Appendix A–D

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our Web site at www.wiley.com.

APPENDIX A
Basic Linear Algebra Concepts

This appendix provides a very basic introduction to linear algebra concepts. Some of these concepts are presented in a somewhat simplified form (i.e., not as general as could be) here on purpose. Our goal is not to teach all the intricacies of this very important field in mathematics, but to enable readers to understand the linear algebra notation and applications described in this book.

A.1 SYSTEMS OF EQUATIONS

Consider a very simple equation:

$$a \cdot x = b.$$

In this equation, x is a variable, and a and b are input data. To find the value of the variable x, we would simply write

$$x = b/a.$$

Equivalently, we could have written

$$x = a^{-1} \cdot b.$$

Consider now a simple system of two linear equations:

$$\begin{aligned} 3 \cdot x_1 + 8 \cdot x_2 &= 46. \\ 10 \cdot x_1 - 7 \cdot x_2 &= -15. \end{aligned}$$

The variables in this system are x_1 and x_2, and the input data consist of the coefficients 3, 8, 10, -7 and the constants to the right of the equal sign,

Simulation and Optimization in Finance by Dessislava A. Pachamanova and Frank J. Fabozzi.

46 and −15. The way we would normally solve this system of equations is to express one of the variables through the other from one of the equations and plug into the other equation:

$$x_2 = \frac{46 - 3 \cdot x_1}{8}.$$

$$10 \cdot x_1 - 7 \cdot \frac{46 - 3 \cdot x_1}{8} = -1.5.$$

Therefore, $x_1 = 2$ and $x_2 = 5$.

It turns out that it is convenient to introduce new array notation that allows us to treat systems of equations similarly to a single equation. Suppose we put together the coefficients in front of the variables x_1 and x_2 into a 2 × 2 array **A**, the constants to the right hand side of the two equations into a 2 × 1 array **b**, and the variables themselves into a 2 × 1 array **x**. (Note that the first index counts the number of rows in the array, and the second index counts the number of columns in the array.) We have

$$\mathbf{A} = \begin{bmatrix} 3 & 8 \\ 10 & -7 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 46 \\ -15 \end{bmatrix}, \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}.$$

We would need to be careful in defining rules for array algebra so that, similarly to the case of solving a single equation, we can express an array of variables through the arrays for the inputs to the system of equations. Namely, we want to be able to write the system of equations as

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$$

and express the solution to the system of equations as

$$\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{b}.$$

This would substantially simplify the notation when dealing with arrays of data.

A.2 VECTORS AND MATRICES

Vectors and matrices are the terms used to describe arrays of data like the arrays **A**, **b**, and **x** in the previous section. Matrices can be arrays of any dimensions, for example, $N \times M$. The array **A** in the previous section was a matrix array. Vectors are matrices that have only one row or column, and are typically written as column arrays of dimensions $N \times 1$. You can imagine them as a listing of coordinates of a point in N-dimensional space. The **b** and the **x** arrays in the previous section were vector arrays. When an array consists of a single number, that is of dimensions 1×1, it is referred to as a *scalar*.

Typically, vectors and matrices are denoted by bold letters in order to differentiate arrays from single elements. Vectors are usually denoted by bold small letters, while matrices are denoted by bold capital letters. An individual element of a vector or a matrix array is represented by a small nonbold letter that corresponds to the letter used to denote the array, followed by its row and column index in the array. The element in the ith row and the jth column of the matrix array **A**, for example, is denoted a_{ij}. For the matrix **A** in section A.1, the element in the first row and second column is $a_{12} = 8$.

Some important matrix arrays include the *null matrix*, **0**, whose elements are all zeros, and the *identity matrix*, usually denoted **I**, which contains 1's in its left-to-right diagonal, and zeros everywhere else. The latter matrix is called identity matrix because every other matrix, multiplied by a matrix **I** of the appropriate dimensions, equals itself. We will introduce matrix multiplication in the next section.

A.3 MATRIX ALGEBRA

Matrix algebra works differently from classical algebra, but after a little bit of getting used to, the definitions of array operations are logical. We list some common operations below.

Matrix equality. Two matrices are equal only of their dimensions are the same and they have the same elements. Thus, for example,

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \neq \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

Transpose. The transpose of an $N \times M$ matrix with elements that are real numbers is an $M \times N$ matrix whose elements are the same as the elements of the original matrix, but are "swapped" around the left-to-right diagonal. The transpose of a matrix **A** is denoted $\mathbf{A}^{\mathrm{T}}$ or $\mathbf{A}'$. For example, the transpose of the matrix **A** in section A.1 is

$$\mathbf{A}' = \begin{bmatrix} 3 & 10 \\ 8 & -7 \end{bmatrix}.$$

The transpose of a vector makes a column array a row array, and vice versa. For example,

$$\mathbf{b}' = \begin{bmatrix} 46 \\ -15 \end{bmatrix}' = [46 \quad -15] .$$

Multiplication by a scalar. When a matrix is multiplied by a scalar (a single number), the resulting matrix is simply a matrix whose elements are all multiplied by that number. For example,

$$5 \cdot \mathbf{A} = 5 \cdot \begin{bmatrix} 3 & 8 \\ 10 & -7 \end{bmatrix} = \begin{bmatrix} 15 & 40 \\ 50 & -35 \end{bmatrix}.$$

The notation **–A** means $(-1) \cdot \mathbf{A}$, that is, a matrix whose elements are the negatives of the elements of the matrix **A**.

Sum of matrix arrays. When two matrices are added, we simply add the corresponding elements. Note that this implies that the matrix arrays that are added have the same row and column dimensions. For example, the sum of two 2 × 3 matrices will be a 2 × 3 matrix as well:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}.$$

Multiplication of matrix arrays. Matrix multiplication is perhaps the most confusing array operation to those who do not have background in linear algebra. Let us consider again the example in section A.1. We found that the values for the variables x_1 and x_2 that satisfy the system of equations are $x_1 = 2$ and $x_2 = 5$. Therefore, the vector of values for the variables is

$$\mathbf{x} = \begin{bmatrix} 2 \\ 5 \end{bmatrix}.$$

Recall also that

$$\mathbf{A} = \begin{bmatrix} 3 & 8 \\ 10 & -7 \end{bmatrix},$$

$$\mathbf{b} = \begin{bmatrix} 46 \\ -15 \end{bmatrix},$$

and that we need $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ to be true for the system of equations if our matrix algebra is to work in a useful way.

Let us compute the array product $\mathbf{A} \cdot \mathbf{x}$. Note that we cannot simply multiply **A** and **x** element-by-element because A is of dimension 2 × 2 and **x** is of dimension 2 × 1. It is not clear which elements in the two arrays "correspond" to each other. The correct way to perform the multiplication is to multiply and add together the corresponding elements in the first row of **A** by the elements of **x**, and the corresponding elements in the second row of **A** by the elements of **x**:

$$\mathbf{A} \cdot \mathbf{x} = \begin{bmatrix} 3 & 8 \\ 10 & -7 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 5 \end{bmatrix} = \begin{bmatrix} 3 \cdot 2 + 8 \cdot 5 \\ 10 \cdot 2 - 7 \cdot 5 \end{bmatrix} = \begin{bmatrix} 46 \\ -15 \end{bmatrix} = \mathbf{b}.$$

In general, suppose that we want to multiply two matrices, **P** of dimensions $N \times M$ and **Q** of dimensions $M \times T$. We have

$$\mathbf{P} \cdot \mathbf{Q} = \underbrace{\begin{bmatrix} p_{11} & \cdots & p_{1M} \\ \vdots & \ddots & \vdots \\ p_{N1} & \cdots & p_{NM} \end{bmatrix}}_{N \times M} \cdot \underbrace{\begin{bmatrix} q_{11} & \cdots & q_{1T} \\ \vdots & \ddots & \vdots \\ q_{M1} & \cdots & q_{MT} \end{bmatrix}}_{M \times T}$$

$$= \underbrace{\begin{bmatrix} \sum_{i=1}^{M} p_{1i} q_{i1} & \cdots & \sum_{i=1}^{M} p_{1i} q_{iT} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{M} p_{Ni} q_{i1} & \cdots & \sum_{i=1}^{M} p_{Ni} q_{iT} \end{bmatrix}}_{N \times T}.$$

In other words, the (i,j)th element of the product matrix $\mathbf{P} \cdot \mathbf{Q}$ is obtained by multiplying element-wise and then adding the elements of the ith row of the first matrix (**P**) and the jth column of the second matrix (**Q**).

Multiplications of more than two matrices can be carried out similarly, by performing a sequence of pairwise multiplications of matrices, but note that the dimensions of the matrices in the multiplication need to agree. For example, it is not possible to multiply a matrix of dimensions $N \times M$ and a matrix of dimensions $T \times M$. The number of columns in the first matrix must equal the number of rows in the second matrix. Similarly, in order to multiply more than two matrices, the number of columns in the second matrix must equal the number of rows in the third matrix, and so on. A product of an $N \times M$, an $M \times T$, and an $T \times S$ matrix will result in a matrix of dimensions $N \times S$. Thus, matrix multiplication is not equivalent to scalar multiplication in more ways than one. For example, it is not guaranteed to be commutative, that is, $\mathbf{P} \cdot \mathbf{Q} \neq \mathbf{Q} \cdot \mathbf{P}$.

It is possible to perform matrix multiplication in a way that is closer to standard arithmetic operations, that is, to multiply two matrices of the same dimensions so that each element in one matrix is multiplied by its corresponding element in the second matrix. However, using direct element-wise matrix multiplication is the special case rather than the default. Element-wise matrix multiplication is referred to as *Hadamard product*, and is typically denoted by "•" rather than "·".

Matrix inverse. We would like to be able to find the vector $\mathbf{x}$ from the system of equations in section A.1 in a way similar to the calculation of the value of the unknown variable from a single calculation. In other words, we would like to be able to compute $\mathbf{x}$ as

$$\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{b}.$$

This necessitates defining what $\mathbf{A}^{-1}$ (pronounced "$\mathbf{A}$ inverse") is. The inverse of a matrix $\mathbf{A}$ is simply the matrix that, when multiplied by the original matrix, produces an identity matrix. In other words,

$$\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I}.$$

How to find $\mathbf{A}^{-1}$ is not as straightforward. Software packages such as MATLAB have special commands for these operations. Intuitively, the way to find the inverse is to solve a system of equations, where the elements of the inverse matrix are the variables, and the elements of $\mathbf{A}$ and $\mathbf{I}$ are the input data.

It is important to note that not all matrices have inverses. However, some kinds of matrices, such as symmetric positive definite matrices, which are typically used in financial applications, always do. (See the definition of symmetric positive definite matrices in the next section.)

A.4 IMPORTANT DEFINITIONS

Some special matrices are widely used in financial applications. Most often, practitioners are concerned with covariance and correlation matrices for asset returns. Such matrices are special in that they are symmetric and, theoretically, need to be positive definite. We explain what these terms mean below.

Symmetric matrix. A matrix **A** is symmetric if the elements below its left-to-right diagonal are mirror images of the elements above its left-to-right diagonal. In other words, it is the same as its transpose, that is, **A** = **A'**. Covariance and correlation matrices are always symmetric.

Positive definite and positive semidefinite matrices. The main idea behind defining a *positive definite matrix* is to create a definition of an array that shares some of the main properties of a positive real number. Namely, the idea is that if you multiply the equivalent of a square of a vector by it, you will obtain a positive quantity. If a matrix **A** is positive definite, then

$$\mathbf{z}'\mathbf{A}\mathbf{z} > 0 \text{ for any vector } \mathbf{z} \text{ of appropriate dimensions.}$$

Similarly, a *positive semidefinite matrix* shares some properties of a *nonnegative* real number. Namely, if a matrix **A** is positive semidefinite,

$$\mathbf{z}'\mathbf{A}\mathbf{z} \geq 0 \text{ for any vector } \mathbf{z} \text{ of appropriate dimensions.}$$

APPENDIX B

Introduction to @RISK

This appendix is a brief introduction to @RISK, a Microsoft Excel add-in that allows users to incorporate simulation into worksheet models. @RISK is a part of a collection of Excel add-ins, Palisade Decision Tools, distributed commercially by the Palisade Corporation, http://www.palisade.com. Other Excel add-ins in the package include Evolver (an add-in for optimization), StatTools (an add-in for statistical analysis), and Risk-Optimizer (an add-in for optimization and simulation).

In this appendix, all path references to commands and items in the @RISK menu are based on @RISK 5.0 and MS Excel 2007. While the locations of the menu items are different depending on the version of @RISK used, the nature of the commands and the items to select from the @RISK menu remains the same. We recommend that you use the @RISK help to search for the commands in any version of @RISK.

B.1 BASIC OPTIONS IN @RISK

When @RISK is activated in Excel, it has its own tab on the Excel ribbon (see Exhibit B.1).

The functions of the most important commands for our purposes are listed in Exhibit B.2.

B.2 EXAMPLE

To give a specific example for how one may create and interpret a simulation model in @RISK, let us consider the following situation. This is a version of a famous problem called the *Newsvendor Problem*. Although this example is from the area of operations management rather than finance, it is very intuitive and makes it easy to illustrate the main features of @RISK.

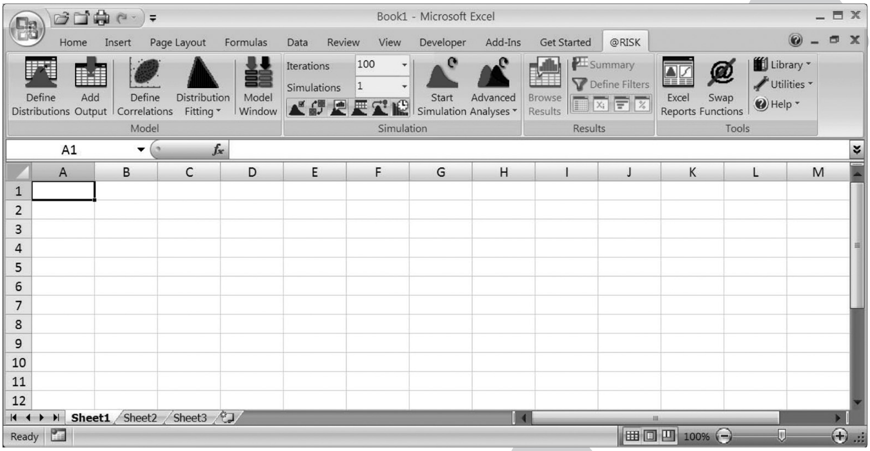


EXHIBIT B.1 @RISK tab on the Excel ribbon.

Suppose you are the manager of a store. It is April, and you need to place an order for a certain number of coats to be sold in your store in the fall. Often, such orders take several months to fulfill, so you need to guess the demand for next year's coats far in advance. The price of a coat is $110, the cost of a coat to you is $70, and a marketing analyst you consulted has forecasted that the demand for this type of coat in your store will be about 3,000 items, with a standard deviation of 500 items. The future demand for the coats, of course, is random. Let us assume that it is a normal random variable.[1] If you order more coats than the quantity demanded by your customers, you will sell the leftover coats at the end of the season at a steep discount—at $40 per item. (If you order fewer items than the quantity demanded by your customers, you will be losing the opportunity cost of selling the items to the customers, but let's not worry about that.)

What is the "optimal" order quantity? An "optimal" order quantity would result in the highest expected profit, with a low degree of "risk." We will discuss these concepts in detail in Chapters 3 and 4 of the book, but for now let us see how we can model the profit. It may seem that the manager should order as many coats as he expects to be demanded on average (i.e., 3,000); however, we will see that this is not necessarily the optimal solution.

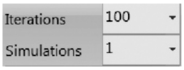
B.2.1 Building an Excel Model

Before using @RISK, we need to build a worksheet model that would compute the profit if we knew the demand with certainty. This is always the

EXHIBIT B.2 Important commands under the @RISK tab.

Define Distributions in the Model group. Allows the user to select the probability distribution for an uncertain input from a gallery of continuous and discrete distributions. Equivalently, one can specify the input distribution directly by typing a formula in the cell designated as an input cell for the model. For example, instead of selecting a normal distribution with mean 50 and standard deviation 120 by clicking the Define Distribution, one can type `=RiskNormal(50,120)`. As you get used to @RISK, you may find that the direct formula specification is a more convenient way to enter the input probability distribution, because it can be made part of Excel formulas, and can include cell references rather than specific numbers. For example, you can write a formula of the kind `=IF(RiskNormal(50,120) > 40, 1, 0)`: if the random number realization satisfies a given condition (in this case, if it is greater than 40), then do something (in this case, assign a value of 1 to the current cell); otherwise, do something else (in this case, assign a value of 0 to the current cell). (This function is explained in more detail in section B.2.)

Add Output in the Model group. Allows the user to specify which cells in the worksheet are output cells, that is, which cells @RISK should track during the simulation. Note that output cells should contain formulas that link them to values of the input cells (or depend on the input cells through references to other cells in the worksheet); otherwise, their value will be constant, and will not be updated when new scenarios are generated, rendering the simulation meaningless. Equivalently, output cells can be specified by entering the formula `=RiskOutput(‘‘Name of Output’’)+` in front of the Excel formula in the designated cell.



Iterations 100

Simulations 1

Iterations/Simulations in the Simulation group. These options allow the user to specify the number of iterations (called also trials or scenarios) for the simulation and the number of simulations to run. The option for specifying the number of simulations to run is used when the function `RiskSimtable` is called in the worksheet. As we will explain later in this appendix, `RiskSimtable` allows the user to test different strategies through simulation. The specific strategies are passed as a table with numerical values. The number of simulations the user chooses in the main menu tells @RISK how many of the numerical values in the `RiskSimtable` to use.

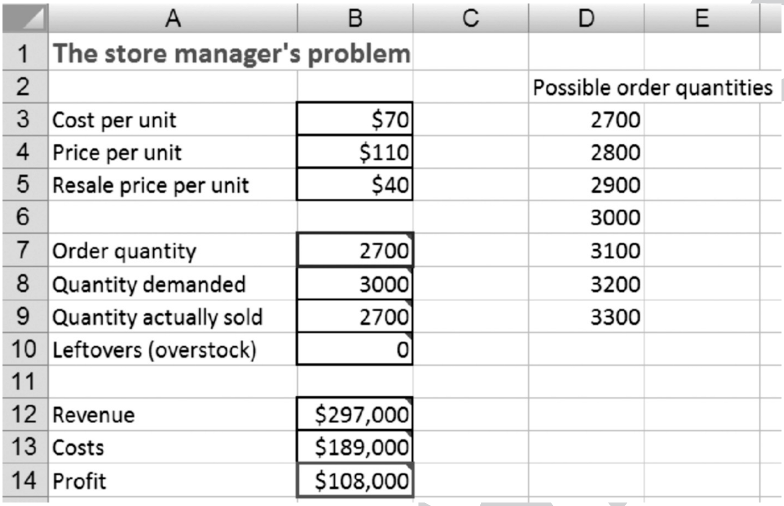
(*Continued*)

EXHIBIT B.2 (*Continued*)

	Simulation Settings in the Simulation group. Allows the user to specify options such as: - Method of scenario generation (see section 4.4.5 in Chapter 4). Under the **Sampling** tab, click drop-down box **Sampling type**. - Whether or not to recalculate the random numbers in the worksheet every time new information is entered. Under the **General** tab, click **Random Values** or **Static Values**. We recommend that you choose **Static Values** while building the model in Excel to make entering data faster, and enable recalculation (**Random Values**) once the model is complete to test whether the model is doing the right thing. The same switch between random values and static values can be accomplished by clicking in the Simulation group under the @RISK tab.
Start Simulation	*Start Simulation* in the Simulation group. Starts the simulation.
Browse Results	*Browse Results* in the Results group. After running the simulation, pressing and unpressing it shows or hides simulation charts on the worksheet.
	Simulation Statistics in the Results group. After running the simulation, clicking this button shows summaries of simulation results, such as means, variances and percentiles for inputs and outputs.
Xi	*Simulation Data* in the Results group. After running the simulation, clicking this button shows the actual scenarios that were generated for the data.

case—before running a simulation, we need a model that works if all inputs are known with certainty. We can then enhance the model by incorporating considerations for uncertainty.

A snapshot of the model is shown in Exhibit B.3. (The model itself can be found in file **B-OptimalOrderQty.xlsx**.) In cells B3:B5, we store the data on the cost, price, and discount value of the coats. Cells D3:D9 contain possible order quantities.



	A	B	C	D	E
1	**The store manager's problem**				
2				Possible order quantities	
3	Cost per unit	$70		2700	
4	Price per unit	$110		2800	
5	Resale price per unit	$40		2900	
6				3000	
7	Order quantity	2700		3100	
8	Quantity demanded	3000		3200	
9	Quantity actually sold	2700		3300	
10	Leftovers (overstock)	0			
11					
12	Revenue	$297,000			
13	Costs	$189,000			
14	Profit	$108,000			

EXHIBIT B.3 @RISK model in file **B-OptimalOrderQty.xlsx.**

Cell B7 contains a value for the order quantity: 2,700. Cell B8 contains a possible value for the quantity demanded −3,000. (This is if we assume that exactly the average forecast will happen.) Cell B9 contains the formula

```
=MIN(B7,B8)
```

This is the quantity of coats that will be actually sold – the store cannot sell more than the number of coats ordered or more than the number of coats that will be demanded.

Cell B10 contains the formula

```
=B7-B9
```

This expression (the difference between the number of coats ordered and the number of coats actually sold) is the number of coats left at the end of the season.

Cell B12 contains the formula

```
=B4*B9+B5*B10
```

This is the revenue that will be realized from sales during the season and from sales of the leftover coats.

The costs are computed in cell B13. They depend on the number of coats ordered, and are incurred independently of the level of demand. The exact formula is

```
=B3*B7
```

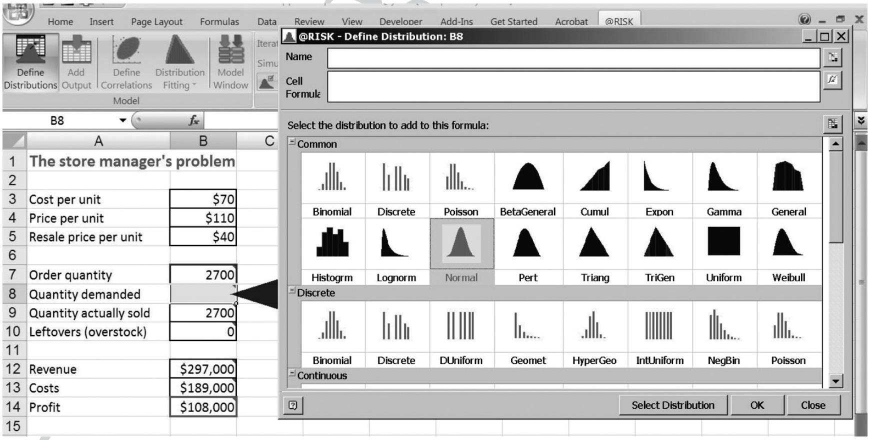
Finally, the profit is computed in cell B14 as the difference between the revenues and the costs:

```
=B12-B13
```

As is evident in Exhibit B.2, if the manager orders 3,000 coats and demand is 3,000 coats, the profit will be $108,000.

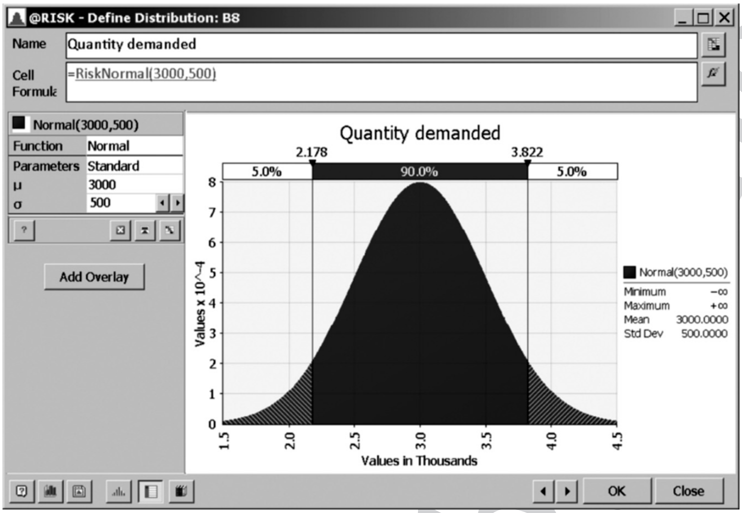
B.2.2 Specifying Inputs and Outputs

Now let us incorporate the information we have about demand uncertainty. Demand will be assumed to be normally distributed with a mean of 3,000 and a standard deviation of 500 coats. This is an *input* to the model. To specify an input distribution, we click the **Define Distributions** command in the Model group under the @RISK tab. We can then choose from a gallery of distributions. (See Exhibit B.4.)



	A	B
1	The store manager's problem	
2		
3	Cost per unit	$70
4	Price per unit	$110
5	Resale price per unit	$40
6		
7	Order quantity	2700
8	Quantity demanded	
9	Quantity actually sold	2700
10	Leftovers (overstock)	0
11		
12	Revenue	$297,000
13	Costs	$189,000
14	Profit	$108,000
15		

EXHIBIT B.4 Gallery of probability distributions that can be selected by clicking on the **Define Distributions** command in @RISK.



@RISK - Define Distribution: B8

Name: Quantity demanded

Cell Formula: =RiskNormal(3000,500)

Normal(3000,500)

Function	Normal
Parameters	Standard
μ	3000
σ	500

Normal(3000,500)

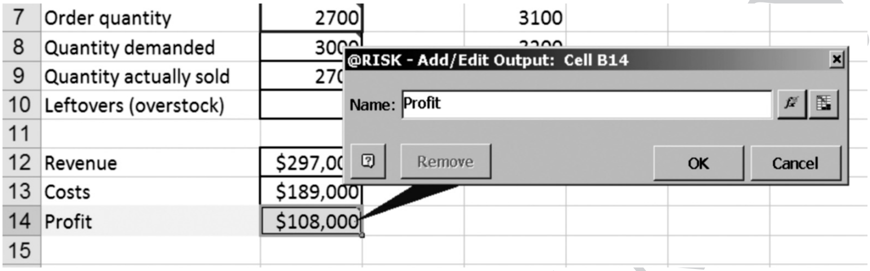
Minimum	−∞
Maximum	+∞
Mean	3000.0000
Std Dev	500.0000

EXHIBIT B.5 Normal distribution dialog box.

We select the normal distribution, and then fill out its dialog box as shown in Exhibit B.4. We specify μ (the mean of the distribution) to be 3,000, and σ (the standard deviation of the distribution) to be 500. The graph on the right in Exhibit B.5 shows the shape of the distribution with these parameters. The second box in the top part of the dialog box shows the formula @RISK will enter in Excel (`=RiskNormal(3000,500)`) after you click **OK**. The numbers in that formula can be replaced with Excel cell references if you prefer to store the information in the worksheet rather than hardcoding it into the @RISK formula. Eventually, as you become more familiar with @RISK, you may find it more convenient to type the formulas for the probability distributions directly into Excel instead of going through the **Define Distributions** dialog box.

After defining the input cell(s),[2] we define the output cell. Every @RISK model should have at least one output cell. The output cell contains a reference to the quantity we would like to track. In this case, it is the profit. To specify the output cell, we click **Add Output** in the Models group under the @RISK tab, and obtain the Add Output dialog box (Exhibit B.6). We type the name of the output cell (in this case, it is “Profit”), and @RISK adds an expression (`RiskOutput(‘‘Profit’’) +`) to the formula in the Excel cell B14. Cell B14 then contains

```
=RiskOutput(‘‘Profit’’)+B12-B13
```



7	Order quantity	2700	3100
8	Quantity demanded	3000	
9	Quantity actually sold	270	
10	Leftovers (overstock)		
11			
12	Revenue	$297,00	
13	Costs	$189,000	
14	Profit	$108,000	
15			

@RISK - Add/Edit Output: Cell B14

Name: Profit

EXHIBIT B.6 Add Output dialog box.

Again, as you get more familiar with @RISK, you may prefer simply to type the expression in the Excel formula instead of going through the **Add Output** dialog box.

Once we have specified the inputs and outputs for @RISK, it is helpful to press the button in the Simulation group under the @RISK tab. Then, every time we press F9, @RISK generates a random number, and the profit is recalculated. This is useful for checking if our model behaves the way we would expect it to behave, and for finding potential errors.

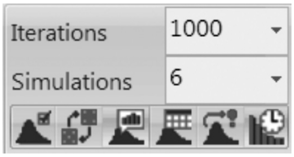
B.2.3 Specifying Options with `RiskSimtable`

In this example, the manager has several options for order quantities. They are listed in cells D3:D9. @RISK has a function, `RiskSimtable`, that allows us to pass this information. We modify the formula in cell B7 (the order quantity) to be

```
=RiskSimtable(D3:D9)
```

and change the number of simulations in the @RISK dialog box to 6 (which equals the number of values passed as arguments in `RiskSimtable`). See Exhibit B.7. This makes @RISK run the simulation for the six order quantities in `RiskSimtable`. If we do not change the number of simulations from the default (which is 1) to 6, @RISK will run the simulation only for the first value in `RiskSimtable`.

Using `RiskSimtable` is helpful because @RISK evaluates all options for the order quantity over the same set of scenarios. In other words, @RISK generates the number of scenarios we ask it to generate, and then computes the profit with a new order quantity, but the same scenarios. This allows for



Iterations 1000

Simulations 6

EXHIBIT B.7 Specifying the number of simulations with `RiskSimtable`.

a fair comparison between different strategies (order quantities), as discussed in section 4.2.3 of Chapter 4.

B.2.4 Running the Simulation

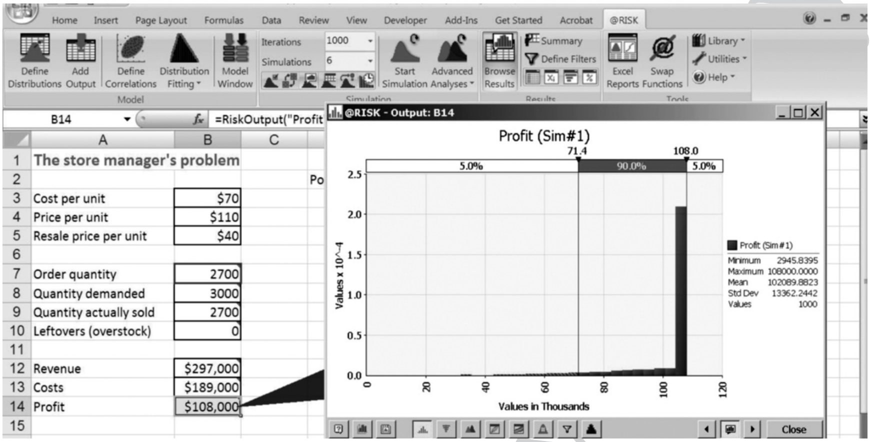
After creating the Excel model and specifying the input and output cells, we are ready to run the simulation. We can specify the number of scenarios (trials) we would like @RISK to generate by typing the number under **Iterations** on the @RISK tab (see Exhibit B.6). In our example, we would like @RISK to generate 1,000 scenarios. Other simulation options can be specified by clicking the button in the Simulations group under the @RISK tab. Some of the options include type of simulation method to use,[3] and whether to run an Excel Macro before or after running @RISK recalculations.

Once we have specified the simulation options, we click the **Start Simulation** button.

B.2.5 Reading Simulation Output

After the simulation has stopped running, the results automatically pop on screen (see Exhibit B.8). The results can be hidden or appear on screen by pressing on the **Browse Results** in Results group under the @RISK tab. If we click on different cells in the worksheet, the simulation results for those cells will appear. We ran six simulations (for six values of order quantity in `RiskSimtable`), and the simulation number (1 through 6) will be explicitly shown in the graph or table with output data.

The pop-up boxes let us analyze the histogram of outcomes (i.e., the plot of values, 1000 in our example, that were generated during the simulation), the descriptive statistics,[4] and sensitivity information such as tornado graphs.[5] By clicking and dragging the bars that define the 90% in the middle of the distribution, we can compute the percentage of scenarios that were



Iterations 1000
Simulations 6

B14 =RiskOutput("Profit

	A	B	C
1	The store manager's problem		
2			Po
3	Cost per unit	$70	
4	Price per unit	$110	
5	Resale price per unit	$40	
6			
7	Order quantity	2700	
8	Quantity demanded	3000	
9	Quantity actually sold	2700	
10	Leftovers (overstock)	0	
11			
12	Revenue	$297,000	
13	Costs	$189,000	
14	Profit	$108,000	
15			

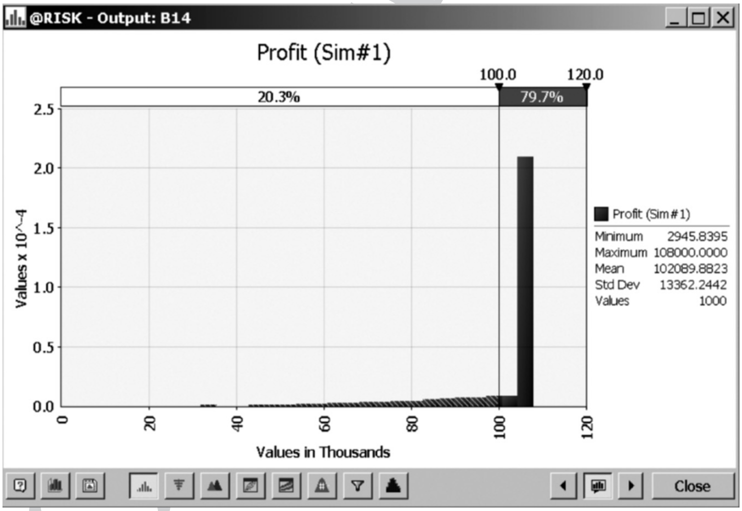
@RISK - Output: B14

Profit (Sim#1)

Profit (Sim#1)	
Minimum	2945.8395
Maximum	108000.0000
Mean	102089.8823
Std Dev	13362.2442
Values	1000

EXHIBIT B.8 Simulation results at the end of a simulation run.

less than a certain number, or greater than a certain number. For example, Exhibit B.9 illustrates that in Simulation #1, when the order quantity was 2,700, 20.3% of the scenarios (i.e., 203 out of the 1,000 generated scenarios) resulted in profits of less than $100,000. These graphs can be copied and pasted into other documents.


@RISK - Output: B14

Profit (Sim#1)

Profit (Sim#1)	
Minimum	2945.8395
Maximum	108000.0000
Mean	102089.8823
Std Dev	13362.2442
Values	1000

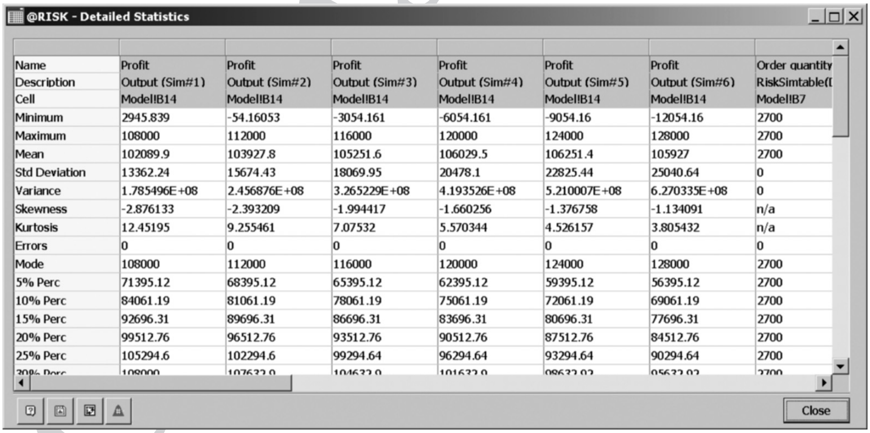
EXHIBIT B.9 Determining percentage of generated scenarios that satisfy a particular condition from the simulation.

To generate the summary statistics of the simulation, click the **Simulation Statistics** in the Results group on the @RISK tab. The output we obtained in this example is shown in Exhibit B.9. (Your numbers would be slightly different, because @RISK will generate different random scenarios every time it runs.) We can use this output to compare the different order quantities. For example, the highest expected (mean) profit of $106,251.40 is obtained in Simulation #5, which was for order quantity of 3,200 (the fifth value in the `RiskSimtable`). The standard deviation of the profit for this order quantity, however, is also quite high relative to the other options – $22,825.44. See Chapter 4 for more examples of useful observations from the simulation statistics.

If we want see the actual scenarios that were generated by @RISK, we click **Simulation Data** in the Results group on the @RISK tab. This feature will be useful when we discuss evaluating portfolio conditional value-at-risk in Chapter 8.

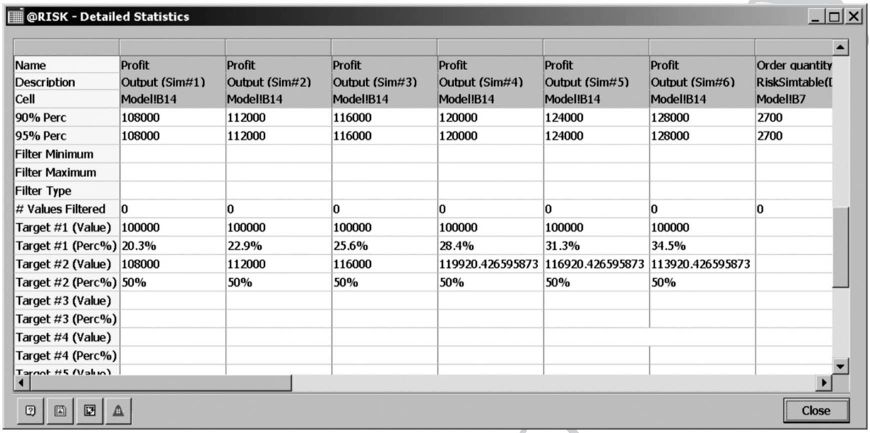
Finally, it is often helpful to generate @RISK reports directly in Excel worksheet format. This can be done by clicking on the button **Excel Reports** in the @RISK tab, and checking off the reports we would like to obtain.

@RISK has many other features that make it easy to create and analyze simulation models; we will encounter some additional ones in this book. For example, if we scroll down the **Summary Statistics** output window (Exhibit B.10), we see a row titled "Target." By entering value (such as 100,000), we can find the percentage of generated values from the simulation that was less than that value. In this case, (see Exhibit B.11), 20.3% of Simulation #1



Name	Profit	Profit	Profit	Profit	Profit	Profit	Order quantity
Description	Output (Sim#1)	Output (Sim#2)	Output (Sim#3)	Output (Sim#4)	Output (Sim#5)	Output (Sim#6)	RiskSimtable([
Cell	Model!B14	Model!B14	Model!B14	Model!B14	Model!B14	Model!B14	Model!B7
Minimum	2945.839	-54.16053	-3054.161	-6054.161	-9054.16	-12054.16	2700
Maximum	108000	112000	116000	120000	124000	128000	2700
Mean	102089.9	103927.8	105251.6	106029.5	106251.4	105927	2700
Std Deviation	13362.24	15674.43	18069.95	20478.1	22825.44	25040.64	0
Variance	1.785496E+08	2.456876E+08	3.265229E+08	4.193526E+08	5.210007E+08	6.270335E+08	0
Skewness	-2.876133	-2.393209	-1.994417	-1.660256	-1.376758	-1.134091	n/a
Kurtosis	12.45195	9.255461	7.07532	5.570344	4.526157	3.805432	n/a
Errors	0	0	0	0	0	0	0
Mode	108000	112000	116000	120000	124000	128000	2700
5% Perc	71395.12	68395.12	65395.12	62395.12	59395.12	56395.12	2700
10% Perc	84061.19	81061.19	78061.19	75061.19	72061.19	69061.19	2700
15% Perc	92696.31	89696.31	86696.31	83696.31	80696.31	77696.31	2700
20% Perc	99512.76	96512.76	93512.76	90512.76	87512.76	84512.76	2700
25% Perc	105294.6	102294.6	99294.64	96294.64	93294.64	90294.64	2700
30% Perc	108000	107632.9	104632.9	101632.9	98632.92	95632.92	2700

EXHIBIT B.10 Summary statistics output from simulation.



Name	Profit	Profit	Profit	Profit	Profit	Profit	Order quantity
Description	Output (Sim#1)	Output (Sim#2)	Output (Sim#3)	Output (Sim#4)	Output (Sim#5)	Output (Sim#6)	RiskSimtable(I
Cell	Model!B14	Model!B14	Model!B14	Model!B14	Model!B14	Model!B14	Model!B7
90% Perc	108000	112000	116000	120000	124000	128000	2700
95% Perc	108000	112000	116000	120000	124000	128000	2700
Filter Minimum							
Filter Maximum							
Filter Type							
# Values Filtered	0	0	0	0	0	0	0
Target #1 (Value)	100000	100000	100000	100000	100000	100000	
Target #1 (Perc%)	20.3%	22.9%	25.6%	28.4%	31.3%	34.5%	
Target #2 (Value)	108000	112000	116000	119920.426595873	116920.426595873	113920.426595873	
Target #2 (Perc%)	50%	50%	50%	50%	50%	50%	
Target #3 (Value)							
Target #3 (Perc%)							
Target #4 (Value)							
Target #4 (Perc%)							
Target #5 (Value)							

EXHIBIT B.11 Target function in the @RISK summary statistics window.

values for profit were less than $100,000, 22.9% of Simulation #2 values for profit were less than $100,000, and so on. Similarly, we can enter a percentage, such as 50% (see Exhibit B.11), and obtain the value from the simulation so that 50% of all other values are less than that value. In this case, 50% of profit values for Simulation #1 were less than $108,000, 50% of profit values for Simulation #2 were less than $112,000, and so on.

NOTES

1. See Chapter 3 for an introduction to random variables and probability distributions.
2. In this example, we have only one uncertainty—the demand, so we have only one input distribution.
3. See discussion in section 4.4.5 of Chapter 4.
4. See section 3.6 of Chapter 3.
5. Tornado graphs are discussed in detail in Chapter 17.

APPENDIX C

Introduction to MATLAB

MATLAB is an interactive computing environment for model development that also enables data visualization, data analysis, and numerical simulation.[1] The core of the MATLAB environment was created as a number-array-oriented programming language, that is, a programming language in which vectors and matrices are the basic data structures. In fact, MATLAB stands for "Matrix Laboratory."[2] Operations involving matrices and vectors can be performed efficiently within the core MATLAB software product. More specialized operations, such as statistical data analysis, optimization, and simulation, can be accessed by purchasing some of MATLAB's specialized toolboxes. Once a toolbox is installed, functions from the toolbox can be called in the same way as standard MATLAB functions, that is, without any special additional syntax. The toolboxes we use in this book are:

- *Statistics Toolbox.* Contains data analysis tools (for multivariate analysis, statistical tests, and statistical plots), random number generation tools, and quasi-random number generation tools, among others.
- *Optimization Toolbox.* Contains solvers for linear, quadratic, nonlinear, and binary optimization.

Other useful toolboxes for financial modeling include:

- *ExcelLink.* Enables the manipulation of Microsoft Excel worksheets from within MATLAB and using MATLAB functions from within Excel.
- *Genetic Algorithms and Direct Search Toolbox.* Contains randomized search optimization subroutines that can be used for solving complex (e.g., mixed integer) optimization problems to near optimality. To use it, one needs to own the Optimization Toolbox as well.

- *Financial Toolbox*. Contains specialized routines for computing frequently used financial quantities, such as present and future value, basic portfolio optimization, term structure of interest rates, bond prices, and derivative prices. It also contains functions that help with the manipulation of typical financial data sets, such as multivariate regression with missing data. Many of these routines can be implemented by using standard MATLAB functions, but the Financial Toolbox puts them together in a convenient package.

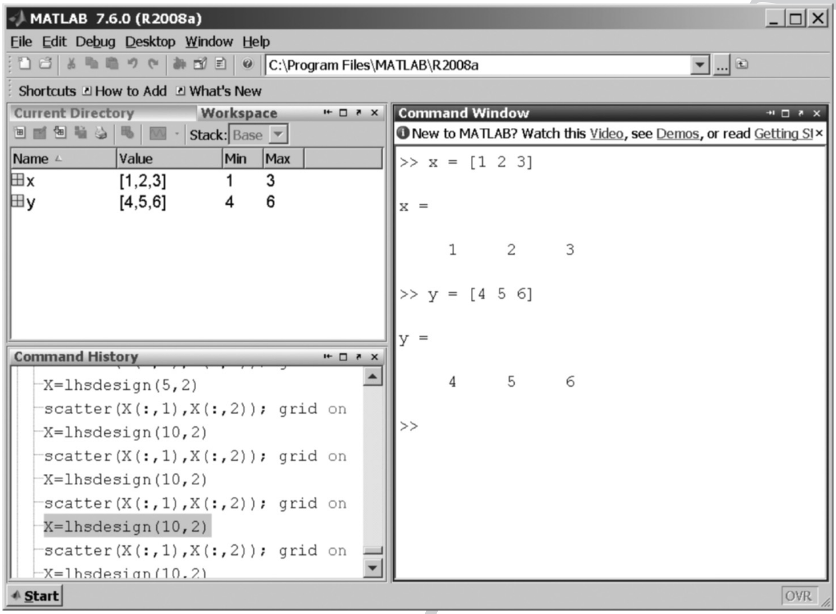
This appendix provides only brief pointers to important aspects of modeling in MATLAB. The actual MATLAB code for different applications discussed in the book is provided at the end of every chapter, and you will find the MATLAB manual and online help useful. Detailed documentation is also provided in MATLAB itself. For example, entering `help` at the prompt in MATLAB lists all major topics. Entering `help` and the name of the desired function at the prompt or in the box in the Help dialog box to access the documentation on that function in MATLAB. When you are unsure of which help topic is relevant for what you would like to accomplish, click on the button with question mark (?) in MATLAB's top menu. It will provide richer search options.

C.1 MATLAB DESKTOP AND EDITOR

The standard MATLAB desktop window contains a **Workspace** window, a **Command History** window, and a **Command** window (see Exhibit C.1). Depending on how you customize the MATLAB desktop window, however, you may see more or fewer windows. To check which windows are currently displayed and view other options, click on **Desktop** in the top MATLAB desktop window menu.

MATLAB commands are entered in the **Command** window. When series of commands need to be given, it is more convenient to list them into an "M-file," which is basically a file with instructions that MATLAB executes sequentially. Such files (scripts) are saved with the file extension .m and can be called from the prompt in the Command window entering their name (without the file extension .m). For example, if you create a file **OptimizePortfolio.m** with instructions on how to perform optimal portfolio allocation, you can call that file from the MATLAB command prompt by entering

```
>> OptimizePortfolio
```


MATLAB 7.6.0 (R2008a)
File Edit Debug Desktop Window Help
C:\Program Files\MATLAB\R2008a
Shortcuts How to Add What's New
Current Directory
Workspace
Stack: Base
Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting St
Command History
Start
OVR

Name	Value	Min	Max
x	[1,2,3]	1	3
y	[4,5,6]	4	6

```
>> x = [1 2 3]

x =

     1     2     3

>> y = [4 5 6]

y =

     4     5     6

>>
```

```
X=lhsdesign(5,2)
scatter(X(:,1),X(:,2)); grid on
X=lhsdesign(10,2)
scatter(X(:,1),X(:,2)); grid on
X=lhsdesign(10,2)
scatter(X(:,1),X(:,2)); grid on
X=lhsdesign(10,2)
scatter(X(:,1),X(:,2)); grid on
X=lhsdesign(10,2)
```

EXHIBIT C.1 Standard MATLAB desktop.

(If the file is saved in a directory other than the default MATLAB directory, you will need to make sure that MATLAB can find the file. Click **Desktop** and select **Current Directory** from the top menu, and navigate to the correct directory before entering the command at the prompt.)

To create an M-file, you can use any text editing program, such as MATLAB's own editor, WordPad, NotePad, and the open source editor Emacs. In general, it is convenient to use an editor that recognizes the MATLAB file extension and provides helpful highlighting for parts of the code that have different characteristics (for examples, comments in the code appear colored differently from commands). MATLAB's own editor can do that, and Emacs can be set up to recognize the MATLAB file format as well.

To call MATLAB's editor in order to create or edit M-files, click **Desktop** and select **Editor** from the top menu. Alternatively, you can use the shortcut buttons at the top of the MATLAB desktop window: the 🗋 button to open the MATLAB editor to write a new file, or the 📂 button to open a file that has already been created.

C.2 BASIC OPERATIONS AND MATRIX ARRAY CONSTRUCTION

C.2.1 Basic Mathematical Operations

MATLAB can perform many kinds of different mathematical operations, such as addition (+), multiplication (* or .*), square root (`sqrt` or `sqrtm`), and power (^). The commands can be entered at the command prompt, e.g., by typing

```
>> 3*sqrt(4) + 15
```

and pressing **Enter** on the keyboard produces the output

```
ans =
    21
```

To suppress output, use the semicolon (;). For example, typing

```
 >> 3*sqrt(4) + 15;
```

does not result in any visible output in the command window. However, MATLAB still performs the calculation. To see this, let us assign the value of the above expression to a `variable, ExpressionValue`:

```
>> ExpressionValue = 3*sqrt(4) + 15;
```

Then, after typing `ExpressionValue` at the command prompt, you get

```
>> ExpressionValue
ExpressionValue =
    21
```

C.2.2 Constructing Vectors and Matrices

As mentioned earlier, MATLAB's core data structures are vectors and matrices. For example, the command

```
>> x = [2 3 4 6]
```

produces a horizontal array (one row) that contains the numbers 2, 3, 4 and 6. The semicolon (;) is used to create new rows. To create a vertical array with the same entries, you can enter

```
>> x = [2; 3; 4; 6]
```

or press **Enter** after entering each number. (MATLAB treats semicolon and carriage return in array declarations as new lines.) The different syntax is useful depending on the source for downloading the data that populate the arrays.

Matrices are declared similarly. For example, a 2 × 2 matrix `X` can be specified as

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

MATLAB is case-sensitive; that is, it will treat the matrix `X` and the vector `x` defined earlier as separate variables.

Special commands exist for declaring types of matrices that are used often. For example,

```
>> I = eye(3,3)
I =
     1     0     0
     0     1     0
     0     0     1
```

produces a 3 × 3 identity matrix.

Similarly, the commands `ones(n,m)` and `zeros(n,m)` can be used to declare matrices that contain only 0's or 1's of the desired dimension (`n` × `m`), and `diag(x)` can be used to create a matrix that has a vector x as its diagonal elements, and 0s everywhere else.

You can also "stack" matrices and vectors. For example,

```
>> Y =[x; X]
Y =
     2     3     4     6
     1     2     3     4
     5     6     7     8
```

C.2.3 Basic Array Operations

To transpose an array A, use the command `transpose(A)` or `A′`. This operation converts a horizontal vector into a vertical one and vice versa, and flips the elements of a matrix which contains real numbers in its entries around the diagonal, keeping the diagonal entries the same.

For example,

```
>> X′
ans =
     1     5
     2     6
     3     7
     4     8
```

To multiply two arrays, you can simply use the multiplication command *. Since * performs a matrix multiplication, you need to make sure that the matrix dimensions agree. For example, an error results in the case when the 1 × 4 array `x` is multiplied by the 2 × 4 array `X`:

```
>> x*X
??? Error using ==> mtimes
Inner matrix dimensions must agree..
```

To multiply `x` and `X` correctly, you can enter

```
>> x*X′
ans =
    44   104
```

If you need to perform an element-by-element multiplication of two arrays (of equal sizes), use the .* operator. For example,

```
>> X.*X
ans =
     1     4     9    16
    25    36    49    64
```

Note that this is different from the matrix product. The matrix product would produce the following result:

```
>> X′*X
```

```
ans =
    26    32    38    44
    32    40    48    56
    38    48    58    68
    44    56    68    80
```

In general, adding a dot (.) before a matrix operation symbol applies the operation elementwise.

As explained in Appendix A, if you multiply a matrix array by a number, all of the array's entries get multiplied by that number. Similarly, in MATLAB, if you add a number to a matrix array, the number will be added to all of the elements of the matrix. For example,

```
>> 10+X
ans =
    11    12    13    14
    15    16    17    18
```

C.2.4 Extracting Information from Arrays

Suppose we have created a matrix array `Data` with financial data on annual stock returns over 10 years for 1,000 companies traded on the New York Stock Exchange, and would like to check the entry for the return on stock 253 in year 7. We are dealing with a 10 × 1,000 matrix array in which each row is a time period, and each column contains the returns on a particular stock. We are looking for the element in row 7, column 253 of this array. This can be requested with the command `Data(7,253)`.

Suppose now that we would like to extract information on all of stock 253's returns over the 10 years. This means that we are looking for the elements of column 253 of the matrix array. This can be requested with the command `Data(:,253)`. Similarly, if we would like to request all elements in the same row (e.g., the returns on all stocks in year 7), we can use the colon operator again: `Data(7,:)`.

To illustrate the output, let us use the matrix array `X`. To find out what the value of the element in row 1, column 3 is, we enter

```
>> X(1,3)
ans =
     3
```

The third column of X is

```
>> X(:,3)
ans =
     3
     7
```

The second row of X can be obtained as

```
>> X(2,:)
ans =
     5     6     7     8
```

C.3 IMPORTANT MATLAB FUNCTIONS

MATLAB supports a number of built-in functions. A function is written as a command, and takes arguments as inputs in parentheses. It processes the inputs by using operations hidden from the user, and passes the final results back to the user. While we cannot cover many of the MATLAB functions in this brief introduction, we illustrate how functions work with an example of the function `find` which can be useful in many situations.

`Find` takes in an array and a condition as arguments, and returns the indices of elements within the array that satisfy the condition. In addition to traditional applications, `find` can be very helpful when dealing with missing data, which happens often with financial time series.

Recall that Y was the matrix array obtained by stacking x and X. Suppose we want to find the indices of the elements in the array that are less than 5. We obtain

```
>> ind = find(Y<5)
ind =
     1
     2
     4
     5
     7
     8
    11
```

Note that MATLAB treated the matrix array as a stacked up collection of column vectors, and the indices in the array `ind` correspond to the index of the element in that long column vector. Obtaining the actual *values* of `Y` that correspond to these indices can be accomplished by entering

```
>> Y(ind)
ans =
     2
     1
     3
     2
     4
     3
     4
```

The indexing of an array as a sequence of stacked columns works well if the array is a vector, but can get confusing if the array is a matrix. In the latter case, we may want to obtain the indices as a row and column index. For example,

```
>> [indRow,indCol] = find(Y<5)
indRow =
     1
     2
     1
     2
     1
     2
     2
indCol =
     1
     1
     2
     2
     3
     3
     4
```

This means that the following elements of Y have values less than 5: (row 1, column 1), (row 2, column 1), (row 1, column 2), etc. Unfortunately, looking up the actual values of Y as `Y(indRow,indCol)` does not work.

C.4 CREATING USER-DEFINED FUNCTIONS

The compactness of the function syntax makes functions desirable when a user needs to call a certain sequence of commands often. Many examples of such applications are in the book. For example, the Black-Scholes formula for pricing European options, covered in Chapter 13, takes a number of steps to compute. It is convenient to have a function that returns one value—the option price—to the user after the user inputs values of factors that determine that price, such as the strike price, the time to maturity, the volatility, and the like.

Functions need to be written in M-files. While general script M-files can contain any sequence of instructions that will be completed when the name of the file is entered at the MATLAB prompt, however, *function* M-files need to start with a specific first line. That line contains the word "function" and a declaration of the function name, inputs and outputs. The function name and the name of the M-file should be the same.

It turns out that the Black-Scholes formula already exists in the Financial Toolbox, so it is convenient to see how the price is computed and discuss important aspects of writing user-defined functions. (We have skipped some lines in the code and replaced them with "..." for the sake of brevity.) Users can view the source code for some of the advanced MATLAB functions in the toolboxes by entering `type` and the desired function name at the prompt.[3]

```
>> type blsprice
function [call,put] = blsprice(S, X, r, T, sig, q)
%BLSPRICE Black-Scholes put and call option pricing.
%   Compute European put and call option prices using a
%   Black-Scholes model.
%
%   [Call,Put] = blsprice(Price, Strike, Rate, Time,
%   Volatility)
%   [Call,Put] = blsprice(Price, Strike, Rate, Time,
%   Volatility, Yield)
%
%   Optional Input: Yield
%
%   Inputs:
%   Price       - Current price of the underlying asset.
%
%   Strike      - Strike (i.e., exercise) price of the
%                 option.
```

```
%
%   Rate        - Annualized continuously compounded risk-
%                 free rate of return over the life of the
%                 option, expressed as a positive decimal
%                 number.
%
%   Time        - Time to expiration of the option,
%                 expressed in years.
%
%   Volatility  - Annualized asset price volatility (i.e.,
%                 annualized standard deviation of the
%                 continuously compounded asset return),
%                 expressed as a positive decimal number.
%
%   Optional Input:
%   Yield       - Annualized continuously compounded yield
%                 of the underlying asset over the life of
%                 the option, expressed as a decimal number.
%                 If Yield is empty or missing. the default
%                 value is zero.
%
%                 For example, this could represent the
%                 dividend yield (annual dividend rate
%                 expressed as a percentage of the price
%                 of the security) or foreign risk-free
%                 interest rate for options written on stock
%                 indices and currencies, respectively.
%
%   Outputs:
%   Call        - Price (i.e., value) of a European call
%                 option.
%
%   Put         - Price (i.e., value) of a European put
%                 option.
...
%
%   See also BLSIMPV, BLSDELTA, BLSGAMMA, BLSLAMBDA,
%    BLSTHETA, BLSRHO.
% Copyright 1995-2005 The MathWorks, Inc.
% $Revision: 1.8.2.5 $   $Date: 2005/09/18 16:19:06 $
...
```

```
if nargin < 5
  error('Finance:blsprice:InsufficientInputs', ...
     'Specify Price, Strike, Rate, Time, and Volatility.')
end
if (nargin < 6) || isempty(q)
    q = zeros(size(S));
end
message = blscheck('blsprice', S, X, r, T, sig, q);
error(message);
...
d1 = log(S(i)./X(i)) + (r(i) - q(i) + sig(i).^2/2) .* T(i);
d1 = d1 ./(sig(i).*sqrt(T(i)));
d2 = d1 - (sig(i).*sqrt(T(i)));
d1(isnan(d1)) = 0;
d2(isnan(d2)) = 0;
call(i) = S(i) .* exp(-q(i).*T(i)) .* normcdf( d1) - ...
    X(i) .* exp(-r(i).*T(i)) .* normcdf( d2);
put (i) = X(i) .* exp(-r(i).*T(i)) .* normcdf(-d2) - ...
    S(i) .* exp(-q(i).*T(i)) .* normcdf(-d1);
...
% Reshape the outputs for the user.
%
call = reshape(call, nRows, nCols);
put  = reshape(put , nRows, nCols);
```

Some aspects of this function are complicated for a beginner, but a review of the function syntax helps create a list of useful pointers to which you can refer when creating your own functions:

- The first line contains the word function followed by a specification of the outputs of the function (in this case, [call,put]). (Note that a function can have more than one output.) After calling the function, MATLAB computes the values for the outputs, and the variable call will contain the price of a European call option, while the variable put will contain the price of a European put option. Next, we have an equal sign followed by the name of the function (blsprice) and the arguments for the function (S for current stock price, X for strike price, r for rate of return, T for time to maturity, sig for volatility, yield for continuous dividend yield (the last argument is optional)).
- When the function is called with specific input values, you can assign the output to variables. For example,

```
>> [callOutput,putOutput] = blsprice(110,100,0.10,2,0.40)
callOutput =
   38.1757
putOutput =
   10.0488
```

- The names of the input variables need to participate in calculations in the function. For example, `S` appears as the current stock price in the first line (`function [call,put] = blsprice(S, X, r, T, sig, q)`), and this is the same variable that is used to store the value of the stock price in the computations. Similarly, the names of the output variables (`call` and `put`) should appear somewhere in the text of the function, and be assigned an expression which can then be returned to the user.
- Note the abundance of the percentage sign (%) in the function code. This sign is used for writing comments that are ignored by MATLAB when executing the code. It is always a good idea to comment abundantly in order to be able to re-trace your reasoning later. The first comment line is called "the H1 line," and is the line that is searched by the MATLAB built-in function `lookfor`. `Lookfor` searches all MATLAB files containing a keyword in their first line. (This is useful if you are not sure which function to use for a specific purpose, and would like to find the names of all functions that may be relevant.) Therefore, it is important to provide a meaningful description of your function in the first commented line. After the first line, you can continue with a more detailed description of the function, and list references.

C.5 CONTROL FLOW STATEMENTS

M-files, whether of a generic or function kind, can contain more advanced operations than matrix manipulation. We briefly review a couple of control flow statements that are often used in such files: the `for` loop and the `if` statement.

The general format of a `for` loop is

```
for n = array
        commands
end
```

The commands inside the `for` loop are executed once for every value in the *column* in the array (typically, the array is a vector of numbers, so the loop is executed once for every number). For example,

```
for n = 1:5
        v(n) = sqrt(n);
end
```

results in

```
v =
    1.0000    1.4142    1.7321    2.0000    2.2361
```

The array 1:5 is equivalent to [1 2 3 4 5]. MATLAB starts out with `n` = 1, computes its square root, and assigns it to `v(1)`. Then it keeps repeating the process until it has computed `v(5)` for `n` = 5.

Loops in MATLAB are often necessary, but as a general rule MATLAB is more efficient in array operations than in loops. For example, the same effect (adding 10 to each element of the vector `x`) can be achieved in two ways:

```
for n = 1:4
        x(n) = x(n) + 10;
end
```

and

```
>> x = x+10
```

Both of them result in

```
x =
    12    13    14    16
```

The second command would typically be completed faster; however, the difference in speed between the two approaches has been greatly reduced in the latest versions of MATLAB.

The `if` statement has the following general format:

```
if expression
        commands
end
```

The `commands` are completed only if all elements in the expression are true. A somewhat more complex `if` statement is

```
if expression1
        commands1
elseif expression2
        commands2
else expression3
        commands3
end
```

`Commands1` are completed if `expression1` is true. Only if `expression1` is not true, MATLAB moves on and checks if `expression2` is true. If `expression2` is true, `commands2` are completed. If `expression2` is not true either, MATLAB moves to `expression3`. If `expression2` is true, `commands3` are completed; otherwise MATLAB exits. The `elseif` or `else` commands are optional in `if` statements.

There are several other useful control flow statements, such as the `while` loop, `switch-case` constructions, and `try-catch` blocks. See the MATLAB manual and help for more detail.

C.6 GRAPHS

MATLAB is well-known for its beautiful graphing capabilities. The most common function for plotting two-dimensional (2-D) plots is `plot`.

For example, suppose we would like to plot the standard normal probability distribution. We will use the function `normpdf` (available from the Statistics Toolbox), which computes the PDF of a normal random variable.

The command

```
>> x = linspace(-6,6,100)
```

creates a vector `x` with 100 values, equally spaced between the minimum value -6 and the maximum value $+6$. (In reality, the normal distribution stretches from negative infinity to positive infinity, but it is highly unlikely that we will obtain realizations that are greater than 6 standard deviations away from the mean of 0.)

The command

```
>> y = normpdf(x)
```

computes the values of the normal probability distribution function for every value in the array for x.

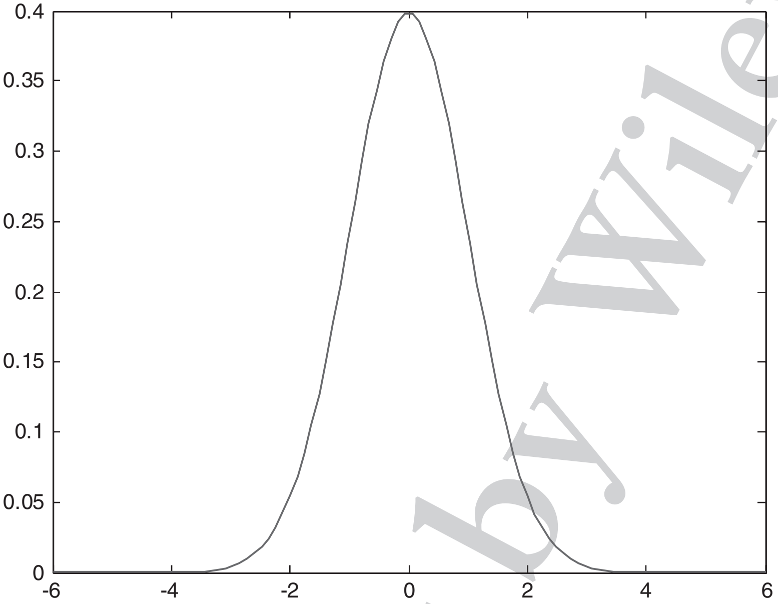


EXHIBIT C.2 Plot of the PDF of the normal distribution.

To plot x versus y, we use

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

The result is the graph in Exhibit C.2.

You can play with the options for the graph. For example,

```
>> plot(x,y,'r:p'); title('Normal PDF'); xlabel('x');
   ylabel('pdf')
```

plots the same graph as a red dotted line with a pentagram symbol, labels the x- and the y-axis, and creates a title for the graph (Exhibit C.3).

To plot multiple graphs on the same picture, use the command `hold on` before you start and `hold off` when you are done with the instructions. For example, suppose we would like to plot the standard normal distribution and a standard *t*-distribution with five degrees of freedom on the same graph in order to compare them. The following sequence of commands accomplishes this.

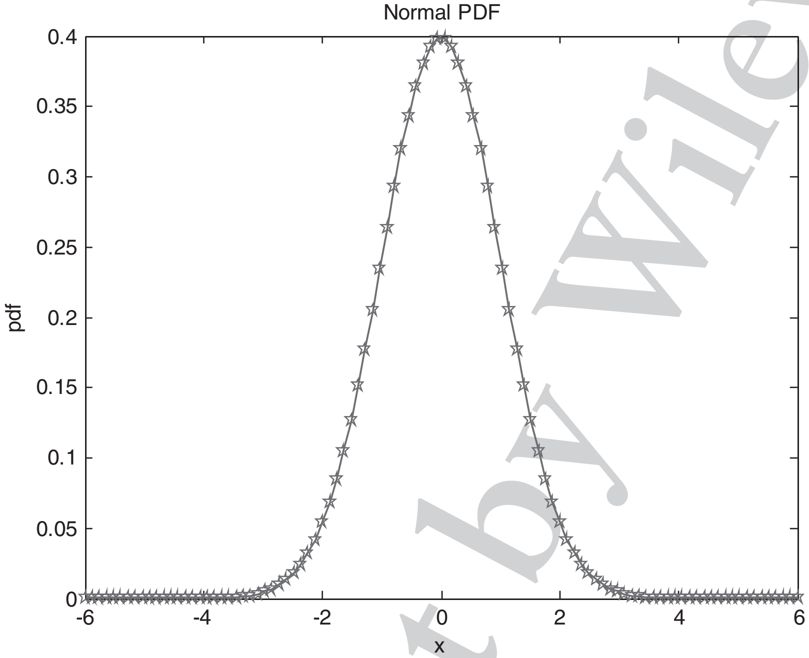


EXHIBIT C.3 Plot of the PDF of the normal distribution (with modified options).

First, we declare a variable that follows a *t*-distribution with five degrees of freedom:

```
>> t=tpdf(x,5);
```

Then, we plot the graph:

```
>> hold on
>> plot(x,y,'r:p'); xlabel('x'); ylabel('pdf')
>> plot(x,t);
>> title('Normal Versus T Distribution');
>> hold off
```

The results are displayed in Exhibit C.4.

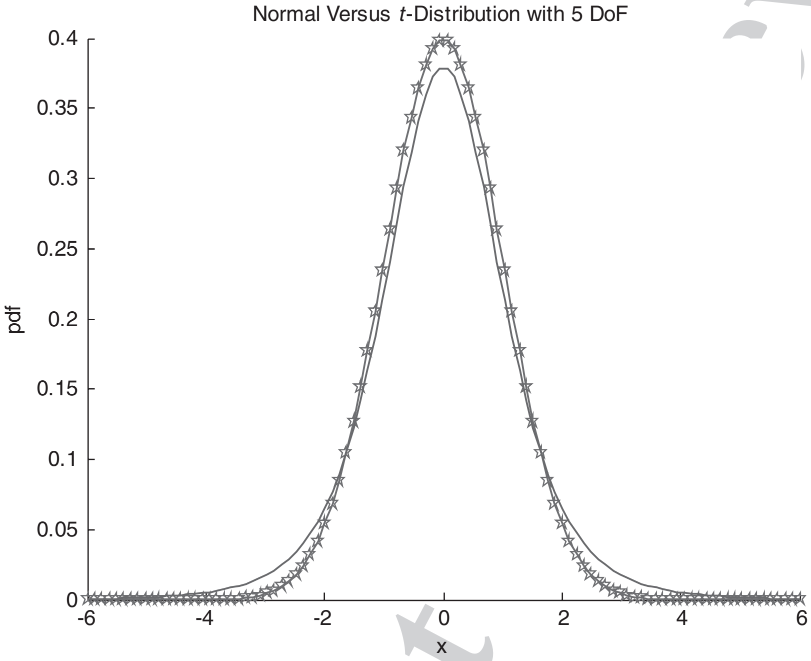


EXHIBIT C.4 Illustration of `hold on/hold off` effect.

Alternatively, you can list several pairs of variables to plot inside the plot function. For example,

```
>> plot(x,y,'r:p',x,t); xlabel('x'); ylabel('pdf')
>> legend('Normal PDF','T PDF')
>> title('Normal Versus T Distribution with 5 DoF');
```

This allows also for creating a legend (Exhibit C.5).

Legend, titles, and other graph attributes can be added and modified also after the basic `plot` command has been given and a graph window has popped up. To do that, click on the corresponding items in the top menu of the graph window.

Suppose now that we would like to plot the two PDFs side by side in the same figure. To graph several separate graphs in the same figure, use the command `subplot(number of rows, number of columns, index of graph within the graph array)`.

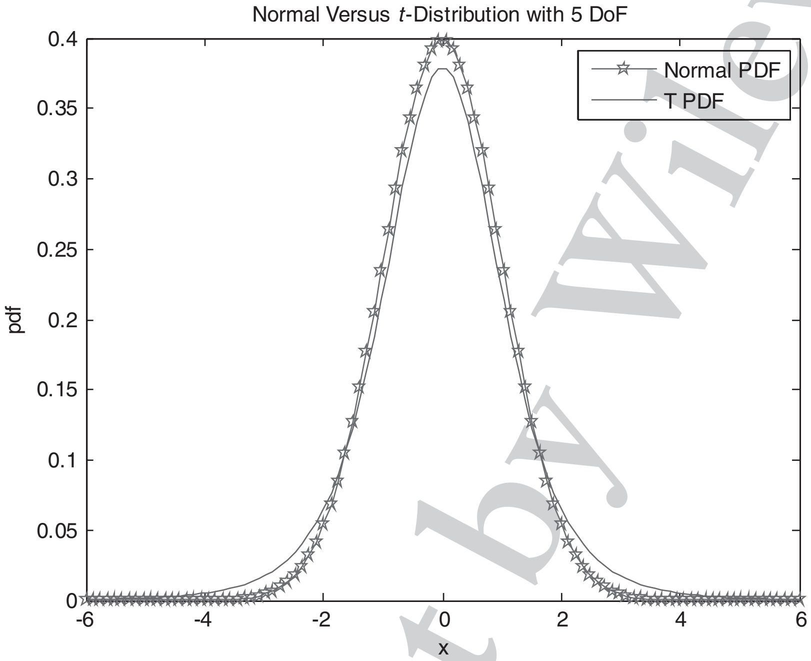


EXHIBIT C.5 Changing defaults and plotting multiple graphs with the `plot` function.

For example, the code

```
>> subplot(1,2,1), plot(x,y,'r:p'); xlabel('x');
   ylabel('pdf')
>> title('(a) Normal PDF')
>> subplot(1,2,2), plot(x,t); xlabel('x'); ylabel('pdf')
>> title('(b) T PDF')
```

produces the graph in Exhibit C.6.

Finally, we discuss three-dimensional (3-D) graphs. They can be created with commands like `plot3` and `surf`, and as a general matter are more complex.

The command `plot3(first variable x, second variable y, third variable z)` plots points in 3-D space whose three coordinates are given by the vectors or matrices `(x,y,z)` in the three arguments of the function. (Note that the arguments need to be arrays of equal sizes.)

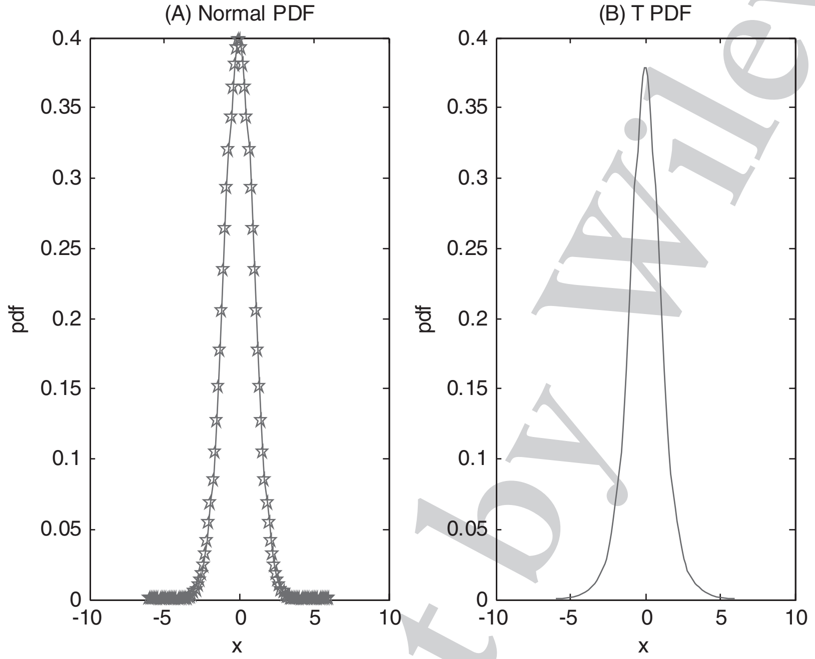


EXHIBIT C.6 Multiple plots within the same figure.

The command `surf(x, y, z)` plots a shaded surface using `z` as the height, and `(x, y)` as the vectors or matrices that define the other two dimensions of the surface. For applications in this book, we mainly need `x` and `y` to be vector arrays, in which case the number of rows for `z` should be the length of the vector array `y`, and the number of columns for `z` should be the length of the vector array `x`.

For example, suppose we would like to plot a multivariate normal distribution for two normal variables, x1 and x2, that have means of 0 and are correlated with covariance matrix `[0.25 0.3; 0.3 1]`. (Note that this notation means that the variance of x1 is 0.25 (the standard deviation of x1 is 0.5), the variance of x2 is 1 (the standard deviation of x2 is 1), and the covariance of x1 and x2 is 0.3.[4])

The multivariate normal distribution can be computed with the MATLAB function `mvnpdf(X,mu,Sigma)`. The arguments `mu` and `Sigma` are the vector array of average (expected) values for the normal random variables and their covariance matrix, respectively. In this case, we have `two` normal random variables, so `mu=[0 0]` and `Sigma = [0.25 0.3; 0.3 1]`.

The first argument in the function (matrix X) provides the points at which the function should be evaluated. The function is evaluated for every row of X, taking the elements in that row as the coordinates of the point at which the function should be evaluated. Therefore, since in our example we are looking at two normal random variables, there should be two columns of the matrix X. Note that we cannot simply provide two columns with, say, equally-spaced values for x1 and x2. If we do, MATLAB would pair each entry of x1 with the corresponding entry of x2, and will only use those combinations of coordinates, so the plot will look two-dimensional. The columns of X should provide a "grid." In other words, we cannot simply provide possible coordinates along each axis, and expect that MATLAB will know to take every combination of possible coordinates to obtain the points at which to plot the function. To create this "grid" of points, we need to go through a couple of steps.

First, we would use the function [X1,X2] = meshgrid(x1,x2). It creates two matrices. The number of *rows* in the first matrix, X, is the same as the number of elements in the vector y (that is, the number of rows equals length(y), another useful MATLAB command). Each row of the column X contains identical entries: the entries of the vector x. The matrix Y contains the same number of columns as the number of elements in the vector x, and each column contains an identical copy of the vector y. While perhaps difficult to imagine at first, X(i,j) and Y(i,j) cover all possible combinations of the elements of the original vectors, x and y.

The second step is to create the array [X(:), Y(:)]. The colon operator (:) has multiple uses, but in the context of being used as an argument for a matrix, it takes all entries of a matrix, column by column, and lists them as a vector array. Therefore, the array [X(:), Y(:)] would contain two columns with every possible combination of coordinates generated by the original list in the vector arrays x and y.

To summarize, here are the commands used to generate 30 points between −4 and 4 along each coordinate x1 and x2, then evaluate the multivariate normal PDF at each combination of coordinates:

```
>> x1 = linspace(-4,4,30); x2 = linspace(-4,4,30);
>> Sigma = [0.25 0.3; 0.3 1]; mu = [0 0];
>> [X1,X2] = meshgrid(x1,x2);
>> y = mvnpdf([X1(:),X2(:)],mu,Sigma);
```

The output of this sequence of commands is a vertical array of values that represent the multivariate normal PDF evaluated at each combination of coordinates. (If you skip the semicolon at the end of the last row with the function mvnpdf, you can see what the output looks like. You can also use the command `size(y)` to check the dimensions of `y`.) Now we would like to plot these values. We will use the `surf` function.

The tricky thing now is that the `surf` function's third argument, `z`, needs to be a matrix whose entries represent the values of the function to be plotted at each combination of coordinates. However, we obtained a vector of values for the PDF. We need to "re-shape" that vector back into a matrix. This can be done with the command `Y = reshape(y,m,n)`. The function `reshape` takes the array `y`, and goes through the elements of `y` column-wise. The first `m` elements of `y` become the first column of the new matrix `Y`, the next `m` elements of `y` become the second column of the `Y`, and so forth until `n` columns for `Y` are created. In this case, we would like to create `length(x1)` columns and `length(x2)` rows. (This may be a bit confusing, but, as we mentioned earlier, the function `surf` expects the third argument to be a matrix with number of columns equal to the size of the first argument, and number of rows equal to the size of the second argument.)

```
>> Y = reshape(y,length(x2),length(x1));
>> surf(x1,x2,Y);
>> title(’Multivariate Normal Probability Density’)
>> axis([-4 4 -4 4 0 0.4]);
>> xlabel(’x1’); ylabel(’x2’); zlabel(’PDF’);
```

The resulting graph is in Exhibit C.7.

C.7 IMPORTING DATA AND INTERACTING WITH SPREADSHEETS

MATLAB recognizes files with extensions .dat as data files. Such files should contain text structured in rows and columns. For example, suppose that the file returns.dat contains daily annual returns on the stocks traded in the NYSE for 10 years. The command

```
>> load returns.dat
```

imports the data in the file into a data structure—a matrix array with rows and columns that can then be referenced using some of the commands we described in section C.2.4.

Many financial companies build their infrastructure around Microsoft Excel. The MATLAB core product contains some useful functions for

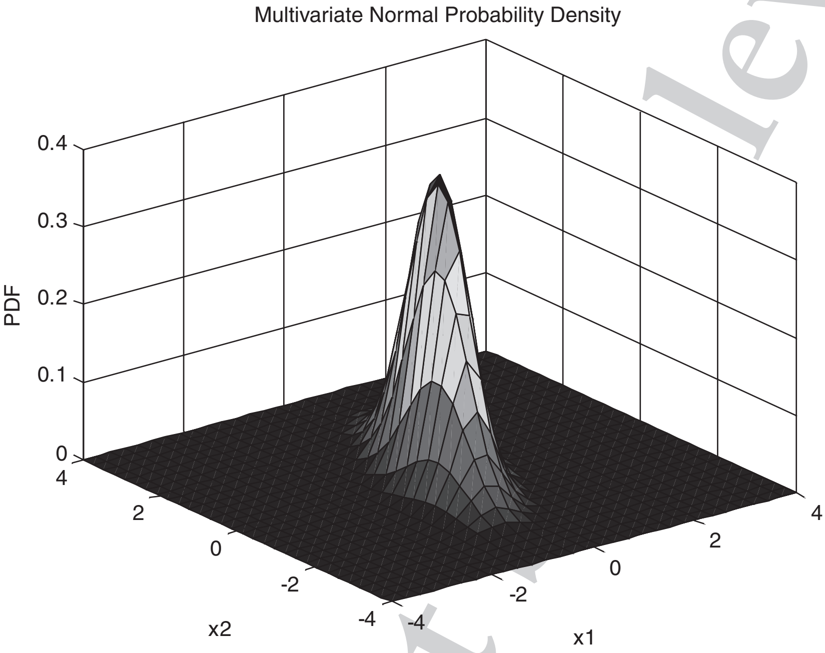


EXHIBIT C.7 Three-dimensional plot of a multivariate normal distribution.

importing Excel data and exporting MATLAB results to spreadsheets. The function

```
>> xlsread('fileName','sheetName','range')
```

allows the user to read into MATLAB the data stored in file `fileName`, worksheet `sheetName`, cells in range `range`. Instead of `range`, you can state an array name if you had named the array of cells in advance. Variations of this command exist, for example,

```
>> xlsread('fileName',-1)
```

allows the user to select the range in `fileName` directly, through interactive selection in Excel. Enter `help xlsread` at the MATLAB command prompt for further information.

The function

```
>> xlswrite('fileName',output,'sheetName','cell')
```

allows the user to export MATLAB results (`output`) to a worksheet (`sheetName`) in an Excel file (`fileName`). MATLAB preserves the dimensions on the output, and writes it to the spreadsheet starting at cell reference `cell`. For example, if `output` is a horizontal array of numbers, MATLAB will write the data in a row in the Excel file, starting at `cell`. MATLAB operations work within the `xlswrite` command. For example, you can transpose the output (`output'`) inside the parentheses of the `xlswrite` command if you desire different output formatting in the Excel spreadsheet.

More sophisticated capabilities exist through MATLAB's Excel Link. With Excel Link, you can call MATLAB's functions directly from within Excel, thus ensuring access to MATLAB's superior computational and graphical capabilities. Excel Link is purchased as a separate toolbox. It can then be made visible from within Excel by selecting it as one of Excel's Add-Ins. There are 11 commands that allow for communicating data back and forth between Excel and MATLAB. They all start with "ML." For example, `=MLAppendMatrix()` creates or appends a matrix in MATLAB with data from an Excel spreadsheet.

A word of caution: Excel Link formulas are not case sensitive. For example, `MLAppendMatrix` and `mlappendmatrix` are the same. However, MATLAB functions and variables called through these links are case sensitive. For example, `x` and `X` would still be treated as two separate variables.

NOTES

1. MATLAB is sold by the MathWorks, Inc., http://www.mathworks.com/.
2. See Appendix A at the companion web site for a quick review of vector and matrix arrays.
3. You can also use the command `edit` to view the code of most MATLAB functions. `Edit` opens the MATLAB editor directly, and lets users modify the code. However, we recommend that MATLAB beginners avoid using this command, because any modifications to the code become permanent in the user's copy of MATLAB.
4. See Chapter 3 for a definition of the multivariate normal distribution and a review of covariance and correlation.

APPENDIX D

Introduction to Visual Basic for Applications

This appendix is a brief introduction to Visual Basic for Applications (VBA), the programming language environment that allows Excel users to automate tasks, create their own functions, perform complex calculations, and interact with worksheets. It does not provide a comprehensive review of the capabilities of VBA, justsufficient background to understand the VBA code and exercises at the end of selected chapter. For further information, see Walkenbach (2004) and Roman (2002). The Excel VBA help is also useful as a quick reference. All Excel commands in this introduction to VBA are for Microsoft Office 2007.[1]

Before reviewing some important characteristics of the VBA language, let us create a simple example of a VBA program. Excel has a tool for recording worksheet tasks that can be replayed as a *macro*. The idea of creating macros in Excel is similar to the idea of creating M-file scripts in MATLAB. It serves to record a sequence of commands so that you do not have to repeat the same set of instructions if you need to perform the task several times. Macros are in effect computer programs whose commands are hidden from the user, but can be seen if you open the VBA editor (VBE). You can access the VBE by using a keyboard shortcut, **Alt-F11**, in all versions of Excel. In Excel 2007, VBE can be accessed from the Developer tab. If the Developer tab is not visible, do the following to set it up: Click the main Microsoft Excel button, then Excel Options. In the **Popular Options** pane, click the **Show Developer Tab in Ribbon** option. Then click the **Developer** tab on Excels ribbon, and then click **Visual Basic** in the Code group to open the editor (see Exhibit D.1).

Use the **Macro Security** command to enable macros. (It is always a good idea to return to the default—disabled macros—after you are finished working with macros.)

Simulation and Optimization in Finance by Dessislava A. Pachamanova and Frank J. Fabozzi.

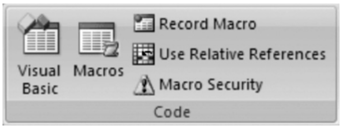
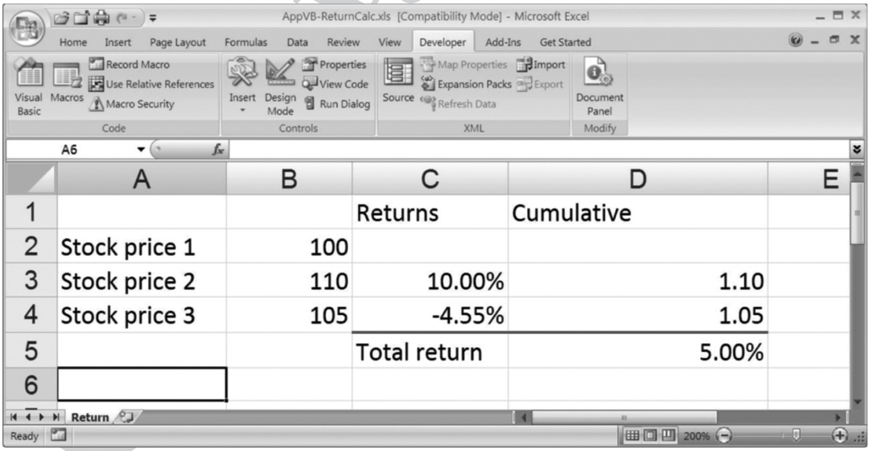


EXHIBIT D.1 The Code group under the Developer tab in Excel 2007 (note the Visual Basic command).

Open a new file. We are trying to create the layout shown in Exhibit D.2. First, enter the text in column A and B, that is, enter stock prices for three points in time. Suppose we want to be able to compute the realized cumulative return over the two time periods for any set of three stock prices in column B. We can do that by, for example, computing the realized returns over each of the two periods in column C, and then computing the cumulative return between time 1 and 3 in cell D5. Let us record the entries and the calculations as a macro. To record a macro, click on **Record Macro** in the Code group under the Developer tab. Delete the default name `Macro 1`, and replace it with something more meaningful,such as `ReturnCalc`. Click **OK**. Once the macro recorder is on, do the following:

1. Enter `=(B3-B2)/B2` in cell C3 (this will compute the return for time period 1 and 2).
2. With the cursor in cell C3, press **Ctrl-C** to copy the contents of cell C3, more the cursor to cell C4, and press **Ctrl-V** to paste. This fills cells C4 with the formula for computing the return between time 2 and 3.



	A	B	C	D	E
1			Returns	Cumulative	
2	Stock price 1	100			
3	Stock price 2	110	10.00%	1.10	
4	Stock price 3	105	-4.55%	1.05	
5			Total return	5.00%	
6					

EXHIBIT D.2 Macro recording example.

3. Select and right-click cells C3:C4 and then select from the shortcut menu **Format Cells > Number > Percentage > Decimal Points 2** to format the returns as percentages.
4. In D3 enter `=(1+C3)`. Then right-click cell D3 and from the shortcut menu select **Format Cells > Number > Number > Decimal Points** 2 to format the contents of the cell as a number.
5. Click cell D4, enter `=D3*(1+C4)`.
6. Type `Total Return` in cell C7.
7. Enter `=D6-1` in cell D7 to compute the total return over the five periods.
8. Select and right-click cells C7:D7. Then select from the shortcut menu **Format Cells > Border**. Select the double-line, and then click the upper line of the cell in the Border window to make the double-line appear. Click **OK**.
9. Click the stop button in the macro recorder to stop recording.

Now let us see what the macro does. You can use your own file, or the file **D-ReturnCalc.xlsm**, which contains the complete exercise above. Delete all contents from the array C3:D5. Press **Alt-F8** or, under the **Developer** tab, click **Macros** in the Code group . Select `ReturnCalc`, press **OK**. The worksheet should fill up with the entries that you entered before. If you had changed the value of the stock price in any of the three cells in column B, the macro should calculate the correct corresponding value for total return.

Behind the scenes, Excel recorded VBA code with instructions that tell Excel what functions to perform when you run the macro. You can see these instructions by opening the VBA editor, and clicking on **Modules > Module 1**. The instructions look like this:

```
Sub ReturnCalc()
'
' ReturnCalc Macro
' Macro recorded month/day/year by you
'

'
   Range("C3").Select
   ActiveCell.FormulaR1C1 = "=(RC[-1]-R[-1]C[-1])/R[-1]C[-1]"
   Range("C3").Select
   Selection.Copy
   Range("C4").Select
   ActiveSheet.Paste
   Range("C3:C4").Select
   Selection.NumberFormat = "0.00%"
   Range("D3").Select
   ActiveCell.FormulaR1C1 = "=1+RC[-1]"
   Range("D3").Select
   Selection.NumberFormat = "0.00"
   Range("D4").Select
   ActiveCell.FormulaR1C1 = "=R[-1]C*(1+RC[-1])"
   Range("C5").Select
   ActiveCell.FormulaR1C1 = "Total return"
   Range("D5").Select
   ActiveCell.FormulaR1C1 = "=R[-1]C-1"
   Range("D5").Select
   Selection.Style = "Percent"
   Selection.NumberFormat = "0.00%"
   Range("C5:D5").Select
   Selection.Borders(xlDiagonalDown).LineStyle = xlNone
   Selection.Borders(xlDiagonalUp).LineStyle = xlNone
   Selection.Borders(xlEdgeLeft).LineStyle = xlNone
   With Selection.Borders(xlEdgeTop)
         .LineStyle = xlDouble
         .Weight = xlThick
         .ColorIndex = xlAutomatic
   End With
   Selection.Borders(xlEdgeBottom).LineStyle = xlNone
   Selection.Borders(xlEdgeRight).LineStyle = xlNone
   Selection.Borders(xlInsideVertical).LineStyle = xlNone
   Range("D5").Select
End Sub
```

Knowing the actions to create the macro, it is relatively straightforward to trace what the program is doing at every step. To understand how the macro works better, however, and to know how to create such scripts without recording them in the worksheet, we need to understand some basic facts about VBA.

D.1 OBJECTS, PROPERTIES, AND METHODS

The most important fact about VBA is that it tries to act as an object-oriented language.[2] This means that it treats every component of Excel, such as a worksheet, a cell, a range of cells, and a chart, as an *object*. Objects are arranged in a hierarchy, and have *properties* (attributes) that can be modified by entering the name of the object followed by dot and a specific command. In addition, objects are associated with *actions* (methods) that the objects can perform or have applied to them. You can view all objects by selecting **View > Object Browser** from the menu bar in the VBE window. In Excel 2007, you can also view a detailed list of objects, their properties, and their methods by clicking on **Help** (pressing **F1**) and selecting **Excel Object Model Reference**.

The largest object, the object on top of the hierarchy, is Excel itself. It is the Application object. Worksheets, ranges, selections, charts, etc. are all objects that are lower in the hierarchy. Objects in the same class are organized in *collections*. For instance, the Workbooks collection contains all workbooks (i.e., Excel files) that are currently open. Similarly, the Worksheets collection contains all Excel worksheets in the files that are currently open, the Sheets collection contains all Excel worksheets and charts in the files that are currently open, etc. Thus, for example, to reference cell C3 in Worksheet Return in file (Workbook) **D-ReturnCalc.xlsm**, you would type

```
Application.Workbooks("D-ReturnCalc.xlsm").Sheets("Return").
   Range("C3")
```

This reference is rather long and, as we can see from the actual VBA code, it is not necessary, as long as the macro is saved within the active Excel workbook, and the identification of the cell range that is referenced is unique. In our example, Range(``C3'') is sufficient to reference cell C3, because the objects higher in the hierarchy, such as the name of the worksheet and the name of the file, are implied in the reference.

An example of an action (method) that can be performed on an object is the command Select. The Select method applies to several objects, including Worksheet, Chart, and Range. Notice that it was used often in the macro we created, because clicking on a cell or selecting an array performed the action. For example, in line 14, we selected the range C3:C4. Similarly, in line 10, we selected the cell C3 with the command]

```
Range("C3").Select
```

Then, the Selection property of an object in the background (the Window object) was used to return a Range object (representing the selected range on the worksheet) on which we can apply other methods, such as Copy (line 11 of the code):

```
Selection.Copy
```

VBA usually suggests actions and properties that can be used with an object, so you can select from a list.

Another example of modifying the properties of the object is in lines 14 and 15 of the VBA code. They request that the format of the cell range C3:C4 be changed to percentage with two digits after the decimal point. Namely, we selected the range C3:C4, and the `NumberFormat` property of the `Range` object that was returned by the `Selection` property was set to percentage with two digits after the decimal point.

While the code we created by recording a macro is helpful in understanding the basics of the VBA language, it can be confusing because it is unnecessarily verbose. For example, the same result as lines 14 and 15,

```
Range("C3:C4").Select
Selection.NumberFormat = "0.00%"
```

can be achieved with the command

```
Range("C3:C4").NumberFormat = "0.00%"
```

which modifies directly the property `NumberFormat` of the object `Range("C3:C4")`.

You can test that this is the case by deleting lines 14 and 15 in the VBA code in the VBA editor window of the file **D-ReturnCalc.xlsm**, and replacing them with `Range("C3:C4").NumberFormat = "0.00%"`. Save the code by pressing **Ctrl-S** or selecting **Save** from the list under the main Microsoft Excel button . Next, delete cells C3:D5 in the worksheet and run the `ReturnCalc` macro again. The result and the formatting should be the same.

The effect of the `With/End` structure in lines 33 through 36 is another piece of code that can be replicated easily through other commands; the advantage of the structure is that it allows you to reduce the number of listed objects in the code, and that it makes the code more readable. A `With/End` statement requires the specification of an object. Inside the `With/End` statement, one can omit mentioning the object with every modification of a property or application of a method to the object. In this particular example, lines 33 through 36 could be replaced with

```
Range("C5:D5").Borders(xlEdgeTop).LineStyle = xlDouble
Range("C5:D5").Borders(xlEdgeTop).Weight = xlThick
Range("C5:D5").Borders(xlEdgeTop).ColorIndex = xlAutomatic
```

with the same effect as the `With/End` statement that references `Range("C5:D5").Borders(xlEdgeTop)`. However, the `With/End` statement is more concise.

In general, when writing VBA code you do not need to select cells explicitly in order to enter data into them. However, if you are new to VBA, it is helpful to record the macro first to see the code VBA suggests, and clean up afterwards. In addition, it is a good idea to “comment out” the redundant statements at first, rather than deleting them. After commenting out overly verbose statements, save the macro by pressing **Ctrl-S**, make sure it still does what you would like it to do, and only then go back and delete the redundant statements. (Commenting out is done by entering apostrophe (’) in the front of the line of code that you wish VBA to ignore.)

A less verbose version of the VBA code is:

```
Sub ReturnCalc()
’
’ ReturnCalc Macro
’ Less verbose
’
    Range("C3").Formula = "=(RC[-1]-R[-1]C[-1])/R[-1]C[-1]"
    Range("C3").Copy
    Range("C4").Select
    ActiveSheet.Paste
    Range("C3:C4").NumberFormat = "0.00%"
    Range("D3").Formula = "=1+RC[-1]"
    Range("D3").NumberFormat = "0.00"
    Range("D4").Formula = "=R[-1]C*(1+RC[-1])"
    Range("C5").Formula = "Total return"
    Range("D5").FormulaR1C1 = "=R[-1]C-1"
    Range("D5").Style = "Percent"
    Range("D5").NumberFormat = "0.00%"
    With Range("C5:D5")
        .Borders(xlDiagonalDown).LineStyle = xlNone
        .Borders(xlDiagonalUp).LineStyle = xlNone
        .Borders(xlEdgeLeft).LineStyle = xlNone
        With .Borders(xlEdgeTop)
            .LineStyle = xlDouble
            .Weight = xlThick
            .ColorIndex = xlAutomatic
        End With
        .Borders(xlEdgeBottom).LineStyle = xlNone
        .Borders(xlEdgeRight).LineStyle = xlNone
        .Borders(xlInsideVertical).LineStyle = xlNone
    End With
    Range("D5").Select
End Sub
```

Notice how the `With/End` structure was used to reduce the number of words we need to use, and how `With/End` structures can be nested inside one another. You can test that this code achieves the same effect by copying replacing the current code in the module in file **D-ReturnCalc.xlsm**, saving the new code, and rerunning the macro `ReturnCalc`.

Before we end this section, we would like to mention a useful property of the `Range` object, `Offset(v,h)`. It points to a cell that is `v` cells above or below (vertical direction) and `h` cells to the right or left (horizontal direction) from a specific cell. For example,

```
Range("C5").Select
ActiveCell.Offset(1,2) = 10
```

sets the value of the cell that is 1 cell down and 2 cells to the right from cell C5 (i.e., cell E6) to 10. Similarly,

```
Range("C5").Select
ActiveCell.Offset(-1,-2) = 20
```

sets the value of the cell that is 1 cell up and 2 cells to the left from cell C5 (i.e., cell A4) to 20.

We saw the idea of referencing cells above and below and to the left and right of the current cells in the example code at the beginning of this section. For example, the formula in line 9 of the original macro,

```
ActiveCell.FormulaR1C1 = "=(RC[-1]-R[-1]C[-1])/R[-1]C[-1]"
```

uses the cell in the same row and one column to the left (`RC[-1]`) and the cell one row up and one column to the left (`R[-1]C[-1]`) to compute the value in the active cell. These kinds of commands help when one prefers to create *relative references*—that is, to perform tasks relative to a prespecified location in the worksheet without changing the code when the starting location is changed. The default in VBA is to record macros in absolute reference mode. To change the mode to relative references, make sure that the Use **Relative References** button in the Code group under the Developer tab before starting the macro recorder.

D.2 PROGRAMMING TIPS

While some desired formatting of an Excel worksheet can be recorded with the macro recorder, knowing basic programming in VBA opens up a whole lot of additional functionality. For example, suppose that you have a set of data on stock returns over several months and, as often happens with real world data, it is not recorded well—there are some duplicate rows. You could record a macro as you go through the worksheet and clean them by hand, but next time you have a set of data, duplicate entries will not be exactly in the same rows as the first set of data. How can you tell Excel to sort through the data and remove duplicate rows in *any* set of data? You need to construct a program from scratch, and make the code general enough to enable the script to be transferable.

Next, we cover some basic VBA programming concepts. We discuss the difference between subroutines and user-defined functions, explain variable declaration in VBA, and introduce some important control flow statements. These concepts are not unique to VBA—versions of them exist in most programming languages.

D.2.1 Subroutines vs. User-Defined Functions

Subroutines and user-defined functions in VBA are both blocks of code saved in modules.[3] The difference is that subroutines are general *scripts*, that is, lists of instructions; whereas functions complete a list of instructions *and* return a value to the user. Subroutines have the general form

```
Sub ()
       [commands]
End Sub
whereas functions have the form
Function FunctionName(list of inputs) As type
       [commands]
       FunctionName = Return value 'Computed from [commands]
End Function
```

The macro recorded in section D.1 was an example of subroutine code. Below, we provide another small example in order to illustrate the difference. Do not worry about the details of the commands right now; each part of the code is explained in subsequent sections.

Suppose we want to calculate $n!$ (pronounced "n factorial"), where n is an integer number the user provides as input.[4] The definition of $n!$ was given in Chapter 3. Module `FactorialExamples` in file **D-Factorial.xlsm** contains examples of subroutines and examples of user-defined functions that accomplish this goal. For example, the subroutine

```
Sub FactorialSub1()
```

```
'Compute factorial using control flow statements

    'Declare the variable that will store the value for
       factorial
    Dim Factorial As Integer
    'Declare the variable that will store the number n
    Dim inNumber As Integer
    'Declare the variable that will be used as counter in
       the loop
    Dim i As Integer

    'Read in the number from cell B1, store it in inNumber
    inNumber = Range("B1").Value

    'Calculate factorial
    Factorial = 1

    For i = 1 To inNumber
        Factorial = i * Factorial
    Next i

    Range("B2").Value = Factorial
End Sub
```

takes the number specified in cell B1, computes the factorial of that number, and sets the value of the cell B2 to the value of that factorial. To see how this subroutine works, open the file **D-Factorial.xlsm**, and delete the contents of cell B2. Press **Alt-F8**, and select `FactorialSub1`. Since the number entered in cell B1 when you open the file is 5, the subroutine fills cell B2 with 120 ($5! = 1\cdot 2\cdot 3\cdot 4\cdot 5 = 120$).

The function `FactorialFun1` computes the same result, but works in a different way. It takes a number as an input (`inNumber`), and returns a number as an output (`FactorialFun1`). The output to be returned should have the same name as the function. The code of the function is shown below:

```
Function FactorialFun1(inNumber) As Integer
    Dim i As Integer
    'Calculate factorial
    FactorialFun1 = 1
    For i = 1 To inNumber
        FactorialFun1 = i * FactorialFun1
    Next i
End Function
```

This function is called in cell B3 of the worksheet with the expression `=FactorialFun1(B1)`. The value of cell B3 is computed to be 120. Notice that the syntax for calling your (user-defined) function is not different from the syntax for calling built-in Excel functions. In fact, Excel has a function for computing factorial, `=Fact(number)`, and if you enter the expression `=Fact(B1)`, you would get the same result (120).

Excel functions can be used also inside VBA code with the prefix `Application`.[5] For example, `FactorialSub1()`and `FactorialFun2()` saved in the file **D-Factorial.xlsm** illustrate how the factorial can be computed by calling the built-in Excel function `Fact`.

```
Sub FactorialSub2()
'Compute factorial using Excel's FACT function within a
    subroutine
    Range("B5") = Application.Fact(Range("B1"))
End Sub
Function FactorialFun2(inNumber) As Integer
    'Calculate factorial
    FactorialFun2 = Application.Fact(inNumber)
End Function
```

What is the advantage of using user-defined functions rather than subroutines? In some cases, you can only use one or the other. However, in cases where both are possible, it may be preferable to structure the script as a function as opposed to a subroutine. User-defined functions are more "transferable," making then easier to use in different places in the worksheet. There are other conveniences—for example, see what happens when the number for *n* in cell B1 is changed from 5 to 6. Cell B3 (which contains the call to the function `FactorialFun1`) immediately updates to 720, which is the correct result. However, cells B2 and B5—those that are output ranges for the subroutines `FactorialSub1` and `FactorialSub2`—do not update until you rerun the macros associated with them.

D.2.2 Variable Declaration

Variables are a basic common concept in computer languages. They are used to store numerical and text data and handle intermediate output in subroutines and functions. For example, `inNumber` in the code for `FactorialSub1` was a variable that stored the value of *n* for which the factorial should be computed. There is no convention for naming variables, but a good practice is to give them meaningful names (rather than *x*, *y*, and *z*), so that your code is easier to follow. We prefer to start names of variables with small letters. If there is a second word in the name, that word starts with a capital letter. It is also a good idea to differentiate variables that store *in*puts (such as `inNumber`), and variables that record *out*put (e.g., `outFactorialValue`).

Depending on their *type*, variables are handled differently, and are allocated a different amount of memory. For example, we specified that `inNumber` should be an integer number by declaring it with the syntax `Dim` *`variableName`* `As` *`variableType`*:

```
Dim inNumber As Integer
```

Other types of variables include `String`, `Single`, `Double`, `Long`, `Boolean`, `Date`, `Object`, `Variant`, and the like. For example, when you need a variable that will hold a fractional (also called "floating point") value, then you should use the `Single` or `Double` data type. When you need a variable to store text data, use the `String` type. The `Variant` type can be used to replace any type; however, it also uses up the largest amount of space, so it is better to specify a particular type for a variable if you know it.

When specifying a variable type, make sure that you have enough space for the data you are planning to store in that variable. If the value gets too large for the variable type, your program may crash. For example, the `Integer` type can store values between −32,768 and 32,767. If you need to store an integer number outside this range, use the `Long` variable type. Similarly, the `Single` (floating point) type can store numbers between −3.402823E38 and −1.401298E-45 for negative values, and numbers between 1.401298E-45 and 3.402823E38 for positive values.[6] If you need to work with fractional numbers outside this range, use the `Double` (floating point) variable type.

Variables can be grouped into arrays. For example,

```
Dim myArray(5) As Integer
```

declares an array of integers of size 6. One of the most confusing things about VBA is the way it handles arrays. The default is to index the first element in arrays as 0, which is the convention in most programming languages, which is why the total number of elements in `myArray` is 6. However, in some special circumstances arrays are treated as starting with the index 1. To ensure consistency and minimize confusion, it is helpful to use the command

```
Option Base 1
```

at the beginning of the module, which makes sure that the indexing of arrays always starts at 1. If this option is stated, then declaring

```
Dim myArray(5) As Integer
```

will result in an array of 5 elements. Those elements can be referenced as `myArray(1)`,. . ., `myArray(5)` later in the program.

You can specify arrays of multiple dimensions as well, for example,

```
Dim myMultiArray(5,2) As Integer
```

will result in an array of 5 rows and 2 columns.

You can also declare dynamic arrays, that is, arrays that do not have specific dimensions from the beginning. This may happen if, for example, you have a set of data, and you need to read it in before you know how many elements it has. In that case, you would declare an array

```
Dim myDynamicArray() As Integer
```

which will be filled as necessary. Once the number of elements is counted, the array can be resized by using the command `ReDim`, for example, `ReDim myDynamicArray(10)`. Note that `ReDim` reinitializes (sets to empty) all values within an array. If you want to preserve the values that are already there, use `ReDim Preserve`, which preserves as many elements as can fit in the new array dimensions.

Working with arrays within VBA is cumbersome, and prone to errors. Often, one needs to resort to loops (see the introduction to loops in the next section) to handle array operations. In many cases, it may be preferable to use built-in Excel array manipulation functions, such as `SUMPRODUCT` (which performs vector multiplication). As we mentioned earlier, these Excel functions can be called with `Application.FunctionName`. For example, `Array3 = Application.SUMPRODUCT(Array1, Array2)` will fill a variable array `Array3` with the result of the elementwise multiplication and summation of the matrix arrays `Array1` and `Array2`.

VBA will assume that you are creating a new variable whenever you use an expression that is not one of the standard commands. Stating the type of variables you use in the program (typically, this is done at the beginning of the program) can save you a lot of headache. Also, we strongly recommend that you write the statement `Option Explicit` in the first line of the module (see file **D-Factorial.xlsm**). This statement makes sure that Excel will report an error if it encounters an undeclared variable in your code.[7] While this may seem like an inconvenience, think about a situation in which you mistype the name of a variable somewhere in your program. If Excel is not in the `Option Explicit` mode, it will treat the mistyped name as a new variable, ignoring any value that your variable may have had at that point in the program, and you will get nonsensical output. If Excel reports an error instead, you will know to fix the typo.

D.2.3 Control Flow Statements: `For` and `If`

Control flow statements in VBA allow us to build more sophisticated programs than simple input and output of data to Excel. We briefly review a couple of important control statements that are used in VBA code: an example of an iterative statement (the `For` loop) and an example of checking a condition (the `If` statement).

The general syntax of a `For` loop in Excel is as follows:

```
For i = 1 to n
        commands
Next i
```

The commands inside the `For` loop are executed once for every value of n.[8] We saw an example of a `For` loop in the code for calculating the factorial of a number n. Let us walk through the `For` loop code inside `FactorialSub1`.

```
’Calculate factorial
    Factorial = 1

    For i = 1 To inNumber
        Factorial = i * Factorial
    Next i
```

The initial value of `Factorial` is set to 1. Suppose the value for `inNumber` is 5. The loop starts at `i = 1`. During the first iteration, the new value of `Factorial` equals the current value of `i` (which is 1) times the current value of `Factorial` (which is 1 as well). At the end of the first run through the loop, the value of `Factorial` is 1. Next, the value of `i` is set to 2. The new value of `Factorial` equals the current value of `i` (which is 2) times the current value of `Factorial` (which is 1), that is, it equals 2. At the third iteration, the value of `i` is 3, and the current value of `Factorial` is 2, that is, the new value of `Factorial` is $3 \cdot 2 = 6$. And so on, and so forth for the next values of `i`, which are 4 and 5. The value of `Factorial` keeps getting updated, until it reaches 720 (=5!) in the last iteration of the loop.

There are other commands that enable us to iterate through commands multiple times, such as the **Do While** and **Do Until**. See VBE's **Help** for description of the syntax and use of these alternatives.

The general form of the `If` statement is

```
If condition Then
      commands
End If
```

When the condition is true, the block of commands executes. More generally, you can use a statement of the kind

```
If condition1 Then
      commands1
ElseIf condition2 Then
      commands2
Else
      commands3
End If
```

`Commands1` will be executed if *`condition1`* is true. If *`condition1`* is not true, then (and only then) *`condition2`* will be checked. If *`condition2`* is true, then `commands2` will be executed. If *`condition2`* is not true, then `commands2` will be executed.

When using `If` statements, one typically needs to check compare values of variables and check whether conditions are true. Therefore, it is useful to know about the logical operators that allow for such comparisons and checks. The comparison operators are: = (tests for equality), <> (tests for inequality), < (tests whether the variable to the left of it is less than the variable on the right), > (tests whether the variable to the right of it is less than the variable on the left), <= and >= (test for less than or equal to / greater than or equal to). Additional useful operators are `AND`, `OR`, and `NOT`. `AND` allows to check whether more than one statements are true at the same time. `OR` returns a `True` result if at least one of the statements is true. `NOT` returns a `True` result if the statement is false.

To illustrate how we can use these operators, consider a couple of simple examples that involve three numerical variables, `var1`, `var2`, and `var3`. Let `var1` = 5, `var2` = 10.

The code

```
If (var1 <> var2) Then
        var3 = 100
```

```
Else
        var3 = -100
End If
```

checks whether the value for `var1` is different from the value of `var2`. If it is (i.e., the value of the logical statement `(var1 <> var2)` is `True`), then the value of `var3` is set to −100; otherwise it is set to 100. In this example, the value of `var3` at the end of the loop is 100, since the value for `var1` (5) is indeed different from the value of `var2` (10).

Consider also the example

```
If (var1 < 5) Or (var2 >= 7) Then
        var3 = 100
Else
        var3 = -100
End If
```

The code checks if at least one of the statements `(var1 < 5)` and `(var2 >= 7)`is true. If it is, then the value of `var3` is set to 100; otherwise it is set to −100. In our case, the first statement is false because the value of `var1` is not less than 5 (it is equal to 5). However, the second statement is true: The value of `var2` (10) is indeed greater than or equal to 7. Therefore, the combined statement `(var1 < 5) Or (var2 >= 7)` is true, and the value of `var3` will be set to 100.

D.2.4 User Interaction in VBA

While we covered the most fundamental concepts about the VBA language, it is fun to learn also about some additional capabilities that enable your programs to interact better with their users. For example, once you have created a macro, you can associate it with a button that the user can press every time he or she wants the macro to run. To do that, click the **Developer** tab if necessary, click **Insert** in the Controls group and then click Form Controls palette. When the Macro dialog box appears, click the macro you would like to have associated with this button.

Sometimes, it is convenient to ask the user to input information through an *input* dialog box. This can be done with the command `InputBox(``question for user'', ``title of the input box'')`. For example,

```
inNumber = InputBox(``Enter an integer'', ``Factorial
   Calculation'')
```

will prompt the user to enter an integer number, and will save that number into the variable `inNumber`. The title of the input box will be `Factorial Calculation`.

Other useful user interaction tools include Message Box (`MsgBox`), which allows you to report output not in a cell on the worksheet, but in a message box. To test how it works, let us go through the following modification of the factorial calculation program (saved as the subroutine `Factorial-SubMsgBox()` in **D-Factorial.xlsm** that can be called with **Alt-F8** as a regular macro):

```
1  Sub FactorialSubMsgBox()
2      Dim inNumber As Variant
3      Dim numberType As Boolean
4      Dim outFactorial As Integer
5
6      inNumber = InputBox("Enter an integer number",
       "Factorial
7 Calculation")
8      numberType = IsNumeric(inNumber)
9
10    If numberType = True Then
11        outFactorial = Application.Fact(inNumber)
12        MsgBox ("The factorial of " &
          inNumber  & " equals " &
13 outFactorial)
14    ElseIf numberType = False Then
15        MsgBox ("Not a number. Please enter an
          integer number.")
16    End If
17 End Sub
```

On line 6, the user is asked to specify the desired number to compute the factorial. Line 8 checks whether this is indeed a number. Note that the variable `numberType` is specified as Boolean, which means that it can only take True or False values. If it is true,that is, if `inNumber` is indeed a number, then the Excel built-in function `Fact` calculates the factorial of this number, and print the statement "**The factorial of** *the number the user entered* **is** *the result obtained*" in a message box on the screen. If it is not true, then the user is prompted to enter a number.

Note that line 2 specifies the type of variable for `inNumber` as `Variant`, which allows it to be anything. If we had declared `inNumber As Integer` and had entered a letter instead of a number, Excel itself would have returned an error, complaining that there is a variable type mismatch between what was declared and what the actual value of the variable is. Thus, declaring the exact type of variable whenever we know the type is very is important for minimizing errors in output.

D.3 DEBUGGING

VBA has useful debugging tools that allow you to look at the code in more detail if your programs do not work as expected. These tools can be accessed through commands in the **Debug** menu, which can be accessed on the VBE menu bar.

The **Step Into** button or keyboard shortcut **F8**, lets you execute your program step by step. When you are executing a program step-by-step, your program is in "break mode." Every time you press **F8**, the "break" is moved to the next command. While the break is set on a particular command, placing the cursor over any variable above the break point will give you an updated stored value for that variable. This makes it easy to catch calculation errors and inconsistencies. You can "step over" (i.e., skip) some subroutines that you are not interested in double-checking—click the Step Over button or the keyboard shortcut **Shift-F8**—and "step out" of the break mode—click the **Step Out** button or the keyboard shortcut **Ctrl-Shift-F8**. Equivalently, you can click on the **Reset** button in the top VBE toolbar (■) to get out of debug mode.

Rather than going through the program step-by-step, it is sometimes helpful in long programs to set breakpoints in advance, so that the program runs until it gets to a particular breakpoint. A breakpoint can be specified by placing the cursor at the place where it should be inserted, and clicking on the Toggle Breakpoint (✋) button in the **Debug** menu (or using the keyboard shortcut **F9**). When the program gets to the breaking point, it automatically goes into break mode and allows you to follow the subsequent commands step-by-step and check the values of the variables at that point in the program. To remove a breakpoint, simply place the cursor in the corresponding line, and click on the breakpoint button again.

NOTES

1. While there were substantial changes in the organization of menu items in Microsoft Excel 2007 relative to earlier versions of Excel, the substance of the commands is very similar, so readers with earlier version of the software can easily use Excel help to search for the corresponding location of the commands we

use. Note that Excel files created in earlier versions of Excel were saved with the extension .xls, independently of whether they contained macros. In Excel 2007, a distinction is made between files that do not contain macros (suffix .xlsx) and files with macros (suffix .xlsm).

2. VBA does not quite qualify as an object-oriented language for technical reasons; however, for all practical purposes it is helpful to remember that VBA shares many of the same concepts as "real" object-oriented programming languages.
3. If you do not see a module when you open VBE and you would like to create one, select **Insert | Module** from the top menu in VBE.
4. Recall, for example, that this expression was used in the calculation of binomial probabilities.
5. It is worthwhile to note here that VBA itself has some built-in numeric functions. In particular, functions such as `Abs` (absolute value), `Exp` (exponential), `Int` (integer part), `Cos` (cosine), `Sin` (sine), `Log` (natural log), `Rnd` (random number generator), `Sign` (sign function), `Tan` (tangent), and `Sqr` (square root) can be used directly within VBA code, without the prefix `Application`. While it seems that this should make things easier, it may also be a source of confusion. Notice that Excel has equivalent numerical functions for formulas that are entered into cells, but the syntax for some of the functions is different. For example, the natural logarithm function in Excel is `Ln`, and the square root function is `Sqrt`. So, typing `Sqr` in your program in VBA is equivalent to typing `Application.Sqrt`. In practice, you would want to use the shorter syntax `Sqr`, but it is important to be aware of this inconsistency.
6. The notation `E` in Excel denotes multiplication by 10 to a specific power. For example, 5E40 means 5·1040, and 5E-45 means 5·10-45.
7. This can be accomplished also by checking **Require Variable Declaration** under **Tools | Options** in the top VBE menu.
8. One can also specify a *step* by stating `For i = 1 to n Step k`. For example, if $n = 10$ and the step $k = 2$, then the commands in the loop will be executed for $n = 1, 3, 5, 7, 9$.