CHAPTER

2

MATLAB Tutorial

2.1. GOAL OF THIS CHAPTER

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

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

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

2.2. BASIC CONCEPTS

2.2.1. Purpose and Philosophy of MATLAB

MATLAB is a high-performance programming environment for numerical and technical applications. The first version was written at the University of New Mexico in the 1970s. The "MATrix LABoratory" program was invented by Cleve Moler to provide a simple

and interactive way to write programs using the Linpack and Eispack libraries of FORTRAN subroutines for matrix manipulation. MATLAB has since evolved to become an effective and powerful tool for programming, data visualization and analysis, education, engineering and research.

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

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

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

2.2.2. Getting Started

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

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

tutorial will be in MATLAB notation, to enhance your familiarity with the MATLAB environment.

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

2.2.3. MATLAB as a Calculator

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

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

These operations are straightforward:

Addition:

```
>> 2 + 3

ans =

   5
```

Subtraction:

```
>> 7 - 5

ans = 2
```

Multiplication:

```
>> 17 * 4

ans =

   68
```

Division:

```
>> 24 / 7

ans =

   3.4286
```

Following are some points to note:

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

```
>> 5          + 12
ans =
   17
```

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

```
>> 2 + 3 + 4 - 7 * 5 + 8 / 9 + 1 - 5 * 6 / 3
ans =
   -34.1111
```

3. Parentheses disambiguate statements in the usual way:

```
>> 5 + 3 * 8
ans =
   29
>> (5 +; 3) * 8
ans =
   64
```

"Advanced" mathematical operators: Powers, log, exponentials, trigonometry. Power: $x \hat{} p$ is x to the power p:

```
>> 2 ^ 3
ans =
   8
```

Natural logarithm: log:

```
>> log (2.7183)
ans =
   1.0000
>> log(1)
ans =
   0
```

Exponential: exp(x) is e^x

```
>> exp(1)
ans =
   2.7183
```

Trigonometric functions; for example, sine:

```
>> sin(0)
ans =
   0
>> sin(pi/2)
ans =
   1
>> sin(3/2 * pi)
ans =
   -1
```

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

```
>> log(2.7183)
ans =
   1.0000
>> format long
>> log(2.7183)
ans =
   1.000006684913988
>> format short
>> log(2.7183)
ans =
   1.0000
```

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

```
>> 5*7

ans =

   35

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

ans =

   35.0000
```

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

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

For example:

```
>> 2^500

ans =

   3.2734e + 150
```

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

x e + y means x * 10 ^ y.

Example:

```
>> 2e3

ans =

   2000

>> 2*10^3

ans =

   2000
```

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

```
>> ans + ans

ans =

   4000
```

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

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

2.2.4. Defining Matrices

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

A simple row vector is as follows:

```
>> [1 2 3]

ans =

   1   2   3
```

It contains the elements 1, 2, and 3.

A simple matrix is as follows:

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

ans =

   2   2   2
   3   3   3
```

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

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

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

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

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

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

a =

   1   2   3   4   5
```

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

b =

   6   7   8   9
```

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

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

```
>> a

a =

   1   2   3   4   5

>> b

b =

   6   7   8   9
```

Note:

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

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

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

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

   uno =

      1   1   1
      1   1   1
      1   1   1

   >> thismatrixisreallyempty = [ ]

   thismatrixisreallyempty =

      [ ]
   ```

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

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

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

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

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

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

```
>> z = x + y
z=80
```

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

```
>> who
Your variables are:
a  ans  b thismatrixisreallyempty  uno  x  y  z
>> whos
```

Name	Size	Bytes	Class
a	1×5	40	double
ans	2×3	48	double
b	1×4	32	double
thismatrixisreallyempty	0×0	0	double
uno	3×3	72	double
x	1×1	8	double
y	1×1	8	double
z	1×1	8	double

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

```
save my_workspace x y z
```

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

```
>> save my_workspace
```

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

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

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

```
>> load my_workspace
>> who

Your variables are:

a ans b thismatrixisreallyempty uno x y z
```

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

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

Your variables are:

a ans b thismatrixisreallyempty uno
```

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

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

```
>> size(a)

ans =

   1   5

>> length(a)

ans =

   5
```

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

```
>> thismatrixisreallyempty = [5]

thismatrixisreallyempty =

   5
```

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

```
>> thisiscool = 4:18

thisiscool =

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

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

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

thisiscool =

   4   6   8   10   12   14   16   18
```

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

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

v =

        10   100   1000   10000   100000
```

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

```
>> a

a =

   1   2   3   4   5

>> a'

ans =

   1
   2
   3
   4
   5
```

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

```
>> b

b =

   6   7   8   9

>> b = a

b =

   1   2   3   4   5
```

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

```
>> a

a =

   1   2   3   4   5

>> a = a'

a =

   1
   2
   3
   4
   5

>> a

a =

   1
   2
   3
   4
   5
```

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

This reassignment also works for elements of matrices and vectors:

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

```
a =

   1
   9
   3
   4
   5
```

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

```
>> a(2)

ans =

   9
```

We will explore indexing more in Section 2.2.6.

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

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

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

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

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

2.2.5. Basic Matrix Algebra

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

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

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

p =

   1   2
   3   4

>> p = p + 2

p =

   3   4
   5   6
```

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

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

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

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

q =

   2   1
   1   1

>> m = p + q

m =

   5   5
   6   7
```

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

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

r =

   2   1
   1   1
   1   1

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

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

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

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

```
>> p

p =

   3   4
   5   6

>> q

q =

   2   1
   1   1

>> p*q

ans =

   10    7
   16   11

>> p.*q

ans =

   6   4
   5   6
```

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

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

a =

   1   2   3   4   5

>> b = [5 4 5 4 5]

b =

   5   4   5   4   5
```

```
>> c = a.*b

c =

   5   8   15   16   25

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

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

```
>> c = a.^2

c =

   1   4   9   16   25
```

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

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

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

We will explore indexing further in the next section.

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

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

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

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

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

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

A =

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

>> rref(A)

ans =

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

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

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

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

Revenue =
   2150
```

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

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

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

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

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

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

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

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

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

2.2.6. Indexing

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

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

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

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

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

```
A =

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

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

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

A =

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

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

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

ans =

    1    3    4
```

Thus, the following command will produce the desired result:

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

A =

    23    23    23    23

    57     6    57    57

    23    20    30    40

    23    60    70    80
```

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

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

i =

    2
    3
```

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

A =

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

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

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

A =

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

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

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

r =

    3

c =

    2
```

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

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

MATLAB also used the following syntax for logical operators:

& (AND)
| (OR)

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

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

2.3. GRAPHICS AND VISUALIZATION

2.3.1. Basic Visualization

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

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

```
>> x = 0:10

x =

   0   1   2   3   4   5   6   7   8   9   10

>> y = sin(x)

y =

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

>> plot(x,y)
```

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

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

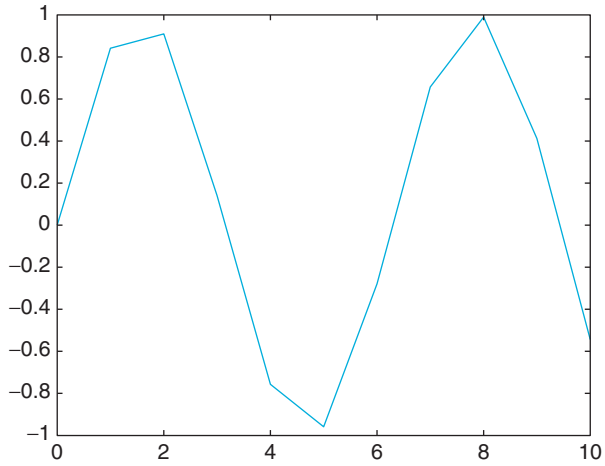

FIGURE 2.1 Crude sinusoid.

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

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

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

So try this:

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

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

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

Upon completing Exercise 2.11 enter the following commands:

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

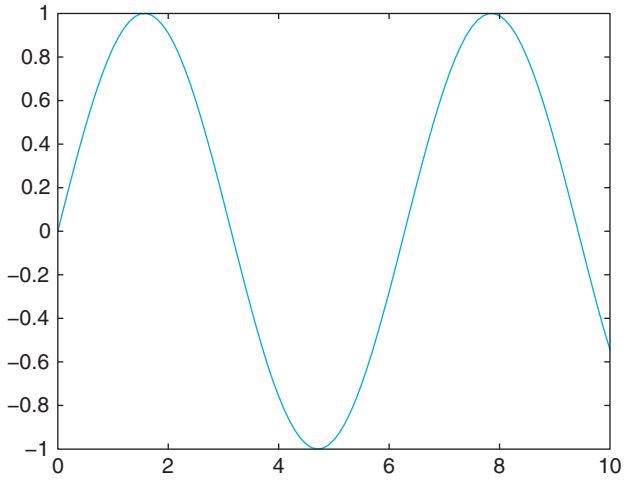

FIGURE 2.2 Smooth sinusoid.

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

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

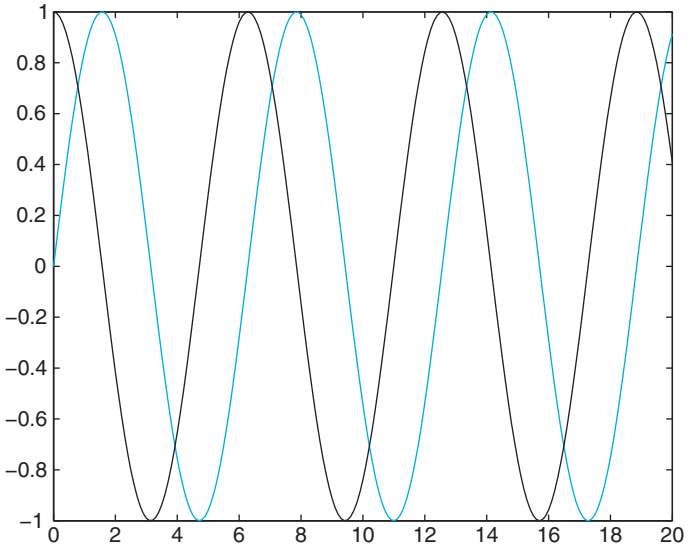

FIGURE 2.3 Sine vs. cosine.

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

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

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

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

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

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

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

results =
   55   30   10    5

>> bar(results)
```

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

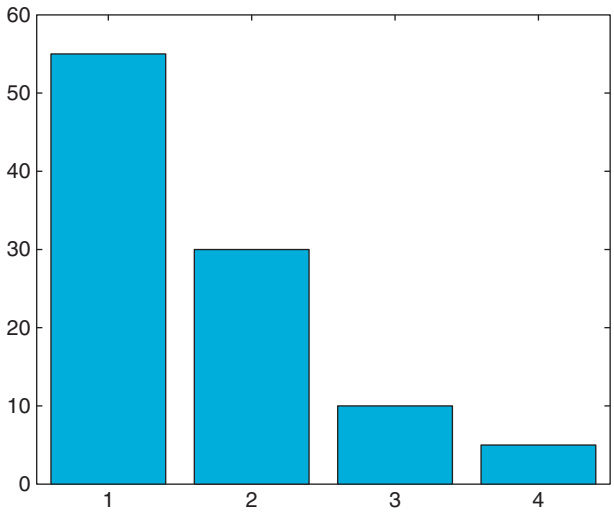

FIGURE 2.4 Lowering the bar.

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

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

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

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

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

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

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

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

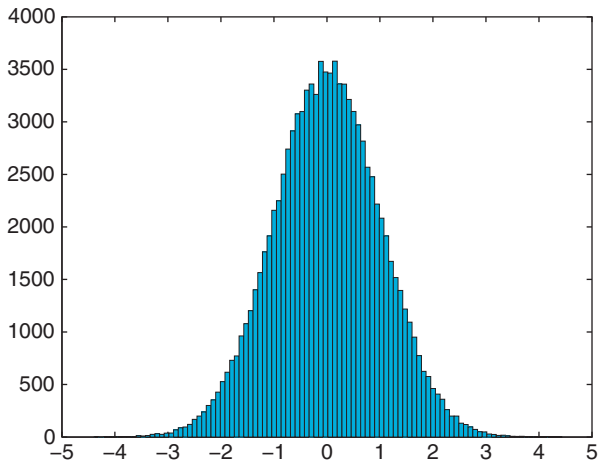

FIGURE 2.5 Gaussian normal distribution.

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

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

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

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

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

2.4. FUNCTION AND SCRIPTS

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

The primary goal of this section is to help you become familiar with M-files within MATLAB. M-files are the primary vehicle to implement programming in MATLAB. So while the previous sections showed how MATLAB can double as a scientific calculator and as a calculator with graphing functions, this section will introduce you to a high-level programming language that should be able to satisfy most of your programming needs if you are a casual user. It will become apparent in this section of the tutorial how MATLAB can aid the researcher in all stages of an experiment or study. By no means is this tutorial the last word on M-files and programming. Later we will elaborate on all concepts introduced in this section—particularly in terms of improving efficiency and performance of the programs you are writing. One final goal of this tutorial is to demonstrate the

remarkable versatility of MATLAB—and don't worry, we'll move on to neuroscience-heavy topics soon enough.

2.4.1. Scripts

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

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

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

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

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

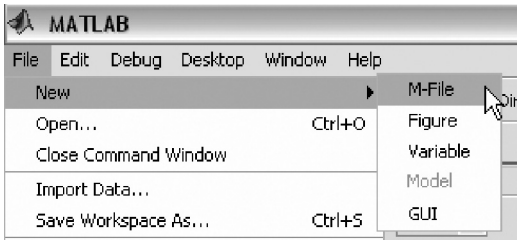

FIGURE 2.6 Creating a new M-File.

```
1 -  figure
2 -  x = 0:0.1:10;
3 -  y = sin (x);
4 -  plot (x,y)
5
```

FIGURE 2.7 The editor.

2.4.2. Functions

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

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

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

```
function r = triple(i)

r=3*i;
```

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

```
>> a=triple(7)

a =

   21

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

b =

   30   60   90
```

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

2.4.3. Control Structures

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

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

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

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

Statements
...
Statements

end

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Try this, save, and run:

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

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

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

if these-conditions-are-met

Execute-these-Statements

else

Execute-these-Statements

end

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

Fun with loops—How to make an American quilt

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4.4. Advanced Plotting

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

Then type the following:

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

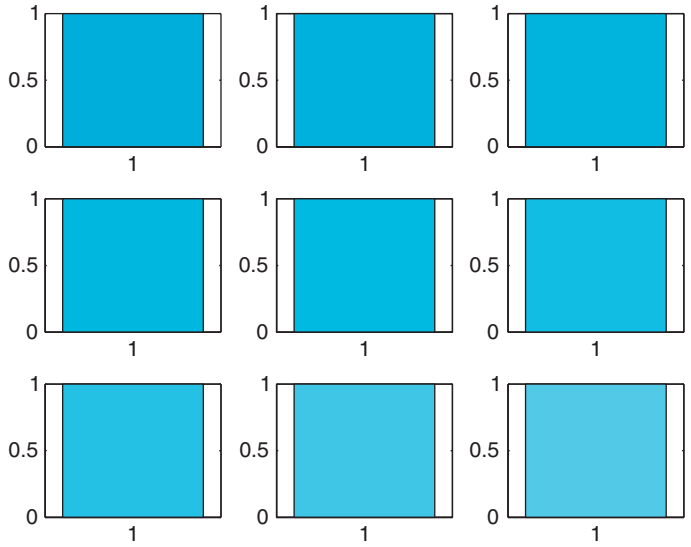

FIGURE 2.8 Color gradient subplots.

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

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

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

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

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

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

FIGURE 2.9 Maximizing a figure.

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

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

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

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

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

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

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

shading interp

Now try the following:

colormap hot

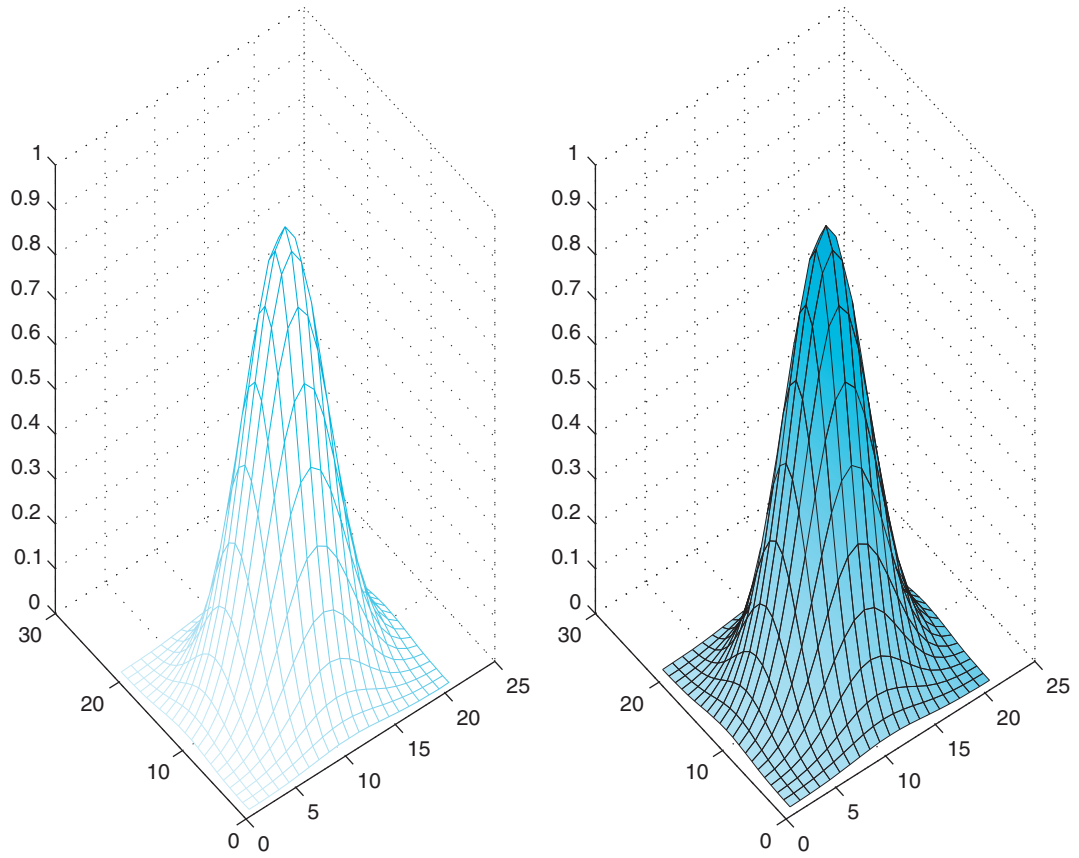

FIGURE 2.10 Three dimensional plotting of a Gaussian.

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

2.4.5. Interactive Programs

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

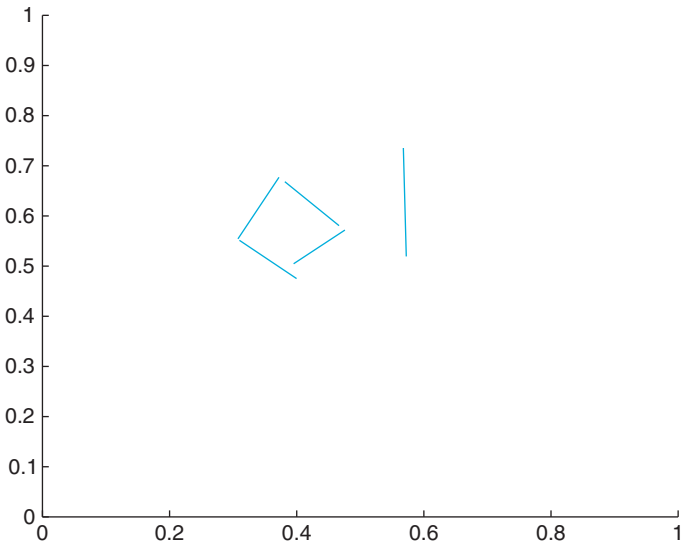

FIGURE 2.11 The luck of the draw.

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

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

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

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

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

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

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

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

Here is the code:

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

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

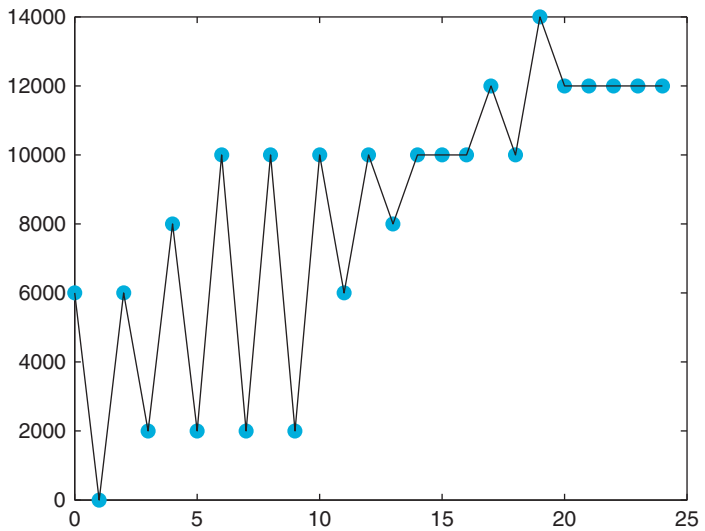

FIGURE 2.12 Game over.

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

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

2.5. DATA ANALYSIS

2.5.1. Importing and Storing Data

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

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

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

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

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

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

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

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

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

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

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

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

ans =
   0

>> isnan(M1(3,1))

ans =
   1
```

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

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

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

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

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

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

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

The result looks something like that shown in Figure 2.13.

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

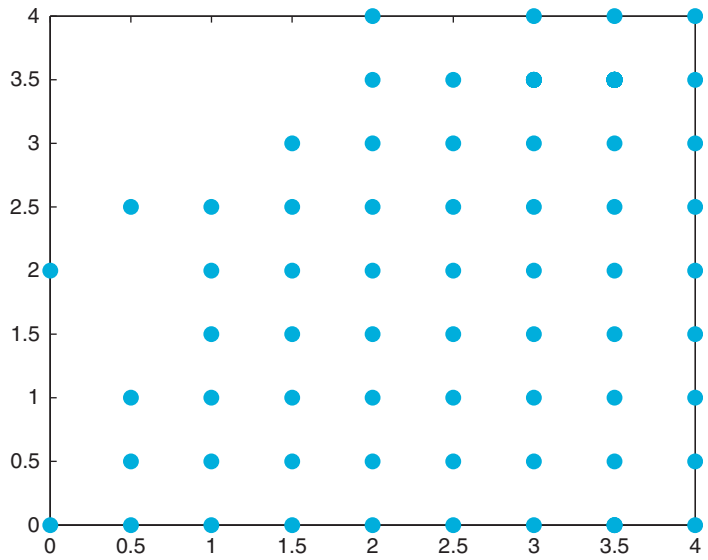

FIGURE 2.13 Low resolution.

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

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

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

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

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

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

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

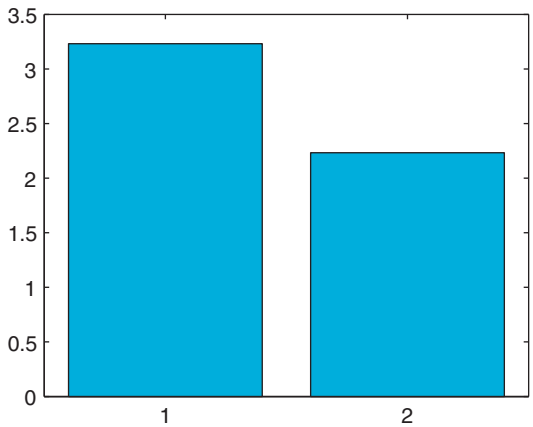

FIGURE 2.14 Means.

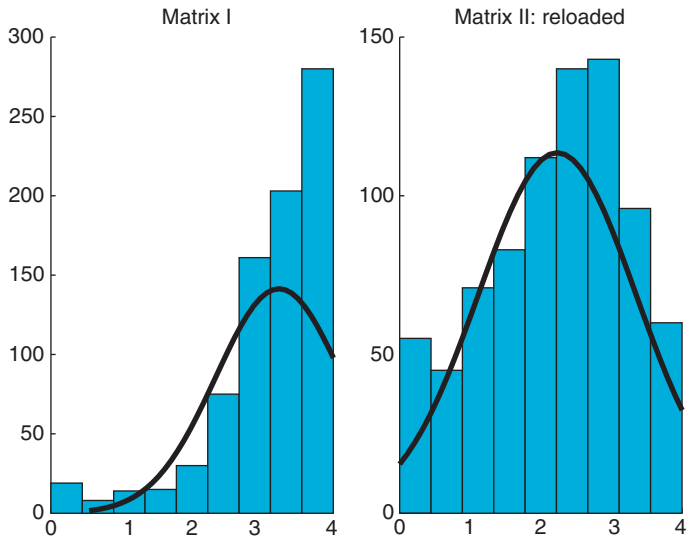

FIGURE 2.15 Variance.

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

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

Nevertheless, add the following code to the program:

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

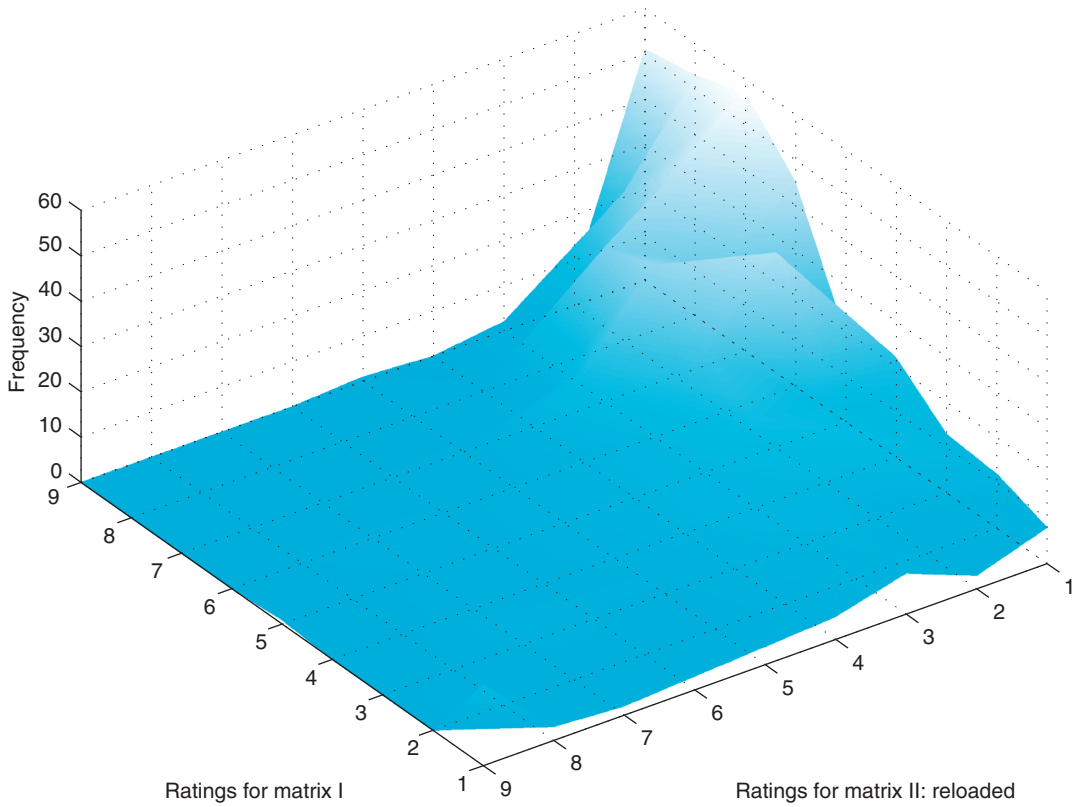

FIGURE 2.16 The real deal.

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

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

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

2.6. A WORD ON FUNCTION HANDLES

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

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

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

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

sin(*x*)

where x is the value of interest.

For example, type

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

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

Predictably, the result is

```
ans =

   0    1.0000    0.0000   -1.0000   -0.0000
```

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

h = @sin

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

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

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

yields

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

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

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

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

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

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

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

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

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

```
>> quad(h,0,pi)
ans =
   2.0000
>> quad(h,0,2*pi)
ans =
   0
```

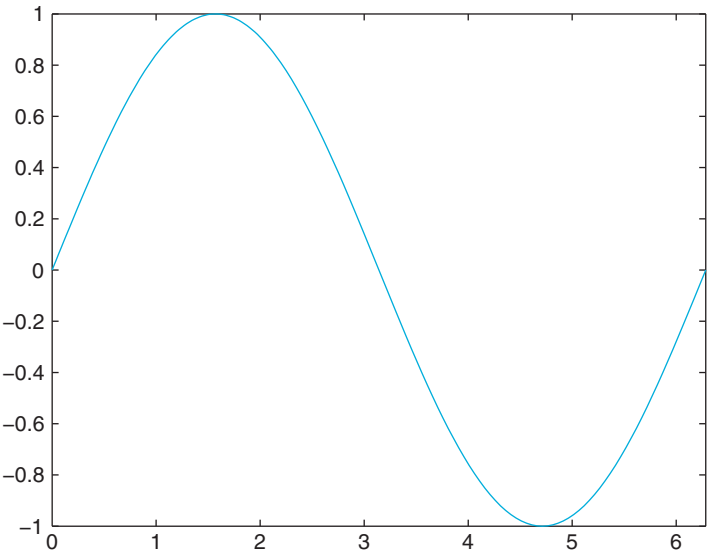

FIGURE 2.17 fplot in action.

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

ans =

    1.0000
```

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

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

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

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

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

The result is shown in Figure 2.18.

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

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

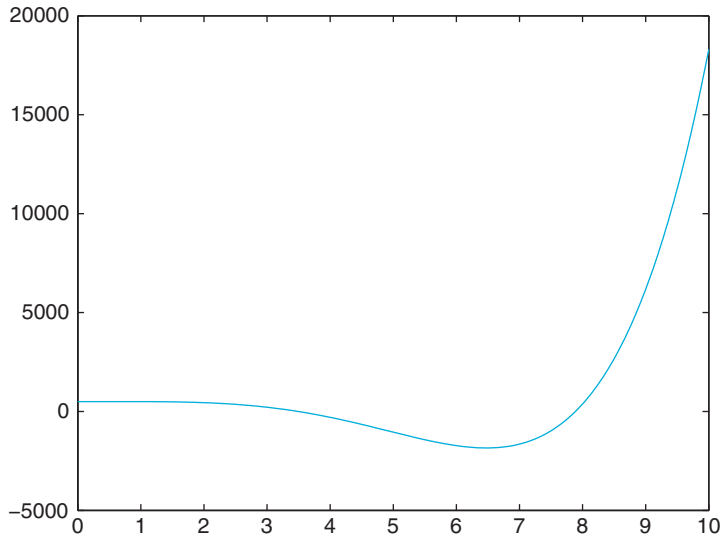

FIGURE 2.18 A polynomial in q, plotted from 0 to 10.

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

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

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

2.7. THE FUNCTION BROWSER

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

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

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

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

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

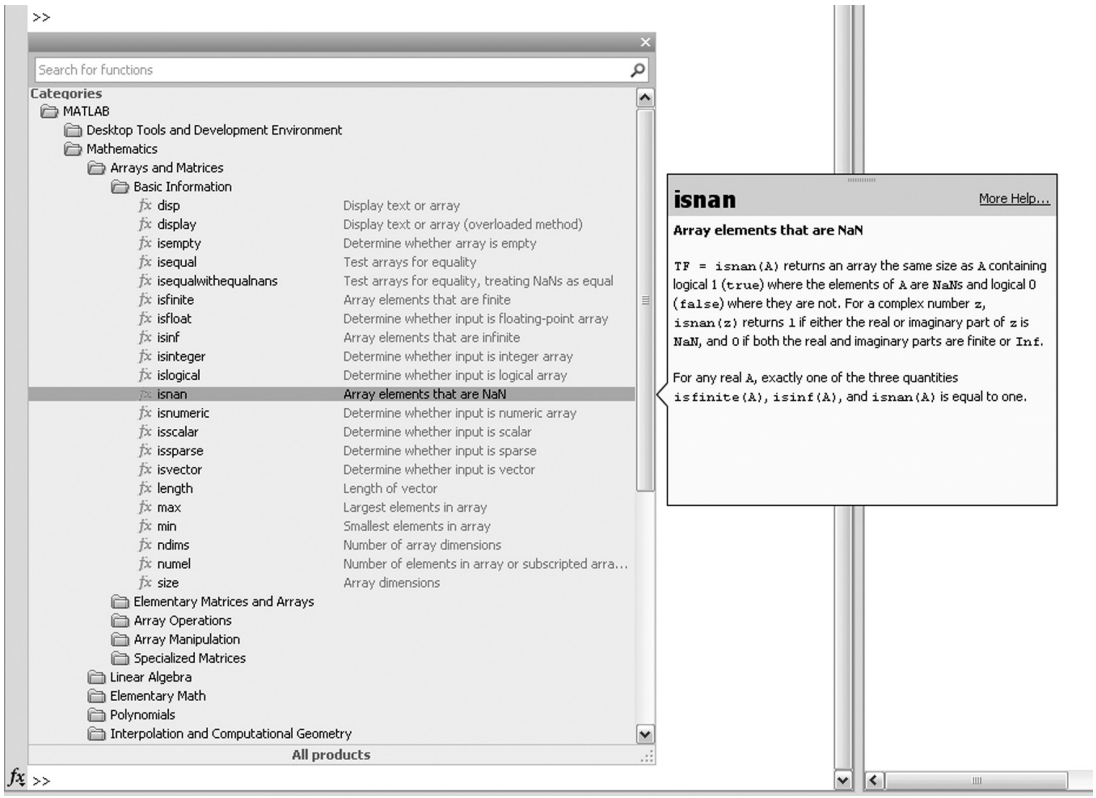

FIGURE 2.19 The function browser.

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

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

2.8. SUMMARY

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

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

MATLAB FUNCTIONS, COMMANDS, AND OPERATORS COVERED IN THIS CHAPTER

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

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