1

An Introduction to Excel VBA

Excel VBA is probably the most commonly used computational tool in financial institutions, particularly when a new model is tested at a preliminary stage within a division. Many traders use Excel VBA to compute their trading strategies. Some data providers allow users to update information in real time using the Excel format. Excel VBA thus allows traders and risk managers to implement their solutions conveniently in real time.

1.1 HOW TO START EXCEL VBA

1.1.1 Introduction

VBA stands for Visual Basic for Application. It is a programming language that enhances the applicability of MS Excel by enabling the users to instruct Excel to perform tasks automatically. As most of the programs in this book are written in VBA, a brief introduction to VBA is provided in this opening chapter. Although we do not assume that readers have prior programming knowledge, programming experience in other languages would be helpful. For readers already familiar with VBA, this chapter serves as a refresher and quick reference. A list of the functions defined throughout the book can be found at the end of the chapter. These functions not only improve readability and traceability but also simplify the programs. For a more thorough understanding of Excel VBA, readers are referred to other books

Handbook of Financial Risk Management: Simulations and Case Studies, First Edition. N.H. Chan and H.Y. Wong.

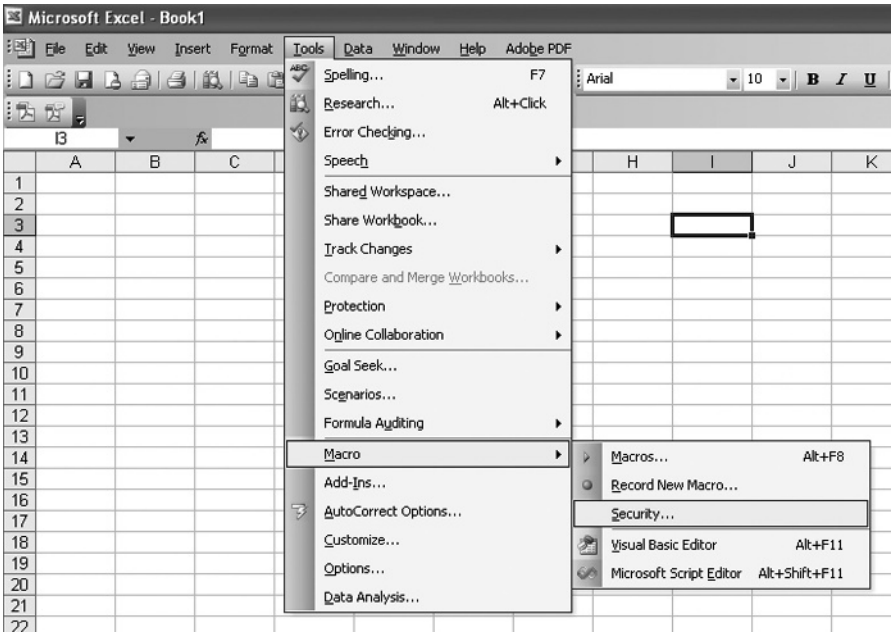

Figure 1.1 Set security level.

specializing in the matter. We believe, however, that this chapter is sufficient to allow a beginner to learn and execute the codes within the book.

MS Excel 2003 is used for illustration in this book. If readers are using another version of Excel, then they may find some minor differences. Nevertheless, if this is the first time for a reader to use Excel VBA, then set the macro security level to Medium or Low and restart Excel to enable the macros:

Click [Tools] → [Macro] → [Security] → [Medium] or [Low] (Fig. 1.1).

MS Excel 2007 users should click the Options button to enable the macros.

1.1.2 Visual Basic Editor

VBE, which stands for Visual Basics Editor, is the environment in which macros are created, modified and managed. Macros (VBA procedures) are the code components that automate repetitive Excel tasks. A macro consists of codes that start with the keyword *Sub* or *Function* and end with the keywords *End Sub* or *End Function*. These codes are known as *Sub* and *Function* procedures. A module contains one or more macros, and a project contains one or more modules. A macro developed in VBE

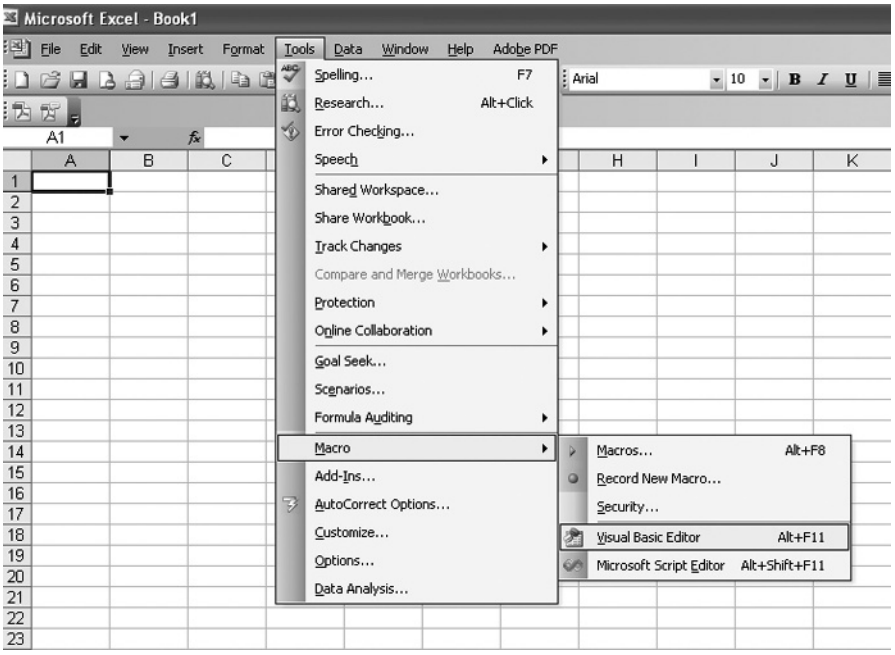

Figure 1.2 Open VBE.

becomes part of a workbook and is saved at the same time that the workbook is saved. To open and edit macros in VBE, follow the procedure below.

1. Open VBE: Click [Tools] → [Macro] → [Visual Basic Editor] or press Alt + F11 (Fig. 1.2).
2. Insert module: In the project window on the left of the VBE, right-click one of the worksheets → [Insert] → [Module] (Fig. 1.3).
3. Edit in VBE: Type the codes in the code window.
4. Execute the program: In VBE, click [Run] → [Run Sub] and choose the macro to be compiled. Equivalently, in Excel, click [Tools] → [Macro] → [Macro] and choose the macro to be compiled.

1.1.3 The Macro Recorder

Excel offers a macro recorder that records the actions of the mouse and/or keyboard and translates them into VBA codes, thus allowing the designated actions to be repeated by running the macro again. Although the macro recorder is sometimes useful, it is unable to generate codes that perform looping, assign variables, or execute conditional statements, which are fundamental components in simulation. In

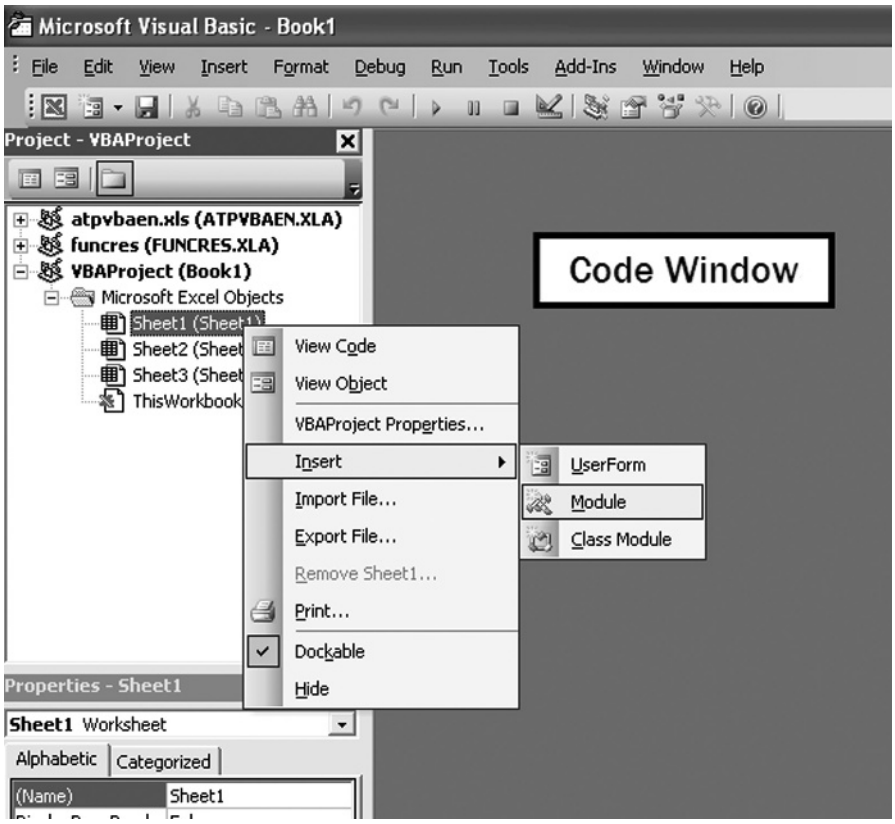

Figure 1.3 Insert modules.

addition, the codes that are generated depend on certain specific settings. To record a macro, follow the procedure below.

1. Open the macro recorder: Click [Tools] → [Macro] → [Record New Macro].
2. Type the macro name and click OK. Note that the name should begin with a letter and contain no spaces or special characters (Fig. 1.4).
3. Perform all of the actions to be recorded. Here, type "Hello" in cell A1.
4. Stop the macro recorder: Click [Stop recording macro] button.

Note that when a macro is recorded, MS Excel automatically inserts a VBA module to keep the recorded codes. To execute the recorded macros or other macros, click [Tools] → [Macro] → [Macros] or Alt + F8 in Excel. Then, select the designated macro to implement and click [Run] (Fig. 1.5). To view the codes in the recorded macro, open VBE and double-click the newly added module (Fig. 1.6).

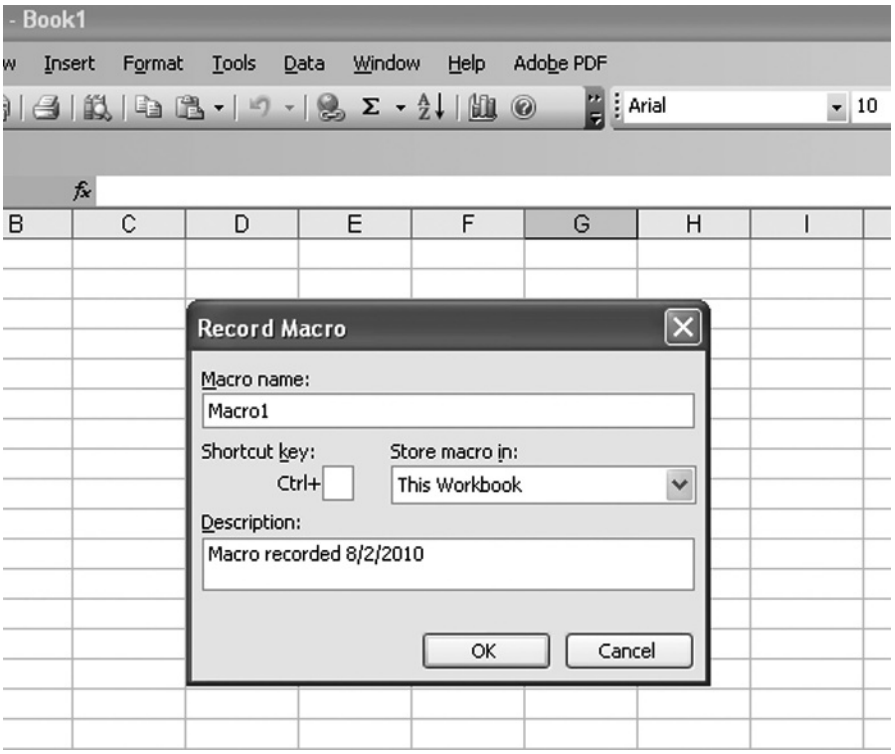

Figure 1.4 Name a macro.

Figure 1.6 shows that the recorded macro is a complete *Sub* procedure. The lines beginning with the symbol ′ are not executed as they are program *comments*. A comment can be added to the code by preceding it with the symbol ′ or by using the keyword *Rem* at the beginning of a line.

1.1.4 Insert a Command Button

Instead of having to remember a shortcut key or choosing a macro from a list, it is more convenient to add a command button to the worksheet to invoke the macro directly. To insert a command button, follow the following procedure.

1. Click [View] → [Toolbars] → [Visual Basic] (Fig. 1.7).
2. Click Control Toolbox.
3. Click Command Button and put it in the Excel worksheet (Fig. 1.8).
4. Edit the macro: Double-click the command button.

To use a *Sub* in the module, type *call* [name of the *Sub*] inside the macro of the command button. The common button can also be edited by clicking *Design Mode*

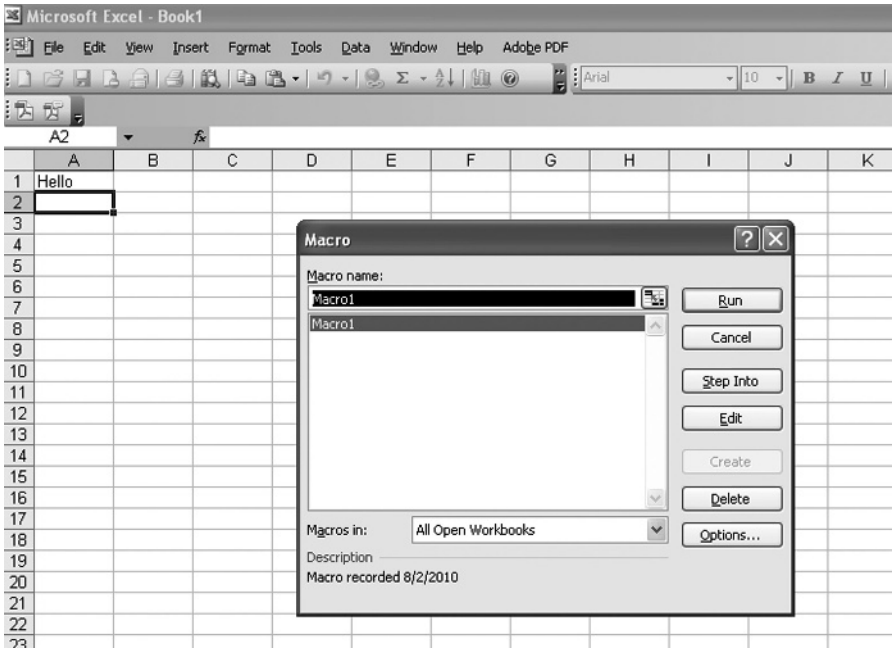

Figure 1.5 Run a macro.

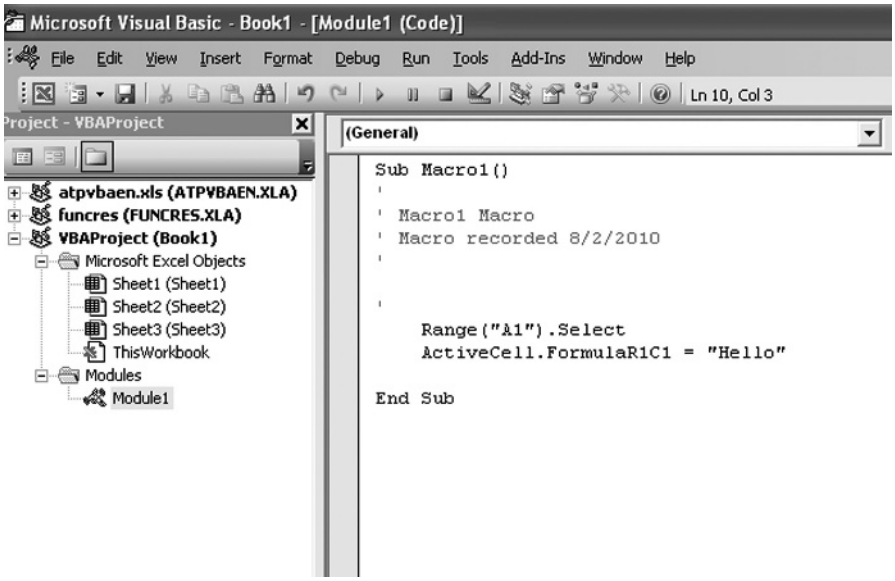

Figure 1.6 View the codes.

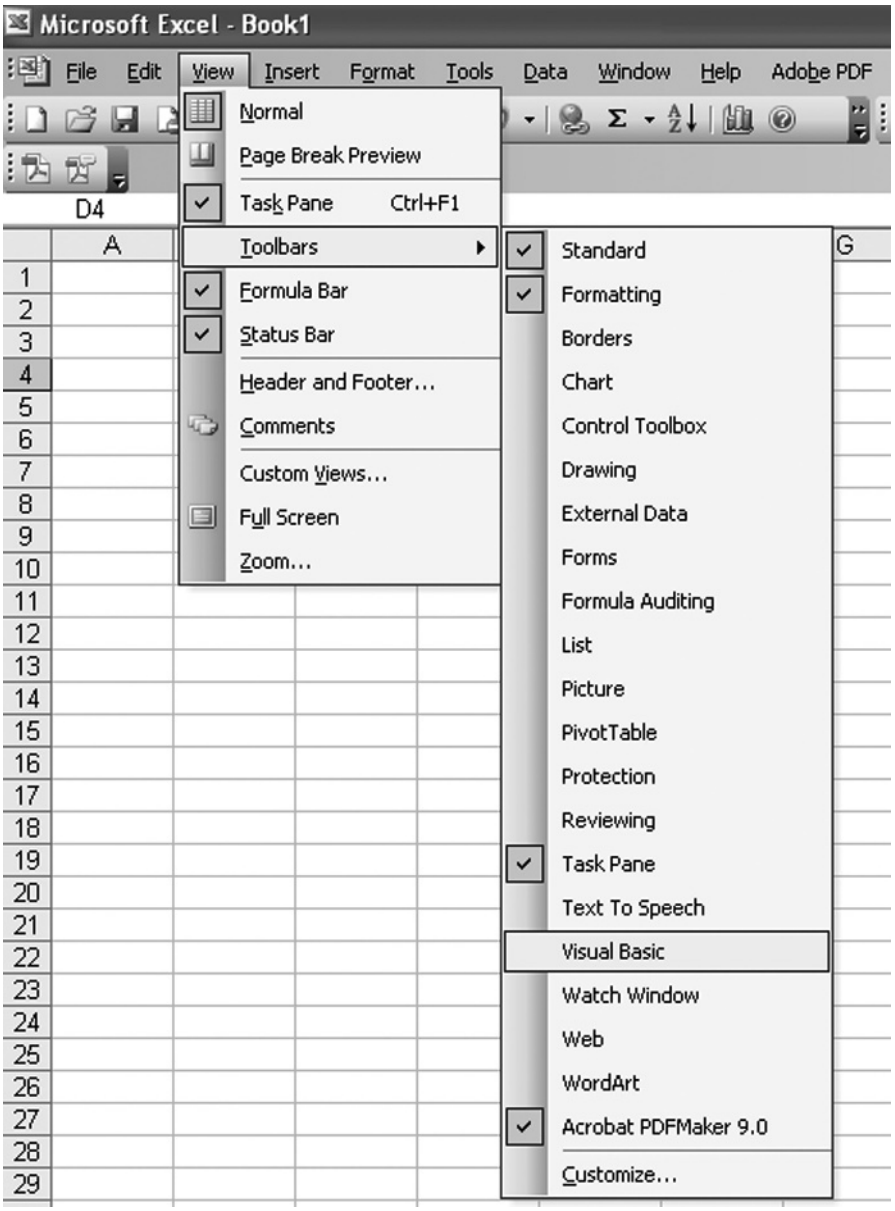

Figure 1.7 Insert command button 1.

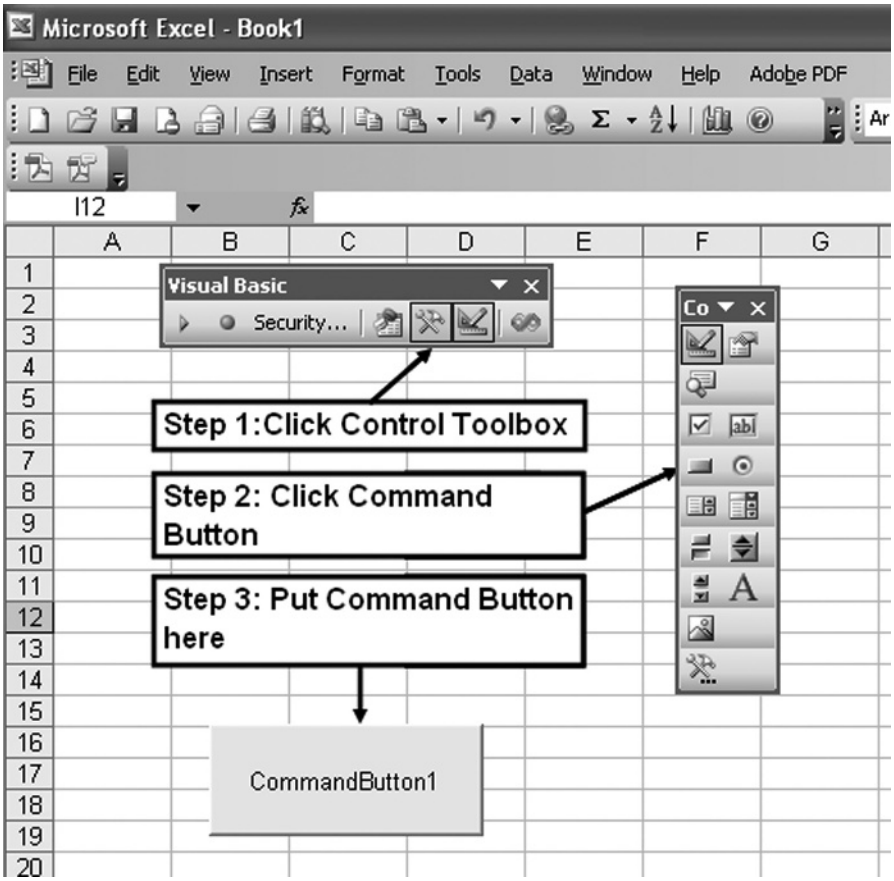

Figure 1.8 Insert command button 2.

in the Visual Basic Control Toolbox, which also contains other useful buttons, such as those for recording a macro and opening VBE.

1.2 VBA PROGRAMMING FUNDAMENTALS

1.2.1 Declaration of Variables

In programming, a *variable* is the name for a place in computer memory in which values or objects are stored. To declare a variable in VBA, use the following statement.

```
Dim varname [As vartype],
```

where *varname* is the variable name and *vartype* is the variable type. A variable name must begin with a letter and contain only numeric and letter characters and underscores. Moreover, the variable name should not be a VBA reserved word, such

as Sub, Function, End, For, Optional, New, Next, Nothing, Integer, or String. It is also important to note that VBA does not distinguish between cases.

Different from other programming languages, specifying the variable type [As vartype] is optional. Other languages require the programmer to define explicitly the data type of each variable used. Although optional in VBA, if the data type is not explicitly specified, then execution is slower and memory is used less efficiently.

1.2.2 Types of Variables

Every variable has a type specifying the type of values it stores. Variables can be classified into four basic types: string data type, date data type, numeric data type, and variant data type. The string data type is used to store a sequence of characters, and the date data type can store dates and times separately or simultaneously. The types that are used most frequently in this book are the numeric and variant data types.

There are several numeric data types in VBA, the details of which are listed in Table 1.1. In general, a user should choose the data type that employs the smallest number of bytes to enhance program efficiency. Doing so may make a big difference in the computational time needed for simulation.

The variant data type is the most flexible data type in VBA. It stores both numeric and non-numeric values. VBA will try to convert a variant variable to the data type, which is able to store the input data. As noted, [*As vartype*] is optional, and the default variable type will be *Variant*.

In addition to normal data, a variant type variable can also store three special types of values: error code, Empty (which indicates that the variable is empty, and is not equal to 0, False, an empty string, or another value), and Null (which means that the variable has not been assigned memory, and is not equal to 0, False, an empty string, Empty, or another value).

TABLE 1.1 Numeric Data Type

Type	Shorthand	Range	Description
Byte		0 to 255	Unsigned, integer number
Boolean		True(−1) or False(0)	Truth value
Integer	%	−32,768 to 32,767	Signed integer number
Long	&	−2,147,483,648 to 2,147,483,647	Signed integer number
Single	!	± 3.402823E38 to ± 1.401298E-45	Signed single-precision floating-point number
Double	#	± 1.79769313486231E308 to ±4.94065645841247E-324	Signed double-precision floating-point number
Decimal		±7.9228192514264337 59E28 with no decimal point and ±7.92281625142643375 9354 with 28 digits behind the decimal point	Cannot be directly declared in VBA; requires the use of a variant data type

Here are some examples of variable declaration statements:

```
Dim a As integer
Dim b 'the type will be variant

Dim c As string
c = "It is a string"

Dim Today As Date
Today = #4/7/2011# 'defined using month/day/year format

Dim Noon As Date
Noon = #12:00:00#
```

1.2.3 Multivariable Declaration

To declare several variables, use the following statement.

```
Dim a As Integer, b As Integer, c As Integer
```

Different from other programming languages, attention must be paid to the following case.

```
Dim a, b, c As Integer
```

If the *Dim* statement is declared as above, then *a* and *b* will be declared as variant types. In this case, the following shorthand can be employed to ensure the cleanliness and readability of the program.

```
Dim a#, b#, c As Double
```

1.2.4 Declaration of Constants

Constants can be declared using a *Const* statement, of which the following are examples.

```
Const interest_rate as Integer = 0.05
Const dividend_yield = 0.03 'without declaring the constant
type
Const option_type as String = "Call"
```

VBA also defines many intrinsic constants that are used in *Sub* and *Function* procedures.

TABLE 1.2 VBA Logical Operators

Operator	What it does
Not	Performs a logical negation on an expression
And	Performs a logical conjunction on two expressions
Or	Performs a logical disjunction on two expressions
Xor	Performs a logical exclusion on two expressions
Eqv	Performs a logical equivalence on two expressions
Imp	Performs a logical implication on two expressions

1.2.5 Operators

This subsection introduces assignment operators, mathematical operators, comparative operators, and logical operators.

The equal sign (=) is an assignment operator and is usually used to assign the value of an expression to a variable or a constant. An expression is a combination of keywords, operators, variables, and constants that yields a string, number, or object. For example,

```
x = 4 * 3
x = x * 5
```

The result of x is 60.

Familiar mathematical operators include *addition*(+), *multiplication*(∗), *division*(/), *subtraction*(−), and *exponentiation*(^).

VBA also supports the comparative operators used in Excel formulas: *equal to* (=), *greater than* (>), *less than* (<), *greater than or equal to* (>=), *less than or equal to* (<=), and *not equal to* (<>).

Table 1.2 presents the logical operators and their uses in VBA.

1.2.6 User-Defined Data Types

Users may sometimes wish to employ a more complex data type to store data. VBA provides the *Type* statement, which allows the creation of a custom data type or a user-defined data type (UDT). The syntax for creating a UDT is

```
[Private|Public] Type typename
    [elementname As vartype]
    [elementname As vartype]
    ...
End Type
```

[Private | Public]: (Optional) It is *public* by default, and indicates whether this UDT can be declared in all modules. If it is declared to be *private*, then the

UDT can be declared only in the same module as that in which the UDT is defined.

typename: (Required) This is the name of the UDT and follows standard variable naming conventions.

elementname: (Required) This is the name of the elements within a UDT and also follows standard variable naming conventions.

vartype: (Required) Unlike the declaration of ordinary variables, the elements within a UDT must be given a data type, which can be any of the aforementioned variable types (including *Variant*) or a UDT.

Declaring a UDT is the same as declaring another built-in variable type. To reference the sub-elements of the UDT, use the period (.) operator. Finally, the UDT should be defined at the top of the module before any procedures, as illustrated in the following example.

Example 1.1 *The following code defines a nested UDT which stores the name and coordinates of a point.*

```
Type Coordinate
    x As Double
    y As Double
End Type

Type Point
    name As String
    c As Coordinate
End Type

Sub UDTEx1()
    'Declare p1 as UDT Point
    Dim p1 as Point

    'Assigning the values
    p1.name = "A"
    p1.c.x = 2.5
    p1.c.y = 3

    'Print out the values to spreadsheet
    Cells(1, 1) = p1.name
    Cells(2, 1) = p1.c.x
    Cells(3, 1) = p1.c.y
End Sub
```

1.2.7 Arrays and Matrices

An array is a collection of variables of the same type that have a common name. An array allows access to the variables through the index number, thereby providing a way to loop through and process a collection of variables of the same type easily.

The following statement declares a one-dimensional (1D) array.

```
Dim varname(LowerIndex to UpperIndex) As vartype
```

In this way, a user can access variables with varname(LowerIndex), varname(LowerIndex +1), . . . , varname(UpperIndex). If he or she specifies only the upper index, that is,

```
Dim varname(UpperIndex) As vartype,
```

then VBA will assume that 0 is the lower index.

The following statement declares a multidimensional array.

```
Dim varname(LowerIndex1 to UpperIndex1, LowerIndex2 to _
UpperIndex2,...,LowerIndexN to UpperIndexN) As vartype
```

For example, to create an array to store the scores of 20 students on three tests, declare:

```
Dim Score(1 to 20, 1 to 3) As Double
```

Here, Score(10, 2) stores the mark of the tenth student on the second test.

Note that both the lower and upper indices must be a constant or a number. If the user wants to employ a variable in the index, then he or she should use a *dynamic array* which has no preset number of elements. The following statement declares a dynamic array.

```
Dim varname() As vartype
```

Before a dynamic array is used, the *ReDim* statement should be employed to specify the number of elements in the array. For example,

```
ReDim varname(LowerIndex to UpperIndex)
```

In this case, the LowerIndex and UpperIndex can be a variable or a constant. In VBA, a matrix is essentially a two-dimensional (2D) array, and a column or row vector is a 1D array. A matrix is an important tool in risk management and finance, as it deals with high dimensional problems. For example, it can be used in multiple linear regression. To declare a matrix of size $m \times n$ containing real numbers, use the following statement.

```
Dim matrixmn() As Double
ReDim matrixmn(1 To m, 1 To n)
```

In the next subsection, we discuss functions related to matrix manipulation.

1.2.8 Data Input and Output

One advantage of Excel VBA is that it allows the VBE and the worksheet to be linked together, affording the user the ability to read and print out the data in the worksheet and execute programs written in VBE. The following statements are usually used for input and output, respectively.

```
'Read in data
Var = Cells(i, j)

'Print out data
Cells(i, j) = Var,
```

where i and j denote the row and column number of a cell, respectively. For example, to print out the score of the sixth student on the last test in cell A2 on the worksheet, write:

```
Cells(1, 2) = Score(6, 3)
```

1.2.9 Conditional Statements

When the program needs to follow different instructions in different cases, we use conditional statements. The two main conditional statements in VBA are *If-Then-Else* statements and *Select-Case* statements.

If-Then-Else Statements

There are two forms of *If-then-else* statements: single-lined and multi-lined. Only one statement can be inserted in the single-lined form whereas several can be inserted in the multi-lined form. With the use of Else statement, the extraneous conditions are not evaluated when an Else statement is used, which improves efficiency. The syntax of the two forms is as follows.

```
'the Else clause is optional
If [condition] Then [statement] (Else [elseStatement])
'... represents other more statements can be included
'these Else clauses are also optional
If [condition] Then
    [statement]
    ...
ElseIf [elseif condition1] Then
    [Statement]
    ...
ElseIf [elseif condition2] Then
    [Statement]
    ...
Else
    [Statement]
    ...
End If
```

In the conditional part of the statement, the users need to specify an expression that can be evaluated as True or False. Use the comparative operators and logical operators discussed in Section 1.2.5.

Select-Case Statements

Select-Case statements are useful for choosing among three or more options and are good alternative to If-Then-Else statements. The syntax for *Select-Case* is as follows.

```
Select Case [testexpression]
  Case expressionlist-n
     [instructions-n]
     ...
  Case expressionlist-n
     [instructions-n]
     ...
  Case Else
     [default_instructions]
     ...
End Select
```

The most common *expressionlist-n* is one of the following.

0 to 20
1, 7
Is >= 10

Example 1.2 *Suppose that the scores of 20 students on three tests have already been stored in the array Score(1 to 20, 1 to 3). Write a* Sub *ensuring that once the student ID and test number are entered into cells B1 and B2, respectively, the program will determine whether the student has passed the test (i.e., achieved a score equal to or higher than 60) and output the result to cell B3.*

The corresponding codes for the *If-Then-Else* statement are:

```
StudentID = Cells(1, 2)
TestNo = Cells(2, 2)

If Score(StudentID, TestNo) >= 60 Then
    Cells(3, 2) = "Pass"
Else
    Cells(3, 2) = "Fail"
End If
```

The corresponding codes for the *Select-Case* statement are:

```
StudentID = Cells(1, 2)
TestNo = Cells(2, 2)

Select Case Score(StudentID, TestNo)
    Case Is >= 60
      Cells(3, 2) = "Pass"
    Case Else
      Cells(3, 2) = "Fail"
End Select
```

1.2.10 Loops

The main purpose of using loops is to allow VBA to perform certain tasks several times. *For-Next* loops and *Do* loops are widely used in VBA programming, with the former, in particular, frequently used in simulations. The syntax for a *For-Next* loop is:

```
For counter = startValue To endValue [Step nStep]
    [statements]
    [Exit For]
    [statements]
Next counter
```

If the *Step nStep* part is omitted, then the counter will increase by 1 each time. We can set *nStep* to be *n* and the counter will then increase by *n* each time.

Example 1.3 *Suppose that the scores of 20 students on three tests have already been displayed in the Range of A1:C20 in the worksheet. To store the scores into the array Score(1 to 20, 1 to 3), we use the following For-Next statement.*

```
Sub LoopEx1()
    Dim Score(1 To 3, 1 To 20) As Double
    For i = 1 To 20
        For j = 1 To 3
            Score(i, j) = Cells(i, j)
        Next j
    Next i
End Sub
```

For a *Do* loop, the syntax is

```
Do [do_condition]
    [statements]
    [Exit Do]
    [statements]
Loop [loop_condition]
```

Although both *do_condition* and *loop_condition* are optional, only one of them can be used for a *Do* loop. If both are omitted, then the user must specify a condition and call `Exit Do` to end the loop. Otherwise, the program will not terminate. The syntax is the same for *do_condition* and *loop_condition*.

```
While|Until condition
```

For *While*, the loop continues as long as *condition* is *True*. For *Until*, the loop breaks once *condition* becomes *True*. If *While* is used, then the loop is also called the *Do While* loop; if *Until* is used, then it is called the *Do Until* loop. The use of *While* or *Until* depends solely on the programmer's preference, as the same task can always be performed either way. However, putting the condition after *Do* or *Loop* depends on the situation, because if it is put after *Loop*, then the loop is repeated at least once. The following example prints 1 to 10 in cells A1 to A10 using different methods.

Example 1.4 *Use five different methods to print 1 to 10 in cells A1 to A10.*

```
'For Loop
For i = 1 to 10
    Cells(i, 1) = i
Next i

'Do Loop Method 1
i = 1
Do while i <= 10
   Cells(i, 1) = i
   i = i + 1
Loop

'Do Loop Method 2
i = 1
Do Until i > 10
   Cells(i, 1) = i
   i = i + 1
Loop

'Do Loop Method 3
i = 1
Do
    Cells(i , 1) = i
    i = i + 1
Loop while i <= 10

'Do Loop  Method 4
i = 1
```

```
Do
    Cells(i, 1) = i
    i = i + 1
Loop until i > 10
```

1.3 LINKING VBA TO C++

Even though C++ was developed back in the 1970s, many programmers still use it today because it is a high-level general-purpose programming language. As many procedures, functions, and algorithms are still developed in the C++ platform, it is important to link VBA to C++. In this section, we link VBA to C++ through .dll (dynamic-link library) by using Visual Studio and calling the functions from VBA. For further details, please refer to *Ch1.3_simplemath.xls*. To build .dll via Visual Studio:

1. Open Visual Studio and select [New Project].
2. Choose [Win32 Project] and enter a name for the project (for example, SimpleMath).
3. In Application Setting, choose [DLL] and select [Empty project]. Click [Finish].
4. Choose [Add New Item] to add new .cpp file. Enter a name (for example, main).
5. Add another item with .def suffix (for example, export.def).
6. In main.cpp, enter the corresponding C++ code for a user-defined function. For example,

   ```
   double minus(double x, double y){
       return x - y;}
   ```

7. In export.def, enter the following code.

   ```
   LIBRARY SimpleMath
   EXPORTS
   minus
   ```

8. Right-click the project file [SimpleMath] in Solution Explorer and choose [Properties].
9. Click [Configuration Properties] → [C/C++] → [Advanced].
10. Choose [__ stdcall(/Gz)] in [Calling Convention].
11. Click [Configuration Properties] → [Linker] → [Input].
12. Choose [.\export.def] in [Calling Convention]. Click [OK].
13. Build the project OR Click [F7].
14. A .dll file (e.g., SimpleMath.dll) is created in the project directory.

To establish linking with VBA:

1. Place the .dll file and .xls file in the same directory.
2. Open VBA editor, and enter the following code in the module.

```
Private Declare Function SetCurrentDirectoryA Lib "kernel32"_
(ByVal lpPathName As String) As Long
Private Declare Function minus Lib "simplemath.dll" _
(ByVal a As Double, ByVal b As Double) As Double

Function test(a As Double, b As Double) As Double
    SetCurrentDirectoryA Application.ActiveWorkbook.Path
    test = minus(a, b)
End Function
```

3. The function "test" can be called up in Excel Worksheet and VBA.

1.4 SUB PROCEDURES AND FUNCTION PROCEDURES

Writing a program in a systematic manner may necessitate the separation of a large program into smaller pieces that can be reused and managed easily. In VBA, a *procedure* is basically a unit of computer code that performs certain tasks. There are two types of procedures: a *Sub* procedure and a *Function* procedure. A *Sub* procedure performs tasks but does not return values, whereas a *Function* procedure does return a value.

The syntax that defines a *Sub* procedure is

```
[Private|Public] [Static] Sub name ([arglist])
    [statements]
End Sub
```

Private|*Public*: (Optional) The *Sub* is *Public* by default if *public* or *private* is omitted. *Public* indicates that the *Sub* is accessible by other *Sub*s or *Function*s in all modules, whereas *Private* indicates that the *Sub* is accessible only to the *Sub*s and *Function*s in the same modules.

Static: (Optional) *Static* indicates that all local variables of the *Sub* are preserved at the end of the *Sub*. If *Static* is omitted, then the values of the local variables will be reset each time the *Sub* ends. See Example 1.5 for an illustration.

name: (Required) This is the identifier of the *Sub* and follows standard variable naming conventions. The *name* must be unique; it cannot be the same as the identifier of other *Sub*s, *Function*s, classes, etc.

arglist: (Optional) This is a list of variables representing parameters that are passed to the *Sub* when it is called. Multiple variables are separated by commas. If the

procedure uses no arguments, then a set of empty parentheses is required. See Examples 1.6 and 1.7 for an illustration.

statements: (Optional) This refers to any group of statements to be executed within the *Sub*.

Example 1.5 *The following Sub SubEx1 adds one to the variable x each time it is called and writes the value of x into cell A1.*

```
Static Sub SubEx1()
    Dim x as integer
    x = x + 1
    Cells(1, 1) = x
End Sub
```

As the previous value of x is preserved each time *Sub SubEx1* is called, cell A1 adds one instead of always printing 1, as in the case of *Static* being omitted. The same effect can be accomplished with the following code.

```
Sub SubEx1()
    Static x as integer
    x = x + 1
    Cells(1, 1) = x
End Sub
```

Example 1.6 *The following procedure for* SubEx2 *calculates* $var1 + var2$ *and outputs the result in cell A1.*

```
Sub SubEx2(var1, var2)
    Cells(1, 1) = var1 + var2
End Sub
```

To call the *Sub*, use one of the two following statements, in which x, y can also be replaced with other constants or variables.

```
Call SubEx2(x, y)
SubEx2 x, y
```

Instead of simply specifying the name of the parameters, each parameter in *arglist* can be specified by the following syntax.

```
[Optional] [ByRef|ByVal] varname [As vartype] [= defaultvalue]
```

Optional: (Optional) This indicates that the parameter is optional and will take *defaultvalue* as its value if it is omitted when the *Sub* is called.

By Ref | *By Val*: (Optional) The parameter is passed to *ByRef* by default. *ByRef* and *ByVal* indicate whether the parameter is passed by address or by value. When calling with *ByRef*, the parameter's memory address is passed to the procedure, and any changes of the parameter value in the procedure cause changes to the original parameter. For *ByVal*, in contrast, a copy of the value of the parameter

is passed and so the original parameter is not affected. See Example 1.7 for an illustration.

varname: (Required) This is the identifier of the parameters.

vartype: (Optional) The variable type is *Variant* by default. It is the variable type of the parameter that has been passed, and can be any of the variable types or a UDT. If the variable passed when calling the *Sub* does not match, then the error message "ByRef/ByVal argument type mismatch" is shown.

defaultvalue: (Optional) This is the value that the parameter takes when the parameter is not specified and the *Sub* is called.

Example 1.7 *The following codes demonstrate the difference between ByRef and ByVal.*

```
Sub SubEx3_Run()
    Dim x as integer, y as integer
    x = 1
    y = 1
    Call SubEx3(x, y)
    Cells(1, 1) = x
    Cells(2, 1) = y
End Sub

Sub SubEx3(ByRef var1 as integer, ByVal var2 as integer)
    var1 = var1 + 1
    var2 = var2 + 1
End Sub
```

The foregoing codes can be copied to a module with *SubEx3_Run*, then run. Cell A1 shows 2, as the change in the value of *var1* in *SubEx3* actually changes the value of x. Cell A2 shows 1, as the change in the value of *var2* in *SubEx3* does not affect the value of y.

VBA also allows the user to create a *Sub* to take an arbitrary number of parameters using *ParamArray*. When using *ParamArray*, the parameters can be passed only by reference and declared as the *Variant* type. They will be stored in an array with the parameter's name. To declare such a *Sub*, use

```
Sub SubEx4(ParamArray var())
    [statements]
End Sub
```

Although a *Function* returns a value, whereas a *Sub* does not, a *Function* can also be used in formulas in the Excel spreadsheet as a user-defined function. The syntax that defines a *Function* is

```
[Private|Public] [Static] Function name ([arglist, ...])
    [as vartype] [statements]
End Sub
```

For *Private|Public*, *Static*, *name*, and *arglist* a *Function* is identical to *Sub*. The only difference between the declaration of a *Function* and a *Sub* is that the user may want to define the return type *vartype* of the *Function*. The return type is *Variant* by default if it is omitted. To return a value for a *Function*, the user needs to store that value in a variable with a name identical to the given function name. See Example 1.8 for an illustration. To call a *Function*, use one of the following statements.

```
Call FuncName(x, y)
FuncName x, y
z = FuncName(x, y)
```

Note that the first two are identical because *Sub* is used. For the third, the return value will be stored in z.

As *Sub* cannot return a value, we may need to use global variables or pass the variables by reference to accomplish certain tasks. Example 1.8 calculates $var1 + var2$ and outputs the result into cell A1, which is analogous to Example 1.6 using *Function*.

Example 1.8 *The following code is to calculate* $2 + 3$ *by calling Function FuncEx4 and output result 5 into cell A1.*

```
Sub SubEx4()
    Cells(1, 1) = FuncEx4(2, 3)
End Sub

Function FuncEx4(var1 as integer, var2 as integer) as integer
    FuncEx4 = var1 + var2
End Function
```

1.4.1 VBA Built-In Functions

VBA has a variety of built-in functions that can simplify calculations and operations. For a complete list of VBA functions, please refer to the VBA Help system. In VBE, one can type VBA to display a list of VBA functions. Table 1.3 presents some commonly used VBA built-in mathematical functions and their return values in descriptive and mathematical forms.

Example 1.9 *The following code calculates* $sin(e^2)$ *and outputs the result into cell A1.*

```
Sub expsquare()
    cells(1, 1) =  sin(exp(2))
End Sub
```

TABLE 1.3 Common Built-In Mathematical Functions in VBA

Function	Return value	Math expression
Abs(x)	Absolute value of the x	$\|x\|$
Atn(x)	Arc-tangent of x in radians	$\tan^{-1} x$
Cos(x)	Cosine of x	$\cos x$
Exp(x)	Exponential of x	e^x
Int(x)	The integral part of x	$[x]$
Log(x)	Natural logarithm of x	$\ln x$
Round(x[, dp])	x rounded to dp decimal place dp is 0 by default if omitted	
Sgn(x)	Number indicates the sign of x -1 for $x < 0$, 0 for $x = 0$, 1 for $x > 0$	$\|x\|/x$
Sin(x)	Sine of x	$\sin x$
Sqr(x)	Square root of x	$\sqrt{x}$
Tan(x)	Tangent of x	$\tan x$

Remarks If the number is negative, then the function *Int* returns the first negative integer that is less than or equal to the number. For example, $Int(-8.3)$ will return -9. If a user wishes to return the first negative integer that is greater than or equal to the number, then he or she should use $Fix(-8.3)$, which will return -8.

Excel VBA also allows users to employ the worksheet functions of Excel, for example, *Average* and *Stdev*. To call the worksheet functions, use one of the following commands.

```
Application.FunctionName([arglist])
WorksheetFunction.FunctionName([arglist])
Application.WorksheetFunction.FunctionName([arglist])
```

For example, to calculate $\sin^{-1}(0.5)$, which is not provided in VBA's built-in function library but is included in Excel, we can use

```
x = Application.Asin(0.5),
```

which will return the value 0.5236 ($\approx \pi/6$) and be stored in x. Note that not all of Excel's worksheet functions can be used in VBA. For example, worksheet functions that have an equivalent VBA function, for example, sqrt and sin cannot be used. For a complete list of Excel's worksheet functions, please refer to Excel Help.

1.4.2 Multiple Linear Regression

A useful function for finding the ordinary least squares (OLS) estimate after defining a function in VBA and using the worksheet functions in Excel is given here. Recall that the general form of a multiple linear regression is given by

$$\begin{aligned} \mathrm{E}[\boldsymbol{Y}|\boldsymbol{X}] &= \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p, \\ \mathrm{Var}(\boldsymbol{Y}|\boldsymbol{X}) &= \sigma^2 \boldsymbol{I}_n. \end{aligned}$$

In matrix notation, it is written as

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{e},$$

where

$$\boldsymbol{Y} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, \ \boldsymbol{X} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1p} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & \dots & x_{np} \end{pmatrix}, \ \boldsymbol{e} = \begin{pmatrix} e_1 \\ \vdots \\ e_n \end{pmatrix}, \boldsymbol{\beta} = \begin{pmatrix} \beta_0 \\ \vdots \\ \beta_p \end{pmatrix}.$$

Also,

$$\mathrm{E}[\boldsymbol{e}] = \boldsymbol{0} \text{ and } \mathrm{Var}(\boldsymbol{e}) = \sigma^2 \boldsymbol{I}_n.$$

The OLS estimate is given by

$$\widehat{\boldsymbol{\beta}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y}.$$

Example 1.10 *Write a function with matrices X, Y as the parameters which returns an array containing the OLS estimate with* $array(i) = \beta_i$ *for* $i = 0, \dots, p$.

```
'Ordinary Least Squares

Public Function OLS(x As Variant, y As Variant) As Variant
Dim Beta As Variant
Dim i As Integer

With Application
  Beta = .MMult(.MInverse(.MMult(.Transpose(x), x)),
         .MMult(.Transpose(x), y))
End With

ReDim res(0 To UBound(Beta) - 1) As Double
For i = 0 To UBound(res)
    res(i) = Beta(i + 1, 1)
Next i

OLS = res
End Function
```

Specifically, Table 1.4 provides a list of worksheet functions and VBA built-in functions used in the OLS function.

TABLE 1.4 Functions Used in the OLS Function

Function	Nature	Return value
MMult (x, y)	Worksheet function	Returns the product of x and y
MInverse (x)	Worksheet function	Returns the inverse of x
Transpose (x)	Worksheet function	Returns the transpose of x
UBound (x)	VBA built-in function	Returns the largest subscript for an array x

1.5 RANDOM NUMBER GENERATION

Monte Carlo simulation requires the use of random numbers. VBA provides a built-in function, *rnd()*, that generates a sequence of pseudo-random numbers. Although they are pseudo-random by nature, they are sufficiently random for general applications in the sense that they satisfy certain characteristics.

The built-in function *rnd()* returns a uniform random number between 0 and 1, and the syntax is:

```
Randomize
x = Rnd()
```

See Table 1.5 for a complete description of *Randomize* and *Rnd*.

Simulation always involves the generation of random variables. In this section, the two main approaches to generating random variables are introduced: inverse transform and the acceptance–rejection method.

1.5.1 Inverse Transform

The inverse transform method makes use of the cumulative density function $F(x)$ of a random variable X. It is simple and easily implemented, but is limited to those random variables that have an analytic form for its cumulative density function.

TABLE 1.5 Description for the Random Number Generator

Procedure/Function	Description
Randomize([x])	The randomize statement is used to initialize the random number generator with an optional argument x as the seed. The system time is used as the seed if x is omitted. If randomize is not used, then the Rnd function (with no arguments) uses the same number as a seed the first time it is called, and thereafter uses the last generated number as a seed value
Rnd([x])	Return the next random number in the sequence if x is omitted. If x is not omitted, then Rnd([x]) returns the same number using x as the seed if $x < 0$; returns the most recent generated number if $x = 0$; and returns the next random number if $x > 0$

TABLE 1.6 Examples of Random Variable Generation Using Inverse Transform

Type	Description
Exponential with mean λ	$X = -\lambda \log(Rnd())$
Normal	$X = Application.NormSInv(Rnd())$

The algorithm of inverse transform is as follows.

1. Generate a standard uniform random variable $Y = \mathrm{U}(0, 1)$.
2. The required random variable is given by $X = F^{-1}(Y)$.

Table 1.6 presents examples of random variable generation using inverse transform.

1.5.2 Acceptance–Rejection Method

The acceptance–rejection method was proposed to address some of the limitations of inverse transform. In this method, suppose that Y with density function g can be simulated easily. Use Y as a basis to simulate $X \sim F$ by first generating Y from g and then accepting the value with probability $f(Y)/(cg(Y))$. More specifically, let c be such that

$$\frac{f(y)}{g(y)} \leq c \text{ for all } y.$$

Note that g should have tails heavier than those of the target distribution. The algorithm of the acceptance–rejection method is as follows.

1. Generate Y from density g.
2. Generate $U \sim \mathrm{U}(0, 1)$.
3. If $U \leq f(Y)/(cg(Y))$, then set $X = Y$.
4. Otherwise, return to step 1.

Example 1.11 *Student-t distribution is similar to normal distribution except that it has heavier tails. This feature is very useful in calculating Value at Risk. However, inverse transform is not possible for t distribution, and so the acceptance–rejection method is used. Simulation of a t distribution with two degrees of freedom is illustrated here. A double exponential with mean 1 is used as the proposed distribution.*

First find the maximum value of $f(y)/g(y)$ via differentiation, and c is found to be equal to 1.046267 (the maximum occurs at $y = 1$). The corresponding code for the acceptance–rejection method is:

```
Sub tdist()
Dim c As Double, p As Double
Dim Y As Double, X As Double, U1 As Double, U2 As Double
```

```
c = 2 * Exp(1) / ((2 + 1 ^ 2) ^ (3 / 2))

Do
'Generate exp(1)
Y = -Log(Rnd())

'Test if Y is accepted or rejected
U1 = Rnd()
p = 2 * Exp(Y) / (c * ((2 + Y ^ 2) ^ (3 / 2)))
Loop Until (U1 < p)
X = Y

'Generate the negative part of the distribution
U2 = Rnd()
If U2 < 0.5 Then
    X = −X
End If

End Sub
```

Generating normal random variables in an efficient way is very important in the simulation of asset prices. Inverse transform in Excel is not efficient as it is computationally intensive. A more efficient method of generating normal random variables is the Box–Muller transform, which states that if U_1 and U_2 are independent random variables that are uniformly distributed in the interval (0, 1], then

$$Z_0 = \sqrt{-2\log U_1}\cos(2\pi U_2)$$
$$Z_1 = \sqrt{-2\log U_1}\sin(2\pi U_2)$$

are independent standard normal random variables. The Box–Muller transform is coded in the following function *rGauss*.

```
Public Function rGauss() As Double
    Static store As Boolean, z As Double
        If store = True Then
            store = False
            rGauss = z
        Else
            z = Sqr(−2 * Log(1 - Rnd())) * Cos(Pi2 * Rnd())
            rGauss = z * Tan(Pi2 * Rnd(0))
            store = True
        End If
End Function
```

1.6 LIST OF FUNCTIONS DEFINED IN THE BOOK

To simplify the codes in the application programs, we have defined a number of constants, UDTs, and functions. This section briefly explains each of the constants, UDTs, and functions used in this book. For details of the code, please refer to the Excel files.

1.6.1 Constants

The following are the constants defined in the book.
xCall = 1
xPut = 2
xStraddle = 3

1.6.2 Types

Type BS_PathType
Type Garch_PathType
Type JD_PathType
Type Heston_PathType

1.6.3 General Functions

rGauss()
Parameters: none
Result: Returns a $\mathcal{N}(0, 1)$ random variable

rCGauss(LArray as Variant)
Parameters: LArray is the lower triangular matrix of the variance–covariance matrix of a vector of multivariate normal random variables
Result: Returns an array of normal random variables with LArray as the lower triangular matrix of the variance–covariance matrix

rGamma(alpha as Long, beta as Double)
Parameters: alpha is the shape parameter and beta is the scale parameter (where mean = alpha ∗ beta)
Result: Returns a Gamma(alpha, beta) random variable

rInvGamma(alpha as Long, beta as Double)
Parameters: alpha is the shape parameter and beta is the scale parameter
Result: Returns an InverseGamma(alpha, beta) random variable

rBeta(alpha as Long, beta as Long)
Parameters: alpha is the shape parameter and beta is the scale parameter
Result: Returns an Beta(alpha, beta) random variable

BS(*S*0 as Double, *K* as Double, rf as Double, *q* as Double, sigma as Double, *T* as Double, optionType as Integer)
Parameters: *S*0 is the initial stock price, *K* is the strike price, rf is the constant risk-free interest rate, *q* is the dividend yield, sigma is the volatility, *T* is the time to maturity in year, and optionType can be xCall, xPut, or xStraddle
Result: Returns the close-form solution of the option price for Black–Scholes formula

Max(Val1 as Double, Val2 as Double, optional Val3)
Parameters: Val1 is the first number, Val2 is the second number, and Val3 is optional
Result: Returns the maximum of the two (three) numbers

Min(Val1 as Double, Val2 as Double, optional Val3)
Parameters: Val1 is the first number, Val2 is the second number, and Val3 is optional
Result: Returns the minimum of the two (three) numbers

OLS(X as Variant, Y as Variant)
Parameters: *X* is the predictor matrix and *Y* is the response matrix
Result: Returns an array (base 0) of the least squares estimate for predictor *X* and response *Y*

Sort(sortArray as Variant, Optional lIndex as Long $= -1$, Optional rIndex as Long $= -1$)
Parameters: sortArray is the array you would like to sort
Result: The array inputted is sorted by the *Quicksort* algorithm
Remarks: This is in fact a *Sub* procedure, not a *Function* procedure

CDecom(VCMatrix as Variant)
Parameters: VCMatrix is a symmetric matrix
Result: Returns the lower triangular matrix of a symmetric matrix VCMatrix after Cholesky decomposition

Percentile(valArray as Variant, quantile as Double)
Parameters: valArray is the array for which you would like to find out a certain percentile
Result: Returns the percentile of valArray
Remarks: valArray need not be sorted before using this function

Average(valArray as Variant)
Parameters: valArray is the array for which you want to find the average of its elements
Result: Returns the average of valArray's elements

netDays(bDay as Date, eDay as Date)
Parameters: bDay is the beginning date and eDay is the ending date
Result: Returns the number of business days between bDay and eDay (measuring from the end of bDay to the end of eDay)

ND(z as Double)
Parameters: z
Result: Returns the density function of N(0, 1) at z

NCD(z as Double)
Parameters: z
Result: Returns the cumulative distribution function (CDF) of N(0, 1) at z

BS_Vega(ByVal $S0$ as Double, ByVal K as Double, ByVal rf as Double, ByVal q as Double, ByVal sigma as Double, ByVal T as Double)
Parameters: $S0$ is the initial stock price, K is the strike price, rf is the constant risk-free interest rate, q is the dividend yield, sigma is the volatility, and T is the time to maturity in years
Result: Returns the vega of the option under the Black-Scholes model

ImpVol(ByVal Price as Double, ByVal $S0$ as Double, ByVal K as Double, ByVal rf as Double, ByVal q as Double, ByVal T as Double, ByVal optionType as Integer)
Parameters: Price is the current market price, $S0$ is the initial stock price, K is the strike price, rf is the constant risk-free interest rate, q is the dividend yield, T is the time to maturity in years, and optionType can be xCall, xPut, or xStraddle
Result: Returns the implied volatility of the option under the Black-Scholes model

1.6.4 Asset Path Simulation Functions

BS_Path(A as BS_PathType)
Parameters: A is a user-defined data type (UDT) variable. Hence, the user has to specify parameters of the Black-Scholes model. An example can be found on page 74–75
Result: Returns a 2D array of asset path S(0 to m, 1 to n)

BS_CPath(A() as BS_PathType, VCMatrix as Variant)
Parameters: A is the UDT and VCMatrix is the variance-covariance matrix of the multi-asset Black-Scholes model. Chapter 3.7 presents an example
Result: Returns a 3D array of asset path S(0 to m, 1 to n, 1 to nAsset) according to the variance-covariance matrix VCMatrix. rf, m, n, dt, T will be read in A(1) only, and so other A(i) can be left empty with these parameters

Garch_Path(A as Garch_PathType)
Parameters: A is the UDT for the GARCH model. An illustrative example is given in Chapter 4.4 and Ch4.4_HSBC_RAN_GARCH(1,1).xls.
Result: Returns a 2D array of asset path S(0 to m, 1 to n) under GARCH

JD_Path(A as JD_PathType, Optional CalculateDrift as Boolean = True)
Parameters: A is an UDT. If Calculate Drift is set as False, then the drift specified in A will be used; otherwise, the risk-neutral drift will be used. An example using this function is given in Chapter 4.5
Result: Returns a 2D array of asset path S(0 to m, 1 to n) under the Jump–diffusion model with method 1.

JDExp_Path(A as JD_PathType, Optional CalculateDrift as Boolean = True)
Parameters: A is an UDT. If Calculate Drift is set as False, then the drift specified in A will be used; otherwise, the risk-neutral drift will be used. An example using this function is given in Chapter 4.5
Result: Returns a 2D array of asset path S(0 to m, 1 to n) under the Jump–diffusion model with method 2; if CalculateDrift is set as False, then the drift specified in A will be used; otherwise, the risk-neutral drift will be used

HestonVol_Path(A as Heston_PathType)
Parameters: A is an UDT for the Heston model. An example using this function is given in Chapter 4.3
Result: Returns a 2D array of asset path S(0 to m, 1 to n) under Heston, with moment matching

HestonVolQE_Path(A as Heston_PathType)
Parameters: A is an UDT for the Heston model. An example using this function is given in Chapter 4.3
Result: Returns a 2D array of asset path S(0 to m, 1 to n) under Heston, with the QE scheme

EO_Payoff(S as variant, K as double, optionType as Integer, optional m as long $= -1$)
Parameters: S is the stock price, K is the strike price, optionType is the type of options of either xCall, xPut, or xStraddle. Please refer to Example 3.1 on page 78
Result: Returns an array of the terminal payoff of the vanilla European option expiring at step m given the price path S

AO_Payoff(S as Variant, rf as Double, dt as Double, K as Double, optionType as Integer, optional m as long $= -1$)
Parameters: S is the stock price; rf is the interest rate; dt is the time step size; K is the strike price, optionType can be xCall, xPut, or xStraddle. Please refer to Example 3.2 on page 85
Result: Returns an array of the terminal payoff of a vanilla American option expiring at step m

EMartingale(S as Variant, rf as Double, q as Double, dt as Double)
Parameters: S is the original asset price path, rf is the risk-free rate, q is the dividend yield, and dt is the interval of each step

Result: Returns a 2D array of the asset price path after empirical martingale correction

1.6.5 Other Functions

ShowStatus(nStep as Long, tStep as Long, sStep as Integer)
Result: Show nStep/tStep in the status bar for each sStep; it can be disabled by setting HideStatus = True

ResetStatus
Parameters: None
Result: Reset the status bar

1.6.6 Remarks

Option Explicit

To force the declaration of all variables used, include the following as the first instruction in the VBA module.

```
Option Explicit
```

This statement causes the program to stop whenever VBA encounters a variable name that has not been declared. The variable must then be declared before proceeding.