Introduction to Programming
With Matlab

J. Michael Fitzpatrick and John D. Crocetti
Introduction to Programming with Matlab

J. Michael Fitzpatrick and John D. Crocetti

Department of Electrical Engineering and Computer Science
School of Engineering
Vanderbilt University
Nashville, TN

© 2000-2011 J. Michael Fitzpatrick and John D. Crocetti
Introduction to Programming with Matlab
which is a book comprising a Preface, a Table of Contents, Chapters 1 through 12, and
Homework Assignments 1 through 6, each of the latter in turn comprising a set of
problems.

Copyright 2000-2010 by J M Fitzpatrick and J Crocetti

All rights reserved. No part of the material protected by this copyright notice may be
reproduced in any form or by any means without the written permission of one or both of
the copyright owners except that permission is granted to any student who is, or who has
been, enrolled at Vanderbilt University in CS 103, Introductory Programming for
Engineers and Scientists.

The authors make no warranty regarding the programs or documentation within this book
and are not liable for any damages resulting from their use.
Preface

This book is meant for students of engineering or science who need both a useful programming language for solving problems in their disciplines and an introduction to ideas from the discipline of computer science. It is provided free of charge to students at Vanderbilt who enroll in CS 103, “Introductory Programming for Engineers and Scientists”.¹ It is being provided to you as a “digital” book, meaning that you are provided the digital computer files that contain it. (Because it is in this format, it is very easy to copy portions of it. You are free to do that, but if you incorporate any part of the book in other work, then you must give a proper citation, just as you would any book printed on paper.) If you are reading the Microsoft Word version of this textbook, the format of the file is compatible with Microsoft Word 1997-2010. If you are reading the PDF version, it is compatible with all recent versions of Acrobat Reader.

CS 103 provides an introduction to programming. If you study computer science or computer engineering further, you should, upon completion of this course, have a strong foundation for advancing in either area as you enroll in more advanced courses that involve programming. If this is your only course in programming, you should feel confident that you can write the programs that you need to solve the engineering or scientific problems that you encounter in your chosen discipline.

Style

Each of the chapters of this book corresponds to approximately one or two weeks of the semester. This textbook emphasizes more of the general concepts from the discipline of computer science than does the typical introduction to Matlab programming. In addition to helping you to learn how to solve problems by writing programs in Matlab, it should also help you to see how the study of programming in Matlab relates to the study of computer science.

Much of the material is presented in a tutorial style (i.e., concepts are illustrated by means of practical examples). The early chapters include lots of introductory tutorial material. This tutorial material has two purposes: (1) To help you get started, and (2) to reinforce your understanding of concepts that are presented in lecture. As you gain experience with Matlab, with concepts from computer science, and with the lecture style of your instructor(s), you will be better able to absorb material more easily. As that happens, the style will become less tutorial, and the rate at which new material is introduced will increase. Unlike most books, this one does not contain sets of worked examples within each chapter. Instead examples are provided in a separate document, called

¹ This book is copyrighted material. It may not be redistributed without permission from the authors, but students in CS 103 may use the book during the course and for as long as they wish after the course is over. They make copies for their own use including paper print-outs.
Highlights.doc, which provides both an outline of the ideas in this book and examples, with commentaries and complete solutions. These problems are presented, discussed, and worked in class while you, as a student, watch and, in most cases, participate. Furthermore, this book does not have sets of homework problems at the end of the chapters. Instead homework problems are provided either in a separate document or at the end of the book, depending on which format you received. Finally, an extended homework problem, called a “Project”, is provided, which require a broader knowledge of the contents of the book and which will require considerably more time to solve. All these documents are made available on line to students in the class.

**Homework**

The homework problems, including the Project, are provided to help you with the understanding of the material presented in this book. Some of these problems and projects may be assigned to you. To complete the assignments you will need to understand the concepts covered both in the book and in the lectures. Furthermore, as in virtually all engineering and science courses, because later concepts build on earlier ones, you will find that each later homework assignment refers to concepts, not only from the most recent lecture and text coverage, but from previous coverage as well. Thus, as you work your way through the homework assignments in this class, you will enforce and re-enforce your understanding of concepts that you learned earlier in the semester. You will also be tested on the material covered in the homework assignments by means of in-class, graded tests. If you learn the material for the homework, you should find that studying for the tests is easy, and you should do well on the tests and in the course.

**Suggestions for using a digital textbook**

*Turning off spelling and grammar checking*

Many of the words in this document are in a computer programming language and are not standard English. The spelling and grammar checking tools of Microsoft Word\(^2\) will highlight these words unless the highlighting feature is turned off. This highlighting can be quite distracting. It is recommended that you turn these features off when reading this text. To do that, click the Microsoft Office Button \(\text{at the upper left. A subwindow will open. Click Word Options at the bottom of that subwindow near the right. Click Proofing in the list at the left. A new subwindow will appear. At the bottom you will see Exceptions for: followed by the name of the document that you have open. Click the checkboxes next to Hide spelling errors in this document only and Hide grammar errors in this document only. When they are both checked, click OK at the bottom of that subwindow.}*

*Indexing with Find*

There is no index for this book! Instead, because the book is in “soft” form, you can use the versatile “Find” tool of Word to do dynamic indexing. That tool can be accessed most

---

\(^2\) The specific version of Word used for these suggestions is Word 2007.
easily by clicking on the tiny circle at the right, bottom of the Word window, as pointed out by the red arrow below:

Clicking there will cause this smaller window to appear: Clicking on the binoculars will bring up Find.

Alternatively, on the taskbar at the top of the Word window you can click Edit and then Find…. In either case you will see the following window appear:
To get the full functionality of this feature, you should click on More at the bottom. You will then see a larger window:

![Find and Replace dialog box](image)

You can choose to use the special features by clicking the checkboxes at the lower left. The “Use wildcards” box is the most versatile. If you choose that option, then you can click the “Special” box below to get a list of wildcards to choose from. You can probably learn what you need to learn about wildcards by experimentation, but you can also learn everything by using the Help facility.

**Getting help with Word**

Word provides an excellent help facility, which can be accessed simply by double clicking the Help icon at the upper right corner of the Word window or hitting the F1 key, and then typing a phrase, such as “find” into the search box and clicking Search.

**Annotating your personal copy of this book**

You can easily write in your personal copy of this book and you can highlight words and sentences, just as would in a paper book. Do not write directly into the text. Instead add comments, using Word’s comment facility. Let’s consider an example. Suppose you want to annotate the first page of Chapter 1 by highlighting the definition of computer science in the second paragraph and inserting a comment about the definition of algorithm. The result might look like this:
To highlight, select a portion of text, then click the icon of a highlighter pen under the Home tab. To remove the highlighting, repeat the process. To insert a comment, select a portion of text, click New Comment under the Review tab. An empty comment “balloon” will appear in the margin. Type your comment into that balloon. (If your comments do not appear in the margin, then click Tracking under the Review tab, click Balloons, and click to the left of Show Only Comments And Formatting In Balloons to put a checkmark there. You will find it easier to work with annotations if you turn these “balloons” on.) To delete a comment, click right on it and choose Delete Comment.

Updating your personal copy
Periodically, you will be find that a chapter has been updated on line. When that happens, you will want to update your personal version. The simplest, but not the best, approach is simply to replace your current version with the new version. There are two problems with that approach: (1) You will not know what changes were made and (2) you will loose all your highlighting and comments. Fortunately, Word provides a way to solve both problems. Here is how you can use it. Let’s suppose that your current document, which might, for example, be Chapter 1 is called Chapter 1.doc.

1. Copy Chapter 1.doc into a file called old_version.doc, but do not open it.
2. Copy the new version into a file called new_version.doc.
3. Open new_version.doc.
4. Click on the Review tab.
5. Click Compare. A drop-down menu will appear. Click Compare there too.
6. Another drop-down menu will appear. Click Compare there as well.
9. Click Compare

10. At this point you will see a three documents in separate windows with many annotations in the document called the Compared Document, each of which indicates how the old version must be changed to become the new version. Click the “x” at the top right of the other documents to remove them from view.

11. At the left, you will find a window in which these changes are listed. There are six categories of changes. The first of these is the only one you need to be concerned with: Main document changes and comments. Under that category there are five types of changes: Insertions, Deletions, Moves, Formatting, and
Comments. You will want to keep all changes other than those that eliminate your highlighting. To undo the elimination of highlighting, scroll down and find every instance of “Not Highlight”. Click right and then click Reject Format Change. (The comments that you have added will remain unless you specifically delete them.)

12. Now, scroll through the document. Look at each change and accept it. To accept a change, click the right mouse button anywhere inside it and select Accept Format Change, Accept Deletion, Accept Insertion, etc. as appropriate. (To reject a change, click the right mouse button anywhere inside it and select Reject … as appropriate.)

As you accept changes, observe which changes are trivial corrections of typographical errors and which are important enough to take note of in your studies. You may even decide to add new highlights or comments as you examine a change. You can do that at any time during this process as well.

13. When you have finished accept and rejecting changes to produce your new annotated version, save it as Chapter 1.doc.

At this point you have updated your personal copy to the latest version, have inspected each change, and have kept all your own annotations.

Software Versions


A note about fonts

We have used the following set of fonts in this book:

- **Section headings are Arial Bold 14 pt.**
- **Section subheadings are Times New Roman Italic 12 pt.**
• Normal text is Times New Roman 12 pt.

• **Vocabulary terms are Times New Roman Bold Italic 12 pt.**

• *Mathematical symbols (not Matlab) are Time New Roman Italic 12 pt.*

• **Text in the Matlab Command Window is Courier New Bold 12pt, 11pt, or smaller.**

• **Text in a Matlab M-file is Courier New(non-Bold) 12pt, 11pt, or smaller.**

• **Text in the Matlab taskbar, menus, and submenus and filenames are Arial 12 pt or smaller.**

**Acknowledgements**

This book was developed at Vanderbilt as part of a course for first-year students in the School of Engineering. We would like to thank over a thousand students, who, during the years from 2000 through 2010, have used it, endured its early versions, and provided valuable feedback that has contributed to its current form.

J. Michael Fitzpatrick
John D. Crocetti
Chapter 1
Introduction

Computers are capable of remarkable feats of data manipulation and numerical calculation, but they do only what they are told to do. They manipulate and calculate at incredible speed and have enormous memories, but unless they are told exactly what to do, they are useless. Humans (so far) are in control, and they control computers by giving them commands. A list of commands that control a computer is a computer program. This book will teach you how to write computer programs and will do that by teaching you a programming language called Matlab, which is very popular among engineers and scientists.

Matlab is one of many programming languages that have been developed since the advent of computers in the 1940s by researchers in the discipline of computer science. Computer science is the study of algorithms for processing information with computers, and the goal of each of the languages developed by computer scientists is to provide a means for describing those algorithms. An algorithm is a precise step-by-step procedure for solving a problem. The differences among computer programming languages arise primarily from the particular problems that are to be solved by means of algorithms. Matlab is a special-purpose language that is an excellent choice for writing small programs (we’ll define “small” to be having fewer than a thousand lines) that solve problems involving the manipulation of numbers. It is especially appropriate when the person who will use the program is also a Matlab programmer (for example, the same person who wrote the program). The design of the language makes it possible to write a powerful program in a few lines. The problems may be relatively complex, while the Matlab programs that solve them are relatively simple: relative, that is, to the equivalent program written in a general-purpose language, such as C++ or Java. These two languages are hugely popular and are studied in computer science curricula all over the world. They are wonderful languages that are the result of 60 years of research in computer science, but they are wonderful only when they are appropriate. For many applications in engineering and science, they are not wonderful at all. For these applications, they are an extremely poor choice when compared with Matlab. Before we begin to learn how to use Matlab, it is a good idea to learn when it is a better choice than a general-purpose language, such as C++ or Java, and when it is not. We will begin our discussion of the choice by means of a simple example.

Suppose you are taking a class in which 3 tests are given and the final course average is determined by taking a weighted average of your grades on those tests. The weighting is such that the first test counts 40 units, the second counts 15 units, and the third counts 35 units. To calculate the average grade, you would use a formula like this:
\[ a = \frac{w_1 g_1 + w_2 g_2 + w_3 g_3}{w_1 + w_2 + w_3}, \]

where \( w_1, w_2, w_3 \) are the weights, \( g_1, g_2, g_3 \) are the grades, and \( a \) is the average. No one would write a computer program to perform such a simple calculation! Let’s do it anyway, and let’s write two versions, one in Matlab and one in C++. (We could use Java in place of C++. The results would be very similar.) Here is our Matlab program:

```matlab
grades = input('Please input grades: ');
weights = input('Please input weights: ');
Weighted_average = sum(ggrades.*weights)/sum(weights)
```

Here is that Matlab program in action:

```
Please input grades: [95 90 100]
Please input weights: [40 15 35]

Weighted_average =

96.1111
```

The program itself typed “Please input grades:” and “Please input weights:”. The user typed in “[95 90 100]” and “[40 15 35]”. The program then typed “Weighted_average = 96.1111” (skipping a line as it did so). The brackets ([]) are required, by the way. If the user were to omit them, the program would print a message stating that the user had made an error and would then stop without giving an answer.
Now, here is our C++ program:

```c++
#include "stdafx.h"
using namespace std;
void main(void)
{
    const int   N = 3;
    float weights[N];
    float sum_weights;
    float grades[N];
    float sum;
    float average;
    int   i;

    cout << "Please input 3 grades: ";
    for(i = 0; i<N; i++){
        cin >> grades[i];
    }

    cout << "Please input 3 weights: ";
    for (i = 0; i<N; i++){
        cin >> weights[i];
    }

    sum = 0;
    sum_weights = 0;
    for(i = 0; i <N; i++){
        sum += grades[i]*weights[i];
        sum_weights += weights[i];
    }

    average = sum/sum_weights;
    cout << "Weighted average = " << average << endl;
}
```

and here is the C++ program in action:

```
Please input 3 grades: 95 90 100
Please input 3 weights: 40 15 35
Weighted average = 96.1111
```
The program itself typed “Please input 3 grades:” and “Please input 3 weights:”. The user typed in “95 90 100” and “40 15 35”. The program then typed “Weighted average = 96.1111”.

Without understanding anything about programming, it is possible to see one important difference: The Matlab program is much simpler than the C++ program. Because of that simplicity, it is possible (1) to write the Matlab program more quickly and (2) to do it with fewer errors. This simplicity is the major reason for using Matlab. The simplicity factor is even greater, though, than might be appreciated at first. Note that in the Matlab program we did not include anything about entering exactly “3 grades” and “3 weights”. Let’s see what happens when we run the same Matlab program and enter, say, 5 grades and 5 weights:

```
Please input grades. [70 82 94 77 89]
Please input weights. [40 15 35 45 20]
Weighted_average =
81.0645
```

It works perfectly. In fact, there is no restriction whatever on the number of grades and weights, except there must be the same number of each. This little program is very versatile.

Now, let’s try this with the C++ program:

```
Please input 3 grades: 70 82 94 77 89
Please input 3 weights: 40 15 35 45 20
Weighted average = 79.8447
```

It seems to work, but it does not work. The average is wrong. It used 70, 82, and 94 as grades, and it used 77, 89, and 40 as the weights. The numbers 15, 35, 45, and 20 were ignored entirely! This is an example of the worst kind of error. The program gives a reasonable looking result, but the result is wrong. A less insidious error would be one in which the program refused to run: Not giving an answer is much better than giving an incorrect answer. It is possible to rewrite the C++ program so that it can handle an arbitrary number of grades (and equal number of weights), just like the Matlab program, but doing that requires that the C++ program be far more complex than the version we have given above. That is why we said that the simplicity factor is even greater than might be appreciated at first.
This versatility of Matlab programs is another major advantage, and it is one of the ingenious aspects of this language. The inventors of Matlab were able to envision numerical problems in a very general way, and, as a result, they succeeded in designing a language that allows you to write simple programs that can handle a wide variety of inputs. It is common for a Matlab programmer to design a program for a certain type of input and then realize that the program, without change, can handle much more.

This simplicity comes at a price. Matlab is the winner for small programs that involve numerical applications and are written for the use of Matlab programmers. C++ becomes the better choice when any of these three qualities are missing. Much of the complexity of the C++ program is there because it reduces the likelihood of error when large programs (more than a thousand lines of Matlab or more than ten thousand lines of C++), such as parts of a computer’s operating system (e.g., Linux, Mac OS, Windows XP), graphics applications, or spreadsheets, are to be written. It is also the better choice when non-numerical applications, such as word processors, web browsers, or database programs are involved. Finally, when the program is to be used by a Matlab programmer, who knows Matlab’s rules and its special requirements, such as delimiting inputs by brackets (e.g., [95 90 100], instead of 95 90 100 in the example above), running the Matlab program is no problem. But when an application, such as a tax calculator or the controller for a microwave oven, is to be used by non-programmers, which includes at least 99% of typical users, the problem tends to be better solved by C++.

In the remainder of this book, you will be led through many examples that fit the three requirements above, i.e., small programs involving numerical applications to be used by Matlab programmers. While these may appear to be restrictive requirements, many thousands of engineers and scientists, researchers and practitioners, both in academics and in business, have found that almost all the programming problems that they encounter fit this profile. For them, and for you, if you plan to study engineering or science or are already working in these areas, when the question is “Which language should I use to solve my problem?”, the right answer will almost always be—Matlab.
Chapter 2
Getting Started with Matlab

To gain the most from this book, you should, as you read it, be continually writing and testing examples in Matlab. To run the examples in this book you should have Version 7 of Matlab installed on your computer. The version you have can be determined by clicking on the Help button and then clicking on “About MATLAB”. You will see the Matlab logo pop up with the version written, e.g., Version 7.7.0.471 (R2008b). The first number after the word “Matlab” is the version number (in this case 7); the numbers after the decimals are subversion numbers (this is the non-sinister type of “subversion”!)

2.1 Starting and stopping Matlab

When Matlab is installed on your computer, the Matlab icon should appear:

To start Matlab, double click on the Matlab icon. You should see the "MATLAB Desktop" appear. (Its appearance will vary slightly according to the operating system and version of Matlab.)

The Desktop displays four windows: the “Current Directory” at the left, the “Command Window” in the middle, the “Workspace” at the upper right, the “Command History” at
the lower right. You will learn to type commands into the command window. When you are ready to quit Matlab, you type `quit` in that window, as shown on the next line,

```
>> quit
```

and hit `Enter`. Matlab will then close all open windows and quit. If you have files open, you will be asked about saving open files and given a chance to save them. You can also quit by clicking with the left mouse button on the × at the top right of the Matlab desktop.

### 2.2 Current Directory

In Matlab, the term “directory”, is synonymous with the term “folder”, as it is used in Windows and in Apple’s operating systems. The term “directory” is the term used by Unix and Linux systems and by Matlab. Since we are using Matlab, we will use the term “directory”. When Matlab is running, there is a special directory, called the "Current Directory" where Matlab expects to find files that you want to open and will store files that you want to save. You should change this directory from the one selected by Matlab to the one in which you want to keep your files. To do this, move your mouse over the three arrows, pointed at in the figure by the arrow. A button will appear. Click on the button.

When you click on this button, a window will pop up showing the directory structure that is accessible from your computer. Click on the directory you wish to use and then click OK. You will then see that the name of the current directory that appears in the space to the left of the button with the three dots has now changed to the one that you chose.

#### 2.2.1 Path

When Matlab fails to find a file that it is looking for in the Current Directory, it does not give up. It looks in other directories. It follows a path searching in one directory after another. A “path”, in this context, is a list of directories through which a program searches for a file. The notion of a search path occurs in operating systems such as Unix, Windows, and Mac OS and in many programming environments. Matlab comes with a path already set up, but you can change it by adding or removing directories.
Adding a directory to your path.
You should add to your path any directories in which you keep files that you have created for use with Matlab. You may choose to use only one directory for this purpose, or you may wish, for example, to have a different one for each project you are involved in, or for each homework assignment in a programming course. In any case, you should put the names of these directories at the bottom of the list, i.e., at the end of the path, not the top.

Let’s add a directory to the end of your path. First, create a new directory using your operating system (e.g., Windows XP, Mac OS) and then, within Matlab, add the directory that you have created to your path as follows:

Click on File at the top left of the Matlab desktop and then click on Set Path… The Set Path window will pop up (See figure). This window contains some buttons on the left and a display of the path on the right. The beginning of your path is at the top of the display. The entire path can be viewed by using the scroll button at the right. Click Add Folder... and a window will pop up called, “Browse for Folder”. Inside this window, click on the directory that you wish to add and then click OK. The directory will
appear at the top of the path. Finally, click **Move to Bottom** to move the directory to the end of the path. While you are in that window you can add as many directories as you wish. If you are running Matlab on your own machine and you want to have this directory appear on your path the next time you run Matlab, click **Save** to save this path for the next time your run Matlab. If you want to leave this window open you can click on the “**Minimize**” button in the upper right corner of the window ( ). Otherwise, click **Close**. If you have not saved the path, a box will appear saying "**Do you wish to save the path for use in future MATLAB sessions?**" Again, if you are running Matlab on your own machine and you want to have this directory appear on your path the next time you run Matlab, click **Yes**. Otherwise, click **No**.

**Removing directories from your path.**

To remove a directory from your path, you will use the same window that you used in the previous section. If is not already open, click on **File** at the top left of the Matlab desktop and then click on **Set Path…** When the **Set Path** window pops up, use the scroll bar in the display at the right to find a directory that you wish to remove. Click on that directory’s name to highlight it and then click the **Remove** button at the left. While you are in that window you can remove as many directories as you wish. Again, as in the previous section, if you are running Matlab on your own machine and you want to have the changes that you have made appear on your path the next time you run Matlab, click **Save** to save this path for the next time your run Matlab. If you want to leave this window open you can click on the “**Minimize**” button in the upper right corner of the window ( ). Otherwise, click **Close**. If you have not saved the path, a box will appear saying "**Do you wish to save the path for use in future MATLAB sessions?**" Again, if you are running Matlab on your own machine and you want to have this directory appear on your path the next time you run Matlab, click **Yes**. Otherwise, click **No**.

### 2.3 The Command Window

The symbol **>>** in the command window (the window at the right of the Matlab desktop) indicates that Matlab is ready to obey your command. The generic name for a symbol, or symbols, used by a program (Matlab is a program) to request input is **prompt**. Thus, Matlab is using the symbols **>>** as a prompt. When you are ready to quit Matlab, you can type **Ctrl-Q** (while depressing the Ctrl key, press q) or select **Exit MATLAB** from the **File** Menu at the top left, or type **quit** and hit **Enter**. If you ever find Matlab taking too long to execute a command, you may type **Ctrl-C** to abort the command (while depressing the Ctrl key, press c).

When you type a command into the Command Window and then hit the **Enter** key, Matlab immediately executes your command and prints its response. For example, the command

```
>> x = 1 + 2
```
produces the response

\[
x = 3
\]

This command tells Matlab to add 1 to 2 and then assign the result to a variable called \(x\). Matlab executes that command and then show you the new value of \(x\). If you wish at any time to learn the value of \(x\), you simply type it, and Matlab will respond as follows,

\[
\text{>> } x \\
x = 3
\]

This value will not change until \(x\) is assigned a new value by later statement. If you ask for the value of a variable that has not yet been given a value, Matlab complains,

\[
\text{>> y} \\
\text{??? Undefined function or variable 'y'.}
\]

The act of assigning a value to \(x\) caused a new variable named \(x\) to be defined and set up a location in memory for its value to be stored. While you might not think about it at first, the value of the variable \(x\) must be stored in some location in the computer’s memory, and that location cannot be used for any other purpose until Matlab exits. The definition of variable (in computer science, as opposed to mathematics) is in fact a named location in memory. Each time you assign a value to a new variable, Matlab puts the name of that variable into a table and sets aside enough room to hold its value. As you use more variables, you use more space. If at any time, you wish to see the variables that you have defined, you can use the \texttt{whos} command, as for example,

\[
\text{>> whos} \\
\begin{array}{llll}
\text{Name} & \text{Size} & \text{Bytes} & \text{Class} \\
c & 1x1 & 8 & \text{double array} \\
x & 1x1 & 8 & \text{double array} \\
y & 1x1 & 8 & \text{double array}
\end{array}
\]

The columns \texttt{Size} and \texttt{Class} will mean more to you later. The column \texttt{Bytes} shows how much space Matlab has allocated for each variable. A byte is 8 bits. A bit is the smallest unit of memory. It can store one of two values: 0 or 1. The number of different values that can be stored in \(N\) bits is \(2^N\).

\textit{Interpretation versus compiling}

The execution of a command given to a computing environment, such as Matlab, is termed \textit{interpretation} of the command. Matlab \textit{interprets} the command as soon as you hit the Enter key. This immediate response is different from the situation with a so-called
In compiled language, such as Java, C, C++, or FORTRAN, in which you write a program from beginning to end and then run it. In these languages, unlike Matlab, before the commands that you type can be run, they must be translated from the language that you are using (e.g., C++) into a language that the computer can understand. This translation is called compiling. Note that the word "interpret", which roughly means "translate" outside the computing community, in fact means "execute" when it refers to a computer language. The word "compile", which roughly means gather and organize outside the computing community, means "translate" when it refers to a computer language. You should learn these strange new meanings right now! Matlab provides an option for compiling, but Matlab is instead used primarily as an interactive language, which means a language in which the user continually sees the results of his or her commands as soon as they are issued. An interactive language is always an interpreted language.

Suppressing printing
Unless you tell it otherwise, Matlab will print the result of a command in the command window, as it did with the \texttt{x = 3} above. If you do not want the result printed, you can type a semicolon (\texttt{;}) after the command. With the semicolon, the command will still assign the value 3 to \texttt{x}, but it will not print the value of \texttt{x}. Try setting \texttt{x} to 5 using the semicolon. Then, to see the value of \texttt{x}, type \texttt{x} alone (and then hit \texttt{Enter}). You should see this:

\begin{verbatim}
>> x=5;
>> x
\end{verbatim}

\begin{verbatim}
x =

5
\end{verbatim}

Spaces
You may have noticed that in the example above, there is no space before or after the equal sign, while in the previous example there were spaces. The spacebar can be used to put spaces where they would occur naturally without having any effect, other than to make the command easier to read. As we will see in the next example, variables in Matlab can have more than one letter and can even include digits, but the letters and digits cannot have spaces between them. Thus, examples of legal names, which are also known formally as identifiers, are \texttt{SNOBOL} and \texttt{Wat4}, but \texttt{SNO BOL} and \texttt{Wat 4} are illegal because they include spaces. That should not be surprising. You would not expect to find spaces inside a name. We will give more precise rules for variable names below.

Continuing a command to the next line
You can indicate to Matlab that a command is to be continued on the following line by typing three dots (periods):

\begin{verbatim}
>> dime = 2 + ...
\end{verbatim}

dime = 

10

Putting more than one command on a line
You can indicate to Matlab that a command is completed by using either a comma or a semicolon. After that, you can begin another command on the same line. The comma indicates that a result should be printed. The semicolon suppresses printing. For example,

\[ \text{>> weight} = 4, \text{acceleration} = 9.8; \text{velocity} = 4.5; \]

weight = 

4

\[ \text{>> acceleration, velocity} \]

acceleration = 

9.8000

velocity = 

4.5000

Here we see that the first line typed into the Command Window, which has semicolons after the assignments to both acceleration and velocity, causes only the weight to be printed, while the second line, which has no semicolons, causes both acceleration and velocity to be printed.

Repeating a command
Hitting the up-arrow key (↑) (not the one in the numeric keypad) will cause the previous command line to appear. Hitting Enter will then cause it to be executed. Repeated pressing of the up-arrow will cause earlier and earlier lines to appear. Pressing the down-arrow key will then cause later commands to appear. When a command has appeared, it may be altered by moving the cursor with the left-arrow and right-arrow keys and then using Backspace and/or Delete, and/or typing new characters. The resulting command is executed only when Enter is pressed. Try typing \[ x = 4 + 1 \] without hitting Enter and then altering it to \[ x = 4 - 1 \] and then hitting Enter.

You can also repeat commands by double clicking them in the Command History window at the lower left of the Matlab desktop. You can also highlight, drag, and drop them from anywhere into the command window and execute them by hitting Enter.
Recording a session with the **diary** command

When you are first learning Matlab, you may wish to review the commands you entered in the Command Window and the responses that Matlab gave you. You can record a session of commands and responses by means of the **diary** command. Here is an example of its use:

```matlab
>> clear
>> x = 1
x =
   1
>> diary('My First Session.txt')
>> whos
    Name      Size            Bytes  Class     Attributes
    x         1x1                 8  double
>> y
??? Undefined function or variable 'y'.
>> y = 2;
>> diary off
>> z = 45
z =
   45
>> diary on
>> whos
    Name      Size            Bytes  Class     Attributes
    x         1x1                 8  double
    y         1x1                 8  double
    z         1x1                 8  double
>> x+y+z
ans =
   48
>> diary off
>> t = 7
 t =
   7
```

Now, if we go to the operating system and look at the file, *My First Session.txt* (double clicking on it should work), we will see this:
Chapter 2  Getting Started with Matlab

Note that even Matlab’s error message showed up! (Diaries should record the bad with the good. It helps us learn from our mistakes!)

**Syntax and Semantics**

Matlab's commands must be typed correctly. If they are not, then an error message is given:

```
>> 1 = x
??? 1 = x
| Error: The expression to the left of the equals sign is not a valid target for an assignment.
```

Matlab is trying to tell the user what is wrong. In this case, the user probably does not realize that the equal sign does not mean "is equal to", but instead means "assign the value of the right side to the variable that is on the left side". This is an **assignment statement**, and constants, such as 1, 2, or –18.9 cannot be assigned values. Thus, constants cannot appear on the left side of an equals sign.

This error is a good example of the violation of the proper form of a Matlab statement. The proper form of a Matlab statement is its **syntax**. A violation of the form is called a **syntax error**. Violations with regard to the form of any computer language (and any spoken language for that matter) are called syntax errors. The reason for this particular **syntactical** rule (“syntactical” is the adjective form of “syntax”) is that only named variables can be assigned values. Putting a constant on the left side of an equals sign does
not fit the meaning of the assignment statement, and it is not allowed. Thus, it is a syntax error.

We call the meaning of a statement, as opposed to the form of a statement, the **semantics** of the statement. So here we have an error that violates both the syntax and the semantics of the assignment statement in Matlab. (The word “semantics” is a singular noun. Thus we might say, “The semantics is simple, or, “The semantics needs to be well understood”.) Note that despite the error, Matlab indicates its readiness for the next command by typing another prompt (>>).

**Variable names**
The syntax of Matlab allows a variable’s name, more formally known in computer science as a variable’s “identifier”, to be a single letter, such as `x`, or a word, such as `weight`. In mathematical expressions, as opposed to programming expressions, we are restricted to single-letter names for variables. Because of that restriction we know that consecutively written letters indicate multiplication. Thus, in mathematics `x = cat` would mean that the values of `c`, `a`, and `t` should be multiplied together and that the value of `x` would be equal to the resulting product. We will see below that multiplication is indicated explicitly in Matlab by typing an asterisk (*, which is shift-8 on most keyboards) between the two variables to multiplied. Thus `x = cat` in mathematics would translate to `x = c*a*t` in Matlab. The use of the asterisk to indicate multiplication is very common in programming languages. It was used for that purpose in the very first major programming language FORTRAN (“an acronym for “Formula Translator”), which was invented in the late 1950s, and it is still used today in Fortran and in both C++ and Java as well. Because programming languages use a symbol to indicate multiplication, they can allow an identifier to consist of more than one letter. In the version of Matlab used in writing this book, the name of a variable may include up to 63 characters, which may be upper or lower case letters. Longer names are legal but characters after the 63rd are simply ignored. (The function `namelengthmax` gives the maximum.) Matlab distinguishes between upper and lower case, so, for example, `X` and `x` are two different variables, as are `hOmEr` and `HoMeR`. Additionally, any of the characters in the name, other than the first one, may be a digit or the underscore (_). These rules are almost identical to those for identifiers in C, C++, and Java. Here are some examples of legal Matlab, C, C++, and Java identifiers:

- `mass145square`
- `weight_with_shoes_on`
- `Init_velocity`
- `hA_RDt_oR_ead_But_legal`

**2.4 Getting help with Matlab from Matlab itself**
Matlab stores values in a form that includes more than the mere 5 digits that you see in the examples above. If you want to see more of those digits printed on the screen, you can use the command `format`:

```
>> format long
>> X
X =
     1.00000000000000   2.00000000000000   3.00000000000000
     3.40000000000000   3.14159265358979  -4.00000000000000
```

As you can see the numbers are now printed with more digits, which in Matlab’s terminology (borrowed from Fortran) is a longer “format”. There are other formats available. To learn what they are, you can use the "help" facility. The most direct way to use it is to type the command `help` followed by a space followed by the name of the command that you want help with. For example,

```
>> help format
```

```
FORMAT Set output format.
All computations in MATLAB are done in double precision.
FORMAT may be used to switch between different output
display formats as follows:
FORMAT         Default. Same as SHORT.
FORMAT SHORT   Scaled fixed point format with 5 digits.
FORMAT LONG    Scaled fixed point format with 15 digits.
. . .
```

There are more formats displayed by this command, but we did not show them all above. The help command will be of great value to you as you learn Matlab. Whenever you forget how some command `c` works, type `help c` at the command prompt.

If you don’t know the name of command or can't remember it, the command `lookfor` might come to the rescue. The command `lookfor x` searches through the on-line keywords (first lines of `help` outputs) given for every command, searching for the word `x`, and it shows you the associated commands. For example,

```
>> lookfor pseudoinverse
PINV    Pseudoinverse.
```

This response tells you that there is a Matlab command named `pinv` that has something to do with pseudoinverse. To learn what that is, you would then type `help pinv` (`help PINV` works too.) Note that `lookfor` and `help` are (fortunately) not case sensitive. To learn more about `lookfor` and to see the names of some other helpful commands type `help lookfor`.

In addition to the help available with the commands `help` and `lookfor` in the command window, there is a more graphical help system available from Matlab’s “Help
Navigator*, which can be reached via the command `helpwin` or via mouse clicks. To access that system, type `helpwin` at the command prompt, or if you want help on a particular command, such as the `format` command, type

```matlab
>> helpwin format
```

To access this same graphical help system via mouse clicks, click on the **Start** button at the bottom left of the Matlab window. A menu will pop up. Slide up to **MATLAB**. A companion menu will appear at the right. Slide onto that menu and then down to **Help** and click again. Alternatively, click on **Help** in the Matlab taskbar and choose **Product Help** or just hit the F1 key on your keyboard. The Help window will pop up with the so-called **Help Navigator** on the left and helpful links in a panel on the right, as shown on the next page.
Follow any links that appear interesting. This help facility is very intuitive and easy to navigate. If you like animated demos, then, after clicking on the start button, slide up to, and click on, Demos (alternatively click on the Demos tab in the Help Navigator, or click Help in the Matlab taskbar and choose Demos).

(The graphical help system can also be summoned by hitting the F1 key, but the navigator does not appear.)

**2.5 The Edit Window and "M files"**

So far we have focused on the command window. The command window acts like a calculator—a really, really powerful calculator. You type a command, and Matlab interprets it (i.e., executes it) and shows you the result immediately (or complains that you made a syntax error!). The command window provides the interaction that makes Matlab an interactive language. It is a great way to try simple commands, but it is a poor way to do anything that involves more than, say, 5-10 steps. To accomplish more complicated tasks, you should produce a so-called M-file. That is done by using the Edit Window. There are three ways to get the edit window: (1) Click the little
white blank-page icon at the left end of the Toolbar at the top of the Matlab window, (2) Click File/New/M-File, or (3) type `edit` in the Command Window and hit Enter. An **Edit Window** will pop up:

If you type into this window, text will appear, but hitting Enter will cause nothing to happen except moving to the next line, because Matlab does not interpret (execute) these commands as you type them. Type the command

```
x=5
```

inside this window. Now save what you have typed into a file called `myfirst.m` by clicking **File/Save** in this Edit window, changing directories to the same directory that you chose earlier for your "current" directory, typing in the file name, and clicking **Save**.

You have just created an “M-file”. The “M” stands for Matlab, and the file extension is always `.m`, which is pronounced “dot m”. So another commonly used name for an M-file is a “Dot-M file”. Now go back to the command window and type `myfirst` (**without** the `.m`, i.e., not `myfirst.m`). You should see this:

```
>> myfirst

x =

   5

>>
```
Matlab has checked to see whether you have defined a variable named `myfirst` (you haven’t), then it looked inside its current directory for a file named `myfirst.m`, found it, and interpreted the commands it found inside. Congratulations! You have written a (very short) Matlab program, stored it in a file, and run it. You ran it by typing the name of the file without the .m extension. It is because of this extension that the files containing Matlab programs are called **M-files**. The conventions regarding file extensions for other languages, such as Fortran, C, C++, and Java are less strict, but typical extension for these languages are .f, .c, .cpp, and .java.

Within the edit window, put a semicolon after `x=5`. Save the file again by clicking on the diskette icon near the upper left. Now return to the command window and type `myfirst` again. As with commands issued in the command window, the output that was originally produced by the command `x=5` is now suppressed. This suppression of output is much more important for programs written in M-files than for commands issued in the command window, because typically only the final result of a long set of commands should be printed. In fact the vast majority of commands in the typical M-file will include a semicolon to suppress printing.

**Source code versus executable code**

A **program** is a sequence of symbols that describes an algorithm. An **algorithm** is a step-by-step procedure for solving a problem. The text that you enter into an M-file by using the edit window is a sequence of symbols that describes an algorithm, and therefore it is a program, but it is sometimes also called **code** or **source code** or **source**. This strange use of the word “code” comes from the fact that the earliest programs (i.e., in the 1940s and 50s) were written in a form that resembled a code and, like a code, it was very difficult for a human to decipher. The reason for the modifier “source” is that, before the computer runs the code that a programmer writes, it typically must translate it into an equivalent program called the **executable code** that is written in another language that is more efficient for a computer to run. Thus, the program that a human writes is not the program that the machine runs, but it is the **source** of the program that the machine runs. The language of the executable code is as difficult to decipher as the early languages were, and for that reason it is very hard to determine how a program works by looking at the executable code and even more difficult to modify the executable code. Companies who produce programs for sale are eager to distribute their executable code (also known as the “executables”) but rarely their source code, thereby keeping their proprietary programming tricks secret. Similarly, your instructors in a programming course can give you an executable version of a solution to a programming assignment so that you can run it and thereby see first-hand how it is supposed to behave without your being able to see how it works.

The text that you type is called alternately the “program” or the “code”, and writing such a program is called alternately **programming** or **coding**. The person who writes the program is the **programmer** (but, for some reason, not the “coder”). It is interesting to note that despite the fact that we tend to think otherwise, the program that the programmer writes is never executed by any computer; it is merely the source of the executable computer program, and almost no one ever sees a program that runs.
Software
A set of files containing source code or executable code or both that describes a single
program or a set of programs is called software to distinguish it from the hardware that
makes up the physical part of a computer. The disciplines of Computer Science and
Computer Engineering each deal with software and hardware, with the former
emphasizing the software and the latter emphasizing the hardware. The software written
by scientists and engineers to solve the problems of their disciplines tends to be focused
on numerical applications describing the behavior of physical systems, such as buildings,
bridges, chemical plants, automobiles, aircraft, audiovisual systems, medical devices, or
even kitchen appliances. Their programs tend to be written in a language, like Matlab,
that is tailored to their applications. Because of the design of the language, powerful
programs can be written in a few hundred lines of source code and can be written by one
person. With these programming languages, the programmer is able to focus on the
physical aspects of the system instead of the logical aspects of the program. The software
written by computer scientists and computer engineers to solve problems of their
disciplines tends, on the other hand, to be written in languages, like C, C++, or Java, that
are designed to handle more general applications, often describing non-physical systems,
such as insurance policies, bank accounts, spreadsheets, databases, payroll systems,
reservation systems, or general document processors (e.g., Microsoft Word, which was
used to write the words that you are reading). Because of the design of these languages
and the complexity of the programs, hundreds of thousands of lines of source code are
required. Such programs are rarely written by one person. A subfield of Computer
Science, called Software Engineering, studies the problems that arise when large groups
of people design large programs and the equally important problem of identifying and
correcting the many unfortunate, but inevitable, errors, or “bugs”, that are present in the
code that they write. In Chapter 5 we will use one of the techniques for program design
that were developed by software engineers.

P-code
As we pointed out in under Interpreting versus compiling in Section 2.3 above, the
Matlab system interprets the code that you type into the command window as soon as you
complete a command and hit the Enter key. What we did not mention there is that the
first phase of the interpretation is a translation into executable code. In Matlab, the
executable code has the special name, p-code. In Java, the executable is called
“bytecode”. In Fortran, C, and C++, and many other languages, it is called simply
“executable” code. When the name of an M-file is part of the command executed in the
command window, the entire program in the file is translated into p-code before the
execution phase. For efficiency, the p-code is also saved in memory as long as Matlab is
running and until the M-file is modified, so that the next execution of the program (before
modification) does not require the translation step. You can translate the contents of any
M-file into p-code and save it into a file for distribution by giving the command, pcode
name, where name is the name of the M-file without its .m extension. The program will
be translated into p-code and written into a P-file that has the same name as the M-file
but with the .m extension replaced by .p. The P-file will be placed into the current
directory. One important aspect of p-code is that it is version dependent. Thus, while an M-file written in version 7.0 of Matlab will run under Matlab 17 (we should live that long!), a P-file written under 7 may not run under any previous version of Matlab. An important aspect of a P-file is that, if the files name.p and name.m (i.e., two files with the same name except for different extension) are present in the same directory, Matlab will always run the .p file instead of the .m file. This can cause great confusion if the programmer modifies the .m file and fails to remove the .p file. When the command name is issued, the .p file will be run, ignoring the changes in the .m file.

Comments
Matlab understands commands that are given it in proper syntax. Humans can understand them too, but you can help other humans understand what you are doing by including additional text that is meant to be read only by humans. As an example, you might type in your name at the top of the M-file to show that you are the author. If you do that, Matlab will complain that you have made a syntax error (unless your name happens also to be the name of an active variable or a function that takes no arguments!). To keep that from happening, you must tell Matlab to ignore the line that contains your name. That is done by starting the line with a percent sign (%). It is customary to include such information about the production of the file in a header at the top. For example, for submitting homework solutions in a class you might be required to include the following information, in this order:

- Your name
- Your section and the registrar’s label for the class
- The date of submission
- The name of the assignment

such as a student has done below for a class called “CS 103”,

```
% Author: Nickolas Zeppos
% Section 1, CS 103
% August 22, 2009
% HW Assignment 1
```

Text such as this, which is included in the program but is ignored by Matlab, is called a comment or comments. We are showing the comment text in a green font because the Matlab editor shows it in a green font as well. Matlab uses color to make it easier to read the text, but color has no effect on the meaning of the text. Comments can also be included on the same line as a Matlab command. The rule is that everything following the % up to the end of the line is ignored by Matlab:

```matlab
number = 6  % number of eggs in basket
```

---

3 Two percent signs together (%%) have a special significance. They indicate the beginning of an “M-file cell”, which is a set of statements that can be run separately from the rest of the code. We will not use this facility, but to find out more, you can search for “Rapid Code Iteration” in Matlab’s Help Navigator.
Every modern language provides some means for including comments, and there is always some character or characters used to indicate those parts of the program that are comments. C, C++, and Java, for example, all use two forward slashes (//) instead of a percent sign. Surprisingly, comments may be more important to a program than the executable commands! That is because most programs must eventually be changed, and humans, who rely on comments to make sense of the code, must make those changes.

Using the path to find an M-file
Inside the edit window, change the 5 to a 6. Then, using File/Save As…, save the result to a new file called mysecond.m in the directory that you added to your path above. Type the name mysecond into the command window. You should see this new version run. Matlab has found this new file in the directory that you added to your path and has run it.

2.6 The Figure Window
So far, the output that Matlab has produced has been simple text inside the Command Window. Graphical output is more fun. To get graphical output, you simply use a command that produces it, such as plot. To try out plot, go back to the command window and create two vectors with the commands,

>> x = [1 2 3 4 5 6 7 8];
>> y = [1 2 3 5 8 13 21 34];

A vector in mathematics is simply an ordered list of numbers, and each number is called an element of the vector. As we will see in the next chapter, a vector can be created in Matlab, as shown above, by using a pair of brackets ([ ]) to enclose a list of numbers separated by spaces, commas, or both. The two commands above set \( x \) and \( y \) equal respectively to two 8-element vectors. Now give the following command:

\[ \text{plot}(x,y) \]

A plot appears in a “Figure” window, which pops up automatically when you give the plot command. If you want to close the figure window, either click on the \( \times \) in the upper right corner or give the command close. If you want to keep that window and put your next plot into a new figure window, give the command figure before the next plot command. If you have more than one figure window open, close will remove only the last one that appeared. If you want to get rid of all figure windows, give the command close all. Your plot should look like this:
It should be clear that \texttt{plot(x,y)} plots the vector \textit{y} vertically versus the vector \textit{x} horizontally. These vectors must be of the same length. For each value of the index, \textit{n}, from \texttt{n} = 1 to \texttt{n} = 8, \textit{x(n)} gives the horizontal position of the point that is plotted, and \textit{y(n)} gives the vertical position. There are many variations possible to this simple plot. Dotted or dashed lines may be used, individual dots may be plotted without being connected by straight lines, symbols other than dots may be plotted, various colors may be used, labels may be put on the axes, etc. You can get an idea of its versatility by trying \texttt{help plot}. We will give more examples in Chapter 3.

Figure windows are used whenever graphical output is required. There are many commands in addition to \texttt{plot} that produce graphical output. All of them put their graphical output into a figure window. As an example, let’s display a color picture. One of the official logos of Vanderbilt University is a letter V whose inside is the outline of an oak leaf and an acorn. The image can be stored digitally in many file formats. One such format, which is useful for simple cartoon-like pictures, is the so-called “GIF” (pronounced either “gif” or “jif”) format. If we put a file, named \texttt{Vandy\_symbol.gif},
which contains this logo in GIF format, into a directory that is on the Matlab path, and issue the following commands, the figure below will pop up:

```matlab
>> [V,map] = imread('Vandy_symbol.gif','GIF');
>> image(V);
>> truesize;
>> colormap(map);
```

The first command causes a two-dimensional array of numbers to be stored in the variable `V`. The second command causes it to be displayed. The computer display screen is composed of pixels ("picture elements"), each of which is capable of displaying one color at a time. The command `truesize` causes each value in the array to be displayed by one pixel. The array `V` is 99 rows by 99 columns. Therefore, thanks to `truesize` the displayed image occupies 99 by 99 pixels. The coordinate tick marks and labels are not usually included with images. They can be hidden by means of the command `axis off`, as in the next example.

This version of the Vanderbilt logo contains only two colors (white and gold), but full-color images can be readily displayed as well. Full-color formats are supported by the commands `imread` and `image`, including the “JPEG” format, which is well suited to display photographic images. To explore the possibilities of image display, you can use the command `help imread`. Here is one last image to put you in a peaceful, academic mood, preceded by the commands that produced it:

```matlab
>> Alumni_lawn = imread('Alumni1.jpg','JPG');
>> figure(2);
>> image(Alumni_lawn);
>> truesize;
>> axis off;
```
Chapter 2  Getting Started with Matlab

**Concepts from this chapter**

- Computer Science and Mathematics:
  - search path
  - prompt
  - variable
  - interpreting versus compiling
  - interactive language
  - identifier
  - syntax
  - semantics
  - assignment statement
  - program
  - algorithm
  - source code versus executable code
  - comments
  - prompt
  - vector and vector element

- Matlab:
  - current directory and path
  - command window
  - prompt
  - variable
  - suppressing printing with semicolon
  - identifier
  - extending a command across lines
  - multiple commands on one line
  - assignment statement
  - the `help` and `lookfor` commands
  - the `format` command
  - edit window
  - M-file
  - p-code and P-file
  - comment
  - figure window
  - vector and vector element
  - `plot` and the plot window
Chapter 3  
Matrices, Operators, and Functions

The basic unit with which we work in Matlab is the matrix (plural is “matrices”). We add them, subtract them, multiply them, and perform many other operations on them. In this chapter, we introduce the basic idea of the matrix. We show how Matlab allows you to define matrices and to operate on them. In addition to providing simple operations on matrices, such as addition or subtraction, which are stated using the usual plus sign and minus sign of mathematics, Matlab provides hundreds of other operations that are carried out by calling built-in functions, such as \( \text{sqrt}(x) \), which is a function that calculates the square root.

3.1 Matrices

The name "MATLAB" stands for "Matrix Laboratory", which suggests (correctly) that Matlab is designed to deal with matrices. A matrix is a 2-dimensional rectangular arrangement of numbers, such as this “2x3”, matrix, which has 2-rows and 3-columns:

\[
\begin{bmatrix}
1 & 2 & 3 \\
3.3 & 3.1416 & -4
\end{bmatrix}
\]

A matrix is a useful form for dealing with sets of two or more equations involving sets of two or more variables, a situation that arises repeatedly in all branches of science and engineering. Scientists and engineers also work with scalars (single numbers). Surprisingly perhaps, Matlab treats a scalar as a 1x1 matrix! In fact even the \( x \) in \( \text{sqrt}(x) \) above is treated in Matlab as if it were a 1x1 matrix! To see the size of a matrix in Matlab, you can use the built-in function called size:

\[
\text{>> size(x)} \\
\text{ans} = \\
1 \quad 1
\]

A function in mathematics is any operation that produces a result that depends only on its input. In Matlab, as in most other programming languages (C, C++, Java, Fortran), a function is any operation that is invoked by giving its name. The major distinction between these two definitions is that a mathematical function will always produce the same output for a given input, whereas that is not necessarily true for a function provided by a programming language. Input to a function is given as a list of values separated by commas inside a pair of parentheses that follows the name. Each such value is called an argument (in both mathematics and programming). Here the size function was given only one argument, \( x \), (so no commas are needed), and it produced as its result two numbers—1 and 1. The first 1 represents the number of rows, or height, of the matrix \( x \) and the second 1 is its number of columns, or width. It is possible to make larger matrices by using brackets and semicolons (;). As another example, let’s create in Matlab the
matrix that we gave at the beginning of this section, assign it to the variable \( X \), and then call \texttt{size} using \( X \) as an argument, as follows:

\[
\begin{align*}
\texttt{>> X = [1 2 3; 3.4 pi -4]} \\
X &= \begin{bmatrix} 1.0000 & 2.0000 & 3.0000 \\ 3.4000 & 3.1416 & -4.0000 \end{bmatrix} \\
\texttt{>> size(X)} \\
\texttt{ans} &= 2 \quad 3
\end{align*}
\]

Note that the matrix is printed in \textit{row-major order}, meaning that all the elements on one row are printed before the elements in the next row. Note also that when the semicolon occurs inside square brackets it means "end of row" (instead of suppressing printing). Hitting the \texttt{Enter} key also indicates end of row, and a comma can optionally be placed after each element within a row. Note that the word \texttt{pi} stands for the constant value for \( \pi \) in Matlab. A \textit{vector} in Matlab is simply a matrix with either exactly one column or exactly one row. They are called, respectively, a \textit{column vector} and a \textit{row vector}. The commands, \( x = [1 4 7] \) and \( y = [1; 4; 7] \) produce a row vector and a column vector, respectively. Try these commands and look at the difference in their outputs. Then check the dimensions of \( x \) and \( y \) with \texttt{size()} . A scalar is treated in Matlab, not only as a 1x1 matrix, but as a vector with only one element.

Matrices are sometimes called \textit{arrays}. For most programming languages, the term array is used exclusively, but for Matlab the choice of term depends on the application, which in turn determines the sort of operations that are performed on them. (Array operations and matrix operations are listed below in the section entitled, \textit{Arithmetic with matrices and vectors}) For example, when the numbers represent values in the cells of a spreadsheet, their arrangement is called an array. When the numbers represent the colors of pixels in an image, the term array is again appropriate. The term \textit{matrix} is strictly appropriate only when the numbers represent the coefficients in a set of linear equations, but both matrix and array are sometimes used generically.

It is possible to have three-dimensional arrays in Matlab and even higher dimensions as well, but few programs employ them. In some applications three dimensions are appropriate to model the three-dimensional space that we live in. In medical imaging for example a computed tomography (CT) image of the human body is a three-dimensional array of intensity values that are high for bone and lower for soft tissue. Magnetic resonance (MR) imaging similarly produces three-dimensional arrays of intensities. In some cases a sequence of these images are combined into a \textit{four}-dimensional array! Medical image researchers world-wide employ Matlab programs to manipulate CT and MR images, but most of their programs deal with two-dimensional slices of the three-dimensional volumes or of four-dimensional time series. One reason for this simplification is that humans can visualize two-dimensional arrangements much more readily than higher dimensions. Another reason is that Matlab is designed for two-dimensional arrays and does not handle higher dimensional arrays nearly as smoothly.
Chapter 3  Matrices, Operators, and Functions

Fortunately, most engineering and science applications do not require these higher dimensional arrays. We will, therefore, focus on the two-dimensional ones. The depiction below shows that the set of arrays includes all matrices, the set of matrices includes all vectors, and the set of vectors includes all scalars.

3.1.1 Complex numbers

Matlab is fully capable of dealing with complex numbers. Complex numbers are numbers that may include the square root of \(-1\), which is imaginary and is symbolized by the letter \(i\) in mathematics. In Matlab, the imaginary part of a complex number is indicated by the suffix \(i\) or the suffix \(j\):

\[
\begin{align*}
\text{>> } z &= 3+4j \\
\text{z } &= 3.0000 + 4.0000i \\
\text{>> } z*z \\
\text{ans } &= -7.0000 + 24.0000i
\end{align*}
\]

Note that Matlab responds with \(i\), instead of \(j\), to indicate the imaginary part of the complex number.

3.1.2 The “colon operator”

An equivalent way to specify the vector \(\mathbf{x} = [1 \ 4 \ 7]\) above is \(\mathbf{x} = 1:3:8\), which means, "Assign \(\mathbf{x}\) to be the vector whose elements begin with 1, are spaced by 3, and go no higher than 8." (The result would be the same if the 8 were replaced by either 7 or 9 in this case.) This shortcut can be used anywhere that a row (not a column) can be specified by giving a list of elements. It is useful for very large vectors or matrices with regularly spaced elements. If the spacing is 1, then there is an abbreviated version available. For example, \(\mathbf{x} = 1:7\) means, "Assign \(\mathbf{x}\) to be the row vector whose elements begin with 1, are spaced by 1, and go no higher than 7." The numbers can be fractional and/or negative, but they cannot be complex. Multiples of these numbers can be used as well. For example, \(7*(1:9)\) is the same as \(7, 14, 21, \ldots 63\). Offsets are also allowed: \(.5 + 7*(1:9)\) is the same as \(7.5, 14.5, 21.5, \ldots 63.5\).
Accessing parts of a matrix
A given element of a matrix can be accessed by giving its row index and column index in parentheses separated by commas. Again, using \( X = \begin{bmatrix} 1 & 2 & 3; & 3.4 & \pi & -4 \end{bmatrix} \), as above, we have

\[
\text{>> } X(2,3) \\
\text{ans } = \\
-4
\]

We can use this access method to change one element of a matrix:

\[
\text{>> } X(2,3) = 7; \\
\text{If we now check the value of } X, \text{ we see that one element has been changed:}
\]

\[
\text{>> } X \\
X = \\
\begin{bmatrix}
1.0000 & 2.0000 & 3.0000 \\
3.4000 & 3.1416 & 7.0000
\end{bmatrix}
\]

This notation is closely related to the standard mathematical notation, in which the two indices are written as subscripts. Thus the mathematical version of Matlab's \( X(2,3) \) is \( X_{2,3} \). In general, \( X(i,j) \) is the same as \( X_{ij} \) (commas are not needed in mathematical notation when the indices are letters instead of numbers).

More than one element can be specified at a time by using the \textit{subarray} operations. These operations access rectangular shaped subarrays within a matrix. They are invoked by giving a vector of integers for each index. Here are three examples. We use the same that we started with:

\[
\text{>> } X = \begin{bmatrix} 1 & 2 & 3; & 3.4 & \pi & -4 \end{bmatrix} \\
X = \\
\begin{bmatrix}
1.0000 & 2.0000 & 3.0000 \\
3.4000 & 3.1416 & -4.0000
\end{bmatrix}
\]

Now, we produce three subarrays:

\[
\text{>> } X(2,[1 2 3]) \\
\text{ans } = \\
\begin{bmatrix}
3.4000 & 3.1416 & -4.0000
\end{bmatrix}
\]

\[
\text{>> } Y = X([1 2],[1 3])
\]
Y =
    1.0000    3.0000
    3.4000   -4.0000
>> Z = X(2, [1 2 3])
Z =
    3.4000    3.1416   -4.0000

Subarrays are more often produced by means of the colon operator. Here are the same
three results accomplished with the colon operator:

>> X(2,1:3)
ans =
    3.4000    3.1416   -4.0000
>> Y = X(1:2,1:2:3)
Y =
    1.0000    3.0000
    3.4000   -4.0000
>> Z = X(2,:)
Z =
    3.4000    3.1416   -4.0000

When the colon is given alone in the second position, as in X(2,:) above, it means "all
columns." If it were in the first position, e.g., X(:,2), it would mean "all rows."

A convenient notation is provided for telling Matlab that the subarray extends to the end
of a row or column:

>> X(1,2:end)
ans =
    2     3
>> X(end,1)
ans =
    3.4000
The colon operator can be used on the left side of the equal sign as well to change rectangular parts of the matrix. For example,

```matlab
>> X(1:end,1) = -4.444
X =
   -4.4440    2.0000    3.0000
   -4.4440    3.1416    7.0000
```

Note that, although we referenced only the first column of X in the statement, the entire matrix is printed (if we do not use the semicolon to suppress printing).

---

**Combining matrices to build new matrices**

Matlab allows you to combine matrices to build new matrices. Matrices may be combined in these ways:

1. Matrices that have the same shape (same number of rows and same number of columns) can be combined in all of the ways that scalars are combined. They can be placed on the same rows or in the same columns, as long as the result is rectangular. For example, if
   
   ```matlab
   A = [1 1 1; 1 1 1],
   B = [2 2 2; 2 2 2],
   C = [3 3 3; 3 3 3]  
   ```
   
   (all have 2 rows and three columns), then the following combinations are legal:

   ```matlab
   >> [A B C]
   ans =
      1     1     1     2     2     2     3     3     3
      1     1     1     2     2     2     3     3     3
   >> [A; B; C]
   ans =
      1     1     1
      1     1     1
      2     2     2
      2     2     2
      3     3     3
      3     3     3
   >> [A B C; B B A]
   ans =
      1     1     1     2     2     2     3     3     3
   ```
2. Matrices that have the same number of rows can be placed on the same row. For example, if $A = \begin{bmatrix} 1; 1 \end{bmatrix}$, $B = \begin{bmatrix} 2 & 2; 2 & 2 \end{bmatrix}$, $C = \begin{bmatrix} 3 & 3 & 3; 3 & 3 & 3 \end{bmatrix}$ (all have 2 rows), then the following combinations are legal:

$$\begin{bmatrix} A & B \end{bmatrix}$$

```
ans =

1     2     2
1     2     2
```

$$\begin{bmatrix} A & B & C \end{bmatrix}$$

```
ans =

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

3. Matrices that have the same number of columns can be placed in the same column. For example, if $A = \begin{bmatrix} 1 & 1 \end{bmatrix}$, $B = \begin{bmatrix} 2 & 2; 2 & 2 \end{bmatrix}$, $C = \begin{bmatrix} 3 & 3; 3 & 3; 3 & 3 \end{bmatrix}$ (all have 2 columns), then the following combinations are legal:

$$\begin{bmatrix} A; B \end{bmatrix}$$

```
ans =

1     1
2     2
2     2
```

$$\begin{bmatrix} A; B; C \end{bmatrix}$$

```
ans =

1     1
2     2
2     2
3     3
3     3
```
4. Combinations of rules 2 and 3 may be used simultaneously as long as the result is rectangular. For example, if \( A = \begin{bmatrix} 1; 1 \end{bmatrix} \), \( B = \begin{bmatrix} 2 & 2; 2 & 2 \end{bmatrix} \), \( C = \begin{bmatrix} 3 & 3 & 3; 3 & 3 & 3 \end{bmatrix} \), \( D = \begin{bmatrix} 4 & 4; 4 & 4; 4 & 4 \end{bmatrix} \), and \( E = \begin{bmatrix} 5 & 5 & 5 & 5; 5 & 5 & 5 & 5; 5 & 5 & 5 & 5 \end{bmatrix} \), then the following combination is legal:

\[
\begin{bmatrix} A & B & C; D & E \end{bmatrix}
\]

\[
\begin{array}{ccccccc}
1 & 2 & 2 & 3 & 3 & 3 & 3 \\
1 & 2 & 2 & 3 & 3 & 3 & 3 \\
4 & 4 & 5 & 5 & 5 & 5 & 5 \\
4 & 4 & 5 & 5 & 5 & 5 & 5 \\
4 & 4 & 5 & 5 & 5 & 5 & 5 \\
\end{array}
\]

3.1.3 The transposition operator

The "orientation" of a matrix may be changed by transposing it, or taking its transpose, which means to interchange all its elements so that \( X(m, n) \) is replaced by \( X(n, m) \). The operator that transposes a matrix is the transposition operator \( ' \) (i.e., one single quote---the one leaning to the right, usually on the same key as the double quote). This operator, which is also called the transpose operator, comes after the matrix that it operates on:

\[
\begin{bmatrix} H \end{bmatrix}
\]

\[
H = \begin{bmatrix} 1 & 2; 3 & 4 \end{bmatrix}
\]

\[
H' = \begin{bmatrix} 4 & 5; 6 & 1 \end{bmatrix}
\]

The \( H \) in the expression \( H' \) is termed the operand on which the transposition operator operates. In fact we use the term operand for any object on which an operation is performed. (The term is, however, rarely used to refer to the arguments of a function.) An operator, such as the transposition operator, that applies to only one operand is called a unary operator. Another example is the unary minus, \( -H \), which negates all elements of its operand. When the symbol for the operation comes after the operand, as in \( H' \), the
operator is said to be a \textit{postfix} operator. When it comes before the operand as in \( \cdot H \), the operator is said to be a \textit{prefix} operator.

There is actually a bit more to the transposition operator. If any of the elements of the original array are complex, meaning that they have a nonzero imaginary component, then, during the transposition, each such element is replaced by its \textit{complex conjugate}, which means that the sign of the imaginary part is changed. Thus, for example, if we change the example above slightly, as follows, then the transpose will reveal this behavior:

\begin{verbatim}
>> H = [1 + 2i 2 3; 4 5 6]
H =
     1.0000 + 2.0000i   2.0000   3.0000
     4.0000   5.0000   6.0000
\end{verbatim}

(Matlab sometimes gives more decimal places than is necessary.)

\begin{verbatim}
>> H'
ans =
     1.0000 - 2.0000i   4.0000
     2.0000   5.0000
     3.0000   6.0000
\end{verbatim}

To take the transpose \textit{without} taking any complex conjugates, it is necessary to precede the prime (\( ' \)) by a period (\( . \)), or “dot”, as follows:

\begin{verbatim}
>> H.'
ans =
     1.0000 + 2.0000i   4.0000
     2.0000   5.0000
     3.0000   6.0000
\end{verbatim}

Of course if all the elements are real, the two versions of the transpose do the same thing.

\section*{3.2 Matrix arithmetic}

Arithmetic is the manipulation of numbers with addition, subtraction, multiplication, and division. The operations are ordinarily thought of as applying to scalars, but they also apply to matrices, and there are additional operations that can be applied to matrices as well.
Addition and subtraction
You can add and subtract matrices (and vectors, which are also matrices) that have the same size (i.e., same number of rows and columns), producing a resulting matrix of the same size. Thus, for example, if $Y$ is the same size as $X$ above, then $Z = X + Y$ is legal. The meaning of addition, which is also called both "array addition" and "matrix addition", is as follows:

$$Z = X + Y \quad \text{means that for each } m \text{ and } n, \quad Z(m,n) = X(m,n) + Y(m,n)$$

Subtraction works similarly. Both $+$ and $-$ operate on two operands. We call such operators binary operators. In Matlab, as in ordinary algebraic notation and as in most other programming languages, the symbol for a binary operator is placed between the two operands (i.e., $X + Y$, instead of $XY+$ or $+XY$). Such operators are said to be infix operators.

Multiplication
Other binary operators include multiplication and division. There are two types of multiplication—array multiplication, which works like addition, and matrix multiplication, which is the standard multiplication operation used in linear algebra. Array multiplication is specified this way, $Z = X \cdot Y$. Note again the period, or “dot”, before the $\cdot$. (As we have mentioned before, Matlab, like almost all computer languages, uses the asterisk to indicate multiplication.) As with addition and subtraction, array multiplication requires that the two matrices be of the same size. The definition of array multiplication is as follows:

$$Z = X \cdot Y \quad \text{means that for each } m \text{ and } n, \quad Z(m,n) = X(m,n) \cdot Y(m,n)$$

Array multiplication is useful for the sort of operations that occur in spreadsheets, where the cells of a sheet correspond to the elements of the array.

Matrix multiplication is specified this way, $Z = X \cdot Y$ (no period this time). It requires that the number of columns in $X$ be equal to the number of rows in $Y$. The resulting matrix, $Z$, has the same number of rows as $X$ and the same number of columns as $Y$. These relationships are illustrated in the equation that follows, where the arrows of the same color show equal dimensions:

The blue arrows show the “inner dimensions” of $X$ and $Y$ in the multiplication. The inner dimensions must be equal or multiplication is impossible. The red and blue arrows show
the “outer” dimensions of \( \mathbf{X} \) and \( \mathbf{Y} \) in the multiplication. The outer dimensions determine the dimensions of the result. The semantics of matrix multiplication in Matlab is the same as the standard definition from linear algebra: If the inner dimensions are equal (so that multiplication is possible), then the definition of matrix multiplication is as follows:

\[
\mathbf{Z} = \mathbf{X} \ast \mathbf{Y} \text{ means that for each } m \text{ and } n, \quad Z(m,n) = \sum_k X(m,k) Y(k,n)
\]

where the summation extends over all columns of \( \mathbf{X} \) (and corresponding rows of \( \mathbf{Y} \)).

An important special case is when \( \mathbf{Y} \) is a vector. In that case, \( \mathbf{Y} \) must be a column vector, and its number of elements must be equal to the number of columns of \( \mathbf{X} \); \( \mathbf{Z} \) will be a column vector whose length is equal to the number of rows of \( \mathbf{X} \). This case will be especially important when we take up problems in linear algebra in Chapter 4.

**Division**

Division in Matlab has four forms---two array forms and two matrix forms. For the array forms, the sizes of the matrices must be the same. The syntax again includes a period:

\[
\mathbf{Z} = \mathbf{X} ./ \mathbf{Y} \text{ means that for each } m \text{ and } n, \quad Z(m,n) = X(m,n) / Y(m,n)
\]

\[
\mathbf{Z} = \mathbf{X} \cdot \mathbf{Y} \text{ means that for each } m \text{ and } n, \quad Z(m,n) = Y(m,n) / X(m,n)
\]

The two forms of matrix division involve the inverse (or “pseudoinverse”) of a matrix and will be taken up in Chapter 4.

**Exponentiation**

Finally, there are exponentiation operations. \( \mathbf{Z} = \mathbf{X} .^\wedge \mathbf{Y} \) requires that \( \mathbf{X} \) and \( \mathbf{Y} \) have the same shape. It means that \( Z(m,n) = X(m,n) ^\wedge Y(m,n) \), where the caret, \( ^\wedge \), means "raised to the power of". \( \mathbf{Z} = \mathbf{X} ^\wedge \mathbf{p} \) has several meanings depending on the shapes of \( \mathbf{Z} \) and \( \mathbf{p} \), but we will consider here only the case in which \( \mathbf{p} \) is an integer (and a scalar), in which case \( \mathbf{X} \) must be square, which means that the number of rows equals the number of columns. In this case \( \mathbf{Z} \) is the result of multiplying \( \mathbf{X} \) by itself \( \mathbf{p} \) times, using matrix multiplication. Thus,

\[
\mathbf{Z} = \mathbf{X} ^\wedge \mathbf{p} \text{ means that } Z = X \ast X \ast X \ldots \ast X
\]

**Operations involving scalars**

There are \( \mathbf{p} \) \( \mathbf{X} \)s.

The rules above about the required shapes of the matrices for various operators were actually overstated just a bit. When one of the operands of addition, subtraction, multiplication, or array division is a scalar, the other operand may have any shape! For example, \( \mathbf{A} = \mathbf{c} \ast \mathbf{B} \), where \( \mathbf{c} \) is a scalar. The result for this example is that \( \mathbf{A} \) has the same shape as \( \mathbf{B} \) and each element of \( \mathbf{A} \) is equal to \( \mathbf{c} \) times the corresponding element of \( \mathbf{B} \). As another example \( \mathbf{A} = \mathbf{B} + \mathbf{c} \) means that \( \mathbf{A} \) has the same shape as \( \mathbf{B} \) and each
element of $A$ is equal to the corresponding element of $B$ plus $c$. Subtraction works similarly, and so does array division. Matrix division works this way when the scalar is the divisor.

*Combining the colon and arithmetic operators*

By combining the colon and arithmetic operators you can specify matrices with complicated patterns. Here are some examples,

```matlab
>> even = 2*[1:5]

even =
    2    4    6    8   10

>> odd_even = [2*[1:3]+1, 2*[1:3]]

odd_even =
    3    5    7    2    4    6

>> aaa = [ [1:3]'*[1:3] , 0*[([1:3];[1:3];[1:3])]

aaa =
    1    2    3    0    0    0
    2    4    6    0    0    0
    3    6    9    0    0    0
```

### 3.3 Operator Precedence

We have seen that matrices, vectors, and scalars can be operated on by $+$, $-$, $*$, $/$, $\backslash$, $^\wedge$, and the “dotted” versions of these operators (e.g., $.*$). Arbitrarily complex expressions can be formed by combining more than one operation, e.g.,

```matlab
>> x = a*b + c;
>> y = c + a*b;
>> z = a*(b + c);
```

It is clear from our experience with algebra that in the first command above, $a$ and $b$ are multiplied first. Their product is then added to $c$. The same sequence is followed for the second command, even though the plus sign precedes the multiplication sign. On the other hand, in the third command, because of the parentheses, $b$ and $c$ are added first, and their sum is then multiplied by $a$.

We know intuitively what to expect because of our familiarity with these operations, but Matlab has no intuition. It must follow a set of rules to determine which operation follows which. It is clear from the first two commands that the order in which the operations are carried out is not necessarily determined by the order in which they occur.
in the command. Instead, the operators are ranked so that some act before others, regardless of the order in which they appear. From this example, it is clear that * ranks above + because it acts before +. An operator’s ranking is called its precedence. Matlab uses operator precedence to determine the order in which to apply operators in an expression that includes more than one type of operator. The highest precedence belongs to parentheses. The precedence of the arithmetic operators, transposition, and the colon operator (we’ll see other operators later) is given in the table below:

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Parentheses</td>
</tr>
<tr>
<td>2</td>
<td>Exponentiation and Transpose</td>
</tr>
<tr>
<td>3</td>
<td>Unary + and Unary –</td>
</tr>
<tr>
<td>4</td>
<td>Multiplication and Division</td>
</tr>
<tr>
<td>5</td>
<td>Addition and Subtraction</td>
</tr>
<tr>
<td>6</td>
<td>Colon operator</td>
</tr>
</tbody>
</table>

If more than one set of parentheses occur, then more deeply nested sets operate earlier than less deeply nested sets, as in ordinary arithmetic expressions. If more than one multiplication and/or division operator is involved, then the order is left-to-right. The order in which operators of equal precedence act is called the **associativity** of the operators. The associativity of exponentiation, multiplication, division, addition, and subtraction is left-to-right. Thus, for example,

\[
2^3^4 \text{ equals } (2^3)^4 \\
2/3*4 \text{ equals } (2/3)*4 \\
2-3+4 \text{ equals } (2-3)+4
\]

While left-to-right associativity is common in most languages for these operators, Fortran employs right-to-left associativity for exponentiation. Its exponentiation operator is **. Thus the first expression would be written in Fortran this way: \(2**3**4\), and it would be equal to \(2**(3**4)\). There is no exponentiation operation in C, C++, or Java. In these languages \(x^y\) is achieved by calling a function: `pow(x, y)`. Thus, for example, in C and C++, the first expression above must be written as `pow(pow(2, 3), 4)`. This construction not only looks confusing but sounds dangerous. Matlab’s syntax is much more natural.

Associativity has another meaning as well. It is a property of some operators. If an operator has associativity, then the order of application of operators does not change the outcome. Consider multiplication, for example. \((A*B)*C\) is equal to \(A*(B*C)\), and so multiplication has this property of **associativity**. It is **associative**. Consider subtraction, on the other hand, for which \(A- (B-C)\) is not equal to \((A-B) - C\). Subtraction is not associative. It does not exhibit associativity.
Chapter 3

Matrices, Operators, and Functions

36

3.4 Functions
The operations that we have studied so far are based on operators. Another class of
operation is the function, which we defined above in Section 3.1. There we gave just two
examples, sqrt(x) and size(x). Matlab provides many more functions—hundreds
of them in fact. The functions that Matlab provides are sometimes called “built-in
functions”. You will learn later how to write your own functions to add to the set of
functions that is available to you. Functions are always specified by means of their name,
which is an identifier such as sqrt. The rules for a legal function identifier are the same
as those for a legal variable identifier in Matlab, as in most programming languages. (The
rules for Matlab identifiers are given in Section 2.4 above.) The function name is almost
always followed by one or more arguments, separated by commas and enclosed in one
pair of parentheses: function_name(arg1, arg2, …, argN). A few functions,
however, can be called with no arguments. In some programming languages, such as C,
C++, and Java, such function calls require that the function name be followed by an
empty set of parentheses, but in Matlab the parentheses are omitted. Here are two
examples of a no-argument call
>> rand
ans =
0.9501
>> rand
ans =
0.2311

The returned numbers here are “random” numbers uniformly distributed in the range
from zero to one. (The same series of numbers will be produced by this function each
time Matlab is started, so they are not truly random.) This function is useful for testing
programs on arbitrary inputs and for writing programs that simulate systems that behave
in some random way. The function rand serves also as an example of a function that
does not follow the mathematical definition, which requires that its output depends only
on its input. For rand the output depends on the number of times it has been called since
Matlab was started (other ways to alter it’s behavior are described by help rand.)
Most functions are designed to be used as part of an expression. The value that the
function returns is then used by the expression. For example,
>> x = 2 + sqrt(4 + 5);


will cause the value 5 to be assigned to \( x \). Here \( 4 + 5 \) gave 9, and \( \text{sqrt} \), which gives the positive square root of its argument, returned the value 3. That 3 was added to 2 to give 5, which was assigned to \( x \).

Before we continue discussing functions whose values are used in expressions, it is helpful to describe some functions that are used to build special matrices. Table 3.1 lists the most important ones.

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns an ( n ) by ( m ) matrix of</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{zeros}(n,m) )</td>
<td>Zeros</td>
</tr>
<tr>
<td>( \text{ones}(n,m) )</td>
<td>Ones</td>
</tr>
<tr>
<td>( \text{eye}(n,m) )</td>
<td>Zeros except that diagonal elements equal 1</td>
</tr>
<tr>
<td>( \text{rand}(n,m) )</td>
<td>Random numbers uniformly distributed in the range from 0 to 1</td>
</tr>
<tr>
<td>( \text{randn}(n,m) )</td>
<td>Random numbers normally distributed with mean equal to 0 and variance equal to 1</td>
</tr>
</tbody>
</table>

For example, \( \text{zeros}(n,m) \) returns an \( n \) by \( m \) matrix, all of whose elements equal zero. Similarly, \( \text{ones}(n,m) \), returns \( n \) by \( m \) ones. An interesting function is \( \text{eye}(n,m) \), which returns an \( n \) by \( m \) matrix that has all zeros, except for those elements on the diagonal, which are equal to one. The diagonal of a matrix is the set of elements whose indices are equal to each other. Thus, in the matrix \( \text{M} \), the diagonal elements are \( \text{M}(1,1), \text{M}(2,2), \ldots \). Why is it called “eye”? Well, that’s because \( \text{eye}(n,n) \) produces an identity matrix, whose symbol in mathematics is \( I \) (pronounced “eye”, get it?). An \( n \) by \( n \) identity matrix can be multiplied by any other \( n \) by \( n \) matrix \( X \), and the result is \( X \):

\[
I = \text{eye}(3,3)
\]

\[
I =
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

\[
>> X = [1 \text{ } 2 \text{ } 3; \text{ } 4 \text{ } 5 \text{ } 6; \text{ } 7 \text{ } 8 \text{ } 9]
\]

\[
X =
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\]

\[
>> \text{I*X}
\]
Chapter 3  Matrices, Operators, and Functions

```matlab
ans =
    1     2     3
    4     5     6
    7     8     9

>> X*I
ans =
    1     2     3
    4     5     6
    7     8     9
```

**Polymorphism**

In the example above, the argument to `sqrt` was a scalar, which is a 1-by-1 array (Note: We will use the terms array and matrix interchangeably when referring to variables). We could have given `sqrt` an array of any size and shape as an argument. Its behavior in that case is to return an array of the same size and shape, such that each element of the result is equal to the square root of the corresponding element of the argument. Thus, if `X` is a 3x4 array, then `Y = sqrt(X)` results in `Y(m,n) = sqrt(X(m,n))` for all elements of `X`.

In the general study of programming languages, when the type of an argument used in a function can vary (as for example, from a scalar to a vector to a matrix) from one call of the function to the next, the function is said to be **polymorphic**. (The term polymorphic means “having multiple forms”, which matches its definition, since it means that the function call can have more than one form.) The characteristic of being polymorphic is called **polymorphism**. (An alternate terminology is to say that the function’s arguments are polymorphic, but the idea is the same.) As can be seen from the `sqrt` example above, Matlab's support for polymorphism is very powerful, and it makes it possible to handle a huge variety of situations (e.g., a huge variety of different matrix shapes) with relatively few functions. It makes it possible for Matlab to do a great deal of work very efficiently (i.e., quickly) with very few statements. Polymorphism is not supported by the most popular older languages, such as Fortran, C, Pascal, and Basic. Many modern languages, on the other hand, such as Ada, C++, Java, and other so-called "object-oriented" languages (and even a few of the older languages, such as LISP) do provide support for polymorphism. It is in fact a major feature of these languages, just as it is a major feature of Matlab.

Unlike most of the other modern languages that support polymorphism, Matlab’s typical function returns an object that is the same shape as the argument that it is given. For example, if `f` is a typical function, then for `y = f(x)`, if the value of `x` is a scalar, then a scalar value will be returned into `y`. If `x` contains a matrix of dimensions `m` by `n`, the
value returned into \( y \) will also be an \( m \) by \( n \) matrix. When the function allows two or more arguments of different types, the returned type will vary according to the function.

Returning an object whose shape is the same as the argument is not always appropriate, however. For example, the function \texttt{sum}, when given a row or column vector, returns a scalar that is equal to the sum of the elements of the vector:

\begin{verbatim}
>> v = [1 -3 5 10];
>> sum(v)
ans =
   13
\end{verbatim}

When \texttt{sum} is given a two-dimensional matrix, it calculates the sum for each column of the matrix and returns a row vector of those elements:

\begin{verbatim}
>> M = [1 10 100; 2 20 200; 3 30 300]
M =
    1     10    100
    2     20    200
    3     30    300
>> sum(M)
ans =
    6     60    600
\end{verbatim}

Thus, \texttt{sum} is certainly polymorphic, since it accepts either a vector or a two-dimensional matrix as its argument, but it does behave typically by returning an object whose shape is the same as that of the argument.

There is a second type of polymorphism. A function is also polymorphic if it accepts varying numbers of arguments. So, for example, if a function can be called with one argument or with two arguments, it is polymorphic. The function \texttt{sum} exhibits this second aspect of polymorphism as well, as shown by the following call:

\begin{verbatim}
>> sum(M,2)
ans =
   111
   222
   333
\end{verbatim}
The argument $M$ in this call is the same matrix as used above, but this time the summation is carried out along each row, and the resulting sums are put into a column vector. The second argument tells `sum` which dimension it is to sum over and whether to return a row vector (second argument equal to 1) or column vector (second argument equal to 2). Since the second dimension is the row index, the call `sum(M,2)` means to sum across the rows. The call `sum(M,1)` means to sum over the column index, so it returns the same vector as that returned by `sum(M)` above. Thus, the second argument is optional, and if it is omitted, the default dimension is 1. When we describe functions in this book, we will not always list all the optional variables, but you can always learn whether there are optional arguments and their meanings by using `help`.

**Returning more than one object**

Some Matlab functions can return more than one object. A good example is the function `max`. This function can be used to find the largest element in a vector, as in the following example:

```matlab
>> a = max([1 4 -5 0])
a =
   4
```

In this example `max` returned one object, the number 4, which is the largest element, but `max` can do more, as shown in the next example:

```matlab
>> [a b] = max([1 4 -5 0])
a =
   4
b =
   2
```

In this example, `max` returned two objects. The first one is the maximum value; the second one is the index of the element that contained the maximum value. The two variables, `a` and `b`, in brackets on the left side of the equal sign in the call to `max` are called *output arguments* in Matlab. This ability to return more than one object is an important feature of Matlab, and that feature is lacking in the major general-purpose languages: Ada, C, C++, Fortran, Java, and Pascal. Furthermore, the ability to call such a function in different ways so that it returns different numbers of output arguments is a third type of polymorphism, and Matlab is one of the very few languages that provides it. Note that some Matlab functions return a vector that contains more than one object. Returning a vector is not equivalent to returning more than one object. The `size` function, which we encountered in Section 3.2, is such a function. It returns one object, which is a two-element vector. The first element is the number of rows of the argument; the second is the number of columns. When only one object is returned by a function, it is possible to capture it in one variable, as in the following example, in an individual variable. The syntax is obvious from the following example,

```matlab
>> v = size([1 2 3; 4 5 6]);
```
Chapter 3  Matrices, Operators, and Functions

>> v
v =
   2   3

As can be seen from the examples above, because \texttt{max} returns two objects, it is \textit{not} possible with \texttt{max} to capture both the maximum value and the index in a single variable. Matlab does, on the other hand make it possible to capture the two elements of the vector returned by \texttt{size} in two separate variables, as follows;

\begin{verbatim}
>> [m n] = size([1 2 3; 4 5 6])
m =
   2
n =
   3
\end{verbatim}

The number of functions provided with a Matlab installation is huge, and many more are available from the Internet. You have seen a few of them in these tutorials and in Table 2. In Table 3 through Table 8 below you will find a few more of them. (Soon you will learn how to write your own functions.) To learn more about them and to learn the names of many more functions use Matlab’s help facility.

\begin{table}[h]
\centering
\begin{tabular}{|l|p{10cm}|}
\hline
\textbf{Function} & \textbf{Returns} \\
\hline
acos(x) & Angle in radians whose cosine equals x \\
acot(x) & Angle in radians whose cotangent equals x \\
asin(x) & Angle in radians whose sine equals x \\
atan(x) & Angle in radians whose tangent equals x \\
atan2(x,y) & Four-quadrant inverse tangent of x \\
cos(x) & Cosine of x (x in radians) \\
cot(x) & Cotangent of x (x in radians) \\
sin(x) & Sine of x (x in radians) \\
tan(x) & Tangent of x (x in radians) \\
\hline
\end{tabular}
\caption{Trigonometric Functions}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|l|p{10cm}|}
\hline
\textbf{Function} & \textbf{Returns} \\
\hline
\texttt{exp} (x) & \textit{e} raised to the x power \\
\texttt{log} (x) & Natural logarithm x \\
\texttt{log2} (x) & Base-2 logarithm of x \\
\texttt{log10} (x) & Base-10 logarithm of x \\
\texttt{sqrt} (x) & Square root of x \\
\hline
\end{tabular}
\caption{Exponential Functions}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|l|p{10cm}|}
\hline
\textbf{Function} & \textbf{Returns} \\
\hline
\texttt{abs} (z) & Absolute value of z \\
\hline
\end{tabular}
\caption{Functions that Analyze Complex Numbers}
\end{table}
Table 6. Rounding and Remainder Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fix(x)</code></td>
<td>Round x towards zero</td>
</tr>
<tr>
<td><code>floor(x)</code></td>
<td>Round x towards minus infinity</td>
</tr>
<tr>
<td><code>ceil(x)</code></td>
<td>Round x towards plus infinity</td>
</tr>
<tr>
<td><code>round(x)</code></td>
<td>Round x towards nearest integer</td>
</tr>
<tr>
<td><code>rem(x,n)</code></td>
<td>Remainder of ( x/n ) (see help for case of noninteger n)</td>
</tr>
<tr>
<td><code>sign(x)</code></td>
<td>( 1 ) if ( x &gt; 0 ); ( 0 ) if ( x ) equals ( 0 ); ( -1 ) if ( x &lt; 0 )</td>
</tr>
</tbody>
</table>

Table 7. Descriptive Functions: Applied to a Vector

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>length(v)</code></td>
<td>Number of elements of v</td>
</tr>
<tr>
<td><code>max(v)</code></td>
<td>Largest element of v</td>
</tr>
<tr>
<td><code>min(v)</code></td>
<td>Smallest element of v</td>
</tr>
<tr>
<td><code>mean(v)</code></td>
<td>Mean of v</td>
</tr>
<tr>
<td><code>median(v)</code></td>
<td>Median element of v</td>
</tr>
<tr>
<td><code>sort(v)</code></td>
<td>Sorted version of v in ascending order</td>
</tr>
<tr>
<td><code>std(v)</code></td>
<td>Standard deviation of v</td>
</tr>
<tr>
<td><code>sum(v)</code></td>
<td>Sum of the elements of v</td>
</tr>
</tbody>
</table>

Table 8. Descriptive Functions: Applied to a Two-Dimensional Matrix

<table>
<thead>
<tr>
<th>Function</th>
<th>Returns a row vector consisting of</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>max(M)</code></td>
<td>Largest element of each column</td>
</tr>
<tr>
<td><code>min(M)</code></td>
<td>Smallest element of each column</td>
</tr>
<tr>
<td><code>mean(M)</code></td>
<td>Mean of each column</td>
</tr>
<tr>
<td><code>median(M)</code></td>
<td>Median of each column</td>
</tr>
<tr>
<td><code>size(M)</code></td>
<td>Number of rows, number of columns</td>
</tr>
<tr>
<td><code>sort(M)</code></td>
<td>Sorted version, in ascending order, of each column</td>
</tr>
<tr>
<td><code>std(M)</code></td>
<td>Standard deviation of each column</td>
</tr>
<tr>
<td><code>sum(M)</code></td>
<td>Sum of the elements of each column</td>
</tr>
</tbody>
</table>

3.5 Keyboard input

Most Matlab programs, which are written in M-files, will require some input from the user through the keyboard. In a typical situation, a message is printed by the program requesting input from the user. This message is called a prompt. We have already seen
that Matlab’s command window uses a very simple prompt: `>>`. In the programs that you will write, a prompt may be more like a sentence, but regardless of the form of the prompt, the user must respond to it by typing something and hitting the **Enter** key. All this is accomplished in a Matlab program by means of the **input** function. For example, if the M-file containing a Matlab program includes the command,

```markdown
fave = input('Please type your favorite vector: '); 
```

When this command is reached, the program will print in the command window,

**Please type your favorite vector:**

and will wait until the user types in something. Note that there must be **single quotes** around the prompt. Characters inside single quotes in Matlab make up a **string**. The term **string** is used in programming languages in general to mean **a sequence of characters**. A string is useful anytime Matlab, or any other programming language, needs to store or display meaningful words, phrases, or sentences in a spoken language. (We will see later that Matlab provides many special purpose operations for strings. It stores a string as a one-dimensional array of characters.)

While Matlab is waiting for input, a vertical bar (`|`) will blink on and off just to the left of the space into which the user is about to type a response. Note that the prompt above ended with a **blank character** (the space between the colon and the single quote). This blank is "printed" after the colon, meaning that when the user types in a response, there will be one space between the colon and the first character of the response. If that space is omitted, Matlab will not include it for you.

Suppose the user types, not a vector, but the matrix `[1 2; 3 4]`. Then the interaction would look like this:

**Please type your favorite vector:** `[1 2; 3 4]`

The result is that `fave` would be given the value `[1 2; 3 4]`. Note that Matlab pays no attention to what is written in the prompt string. Regardless of the type of the object that the user types in---a scalar, a complex number, a vector, a matrix, or even a string---it will be accepted. The fact that the prompt specifies a vector is of no consequence.

### 3.6 Formatted Output

A program written in an M-file will need to communicate results to the user. So far, we have taken advantage of the fact that a command that is not followed by a semicolon automatically prints any value it computes to the command window. We have also seen that it is possible to change the way in which computed results are displayed by using the **format** command.
This method of communication is workable for simple programs, but Matlab provides a far more sophisticated means for printing information by means of a function, borrowed from the languages C and C++, called `fprintf`. Its name stands roughly for “formatted printing to a file”, and the function is versatile enough to allow us to print either to the Command Window or, as we will see later, to a file. It requires at least one argument. Its first argument (when it is being used to print to a file, it is the second argument) is a string that specifies the format, such as the number of decimal places, of its output to the command window. A string is simply a sequence of characters. To indicate to Matlab that a sequence of characters is not to be interpreted as the name of a variable or function, it is started and terminated with the single quote symbol ('). In that format string, the format of each value that is to be printed is specified individually. So, for example, one value may be printed with 4 decimal places and another with 9. Additionally, any text that is to be printed along with the values is included in that format string. Values that are to be printed are given as arguments, in the order that they are to be printed, following the format. For example,

```matlab
>> fprintf('No.: %d, cost= %5.2f.\nTotal is %7.3f\n', 4, 3.1, 4*3.1);
No.: 4, cost=  3.10.
Total is  12.400
```

**Conversion characters and escape characters**

In this particular format string, the `%d`, `%5.2f`, and `%7.3f` are conversion characters. Instead of being printed, a percent sign (%) and the characters immediately after the percent sign indicate to Matlab that the value of one argument after the format string, is to be printed in a specific format. The percent sign acts as a so-called escape character (a term used in other computer languages as well as Matlab), which tells Matlab that the next character or characters have a special meaning. For each percent sign there will (usually) be exactly one argument after the format string to go with it. One argument's value is printed for each percent sign in the order that they appear. For example,

- The meaning of the `d` in `%d` is "If the value is an integer, print it without a decimal point; otherwise print it in scientific notation."
- The `%5.2f` means "Print using fixed-point notation in 5 positions (counting a minus sign if the number is negative, the digits before the decimal place, the decimal point itself, and the digits after the decimal point), and use 2 of these 5 positions to hold the digits after the decimal point.
- See Table 9 for additional format specifiers.

The meaningful sequence of characters that begins with an escape character is called an escape sequence. Another escape sequence is `\n`, which means "go to a new line". The backslash, which we identified earlier as one of the division operators, has a different meaning when it occurs inside a string. Here is an escape character. The special meaning of `\n` is "newline". To print a backslash use two of them: `\`. To print a single quote, use two of them: `'`.
Table 9. Format Specifiers for fprintf

<table>
<thead>
<tr>
<th>Format Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>single character</td>
</tr>
<tr>
<td>d</td>
<td>decimal notation (but no decimal if integral)</td>
</tr>
<tr>
<td>e</td>
<td>exponential notation</td>
</tr>
<tr>
<td>E</td>
<td>exponential notation with an upper case E</td>
</tr>
<tr>
<td>f</td>
<td>fixed-point notation</td>
</tr>
<tr>
<td>g</td>
<td>shorter of e or f</td>
</tr>
<tr>
<td>G</td>
<td>shorter of e or f but with an upper case E</td>
</tr>
<tr>
<td>o</td>
<td>unsigned octal notation</td>
</tr>
<tr>
<td>s</td>
<td>string</td>
</tr>
<tr>
<td>u</td>
<td>unsigned decimal notation</td>
</tr>
<tr>
<td>x</td>
<td>hexadecimal notation</td>
</tr>
<tr>
<td>X</td>
<td>hexadecimal notation with uppercase</td>
</tr>
</tbody>
</table>

Partial use and recycling of the format string and matrix arguments
If the number of arguments to be printed (i.e., those after the format string) is smaller than the number of percent signs in the format string, then the printing will stop just before the first unused percent sign is encountered. If there are more arguments to be printed than percent signs, the format string will be recycled repeatedly until all of the arguments are printed. If any or all of the arguments is a matrix, then the printing will behave as if the matrix were replaced by its elements written as separate arguments in column-major order.

Imaginary parts ignored
A strange limitation to fprintf is that it cannot be used to print complex numbers properly. If a complex argument is given to it, fprintf will ignore the imaginary part! To print complex numbers, you may use another function called disp. For example,

```matlab
>> z = 2 + 3i;
» fprintf('z = '); disp(z);
z = 2.0000 + 3.0000i
```

Alternatively, you can extract the real and imaginary parts of a complex number by using the functions real and imag. The following example reveals how it can be done:

```matlab
>> fprintf('z = %.2f + %.2fi',real(z),imag(z))
z = 2.00 + 3.00i
```
3.7 Plotting

Matlab has an extremely powerful plotting facility, which is supported by functions, including `plot`, `title`, `xlabel`, `ylabel`, `grid`, `semilogx`, `semilogy`, `loglog`, and many others. With these functions it is possible to (a) plot more than one function at a time on a given plot, (b) select distinct symbols, called "markers", for distinct plots, (c) select distinct colors for distinct plots, (d) print a title with a plot, (e) label the axes, (f) provide a "graph-paper" grid, (g) change one or both axes to be scaled according to the logarithms of the values. Many other options are available, including three-dimensional plotting.

You have already seen the basic approach to plotting. The idea is to create two vectors of the same length and give them as arguments to `plot`. Thus, if \( x \) and \( y \) are two vectors of length \( N \), then `plot(x, y)` produces a plot of \( N \) points with \( x \) giving the horizontal positions and \( y \) giving the vertical positions. It is easy to extend this idea to put more than one plot in a figure because plot is polymorphic. It accepts a varying number of arguments. For example,

\[
\begin{align*}
&\text{>> } x1 = 0:0.1:2*pi; \quad y1 = \sin(x1); \\
&\text{>> } x2 = \pi/2:0.1:3*pi; \quad y2 = \cos(x2); \\
&\text{>> } plot(x1,y1,x2,y2)
\end{align*}
\]

produces the following plot:
The vector \( y_1 \) is plotted versus the vector \( x_1 \) and the vector \( y_2 \) is plotted versus the vector \( x_2 \), separately on the same set of axes. The ranges of the \( x \) and \( y \) axes are automatically chosen to encompass all the values in the vectors. Any number of pairs of vectors may be given to plot. It will plot them all on the same set of axes.

It can be seen from the figure that the color blue is used for the first plot and the color green for the second plot. It is possible to alter the appearance of the plots by using additional arguments in the form of strings. For example, the command

\[
\text{>> plot}(x_1, y_1, 'r', x_2, y_2, 'k:')
\]

produces this version

![Figure No. 1](image)

The string ‘\( r \)’ means “red”. The string ‘\( k: \)’ means black, dotted (\( k \) means black; the colon means dotted). One, two, or three character strings can be included optionally for every pair of vectors to be plotted. Table 10 gives a list of possible symbols and their meanings. A few additional line styles can be found by using \texttt{help plot}.

It is also possible to plot more than one function on the same axes by means of sequential plot functions by means of the command \texttt{hold}. Once the command \texttt{hold on} has been issued all subsequent plots will be made on the same axes with the ranges of the axes being adjusted accordingly. When the command \texttt{hold off} is issued, all
subsequent calls to plot will employ a new figure. The command `hold` (not followed by `on` or `off`) can be used as a toggle to turn plot holding on or off. Thus, the following commands could be used to produce the same plot:

```matlab
>> hold on
>> plot(x1,y1,'r')
>> plot(x2,y2,'k:')
>> hold off
```

There are additional options, not reachable through the plot command that will enhance the visual quality of a plot. The command `grid` turns on a graph-paper like grid. Repeating the command turns it off. The `grid` command affects only the currently active figure. (The “active” figure is the one most recently created or the one which has received the most recent mouse click, whichever happens last.)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Color</th>
<th>Symbol</th>
<th>Line Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>blue</td>
<td>.</td>
<td>point</td>
</tr>
<tr>
<td>c</td>
<td>cyan</td>
<td>o</td>
<td>circle</td>
</tr>
<tr>
<td>g</td>
<td>green</td>
<td>x</td>
<td>cross</td>
</tr>
<tr>
<td>k</td>
<td>black</td>
<td>+</td>
<td>plus</td>
</tr>
<tr>
<td>m</td>
<td>magenta</td>
<td>*</td>
<td>star</td>
</tr>
<tr>
<td>r</td>
<td>red</td>
<td>s</td>
<td>square</td>
</tr>
<tr>
<td>y</td>
<td>yellow</td>
<td>d</td>
<td>diamond</td>
</tr>
<tr>
<td>w</td>
<td>white</td>
<td>-</td>
<td>solid (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>:</td>
<td>dotted</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
<td>dashdot</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--</td>
<td>dashed</td>
</tr>
</tbody>
</table>

A plot can be adorned with a title at the top and labels along the x and y axes by means of the functions,

```matlab
>> title(string)
>> xlabel(string)
>> ylabel(string)
```

each of which takes a single string (the label) as an argument. So, for example, `string` might be `'My nifty title'`, or it might be the name of a variable that contains a string as its value. It is also possible to change the ranges of the axes with the function `axis`, which accepts a single four-element vector as its argument:

```matlab
>> axis([xmin xmax ymin ymax])
```

To return the range to its original value the command
can be used. Notice that this command does not have the form of the normal function with arguments in parenthesis. Another use of the `axis` command is to alter the relative lengths of the x and y axes match. The command

```
>> axis square
```

will make the length of the axes be equal, resulting in a square plot area. The command

```
>> axis equal
```

will adjust the axes so that equal tick mark increments on the x-, y- and z-axis are equal in for x and y. To see other uses of the command, use `help axis`.

Here is an example of the use of all these functions after the previous plot has been displayed:

```
>> grid
>> title('A Sine Plot and a Cosine Plot');
>> xlabel('The argument of sine and cosine');
>> ylabel('The value of the sine or cosine');
>> axis([-2 12 -1.5 1.5])
```
Finally, here is fancy plot that uses all these options. Six plots are done simultaneously, each in a different color:

- A cosine multiplied by an exponential function and plotted in blue
- Four exponential functions
  - two in red
  - two in green
- One black circle.

It doesn’t mean anything, but it is a pretty ending for this chapter!
Chapter 3  Matrices, Operators, and Functions

Concepts from this chapter

- Computer Science and Mathematics:
  - matrix
  - scalar
  - function
  - argument
  - row-major order
  - vector
    - column vector
    - row vector
  - complex numbers
  - operand
  - unary operators
  - postfix and prefix operators
  - transposition
  - matrix arithmetic
  - operator precedence and associativity
  - polymorphism
  - string
  - escape character

- Matlab:
  - matrix
  - scalar
  - function
  - argument
  - vector
    - column vector
    - row vector
  - complex numbers
  - colon operator
  - subarray operations
  - combining matrices
  - transposition
  - matrix operations and array operations
  - operator precedence and associativity
  - important functions
    - building matrices
    - trigonometric
    - exponential
    - complex number analysis
    - rounding and remainder
    - descriptive
  - polymorphism
  - returning more than one object with output arguments
- **input**
- **string**
- Formatted output: `fprintf`
  - format string
  - conversion characters
  - escape characters
- **disp**
- **plot**
  - multiple functions on one plot
  - selecting colors and markers
  - titles, labels, log plotting
Chapter 4
Linear Algebra in Matlab

Mathematical problems involving equations of the form $Ax=b$, where $A$ is a matrix and $x$ and $b$ are both column vectors are problems in linear algebra, sometimes also called matrix algebra. Engineering and scientific and disciplines are rife with such problems. Thus, an understanding of their set-up and solution is a crucial part of the education of many engineers and scientists. As we will see in this brief introduction, Matlab provides a powerful and convenient means for finding solutions to these problems, but, as we will also see, the solution is not always what it appears to be.

4.1 Solution difficulties

The most common problems of importance are those in which the unknowns are the elements of $x$ or the elements of $b$. The latter problems are the simpler ones. The former are fraught with many difficulties. In some cases, the “solution” $x$ may in fact not be a solution at all! This situation obtains when there is conflicting information hidden in the inputs (i.e., $A$ and $b$), and the non-solution obtained is a necessary compromise among them. That compromise solution may be a perfectly acceptable approximation, even though it is not in fact a solution to the equations. In other cases Matlab’s result may be a solution, but, because of small errors in $A$ or $b$, it may be completely different from the exact solution that would be obtained were $A$ and $b$ exact. This situation can have a serious impact on the application at hand. The elements of $A$ and $b$ are typically measurements made in the field, and even though much time, effort, and expense goes into reducing the error of those measurements, the calculated $x$ may have an error that is so large as to render those measurements useless. This situation occurs when the solution $x$ is highly sensitive to small changes in $A$ and $b$, and it is just as important, or in many cases more important, than the problem of the approximate non-solution.

Neither of these problems is Matlab’s “fault”, nor would it be the fault of any other automatic solver that you might employ. It is the nature of linear algebra. The person who relies on a solution to $Ax = b$ provided by Matlab, or by any other solver, without investigating the sensitivity of the solution to errors in the input, is asking for trouble, and will often get it. When accuracy is critical (and it usually is), the reliability of the solution must be investigated, and that investigation requires some understanding of linear algebra. The study of linear algebra can be life-long, and there are many books and courses, both undergraduate and graduate, on the subject. While it is not possible for every engineer and scientist to master the subject, it is possible for him or her to be aware of the nature of the difficulties and to learn how to recognize them in his or her discipline. The goals of this chapter are to give you some insight into the nature of the difficulties associated with the solution of $Ax=b$ when $x$ is the unknown and to show you how to use Matlab to find the solution and to investigate its reliability.
4.2 Simultaneous Equations

Because of the definition of matrix multiplication (See Chapter 3), the matrix equation $Ax=b$ corresponds to a set of simultaneous linear algebraic equations:

$$
A_{i1}x_1 + A_{i2}x_2 + \ldots + A_{iN}x_N = b_i \\
A_{21}x_1 + A_{22}x_2 + \ldots + A_{2N}x_N = b_2 \\
\vdots \quad \vdots \quad \vdots \quad \ldots \quad \vdots \\
A_{M1}x_1 + A_{M2}x_2 + \ldots + A_{MN}x_N = b_M
$$

(4.1)

where the $A_{ij}$ are the elements of the $M$-by-$N$ matrix $A$, the $x_j$ are the elements of the $N$-by-1 column vector $x$, and the $b_i$ are the elements of the $M$-by-1 column vector $b$. This set of equations is sometimes written in tableau form to indicate more clearly the relationship between the matrix form and the equation forms and to show shapes of the arrays:

$$
\begin{bmatrix}
A_{11} & A_{12} & \ldots & A_{1N} \\
A_{21} & A_{22} & \ldots & A_{2N} \\
\vdots & \vdots & \ldots & \vdots \\
A_{M1} & A_{M2} & \ldots & A_{MN}
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_N
\end{bmatrix}
=
\begin{bmatrix}
b_1 \\
b_2 \\
\vdots \\
b_M
\end{bmatrix}
$$

(4.2)

Note that, despite the row-like arrangement of the $x_j$ in Eq. (4.1), $x$ is in fact a column vector. Note also that the length of $x$ is $N$, which is always equal to the width (not the height) of $A$. The length of $b$ is $M$, which is always equal to the height of $A$. We have formed this tableau to indicate clearly that it is a case in which there are more equations than unknowns. In other words $M>N$. While it is more common to have $M=N$, in which case the height of the $x$ vector in Eq. (4.2) is the same height as $A$ and $b$, in many situations $M$ is in fact larger than $N$. The third situation, in which $N<M$, is far less common.

Typically the $A_{ij}$ and $b_i$ are given, and the $x_j$ are unknown. The equations are “simultaneous” because one set of $x_i$ must satisfy all $M$ equations simultaneously. They are “linear” because none of the $x_i$ is raised to any power other than 1 (or 0) and there are no products $x_iy_j$ of unknowns. When $A$ and $x$ are given, the problem is relatively easy, because finding $b$ requires only matrix multiplication. The number of unknowns
(i.e., the length of $b$) is always equal to the number of equations (otherwise $Ax$ cannot equal $b$), and typically the elements of $b$ are not highly sensitive to small errors in $A$ or $x$. More often, however, the problem confronting us is to find $x$ when $A$ and $b$ are given. This problem is much more difficult and is the subject of many books and many courses.

### 4.3 Matlab’s solutions

Most programming languages (e.g., C, C++, Fortran, Java) provide no commands at all for solving either of these linear algebra problems. Instead, the user of the language must find a function written by another programmer or write his or her own function to solve the problem and then include that function in a program. Writing a function to find $x$ is in fact no simple matter, and the likelihood that a programmer unskilled in linear algebra will produce a reliable one is virtually negligible. By contrast, Matlab provides one simple operation to solve each of the two problems: When $A$ and $x$ are given, the Matlab solution is $b = A*x$; when $A$ and $b$ are given, the Matlab solution is $x = A\backslash b$. There are amazingly few restrictions on $A$, $x$, and $b$. As explained in Section 3.2 of Chapter 3, entitled, “Matrix arithmetic”, in order for matrix multiplication to make sense, the number of columns of $A$ must equal the number rows of $x$. Similarly, in order for the division $A\backslash b$ to make to make sense, the number of rows of $A$ must equal the number of rows of $b$. That’s it!

Suppose, for example, we need to solve the equations:

$$
\begin{align*}
4x_1 + 5x_2 &= 6 \\
3x_1 - 2x_2 &= 14
\end{align*}
$$

Here $M = 2$ and $N = 2$. We can set up this problem and solve it in Matlab as follows:

```matlab
>> A = [4 5; 3 -2]

A =

4 5
3 -2

>> b = [6; 14]

b =

6
14

>> x = A\b

x =
```

Suppose, for example, we need to solve the equations:
3.5652
-1.6522

Note that, as always, the number of rows of $A$ is equal to the number of rows of $b$. (The number of columns of $A$ is also equal to the number of rows of $b$, but that equality is not necessary).

We can check the result to make sure that it solves the equation:

```matlab
>> A*x
ans =
   6
  14
```

Since the elements are the same as those of the vector $b$, we can see that Eqs. (1.3) have both been solved simultaneously by the two values $x_1 = 3.5652$ and $x_2 = -1.6522$.

Another way to check is to look at the difference between $A*x$ and $b$:

```matlab
>> A*x - b
ans =
   0
   0
```

Since the difference between each element of $A*x$ and the corresponding element of $b$ is zero, we see again that Matlab has given us the solution.

As we mentioned at the beginning of this chapter, we will soon see that there are some situations in which the result given by $A\backslash b$ does not solve the equations or, while it solves the equations, the solution is not trustworthy. In those cases, Matlab may warn you, but not always. Such situations can cause serious trouble if the user does not know what is going on. The trouble is not the fault of Matlab. It is the fault of the user who is solving problems in linear algebra without knowing enough about linear algebra, or perhaps the fault of the instructor who failed to teach the user enough about linear algebra! We will learn below when this will happen and why it happens, and we will learn the meaning of the result that Matlab gives us in these situation.

In the problem above there were two equations in two unknowns. Because there were two equations, there were also two rows in both $A$ and $b$. Because there were two unknowns, the number of columns of $A$ was also 2 and the resultant answer $x$ had a length of two. Problems such as this one, in which there are $N$ equations and also $N$ unknowns, are the most common linear algebraic problems that occur in all of engineering and science.
They are solved in Matlab by setting up an $N$ by $N$ matrix $A$, and a column vector $b$ of length $N$.

### 4.4 Inconsistent equations

In most cases for which $M = N$ a solution exists, but occasionally there is no solution. The simplest example is given by two equations in which the left sides are the same but the right sides are different, as for example,

\[
\begin{align*}
4x_1 + 5x_2 &= 6 \\
4x_1 + 5x_2 &= 12
\end{align*}
\]

Clearly this pair of equations is inconsistent. A plot of the lines represented by each of the equations will reveal that they are parallel. Thus, they never intersect. If we try to solve this pair of equations using Matlab, it will issue a warning that the matrix is “singular” and will give as its answer Inf for each element of $x$. Inf is a special value that means “infinity” and is simply a sign to the user in this case that the solution is meaningless.\(^4\)

### 4.5 Under-determined problems

Problems for which $M = N$ are not the only problems of importance. In the general case, the number of equations $M$ may not be equal to the number of unknowns $N$. For $M < N$, the problem is under-determined or undetermined, meaning that there is not enough information provided by the equations to determine the solution completely. The simplest example is a single equation in two unknowns:

\[
4x_1 + 5x_2 = 6
\]

Here we have given only the first equation from Eqs. (1.3), so $M = 1$ and $N = 2$. If we solve this equation for $x_2$, we get $x_2 = -(4x_1 - 6)/5$. This relationship shows us that we can choose an infinite set of values of $x_1$ from $-\infty$ to $\infty$, as long as we choose the right $x_2$ to go with it. If Matlab is asked to solve for $x$ in this situation, it will find one, and only one, of the infinite solutions\(^5\):

\[
\gg A = \begin{bmatrix} 4 & 5 \end{bmatrix}; \ b = 6;
\]

\(^4\) This situation and many other special situations are discussed in books on the subject of numerical analysis, such as, Numerical Analysis by R. L. Burden and J. D. Faires, Brooks/Cole, 1997.

\(^5\) How does Matlab decide which of the infinite choices to give you? It chooses one for which as many of the elements as possible are equal to zero. If there is still more than choice, it chooses the one for which the norm of the solution vector is minimal.
Chapter 4  Linear Algebra in Matlab

>> x = A\b

x =

0
1.2000

Inadequate information can sometimes occur when $M=N$ and even when $M>N$. This happens when the information in some of the equations is redundant. For example, it happens for these three equations:

\[
4x_1 + 5x_2 = 6 \\
8x_1 + 10x_2 = 12 \\
-2x_1 + \frac{5}{2}x_2 = -3
\] (1.6)

There is no more information in Eqs. (1.6) than there is in Eq.(1.5). That reason is that each of the second and third equations can be obtained from the first by multiplying all terms by the same number (2 for the second equation and -1/2 for the second). Thus, any two of these equations are redundant relative to the third. The set of solutions to these three equations is the same infinite set of solutions that solve Eq. (1.5).

4.6 Over-determined problems

A more common situation when $M > N$, is that the problem is over determined, meaning that there is too much information provided by the equations. The equations in this case cannot all be satisfied simultaneously by any value of the vector $x$. The Matlab command $x = A\b$ will still result in a value being calculated for $x$, even though that value does not satisfy the equality $A*x = b$! To understand the meaning of the “solution” that Matlab gives when the equations are over determined, we will begin by looking at Eqs. (1.3), which are in fact not over determined. It is helpful to plot the lines defined by the equations. We begin by solving the first equation for $x_2$, and get

$x_2 = -(4x_1 - 6)/5$. To plot $x_2$ versus $x_1$, we set up a Matlab variable called $x1$ that contains an assortment of values. We pick $-10, -9, -8, \ldots, 9, 10$. We then set up a variable called $x2a$ (the $a$ indicates that we are plotting the first equation), which we set equal to $-(4*x1 - 6)/5$. We do a similar thing for the second equation, using $x2b$ this time, and we plot both lines:

```matlab
>> x1 = -10:10;
>> x2a = (-4*x1 + 6)/5;
```
>> x2b = (3*x1-14)/2;
>> plot(x1,x2a,x1,x2b)
>> grid on
>> xlabel('x1')
>> ylabel('x2')
>> legend('x2a','x2b')

The resulting plot is shown below. It can be seen that the two lines cross at approximately the point (3.6, -1.7). Their intersection is the lone point at which both equations are satisfied. In other words, it is the solution of the simultaneous equations. We solved these equations when we encountered them before by using \( \mathbf{x} = \mathbf{A}\backslash\mathbf{b} \). We found that \( \mathbf{x}(1) \), which corresponds to \( x_1 \), equals 3.5652, and \( \mathbf{x}(2) \), which corresponds to \( x_2 \), equals \(-1.6522\). These values can be seen to agree with the intersection point in the plot.

Thus, for the case of two equations in two unknowns, Matlab gives us the exact (to within some small round-off error) solution. We can study a case in which there are more equations than unknowns by considering the following set of equations:

\[
\begin{align*}
4x_1 + 5x_2 &= 6 \\
3x_1 - 2x_2 &= 14 \\
7x_1 - x_2 &= 25
\end{align*}
\]  

(1.7)
which are the same equations as Eqs. (1.3) plus one more. We can plot these three equations using the commands above plus these additional commands:

```matlab
>> x2c = 7*x1 - 25;
>> hold on
>> plot(x1,x2a,x1,x2b,x1,x2c)
```

While it appears that the three lines cross at a point, in fact they do not. To get a better look, we can click on the magnifying-glass icon (just below the word “Help” at the top of the figure window) and then click a few times near the area where the lines appear to cross. Under magnification we will see that the lines do not in fact all cross at the same point, which means that there is no solution to Eqs (1.7). Nevertheless, can ask Matlab to try to “solve” them as follows:

```matlab
>> A = [4 5; 3 -2; 7 -1]
A =
    4    5
    3   -2
    7   -1
>> b = [6; 14; 25]
```
Chapter 4  Linear Algebra in Matlab

The resulting solution point can be plotted on the same graph:

\[
\text{plot}(x(1),x(2),'k*')
\]

The resulting plot (after magnification) shows that Matlab has chosen a point that comes close to all three lines.

The errors in the solution can be seen by comparing \( A\times x \) to \( b \):

\[
\text{error} = A\times x - b
\]

\[
\text{error} =
\begin{align*}
-0.1875 \\
-0.6647 \\
0.3920
\end{align*}
\]

If the solution were exact, all three elements of error would be zero. If an exact solution did exist (i.e., if all three lines crossed at the same point), Matlab would have given us that solution. In all other cases, Matlab gives that solution for which the sum of the squares of the elements of error is as small as possible. This value can be
calculated using Matlab’s built-in function \texttt{norm(x)} which gives the square root of the sum of the squares of the elements of \( x \):

\[
\text{norm}(x) = \sqrt{\sum_{i} x_i^2}.
\]

There are other “norms”, each of which measures the size of the vector in some sense (try \texttt{help norm}), but this is the most common and most useful one. In this case we find that \texttt{norm(error)} gives the value 0.7941. We can try other values for \( x \). Each of them will produce a nonzero error, and the norm of the error will always be greater than 0.7941. A solution for which \texttt{norm(error)} is as small as possible is called the \textit{optimum solution in the least-squares sense}.

### 4.7 Ill-conditioned problems

We have seen that Matlab will solve problems of the form \( Ax = b \) when they have exact solutions, and it will give an optimum solution in the least-squares sense to problems that are over determined. In either of these situations an important aspect of the problem is its stability to input error. While there are several definitions of stability, we will concentrate on the most common one: the effect for a given matrix \( A \) and vector \( b \) that changes in \( b \) have on \( x \). This problem is confronted by engineers and scientists when (a) a problem of the form \( Ax=b \) must be solved for \( x \), (b) the values of \( A \) are calculated or known with high accuracy, and (c) the values of the elements of the vector \( b \) must be measured. There is always some level of error in the measurements, and if \( x \) changes drastically when small errors are made in \( b \), then the problem is said to be \textit{ill-conditioned}. The ill
conditioning depends on both $A$ and $b$, but it is possible to have a matrix $A$ for which the problem is well behaved for any $b$.

To study the phenomenon of ill conditioning, we will consider the following two equations:

\[
\begin{align*}
24x - 32y &= 40 \\
31x - 41y &= 53
\end{align*}
\]

We calculate a solution to these equations using $x = A\backslash b$ as usual and we find that $x = 7$ and $y = 4$. If we perturb the values of $b$ slightly to 40 and 53.1, we find that $x = 7.4$ and $y = 4.3$. While these may not seem to be significant changes it is interesting to make some relative comparisons. Let’s define the following convenient quantities:

- $bp$ is the perturbed version of $b$
- $db$ is the change in $b$: $db = bp - b$
- $xp$ is the perturbed version of $x$: $xp = A\backslash bp$
- $dx$ is the change in $x$: $dx = xp - x$

We will use the norm as a measure of the size of some of these vectors, and we will examine the ratio of some of them. In particular, we find that

\[
\text{rel}_\text{input}_\text{error} = \frac{\text{norm}(db)}{\text{norm}(b)} = 0.0015
\]

and

\[
\text{rel}_\text{output}_\text{error} = \frac{\text{norm}(dx)}{\text{norm}(x)} = 0.0620
\]

These two quantities represent the fractional change, respectively, in the sizes of $b$ and $x$ resulting from the addition of $db$ to $b$. This addition represents a possible imprecision or error in the measurement. Because $b$ is given, we call it the input ($A$ is input as well, but, as we mentioned above, we are not considering the effects of errors in its determination.) Because $x$ is calculated, we call it the output. This is the situation that would arise if errors in the measured values of the elements of $b$ were to change by the amounts in $db$. These errors would result in the changes in $x$ represented by $dx$. The fractional changes represent the seriousness of the changes in comparison to the quantities that are changing. In this case a 0.15% change in $b$ results in a 6.2% change in $x$. Neither of these changes seems impressive, but their ratio is. The relative change in $x$ is 6.2/0.15, or 41 times larger than the change in $b$. Thus the percent error is increased by over four thousand percent! By comparison, if we perturb $b$ for the first set of equations that we studied in this section from 6 and 4 to 6 and 4.1 and do a similar calculation, we find that the percent error is increased by a factor of only 1.1. This second situation is far more stable then the former one.
Matlab provides a function to help you detect ill-conditioned problems. The function is called \texttt{cond}. To use it, one simply calls \texttt{cond} on the matrix \texttt{A}. If \texttt{cond(A)} returns a large value, then \texttt{A} is ill-conditioned; otherwise it is not. The number returned by \texttt{cond} is called the \textit{condition number} of \texttt{A}. Its precise definition is as follows: For any input vector \texttt{b} and any perturbed input vector \texttt{bp}:

\[
\frac{\text{rel\_output\_error}}{\text{rel\_input\_error}} \leq \text{cond}(A) \times \text{rel\_input\_error}
\]

Returning to the two examples above we find that \texttt{cond([24 \ -32; \ 31 \ -41])} returns 530.2481, while \texttt{cond([4 \ 5; \ 3 \ -2])} returns 1.7888. The inequality above is clearly satisfied for both cases, and it is clear that the former is far less stable than the latter.

It is well to remember these examples when confronting any problem of the form \texttt{Ax=b} when the unknown is \texttt{x}. If the condition number of \texttt{A} is large, the problem probably needs to be reformulated with more and/or different measurements taken to avoid having large errors in the output, even when the inputs are measured with relatively high accuracy.

\section*{Concepts from this chapter}

- Computer Science and Mathematics:
  - linear algebra, matrix algebra
  - simultaneous linear algebraic equations
  - under determined
  - over determined
  - norm of a vector
  - ill-conditioning, ill-conditioned
  - relative input error and relative output error
  - condition number of a matrix

- Matlab:
  - \texttt{b = A*x}
  - \texttt{x = A\b}
  - \texttt{Inf}
  - \texttt{norm(x)}
  - \texttt{cond(A)}
Chapter 5
Top-down Design and Branching Statements

This chapter introduces a systematic method for designing programs called “top-down design”, and it also introduces a new class of programming statements, called “branching” statements. The two are related because branching statements are often used in the design method. We begin by considering the design method. It is one of many methods developed by the researchers in program design who deal with large and complex problems. Their discipline is called Software Engineering.

5.1 Software Engineering

Matlab programs are written to solve problems. The problems may be so simple as to require only a few lines of code, or so complicated that dozens of pages of code are involved. The simple ones can often be written intuitively with little planning. The complicated ones require considerably more skill. Learning how to get started, what to do next, and how to reduce the number of errors in the code all become more important as the problems get more complex. To a certain extent, anyone who wants to become skilled in solving problems by writing programs must learn by practice. However, there are some techniques that have been developed by experienced programmers that can make the typical novice programmer more efficient more quickly. These techniques become more important as the problems get more complicated and as the programs that solve them get larger. This chapter deals with one of those methods.

Simple problems that require programs of no more than fifty to one hundred lines of code or so can solved by one person in one sitting. They are typically used only once, or a few times, to find the answer to a small set of problems. An example is the problem of calculating a course grade from the scores on graded assignments and the weights of these scores. Complex problems, on the other hand, may require programs of thousands, tens of thousands, or even hundreds of thousands of lines of code. Writing such large programs typically requires more than one person and involves months or years of work, and they are typically used many times by people who often know little or nothing about programming. An example is a game-playing program or a spreadsheet designed for calculating taxes. The solution of such complex problems is so difficult that an entire discipline has arisen to study them. That discipline is called Software Engineering. The goals of Software Engineering are to understand the source of the difficulties of solving large programming programs and to develop effective methods for writing good programs in the face of these difficulties in a predictable period of time. “Good” programs are those that not only have few errors, but also are easy to modify. Errors, also called “faults” or “bugs”, are frustrating for the user and also expensive and, in some cases, dangerous. Ease of modification reduces the effort necessary both to eliminate errors when they are discovered and to add functionality when it is needed. Many studies have shown that good software engineering, which attacks both these problems, saves
time, money, and even lives. One of the design techniques developed by software engineers is called “top-down design. It is that technique that we present in this chapter.

5.2 Top-down design

Software engineers made the important discovery early on that the program to solve a problem should not be written until a design phase has been completed. The design phase is done on paper, or with the help of special software that provides “computer assisted software engineering” (CASE). This phase, whether it is done on paper or with CASE tools, involves no programming. In other words, no code is written during this phase. One of the best design methods that have been developed by those who study software engineering is called top-down design. Top-down design is the process of describing a problem as a set of simpler problems and then describing each of these subproblems, in turn, as a separate set of even simpler problems, and so on, until elementary subproblems are encountered that are so simple that they can be easily solved directly (i.e., without applying top-down design). The process of describing a problem as a set of simpler problems is called decomposition. The repeated application of decomposition to the resulting subproblems until elementary problems are encountered is called stepwise refinement. Thus, top-down design is the process of describing a problem by means of decomposition and stepwise refinement. Once the top-down design phase is complete, and only when it is complete, the resulting description of the problem can be translated into code. That code is tested on example inputs. Testing may reveal errors in the design phase.

A five-step approach to top-down design

The method of top-down design, when it is applied to very large, complex problems, is facilitated by many special techniques for improving the efficiency of large group projects. We will confront smaller problems which can be solved effectively by the following five-step approach to program design:

1. State the problem
2. Define the inputs and outputs
3. Design the algorithm
   a. Decompose into subproblems
   b. Perform stepwise refinement
4. Convert the algorithm into a Matlab program (by typing into the editor window)
5. Test the program (by running the program in the command window)

Usually the most difficult part of this process is step 3, “Design the algorithm”. The algorithm, which we defined in Chapter 2 as a step-by-step procedure for solving a problem must be described precisely and unambiguously, not to mention correctly. Translating a vague solution method into an algorithm is a central issue of computer science. It is not programming, but it is an important adjunct to programming. It is during the design of the algorithm that vagueness or inconsistencies in the solution method first become apparent. Removing those vague and inconsistent aspects of the solution and
reducing the required number of steps or the required amount of computer memory to solve the problem can be accomplished much more efficiently at this stage than during the coding stage. For that reason, skill in algorithm design is often considered to be the most important skill in computer science. It can be seen also from the five-step approach above that both decomposition and stepwise refinement, which are the essential aspects of top-down design involve only algorithm design. They are not part of the conversion to a program.

_Pseudocode_
Thus, the top-down design phase comprises steps 1-3. If this phase is done well, then the actual writing of the code, which is done in step 4, is relatively straight-forward. It is especially easy if the designer is skilled in the writing of _pseudocode_, which is a combination of English (or French, German, Chinese, or whatever your native language) and Matlab (or C, C++, Java, Fortran, or whatever target programming language is to be used for the programming). The first two steps are often said to describe a _black box_, which is any system whose inner workings are unknown but whose inputs and outputs are known. The overall process, which starts with inputs and produces outputs, is known too, but the manner in which it is implemented (e.g., by a Matlab program) is not. In fact “State the problem, is equivalent to “describe the overall process that is to be implemented”. There is rarely any pseudocode used in that description. The pseudocode is used primarily in step 3, and it becomes more Matlab-like (assuming now that we are programming in Matlab) with each refinement.

_An iterative process_
While it is not explicit in the five-step approach shown above, top-down design is an iterative process. (In fact any design technique is iterative.) In step 3b above, “refinement” implies that this step is iterated (repeated) until the subproblems reach the elementary stage described above, in which further refinement is no longer necessary. This iteration may involve only two or three steps, or for more complex problems it may involve dozens of steps. We will see below that there is another need for iteration, a need caused by errors that the designer makes during the design process. What follows is a very simple example of an application of the five-step approach to top-down design.
Example of the five-step approach to top-down design for Matlab

1. State the problem:
   Multiply a matrix times a vector and give the sum of the resulting elements.

2. Define the inputs and the outputs
   Input is a 3x3 matrix and a 3-element column vector.
   Output is the sum of the elements.

3. Design the algorithm
   a. Decompose into subproblems
      - Subproblem 1: Read inputs
      - Subproblem 2: Carry out calculations
      - Subproblem 3: Print result
   b. Perform 1st stepwise refinement:
      - Read inputs
        i. Get a 3x3 matrix from input and copy it into M.
        ii. Get a 3x1 vector from input and copy it into x.
      - Carry out calculations
        i. \( y = M \times x \)
        ii. \( \text{output} = \text{sum of elements of } y \)
      - Print result in one-line sentence.
   Perform 2nd stepwise refinement:
   - Read inputs
     i. \( \text{matrix } M = \text{input('please give a 3x3 matrix')}; \)
     ii. \( 3x1 \text{ vector } x = \text{input(similar prompt)} \)
   - Carry out calculations
     i. \( y = M \times x \)
     ii. \( \text{output} = \text{sum of elements of } y \)
     iii. \( \text{fprintf("The sum of the elements is ", output);} \)

4. Convert the algorithms into a Matlab program:

```
M = input('Please give a 3x3 matrix');
x = input('Please give a 3x1 vector');
y = M*x;
output = y(1) + y(2) + y(3);
fprintf('
The sum of the elements is %f\n', output);
```

5. Test the program:

```
>> multiply_and_sum
Please give a 3x3 matrix [1 2 3; 1 2 3; 1 2 3]
Please give a 3x1 vector [1;2;3]
The sum of the elements of the product is 42.000000
>>
```
Design Errors

One major difference between the example shown above and actual experience is that the example is perfect: The designer made no mistakes in steps 1-4, nor did the programmer (the same person in this case) make any mistakes in step 4. Thus, step 5 revealed no errors. A more typical situation would find errors in all steps! Syntax errors, which are made in step 4, will be caught by Matlab’s interpreter during step 5. An example might be writing $y = Mx$ instead of $y = M \times x$. These errors are easy to correct. The more difficult problems are those caused by semantic errors in steps 3 and 4. Such errors will cause the program to produce incorrect errors, but are not noticed by the interpreter. In other words, the program will run, but it produces incorrect results. An example might be writing \texttt{output = y(1) + y(1) + y(3)}. Semantical errors must be caught during step 5. When such errors are noticed, it is necessary to return to the step where they first appeared, correct them there, and retrace the design from that point. An important aspect of step 5 is the test data used as input. That input must be chosen carefully so as to catch as many errors as possible. For example, the input given above would not catch the semantical error given here because $y(1)$, $y(2)$, and $y(3)$ are all equal.

Thus, as mentioned above, the design process is an iterative one for two reasons. Within step 3, stepwise refinement requires that problems be repeatedly broken into simpler subproblems. That makes the process iterative, but, in addition, when errors are made during the design process, some of the steps must be repeated to correct the errors. Most of the time the repeated steps are 3 and/or 4.

5.3 Branching statements

So far, each Matlab program that we have encountered consists of a sequence of commands, each one of which is carried out immediately after the one that came before it in the program. This flow of control from one statement to the next one after it in the order that they are written in the program is called sequencing, or sequential control. It is one of the control constructs provided by Matlab. A control construct is a method by which the interpreter selects the next statement to be executed after a previous statement is concluded.

Matlab, like all other modern programming languages, provides other control constructs as well. One of these, which is called selection, makes it possible for one block of statements to be selected from a set of two or more with the choice depending on the values of variables. For example, consider the following code:

```matlab
x = input('Please type a non-zero integer: ');
if x > 0
    fprintf('Thanks for your positive response!\n');
elseif x == 0
    x = 2;
    fprintf('Zero is not allowed. I changed it to 1.\n');
else
    fprintf('Please enter a non-zero integer.\n');
```

This example illustrates the use of the \texttt{if}, \texttt{else}, and \texttt{elseif} constructs in Matlab.
fprintf('Nice choice!
');
end

These lines will cause “Thanks for your positive response!” to be printed if the user inputs a positive number. If the user types 0, then \( x \) will be immediately changed to 2 and the program will print, “Zero is not allowed. I changed it to 1.” (Some programs tell lies! There is a bug in this program.) If the user types a negative number, the message “Nice choice” will be printed instead. The following diagram, called a “flow chart”, depicts schematically the flow of control through these statements:

This selection construct is called an \textit{if statement}, or \textit{if construct}. It is an example of a \textit{selection statement}, also known as a \textit{branching statement}, or a \textit{branch statement}, or simply, a \textit{branch}. The expressions \( x > 0 \) and \( x = 0 \), above are called \textit{conditionals}, which are expressions whose result determines whether a section of code is to be executed. The statements between the keywords \texttt{if} and \texttt{elseif} make up a \textit{block} that will be executed if and only if \( x \) is greater than zero. A \textit{keyword} is an identifier that is
defined by the language to have special meaning. By default the Matlab editor highlights its keywords in blue. We have also done that below to make the statement structure a bit clearer. The elseif clause and/or the else clause may be omitted, leaving

\[
\begin{align*}
&\textbf{if } \text{conditional} \text{ block else block end}, \\
&\text{or} \\
&\textbf{if } \text{conditional} \text{ block elseif conditional block end,} \\
&\text{or} \\
&\textbf{if } \text{conditional} \text{ block end.}
\end{align*}
\]

Furthermore, additional elseif clauses may be added (no limit). In all cases, no more than one block will be executed by an if construct. If there is no else clause, then, if none of the conditionals are satisfied, all of the blocks will be skipped and the if statement will do nothing. If there is an else clause, however, it is guaranteed that one block will be executed. Each part of the if construct must begin on a new line or be separated from the previous part by a comma or semicolon. The semicolon suppresses printing as usual.

Matlab has a second form of the selection statement called the switch statement, or switch construct:

\[
\begin{align*}
&\textbf{switch } \text{switch_expression} \\
&\textbf{case } \text{case_expression}, \\
&\quad \text{block} \\
&\text{case } \text{case_expression}, \\
&\quad \text{block} \\
&\quad \ldots \\
&\textbf{otherwise} \\
&\quad \text{block} \\
&\textbf{end}
\end{align*}
\]

Here switch_expression is any scalar or string and case_expression is any Matlab expression. The flow of the switch construct is the same as that of the if-statement depicted above. Exactly one block will be executed by the code above. It will be either the first block whose case expression equals the result of the switch expression or, if none of the case expressions equal the switch expression, then the block between otherwise and end will be executed, much like the else clause at the end of the if-statement. The otherwise clause may be omitted. If so, then it may be that no block is executed, so, as for the similar case with the if-statement, the switch-statement will do nothing. The commas may be replaced by semicolons. If the block starts on a new line, as in the layout above, neither a comma nor a semicolon is required.

The switch-statement and the if-statement are equally powerful (each can do what the other can do), but the switch-statement is especially well suited to handle cases with only a finite set of choices For example, the following two pieces of Matlab code below are equivalent, but the switch statement is easier to read and matches the task better:
if x == 1
    fprintf('One');
elseif x == 2
    fprintf('Two');
elseif x == 3
    fprintf('Three');
else
    fprintf('%d', x);
end

switch x
    case 1
        fprintf('One');
    case 2
        fprintf('Two');
    case 3
        fprintf('Three');
    otherwise
        fprintf('%d', x);
end

There is one alternative method of specifying cases in the switch construct. Any case_expression may be replaced by a list of case expressions in braces:

case {case_expression_1, case_expression_2, ...},
    block

The meaning is that if the switch expression matches any of the case expressions in the list, the block will be executed.

Relational Operators and Logical Operators

The operators, > and ==, that appear in the if statement above are examples of relational operators. Note that == comprises two equal signs (a notation borrowed from C/C++). It means “is equal to”. It is different from the assignment operator, which uses just one equal sign. The relational operators are given in Table 11 below. Expressions involving relational operators actually have a value. The value is 1 if the relationship is true and 0 of the relationship is false. (The idea of using 1 to stand for truth and 0 to represent falseness came from the C language.) In fact any expression that is equal to 0 will be treated as “false” by the if construct. Furthermore, any expression

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>not equal to</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
that does not equal zero will be treated as true by the `if` construct (i.e., it does not have to be 1). Thus the statement

```plaintext
if -14; x = 5; end;
```

will cause `x` to be assigned the value 5 because the conditional evaluated to non-zero (-14 is non-zero), which means “true”.

Expressions involving relational operators can appear anywhere. So for example,

```plaintext
fprintf('The value of 1 > 0 is %f \n', 1 > 0);
fprintf('The value of 1 < 0 is %f \n', 1 < 0);
```

does these lines to be printed out,

```
The value of 1 > 0 is 1.000000
The value of 1 < 0 is 0.000000
```

Conditionals may also involve the so-called **logical operators**. These operators are given in Table 12

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&amp;&amp;</code></td>
<td>and</td>
</tr>
<tr>
<td>`</td>
<td></td>
</tr>
<tr>
<td><code>~</code></td>
<td>not</td>
</tr>
</tbody>
</table>

The not operator is a unary prefix operator, which means that it takes only one operand and it comes before its operand. If its operand equals zero, which represents falsehood, its output is 1, which, since it is nonzero, represents truth. Thus `~0` equals 1 and `~1` equals 0. The other two logical operators are binary infix operators, which means that, like `+`, `-`, `*`, `/`, and `\`, they each take two operands and they come between their operands. For example, `x && y` and `x || y` are logical expressions involving these two operators. Expressions involving these operators always return 1 for truth and 0 for falseness, exactly as do the relational operators. The outputs of these two logical operators and
those of the related function xor ("exclusive or") are given for all possible inputs in Table 13.

| Inputs   | & & | || xor |
|----------|-----|-----|-------|
| 0        | 0   | 0   | 0     |
| 0        | non-zero | 0 | 1    | 1   |
| non-zero | 0   | 0   | 1     |
| non-zero | non-zero | | 1    | 1   |

Table 13. Logical Operators

Here are some examples of both the relational operators and logical operators in action. The following statements:

```c
fprintf('The value of (1>0) && (2>1) is %d 
', (1>0) && (2>1) );
fprintf('The value of (1>0) && (2<1) is %d 
', (1>0) && (2<1) );
fprintf('The value of (1>0) || (2<1) is %d 
', (1>0) || (2<1) );
fprintf('The value of (1<0) || (2<1) is %d 
', (1<0) || (2<1) );
fprintf('The value of ~(1>0)         is %d 
', ~(1>0)         );
```

cause these lines to be printed out,

The value of (1>0) && (2>1) is 1  
The value of (1>0) && (2<1) is 0  
The value of (1>0) || (2<1) is 1  
The value of (1<0) || (2<1) is 0  
The value of ~(1>0)         is 0  

If one thinks of 1 and 0 as true and false, all these answers make sense. The three logical operators and xor() each treat their arguments as if zero represents falsehood and anything else (i.e., nonzero) represents truth. Thus,

```c
fprintf('The value of ~0            is %d 
', ~0             );
fprintf('The value of ~14           is %d 
', ~14            );
fprintf('The value of 3 && -1       is %d 
', 3 && -1        );
fprintf('The value of sqrt(11) | 0  is %d 
', sqrt(11) || 0  );
```

causes the following lines to be printed,

The value of ~0            is 1  
The value of ~14           is 0  
The value of 3 && -1       is 1  
The value of sqrt(11) | 0  is 1  

5.4 Plotting features

We have plotted more than one function of the same graph, but we have never put more than one graph on the same figure. It is possible to put multiple graphs on the same figure. The Matlab function that provides this feature is called subplot. This feature and several other plotting features are described in Section 3.5 of Chapman. The figure below, for example, was produced using this feature. The first graph on the upper left was produced by first executing the command subplot(2,3,1) and then giving the appropriate plot and title commands. Here “2,3” means “6 graphs, arranged in 2 rows and 3 columns”. The third argument (1) indicates which of these six graphs, or “subplots” will be affected by subsequent plotting command(s) (plot, title, etc.). The graph at the upper right was produced by executing subplot(2,3,3), followed by the command polar(theta,r), where theta and r are vectors of the same length. This command is a variant of plot. It plots one point for each corresponding pair of elements theta(n) and r(n), but it treats them as the polar coordinates r and \theta.

Like plot, polar allows the user to specify the line style in an optional argument: polar(theta,rho,s), where s is a string, such as ‘*’ and ‘k’. The available styles are the same as those for plot and are listed in the help information for plot (help plot). The lower plot was produced by executing the command subplot(2,1,2), which means “2 graphs, arranged in 2 rows and 1 column and the next plotting command pertains to the second one”. Thus, different arrangements can be mixed within the same figure. However, if a later subplot request will cause an overlap with a previous, one, the previous one will be erased.

As we mentioned earlier, the plotting limits for the x and y axes are chosen by Matlab for you, but you can override its choices by using the axis command. This command was used in the lower plot. In that plot, while the x value of the plot itself ranged from 0 to 30, the plotting limits include the range from -5 to 35. That is why there is space to the left and right.

Finally, note that at the upper left of the figure, is the label, Figure No. 2. The number 2 was produced because we used the command figure(2). This command makes it possible to have more than one figure active at a time. After the command figure(n) is given, the next plot commands will pertain to the figure window labeled “Figure No. n”. As soon as figure is called with a different number, the subsequent plotting commands will pertain to a different figure window. Each window will remain open on the desktop. It is possible to return to a previously used window by giving the figure command with that window’s number as the argument.
Concepts from this chapter

- Computer Science and Mathematics:
  - software engineering
  - top-down design
  - black box
  - decomposition
  - stepwise refinement
  - pseudocode
  - keyword
  - control constructs
    - sequencing
    - selection
  - conditionals
  - relational operators
  - logical operators

- Matlab:
  - branching statements
    - if construct
    - switch construct
  - relational operators
  - logical operators
  - plotting:
    - figure
    - subplot
    - polar
Chapter 6

Loops

In the last chapter, we introduced the idea of a control construct, a method by which Matlab will depart from its customary sequential processing. Initially, we accomplished this task through the use selection. That is, we used either an if or a switch statement to choose among a series of alternative commands to be executed. In this chapter, you will be introduced to another control construct: loops.

6.1 Repetition

Imagine that you have just walked up to a large breakfast buffet. As you survey the offerings, you are reminded that you do not like green eggs or ham. Seeking to avoid digestive discomfort, you begin walking down the buffet line checking each dish in turn to ensure it does not include the offending foods. If we were to write this process as a program, you might create a long series of if statements to evaluate each dish for the presence of green eggs or ham. In programming, we often need to do something more than once. Add 10 to a series of numbers. Perform an element-by-element operation. Search through a set of numbers to find its maximum value. Review a data set to ensure no values fall below a specified safety limit. Each of these operations involves performing a single set of actions repetitively, though with small variations. In sequential processing, we would have to write a lengthy program that repeated those actions for each element. It would be much more efficient if we could write the code for those actions once and then simply repeat them. A loop is a control construct that does just that. Matlab supports two types of formal loops: the for loop and the while loop.

6.2 The for Loop

The for loop is designed to execute a series of statements a finite number of times. Its syntax is straightforward:

```
for variable = expression
    statements
end
```

The variable above is known as an index and may be referenced like a variable by your Matlab program. The most basic function of an index is to keep track of how many times your loop has been executed. Traditionally, most programmers use the letter i for the index. In Matlab, however, i is used to represent imaginary numbers. To avoid
confusion, we’ll use ii as our index variable. As a Matlab programmer, you’re free to use any valid Matlab variable name for your index.

The expression is used to determine how many times the loop should be executed. The expression takes the form of a vector. Usually this is created in shorthand form by means of the colon operator.

The statements are the set of commands Matlab is expected to execute each time the loop is repeated.

Finally, every loop is concluded with the word end.

To understand how loops function, consider the following commands:

```matlab
for ii = 1:10
    fprintf('
 Loop Count: %d', ii);
end
```

When Matlab reaches the loop initially, a control structure is created. In this example, that structure is an array from 1 to 10: [1 2 3 4 5 6 7 8 9 10] as specified in the expression. Being the first time through the loop, our index variable, ii is set to the first element in our control structure: 1. Now, our statements in the body of the loop are executed, so the phrase Loop Count: 1 is printed on a new line. Once Matlab reaches the word end, representing the end of the loop, our index variable is assigned to the next number in our control structure, 2. At this point, the statements in the body of the loop are executed, so the phrase Loop Count: 2 is printed on a new line. When we reach the word end, our index variable is assigned the next number in our control structure, 3. This process will repeat until we reach the end of our control structure. When our index variable is equal to 10 and we reach the end statement, the loop will terminate and our program will continue on to whatever code might exist after the loop. It’s worth noting that our index variable, ii, will continue to exist and will keep a value of 10 unless modified later.

### 6.2.1 Modifying the Control Expression

For loops are extremely easy to write because Matlab does most of the hard work for you: it keeps track of how many times you’ve executed the loop. For loops do have one limitation, however: a for loop will only execute a finite number of times. The required values of your index variable must be known before the loop begins. This means that creating the control expression is often the most difficult or most important part of the loop.

Consider our previous example of counting from 1 to 10. Suppose instead that we wanted to count from 1 to 100. The obvious solution is to modify our control expression to read:

```matlab
for ii = 1:100
```
Now suppose that we only want to count the even numbers. We certainly know enough about conditionals to write an `if` statement to determine whether or not a particular index value is even or odd. But we can greatly simplify this process by modifying the control expression instead to only generate even numbers in our control array. The colon operator makes this easy:

```matlab
for ii = 2:2:100
```

The colon operator’s ability to increment values allows us to create control arrays that feature fractions, even or odd numbers, multiples or even arrays that decrease in number.

Sometimes, however a standard increment value can be impossible to define. In that case, Matlab allows you to define your control array directly.

```
Example 6.1 – Filtering Data

Suppose you have been given a set of data in an array, x.

```matlab
x = [-2  4  6  -7  3  5  -45  1  -0.5  2.6]
```

You have been asked to write a program that will print the positive numbers in x. A solution is to write a simple for loop. Examining array x, however, we find it impossible to describe its values using the colon operator, so we will manually define our control array instead.

```
for ii = [-2  4  6  -7  3  5  -45  1  -0.5  2.6]
  if ii > 0
    fprintf('%f \n', ii);
  end
end
```

In the solution above, each time Matlab completes an iteration of the loop, the index value will move to the next element in the array. So the first time through the array, `ii=-2`, the second time through the loop, `ii=4`. The third time through the loop, `ii=6` and so forth.

Performing iteration on an array (that is to say, repeating a task on a series of values) is one of the most common applications of loops in Matlab. Because we can access individual elements in a matrix numerically (for example, `A(3,2)` or `x(4)`, etc.), a well-defined control expression allows us to use our index to cycle through a Matlab array of values. This is an important concept. Recall that because for loops are only executed a finite number of times, the control array (the values of our index variable) must be known beforehand. But that only means the values must be known at the time the loop is first executed, not necessarily at the time the program is written!
Chapter 6  
Loops

To this point, we have used our control expressions to generate a one dimensional array of values for our index variable to cycle through. An important consideration, however, is that in Matlab, our index variable can be a vector instead. Consider the following control expression:

```matlab
for ii = [4 7 9; 2 3 4; 1 7 8]
```

In this example, the loop will execute three times, once for each column. Each time through the loop, our index variable, ii, will be set equal to one of the column vectors. So the first time through the loop, \( ii = \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix} \), the second time through the loop, \( ii = \begin{bmatrix} 7 \\ 3 \\ 7 \end{bmatrix} \), and the last time through the loop, \( ii = \begin{bmatrix} 9 \\ 4 \\ 8 \end{bmatrix} \).

6.3 The while Loop

A while loop is designed to repeat as long as a condition is met. Unlike a for loop which is always finite, a while loop is indefinite and may be infinite. Its syntax may look similar to a for loop, but there are a number of important differences.

```matlab
while  expression
  statements
end
```
Unlike a for loop, the expression in a while statement consists of a logical or relational test. As long as that expression evaluates to true (non-zero), the loop will repeat. When the expression evaluates to false (zero), the loop will terminate.

The statements are the series of commands to be executed each time through the loop. Again, the loop construct is concluded with the end statement.

Perhaps the most obvious difference between for and while loops is that the while loop lacks an index variable. Indeed there is no built-in index functionality in a while loop. You can achieve the same functionality, but you’ll have to define and increment an index yourself. In fact, there is no problem you can solve with a for loop that can’t also be solved using a while loop. However, you’ll find that it’s often more complicated to do so in a while loop.

### 6.3.1 Working With while Loop Control Expressions

The control expression in a while loop is the most important and usually one of the most challenging aspects to write. More programming errors occur because of the control expression than in any other part of a while loop. Consider the following example:

```matlab
while 1
    fprintf('Hello World! \n');
end
```

The control expression is simply “1”. In a while loop, as long as the control expression evaluates to true (in Matlab, any non-zero value) the loop will continue to run. So in this example, the phrase “Hello World” will be printed on the screen forever. This is an example of an “infinite loop.” In programming, infinite loops are generally considered to be something to avoid. Note: if you find yourself caught in an infinite loop, try hitting cntl-c (the control and c keys pressed simultaneously).

The above example was fairly obvious. However, infinite loops are common problem. One of the most frequent causes of an infinite loop is the failure to update a counter. For example, suppose we wanted to sum the first ten numbers and we wrote the following code:

```matlab
n = 1;
total = 0;
while n <= 10
    total = total + n;
end
```

This potential solution causes an infinite loop! Our programmer-created index, \texttt{n}, begins with the value of 1. When we reach the while loop for the first time it asks whether \texttt{n} is less than or equal to 10. Since 1 meets this criteria, the expression evaluates to true and
the statements within the loop are executed. The value of `total` then is set to 1. We reach the `end` statement, and control returns to the top of the while loop. Again, Matlab evaluates whether `n` is less than or equal to 10. The value of `n`, however, hasn’t changed. 1 is still less than or equal to 10, so the loop will execute again and again and again. Indeed, the loop will never end because the value of `n` will always be 1 and will always be less than or equal to 10. As for the solution to our original problem, our grand total will well exceed its task of finding the sum of the first 10 numbers as it heads toward infinity.

Updating our created index will solve this problem.

```matlab
n = 1;
total = 0;
while n <= 10
    total = total + n;
    n = n + 1;
end
```

Now, each time through the loop, `n` will be incremented. The last time through the loop, `n` will have a value of 10. The variable `total` will be recalculated, then `n` will be incremented and set to 11. The `end` statement is reached and control returns to the top of the while loop. The expression is evaluated. This time, when Matlab asks whether `n` is less than or equal to 10, the expression evaluates to `false`. The loop terminates and control is shifted to the next line in the program immediately after the end of the while loop.

You may have noticed that the logical expression in the example above evaluated to true as long as `n` was less than or equal to 10. Another common error in writing control expressions is to terminate the loop too quickly. Consider what would have happened if the control expression had been written as:

```matlab
n = 1;
total = 0;
while n < 10
    total = total + n;
    n = n + 1;
end
```

The last time through the loop would be when `n = 9`. At that time, the variable `total` is adjusted, `n` is incremented and set equal to 10. When control returns to the top of the while statement, the expression evaluates to `false` because (with a value of 10) is not less than 10. The error here is one of logic rather than syntax. The loop will run and terminate normally, but does it solve the original problem which was to sum the first ten numbers? It does not. In many ways this is a more insidious type of error than an infinite loop. When an infinite loop occurs, you know there is a problem and roughly where you need to look to fix it. In this case, the program will run normally and you might never detect that there is an error unless you are checking the output very carefully and recognize that the numeric solution Matlab returns is incorrect.
Early termination is one of a set of related problems in building control expressions in loops. More generally, we can think of these problems as dealing with the endpoints of our data. Consider our earlier problem where we were given an array of numbers and asked to print out the positive values in that array. We solved that problem using a for loop. But any problem that can be written as a for loop can also be solved using a while loop.

Example 6.3 – Filtering Data With a While Loop

Suppose the user has been asked to enter an array of data values, x, and you have been asked to write a program that will retrieve those values and print out only the positive numbers.

```matlab
x = input('Please define an array: ');  
n = 1;  
while n <= length(x)  
    if x(n) > 0  
        fprintf('%f \n', x(n));  
    end  
    n = n + 1;  
end
```

This problem can be solved using a while loop, though it’s slightly more complicated because we’re forced to use an array referencing structure rather than simply using the array itself as we could in a for loop. This means we have to define and increment our own index.

Again, to avoid early termination, we set our control structure to the format:

```matlab
while n <= length(x)
```

However, there are several other alternatives that will produce the same result. For example:

```matlab
while n < length(x)+1
```

Or, with slight modifications to our program, other possibilities emerge:

```matlab
n = 0;  
while n <= length(x) - 1  
    n = n + 1;  
    if x(n) > 0  
        fprintf('%f \n', x(n));  
    end  
end
```

```matlab
n = 0;  
while n < length(x)  
    n = n + 1;  
    if x(n) > 0  
        fprintf('%f \n', x(n));  
    end  
end```
The point is that you must be aware of how your loop will treat the beginning and end of your loop. There are several potential methods for addressing these endpoints. The method of choice is up to you.

6.4 Programming With Loops

Although the basic syntax and functionality of for and while loops are easy to understand, many students find their application to be much more challenging. There are a number of considerations for the programmer when developing loops that need to be mastered.

6.4.1 Good Programming Practices

In all the previous examples, the loops were written across multiple lines and with indented spaces. This isn’t required for Matlab, of course, as long as each command is separated by a semicolon. For humans however, this type of spacing makes your code much more readable and easier to debug.

When writing loops, it is considered good programming practice to *always indent the statements in the body of your loop by 2 – 5 spaces*. Additionally, you should consider placing each command on its own line as this makes it easier to read overall and usually much easier to debug errors.

Another consideration involves modifying the index variable. A while loop, of course, requires you to increment or make any other changes to an index. A for loop, however, handles that automatically. It is considered good programming practice to *avoid modifying the value of a loop index variable inside the body of a for loop*. Consider the following code:

```matlab
total = 0;
for ii = 1:10
    ii = ii * 2;
    total = total + ii;
end
```

The first time through the loop, \( ii \) is initially set equal to 1. Then in the first statement in the body of the loop, \( ii \) is set equal to 2 \((1 \times 2)\). At the end of the loop, what should happen? Will \( ii \) now be equal to 2 or 3? In Matlab, \( ii \) will automatically be set equal to the second element in the control array, in this case 2. But you can see why this might be confusing. During debugging, keeping track of \( ii \) and how many times you’ve executed a loop becomes extremely difficult if you modify the index yourself. Syntactically, there
are no errors in this example. Logically, however, you’re creating unnecessary headaches for yourself. A far better solution would be:

```matlab
total = 0;
for ii = 1:10
    total = total + 2*ii
end
```

Finally, it’s worth noting that in many other programming languages, modifying the index in a for loop will cause that loop to skip ahead (or backward) as though it had been executed the exact number of times as indicated by the value of your index variable.

### 6.4.2 Choosing Between for and while Loops

For most problems you will have a choice between using a for or a while loop. The primary differences may be summarized as:

<table>
<thead>
<tr>
<th></th>
<th>For Loops</th>
<th>While Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite</td>
<td>(will run a specified number of times, as determined when the loop is first executed)</td>
<td>Indefinite (will run until a given logical or relational condition evaluates to false or 0)</td>
</tr>
<tr>
<td>Built-In Index</td>
<td>(Matlab will define and increment an index automatically)</td>
<td>User-Defined Index (You have to define an index before the loop and increment the index inside the body of the loop)</td>
</tr>
</tbody>
</table>

In practical terms, you’ll likely find that for applications involving an array of numbers to be manipulated, a for loop is far easier to program. The while loop is most often used when you simply don’t have the option of using a for loop: when there’s no way to determine how many times a loop should run.

The classic example of this situation involves data validation. Suppose we ask a user to enter a number between 1 and 10. Invalid entries, such as numbers outside the range or text should be rejected. A simple conditional statement can be used to implement this process. But suppose we want the program to keep prompting the user for an entry until a valid response is given? How many times will that loop run? There’s no way to tell for certain. The simple answer is that it will run until the user gives a valid response. A while loop is the perfect solution to this problem.

```matlab
valid_response = 1;
while valid_response
    choice = input('Please enter a number between 1 and 10');
    if 1 <= choice && choice <= 10
        valid_response = 0;
    end
end
```
Chapter 6  Loops

This loop will run continually until the value of choice falls between 1 and 10, at which point the value of valid_response will be set to 0, the logic of the while loop expression will evaluate to false and the loop ends.

You may have noticed what seems to be a logical discrepancy in the example above. The variable valid_response maintains the value true until what is actually a valid choice by the user occurs, at which time valid_response is set to false. Doesn’t that seem backward? Remember that the test occurs in the expression of the while loop. As long as the expression evaluates to true, the loop will run. In order to avoid this apparent inconsistency, however, you’ll often see the following modification to the error validation code:

```matlab
valid_response = 0;
while ~valid_response
    choice = input('Please enter a number between 1 and 10');
    if 1 <= choice && choice <= 10
        valid_response = 1;
    end
end
```

Placing the ~ (not) in the expression means that as long as valid_response is not true, the loop will repeat.

### 6.4.3 Nested Loops

A control construct may be placed inside the body (often referred to as the block) of any other construct. This is called nesting. The ability to nest loops inside one another allows for the creation of particularly powerful applications. Let’s consider a simple example of a multiplication table. Suppose we wanted to print a 12x12 table of multiplied values. Nested loops allow us to solve this problem with minimal programming:

```matlab
for j = 1:12
    for k = 1:12
        fprintf('%d  ', j*k);
    end
    fprintf('
');
end
```

When the first loop (with index j) occurs, j is assigned the value 1. The first statement in the that loop (called the outer loop) is a nested for loop (with index k). This new loop is called the inner loop. The initial value of k is also 1. At this point, Matlab will begin to execute the inner loop block, meaning the statements within the loop with index k. The first end statement signifies the conclusion of the inner loop. At this point, Matlab will return control to the top of the inner loop. The index k will now be assigned the value 2. However, the index j will still have the value 1. This will continue until k is equal to 12. After the inner loop ends, a new line is created and the conclusion of the first iteration of the outer loop occurs. Now, j is assigned the value 2. The statements
for the outer loop will be executed a second time, which means a brand new inner loop with index $k$ begins. Because this is a new loop, $k$ will be assigned the value 1 and will run through a complete cycle up to 12 again before $j$ is incremented a second time.

A few programming tips to remember:

- An end statement is always associated with the innermost currently open control construct, no matter whether that is a loop or a conditional. The closest parallel to this idea is parentheses. You always have the same number of open and closed parentheses and they are grouped such that the closed parentheses associates with the last open parentheses that hasn’t already been closed. The same logic applies to control constructs and their end statements.

- Nested loops should always have independent index variables.

### 6.4.4 The **break** statement

Previously, we were concerned with writing control expressions that avoided early termination of our loops. Sometimes, however, we want a loop to end early. This process is known as “breaking out” of a loop. In Matlab, breaking out of a loop is accomplished with the break statement.

The break statement causes the execution of a loop to terminate and control is passed to the first executable statement after the loop.

Suppose we have a large data set, but to be valid, all the values in that set must be positive. One possible solution would be:

```matlab
x = load('sample_data.dat');
for ii = 1:length(x)
    if x(ii) <= 0
        fprintf('The data contains invalid values.')
        break;
    end
end
```

This solution will examine every data point in $x$. If a negative value is found, the error message is printed and the loop terminates. No further action is taken by the loop, meaning no other elements in $x$ are evaluated.

Note that if a break statement is executed inside a set of nested loops, then the break statement applies only to the innermost loop that contains it. Any outside loops would continue to run normally.
Chapter 6  Loops

6.4.5 Preallocation

Now suppose instead of simply providing the value of N! as output, we wish to create an array of factorial values from 1 to N. With a slight modification to our program above, we can create this array:

```
n = input('To calculate N!, what is N?');
factorial(1) = 1;
for ii = 2:n
    factorial(ii) = factorial(ii-1)*ii;
end
fprintf('%d', factorial);
```

The above example certainly seems to work. But what happens if you try to calculate the factorial for 30,000? You find that the answer is infinity, or at least large enough that the student edition of Matlab won’t calculate it. You may also notice that it takes quite a few seconds for Matlab to reach this calculation. Without clearing your variables, try running the program again, this time for 29,999. The value is still infinity, but did you notice that the answer popped up almost immediately? Could one fewer calculation make that significant a difference in processing time? No, but what’s happening behind the scenes accounts for this unusual behavior.

Every iteration through the loop, the variable array `factorial` is growing larger in size. Each iteration, your computer is going to memory and allocating additional RAM storage space for this variable. That takes time. In fact, it takes more time for your computer to interact with RAM to gain additional space than it does to actually make the calculation. With small arrays, this process is virtually invisible. With larger arrays (greater than 2000, for example) you may find a delay in Matlab’s response.

Example 6.4 – Factorial Calculations

A factorial is defined for a number, N, such that \( N! = 1 \times 2 \times 3 \times \ldots \times N-2 \times N-1 \times N \)

For example, \( 10! = 1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 \times 8 \times 9 \times 10 = 3,628,800 \)

We can use a `for` loop to write a program that calculates N!, where N is provided by the user.

```
n = input('To calculate N!, what is N?');
factorial = 1;
for ii = 2:n
    factorial = factorial * ii
end
fprintf('%d', factorial);
```
So why did this problem disappear the second time? After the first run through, the variable `factorial` was already defined in memory as a 1x30000 array. Your computer didn’t have to keep asking RAM for more space, thus the speed was significantly increased. You can test this yourself by clearing your variables and running the program again. This time if you only ask for a factorial of 25,000, you’ll still notice a time delay.

This process of claiming the variable space beforehand is called preallocation. It is considered good programming practice to always preallocate your array variables in Matlab before beginning a loop. This is particularly important for large arrays. We commonly use the `zeros()` function in Matlab to preallocate our arrays.

The example below uses a preallocated array. Try it in Matlab and notice the significant improvement in speed with really large factorials, such as 100,000.

```matlab
n = input('To calculate N!, what is N?');
factorial = zeros(1,n)
factorial(1) = 1;
for ii = 2:n
    factorial(ii) = factorial(ii-1)*ii;
end
```
Logical Arrays and Logical Indexing

While looping provides a very versatile method for performing operations on all of an array or on selected elements, these operations can often be implemented much more simply by means of logical indexing. Logical indexing involves the use of a special type of array called a logical array. We introduce both concepts in this section.

Let \( a = [12, 3, 45] \) and let \( b = [1, 0, 1] \). The command \( a(b) \) causes an Error. But if we make \( b \) a logical vector as follows: \( b = \text{logical}([1, 0, 1]) \), then we get this:

\[
\begin{bmatrix}
12 & 45
\end{bmatrix}
\]

Matlab has returned a vector, \([a(1), a(3)]\), made up of the elements of \( a \) that are in the positions of the nonzero elements of \( b \). The length of \( b \) must be the same as the length of \( a \), but note that the length of the result may be shorter than the length of \( a \). This feature is called logical indexing. It works only if \( b \) is a logical vector.

There are other ways to create logical arrays. The most important method is the use of the relational operators, \(<\), \(>\), \(==\), \(<=\), \(>=\), and the “element-wise” logical operators, \&\& and \~\~. For example:

\[
c = [2>1, 2<1, -(3>2 & 4>5)]
\]

makes \( c \) a logical vector with the value \([1, 0, 1] \). This happens because the relational and logical operators return values whose type is “logical”. (To determine a variable’s type use the function named \text{class}, as, for example, \text{class}(c).) The operators \& and \&\& each signify the “and” operation. There are, however, two differences: (1) \& is the array operator, meaning that its operands, unlike those of \&\& are not required to be scalars. The two operands are required only to have the same dimensions. The operator acts on each corresponding pair of elements in the two operands and returns an array of the same dimensions. For example, \([1 \ 0 \ -3] \& [0 \ 0 \ 1] \) returns \([0 \ 0 \ 1] \). (1) The operator \&, unlike \&\&, does not short-circuit. The operators \| and || have a similar relationship.

A common use of logical indexing is illustrated by this example. We have the vector \( a \) above whose elements are 12, 3, and 45. We wish to form a new vector consisting of those elements of \( a \) that are greater than 10. We do it as follows:

\[
\begin{bmatrix}
12 & 45
\end{bmatrix}
\]

What has happened is that a new vector is formed by the operation \( a>10 \). That vector looks like this \([1, 0, 1] \) and it is a logical vector. That vector then provides logical indexing into \( a \), and the result is that a new vector is formed (as in the example \( a(b) \) above) consisting of the two elements of \( a \) that appear at the indices where the values of
the logical vector are nonzero (positions 1 and 3). The result, \( c \) is not a logical vector. It is of the same type as \( a \), which in this case is **double**.

It can be seen from this example how logical indexing can simplify the code. Here is a loop version of \( c = a(a>10) \):

```matlab
jj = 0;
for ii = 1:length(x)
    if x(ii) > 10
        jj = jj + 1;
        y(jj) = x(ii);
    end
end
```

Logical indexing can be used on the left side of the equal sign as well. We will introduce left-side logical indexing with a simple example. Once again, we let \( a = [12, 3, 45] \). Now we execute the following command:

```matlab
>> a(a>10) = 99
a =
    99     3     99
```

The meaning here is that all elements of \( a \) that are greater than 10 are set to 99. Once again logical indexing is much simpler in comparison to explicit looping, as can be seen by comparing \( a(a>10) = 99 \) with the following semantically equivalent loop:

```matlab
for ii = 1:length(a)
    if a(ii) > 10
        a(ii) = 99;
    end
end
```

In each of the last two examples—one with logical indexing on the right and one with logical indexing on the left of an assignment statement, this explicit loop reveals something that is hidden by the far more succinct form of logical indexing: namely, that each element of the vector must be accessed and operated on. With the syntax of logical indexing, it appears that an entire vector is being operated on at once. That is not true. It is not even possible with conventional computers, such as the ones that Matlab typically runs on. Matlab commands may appear to operate on entire vectors, but in fact they cause hidden loops to be activated by the Matlab interpreter behind the scenes.

### 6.5.1 Logical indexing with arrays

So far, we have used logical indexing only with vectors. Here is an example of logical indexing with an array. Our task is to finds the elements in the matrix \( M \) that are greater than 0.1 and replace each one with its square root. First we give a version employing explicit looping:
[m,n] = size(M);
for ii = 1:m
  for jj = 1:n
    if M(ii,jj) > 0.1
      M(ii,jj) = sqrt(M(ii,jj));
    end
  end
end

Here is a version using logical indexing:

M(M > 0.1) = sqrt(M(M > 0.1));

This example shows both that logical indexing works with arrays and that logical indexing can be used on both sides of the equal sign in the same command.

At this point it may seem that the difference between vectors an arrays is minor with logical indexing. It is not. The situation with arrays is more complex, as can be seen with the following example:

>> A = [1 2 3; 4 5 6]
A =
    1  2  3
    4  5  6
>> B = A(A>2)
B =
    4
    5
    3
    6

As expected, only those elements of A that are greater than 2 made it into B, but surprisingly, we find that B is a column vector! What is the reason for this change of shape, and how did Matlab choose the order of the elements in B? The answers have to do with the requirement that arrays must be rectangular. In other words each row must have the same length. If that were not required, then we might well have seen something like this:

>> B = A(A>2)
B =
    4  5  6  3

In this fictitious output, the first row of B has only one element and it is “sitting” at the far right (whatever that means!), and the second row of B has three elements. This
scenario cannot work because later operations with \( B \) would be poorly defined, so Matlab does not behave this way. Instead, when logical indexing is applied to an array on the right side of the equal sign, if the array is not a vector, then that array is replaced by a column vector in which the elements are placed in column-major order. Column major order means that, if the successive indices are written out, then the column index changes fastest. For an \( M \)-by-\( N \) matrix \( X \), the order would be as follows:

\[
(1,1), (2,1), (3,1), \ldots, (M,1), (1,2), (2,2), (3,2), \ldots, (M,2), (1,3), (2,3), (3,3), \ldots, (M,3), \ldots, (1,N), (2,N), (3,N), \ldots, (M,N),
\]
as is illustrated below:

\[
\begin{array}{cccccc}
1,1 & 1,2 & 1,3 & \ldots & 1,N \\
2,1 & 2,2 & 2,3 & \ldots & 2,N \\
3,1 & 3,2 & 3,3 & \ldots & 3,N \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
M,1 & M,2 & M,3 & \ldots & M,N
\end{array}
\]

The command \( x = X(:,) \) puts the elements of to \( x \) as a column vector in column-major order. Here is what the result looks like for \( A \):

\[
\begin{bmatrix}
1 \\
4 \\
2 \\
5 \\
3 \\
6
\end{bmatrix}
\]

In the command, \( B = A(A>2) \), it is this column vector whose elements are selected for \( B \). Only those elements that are greater than 2 are selected. Thus 1 is omitted, 4 is selected, 2 is omitted, and 5, 3, and 6 are selected. The result is, as we saw above, is

\[
\begin{bmatrix}
4 \\
5
\end{bmatrix}
\]
6.5.2 Implicit Looping and Vectorization

Logical indexing is the latest of many examples of implicit looping. We learned in Chapter 3.4 that many numerical functions, sqrt, for example, when given an array as input will perform a calculation on each element of the array and return an array of results. While it may not have been clear to you then, it is probably obvious now that a hidden loop must be involved behind the scenes when an operation, such as taking the square root is repeated many times. This is in fact exactly what happens when sqrt, sin, zeros, or many other functions return an array of results. It also happens with the colon operator with array operations and with matrix multiplication, as in the following examples:

- x = 1:1000: loops through the numbers 1, 2, ..., 1000 and sets x(1) = 1, x(2) = 2, ..., x(1000) = 1000
- A.*B loops through all the elements of A and B, and returns an array of the products of their elements (if A and B are not vectors, there will be nested loops)
- A*B loops through the rows of A and the columns of B, forming an array of dot products (unless A and B are scalars, there will be nested loops)

Commands that operate on entire vectors or entire arrays are said to be “vector commands” (the more general term “array” commands is rarely used), and the translation of Matlab code from a version that uses a loop into one that uses a vector command is called vectorization. In every case of vectorization, an explicit loop is avoided. Looping is however implicit, because the programmer knows that (for the typical computer), only one element of the vector can be accessed by the CPU at a time. There are many instances in which vectorization results in a significant simplification in the code required to accomplish a task. Furthermore, the vectorized version typically runs faster than the loop version, because the vectorized version runs more efficiently. Because of the simplification in code and the improved execution efficiency, implicit looping represents a major advantage over most other languages provided by Matlab in numerical applications, and vectorization is an important way to exploit this advantage. Skill at writing vectorized code is essential for large numerical applications in Matlab, and that skill comes both with practice and by observation of experienced Matlab programmers.
Concepts from this chapter

- Computer Science and Mathematics:
  - loops
  - nested loops
  - preallocation of arrays
  - column-major order
  - implicit looping
  - vectorization

- Matlab:
  - `for` loop
  - `while` loop
  - `break` statement
  - logical arrays
    - logical
  - logical indexing
  - `A(:)`
Chapter 7

User Defined Functions

To this point, the programs you have written have relied almost exclusively on sequential control. And while you have used control constructs such as loops and conditional statements to alter that sequential processing, there has been one constant in your programming: you have written single m-files to accomplish your task. For small programs, this is not an issue. But imagine you’re writing a program that contains thousands of lines of code or that you’re working on this program with several people. In this scenario, several problems arise. You find that you can’t work on the same m-file simultaneously. Your group members tend to use the same common variable names, so important data is overwritten and lost in the program. Common tasks are programmed repeatedly. In short, you find that using a single m-file creates an inefficient and confusing jumble of code.

Thankfully, there is a more efficient way to organize your programming. That method is to use functions. From the moment you started working with Matlab, you have been using functions. Matlab has hundreds of pre-programmed functions to perform common tasks. This chapter will introduce you to a number of concepts geared toward the creation of your own unique functions.

7.1 Introduction to User Defined Functions

A function is nothing but a small program that performs a specific task. Recall that in top-down design, algorithm creation is a two-step process. First, you decompose your task into a series of sub-problems and then you perform stepwise refinement to break those sub-problems down even further. Software engineers will typically create a function for every sub-problem in their algorithm. This organization provides a number of advantages:

- **Independence in Writing and Testing**
  Each function acts as a stand-alone program, so when your algorithm is divided into a series of sub-problems or sub-tasks, each programmer could work on an individual task, programming and debugging their functions simultaneously. The separate functions are brought together at the end and used as a seamless program.

  Perhaps more important than the time savings offered by the division of labor is the independence of variables. An important side effect of using functions is that because each function acts as a stand-alone program, variables are kept separate and independent. This means that identical variable names can be used in more
than one function, but their values won’t be confused or overwritten. This is an extremely important computer science concept known as variable scope, and will be discussed at length later in the chapter.

- **Ability to Reuse Code**
  It is often the case in programming that a single task is needed over and over again in a program. A loop can be used to repeat a task, but that loop might only occur once in a program. By writing that task as a function, you can execute that task at any time in your program. This saves an enormous amount of time because your task only needs to be programmed and debugged once. Perhaps just as significantly, your function can be used in every program you write thereafter, not just in the program for which it was originally created!

### 7.2 Function Syntax

Thanks to Matlab’s pre-defined functions, you are already very accomplished at using functions. When we command Matlab to execute a function, we say that we are calling a function. As you know, some functions require you to provide data or variables to perform its task. Similarly, some functions will return one or more variables as output. These variables, whether input or output, are called arguments when associated with a function. We say that we are passing an argument when they are being sent to or from a function. It should be noted that the words argument, variable and parameter are often used interchangeably when we speak of passing arguments to and from functions. Technically, there are distinctions between the three, but those distinctions are best left for computer science theorists to debate.

The basic syntax for both calling and defining (writing) a function varies depending on how many arguments are used for input and output.

#### 7.2.1 Calling Functions

The default syntax for calling a function is as follows:

```
[output_argument1, output_argument2, ...] = 
    function_name(input_argument1, input_argument2, ...)
```

Multiple input and output arguments are placed in lists, with the output arguments listed inside square brackets and set equal to the function name with the input argument list passed inside parentheses.

Variations in this syntax abound depending on the combination of input and output arguments required:

<table>
<thead>
<tr>
<th>function_name</th>
<th>No input our output arguments</th>
</tr>
</thead>
</table>
Because variables are treated independently by functions, there is often a great deal of confusion between the arguments we use to call a function and those that exist inside the function. To help keep them separate in our minds, it’s helpful to describe the arguments (or variables or parameters) we use in our function call as actual arguments. We call them this because these are the actual variables used in our main program as opposed to the values which are passed to the function.

### 7.2.2 Defining Functions

Because functions act like independent programs, they are written in their own m-files. Creating a function is actually quite simple, but there are three important rules that must be followed:

1. Your function declaration must appear in the first line.
2. All possible input and output arguments must be defined in the function declaration.
3. The name of your function should match the name of the m-file you save.

The generalized form of a function declaration is as follows:

```matlab
function [output_argument1, output_argument2, ...] = function_name(input_argument1, input_argument2, ...)
```

You may have noticed that the only difference in syntax between the generalized form of a function declaration and a function call is the word “function” which appears as the first word in the declaration. Just like a function call, the syntax of a function declaration will vary slightly depending on the input and output arguments required.

<table>
<thead>
<tr>
<th>function</th>
<th>function name</th>
<th>No input or output arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>function name(input arguments)</td>
<td>Input arguments only</td>
</tr>
<tr>
<td>function</td>
<td>output arguments = function name</td>
<td>Output argument only (Square brackets would be used if multiple output arguments were required)</td>
</tr>
<tr>
<td>function</td>
<td>[output arguments] = function name(input arguments)</td>
<td>Input and output arguments</td>
</tr>
</tbody>
</table>
The arguments we define inside a function are called **dummy arguments**. You can think of these as temporary variables for the values of the actual arguments passed to the function.

At this point, it’s instructive to see a function in action. Recall from Chapter 6 the multiplication table problem. We created a simple program that printed a 12x12 multiplication table.

```matlab
for j = 1:12
    for k = 1:12
        fprintf('%d  ', j*k);
    end
    fprintf('
');
end
```

Let’s modify this program to act as a function. Instead of always printing a 12x12 table, we’ll allow the user to determine our table size. Recall our rules for writing functions. The first rule says our function declaration must appear in the first line. The second rule says that all possible input and output arguments must appear in that declaration. This function will require two input arguments, M and N, representing the MxN table that our user wishes our function to calculate. Will there be any output arguments? You might think so because the function will print out a multiplication table for the user, but this is output that appears on the screen rather than output that will be returned as an output variable, so there are no output arguments in this function.

Our new function might appear as follows:

```matlab
function multiply(M, N)
%MULTIPLY  Produces an MxN multiplication table

for j = 1:M
    for k = 1:N
        fprintf('%d ', j*k);
    end
    fprintf('
');
end
```

To use our function, we must apply the third rule of writing functions. That is, when we save this m-file, we must save it with the same name as our function. So this m-file should be saved with the name “multiply.m”.

The first comment line beneath the function declaration serves a special purpose. This line is searched and displayed by the lookfor command in Matlab. Therefore it should always contain a one-line summary of the function. All comments placed immediately beneath the function declaration will appear in the help command and help window up until the first blank line or non-comment line.

A couple quick notes about saving functions:
Chapter 7 User Defined Functions

- The location where you save your function is actually rather important. To be useable, the m-file containing your function must be saved either in the current directory (presumably where your main program or script file is saved) or somewhere in the path. If the function you write is intended for use with a single program, you will typically save it in the same directory as that program. However, if the function is intended to be used over and over again by many different programs, it is not uncommon to create a separate directory for those commonly used functions and then place that directory in Matlab’s path.

- The rule requiring your function name to be identical to the name of the m-file is actually more of a guideline than a hard and fast rule. But it’s a guideline meant to help your organization. When a function is called, Matlab will actually search the current directory (and then later the path) for an m-file with the same name as the function you call. Matlab will execute the code in that m-file regardless of the function name in the declaration.

Example 7.1 -- Finding Prime Numbers

Write a function that will return a 1xN vector containing all the prime numbers between 1 and N.

```
function list_of_primes = myprimes(N)
% MYPRIMES Returns a 1xN vector of all the primes from 1 to N
% Method: Sieve of Eratosthenes (ca 240 BC)

% Variables:
% prime - 1xN, prime(n) = 1 if n is prime, 0 if n is not prime.

% (1) Assume that all the integers from 2 to N are prime.
prime = ones(1,N);
prime(1) = 0;

% (2) Mark all the numbers that are multiples of a number:
for n = 2:sqrt(N)  % No factor can be larger than sqrt(N)
    if prime(n)  % Multiples of primes include all multiples
        prime(n+n:n:N) = 0;
    end
end

% (3) Put the survivors on a list:
prime = logical(prime);
temp = 1:N;
list_of_primes = temp(prime);
```

EDU>> myprimes(25)
ans =
     2     3     5     7    11    13    17    19    23
7.2.3 Function Execution and Control

Using functions means that control will shift from one program to another. This can make run-time debugging more difficult. Testing each function independently of a main program or script file is important to avoid conflicts later. The following example illustrates how control shifts between a script and various functions.

At each stage, sequential processing operates. When a function is called, control shifts to the new function along with the value of the actual argument. The function is then executed as though it were a program all by itself, meaning sequential processing is again the norm. If the function should call another function, control would shift yet again.

At the conclusion of a function, control returns to wherever the function was called. This might be the main program or script file, another function or the command window. If an output argument was defined in the function, its value will be returned as well.
A function terminates when it reaches the end of its code or encounters a `return` statement. Recall that `return` will end a program. In the case of a function, `return` acts to end that function and control is reassigned to the location where the function was called. It is not necessary to end your functions in Matlab with the `return` statement. However, many other programming languages require such a concluding statement to end their functions.

### 7.3 Variable Scope

Previously we described one of the primary benefits of using functions to be their independence. Acting as individual programs, functions have the unique ability to reuse variable names because they’re treated as separate entities. This is a property of variable scope. The `scope` of a variable describes where that variable exists in a computer’s memory and what parts of a program are allowed to access it.

When you use variables, it’s sometimes helpful to think of them merely as reference markers. A piece of data (whether it’s an array, a scalar, or a text string) is stored in your computer’s memory. A variable is a handy way for you to tell the computer that you wish to access the data located in a specific spot in memory. Thankfully, Matlab and your computer’s operating system handle the details of controlling which variable name is associated with which physical location in memory. But the idea that a variable name is merely a marker or a signpost to a specific memory location is important in understanding how variable scope works.

Your computer’s memory is an actual, physical location in the real world. It exists as a series of microscopic transistors on a silicon RAM chip. Computers today typically have millions, if not billions of these transistors. The actual workings of how and where data is stored on these transistors is not important for this book, though high-end computer programmers are concerned with such details! For now, you can think of your computer’s memory as a large space with many locations where data could be stored. (Those locations are common to the whole computer, meaning Matlab has to share that space with every other program you’re running.) Each one of those locations has a unique address. We can visualize this as a really large matrix where each cell represents a different memory location. Small data values might only take up 1 cell. Large arrays might use multiple cells. A variable name is a shorthand notation for the specific memory locations where your data is stored.

#### 7.3.1 Local Variables

By default, variables are considered to have local scope. Local scope means that the variable only exists for that function (or program) and no other function (or program) knows it’s there. In practical terms, this means two things:

1. Arguments and variables are only viewable, readable and able to be changed by the program or function that created them. To everything else, it’s as
though those variables do not exist.

2. Arguments and variables created by a function are deleted from memory when a function ends.

This is how functions are able to have duplicate variable names. Because the scope of the arguments and variables in each function are local, you can create variables with previously used names and Matlab is none the wiser. Consider the following example:

```
% Main Program
A = 13
B = my_function(A)

function [B] = my_function(A)
B = sqrt(A);
```

In this example, the main program has two actual arguments, A and B, each of which refer to their own locations in memory (represented in blue). The value stored in A is passed to the function “my_function” when it’s called. The function creates two dummy arguments, A and B (represented in yellow). Although they have the same name as the actual arguments, you can see that they point to completely different memory locations. In this way, they’re treated as totally separate variables and may contain different values. When the function ends, the value stored in dummy argument B’s memory location is passed back to the main program, and that value is then assigned to the memory location B (in blue). When the function ends, the dummy arguments A and B (yellow) are wiped from memory.

Should you have identical variable names in your main program and functions? That’s entirely up to you. Some programmers like to keep it that way for consistency. Others try very hard to avoid duplicating variable names to avoid confusion. The choice is yours, but the advantages of local variables are clear. When you have several people working on a program using functions, you no longer have to worry about your programmers duplicating variable names because Matlab will keep the local variables separate for you.

### 7.3.2 Global Variables

By default, every variable created in Matlab is local in scope. However, there are two notable exceptions. The first is variables that are global in scope. *Global variables* are
Chapter 7  User Defined Functions

those that can be used by a Matlab program and any or all of its functions. This is accomplished by having your main program and functions reference the same memory location. To become a global variable, you must first declare that variable to have global scope in the main program and all functions that use that variable. The syntax for a global declaration is straightforward:

    global  variable_name

Again, this declaration must appear everywhere that the global variable is to be used.
Let’s modify our previous example:

```
% Main Program
global  x
x = 15;
A = 13;
B = my_function(A);

function [B] = my_function(A)
global x
B = sqrt(A);
x = 10;
```

In this modified example, A and B are still local variables in the function and the main program. However, the global declarations indicate that the variable x referred to in both the main program and the function point to the same memory location. Thus, any change to the variable x will carry over. So if we were to execute the main program, it would begin by declaring the variable x to be global in scope. x is assigned the value 15. A is assigned the value 13 and then my_function is called, with the value of variable A (13) passed to the function.

Control is now shifted to the function. The value 13 is assigned to the dummy argument A. Next, the variable x is declared to be global. Matlab recognizes that this is the same x as was defined in the main program and has the value 15. Dummy argument B is now assigned the value of the sqrt(13). Finally, the value in x is now changed to be 10. The function ends and control is returned to the main program. The variable B in the main program is now assigned the value that the function returns. The function is defined in such a way that it returns the value in dummy argument B. That is, it returns the square root of 13.

There are no more commands in the main program, so the program terminates. Dummy arguments A and B no longer exist in memory. However, local variables A and B in the
main program persist. A still has the value of 13. B is now equal to the square root of 13. And x? Its value was changed in the function, and because it’s global, that changed value persists even when we’ve returned to the main program. Thus, the value of x is 10.

In this way, global variables are a way of exchanging information between the main program and its functions (or function to function for that matter) without having to pass the data as arguments in a function call. Global variables, therefore, should never be used as arguments passed to a function. You might be asking yourself why shouldn’t all variables be made global? While there are many times when global variables are appropriate, their use is generally discouraged. Their very nature causes a function to lose its independence. A global variable will usually be global over an entire program and all its functions, restricting the ability to duplicate variable names. It also raises the risk of accidental overwriting of its value in other functions. Finally, global variables are not a particularly efficient means of managing memory. When in doubt, try to use local variables instead.

7.3.3 Persistent Variables

A second exception to the local variable rule are those that are persistent in scope. **Persistent variables** are those that remain local to a particular function, but whose values are maintained between function calls. What does that mean? You can think of persistent variables as a hybrid or offshoot of local variables. Persistent variables only exist in functions, and like local variables, they can only be viewed, read and changed by the function that created them. Unlike local variables, however, their values aren’t deleted after the function ends. Their values persist in memory until the next time the function is called, at which point that previously saved value may be accessed.

A persistent variable must be declared as such within a function. The syntax is similar to that of a global variable:

```
persistent variable_name
```

To see persistent variables in action, consider the following example:
Chapter 7  User Defined Functions

Try running this program a few times. You should receive results similar to the ones below.

The value in total is kept in memory and used on each successive call of the function. However, that variable is not accessible outside the function.

7.3.4 Passing Arguments

A few words should be said about passing values to and from functions. Arguments passed to a function are assigned in the order in which they appear. For example, given the following function call and definition:

```matlab
function count = running_total(x)
%running_total Maintains a running total of all values passed to the function
% Function running_total will print the sum total of all values passed to it.
persistent total
% Declares the variable total to be persistent.
% Note: the local variable count defined in the function declaration can not be persistent!
% That’s because any variable that is going to be persistent %must be declared that way before appearing anywhere else in the function.
% That restriction includes the function declaration.
if isempty(total)
    total = 0;
end
% The first time this function is run, total exists as a persistent variable, %but it’s an empty matrix. This conditional checks to see if the matrix is %empty. If it is, that means it’s the first time the function has been run %and the value of total is set to zero.
    total = total + x;
    count = total;
```

EDU>> running_total(2)
ans =
   2
EDU>> running_total(6)
ans =
   8
EDU>> running_total(10)
ans =
  18
EDU>> running_total(-4)
ans =
   14
EDU>> total
??? Undefined function or variable ‘total’
The number 50 is assigned to dummy argument a. The number 25 is assigned to dummy argument b. And the value of actual argument x is assigned to dummy argument c. In this way, the value of each actual argument is assigned in order to the dummy arguments in the function definition.

An interesting property of Matlab is that it allows you to call a function with fewer arguments than are defined. For example, consider the following function call and definition:

```
sample_function2(50, 25, 30)
function sample_function2(a, b, c, d)
```

Like the previous example, the values of our actual arguments are assigned in order to the dummy arguments. Thus, a = 50, b = 25 and c = 30. But what about d? If a fourth actual argument is not passed, the function will be called but d will be an undefined variable. There is, in fact, nothing wrong with that. The problem arises when the code in your function depends on having a value for argument d. Robust programming requires you to write functions that can handle function calls with fewer arguments than you might expect.

It should be noted that although Matlab will allow you to call a function with fewer arguments than are defined, you can never call a function and attempt to pass more arguments than are defined. In that case, Matlab will simply return an error without calling the function.

Returning arguments from a function is perhaps an even simpler model. Consider the following function definition:

```
function [a, b, c, d] = test_function(x, y)
```

The function above has four arguments that may be returned: a, b, c, and d. To pass these values back, the dummy arguments must have values at the time the function ends. At that point, whatever values are assigned to the dummy arguments will be passed back in order. For example:
[A, B, C] = test_function(x, y)

function [s, t, v] = test_function(a, b)

Should a dummy argument not be defined at the time the function ends, it will not be passed back.

Perhaps the biggest trick in dealing with returning values to the function call is in writing the function call itself. What would happen if the function call and function definition above were modified slightly?

test_function2(x, y)

function [s, t, v] = test_function2(a, b)

What happens to the values of s, t, and v? By default, the first output argument in the function declaration will always be returned. If no actual output argument is defined in the function call, then that value will be placed in the ans variable. The values of t and v might be returned, but having no place to go, they will be immediately discarded.

Defining more output arguments than are actually called is not necessarily a bad thing. But a robust programmer will write their functions in such a way that the function doesn’t calculate and return more data than is actually called for.

7.4 Programming Functions

Robust programming requires that your functions be able to handle function calls with varying input and output arguments. Sometimes, that simply means returning an error message to the user informing them that you need additional arguments. Other times, it might completely change what your function does. Matlab has a series of built-in functions that will help you determine how your function has been called. The three most important of these functions are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nargin</td>
<td>Returns the number of actual input arguments used to call the function</td>
</tr>
<tr>
<td>nargout</td>
<td>Returns the number of actual output arguments requested by the function call</td>
</tr>
<tr>
<td>nargchk(min, max, nargin-or-nargout)</td>
<td>Returns a standard error message if your function was called with too many or too few arguments. (Remember…a function will never actually be called with more arguments than have been defined.)</td>
</tr>
</tbody>
</table>
Chapter 7  User Defined Functions

These functions should be used to determine how many arguments have been passed to the function in the function call and then how many arguments have been requested as output. The function you write should be able to accommodate the varying possibilities.

The following example problem illustrates the use of nargin and nargout to write robust functions.

Example 7.2 – Finding Prime Numbers (Revisted)

Modify the prior function to find the prime numbers between M and N. If only one argument is passed when the function is called, then the function should return an array of prime numbers between 1 and N. If requested, the function should also return as output the average of the prime numbers found.

```matlab
function [list_of_primes, avg] = ourprimes(M,N)
% OURPRIMES  Returns a 1xN vector of all the primes from 1 to N
% OURPRIMES(M,N)  Returns a 1x(N-M+1) vector of all the primes from M to N
% Method: Sieve of Eratosthenes (ca 240 BC)
% Variables:
% prime – 1xN, prime(n) = 1 if n is prime, 0 if n is not prime
% Check to see whether we have enough arguments.
% No need to check if we have too many because Matlab will not call
% a function with too many arguments.
if nargin == 0
    error('ourprimes requires 1 or 2 arguments');
end
% If only one argument, then we want to go from 1 to N.
if nargin == 1
    N = M;
end
% (1) Assume that all the integers from 2 to N are prime
prime = ones(1,N);
prime(1) = 0;
% (2) Mark all the numbers that are multiples of a number:
for n = 2:sqrt(N)  % No factor can be larger than sqrt(N)
    if prime(n)  % Multiples of primes include all multiples
        prime(n+n:n:N) = 0;
    end
end
(continued below)
```
% (3) Put the survivors on a list:
% If we’re going from M to N, clear all values from 1 to M-1.
if nargin == 2
    prime(1:M-1) = 0;
end
prime = logical(prime)
temp = 1:N;
list_of_primes = temp(prime);

% If it was requested, calculate the average prime:
if nargout == 2
    avg = sum(list_of_primes)/length(list_of_primes);
end

EDU>> ourprimes(25)
ans =
    2    3    5    7   11   13   17   19   23

EDU>> ourprimes(10,50)
ans =
     11   13   17   19   23   29   31   37   41   43   47

EDU>> [list, avg] = ourprimes(10)
list =
    2    3    5    7
avg =
    4.2500

EDU>> [list, avg] = ourprimes(5, 35)
list =
    5    7   11   13   17   19   23   29   31
avg =
    17.2222
Robust programming doesn’t just mean that your functions need to be able to handle varying numbers of input and output arguments, it also means it needs to be able to handle outputs of varying type. What happens if your function is expecting a number and it receives a text string instead? What happens if it receives an imaginary number instead of an array? There will be many occasions when you’ll wish for your function to perform a different operation depending on the type of your input arguments.

Example 7.3 -- Determining Complex Numbers

Write a function that will determine whether a given input is a complex number.

```matlab
function out = iscomplex(input)
    % ISCOMPLEX Determines whether the input argument is complex
    % The function will return 1 if the input is a complex number and
    % will return 0 if the input is not complex.
    if imag(input) > eps
        out = 1;
    else
        out = 0;
    end

EDU>> iscomplex(14)
ans = 0

EDU>> iscomplex(-4)
ans = 0

EDU>> iscomplex(3 + i)
ans = 1
```

The output of the example above is entirely dependent on the type of the input (whether it’s complex or real). Dynamic Polymorphism is a property of some functions. It exists in a function where the output is determined during the execution of the function by the type of the input arguments. Matlab subscribes to loose data typing. This means that variables in Matlab can change data types at will. A variable can go from holding an integer to a matrix to a text string to a decimal all in a few lines of code. This makes creating functions with dynamic polymorphism very easy in Matlab. However, most programming languages are not that way. They exhibit strong data typing, which means
once a variable is declared to be of a certain type, it’s very difficult (and sometimes impossible) to change that type. In those languages, dynamic polymorphism is difficult to accomplish, but is often a very lofty goal. The most common examples of dynamic polymorphism you’re likely to see might be found in data validation. Your function might evaluate the input arguments to ensure they are of the proper data type. If not, they will return a specific error message. In this way, the output (whether an error message or an actual result) may be determined by the type of input.

The example also illustrates another type of special function: a predicate. A predicate is a Boolean function that returns one of two values: true or false. In Matlab, this is represented by 0 for false and any non-zero value for true. Again, we often think of this as 1.

**Concepts from this chapter**

- Computer Science and Mathematics:
  - none

- Matlab:
  - defining a function
    - input arguments
    - output arguments
  - scope
    - local variables
    - global variables
  - persistant variables
  - passing arguments
Chapter 8
Data Types

Every modern programming language provides means for storing numbers in variables, operating on them, and printing them. Matlab is no exception, and we have seen countless examples of numbers being stored, operated on, and printed. The examples suggest that these numbers are the same as the real numbers and complex numbers of mathematics, where there is no upper limit to absolute value of a number and no lower limit to the absolute difference between two numbers. On the other hand, since the space of mathematical numbers is infinite, while the memory of a computer is finite, it is clear that the use of a computer must impose some limits. It does. There is an upper limit to the size of a number on the computer, and there is a lower limit to the absolute difference between two numbers.

The set of numbers that can be represented by a Matlab variable is, therefore, finite. We are about to learn that that there is more than one type of number in Matlab, and that each type provides a different set of allowed numbers. Furthermore, we will learn that there are types other than numeric types that hold other data. These other types cannot be used in arithmetic operations, but do allow other operations. A data type is in fact completely determined by the set of all possible values it can have and the set of operations that can be performed on it. Indeed, in the formalism of computer science, a data type, or simply, type, is defined as a set values and a set of operations that can be performed on those values.

It is common to employ variables of several different types in a single function. It is even possible to use variables of different types in the same operations. Matlab does, however, enforce one rule of uniformity: All elements of a given array must be of the same type. We call the types of these elements elementary types. Thus, when we speak of the type of a variable that is not a scalar, it is not necessary to specify the type of each element in the array. We need specify only one elementary type. A complete description of the type of an array will include: the number of dimensions, the size in each dimension, and the type of the elements. In Matlab there are always two or more sizes because the number of dimensions is always 2 or more (even a scalar is a one by one matrix!) Example descriptions of types are

- \( x \) is a 1 by 1 matrix with elements of type \texttt{double}
- \( x \) is a scalar of type \texttt{double}
- \( y \) is a row vector with 10 elements of type \texttt{single}
- \( y \) is a 1 by 10 vector with elements of type \texttt{single}
- \( A \) is a 2 by 3 matrix with elements of type \texttt{int8}
- \( B \) is a 2 by 3 by 4 by 5 array with elements of type \texttt{logical}
**The class function**

To determine the data type of a Matlab variable, use the function `class`. For example,

```matlab
>> x = 14
x =
    14
>> class(x)
ans =
double
```

(Matlab uses the name `class` and reserves the name `type` for a command that prints out the contents of an M-file.)

Values have types, even when they are not stored in a variable. Thus,

```matlab
>> class(14)
ans =
double

>> class('Mary had a little lamb. ')
ans =
char

>> class(class(14))
ans =
char
```

The first example shows that the type of the typical number is `double`. The second, shows that the type of a string is `char` (It is a matter of taste, whether “char” is pronounced like the first syllable of “charcoal” or like the word “care”). The third example shows that the type that is returned by `class` itself is of type `char`. Thus, `class` returns a string that spells out the name of the type.

While Matlab supports complex numbers, it does not distinguish among, real, imaginary, and complex numbers when classifying them into types.

```matlab
>> class(sqrt(-1))
ans =
double
```

Here we see that a clearly imaginary number, \( i = \sqrt{-1} \), is classified as type `double`.
8.1 Numeric types

Matlab uses many different internal representations to store numbers. Each representation is a numeric type. Most of the time a Matlab programmer need not be concerned with the specifics of the types of numbers used in a calculation. The default data type used by Matlab to store a number is **double**. Double is so versatile that it is capable of handling almost any numerical problem that comes up in engineering and science applications. The name “double” has historical roots. It was introduced as the name of a numeric data type in the 1960s in programming languages that supported two types of real numbers, one of which required twice as much memory space per number as the other. The one with twice the space was called “double precision”. The other one was called “real” and was said to employ “single precision”. In the 1970s these two types were renamed “double” and “float” in some new languages. The latter name is meant to indicate that a “floating-point” format is used to represent the number. (The specifics of the numeric formats is beyond the scope of this book.) Double uses floating-point representation as well, and, thanks to improvements in memory and CPU speed, it is now the default representation in most languages. The language C uses this name. C++ and Java use it as well. Today, if a computer supports any of these languages—Matlab C, C++, and/or Java—it will use the same amount of memory space for a number stored using the double type for programs written in any of these languages. C, C++, and Java employ the name “float” for single-precision numbers, but Matlab has adopted the term **single** for them.

Table 14 includes all of Matlab’s numeric data types along with the range of values that each supports. The names are mnemonic: The embedded word int in all but two of the names means “integer”, uint means “unsigned integer”. (The numbers indicate the number of bits used for storage, as explained below.)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Range of values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>int8</strong></td>
<td>-2^7 to 2^7-1</td>
</tr>
<tr>
<td><strong>int16</strong></td>
<td>-2^15 to 2^15-1</td>
</tr>
<tr>
<td><strong>int32</strong></td>
<td>-2^31 to 2^31-1</td>
</tr>
<tr>
<td><strong>int64</strong></td>
<td>-2^63 to 2^63-1</td>
</tr>
<tr>
<td><strong>uint8</strong></td>
<td>0 to 2^8-1</td>
</tr>
<tr>
<td><strong>uint16</strong></td>
<td>0 to 2^16-1</td>
</tr>
<tr>
<td><strong>uint32</strong></td>
<td>0 to 2^32-1</td>
</tr>
<tr>
<td><strong>uint64</strong></td>
<td>0 to 2^64-1</td>
</tr>
<tr>
<td><strong>single</strong></td>
<td>-3.4x10^38 to 3.4x10^38</td>
</tr>
<tr>
<td><strong>double</strong></td>
<td>-2.2x10^308 to 2.2x10^308</td>
</tr>
</tbody>
</table>

If you don’t have this table handy, the maximums and minimums can be gotten from Matlab by using these four functions: **intmax**, **intmin**, **realmax**, and **realmin**. Each takes one string as input. The string for the first two is the name of the integer type of interest. The string for the last two is ‘**double**’ or ‘**single**’. For example,
Chapter 8  Data Types

>> intmax('int32')
ans =
     2147483647

>> intmin('int32')
ans =
    -2147483648

>> realmax('single')
ans =
    3.4028e+038

>> realmin('single')
ans =
     1.1755e-038

Note that \texttt{realmin} gives the smallest positive number representable. The smallest negative number, which is shown in the table, is equal to \texttt{-realmax}.

\textit{Conversion functions}

To produce a numeric value of specific type Matlab provides “conversion functions”. The conversion function for a given data type has the same name as that type. It takes one input argument of any numeric type and returns one output argument of its specified type. Here are examples of conversion functions in action:

>> x = 10
x =
     10

>> class(x)
ans =
    double

From the commands above, we see that, the default numeric data type is \texttt{double}.

>> x = int8(10)
x =
     10

>> class(x)
ans =
    int8
From the two commands above, we see how the conversion function \texttt{int8} can be used to produce a numeric type of \texttt{int8} as output, when given an input type of \texttt{double}. We know that the input argument 10 is \texttt{double} because \texttt{double} is the default numeric type.

\begin{verbatim}
>> y = x
y =  
   10

>> class(y)
ans =
int8
\end{verbatim}

From the two commands above, we see, as we would expect, that the data type of \texttt{y} becomes the same as the data type of the value that is assigned to it.

\begin{verbatim}
>> y = double(y)
y =  
   10

>> class(y)
ans =
double
\end{verbatim}

From the two commands above, we see that the conversion function \texttt{double} can be used to produce an output of type \texttt{double}.

With the exception of \texttt{int64} and \texttt{uint64}, the advantage of each of the first 9 data types over \texttt{double} is that they require less memory space per number. Computer memory is measured in \texttt{bits}, which are the smallest units of memory on any computer, and \texttt{bytes}, which are groups of 8 bits. All but the last two of the data types in Table 14 include a number as part of their names. That number is equal to the number of bits required to store a single number of that type. The last two do not include a number in their names, but \texttt{single} requires 32 bits and \texttt{double} requires 64. Because computers manipulate their memories most efficiently by accessing a byte at a time, or a power of two bytes at a time, each type requires an integral number of bytes of storage—1, 2, 4, or 8.

All but two of the data types include “\texttt{int}” as part of their names. These are integer types, which means that they are capable of storing only integers. The integer types cannot store fractional parts. An attempt to convert a number with a fractional part to an integer type will result in rounding, as in the following examples:

\begin{verbatim}
>> int8(9.5)
ans =
   10
\end{verbatim}
Chapter 8 Data Types

>> int8(9.4)
ans =
   9

>> int8(-9.4)
ans =
  -9

>> int8(-9.5)
ans =
 -10

Furthermore, if a conversion function is given a value outside its range, it will return the value that is at the closest end of its range. For example,

>> int8(128)
ans =
  127

>> int8(-1000)
ans =
 -128

Some of the integer types have names that begin with the letter “u”. These types can store only non-negative integers:

>> uint8(-13)
ans =
    0

Arithmetic operations

We have seen many examples of arithmetic in Matlab, all involving the arithmetic operators +, -, *, /, \, ^, .*, ./, .\, and .^ . However, up to now, the operands have always been of double type. With the exception of int64 and uint64, all other numeric types can be used in arithmetic operations in Matlab as well. However, there are important restrictions on these operations. When the two operands of a binary arithmetic operator (see Chapter 3, Section 3.2) are of different types, the resulting operation is called mixed-mode arithmetic. In Matlab there are severe restrictions on mixed-mode arithmetic. For the expression x op y where op is an arithmetic operator, the following list gives in braces the set of allowed operators for each given pair of operand types:

---

6 These types cannot not be used in arithmetic operations in versions of Matlab before Version 7.
1. \(x\) and \(y\) are of floating-point types: \{ any arithmetic operator \}
2. \(x\) is of integer type and \(y\) is a floating-point scalar: \{+,-,\*, ./, .\, .^, *, /\}
3. \(x\) is a floating-point scalar and \(y\) is of integer type: \{+,-,\*, ./, .\, .^, *, \}
4. \(x\) and \(y\) are of the same integer type: \{+,-,\*, ./, .\, or .^\}
5. \(x\) and \(y\) are of the same integer type and \(y\) is a scalar: \{ / \}
6. \(x\) and \(y\) are of the same integer type and \(x\) is a scalar: \{ \}
7. \(x\) and \(y\) are of different integer types: NONE!

The last category shows that arithmetic operations involving differing integer types are always illegal.

A question arises immediately in mixed-mode operations: What is the type of the result?
When an operation of type 1 is carried out, the result has the type,

- **double**, if both operands were of type **double**, both were of type **char**, or one was **double** and one was **char**.
- **single**, if either of the operands was of type **single**.

When an operation involves an operand of integer type, which includes all types from 2 to 6, the result has that same integer type. Note that, except when a **char** is involved, Matlab uses the rule that the output type of an arithmetic operation is the same as the type of the input type with the smaller range. (This behavior is opposite that of C, C++, and Java. Their rule is that the output type is the same as the type of the input with the larger range.)

When the result of an arithmetic operation is out of range, Matlab gives the expression a special value according to the type of the result (\texttt{Inf} represents “infinity”):

- **double**: \texttt{Inf} or \texttt{–Inf}
- **single**: \texttt{Inf} or \texttt{–Inf}
- integer type: maximum or minimum of the range of that type

In summary, Matlab usually chooses the “smaller” type for its result, and, if the magnitude of the result is too large or too small for that type, it is set to the closer end of the range.

**Relational operations**

The relational operators, ==, \texttt{~=}, <, >, <=, and => all allow mixed-mode operands of any numeric types. They return a value of type **logical**. For example,

\[
\texttt{>> int8(4) < single(4)}
\]
\[
\texttt{ans =}
\]
\[
0
\]
>> class(ans)
ans =
logical

8.2 Strings

Strings were introduced in this book in Chapter 3 (Section 3.5) and have been used repeatedly as inputs to the functions input, in which they were used to provide a prompt to the user, and fprintf, in which the first argument was a string, the “format string”, which told fprintf what to print and how to print it. There is, however, much more to strings. A string is a value, like a numeric value, that can be assigned to a variable and manipulated by functions.

The ASCII encoding scheme

As we saw in an example at the very beginning of this chapter, a string is of type char. It is in fact stored as a sequence of numbers, each of which represents one character. The coding scheme is shown in Table 15. The encoding scheme assigns a number from 0 to 127 to each character. The encoding number is given to the right of each character. For example, the number that encodes an uppercase “A” is 65, while the number that encodes a lowercase “a” is 97.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Character</th>
<th>Encoding</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(nul)</td>
<td>96</td>
<td>`</td>
</tr>
<tr>
<td>1</td>
<td>(soh)</td>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>(stx)</td>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>(etx)</td>
<td>67</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>(eot)</td>
<td>68</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>(enq)</td>
<td>69</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>(ack)</td>
<td>70</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>(bel)</td>
<td>71</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>(bs)</td>
<td>72</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>(ht)</td>
<td>73</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>(nl)</td>
<td>74</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>(vt)</td>
<td>75</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>(np)</td>
<td>76</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td>(cr)</td>
<td>77</td>
<td>M</td>
</tr>
<tr>
<td>14</td>
<td>(so)</td>
<td>78</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>(si)</td>
<td>79</td>
<td>O</td>
</tr>
<tr>
<td>16</td>
<td>(dle)</td>
<td>80</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>(dc1)</td>
<td>81</td>
<td>Q</td>
</tr>
<tr>
<td>18</td>
<td>(dc2)</td>
<td>82</td>
<td>R</td>
</tr>
<tr>
<td>19</td>
<td>(dc3)</td>
<td>83</td>
<td>S</td>
</tr>
<tr>
<td>20</td>
<td>(dc4)</td>
<td>84</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>(nak)</td>
<td>85</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>(syn)</td>
<td>86</td>
<td>V</td>
</tr>
<tr>
<td>23</td>
<td>(etb)</td>
<td>87</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>(can)</td>
<td>88</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>(em)</td>
<td>89</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>(sub)</td>
<td>90</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>(esc)</td>
<td>91</td>
<td>[</td>
</tr>
<tr>
<td>28</td>
<td>(fs)</td>
<td>92</td>
<td>\</td>
</tr>
<tr>
<td>29</td>
<td>(gs)</td>
<td>93</td>
<td>]</td>
</tr>
<tr>
<td>30</td>
<td>(rs)</td>
<td>94</td>
<td>^</td>
</tr>
<tr>
<td>31</td>
<td>(us)</td>
<td>95</td>
<td>(del)</td>
</tr>
</tbody>
</table>

The name of this character encoding scheme, which is used by almost every programming language and, for that matter, almost every program that uses the Latin alphabet, is ASCII (American Standard Code for Information Interchange). The abbreviations in parentheses have special meanings to computer systems that use communication protocols that are of no interest here, except for these: (nl) means “newline”, (ht) means “horizontal tab”, and (sp) means space. These have meaning to

---

7 A far more general encoding scheme which is gradually replacing ASCII is Unicode. Unicode has three forms, one of which uses 32 bits and is capable of encoding over 4 billion characters.
fprintf, and there is a special escape sequences for each of them: \n produces (nl), and \t produces (ht). We saw the first one in Chapter 3 (Section 3.6). The ASCII scheme is designed for an eight-bit variable, which can represent 256 different characters, 128 of which are the standard characters given in the table. Matlab reserves 16 bits, or two bytes for each character, and it includes some characters in an “extended” ASCII set. We will not need them.

A string is a row vector

The sequence of ASCII numbers that encode a string is stored as a standard row vector. The length of the vector is equal to the number of characters in it, and it can be determined by using the same length function that we have applied before to numeric vectors. Furthermore, each individual element can be accessed using the usual indexing scheme. Here is an example:

```matlab
>> book_title = 'Matlab for Smarties'
book_title =
Matlab for Smarties

Note that, when Matlab prints the value of a string, it omits the quotes. Thus, it prints Matlab for Smarties, instead of 'Matlab for Smarties'.

>> length(book_title)
ans =
   19

There are 19 characters in 'Matlab for Smarties’, and that is the value returned by length. These characters include the two spaces, and, if there were any punctuation marks, they would have been counted too. Spaces and punctuation markers are all part of the string, and each one is encoded with its own ASCII number.

>> book_title(1)
ans =
   M

>> book_title(4:16)
ans =
   lab for Smart

>> book_title(7:11)
ans =
   for

Note that the word for on the last line is shifted slightly to the right. That is because the first character printed by the last command is a space. There is a trailing space there too,
but it is invisible. Checking the ASCII table above we see that the number that encodes a space (sp) is 32. We can use that information to look for spaces in a string. We also need to be able to convert the characters into their codes. Matlab enables you to do that by converting from the type char to the type double using the conversion function double. Thus,

\[
\text{>> double(' ')} \\
\text{ans = } \\
\text{32}
\]

Here, we have given double an argument that is a one-element character string consisting of one space. Now let’s look for spaces in book_title:

\[
\text{>> double(book_title)} \\
\text{ans = } \\
\begin{array}{ccccccccc}
\text{Columns 1 through 9} \\
77 & 97 & 116 & 108 & 97 & 98 & 32 & 102 & 111 \\
\text{Columns 10 through 18} \\
114 & 32 & 83 & 109 & 97 & 114 & 116 & 105 & 101 \\
\text{Column 19} \\
115
\end{array}
\]

As expected there are two 32s. The first one is followed by 102, which is the ASCII code for the letter ‘f’. The second one is followed by 83, which is the code for ‘S’. Numbers and punctuation marks are treated no differently. If they occur in a string, they are encoded in ASCII:

\[
\text{>> pi_digits = '3.14159'} \\
\text{pi_digits = } \\
3.14159
\]

\[
\text{>> double(pi_digits)} \\
\text{ans = } \\
\begin{array}{cccccccc}
51 & 46 & 49 & 52 & 49 & 53 & 57
\end{array}
\]

Here we have assigned the 7-character string ‘3.14159’ to pi_digits. These are the first digits of the number \(\pi\), but, in this case, instead of assigning the number 3.14159, which would be of type double, we have, by enclosing 3.14159 in quotes, assigned a vector consisting of seven elements, each of which is the ASCII code for one character. Since there are seven elements in this vector, the result of applying the function length to my_number should not be surprising:

\[
\text{>> length(my_number)} \\
\text{ans = } \\
7
\]
Like other vectors, new strings can be made from parts of existing ones:

```plaintext
>> s = 'Two is company, three a crowd.';
>> ssub = s(13:end)
ssub =
ny, three a crowd.
```

Also, new strings can be made by concatenating existing strings:

```plaintext
>> a = 'Three, two, one'; b = ', '; c = 'BLASTOFF!';
>> count_down = [a,b,c]
count_down =
Three, two, one, BLASTOFF!
```

Strings can even be put together to make arrays with more than one row and column:

```plaintext
>> first = 'abcdefgh'
first =
abcdefgh

>> second = '12345678'
second =
12345678

>> both = [first;second]
both =
abcdefgh
12345678

>> double(both)
anst =
   97  98  99 100 101 102 103 104
   49  50  51  52  53  54  55  56

>> both'
anst =
a1
b2
c3
d4
e5
f6
g7
h8
```

Here we have made an array with two rows in which all the elements are of type char. It is important, though, to remember that arrays must be rectangular. That is, each row
must have the same number of elements. In this case, the variables `first` and `second` are both of length 8, so each row of both has 8 elements. It is not possible to make a “ragged” array, in which rows contain strings of differing lengths, as the following attempt shows:

```matlab
>> long = 'Matlab'; short = 'Java';
>> languages = [long; short]
??? Error using ==> vertcat
All rows in the bracketed expression must have the same number of columns.
```

(The reference to `vertcat` refers to the internal function that Matlab uses to put row vectors together into arrays.)

**Arithmetic operations**

It may seem odd at first, but, like the numeric data types, strings can be used in arithmetic operations! When we do arithmetic on them, Matlab treats them as a vector of ASCII numbers, instead of a string of characters. Like the integer types, a string can be used in mixed-mode arithmetic. A major difference, though, is that when a string is used in mixed-mode arithmetic, the result of the operation is the type of the numeric operand, whereas, as pointed out in Section 8.1, when an integer variable is used in mixed-mode arithmetic, the result is of the same integer type. Here is an example of arithmetic on strings:

```matlab
>> language = 'Matlab'
language =
Matlab

>> double(language)
ans =
77   97  116  108   97   98

>> big_language = 2*language
big_language =
154  194  232  216  194  196

>> class(big_language)
ans =
double
```

We see that when we multiply `language`, which is of type `char`, by 2, which is of type `double`, the result is of type `double`, as predicted above.
Note also the non-Latin characters that showed up when we converted back to char. These are some of the characters mentioned above that Matlab supports in addition to the Latin ones. The little square box ( ) indicates that there is no printable character (not even a space) associated with the number 154.

>> char(big_language/2)
ans =
Matlab

This last command produces the original string again by dividing out the factor of two and converting once again to the type char. We have our original string.

This forward-backward conversion suggests a simple (and easy to crack) coding scheme for secret messages: Any operation that can be undone is suitable to encode a message. Here is another example,

>> secret_plans = '1pm: Stairs behind door at right to basement';
>> coded_plans = char(158-secret_plans)
coded_plans =
m.1d~K*=5,+~<9650:~://,~=*~,576*~*/~<=+9190*

Well, this is certainly a mess!

>> decoded_plans = char(158-coded_plans)
decoded_plans =
1pm: Stairs behind door at right to basement

We are back to our original message again.

Relational operations

The relational operators, ==, ~=, <, >, <=, and >= all allow mixed-mode operands of any numeric type and of type char. They return a value of type logical (see below). As when we do arithmetic on them, Matlab’s relational operators treats strings as a vector of ASCII numbers, instead of a string of characters. For example,

>> int8(97) < 'a'
ans =
0
Chapter 8   Data Types

>> single([97 97]) < 'ab'
ans =
    0     1

Here each element of the vector [97 97] is compared in turn with the corresponding element of the vector of ASCII numbers [97 98] that encode ‘a’ and ‘b’. The following example shows how this ability compare on the basis of ASCII codes might be useful:

>> 'agbe' < 'ah f'
ans =
    0     1     0     1

Thanks to the sensible order of the ASCII codes, letters that come earlier in the alphabet also have smaller numbers. Thus, a program that is required to alphabetize words can compare the ASCII numbers in order to determine which letters come first. It is interesting to note that the upper case letters all precede the lower case ones. It is for this reason that many computer-sorted word lists put all capitalized words before all the uncapsitalized ones. It may not be the preferred order for a given application, and it is not necessary to order them that way. However it is the way that requires the least complex program, and so programmers have often done it that way.

String functions

While it is possible to do almost anything we wish with strings simply by using the ordinary tools for manipulating vectors, such as indexing, the colon operator, concatenation, arithmetic, relational operators, etc., there are many tasks that have already been written as functions for us by the Matlab programmers. Matlab provides a number of built-in functions that include strings as input arguments and/or output arguments. Table 16 gives a partial list. One of these functions is very similar to a function that we have already learned about in great detail. It is sprintf. It is similar in functionality to fprintf. Both functions permit you to stipulate in detail how data is to be printed. The difference is that, while fprintf causes its output to appear directly in the command window, sprintf puts its output into a string. Here is a simple example that reveals the difference:

>> x = pi;

>> fprintf('x:\nto three decimal places: %6.3f
', x);
x:
to three decimal places:  3.142

>> s=sprintf('x:\nto three decimal places: %6.3f
', x);

>> s
Chapter 8  Data Types

$s =$
$x:$
to three decimal places:  3.142

We see by this example that \texttt{fprintf} itself forms a string, but it is not possible to capture
the string in a variable, as it is with \texttt{sprintf}. We see also that, with both \texttt{fprintf} and
\texttt{sprintf} it is possible to use the escape characters ‘\n’ to produce the newline
character. The following example makes that clearer:

\begin{verbatim}
>> s = sprintf('\n')
s =

>> double(s)
ans =
   10
\end{verbatim}

Checking Table 15 we see that the ASCII code for newline (nl) is indeed 10.

\begin{table}[h]
\centering
\begin{tabular}{|c|p{10cm}|}
\hline
Name & Description \\
\hline
\texttt{char} & converts type to char \\
\texttt{findstr} & finds the positions of a substring in a string (see also \texttt{strfind}) \\
\texttt{ischar} & returns 1 if argument is a character array and 0 otherwise \\
\texttt{isletter} & finds letters in string \\
\texttt{isspace} & finds spaces, newlines, and tabs in string \\
\texttt{isstrprop} & finds characters of specified type in string \\
\texttt{num2str} & converts number to string \\
\texttt{length} & determines the number of characters in string \\
\texttt{lower} & converts letters in string to lower case \\
\texttt{regexp} & finds substrings defined by a regular expression \\
\texttt{sprintf} & writes formatted data to string (cf. \texttt{fprintf}) \\
\texttt{strcat} & concatenates strings \\
\texttt{strmatch} & search array for rows that begin with specified string \\
\texttt{strncmp} & applies \texttt{strcmp} to the first n characters of strings \\
\texttt{strcmp} & returns 1 if two strings are equal and 0 otherwise \\
\texttt{strcmpi} & like \texttt{strcmp} but independent of case \\
\texttt{strncmp} & compares first n characters of strings \\
\texttt{strncmpi} & like \texttt{strncmp} but independent of case \\
\texttt{str2num} & converts string to number \\
\texttt{upper} & converts string to upper case \\
\hline
\end{tabular}
\caption{String Functions}
\end{table}

For the remaining functions in Table 16 it is recommended that, when you have a task to
perform with strings, such as alphabetizing a list, looking something up in a database of
strings, or using strings in a graphical-user-interface, you consult this table, find possibly
useful functions, and use the Matlab `help` command to learn the details of their operation. The most powerful of these functions for searching for substrings is `regexp`. To learn about regular expressions go to the Help Navigator (Chapter 2, Section 2.4) and click on MATLAB/Programming/Basic Program Components/Regular Expressions.

### 8.3 Structs

An array in Matlab, C++, Java, Fortran, and most other languages, consists of a collection of elements, each of which is of the same elementary data type. The array is said to be homogeneous with regard to the types of its element. It is not possible, therefore, to have a matrix whose first row consists of elements of type `int16` and whose second row consists of elements of type `char`. Such an array, were it legal, would be heterogeneous with respect to the types of its elements. An attempt to set up any heterogeneous array will be thwarted by Matlab. Here is an example of what happens,

```matlab
>> A = [int8(1),int16(2);int32(3),uint32(4)]
Warning: Concatenation with dominant (left-most) integer class may overflow other operands on conversion to return class.
Warning: Concatenation with dominant (left-most) integer class may overflow other operands on conversion to return class.
Warning: Concatenation with dominant (left-most) integer class may overflow other operands on conversion to return class.
A =
    1    2
    3    4

>> class(A)
ans =
int8

>> class(A(1,1)),class(A(1,2)),class(A(2,1)),class(A(2,2))
ans =
int8
ans =
int8
ans =
int8
ans =
int8
```
Matlab imposes its homogeneity restriction by converting all but the first element in the list to be of the same class as that first element, which in this particular case is \texttt{int8}. It also prints a warning for every element that is converted. The homogeneity restriction is actually quite reasonable because of the important feature of arrays that allows an operation to be specified simultaneously for all elements of the array, as for example \( x = \texttt{int8}(2)*A \). This operation would legal for \( A \) above, since the elements of \( A \) are all of type \texttt{int8}, but it would legal for only \( A(1,1) \), if Matlab allowed heterogeneous arrays.

One might argue about the importance of homogeneity to the functionality of arrays, but it is also important from an efficiency standpoint. Accessing individual elements of an array can be handled internally by Matlab, or any other language for that matter in an especially efficient way if the types of the array’s elements are all the same, but efficiency of the inner workings of Matlab is beyond the scope of this book, and when array-wide operations are not important, then the homogeneity restriction becomes onerous. If, for example, we wish to store information about a person in an array, we might want to put both his or her social security number and his or her name in the same data structure. In this case, a first row of type \texttt{int16} and a second of type \texttt{char} would work very well.

Fortunately, for applications in which array-wide operations are not required and heterogeneous elements are desired, Matlab supports a perfect data type. It is called a \texttt{struct}. Like an array, a \texttt{struct} consists of a set of elements, but unlike an array those elements can be of differing data types. There is another important difference as well. Each element is indexed by user-defined name instead of numerical indices. Since that description is hard to understand at first, we will quickly provide an example:

\begin{verbatim}
>> r.ssn = 568470008
r =
   ssn: 568470008

>> class(r)
ans =
struct

>> class(r.ssn)
ans =
double
\end{verbatim}

Here we have created a user-defined name, \texttt{ssn}, to serve as an index for an element of a variable \( r \). The dot between \( r \) and \texttt{ssn} indicates that the name to the right of it is the name of an element of the variable to the left of it. A name used as an index is called a \texttt{field name}, and the value associated with that name is a \texttt{field}. The variable \( r \) is now a \texttt{struct}. It has one field, whose name is \texttt{ssn} and whose type is \texttt{double}. We can add as many fields as we like:

\begin{verbatim}
>> r.name = 'Homer Simpson'
\end{verbatim}
Chapter 8   Data Types

\[
\begin{align*}
\text{r} &= \\
&= \\
&\quad \text{ssn: 568470008} \\
&\quad \text{name: 'Homer Simpson'}
\end{align*}
\]

>> r

\[
\begin{align*}
\text{r} &= \\
&= \\
&\quad \text{ssn: 568470008} \\
&\quad \text{name: 'Homer Simpson'}
\end{align*}
\]

Each field can be of any type. It can, for example, be another struct:

>> r.address.street = '742 Evergreen Terrace'

\[
\begin{align*}
\text{r} &= \\
&= \\
&\quad \text{ssn: 568470008} \\
&\quad \text{name: 'Homer Simpson'} \\
&\quad \text{address: [1x1 struct]}
\end{align*}
\]

We have added a new field to r, called address and we have made the type of that field be a struct. That inner struct has one field, called street, and its value is the string '742 Evergreen Terrace'. Notice that when the structure gets this complex, when Matlab is commanded to display the contents, it resorts to abbreviating the inner structure a bit. Thus, instead of showing the structure of the inner struct, it simply shows that it is a struct. It shows that by means of the struct in the phrase, [1x1 struct]. (The “1x1” does not mean that the struct has just one field. Its meaning will be made clear below.) We can add an additional field to the inner struct:

>> r.address.city = 'Springfield'

\[
\begin{align*}
\text{r} &= \\
&= \\
&\quad \text{ssn: 568470008} \\
&\quad \text{name: 'Homer Simpson'} \\
&\quad \text{address: [1x1 struct]}
\end{align*}
\]

There is no visible change in Matlab’s display of the contents of r because, while we have added another field to the struct within the third field of r, we have not changed the fact that the third field of r is a struct. If we want to look at the inner struct itself, we can do it as follows:

>> r.address

\[
\begin{align*}
\text{ans} &= \\
&= \\
&\quad \text{street: '742 Evergreen Terrace'} \\
&\quad \text{city: 'Springfield'}
\end{align*}
\]
Arrays and structs

The value of any field of a struct can be an array of any dimension. Thus, for example,

```matlab
>> r.years_on_air = 1989:2005;
>> r
r =
    ssn: 568470008
    name: 'Homer Simpson'
    address: [1x1 struct]
    years_on_air: [1x17 double]
```

Here we see that, when Matlab displays the contents of the last field, it shows only that it is a 1 by 17 array type double. The contents of the field address is shown as a 1 by 1 array of type struct. Matlab chooses to show only the types when the values would occupy too much space on the command window. Smaller arrays are displayed in full, as for example:

```matlab
>> pixel.position = [567, 688];
>> pixel.rgb = [145, 155, 134];
>> pixel
pixel =
    position: [567 688]
    rgb: [145 155 134]
```

Since short numeric vectors fit conveniently on one line, they are displayed in full. As a special case of this sort of display, strings, which are stored as row vectors of type char, are displayed in full, as was, for example 'Homer Simpson' above.

Accessing individual elements of an array that is a field of a struct is done as usual. Thus,

```matlab
>> pixel.rgb(1:2)
an =
    145  155
```

Structs can also be elements of an array themselves, as long as the homogeneity of the array is maintained. The type of a struct is specified by its field names. Thus, if an element of an array is a struct, then all elements of that array must be structs and all must have exactly the same set of field names:

```matlab
>> clear picture;
```

```matlab
>> picture(2,2) = pixel
picture =
```
2x2 struct array with fields:
  position
  rgb

The clear command is included above to show that picture is non-existent before the second command assigns it a value. As we saw in Chapter 3, when a value is assigned to an element of variable on a column or row that did not previously exist for that variable, then a new row and column is created along with all the rows and columns necessary to produce a rectangular array of values. In doing that Matlab creates elements in addition to the one specified in the command, and it gives each of them an initial value. If the type of the array is numeric, the initial value is zero. If the type is struct, then the initial value is a struct with the value for each field set to the empty matrix. Thus,

```matlab
>> picture(1,2)
ans =
    position: []
    rgb: []
```

Finally, we note that, since the type of a struct is defined completely by the list of its field names, the types of the values of those fields does not affect the type of the struct. As a result, we can, for example, without violating Matlab’s rule that an array be homogeneous, give the rgb field of element (2,1) of the array a value of type char,

```matlab
>> picture(2,1).rgb = 'k';
```

while the rgb field of element (2,2) retains its previously assigned value of the 3-element numeric row vector:

```matlab
>> picture(2,1).rgb
ans =
    k
>> picture(2,2).rgb
ans =
    145   155   134
```

We have seen that structs can have fields that are structs or arrays that and arrays can have fields that are structs. This nesting of one data type within another can extend to any depth.

**Arithmetic and Relational operations**

Objects of type struct cannot be used as operands of any of the arithmetic operators or relational operators or as the arguments of a function that expects numeric arguments. However, it is perfectly legal to apply these operators or functions to any field of a struct, if it is of numeric type. Here are a couple examples:
>> picture(1,2).position = picture(2,2).position + 1;

>> picture(1,2).position
ans =
    568   689

>> picture(1,2).rgb = round(picture(2,2).rgb./[1 2 3])

>> picture(1,2).rgb
ans =
    145    78    45

Dynamic Field Names

When writing functions that manipulate structs, it is sometimes desirable to create field names during the execution of the function. Creation of field names during execution requires additional functionality. One convenient approach is to use Matlab’s dynamic field-naming operation. Here is an example, of this operation,

>> zoo.lions = 3;

>> zoo.('tigers') = 2;

>> zoo
zoo =
    lions: 3
    tigers: 2

The first command is a conventional struct assignment. The second one is the dynamic operation. Note the dot before the left parenthesis. The string 'tigers', is read by the “dot-parentheses” operator and converted into a field name. To see how this operator might be used in a program, consider the following function, which allows the user to create a customized structure during its execution,

```
function s = create_struct_dynamic
% S = CREATE_STRUCT_DYNAMIC Create a structure from field names
% and values that are input by the user
while 1
    field_name = input('Enter a field name (zero to quit): '); if field_name == 0
        break;
    end
    field_value = input('Enter value for this field: '); s.(field_name) = field_value;
end
```

8 Dynamic field names are not available in versions of Matlab previous to version 7.
Then, we use \texttt{create\_struct\_dynamic} to create a structure,

\begin{verbatim}
>> children_ages = create_struct_dynamic;
Enter a field name (zero to quit): 'Katie'
Enter value for this field: 20
Enter a field name (zero to quit): 'Johnny'
Enter value for this field: 18
Enter a field name (zero to quit): 0
\end{verbatim}

and we check to see that the fields that we input are indeed the names of the fields of our new \texttt{struct}:

\begin{verbatim}
>> children_ages
children_ages =
    Katie: 20
    Johnny: 18
\end{verbatim}

\textit{Struct functions}

There are a number of useful functions designed specifically for working with \texttt{structs}. A partial list is given in Table 17.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Name & Description \\
\hline
\texttt{getfield} & returns the value of a field whose name is specified by a string \\
\texttt{isfield} & true if a \texttt{struct} has a field whose name is specified by a string \\
\texttt{isstruct} & returns true if argument is of type \texttt{struct} \\
\texttt{orderfields} & changes the order of the fields in a \texttt{struct} \\
\texttt{rmfield} & removes from a \texttt{struct} a field whose name is specified by a string \\
\texttt{setfield} & assigns a value to a field whose name is specified by a string \\
& -or- if the field is not present, adds it and assigns it the value \\
\texttt{struct} & create a \texttt{struct} with fields whose name are specified by strings \\
\hline
\end{tabular}
\caption{Table 17 Struct Functions}
\end{table}

Several of the descriptions in the table include the phrase, \textit{“a field whose name is specified by a string”}. Like the dynamic field-naming operation, these functions allow for the possibility of a string that is formed during the execution of a program to be used with \texttt{structures}. Here is a function that produces the same result as
create_struct_dynamic above, but it does it by means of two of the functions in the table, instead of the dot-parentheses operator:

```matlab
function s = create_struct
% S = CREATE_STRUCT Create a structure from field names
% and values that are input by the user
first_field = 1;
while 1
    field_name = input('Enter a field name (zero to quit): ');
    if field_name == 0
        break;
    end
    field_value = input('Enter value for this field: ');
    if first_field
        s = struct(field_name,field_value);
    else
        s = setfield(s, field_name, field_value);
    end
end

Note that we must use the function struct to create the struct initially, but once the struct has been created, we must use setfield to add fields to it.

8.4 Cells

We noted above that structs and arrays can be nested within each other and structs can be nested within structs. What about arrays within arrays? Matlab provides that option as well, albeit indirectly. It is provided by means of a new data type, called a cell. To understand the cell, we must understand a bit about the way in which a program references its variables.

The table of variables

As we saw in Chapter 7, when we were discussing the idea of local variables, each variable has its own location in the computer’s memory. The operating system (e.g., Mac OS, Linux, or Windows) allocates portions of the computer memory for Matlab’s exclusive use while it is running, and Matlab, in turn allocates portions of its portion to variables as they are used. Each time Matlab encounters a variable in a command, either in its Command Window or during the running of a function, it looks up the name in a table of names to see whether it has already been defined. If not, then it checks the command to see whether it requires that the variable’s value be looked up. If the value
must be looked up when no such variable exists, then Matlab halts execution of the program and issues an error message, as for example,

```matlab
??? Undefined function or variable 'x'.
```

If, on the other hand, the command assigns a value to a variable that has not already been defined, it defines it immediately by placing its name into a table and choosing an area of memory in which to store the variable’s values. The area of memory is a set of bytes, and each byte in the memory has its own unique address. Matlab will choose a contiguous set of bytes and will put the address of the first byte into the table, along with information about the type, size and shape of the variable. It reserves this area and will not use it for any other variable. Every variable has its own unique area of memory.

You can look at part of the contents of the table. You see it by issuing the command `whos`. For example,

```matlab
>> a = 5;
>> b = int8([5 6;-9 9]);
>> c = 'Tiger';
>> whos
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1x1</td>
<td>8</td>
<td>double array</td>
</tr>
<tr>
<td>b</td>
<td>2x2</td>
<td>4</td>
<td>int8 array</td>
</tr>
<tr>
<td>c</td>
<td>1x5</td>
<td>10</td>
<td>char array</td>
</tr>
</tbody>
</table>

Grand total is 10 elements using 22 bytes

Here we see that three variables are defined. (In the terminology of computer science the set of defined variables is called an “activation record”. In Matlab terminology, it is called a “Workspace”. What is displayed above is the activation record, or workspace, of the command window. Each function also has an activation record when it is running. We will return to this concept of multiple workspaces in Chapter 9, when we study recursion.) The dimensions of each variable in the table is given in the second column, labeled “Size”; the amount of memory occupied is given in the “Bytes” column, and its type is given under “Class”. The addresses are in the table too, but they are not shown by the `whos` command.

When a command references a variable that is already in the table, Matlab finds the name in the table, obtains the size, type, and address associated with the name, and either uses the value that it finds at that address or gives the variable a new value. The following command gives an example of both uses,

```matlab
>> c = a
```

```matlab
ans =
    5
```
>> whos
Name      Size                    Bytes  Class
  a         1x1                         8  double array
  b         2x2                         4  int8 array
  c         1x1                         8  double array

Grand total is 6 elements using 20 bytes

Here, Matlab first looks up the variable a, finds it in the table, discovers that it is a 1x1 array of type double, obtains its address, goes to that address in the computer’s memory, and discovers the value 5 located there. It then looks up the variable c, finds it in the table, discovers that it is a 1x5 array of type char, and it replaces that value by a 5 of type double. In order to accomplish that, it changes the size and type for c in the table to 1x1 and double and stores the new value, in this case the number 5, at the address that is given in the table for c. If the new value is too large to fit in the space already reserved for that variable, then Matlab allocates for c enough new space to hold the new value, puts the value there, and changes the address for c to the that of the new space. That would happen, for example, if the new value were a 10x10 array, as in the command,

>> c = rand(10);

>> whos c
Name      Size                    Bytes  Class
  c        10x10                      800  double array

Grand total is 100 elements using 800 bytes

It is possible at any time, both while issuing commands in the Command Window, and within a running function, to determine whether a variable exists in the table by using the function exist, which returns true if and only if the variable whose name is input as a string, is found in the table, as for example,

>> exist('c')
an =
   1

It is also possible to remove a variable from the table at any time using the command clear:

>> clear('c')
>> exist('c')
an =
Pointers

What is not shown in the table of variables is the address of each variable. That information is never revealed in the variable table, or any table, by Matlab for any variable. There is no option to do so, and addresses are not available via any function or operation. Matlab is different in this regard from some languages, such as C and C++, which provide an operator that returns the address of any variable whose name is given as an operand to the operator. These languages also permit one variable to hold the address of another variable, which is another practice that Matlab does not allow. In these languages the former variable is said to hold a pointer to the other variable. In fact, in the parlance of programming languages the term pointer means simply the address of an object in memory. When a variable holds an address, it is called pointer variable.

Hidden addresses

In C and C++ a pointer’s value can be set to any desired value, printed, assigned as the value of a variable, and generally treated like any other number. Matlab’s implementation is a bit more like that of Java, which also keeps the addresses of its variables hidden. There are debates about the decision to reveal or keep hidden the addresses used during the running of a program. When they are revealed, the program has the power to select specific addresses for special uses, as for example input and output to hardware devices using registers with specific addresses. The ability to specify in the code a specific address is particularly important feature for embedded applications, which are applications that run on small, special-purpose computers in appliances, cars, etc. Having addresses open to direct manipulation is advantageous in these applications, but there are disadvantages too. When addresses are kept hidden, it is far less likely that a change in the address space in which the program runs will affect the outcome of the program. Thus, a Matlab or Java program that runs properly on one computer is made more likely to run properly on another computer because changes in the address space from one computer to another cannot affect their programs.

Indirect implementation of heterogeneous arrays

With regard to pointers, Matlab is, however, more restrictive even then Java, because Matlab does not allow the creation of a pointer to a variable. Matlab uses pointers only as a means to provide an indirect implementation of heterogeneous arrays. The indirect implementation is accomplished through the use of a data type called the cell. The cell is the data type in Matlab that holds an address. (The name “cell” is used because a single memory location is often called a memory cell.) Thus only a variable of type cell can hold a pointer. A cell variable may be created in one of three ways. The first way is by means of the function called cell:
The function `cell` creates an array of type `cell` and initializes each element to contain the address of an empty matrix. (This operation is analogous to the use of the function `zeros` to produce a numeric array of zeros.) The result is depicted schematically in Figure 1.

```
>> p = cell(2,3)
p =
    {}    {}    {}
    {}    {}    {}
```

Like any matrix in Matlab, `p` is homogeneous: All its elements are of the same type, and in this case the type is `cell`. Also, like any matrix, we can access any element of `p` by putting the indices of the element in parentheses,

```
>> p(1,2)
an =
    {{}}
```

The set of curly braces around the empty matrix indicates that `p(1,2)` is, as we know it must be, of type `cell`. Those braces are Matlab’s way of telling us that `p(1,2)` contains the address of the empty matrix. The value inside the braces, in this case the empty matrix, is the object whose address is stored in `p(1,2)`.

Curly braces are featured in much of the syntax involving arrays of type `cell`, as we will see now. For example, curly braces are part of the syntax that allows us to gain access to the object to which `p(1,2)` points. We do that by replacing the parentheses by curly braces in the indexing operation,

```
>> p{1,2}
an =
    {}
```

The phrase `(ii, jj)` refers to the pointer that is stored at position `ii, jj`, while the phrase `{ii, jj}` refers to the object to which it points.

Let’s look at what `whos` will tell us about `p`:

```
>> whos p
    Name             Size          Bytes  Class
    p                2x3 cells     48 bytes  cell
```
Figure 2  Pointer to a scalar array is assigned

The variable \( p \), which is of size 2x3, contains 6 elements and, as shown in the table, occupies 24 bytes. Each element consists of the address of one location in memory. Matlab uses 4 bytes to encode each address. (This choice was made because the number of distinct address that can be stored in 4 bytes—\( 2^{32} \)—or 2,147,483,647, which is over 2 billion bytes, seemed to be big enough to handle any conceivable computer at the time, and it complies with the address formats of current CPUs. Other languages also use four bytes to store addresses, but with the growth of computer memories, these language designers may soon be forced to increase the size of their address holders.)

Each of the 6 pointers points to a location in memory that stores an empty matrix, as Figure 1 shows. A more useful array would include pointers to matrices with something in them. Let’s put a single scalar into the first element,

\[
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\]

Oops. Matlab “thinks” that we are trying to set the address that is stored at \( p(1,1) \) to 17. As mentioned above, Matlab does not allow us to choose the value of a pointer (or even to know that value that Matlab chooses). The notation that we use to let Matlab choose the address, but let us tell Matlab what to put at the address, is a pair of curly braces,

\[
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\]

This example shows the syntax for assigning a value to an element of a cell array. The value that is assigned is not the number 17. Instead, as is illustrated by Figure 2, it is a pointer to a location in memory where Matlab will put the number 17. We get to stipulate what array the element points at. Matlab chooses the address at which to place that array, and then it writes that address into \( p(1,1) \). Matlab does not reveal the address at which the new array is stored, but it will tell us how much memory it requires. We can do that with another whos command,
By adding one single scalar, we have increased the number of bytes occupied by \( p \) from 24 to 88, an increase of 64! Why are so many additional bytes needed for the number 17? First there are 8 bytes required to hold the number, because it is by default of type \texttt{double}. Note, by the way, that the “Grand total” of 7 elements includes the six pointers plus the one number. The remaining 56 bytes are employed by Matlab to encode a descriptor of the scalar 17. That descriptor holds information, such as \texttt{Size = 1x1} and \texttt{Class = array of doubles}, along with other (secret) information that Matlab will need when it is eventually required to use that object in a calculation. These elements, and all these 88 bytes are associated with the variable \( p \), in Matlab’s system. If \( p \) is cleared, then all these elements and all these bytes disappear from the workspace.

Now, let’s put a pointer to something else at element (1,2), say a two-element row vector of type \texttt{double},

\[
>> \ p\{1,2\} = [-8 \ \texttt{pi}]
\]

\[
\begin{array}{c|c|c}
\hline
\text{Name} & \text{Size} & \text{Bytes} \\
\hline
\text{p} & \text{2x3} & \text{160} \\
\hline
\end{array}
\]

This time, Matlab gives an abbreviated description of the object that is pointed at by the second element, just as it did when it was displaying the contents of the fields of \texttt{structs} in Section 8.3. Note that the total number of elements increased by two when we added the vector of two elements. The number of bytes increased by 72 (= 160 – 88). Sixteen of these bytes are required for the two additional numbers, \(-8\) and \texttt{pi}, which are of type \texttt{double}; remaining 56 bytes are again needed for a descriptor.

Now, just to show the great variety of possible objects that can be pointed at by elements of a \texttt{cell} array, let’s make the first element on the second row a pointer to a \texttt{uint16}, the second element a pointer to a \texttt{2x4} array of \texttt{int8}, and the last element on the second row a pointer to a string,

\[
\begin{align*}
& >> \ p\{2,1\} = \texttt{uint16}(34978); \\
& >> \ p\{2,2\} = \texttt{int8}([1 \ 10; 2 \ 20; 3 \ 30; 4 \ 40]) \\
& >> \ p\{\text{end,end}\} = 'Awesome!'
\end{align*}
\]
With this example we can see clearly how Matlab achieves its “indirect” implementation of heterogeneous arrays. Figure 3 illustrates the situation. The array \( p \) is certainly not heterogeneous, since all its elements are of type cell, but the pointers in those elements point to objects of differing types, and that is why we say that the cell array is an indirect version of a heterogeneous array. The difference in the types is illustrated in the figure both by the difference in the size of the boxes—large for the 8-byte double type and small for the two-byte int16 and chars—and by the quotes around the values of type char. In this regard Matlab is more flexible than C, C++, or Java. These three languages certainly provide for arrays of pointers, but they each require that all pointers in the same array point to objects of the same type.

Throughout this subsection, we have been using curly braces to manipulate a cell array that we created by means of the cell function. There are three other ways to create a cell array, each involving curly braces. Here are examples showing the syntax of each,

First:

\[
>> A = \{1, \text{'hello'}; \text{int8}(7), \{1 2; 3 4\}\};
\]

Second:

\[
>> A(1,1) = \{1\};
>> A(1,2) = \{'hello'\};
>> A(2,:) = \{\text{int8}(7), \{1 2; 3 4\}\};
\]

Third:

\[
>> A{1,1} = 1;
>> A{1,2} = \text{'hello'};
>> A{2,1} = \text{int8}(7);
\]
>> A{2,2} = [1 2; 3 4];

All three of these examples produce the same cell array,

>> A
A =
[1]    'hello'
[7]    [2x2 double]

Accessing subparts of cell arrays

As we have seen, we can access individual elements of cell arrays by using subscripting and there are two options for enclosing the subscripts, parentheses and curly braces. The first option gives us a pointer; the second gives us the object pointed at. What if that object is an array, and we wish to access individual elements within it? We use a concatenation of indices in curly braces followed by indices in parentheses. Here is an example, which is depicted in Figure 4,

>> p{2,2}(3,2)
an =
3

The first set of indices, which are enclosed in curly braces, identify element 2,2 in the cell array. The second set of indices, which are enclosed in parentheses, identify element 3,2 within the array pointed at by cell element 2,2.

The colon operator can be used to assign a subarray, as usual. It can be used within braces, as for example,

>> q = p(1:2,2:3)
q =
[1x2 double]        []
[4x2 int8  ]        'Awesome!'

and it can be used in the concatenated parentheses,

>> q{2,2}(2:4)
an =
wes
Distinguishing between a pointer and the object at which it points

Matlab uses various notations to indicate that the elements of a cell array are not actually matrices, but instead are pointers to matrices. In its depiction of the contents of \( p \) above, for example, it encloses all numeric matrices in square brackets and it encloses a string in single quotes. That notation is supposed to say, “pointers to” the respective objects. The distinction is a bit clearer in the following examples:

\[
\begin{align*}
&\text{} \text{>> } q = p(1,1) \\
&q = \\
&[17]
\end{align*}
\]

Here we have copied the pointer in \( p(1,1) \) into the variable \( q \), making \( q \) of type \text{cell}. Matlab indicates that the value stored in \( q \) is a pointer by enclosing 17 in square brackets. We can verify the type of \( q \) with the class function,

\[
\begin{align*}
&\text{} \text{>> } \text{class}(q) \\
&\text{ans} = \\
&\text{cell}
\end{align*}
\]

If we want access the object that is pointed at by \( p(1,1) \), instead of the pointer that is stored in \( p(1,1) \), we use curly braces, as follows:

\[
\begin{align*}
&\text{} \text{>> } r = p{1,1} \\
&r = \\
&17
\end{align*}
\]

The type of the object retrieved by the using \( p{1,1} \) is \text{double}, and it is assigned to \( r \). Therefore, \( r \) is of type \text{double}, not \text{cell}, as can also be verified with the function \text{cell},

\[
\begin{align*}
&\text{} \text{>> } \text{class}(r) \\
&\text{ans} = \\
&\text{double}
\end{align*}
\]

Finally, compare that way Matlab displays the contents of \( p(2,3) \),

\[
\begin{align*}
&\text{} \text{>> } p(2,3) \\
&\text{ans} = \\
&'Awesome!'
\end{align*}
\]

This element contains a pointer to a string. To indicate that it is a pointer to a string, and not a string, Matlab encloses the string in single quotes. When, we access the string itself, Matlab omits the quotes. Thus,

\[
\begin{align*}
&\text{} \text{>> } p{2,3} \\
&\text{ans} = \\
&'
\end{align*}
\]
Awesome!

The characters in the string are printed without quotes, as they are for the display of any string,

```matlab
>> s = 'Simply amazing!'
s =
Simply amazing!
```

Uses for cell arrays

Cell arrays are useful when it is necessary to combine sets of data of differing types and/or sizes, and it is required to use numerical indices, instead of field names, as are used in `structs`. For example, if a user is inputting a set of strings that will later be operated on, it is convenient to put pointers to each of them in a cell vector:

```matlab
n = 0;
while 1
    s = input('Name please: ');
    if isempty(s)
        break;
    else
        n = n+1;
        name_list{n} = s;
    end
end
fprintf('Here is a numbered list
');
for ii = 1:n
    fprintf('%3d: %s
', ii, name_list{ii});
end
```

Here is an example of this code in action,

```
Name please: 'Marge'
Name please: 'Homer'
Name please: 'Maggie'
Name please: 'Lisa'
Name please: 'Bart'
Name please: []
Here is a numbered list
  1: Marge
  2: Homer
  3: Maggie
  4: Lisa
  5: Bart
```
**Arithmetic and relational operators**

Neither arithmetic operators nor relational operators allow the type `cell` for their operands.

**Variable numbers of input and or output arguments**

Sometimes it is desired to write a function that can take an arbitrary number of input arguments or output arguments. We know how to write a function that can take differing numbers of arguments, but not how to write one that can take any number of arguments. Matlab provides a facility for arbitrary numbers of arguments through two special argument names: `varargin` (for “variable number of arguments in”) and `varargout` (for “variable number of arguments out”).

We will explain their uses through examples. Suppose we want a function that will take one or more numbers as input arguments and print them each on a separate line. We can write that function with the help of `varargin`, as follows:

```matlab
function print_all_args(first, varargin)
    fprintf('%d
', first);
    for ii = 1:nargin-1
        fprintf('%d
', varargin{ii});
    end
```

The special argument `varargin` must be the last dummy argument in the list. It can also be the only dummy argument. When the function is called with more input arguments than there are “normal” dummy arguments, `varargin` receives the surplus. It is a cell vector, and each element points to one argument. Inside the function it is treated as an ordinary `cell` array. Furthermore, the function `nargin`, which returns a count of the number of arguments used in the call, counts all arguments, both those that are captured by the normal dummy arguments and those that are captured by `varargin`. Here is this function in action,

```matlab
>> print_all_args(14)
14
>> print_all_args(14,15,16,17,18)
14
15
16
17
18
```
Now suppose we want a function that will take an input vector and copy the value of each of its elements into separate output arguments. We can accomplish that with \texttt{varargout}. Here is a function that does it:

\begin{verbatim}
function \ [first, varargout] = distribute(v)
first = v(1);
for ii = 1:length(v)-1
    varargout{ii} = v(ii+1);
end
\end{verbatim}

As is the rule with \texttt{varargin}, \texttt{varargout} must be the last dummy output argument in the list. Also, in analogy to \texttt{varargin}, it holds any excess output arguments after the “normal” ones. Here is a function in the expected situation, namely, the length of the input vector is equal to the number of output arguments:

\begin{verbatim}
>> \ [a,b,c] = distribute_to_args([14,15,16])
a =
     14
b =
     15
c =
     16
\end{verbatim}

If fewer output arguments are given, there is no error. The extra elements placed into \texttt{varargout} are simply ignored:

\begin{verbatim}
>> \ [a,b] = distribute_to_args([14,15,16])
a =
     14
b =
     15
\end{verbatim}

On the other hand, if there are not enough elements in \texttt{varargin} to handle all the remaining output arguments, Matlab complains and halts execution,

\begin{verbatim}
>> \ [a,b,c,d] = distribute_to_args([14,15,16])
??? Error using ==> distribute_to_args
Too many output arguments.
\end{verbatim}

\textit{Cell functions}

Matlab provides a number of functions designed to assist you in using cell arrays. Table 18 gives a partial list.
### Table 18 Cell Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cell</code></td>
<td>create an array of type <code>cell</code></td>
</tr>
<tr>
<td><code>celldisp</code></td>
<td>show all the objects pointed at by a <code>cell</code> array</td>
</tr>
<tr>
<td><code>cellfun</code></td>
<td>apply a function to all the objects pointed at by a <code>cell</code> array</td>
</tr>
<tr>
<td><code>cellplot</code></td>
<td>show a graphical depiction of the contents of a <code>cell</code> array</td>
</tr>
<tr>
<td><code>cell2struct</code></td>
<td>convert a <code>cell</code> array into a <code>struct</code> array</td>
</tr>
<tr>
<td><code>deal</code></td>
<td>copy a value into output arguments</td>
</tr>
<tr>
<td><code>iscell</code></td>
<td>returns true if argument is of type <code>cell</code></td>
</tr>
<tr>
<td><code>num2cell</code></td>
<td>convert a numeric array into a <code>cell</code> array</td>
</tr>
</tbody>
</table>
Concepts from this chapter

- Computer Science and Mathematics:
  - data type, or type
  - mixed-mode arithmetic
  - strings
    - encoded characters
    - ASCII encoding
  - structs
    - heterogeneous structure
    - element is called a field
    - indexed by field names
  - pointers

- Matlab:
  - the function class returns the type of its argument
  - conversion functions
  - numeric types
    - floating point
      - double, single
    - integer
      - int8, 16, 32, and 64
      - uint8, 16, 32, and 64
  - string
    - a vector
    - type is char
  - struct type
  - cell type
    - holds pointers
    - created with function cell
    - syntax of commands employs curly braces
    - cell array may hold pointers to arrays of differing types
Chapter 9
Recursion

This chapter introduces a fundamental new method of programming called “recursion”. Recursion is a powerful idea that makes it possible to solve some problems easily that would otherwise be quite difficult. Recursion cannot be used in all programming languages, but it is supported by most modern languages, including Matlab.

9.1 Activation records

In Chapter 7, we learned that Matlab assigns a unique memory location for each variable and that it maintains separate sets of variables for the command window and for each function that is called during the running of a program. The local variables within the scope of one function are distinct from those within the scope of another function and from those of the command window. Two such variables can have the same name, as long as each belongs to a different scope. In this chapter we will learn that Matlab accomplishes this feat by maintaining one area of the computer memory to hold variables used in the command-window and a separate area for each function. The term used in computer science for one such memory area is activation record. An activation record is simply an area of memory in which all information is stored that is needed during the execution of a function or during the execution of a command in the command window. In the design of some early languages, including, notably FORTRAN, which is still a popular language today (and was, incidentally, the language in which the original version of Matlab was written), each activation record was set aside for a specific function before the program ran and was available for the exclusive use of that function whenever it was called. As you will see, Matlab uses a different scheme for allocating areas of memory to its functions, but for now, we will use that model.

Suppose we define three Matlab functions, tax, high, and low, as shown in Figure 5. Suppose we then call tax as follows in the command window:

```matlab
>> wages = 5000;
>> due = tax(wages)
due =
    625
```

The memory allocation might look something like that shown in Figure 6. The first column of Figure 6 gives the name of the function whose activation record, abbreviated “AR”, is shown. The allocation of memory at the bottom (in light blue) is for the command window, in which two variables are defined—small and due. Since there is always a command window and it always has an activation record to hold the variables defined in it, we have not labeled it. Stacked on top of the memory allocated for the command window is the activation record allocated for the function tax. It contains space for the local variables, income and y; the activation record for the function high is next. Finally, at the top is the activation record for low.
Stacked on top of the memory allocated for the command window is the activation record allocated for the function `tax`. It contains space for the local variables, `income` and `y`;

```matlab
function y = tax(income)
if income > 100000
    y = high(income-45000);
elseif income > 2500
    y = low(income-2500);
else
    y = 0;
end
```

```matlab
function low_tax=low(subject)
    rate = 0.25;
    max_tax = 18750;
    raw_tax = rate*subject;
    if raw_tax > max_tax
        low_tax = max_tax;
    else
        low_tax = raw_tax;
    end
end
```

```matlab
function high_tax=high(subject)
    rate = 0.33;
    min_tax = 18750;
    raw_tax = rate*subject;
    if raw_tax < min_tax
        high_tax = min_tax;
    else
        high_tax = raw_tax;
    end
end
```
the activation record for the function \texttt{high} is next. Finally, at the top is the activation record for \texttt{low}. We have depicted the first allocation to be at the bottom and the last at the top, but the spatial ordering here is not important. The important thing is that each function has its own area. We will see later that the idea of “stacking” one activation record on top of the other makes sense. That is the reason for showing them this way in the figure.

Figure 7 shows a \textit{snapshot} of the memory during execution. A snapshot is a list of the values in all memory locations for a particular point \textit{in time} during execution. The time point always falls between the executions of two commands. In this case, execution, which started in the command window with the function call—\texttt{due = tax(small)}—has been suspended during execution of the function \texttt{low} at the point in time between the completion of the statement—\texttt{raw_tax = rate*subject;}; and the beginning of the statement—\texttt{if raw_tax > max_tax}. The memory locations containing the three dashes (---) have not yet been given values. Thus, for example, since the \textit{if}-statement has not yet begun, the statement—\texttt{low_tax = max_tax}; has not yet executed, so no value has yet been assigned to \texttt{low_tax}. Thus, three dashes (---) appear in its memory location in place of a value.

\section*{9.2 Breakpoints}

It is easy to make your own snapshot by setting a \textit{breakpoint} in one of your functions. A breakpoint is a point in the text of a function at the beginning of one of its command that is selected by the user as a stopping point. Figure 8 shows an example. A breakpoint is set by clicking on the dash (-) just to the left of the statement between the number of the statement and the statement itself. The setting of the breakpoint is indicated by a red circle. A breakpoint can be removed simply by clicking the mouse on that red circle. Additional breakpoints can be set by clicking next to other commands. When the function is executed, execution will be suspended just as the statement \texttt{if raw_tax > max_tax} is about to begin. If we call \texttt{tax}, as we did in the example above, then after the program reaches the breakpoint, the command window will look like this:

\begin{verbatim}
>> wages = 5000;
>> due = tax(wages)
K>>
\end{verbatim}

The prompt K>>, instead of >>, indicates that a function has reached a breakpoint and that execution is suspended. At this point, if we give the command \texttt{who}, Matlab will show us the local variables that have been assigned a value in the function that has reached the breakpoint, instead of the variables that have been assigned a value in the command window:

\begin{verbatim}
K>> who
\end{verbatim}
Your variables are:

max_tax  rate     raw_tax  subject

Note that these variables are among those listed in the activation record of the function low in Figure 7. From this example, we learn that the command who lists the local variables of the currently active function that have been assigned a value. The command whos behaves in the same way. The Matlab terminology for this set of variables is workspace. Thus, the workspace is the set of variables in the current activation record that have been assigned a value. The letter “w” in who and whos comes from that term. Heretofore, when we used either of these commands, they displayed the variables of the command window, but now we see that they can also be used to display local variables as well. (These two commands can be issued within a function as well, and, as expected, they will display the assigned local variables of the function.) During suspension, these variables are active in every sense. If we type their names in the command window, we can see their values, and, if we wish, we can even alter their values before we resume execution. Suppose, for example, we wanted for some reason to see what would happen if the value of raw_tax were negative. We could try some random negative value, say, −17, by simply assigning raw_tax that value in the command window:
K>> raw_tax = -17;

After this command, the memory would appear as in Figure 9. All is the same as Figure 7 except for the new value of \texttt{raw\_tax}.

After we have inspected and/or changed these variables, we can resume execution. There are two ways to do that: (1) by giving the command \texttt{dbcont} in the command window, and (2) by clicking an icon in the editing window. There are several icons that are useful when using breakpoints, and the most important ones are highlighted in Figure 10. Because breakpoints are used by programmers when they are trying to find errors and eliminate bugs, the breakpoint system is considered to be part of the \texttt{debugging} system. Debugging is simply the process of identifying errors in programs and correcting them. Often during debugging, it is helpful not only to suspend execution with a breakpoint, but to execute one step at a time, checking the values of variables after each step. That is easy to do by clicking on the single-step icon labeled (b) in Figure 10.

Matlab has an excellent debugging system, which has other features as well (try \texttt{help debug}), but the most important things to learn are that (a) you can cause execution to be suspended by setting breakpoints, (b) you can inspect or change local variables while execution is suspended, and (c) you can execute one step at a time. The “End debugging” icon stops execution and returns control to the command window, but the breakpoints are not removed. They will suspend execution each time the function reaches one of them.
Let’s return now to our suspended example in which we have forced \texttt{raw\_tax}, to have the value -17. If we resume execution with \texttt{dbcont} or by clicking the icon in Figure 10(c), we will see this result:

\begin{verbatim}
K>> dbcont
due =
   -17
\end{verbatim}

It can be seen that the change we made while execution was suspended caused the calculation to give a strange, but predictable result.

### 9.3 Dynamic allocation

As we mentioned earlier, the scheme used by Matlab to allocate memory storage for functions is somewhat different from that described above. The scheme above is called \textit{static allocation}. Most modern languages, including Matlab, use \textit{dynamic allocation}. The difference is that with static allocation the position in memory of each activation record is determined before the program runs and remains the same throughout execution, while with dynamic allocation the position is determined during execution, and it may change during execution. Static allocation allows for slightly faster execution times, whereas dynamic allocation requires slightly less memory usage in most cases, but the main advantage to dynamic allocation is that it makes recursion possible. Before we see the connection with recursion, let’s learn how dynamic allocation works.

Returning to the example above, let’s remove the breakpoint from the function \texttt{low} and insert one instead in the function \texttt{high} before the statement—\texttt{if raw\_tax < min\_tax}. If we then issue the following commands in the command window:

\begin{verbatim}
>> wages = 200000;
>> due = tax(wages)
\end{verbatim}

a snapshot of the memory will look like that shown in Figure 11. The situation is similar to that of Figure 7 because four activation records are allocated while only three activation records are used, one for the command window and one each for two functions. In Figure 7 the functions whose activation records were used were \texttt{tax} and \texttt{low}; in Figure 11 the functions are \texttt{tax} and \texttt{high}. Also, in both cases, when execution is resumed, only three activation records will be used for the entire execution. The space set aside for the fourth activation record is

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figure_11.png}
\caption{Snapshot of memory during execution.}
\end{figure}
never used. This situation is unavoidable when static allocation is employed because with static allocation activation records are allocated before execution begins and there is no way to know which functions will be used before execution begins and inputs are received from the user (in this case the argument to \texttt{tax}).

This problem of wasted memory is solved by means of dynamic allocation. With dynamic allocation, an activation record is allocated \textit{only when a function is called}. Let’s follow the function calls and watch how activations are allocated dynamically for the current example as execution proceeds from just before the call of \texttt{tax} in the command window through the suspension of execution at the breakpoint in \texttt{high}. For simplicity, we’ll label just the activation records without labeling the variables within them.

![Diagram of activation records](image)

No activation record is allocated for the function \texttt{low} because \texttt{low} has not been called, so no space is wasted. Let’s continue execution now and follow the allocation of activation records through to the end:

![Diagram of activation records](image)

The activation record for the command window is always there, but the other ones come and go as their functions are called and return. Now we can see why it makes sense to put the first activation record on the bottom and the most recent activation on the top. That way it appears that the records are in a stack with new records being put on the top of the old ones and, when a function returns, the most recently added record is taken from the top, just the way objects are added to, and removed from, a physical stack. That is the way computer scientists like to think about this arrangement. In fact when memory is allocated dynamically in this way, the collection of activation records is called the
Chapter 9  Recursion

**activation-record stack.** If we were to watch the behavior of the activation-record stack that results from the commands,

```matlab
>> wages = 5000;
>> due = tax(wages)
```

due =

```
625
```

we would see no difference except at the third stage, in which the activation record for `low` would appear in place of that for `high`. Thus, the same area of memory would be used for two different functions at two different times.

### 9.4 Recursive definitions

We are now almost ready to see recursion in action. The classic way to introduce recursion is through the factorial function, and that is just the way we will do it. We considered that function in Chapter 6, where the factorial function \( f(n) \), which is usually written in mathematics as \( n! \), was defined as follows:

\[
\begin{align*}
    f(n) &= 1 \times 2 \times 3 \times \ldots \times (n-2) \times (n-1) \times n \\
    f(1) &= 1! = 1 \\
    f(2) &= 2! = 1 \times 2 = 2 \\
    f(3) &= 3! = 1 \times 2 \times 3 = 6 \\
    f(4) &= 4! = 1 \times 2 \times 3 \times 4 = 24 \\
    f(5) &= 5! = 1 \times 2 \times 3 \times 4 \times 5 = 120
\end{align*}
\]

In Section 6.4.5, we gave a loop to calculate the factorial. In this chapter we will show how to calculate it using recursion.

To get started, we first note that there is another way to define the factorial function:

\[
f(n) = \begin{cases} 
    1, & \text{for } n = 1 \\
    f(n-1) \times n, & \text{otherwise}
\end{cases}
\]  

(4.8)

We can see that the second part of this definition makes sense by looking, for example, at \( f(4) \). By carefully grouping the four factors that make it up, we see the following:
Eq. (4.8) is a **recursive definition**. A recursive definition is one that defines a concept in terms of the concept itself. That would seem to be nonsense because the definition seems to require that the object already be defined!

Looking more closely at Eq. (4.8), we see that it is only the second part that presents the potential problem because it is only in that part that the function \( f \) shows up in its own definition. It seems at first to be circular reasoning, but it is not. The crucial aspect of the second part that avoids circular reasoning is that \( f(n) \) is not defined in terms of \( f(n) \), but in terms of \( f(n-1) \). Thus, the definition of \( f(n) \) requires only that \( f(n-1) \) be previously defined, and that is not circular reasoning. It is more like spiral reasoning because, instead of going round and round in a circle, we are spiraling down from \( f(n) \) to \( f(n-1) \), and then to \( f(n-2) \), etc., ever closer and closer to the case \( f(1) \), where at last we will reach the first part of the definition in Eq. (4.8) and stop.

If we use Eq. (4.8) to determine the value of \( f(3) \), for example, we would proceed this way:

\[
f(3) = f(2) \times 3
\]  
(4.10)

We cannot carry out the multiplication until we know the value of \( f(2) \). So we make a note to carry out Eq. (4.10) after we determine the value of \( f(2) \), and we go back to Eq. (4.8) to calculate \( f(2) \):

\[
f(2) = f(1) \times 2
\]  
(4.11)

We cannot carry out the multiplication until we know the value of \( f(1) \). So we make a note to carry out Eq. (4.11) after we determine the value of \( f(1) \), and we go back to Eq. (4.8) to calculate \( f(1) \):

\[
f(1) = 1
\]  
(4.12)

**Finally**, we have a case that does not require a further invocation of Eq. (4.8)! We have arrived at the part of the definition that is not recursive. Every recursive definition *must* include such a non-recursive part. We call the non-recursive part the **base case**. The recursive part of the definition is called, appropriately enough, the **recursive case**.
Chapter 9  Recursion

Now that we have spiraled down to the base case, we look at the notes that we made to see what remains to be done. Following our last note, we carry out the operation in Eq. (4.11) to get $f(2) = 2$; then, following the note that we made before the last one, we carry out the operation in Eq. (4.10) to get $f(3) = 6$. There are no more notes remaining, so we are done.

We can use our recursive definition to help us write Matlab code to calculate $f(n)$. Suppose we create the four functions, $f_1$, $f_2$, $f_3$, and $f_4$, shown in Figure 13. Each function has the same form, and the form mirrors the definition in Eq. (4.8). We can use these functions to calculate $f(1)$, $f(2)$, $f(3)$, or $f(4)$. We always call $f_1$ from the command window, and we give it the number whose factorial is required. Suppose, for example, we want to calculate $f(3)$. We give the call, $f_1(3)$ in the command window. The important steps in the resulting execution are highlighted in Figure 13 and are itemized in Figure 12. Figure 14 depicts the state of the activation-record stack at each of these steps ("CW" means Command Window).

It is a good idea for the reader to take the time to follow these steps through in detail to get a clear idea of what is happening. Each function has a base case and a recursive case. When it is called it makes a decision based on the value of its dummy argument $n$: If $n$ is equal to 1, it uses the base case, for which the answer is 1, and it returns 1. If $n$ is greater than 1, the function uses the recursive case: It calls the next function on the list with the argument $n-1$.

There is a problem with using these four functions to calculate $f(n)$. The problem arises when we try to calculate $f(5)$. We do that by giving the command, $f_1(5)$. There is no problem in $f_1$ which issues the call, $f_2(4)$, and is there no problem in $f_2$, which issues the call $f_3(3)$, or $f_3$, which issues the call $f_4(2)$. The problem occurs when $f_4$ issues the call $f_5(1)$. There is no $f_5$! Matlab issues the error: ??? Undefined function or variable 'f5'. We could of course define an $f_5$, and an $f_6$, etc., but that is a lot of trouble, and we do not need to. Matlab provides a better way: recursion.
1. \(f1\)’s dummy argument \(n = 3\).
2. \(n == 1\) is false.
3. \(f2\) is called with argument 2.
4. \(f2\)’s dummy argument \(n = 2\).
5. \(n == 1\) is false.
6. \(f3\) is called with argument 1.
7. \(f3\)’s dummy argument \(n = 1\).
8. \(n == 1\) is true.
9. \(nfact\) is set to 1
10. \(f3\) returns the value 1
11. The value 1 returned by \(f3\) is multiplied times the value 2 of the local variable \(n\) in \(f2\) (see step 4 above). The resulting value of 2 is assigned to \(nfact\).
12. \(f2\) returns 2.
13. The value 2 returned by \(f2\) is multiplied times the value 3 of the local variable \(n\) in \(f1\) (see step 1 above). The resulting value of 6 is assigned to \(nfact\).
14. \(f2\) returns 6.
15. The command window prints the value 6 returned by \(f1\).

Figure 12. Description of steps in Figure 13.

Figure 13. Functions for calculating the factorial with numbered steps in their execution.

Figure 14. Activation record stack at each of the numbered steps in Figure 12.
9.5 Recursive functions

The better way is to write a function that calls itself! A function that calls itself is a recursive function. The call itself is referred to as a recursive function call, and the use of a recursive call in a function is called recursion.

9.5.1 The mechanics of recursion

Let’s see how recursion works. The function shown in Figure 15 is a recursive function that calculates the factorial. It has the same form as the functions f1, f2, f3, and f4 above, except that carries out the recursive part of the definition in Eq. (4.8) by calling itself. That might at first seem to be impossible because a function that is already active would seem to “used up” and unavailable to be called again. That is exactly the way it is for a programming language that uses static allocation of activation records, but with dynamic allocation, more than one copy of the function can be active at a time because each activation has its own activation record. Each activation record holds its own separate set of local variables, so there is no confusion from one activation to another. It is as if a separate copy of the function is generated on the fly with each call: f1, f2, f3, f4, f5, f6, and on and on. If we issue the call f(3) in the command window and refer to the first, second, and third copies of the function that are generated by the recursive calls as f1, f2, and f3, respectively, then a trace of the execution looks exactly the same as that shown in Figure 12 and Figure 14. The difference is that we can also issue the calls f(5), f(10), and higher—much higher:

```matlab
def function nfact = f(n)
if n == 1, nfact = 1;
else nfact = f(n-1)*n;
end
```

Figure 15. Recursive function to calculate factorial.

With this example, the activation-record stack gets very tall. When the base case is reached, there are 71 activation records on it: 70 corresponding to calls of f and one for the command window. The stack is tall, but when referring to the degree of recursion, it is customary to say that the recursion level is deep. In this case the depth of the recursion is 70 (or arguably 69, since the record for the first call is not recursive).

For a given computer system, there is always a limit to how tall the stack can get because every activation record occupies memory space, and the memory space on any computer is finite. Matlab itself puts a limit to the depth of recursion, which defaults to 500. We can demonstrate that limit as follows:

```matlab>> f(0)
??? Maximum recursion limit of 500 reached. Use
set(0,'RecursionLimit',N)
```
to change the limit. Be aware that exceeding your available stack space can crash MATLAB and/or your computer.

Error in ==> C:\Documents and Settings\Matlab\f.m
On line 3  ==> else nfact = f(n-1)*n;

What happened is that we have bypassed the base case. Since we started with $n = 0$, it will never be the case that $n$ equals 1, so $n = 1$ will never be true. Thus, the command $f(0)$ results in infinite recursion! There is, mathematically, no limit to the depth of the recursion. However, Matlab imposes a limit: As we continue subtracting 1 from $n$ and issuing recursive calls, we are also adding activation records to the stack. Matlab keeps a tally of the number of records that are simultaneously on the stack. When that number reaches 500, Matlab refuses to add a 501st record, and it issues an error message. This problem could be avoided by including a statement at the beginning of the function that checks the argument. If the argument is less than 1, it would issue an error message and halt. The resulting function would be more robust than the simple version in Figure 15. It would not calculate $f(0)$, but it would end gracefully with an error message explaining the user’s error, instead of causing the stack to fill to overflowing and Matlab to issue its own error. Such functions, which check for erroneous input, are far more helpful to users when things go wrong than those that do not.

It is easy to write recursive functions when we are given recursive definitions like that given in Eq. (4.8). Here is another example, which serves no purpose except to show that virtually any recursive definition can be implemented with a recursive function,

$$g(x) = \begin{cases} 
-0.5, & \text{for } x \leq 0 \\
\tan^{-1}((\sin x + 4)g(x-0.7)), & \text{otherwise}
\end{cases} \quad (4.13)$$

A recursive function that implements Eq. (4.13) is given in Figure 16. Note that the base case includes every negative value of $x$. Unlike the function in Figure 15, it is not possible to bypass this base case for this function. Thus, there is no need for it to check for erroneous input.

It is unfortunately easy to write functions that include cases of infinite recursion. If, for example, in Eq. (4.13) we were to change the base case to $g(x) = -0.5$, for $x = 0$ (i.e., equal to zero instead of less than or equal to zero) and make the equivalent change in Figure 16, then we would run into infinite recursion for almost any call. The call $g(1)$, for example would spawn the recursive call, $g(0.3)$, which in turn would spawn the recursive call, $g(-0.4)$, followed by a sequence of calls all of which

```
function a = g(x)
if x < 0
    a = -0.5;
else
    a = atan((sin(x)+4)*g(x-0.7));
end
```

Figure 16. A recursive function.
have negative arguments. The arguments would never hit zero. Matlab would halt the function with an error when the stack reached 501.

It should be clear that the recursive call should move the function closer to the base case. In the factorial problem of Figure 15, the argument moves one closer to \( n = 1 \) each time. In Figure 16, \( x \) moves closer to being negative by 0.7 each time. A common error is to write a recursive function that results in infinite recursion because the argument moves further from the base case. That could happen in Figure 16, for example, if the recursive call had been, \( g(x + 0.7) \), instead of \( g(x - 0.7) \).

9.5.2 Multiple recursive calls per activation

The functions in Figure 15 and Figure 16 each makes one recursive call per activation. There are many examples of functions that make two or more recursive calls per activation. We will look at two of them: the Fibonacci series and the derivatives of a function.

**Example 1: The Fibonacci Series**

A famous series that can be generated by a function that makes two recursive calls per activation is called the Fibonacci series, named after the 13\(^{th}\) century mathematician who invented it. Fibonacci posed this problem:

\[
\text{Suppose a pair of rabbits, one male and one female, is born on January 1. Suppose further that this pair of rabbits gives birth to a new male and female on the last day of their second month of life, their third month of life, their fourth month, etc. forever (mathematicians do not care about reality). Finally, suppose that all new pairs of rabbits do the same. How many pairs of rabbits would be alive on the first day of the Nth month (and who but a mathematician would care)?}
\]

To help us understand the problem, let’s consider the first few months. Figure 17 is a schematic representation of the development of the rabbit population from January 1 through July 31. Each solid horizontal line is the life-line of a pair of rabbits; the first pair has the longest line. Each vertical line represents the birth of a new pair.

The number of pairs of rabbits alive at the beginning of a month is equal to the number of solid horizontal lines that intersect the vertical dotted line for that day. By counting the intersections we can see the following progression in the number of pairs:

\[
1, \ 1, \ 2, \ 3, \ 5, \ 8, \ 13
\]

These numbers are a consequence of the rules Fibonacci gave for the births. To see how these rules produce these numbers, we can look the changes, month by month:
**January 1 and February 1:** The first two months are simple. Since the first pair requires two months to produce offspring, there is one pair on January 1 and still just one pair on February 1.

**March 1:** On the last day of February, the first pair produces its first offspring, so there are two pairs on March 1.

**April 1:** April is the first interesting month. It is interesting because some rabbits reproduce and some do not. Only the rabbits that are at least two months old reproduce, so only the pairs that were alive on February 1 will produce a new pair. There was only one pair alive then, so one new pair is added to the number alive March 1 to bring the total to three. Thus, the number of pairs on April 1 is equal to the number of pairs on February 1 plus the number of pairs on March 1.

**May 1:** May is interesting as well for the same reason. Again only those rabbits that are at least two months old will reproduce, so only the pairs that were alive on March 1 will produce a new pair. There were two pairs alive then, so two new pairs are added to the number that were alive on April 1. Thus, the number of pairs on May 1 is equal to the number of pairs on March 1 plus the number of pairs on April 1.

By now, we can see a pattern: To determine the number of pairs alive at the beginning of month \( N \), we add the number of pairs alive at the beginning of month \( N - 2 \) to the number of pairs alive at the beginning of month \( N - 1 \). Letting \( F(N) \) stand for the number of pairs of rabbits alive on the first day of the \( N \text{th} \) month, we have
A recursive function that implements Eq. (4.14) is shown in Figure 18.

\[
F(N) = F(N - 2) + F(N - 1)
\]  

(4.14)

To check things out let’s use the function in Figure 18 to calculate the number of pairs for a month whose answer we already know. We’ll try May:

\[
\text{>> fibonacci(5)}
\]

??? Maximum recursion limit of 500 reached. Use set(0,'RecursionLimit',N) to change the limit. Be aware that exceeding your available stack space can crash MATLAB and/or your computer.

Error in ==> C:\Documents and Settings\Matlab\fibonacci.m
On line 2 ==> pairs = fibonacci(month-2)+fibonacci(month-1);

Can you guess what happened? We forgot to include a base case! Calls of fibonacci were made with \text{month} = 5, 3, 1, -1, -3, -5, ..., -993. At this point there were 500 activation records on the stack. The next call caused the error. Our error came from Eq. (4.14). A corrected version of Eq. (4.14) is given in Eq. (4.15).

\[
F(N) = \begin{cases} 
1, & \text{for } N = 1 \\
1, & \text{for } N = 2 \\
F(N - 2) + F(N - 1), & \text{otherwise}
\end{cases}
\]  

(4.15)

We now have a base case. In fact we have two of them, one for \( N = 1 \) and one for \( N = 2 \). A corrected version of \text{fibonacci}, implementing Eq. (4.15) is shown in Figure 19.

\[
\text{function} \ \text{pairs} = \text{fibonacci(month)} \\
\text{if} \ \text{month} == 1 \\
\text{pairs} = 1; \\
\text{elseif} \ \text{month} == 2 \\
\text{pairs} = 1; \\
\text{else} \\
\text{pairs} = \text{fibonacci(month-2)} + \text{fibonacci(month-1)};
\]

Figure 18. Recursive Fibonacci function.

This time our testing is successful,
Example 2: Derivatives of a function

Another example of recursion involving two recursive calls per activation arises when we estimate numerically the 1\textsuperscript{st}, 2\textsuperscript{nd}, 3\textsuperscript{rd}, etc., derivatives of a function. The definition of a derivative is, from calculus,

\[ f^{(1)}(x) = \lim_{{\Delta x \to 0}} \frac{f(x + \Delta x) - f(x)}{\Delta x} \]  \hspace{1cm} (4.16)

We can estimate numerically the derivative of a function at \( x \) by using the following formula:

\[ f^{(1)}(x) = \frac{f(x + \frac{1}{2} \Delta x) - f(x - \frac{1}{2} \Delta x)}{\Delta x} \]  \hspace{1cm} (4.17)

where we assume that \( \Delta x \) is small enough to produce a good approximation to the limit as \( \Delta x \) approaches zero. The second derivative is defined as follows:

\[ f^{(2)}(x) = \lim_{{\Delta x \to 0}} \frac{f^{(1)}(x + \Delta x) - f^{(1)}(x)}{\Delta x} \]  \hspace{1cm} (4.18)

and the corresponding numerical approximation is

\[ f^{(2)}(x) = \frac{f^{(1)}(x + \frac{1}{2} \Delta x / 2) - f^{(1)}(x - \frac{1}{2} \Delta x / 2)}{\Delta x / 2} \]  \hspace{1cm} (4.19)

Both of these definitions involve recursion. We can give complete recursive definitions of all derivatives and their approximations as follows:
Chapter 9  Recursion

\[
f^{(n)}(x) = \begin{cases} 
    f(x), & \text{for } n = 0 \\
    \lim_{\Delta x \to 0} \frac{f^{(n-1)}(x + \Delta x) - f^{(n-1)}(x)}{\Delta x}, & \text{otherwise}
\end{cases} \tag{4.20}
\]

where we have added the concept of the 0th derivative, \( f^{(0)}(x) = f(x) \). Similarly, the recursive definition for the numerical approximations to these derivatives is

\[
f^{(n)}(x) = \begin{cases} 
    f(x), & \text{for } n = 0 \\
    \frac{f^{(n-1)}(x + \frac{1}{2} \Delta x / n) - f^{(n-1)}(x - \frac{1}{2} \Delta x / n)}{\Delta x / n}, & \text{otherwise}
\end{cases} \tag{4.21}
\]

Eq. (4.21) can be implemented easily by means of recursion. An implementation is given in Figure 20. The base case is implemented by lines 7-8 and the recursive case by lines 10-12. When the function is called, the first argument must be a function handle, which is obtained by means of the \(@\) operator. For example, if we wish to calculate the 3rd derivative of the sine evaluated at 0.4, we would give the following command:

\[
\text{>> derivative(@sin,0.4,3)}
\]

\[
\text{ans = -0.92106}
\]
This answer can be easily checked by comparing with the 3\textsuperscript{rd} derivative of the sine, which is simply negative of the cosine:

\[
\text{>> } -\cos(0.4) \\
\text{ans } = \\
-0.92106
\]

There are many, many other examples of recursive problems and recursive functions. Two important applications that involve many recursive problems are searching and sorting. The next chapter introduces these two applications and gives examples of recursion in both.
**Concepts from this chapter**

- Computer Science and Mathematics:
  - activation record
  - snapshot
  - breakpoint
  - debugging
  - static allocation
  - dynamic allocation
  - activation-record stack
  - recursive definition
  - base case
  - recursive case
  - recursive function
  - recursive function call
  - recursion

- Matlab:
  - breakpoint
  - debugging system
  - workspace
Chapter 10
Searching and Sorting

In this chapter we will look at two closely related problems: searching and sorting. Both problems appear in applications involving databases, and without efficient solutions to these problems databases would be virtually useless. They would be useless because databases that are large enough to hold the information we need would be too large to use. They would be too large to use because it would take too long to find what we are looking for inside them. There is a large variety of specific applications of searching and sorting, but the most basic ones involve locating an element in a vector whose value is equal to a target value. We will use that simple problem as our driving example throughout this chapter.

10.1 Searching

Searching is important in many areas of computer science. Many large computer systems include databases that must be searched when an inquiry is made. For example, when you phone a credit-card company to inquire about your bill, you will be asked for your card number. In a few seconds the company representative will be looking at your data. Your data and that of tens of millions of other customers are all stored in a database. Your information is found by searching among all those millions of card numbers. The target of the search is your unique number. Furthermore, every day, millions of transactions, each involving one of those unique numbers take place, and each one requires that a target number be located in a list of all card numbers. In each of these examples, in the calculation of account balances, and many other cases, the first task it to find the account with the target number, and it is accomplished by a searching algorithm.

10.1.1 Sequential search

The simplest search method is the **sequential search** method, also known as the **linear search** method. In this method the target value that is being sought is compared with the first member of the database, then with the second, etc., until either the number is found, or the end of the list is reached, whichever comes first. Instead of dealing with a real database, let’s work on the simplest version of the search problem. We are given a vector of numbers, for example \([45 \ 23 \ 17 \ 12 \ -2 \ 100 \ 34]\), and we are given a target value, for example 17. Our task is to search for that target value in the vector. If we find it, then the answer we give is the index of the element that we found, which in this case would be 3.

Suppose instead, that we were given the target value 99. In that case, we would not find it. To indicate that it is not on the list, we return an impossible index, say, -1, as a **flag** indicating that the search failed. That flag is crucial to the search algorithm because there must be some means of informing the searcher that the target is not there. The term “flag” is a special term in computer science, and it is used in many contexts. A flag is a value
indicating a special condition. In this case the condition is “target not found”, or “search failure”. In other situations it might mean “modem is ready to receive data” (i.e., good news) or “modem is not ready to receive data” or “buffer empty” or “disk full”, etc. A given flag may also have many possible meanings depending on which of a set of meaningful values it has. In the present case, in which we are searching for a target, a positive index can be considered to be a flag that means “target found” or “search successful”, and it conveys further information as well—the index of the element in which the target was found. Thus, when we returned 3 above, it meant “The search was successful, and the target was found at index 3”.

The function, `sequential_search`, below carries out the sequential search:

```matlab
function index = sequential_search(vector, target)
%SEQUENTIAL_SEARCH  sequential search
%   SEQUENTIAL_SEARCH(VECTOR,TARGET) returns smallest index
%   for which TARGET == VECTOR(index), or returns -1, if
%   TARGET is not found in VECTOR.

Found = 0; % Assume the target is not in vector
first = 1;
last  = length(vector);

for n = first:last
    if target == vector(n)
        Found = 1; % We found it..
        break;     % ..so we quit looking for it!
    end
end

index = n;
if ~Found
    index = -1;
end
```

As promised, this function returns –1, as a flag indicating failure, where failure means that the target value is not equal to any of the elements in the vector. It is customary in computer science to refer to vector of numbers as a list, and if one of the elements of the list has a value equal to the target, it is customary to say that the target is on the list or the target is in the list. If the target is on the list, then `sequential_search` returns the first position at which it finds the target. So, for example, suppose the function is applied to the list `[45 23 17 17 -2 100 34]`. In that case, the number 17 would be found at index 3 after the target had been compared with three numbers: 45, 23, and 17 (the first 17, that is). When searching for a target that is not on the list, 82, for example, the function will have to compare the target with every number on the list, requiring 7 comparisons. These few comparisons will not take long, but if there are a million numbers on the list, time can become a significant factor. If the target is not on the list, or if it is last on the list, a million comparisons are required. Numbers close to the beginning will be found quickly, numbers toward the end will take longer, but if all elements are
equally probable, the mean number of comparisons required will be about half the length of the list: in this case, half a million. Faster searching methods are available, but only if the list has been sorted before the search is undertaken. For an unsorted list, the sequential search is the only possible choice.

10.1.2 Binary search

If a list has been sorted before the search is undertaken, searches can be carried out much more quickly. The search method that should be used in this case is the binary search. It is more complicated than the sequential search method, but the complication is well worth it: A binary search requires no more than 40 comparisons to find a number on a list of one million numbers, whereas a sequential search can require a million of them.

10.1.2.1 Recursive implementation of the binary search

If you have ever looked up a number in a phone book, the recursive approach to the binary search is one that you already know about. You don’t look at every name sequentially, as in the sequential search algorithm. Instead, you look at a page near the middle of the phone book. You look at one name on that page and note whether the target name that you are searching for comes before or after it. If the target name comes before the name on that page, then you have eliminated about half the names in the book. Then, you go to a second place in the phone book closer to the beginning and repeat the process.

The “process” that you are repeating is the process of finding a name in a list by looking near the middle and eliminating about half the names in the list. Each time you are applying the process to a smaller list, but otherwise the process is the same. Since the process includes the application of itself (to a smaller list), it is recursive. Near the end of the recursive process, you will probably abandon the approach of eliminating sections of the book and just scan through the names on a page or two, but the major work was done when you eliminated large chunks of the book at the beginning with the first few recursive steps.

When you use this approach you are carrying out your own recursive approximation of the binary search, and that is the version of binary search that we will consider first. In Figure 21 a recursive function is shown that implements the binary search algorithm. Note that there are four arguments. Argument 1 is the vector to be searched, and that argument is assumed to be sorted in ascending order. Argument 2 is the target that we are looking for. Arguments 3 and 4 give the beginning and end, respectively of the range of indices over which we wish to search. Most of the time we will set first to 1 and last to the length of vector, but sometimes we may wish to search only some limited range. Including arguments 3 and 4 provides this option. So for example, if we wish to search all of vector, we would give the call,

```plaintext
>> binary_search_recursive(vector, target, 1, length(vector))
```
but, if we wish to search only within vector(45:56), we would give the call,

```plaintext
>> binary_search_recursive(vector, target, 45, 56)
```

We will see that allowing the range to be specified is also crucial to the recursion. The function works as follows:

- **Line 7**: The middle of the range is determined. If `first` is odd and `last` is even, or vice versa, then that average will not be an integer. Since `mid` must be an integer in order to be an index, the `fix()` function is used to discard the fractional part (which is always 0.5), if there is one.
- **Lines 8-9**: If `first` and `last` are out of order, then the range is empty, so `target` cannot possibly be in the range. Therefore failure is flagged by returning -1.
- **Lines 10-11**: The value of `target` is compared to the element at the middle of the range. If the target and the element are equal, then the search is successful. Therefore the middle index is returned.
- **Line 12**: If `target` is less than that middle number, then the middle element and all the elements that come after the middle element can be eliminated. The search can now be confined to the range of elements that come before the middle one.
- **Line 13**: A recursive call is made to search only the range of elements that come before the middle. Whatever index is received from this recursive call is returned.
- **Line 14**: If this statement is reached, then `target` must be greater than the middle element. That means that the middle element and all the elements that
come before the middle element are eliminated. The search can now be confined
to the range of elements that come after the middle one.

- **Line 15:** A recursive call is made to search only the range of elements that come
  after the middle. Whatever index is received from this recursive call is returned.

It is important to note that, during the first call of this function, about half (actually just
over half) the numbers are eliminated before the first recursive call is made. This search
algorithm is in fact named “binary” search because the list is repeatedly divided into two
parts. (Actually it is divided into two big parts and one very small one. The very small
part is the single element in the middle.) Searching a subrange of the list would not be
possible without the provision of the 3rd and 4th arguments that allow us to restrict the
range that we search. Thus, these two arguments are crucial to the recursion. The
elimination of about half the remaining elements happens again on the second recursive
call, and on the third, and on each successive recursive call. Thus, each call cuts the size
of the problem roughly in half.

This is a powerful approach to searching because relatively few divisions are required to
trim a large list down to one number. If, for example, we search a list of one million
elements for a target that is equal to the last number on the list, the successive numbers of
elements to be searched with this algorithm are 1000000, 500000, 250000, 125000,
62500, 31250, 15625, 7812, 3906, 1953, 976, 488, 244, 122, 61, 30, 15, 7, 3, 1. This can
be shown to be the worst case. Thus, if the number is in the list, even in the worst case
there will be only 20 calls of `binary_search_recursive` required to find a target
number within in a list of 1 million numbers. It takes no more than 21 calls to hit failure
when the target is not there. In either case (i.e., the target is on the list or not on the list),
during those 20 or 21 calls, no more than 40 comparisons will be made between `target`
and the elements of `vector`. This is a very small number compared to the million
comparisons required by the sequential search: a factor of 25,000 fewer!

The strategy of dividing a problem into smaller problems and then attacking the smaller
problems recursively is a standard approach in the field of `algorithms`, which is the field
of computer science that deals with the efficiency of computer programs. This strategy is
called `divide-and-conquer`\(^9\). We will look more closely at the idea of efficiency in a later
section of this chapter.

It is informative to follow the algorithm as it works its way through a few specific
searches. Suppose we are searching for targets in a vector named \texttt{A}, which contains 11
elements:

\[
\texttt{A} = [2 \ 7 \ 17 \ 17 \ 24 \ 43 \ 74 \ 77 \ 80 \ 88 \ 97]
\]

In our first search we will look for the number 88:

---

\(^9\) after the ancient Roman, *divide et impera*, “Divide and rule.”
>> binary_search_recursive(A,88,1,length(A))
ans =
   10

This search involved three calls of the function (the second two being recursive calls). The following is a summary of the search progression showing the search ranges for each successive function call along with the element \( A(\text{mid}) \) in the middle of the range that was compared to the target:

- Range = \( A(1) = 2 \) to \( A(11) = 97 \), middle element \( A(6) = 43 \)
- Range = \( A(7) = 74 \) to \( A(11) = 97 \), middle element \( A(9) = 80 \)
- Range = \( A(10) = 88 \) to \( A(11) = 97 \), middle element \( A(10) = 88 \)

Thus, the three numbers, 43, 80, and 88, were compared with the target.

Now let’s search for 17:

>> binary_search_recursive(A,17,1,length(A))
ans =
   3

Here is the summary of this search:

- Range = \( A(1) = 2 \) to \( A(11) = 97 \), middle element \( A(6) = 43 \)
- Range = \( A(1) = 2 \) to \( A(5) = 24 \), middle element \( A(3) = 17 \)

Here, only the two numbers, 43 and 17, were compared with the target.

It is interesting to note that the first number compared with the target for this list is, in each case, 43. This is always the first number for this list, because the algorithm always starts at the middle of the list, regardless of the value of the target. It is also noteworthy that if there are two or more instances of the target on the list, only one will be found (not necessarily the first one).

Finally, let’s try one example in which the target is not on the list:

>> binary_search_recursive(A,90,1,length(A))
ans =
   -1

The summary looks like this:

- Range = \( A(1) = 2 \) to \( A(11) = 97 \), middle element \( A(6) = 43 \)
- Range = \( A(7) = 74 \) to \( A(11) = 97 \), middle element \( A(9) = 80 \)
- Range = \( A(10) = 88 \) to \( A(11) = 97 \), middle element \( A(10) = 88 \)
- Range = \( A(11) = 97 \) to \( A(10) = 88 \)
This time, the numbers 43, 80, 88, and 97 were compared with the target. In the last range the first and last elements are out of order, so the algorithm treats the range as empty. No element is compared with the target. The algorithm halts and returns -1 to indicate that the target is not on the list.

10.1.2.2 Iterative implementation of the binary search

Every recursive function can be replaced by an iterative function that produces the same result. Figure 22 contains an iterative version of the binary search.

As in the recursive version, the list is repeatedly divided into smaller and smaller parts, but instead of executing a recursive call after each division, another iteration of the
**10.2 Sorting**

Thanks to the advantage provided by the binary search algorithm over sequential search, it should be obvious that large lists that are going to be searched repeatedly should first be sorted in order to make subsequent searches more efficient. Sorting is another important task in computer applications and is carefully studied by students of that field. It is important because of the large savings in search time possible when using the binary search instead of the sequential search. Many sorting methods have been developed, some of which are quite complicated (far more complicated than the binary search algorithm given above, for example). The benefit in reduced sorting time is, however, worth the programming effort. Students who choose computer science or computer engineering as their major spend a good deal of time mastering the concepts involved in efficient sorting algorithms. The simplest form of the problem is to sort into ascending order a vector of numbers. We will confine our attention to that problem.

**10.2.1 Selection sort**

A sorting algorithm that is not very efficient, but is simple enough to be readily understood is the **selection sort**. A function that implements the selection-sort algorithm is given in Figure 23.
It is used as follows,

```
>> vs = selection_sort(vu)
```

where `vu` is the vector to be sorted and `vs` is a vector containing the same elements as `vs` but arranged in ascending order.

To learn how the selection sort works, suppose `selection_sort` is given the following list:

```
v = [28    27    86    15    28    64]
```

Step 1: Find (i.e., to “select”, hence the name selection sort) the smallest element, which in this case is `v(4)`, which equals 15, and swap it with the first element, `v(1)`, which

```
function v = selection_sort(v)
    %SELECTION_SORT sort in ascending order
    % A = SELECTION_SORT(V) sorts vector V in ascending order and returns it in A.
    % The method used is selection sort.
    for m = 1:length(v)-1
        m_min = m;
        for n = m+1:length(v)
            if v(n) < v(m_min)
                m_min = n;
            end
        end
        if m_min ~= m
            temp = v(m);
            v(m) = v(m_min);
            v(m_min) = temp;
        end
    end
```

Figure 23. An implementation of the selection-sort algorithm
equals 28:
\[ v = [28 \ 27 \ 86 \ 15 \ 28 \ 64] \]

The result is
\[ v = [15 \ 27 \ 86 \ 28 \ 28 \ 64] \]

Now we have taken care of \( v(1) \), and all that remains is to sort the items in \( v(2:end) \).

**Step 2:** Find the smallest element in \( v(2:end) \) and swap it with \( v(2) \). As it happens in this case, the smallest element in \( v(2:end) \) is already in \( v(2) \). So no swap is necessary. The list is still
\[ v = [15 \ 27 \ 86 \ 28 \ 28 \ 64] \]

Now we have taken care of \( v(1:2) \), and all that remains is to sort the items in \( v(3:end) \).

**Step 3:** Find the smallest element in \( v(3:end) \) and swap it with \( v(3) \). As it happens there are two smallest elements: both \( v(4) \) and \( v(5) \) have the smallest value, 28. We choose to use the first of these, swapping with \( v(3) \): \n\[ v = [15 \ 27 \ 86 \ 28 \ 28 \ 64] \]

The result is
\[ v = [15 \ 27 \ 28 \ 86 \ 28 \ 64] \]

Now we have taken care of \( v(1:3) \), and all that remains is to sort the items in \( v(4:end) \).

Each of these three steps is carried out by one iteration of the outer `for` loop in lines 7-19 of `selection_sort`. There have thus far been three iterations, during which \( m \) has taken on the values 1, 2, and 3. Each iteration of that loop involves finding the smallest element in \( v(m:end) \) and swapping it with \( v(m) \), as we have been doing above. When iteration \( m \) is over, \( v(m:end) \) has the correct elements and \( v(m+1:end) \) remains to be sorted.

The work of finding the smallest element is done by lines 8-13. (We could use Matlab’s `min` function to find it, but that would hide some of the complexity of the algorithm.) When those lines are completed, the variable `m_min` holds the index of the smallest
element in $v(m:\text{end})$. Those lines begin with the “guess” on line 8 that the smallest element is at the beginning of $v(m:\text{end})$. Thus, $m_{\text{min}}$ is set equal to $m$. The inner loop in lines 9-12 then compares that guess against the first element after $m$, namely, $v(m+1)$. If it is smaller, then $m_{\text{min}}$ is set equal to $m+1$, which is the new guess. The loop index $n$ holds the index of each element to be compared with the latest guess. When $n$ has reached the end of the vector, and element $v(\text{end})$ has been checked, the guess has become a certainty. At this point we know that $v(m_{\text{min}})$ is the smallest element in the $v(m:\text{end})$.

Now it is time to swap $v(m)$ and $v(m_{\text{min}})$. That work is handled by lines 14-18. The if statement checks to see whether a swap is necessary at all. It is unnecessary if the smallest element is already at $v(m)$, so the swapping is avoided if $m_{\text{min}}$ is equal to $m$. If a swap is required, lines 15-17 do the job. These three lines are very common in computer programming. In order to swap two numbers, it is always necessary to hold one of them in a temporary variable, often called temp, as in this example. The swap always begins (line 15) with the copying into the temporary variable of the first one of the values to be swapped, in this case $v(m)$. The next step (line 16) always copies the second of the values to be swapped, in this case $v(m_{\text{min}})$ into the first variable. If it were not for the fact that we have saved a copy of the original value of first variable in temp, this copying (i.e., line 16) would cause it to be lost. The final step (line 17) always retrieves that saved value and copies it into the second variable.

The selection sort continues in this way with its outer loop incrementing $m$ and working on a smaller and smaller part of the list, $v(m:\text{end})$, each time using the inner loop to find the smallest element in $v(m:\text{end})$, and using lines 14-18 to swap it with element $v(m)$. The outer loop stops one position short of the end of the loop when the index $m = \text{length}(v)$. It can stop there because the previous step has established that the last element is not smaller than the next-to-last element. Therefore, the sort is complete.

10.2.2 Quicksort

A sorting method that is much more efficient for most lists than selection sort, and is fairly easy to understand, is quicksort. A function that implements the quicksort

---

algorithm is given in

```plaintext
function v = quicksort(v)
% QUICKSORT sort in ascending order
% Uses vectorized operations and
% logical variables. The method
% used is quicksort.

% Base case (empty or singleton)
if length(v) <= 1, return, end

% Recursive case
less = v < v(1);
smaller = v(less);
remainder = v(~less);
remainder = remainder(2:end);
left = quicksort(smaller);
right = quicksort(remainder);
v = [left v(1) right];
```

Figure 24. Quicksort employs the divide-and-conquer strategy. The central idea is to divide the vector \( v \) that is to be sorted into three parts:

(a) \( v(1) \)

(b) a list, which we will call \texttt{smaller}, that consists of all the elements in \( v \) that are smaller than \( v(1) \)

(c) a list, which we will call \texttt{remainder}, that holds the remainder of the elements in \( v \), \textit{i.e.}, those elements in \( v[2:end] \) that were not copied into \texttt{smaller}.
Then, we apply \texttt{quicksort} recursively to \texttt{smaller} to produce a sorted list, which we will call \texttt{left}, and then we apply \texttt{quicksort} recursively to \texttt{remainder} to produce a second sorted list, which we call \texttt{right}. Finally we set

\[
\texttt{v = [left, v(1), right]}
\]

which shows why we named those sorted parts \texttt{left} and \texttt{right}. The resulting list is now sorted.

As an example, let’s start again with the list we used above with the selection sort:

\[
\texttt{v = [28 27 86 15 28 64]}
\]

The first (non-recursive) call,

\[
\texttt{>> quicksort(v)}
\]
begins by dividing $\mathbf{v}$ into three parts, as described above,

(a) $\mathbf{v}(1)$, which equals $28$
(b) smaller, which equals $[27 \ 15]$
(c) remainder, which equals $[86 \ 28 \ 64]$

Step (b) is accomplished by means of lines 11-12. Together these two lines provide an example of Matlab’s “logical indexing” feature, which is the use of a logical array to select the desired elements of an array, instead of the normal method of listing the desired indices. Logical indexing was introduced in Chapter 6.

- The right side of line 11 produces a vector whose elements each equal 1 or 0, with 1 meaning true and 0 meaning false. The vector is then assigned to less. The true/false value of each element less$(i)$ is based on whether $\mathbf{v}(i) < \mathbf{v}(1)\ i = 1, 2, \ldots$. As explained in Chapter 6, the result is called a “logical array”, meaning that its type is “logical” and can therefore be used in operations like the one on the right side of line 12. The value that is assigned to less in this example is $[0 \ 1 \ 0 \ 1 \ 0 \ 0]$, since $\mathbf{v}(2)$ and $\mathbf{v}(4)$ are the (only) elements that are less than $\mathbf{v}(1)$.

- The right side of line 12 produces a vector consisting of those elements of $\mathbf{v}$ that are located at the same positions as the 1s in less. They are kept in the same order too. Thus, the vector assigned to smaller is $[27 \ 15]$, since $\mathbf{v}(2) = 27$ and $\mathbf{v}(4) = 15$.

The total effect of lines 11-12 is to remove the elements of $\mathbf{v}$ that do not satisfy the rule, $\mathbf{v}(i) < \mathbf{v}(1)$.

Step (c) is accomplished by lines 13-14.

- Line 13 is similar to line 12, but the logical array less has been replaced by $\neg$less, whose value is “opposite” to that of less: $[1 \ 0 \ 1 \ 0 \ 1 \ 1]$. Thanks to the “not” operator $\neg$, the vector assigned to smaller has 1s at the positions for which the elements $\mathbf{v}(i)$ is not less than $\mathbf{v}(1)$. These elements are $\mathbf{v}(1) = 28, \mathbf{v}(3) = 86, \mathbf{v}(5) = 28$, and $\mathbf{v}(6) = 64$. Logical indexing causes each of these elements to be copied into remainder, which now equals $[28 \ 86 \ 28 \ 64]$. Note that $\mathbf{v}(1)$ is on the list. While it is true that $\mathbf{v}(1)$ is not smaller than $\mathbf{v}(1)$, we do not want it on this list.

- Line 14 is responsible for removing $\mathbf{v}(1)$ from remainder. Because the logical indexing operation in line 13, does not change the order of the elements, $\mathbf{v}(1)$ is still in the first position. Since remainder(2:end) contains everything in remainder except its first element, we can be assured that $\mathbf{v}(1)$ has been left off.
Now we carry out the first recursive call, line 15:

left = quicksort(smaller)

which returns left equal to the sorted list, [15  27]. Our second recursive call, line 16,

right = quicksort(remainder)

sets right equal to [28  64  86]. Finally, we combine the three parts into one sorted list:

v = [left, v(1), right], which equals [15  27  28  28  64  86].

The algorithm is completed and the v is sorted.

Every recursive function must have a base case, and, as can be seen in step 8 of

```matlab
function v = quicksort(v)
% QUICKSORT sort in ascending order
% Uses vectorized operations and
% logical variables. The method
% used is quicksort.

% Base case (empty or singleton)
if length(v)<=1, return, end

% Recursive case
less = v<v(1);
smaller = v(less);
remainder = v(~less);
remainder = remainder(2:end);
left = quicksort(smaller);
right = quicksort(remainder);
v = [left v(1) right];
```
Figure 24. the base case is determined by the length of the list. If the length is one, then there is no work to do. The list is returned without change. The same holds for an empty list (length equal to zero). For all other lengths recursion is employed.

The quicksort algorithm is much faster than selection sort for most lists. As an example, on one machine the selection sort takes 19 times as long as quicksort to sort 100 thousand numbers.\textsuperscript{11}

10.2.3 Merge sort

The quicksort is not the best sorting algorithm available. It has a problem when the list that it is given to sort happens to be almost sorted. The problem is that its divide-and-conquer strategy no longer works well. In that case the list smaller is always empty, so the first recursive call does nothing. The recursive call, \texttt{right = quicksort(remainder)} is given a list that is shorter than the original list by only one element. As a result there is one recursive call required for each element on the list. For all but very small lists (e.g., less than 500 elements) the stack grows too large and Matlab gives an error message. This stack-overflow problem can be solved by using a nonrecursive version of \texttt{quicksort} (remember that there is always a nonrecursive version), but, because the division of the list is so one-sided, the search is in this case no more efficient than selection sort.

\textsuperscript{11} 160 seconds for \texttt{selection\_sort}, 19 seconds for \texttt{quicksort} on a 1.3 GHz Pentium processor with 512 MB of RAM
Chapter 10  Searching and Sorting

There are sorts that do not suffer from this problem. One of them is the **merge sort**. Like quicksort, it uses divide and conquer, but it is guaranteed to divide the list into two equal parts, or, if there is an odd number of elements parts whose lengths differ by only one. A recursive implementation of merge sort is shown in Figure 25. The algorithm is fairly simple, but this implementation hides a bit of the complexity because of the call to the function `merge_sorted_lists`. The base case is determined by the length of the list as in quicksort: lists with 0 or 1 elements are already sorted, so the function simply returns. Line 11 begins the interesting part of the algorithm. The midpoint is determined in line 11. In line 12 the elements from the first up to this midpoint are sorted. In line 13 the rest of the numbers are sorted. Line 14 does the rest of the work by calling `merge_sorted_lists`. That function combines the two sorted lists into one sorted list. We leave a discussion of that function to the exercises that accompany this chapter. It is line 11 that guarantees that this algorithm, unlike quicksort, will always divide the list into approximately equal halves, regardless of the elements in it.

There are many other sorting algorithms available, and computer scientists spend a good deal of time studying them. Matlab provides an excellent sorter called sort. It uses varying strategies according to the size of the list and the size of the available memory. A study of these strategies and of sorting in general is beyond the scope of this textbook.
10.3 Algorithmic Complexity

When a computer scientist says that an algorithm is fast, he or she means something very specific. A “fast” algorithm is not simply an algorithm that runs fast. One problem with that definition is that the speed at which an algorithm runs depends on the hardware that it runs on. Since computers are different, and since computers improve every year, it is meaningless to measure the speed of an algorithm by timing it. Furthermore, the relative speeds of two algorithms that are running on the same machine may depend on the size of the data set that they are working on. For example, for a list of 4 numbers, the sequential search may take less time than the binary search, while for larger lists the binary search will beat the sequential search (and beat it soundly).

10.3.1 Worst-case analysis

The more important aspect of an algorithm is the way in which the number of operations that it must perform grows as the size of the dataset grows. That number may depend on the data involved. An example of that data dependence is provided by the sequential search. The best case for the sequential search that in which the first item on the list is the one we are looking for. In this trivial case only one comparison is required. In designing algorithms, however, the worst case is usually of most concern. This makes sense in the example of the search for credit-card information above, for example, where we would want to minimize longest time that a customer would have to wait.

Therefore, most algorithms are subjected to a worst-case analysis. For example, an analysis of the sequential search algorithm above reveals that there are two worst cases (i.e., they are equally bad and are worse than all other cases): (a) the target is not on the list at all or (b) it occurs only once and is the last item on the list. In each of these cases the number of comparisons is equal to the number $N$ of items on the list. While comparisons are not the only work done by the algorithm, the number of comparisons is a good measure of the algorithm’s work because, in the general form of the algorithm, comparing a target to an item on the list is the most time consuming step and also because the number of other required operations is approximately proportional to the number of comparisons.

For the binary search there is one worst case: the target is larger than all the items on the list. In this case, the number of comparisons is equal to $2^{\text{ceil}\left(\log_2(N+1)\right)}$, where $\log_2(x)$ is the power to which 2 must be raised to get $x$. It is fairly easy to see that this is indeed the number of comparisons required by considering some specific worst-case examples.

As mentioned above, the time required to complete the search is approximately proportional to the number of comparisons required. Thus, the worst-case time necessary to complete these algorithms on a list of length $N$ are related to $N$ as follows:
sequential: \( a + bN \),  
binary: \( a + b \cdot \text{ceil}(\log_2(N+1)) \),

where the \( a \) and \( b \) represent constants that will be different for the two algorithms (i.e., two different values for \( a \) and two different values for \( b \)). The value of \( a \) represents the start-up time for the algorithm (including the time for Matlab to set up the activation record for the function). The value of \( b \) represents the amount of time required per comparison in the sequential search and represents twice the amount of time required per comparison in the binary search. The plot below shows a sample behavior for these functions when \( a=100 \) and \( b=5 \) for the sequential search and \( a=200 \) and \( b=10 \) for the binary search:

![Comparison of Search Times](image)

Note that the binary search (these plots are for fictitious implementations) actually requires more time for small lists (fewer than about 30 items), but after the size of the list has grown to 100, the sequential search takes over twice as long. The advantage of binary search grows as \( N \) grows. For \( N = 1000 \), the ratio is 8 to 1, and for \( N = \) one million the ratio is over 6000 to one!

### 10.3.2 Order Notation

The important difference between the two plots above lies in the difference between their general shapes. Both plots increase monotonically with the number of items, but the red plot (the binary search) has a different shape. It curves downward. Because of that
downward curve, it will eventually go below the straight-line blue plot (the sequential search).

There are three things that differ between the two formulas above. Each formula has different values of $a$ and of $b$ and one has $\text{ceil}(\log_2(N+1))$ where the other has simply $N$. Of these, the difference that determines the shape of the curve is the last one—the dependence on $N$. That is so because, for any values of $a$ and $b$ in either or both formulas, the second formula will produce a plot that curves downward and hence will cross the straight line for sufficiently large $N$. That means that for a big enough problem the binary search will win, even if its $a$ and $b$ are bigger then the $a$ and $b$ for the sequential search (as they are in the example above). Furthermore, as the plot below shows, there is little importance to either the $\text{ceil}()$ or the $+1$ in $\text{ceil}(\log_2(N+1))$. When we omit them, we are left with $\log_2(N)$, and the resultant plot (the dashed line) is almost the same. Finally, the importance of $\log_2(N)$ is the fact that we are taking the $\log$ of $N$, regardless of what the base is. As the plot below shows, if we were to replace $\log_2$ with $\log_{10}$ (log base 10), the general shape of the plot would be the same. It is curved downward, so it will cross the plot of the sequential search. It crosses the plot of the sequential search at a different place, but the important thing is not where it crosses, but the fact that it must cross somewhere.

Thus, the important aspects of the worst-time behavior of the binary search can be described simply by saying that the plot of the search time versus $N$ has the same shape as $\log(N)$ for any base. Likewise the important aspects of the worst-time behavior of the sequential search are captured by saying that the plot of the search time versus $N$ has the
same shape as $N$. These statements are said more formally as follows: The worst-case time complexity of the sequential sort is order $N$. The worst-case time complexity of the binary sort is order $\log N$. The phrase “order $N$” is written $O(N)$. The phrase “order $\log(N)$” is written $O(\log N)$. Usually a computer scientist will simply write, “the binary search is $O(\log N)$”, leaving out the words “time” and “complexity”, which are to be understood. (Sometimes we are interested in the increase in memory required by an algorithm as $N$ increases, as opposed to the increase in time. In that case, we measure space as a function of $N$ and determine the “space complexity”). This notation is called Order notation, Big-O notation, or sometimes, $O$ notation.

The selection sort is $O(N^2)$. The quicksort algorithm is also $O(N^2)$, because of its difficulties with its worst case, a list that is almost in order. Merge sort and all the best sorting algorithms, Matlab’s sort for example, are $O(N \log N)$.

We have now discussed four complexities: $O(\log N)$, $O(N)$, $O(N \log N)$, and $O(N^2)$. As the plots below show, they are listed here from fastest to slowest. The relative sizes are so different that in order to show them all on the same plot, we have multiplied the first three by 10,000, 500, and 100, respectively.

![Figure No. 1](image-url)
10.3.3 Dominance

It is clear from the discussion above that the value of $a$ in the formulas is unimportant. That is because the second term *dominates* $a$ for very large values of $N$. A similar thing happens for formulas like this: $a \log(N) + bN$. Here, the second term is the only important one because $N$ dominates $\log(N)$. The *dominance* of $N$ over $\log(N)$ can be stated this way: the ratio, $N/\log(N)$ can get as large as desired, provided $N$ is made large enough. In terms of algorithms, that means that if an algorithm’s time-behavior is of the form $a \log(N) + bN$, then the algorithm is $O(N)$, not $O(\log N + N)$. The dominant term determines the order of the algorithm because the other terms become insignificant for very large $N$. 
Concepts from this chapter

- Computer Science and Mathematics:
  - sequential search, also called linear search
  - flag
  - binary search
  - algorithms (i.e., the field of study)
  - divide and conquer
  - selection sort
  - quicksort
  - merge sort
  - algorithmic complexity
    - worst-case analysis
    - time complexity
    - order notation
      - big-O
      - $O(\log N), O(N), O(N \log N), O(N^2)$
    - dominance

- Matlab:
  - [No new concepts specific to Matlab]
Chapter 11

Graphical User Interfaces

By default, Matlab runs through a command line interface. That is, individual commands, functions or scripts are executed by typing instructions into the command window and pressing enter. For many years, this was the primary method for interacting with a computer. Contemporary applications of the command line interface still abound. The DOS prompt in Microsoft’s command window, as well as the UNIX and Linux operating systems, for example. Since the 1990’s, however, the command line has been replaced as the dominant method for computer control by the graphical user interface. Popularized by Apple, and later Microsoft, the graphical user interface provides a more user-friendly and intuitive means for human-computer interaction. Long, text commands, prone to error, have been replaced by clicks of the mouse on icons and buttons. Virtually all programs you use today utilize graphical user interfaces. Even Matlab’s command window is encapsulated within such an interface. Even though Matlab programming is primarily designed for small groups of experienced users, the importance of graphical user interfaces has necessitated their inclusion.

Matlab’s capabilities in regard to graphical user interfaces have evolved significantly over the past few years. Each subsequent release has included improvements, refinements, and additions. Therefore, the features discussed in this chapter may not be representative for your version.12

11.1 Object-Oriented Programming

Any discussion of programming graphical user interfaces needs to begin with an understanding of what goes on “behind the scenes”. This involves a certain degree of computer science theory. These concepts are important, however, as they form the basis of most major programming languages in use today.

To begin, graphical user interfaces became possible (and practical) thanks to the development and spread of object oriented programming languages. Object-Oriented Programming (OOP) is a method of organizing data and the functions that operate on that data. In the old days (a relative term, to be sure, as this means in the mid-to-late 80’s and before), most languages had a strict separation between data and functions. OOP bridged that gap by linking functions and their related data together. This distinction may seem trivial, but it was a revolutionary idea in computer science. OOP allows for complicated operations to be performed with simpler coding (in many, but not all cases). It allows for easier replication of code, improved efficiency and more rapid development. The result has been that over the past decade, many previously widespread programming languages

12 This chapter is based on Matlab Version 7.7.0.471 (R2008b), released September 17, 2008.
have practically died out and have been replaced with C++, Java, Visual Basic and others. Matlab is, in fact, an OOP language.

11.1.1 Objects

OOP requires, by definition, the use of objects. This suggests the question, of course, what is an object? In everyday life, we can think of an object as a self-contained, uniquely identifiable “thing”. That is, if you look around your desk, you’re likely to see many different objects: a pencil, your computer monitor, a math textbook. Each of these objects is distinct from the others, even if only in small ways. For example, two pencils might look identical, but they exist in two different spatial locations, and are therefore distinct, separate objects.

To a certain extent, this definition is a slippery slope: How far do you go in defining distinct objects? For example, take a look at your computer keyboard. We might think of that as an object. But a keyboard is made up of individual keys. So what’s the object: the keyboard or the keys? The answer actually can be both. An object may consist of a group of other objects. But you probably see how this can get confusing. There is really a great deal of latitude in defining an object. As a programmer, you get to make the decisions as to what is or is not an object.

From a programming perspective, an object is defined as a set of data values and all the functions that operate on that data. The user of a programming language that supports object oriented programming is able to specify both the type of data and the functions for new objects. Thus, an object is simply an instance of a user-defined data type. The data may be other objects, which may, in turn be objects, until we get finally to an object whose data are of types that are defined by the language itself—types that “come with” the language. For example, an object may be defined that consists of a two-element vector and a function that calculates the mean of these two elements. Another object may be a collection of other objects and functions that operate on that collection. In programming languages such as C++ or Java, this very general approach to the meaning of object is supported by an incredibly flexible environment for defining and creating objects. For our purposes, we will take a much more restrictive, but easier to understand, view. We will consider only a set of predefined objects in graphical user interfaces are very well defined and rather intuitive to program.

Think about your Matlab operating environment. You have a series of windows (command window, workspace, current directory, etc.) There is a menu along the top as well as a series of icon buttons for new document, cut, paste, etc. Each window is an individual object. So is each button. We can also think of the overall operating environment as an object as well….one that contains a series of other objects (the windows, buttons, etc.)
11.1.2 Properties
Objects are described by their properties. These are typical descriptive characteristics such as color, height, width, location, etc. The objects in our Matlab operating environment all have a distinctive set of properties. For instance, the command window has properties describing its size, background color and location on the screen. Even the text that appears inside the window is a property. The buttons have properties describing their color, location, the image that appears on them and even whether they’re active or inactive (grayed-out).

Much of the work that’s involved in programming graphical user interfaces involves manipulating the properties of objects on your screen. For example, if you designed a program that would multiply two numbers and return the answer at the click of a button, your program would have to change the String property of an object to display the product.

In most OOP languages, the syntax for manipulating properties resembles that of structure arrays:

```
Object_Name.Property_Name = Property_Value
```

In Matlab, you use a function, called `set` to change the value of a property, and a function called `get` to read the value of a property. We describe these functions below.

11.1.3 Event-Driven Programming
A recent offshoot of OOP is a subcategory called Event-Driven Programming. This has been made more prominent by the proliferation of graphical user interfaces and Microsoft’s so-called “Visual” programming languages. The idea is that your program exists at a steady state. That is, in the normal scheme of things, your program sits there and does nothing but wait. It waits for something to happen. That “something” is called an event.

There are many possible events in graphical user interfaces: the click of a button, entering text in a box, hitting the enter key, closing a window, and even something as simple as moving the mouse. When one of these events occurs, your program will execute a function corresponding to that event. Oftentimes, this is the bulk of your programming. You have to write the code that tells Matlab what to do when specific events occur.

In most of the “Visual” languages (such as Visual Basic, Visual C++, Visual J++, etc.), there are literally hundreds of possible events that must be accounted for. In Matlab, however, there are only a few possible events. In general, each object has one potential event associated with it. This significantly reduces the complexity and quantity of the programming required for a graphical user interface.
11.2 Graphical User Interfaces in Matlab

Programming a graphical user interface (GUI), (pronounced either G – U – I or gooey) has become much easier in Matlab over the years. Matlab GUls consist of two files: a figure file (.fig), which consists of the physical layout of your GUI and its objects in a window and an M-file (.m) that contains the code that describes the activities that you wish to take place when certain events occur, along with some other code provided automatically by Matlab to operate your GUI.

When creating a GUI, you have three primary tasks:

1. Decide on the objects you wish to use in your GUI and place them within a figure window.
2. Set the initial properties for your objects.
3. Write the functions to be called for each event

11.2.1 Handles

In most OOP languages, setting the properties of objects is relatively straightforward. As mentioned previously, however, it’s a little more complicated in Matlab. The reason has to do with the way in which Matlab stores object information in GUls. Matlab holds every object and all their properties in a large cell array. To keep track of individual objects, Matlab assigns each object a reference number, called a handle. Rather than referring to an object by its name, we have to refer to it by its handle.

Handle numbers are generated by Matlab, and in general, we don’t need to know their values. We need to pass them as arguments to some important GUI functions, however, so there must be a way to assign them to variables. In earlier versions of Matlab (before 6.0) it was necessary in every case to use one of these three functions to find an object’s handle.

- gcbo – returns the handle of the object that initiated the event
- gcbf – returns the handle of the figure that contains the object that initiated the event
- findobj(figure, property, value) – returns the handle of any object in a given figure that has a matching value for the given property

In all subsequent versions, however, there are always two variables available that contain the handles that we need: First, a variable named hObject contains a copy of the handle of the object that initiated the most recent event, thus removing the need for calling gcbo. Second, a variable named handles is a struct, whose field names are the tags of all the objects in the GUI figure, including the figure itself, and whose field values are
their handles, removing the need to use `gcbf` to find the figure handle or `findobj` to find the handle of an object by giving it the object’s tag name.\textsuperscript{13}

### 11.2.2 Callbacks

Every event can have a function associated with it. For example, if a user clicks on a button, and that event has a function associated with it, Matlab will immediately call that function. Such functions are known as `callback` functions, or simply `callbacks`. We’ll learn below how to specify the particular callback function that is associated with a particular event. As a programmer, one of your tasks is to write code in each callback function that performs the specific task that you wish to be associated with that specific event.

Thankfully, Matlab will actually automatically generate the code that provides the nuts and bolts of your GUI (such as moving the mouse across the screen, displaying the proper graphic for a button depression, etc.) Matlab will even generate the function calls and basic function definitions for your callback functions! All you have to do is insert additional the code inside the callback that does what you want it to do.

### 11.2.3 Working With Properties

Much of your work in writing callback functions will involve manipulating the properties of the various objects within your GUI. Indeed, any changes to the appearance during the running of your GUI is done by altering the properties of your objects. Reading input data typed by the user is achieved by accessing property values as well. There are two primary functions used to work with properties in Matlab GUIs.

The function `get` is used to read property values of objects. The basic syntax of the `get` function is as follows:

```matlab
get(handle, 'property')
```

Two input arguments are required: the handle of the object in question and the name of the property you wish to access. Note that the name of the property is stored as a string and therefore must be surrounded by single quotes.

\textsuperscript{13} The function `findobj` was used primarily to find the handle of any object other than the active figure window or the object which initiated the event function call. For example, if, when a user clicks a button, we wish for text to appear in a textbox as a result, we could use `findobj` to find the handle of that textbox. The function searches through all the objects in a given figure window and returns the handles of those objects with property values that match the value passed to the function. We would use the Tag property to give each object a unique tag and then use `findobj` to search for an object with that specific tag name.
The `set()` function is used to alter the property values of objects. The syntax of `set` is as follows:

```matlab
set(handle, 'property', value)
```

Three input arguments are required for the `set` function: the handle of the object you wish to manipulate, the name of the property to be changed and the value you wish to set that property to. Just as with the `get` function, the name of the property is stored as a string, so it must be surrounded by single quotes. Note that many property values of objects in Matlab are stored as strings. In those cases, the value you set should also be surrounded by single quotes. In some cases, however, the values will be arrays or other numeric types in which case quotes are not appropriate.

### 11.3 Using Guide

By this point, you may have come to the conclusion that programming GUIs is complicated business. You’re right. The good news, however, is that Matlab has taken all of the most difficult pieces of the work required to make GUIs and has done them for you! These pieces have been put together in a tool called `guide` (stands for the GUI Development Environment). Guide is an easy-to-use tool for creating the layout of your GUI. Guide allows you to control the initial property settings of all your objects: alignment, spacing, location placement, color, etc.

To begin, you simply type the word `guide` in the Matlab Command Window.

In the newest version of Matlab, this will launch the `guide` quick start window where you will be given a number of choices.
You may choose to create a new GUI from a number of pre-defined templates, or you may open an existing GUI. For now, let’s start with a clean slate and select the default, a blank GUI. We do that by clicking OK.

After initializing, guide will present us with our workspace. (Displayed on the next page)

Guide’s initial figure window presents us with a grey grid workspace. This is the area where we will design the layout of our GUI. Note that the grid will not actually show up on the finished product. The grid lines are there simply for your benefit to assist with the visual placement and alignment of your objects.

Along the left hand side of the window is the “component pallet”, which consists of a set of buttons, all but one of which represents a type of object, which the Matlab help utility terms a “component”, that can be added to a GUI. The button on the top left featuring the cursor arrow does not represent a component. It is used to point at and click on components already in the GUI. The remaining buttons make up the “Component Pallet”, and are used to create objects.

Along the top are a series of menu items and tools to assist you in the creation of your GUI. In addition to standard features like save, cut and paste, are tools such as the menu editor, align-objects tool, the property inspector and a run button (green arrow) for testing your GUI. These tools will be covered a little later in this section.
Each of those buttons on the left has a name that tells you what object they will produce in the GUI workspace. To see their names, you can click File at the top left, and then Click Preferences..., then click on the box that says “Show names in component palette”. Then click OK. You will see a name added to each of those buttons. (We will not include those in figures below to save space, but you will want to use them at first, until you get used to the icons and no longer need the names.) The most important ones are described in Table 19.
## Table 19 Standard Objects in Matlab GUIs

<table>
<thead>
<tr>
<th>Object Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pushbutton</td>
<td>Creates a standard pushbutton which triggers a callback function when clicked.</td>
</tr>
<tr>
<td>Toggle Button</td>
<td>Creates a toggle button. Toggle buttons differ from pushbuttons in that a toggle button has two states: “on” or “off” and that state changes each time the toggle button is clicked. Each click of the mouse on a toggle button triggers a callback function.</td>
</tr>
<tr>
<td>Radio Button</td>
<td>Radio buttons are similar to toggle buttons in that they have two states: “on” or “off”. They appear as small circles. A dot appears within the circle when the radio button is in the “on” state. Typically, radio buttons are grouped together to provide for mutually exclusive choices. Each click on a radio button triggers a callback function.</td>
</tr>
<tr>
<td>Checkbox</td>
<td>Checkboxes are similar to toggle buttons and radio buttons because they also have two states: “on” or “off”. They appear as small squares and have a check mark when in the “on” state. Each click on a checkbox triggers a callback function.</td>
</tr>
<tr>
<td>Edit Text Box</td>
<td>The edit text box displays a string which can be modified by the user. This is the only practical method for receiving text input from the user. After a user has entered information, a callback function is triggered when they press the enter key.</td>
</tr>
<tr>
<td>Static Text Box</td>
<td>The static text box displays a string to the user. This string can not be modified by the user. It should be noted that by default, no actual box appears on your GUI (unless you set the properties in such a way for one to appear). This is a convenient method for placing labels, titles and other informational text within your GUI. A static text box will never trigger a callback.</td>
</tr>
<tr>
<td>Slider</td>
<td>A slider allows you to change a numeric value in a continuous fashion by dragging a marker or clicking on an arrow button at either end. The most common application of sliders is to scroll up and down the viewable part of a window. In Matlab, however, sliders are used to change a numeric value. This can ultimately be used in the traditional manner, of course, but can also be used to change variable values, etc. Every change in the slider value triggers a callback function.</td>
</tr>
<tr>
<td>Frame</td>
<td>A frame is simply a rectangular box you can use to highlight, group or emphasize an area of your GUI. It should be noted that frames are purely aesthetic – they have no impact on your program at all other than look and feel. Frames never trigger callback functions.</td>
</tr>
<tr>
<td>List Box</td>
<td>A list box contains a series of string values in a vertical list. A user may select an item on the list by single or double-clicking on the time. Each time the user selects a string in the listbox, a callback function is triggered.</td>
</tr>
</tbody>
</table>
| Popup Menu     | Known as a select box in HTML programming, the popup menu
displays a series of string values in a vertical list. The list is accessed by pressing a button at the side of the menu. The user can then scroll down the menu and highlight an individual string to select it. A callback function is triggered each time the user selects one of the values.

**Axes**

Axes are used to display image or graph data within your GUI. Axes never trigger callback functions, but are still very important as they are the only means for showing plots, pictures and other graphics within your GUI.

To illustrate the use of guide, we’ll begin with a simple example. Suppose we want to create a GUI that can change the background color of its window. We might have a series of buttons that when clicked will change the figure window color appropriately.

*Setting up the Initial Layout*

To begin, we need to choose an appropriate size for our GUI figure. Notice that we can click and drag the bottom right corner of our layout grid to resize our workspace. This grid is an accurate representation of the size of our resulting GUI. An important note: it is an unfortunate characteristic of Matlab that guide doesn’t handle resizing the workspace window very well after you have placed objects on the workspace. Once you have begun placing objects, you’ll find that resizing the window often causes those objects to suddenly overlap or disappear altogether. (They haven’t really disappeared, they’ve just moved off screen.) In any event, your objects will no longer appear where you want them. As a result, you should always select the size of your workspace first before placing any objects in your GUI.
Having selected an appropriate size for our GUI, we’ll place a series of pushbuttons on
the screen, one for each color we want to use. First click Layout and uncheck “Snap to
Grid”. Now click on the pushbutton icon at left. Then, while holding down the left mouse
button, drag your cursor across the workspace to create an outline of the button.

Then, release the mouse button to create the pushbutton on your workspace.
By selecting the push button, you can click and drag it about the workspace to position it where you want it. By moving your cursor over its corner markers, you can also click and drag to resize the pushbutton.

Now, click right on the button and select Copy. Then click right on the background and click Paste. (or use Cntl-C and Cntl-V). Do this until you have four identical buttons.

The buttons are all arranged haphazardly. There’s nothing inherently wrong with that, but for a professional design, they should be aligned. The grid can help you move line then up if you have a good eye. But the Align Objects tool makes this much easier.

Begin by highlighting two buttons that you wish to align. (Hold down the Ctrl key while clicking on the top two buttons.) The little squares at their corners indicates highlighting.
Then either click Tools/Align Objects…, or find the little Align-Objects shortcut icon on the row below Tools (two yellow rectangles with a vertical line through them. That will bring up the toolbox.

The Align Objects toolbox allows you to choose a horizontal or vertical alignment. You can even set the spacing between objects. In this example, we hope to align the top two pushbuttons vertically. That means we want either the tops of the pushbuttons to align, their centers to align or the bottoms to align. Let’s try aligning them along their tops.

The vertical alignment options are in the first row. The first button is off, but the second button allows us to align based on the top of the pushbutton. Click that option and then click OK.

Our top row of pushbuttons are now aligned. Note that they’re aligned based on the button that had the maximum height. In other words, the button on the left came up to match the button on the right.

We can repeat this process for the buttons in the second row, but to learn a new feature, instead of using Cntl-left click to highlight two buttons, Just click in the background, hold
the click and move the mouse to produce a rectangle surrounding the lower buttons. They will be highlighted.

Align the lower pushbutton as you did the upper ones. Now horizontally align the two on the left as follows: Select them, and open Align Objects. Next (and this is important) click OFF in the Vertical section at the top. Then, select any of the Align buttons in the Horizontal section at the right and click. OK With a similar process you can then produce horizontal alignment for the two on the left right. After that you have four aligned, identical buttons.
The Property Inspector

The object of this GUI, you may remember, is to click on the buttons to change the color of the background figure window. But with four buttons that say “Push Button”, it’s tough for the user to know what to do. It would be much better if we could indicate what color each button might affect. The words that appear on the pushbuttons are part of their properties. Guide has a convenient interface to allow you to set the initial properties of the objects in your GUI. This is known as the Property Inspector.

You can activate the property inspector by double clicking on an object, or by pressing the property inspector button along the top menu bar. Let’s begin by double-clicking on the top left button.

The property inspector appears as a list, and indeed that’s what it is. It’s an alphabetical list of all the properties that can be set for a particular object along with their values. In this case, by scrolling down you can see 37 different properties that can be set for a pushbutton. In most cases, you’ll leave the default values alone, but you can change any of the that are not grayed out. These values are given to the object when the GUI starts to run. They can be changed, though, through commands during execution. Each object has specific individual properties of special importance, some of will be discussed in the next section, but there are a few properties that are common to most objects.

Most objects will have a property to define their colors. In the case of a pushbutton, that property is called “BackgroundColor”. Clicking on the little paint can to the right of this word, will bring up a series of colors to choose from. You can even define custom colors by setting the intensity of the red, green, and blue components—the so-called “RGB” values. To see its current RGB values, click on the “plus” sign to the left of BackgroundColor.
Some properties can have arbitrary values, which you can type in. They are indicated by the little pencil at the right. Others, like “Enable” have only a finite set of allowable settings, indicated by the little down arrow at the right. Enable has three settings: on, off and inactive, which you can see by clicking its value, which is by default on. Inactive means that when the user clicks on it, nothing happens. The inactive state is appropriate when the function of the object is not needed until some event has taken place. For example a file-save button might made inactive until a file name has been entered. Typically, however, objects will remain active, and we will leave ours in that state.

The “ForegroundColor” is another way of saying font color. It sets the color of the text on top of the button.

“Position” is stored in an array format. The four numbers correspond to the x and y locations of the top left corner of your object followed by the width and height. You can write code that will change the position of your object as the program runs. This will have the effect of moving your object about the screen. Crude animation is possible, though it will never be confused as a Matlab strength! You can choose the units of these dimensions through the Units property, which includes the choices, inches, centimeters, normalized, points, pixels, and characters, with the default
being characters. Other than characters, the most useful is probably normalized, whose range is 0 to 1, and whose meaning is “fraction of the width or height of the GUI window. Its usefulness is that, if the user changes the size of the GUI window while it is running, the position and size of the object will change proportionally.

The “String” property defines the text string that appears on the button. Let’s suppose we want this button to change the background color to red. To let the user know the purpose of this button, let’s replace the words “Push Button” with “Red” by clicking on the little text icon to the right of String, and replace “Push Button” in the box that opens with “Red”. Click OK and notice that the button changes immediately in the GUI. Then click the paint can to the right of ForegroundColor and choose the color red. Now Red appears in the GUI as the label for that button. Click the value (normal) for FontWeight and select bold.

An extremely important property that is common to all objects is the “Tag” property. The “tag” acts more or less like the name of the object. The tag will be used as part of the name of the callback function for the object. (It can also be used to find the handle of the object using the findobj function, but we will not need to do that.) Matlab will automatically create a unique tag name for each object in your GUI. You are free to change the tag name to whatever you wish, as long it is a legal Matlab variable name, and you should make every tag name both meaningful to anyone reading your program, as well as unique among the tags in your GUI. This tag will not be seen or known by the users of this GUI, but a meaningful name makes your code easier to understand. Let’s choose “Red_Button” for this tag, since that will be easy for us to remember while we are programming. Double-click the tag name, which is to the right of the word Tag (it will be of the form pushbutton#, where # is some number, in the case above it is pushbutton7), and type Red Button (no quotes required here).

Finally, the “Visible” property will make your object visible or invisible to the user. It has two settings: “on” and “off”. Switching the visible property to off will cause the object to disappear from the user’s screen. Switching this property to on will cause it to appear again. Like the inactive state, invisibility is appropriate when the function of the object is not needed until some event has taken place. For example a file-save button might be kept hidden until a file name has been entered. Typically, however, objects will remain visible, and will leave ours in that state. The property Inspector sets the initial states.
By double clicking any of the other buttons on your GUI, you can make their properties for appear in the property inspector.

Let’s round out our buttons by creating one for green, one for blue and one for purple. Then highlight them all and move them to be in the middle of the GUI by clicking left on anyone of them and while holding the mouse down, moving the mouse.

In addition to the obvious objects having properties, the figure window itself is also an object, and its properties can be viewed and changed in the property inspector as well. Double click on the grid area itself to view the figure window’s properties.

Once we’re satisfied with the initial layout and initial property settings of our GUI, it’s time to save them. From the file menu, select “Save As…” and give a name to your file. Any legal Matlab variable name will do. Let’s call this one MyVeryOwnGUI. Your GUI will be saved as a “.fig” format file: **MyVeryOwnGUI.fig**. Matlab calls files in this format *figure files*. They are used to store any figure, including the figure that holds a GUI. The file includes all the layout information and all the properties of the objects in the figure. You cannot inspect a figure file with an editor. You inspect it only indirectly by opening the figure file in **guide** and looking at its properties with the Property Inspector.

*The GUI M-file*
As you save your figure file, you will notice something unusual happening. Matlab has automatically created a new M-file as well, with the same name as the Fig-file that it just created: MyVeryOwnGUI.m. This file is used in conjunction with the figure file to run the GUI.

MyVeryOwnGUI.m consists of a primary function named MyVeryOwnGUI followed by a set of subfunctions. The primary function is the first function that is called when you run your GUI. You run the GUI, as you would run any function, simply by giving the name of the M-file in a command, either in the Command Window or within a function. As, for example,

```
>> MyVeryOwnGUI
```

Typically, as in this example, the call includes neither input arguments nor output arguments, but the function definition employs both varargin and varargout (See Chapter 8, Section 8.4) so that an arbitrary number of arguments of any types are supported. When we give this command, a figure pops up that looks like the figure on the left:

![MyVeryOwnGUI](image)

If we move our mouse over one of the buttons, it will be highlighted, as shown on the right, but clicking inside it does nothing. The event of clicking the mouse is, as this point, ignored by the GUI. We’ll make something happen below.
A short-cut to saving and running a GUI

You will often find yourself making a change to the layout or to the properties of one or more of the objects in the layout. Each time, you will want Matlab to (a) modify the figure file (or generate a new figure file, if it does not already exist), (b) modify the M-file (or generate it, if it does not already exist), and (c) call the primary function in the M-file. Since these three actions are needed over and over, Matlab has provided a short-cut to get them all done. Note the small green arrow pointing to the right in the tool bar at the top of the GI workspace:

Clicking that arrow will accomplish all three.

The primary function

As we will see, you determine the specific actions of your GUI by altering the subfunctions, but you will not need to alter the primary function. Indeed, Matlab includes a strong suggestion at the end of the primary function that you “DO NOT EDIT” this function, as you can see in the code for the primary function of `MyVeryOwnGUI.m`, which we have included below:

```
function varargout = MyVeryOwnGUI(varargin)

% MYVERYOWNGUI M-file for MyVeryOwnGUI.fig
%      MYVERYOWNGUI, by itself, creates a new MYVERYOWNGUI or raises the existing
%      singleton*.
%      
%      H = MYVERYOWNGUI returns the handle to a new MYVERYOWNGUI or the handle to
%      the existing singleton*.
%      
%      MYVERYOWNGUI('CALLBACK', hObject, eventData, handles,...) calls the local
%      function named CALLBACK in MYVERYOWNGUI.M with the given input arguments.
%      
%      MYVERYOWNGUI('Property','Value',...) creates a new MYVERYOWNGUI or raises the
%      existing singleton*. Starting from the left, property value pairs are
%      applied to the GUI before MyVeryOwnGUI_OpeningFcn gets called. An
%      unrecognized property name or invalid value makes property application
%      stop. All inputs are passed to MyVeryOwnGUI_OpeningFcn via varargin.
%      
%      *See GUI Options on GUIDE's Tools menu. Choose "GUI allows only one
%      instance to run (singleton)".
%      
%      See also: GUIDE, GUIDATA, GUIHANDLES
```

Multiple versus Singleton Instances

Except for the fact that it uses, as a function name, the name that you choose for your .fig file, the code for the automatically generated primary function is always the same, except for the date/time in the comments and one other one possible difference. The statement `gui_Singleton = 1` may be changed to `gui_Singleton = 0`. If it is the former case, then you cannot have simultaneous instances of your GUI running simultaneously. You can issue the command `MyVeryOwnGUI` as many times as you wish, but only the first one will produce a figure. Your GUI will always run as a “singleton”. If it is the latter case, then each command will produce an additional instance of the GUI, and the multiple instances are all simultaneously usable. You can change this line to yourself to `gui_Singleton = 0` with the editor, but there is also a way to specify that you want to allow multiple instances directly within `guide` by clicking on Tools at the top, clicking GUI Options..., and unchecking the box that says GUI allows only one instance to run (singleton). If you do that, then, when Matlab first generates the M-file, you will find `gui_Singleton = 0`. Let’s change it to zero with the editor and invoke our little GUI two more times in the Command Window:

```matlab
>> MyVeryOwnGUI
>> MyVeryOwnGUI
>> MyVeryOwnGUI
```

The result will appear at first to be only one GUI, but that is because they are all generated in exactly the same place and are exactly the same size. By moving them with the mouse, we can see all three of them, as shown here.
Adding Functionality to the GUI

Functionality is provided to the GUI by changing the subfunctions. The first two subfunctions that follow the primary functions are just as standard as the primary function. Like the primary function, they incorporate the name of the figure file:

```matlab
% --- Executes just before MyVeryOwnGUI is made visible.
function MyVeryOwnGUI_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to MyVeryOwnGUI (see VARARGIN)

% Choose default command line output for MyVeryOwnGUI
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes MyVery OwnGUI wait for user response (see UIRESUME)
% uiwait(handles.figure1);

% --- Outputs from this function are returned to the command line.
function varargout = MyVeryOwnGUI_OutputFcn(hObject, eventdata, handles)
% varargout    cell array for returning output args (see VARARGOUT);
% hObject      handle to figure
% eventdata    reserved - to be defined in a future version of MATLAB
% handles      structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;
```

You will sometimes find it necessary to make changes in the first subfunction, and, rarely, in the second. However, you will always find it necessary to make changes to the third and following subfunctions. These are the callback functions. As explained above, these functions are called when events occur in the GUI figure. When such an event occurs, the primary function (n.b., not a callback function!) is called by Matlab. Unlike the call that we give in the command window to activate an instance of the GUI, however, the call that Matlab makes when an event takes place includes arguments. The
first of these arguments is the name of the callback function associated with the object in the figure in which the event took place. Matlab finds the name of that callback function in the .fig file for the GUI. The name of the callback function is also shown in the Property Inspector in the field called Callback. In fact, the call of the primary function is given there verbatim complete with the arguments that Matlab is to use.

Callback functions are called by other functions when their names are passed through an input argument into the calling functions during execution. When an event takes place in the GUI (mouse click, mouse moves, mouse released, or keyboard activity), a callback function is called by the function gui_mainfnc. That function is called by the primary function, and when it does, it passes to gui_mainfnc the name of the subfunction that is to be called. The full sequence is as follows:

1. The user causes an event with the mouse or keyboard.
2. Matlab’s GUI system detects the event and calls the primary function, which is in our current case MyVeryOwnGUI, passing as an argument the name of the callback function
3. The primary function calls gui_mainfnc,
4. gui_mainfnc calls the callback function.
5. The callback function executes.

The name of a callback function is determined by the tag (some of which we changed by using the Property Inspector) of the object with which it is associated. Here are the callback functions in MyVeryOwnGUI.m:

```matlab
% --- Executes on button press in Red_Button.
function Red_Button_Callback(hObject, eventdata, handles)
% hObject  handle to Red_Button (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles  structure with handles and user data (see GUIDATA)

% --- Executes on button press in Green_Button.
function Green_Button_Callback(hObject, eventdata, handles)
% hObject  handle to Green_Button (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles  structure with handles and user data (see GUIDATA)

% --- Executes on button press in Blue_Button.
function Blue_Button_Callback(hObject, eventdata, handles)
% hObject  handle to Blue_Button (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles  structure with handles and user data (see GUIDATA)

% --- Executes on button press in Purple_Button.
function Purple_Button_Callback(hObject, eventdata, handles)
% hObject  handle to Purple_Button (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles  structure with handles and user data (see GUIDATA)
```

Note that the name of each callback function incorporates the name of the tag of the object that triggers it. It can be seen that we changed the tag of each of our four buttons to
give them names that we can easily remember as associated with the color that they are supposed to produce.

It’s time to make the Red button do what we want it to do: change the background color of the figure to red. To do that, we need to put a command inside the callback function for that button, which is the function named Red_Button_Callback. The command must change a property of the figure. The name of the property is probably “Color”, but it might be “BackgroundColor”. By the way, unlike the situation with variable names, function names, and file names, capitalization is of no consequence with property names. Thus, color, COLOR, cOlOr, and Color are all treated equally. To determine that name of the color property for the figure, we can consult Guide’s Property Inspector. We double click anywhere in the background of the figure (the grid area). There is no BackgroundColor, so we guess (correctly) that the property Color, is what we want. In order for the callback function to change the color of the figure, it must first obtain the handle for the figure. To write code that will do that, we need to know the Tag of the figure. Looking further down the property list, near the end, we find Tag, and we see that the name is figure1. We now have all the information we need to write code to change the figure’s color. The command that will do it is

```matlab
set(handles.figure1,'Color',[1 0 0]);
```

As mentioned above, handles is a struct whose field names are the tags of all the objects in the figure, including the figure itself. A copy of that struct is passed as an input argument to each callback function along with two other arguments, hObject, which is the handle of the object that triggered the callback and eventdata, which is an unused argument that Matlab has been reserving (for years) for possible future use. The value of eventdata is always (up to this writing) the empty matrix. (We will make no use of it in order to maximize the chance that our dandy GUI will work in futures version of Matlab that use it).

The value that we are giving to the Color property is the vector [1 0 0], which means: “Give the red component full intensity and give the green and blue components zero intensity.” (The string 'Red' in place of [1 0 0] has the same effect.).

After we add this command, the callback for the red button looks like this:

```matlab
function Red_Button_Callback(hObject, eventdata, handles)
% hObject  handle to Red_Button (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles  structure with handles and user data (see GUIDATA)
set(handles.figure1,'Color','r');
```

We add a similar command to Green_Button_Callback,

```matlab
set(handles.figure1,'Color',[0 1 0]);
```
and similar commands for the other two buttons, changing the intensities to \([0 \ 0 \ 1]\) for blue and \([0.7 \ 0 \ 1]\) for purple. Now when we activate our GUI and click on the Red Button, the result looks like this:

![GUI Interface](image)

Pressing the other buttons changes the color as expected. We can press them in any order as many times as we wish. The callbacks will be called, and the color will change.

We’ve done it! MyVeryOwnGUI is working. But…This color is pretty intense. It would have been nice if we had included a capability to be able to dim it a bit. Fortunately, we can now go back and add objects that we didn’t think of when we started, and we can add callbacks for them without losing the work that we’ve already done. We simply open the GUI figure file in guide, add an object or objects, set their properties with the Property Inspector, and save as before. The figure file and M-file will be modified, and we will find a brand new callback function at the end of the M-file right after the `Purple_Button_Callback` function. In the next section will look at the slider, which is just what we need for this dimming task.

**The Slider**

The slider is an object that looks like a horizontal or vertical scroll bar. It allows the user to adjust a number by moving a rectangular slider, which shifts from left to right within a “trough”. Let’s put a slider at the bottom of our GUI. To do that, we click on the Slider button in the component pallet at left and drag out a rectangular shape, as we did for the
buttons. The result looks like this (we’ve turned on the Show-names-in-component-palette again so that you can easily identify the Slider button):

There are three ways to move a slider: The first is to click on the slider and move it before releasing. The second is to click somewhere in the trough, and the third is to click on either of the darkened squares with little arrows on them at the left or right end. In second or third case, the slider will jump a set distance toward the clicked point.

Of course, we do not have a working slider yet. We have laid it out, but we need to set its properties and then work on its callback function. First, we’ll set the properties. If we double click the slider that we have created, we will see its property list in the Property Inspector. As in most cases, we first change the tag to something appropriate for the task that the object is to perform. In this case we will chose “intensity_slider”. The only other properties that we need to work on are Max, Min, SliderStep, and Value. The value property is the key to the operation of the slider. When the user moves the slider, only one thing happens, its Value property changes. It will increase or decrease, but it will remain within the range Max-to-Min. We are going to use the Value as a multiplier to reduce the color intensity from its full value to lower values all the way down to zero. So we want the range to be 0 to 1. These are the most common values, and so they are already set for us by Matlab. So, no need to change Min and Max. The slider step is easy too. Its value is a two-element vector, and the initial value is [0.01 0.1]. The first number is the percent change of the value when we click on one of the arrows at either end; the second number is the percent change when we click in the trough. These values are fine, so no need to change SliderStep. We want our initial value to be 1, so we have to change the value of Value from Matlab’s default of 0. To do that we click on the icon to the left of the [0.0]. That icon represents a vector because we can in fact specify that the slider simultaneously changes every element in a vector by the same factor. When we click on it we see this window
Chapter 11  Graphical User Interfaces

It shows that the default value is the scalar 0. We change it by clicking it and typing in 1. (If we wanted to add elements to this vector, we could do that by clicking the Append button; to remove elements, we click Delete.) When we are done, we click OK, and then close the Property Inspector.

Now, we are ready to generate a new figure file and new M-file and then go to work on the new callback function that Matlab will put in the M-file. We click the green arrow in the GUI toolbar. This little box pops up:

We want to save changes (virtually always), so we click Yes (or hit Enter) and the GUI pops up.

When the GUI pops up the slider is now sitting at the right, because we gave an initial value of 1 and that is the maximum of the range (which runs from left to right). We can click on the slider and move it, but nothing happens because we have not changed the callback. If we open the MyVeryOwnGUI.m in our editor, we find this new code at the end:

```matlab
% --- Executes on slider movement.
function intensity_slider_Callback(hObject, eventdata, handles)
% hObject handle to intensity_slider (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)
% Hints: get(hObject,'Value') returns position of slider
% get(hObject,'Min') and get(hObject,'Max') to determine range of slider

% --- Executes during object creation, after setting all properties.
function intensity_slider_CreateFcn(hObject, eventdata, handles)
% hObject handle to intensity_slider (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)
% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject, 'BackgroundColor', [.9 .9 .9]);
end
```
This time there are two new functions, the callback, as expected, and a second function, `intensity_slider_CreateFcn`. This second function is called only once, when the slider is created upon GUI activation.

Now we work on the slider’s callback. Matlab has included a helpful comment (Hints: ...) that tells us how to read the value of slider and its maximum and minimum values. All we need is the value, so we add the command,

```matlab
intensity_factor = get(hObject,'Value');
```

Now we want to multiply `intensity_factor` times the most recent color that was selected by the buttons. Each time the slider is adjusted, this callback will be called to do that. Let’s imagine that we have clicked the red button to produce the bright red background that we saw above. Now we move the slider two times. Suppose the `intensity_factor` changes from one, down to, say, 0.5, and then from 0.5 back up to 0.75. We want to see the color go from bright red to half-red and back to ¾ red. In order to do that, we need to multiply the red color vector, which is `[1 0 0]` first by 0.5 and then by 0.75. That must be accomplished by commands in this callback. Consider the following code, which seems, at first glance, to do what we want:

```matlab
current_color = get(handles.figure1,'color');
new_color = intensity_factor*current_color;
set(handles.figure1,'color',new_color);
```

Consider the first time the slider is moved after the red button is clicked. The slider’s Value property is set to 0.5 and the callback is called. The first command reads the current color—`[1 0 0]`. The second one calculates the new color—`0.5*[1 0 0]`, which equals `[0.5 0 0]`, and the third one writes the new color into the color property of the figure. As soon as the new color is written into the color property of the figure, the background will change on the screen. All seems well. Unfortunately, however, if we move the slider again, the results will not be what we want.

Suppose that the second motion of the slider results in a slider value of 0.75, as we imagined above. The slider’s callback will be called again, and these three commands will be executed again. The first one will read the current color, which is still `[0.5 0 0]`. The second will set the new color to `0.75*[0.5 0 0]`, which equals `[0.375 0 0]`. What we wanted instead was to multiply 0.75 times `[1 0 0]`, which would give `[0.75 0 0]`. The problem is that we have not stored the original color that was set when the color button was clicked. We need to do that in the callback of each of the four buttons, and we need to be able to read that value from within the slider’s callback.

**Sharing values among callbacks**

We are presented with a problem. We need to set a value in one callback, for example `Red_Button_Callback`, and read that value later in another callback—namely `intensity_slider_Callback`. As we have learned in our study of functions, each
function’s variables have only local scope, so the value we set in one function cannot be accessed by a second function unless the value is returned by the first function to the second. Since these functions are not calling each other that method is impossible. Furthermore, the definitions of the functions are beyond our control. These callback functions do not return anything! We learned a method for sharing values by means of the global attribute, but that option is a bad one because it means that, if the user happens also to declare a variable with the same name to be global in, say, the Command Window, commands there would interfere with the behavior of the GUI. Thus, we have a difficult problem.

Fortunately, Matlab has anticipated both the need for sharing values among callbacks and the danger of using the global attribute. It has provided a safe and relatively easy alternative. Since the call of each callback function includes the argument handles, we can store the values that we want to share inside it. To see how this scheme works, we will show a modification of the Red_Button_Callback and the intensity_slider_Callback that implements sharing through handles (we have omitted the comments to save space):

```matlab
function Red_Button_Callback(hObject, eventdata, handles)
    set(handles.figure1,'Color',[1 0 0]);
    (1) handles.current_color = [1 0 0];
    (2) guidata(hObject, handles);

function intensity_slider_Callback(hObject, eventdata, handles)
    intensity_factor = get(hObject,'Value');
    (3) new_color = intensity_factor*handles.current_color;
    set(handles.figure1,'color',new_color);
```

The numbers indicate the new commands. (1) stores the red value in a new field of handles, called current_color. As we have seen in Chapter 8, if this field is already there, it will be used; if not, it will be added to the existing fields of handles. (2) is the command that makes the value accessible to other callback functions. It is a call to a function, guidata, which stores the value of its second argument in the figure file. The data that it saves in that file is simply a struct. The struct must have one field for each object in the figure, but it can have additional fields as well. Once we have added this new field and stored a value in it, each callback will receive it as an argument when it is called.

To understand fully the need for the call to guidata, we must remember that the variable handles, like any input argument of any function in Matlab, is a local variable. When the function is called, a value is copied into handles, but when the function exits, nothing is copied back (a few languages do support this behavior via so-called “in-out” arguments, but not Matlab). Thus, adding a field and/or changing the value of a field in the local variable handles will have no effect on any variable or value outside the function. That is why the call to guidata is required.
Continuing with our analysis of the code above, we come to (3), which retrieves the color that was most recently set by one of the buttons (in the case we are analyzing that would be the red button). It gets it from the current_color field of handles, which is a local variable (i.e., not the same variable as that in a button’s callback function), but which received its value when it was called by gui_mainfcn.

If we run our GUI now it works as expected—almost! A new problem has shown up. When we click a button and then move the slider to, say the middle, which produces a 50% intensity, all works properly. However, when we click a button after we have moved the slider, we discover a problem. The button click produces a color at 100% intensity, but the slider is showing 50%. If we move the slider the intensity is adjusted properly, but until we do, the GUI is in a confusing state. To solve that problem we could do one of two things: (1) Move the slider all the way to its maximum position when we click a button, or (2) Use the current intensity factor when we click a button. We will choose the former, only because it shows how to change the state of one object from within the callback of an other object. Here is the modified callback function for the red button:

```matlab
function Red_Button_Callback(hObject, eventdata, handles)
set(handles.figure1,'Color',[1 0 0]);
handles.current_color = [1 0 0];
guidata(hObject, handles);
set(handles.intensity_slider,'Value',1);
```

The only new command is the last one. We simply set the Value to 1 for the slider. When that is done, the slider moves immediately to the right on the screen. We could have placed this command anywhere within this function. We put it last to show that it does not have to come before the call to guidata. It is not necessary to employ guidata for this change because it is not handles that we are changing, but the property of an object whose handle is stored in handles.intensity_slider.

Our GUI now works properly.

The Static Text Box

But we’re not done yet! The GUI works properly, but we need to add something: a label to inform the user of the purpose of that slider. Our buttons suggest that they will produce a color, but the slider will seem mysterious. We will add the label, “Intensity Adjustment”. We can add a label anywhere in the figure by means of a Static Text Box. To do that, we first click on the bottom right of the background and stretch it down a bit to provide more room, and then we click on the button in the component pallet that says Static Text and drag a rectangle on the workspace just below the slider. When we release the mouse button, we see this,
We double click our new static text box to bring up the Property Inspector. Unlike the other property lists that we have seen, a static text box is not automatically given a callback, because, as the name implies, it is not meant to be active. It is used simply to provide static information. Its visibility can be changed, its position can be changed, its message can be changed, and other things can be changed, all by changing its properties, but typically, as in our case, all we need to do is specify that its string be “Intensity Adjustment”, and choose the right size and weight for the font. We choose a FontSize of 12.0, and for FontWeight we choose bold. With this increased font size it is necessary to widen the box and re-center it.

After we make these simple changes and save our GUI, we are done (really!). The image below shows it in action after the red button has been clicked and the intensity has been reduced. We could do much more by adding other objects and other callbacks, but the basic ideas are the same: Drag a component from the component pallet, modify the
properties, save the figure file and the M-file, and modify the callbacks. As we have seen above, this process tends to be iterative. Surprises always arise. When they do, we will typically find it necessary to alter one or more callbacks. Occasionally, we will find that we need to add a component or change a property, but most of the time is spent with programming.

11.4 Matlab Help on GUIs

Most of the standard objects provided by the component pallet are listed in Table 19. We have discussed the Pushbutton and the Slider in some detail and given an example of each. To learn about the other objects, you should consult Matlab’s detailed Help facility. To do that, click Help in Matlab’s taskbar and click Product Help. The Help tool will appear with the Help Navigator at the left. Click the “plus” sign to the left of MATLAB and then click the “plus sign to the left of Creating Graphical User Interfaces to expand a topic to reveal its subtopics. There you will see these topics:

- Introduction to Creating GUIs
- Creating GUIs with GUIDE
- Creating GUIs Programmatically

There is a wealth of information among these three topics. To get descriptions of the all the components in the component pallet, expand Creating GUIs with GUIDE, then expand Laying Out a GUIDE GUI, then expand Adding Components to the GUI, and click on Available Components.
Chapter 11 Appendix
Handy Commands for GUIs

Here are some functions that will often come in handy when writing GUIs

**axes**
AXES(H) makes the axis with handle H current.

It is helpful when plotting is to be done in a GUI (but see subplot below!). When the GUI was set up, an Axes component would have been selected from the component pallet and placed somewhere in the GUI. While the GUI is running, plotting can be directed to that component by means of the axes command. For example, suppose the plot is to appear in an axes object whose tag is foo. Then the following commands would do it:
```matlab
axes(handles.foo);
plot(x,y);
```

**close**
CLOSE closes the current figure window

It is the command that will stop the GUI and close its window (i.e., its figure). This is the command that you should put in the callback function of the GUI’s Quit button.

**num2str**
T = NUM2STR(X) converts the matrix X into a string representation T with about 4 digits and an exponent if required.

It is helpful when a number is to be displayed in a textbox. For example if a textbox whose tag is foo must display the value in the variable v. Then the following commands would do it:
```matlab
text = num2str(v);
set(handles.foo,'string',text)
```

**str2double**
X = STR2DOUBLE(S) converts the string S, which should be an ASCII character representation of a real or complex scalar value, to MATLAB's double representation. The string may contain digits, a comma (thousands separator), a decimal point, a leading + or - sign, an 'e' preceding a power of 10 scale factor, and an 'i' for a complex unit.

It is helpful when a number in a textbox needs to be read and converted to a number for use in some function. For example, if a textbox whose tag is foo, contains the string ‘4’, and we need to assign that number to a variable v. Then the following commands would do it:
```matlab
text = get(handles.foo,'string')
v = str2double(text);
```
**subplot** (new use)
SUBPLOT('position',[left bottom width height]) creates an axis at the specified position in normalized coordinates (in the range from 0.0 to 1.0).

You have seen the command subplot before, but this is a new use. When you wish to use subplot in a GUI, it is important to note that subplot use previously existing axes, so it does no good to layout axes in the GUI layout tool. Subplot produces its own axes and places them somewhere within the GUI figure. The next call to `plot` (or to `plot3`, or `hist`, or `image`, or any other plotting command) causes plotting to go into that set of axes.

This special use of subplot allows you to be very specific about where in the figure the plot is placed. As can be guessed from the template command above, you specify the horizontal and vertical positions of the lower left corner, and the width and height. Interestingly, all four of these dimensions are given as fractions of the figure dimensions. For example, the following commands

```matlab
subplot('position',[0.3, 0.2, 0.4, 0.6])
plot([1,10],[1,10])
```

would cause a line to be plotted, as shown below, in a rectangular area whose lower left corner is 0.3 of the width of the figure from the left of the figure, 0.2 of the height of the figure from the bottom of the figure, with a width equal to 0.4 of the figure width, and a height equal to 0.6 of the figure height.

If you wish to place, say, three plots on the right side of the GUI, you might, for example, use these commands

```matlab
subplot('position',[0.5, 0.1, 0.4, 0.25])
plot([0:10],[0:10],'r')
subplot('position',[0.5, 0.4, 0.4, 0.25])
plot([0:10],sin([0:10]),'g')
subplot('position',[0.5, 0.7, 0.4, 0.25])
```
which produce the following result:

```
plot([0:10],cos([0:10]))
```

**uiputfile**

UIPUTFILE Standard save file dialog box.

```
[FILENAME, PATHNAME, FILTERINDEX] = UIPUTFILE(FILTERSPEC, TITLE)
```

displays a dialog box for the user to fill in and returns the filename and path strings and the index of the selected filter.

It is useful when it is time to save information into a file. This command brings up a standard dialog box provided by the operating system, and allows the user to choose any directory on the file system into which to save the file. For example, if the following command is given:

```
[filename, pathname] = uiputfile('*.','Save file as')
```

Then a dialog box will open up that looks something like this (under Windows XP):
If the user types the name Popcorn.wav in the File name box and clicks Save, then the function will return two values:
filename = Popcorn.wav
pathname = C:\Documents and Settings \My Documents\MATLAB\

These two values can then be used to write values into a file. For example, the GUI might use the Matlab-provided function \texttt{wavwrite}, to save audio information in a file:
\texttt{wavwrite(audio\_array, sampling\_rate\_p,16,filename)}.

This is the typical use of \texttt{uiputfile}: Despite the fact that the user clicked Save during the call to \texttt{uiputfile}, nothing is saved while this function is running! The call to \texttt{uiputfile} does nothing but return a file name and pathname chosen by the user by means of the dialog box. It does not create a file or save anything into it! The file creation and the writing into the file, or the writing into a pre-existing file, is done by some other function.
Concepts from this chapter

- Computer Science:
  - object-oriented programming
    - objects
      - property
      - property value
  - graphical user interface (GUI)
    - event-driven programming
    - handle
    - callback

- Matlab:
  - graphical user interface
  - handle
  - properties
    - get and set
  - guide
    - GUI workspace
    - component pallet
    - Property Inspector
  - standard objects:
    - Pushbutton
    - Slider
    - Static Text Box
    - Access Matlab’s help facility on:
      - Toggle Button, Radio Button, and Checkbox
      - Edit Text Box, Static Text Box, List Box
      - Popup Menu
      - Axes
  - handy commands for use with GUIs
    - axes, close, num2str, str2double, subplot (new use), uiputfile
Chapter 12

Symbolic Mathematics

When mathematicians work, they use symbols, such as $x$ and $y$, to represent numbers and the functional notation $f(x)$ to represent a function of $x$. We have used the same symbols in Matlab, and yet the meanings are subtly different. The Matlab symbols $x$ and $y$ refer to floating point numbers or integers of the types we considered in Chapter 8, whose ranges and accuracies are limited, and $f(x)$, $\sqrt{x}$, etc. refer to functions that operate on these numbers. A common feature of all the numbers that can be represented as floating point numbers or integers is that they are rational numbers, i.e., numbers that are equal to ratios of integers.

The $x$ and $y$ of mathematics, on the other hand, refer to real (or complex) numbers whose values are often defined only by being solutions to equations. These solutions are sometimes not rational, as for example the solution to the equation, $x^2 = 2$, which we can represent by $x = \pm \sqrt{2}$ or $x = \pm 2^{1/2}$. In Matlab we can easily set $x = 2^{(1/2)}$ or $x = -2^{(1/2)}$, but the meaning is not the same. The result of Matlab’s calculation of $2^{(1/2)}$ is approximately equal to $2^{1/2}$, but it is certainly not exact because $\sqrt{2}$ is an irrational number and cannot be represented to perfect accuracy with any floating point number. The statement $x = \pm 2^{1/2}$, on the other hand, means “$x$ is equal to the number that solves the equation, $x^2 = 2$”. The equation is the central feature here, not the number.

Furthermore, suppose a mathematician is interested in the function $f(x) = x^2$ and wants to know the rate at which $f(x)$ changes with $x$. The rate of change of $f(x)$ with respect to $x$ is the derivative of $f(x)$, expressed symbolically in standard calculus notation as

$$\frac{df(x)}{dx}.$$

We learn from calculus that the derivative is another function, in this case,

$$\frac{df(x)}{dx} = 2x.$$

In this case there is no number involved at all. We have, by taking a derivative, manipulated one function to produce another function. There are many such relationships between functions of importance in mathematics. For example, if we treat $2x$ as a function of $x$, then its integral gives a family of functions:

$$\int 2x dx = x^2 + c,$$
each with a different value for $c$. As another example of the manipulation of functions, if
\[ g(x) = x^2 - 2x + 2, \]
then it can be shown that \[ g(x) = p(x) \times q(x), \]
where \[ p(x) = x + 1 \]
and \[ q(x) = x - 2. \] Here we have “factored” a function, or “expression”, into two
functions, or “terms”, whose product is equal to the original function.

In each of these cases, manipulations of equations and functions are the central
problems; specific numbers play only a supporting role.

Such manipulations cannot be done by the Matlab with which we have become familiar
in the previous chapters. That Matlab deals with rational numbers and with functions and
programs that manipulate rational numbers to produce other rational numbers. What we
have seen is the numerical side of Matlab. Numerical mathematics is the greatest strength
of Matlab, and it is the most well known part of Matlab. Indeed it is the only side of
Matlab that most people know. However, Matlab has another side—symbolic
mathematics. **Symbolic mathematics** is the manipulation of symbols according to
mathematical rules. It is the sort of thing that we learn first in algebra and then in
calculus. We learned in algebra, for example, how to factor a polynomial, such as \( g(x) \)
above, and we learned how to differentiate and integrate functions in calculus.

In this chapter we will open Matlab’s symbolic capabilities by giving the command
**syms**, and once we are there, we will use those capabilities to manipulate equations and
functions. We will also learn how, after we have got the function we want, to plug in
values for variables to calculate numerical answers to numerical problems. The symbolic
approach is quite different and will take the reader a while to assimilate. The best way to
do that is to repeat the examples shown in the chapter.

(Note: The examples in this chapter should be carried out in order during the same
command session because some of them depend on the fact that previous ones have been
done. If you have to start in the middle, note that in some cases a **syms** statement is
needed and also, possibly, an assignment statement or two. You can look back to the
earlier examples to find the missing statements.)

You should use Matlab’s **help** command on the functions discussed below as a further
means for understanding them. However, you should know that when you ask Matlab for
help, you will, in some cases get help on the wrong function! That happens when Matlab
has two functions with the same name, one of which is called when non-symbolic
arguments are given to it and the other of which is called when symbolic arguments are
given to it. Matlab uses the computer-science term “overloaded method” to describe a
function that has both a numerical and a symbolic definition. The word “method” here
means simply “function”. The use of the word “method” to mean function is common in
object-oriented programming languages, such as C++ and Java. Matlab, however, while it
does support some object-oriented functionality, particularly in its GUI development (See
Chapter 11) is not an object-oriented language.
As an example, the function `factor` is an overloaded method. It exhibits one behavior when used in numerical operations and a totally different behavior when used in symbolic operations. Its behavior is determined by whether its arguments are numerical arguments or symbolic arguments. If, for example you type `help factor` in the Command Window, you will see the help text for the non-symbolic version of `factor`, but at the bottom you will find this:

**Overloaded methods**

`help sym/factor.m`

To get help for the symbolic version, type `help sym/factor` (the `.m` is optional). This technique: `help sym/function_name`, works for any symbolic function.

In versions of Matlab previous to Matlab 7.7.0 (R2008b), symbolic mathematics and variable-precision arithmetic (see next section) were handled by an “engine”, which is a program running in the background. This engine is called Maple, and it is available as a separate stand-alone product from Waterloo Maple, Inc. ([www.maplesoft.com](http://www.maplesoft.com)). Beginning with version 7.7.0 (R2008b), these functions are handled by a different engine, called “muPAD”. MuPAD is not available as a separate product, but it is also runs in the background of other mathematical products. The MathWorks, Inc., ([www.mathworks.com](http://www.mathworks.com)), which is the company that makes and sells Matlab, paid Waterloo Maple, Inc. for the rights to incorporate Maple into Matlab, but in 2008 it purchased the company that formerly distributed muPAD. The basic functionality has not changed significantly (after all, the derivative of $x^2$ is still $2x$), but with the previous versions, purchase of Matlab did not include symbolic mathematics or variable-precision arithmetic. The user had to pay separately for a symbolic toolbox. Beginning with Matlab 7.7.0 (R2008b), they are included with all Matlab installations.

In addition to Mathworks and Waterloo Maple, another company that provides symbolic mathematics capabilities is Wolfram Research, Inc. ([www.wolfram.com](http://www.wolfram.com)), which makes and sells *Mathematica*. If symbolic mathematics is the major thrust of your work, then you might want to look into these other programs.

### 12.1 Variable Precision Arithmetic

Before we get started with equations and functions, we will learn about Matlab’s variable-precision floating-point arithmetic facility, which is included as part of its symbolic mathematics package (even though it is not symbolic mathematics).

When Matlab does a calculation, for example, when it calculates the value of pi, it uses a special internal floating-point number representation that limits the accuracy of the calculation. It does this because it is faster to do it this way. It is faster because the

---

computer on which Matlab runs has special hardware for calculating that works with this representation. If you wish to carry out calculations with greater accuracy, also known as greater “precision”, you can use Matlab’s facility for variable precision arithmetic. It is provided through the function \texttt{vpa}.

Here are some examples that show how it works. The function \texttt{vpa} takes two arguments. The first argument is a string containing the expression that is to be evaluated\textsuperscript{15}, and the second argument is the number of digits of accuracy requested:

\begin{verbatim}
>> vpa('pi',50)
ans =
3.1415926535897932384626433832795028841971693993751

>> vpa('exp(1.2)',50)
ans =
3.3201169227365472380597566370852291584014892578125
\end{verbatim}

That second argument defaults to 32:

\begin{verbatim}
>> vpa('pi')
ans =
3.1415926535897932384626433832795
\end{verbatim}

The default can be changed with the function \texttt{digits}:

\begin{verbatim}
>> digits(10)
>> vpa('pi')
ans =
3.141592654
\end{verbatim}

Once a variable is given a variable precision value, it passes it to other variables in assignment statements:

\begin{verbatim}
>> x = vpa('pi');
>> y = x
y =
3.1415926535897932384626433832795
\end{verbatim}

\textit{Evaluating expressions with variable precision}

We can evaluate any expression with high precision by giving \texttt{vpa} a string representing the expression, instead of giving it the expression directly, as for example,

\begin{verbatim}
>> vpa('log(2) + 1')
ans =
\end{verbatim}

\textsuperscript{15} An exception is that, if the expression is not a string but equals 3.14159265358979, which is $\pi$ to double precision, \texttt{vpa} behaves as if its first argument were in fact \texttt{'pi'} (i.e., complete with the quotes).
which gives an answer correct to 32 digits. If we omit the quotes, the \texttt{log} function operates without high precision returning different number, which is then printed by \texttt{vpa} to 32 digits, the last 17 of which (underlined below) are different from the high-precision version:

\begin{verbatim}
>> vpa(log(2) + 1)
ans =
1.69314718055994528622676398299518
\end{verbatim}

We will see another way to persuade \texttt{vpa} to evaluate an expression at high precision in Section 12.3.

\section*{12. 2 Symbolic Expressions}

To use Matlab’s facility for doing symbolic mathematics, it is necessary\footnote{Necessary, that is, for doing it in the command window or an M-file. However, there is an alternate approach which provides beautifully type-set mathematics, which can be accessed by typing \texttt{mupad} in the command window. To learn it, click its Help button and then select Getting Started.} to declare at least one variable to be “symbolic”. The best way to do that is to use the \texttt{syms} declaration statement:

\begin{verbatim}
syms a b x y;
c = 5;
E = a*x^2 + b*x + c;
\end{verbatim}

The \texttt{syms} statement turns all the variables listed with it into symbolic variables and gives each of the variables a value that is equal to its own name! Thus, the value of \texttt{a} is \texttt{a}, the value of \texttt{b} is \texttt{b}, etc. The variable \texttt{E} is now symbolic because it was assigned to be equal to an expression that contained one or more symbolic variables. Its value is \texttt{a*x^2 + b*x + 5}.

Typing \texttt{whos} will show which variables are symbolic:

\begin{verbatim}
» zebra = 10;
» whos
Name      Size           Bytes  Class       Attributes
E          1x1            142    sym object
a          1x1            126    sym object
b          1x1            126    sym object
  c         1x1             8    double array
\end{verbatim}
If we now make \( c \) symbolic, \( E \) will not be affected:

\[\begin{align*}
\text{>> syms } & c \\
\text{>> } & E \\
\text{E } & = \\
x+y+a+b+5
\end{align*}\]

If we give the same assignment again for \( E \), after \( c \) has been made symbolic, its value will now be affected,

\[\begin{align*}
\text{>> } & E = a*x^2 + b*x + c \\
\text{E } & = \\
a*x^2+b*x+c
\end{align*}\]

12.3 Symbolic Evaluation

Symbolic evaluation of an expression involves producing an equivalent expression that has some desirable property. For example, the equivalent expression may be simpler. The notion of what is “simpler” is not always obvious, but Matlab (i.e., Maple) uses some rules that tend to produce a shorter expression (i.e., fewer symbols). The expression, \( \sin(\pi) \), can be simplified to 0, for example. Another example of simplification is the factoring of a polynomial. For example, the polynomial \( x^4 + 4x^3 + 6x^2 + 4x + 1 \) can be factored into \((x+1)^4\). The opposite operation is expansion, in which a factored, or partially factored, polynomial is written out in terms of powers of the variable. All three of these ideas are provided by the symbolic Matlab functions, \textit{simple}, \textit{factor}, and \textit{expand}.

\[\begin{align*}
\text{>> } & \sin(\pi) \\
\text{ans } & = \\
1.2246\text{e-016}
\end{align*}\]

\[\begin{align*}
\text{>> } & \sin(\pi/3) \\
\text{ans } & = \\
0.86603
\end{align*}\]

\[\begin{align*}
\text{>> syms } & \pi \\
\text{>> } & \sin(\pi/3) \\
\text{ans } & = \\
1/2*3^{(1/2)}
\end{align*}\]
Here we see that the numeric side of Matlab found an approximate floating point number for the sine of one-third pi, whereas the symbolic side of Matlab found a simpler expression for it, namely $\sqrt{3}/2$.

If we want a numerical version of the symbolic answer we can get it by means of the conversion function double (See Chapter 8),

```matlab
>> double(ans)
an =
  0.86603
```

The result is the numerical approximation to $\sqrt{3}/2$ represented by the floating-point number 0.86603. If we want more digits, we can get that by using `vpa`,

```matlab
>> vpa(sin(pi/3),25)
an =
  .8660254037844386467637230
```

The astute reader may have noticed that we evaluated the expression `sin(pi/3)` at high precision without putting it in the quotes that were stipulated earlier. High precision was gotten in this case without the quotes because we had declared `pi` to be a symbolic constant. When the function `sin` is given a symbolic constant or a symbolic variable, it performs a symbolic calculation and returns a symbolic result. That symbolic result is evaluated by `vpa` to the requested precision. No quotes are needed. The ability of sin to vary according to the type of its actual parameter is another example of polymorphism.

Sometimes there is no simpler form:

```matlab
>> sin(pi/7)
an =
  sin(1/7*pi)
```

and sometimes our idea of simpler is not the same as Matlab’s:

```matlab
>> syms x
>> (x+1)*(x+2)
an =
  (x+1)*(x+2)
```

In fact, when dealing with polynomials, we need to tell Matlab what we want. For example, if we wish to expand a product of polynomials into the standard polynomial form with the highest degree term first followed by terms of lower degree, we use the function `expand`:

```matlab
>> expand((x+1)*(x+2))
```
ans =
x^2+3*x+2

If we suspect that a polynomial can be factored, we use the function **factor**:

```matlab
>> factor(x^2+3*x+2)
an =
(x+1)*(x+2)
```

Here is another example of these two functions in action:

```matlab
>> expand((1+x)^6)
an =
x^6+6*x^5+15*x^4+20*x^3+15*x^2+6*x+1
>> factor(ans)
an =
(x+1)^6
```

The function **expand** is more versatile than direct polynomial expansion. It can expand trigonometric functions whose arguments are polynomials as well:

```matlab
>> expand(sin(x+y))
an =
sin(x)*cos(y)+cos(x)*sin(y)
>> expand(tan(x+y)^2)
an =
(tan(x)+tan(y))^2/(1-tan(x)*tan(y))^2
```

The function **simple** will look for a “simple” version of an expression, where the definition of simple is roughly “few characters required for the representation”. It prints out every version that if finds that requires fewer characters and returns one that has the fewest. It typically agrees with human intuition about what is simple. In the following example, we show only part of the output,

```matlab
>> simple( (sin(x))^2 + (cos(x))^2 )
simplify:
1
radsimp:
...
an =
1
```

**Simple** tries many methods for simplification. To suppress the printing of all the methods tried, use it in the form **f=simple(e)**, where **e** is the symbolic expression that
is to be simplified. The variable \( f \) will become a symbolic variable containing the simplified result.

```matlab
>> f = simple(x^3+(a-2*b)*x^2+(-2*a*b+b^2)*x+a*b^2)
f =
(x-b)^2*(x+a)
```

```matlab
>> g = simple(sin(a)/cos(a))
g =
tan(a)
```

```matlab
>> h = simple((sin(x) + cos(x))*(sin(x) + cos(x)) - sin(2*x))
h =
1
```

```matlab
>> r = simple(cos(a) + i*sin(a))
r =
exp(i*a)
```

The last example uses the complex equivalence, \( \cos(a) + i\sin(a) = e^{ia} \).

### 12. 4 Solving Equations

Suppose that \( E \) still contains the expression given it above, \( a*x^2+b*x+c \). Consider the following command:

```matlab
>> solve(E)
ans =
[ 1/2/a*(-b+(b^2-4*a*c)^(1/2))]
[ 1/2/a*(-b-(b^2-4*a*c)^(1/2))]
```

The command \texttt{solve(E)} assumes that you wish to solve the equation \( E = 0 \). If \( E \) contains the variable \( x \), then it solves for \( x \). In this case, there are two solutions. It gives the two solutions as two elements to a single symbolic vector. The assumption that \( x \) is the unknown is always made when there is an \( x \) in the equation, unless you give as a second argument the variable for which to solve:

```matlab
>> solve(E,c)
ans =
-a*x^2-b*x
```

This time there is only one solution. If there is an exact numerical solution to the equation, \texttt{solve} will give that instead of a symbolic expression. For example,

```matlab
>> F = 14*x^2+50*x-24;
```
Here again, if we wish to get a floating point version of the answer, we can use `double`,

>>> double(ans)
an =
 0.4286
-4.0000

Solving simultaneous equations

If you have two or more equations in two or more unknowns, you simply give all the equations to solve as arguments, followed by a list of the unknowns. For example:

```matlab
>> [xval,yval] = solve(x+3*y-7, 2*x+y, x, y)
xval =
-7/5
yval =
14/5
```

If the list of unknowns is omitted, `solve` will try to guess them. It starts with `x`, `y`, and `z`. `solve` can also be given its equations in the form of strings. Use `help` to learn more about `solve`.

We have learned previously how to solve simultaneous linear algebraic equations by using matrix division. We can do the same thing symbolically. Here is an example:

```matlab
>> b = [3; a+2];
>> A = [a*c, a; 3+c, a+c];
>> X = A\b
```

The use of `simple` is important. Without it, the result is very complicated. It does not appear to be equal to zero, but, upon simplification, everything cancels out. Matlab tends (using Maple) not to simplify unless it is explicitly commanded to do so.
12.5 Substituting Values into a Symbolic Expression

Once an expression involving symbols has been obtained, it is often desirable to substitute numeric values for at least some of the symbols. To substitute values for symbols in an expression use the function \texttt{subs}. For example,

\begin{verbatim}
>> syms a b c
>> subs(a^3+b^2-c,{a,b,c},{3,4,5})
ans =
   38
\end{verbatim}

Note the braces \{, and \}, above. Both the list of symbols and the list of the values to be substituted for them are cell arrays. This same function can be used to substitute symbolic expressions for some (or all) of the symbolic variables. Thus, for example,

\begin{verbatim}
>> subs(a^3+b^2-c,{a,b,c},{3,4,a+b})
ans =
  43-a-b
\end{verbatim}

12.6 Changing Symbols into Strings

Sometimes is desirable to have a string representing a symbolic expression. One purpose is to print out the expression. To change any symbolic expression into a string, use the \texttt{char} function.

\begin{verbatim}
>> syms x
>> E = sin(x);
>> Esstring = char(E);
>> fprintf('Here is the function: %s\n', Esstring);
\end{verbatim}

12.7 Plotting functions (as opposed to plotting vectors)

Symbolic functions are often plotted in books on mathematics to give an idea of their behavior. Such plots do not represent a discrete set of values, as in the plots we have done previously of one vector relative to another, but a continuous set. Matlab provides means to plot symbolic functions in this way through two functions \texttt{ezplot}, which is used for functions of one variable, and \texttt{ezsurf}, which is used for functions of two variables.
**Functions of one variable**

Here is an example of the use of `ezplot`:

```
>> ezplot(sin(x)/x)
```

Note that when x equals zero, both the numerator and the denominator of the expression \( \frac{\sin(x)}{x} \) are equal to zero. If we were to plot the vector \( \sin(x) ./ x \) versus \( x \), the numerical side of Matlab would give us a divide-by-zero error there. Its symbolic side (Maple), however, “knows” how to evaluate the expression by taking a limit as \( x \) approaches zero!

Ezplot can also be given the expression to plot as a string. A second, optional argument gives the range over which to plot the first argument. The default range is -2*pi to +2*pi.

**Functions of two variables**

Here is an example of the use of `ezsurf`:

```
>> ezsurf(sin(sqrt(x^2+y^2))/sqrt(x^2+y^2))
```

`Ezsurf`, like `ezplot`, takes its expression either as a string or as a symbolic expression. It plots a surface lofted above the \( x-y \) plane with the height (\( z \) value) equal to the value of the expression for the values of \( x,y \).

### 12.8 Power Series Approximations

The function `taylor` will give a polynomial approximation of a function. For example,

```
E = taylor(sin(x),11)
```
Its first argument is the expression to be expanded. Its second argument is the number of terms to include. The result of the expression above is

\( x - \frac{1}{6}x^3 + \frac{1}{120}x^5 - \frac{1}{5040}x^7 + \frac{1}{362880}x^9 \)

which has all of the first eleven terms, six of which equal zero. Another way to say what the 11 means here is that all the terms up to \( x^{(11-1)} \) are given.

The polynomial expansion of an expression is a good approximation to the expression when the argument (\( x \), in this case) is small. How small must \( x \) be? How good is the approximation? To get an idea of the answer to each of these questions it helps to plot both functions to compare one to the other.

\[ \text{>> ezplot(E)} \]
\[ \text{>> hold on} \]
\[ \text{>> ezplot(sin(x))} \]

The resulting plot is show here.

It can be seen that the approximation is good in the range from -3 to 3, but is bad if \(|x|\) is greater than 4.

### 12. 9 Calculus

The two central operations in calculus are taking the derivative and integrating.
Derivatives

The Matlab function that performs differentiation is \texttt{diff}. These operations show how it works:

\begin{verbatim}
>> diff(x^2)
ans =
 2*x
>> diff(x^n)
ans =
x^n*n/x
>> diff(sin(x)^2)
ans =
2*sin(x)*cos(x)
\end{verbatim}

\texttt{Diff} takes a second argument that is the name of the variable with respect to which the derivative is to be taken (\texttt{x} is the default) or, if it is a number (i.e., not symbolic), then it indicates the number of times the derivative is to be taken. For example,

\begin{verbatim}
>> diff(x^3,2)
ans =
 6*x
\end{verbatim}

Integrals

Integration is done by the function \texttt{int}. If \texttt{f} is a symbolic expression in \texttt{x}, then

\begin{equation}
\int f(x) \, dx
\end{equation}

Here are some examples:

\begin{verbatim}
>> int(x)
ans =
1/2*x^2

>> int(a^x)
ans =
1/log(a)*a^x

>> int(a^x^2)
ans =
1/2/log(a)*(a^x)^2

>> int(x*sin(x))
\end{verbatim}
ans = 
\sin(x) - x\cos(x)

>> int(a^x, a)
ans =
a^{(x+1)}/(x+1)

When there are two arguments, as in the last example, the second argument is the variable of integration. These are all indefinite integrals, meaning that there are no limits of integration. Note that Matlab does not add the constant associated with an indefinite integral. For example, the answer given by Matlab for \texttt{int(x)} is \(1/2\cdot x^2\), not \(1/2\cdot x^2 + \text{constant}\).

A definite integral can be taken by giving three arguments. The second and third arguments in that case are the first and second limits of integration.

\[
\int_{x=a}^{x=b} f(x) \, dx
\]

Here are some examples,

>> int(x, 1, 2)
ans =
3/2

>> int(x*sin(x), -2, 7)
ans =
\sin(7) - 7\cos(7) + \sin(2) - 2\cos(2)

>> int(a^x, a, 3, 4)
ans =
\frac{-(-4\cdot 4^x + 3\cdot 3^x)}{(x+1)}

The last example has four arguments. In that case, the second argument is the variable of integration, and the last two arguments are the limits of integration.

Sometimes Matlab will give back an answer that includes a function that you may not recognize. The following command, for example, returns an expression involving the function, \texttt{erf()}.

>> int(exp(-x^2), 0, 1)
ans =
1/2*erf(1)*\pi^{(1/2)}
The function \texttt{erf()} is the “error function”, which is well known to those who work in statistics. If you don’t recognize a function that Matlab gives you, use \texttt{help}, e.g., \texttt{help erf}.

\textit{Differential Equations}

Suppose a problem can be written as a differential equation. The most common form is that in which the first derivative of a function of time, \( y(t) \), is related to the function \( y(t) \) itself and some given function of time. For example,

\[
\frac{dy}{dt} + 2y = 12t^2
\]

Here the function of time is \( 12t^2 \). Matlab can find the solution for you via the function \texttt{dsolve}, as follows:

\[
\text{ans} = 6t^2-6t+3+\exp(-2t)\times C1
\]

This function takes a string, as opposed to a symbolic expression, as its argument. The letter “D” has a special meaning and cannot be used otherwise inside \texttt{dsolve}. It means “first derivative of”. The C1 is a constant of integration

Higher derivatives can be handled as well. The letters “D2” mean “second derivative of”, etc. Here’s an example a second-order equation:

\[
\text{ans} = C1\times \sinh(c\times t)+C2\times \cosh(c\times t)
\]

This time there are two constants of integration, C1 and C2. The functions \texttt{sinh} and \texttt{cosh} are the “hyperbolic” sine and cosine, respectively, and are defined as follows:

\[
\sinh(x) = \frac{1}{2}(e^x - e^{-x}) \quad \text{and} \quad \cosh(x) = \frac{1}{2}(e^x + e^{-x}).
\]

It’s easy to specify initial conditions with extra arguments (all strings):

\[
\text{ans} = \cosh(c\times t)
\]
Here the initial value of $y$ is 1 and the initial value of the derivative of $y$ is 0. Matlab cannot solve some differential equations. Such equations may require numerical approximation methods. Courses in numerical mathematics treat such methods.

12.10 Summations

A summation of the form, $E(1) + E(2) + \ldots + E(n)$ can be evaluated with Matlab by using the function `symsum`. If $E$ is a symbolic expression in $k$, then

$$\text{symsum}(E, a, b) \rightarrow \sum_{k=a}^{b} E_k$$

For example, if $E_k = \sin(k)$, then $E_1 + E_2 + E_3 + E_4$ looks like this in Matlab,

```matlab
>> syms k
>> symsum(sin(k),1,4)
ans =
sin(1)+sin(2)+sin(3)+sin(4)
```

There is nothing that Matlab can do with this one, but $\sum_{k=1}^{n} k^2$ has a simple form, so Matlab gives the simpler form as an answer:

```matlab
>> symsum(k^2,1,n)
ans =
1/3*(n+1)^3-1/2*(n+1)^2+1/6*n+1/6
```

12.11 Learning More

There is much more to the symbolic side of Matlab. To learn more, you should use the graphical help facility. To do that, you should To do that, click Help in Matlab’s taskbar and click MATLAB Help. The Help Navigator will appear on the left. Click the “plus” sign to the left of Symbolic Math Toolbox to expand a topic to reveal its subtopics. There you will see these topics (and a couple others):

- Getting Started
- Using the Symbolic math Toolbox
- Function Reference

Much of the material in the first two items is covered in this chapter. Under Function Reference, is a categorical list of all symbolic functions(Calculus, Simplification, Variable Precision Arithmetic, etc.) plus a link to an alphabetical list.
Concepts from this chapter

- Computer science and mathematics:
  - variable precision arithmetic
  - symbolic mathematics
    - solving algebraic equations
    - calculus
      - derivatives
      - integrals
        - indefinite
        - definite
      - differential equations
      - summations
- Matlab:
  - `vpa`
  - symbolic expressions
    - `syms, expand, factor, simple, solve, subs`
    - calculus
      - `diff, int, dsolve, taylor, symsum`
  - plotting symbolic expressions
    - `ezplot, ezsurf`
1. [Answer in English] Suppose you complete all the problems for this homework except the last and are ready to turn them in. Then you realize that you cannot do the last one!
   a. Is it a violation of the Honor Code for you to get help at that time from another student on that last problem?
   b. If you do get help, then, according to our Honor-Code rules in this class, is it required that, before you turn anything in, you throw away all your work on this assignment, start over on it, and redo everything without looking at your previous work?

2. [Answer in English] Consider the two languages C++ and Matlab. For each of the following problems, which language would be preferable
   a. Writing a new interactive spreadsheet to compete with Microsoft’s Excel.
   b. Calculating the current in electrical circuits for which you are given the resistances and the voltages.
   c. Writing a new interactive tool to be used by people who are unfamiliar with any programming language but want to calculate the current in electrical circuits when given the resistances and voltages.
   d. Designing a heart pump.
   e. Calculating the specific heat of a transformer oil when given the changes in temperature and masses involved.
   f. Writing a database management system.

3. [Answer in English]
   a. Suppose you write a new version of $\text{sqrt}$, which you put in the directory C:\Documents and Settings. Suppose further that you want your new version to be called automatically instead of Matlab’s built-in version whenever you issue the command $\text{sqrt}$ in the Command Window. Should you put C:\Documents and Settings at the beginning of the Matlab path or at the end?
   b. What is a program?
   c. Consider this sentence, “Add one to all the positive integers”. Is it an algorithm? If not, why not?
   d. When statements in a program are interpreted, are they always translated or always executed or both?
   e. When statements in a program are compiled, are they always translated or always executed or both?
   f. Which statements are more likely to be interpreted, the statements in source code or the statements in executable code?
4. [Answer in English] Each part below gives a sequence of characters that represents a single potential variable or function name. Identify each one as being legal or illegal. If it is illegal, say why it is illegal.

a. Not_a_valid_Matlab_name
b. LoW_MONthly_mOrtg_age_RATeS
c. L7
d. 7L
e. Fortran99
f. Guaranteed*Platinum
g. _xyz
h. Low.Rates
i. sqrt
j. $BASIC

5. [Answer in English]
   a. Is the option of using commas to separate the columns in a matrix an illustration of variation in Matlab’s syntax or a variation in Matlab’s semantics?
   b. What is the difference in syntax between the two types of multiplication in Matlab?
   c. What is the difference in semantics between the two types of multiplication in Matlab?
   d. Is exponentiation a binary or unary operation?

6. [Answer in English] Problems on the legality of arithmetic operations:
   a. If \( \text{size}(X) \) returns 2 5, which of the following operations is(are) legal:

   \[ X + X, \ X \times X, \ X \div X, \ X + X', \ X \times X', \ X \div X', \ X' \div X, \ X' \div X' \] 

   b. Suppose \( \text{size}(X) \) returns two integers, \( m \) and \( n \). What relationship, if any, between \( m \) and \( n \) will make all of the operations above legal?

7. [Answer in Matlab] Problems involving the colon operator:
   a. Assign \( X \) to be a 6x4 (6 rows and 4 columns) matrix such that row \( m \) consists of the numbers from \( m \) to \( m+3 \). Do it by using the colon operator once for each of the 6 rows.
   b. Assign \( X \) to be the same matrix as before, but this time use the colon operator once for each of 4 rows to produce the 4x6 transpose of \( X \) and use the transpose operator to make it a 6x4 matrix.
   c. Using the subarray operation with the colon operator, give a command that sets \( Y \) to be the vector that holds the odd numbers on the 4\textsuperscript{th} row of \( X \).
d. Using the subarray operation with the colon operator, give a command that sets \texttt{Lower Right} to be a matrix that holds the subarray that makes up the lower right quadrant of \texttt{X}.

8. [Answer in Matlab] Problems on the colon operator:
   a. Assign \texttt{X} to be a 5x2 (5 rows and 2 columns) matrix such that row \( m \) consists of the numbers from \( m-2 \) to \( m-1 \). Do it by using the colon operator once for each of the 5 rows. Your answer will look something like this:

   \[
   \texttt{X} = [-1:0; 0:1; \ldots]
   \]

   b. Assign \texttt{X} to be the same matrix as in the previous part, but this time use the colon operator once for each of only 2 rows to produce the 2x5 transpose of \texttt{X} and use the transpose operator to change it into the same 5x2 matrix that you produced for part a. (This problem is meant to show you that sometimes it is easier in Matlab to produce the transpose of a matrix, then to produce the matrix itself.) Your answer will look something like this:

   \[
   \texttt{X} = [\ldots]'
   \]

   c. Using the subarray operation with the colon operator, give a command that sets \texttt{Y} to be the matrix that holds the odd rows of \texttt{X} (i.e., rows 1, 3, and 5).

   d. Using the subarray operation with the colon operator and the transposition operator, give a command that sets \texttt{Z} to be a row vector that contains the elements of the second column of \texttt{X}.

9. [Answer in Matlab]
   a. Create a 6x3 matrix (6 rows and 3 columns) named \texttt{x} such that row \( m \) consists of the numbers from \( m \) to \( m+2 \). Do this for each row by using the colon operator to incrementally assign the values.

   b. Create a matrix, \texttt{w}, whose value is equal to the transpose of \texttt{x}. Again, do this for each row by using the colon operator to incrementally assign the values.

   c. Using the subarray operation with the colon operator, give a command that sets \texttt{y} to be the matrix that holds the odd columns of \texttt{x}. (i.e. columns 1 and 3)

10. [Answer in Matlab in the order stated] Exercise with arithmetic operators:
a. Assign $X$ to be a 3x4 (3 rows and 4 columns) matrix each of whose elements is equal to the absolute value of the difference of its indices. (Just type in the values, nothing fancy required here.)
b. Make a new matrix $Y$ that can be added to $X$ such that each of its elements is equal to the sum of its indices (again, you may, if you wish, just type it in).
c. Assign $P$ the value $X+Y$.
d. Assign $M$ the value $X-Y$.
e. Assign $E$ to be equal to $P \cdot \cdot P$.
f. Assign $F$ to be equal to $P' \cdot \cdot P$.
g. Assign $G$ to be equal to $P \cdot \cdot P'$.
h. Assign $H$ to be equal to $M \cdot \cdot M$.
i. Assign $I$ to be equal to $M' \cdot \cdot M$.
j. Assign $J$ to be equal to $M \cdot \cdot M'$.

11. [Answer in English] Let $P$ and $M$ be non-square matrices with the same shape and let $G$ be equal to $P$ or $P'$ or $M$ or $M'$. Let $H$ be equal to $P$ or $P'$ or $M$ or $M'$. Let $op$ stand for $\cdot \cdot$, $\cdot\cdot$ or $+$. 
a. How many legal combinations of $G \cdot \cdot H$ are there? 
b. How many legal combinations of $G \cdot \cdot H$ are there? 
c. How many legal combinations of $G + H$ are there?

12. [Answer in English] Will $M' \cdot \cdot M$ always, sometimes, or never be equal to $(M' \cdot \cdot M)'$ for an arbitrary matrix $M$? You can guess the answer by experimenting with arbitrary examples for $M$.

13. [Answer in Matlab] Using the colon operator, the transpose operator, and arithmetic operations give one-line assignment statements that produce the following two matrices (You will get the most credit by using the operators efficiently. You will get no credit for merely typing in the numbers directly):
a. $\textbf{series1}$ has one column consisting of the 9 integer multiples of $\pi$ from 5 $\pi$ to 13 $\pi$ in descending order (i.e., 13 $\pi$ down to 5 $\pi$). You may use the predefined constant, pi, only once.
b. $\textbf{series2}$ has two columns. The first column consists of the odd integers from $-17$ to 5, and the second column must be such that $\textbf{series2}(m,2) = 2 \cdot \cdot \textbf{series2}(m,1)$. You may use only the numbers, 2, 5, and $-17$ in your answer.

14. [Answer in Matlab] Using the colon operator, the transpose operator, and arithmetic operations, give one-line assignment statements that produce the following two matrices. (You will get the most credit by using the operators efficiently. You will get no credit for merely typing in the numbers directly.)
a. $\textbf{series1}$ has one column consisting of integer and half-integer multiples of $\pi$ from 3.5 to 10 (i.e., 3.5$\pi$; 4$\pi$; 4.5$\pi$; etc.). You may use the predefined constant, pi, only once.
b. $\textbf{series2}$ has two columns. The first column consists of the even integers from $-14$ to 7, and the second column must be such that $\textbf{series2}(m,2) =$
Homework Problems

2*series2(m, 1). You may use only the numbers 2, 7 and –14 in your answer.

15. [Answer in Matlab]
   a. Using both the colon operator and an exponentiation operator, give a one-line
      assignment statement that produces a matrix called squares
      that contains one row of the numbers 100 through 110
      a second row containing the squares of these numbers in order, and
      a third row containing the cubes of these numbers in order.
      (Write your answer concisely. You will not get credit for merely typing these
      numbers in.)
   b. Produce two vectors: x contains same thing as the first row of the matrix squares
      in the exercise above and y contains the third row. Suppress printing.
   c. Plot x versus y so that x is on the horizontal axis.

16. [Answer in Matlab]
   a. Using both the colon operator and an exponentiation operator, give a one-line
      assignment statement that produces a matrix called quads
      that contains one row of the numbers 1 through 10, a second row containing the squares of these
      numbers in order, a third row containing the cubes of these numbers in order and
      a fourth row containing each number to the fourth power in order.
   b. Create two vectors: x contains the first row of the matrix quads (from above) and
      y contains the fourth row. Suppress printing.
   c. Plot x and y (with x on the horizontal axis). Provide an appropriate title and labels
      for your plot.

17. [Answer in comments] How many different values can a variable have if its value is
    represented by 8 bytes?

18. [Answer in comments]
   a. What is the main difference between mathematical functions and the function of a
      programming language?
   b. X is 10 by 10 matrix. In row major order, which element follows X(3,4)?
   c. Suppose we use prefix notation for addition and multiplication. What would be the
      value of +(8 * (3 5))?
   d. In the expression, 4^2^2, which exponentiation operation is carried out first, the
      one on the left or the one on the right?
   e. In Fortran, what is the value of 7**3**2?

19. [Answer in Matlab] Give commands to do the following:
   a. Create a row vector that contains the sines of the angles 0.1, 0.2, ... 10.0 radians.
   b. Create a row vector that contains the cosines of the same angles.
   c. Plot one vector versus the other with the sines along the horizontal axis.

20. Give a command of only 6 keystrokes or fewer that produces this response (the
    command [1 4 7 10 13 16 19] has 19 keystrokes):

        ans =
21. Suppose $A$ is an $M$ by $N$ matrix (M rows and N columns) and $B$ is a $K$ by $L$ matrix. Give the restrictions on $M$, $N$, $K$, and $L$ so that the following operations are legal: for example: $A - B$ (Answer: $M = K$ and $N = L$)

a. $A \cdot B$

b. $A* B$

c. $B* A$

d. $A* A$

e. $A / B$

22. [Answer in Matlab] For part below, give the simplest command that accomplishes each task (assuming that the previous tasks have been completed in each case):

a. Set $E = \text{a 5 by 6 matrix containing all zeroes except that each element on the diagonal is equal to 7.}$

b. Set $f = \text{a row vector of 40 elements containing the even numbers in order starting with 2 and ending with 80.}$

c. Set $F = \text{a matrix each of whose rows is equal to } f \text{ and such that } E* F \text{ is legal.}$

d. Set $G = E* F$

e. Replace $G$ with a matrix that is equal to $G$, except that all elements $G(m,n)$ for which $2 \leq m \leq 4$ and $n > 15$ are equal to -3.
Homework Set 2

23. [Answer in Matlab] In these two problems you may use the operators +, -, *, /, and ^, but you may not use parentheses:
   a. Using only the numbers 1, 2, 3, and 4 once each, give an expression that has the smallest absolute value possible.
   b. Using only the numbers 1, 2, 3, and 4 once each, give an expression that has the largest value possible.

24. [Answer in Matlab] An important quality of a random number generator is that there is minimal correlation between one random number and the next one. In this problem you will write code to test the correlation of the numbers produced by \texttt{randn}. The “n” on the end of this word stands for “normal distribution”, and one quality of the normal distribution is that numbers are equally likely to be positive or negative.
   a. Give one command to generate 100,000 random numbers using \texttt{randn} and store them in a vector \texttt{r} (suppress printing).
   b. Give a second command that computes the sum, $\sum_{i=1}^{N} r_i^2$.
   c. Give a third command that computes the absolute value of the sum, $\sum_{i=1}^{N-1} r_i r_{i+1}$.
   d. Give a fourth command that calculates the ratio of the second sum divided by the first. This ratio should be small, if there is a small correlation between the signs of successive numbers.

25. [Answer in English] Consider the statement, \texttt{fave = input('Please ...')}, given at the beginning of Section 3.5 of Ch. 3. Describe what happens in each of the following situations. Give any error messages that Matlab produces in each situation verbatim (by cutting and pasting from the command window) and tell the value of given to \texttt{fave}.
   a. The user types in a 2x2 matrix instead of a vector.
   b. The user types in \texttt{hello}.
   c. The user types in the name of a variable that already has been assigned the value [5 25 45].
   d. The user types in the name of a variable that does not currently have an assigned value.
   e. The user types in a string (i.e., anything inside single quotes).

26. [Answer in English] Consider this command:

\begin{verbatim}
fprintf('No.: %g, price= %5.2f.\nTotal is %7.3f\n', 80, 3.1, 4*3.1);
\end{verbatim}

   a. What happens if \texttt{\indent} is replaced by \texttt{\tab}?
b. While it is rarely used, `fprintf` does return a value (*i.e.*, in addition to printing something into the command window). The value is the number of characters printed to the screen. Repeat the first `fprintf` command above in your command window but this time capture the value that is returned as follows:

```matlab
» x = fprintf('No.: %g, price= %5.2f. \nTotal is %7.3f\n', 80, 3.1, 4*3.1);
```

What value does `x` get?

c. Does this value agree with the number of characters visible on the screen? If not, why not?

27. [Answer in Matlab] The relation between the velocity and acceleration of an object is given as:

\[ v = u + at \]

where
- \( u \) = initial velocity in miles/hour
- \( v \) = final velocity in miles/hour
- \( a \) = acceleration in miles/hour\(^2\)
- \( t \) = time of acceleration in hours

Write a sequence of commands to accept the initial and final velocities (both integers) and the time of acceleration (in minutes) from the user, and calculate acceleration in miles per hour\(^2\). (Your answer will probably be a floating point value). Print out your answer using `fprintf` in the following format (numbers should appear where there are blanks below) with two digits to the right of the decimal place for each:

The object has an initial velocity of _____ mph, and undergoes an acceleration of _____ mph\(^2\) for ______ minutes, to achieve a final velocity of ___ mph.

28. [Answer in Matlab] A course involves 7 graded assignments. The grades in these assignments contribute the following respective percentages of the course grade: 20%, 10%, 30%, 5%, 5%, 15%, and 15%. There are 10 students in the course.

a. Give one command that sets up a column vector containing these percentages (use 0.20 for 20%, etc.)

b. Give a second command that sets up a matrix containing (fictitious) grades for each student for each assignment, letting each grade fall within the range from zero to 100.

c. Give a third command that calculates the weighted average for each student and puts them in a column vector.

29. [Answer in English] Add parentheses to the following expressions to show the order in which the operations are carried out. (Don’t be fooled by the spacing!) The first one is answered for you to show how to answer the rest of them.
Homework Problems

a. \(1 + 2^3 \cdot 4\)
   
   \textbf{ANSWER:} \(1 + ((2^3) \cdot 4)\)

b. \(1 + 2 - 3/4 \cdot 5\)

c. \(2^3^\cdot 3^\cdot 5\)

d. \(1 - 2^3 \cdot 3 - 2\)

e. \([1 \ 2; \ 3 \ 4] \cdot [1 \ 2; \ 3 \ 4]^2\)

f. \(1 + 2/3/4 \cdot 5\)

g. \(1/2^5 + 6^2 + 3\)

h. \(2^4 / 3 \cdot 5\)

i. \([1 \ 2 + 5 - 2 + 3\]

j. \([1 \ 2; \ 3 \ 4] \cdot [1; \ 2]^\cdot 3\]

k. \(1^\cdot 3 - 3^\cdot 4^\cdot 5\)

l. \(2^3^\cdot 3^\cdot 5\)

m. \(1 - 2^3 + 3 - 2\)

n. \([1 \ 2; \ 3 \ 4] \cdot [1 \ 2; \ 3 \ 4]^2 \cdot [1 \ 2; \ 3 \ 4]\)

30. [Answer in Matlab.] Let \(p\) = the precedence vector arithmetic operators, where \(p(1)\), \(p(2)\), \(p(3)\), and \(p(4)\) give the precedences of parentheses, exponentiation, multiplication and division, and addition and subtraction, respectively. Give the Matlab command that sets \(p\) equal to the appropriate row vector for each of the following, but in each case leave the precedence for parentheses the same as the standard Matlab value:

a. The standard Matlab precedences.

b. A set of precedences that would cause the expression \(27 \cdot 80 - 6 \cdot 10\) to be equal to 19980.

c. A set of precedences that would cause the expression \(10^\cdot 2 + 3 \cdot 2 = 4\) to be equal to \(1e-10\).

d. A set of precedences that would cause the expression \(2 - 4^\cdot 3 \cdot 2\) to be equal to \(-4094\).

31. [Answer in Matlab] Write code that produces the last plot (the “fancy plot”) of Section 3.7 of Ch. 3. Hint 1: Plot five vectors \(y_1, y_2, y_3, y_4, y_5\) versus the vector \(x\). The vector \(x\) is a vector whose first element is zero, whose elements are spaced by 0.1, and whose largest value is 200. Hint 2: The following mathematical functions will be useful:

- \(\cos(x)\)
- \(\exp(-(x-100)^2/200)\)
- \(-\exp(-(x-100)^2/200)\)
- \((1/2)\exp(-(x-100)^2/200)\)
- \(-(1/2)\exp(-(x-100)^2/200)\)

Place the black circle at the center. Suppress all printing to the command window. Don't forget to include the title of the plot and the labels for the \(x\) and \(y\) axes. NOTE: You must not use any of Matlab's looping or selection facilities (e.g., for loop, if statement).
32. Suppress all printing. Give commands to produce a plot of the following two functions over the range from 0 to $2\pi$:

\[
\begin{align*}
  f(x) &= 2(\sin x)e^{-x} \\
  g(x) &= \frac{4x^3 + 10x + 5}{20x^4 + 30x + 3}
\end{align*}
\]

a. [Answer in Matlab] Put $x$ on the horizontal axis (as usual). Make the horizontal distance between plotted points 0.001 or less. Plot the first function as a red line and the second as a blue line. Add a grid. Include a title that says “Plots of two functions of $x$”. Label the horizontal axis “$x$” (no label for the vertical axis). Give a legend that identifies the two plots as “$f(x)$” and “$g(x)$”, respectively (use help legend). You do not need to worry about the tick marks or the numbers that Matlab uses to label them. Just use what appears automatically.

b. [Answer in English] Using the plot that you have just produced, determine to within two decimal positions to the right of the decimal point the smallest value of $x$ within the range 0 to $2\pi$ for which $f(x) = g(x)$.

33. [Answer in English] Find the largest value of $x$ within the same range for which $f(x) = g(x)$.

Write Matlab code to do the following things. Use the colon operator to reduce the size of your commands as much as possible:

a. Produce a column vector $x$ that contains the numbers from -10 through 10, increments of 0.01.

b. Suppose there is a circle whose center is at (0, 0) and whose radius $r = 10$. For each index $m$, let the elements $x(m)$ and $y_p(m)$ of vectors $x$ and $y$ be equal to the $x$ and $y$ coordinates, respectively, of a point $P(x, y)$ on the upper half of the circle, which is the half on which $y \geq 0$. Let the elements $x(m)$ and $y_n(m)$ of vectors $x$ and $y$ be equal to the $x$ and $y$ coordinates of a point $P(x, y)$ on the lower half of the circle, for which $y \leq 0$. Use the equations, $y = (r^2 - x^2)^{1/2}$ and $y = -(r^2 - x^2)^{1/2}$, to produce $y_p(m)$ and $y_n(m)$.

c. Plot $y_p$ versus $x$ and $y_n$ versus $x$ in the same figure to produce a red circle. After you have used plot, give the command axis equal to see a truly circular shape.

34. [Answer in Matlab] The following set of equations is given:

\[
\begin{align*}
  ax + by + cz &= f \\
  dx + by + ez &= g \\
  bx + cy + cz &= h
\end{align*}
\]

The following values are given: $a = 4, b = 6, c = 7, d = -3, e = 9$ and $x = -4, y = 3, z = 6$. The values of $f, g,$ and $h$ can be calculated easily with Matlab.
Homework Problems

using three commands:

a. Give a command that sets up the appropriate matrix.
b. Give a second command that sets up the appropriate vector.
c. Give a third command that calculates the unknowns.

35. [Answer in Matlab] Water is pumped into a system of pipes at points A and B in the diagram below:

At each of the junctions C, D, E, F, G and H, the pipes are split and water flows according to the proportions indicated in the diagram. If water flows into A and B at \( p \) and \( q \) gallons per minute respectively, determine the following:

a. Consider the junctions C, D, E and their associated water flows, W1, W2 and W3.

Assign a matrix \( X \), such that

\[
\begin{bmatrix}
W1 \\
W2 \\
W3
\end{bmatrix}
= 
X
\begin{bmatrix}
p \\
q
\end{bmatrix}
\]

b. Assign a matrix \( Y \), such that

\[
\begin{bmatrix}
I \\
J \\
K
\end{bmatrix}
= Y
\begin{bmatrix}
W1 \\
W2 \\
W3
\end{bmatrix}
\]

c. Assign a matrix \( Z \), such that the outputs at I, J and K may be represented as

\[
\begin{bmatrix}
I \\
J \\
K
\end{bmatrix}
= Z
\begin{bmatrix}
p \\
q
\end{bmatrix}
\]

36. [Answer in English] Suppose that a particular region with a fixed population is divided into three areas: the city, suburbs and country. We wish to track the demographic flow of people from year to year between the city, suburbs and country. The probability that a person living in the city moves to the suburbs (in one year) is 0.10 and moves to the country is 0.50. The probability that a person living in the suburbs moves to the city is 0.20 and moves to the country is 0.10. The probability
that a person living in the country moves to the city is 0.20 and moves to the suburbs is 0.20. Suppose that currently 650,000 people live in the city, 335,000 people live in the suburbs and 185,000 people live in the country.

a. How many people would you expect to be living in the city, suburbs and country (respectively) after one year? (Hint: start by devising a system of linear equations representing the new populations as a function of the current populations and their likelihood of moving.)

b. How many people would you expect to be living in the city, suburbs and country (respectively) after two years?

c. How many people would have been living in the city, suburbs and country (respectively) one year ago?

37. [Answer in English] A paper distribution company is putting together shipments containing various types of computer printer paper. On hand, they have 14 gross of high brightness inkjet paper, 25 gross of medium brightness inkjet paper and 15 gross of all-purpose paper. The company ships three types of pallets containing different quantities of paper according to the table below:

<table>
<thead>
<tr>
<th></th>
<th>Pallet Type 1</th>
<th>Pallet Type 2</th>
<th>Pallet Type 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>High brightness inkjet paper</td>
<td>4 gross</td>
<td>1 gross</td>
<td>3 gross</td>
</tr>
<tr>
<td>Medium brightness inkjet paper</td>
<td>6 gross</td>
<td>4 gross</td>
<td>2 gross</td>
</tr>
<tr>
<td>All purpose paper</td>
<td>5 gross</td>
<td>1 gross</td>
<td>2 gross</td>
</tr>
</tbody>
</table>

Determine the number of each type of pallet the company can assemble and ship.

38. [Answer in Matlab] Money flows into a certain medical foundation from two sources: donations and interest earned on endowment. These funds are then used as follows:

- Donations: Thirty percent goes into fundraising and the rest goes into research and clinics.
- Interest: Ten percent goes into fundraising and the rest goes into research and clinics.

All money that flows into research and clinics is divided this way: Forty percent goes into research and the rest goes into clinics. Finally, the money that goes into fundraising, research, and clinics flow into material costs and personnel costs according to the table below:

<table>
<thead>
<tr>
<th></th>
<th>Research</th>
<th>Clinics</th>
<th>Fundraising</th>
</tr>
</thead>
<tbody>
<tr>
<td>Material Costs</td>
<td>80%</td>
<td>50%</td>
<td>70%</td>
</tr>
<tr>
<td>Personnel Costs</td>
<td>20%</td>
<td>50%</td>
<td>30%</td>
</tr>
</tbody>
</table>

a. Let $s$ (for “source”) be a column vector such that $s(1)$ equals the money from donations and $s(2)$ equals the money from interest. Let $g$ (for “goals”) be a
column vector such that \( \mathbf{g}(1) \) equals the money spent on research, \( \mathbf{g}(2) \) equals the money spent on clinics, and \( \mathbf{g}(3) \) equals the money spent on fundraising. Give the proper values to the elements of a matrix \( \mathbf{A} \) such that \( \mathbf{A}s = \mathbf{g} \).

b. Let \( \mathbf{c} \) (for “costs”) be a column vector such that \( \mathbf{c}(1) \) equals material costs and \( \mathbf{c}(2) \) equals personnel costs. Give the proper values to the elements of a matrix \( \mathbf{B} \) such that \( \mathbf{B}^* \mathbf{g} = \mathbf{c} \).

c. Give an expression in terms of \( \mathbf{A} \) and \( \mathbf{B} \) for a matrix \( \mathbf{C} \) such that \( \mathbf{C}s = \mathbf{c} \).

d. Suppose the foundation has an annual budget calling for $2.5 million to be spent on research, $4 million to be spend on clinics, and $2 million to be spent on fundraising. Using one of the matrices, \( \mathbf{A} \), \( \mathbf{B} \), or \( \mathbf{C} \), calculate the vector \( \mathbf{s} \) that gives the required amounts of donations and interest.

e. Using the same budget, as that given in the previous part, calculate the vector that gives the resulting material costs and personnel costs.

39. [Answer in Matlab] Use the same values for \( a \) through \( e \) that were given in the previous problem, but this time treat \( x \), \( y \), and \( z \) as unknowns, and give the following values to \( f \), \( g \), and \( h \): 14, 15, and 16.

a. Give a command that sets up the appropriate matrix.

b. Give a second command that sets up the appropriate vector.

c. Give a third command that calculates the unknowns.

40. [Answer in Matlab] An electrical circuit is set up as follows:
where the gray boxes are resistors, the two thick lines labeled “5.00 volts” represent a battery, and the arrows represent the flows of current. The following equations give the relationships among the resistors, the currents, and the battery:

\[ i_2 + i_4 = i_1 \]
\[ i_2 + i_3 = i_5 \]
\[ i_1 + i_3 = i_6 \]
\[ i_1 R_1 = i_3 R_3 \]
\[ i_4 R_4 = i_4 R_5 \]
\[ i_1 R_1 + i_4 R_4 = E \]
\[ i_2 R_2 + i_5 R_5 = E \]

The Rs give the resistances of the respective resistors, which are given as labels in the diagram. Thus, \( R_1 = 100, R_2 = 200, R_3 = 40.0, R_5 = 50.0 \) (ohms). Note that there is no \( R_2 \). The strength of the battery is represented by \( E \), and \( E = 5.00 \) volts.

This set of simultaneous equations in the unknown currents \( i_1, i_2, \ldots, i_6 \) can be written in the form \( Ax = b \), where the currents are the elements of the vector \( x \). In order to determine the elements of \( A \) and \( b \), you must first rewrite the equations above so that the unknowns are all on the left. For instance, the first equation can be rewritten as \( i_2 + i_4 - i_1 = 0 \). It also helps then to rewrite that equation into a form that has all the variables in the ascending order by subscripts with zeros multiplying the ones that are missing: \(-1*i_1 + 1*i_2 + 0*i_3 + 1*i_4 + 0*i_5 + 0*i_6 = 0\). Note that we have also multiplied the other variables by 1, to show that even they have a coefficient. After you have written them that way (not to turn in), do the following:

a. Write the command that sets up the matrix \( A \).
b. Write the command that sets up the matrix \( b \).
c. Write the command that solves \( Ax = b \) for the currents in \( x \).

41. [Answer in English] There are 7 equations in 6 unknowns in problem 40, but because of redundancy among the equations, they are not over determined. Thus, the solution that you found above is exact (within round-off error). Since they are not over determined, it is possible to find the same 6 currents by solving only 6 equations. Suppose you were allowed to use only 6 of the equations in the previous problem to find the 6 currents. Which equation would you omit: 1, 2, 3, 4, 5, 6, or 7?

42. Consider the matrices \( H \) and \( K \) and the column vector \( c \), given below
a. [Answer in Matlab] Give the commands to solve each of the following equations for $x$:

$$Hx = c,$$
$$Hx = d,$$
$$Kx = c,$$
$$Kx = d$$

b. [Answer in English] Calculate condition numbers and use them to explain the similarities and differences among the solutions that you got to part a.

43. A television tower is 1200 feet tall and is held firm against the wind by 4 cables that are each fastened to the top of the tower at one end and to the ground at the other, as shown in the figure:
Each cable is under tension, as follows: Cable 1 is at 1000 pounds, Cable 2 is at 1500 pounds, Cable 3 is at 870 pounds and Cable 4 is at 1500 pounds.

There are three components of force exerted by each cable on the tower: two horizontal components in the East/West direction and North/South direction and one vertical component in the downward direction. The orientation of each cable determines the ratio of each of its component forces to its tension. For example, the East/West component of force exerted on the tower by Cable 1 divided by the tension in Cable 1 is -0.61546. The minus sign means that the component is toward the West. A positive number means East. The North/South ratio is also -0.61546. Here negative means Southward and positive means Northward. The downward ratio is 0.49237. These ratios are determined completely by the direction of the cable. If the tension on the cable were doubled, each component would double also, and the ratios would remain the same. This is true for all the cables. It’s a property of the physics of the structure.

Here are the ratios for all components for all cables in this order—East/West, North/South, Down:

- Cable 1:  -0.61546     -0.61546      0.49237
- Cable 2:  0.90815      0.27245      0.31785
- Cable 3:  0.88697     -0.29566      0.35479
- Cable 4:  -0.91863      0.22966      0.32152

a. [Answer in Matlab] Give the commands to calculate the three components of the net force exerted by these cables on the tower. (You calculate the net force by adding the component forces for each cable including the signs in the calculations.)

b. [Answer in Matlab] Give commands to calculate the three components of net force exerted on the tower if Cable 1 breaks and the other cables’ tensions each remain the same. (They would remain the same for only a brief instant after Cable 1 broke.)

c. [Answer in Matlab] Suppose we could adjust the tensions in the four cables to any values we wished. We might want to adjust them so that there is no net horizontal force on the tower. What tensions should we have on the four cables to produce a zero net force in the East/West direction and a zero net force in the North/South
direction and a force of 1500 pounds in the downward direction?

d. [Answer in English] According to your answer for c., which cable is redundant?
1. Suppose a Matlab program is to be written that prints the products of each ordered pair of integers between 0 and \( n \), where \( n \) is a positive number given as input by the user. Thus, for \( n = 3 \), it will print the sixteen products, \( 0*0 = 0, 0*1 = 0, 0*2 = 0, 0*3 = 0, 1*0 = 0, 1*1 = 1, 1*2 = 2, 1*3 = 3, 2*0 = 0, \ldots, 3*3 = 9 \).

   a. [Answer in English] Write a black-box description of the problem. In other words complete steps 1 and 2 of the five-step design process presented above. In step 2 be clear that both \( x*y \) and \( y*x \) will be included (e.g., 6 will be printed twice: once for \( 2*3 \) and once for \( 3*2 \)), and include a formula that will give the number of products that will be printed in terms of the number that is given as input by the user.

   b. [Answer in Matlab] Write a solution in Matlab to the problem. DO NOT USE BRANCHING OR LOOPS. All products can be produced with ONE COMMAND.

2. Suppose we wish to create an astrology program in Matlab. (Note: You are not being asked to write the Matlab program, but only to complete the first three steps of the design process.) The user will type in their birthday, and the program responds with the user’s sign and horoscope. The month may be entered as a number from 1 to 12. Use a newspaper (or the internet) to find the horoscopes and dates of each sign. Then enhance your program so that, if the user is only one or two days away from an adjacent sign, the program announces that the user is on a “cusp” and also outputs the horoscope for the nearest adjacent sign.

   a. [Answer in English] Write a black box description of the problem. In other words, complete steps 1 and 2 of the 5-step design process.

   b. [Answer in English and/or Pseudocode] Design an algorithm to solve this problem.

3. Suppose a Matlab program is to be written that will accept three positive-valued and non-equal integers (\( \text{intU}, \text{intV}, \text{and intW} \)) as input from the user. The program will then compare the numbers and assign the largest number to \( \text{intBig} \), the smallest number to \( \text{intLittle} \) and the middle number to \( \text{intMiddle} \).

   a. [Answer in English] Write a black box description of the problem. In other words, complete steps 1 and 2 of the 5-step design process.

   b. [Answer in Matlab] Write a solution in Matlab to the problem.
4. Weight lifters sometimes need a table for converting from mass in kilograms (kg) to weight in pounds (lb). Suppose that the masses come in 5 kg units. You are to write a program that generates a conversion table beginning with 20 kg and going up by steps of 5, stopping at a limit specified by the user. Each row of the table will have two entries—the mass and its corresponding weight. The first row will be for 20 kg. The second row will be for 25 kg, etc. The user is prompted to specify the largest mass desired. The table goes no higher than that.

Thus, if the user specifies the value \( N \) (kg), then the table should stop at the largest multiple of 5 that is not greater than \( N \). For example, if \( N = 97 \), the largest mass in the table would be 95 kg. The weight of 1 kg is 2.205 lb. You do not need to print out the units in your table. It is to be a table of numbers.

a. [Answer in English] Write a black-box description of the problem. In other words, complete steps 1 and 2 of the five-step design process presented above.

b. [Answer in Matlab] Write a Matlab solution to the problem. DO NOT USE BRANCHING OR LOOPS.

5. A projectile is fired from the origin with an initial speed of \( v \) at an angle of \( \theta \). You wish to write a program that will ask the user to input \( v \) and \( \theta \) and then calculate and report the horizontal distance traveled by the projectile.

a. [Answer in English] Write a black box description of the problem. In other words, complete steps 1 and 2 of the 5-step design process.

b. [Answer in English and/or Pseudocode] Design an algorithm to solve this problem.

6. You wish to create a program to determine the minimum number of coins needed to give a specific amount of change requested by the user. The program should produce output like the following:

89 cents can be given as:
3 quarters 1 dime and 4 pennies

-or-

6 cents can be given as:
1 nickel and 1 penny

In this way, we strive for a grammatically correct output and avoid printing pointless information (like 0 quarters).

a. [Answer in English] Write a black box description of this problem. In other words, complete steps 1 and 2 of the 5-step design process.
b. [Answer in English and/or Pseudocode] Design an algorithm to solve this problem.

7. [Answer in Matlab] Write a nested if statement that does the following: It prints “Even” on one line if x is an even integer and, if x also lies in the range, 0 to 1000, inclusive, it prints “In range” on the next line. It prints “Odd” on one line if x is an odd integer with no check of its the range. If x is neither an even integer nor an odd integer, it prints “Non-integer” on one line otherwise. You must do this with ONLY FOUR fprintfs.

8. [Answer in Matlab] Write a set of nested if statements to perform the logic shown in the diagram below.

9. A bicycle salesperson is offered a choice of wage plans: (1) a straight salary of $300 per week; (2) $3.50 per hour for 40 hours plus a 10% commission on sales; (3) a straight 15% commission on sales with no other salary. Write a program that takes as input the salesperson’s expected weekly sales and outputs the wages paid under each plan as well as announcing the best-paying plan.

10. [Answer in Matlab] Suppose you are programming a DVD player. When you turn the player on, it provides the following menu of choices:

1. Play the disk
2. Settings
3. Select Language
4. Help
5. Exit
Once the user enters a choice, the program must call one of several functions to process the request. In a preliminary version of the program, instead of calling a function, a message is printed indicating the choice. Write the code that requests input of a numerical choice (from 1 to 5) from the user, and then use a switch construct to print out the user’s choice in words. For example, if the user chooses 1, your code should print the message “The user wants to play the disk.” (or a similar message.) You should include a response for invalid input.

11. [Answer in Matlab] Write a nested if statement that does the following:
   - It prints “Complex Number” on one line, if \( x \) is a complex number. [Hint: use help to look up the function \( \text{imag}() \).]
   - If \( x \) is not complex but is positive, it prints “Positive Number” on one line.
   - It prints “Zero”, if \( x \) equals zero.
   - It prints “Negative” if \( x \) is less than zero.

Use \texttt{input} to get the \( x \) value from user. Include a reasonable prompt.

12. [Answer in Matlab] Write a program that will ask the user how fast he or she drives, and then print out what response a police officer would make to the following speed ranges: >75, >55, >45, <45. Use nested if-else statements.
13. [Answer in Matlab] Write a set of nested if statements to perform the logic shown in the diagram below.

![Flowchart Diagram]

14. [Answer in Matlab] Write a switch statement that prints the name of the weekday corresponding to the number stored in x: If x equals 1, then “Monday” is printed, if x equals 2, then “Tuesday” is printed, etc.. If x is not in the range, 1 to 5, then the statement should print “Not a weekday”.

15. [Answer in Matlab] Suppose a Matlab program is to be written that prints the name of the month and the number of days in that month corresponding to a number input by the user. For the purposes of this assignment, you may disregard leap years.

   a. Write the Matlab code to read a value input from the keyboard and assign it to the variable x.

   b. Write a switch statement that prints the name of the month corresponding to the number stored in x: If x equals 1, then “January” is printed, if x equals 2, then
“February” is printed, etc. If $x$ is not in the range, 1 to 12, then the statement should print “Not a month”.

c. Write a second switch statement that will print the number of days in the month specified by the user. If $x$ is not in the range, 1 to 12, no output should be printed.

16. [Answer in Matlab] Suppose an automated library catalogue has the following menu of choices:

1) Search for a title
2) Search for an author
3) Search for a subject
4) Get help
5) Quit

Once the user enters a choice, the program must call one of the several functions to process the request. In a preliminary version of the program, instead of calling a function, a message is printed indicating the choice. Write code that requests the input of a numerical choice from 1 to 5, and then uses the `switch` construct to print out the user’s choice in words. For example, if the user chooses 1, your code should print the message “The user wants to search for a title.” Include a response for invalid input.

17. [Answer in Matlab] Suppose 200 hundred freshmen, who each are given a unique identification number in the range [1, 200], are organized into 4 groups according to the value of the identification number, i.e., [1: 50] are in group 1, [51:100] are in group 2, and so on. Suppose an identification number is stored in a variable named `number`. Use the `if` construct to set a variable named `group` to be equal to the appropriate group number according to `number`. If number is out of range, the `if` construct should print an error message instead of setting the value of `group`.

18. [Answer in Matlab] The equations for various conics and the values of $x$ to be used while plotting them are given below:

1) Circle: $\ y = \sqrt{16 - x^2}$ where $-4 < x < 4$
2) Hyperbola: $\ y = \sqrt{x^2 - 16}$ where $4 < x < 8$
3) Parabola $\ y = \sqrt{4 \ast (x - 6)}$ where $6 < x < 10$
4) Ellipse $\ y = \sqrt{1 - (x^2)/4}$ where $-2 < x < 2$

Write a program in Matlab to accept an integer representing the conic section to be plotted. Using subplot(2,2,1), where I is the value given by the user and a switch statement, plot the equation in the appropriate subplot. (e.g. A parabola would be plotted using subplot(2,2,3)).

Provide an appropriate title for your graphs. When initializing $x$, keep the step size small enough so that the graph appears continuous. Set the axis ranges using the axis
command so that all four quadrants can be seen in the graph.

19. [Answer in Matlab] Write a program that reads in the radius of a circle from the user, presents a table of numbered options, and then outputs one of the following depending on which option number the user enters: 1. circumference of the circle, 2. area of the circle, or 3. diameter of the circle. The output should be in sentence format. For example, “With a radius of 1.75, the area of the circle is 5.495.” Limit your output to three digits to the right of the decimal point.

20. [Answer in Matlab] You are to give the user an option of calculating a function of a number, x, that you give. The choices are as follows:

1  Sine of x
2  Cosine of x
3  Tangent of x
4  Exponential of x
5  Natural Logarithm of x
6  Base 10 Logarithm of x
7  Square Root of x
8  Quit

The user should be prompted to type in the number x and then be shown the menu above and asked to make a selection. The program should print the value of the selected function by using a switch construct. If Quit is the selection, then nothing should be printed. Include a reasonable response for a number out of range. You may assume that the user will type in a real scalar.

21. [Answer in Matlab] Write a program to compare the car emissions for various pollutants to given emission limits, taking the age of the car into consideration. Print “Within emission limit” if the emissions are within legal limits, else print “Your car is a danger to the environment”. The algorithm is as follows:

The pollutant is selected by choosing an integer (pol_number) from a menu of the form:

(1) Carbon Monoxide
(2) Hydrocarbons
(3) Nitrogen Oxides
(4) Non-methane Hydrocarbons

The user is prompted to enter the emission (emission) in grams per mile. The user is prompted to enter the odometer reading (odom) and the year in which the car was manufactured (year_man).

The entered values are compared to the limits given in the table below and the appropriate message is printed.
22. [Answer in English] Assume P and Q are simple conditions. The following logical equivalences are known as DeMorgan’s Laws:

\[
\text{Not (P or Q) is equivalent to Not P and Not Q} \\
\text{Not(P and Q) is equivalent to Not P or Not Q}
\]

For example, the logical equivalence:

\[
\text{Not ((P and Q) and (Not P and Q))}
\]

can be broken down to the following logical equivalent in two steps:

\[
\text{Step 1 = Not (P and Q) and Not (Not P and Q)} \\
\text{Step 2 = (Not P or Not Q) and (P or Not Q)}
\]

Use DeMorgan’s Laws to write a logical equivalent for each of the following:

a. Not P or Not Q  
b. Not ((Not P) or Q)  
c. Not (Not P and Q)  
d. Not (Not P and Not Q)

23. The word “Boolean” is derived from the name of George Boole, a nineteenth century English mathematician who is considered the father of symbolic logic. Boole was a self educated scholar with limited formal training. He began his teaching career at the age of 16 as an elementary-school teacher and eventually progressed to a professorship at Queen’s College in Cork.

a. [Answer in English] The expression \textbf{A or B} evaluates to true provided that the value of either or both of the variables is true. Design a Boolean expression that evaluates to true provided that the value of exactly one of the two variables is true.

b. [Answer in English] The following four Boolean expressions divide into two groups with two equivalent expressions in each group. What are the two groups? (Two expressions are equivalent if they evaluate to the same value of true or false for each possible way of setting Footloose and FancyFree to true or false)

\[
\text{not(Footloose) and not(FancyFree)}
\]
not(Footloose) or not(FancyFree)
not(Footloose and FancyFree)
not(Footloose or FancyFree)

24. [Answer in Matlab] Write one single expression that involves only operators (i.e., no function calls) and only the variables x and y. Your expression must evaluate to:

\[
\begin{align*}
&1, \text{ if } x \text{ or } y, \text{ or both } x \text{ and } y, \text{ equal zero} \\
x/y, \text{ if } x \text{ is greater than } y \text{ and } x \text{ and } y \text{ are not 2 or 4} \\
&1, \text{ if } x \text{ is equal to } y \\
y/x, \text{ if } x \text{ is less than } y \text{ and } x \text{ and } y \text{ are not 2 or 4} \\
x*y, \text{ if } x^y \text{ is equal to } y^x \text{ and } x \text{ is not equal to } y
\end{align*}
\]

Use `input` to get x and y. Use a reasonable prompt.

**DO NOT USE BRANCHING OR LOOPS. GIVE JUST ONE SINGLE EXPRESSION!** [Hint: Use relational operators and logical operators along with the arithmetic operators. For example: \((x>y)*x*y\) will be equal to the product of \(x\) and \(y\) if \(x\) is greater than \(y\), but it will equal zero if \(x\) is less than or equal to \(y\).]

25. [Answer in Matlab] A prime number (or prime integer, often called simply a “prime” for short) is a positive integer, \(p > 1\), that has no positive integer divisors other than 1 and \(p\) itself. One way to test whether a number is prime is to simply divide \(p\) by all the integers smaller than itself to determine if it has additional divisors. Use a while loop and an if statement to determine whether a number \(N\) (where \(N > 2\)) input by the user is a prime number. (Hint: the function \(\text{mod()}\) may prove useful.)

26. [Answer in Matlab] The Fibonacci series is represented as follows: 1, 1, 2, 3, 5, 8, 13, ...
It is defined as follows:

\[
x_1 = 1, \\
x_2 = 1, \\
x_i = x_{i-1} + x_{i-2} \text{ for all integer } i \text{ greater than 2.}
\]

a. Write a program using loops to print the first 20 numbers in the Fibonacci series.

b. Generate a semi-logarithmic plot of the Fibonacci series you generated in part a. You will want to make the y-axis logarithmic.

27. [Answer in Matlab] One place that the Fibonacci series (see problem 26) regularly occurs is in population growth rates. If a population has no deaths, then the series shows the increase in population after each generation. A generation is the time it takes a member to reach reproductive maturity. The formula applies most straightforwardly to asexual reproduction at a rate of one offspring per parent per generation. In any event, the green crud population grows at that rate and produces
one generation every five days. Hence, if a green crud population starts out as 10 pounds of crud, then in 5 days there is still 10 pounds of crud; in 10 days there is 20 pounds of crud, in 15 days there is 30 pounds, in 20 days there is 50 pounds, and so forth. Write a program that takes as input the initial size of a green crud population (in pounds) and a number of days and then outputs the number of pounds of green crud after that many days. Assume that the population size remains the same for four days and then increases on the fifth day.

28. [Answer in Matlab] The function $f(n)$ is defined as follows:

\[
\begin{align*}
    f(1) &= 7 \\
    f(2) &= 3 \\
    f(n) &= 2f(n-1) - 3f(n-2) \quad \text{for } n \geq 3
\end{align*}
\]

So, for example, $f(3) = 2f(2) - 3f(1) = 6 - 21 = -15$, etc.

Use an `input` statement with a reasonable prompt to get a value from the user for $n$. Check to make sure that $n$ is greater than 2. If is not, just print an error message. If it is, use a `while` loop to calculate the function for $n$ and print only $f(n)$. Thus, for example, if the user types in 3, the program would print –15.

29. [Answer in Matlab] Use a nested `for` loop to print a triangle of stars with the number of stars on each line equal to the corresponding line number and with the stars on the same line separated by two white spaces from each other. The total number of lines should be entered by the user. Include a reasonable prompt. An example run is shown below for 6 lines. The first star on the last row is at the beginning of the line (there is no space in front of it).

```
* 
** 
*** 
**** 
***** 
******
```

Ignore the fact that, if the requested number is too large, some of the longer lines in the pattern will wrap-around from one line to the next.

30. [Answer in Matlab] The formula for multiplying two matrices, $A$ and $B$, is given as:

\[
C_{ij} = \sum_{k=1}^{K} A_{ik} B_{kj}
\]

Where

- $A$ is an $M$ by $K$ matrix,
- $B$ is a $K$ by $N$ matrix and
$C$ is the resultant $M$ by $N$ matrix

Write a program using for loops and if-else statements in Matlab to accept two matrices, $A$ and $B$, from the user, check if their inner dimensions agree (verify they can be multiplied) and then print $C$, the result of their multiplication. (You may print it anyway you wish.) Use for loops to perform the math, DO NOT use Matlab’s built-in matrix multiplication abilities. If the inner dimensions of the two input matrices do not agree, report that information to the user.

31. [Answer in Matlab] Write a program using loops to find all integer solutions to the equation $4x + 3y - 9z = 5$ for values of $x$, $y$ and $z$ between 0 and 100.
Homework Set 4

Problems that do not explicitly say "Answer in English" are to be answered in Matlab.

1. Write a predicate called `isscalar` that takes one argument that determines whether or not that argument is a scalar (i.e., returns 1 if for scalar, 0 otherwise).

2. Write a predicate called `are_equal` that takes two arguments and determines whether or not they are equal. In order to be equal they must have the same sizes, and have equal elements.

3. Write a stub for a function called `Vandy` that takes 3 arguments and returns two output arguments.

4. Write a stub for a function called `dualfunction` that takes 2 arguments and returns two output arguments. The first equals the sum of the two input elements, the second equals the difference of the two elements.

5. Write a predicate called `is_even` that takes one argument and determines whether or not that argument is an even number.

6. Write a predicate called `string_equal` that takes two string arguments and determines whether or not they are equal. In order to be equal they must have the same number of characters and have the same elements.

7. Write a function called `order3` that takes zero to three arguments and returns three values. Example calls:

   Case 1: `[x,y,z] = order3;`
   Case 2: `[x,y,z] = order3(a);`
   Case 3: `[x,y,z] = order3(a,b);`
   Case 4: `[x,y,z] = order3(a,b,c);`

   In the first case `x`, `y`, and `z` are set to zero. In the second case, `x`, `y`, and `z` are set equal the value of `a`. In the third case, if `a` is equal to `b`, then `x`, `y`, and `z` are set equal to `a`, but, if `a` and `b` are unequal, then `x` is set equal to the smaller one, and `y` and `z` are both set equal to the larger one. In the fourth case, `x`, `y`, and `z` receive the values of `a`, `b`, and `c` in ascending order. Hint: `nargin` will be helpful. If fewer than three output arguments are given, then the function should not calculate the missing values. Hint: `nargout` will be helpful.

8. [Answer in English] In your own words, explain the differences between local, global and persistent variables in Matlab.

9. [Answer in English.]
a. What is short-circuiting and describe a situation in which it might be used.
b. What is the difference between persistent and global variables in Matlab?
c. What does it mean for a function to be polymorphic? Give an example of a matlab-defined polymorphic function.

10. [Answer in English.]
   a. What is the difference between a script file and a function?
   b. What is the meaning of “persistent” in Matlab?
   c. What is the meaning of “short-circuit” in Matlab?

11. Write a function, `invert()`, that takes as an argument the name of an image. The function should invert the colors in the image and display the result. Inversion implies “reversing” colors. In other words, the RGB values of each pixel should be made to \(255 - \text{Current Value}\). Use loops to accomplish this task.

12. Write a function, `rollem()`, that will simulate the casting of two dice to produce a value between 2 and 12. Use a random number generator to determine the “roll” of each die. The function should print out “Snake Eyes!” when the value of each die is 1. It should return the sum of the dice if there is only one output argument. It should return the value of each die if there are two output arguments.

13. Write a function named `countcalls` that uses a persistent variable to count the number of times the function has been called. The first time it is called, it should return 1, the next time, 2, etc.

14. Write a function named `dualplot` that plots one function versus a second function. The function takes three arguments. The first argument is a string that gives the name of some function that takes one argument. The second argument is another string that also gives the name of a function that takes one argument. The third argument is a vector. An example call might look like this,

   \[
   \text{dualplot('sqrt','tan',0:.1:2*pi);}
   \]

   This call would cause the square root of \(x\) to be plotted horizontally versus the tangent of \(x\) vertically for values of \(x\) from 0 to \(2\pi\) in steps of 0.1. The resulting plot would be equivalent to

   \[
   \text{plot(sqrt(0:.1:2*pi),tan(0:.1:2*pi);}
   \]

   You will need to use Matlab's `function function` feature, and the built-in function `feval`.

15. a) Write a function called `simplestats()` that will accept up to five input arguments and will return up to four output arguments. The input arguments should all be scalar values. The following values for output arguments should be
returned in this order: the sum of the arguments, the product of the arguments, the mean of the arguments, the standard deviation of the arguments.

b) Write a function call for `simplestats()` that will calculate the sum, product and mean of the following values: 10, 25, 31, 44

16. [Answer in English] Look at the code that is below. The loop iterates 100 times. Inside that loop there are two special cases that are sometimes set by if statements.

```matlab
for i = 1:100
    a = rand(1); b = rand(1); c = rand(1);
    if rand(1)>.5
        % Special case 1:
        b = a;
    end
    if rand(1)>.5
        % Special case 2:
        c = a;
    end
    a.
```

a. Approximately how many times would you expect this special case to be set? In other words, approximately how many times would you expect the statement, \( b = a \), to be executed during this loop?

b. Approximately how many times would you expect both “Special case 1” and “Special case 2” to be set at the same time. In other words, approximately how many times would you expect both the statements \( b = a \) and \( c = a \) to be executed during the same iteration?

17. Copy the following function into your M-file. The function takes two arguments, \( x \) and \( y \), each of which is a vector, and it returns the largest index \( i \) such that \( x(i) \) is no bigger than \( y(i) \). Insert a conditional into the while statement so that it stops when the value of \( x(i) \) exceeds that of \( y(i) \) or the length of either vector is exceeded by \( i \). Use short-circuiting to avoid index errors.

```matlab
function i = no_bigger(x,y)
    i = 1;
    while %replace this comment with a conditional
        i = i+1;
    end
    i = i-1;
```

18. Star-delta transforms (sometimes called Y-delta transforms) are a form of equivalent circuit. They are used to simplify a complex network of resistors by converting it into a form which can more easily be solved using standard series/parallel equations.
To convert from a delta-connected network to a star-connected network, the equations are:

\[ R_A = \frac{R_{AB} * R_{AC}}{R_{AB} + R_{AC} + R_{BC}} \]

\[ R_B = \frac{R_{AB} * R_{BC}}{R_{AB} + R_{AC} + R_{BC}} \]

\[ R_C = \frac{R_{AC} * R_{BC}}{R_{AB} + R_{AC} + R_{BC}} \]

To convert from a star-connected network to a delta-connected network, the equations are:

\[ R_{AB} = \frac{R_A * R_B + R_A * R_C + R_B * R_C}{R_C} \]

\[ R_{AC} = \frac{R_A * R_B + R_A * R_C + R_B * R_C}{R_B} \]

\[ R_{BC} = \frac{R_A * R_B + R_A * R_C + R_B * R_C}{R_A} \]

Write functions to convert delta to star \([Ra, Rb, Rc] = DtoS(Rab, Rac, Rbc)\) and convert start to delta \([Rab, Rac, Rbc] = StoD(Ra, Rb, Rc)\) connections.

19. Write a function called `FtoC(F)` to convert Fahrenheit temperatures into Celsius. Use the following formula:

\[ C = \frac{5}{9} (F - 32) \]

20. Write a function \(s = \text{sinewave}(\text{freq}, \text{dur}, \text{srate})\) to return a sine wave of a given frequency, duration, and sample rate. For example,

\[ s = \text{sinewave}(440, 0.5, 8192) \]

Sinewave can be generated by using \(\sin(2\pi \times \text{freq} \times t)\), where \(t = 0:(1/\text{srate}):\text{dur}\). You can use `sound(s, srate)` in MATLAB to check your function. You will hear sound vary depending on frequency, duration, and sample rate.
21. Write a predicate called `is_singular` that takes one argument that determines whether or not that argument is a singular matrix (i.e., returns 1 if for singular matrix, 0 otherwise). A singular matrix is a square matrix that does not have a matrix inverse. A square matrix has an inverse if and only if its determinant is nonzero. It also has an inverse if and only if its condition number is not infinity. Use `Help det` to determine which method Matlab recommends for determining whether a matrix is singular and use it. Hint: You can conclude that there is no inverse if the condition number is larger than 1/eps. `eps` is a function that returns the smallest possible fractional difference between two numbers in Matlab.

22. Write a predicate called `are_complex_conj` that takes two arguments and determines whether or not they are complex conjugate to each other. In order to be complex conjugate, they must be the same size, their real parts must be equal, the absolute values of their imaginary parts must be equal, and the signs of the imaginary parts of one argument must be opposite that of the corresponding imaginary parts of the other argument. For example:

- \([2 + 3i, 4 – 5i]\) and \([2 – 3i, 4 + 5i]\) are complex conjugate to each other.
- \([2 + 3i, 4 – 5i]\) and \([2 + 5i, 4 – 5i]\) are not complex conjugate to each other.
- \(2 + 3i\) and \([2 – 3i, 4 + 5i]\) are not complex conjugate to each other.

23. Write a stub for a function called `Commodore` that returns 2 output arguments if user calls it with 3 input arguments but returns 3 output arguments if user calls it with 4 input arguments. Hint: The use of `nargin` function should be helpful.

24. Write a function called `Loanbal` that is called this way:

   \[ y = \text{Loanbal}(y_0, I, \text{payment}, N) \]

   It must compute a vector \(y\) of loan balances where \(y(n)\) is the balance at month \(n\). The amount borrowed is \(y_0\), the annual interest rate is \(I\), and the amount paid each month is \(\text{payment}\). The maximum number of payments is \(N\), but the balance should never fall below zero. The formula to compute the loan balance is expressed by

   \[ y[n] = \left(1 + \left(\frac{I}{12}\right)\right) * y[n-1] - \text{payment} \]

25. Given a positive real number \(a\), the square root of \(a\) can be computed from the nonlinear difference equation

   \[ y[n] = 0.5 * \left( y[n-1] + \frac{a}{y[n-1]} \right) \]
Homework Problems

For any initial value \( y[0] > 0 \), this expression for \( y[n] \) converges to \( \sqrt{a} \) as \( n \) gets larger and larger. For example, setting \( a = 2 \) and \( y[0] = a = 2 \) gives the following first 3 estimates:

\[
y[1] = 0.5 \left( \frac{2 + \frac{2}{2}}{2} \right) = \frac{3}{2} = 1.5
\]

\[
y[2] = 0.5 \left( \frac{3 + \frac{2}{3}}{2} \right) = \frac{17}{12} = 1.416667
\]

\[
y[3] = 0.5 \left( \frac{17 + \frac{2}{17}}{2} \right) = 1.4142157
\]

Clearly, \( y[n] \) is converging to \( \sqrt{a} = 1.4142136 \).

Write a function named \( \text{sqrt103}(a, N) \) that computes the \( N \) estimates, \( y[1], y[2], y[3],..., y[N] \) of the square root of \( a \) by starting with \( y[0] = a \) and returns them as the elements of a row vector of length \( N \). If the second argument is omitted, as, for example, \( \text{sqrt103}(a) \), then it will continue the series until the \( (y[N])^2 \) is approximately equal to \( a \) to within 1/1000 of one percent of \( a \).

26. Write a function named \( \text{addup} \) that uses a persistent variable to record the total money that has been added to an account. Initially, there is no money in the account. Each time \( \text{addup} \) is called, the value of the argument \( x \) passed to the function is added to the account and the new balance is printed for the user.

27. Write two functions, \( \text{addup2} \) and \( \text{debit2} \) that will record monetary transactions to a bank account. The function \( \text{addup2} \) should accept one argument, \( x \), and add the value of that argument to the account and then print the new balance for the user. The function \( \text{debit2} \) should accept one argument, \( y \), and subtract the value of that argument from the account and then print the new balance for the user. Initially, there is no money in the account.

28. Write a function called \( \text{simplestats} \) that will take up to five input arguments and will return up to four output arguments. The input arguments should all be scalar values. The following values for output arguments should be returned in this order: the sum of the arguments, the product of the arguments, the mean of the arguments, the standard deviation of the arguments. Hint: the use of \( \text{nargin} \) and \( \text{nargout} \) will be very useful.
29. Copy the following code into your program:

```matlab
function val = bigger2(x,y)
    if x >= y
        val = x;
    else
        val = y;
    end
```

The function, `bigger2`, takes two arguments and returns the larger value. Use `feval` and the `bigger2` function to write two new functions, `bigger3` and `bigger4` that take three and four arguments respectively and returns the largest value. You should use the `function function` feature of Matlab as well as the `feval` function to solve this problem.

30. Use the following code for this problem:

```matlab
function ii = is_equal(x,y)
    ii = 1;
    is_find = false;
    while  %replace this comment with a conditional of your choosing
        if x(ii) == y(ii)
            is_find = true;
            ii = ii + 1;
        end
    end
    if (is_find == false)
        ii = 0;
    end
end
```

Modify the code above to accomplish the following: the function should take two arguments, `x` and `y`, each of which is a vector. You want the function to return the value of the first index, `ii`, such that `x(ii)` is equal to `y(ii)`. If this condition does not exist, the function should return `0`. Use short circuiting to stop the loop once the desired index is found. Additionally, you should be sure to include code to avoid index length errors in your loop.

31. Write a function called `my_log(x,N)` that will use a for loop to calculate an approximate natural log of `x`, using `N` terms in this summation:

\[
\log(x) \approx 2 \sum_{n=1}^{N} \frac{1}{2n - 1} \left( \frac{x - 1}{x + 1} \right)^{2n-1}
\]
Include a check that x is positive (“positive” means > 0, not ≥ 0). If x is negative, your function must print “x must be positive” and return zero.

32. a) The Taylor series approximation for $e^x$ is given as:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Write a function, `taylor(x)` that will use this summation to calculate and return the value of $e^x$ for any x entered by the user. You should continue to sum terms until the absolute value of the last term added is < 0.001. Hint: Matlab’s built-in function `factorial` will be useful.

b) Write a function, `taylor_compare(x)`, that will take the Taylor series approximation for $e^x$ and will return the percentage error between the approximation and the result from the Matlab function exp(x)

33. Write a function, `permute(n,r)`, that will take two arguments, n and r, and return the permutation $^nP_r$. Permutation is calculated using the following formula:

$$^nP_r = \frac{n!}{(n-r)!}$$

You should write a single subfunction: `facti()` that will calculate the factorial values needed in your calculation. You will also wish to make your function robust by ensuring the required arguments have been passed to the function and that $n \geq r$.

34. The greatest common divisor of two positive integers is the largest integer that divides them both. For example, the greatest common divisor of 9 and 6 is 3. Write a function that accepts two integer input arguments and returns their greatest common divisor.

35. Suppose you are given a two-dimensional cell, `c`, whose values contain contact information for a company’s employees. Each row contains information for a different employee. The first column contains the employee’s ID number, the second column contains their name, the third column contains their office location and the fourth column contains their work phone number. Write a function called `cell_to_struct()` that will take the information contained in cell `c` and transfer it to a new structure array called `personnel`. The structure array personnel should contain the following fields: `id_num`, `emp_name`, `office_loc`, and `phone`. You may assume that the cell is fully populated, that is, each row contains data in all columns.
36. Write a function called `solve_and_check` that solves $Ax=b$ for $x$ when given $A$ (first argument) and $b$ (second argument), returns $x$, and also checks the answer is giving to see whether it is correct. If not, it still returns the least-squares solution, but it also issues the statement “Warning: At least one of the equations is not solved to within round-off by this solution.” To determine whether or not to issue the warning, you may use only one IF statement, or possibly one IF-ELSE statement with only one conditional, and it should do that if and only if at least one of the elements of $Ax-b$ is larger in absolutely than $b/10^6$, and no loops are allowed in this function. (HINT: An if statement’s conditional is deemed true if and only if all of the elements of the conditional are true.)

37. Write two functions, `deposit()` and `withdraw()` that will record monetary transactions to a single (identical) bank account. Both functions should accept one input argument. `deposit()` should add its argument’s value to the bank account. `withdraw()` should subtract its argument’s value to the bank account. Both functions should print the new balance after the transaction, using `fprintf()`. Additionally, the user should be notified if they’re overdrawn (that is, if they have a negative balance). Initially, there is no money in the account. HINT: Use a global variable.
Homework Problems

Homework Set 5

Problems that do not explicitly say "Answer in English" are to be answered in Matlab.

User-defined data types

1. Write three functions called `arrives(x)`, `leaves`, and `queue(q)`, which together implement a queue of Matlab numbers. The function `arrives(x)` puts its argument in the queue. If there are already numbers in the queue, it puts `x` at the back of the queue. The function `leaves` removes a number from the queue. The function `queue(q)` is polymorphic. It can be called with no arguments or one argument. If it is called with no arguments it returns a list of numbers in the queue. If it is called with one argument, it replaces the current queue with the value of the argument. You may NOT use any global variables. You may assume that the user always calls `arrives` with arguments of the same numeric type. [Hint: Use a persistent variable to hold the current list inside `queue`. Use the stack of cards example given in the Highlights for this class as an example.]

Strings

2. Write a function called `countspaces` that will accept one input argument, a string, and return as an output argument the number of space characters in that string. You may assume that the argument passed is guaranteed to be a string. 
   Hint: Use the `isspace` function.

3. Write a function called `rot_wrap` that takes two input arguments and returns one output argument as follows:
   \[ \text{code} = \text{rot_wrap} \left( \text{message}, n \right) \]
   where `code` and `message` are each strings (i.e., vectors of type `char`), and `n` is a positive number. These two strings are the same at every position except at positions of lower case letters. If it is a lower-case letter, then there is a change. The change is that `n` is added to the ASCII code for the letter. Furthermore, if the result of the addition is outside the range of the codes for the lower-case letters than the result is “wrapped” around to the beginning of the set of lower-case letters. It is as if the alphabet were written on a circle and the new letter is the result of shifting by `n` around the circle as shown by the example in the figure below. The example shows that for `n = 5`, the letter ‘x’ will be changed to the letter ‘c’. If the second argument is omitted, then the default `n = 5` should be used. If the second argument is negative, the function must give an error message and return. If the first argument is not a string, the function must give an error message and return.
Recursion

4. [Answer in English] For each of the following situations, give the maximum number of activation records on the stack. In each case assume that there is always one record on the stack for the command window.

a. A is called from the command line, A calls B, B calls C, C calls D, D returns, C returns, B returns, A returns.

b. A is called from the command line, A calls B, B calls C, C calls D, D returns, C calls E, E calls F, F returns, E returns, C returns, B returns, A returns.

c. A is called from the command line, A calls B, B calls C, C calls C, C returns, C returns, B returns, A returns.

d. A is called from the command line, A calls B, B calls C, C calls A, A returns, C returns, B returns, A returns. (This is an example of indirect recursion. Indirect recursion is the situation that happens when A calls B and before B returns another function calls A.)

e. The function fibonacci (Ch. 9, Fig. 15) is called with month = 5.

5. Write a recursive function called \texttt{hw5\_5} to implement this recursive definition:

\[ f(n) = \begin{cases} 2, & \text{for } n \leq 1 \\ \log |f(n-1)|, & \text{otherwise} \end{cases} \]

6. Write a recursive function called \texttt{hw5\_6} to implement this recursive definition:
\[ f(n) = 2, \text{ for } n \geq 1 \]
\[ f(n) = \sqrt{|f(n+1)|} - n, \text{ otherwise} \]

7. The following function takes a very long time to execute when its argument is 20 or more:

```matlab
function a = hw5_slow(n)
    if n < 1;
        a = 10;
    else
        a = hw5_slow(n-1) + 1/ hw5_slow(n-1);
    end
end
```

Change it so that it is fast (should be almost instantaneous for \( n < 500 \)) but still calculates the same thing. Call your new version, `hw5_7`.

8. Consider the function `fibonacci`
   a. Make a new version, called `fib_sooner` that is the same as `fibonacci`, except that the infertile interval is one month instead of two.
   
   b. Make a new version, called `fib_faster` that is the same as `fibonacci`, except that there are two pairs per month born to each fertile pair instead of one.
   
   c. Plot the growth of both on the same graph. Which one grows faster? (You do not have to submit a function or a plot. Just say which is faster.)

9. Consider the following recursive function:

```matlab
function a = permutations(v)
    vlen = length(v);
    if vlen <= 1, a = v;
    else
        a = [];
        for jj = 1:vlen
            tail = permutations(v([1:jj-1, jj+1:end]));
            H = size(tail,1);
            head = v(jj)*ones(H,1);
            a = [a;[head, tail]];
        end
    end
end
```
end

a. [Answer in English] Run the function on some vectors and look at the output of the function. What does the function do? (i.e., what is the purpose of the function?).

b. [Answer in English] What is the base case?

c. Make the smallest alteration to the recursive case that you can so that the rows of the output appear in reverse order. By “reverse” we mean here that the numbers on a row are the same, but the first row becomes the last row, the 2nd row becomes the next-to-last row, etc. In other words, suppose the answer produced by the code above for argument \( V \) is array \( M \). Then, after you have changed the function, its answer, when given the same argument \( V \), would be \( M(\text{end}:-1:1,:) \). Name your function \texttt{permutations}.

10. [Answer in English] The following code contains an error. It can be corrected by changing one line. How can the error be corrected? Give the answer in words. You do not have to submit an altered function.

```matlab
function m_min = min_index(v)
%MIN_INDEX find index of smallest element
% MIN_INDEX(V) returns the index of the smallest element in the vector V.
% Method uses recursion.

if isempty(v), error('v is empty');
elseif length(v) == 1, m_min = 1;
else
    m_min = 1;
    mp2_min = min_index(v(2:end));
    if v(mp2_min) < v(m_min)
        m_min = mp2_min;
    end
end
```
Searching problems

1. [Answer in English] Consider the list: \( L = [4 \ 5 \ 12 \ 14 \ 17 \ 24 \ 25 \ 27 \ 28 \ 31 \ 35 \ 36 \ 39] \).

   Suppose you are using the binary search algorithm. For each of the following targets, give a list of the numbers in \( B \) (not their indices) that would be compared with the target in the order in which they would be compared with it.
   
   a. 24  
   b. 4  
   c. 17  
   d. 15

2. [Answer in English] Suppose the binary-search algorithms are applied to unsorted lists:
   
   a. Will the recursive version ever respond that a target is on the list, when that target is in fact not on the list?  
   b. Will the recursive version ever give the wrong index for a target that is on the list?  
   c. When the recursive version responds that a target is not on the list, will it ever be wrong?  
   d. Will the iterative version ever respond that a target is on the list, when that target is in fact not on the list?  
   e. Will the iterative version ever give the wrong index for a target that is on the list?  
   f. When the iterative version responds that a target is not on the list, will it ever be wrong?

Sorting Problems

3. [Answer in English] Suppose that in the function, \texttt{selection\_sort}, the following portion,

   \[
   \text{if } m_{\text{min}} \approx m \\
   \quad \text{temp} = v(m) \\
   \quad v(m) = v(m_{\text{min}}) \\
   \quad v(m_{\text{min}}) = \text{temp} \\
   \text{end}
   \]

   is replaced by
temp = v(m);
v(m) = v(m_min);
v(m_min) = temp;

a. How will the output be changed?

b. Will the average number of swaps be increased, decreased, or unchanged?

c. Will the number of statement executions required for a sort always be increased, always be decreased, or is it indeterminate (i.e., three possible answers).

4. [Answer in English] The sort function, selection_sort, which is given in Chapter 10, is applied to each of the following vectors. In each case, give the vector after the 3rd swap.
   a. 621, 687, 74, 479, 118, 142, 467, 445
   b. 136, 20, 28, 82, 345, 334, 303, 939, 298, 947
   c. 249, 941, 705, 531, 718, 964, 449, 660, 478, 928, 841

5. [Answer in English] In the merge_sort function given in Chapter 10, the subfunction merge_sorted_lists(v1,v2) combines v1 and v2 into one list that is sorted. Here is a partial listing of that function:

```plaintext
function x = merge_sorted_lists(x1,x2)
    % The elements of x1 and x2 are assumed to be
    % in ascending order. Combine them into
    % ascending order and return them in x.
    x = [];
    while ~isempty(x1) && ~isempty(x2)
        if x1(1) < x2(1)
            x = [x, x1(1)];  % add x1(1) to x
            x1 = Answer(a);
        else
            x = [x, x2(1)];  % add x2(1) to x
            x2 = Answer(b);
        end
    end
    x = [x,x1,x2];
```

a. What goes in place of Answer(a)?
b. What goes in place of Answer(b)?
Problems on algorithmic complexity

6. [Answer in English]. Listed below is the worst-case time behavior $t(N)$ of some fictitious algorithms, where $N$ is the size of the input. Give the order of each algorithm using “Big-Oh” notation:

a. $2N^3 + 14N^2 + 3$

b. $a + b \log(N^3)$

c. $6\log(N) + 2\log(\log(N))$

d. $4 + 3.2N + 2/N$

Problems on symbolic mathematics:

Problems 7 through 9 have to do with the following situation, depicted in the figure:
A car is traveling at night along a highway. The highway is shaped like a parabola with its vertex at the origin. The car starts at a point 100 meters west and 100 meters north of the origin and travels in an easterly direction. There is a statue located 100m east and 50m north of the origin. Let $x$ represent the distance of the car east from the origin (negative values of $x$ indicate that the car is west of the origin); let $y$ represent the distance of the car north from the origin.
7. [Answer in English]. Give an equation that relates $y$ to $x$. (Hint: We are simply asking for the equation of a parabola. You can look it up anywhere, so this problem does not count much.)

8. [Answer in Matlab] Using Matlab’s symbolic mathematics facility, give commands that will determine the position of the car on the highway at which the car’s headlights will illuminate the statue. Assume that the statue is a point object.

9. [Answer in English]. What is the equation of the headlight’s illumination when it strikes the statue? Give the numerical values of the coefficient and the constant in this equation. You will probably need to use Matlab to find them.

10. [Answer in Matlab] When gas expands in a cylinder with radius $r$, the pressure at any given time is a function of the volume: $P = P(V)$. The force exerted by the gas on the piston (as illustrated in the figure below) is the product of the pressure and the area:

$$F = \pi r^2 P.$$  The work done by the gas when the volume expands from $V_1$ to volume $V_2$ is therefore:

$$\int_{V_2}^{V_1} PdV$$

In a steam engine the pressure $P$ and volume $V$ of steam satisfy the equation $PV^{1.4} = k$, where $k$ is a constant. (This is true for adiabatic expansion – that is, expansion in which there is no heat transfer between the cylinder and its surroundings.) Using Matlab’s symbolic mathematics facility, give commands that calculate the work done by the engine during a cycle when the steam starts at a pressure of 160 lb/in² and a volume of 100 in³ and expands to a volume of 800 in³.