1

PRELIMINARIES OF VBA

1.1 INTRODUCTION

This chapter introduces the elementary programming skills in Visual Basic for Applications (VBA) that we use for numerical computation of the examples in the book. Experienced readers can read this chapter as a quick review.

1.2 BASIS EXCEL VBA

Microsoft Excel is widely used in the financial industry for performing financial calculations. VBA is a common programming language linked to Excel and other Microsoft Office software that was developed to automatically control and perform repetitive actions. In this section, we guide readers on how to start a VBA in Microsoft Excel and give some popular algorithms for performing repetitions. In most cases, simple algorithms will be sufficient to perform the computations in the examples and exercises. We provide the illustrations in Excel 2010, although other versions can be set up in a similar way. For a comprehensive reference, readers are referred to other books.
1.2.1 Developer Mode and Security Level

For first-time users of VBA in Excel, it is more convenient to switch on the developer mode, where many of the VBA functions can be easily accessed. To open the developer mode, follow the following steps:

Click [File] → [Options] (Fig. 1.1) → [Customize Ribbon] (Fig. 1.2) → [Developer].

Figure 1.3 shows the ribbons at the top of Excel after switching on the developer mode. Macros refer to the codes executed in the VBA language. To execute the macros promptly, users are recommended to turn down the security level as follows:

Click [Macro Security] (Fig. 1.3) → Macro Settings [Enable all macros] (Fig. 1.4).

1.2.2 Visual Basic Editor

To edit the VBA codes, Microsoft provides a Visual Basic editor (VBE) in Excel for editing the macros. Macros are created, edited, and debugged in the VBE before being executed. A macro is usually created as a Sub or Function procedure that can perform automatic tasks, while a module consists of one or more macros. Similarly, a project has one or more modules. Sub and Function are reserved keywords in VBA. Users need to avoid using keywords when defining new variables. The codes in the

![Figure 1.1 Excel [Options].](image-url)
VBE are saved together with the Excel worksheet. In Excel 2010, these worksheets can be saved as .xlsm as an Excel Macro-Enabled Workbook file. To open and edit macros in VBE, follow the following procedure:

1. Open VBE: click the [Visual Basic] button under the developer mode (Fig. 1.3) or press ALT+F11.
2. Insert module: in the project window of the VBE, right-click on one of the worksheets, and select [Insert] → [Module] (Fig. 1.5).
3. Edit in VBE: type the codes in the panel on the right.
Figure 1.4  Macro security.

Figure 1.5  Visual basic editor.
4. Execute the program: in the VBE, select the module, and click the “Play” ribbon or press F5. In the Excel worksheet (Fig. 1.3), click the [Macros] button, choose the macro to be run, and click [Run]. A command button for a specific macro can be inserted in the Excel worksheet to facilitate the execution. See Section 1.2.4 for details.

1.2.3 The Macro Recorder

The macro recorder can record the actions that you perform in the Excel worksheet, such as building a chart or typing words, and transfer the actions into the macros in the VBE. This will be useful when you do not know how to code the actions and need to repeat them later. However, the macro recorder cannot handle codes that involve using the For loop or other repetitive loops and assigning variables. Different environments in Excel may generate different codes for the same task. Nevertheless, it can be a handy tool for learning new VBA codes. To record a macro, do the following:

1. Open the macro recorder: in the developer mode (Fig. 1.3), click [Record Macro].
2. Type the name to be used for the macro and a description of it so that you can recognize the macro next time (Fig. 1.6), then click [OK]. Note that the name should begin with a letter and contain no spaces or special characters.
3. Perform the tasks to be recorded; for example, type “Hello World” in cell A1.
4. Stop the macro recorder: click the [Stop Recording] button.
5. Go to the VBE to see the codes generated by the computer (Fig. 1.7).

![Figure 1.6 The macro recorder.](image)
The recorder creates a new *Sub* module in the VBE (Fig. 1.7). To run this macro, just click [Macro] at the top ribbon in Excel and select the macro you want to run. In the recorded codes, the words following the symbol `′` are not executed and serve only as comments. These comments are added to the codes to increase the readability for other users. It is a good programming habit to provide comments inside the codes to explain the details of the algorithm or define the variables. Comments can also be added by putting the keyword *Rem* at the beginning of the line.

### 1.2.4 Setting Up a Command Button

To run a specific macro in the Excel spreadsheet without selecting the macro procedure list, it is more convenient to designate a command button for each frequently used macro. To run the macro, the user just needs to press the command button. To insert a command button, follow the following procedure:

1. Click the [Insert] icon in the developer mode ribbon, and click the Command Button under [Form Controls] (Fig. 1.8).
2. Drag the mouse over a rectangle in the spreadsheet and release, then select the macro for the button.
3. To edit the button, left-click the name of the command button to change the name. Right-click the command button and select [Assign Macros] (Fig. 1.9) to change the macro.
4. Click on the command button to run the macro.

With this command button, users can quickly execute a macro.
Figure 1.8  Creating command button.

Figure 1.9  Assigning a macro to a command button.
1.3 VBA PROGRAMMING FUNDAMENTALS

1.3.1 Declaration of Variables

A variable in programming is the name of a place in the computer’s memory where some values or objects can be stored. To declare a variable in VBA, we 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. The name should not be the same as a VBA reserved word, such as `Sub`, `Function`, `End`, `For`, `Optional`, `New`, `Next`, `Nothing`, `Integer`, and `String`. However, VBA does not distinguish between cases.

For the `[As vartype]` part, it is optional to specify the type of variable. This is different from other programming languages, which require the programmer to explicitly define the data type of each variable used. However, if you do not specify the data type explicitly, VBA will be slower to execute and use memory less efficiently.

1.3.2 Types of Variables

Every variable can be classified into one of four basic types: string data, date data, numeric data, and variant data. The string data type is used to store a sequence of characters. The date data type can hold dates and times separately and simultaneously. The types used most frequently in this book are numeric data and variant data.

There are several numeric data types in VBA, and the details of each type are listed in Table 1.1. In general, it is more efficient to use the data type that uses the smallest number of bytes. This can significantly reduce the computational time for simulations.

The variant data type is the most flexible because it can store both numeric and non-numeric values. VBA will try to convert a variant variable to the data type that can hold the input data. Defining `[As vartype]` is optional part, so an undeclared type of variable will be stored as `Variant` by default.

A variant type variable can also hold three special types of value: error code, `Empty` (indicating that the variable is empty and is not equal to 0, `False`, an empty string, or other value), and `Null` (the variable has not been assigned to memory and is not equal to 0, `False`, an empty string, `Empty`, or other value).

The following codes show some examples of variable declaration statements:

```
Dim x As integer
Dim z As string
z = "This is a string"
Dim Today As Date
Today = #1/9/2014# 'defined using month/day/year format
```
### 1.3.3 Declaration of Multivariable

We use the following statement to declare several variables:

```vba
Dim x As Integer, y As Integer, z As Integer
```

However, the declaration that

```vba
Dim x, y, z As Integer
```

denotes z as the Integer type only, while x and y are declared as variant types. We can use shorthand (Table 1.1) to improve the cleanliness and readability of the program:

```vba
Dim x#, y#, z As Double
```

### 1.3.4 Declaration of Constants

Constants are declared in a `Const` Statement as follows:

```vba
Const interest_rate as Double = 0.02
Const dividend_yield = 0.02 'without declaring the constant type
Const option_type as String = "Put"
```

### 1.3.5 Operators

This section introduces the assignment operator, mathematical operators, comparative operators, and logical operators. The equal sign (=) is an assignment operator that
is used to assign the value of an expression to a variable or constant. An expression is a combination of keywords, operators, variables, and constants that yields a string, number, or object.

For example,

\[
\begin{align*}
y &= 3 \times 2 \\
y &= y \times 6
\end{align*}
\]

Then \(y\) is evaluated as 36.

Other common mathematical operators include addition (+), multiplication (*), division (/), subtraction (-), and exponentiation (^).

VBA also supports the same 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 lists the logical operators and their functions in VBA.

### 1.3.6 User-Defined Data Types

VBA provides the **Type** statement to allow users to create a more complex custom data type or user-defined data types (UDTs). The syntax for creating a UDT is as follows:

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

- **[Private|Public]**: (optional) this is Public by default. If it is declared as Private, the UDT can only be declared in the same module as the UDT.
- **typename**: (required) this is the name of the UDT, and it follows the standard variable naming conventions.
- **element_name**: (required) this is the name of the elements within a UDT, and it follows the standard variable naming conventions.
vartype: (required) unlike declaring ordinary variables, the elements within a UDT must be assigned a data type, which can be any variable type (including Variant) or a UDT.

UDT can be defined at the top of the module before any procedures. To refer to the subelements within the UDT, use the period (.) operator. See the following example for illustration.

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
    z As Coordinate
End Type

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

    ' Assigning the values
    p1.name = "A"
    p1.z.x = 3.5
    p1.z.y = 3.1

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

1.3.7 Arrays and Matrices

An array is a collection of variables of the same type that have a common name. The index numbering makes it easy for users to perform looping in repetitive tasks.

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

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

In this way, users can access the variables with varname(LowerIndex), varname(LowerIndex +1), ..., varname(UpperIndex).

If only the upper index is specified, that is,

```
Dim varname(UpperIndex) As vartype,
```
VBA will assume that 0 is the lower index. A multidimensional array can be declared as:

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

Note that both the lower index and the upper index must be a constant or a number. A dynamic array should be used for the variable index, which does not have a preset number of elements. The following statement declares a dynamic array.

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

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

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

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)
```

### 1.3.8 Data Input and Output

One advantage of using Excel VBA is that it can link the VBE and worksheet so that users can read in and print out data in the worksheet and execute the programs written in VBE. The following statements are usually used for input and output data, respectively.

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

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

where $i$ and $j$ denote the row number and the column number of a cell, respectively.

### 1.3.9 Conditional Statements

Conditional statements allow users to perform different tasks subject to different conditions. The two main conditional statements in VBA are `If-then-else` and `Select-Case` statements. There are two forms of the `If-then-else` statement: single-lined and multi-lined. Only one statement is allowed in the single-lined form, whereas many statements can be inserted in the multi-lined form. The syntax of the `If-then-else` statements is as follows:
In the conditional part of the statement, the user needs to specify an expression that can be evaluated as True or False. The comparative operators and logical operators in Table 1.2 can help to express more complex conditions.

The Select-Case statement is useful for choosing among three or more options and is a good alternative to the If-Then-Else statement. The syntax for Select-Case is as follows:

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

### 1.3.10 Loops

The use of the loops algorithm allows users to perform certain tasks several times. For-Next loops and Do loops are widely used in VBA programming. In particular, For-Next loops are frequently used in simulations. The syntax for a For-Next loop is as follows:

```vbnet
For counter = startValue To endValue [Step nStep]
    [statements]
    [Exit For]
    [statements]
Next counter
```
If the Step nStep part is omitted, 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.

For a Do Loop, the syntax is as follows:

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

Although both the do_condition and the 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 will continue as long as condition is True. For Until, the loop breaks once condition becomes True. Whether to use While or Until depends solely on the programmer’s preference, as the same task can be performed by either loop. However, whether to put 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 illustrates the uses of different loops to perform the same task.

Example 1.2 Use five different methods to print out 1 to 10 to 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
Loop
VBA PROGRAMMING FUNDAMENTALS

\[i = i + 1\]
Loop while \(i \leq 10\)

'Do Loop Method 4
\[i = 1\]
Do
    Cells(i, 1) = i
    i = i + 1
Loop until \(i > 10\)

1.3.11 Sub Procedures and Function Procedures

Large programs often need to be divided into smaller pieces for easier management and maintenance. In VBA, a procedure is basically a set of computer codes 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, while a Function procedure returns a value at the end of the procedure.

The syntax that defines a Sub procedure is as follows:

\[
[\text{Private}|\text{Public}] \ [\text{Static}] \ \text{Sub} \ \text{name} \ ([\text{arglist}]) \\
\{\text{statements}\} \\
\text{End Sub}
\]

*Private|Public*: (optional) the Sub is Public by default if Public or Private is omitted. Public indicates that the Sub is accessible by other Subs or Functions in all modules. Private indicates that the Sub is accessible only to the Subs and Functions in the same modules.

*Static*: (optional) static indicates that all local variables in the Sub are preserved at the end of the Sub. If Static is omitted, the values of the local variables will be reset each time the Sub ends.

*name*: (required) this is the identifier of the Sub. It follows the standard variable naming conventions and must be unique and cannot be the same as the identifier of other Subs, Functions, classes etc.

*arglist*: (optional) this is a list of variables representing the parameters that are passed to the sub when it is called. Multiple variables are separated by commas. If the procedure uses no arguments, a set of empty parentheses is required.

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

**Example 1.3** The following procedure, SubEx2, calculates \(\text{var1} + \text{var2}\) 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 following two statements where \( x, y \) can also be replaced by other constants or variables.

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

Instead of just specifying the name of the parameters, each parameter in \( \text{arglist} \) can be specified with the following syntax:

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

- **Optional**: (optional) this indicates that this parameter is optional and will take the `defaultvalue` as its value if it is omitted when the Sub is called.
- **ByRef|ByVal**: (optional) the parameter is passed `ByRef` by default. `ByRef` and `ByVal` indicate whether the parameter is passed by value or by address. When calling with `ByRef`, the memory address of the parameter is passed to the procedure and any change in the parameter value in the procedure will change the original parameter. For `ByVal`, a copy of the value of the parameter is passed so the original parameter will not be affected.
- **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 passed, which can be any of the variable types or a UDT. If the variable that is passed when calling the Sub does not match, an error "ByRef/ByVal argument type mismatch" is shown.
- **defaultvalue**: (optional) this is the value that the parameter will take when the parameter is not specified and the Sub is called.

**Example 1.4** The following codes demonstrate the difference between `ByRef` and `ByVal`:

```vba
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
```

Cell A1 shows that 2, as the change in the value of `var1` in `SubEx3`, actually changes the value of \( x \). Cell A2 shows that 1, as the change of the value of `var2` in `SubEx3`, does not affect the value of \( y \).
VBA also allows the user to create a `Sub` that takes 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 are stored in an array with the parameter’s name. To declare such a `Sub`, use

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

Unlike a `Sub` module, a `Function` can be used in an Excel spreadsheet as a user-defined function. The syntax that defines a `Function` is as follows:

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

For `Private|Public, Static, name, and arglist, a Function` is identical to a `Sub`. The only difference between the declaration of `Function` and `Sub` is that when defining `Function`, the user may want to define the return type `vartype`. Otherwise, the return type is `Variant` by default. To return a value for a `Function`, the user just needs to store that value in a variable with the same name as the function name. To call a `Function`, use one of the following statements:

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

Note that the first two statements are identical to those used for `Sub`, so one can treat `Function` as `Sub` if the return value does not matter. For the third statement, the return value will be stored in `z`.

As `Sub` cannot return a value, to accomplish certain tasks, it may be necessary to use global variables or pass the variables by reference. Example 1.5 calculates `var1 + var2` and outputs the result into cell A1, which is analogous to Example 1.3 using `Function`.

**Example 1.5** The following code calculates $3 + 4$ by calling `Function FuncEx4` and outputs the sum of the two numbers, 5, into cell A1.

```vba
Sub SubEx4()
    Cells(1, 1) = FuncEx4(3, 4)
End Sub

Function FuncEx4(var1 as integer, var2 as integer) as integer
    FuncEx4 = var1 + var2
End Function
```
TABLE 1.3  Common Built-In Math Functions in VBA

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Value</th>
<th>Math Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs(x)</td>
<td>Absolute value of the x</td>
<td></td>
</tr>
<tr>
<td>Atn(x)</td>
<td>Arc-tangent of x in radians</td>
<td>tan⁻¹x</td>
</tr>
<tr>
<td>Cos(x)</td>
<td>Cosine of x</td>
<td>cos x</td>
</tr>
<tr>
<td>Exp(x)</td>
<td>Exponential of x</td>
<td>e^x</td>
</tr>
<tr>
<td>Int(x)</td>
<td>The integral part of x</td>
<td>[x]</td>
</tr>
<tr>
<td>Log(x)</td>
<td>Natural logarithm of x.</td>
<td>ln x</td>
</tr>
<tr>
<td>Round(x[, dp])</td>
<td>x rounded to dp decimal place</td>
<td>dp is 0 by default if omitted</td>
</tr>
<tr>
<td>Sgn(x)</td>
<td>Number indicates the sign of x</td>
<td></td>
</tr>
<tr>
<td>Sin(x)</td>
<td>Sine of x</td>
<td>sin x</td>
</tr>
<tr>
<td>Sqr(x)</td>
<td>Square root of x</td>
<td>√x</td>
</tr>
<tr>
<td>Tan(x)</td>
<td>Tangent of x</td>
<td>tan x</td>
</tr>
</tbody>
</table>

1.3.12 VBA’s Built-In Functions

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

Remarks: If the input number is negative, then the function Int returns the first negative integer that is less than or equal to the number and the Fix function returns the first negative integer greater than or equal to the number. For example, Int(−8.3) returns −9, whereas Fix(−8.3) gives −8.

Excel VBA also allows users to use Excel worksheet functions such as Average, 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, one can use

x = Application.Asin(0.5).

This returns the value 0.5236 (≈ π/6) and is stored in x. Note that not all Excel worksheet functions can be used in VBA. In particular, worksheet functions that have an equivalent VBA function, such as sqrt and sin, cannot be used. For a complete list of Excel worksheet functions, please refer to the Excel help pages.