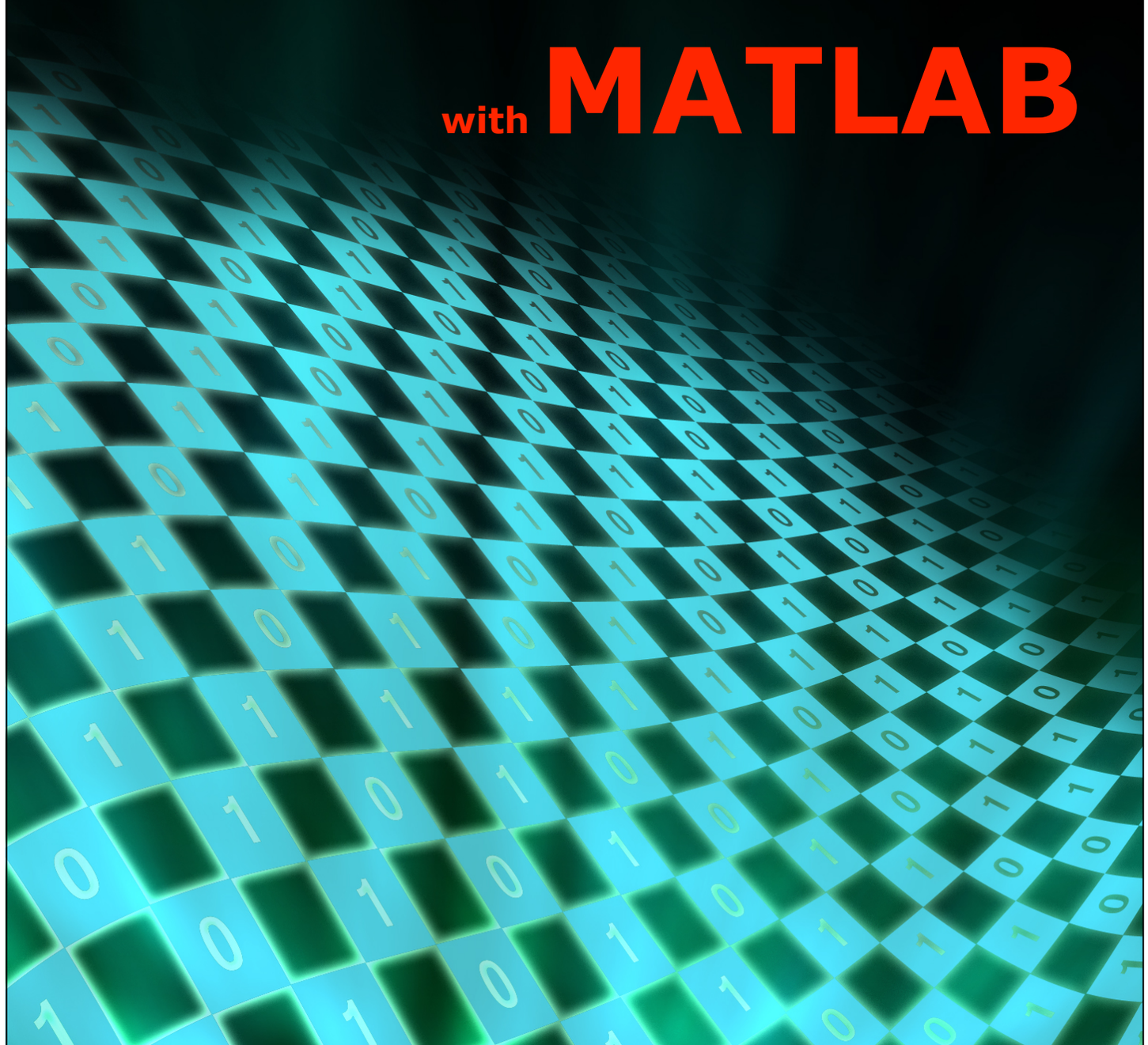
COMPUTER PROGRAMMING

with MATLAB



COMPUTER PROGRAMMING WITH MATLAB

J. MICHAEL FITZPATRICK AND ÁKOS LÉDECZI

Computer Programming with MATLAB
J. Michael Fitzpatrick and Ákos Lédeczi
1st Revised PDF Edition
June, 2015

DEDICATION

This book is dedicated to our wives,

***Patricia Robinson** and **Barbara Lengyel**,*

for their patience and understanding while we were devoting so much time to this book.

THE AUTHORS

John Michael (Mike) Fitzpatrick, Professor Emeritus of Computer Science at Vanderbilt University, retired from the classroom in 2011 after teaching at the college level for thirty-five years, teaching computer science for twenty-nine years, and teaching computer programming with MATLAB® for eleven years. He received a BS in physics and an MS in computer science from the University of North Carolina at Chapel Hill and a PhD In physics from Florida State University in Tallahassee. He has been a member of the Vanderbilt faculty since 1982, where he uses MATLAB in his research in computer-assisted surgery. He is married with two children and lives in Nashville, Tennessee.

J. Michael Fitzpatrick

Ákos Lédeczi, Associate Professor of Computer Engineering and Senior Research Scientist at the Institute for Software Integrated Systems at Vanderbilt University, has been doing research on model-integrated computing and wireless sensor networks for a couple of decades. He has been teaching computer programming with MATLAB for eight years now. He received an MS from the Technical University of Budapest in Hungary and a PhD from Vanderbilt University, both in electrical engineering. He has been a member of the faculty at Vanderbilt University since 1998. He is married with three children and lives in Nashville, Tennessee.

Ákos Lédeczi

Table of Contents

Preface

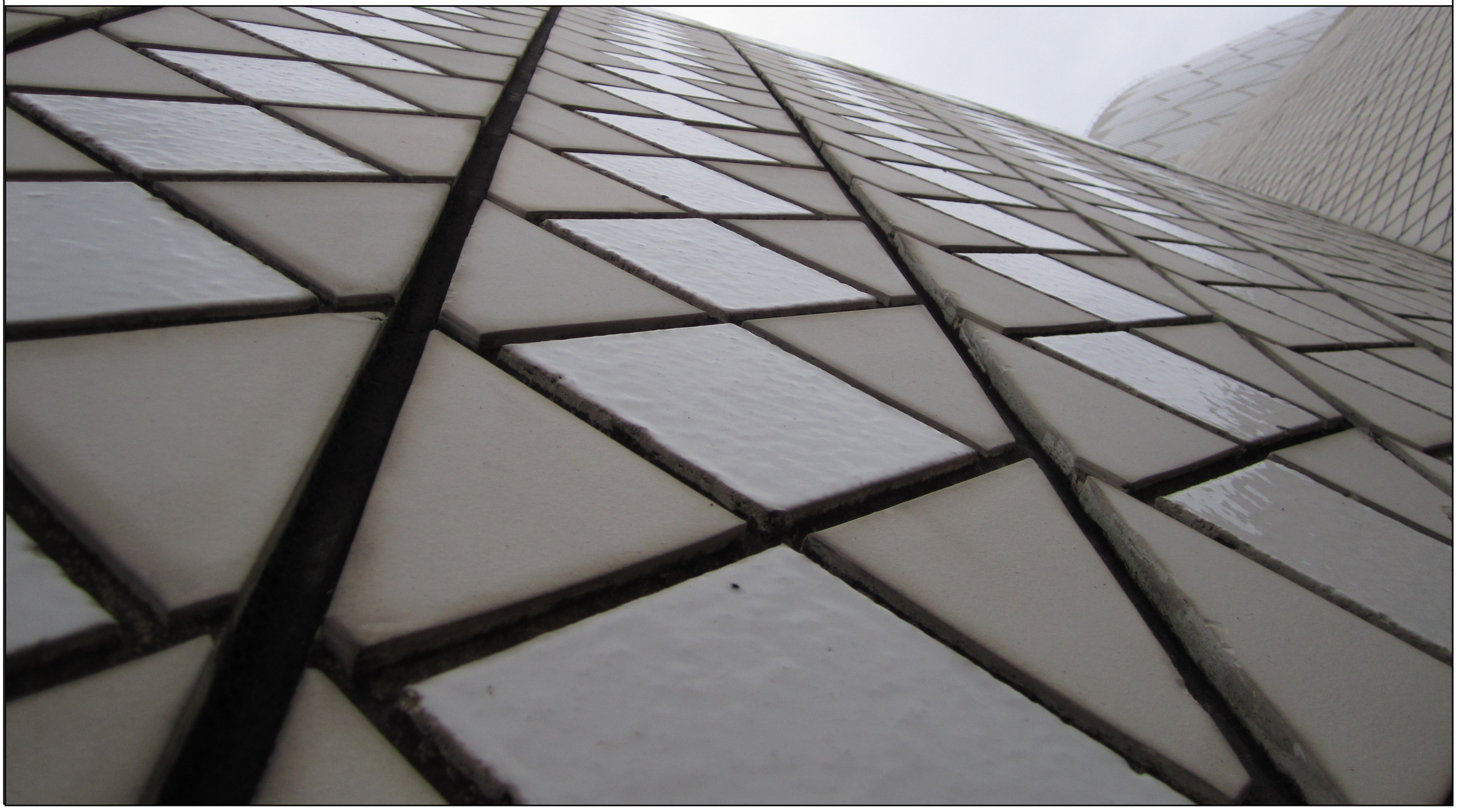


Photo credit: Anna Ledeczi

The primary purpose of this book is to teach computer programming to those with little to no previous experience. It uses the programming system and language called MATLAB® to do so because MATLAB is easy to learn and, at the same time, is an extremely versatile and useful programming language and programming environment. MATLAB is a special-purpose language that is an excellent choice for writing moderate-size programs (let's say, fewer than a thousand lines) that solve problems involving the manipulation of numbers. 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. As a result, MATLAB is being used worldwide in a great variety of domains from the natural sciences through all disciplines of engineering to finance and beyond, and it is heavily used in industry. Hence, a solid background in MATLAB is an indispensable skill in today's job market.

Nevertheless, this book is not merely a reference manual for MATLAB or a MATLAB tutorial. It is an introductory programming textbook that happens to use MATLAB to illustrate general concepts in computer science and programming. As a side effect, the reader will gain a solid foundation in MATLAB, but an experienced computer programmer who wants merely to learn MATLAB should probably look elsewhere.

This book is a good fit for an introductory college-level course in computer programming for engineering and science students. In fact, it is being used as the textbook for such a course at Vanderbilt University. It serves the dual purpose of teaching computer programming and providing a background in MATLAB, which is used in higher-level courses in many majors.

This book is also suitable to teach programming to high school students. The material assumes no background in mathematics that is not part of standard high school curricula, and MATLAB is much more accessible as an introduction to programming to the average student than Java, C, C++, or other general-purpose languages.

Logistics

A disadvantage of using MATLAB as the language of choice for this book is that the MATLAB programming environment is not free. However, a student version is available for the reasonable price of $99. It can be ordered directly from MathWorks®, the company who develops and distributes MATLAB. Furthermore, many colleges have site licenses for the software. Note that MATLAB comes with many additional products, such as Simulink®, Stateflow®, and a number of specialized "Toolboxes" that add to its power. None of these is required for this book.

This eBook comes with a companion website (http://cs103.net) that contains program listings from this book, as well as solutions to selected practice problems. The website also provides links to 11 hours worth of video lectures by the authors. All sections in Chapter 2 include practice problems at the end. Each odd-numbered problem is followed by a red question mark. Clicking on it takes you to the solution on the website. Note that problems come in pairs: each one with a solution is followed by a similar, typically somewhat more difficult problem with no solution.

Style

Two distinct stylistic features have been employed in writing this textbook to make it more useful.

First, this book places more emphasis on the general concepts from the discipline of computer science than does the typical introduction to MATLAB. Both the terminology (e.g., "polymorphism", "stack frame") and some of the topics (e.g., recursion, object-oriented programming) allow the student to become conversant in the language of the computer scientist while learning the MATLAB approach to numerical problem solving (e.g., matrix and array op-

erations, vectorization). Care has been taken to keep the usage of terminology consistent. As a result, the student who moves from engineering or the physical sciences into computer science or vice versa, does not have as much new to learn and does not have to "unlearn" anything. A list of specialized terms is provided in a combined index and glossary at the end of the book, each such term is highlighted in blue when it is introduced in the text itself.

Second, much of the material in this book is presented in a graduated tutorial style, i.e., concepts are illustrated by means of practical examples. The early sections include lots of introductory tutorial material to help the reader get started. As readers gain experience with MATLAB and with the concepts of computer science and move into later sections, they will be able to absorb new material more readily. As that happens, the style becomes less tutorial, and the rate at which new material is introduced increases, but from beginning to end, this book emphasizes the approach of teaching via examples.

A wise Chinese proverb says, "I hear and I forget; I see and I remember; I do and I understand." Nowhere is this adage more true than in computer programming. The only way to understand it is to do it. The sets of worked examples and practice problems within each section provide ample opportunity for the reader to practice the new material.

About The PDF Edition

The original eBook upon which this edition is based was created using Apple's iBooks® Author. As not everybody owns an iPad or a Mac computer, we decided to create a PDF edition so that the book can be enjoyed on any computer. Unfortunately, PDF does not support all the interactive feature that an Apple textbook does. The most significant limitation is that links that allowed the user to jump from one place in the book to another, do not work in this edition. These missing links would be most problematic for navigating from the table of contents to specific chapters and sections and from the glossary entries to the pages on which they appear in the text. To solve that problem, we have done two things: (1) We have added bookmarks at the beginning of each chapter and section, so that the PDF bookmarks panel now serves as an electronic table of contents with the same facility as that of the Apple textbook for navigation to chapters and sections. (2) We replaced the glossary with a combined Index and Glossary that includes both page numbers and definitions. Also, video clips and animations that play inside the original book are available on YouTube and are properly linked from this PDF edition. Also, links that point to outside resources work just fine from this book as well.

Software Versions

The MATLAB examples shown in this book were tested using MATLAB Versions R2012a and R2012b. The operating systems used include both Windows 7®and OS X® 10.7.

Acknowledgements

Material for this eBook was developed at Vanderbilt University for a course for freshmen in the School of Engineering. We would like to acknowledge the late John D. Crocetti, who co-wrote with one of us (JMF) an earlier, traditional textbook on the same subject. He was a good friend and a great teacher, and he will be missed. We would also like to thank over two thousand students at Vanderbilt who, during the years from 2000 through 2012, used that textbook and provided valuable feedback that has contributed to this eBook. We are grateful to Bill Hilton, Jose Santos, Barry Duncan, John Cardoza, Marko Rokvic, Madison Stott, Maria Linn, Charles Gagne and other students who found typos and other mistakes in the first edition of this book.

We are indebted to Szabolcs Kövi, who granted permission to use his wonderful song, Secret Garden, from the album, Cycle - Best of Szabolcs Kövi, to accompany the introductory slideshow. Anna Ledeczi has contributed her voice to Movie 3.3. Finally, we gratefully acknowledge Tamás Fodor who designed the cover art for this book.

An invitation...

CHAPTER 1

Getting Started

It is time to embark on our journey to learn computer programming and MATLAB at the same time. The only way to learn programming is by doing. You are encouraged to try out each new concept as it is introduced in the book. By doing so, you will understand the material more quickly and more deeply, you will discover common mistakes early on, and you will remember how to avoid them.

Introduction to MATLAB

Objectives

MATLAB is both a powerful programming language and a convenient programming environment. We will introduce both in this section:

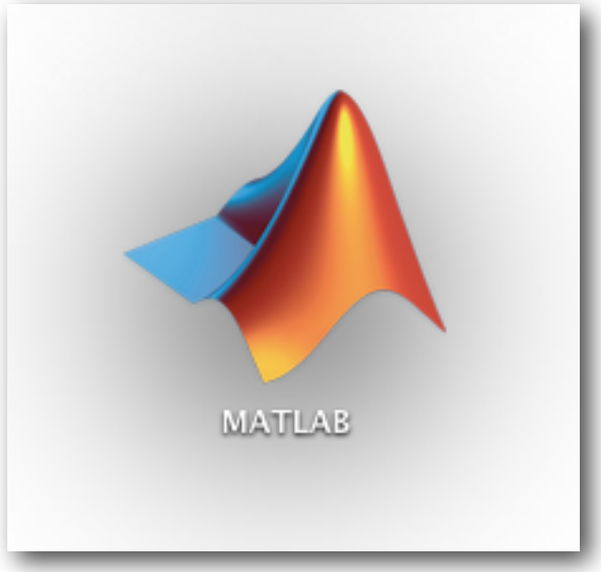
(1) We will learn how to start MATLAB and how to specify folders to work in.

(2) We will learn how to use MATLAB as a sophisticated calculator by entering commands into its Command Window and how to get help with unfamiliar commands.

(3) We will encounter terminology from computer science involving the rules for writing statements and methods of execution.

(4) We will use MATLAB's editor to write programs and save them in "M-files" .

(5) We will be introduced to MATLAB's powerful data visualization facilities.



If you are into engineering, mathematics, economics or the natural sciences, chances are that you are familiar with the MATLAB logo above. It is time to learn what MATLAB is all about!

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 ideally have MATLAB version R2012a or later installed on your computer. MathWorks calls each new version a new "Release". R2012a is Release 2012a, R2012b is Release 2012b, R2013a is Release 2013a, etc. Recently two versions have been released each year—first a, then b,

Earlier versions can be used as well with this book but may, in a few cases, not behave as expected.

When MATLAB is installed on your computer, the MATLAB icon above should appear on your Mac® launchpad or your Windows desktop. Clicking (Mac) or double clicking (Windows) it will start MATLAB.

Figure 1.1 *The MATLAB desktop (R2012a and earlier)*

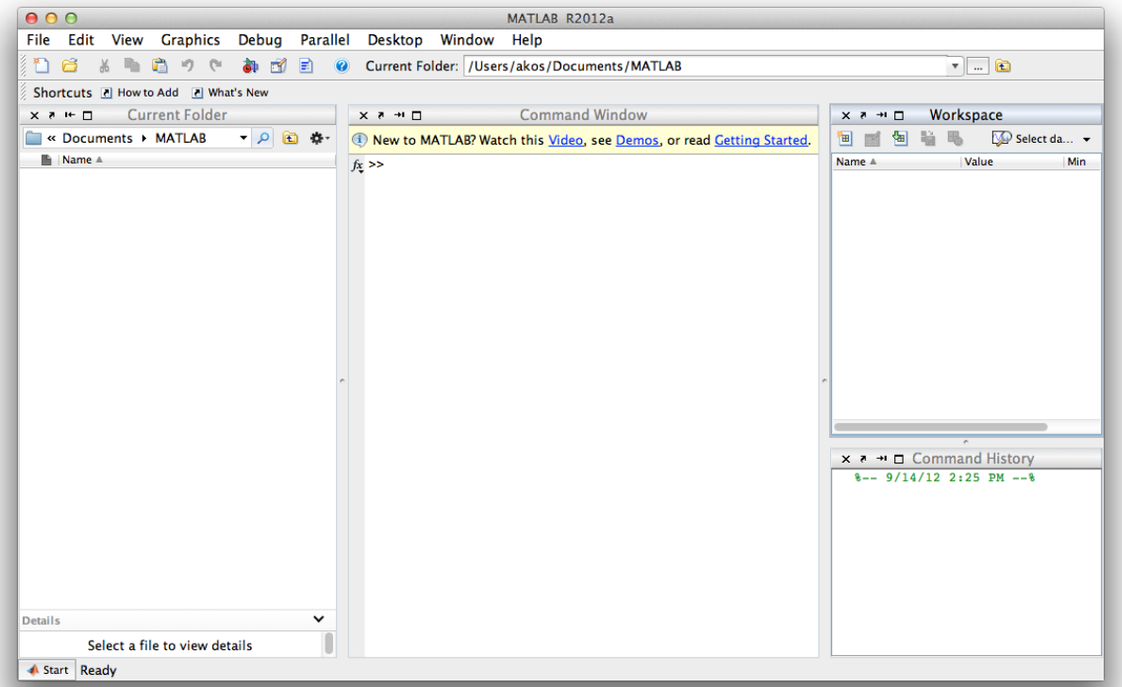
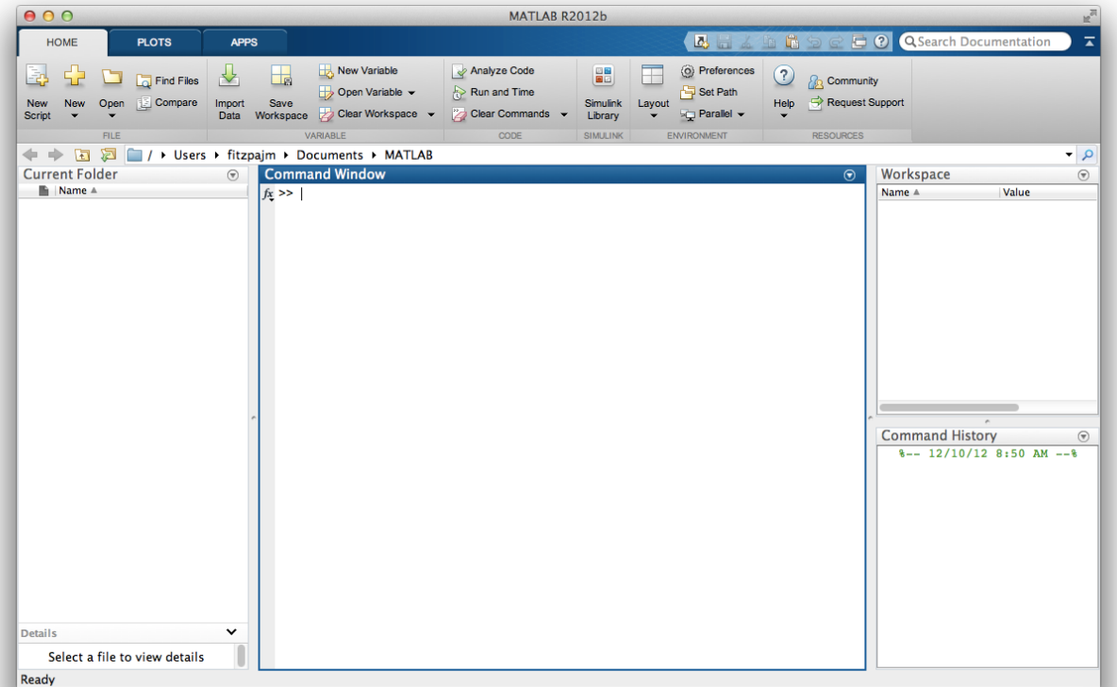


Figure 1.2 *The MATLAB desktop (R2012b and later)*



When you start MATLAB, the "MATLAB Desktop" will appear. Its layout will vary according to the operating system and version of MATLAB. Figure 1.1 shows versions R2012a and earlier; Figure 1.2 shows version R2012b and later. In both figures MATLAB is running on a Mac using the OS X operating system. MATLAB's layout changed in only minor ways for many years until the advent of R2012b, which replaced the menus and toolbars at the top of the Desktop with a "ribbon", much as Microsoft did when it brought out its new Office® applications in 2007. Version R2013a continued the ribbon interface. There are other changes too, but almost all the changes are cosmetic; the basic MATLAB functionality remains unchanged. All the functionality previously available in the toolbar and menus is still there in the ribbon, but it is distributed among three tabs, named Home, Plots, and Apps. The Home tab is by far the most useful tab for programming purposes. We will focus on this tab in this book.

The Desktop displays four windows: the "Workspace" at the upper right, the "Command History" at the lower right, the "Current Folder" at the left and the "Command Window" in the middle. Commands are typed into the Command Window. In fact, if you click inside that window, as we have done in Figure 1.2, you will see a blinking vertical bar (|) just to the right of the **>>** showing you where your first command will appear. Before you start, you should know that when you are ready to quit MATLAB, you can type **`quit`** in that window, like this,

```
>> quit
```

and hit Enter. MATLAB will then close all open windows and quit. If you have files open, you will be asked about saving them and given a chance to save them. You can also quit by clicking with the mouse on the red button at

the top left corner of the MATLAB window on a Mac or the × at the top right of the MATLAB window on Windows.

Current Folder

When MATLAB is running, there is a special folder, called the "Current Folder" where MATLAB expects to find files that you want to open and where it will store files that you want to save. You should change this folder from the one selected by MATLAB to the one in which you want to keep your files. Some people call a folder a “directory”, which is a synonymous term. We will therefore sometimes refer to a folder as a folder-directory.

Changing the current folder in versions R2012a and earlier

To change the current folder in versions R2012a and earlier, move your mouse cursor over the button with three dots, pointed at in Figure 1.3 by the red arrow and click.

A window will pop up showing the folder structure that is accessible from your computer. Click on the folder you wish to use and then click OK. You will then see that the name of the current folder that appears in the space to the left of the button with the three dots has now changed to the one that you chose and so has the name at the top of the “Current Folder” window on the left side of the MATLAB desktop (green arrow). Any folders and files within the current folder will simultaneously appear in the Current Folder window. You can also change the current folder by typing the name directly into that box in the Current Folder window.

Changing the current folder in versions R2012b and later

To change the current folder in versions R2012b and later, move your mouse cursor over the File icon with the green arrow indicating the opening of a file, pointed at by the large red arrow in Figure 1.4, and click.

Figure 1.3 *Changing the current folder (R2012a and earlier)*

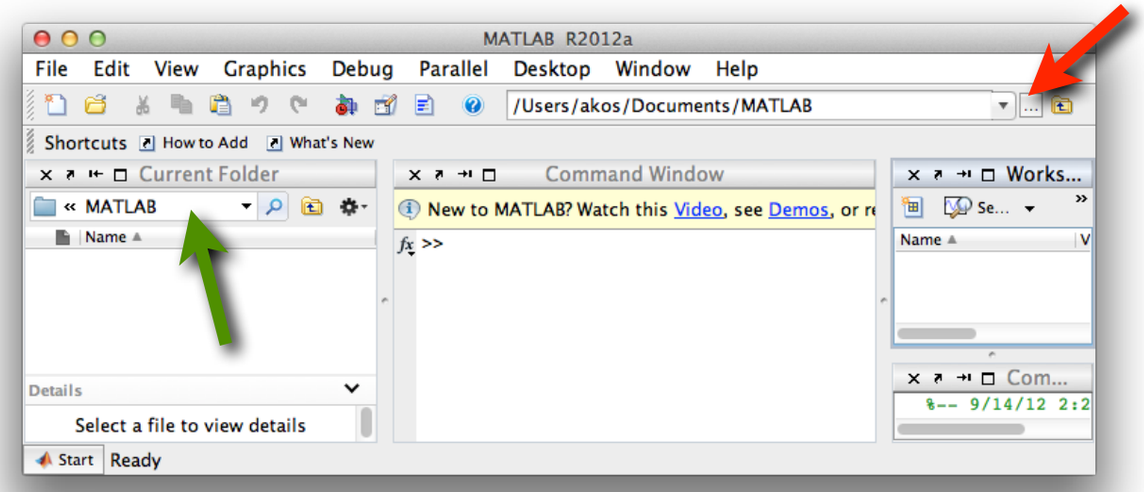
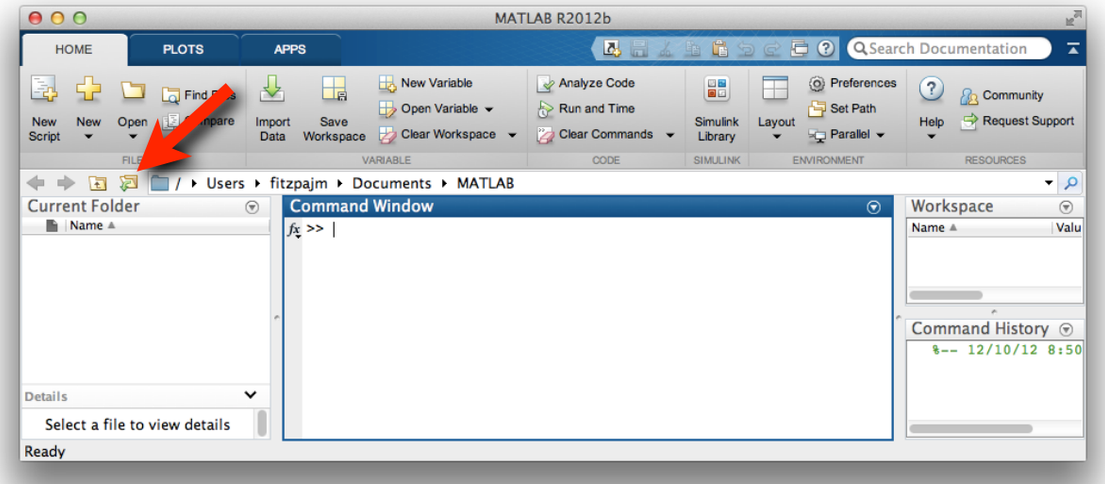


Figure 1.4 *Changing the current folder (R2012b and later)*



A window will pop up showing the folder structure that is accessible from your computer. Click on the folder you wish to use and then click OK. You will then see that the name of the current folder that appears in the space to the right of the file icon has now changed to the one that you chose. Any folders and files within the current folder will simultaneously appear in the Current Folder window.

And speaking of the Current Folder window, if there are subfolders in that window, then you can make any one of those become the current folder by simply double-clicking it. This method works for all versions of MATLAB.

The Path

When MATLAB fails to find a file that it is looking for in the current folder, it does not give up. It looks in other folders. It follows a path searching in one folder after another. A "path", in this context, is a list of folders 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 folders.

Adding folders to your path and removing them from your path.

You should add to your path any folders in which you keep files that you have created for use with MATLAB. You may choose to use only one folder 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 folders at the bottom of the list, i.e., at the end of the path, not the top. Let's add a folder to the end of your path. First, create a new folder using your operating system (e.g., Windows, Mac OS) and then, within MATLAB, add the folder that you have created to your path as follows:

First, you need to open the "Set Path" window. To open that window in versions R2012a and earlier, click on File menu at the top left of the MATLAB desktop and then click on Set Path command. The window will pop up. To open it it in versions R2012b and later, click the Set Path button in the HOME ribbon, pointed at by the red arrow in Figure 1.5. When the Set Path window pops up which is shown in Figure 1.6, you will see some buttons on the left and a display of the path on the right. The beginning of your path is at the

Figure 1.5 *Opening the Set Path window (R2012b and later)*

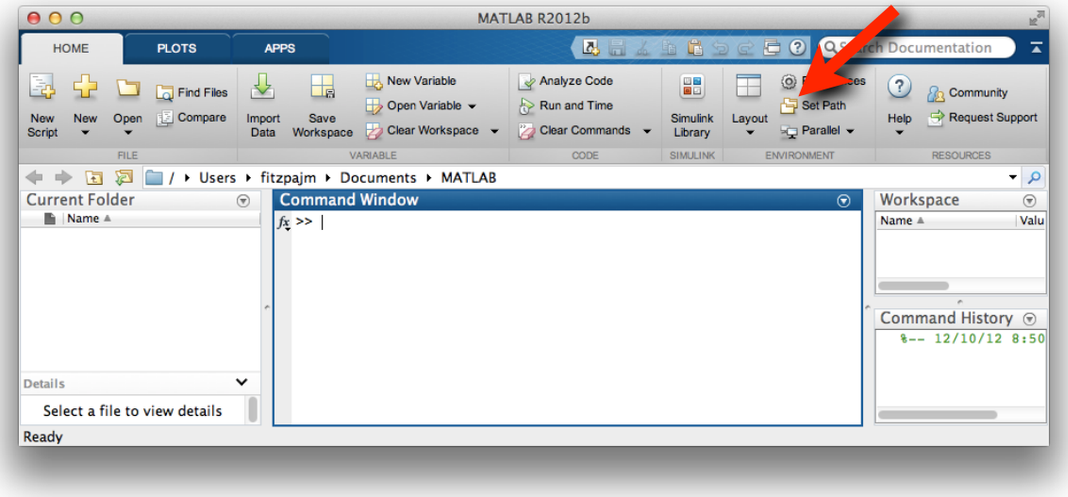
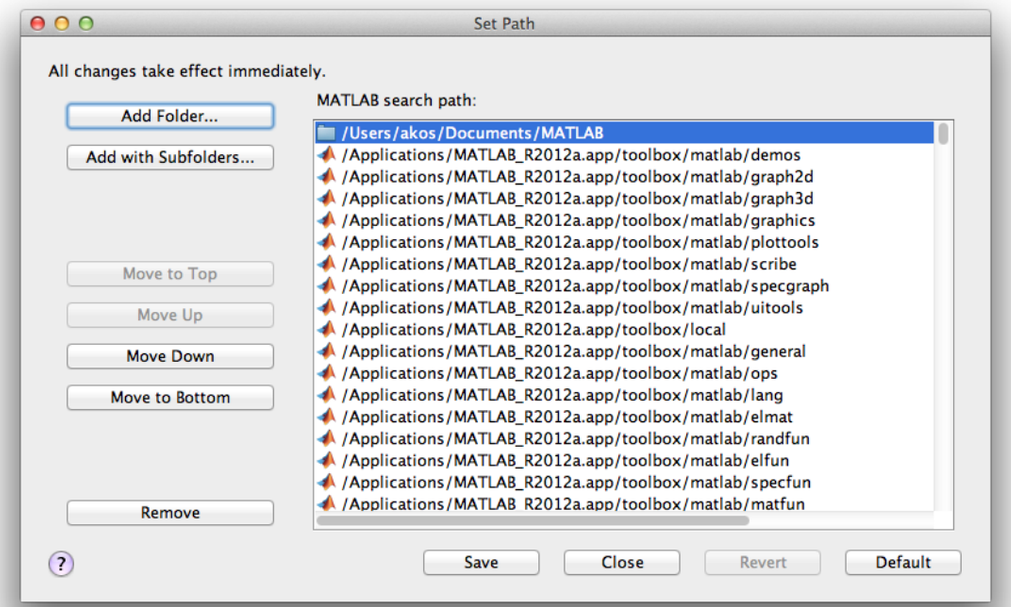


Figure 1.6 *Setting the path (same for most versions)*



top of the display. The entire path can be viewed by using the scrollbar at the right. Click “Add Folder…” and a window will pop up called “Select a Directory” or “Browse for Folder”. Inside this window, click on the folder that you wish to add and then click “Open” or “OK”. The folder will appear at the top of the path. Finally, click “Move to Bottom” to move the folder to the end of the path. If you wish to remove a folder from the path click (at the risk of being obvious) “Remove”. While you are in that window you can add and/or remove as many folders as you wish.

If you are running MATLAB on your own machine and you want to have this folder 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 left (Mac) or upper right (Windows) corner of the window. Otherwise, click “Close”. If you have changed the path but 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 folder appear on your path the next time you run MATLAB, click “Yes”. Otherwise, click “No”.

The Command Window

The symbol **>>** in the Command Window (the window in the middle of the MATLAB desktop) indicates that the genie inside MATLAB is ready to obey your every command. In computer-science parlance, this symbol is called a “prompt”. A **prompt** is a symbol, or symbols, used by a program (MATLAB is a program) while it is running to request an action from the user–such as typing a command. Thus, MATLAB is using the double “greater-than sign” **>>** as its command prompt. As mentioned at the beginning of this section, when you are ready to quit MATLAB, you can type **quit** in the Command Window or click on the red button at the top left corner of the MATLAB window on a Mac or on the × at the top right of the MATLAB window on Windows.

Alternatively, you can quit on a Mac by typing **⌘q** (while depressing the Command key, press q), and you can quit on Windows by typing Ctrl-q (while depressing the Ctrl key, press q). Finally, in version R2012a, you can select Exit MATLAB from the File Menu at the top left.

Regaining control

When you issue a command to MATLAB, it takes control and begins its work. Sometimes that work takes a long time, longer perhaps than you want to wait. So, before you learn how to give commands, it is important to know how to abort them. On both Mac OS and Windows, if you ever find MATLAB taking too long to execute a command, you may type Ctrl-c to abort the command. MATLAB will not quit. Instead it will merely end the execution of the command and give you control of the Command Window once again. And it will present you with a fresh command prompt, ready once more to do your bidding.

Issuing commands

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

>>
```

Our command told MATLAB to add 1 to 2 and then assign the result to a variable called **x**. MATLAB executes that command and then shows you the new

value of **x** followed by its prompt to let you know it is ready for another command. If you wish at any later time to learn the value of **x**, you simply type it, and MATLAB will respond as follows,

```
>> x

x =

     3
```

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

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

The act of assigning a value to **x** caused a new variable named **x** to be defined and caused a location in memory to be designated where its value is 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 or the variable is removed with the command **clear x** or the command **clear** (i.e., with no **x**), which clears all variables that have been defined up to that point in time in the Command Window. 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.

You may have noticed that MATLAB includes blank lines before and after the **x =** line above, and after the last line of the values in **x**. That can get annoying after a while. Fortunately MATLAB allows you to suppress the printing of those extra lines by using a command named "**format**" and telling it that you want a compact notation, in which there are no spaces between lines:

```
>> format compact
```

The mauve color of the word **compact** is interesting. We will see this color repeatedly. It means that the word "compact" does not stand for a variable, as **x** does above and that it is not a command, like **format**. It is just a string of characters that means something to the command. We'll find out more about this distinction in subsection Commands and Strings of the first section of Chapter 2.

Now let's ask MATLAB for the value of **x** again:

```
>> x
x =
     3
```

If you like blank lines, you can have them again by issuing the command **format loose**, which returns to the default spacing that MATLAB is set to when it is installed.

Suppressing printing

Unless you tell it otherwise, MATLAB will print the result of a command in the Command Window, as it did with the **x = 3** above. If you do not want the result printed, you can type a semicolon (**;**) after the command. With the semicolon, the command will still assign the value **3** to **x**, but it will not print the value of **x**. This feature is essential when **x** is a matrix of numbers. We will study matrices in great detail in the next section, which is entitled, Matrices and Operators), and we will use them throughout this book, but for now we will simply note that a matrix is a set of numbers arranged rectangularly that can be stored in a single variable and that there may a lot of them in that variable—possibly tens of thousands, or even more! Let's try using the semicolon:

```
>> XMen=5;
>>
```

We showed the following **>>** prompt to emphasize that the value of **XMen** is not printed after the semicolon. If we want to see its value, we can type its name and hit Enter:

```
>> XMen
XMen =
     5
```

Note that we have used uppercase letters in **XMen**. Uppercase not only looks different to us but is treated differently by MATLAB. **XMen** is a different variable from **xmen**, **xMen**, **Xmen**, etc. You may also have noticed that in the **XMen** example, there is no space before or after the equal sign, while in the **x = 1 + 2** 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. Variables in MATLAB can have more than one letter and, as we will see in the next example, can even include digits, but the letters and digits cannot have spaces between them. Thus, examples of legal names, are **SNOBOL** and **Wat4**, but **SNO BOL** and **Wat 4** are illegal because they include spaces. That should not be surprising. You might expect to see spaces around an equal sign (or not), but you would not expect to find spaces inside a name. Whenever possible, MATLAB, and all reasonable languages, follow the same rules that people use. We give more precise rules for MATLAB's variable names below in the subsection named (what else?) Variable names.

Using the Command Window as a calculator

Let's define some more variables:

```
>> y45 = 2*x
y45 =
     6
>> c = y45^2
c =
    36
>> rock = c/x
rock =
    12
```

The expression **2*x** means "two times **x**"; the expression **y45^2** means "**y45** raised to the power of two"; and **c/x** means "**c** divided by **x**". The variable named **rock** shows that a variable can have a name with more than one character. You can see from these simple examples that the MATLAB Command Window is a nifty (if somewhat expensive) calculator. This calculator can handle any number you are likely to need, large or small. Want a big number? Let's set a variable named "**earth**" equal to 5972000000000000000000000. That's a pretty big number that happens to be the mass of the earth in kilograms:

```
>> earth = 5972000000000000000000000
earth =
    5.972e+24
```

MATLAB chooses not to echo the number back to you with all those zeros. Instead, for numbers equal to one billion or larger, it uses **scientific notation**, which means that a number is written followed by "**e**" followed by a positive integer. That positive integer is the power of ten that must be multiplied by the number that is written before the **e**. In this case the number is 5.972×10^{24}. Scientific notation is handy for us too when we want to enter big numbers:

```
>> earth = 5.972e24
earth =
   5.9720e+24
```

Note that we don't have to include four digits to the right of the decimal point or put a plus sign in the exponent, as MATLAB does. Furthermore, MATLAB always picks a power of 10 so that the number to the left of the decimal is nonzero, but we can write numbers in scientific notation any way we want as long as we put an integer after the **e**:

```
>> earth = 0.5972e25
earth =
   5.9720e+24
>> earth = 0.0005972e28
earth =
   5.9720e+24
>> earth = 0.000597200000e28
earth =
   5.9720e+24
```

Let's use our calculator and scientific notation to calculate something as an illustration: the weight of a cell phone. According to Motorola's website, the mass of its Droid RAZR HD is 140 grams, which is 0.140 kilogram. That is not its weight. It's weight is the force with which the earth pulls on its mass. We could use a conversion formula to determine that force, but we are going to calculate it from basic principles of physics by using Newton's formula for universal gravitation. The formula is $w = G\frac{Mm}{r^2}$, where w is the force, G is Newton's universal gravity constant, M is the mass of the earth, m is the mass of the cellphone, and r is the distance of the cellphone from the center of the earth. MATLAB does not know this formula. Finding the formula is our responsibility. Nor does it know the values of the quantities in the formula. Here is the calculation as it appears in the Command Window beginning with setting the of G:

```
>> G = 6.6738e-11
G =
   6.6738e-11
```

The specification of the value of G utilizes scientific notation, this time to produce a very small number, which like very large numbers must otherwise be written with lots of zeros, 0.0000000000667384. Now we need to assign values to the rest of the variables in the expression on the right side of Newton's equation:

```
>> M = 5.972e24
M =
    5.972e+24
>> m = 0.140
m =
         0.14
>> r = 6378e3
r =
     6378000
```

The value we have chosen for **r** is the radius of the earth, because we are assuming that we want to know the cellphone's weight when we are holding it and standing on the ground. Finally, we use the expression to calculate the weight:

```
w_in_newtons = G*M*m/r^2

w_in_newtons =
       1.3717
```

The asterisks (*****) on the right side of the equal sign signify multiplication; the slash (**/**) signifies division, and the caret (**^**) signifies exponentiation. Thus, **G*M*m/r^2** has the same meaning in MATLAB that $G\frac{Mm}{r^2}$ has in algebra.

The variable name, **w_in_newtons**, shows that the underscore character (_) can be used in the name of a variable. It is often used, as we have used it here, to separate words within a variable name, in this case words that indicate the units of calculated force. MATLAB handles the numerical calculations for us, but it is up to us to keep track of the units in the results of those calculations. If we want to know the weight in units of pounds, we can perform a direct conversion. For that we need to know the conversion factor, which is 0.2248:

```
>> w_in_pounds = 0.2248*w_in_newtons
w_in_pounds =
       0.30835
```

Now, thanks to MATLAB (and Isaac Newton) we know that the Droid RAZR HD (and every other object whose mass is 140 grams that is near the surface of the earth) weighs 0.308 pounds. We round our answer to three digits because the mass is given to only three digits accuracy. As with the determination of the units of calculations, MATLAB leaves decisions about round off to us.

This simple example is meant to give a glimpse of the power of the MATLAB Command Window as a calculator. Many other examples can serve the same purpose, and all of them include the operations of assigning values to variables and then performing arithmetic operations on those variables to calculate new values, which are in turn assigned to other variables.

As we use more variables, we use more memory space. MATLAB, in fact, calls a collection of defined variables a **workspace**. If at any time, you wish to see the variables that you have brought into existence by assigning values to them, you can either look at the Workspace Window or use the **whos** command. For example, as a result of the commands we have entered in this subsection, **whos** would show us this:

```
>> whos
  Name              Size      Bytes  Class    Attributes

  G                 1x1           8  double
  M                 1x1           8  double
  XMen              1x1           8  double
  c                 1x1           8  double
  earth             1x1           8  double
  m                 1x1           8  double
  r                 1x1           8  double
  w_in_newtons      1x1           8  double
  w_in_pounds       1x1           8  double
  x                 1x1           8  double
  y45               1x1           8  double
```

The left column gives the variable names in alphabetical order (uppercase first). The columns labeled Size, Class, and Attributes (Attributes may be omitted in your version of MATLAB) will mean more to you later. The column labeled Bytes shows how much memory space MATLAB has allocated for each variable. Computer memory is measured in terms of bits and bytes. A **byte** is 8 bits, and a **bit** is the smallest unit of memory. A bit can store only one of two values: 0 or 1. The number of values that can be stored in N bits is equal to the number of ways that 0s an 1s can occur in N bits, which is 2^N. Therefore one byte can hold $2^8 = 256$ different values. As indicated in the Bytes column each these variables occupies eight bytes, which is 64 bits, so each of them can store $2^{64} = 1.8447 \times 10^{19}$ values. We will see what particular values can be stored in the section entitled, Data Types.

Saving variables

As seen from the output of the **whos** command, we have defined eleven variables. If we now exit from MATLAB, all these variables will be lost. Suppose we need to calculate additional weights, perhaps for a homework assignment, and we get a call from a friend asking us to lunch. If we shut down our computer, we will lose everything and will have to start from scratch after lunch. Alternatively, we might put our computer to sleep or in hibernation so that MATLAB will start up in the state we left it in with our variables intact. Sleeping and hibernating are not bad solutions, but MATLAB provides an even better way. Simply type the command **save**:

```
>> save

Saving to:
/Users/fitzpajm/Documents/Fitzpatrick&Ledeczi/matlab.mat

>>
```

This command causes all the variable names and their values to be saved in a file named **matlab.mat**, and it also tells us where that file is being saved: in the current folder. As can be seen in its response above, it gives the name of the current folder, **Fitzpatrick&Ledeczi**, before the name **matlab.mat** separated by a forward slash (**/**). It gives the name of the folder that contains the current folder before that, also separated by a forward slash, and so forth back to the top-level directory, which in this case is **Users**. Now that we

have saved our work, we can exit MATLAB and lose nothing, whether we shut down our computer or not, and head off to lunch. When we return and are ready to continue our work, we can restart MATLAB as we would normally. When it starts, our variables will be missing, but we can get them back easily with the command **load**:

```
>> load

Loading from: matlab.mat
```

which tells us the name of the file from which it retrieved our data. By the way, because of its file extension, **mat**, this file is called by the MATLAB community a **MAT-file**. We will learn more about MAT-files and other types of files for storing and retrieving data in Chapter 2 in the section entitled, File Input/Output.

Continuing a command to the next line

Sometimes a command is so long that it won't fit on one line. When that happens, you must tell MATLAB that the command is to be continued to the following line by means of the "line-continuation" operator (also known as an "ellipsis" operator), which is symbolized by three dots (periods, full stops):

```
>> dime = 2 + ...
8

dime =
    10
```

Putting more than one command on a line

You can indicate to MATLAB that a command is completed by typing 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,

```
>> weight = 4, acceleration = 9.8; velocity = 4.5;

weight =
     4

>> acceleration, velocity

acceleration =
    9.8000
velocity =
    4.5000
```

Repeating a command

Hitting the up-arrow key (↑) will cause the previous command line to appear. Hitting Enter (Return) 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. As an example, you might 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 right of the MATLAB desktop. You can also highlight, drag, and drop commands into the Command Window from anywhere and then execute them by hitting Enter.

Interpreting versus compiling

In the language of computer science, executing a command by a computing environment, such as MATLAB, is termed **interpreting** the command. MATLAB interprets (i.e., executes) the command typed in the command window as soon as you have completed the command and have hit the Enter key. This immediate response is different from the situation with a so-called "compiled" language, such as Java, C, C++, or FORTRAN, in which you write a

program from beginning to end and then run it all at once. In these languages, unlike MATLAB, before the commands that you type can be run, they must all be translated from the language that you are using (e.g., C++) into a language that the computer hardware uses. Such translating is called **compiling**. Note that the word "interpret", which roughly means "translate" outside the computer science community, in fact means "execute" when computer scientists use it in reference to a computer language. Outside the computing community, the word “compile" means gather and organize, but it means "translate" when computer scientists use it in reference to a computer language. MATLAB provides options that allow for compiling, but MATLAB is primarily used as an interactive language, which means a language in which users continually see the results of their commands as soon as they are issued. An interactive language is always an interpreted language.

Syntax and Semantics

The form of MATLAB's commands must obey certain rules. If they do not, then MATLAB cannot interpret them, and it gives an error message in bright red:

```
>> 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 us what is wrong. In this case, the user probably does not realize that the equal sign does not mean "is equal to". This is an **assignment statement**, which means instead, "Assign the value of the expression on the right side of the equal sign to the variable that is on the left side." Constants, such as 1, 2, or –18.9, cannot be assigned values. Thus, constants cannot appear on the left side of an equal sign.

This error is a good example of the violation of the proper form of a MATLAB statement. The form of a statement is its **syntax**. Any violation of the form is called a “syntax error”. In fact 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 another 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 definition 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 forgives us (we are, after all, its master) and 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 allow consecutively written letters to 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 must be indicated explicitly in MATLAB by typing an asterisk between the two variables to be 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. The asterisk is still used today in

most programming languages including both C++ and Java. Because programming languages use a symbol to indicate multiplication, they can allow an identifier to consist of more than one letter.

MATLAB provides wide latitude in variable naming. 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 command

```
>> namelengthmax
```

gives the maximum for the version that you are using. As mentioned above, 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`**

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`**:

```
>> X = [ 1 2; 3.4 pi]

X =
    1.0000    2.0000
    3.4000    3.1416

>> format long
>> X

X =
   1.000000000000000   2.000000000000000
   3.400000000000000   3.141592653589793

>>
```

As you can see, the numbers are now printed with more digits, which in MATLAB's terminology is a longer "format". There is another option with the format command. If you now decide that don't want the extra digits, you can issue the command **`format short`** to drop the digits. It won't effect the line spacing:

```
>> format short
>> X
X =
    1.0000    2.0000
    3.4000    3.1416
>>
```

There are other formatting options as well. 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.
    format with no inputs sets the output format to the default appropriate
    for the class of the variable. For float variables, the default is
    format SHORT.

    format does not affect how MATLAB computations are done. Computations
    on float variables, namely single or double, are done in appropriate
    floating point precision, no matter how those variables are displayed.
```

```
    Computations on integer variables are done natively in integer. Integer
    variables are always displayed to the appropriate number of digits for
    the class, for example, 3 digits to display the INT8 range -128:127.
    format SHORT and LONG do not affect the display of integer variables.

    format may be used to switch between different output display formats
    of all float variables as follows:
      format SHORT     Scaled fixed point format with 5 digits.
      format LONG      Scaled fixed point format with 15 digits for double
                       and 7 digits for single.
. . .
```

The **help** command always responds by repeating the command name (**format**, in this case) giving a short summary of its meaning (`Set output format`, in this case) and then giving a more detailed description. (We did not show all of the output from the help command 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. There is a fancier version too: the **doc** command. The command **doc c** gives the same information as **help c**, but it gives it in a much nicer format with better fonts.

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 xyz** searches through the command summaries (first part of the **help** output) given for all the commands, searching for the word **xyz**, and it shows you the associated commands. For example,

```
>> lookfor pseudoinverse
pinv                            - Pseudoinverse.
```

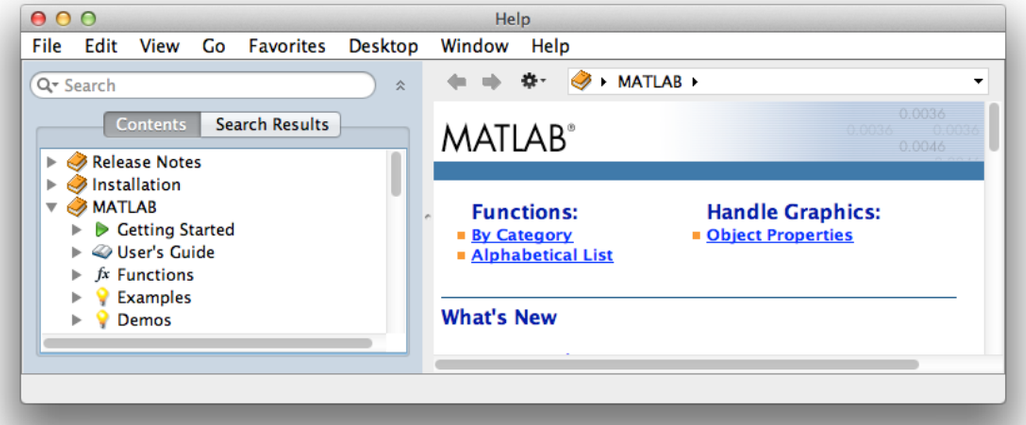
This response lists all the commands, in this case just **pinv**, for whose summaries include the word "pseudoinverse". To learn what **pinv** does, you could then type **help pinv**, or you can simply click on **pinv**. The blue lettering and underlining means that clicking on the term is the same as using **help**. 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**, **doc**, and **lookfor** in the Command Window, there is an even more elaborate help documentation available. In version R2012a and earlier versions, this system can be accessed as follows: click on the Help menu and select the Product Help command. Alternatively with these versions, you can click on the Start button at the bottom left of the MATLAB window. A menu will pop up. Select the Help command. The Help window will pop up with a "Help Navigator" on the left and helpful links in a panel on the right.

For version R2012b and later, you can click on the question mark in the small circle in the Home Ribbon, click MATLAB in the two-column list that appears, and then click MATLAB Functions at the bottom of the window that pops up. At this point you can click "By Category" or "Alphabetical List" (as shown in Figure 1.7 for version R2012a) (at the right of the window in R2012b). At this point in either system, you can click the link of any function that appears interesting. This help facility is very intuitive and easy to navigate.

There are animated demonstrations with voice-over in the help system. In version R2012a, select Demos in the Help Navigator. In version R2012b, after clicking the question mark in the circle and clicking MATLAB as above, click Examples.

Figure 1.7 *MATLAB Help Navigator*



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 write your commands into a file. That is done by using an Edit Window.

In each version of MATLAB there are two ways (and more) to pop-up an edit window. In Versions R2012 and earlier you can:

(1) Click the little white blank-page icon at the left end of the Toolbar at the top of the MATLAB window and then click File/New/Script

or

(2) Type **`edit`** in the Command Window and hit Enter.

Either of these two methods will cause the Edit Window to pop up, as shown in Figure 1.8.

In Versions R2012b and later, you can

(1) Click New Script at the left of the HOME ribbon and then click New and select Script

or

(2) Type **`edit`** in the Command Window and hit Enter.

Either of these two methods will cause the Edit Window to pop up as shown in Figure 1.9.

Figure 1.8 *The Edit Window in Version R2012a and earlier*

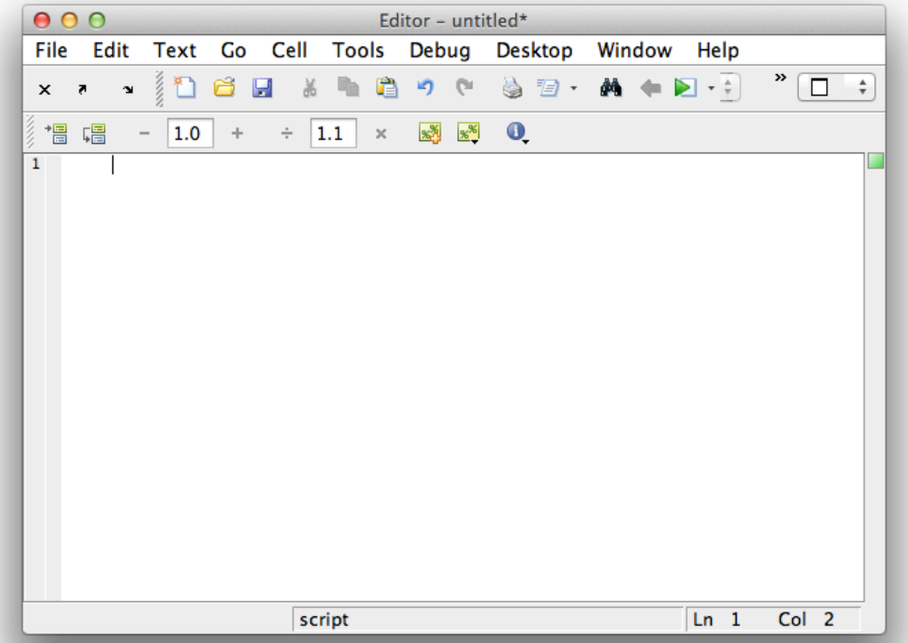
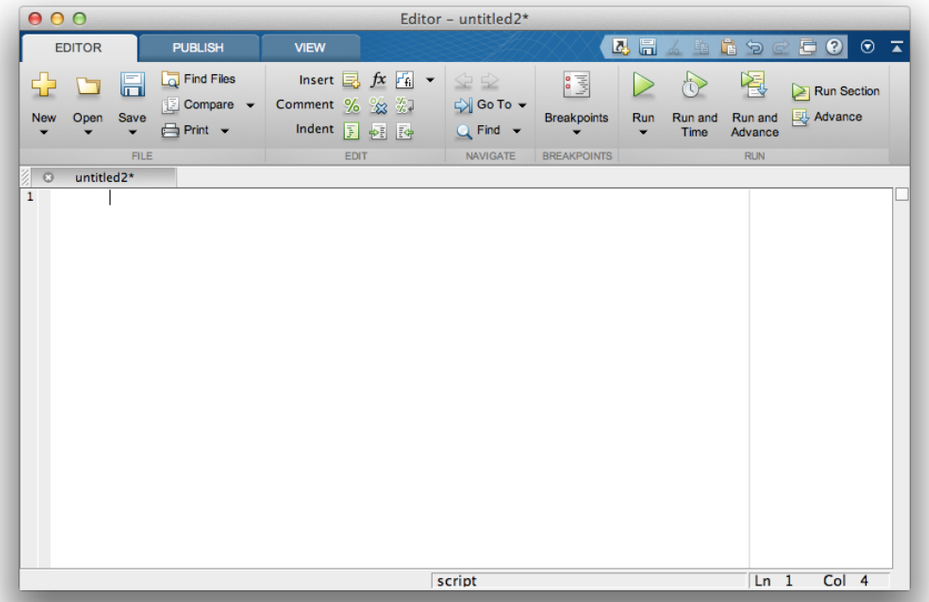


Figure 1.9 *The Edit Window in Versions R2012b and later*



Once you have the Edit Window open, you can type into it and text will appear, but unlike the situation in the Command Window, hitting Enter will cause nothing to happen except that the cursor will move to the next line. Nothing else happens because MATLAB does not interpret (execute) commands as you type them into the editor. Type the command

```
x = 5
```

inside this window. Now save what you have typed into a file. The first step to do that in versions R2012a and earlier is to click File/Save As... The first step in versions R2012b and later is to click Save. In both cases you will be presented with a familiar file-saving window for your operating system in which you can choose an appropriate folder to hold the file and an appropriate name for the file. In this case, you should choose the same folder that you chose earlier for your "current" folder, and you should use the file name, **`myfirst.m`**. When you click Save in this window, you will have created an **M-file**. The "M" stands for MATLAB, and the file extension used by the MATLAB editor is always **.m**, which is pronounced "dot m". It is because of that extension that the file is called an M-file. Because of the dot, another commonly used name for an M-file is a "Dot-m-file". Now go back to the Command Window and type **`myfirst`** (not **`myfirst.m`**). You should see this:

```
>> myfirst
x =
     5
>>
```

(assuming that you have previously given the command **`format compact`**, to avoid the blank lines). MATLAB has checked to see whether you have defined a variable named **`myfirst`** (you haven't), then it looked inside its current folder to see whether you have created a file named **`myfirst.m`**, (you have), and interpreted (i.e., executed) the commands it found inside that file.

Congratulations! You are a computer programmer, and you have proved it: You have written a (somewhat shortish) MATLAB program, stored it in a file, and run it. You ran it by simply typing the name of the file without the **`.m`** extension. Having a preferred filename extension for files that contain its programs is not particular to MATLAB. Conventions regarding filename extensions are common for all programming languages. Other languages, such as Fortran, C, C++, and Java, are less strict, but it is very common for programmers to use the extensions **.f**, **.c**, **.cpp**, and **.java**., respectively for these languages.

Within the Edit Window type a semicolon after **`x = 5`**. Save the file again, this time by clicking on the blue diskette icon. That icon will turn gray, indicating that the file has been saved since the last key you typed in that window, and it turns gray regardless of the method by which you save the file. Now return to the Command Window and type **`myfirst`** again. This time, all you see is this:

```
>> myfirst
```

As with commands issued in the Command Window, the output that was originally produced by the command **`x = 5`** is now suppressed because of the semicolon at the end of the line. The command still executed; **`x`** was still set (again) to 5. The only difference is that the value of **`x`** after the assignment is not printed in the Command Window. You can prove that by typing **`x`** in the Command Window:

```
>> x
x =
     5
```

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 in an M-file should be printed. In fact, typically *every* line in the typical M-file will include a semicolon to suppress printing.

Using the path to find an M-file

Inside the edit window, change the **5** to a **6**, hit enter and type **y = -9**. Then save the result to a new file called **mysecond.m** in the folder that you added to your path in the section above entitled, The Path. Make sure that the current folder is not this same folder, changing folders if necessary, as described before. Type the name **mysecond** into the Command Window. The command in the file will be executed. Even though your current folder does not contain this M-file, MATLAB has found it by looking through every folder on your path until it found it in the folder that you added to that path. It does that with the help of your operating system, and it happens so quickly that there is no noticeable delay. After it found it, it ran it. If you check the values of **x** and **y**, you will see that they are now equal to 6 and –9:

```
>> x
x =
     6

>> y
y =
    -9
```

There is one more name for the M-files that we have written: "scripts". They are called that because when an M-file like **myfirst.m** or **mysecond.m** is run, the MATLAB slavishly executes the commands in it, as if it were an actor following a script. We will learn much more about scripts

Source code versus executable code

Computer science provides some precise terminology for what we produce when we write a program. A **program** is any sequence of symbols that describes an algorithm. An **algorithm** is a step-by-step procedure for solving a problem. A **computer program** is a program that describes an algorithm in a language that can be executed on a computer. The text that you enter into an M-file by using the edit window is a sequence of symbols that describes an algorithm in a language that can be executed on a computer, and therefore, it is a computer 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 language that resembled a code because 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 **executable code** that is written in language that, unlike source code, can be executed directly by the computer..

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, and that is why it is called "source" code. 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 it. Companies who produce programs for sale are eager to distribute their executable code (also known as the "executables") but rarely their source code, thereby making more difficult for others to learn their proprietary programming secrets. Similarly, your instructors in a programming course might give you an executable version of a solution to a programming assignment so that you can run it and see first-hand how it is supposed to behave without your being able to see how it works.

When you write a program, such as the one in the file **myfirst.m**, the text that you type is called alternately the "program" or the "code", and writing such a program is called alternately **computer programming**, **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 actually runs on a computer.

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 or governing 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 special-purpose language, like MATLAB, that is tailored to their applications. Because of the design of such languages, 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 physical system instead of the logical aspects of the computer program. The software written by computer scientists and computer engineers to solve the problems of their disciplines tends, on the other hand, to be written in languages like C++, or Java, which 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., iBooks Author , 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 typically 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.

P-code

As we pointed out above under Interpreting versus compiling , the MATLAB system interprets (i.e., executes) 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 another language. This language is more efficiently executed, but it is not the language of the hardware, which varies with the type of computer, such as, for example, a Mac or a Windows machine,. Instead, it is a language that is interpreted by the MATLAB system and as a result is portable from one type of computer to another (as long as MATLAB is supported on it). The computer-science term for such a language is "portable code" or **p-code**. Java works in much the same way (it's p-code is also called "bytecode"). Fortran, C, and C++, and many other languages, on the other hand, are usually compiled, in which case there is no executing of p-code for those languages.

When an M-file is executed in the Command Window, the entire contents of the file is translated into p-code before the execution phase begins. 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 folder.

One important aspect of p-code is that it is version dependent. Thus, while an M-file written in one version of MATLAB will run under another version, a P-file written under one version may not run under another version. Another

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 folder, 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. A final aspect of p-code is that it is encrypted, meaning that during its translation MATLAB uses a secret code (i.e., the other kind of code – like spies use) so that others cannot read it. This allows the distribution of p-code in the same way that executable code is distributed, so that it can be used without the user seeing how it works.

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!). 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 set of multiple comment lines 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: Nicholas S. Zeppos
% Section 1, CS 103
% September 22, 2013
% HW Assignment 1
```

Text such as this, which is included in the program but is ignored by the system that is interpreting it, such as MATLAB, or compiling it, such as a C++ compiler, 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:

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

MATLAB provides a way to comment consecutive lines of comments without putting a percent sign on every line. Here is the same block of comments as above using the block-comment option:

```
%{
Author: Nicholas S. Zeppos
Section 1, CS 103
January 22, 2012
HW Assignment 1
%}
```

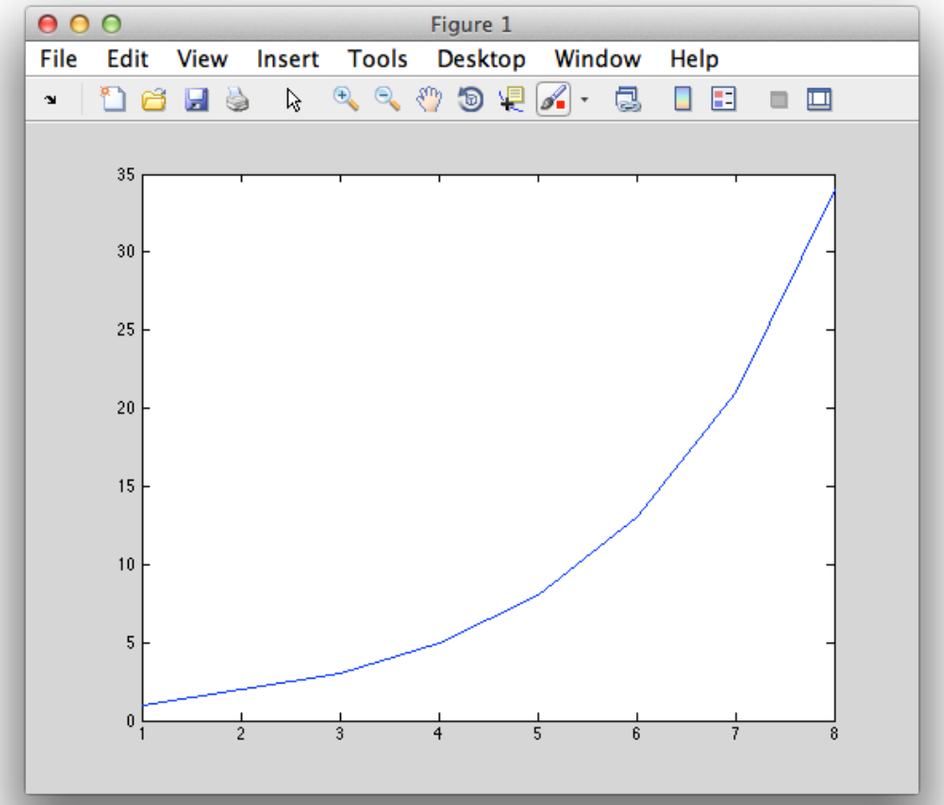
Both `%{` and `%}` must appear alone on their own lines. Every line between those two lines will be highlighted in green as a comment and will otherwise be ignored by MATLAB. Every modern language provides some means for including comments, and there is always some character or characters used to indicate those parts of the program that are comments. C, C++, and Java, for example, all use **`/*`** to begin a block comment and **`*/`** to end it. C++ and Java also use **`//`** for a single-line comment. 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.

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];
```

Figure 1.10 *A MATLAB figure window showing a plot*



A **vector** in mathematics and in computer science is simply an ordered list of numbers, and each number is called an **element** of the vector. As we will see in the next section, 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:

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

A plot appears in a "Figure" window, which pops up automatically when you give the **`plot`** command, as shown in Figure 1.10.

If you want to close the figure window, either click with the mouse on the red button at the top left corner of the window on a Mac or on the × at the top right of the window on Windows or type the command **`close`** in the Command Window. If you want to keep your figure window open 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
```

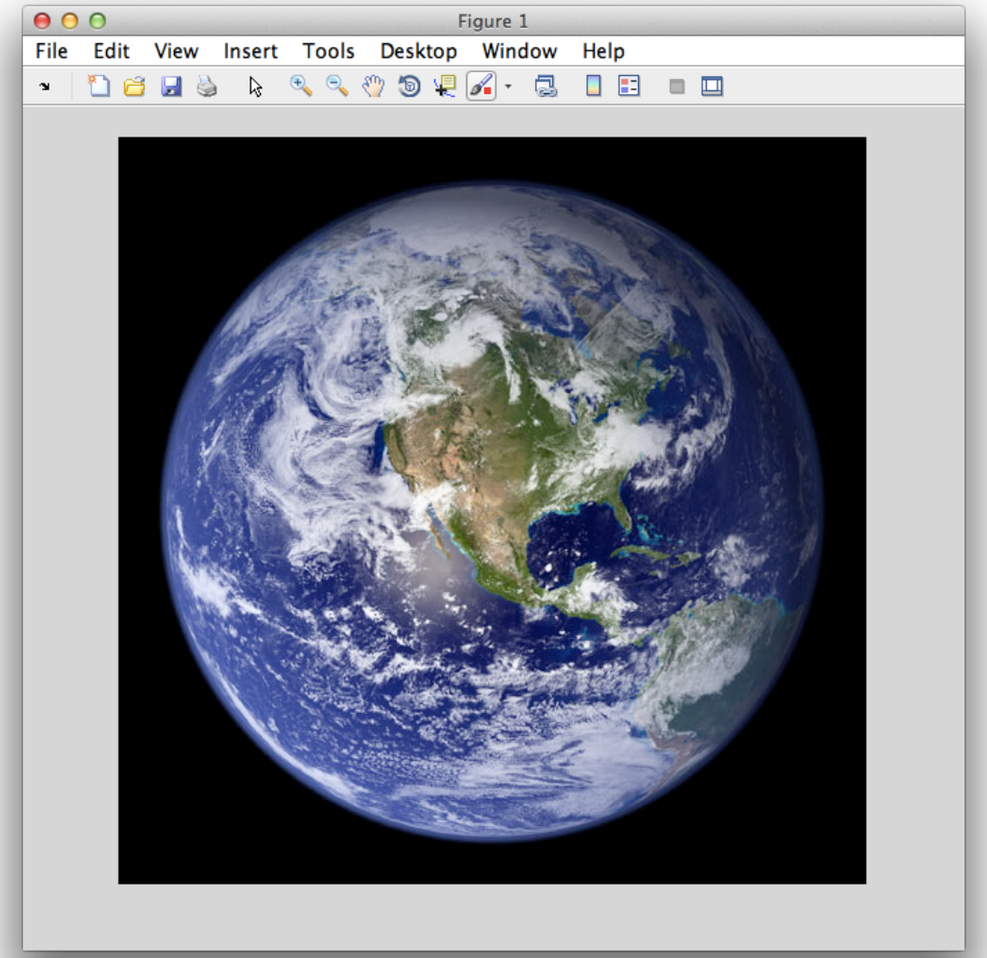
It should be clear that **`plot(x,y)`** plots the vector **`y`** vertically versus the vector **`x`** horizontally. These vectors must be of the same length. For each value of the **index**, **`n`**, from **`n = 1`** to **`n = 8`**, **`x(n)`** is equal to the horizontal position of the point that is plotted, and **`y(n)`** is equal to the vertical position. We note that we have introduced a new term, "index", which in MATLAB terminology means a positive integer that enumerates the elements of a vector.

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 **`help plot`**. We will give more examples in the Programmer's Toolbox section.

Figure windows are used whenever graphical output is required. There are many commands in addition to **`plot`** that produce graphical output, but all of them put their graphical output into a figure window. As an example, let's display a picture. The US National Aeronautics and Space Administration (NASA) is a source of many beautiful images, not the least of which are those

Figure 1.11 *Displaying an image file*



taken of the earth from space. The images are available from the Internet at www.nasa.gov and are stored digitally in standard formats. A common format on that website and many other websites is the so-called JPEG format (Joint Photographic Experts Group), and files containing images in that format are typically named with the extension **jpg**. If we download the file named **`globe_west_540.jpg`**, which contains a satellite image of the western hemisphere of the earth, into the current folder or into a folder that is on the MATLAB path and issue the following commands, the picture shown in Figure 1.11 will pop up.

```
>> west_earth = imread('globe_west_540.jpg');
>> image(west_earth);
>> truesize;
>> axis off;
```

The first command causes an array of numbers to be read from the file and stored in the variable **`west_earth`**. An array is a three-dimensional version of matrix, and we will study them in detail in the next section. This particular one is a 540x540x3 array of numbers. The second command, **`image(west_earth)`**, causes the array in **`west_earth`** to be displayed.

The display screen of a computer is divided up into hundreds of thousands of tiny pixels. The word **pixel** means "picture element", and a pixel is the smallest region of an image that can be controlled individually. Each pixel is one square piece of a digital image, like a tile in a mosaic, and each piece is capable of displaying only one color at a time. Each color is determined by three numbers, and there are 540x540 = 291,600 sets of three numbers in the array **`west_earth`**, each of which determines the color of one pixel.

The last two commands make the picture look its best. The command **`truesize`** causes each color in the array to be displayed by exactly one pixel, so that the shape of the earth is correct (i.e., circular). Finally, the coordinate tick marks and labels that can be seen in Figure 1.10, at the edges of the plot are removed by the command **`axis off`**.

To explore the possibilities of image display further, you can use the command **`help imread`**. We will learn much more about images, colors, and pixels and will learn how images can be altered by means of image processing when we get to the Loops section of Chapter 2.

Additional Online Resources

- Introducing MATLAB, a video by MathWorks
- Getting Started with MATLAB, a video tutorial by MathWorks
- Introducing MATLAB Mobile, a video tutorial by MathWorks
- Video lectures by the authors:

  Lesson 1.1 Introduction (11:43)

  Lesson 1.2 The MATLAB Environment (20:41)

  Lesson 1.3 MATLAB as a Calculator (14:25)

  Lesson 1.4 Syntax and Semantics (5:01)

  Lesson 1.5 Help (8:37)

  Lesson 1.6 Plotting (19:06)

Concepts From This Section

Computer Science and Mathematics:

search path
prompt
variable
scientific notation
interpreting versus compiling
interactive language
identifier
syntax
semantics
assignment statement
program, programmer
computer program, code
algorithm
source code
executable code
bugs
portable code (p-code)
comments
vector, vector element

MATLAB:

current folder and path
Command Window
prompt
variable
suppressing printing with semicolon
scientific notation
identifier
workspace
the **`whos`** command
Mat-files

the commands **`save`** and **`load`**
one command on multiple lines
multiple commands on one line
assignment statement
the commands **`help`** and **`doc`**
the **`lookfor`** command
the **`format`** command
Edit Window
M-file
P-file
comments
figure window
vector and vector element
`plot`
image display

Matrices and Operators

Objectives

The basic unit with which we work in MATLAB is the matrix. We solve problems by manipulating matrices, and operators are the primary means by which we manipulate them.

(1) We will learn how to define matrices, extract parts of them and combine them to form new matrices.

(2) We will learn how to use operators to add, subtract, multiply, and divide matrices, and we will learn that there are several different types of multiplication and division.

(3) Finally, we will learn MATLAB's rules for determining the order in which operators are carried out when more than one of them appear in the same expression.

Image by Tamas Fodor

The matrix is the basic unit of MATLAB.

The primary purpose of this book is to teach computer programming with MATLAB. MATLAB is a good tool for learning about computer programming because it includes all the major computer-programming constructs used by all major languages, including C, C++, and Java, and MATLAB is easier to learn than those languages. Moreover, MATLAB includes additional features that make it especially useful for numerical applications. The most notable such features are the ones that involve special operations on arrays of numbers. The most common array of numbers is called a "matrix", and the basic unit with which we work in MATLAB is the matrix. In comparison to general-purpose programming languages, MATLAB makes it far easier to add one matrix to another, to subtract them, and to multiply them, and it allows us to perform many other operations on them as

well. This focus on matrices is apparent from the name, "MATLAB", which is an abbreviation of the phrase "Matrix Laboratory". Its matrix operations are to a large extent what makes MATLAB such a good language for programming solutions to problems in engineering and science, each of which often involves matrices, and they are also a distinguishing feature that makes MATLAB—well—MATLAB. In this section, we will introduce the basic idea of the matrix, and we will show how MATLAB allows you to define matrices and to operate on them.

Matrices And Arrays

A **matrix** is a two-dimensional, rectangular arrangement of numbers, such as this 2-by-3, matrix, which has 2 rows and 3 columns:

$$\begin{bmatrix} 1 & 2 & 3 \\ 3.3 & \pi & -4 \end{bmatrix}$$

We should note that, while we have called this a "2-by-3" matrix above, it is also common to see "2-by-3" written this way: "2x3". A matrix whose number of rows equals its number of columns, as for example, 7-by-7, is called a **square matrix**. A matrix is useful for dealing with sets of numbers and also with sets of equations involving sets of variables, a situation that arises repeatedly in all branches of science and engineering. A matrix is a special case of an **array**, which can have more than two dimensions. A **scalar**, which is a single number in mathematics, is treated in MATLAB, surprisingly perhaps, as a 1-by-1 matrix or array! To see the size of a matrix or array in MATLAB, we can use the function called **`size`**:

```
>> x = 5
x =
     5
>> size(x)
ans =
     1     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 (e.g., C, C++, Java, Fortran), a function, like **`size`** or **`plot`**, which was introduced in the previous section, is any operation that is invoked by giving its name. The major distinction between mathematical and programming definitions of "function" is that a mathematical function will always produce the same output for a given input, whereas a programming function may not (e.g., **`rand`**, which we will meet in the next chapter). Our input to **`size`** was **`x`**, but in general the input to a function is given as a list of values separated by commas inside a pair of parentheses that follows the name, as in **`plot(x,y)`**. Each such value is called an **argument**. This term is used this way in both mathematics and computer science. However, as we will see later in this section, computer science expands the definition to include output from functions as well. In the example above, **`size`** was given only one argument, **`x`**, (so no commas were 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 (**`;`**). To see how, let's create with MATLAB the matrix that we gave at the beginning of this section, assign it to the variable **`X`**, and then call **`size`** using **`X`** as an argument, as follows:

```
>> X = [1 2 3; 3.4 pi -4]
X =
    1.0000    2.0000    3.0000
    3.4000    3.1416   -4.0000
```

The square brackets (**`[ ]`**) indicate that we are asking MATLAB to form a matrix, and they mark its beginning and its end. Individual elements are separated by spaces (one or more). A semicolon marks the end of a row (instead of causing printing to be suppressed as it does when it comes at the end of a command). So, in the example above, **`1 2 3;`** means that the first three elements are **`1`**, **`2`**, and **`3`**, and that is the end of the row. The next element after the semicolon, which is **`3.4`**, starts the next row. The second element on the

second row may be a bit of a surprise: **pi** is actually a function call that returns π, showing us that, if no argument is being input, the parentheses can be omitted. To be specific, since π is an irrational number, **pi** gives an approximation to π. It's a pretty good approximation: 3.141592653589793. MATLAB displays only the first four digits to the right of the decimal because **format short** is in effect (**format** was introduced in the previous section), but all digits are used in calculations. Since a matrix is by definition rectangular, there must be the same number of elements on the second row as on the first row. Thus it must have one more: The third, and final, element on the second row is **-4**. The **size** function tells us what we already know in this case:

```
>> size(X)
ans =
     2     3
```

namely, that there are **2** rows and **3** columns. Here is a larger example:

```
>> Y = [1 2 3 6 4 1 12; 3.4 -8 3 3 0 pi .2]
Y =
  Columns 1 through 5

  1.0000  2.0000  3.0000  6.0000  4.0000
  3.4000 -8.0000  3.0000  3.0000       0

  Columns 6 through 7

  1.0000 12.0000
  3.1416  0.2000
```

Note that, since there was not enough room for an entire row of the matrix to fit on one line of the Command Window, the 6th and 7th columns were grouped after columns 1 through 5, and the range of column numbers is given for each group. The default in MATLAB is that a matrix is printed in **column-major order**, meaning that all the elements of one column are processed, in this case printed, before the elements of the next column. The other order is called, imaginatively enough, **row-major order**. Column-major order is the default throughout MATLAB; for other languages, such as C++, the default is row-major order. If the end of the Command Window is reached before all the columns have been printed, a new group of columns is begun below the current group.

There are two alternate ways of doing things when entering the elements of a matrix. First, an optional comma can be typed after any element:

```
>> Z = [1 2,3, 4; 5, 6 7, 8,]
Z =
     1     2     3     4
     5     6     7     8
```

and second, hitting the Enter key (i.e., instead of typing a semicolon) also indicates the end of the row:

```
>>  Z = [1 2, 3, 4
5 6 7 8]

Z =
     1     2     3     4
     5     6     7     8
```

Enter key was hit after the 4

Here, we hit the Enter key after the **4**, which moved the cursor to the next line and ended the first row.

A vector in MATLAB is simply a matrix with exactly one column or exactly one row. These two types of vectors are called, respectively, a **column vector** and a **row vector**. The command

```
>> x = [1 4 7]
x =
     1     4     7
```

produces a row vector, and the command

```
>> y = [1; 4; 7]
y =
     1
     4
     7
```

produces a column vector, as can be seen by the vertical display of the elements.

When its argument is a vector, the **size** function always gives one dimension equal to 1:

```
>> size(x)
ans =
     1     3

>> size(y)
ans =
     3     1
```

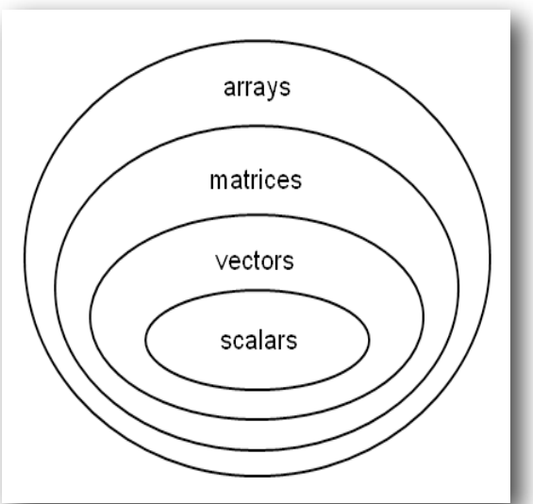
A leading **1** indicates a row vector; a trailing **1** indicates a column vector.

As mentioned at the beginning of this section, matrices are also called arrays. For most programming languages, the term "array" is used exclusively, but for MATLAB the choice of term depends on the sort of operations that are performed on them—matrix operations or array operations. Array operations and matrix operations are listed in the subsection entitled, Arithmetic with matrices and vectors. The appropriate operations—array or matrix—are determined by the meaning of the set of numbers being operated on. For example, when the numbers represent values in the cells of a spreadsheet, array operations are appropriate, and so we call the set of numbers an "array". When the numbers represent the colors of pixels in an image, array operations are again appropriate, and the term "array" is again used. The term "matrix" is strictly appropriate only when the numbers represent the coefficients in a set of linear equations, but both "matrix" and "array" are often used interchangeably.

It is possible to have three-dimensional arrays (not properly called "matrices") in MATLAB. Such arrays are said to have rows, columns, and "pages". So, for example, the element **A(2,3,4)** is on the second row of the third column of the fourth page of the array **A**. Even higher dimensions are available as well. In fact, there is no limit on the number of dimensions, but few programs employ more than three.

Figure 1.12 shows that the set of arrays includes all matrices, the set of matrices includes all vectors, and the set of vectors includes all scalars. In some applications, three dimensions are used 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 relatively large positive numbers for bone, lower positive numbers for soft tissue, approximately zero for water, and a negative value for air. (typically -1024). Magnetic resonance (MR) imaging similarly produces three-dimensional arrays of intensities, and in some cases MR values are complex numbers. A time sequence of CT or MR images may be combined into a four-dimensional array. Medical image researchers throughout the world employ MATLAB programs to manipulate CT and MR images, but many of their programs deal with two-dimensional slices of three-dimensional vol-

Figure 1.12 *Arrays, matrices, vectors and scalars*



umes. One reason for this simplification is that humans can visualize two-dimensional arrangements much more readily than higher dimensions. MATLAB handles all dimensions, but it is focused on two-dimensional arrays and provides special operations for handling them. Fortunately, most engineering and science applications do not require arrays of three or more dimensions. We will, therefore, focus primarily (but not exclusively) on the two-dimensional ones, and, when we deal with a two-dimensional array, we will typically call it a matrix.

Complex Numbers

MATLAB is fully capable of dealing with complex numbers. A scalar can be complex, a vector can have complex elements, and matrices and arrays can have them as well. Having complex elements does not affect the dimension of an array. A 3-by-4-by-5 array has the same shape whether some, all, or none of the numbers are complex. A **complex number** is a number that includes, $\sqrt{-1}$, 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`:

```
>> z = 3 + 4j
z =
   3.0000 + 4.0000i

>> z*z
ans =
  -7.0000 +24.0000i
```

Note that MATLAB responds with `i`, instead of `j`, to indicate the imaginary part of the complex number.

MATLAB also provides two functions, `i` and `j`, whose outputs are $\sqrt{-1}$. Like the function `pi`, which we met in the previous section, they require no input.

Here are examples of how these functions can be used to produce complex numbers:

```
>> a = i
a =
        0 + 1.0000i

>> b = 3 - pi*j
b =
   3.0000 - 3.1416i

>> c = 16*j
c =
        0 +16.0000i
```

We can, however, override these functions by assigning values, real or complex, to variables named `i` and/or `j`:

```
>> i = 23
i =
    23

>> d = 3 + 4*i
d =
    95

>> j = 10i
j =
        0 +10.0000i

>> e = 3 + j
e =
   3.0000 +10.0000i
```

Because of the confusion that can be caused by assigning values other than the square root of –1 to `i` and/or `j`, most MATLAB programmers completely avoid assigning values to these variables. However, *i* and *j* are very commonly used as real integers in mathematics, so, when variables are needed to hold integers in MATLAB, most programmers substitute the variables `ii` and `jj` in place of `i` and `j`. In keeping with this double-letter style, it is common also to see `kk`, `ll`, `mm`, and `nn` used to hold integers as well.

The "Colon Operator"

The elements of the vector `x = [1 4 7]` are regularly spaced: they increase regularly by 3. MATLAB provides a convenient way to produce this vector: `x = 1:3:7`, which means, "Assign `x` to be the vector whose elements begin with 1, increase by 3, and go no higher than 7." This expression is an example of the use of a special MATLAB operator. An **operator** is a function that is invoked by a symbol, the most familiar examples of operators being `+`, `-`, `*`, and `/`. Operators are similar to ordinary functions except that they do not surround their input arguments with parentheses, and their symbols typically sit *between* their arguments. There is also a special name for their input arguments: An input argument to an operator is called an **operand**. The action of an operator on its operands is called an **operation** (not to be confused with surgery). We have just introduced a new operator, `:`, which is called the **colon operator**. A colon operator specifies a regularly spaced list of numbers. Most people find the regular spacing to be intuitive but will at first misunderstand the rule at the upper end of the list. Let's explore this rule by comparing the outputs of some examples:

```
>> x = 1:3:7
x =
     1     4     7

>> x = 1:3:8
x =
     1     4     7
```

To see why these two different expressions produce exactly the same sequence, note that the second one could not require that the sequence end at `8`. That would be impossible for a sequence that begins with `1` and increases by `3`, because, after hitting `7`, it would hop over `8` and land at 10. To resolve such a mismatch between the early part of the sequence and the upper end, MATLAB uses the **Price is Right**® rule: The sequence stops at the number that comes the closest to `8` without going over. The rule might be made clearer with a few more examples. First, let's increase the limiting number to 9:

```
>> x = 1:3:9
x =
     1     4     7
```

Upping the limit to `9` has had no effect here. Ending the sequence at `9` would have the same problem as ending it with 8. The sequence would hop over `9`, just as it would hop over 8. Now let's increase it to 9.9:

```
>> 1:3:9.9
ans =
     1     4     7
```

There is no problem with using a fractional number like `9.9` as the limiting value, but `9.9` produces the same result as 8 and 9, because the sequence would hop over `9.9`, just as it would hop over 8 or 9. Now let's increase the upper limit to 10:

```
>> x = 1:3:10
x =
     1     4     7    10
```

At last! When we raise the limiting value to `10`, we finally get an additional number in the sequence, because we can add `3` to `7` without going over `10`.

The colon operator can be used anywhere that a row vector of equally spaced numbers is needed (we'll see how to change a row vector into a column vector later in the subsection entitled The Transposition Operator). It is especially useful for very long lists, for which an explicit enumeration would require way too much typing, such as `0:2:9999`, which produces five thousand even numbers or `1:2:9999`, which produces five thousand odd numbers.

The most common spacing used with the colon operator is 1, as for example:

```
>> x = 1:1:7
x =
     1     2     3     4     5     6     7
```

For this spacing, there is an abbreviated version of the colon operator available, in which the `:1:` in the middle is abbreviated simply as, `:`. Here is an example:

```
>> x = 1:7
x =
     1     2     3     4     5     6     7
```

Any of the operands of the colon operator can be fractional and/or negative. Here is an example: `.354:.067:4`. This expression produces a list of 65 numbers starting with 0.354 and ending with 3.972. The only restriction on the numbers used as operands with the colon operator is that they not be complex.

When the middle operand is negative, the numbers decrease, for example:

```
>> x = 7:-3:1
x =
     7     4     1
```

This expression means, "Assign **x** to be the vector whose elements begin with **7**, decrease by **3**, and go no lower than **1**." Thus, when the middle number is negative, MATLAB uses the opposite of the Price is Right rule to stop the sequence: The sequence stops at the number that comes the closest to 1 without going under.

Many people are surprised to see what happens when they use expressions like the following one, in an effort to produce a decreasing sequence:

```
>> x = 7:3:1
x =
   Empty matrix: 1-by-0
```

Empty matrix? 1-by-0? What is all this? You might have expected the result to be **x = [7 4 1]**, as it was for the previous example. This example highlights a very common programming error made even by experienced programmers (such as the authors of this book), when a decreasing sequence is desired.

Let's look more closely at what we have asked MATLAB to do for us here: We have asked it to form a row vector that includes all the numbers that start with 7, increase by 3 and are no larger than 1. Well, there are no such numbers! Even 7 is higher than 1. So this is a non-starter. Instead of telling us that we have made an error, that we have asked for the impossible, or that we are chasing rainbows, MATLAB simply sets **x** to an empty matrix. An **empty matrix** is a matrix with no elements. Note that this is not a matrix that contains a zero. This is a matrix that contains nothing. Nothing at all. Not a zero, not a one, not anything. It is empty. It could have been called an "empty array" too, instead of an empty matrix. Either name would work, but this is MATLAB, not ARRLAB, so an empty matrix it is.

What MATLAB has done is follow our instructions to the letter. By using the colon operator, we asked for a row vector, and by giving 7 as the starting element and 1 as the upper limit, we gave constraints that are satisfied by no elements. So MATLAB gave us a row vector with no elements. The **size** function proves that:

```
>> size(x)
ans =
     1     0
```

The answer that **size** gives is that the number of elements in each column is one and the number of elements in each row is zero. This is the **size** function's way of saying there is one row, which contains zero elements. There are other empty matrices as well. The one that seems the emptiest is the one with no rows and no columns. Here is how to get it:

```
>> x = []
x =
     []
```

and here is what **size** says about its dimensions:

```
>> size(x)
ans =
     0     0
```

We will see uses for empty matrices later. For now, they are useless objects that pop up when you have made a mistake. Here are other mistakes that produce empty matrices:

```
>> x = 1:-3:7
x =
   Empty matrix: 1-by-0
```

There are no numbers that start with 1 decrease by 3 and go no lower than 7.

```
>> x = 1:0:7
x =
   Empty matrix: 1-by-0
```

One might argue that a sequence consisting of elements that start at one, increase by zero, and go no higher than 7 would be an infinite set of ones, but MATLAB chooses to define any sequence that increases (or decreases) by zero as empty.

Accessing Parts Of A Matrix

An element of a matrix can be accessed by giving its row index to specify its **row**, which is the position in the first dimension of the matrix, and its column index to specify the **column**, which is the position in the second dimension. These are two positive integers in parentheses separated by commas. An integer used in this way is also commonly called a **subscript**. Again, using the example matrix, **X = [1 2 3; 3.4 pi -4]** that we used above, we have

```
>> X(2,3)
ans =
    -4
```

showing that the element of **X** on the 2nd row and 3rd column is –4. We can also use this access method to assign a value to one element of a matrix:

```
>> X(2,3) = 7;
```

If we now check the value of **X**, we see that third element on the second row has been changed:

```
>> X
X =
    1.0000    2.0000    3.0000
    3.4000    3.1416    7.0000
```

This notation is closely related to standard mathematical notation, in which the two indices of a matrix are typeset 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). If **x** is a row vector, then **x(1,i)** is equivalent to **x(i)**, which in mathematical notation would be x_i . If **x** is a column vector than **x(i,1)** is equivalent to **x(i)**. (Remember to use **ii** and **jj** in MATLAB to avoid confusion with imaginary numbers. Here we only used **i** and **j** because mathematical notation uses single letters only.)

An interesting question is this: What happens, if a value is assigned to an element of a matrix when the matrix does not yet exist? Let's try it. First, let's establish that the matrix **Dumbledore** does not (yet) exist:

```
>> Dumbledore
Undefined function or variable 'Dumbledore'.
```

This is MATLAB's quaint way of telling us that **Dumbledore** does not exist. It can't know whether we are trying to call a function named **Dumbledore** or trying to read the value of a matrix named **Dumbledore**, but it has checked

its inventory of currently defined functions and variables, and there is nothing there with this name. Now that we know that there is no such thing as **Dumbledore**, let's try to assign a value to an element of it:

```
>> Dumbledore(2,2) = 1881
Dumbledore =
           0           0
           0        1881
```

Like magic, **Dumbledore** is there before us! How did he do that, and why did MATLAB not give us an error message again? Giving an error is exactly what the languages C, C++, or Java, would do in this case, but MATLAB takes a different approach. It instantly defines a matrix named **Dumbledore** for us with the minimum number of rows (two) and the minimum number of columns (which happens also to be two in this case) required to provide a place for element **(2,2)**, and it fills it with zeros. Then it does what we asked it to do: It assigns the value **1881** to element **(2,2)**.

Why did the engineers at the MathWorks decide to adopt this approach, i.e., creating **Dumbledore** instead of chastening us with a red error message for not creating it before we wrote something into it? They did this because, while C, C++, and Java have special statements for defining new variables, the only way to bring a variable into existence in MATLAB is to assign a value to it, and that is exactly what we asked MATLAB to do. Since we chose to assign a value to an element other than (1,1), and since matrices must be rectangular, MATLAB had to fill in some values to earlier elements in the matrix to avoid leaving "holes" in it. Zero seems to be a sensible value to use, and MathWorks engineers are exceptionally sensible.

OK. So we have a 2-by-2 **Dumbledore**. What happens if we now tell MATLAB to assign a value to an element of the existing **Dumbledore** outside the range of its rows and/or columns?

```
>> Dumbledore(3,4) = 1998
Dumbledore =
       0           0           0           0
       0        1881           0           0
       0           0           0        1998
```

No problem. MATLAB immediately enlarges the matrix and fills in some more zeros, but it leaves the existing elements alone. This is exactly what we want it to do, because, when we are assigning elements to a matrix one by one, we do not want a fixed order in which we need to do it. We would not want to lose the values already in there when we later caused the matrix to grow by putting new values into it. Of course, if we decide to put, say, **pi**, into an element that already exists, say, element (1,1), we can do that also without affecting the rest of the matrix:

```
>> Dumbledore(1,1) = pi
Dumbledore =
   3.1416           0           0           0
        0        1881           0           0
        0           0           0        1998
```

WARNING! There is a pitfall here. Since, as we learned in the beginning of this section, a scalar is treated in MATLAB as if it were a 1-by-1 matrix, we might be tempted to think that the following command is equivalent to the previous command:

```
>> Dumbledore = pi
Dumbledore =
       3.1416
```

It is not! We have just overwritten the matrix with a single scalar. The **Dumbledore** we had grown to love has been replaced by a new **Dumbledore**. The old one is dead. We cannot get it back. The Command Window has no undo for commands once they are entered.

Accessing multiple elements

More than one element can be specified at a time by using the subarray operations. A **subarray operation** accesses rectangular parts of an array. It is invoked by specifying a vector of integers for each index. The result is that elements from multiple rows and/or multiple columns can be read or written. Here are three examples of reading elements with subarray operations based on the same `X` that we have used before:

```
>> X = [1 2 3; 3.4 pi -4]
X =
    1.0000    2.0000    3.0000
    3.4000    3.1416   -4.0000
```

Example 1. Multiple columns

```
>> X(2,[1 3])
ans =
    3.4000   -4.0000
```

The `[1 3]` is the new part of the command. It means that we want to see columns 1 and and 3. Since we specified a 2 for the first index, we get elements only from row 2. By the way, the comma is required here. Without it, MATLAB will not process a subarray command:

```
>> X(2 [1 3])
 X(2 [1 3])
     |
Error: Unbalanced or unexpected parenthesis or bracket.
```

Example 2. Multiple rows can be specified too:

```
>> X([2,1], 2)
ans =
    3.1416
    2.0000
```

And, as can be seen from this example, the rows do not have to be in order. Here we requested the order to be row 2, followed by row 1.The same holds for columns.

Example 3. Both multiple rows and multiple columns can be specified:

```
>> Y = X([2,1,2],[3,1,1,2])
Y =
   -4.0000    3.4000    3.4000    3.1416
    3.0000    1.0000    1.0000    2.0000
   -4.0000    3.4000    3.4000    3.1416
```

And, as this example shows, rows and/or columns can be repeated.

Subarrays are more often specified by means of the colon operator. Here are some examples:

```
>> 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
```

This last example introduces a handy new feature. When the colon is given alone in the second position, as in `X(2,:)` above, it means "all columns." When it is in the first position, e.g., `X(:,2)`, it means "all rows":

```
>> X(:,2)
ans =
    2.0000
    3.1416
```

A convenient notation is provided for specifying the last row or column in a subarray operation. Instead of giving the number of the last row or column, we use a **keyword.** A keyword is a word that is defined by the language to have special meaning, and in this case the keyword is **end**:

```
>> X(1,2:end)
ans =
     2     3
```

It works for direct element specification too:

```
>> X(end,1)
ans =
    3.4000
```

Arithmetic can be used with **end** , to specify a position relative to the end as in this example:

```
>> X(1,end-1)
ans =
     2
```

Since there are 3 columns in **X**, the expression **end-1** represents 3 – 1, which equals 2, so the command above is equivalent to **X(1,2)**. This trick is more commonly used in specifying a subarray:

```
>> X(:,end-1)
ans =
            2
       3.1416

>> X(:,1:end-1)
ans =
            1            2
          3.4       3.1416
```

Fancier expressions are fine as long as (a) the result is an integer and (b) the integer is within the range of indices of the array:

```
>> X(1,(end+1)/2)
ans =
     2
```

In the command above **(end + 1)/2 =** $(3 + 1)/2 = 4/2 = 2$.

The colon operator can be used on the left side of the equal sign as well, allowing us to assign values to rectangular parts of the matrix. For example,

```
>> X(1:end,1) = -4.444
X =
   -4.4440    2.0000     3.0000
   -4.4440    3.1416    -4.0000
```

Note that, although we referenced only the first column of **X** in the statement, the entire matrix is printed (unless we use the semicolon to suppress printing, in which case nothing is printed).

Note also the single scalar value, **-4.444**, given on the right of the equal sign was copied into each of the two elements specified by the colon operator. There is no limit on the number of replications of a single element that MATLAB will use to fill a subarray:

```
>> X(1:end,2:3) = 9
X =
   -4.4440    9.0000    9.0000
   -4.4440    9.0000    9.0000
```

If, instead of giving a single scalar value, we specify an array on the right that matches the shape of the subarray on the left, the array on the right will replace the subarray on the left:

```
>> X(1:2,2:3) = [10 12; 100 120]
X =
   -4.4440   10.0000   12.0000
   -4.4440  100.0000  120.0000
```

If we fail to match the dimensions exactly, MATLAB complains:

```
>> X(1:2,2:3) = [10 12]
Subscripted assignment dimension mismatch.
```

Here we specified a 2-by-2 subarray on the left but gave a 1-by-2 array on the right. Tsk, tsk!

Finally, when we specify, on the left of an assignment statement, a subarray of a matrix that does not exist, MATLAB will instantly define a new matrix for us just as when in the previous subsection we specified a single element of a non-existent matrix:

```
>> ultimate_answer(2,4:5) = 42
ultimate_answer =
     0     0     0     0     0
     0     0     0    42    42
```

Also, as in the previous section, when we specified a single element outside the range of an existing matrix, MATLAB will immediately enlarge a matrix if elements are added to a subarray outside its range:

```
>> ultimate_answer(1:2,5:6) = [6 28; 496 8128]
ultimate_answer =
   0     0     0     0     6    28
   0     0     0    42   496  8128
```

Here is an example in which we use this enlargement feature of the subarray operation to form a three-dimensional array:

```
>> A3d(:,:,3) = [1 2 3;4 5 6]
A3d(:,:,1) =
     0     0     0
     0     0     0
A3d(:,:,2) =
     0     0     0
     0     0     0
A3d(:,:,3) =
     1     2     3
     4     5     6
```

Before this command was issued, the array named **A3d** had not been defined. In the command, on the left of the equals sign, we used three indices, which means that **A3d** must be treated as a three-dimensional array. We used a colon for the row index, we used another colon for the column index, and we we used a **3** for the third index. As we mentioned at the beginning of this section in subsection Matrices and Arrays, like "row", and "column", the third dimension has a name too. It is called the **page** of the three-dimensional array. Thus, we specified page **3** of **A3d**. Said another way, we have told MATLAB to assign something to the third page of **A3d**. Since we used colons for the other dimensions, we have provided MATLAB with values to the entire third page.

MATLAB defines **A3d** for us and it assigns the value that we specified on the right, which is a 2x3 array. We can see from the result that MATLAB displays a three-dimensional array a bit differently from a matrix (i.e., two-dimensional array) or vector (i.e., one-dimensional array). It gives each page separately, printing the page in the same 2x3 tableau form that it uses for a matrix. It can also be seen, that, as always, it fills in zeros to maintain a rectangular shape, in this case a hyper-rectangular, 2x3x3 shape.

Combining Matrices To Build New Ones

MATLAB allows us to combine matrices in order to build new matrices, and it gives us four options:

1. Matrices that have the same size and shape (same number of rows and same number of columns) can be placed together in any arrangement that forms a rectangular array just as can be done with scalars. They can be placed on the same rows or in the same columns, as long as the result is rectangular. For example, if **A1 = [1 1 1; 1 1 1]**, **A2 = [2 2 2; 2 2 2]**, and **A3 = [3 3 3; 3 3 3]** (all have 2 rows and three columns), then the following combinations are legal:

```
>> [A1 A2 A3]
ans =
  1   1   1   2   2   2   3   3   3
  1   1   1   2   2   2   3   3   3
```

```
>> [A1; A2; A3]
ans =
  1     1     1
  1     1     1
  2     2     2
  2     2     2
  3     3     3
  3     3     3

>> [A1 A2 A3; A2 A2 A1]
ans =
  1   1   1   2   2   2   3   3   3
  1   1   1   2   2   2   3   3   3
  2   2   2   2   2   2   1   1   1
  2   2   2   2   2   2   1   1   1
```

2. Matrices that have the same number of rows can be placed side-by-side on the same row. For example, if we construct these three matrices, all of which have two rows:

```
>> B1 = [1;1]
B1 =
     1
     1

>> B2 = [2 2; 2 2]
B2 =
     2     2
     2     2

>> B3 = [3 3 3; 3 3 3]
B3 =
     3     3     3
     3     3     3
```

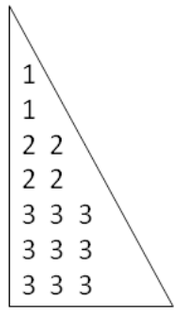
then the following combinations are legal:

```
>> [B1 B2]
ans =
     1     2     2
     1     2     2

>> [B1 B2 B3]
ans =
     1     2     2     3     3     3
     1     2     2     3     3     3
```

However, since it does not lead to a rectangular array, stacking them vertically is illegal:

```
>> [B1;B2;B3]
Error using vertcat
CAT arguments dimensions are not consistent.
```

Here is a picture of what we were trying to create:



It looks nice, and it might be even useful (somehow), but it is not a rectangular arrangement, and so MATLAB will not make it into an array.

(NOTE: **`vertcat`**, which appears in the error message above, is a function that MATLAB calls behind the scenes to stack matrices vertically. The suffix "cat" stands for "catenate", which, like "concatenate" means "to link together".)

3. Matrices that have the same number of columns can be placed atop one another in the same column. For example, if we construct these matrices, all of which have two columns:

```
>> C1 = [1 1]
C1 =
       1     1

>> C2 = [2 2; 2 2]
C2 =
     2     2
     2     2

>> C3 = [3 3; 3 3; 3 3]
C3 =
     3     3
     3     3
     3     3
```

then the following combinations are legal

```
>> [C1; C2]
ans =
     1     1
     2     2
     2     2

>> [C1; C2; C3]
ans =
     1     1
     2     2
     2     2
     3     3
     3     3
     3     3
```

However, since it does not lead to a rectangular array, putting them on the same row is illegal:

```
>> [C1 C2 C3]
Error using horzcat
CAT arguments dimensions are not consistent.
```

(NOTE: **horzcat**, which appears in this error, is function that MATLAB calls behind the scenes to place matrices side-by-side. See also **vertcat** above.)

Combinations of rules 2 and 3 may be used simultaneously as long as the result is rectangular. For example, if we define these arrays:

```
A1 = [1; 1],
B2 = [2 2; 2 2],
C3 = [3 3 3; 3 3 3],
D4 = [4 4; 4 4; 4 4]
E5 = [5 5 5 5; 5 5 5 5; 5 5 5 5]
```

then the following combination is legal:

```
>> [A1 B2 C3; D4 E5]

ans =
     1     2     2     3     3     3
     1     2     2     3     3     3
     4     4     5     5     5     5
     4     4     5     5     5     5
     4     4     5     5     5     5
```

Below we have drawn boxes around the assembled parts to show how they fit together:

1	2	2	3	3	3
1	2	2	3	3	3
4	4	5	5	5	5
4	4	5	5	5	5
4	4	5	5	5	5

The Transposition Operator

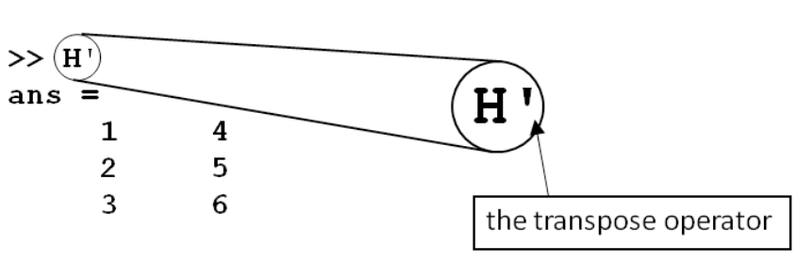
A matrix may be changed by **transposing** it, or taking its **transpose**, which means to interchange all its elements so that `X(m,n)` is replaced by `X(n,m)`. The results are that

- Each row of the new matrix is a column of the old matrix and vice versa.
- The number of rows of the new matrix equals the number of columns of the old matrix and vice versa.

The operator that is used to transpose a matrix is the transposition operator, and its symbol is an apostrophe, `'`, or single quote (usually found on the same key as the double quote). This operator, which is called the **transposition operator** , the **transpose operator**, or the **apostrophe operator** comes *after* its operand, which is the matrix that is to be transposed:

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

>> H'
ans =
     1     4
     2     5
     3     6
```

You might have to look close to see this operator:



So, we might say that `H'` is the transposition of `H`, or `H'` equals `H` transposed.

The `H` in the expression `H'` is the operand which the transposition operator operates on. An operator, such as the transposition operator, that takes only one operand is called a **unary operator**, while an operator, like `+` (plus), which operates on two operands (e.g., `1 + 2`) is called a **binary operator**. The phrase "binary operator" has nothing to do with the fact that the numbers are encoded in the computer in binary (though that is true). In this context the adjective "binary" instead describes the fact that there are two operands involved. Unary operators are less common than binary operators, other examples being the unary minus, `-H`, which negates its operand and the unary plus, `+H`, which, unlike the binary plus, does nothing(!). When the symbol for the operation comes after the operand, as in `H'`, the operator is said to be a **postfix** operator. When it comes before the operand as in `-H`, the operator is said to be a **prefix** operator. When it comes between operands, it is called an **infix** operator, as for example the plus operator in `1 + 2`.

Note that that transposition of a row vector changes it to a column vector:

```
>> x = [1 4 7]
x =
     1     4     7

>> x'
ans =
     1
     4
     7
```

and, vice versa,

```
>> y = [1;4;7]
y =
     1
     4
     7

>> y'
ans =
     1     4     7
```

By employing the transposition operator, we can change the row vectors that are always produced by the colon operator into column vectors:

```
>> x = (1:3:7)'
x =
     1
     4
     7
```

The parentheses are important here. If we omit them, nothing happens:

```
>> x = 1:3:7'
x =
     1     4     7
```

The reason that nothing happened is that only `7` is transposed when the parentheses are omitted, and `7' = 7`, because interchanging row and column for a scalar, for which they are both equal to one, does nothing. The parentheses force the transposition operator to operate on the entire row vector, changing it to a column vector.

We need to point out a peculiar detail regarding 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 will be replaced by its **complex conjugate**, which means that the sign of the imaginary part is changed. If we alter the example above to include a complex element, then the transpose will reveal this behavior:

```
>> H = [1 + 2i 2 3; 4 5 6]
H =
        1 + 2i        2             3
        4             5             6

>> H'
ans =
        1 - 2i        4
        2             5
        3             6
```

The imaginary component of element (1,1) has been changed from `+2i` to `-2i`. To take the transpose without taking any complex conjugates, it is necessary to precede the prime (`'`) by a period (`.`), or "dot", as follows:

```
>> H.'
ans =
        1 + 2i        4
        2             5
        3             6
```

This version of the transposition operator is called the **dot-apostrophe operator**. Of course if all the elements are real, these two versions of the transpose operator do the same thing.

Arithmetic And Matrix Arithmetic

Arithmetic is the manipulation of numbers with addition, subtraction, multiplication, division, and exponentiation (raising a number to a power). The operations are ordinarily thought of as applying to scalars, but they also apply to arrays.

Addition and subtraction

You can add and subtract matrices (and vectors, which are just special cases of 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`, then `Z = X+Y` is legal. The meaning of addition, which is also called both "array addition" and "matrix addition", for two dimensional arrays is as follows:

`Z = X + Y` means that for each `m` and `n`,

```
Z(m,n) = X(m,n) + Y(m,n)
```

which means that

```
Z( 1 , 1 )   =   X( 1 , 1 )   +   Y( 1 , 1 )
Z( 1 , 2 )   =   X( 1 , 2 )   +   Y( 1 , 2 )
. . .
Z( 1 ,end)   =   X( 1 ,end)   +   Y( 1 ,end)

Z( 2 , 1 )   =   X( 2 , 1 )   +   Y( 2 , 1 )
Z( 2 , 2 )   =   X( 2 , 2 )   +   Y( 2 , 2 )
. . .
Z( 2 ,end)   =   X( 2 ,end)   +   Y( 2 ,end)

. . .

Z(end, 1 )   =   X(end, 1 )   +   Y(end, 1 )
Z(end, 2 )   =   X(end, 2 )   +   Y(end, 2 )
. . .
Z(end,end)   =   X(end,end)   +   Y(end,end)
```

1st row

2nd row

lastrow

Here is a simple example involving only integer elements:

```
>> X = [1 5 -2; 3 0 7]
X =
     1     5    -2
     3     0     7

>> Y = [6 0  6; 2 2 1]
Y =
     6     0     6
     2     2     1

>> Z = X + Y
Z =
     7     5     4
     5     2     8
```

A separate addition is carried out for each of the six pairs of elements as depicted here:

1 + 6 → 7	5 + 0 → 5	−2 + 6 → 4
3 + 2 → 5	0 + 2 → 2	7 + 1 → 8

Array addition works for arrays with any number of dimensions, as long as the two operands have the same dimensions. Subtraction works similarly.

Both of the operators **+** and **−** operate on two operands. As mentioned in the previous section, we call such operators binary operators, and now we have seen that their operands can be arrays. Furthermore, both are infix operators, as they are in ordinary algebraic notation and in most other programming languages, (i.e., **X+Y**, instead of **XY+** or **+XY**). In fact all binary operators in MATLAB, as in algebra and most other programming languages, are infix operators.

Multiplication

Other binary operators include multiplication and division. There are two types of multiplication in MATLAB—array multiplication, which multiplies corresponding elements of the operands, and matrix multiplication, which performs the standard multiplication operation used in linear algebra. As we have mentioned before, MATLAB, like almost all computer languages, uses the asterisk to indicate multiplication. Array multiplication is specified this way, **Z = X.*Y**. Note the period, or "dot", before the *****. As with addition and subtraction, array multiplication requires that the two matrices be of the same size. The definition of array multiplication for two-dimensional arrays is as follows:

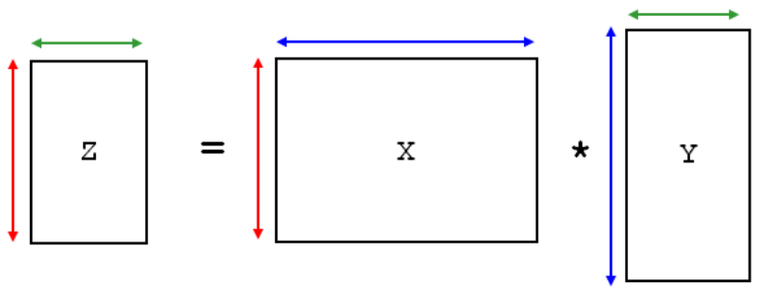
Z = X.*Y means that for each **m** and **n**,

Z(m,n) = X(m,n)*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. Like

array addition and subtraction, array multiplication works for arrays with any number of dimensions.

Matrix multiplication is different from array multiplication, as we mentioned above. It is specified as follows: `Z = X*Y` (no dot this time). The requirements on the dimensions of its operands is different from array multiplication as well. It requires that its operands have no more than two dimensions, and 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 schematic equation that follows, where the arrows of the same color have equal dimensions:



The blue lines are the "inner" dimensions of `X` and `Y` in the multiplication. The inner dimensions must be equal or multiplication is impossible. The red and green lines show the "outer" dimensions of `X` and `Y` in the multiplication. The outer dimensions determine the dimensions of the result.

The semantics of matrix multiplication in MATLAB (i.e., the meaning of matrix multiplication) 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:

`Z = X*Y` means that for each row `m` and column `n`,

$$Z(m,n) = \sum_k X(m,k)\,Y(k,n)$$

where the summation extends over all columns of `X` (and corresponding rows of `Y`).

Here is an example:

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

>> Y = [2 -2; 3 8; 7 4]
Y =
     2    -2
     3     8
     7     4

Z = X*Y
Z =
    29    26
    65    56
    22     0
   -18    -4
```

A separate summation is carried out for each of the eight elements as depicted below:

`1*2 + 2*3 + 3*7  → 29`	`1*(-2) + 2*8 + 3*4  → 26`
`4*2 + 5*3 + 6*7  → 65`	`4*(-2) + 5*8 + 6*4  → 56`
`6*2 + 1*3 + 1*7  → 22`	`6*(-2) + 1*8 + 1*4  → 0`
`0*2 + 1*3 + (-3)*7 → -18`	`0*(-2) + 1*8 + (-3)*4 → -4`

An important special case of matrix multiplication `X*Y` occurs when `Y` is a vector. In that case, the rule that the inner dimensions must match requires that `Y` be a column vector and that its number of elements be equal to the

number of columns of **X**. The result will be a column vector whose length is equal to the number of rows of **X**. This case will be especially important when we take up problems in linear algebra in the Linear Algebra section of Chapter 3. Here is an example of this special case using the same **X** as above with lower case letters being used for vectors, as is customary:

```
>> y = [2;3;7]
y =
     2
     3
     7

>> z = X*y
z =
    29
    65
    22
   -18
```

We note that **y** is equal to the first column of **Y**, and therefore the operations performed in calculating **X*y** are shown in the first column of the depiction of **X*Y** above.

Division

Division in MATLAB has four forms—two array forms and two matrix forms. For the array forms, the operands must be of equal size and shape, as for addition, subtraction, and array multiplication. The syntax includes a period, as it does for array multiplication. Once again we use two-dimensional arrays to illustrate:

Z = X./Y means that for each **m** and **n**,

Z(m,n) = X(m,n)/Y(m,n)

Z = X.\Y means that for each **m** and **n**,

Z(m,n) = Y(m,n)/X(m,n)

MATLAB is using the backslash operator in an interesting way here. Note that in this second example of division the elemental operation is **Y(m,n)** divided by **X(m,n)**, while in the first example, it was the other way around. You might say that **X./Y** means "**X** over **Y**", while **X.\Y** means "**X** under **Y**", and in fact it even looks a bit like the **X** is sliding under the **Y** with the backslash. At the least it is an easy way to remember which one is on top.

As with addition, subtraction, and array multiplication, array division works for any number of dimensions.

The two forms of matrix division involve the inverse (or "pseudoinverse") of a matrix and are a bit more complicated. Well, actually they are a lot more complicated. We will take them up in Chapter 3, which covers advanced concepts, in the section entitled Linear Algebra.

Exponentiation

Finally, there are exponentiation operations, also called "power operations". Array exponentiation, **Z = X.^Y**, requires that **X** and **Y** have the same shape. It means that for each **m** and **n**, **Z(m,n) = X(m,n)^Y(m,n)**, where the caret, **^**, means "raised to the power of". Matrix exponentiation, **Z = X^p**, has several meanings depending on the shapes of **Z** and **p**, but we will consider only the case in which **p** is a scalar with an integer value, in which case **X** must be a square matrix (same number of rows and columns, as defined at the beginning of this section). In this case, **Z** is defined to be equal to the result of multiplying **X** by itself **p** times, using matrix multiplication. Thus,

Z = X^p means that **Z = X*X*X...*X**

There are **p** Xs.

The reason that **X** must be a square matrix can be seen by noting that matrix multiplication requires that the inner dimensions be equal (see the subsection

Multiplication above), and when **X** is multiplied by **X**, the inner dimensions are in fact the rows and columns of **X** .

Operations involving scalars

The rules given above about the required shapes of the arrays and matrices for various operations were actually overstated just a bit. There is a special case in which one of the operands of addition, subtraction, multiplication, or array division is a scalar. For that case, the other operand may have any shape. Furthermore, in that case, array and matrix operations are the same! For example, **Z = c + Y**, where **c** is a scalar. For each operation the result is that **Z** has the same shape as **Y** and each element of **Z** is equal to **c** plus the corresponding element of **Y**. Here is a depiction for two-dimensions:

```
Z( 1 , 1 )   =   c  +  Y( 1 , 1 )
Z( 1 , 2 )   =   c  +  Y( 1 , 2 )      } 1st row
. . .
Z( 1 ,end)   =   c  +  Y( 1 ,end)

Z( 2 , 1 )   =   c  +  Y( 2 , 1 )
Z( 2 , 2 )   =   c  +  Y( 2 , 2 )      } 2nd row
. . .
Z( 2 ,end)   =   c  +  Y( 2 ,end)

. . .

Z(end, 1 )   =   c  +  Y(end, 1 )
Z(end, 2 )   =   c  +  Y(end, 2 )      } last row
. . .
Z(end,end)   =   c  +  Y(end,end)
```

Y + c is equal to **c + Y**. Here is an example of each:

```
Y = [1 2 3; -4 5 0]
     1     2     3
    -4     5     0

>> c = 6
c =
     6

>> Z = c + Y
Z =
     7     8     9
     2    11     6

>> Z = Y + c
Z =
     7     8     9
     2    11     6
```

Subtraction works similarly, and likewise multiplication: **c.*Y**, **c*Y**, **Y.*c**, and **Y*c** each mean that every element of **Y** is multiplied by **c**. There are even more ways to do division involving an array and a constant: **c.\Y**, **Y./c**, **Y/c**, and **c\Y** each mean that every element of **Y** is divided by **c**, while **c./Y** and **Y.\c** each mean that **c** is divided by every element of **Y** to (neither **c/Y** nor **Y\c** is legal).

Combining the colon and arithmetic operators

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

```
>> 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
```

Functions For Transforming Arrays

MATLAB provides built-in functions for transforming existing matrices into new ones. Table 1.1 lists some handy ones.

Table 1.1 Functions for transforming matrices

FUNCTION	DESCRIPTION
`circshift`	shifts existing values along rows or columns, leaving shape the same
`permute`	permute the dimensions of an array
`repmat`	makes an array consisting of copies of an existing array
`reshape`	produces a new shape by reorganizing elements of an existing array

These functions can most easily be understood by means of some simple examples. They all work with input arrays of any number of dimensions, but their behavior can all be illustrated, as usual, with two-dimensional inputs. We begin with **`circshift`**:

```
>> A = [1 2 3 4;5 6 7 8;9 10 11 12; 13 14 15 16;17 18 19 20]
A =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
    17    18    19    20

>> circshift(A,1)
ans =
    17    18    19    20
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

It can be seen that all but the last row of **`A`** has been shifted down; the last row was moved to the top. The name **`circshift`** means "circular shift", and the meaning should be clear. The values move in a circular way with the values shifted down from the top row replaced by the ones at the bottom, which "circle" around to the top. This function is sometimes needed when an array of data is produced by one function in the right order to be processed by another function but beginning at the wrong place. For example, a function may produce weather data for days of the week on successive rows beginning with Monday, while the function to process it expects the week to start with Sunday.

The shift can be increased by increasing the second argument:

```
>> circshift(A,3)
ans =
     9    10    11    12
    13    14    15    16
    17    18    19    20
     1     2     3     4
     5     6     7     8
```

A negative shift goes the other way:

```
>> circshift(A,-3)
ans =
    13    14    15    16
    17    18    19    20
     1     2     3     4
     5     6     7     8
     9    10    11    12
```

And what if we want to shift horizontally, instead of vertically? That is only a bit more difficult. We specify that the shift is for the second dimension, by making the second argument a vector, and putting the desired shift in the second element:

```
>> circshift(A,[0,1])
ans =
     4     1     2     3
     8     5     6     7
    12     9    10    11
    16    13    14    15
    20    17    18    19
```

It is possible to shift in both directions at the same time:

```
>> circshift(A,[1,1])
ans =
    20    17    18    19
     4     1     2     3
     8     5     6     7
    12     9    10    11
    16    13    14    15
```

Note that the order in which MATLAB carries out the two shifts is unimportant; it has no effect on the result.

Now, let's look at **permute**, which allows you to turn rows into columns and columns into rows, just as the transpose operator does for two-dimensional arrays and also allows you to interchange dimensions for three-dimensional arrays and higher. First, just to get acquainted with it, let's use it to do something that we would never need to do: Let's see how we can use it to duplicate the action of the transpose operator. We start with a 2x3 matrix:

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

We first transpose it with the transpose operator:

```
>> A_transpose = A'
A_transpose =
     1     4
     2     5
     3     6
```

As expected, the dimensions have been swapped to 3x2.

Now we do the same thing using **permute**:

```
>> A_permute = permute(A,[2,1])
A_permute =
     1     4
     2     5
     3     6
```

The function permute takes two input arguments. The first argument is the array to be transformed; the second one is a vector of numbers representing the new order of dimensions. The **2** in the vector **[2,1]** means "change the 2nd dimension to be the 1st (change rows to columns), and the **1** means "change the 1st dimension to be the 2nd (change columns to rows). The second number is redundant when only two dimensions are involved, but not for higher dimensions. OK. Transpose is a perfectly good operator, so we don't need permute for transposing matrices, but we will see that it comes up short for higher-dimensional arrays. Let's suppose we have defined this 2x3x4 array:

```
>> A = randi(89,2,3,4)+10
A(:,:,1) =
    42    91    45
    50    85    76
A(:,:,2) =
    36    37    92
    37    13    72
A(:,:,3) =
    31    81    30
    55    44    75
A(:,:,4) =
    79    66    56
    72    78    40
```

We might want to transpose each of the pages. Let's try that with the transpose operator:

```
>> A'
Error using  '
Transpose on ND array is not defined.
```

MATLAB complains that its transpose operator does not work for an "ND array", which means an n-dimensional array in which n is greater than 2. So let's put permute to work on the problem:

```
>> A_permute = permute(A,[2,1,3])
A_permute(:,:,1) =
    42    50
    91    85
    45    76
A_permute(:,:,2) =
    36    37
    37    13
    92    72
A_permute(:,:,3) =
    31    55
    81    44
    30    75
A_permute(:,:,4) =
    79    72
    66    78
    56    40
```

We have gone from a 2x3x4 array to a 3x2x4 array. Finally, let's scramble all three dimensions. We'll put the 3rd dimension first, the 1st dimension second, and the 2nd dimension third:

```
>> A_permute = permute(A,[3,1,2])
A_permute(:,:,1) =
    42    50
    36    37
    31    55
    79    72
A_permute(:,:,2) =
    91    85
    37    13
    81    44
    66    78
A_permute(:,:,3) =
    45    76
    92    72
    30    75
    56    40
```

We now have a 4x2x3 array.

Next we look at examples for **repmat**, whose name means "replicate matrix":

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

>> repmat(A,1,2)
ans =
     1     2     3     1     2     3
     4     5     6     4     5     6
```

It can be seen from this example that **repmat** has returned a matrix that contains two copies of **A**, arranged side by side. The second and third arguments instructed **repmat** to make one copy in the first dimension of **A** (vertical direction) and two copies in the second dimension (the horizontal direction). Here is an example of its use. Suppose we have measured a set of three-dimensional positions on a rigid object, and each point is stored in one column of the array **P**—like this:

```
>> P = [1 5 2 7 5 4; 3 5 3 6 7 0; 4 3 5 4 3 8]
P =
     1     5     2     7     5     4
     3     5     3     6     7     0
     4     3     5     4     3     8
```

Now, suppose the object is nudged by the following displacement vector

```
>> nudge = [0.2; 0.1; 0.3]
nudge =
          0.2
          0.1
          0.3
```

To change the points to their new positions, maybe we can add nudge to **P**:

```
 > P_nudged = P + nudge
Error using  +
Matrix dimensions must agree.
```

As we have seen above, MATLAB will let us add a scalar to an array but not a vector. The only acceptable shape of non-scalar array we can add to **P** is one with the same shape as **P** itself—**3**-by-**6**. What we need is to put six copies of shift side-by-side into an array, and **repmat** can do just that:

```
>> P_nudged = P + repmat(nudge,1,6)
ans =
  1.2    5.2    2.2    7.2    5.2    4.2
  3.1    5.1    3.1    6.1    7.1    0.1
  4.3    3.3    5.3    4.3    3.3    8.3
```

Mission accomplished! There is a second option for specifying the degree of replication along the dimensions of **A**:

```
>> repmat(A,[1,2])
ans =
     1     2     3     1     2     3
     4     5     6     4     5     6
```

With this second option, the replication numbers are listed in a vector. This option is handy when the number of dimensions of **A** is not known until the **repmat** is called. It is also the more general option as can be seen when we try to specify replication in three dimensions:

```
>> repmat(A,2,2,3)
Error using repmat
Too many input arguments.
```

Three dimensions is too many for the first option, but it is no problem for the second one:

```
>> repmat(A,[2,2,3])
ans(:,:,1) =
     1     2     3     1     2     3
     4     5     6     4     5     6
     1     2     3     1     2     3
     4     5     6     4     5     6
ans(:,:,2) =
     1     2     3     1     2     3
     4     5     6     4     5     6
     1     2     3     1     2     3
     4     5     6     4     5     6
ans(:,:,3) =
     1     2     3     1     2     3
     4     5     6     4     5     6
     1     2     3     1     2     3
     4     5     6     4     5     6
```

We see from this result that **repmat** has the ability to increase the number of dimensions—in this case from two to three.

The next function is **reshape**. Like **repmat**, it can change the number of dimensions, but like **circshift** , it does not change the number of elements, as can be seen in this example:

```
>> B = [11 12 13 14; 21 22 23 24;
        31 32 33 34; 41 42 43 44;
        51 52 53 54; 61 62 63 64]
B =
    11    12    13    14
    21    22    23    24
    31    32    33    34
    41    42    43    44
    51    52    53    54
    61    62    63    64

>> C = reshape(B,8,3)
C =
    11    32    53
    21    42    63
    31    52    14
    41    62    24
    51    13    34
    61    23    44
    12    33    54
    22    43    64
```

B is a **6**-by-**4** array, which has **24** elements, and **reshape** has created a new array, **C**, which is an **8**-by-**3** array, which also contains **24** elements. The elements are taken by **reshape** from its input matrix in column-major order and they are inserted into its output matrix in column-major order. Thus, **B(1,1)** is copied into **C(1,1)**, **B(2,1)** is copied into **C(2,1)**, …, **B(6,1)** is copied into **C(6,1)**, and we have reached the end of the first column of **B**. The sequence then moves to the first element of the second column of **B**—**B(1,2)**, which is copied into **C(7,1)**, then **B(2,2)** is copied into **C(1,8)**, and we have reached the end of the first column of **C**. Copying into **C** continues from the beginning of its second column with **B(3,2)** being copied into **C(1,2)**, etc.

In the example above, the number of dimensions—two—was left unchanged, but in the next example, **reshape** produces a three-dimensional output array **D** from the two-dimensional input array **B**:

```
>> D = reshape(B,2,3,4)
D(:,:,1) =
    11    31    51
    21    41    61
D(:,:,2) =
    12    32    52
    22    42    62
D(:,:,3) =
    13    33    53
    23    43    63
D(:,:,4) =
    14    34    54
    24    44    64
```

Any combination of input and output dimensions is possible as long as the number of elements, which equals the product of the dimensions, is the same for the input and output matrices. This rule allows **reshape** to provide a shorthand version of dimension specifications. If one of the dimensions is the empty matrix, **[]**, then that empty matrix will be replaced by the number that will cause the output array to have the same number of elements as the input:

```
>> D = reshape(B,2,[],4)
D(:,:,1) =
    11    31    51
    21    41    61
D(:,:,2) =
    12    32    52
    22    42    62
D(:,:,3) =
    13    33    53
    23    43    63
D(:,:,4) =
    14    34    54
    24    44    64
```

In this case **[]** was replaced by **3**, and the result is the same as the example in which all three dimensions were given explicitly.

Like **repmat**, **reshape** also accepts its dimensions in vector form. For example, **reshape(B,[2,3,4])** is equivalent to **reshape(B,2,3,4)**.

We conclude with one final example:

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

>> v = reshape(A,6,1)
v =
     1
     4
     2
     5
     3
     6
```

In this case reshape returns a column vector of the elements in **A**. An equivalent result is produced by the command **v = reshape(A,[],1)**, which means put all the elements of **A** into a column vector **v**. There is an even simpler way to get this job done:

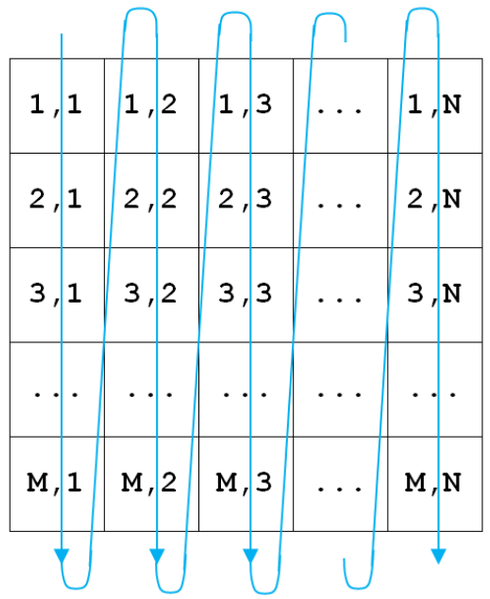
```
>> A = [1 2 3;4 5 6]
A =
     1     2     3
     4     5     6

>> v = A(:)
v =
     1
     4
     2
     5
     3
     6
```

One might have expected MATLAB to issue an error message, since only one index was given for a two-dimensional array, but it is not an error. This is an example of **linear indexing**, which is the specification of just one index for an array, regardless of its number of dimensions. The meaning is that the array is treated as a column vector and the indexing is in column-major order. If the successive indices are written out, then the row index changes fastest. For an **M**-by-**N** matrix **X**, column-major order would be as follows:

```
(1,1), (2,1), (3,1), ..., (M,1), (1,2), (2,2), (3,2), ...,
(M,2), (1,3), (2,3), (3,3), ..., (M,3),..., (1,N), (2,N),
(3,N), ..., (M,N),
```

as is illustrated in Figure 1.13.

Figure 1.13 *Column-major order*



When the symbol **:** is used as a linear-index, it means "all the elements" in column-major order, just as it does for an ordinary vector. The number **5**, when used as a linear index, means the fifth element in column-major order, and all other linear indexing works in the same way: **1:3** means the first **3** elements, **end-3:end** means the last four elements, etc., in column-major order, as the following excerpts from the Command Window illustrate:

```
>> A(5)

ans =
     3
```

```
>> A(1:3)
ans =
     1     4     2

>> A(end-3:end)
ans =
     2     5     3     6
```

Operator Precedence And Associativity

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

```
>> 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 the order in which to apply operations.

Precedence

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 `+` in the expression `c + a*b`. An operator's ranking relative to other operators to determine the order in which they are applied within an expression 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 is given in Table 1.2. A lower number indicates higher precedence and means that the operator is applied earlier in the expression. The complete table for all MATLAB operators is given in Chapter 2 in Table 2.13 of the section entitled Selection.

Table 1.2 *Operator Precedence*

PRECEDENCE	OPERATOR
0	Parentheses: `(...)`
1	Exponentiation `^` and Transpose `'`
2	Unary `+`, Unary `–`, and logical negation: `~`
3	Multiplication and Division (array and matrix)
4	Addition and Subtraction
5	Colon operator `:`

If a pair of parentheses occurs in an expression, as, for example in `z = 8*(4+2)`, then the expression inside the parentheses is evaluated first. So `4 + 2` is evaluated to get `6` before the multiplication is carried out, and the result is `8*6`, which produces `48`. 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.

While we are familiar enough with the precedences of the arithmetic operators that we can guess what will happen when they are combined, we have no experience outside MATLAB with the colon operator, so we may not know how to interpret `1:3 + 10`, which combines the colon operator and the plus operator. If the colon were to operate first, then the result would be `[1,2,3] + 10`, which equals `[11,12,13]`. If instead the plus operates

first, then the result will be `[1:13]` , which equals `[1,2,3,4,5,6,7,8,9,10,11,12,13]`. MATLAB follows its precedence table slavishly, so we can consult it to find out which answer is correct. When we do, we find that the precedence of the plus operator is 2, while the precedence of the colon operator is 5. Lower numbers means higher precedence, so plus has the higher precedence and will operate first. Let's check it out in the Command Window:

```
>> 1:3 + 10
ans =
     1    2    3    4    5    6    7    8    9   10   11   12   13
```

As expected, MATLAB has followed its precedence rules: the plus operation was carried out before the colon operation.

Associativity

If more than one binary operator of the *same* precedence occurs in an expression, then the precedence table is no help. For example, in the expression `4-8-2`, the table does not tell us whether the left `-` acts first or the right `-`. In other words, the table does not tell us whether this expression means `(4-8)-2`, for which the lefthand `-` acts first and which equals `-6`, or `4-(8-2)`, for which the righthand `-` acts first and which equals – 2. In fact, the first one is correct. MATLAB's rule for the order for applying multiple operators of the same precedence is that the order is left-to-right. The order in which operators of equal precedence are applied is called the **associativity** of the operators. Operators that operate from left-to-right, like all those in MATLAB, are called left-to-right associative or left-associative. As a second example, consider `8/4*2`, which involves two operators, `*` and `/`, that have the same precedence. It is very tempting to think that this expression means `8/(4*2)`, which equals `1`, but, in fact, since the order of application of operators of the same precedence is always left-to-right in MATLAB, the meaning is in fact `(8/4)*2`, which equals `4`. Here are some more examples of the left-to-right rule:

`2^3^4` equals `(2^3)^4` , which equals $(2^3)^4$, which equals `4096`

`2/8*4` equals `(2/8)*4`, which equals `0.25 * 4`, which equals `1`

`2-3+4` equals `(2-3)+4`, which equals `3`

Left-to-right associativity for operators of the same precedence is common in computer programming languages. Fortran, however, employs right-to-left associativity for exponentiation. Its exponentiation operator is `**`. Thus the MATLAB expression `2^3^4` , which MATLAB evaluates left-to-right as `4096`, would be written in Fortran this way: `2**3**4`, but it would be equal to `2**(3**4)`, which equals a somewhat larger number: 2417851639229258349412352. There is no exponentiation operation in C, C++, or Java. In these languages `x^y` is achieved by calling a function called "pow", for "power": `pow(x,y)`. Thus, for example, in C and C++, the first expression above must be written as `pow(pow(2,3),4)`. This expression looks confusing (and sounds dangerous!). MATLAB's syntax is much more natural.

Associativity has another meaning as well. It is a property that some operators possess. 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 the property of associativity. It is said to be associative. Addition is associative as well. Consider subtraction, on the other hand, for which `A-(B-C)` is not equal to `(A-B)-C`. Subtraction is not associative, nor are division and exponentiation. In other words, neither division nor exponentiation possesses associativity.

Additional Online Resources

- Video lectures by the authors:

  Lesson 2.1 Introduction to Matrices and Operators (11:25)

  Lesson 2.2 The Colon Operator (8:45)

  Lesson 2.3 Accessing Parts of a Matrix (21:33)

  Lesson 2.4 Combining and Transforming Matrices (10:06)

  Lesson 2.5 Arithmetic Part 1 (17:49)

  Lesson 2.6 Arithmetic Part 2 (11:52)

  Lesson 2.7 Operator Precedence (13:31)

Concepts From This Section

Computer Science and Mathematics:

matrix
scalar
function
argument
column-major order
vector
    column vector
    row vector
complex numbers
operand
unary operators
postfix and prefix operators
transposition
matrix arithmetic
operator precedence and associativity

MATLAB:

matrix
empty 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