*(Co-C).*(Kn+C)./(V*alpha*Kmax);  %For C Nullcline

figure(1)
plot(N,Cvec)
hold on
plot(Nvec,C)
```

Create an image called "ChemostatNullclines.jpeg" that is of publication quality. You should use your best judgment and what you have learned in this chapter as a guide.

CHAPTER 11

Toolboxes

11.1 INTRODUCTION

As mentioned in chapter 1, Matlab is a number of environments all rolled into one. As originally envisioned it is a programming environment, scripting language and a graphics package. Since the early days of Matlab, however, many outside of Mathworks have contributed. Often these new contributors are a group of researchers or industrial scientists writing a series of Matlab scripts (".m" files) that perform related functions. These suites of functions, after an evaluation process, sometimes become part of Matlab. Matlab calls these packages *toolboxes* and allows users to purchase them as products separate from the standard Matlab distribution. To check the packages in your version of Matlab

```
>> ver
```

will display the current version of Matlab and any installed toolboxes. There are hundreds of Matlab toolboxes that can be purchased. Some researchers also offer their own Matlab toolboxes free of charge, although with no guarantees of proper functionality or optimization from Mathworks.

There are three ways to get help for toolboxes. First, Matlab has a website at www.mathworks.com for each toolbox under the Support tab. The website gives a brief overview of the functionality of the toolbox along with any new versions. Second, Matlab maintains a user's guide that can be either downloaded as a pdf or viewed online. It is here that all of the functions (".m" files) will be listed along with how to call them. This feature is nice because it allows a user to read a manual before purchasing the toolbox. You can also find examples of how to use the toolbox and demonstrations. Once a toolbox is installed, the third option is to view the built-in help.

```
>> help
```

will display the high-level help that is available in Matlab. Of these functions, some are the help for the toolboxes listed in *ver*. For example,

```
>> help symbolic
```

will display the help for the Symbolic Math Toolbox.

In this chapter, we only cover a few of the toolboxes which are of most interest to Biomedical Engineers and are readily available at most institutions.

11.2 STATISTICAL ANALYSIS AND CURVE FITTING

Much of this text has focused on the simulation of mathematical models. But much biological and biomedical research exists in the experimental realm. Here Matlab can be useful in the analysis of data, specifically to perform statistical analysis and best fits. The Matlab statistics toolbox contains functions for descriptive statistics for vectors and matrices of data, *e.g., skewness* for Skew, *cov* for covariance, as well as sophisticated random number and probability distribution generators. It also contains linear, *e.g., anova1* and *anova2* for one and two way analysis of variance, *lscov* for least-squares, and non-linear, *e.g., nlinfit*) data fitting. There are even commands to help design experiments and specialized graphic utilities. See

```
>> help stats
```

for more information.

11.2.1 DATA FITS TO NONLINEAR FUNCTION

It is often the case that an engineer or scientist will collect a series of discrete data points and then need to move from the data to an analytical model, *e.g.,* mathematical function. Typically, an experimentalist will collect data as a series of points as a function of some independent variable that can either be controlled or observed, *e.g.,* time, space, concentration, current. For example, the impedance of a biological material is the resistance (R) to current (I) as a function of the frequency (ω) of a sinusoidal forcing function. The experiment would send in an alternating current with a particular frequency ω and then record the amplitude of the resulting voltage (also a sinewave). The experimenter would then record the peak-to-peak amplitude of the voltage sinewave as a function of the frequency.

```
>> omega = [0:10:100];      %frequency in Hz
>> SineWaveAmplitude = [2 5 10 37 59 41 12 4 3 1 0.1];
>> plot(omega,SineWaveAmplitude,'*');
>> hold on                  %You will add to this plot later
```

Rearranging Ohm's Law ($V = IR$) to $R = \frac{V}{I}$ and assuming I is held constant, the measurement of V is proportional to the impedance, R. From the data, it is then clear that this material will pass current best at low and high frequencies, *i.e.,* the impedance is highest around 40Hz.

In many biological applications, to begin creating a model, we need to fit a function to results of an experiment. In the experiment above, we may guess that the data is best fit to a bell-shaped curve, known as a Gaussian Function.

$$G = \frac{A}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{11.1}$$

What we need to estimate are the parameters μ (mean), σ (standard deviation) and A (area under the curve). To perform this fit we will use the nonlinear least-squares fit, *nlinfit*, function in Matlab. The function has the following command line call

```
Beta = nlinfit(x,y,Model,beta0);
```

where x and y are the independent and dependent variables of the real data. *beta*0 is the initial guess for the parameters and *Model* is the name of a function that contains the guessed function. You should note that the *Model* function and parameters, *beta*0, must match. The function returns the best fit, *Beta*, for the parameters.

For initial guesses, we can assume the mean is at 40. We can also guess that our standard deviation is approximately 15. Lastly, area we can estimate as 1000.

```
>> beta0 = [40 15 1000];
```

The last step is creating the function to pass to *nlinfit*. Open up a script called "Gauss.m" and enter the following

```
function [G] = Gauss(beta,x);

mu = beta(1);    %define mean
variance = beta(2)^2;   %define standard deviation
area = beta(3);  %define area

G = (area/sqrt(2*pi*variance)).*exp(-((x-mu).^2)./(2*variance));
```

To test this function

```
>> x = 0:1:100;
>> G = Gauss(beta0,x);
>> plot(x,G,'r')
```

We are now ready to tune the parameters

```
>> Beta = nlinfit(SineWaveAmplitude,omega,@Gauss,beta0);
>> Beta
```

You should note that the `@` symbol is used to reference a function. After many iterations, you will have a best fit to the Gaussian parameters contained within *Beta*. You can then check the fit

```
>> NewG = Gauss(Beta,x);
>> plot(x,NewG,'g');
```

You should view the help for *nlinfit* to see how to measure the quality of the fit, as well as how to bound certain parameters such as the maximum number of iterations to take in searching for a best fit.

Some other common functions to fit are linear and exponentially increasing or decreasing. There are two other functions that deserve mention in the context of biomedical engineering because they appear often. The first is the monotonically increasing or decreasing sigmoid.

$$S = \frac{1}{1 + e^{ax}} \tag{11.2}$$

where x is the independent variable. The sign of a (+ or -) will determine whether the function increases or decreases and the magnitude of a will determine the rate of increase or decrease. You may wish to generate a few lines of Matlab code to plot the sigmoid function - it should be "S" shaped.

A more general form of the Sigmoid function is the Boltzmann distribution

$$B = \frac{1}{1 + e^{(x-x_o)/k}} \tag{11.3}$$

where k is the inverse of a, and therefore controls the slope and increasing or decreasing trend. You may have noticed that the sigmoid was centered around zero. The term x_o is an offset and controls what is known as the half-max crossing point and will therefore translate the sigmoid. Note that both functions range from 0 to 1, making scaling to fit experimental data a simple task.

Note that Matlab also has a series of commands for fitting data to surfaces. The idea is the same, but now there are two independent variables and one dependent variable (often thought of as the height of the surface).

11.2.2 INTERPOLATION AND SPLINES

Two very useful operations that follow directly from curve fitting are interpolation and extrapolation. In the impedance example above, we may wish to estimate the voltage output every 1Hz even though you only measured it every 10Hz. The function *interp1* will allow you to interpolate values.

```
>> figure(2)
>> omega = [0:10:100];      %frequency in Hz
>> SineWaveAmplitude = [2 5 10 37 59 41 12 4 3 1 0.1];
>> plot(omega,SineWaveAmplitude,'*');
>> DesiredOmega = [0:1:100];
>> NewV = interp1(omega,SineWaveAmplitude,DesiredOmega);
>> hold on
>> plot(DesiredOmega,NewV,'r');
```

The inputs to *interp1* are the original independent and dependent variables, along with the desired independent variable. The output, $NewV$, is the new vector of voltages that correspond to $DesiredOmega$. Finally, we plot the interpolated data over the original data.

With no options *interp1* will default to a "linear" interpolation, meaning that all data points in $SineWaveAmplitude$ will be connected by straight lines. Interpolation can become much more sophisticated by using higher order fits between data points. The most important options for *interp1* are "spline" and "cubic". For example,

```
>> NewVSpline=interp1(omega,SineWaveAmplitude,DesiredOmega,'spline');
>> plot(DesiredOmega,NewVSpline,'g');
```

You should note that a spline is simply a local polynomial fit to data. In other words, a low order polynomial is fit to only a few local points. A different fit is then created for other local points.

You should be very careful not to confuse $NewV$ with real measured experimental data or the analytical function, created above. To get an analytical function we must assume a form for the equation and then fit the parameters using a best fit. This fit was performed on the entire data set. With $NewV$ the interpolation was created without assuming any function, only very simple functions, *e.g.*, polynomials, that span a few points between neighboring points.

In our example, we used all evenly spaced data points, but this does not need to be the case. In other words, the independent variables *omega* and $DesiredOmega$ could have been an irregularly spaced vector. You should view the help for *interp1* for more details. You can also view the functions *interp2*, *interp3* and *interpn* which perform 2, 3 and N dimensional interpolation.

The spline toolbox also has tools for extrapolation, confidence intervals, tools to remove outliers and fitting to surfaces. You should type

```
>> help splines
```

for more information.

11.3 DIFFERENTIAL AND INTEGRAL EQUATIONS

In Section 7.3, we introduced Euler's Method for the numerical integration of a differential equation. And an example in chapter 7 demonstrated how to use Euler's Method for multiple coupled differential equations. Although the Euler Method is easy to understand and program, it is limited in that it is only first order accurate. The idea of the *order* of the method is one that comes up in many numerical methods, and it signifies how well of an approximation the method will yield. In the case of differential equations, we can write the solution to a differential equation as a Taylor series. Then the order of the numerical integration technique is given as the number of terms in the Taylor series that are included in the approximation. In the case of Euler, only the first term is included. There are other numerical integration methods that use many more orders to improve accuracy. The most popular are a series of methods with any desired order known as the Runga-Kutta method.

Although, in principle, the order can be increased to obtain more and more accurate solutions, the computational cost (memory and time) increases as order increases. For most applications, the 4-5 order is where the gain in numerical accuracy balances out the computational cost.

To show how the built-in solvers can be used, we will solve the same FitzHugh-Nagumo

model of a neuron as in chapter 7

$$\frac{dV}{dt} = V - \frac{V^3}{3} - W + I \tag{11.4}$$

$$\frac{dW}{dt} = a * (V + b - cW) \tag{11.5}$$

where V is the cell membrane potential, W is a recovery variable and I is a stimulus current. We will assume the constants are $a = 0.08$, $b = 0.7$ and $c = 0.8$. We will use the *ode45* solver that has the following command line call

```
[t,y]=ode45(odefun,tspan,y0);
```

where $odefun$ is a function that will evaluate the derivatives given the variables in the vector y. $y0$ are the initial conditions and $tspan$ is a vector with two elements, $[T0, Tfinal]$. First, open a Matlab function "FHNFunction.m" and enter the following text

```
function dy = FHNFunction(t,y)

a = 0.08;
b = 0.7;
c = 0.8;
I = 0.556;   %Simulus Current

V = y(1);
W = y(2);

dy = zeros(2,1);
dy(1) = V-(V^3)/3-W+I;
dy(2) = a*(V+b-c*W);
```

In this function, all that is reported is how to compute the right-hand term in the differential equations. This is done because there is no assumption made about how the time step (Δt) will be picked or how the solution will be advanced forward. Next, enter the following on the command line.

```
>> tspan = [0 100];
>> y0 = [0 0];
>> [t,y]=ode45(@FHNFunction,tspan,y0);
>> plot(t,y);
```

You should note that the output t contains a vector of the times and y contains both the V and W vectors.

One reason to use one of Matlab's built-in solvers is that they very often have what are known as adaptive time steppers. To view how Matlab has changed Δt during the simulation

```
>> plot(diff(t));
```

You should note that unlike the Euler method used in chapter 7, the time step changes. In general, Δt becomes large when the solution is not changing much and becomes small when the solution is changing rapidly.

The help for *ode45* contains a list of the other solvers that can be used, functions for evaluating the accuracy of the solution (*e.g.*, *deval*) as well as some demonstrations. You should also note that there is a partial differential equation toolbox for handling cases where dynamics are occurring in both time and space.

11.3.1 INTEGRALS AND QUADRATURE

Finding the area under a curve can often yield valuable insight into a biological problem. For example, we may need to find the area under a curve to compute total charge as a current that flows over time

$$Q = \int_{t_1}^{t_2} I(t)dt \tag{11.6}$$

or mechanical work as the integral of force applied over some distance.

$$W = \int_{x_1}^{x_2} F(x)dx \tag{11.7}$$

The problem is that rather than have the analytic functions $I(t)$ or $F(x)$, we have discrete points in the vectors I or F. There are various methods for numerically computing areas under curves formed by discrete points, and they all fall under the general category of *Quadrature*. For example,

```
>> x = 0:0.01:1;
>> y1 = 20*exp(x);
>> y2 = 10*rand(length(x),1);
>> Int1 = trapz(x,y1);
>> Int2 = trapz(x,y2);
```

And it is a simple matter to then get the area between the two curves as

```
>> Int1-Int2
```

The above example uses the trapezoidal approximation. Matlab has many other quadrature methods, each with their advantages and disadvantages. As with most numerical methods, there is a trade off between accuracy on the one hand and computing cost on the other. Some quadrature methods can be found in the command *quad*. You should note that you can also evaluate double (*quad2d*) and triple (*triplequad*) integrals.

11.4 SIGNAL PROCESSING TOOLBOX

Biological signals are often noise, low in amplitude and composed of many superimposed streams of information. Signal processing is what is necessary to isolate features of interest. For example, when analyzing an EEG signal from the scalp, it may be important to determine the relative contribution of a particular frequency band to the signal. Alternatively, it may be important to look for what is known as a complex - a series of spikes and waves that are signatures of specific events in the brain. Using these two types of information, a clinical neurologist can gain a great deal of information about the healthy or abnormal function of a patient's brain.

The type of operation to perform is nearly always some sort of filter, and these are typically either in the time or frequency domain. In the time domain, the two most useful are moving averages and correlations. Moving averages are important because a serious problem with most experimental data is that it is noisy. The result is that an upward or downward trend can often be lost when curve fitting or looking for trends. One often used solution is to smooth the data before fitting to a function. The key is to average nearby points, but the weighting of those points may vary.

```
>> w1 = hamming(64);
>> wvtool(w1)
>> Sig = rand(1000,1);
>> SigWin = conv(Sig,w1,'same');
```

will create and then display the Hamming Window with 64 samples. The idea is that this window will be moved over the entire dataset, and each new point will then be a weighted average of the 64 points around it. Then the window is moved one step forward the averaging occurs again. For example,

```
>> t = 0:0.01:1;
>> Sig = sin(2*pi*t);
>> Sig = Sig + rand(1,length(t));              %noisy sinusoid
>> plot(t,Sig);
>> hold on
>> SigWin = conv(Sig,w1,'same');   %smoothed sinusoid
>> plot(t,SigWin,'r');
```

Note that the general shape of the sine wave has been recovered, but the amplitude is off. More processing would be needed to correct this. Also note that the convolving function (*conv*) was used to apply the filter to all points. Other useful windows can be found in the help for *window*.

Another very common operation is to gain some quantitative measure of how similar two signals are to one another. Here we are looking for correlations. Matlab has a number of functions, *e.g.*, *conv*, *cov*, *corrcoef*, for performing these types of operations. The theory behind these operations

will not be covered here, but you can view the help within Matlab and online for more information.

In the frequency domain, we typically think of designing a filter that will eliminate some frequencies but keep others. For example, a bandpass frequency filter may be needed to analyze the results of fatigue testing on a muscle, but a low pass filter may be desirable for isolating low frequencies in an EEG. To gain more appreciation for the options in the signal processing tool box, you should open up an internet browser and navigate to www.mathworks.com. Click on Products and Services and then on Product List. You should see a full list of the Matlab toolboxes. Scroll down to the Signal Processing Toolbox and click on the link. On the left-hand menu, you will find a link to Demos and Webinars. You should watch the short Introduction demo. On the left-hand side, you will also find a description of the toolbox along with a complete list of all of the functions. You should note that nearly all toolboxes in Matlab have good tutorials to help you get started.

11.5 IMAGING PROCESSING TOOLBOX

Biomedical engineers often generate images or display their data as images. In both cases, the raw images, like biological signals, are typically noisy and low contrast. The image processing toolbox contains algorithms for bringing features of interest to the forefront. There are also times when a user may be looking for correlations between different parts of an image, for example to track a cell as it crawls across a series of images that comprise a movie (known as image registration). Again, the image processing toolbox contains routines for just such a task. Another important feature of the image processing toolbox is edge detection. Below is a very brief demonstration.

```
>> IMAGE = imread('circuit.tif');
>> figure(1); imshow(IMAGE);
>> IMAGE2 = edge(IMAGE,'prewitt');
>> figure(2); imshow(IMAGE2);
>> IMAGE3 = edge(IMAGE,'canny');
>> figure(3); imshow(IMAGE3);
```

The above lines highlight three commands. The first is *imread* which is a general purpose reader for many different image formats. Second is *imshow* which is a general function for displaying grayscale images. Last is the *edge* command which finds edges of an image in the regions where there is a large contrast in grayscale. *edge* contains many different methods and options, and you may want to view the help file.

There are a number of other helpful commands, such as *imresize*, *imrotate*, *imcrop* and *imtransform* as well as a number of sample images to test image processing algorithms, *e.g.*, "board.tif", "trees.tif","cameraman.tif".

You may wish to view some of the built-in demonstrations that will give you some sense of the power of the image processing toolbox.

```
>> iptdemos
```

11.6 SYMBOLIC SOLVER

Thus far, we have focused on programming techniques and toolboxes which perform numerical approximations. Matlab also has a toolbox for performing symbolic math, allowing for direct analytical solutions. The functions in the Symbolic Toolbox are very similar to two other well-known symbolic math processors, Mathematica (Wolfram Research) and Maple (MapleSoft). It should be noted that Mathematica and Maple both allow algorithmic computing and graphics, similar to the commands in the first 10 chapters, but their focus is on symbolic math. Matlab, on the other hand, is designed for numerical approximations and has the symbolic toolbox as an addition.

In keeping with the theme of only showing the surface level of each toolbox, we will show a few examples where the Symbolic Toolbox can be useful. An important distinction between Symbolic and Algorithm solutions must be made first. Consider the following equations that describe a compartment model of a drug in the body.

$$\frac{dA}{dt} = -k_o A \tag{11.8}$$

$$\frac{dB}{dt} = k_o A - k_1 B \tag{11.9}$$

$$\frac{dE}{dt} = k_1 B \tag{11.10}$$

where A is concentration at the absorption site, B is the concentration in the body and E is the concentration being eliminated. We have already learned a variety of ways to solve this problem given the initial conditions $A(0) = A_o$, $B(0) = 0$ and $E(0) = 0$. These range from creating a matrix, to finding eigenvalues to solving the equations numerically using the toolboxes described above. In all cases, Matlab is using some form of a numerical method, *e.g.*, a matrix solve, numerical integration. Below are the commands for solving the equations above

```
>> % To be entered on the same line
>> [A,B,E]=dsolve('DA=-k0*A','DB=k0*A-k1*B','DE =k1*B',
>> 'A(0)=A0','B(0)=0','E(0)=0');
>> A = simplify(A);
>> B = simplify(B);
>> E = simplify(E);
```

Note that the solution is not a vector of values, but rather an equation. This is what makes symbolic math different from numerical methods.

The symbolic math toolbox can also perform differentiation (*diff*) and integration (*int*). It may seem strange that the *diff* command could either be used to find the difference between elements in a vector or perform symbolic differentiation. This is an example where Matlab has what is called overload methods. If you type

```
>> help diff
```

you will notice that at the bottom of the help is a section on Overload Methods, and one of the listed methods is *sym/diff*. If you click on this link, you will be taken to the help for the symbolic differentiation.

This still does not explain how Matlab seems to know which version of *diff* to use. The answer is in the values that are passed in. You should note that both commands take one variable, but the numerical *diff* takes a vector whereas the symbolic *diff* takes a function.

```
>> x = sym('x'); % create a variable called x
>> t = sym('t'); % create a variable called t
>> diff(sin(x^2))            %analytic result
>> diff(t^6,6)               %evaluate analytic result
```

The same logic applies to *int*, which can be used either in a numerical or analytic way in Matlab. You should note that the *sym* command was used to create a symbolic variable. You should view the Matlab memory to verify that x and t are in fact of the variable class "symbolic". Likewise, you may want to force one function to be substituted into another using the *sub* command.

Above a series of differential equations were solved, but a more simple solve is for simultaneous algebraic equations

```
>> [x,y] = solve('x^2 + x*y + y = 3','x^2 - 4*x + 3 = 0')
```

You can even perform more advanced analytical processes such as a Taylor series expansion.

11.7 ADDITIONAL TOOLBOXES AND RESOURCES

There are a wide range of additional toolboxes available through Matlab. Below are some that have direct ties to biology and biomedical engineering.

1. **Simulink** is a graphical programming interface for simulating systems. It consists of a series of graphical blocks that can be connected by "wires". In this way, the flow of the program is visibly apparent and does not require the writing of a script.

2. **Graphical User Interfaces** (GUIs) allow a user to interact with programs (similar to those described in Section 6.7), but in a custom designed window. Many of the same web-like inputs are supported, including radio buttons and text boxes, but Matlab also supports sliders, dials and the ability to embed custom graphics into the window, *e.g.*, two and three-dimensional plots, subplots. GUIs are a very powerful way to make a complex program more user friendly. For more information, there are a number of good web resources as well as a built-in GUI creator called *guide*.

3. **Neural Networks** are an abstraction of how real neurons connect to one another and perform pattern recognition functions on data. They typically need to be trained on some data set,

where the answer is known. Parameters of the network are adjusted to give the minimum error between the actual output of the network and the desired output of the network. In this way, neural networks are very similar to filters, but they are adaptable and tunable.

4. **Genetic Algorithms** are an abstraction of how evolution is thought to use random variation, mutation, selection and mating to produce good (fit in the language of genetic algorithms) solutions to a problem. For some problems, the possible solution is not obvious. Even worse, because there may be many parameters involved, it would take an enormous amount of time to perform a parametric study. In these situations, we can generate a random sampling of possible solutions that must compete with one another for some resource. The most fit solutions will outcompete the less fit. These most fit solutions will have the opportunity to "breed" with other fit solutions in the hope of creating an even more fit solution. Superimposed on these dynamics may also be mutation, which will widen the exploration of the solution space.

5. **Control Systems** is a diverse field that spans many engineering disciplines and is largely about the idea of self-regulation and system characterization. In biological systems, this is similar to the idea of homeostatis and there is an entire field called systems physiology that studies biological function from a quantitative, systems point of view. Likewise, a biomedical engineer often must create devices that compensate for some poorly functioning part of the body. The control systems toolbox contains basic routines for simulating and characterizing systems as well as special graphics routines.

6. **SimBiology** is a graphical interface for modeling systems in biology and pharmacokinetics. It is similar in many ways to Simulink and also provides the user with some unique solvers and analysis tools.

7. **MEX**, short for Matlab executable, allows users to write a function in a compiled language, *e.g.,* C, C++, FORTRAN, and then use that function in Matlab. MEX code is not written in Matlab and therefore requires a user to have knowledge of another computing language. MEX functions are very useful when an algorithm cannot make good use of matrix-vector operations, *i.e.,* it contains many loops. These functions will appear as built-in functions in Matlab - recall an attempt in an earlier chapter to viewing the ".m" file for *sum*.

8. The **Matlab Compiler** allows a user to compile their Matlab code. There are at least two reasons to compile code. First is to speed up simulation time. Remember that Matlab is an interpreted scripting language, meaning that it is flexible but slow. Compiling could greatly increase the speed of the code. Second is if you wish to share the function of your code, without sharing the code itself. This can be useful if you work for a company and do not wish to share the algorithms that are used.

9. Mathworks recently added a **Parallel Computing** toolbox to allow users with networked computers to break up a large computational task into smaller tasks that can then be sent

to individual computers. It should be noted that some algorithms lend themselves to easy adaptation to parallel computing, whereas others do not.

11.7.1 MATLAB CENTRAL AND OTHER ONLINE HELP

There are a number of other very helpful online resources. The most important is MatlabCentral, a place for Matlab users to ask for help, post help and post new code. It is located at http://www.mathworks.com/matlabcentral/ and should be the first place you look if you have an algorithm to write. It is accepted practice in the coding world to use the code of others and to cite them appropriately. You may also find code by using a search engine to find the websites of others. Both can be excellent sources of code, but you should remember that Mathworks does not verify the code from outside parties.

Author's Biography

JOSEPH V. TRANQUILLO

Joseph Tranquillo is an associate professor of biomedical engineering at Bucknell University where he has been a faculty member since 2005. He received his Doctor of Philosophy degree in biomedical engineering from Duke University (Durham, NC) and Bachelor of Science degree in engineering from Trinity College (Hartford, CT).

His teaching interests are in biomedical signals and systems, neural and cardiac electrophysiology, and medical device design. Nationally Joe has published or presented over 40 peer reviewed or invited works in the field of engineering education. He was the founder and inaugural chair of the Undergraduate Research Track at the Biomedical Engineering Society (BMES) conference, co-organized the Biomedical Engineering Body-Of-Knowledge Summit and currently serves on the board of the Biomedical Engineering Division of the American Society of Engineering Education (ASEE). He is the winner of the 2010 National ASEE Biomedical Engineering Teaching Award.

His technical research interests are in non-linear dynamics in the heart and brain. He has over 50 publications and presentations, and he has authored a textbook, *Quantitative Neurophysiology*. He is a member of the Biomedical Engineering Society, IEEE Engineering in Medicine and Biology Society, American Physical Society and is an elected member of Sigma Xi and Heart Rhythm.

When not teaching or doing research, he enjoys improvisational dance and music, running trail marathons, backpacking, brewing Belgian beers, and raising his two children Laura and Paul.