

Introduction to EXCEL Programming with Visual Basic for Applications (VBA)

By Gilberto E. Urroz, June 2013

What is EXCEL?

EXCEL (also known as *Microsoft EXCEL* or, simply, *MS EXCEL*) is the spreadsheet software available in the *Microsoft Office* office suite. The examples presented herein were developed using *EXCEL* 2013, however, they should be easily adapted to *EXCEL* 2010, or even *EXCEL* 2007.

Starting Excel 2013

Opening Excel 2013 opens the following page which lets you select the type of document to open:

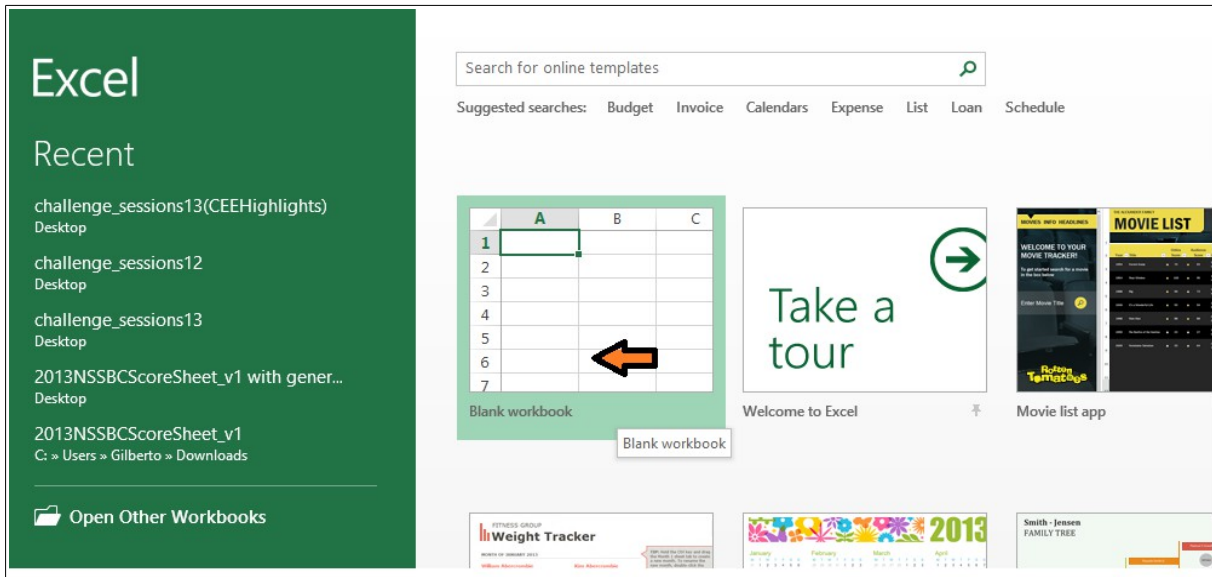


Figure 1. Options when opening *EXCEL*

Next, select “Blank workbook” (indicated by the orange arrow), to open a new worksheet, as shown below:

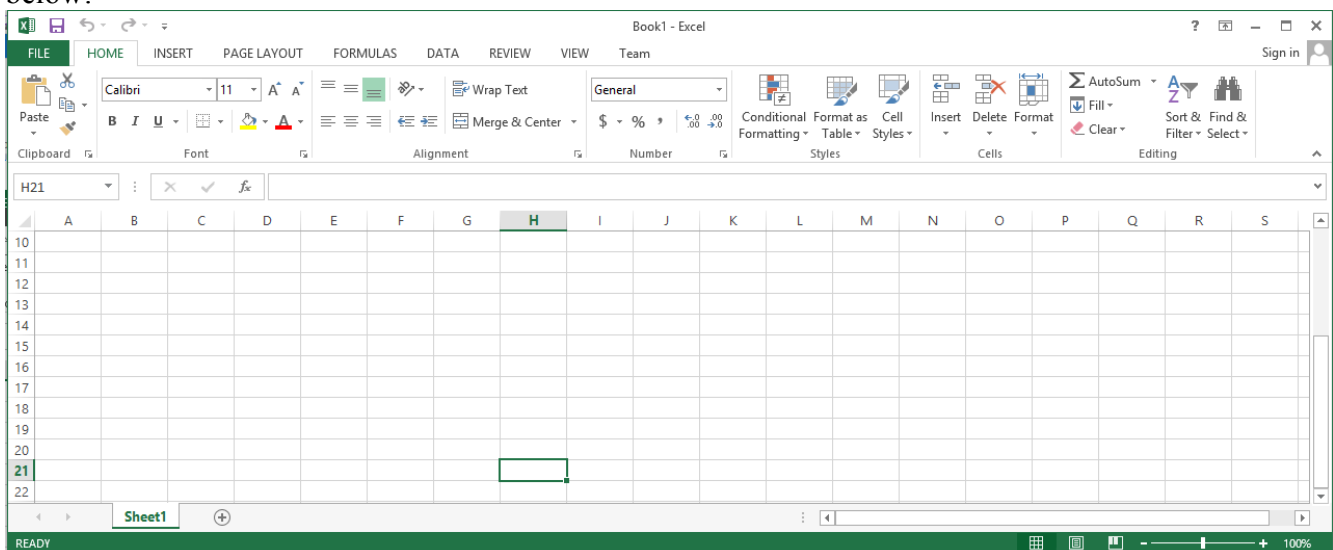


Figure 2. New *EXCEL* workbook with a new worksheet (*Sheet1*) showing the *HOME* tab in the *Ribbon*

The *EXCEL* 2007 version introduced the *Command Ribbon*, an area atop of the spreadsheet interface consisting of a number of tabs with specialized applications. Each tab in the *Ribbon* contains a number of buttons, fields, etc., related to the tab's theme. The default tab in *EXCEL* is the *HOME* tab, illustrated in Figure 2. The buttons and fields in the *HOME* tab are related mainly to editing functions (cut, paste, text format, text and background color), cell formatting (numbers, currency), inserting and deleting rows and columns, etc.

You are invited to explore other tabs in the *EXCEL Ribbon*. Here's a quick rundown of the various tabs' functions:

- *INSERT*: insert pictures, graphs, text boxes, equations, and symbols
- *PAGE LAYOUT*: set margins, page orientation, object positioning, etc.
- *FORMULAS*: insert worksheet functions, define variable names, show and evaluate formulas, calculation options
- *DATA*: sort data, perform What-If Analysis, link to Solver add-in
- *REVIEW*: spelling, comment spreadsheet, protect spreadsheets
- *VIEW*: page layout, zoom in and out, arrange screens, link to macros
- *DEVELOPER*: link to VBA IDE, record macro, insert controls
- *Team*: links to collaborative work online

Adding the *DEVELOPER* tab

Since the *DEVELOPER* tab is typically used for programming, it may not be readily available in your *EXCEL* application. In order to add the *DEVELOPER* tab proceed as follows:

- Click on the *FILE* tab
- Double click on the *Options* link in the left column
- Double click on the *Customize Ribbon* link
- In the right-hand side box, select the option *Developer*
- Press the [OK] button
-

The *DEVELOPER* tab should now be available. Click on it to see the following buttons:

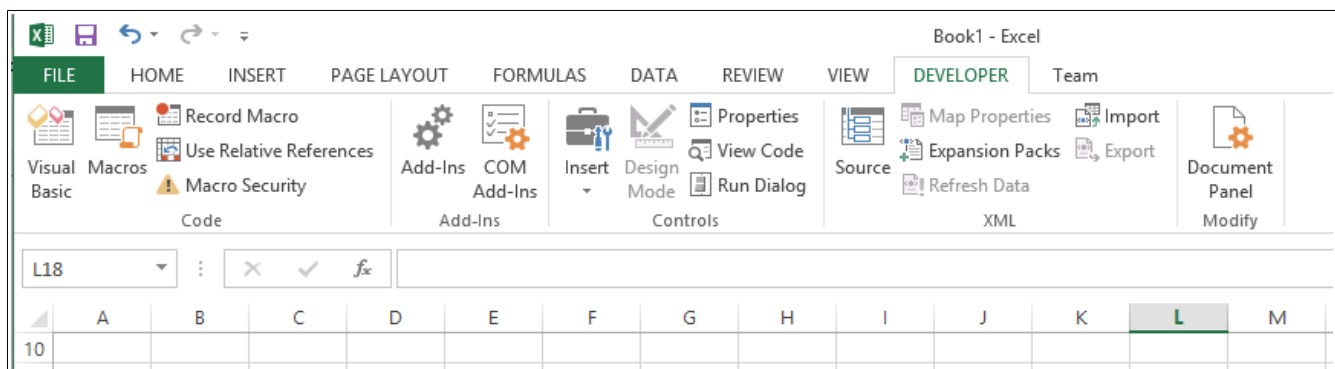



Figure 3. Buttons in the *DEVELOPER* tab.

The following buttons in the *DEVELOPER* tab, are of interest to programming *Excel* with *VBA*:

- *Visual Basic*: opens up the *VBA IDE*
- *Record Macro*: a sequence of actions executed by hand in *Excel* can be recorded into a *VBA* program called a *Macro*.
- *Add-Ins*: access the add-in collection (as shown in the top figure of page 2, above)
- *Insert*: opens up two sets of tools for activating programs or providing input-output: (1) *Form Controls*, and (2) *ActiveX Controls*
- Select the menu option *File > Options > Customize Ribbon*. There are two areas displayed in the resulting window: one showing commands and tabs that are available and another showing those that are selected for display.
- In the area on the right, ensure that the box next to *Developer* is checked. When you click OK, the *Developer* tab will appear in the ribbon.
- If you want to remove the *Developer* tab, repeat these steps but instead uncheck the *Developer* tab box.

Adding the *Solver* Add-In

In the listing of tab functions earlier on, we mentioned that the *DATA* tab includes a link to activate the *Solver Add-In*. The *Solver* add-in is very useful in solving equations and solving optimization problems, therefore, it is recommended that you add it to your *EXCEL* application if it is not available under your *DATA* tab. To install the *Solver* add-in follow these instructions:

- Click on the *FILE* tab
- Double click on the *Options* link in the left column
- Double click on the *Add-Ins* link
- At the bottom of the form you will see a field called *Manage*:. Make sure that the option *Excel Add-ins* (default value) is selected, and press the [ . . .] button.
- Select the *Solver Add-in* option in the *Add-Ins* window, → → press [OK].

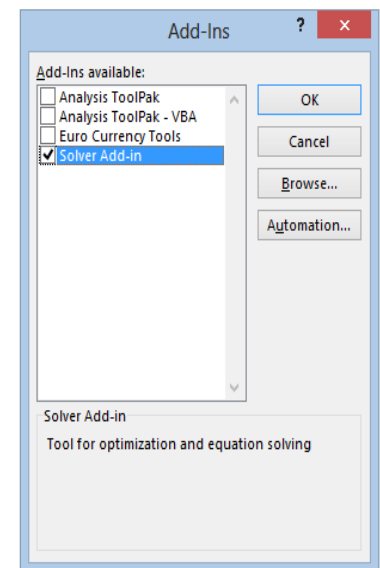


Figure 4. *Solver* Add-In.

To verify that the *Solver* Add-In has been added, click on the *Data* tab, and check that the button ? *Solver*, as shown below, is present. If not, repeat the procedure listed above.

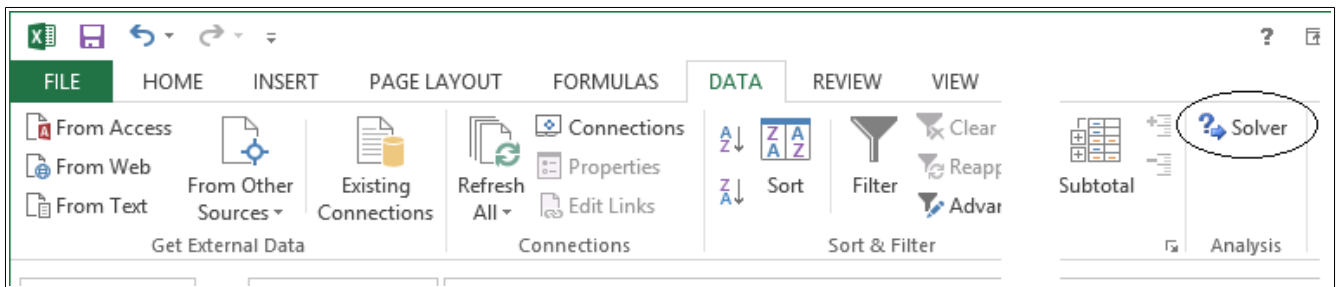


Figure 5. The *DATA* tab in the *Ribbon* showing the *Solver* add-in link

NOTE: Other add-in tools include:

- (1) *Analysis ToolPak*: useful for statistical calculations.
- (2) *Analysis ToolPak - VBA*: same as above, but allowing access to the VBA code for modifying the functions included in the *Analysis ToolPak*.
- (3) *Euro Currency Tools*: conversion of currency to *Euros*, the currency of the European Union.

If you are planning to perform statistical calculations, I recommend you add also one of the two *Analysis ToolPak* add-ins. To add any other add-in follow a similar procedure as that listed above for the *Solver* add-in.

Macros in *EXCEL* – The *VBA IDE*

A *macro* is a term used traditionally to refer to programs developed to manipulate data or perform tasks in a spreadsheet application. Macros in *EXCEL* can be created using the *Visual Basic for Applications' Integrated Development Environment (VBA IDE)*, available through the *Visual Basic* link in the *DEVELOPER* tab. The *VBA IDE*, as it first appears, is shown next.

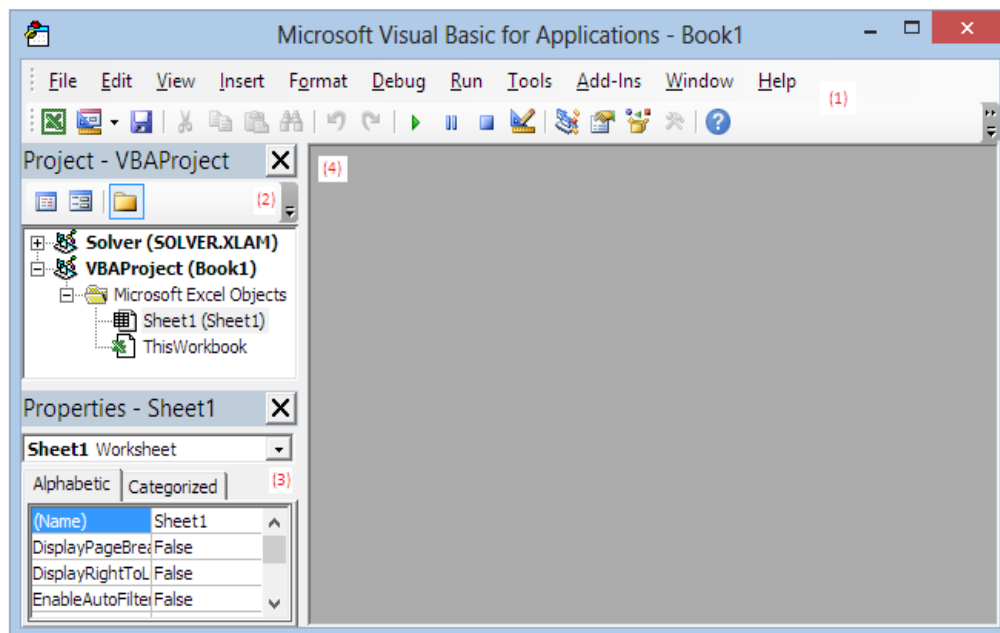


Figure 6. The VBA IDE (Integrated Development Environment).

In the VBA IDE of Figure 6, we distinguish 4 major areas:

- (1) The *Menu Bar*, followed by the *Toolbar*, both at the top
- (2) The *Project Explorer*
- (3) The *Properties Window*
- (4) The *Code Window*

Elements of the VBA IDE

The *Menu Bar* in (1), in Figure 6, contains menus that look familiar to any other *Windows OS* application, such as *File*, *Edit*, *View*, *Insert*, *Format*, *Window*, and *Help*, and some that are specific to programming in VBA, namely: *Debug*, *Run*, *Tools*, and *Add-Ins*. The contents of the *View*, *Insert*, *Run*, and *Debug* menus are shown in Figure 7.

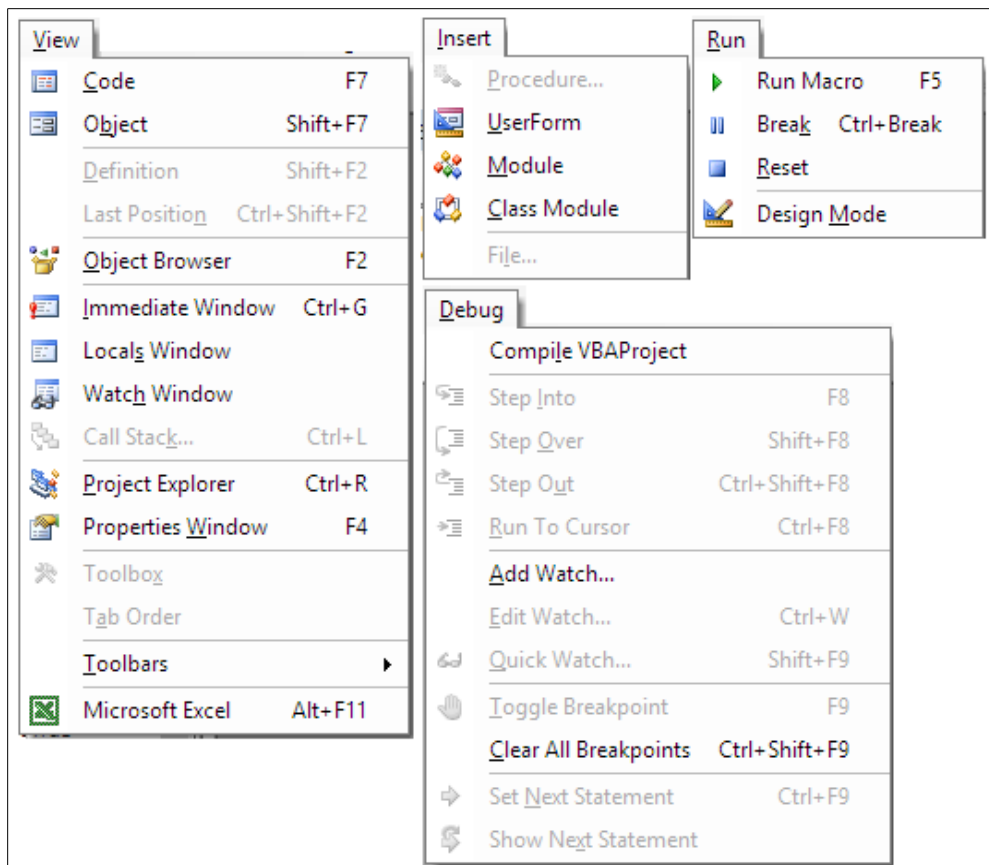


Figure 7. The *View*, *Insert*, *Run*, and *Debug* menus in the VBA IDE.

The *Toolbar* in the *VBA IDE* includes the buttons shown in the following figure.



Figure 8. The *Toolbar* in the VBA IDE.

The buttons shown in Figure 8 correspond to the following operations:

- (1) *View Microsoft Excel* (Alt+F11): switches to the associated Excel file
- (2) *Insert User Form*: user forms are used for input/output (more on these later)
- (3) *Save File* (Cntl+S)
- (4) *Cut* (Cntl+X)
- (5) *Copy* (Cntl+C)
- (6) *Paste* (Cntl+V)
- (7) *Find* (Cntl+F)
- (8) *Undo* (Cntl+Z)
- (9) *Redo*
- (10) *Run Macro* (F5): Run macro currently listed in the *Code Window*
- (11) *Break* (Cntl+Break)
- (12) *Reset*
- (13) *Design Mode*: In *Design Mode* you can design an interface
- (14) *Project Explorer* (Cntl+R)
- (15) *Properties Window* (F4)
- (16) *Object Browser* (F2)
- (17) *Toolbox*
- (18) *MS VBA Help*

The *Project Browser* lists the files currently open and available for editing in the *VBA IDE*. In the Project Browser of Figure 6, the files listed are:

- *Solver(SOLVER.XLAM)*: this is the Solver add-in
- *VBAProject(Book1)*: this is the only currently active EXCEL file, using the default name, Book1, and showing that it contains the following:
 - One worksheet: *Sheet1(Sheet1)*: The object *Sheet1* with name *Sheet1*
 - One workbook: *ThisWorkbook*

The *Code Window* in Figure 6 is empty. You can open code windows associated with a worksheet or with the workbook. Simple double-click on the corresponding link in the *Project Explorer*. Many a time, however, you want to write code in a code window associated with an object that is independent of a worksheet or of the workbook. In that case, you need to add a *Module* where the code will be written, as indicated in the following example.

Creating and running a new *Macro* in *EXCEL*

Use the following steps to create a simple *macro*:

1. Open a new *EXCEL* document, and name it, for example, *MyFirstMacro.xlsm* (i.e., save it as an *Excel Macro-Enabled Workbook*). NOTE: If you save it in a regular *Excel Workbook*, with suffix *xlsx*, macros will not be added to your file.
2. Select the *DEVELOPER* tab, and click on the *Visual Basic* button to open the VBA IDE.
3. Use the menu option *Insert > Module*. This creates a folder called *Modules* in the VBA IDE's *Object Explorer* and adds to it a module whose default name is *Module1*. This is shown in the following figure.

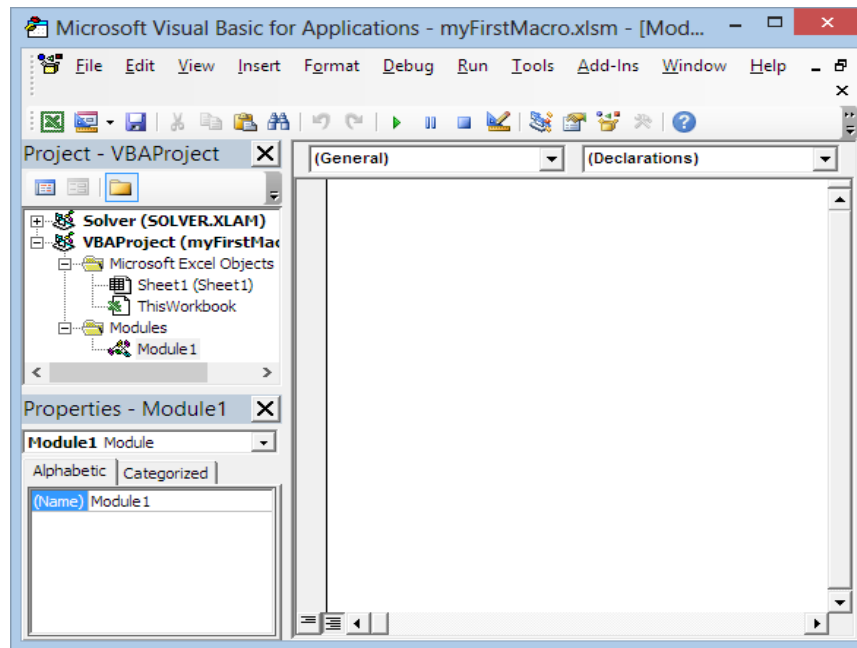


Figure 9. The VBA IDE after inserting module *Module1*.

4. When you added the module *Module1*, the *Properties Window* was automatically open to the properties of such module. At this point, you could change the name of the module from the default name, *Module1*, to, say, *MyMacrosModule* (or any other name that is significant to your project). If you change it to *MyMacrosModule*, then, the VBA IDE will now look as follows:

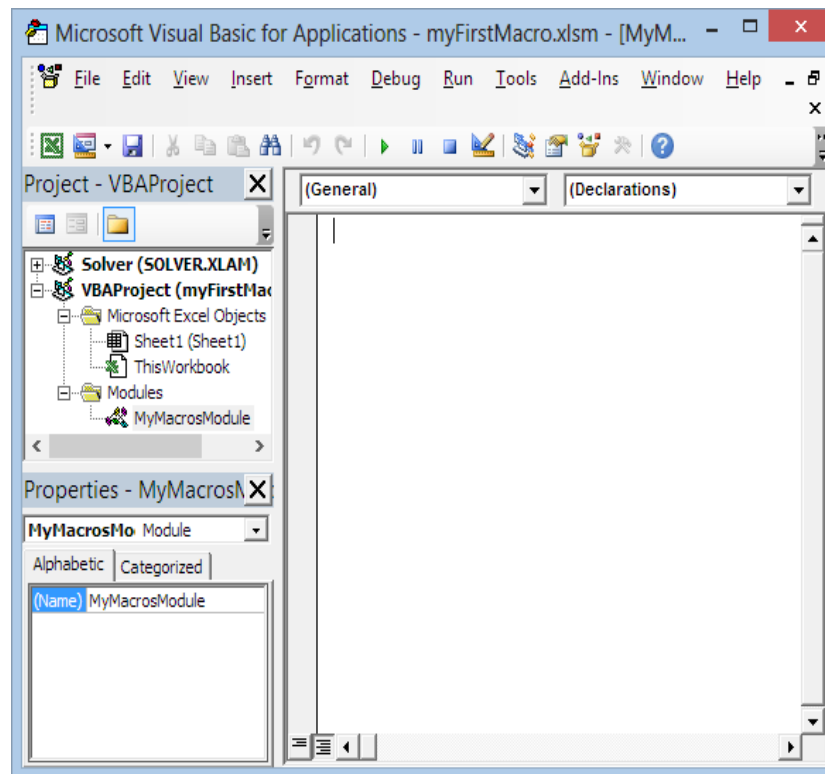


Figure 10. The VBA IDE after changing the name of *Module1* to *MyMacrosModule*.

NOTE: As indicated earlier, the term *IDE* stands for *Integrated Development Environment*. Figures 9 and 10, above, show the *Visual Basic for Applications' Integrated Development Environment (VBA IDE)*. An *Integrated Development Environment* is an application that typically includes an easy way to browse files, write and execute code, and debug code for errors. The use of IDEs facilitate *rapid application development (RAD)*, i.e., a faster way to get from coding to a finished program.

5. Enter the code as shown in Figure 11, below.

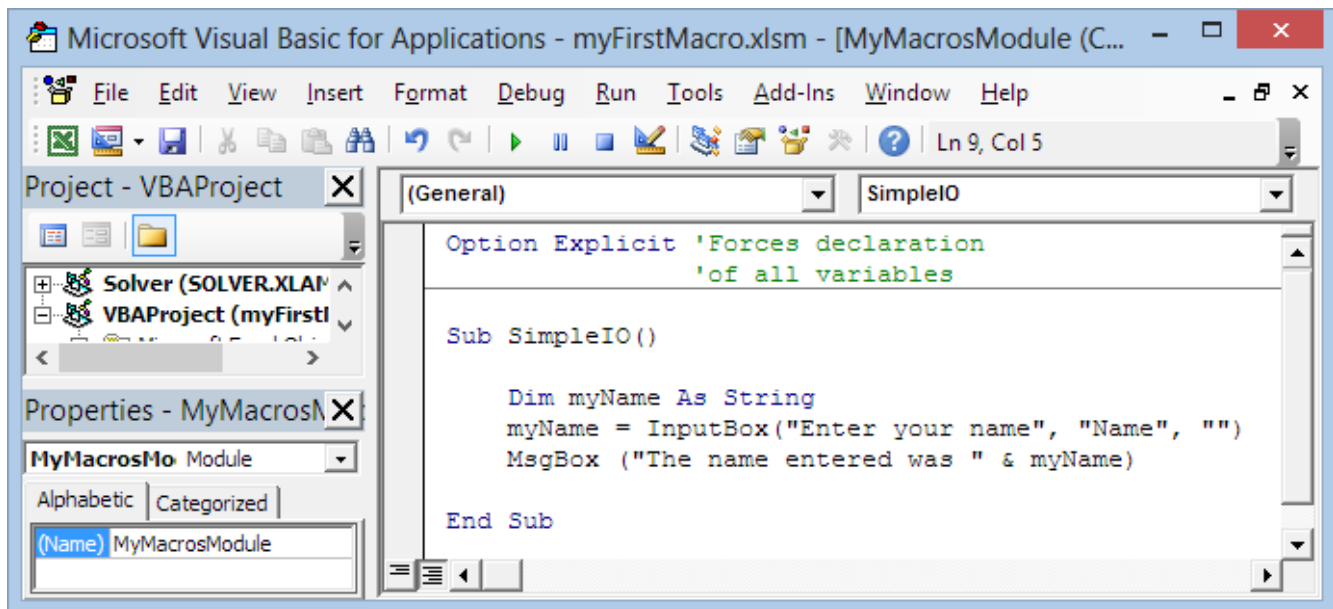


Figure 11. EXCEL VBA IDE showing a VBA Sub (subroutine) called *SimpleIO*

NOTES on the code for Sub SimpleIO:

- (1) A *Sub* (or subroutine) is one of the two types of *VBA* programs. The second type of programs is called a *Function*.
- (2) The line `Sub SimpleIO` marks the start of a subroutine that illustrates the use of simple input/output (IO) using VBA.
- (3) The line `End Sub` marks the end of the subroutine. The subroutine code are the lines contained between the `Sub` and `End Sub` lines.
- (4) The statement `Dim myName As String` defines a variable called *myName* as a *String*.
 - A *variable* is a name that represents a memory location in the computer where information can be stored.
 - A *string* is a collection of characters (letters, numbers, punctuation marks, other) enclosed between double quotes.
 - The word *Dim* (short for *Dimension*) is an old reference to a variable declaration that survives in *VBA* and other *BASIC* languages. *Dim* is a *reserved word* in *BASIC*, therefore, it cannot be used as a variable name (in other words, a variable to be called *Dim* is not allowed).

(5) *Built-in functions* (as opposite to *user-defined functions*) are functions pre-defined in the software. For example, mathematical functions such as *Sin* (sine), *Cos* (cosine), *Exp* (exponential), etc., are available in *VBA* (or *EXCEL BASIC*) for the programmer to use as built-in functions.

(6) The code in this example use two built-in VBA functions: *InputBox* and *MsgBox*.

(7) Function *InputBox* is used to enter data into a *VBA* program. The data is entered as a string in an input box produced by the function. The string thus entered is assigned to the variable *myName* through the *assignment operator* (the equal sign, =).

(8) The simplest form of the *InputBox* function is:

```
InputBox(prompt[,title][,default])
```

(9) Function *MsgBox* is used to show a string in a message box produced by the function.

(10) The simplest form of the *MsgBox* function is:

```
MsgBox(message_string)
```

(11). In this example, the *message_string* for the *MsgBox* function is given by the string:

```
"The name entered was " & myName
```

This is actually a string composed of two strings, the string constant (as opposite to a string variable): "The name entered was " and the string variable *myName*. The two strings are joined together, or *concatenated*, by the *concatenation symbol* &.

6. To see the code in action, press the *Run* button in Figure 11 (i.e., button (10) in Figure 8). The call to function *InputBox* produces the input box shown in Figure 12. Enter your name in the box, and press [OK].

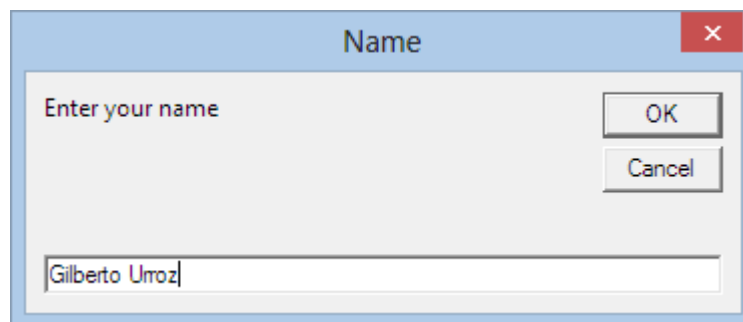


Figure 12. Sample input box produced by function *InputBox*

7. The call to function *MsgBox* produces the message box shown in Figure 13. Press the [OK] button to end the program.

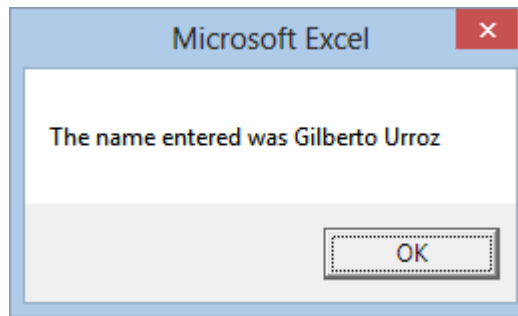


Figure 13. Message box produced by function *MsgBox*

8. Back in the *EXCEL* IDE, use *File > Save* to save the project (i.e., the *EXCEL* file you've been working on). At this point, you can click the *VBA IDE* window off, and return to the *EXCEL* spreadsheet to finish this simple programming exercise.

Changing Macro Security in EXCEL

When you open a macro-enabled *EXCEL* file that you may have received from another user there is always the risk that a macro contained in that file may be a *virus* (i.e., a program that damages your computer memory or alters the operation of your computer in a negative way) or other malicious program. To minimize computer virus infection in your computer, you can select the level of *macro security* that you would allow by default when opening macro-enabled *EXCEL* files.

To change the *macro security level* in you *EXCEL* application use the button *Macro Security* in the *DEVELOPER* tab. This opens up the following *Trust Center* window with the option *Macros Settings* preselected.

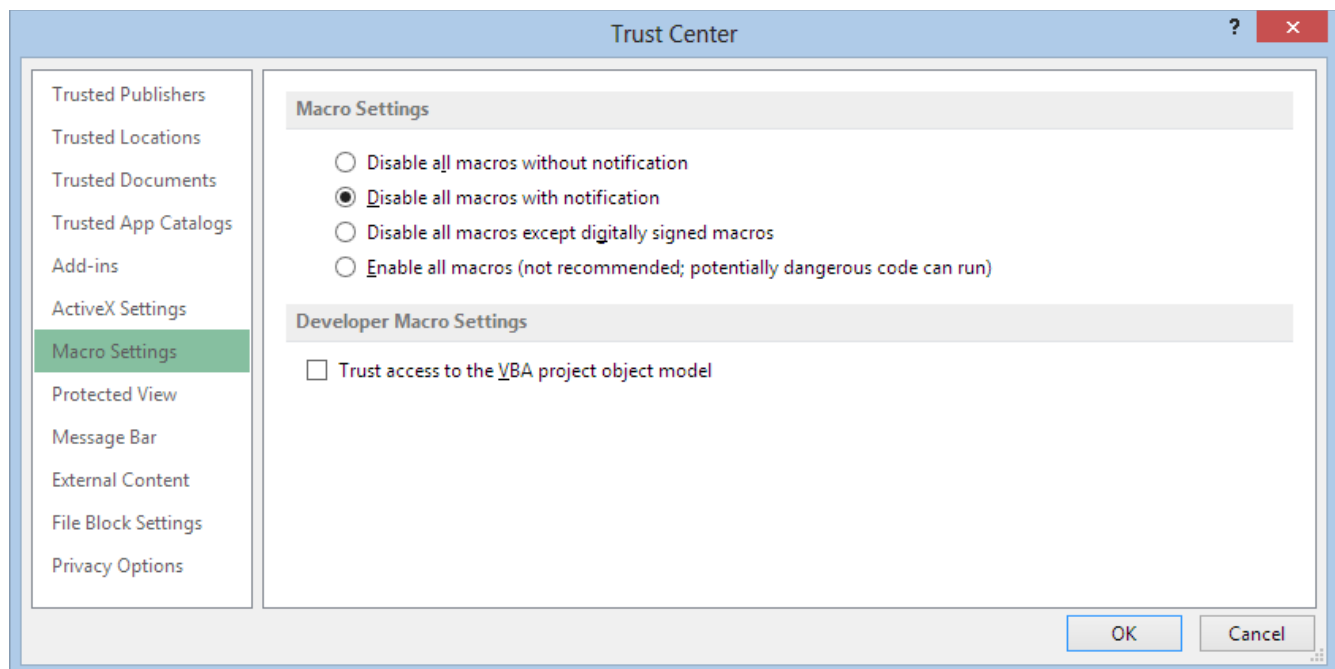
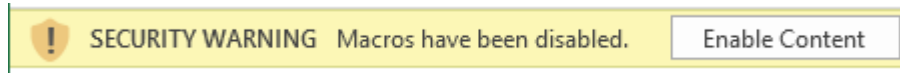


Figure 14. The *EXCEL Trust Center's Macro Settings* Options.

A commonly used setting for macro security is the one shown in Figure 14, namely, *Disable all macros with notification*. With this setting, when you open a macro-enabled *EXCEL* workbook, you get the following message above the spreadsheet row headings:



If you press the [Enable Content] button, then macros will be available for you to use. Alternatively, instead of pressing [Enable Content], you can access the VBA IDE, through the *DEVELOPER* tab, and review the macros available there. Then, you can decide, after looking up the code, whether to allow the activation of macros when you next open this particular *EXCEL* file.

A Simple program – Programming steps

A simple programming exercise with *InputBox* (for input) and *MsgBox* (for output) is developed in this section to introduce the idea of variables and data types, as well as other programming concepts, in VBA. The problem to be solved consists in converting a temperature given in Celsius (centigrade) degrees to its corresponding value in Fahrenheit degrees. The user will enter the temperature in Celsius degrees in a properly-prompted input box, and the program will return the converted temperature in Fahrenheit degrees in a properly- documented message box.

Typical programming steps

In general, to develop a program to solve a particular problem, the following steps are required:

1. *Program description*: describe the problem to be solved in as much detail as possible
2. *Identify input and output*: describe in detail the information the user needs to provide to the program, as well as the information the program must provide back to the user
3. *Develop a solution algorithm*: describe, in as much detail as possible, the steps necessary to accomplish a task, or sub-tasks, necessary to provide the desired solution. Many a times this is accomplished by writing the program tasks in English-like sentences, an approach referred to as *writing pseudo-code*.
4. *[Design and prepare the program interface*: this step depends on whether you are going to use a *EXCEL* spreadsheet or input form as the program interface. If you just want to run the program from the VBA IDE, there is no need to design an interface. In designing the interface you may decide to add buttons, drop-down lists, entry forms, etc. We will describe the use of *Form Controls* and *Dialogs* at a later section.]
5. *Code the algorithm*: Translate the algorithm developed in step 3 into code (e.g., VBA code or other code, in general). As you type the code, the VBA editor will correct syntax errors).
6. *Run the program with known data sets*: To verify that the program is working correctly, test it by running it with one or more known data sets. Check for run time errors at this steps.
7. *Debug the program*: If the program is producing run time errors, or it's giving the wrong results for known data sets, debug the program using the debugging tools in the VBA IDE.

Programming steps for the temperature conversion

1. *Program description*: Given a temperature in Celsius degrees (*temp_C*) calculate the equivalent temperature in Fahrenheit degrees (*temp_F*).
2. *Identify input and output*: Input: temperature in Celsius (*temp_C*). Output: temperature in Fahrenheit (*temp_F*).

3. *Develop a solution algorithm*: the following pseudo-code represents a possible algorithm for the solution:

```
Start program
Input      temp_C
calculate  temp_F = 9/5*temp_C + 32
Output     temp_F
End program
```

4. *[Design the interface*: No interface will be used. This step is skipped for this example].
5. *Code the algorithm*: In a *EXCEL* spreadsheet – call it, *TemperatureConversion.xlsm* (remember to save it as a *Macro-enabled Excel File*) –, create a module to be called *MyTemperatureModule*:
- In the *DEVELOPER* tab, click on the *Visual Basic* button to open the VBA IDE.
 - Select the menu option *Insert > Module* to insert a new module (default name: *Module1*)
 - In the *Properties Window* of *Module1*, change the name of the module to *MyTemperatureModule*

With the *MyTemperatureModule* module selected in the *Object Explorer* in the VBA IDE, enter the code as shown in Figure 15 (this is a translation of the pseudo-code of step 3, but including the *Explicit* option, comments, and variable declarations, e.g., *Dim temp_C As Double*, etc.). The *Sub* (subroutine) program, into which the pseudo-code of step 3, above, is translated, will be called *CelsiusToFahrenheit()*. Type the code as shown when trying this exercise in your own *EXCEL* spreadsheet.

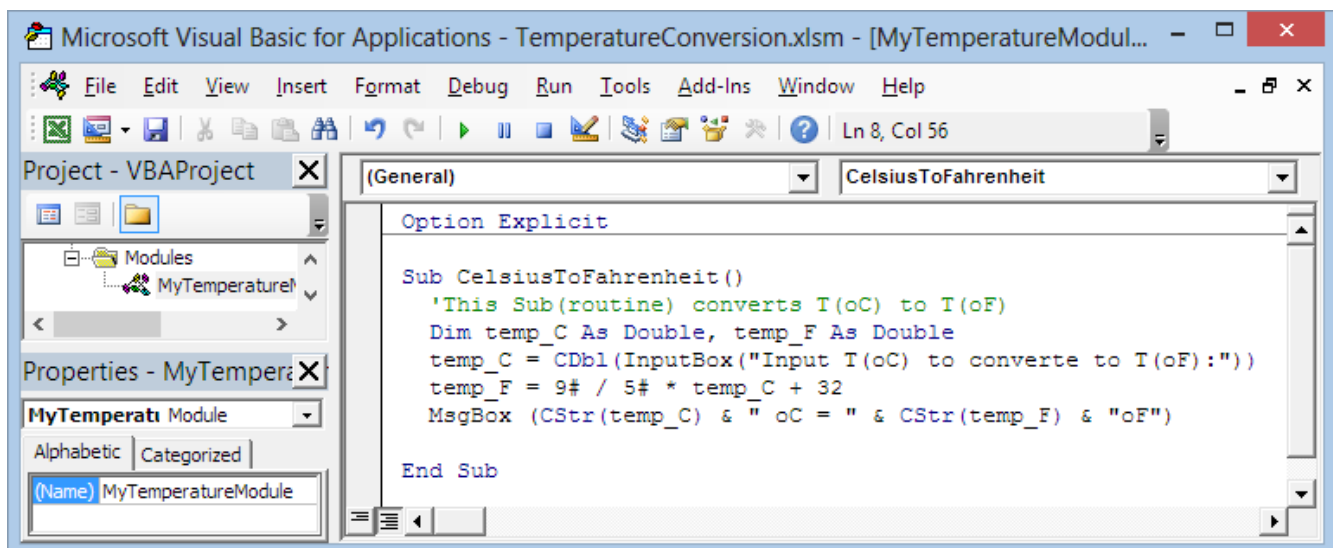
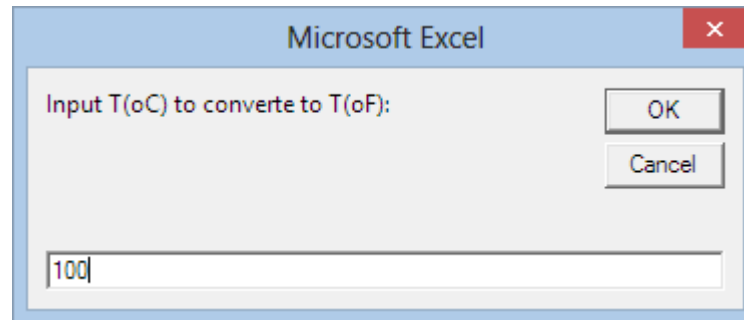
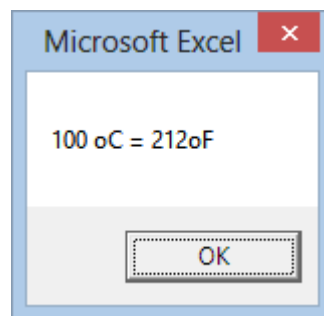


Figure 15. VBA IDE showing Celsius-to-Fahrenheit temperature conversion

6. *Run the program with known data sets:* From what we know of temperature scales we know that $0^{\circ}\text{C} = 32^{\circ}\text{F}$, $100^{\circ}\text{C} = 212^{\circ}\text{F}$, and $-40^{\circ}\text{C} = -40^{\circ}\text{F}$. Let's try these three values with our program. To run the program, in the *VBA IDE* of Figure 15 press the *Run Macro* button (see button (10) in Figure 8), or press [F5] in your keyboard. Enter a value of 100, for example, in the resulting input box, and press [OK]:

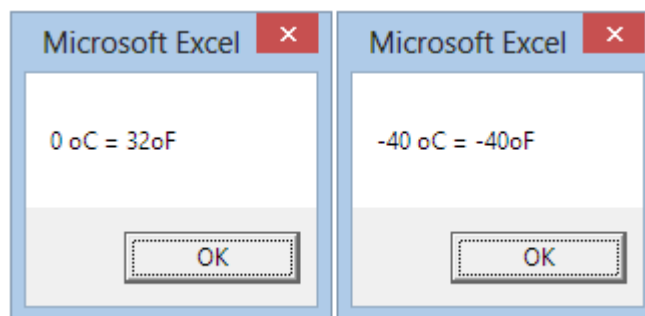


This results in the following output in a message box:



Press [OK] to end the program.

You can check that for 0°C and -40°C the corresponding Fahrenheit temperatures are indeed 32°F and -40°F , respectively:



7. *Debug the program:* This program is working as expected (you can check a few more conversion yourself), therefore, no debugging is needed.

The discussion of the details of the program of Figure 15 requires introducing the idea of numerical and string data types. We will come back to this program, but now, we'll review what we have learned about the *String* data type and introduce numerical data types using VBA.

Summary - Basic VBA data types: String

The program of Figure 11 was used to introduce the simple input-output functions *InputBox* and *MsgBox*, the idea of variables, and the *String* data type. The same functions, *InputBox* and *MsgBox* were used also in the program of Figure 15. This is what we know about strings:

- (1) A *string constant* is a collection of characters (letters, numbers, punctuation marks) enclosed within double quotes (“”) that is explicitly used in a program. In the example of Figure 15, there are three string constants used, namely: "Input T(oC) to convert to T(oF):", " oC = ", and " oF".
- (2) The particle *Dim* can be used to define a *string variable*, as done in the example of Figure 11 for the variable *myName*, e.g., `Dim myName As String`. String variables can be assigned string constants.
- (3) Strings can be *concatenated* (i.e., put together, one next to the other, as in a chain) by using the *concatenation operator* &. For example, in the case of Figure 11 there is one concatenation shown, in the argument of function *MsgBox*, namely:

```
MsgBox("The name entered was " & myName).
```

- (4) Strings, as shown in the *MsgBox* in (3), can be used to document input or output and make your program more readable.
- (5) Strings of characters that start with an apostrophe (') in the code window, are referred to as *comments*. Comments are used for documenting your code, and constitute non-executable *statements*. In other words, when *VBA* encounters an apostrophe, it ignores the rest of that particular line. A comment can be contained in a single line, or can be placed after an *executable statement* (*Dim* statements, assignment statements, input-output statements, etc).
- (6) The built-in function *CStr*(number) can be used to convert numerical data to string data for output purposes.

Basic data types: numbers

For everyday arithmetic operations we use a number system based on ten *digits*: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, i.e., a *decimal system*. Computers, on the other hand, at its most basic level, use a number system based on two *binary digits* (or *bits*): 0 and 1. Thus, to perform operations with numbers in the computer, decimal data is first converted to binary data. This conversion works in a straightforward manner if the number entered is an integer (i.e., numbers with no fractional, or decimal, part). However, converting decimal data to binary may cause loss of precision if the number has a *fractional part* (also referred to as a *decimal part* – however, for clarity, I will use the term *fractional* instead). This is due to the way that the conversion is effected (more on this later). Additional loss of precision can occur when operating upon the converted binary data, and even further while converting the binary data back to decimal data with a fractional part. In order to account for the data storage differences between integer numbers and numbers with a fractional part, programming languages distinguish between *integer data* and *floating-point data*.

Integer and floating-point numbers

Numbers representing things that can be counted one by one are referred to as *integer numbers*, or, simply, *integers*. For example, a class may have 23 students, or a bus may carry 15 passengers. Those

numbers, 23 and 15, are integer numbers. They have no decimal parts. On the other hand, if we measure the length of a steel re-bar using a measuring tape graduated in feet and tenths and hundredths of a foot, we may report such measurement as, for example, 5.25 ft. Or, if we read the temperature in a thermometer graduated in Celsius (or centigrade) degrees, we may read, for example, 22.5 °C. Numbers with a decimal part are referred to as *floating-point numbers* in the sense that they have a decimal point that could “float”, i.e., be moved up and down if using scientific (power-of-ten) notation.

Converting binary integers to decimal integers

Storage of integer data converted to binary is straightforward. The decimal system we use is a positional system in the sense that the position where we place a digit implies a multiplication with a power of 10. Thus, the number 325 is equivalent to:

$$325 = 3 \times 10^2 + 2 \times 10^1 + 5 \times 10^0 = 300 + 20 + 5$$

Similarly, a binary integer number can be expanded using the positional notation. Thus, to convert a binary number, say 101101_2 (remember, there are only two binary digits, or bits, 0 and 1), to its decimal equivalent, we use:

$$101101_2 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 32 + 0 + 8 + 4 + 0 + 1 = 45_{10}$$

The notation 101101_2 means that the number “101101” is of base 2, i.e., a binary number, while the notation 45_{10} means the number “45” is of base 10, i.e., a decimal number.

Converting a decimal integer to a binary integer

To convert a decimal number to binary, start by performing integer division of the number by 2, and then, the resulting quotient again by 2, and so on, keeping track of the residuals (which should be 0's and 1's only), until the resulting quotient is 0 or 1. The last division is, therefore, either $1 \div 2$, with residual = 1, or $0 \div 2$, with residual = 0. The residuals thus found are placed according to the position they were calculated, starting from the last and moving up to the first. For example, to convert the number 55_{10} to binary, we use:

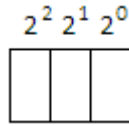
$$\begin{aligned} 55 \div 2 &= 27, \text{ residual} = 1 \text{ (i.e., } 55/2 = 27 + 1/2) \\ 27 \div 2 &= 13, \text{ residual} = 1 \text{ (i.e., } 27/2 = 13 + 1/2) \\ 13 \div 2 &= 6, \text{ residual} = 1 \text{ (i.e., } 13/2 = 6 + 1/2) \\ 6 \div 2 &= 3, \text{ residual} = 0 \text{ (i.e., } 6/2 = 3 + 0/2) \\ 3 \div 2 &= 1, \text{ residual} = 1 \text{ (i.e., } 3/2 = 1 + 1/2) \\ 1 \div 2 &= 0, \text{ residual} = 1 \text{ (i.e., } 1/2 = 0 + 1/2) \end{aligned}$$

At the end of this series of divisions we collect the residuals from the bottom to the top, i.e., $110111_2 = 55_{10}$. To check that the result is correct, try:

$$110111_2 = 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 32 + 16 + 0 + 4 + 2 + 1 = 55_{10}$$

Storing integers in the computer

Computer memory for storing integer numbers can be thought of collection of cells each containing a 0 or a 1 bit. For example, consider an old computer which can only store binary data as collections of three cells, as illustrated in the sketch below. We are going to refer to this collection of cells as this computer's *word*. We say, therefore, that this computer's word contains only three bits.



The smallest number that can be stored in this computer's word is $000_2 = 0_{10}$. If we add 1 to 000, in binary, we have, obviously, $001_2 + 000_2 = 001_2$, i.e., the binary zero behaves just like the decimal zero, it adds nothing to a number. However, $001_2 + 001_2 = 1_{10} + 1_{10} = 2_{10} = 10_2$. Thus, a basic binary addition table is as simple as (all base 2): $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$, $1 + 1 = 10$. Obviously the first three operations look exactly like the summation tables of base 10, but $1_2 + 1_2 = 10_2$. By adding 1, binary, starting with 000, we can figure out all the numbers that can be stored in a computer word of three bits, namely:

$000_2 = 0_{10}$, $000_2 + 001_2 = 001_2 = 1_{10}$, $001_2 + 001_2 = 010_2 = 2_{10}$, $010_2 + 001_2 = 011_2 = 3_{10}$, $011_2 + 001_2 = 100_2 = 4_{10}$, $100_2 + 001_2 = 101_2 = 5_{10}$, $101_2 + 001_2 = 110_2 = 6_{10}$, $110_2 + 001_2 = 111_2 = 7_{10}$

Thus, a 3-bit computer word can store $8 = 2 \times 2 \times 2 = 2^3$ integer decimal numbers (0, 1, 2, 3, 4, 5, 6, and 7). It is easy to demonstrate that a computer word of n bits can store 2^n decimal numbers, starting with 0. The following table shows the number of integer decimals that computer words of n bits, from $n = 2$ to $n = 8$, can store:

Table 1. Values that can be stored in a computer word of n digits

n	Number of decimal integer values that can be stored	Minimum integer value	Maximum integer value
2	4	0	3
3	8	0	7
4	16	0	15
5	32	0	31
6	64	0	63
7	128	0	127
8	256	0	255

Storing negative integer decimal numbers in a 3-bit word

What about the storage of negative numbers? Using our imaginary 3-bit word computer, we can make the first bit to represent a positive or negative sign, and the remaining 2 bits to represent the absolute value of an integer number. Thus, the bit in the first position, instead of representing the bit accompanying 2^2 , represents a positive number if that bit is 0, or a negative number if the bit is 1. Since you can only use the last 2 digits for the absolute value of a number, effectively our 3-bit word, has become a 2-digit word, with the available numbers to store (absolute values) being $00_2 = 0_{10}$, $01_2 = 1_{10}$, $10_2 = 2_{10}$, and $11_2 = 3$. The positive numbers that can be stored using our 3-bit word, with 0 in the first bit, are thus, the numbers $000_2 = 0_{10}$, $001_2 = 1_{10}$, $010_2 = 2_{10}$, and $011_2 = 3$. Since a 3-bit word can store up to 8 numbers, and we have figured out how to store the decimal numbers 0, 1, 2, and 3, the four remaining numbers that can be stored using a 1 in the first bit to represent a negative sign are -1, -2, -3, and -4. Next we explain how to store these numbers.

To store a negative number, take its absolute value and produce what is known as its *2's complement*. The latter consists of switching the bits in the absolute value of the negative number (this is called the *1's complement* of the absolute value of the number to be stored), and adding 1 to it. The number that results, i.e., the 2's complement of the absolute value, would be the binary representation of the negative number. For example, to store -3, take $|-3| = 3$, which is represented as 011_2 , form its 1's complement, i.e., 100_2 , and add 1 to it, resulting in $100_2 + 001_2 = 101_2$. Thus, 101_2 represents a -3 in our 3-digit *signed* word. In the earlier example, where no signs were included in the 3-bit word, the word is referred to as *unsigned*.

The opposite operation, i.e., figuring out which negative number is stored, given that the first bit is a 1, proceeds as follows. Start with the given number, and subtract 1. Since subtracting 1 is the same as adding -1, we need to find the binary representation of -1, with $|-1| = 1_{10} = 001_2$, its 2's complement is $110_2 + 001_2 = 111_2$. Thus, we need to take the original number and add 111_2 . If any *overflow occurs*, i.e., an extra 1 appears to the left of the first bit, this overflow is ignored. For example, if you are given the (obviously) negative number 101_2 , and add 111_2 , we get $101_2 + 111_2 = 1100_2$. The overflow 1 is ignored, resulting in the number 100_2 , whose 1's complement (i.e., switching bits) is $011_2 = 3_{10}$. Thus, the absolute value of the number of interest is 3, but we know it's got to be a negative number, thus, 101_2 represents the number -3.

In fact, the entire range of numbers that can be stored in our made-up 3-bit word is:

$$\begin{aligned} 000_2 &= 0_{10}, 001_2 = 1_{10}, 010_2 = 2_{10}, 011_2 = 3_{10}, \\ 100_2 &= -4_{10}, 101_2 = -3_{10}, 110_2 = -2_{10}, 111_2 = -1_{10} \end{aligned}$$

All this information is useful for understanding how the computer handles integer binary numbers that result from storing integer decimal numbers, whether signed or unsigned. However, unless you're programming computer chips at the binary level, you need not to worry about the binary representation of numbers. In a later section we'll talk a little bit about how floating-point data is stored in the computer, to illustrate the case that not all floating-point data can be converted to a precise equivalent in binary format. Next, we talk about units of memory used by computers.

Bits, bytes, ASCII characters

While *bits* (0's and 1's) are the smallest units of memory storage possible, typically we are most interested in a unit called *byte* and its multiples. An 8-bit word constitutes a *byte* (1 *byte* = 8 bits). A byte can store 256 integer decimal unsigned numbers, from 0 to 255. Bytes can be used to store characters in the computer. Conventions such as *ASCII* (American Standard Code for Information Interchange) use the numbers 0 through 255, stored in a byte, to represent characters that you can produce from your keyboard. Using *EXCEL* you can figure out the character corresponding to a particular integer value between 0 and 255 by using a function called *CHAR*. Click on an empty cell in your *EXCEL* spreadsheet, and type, for example:

= CHAR(156)

and press [ENTER]. The cell will turn to the character corresponding to the code 156, i.e., the letter *k*.

Using *EXCEL* you can produce all 256 ASCII characters through function *CHAR*. I prepared such a table as shown in Table 2, in next page.

Table 2. Table of ASCII characters

number	ASCII	number	ASCII	number	ASCII	number	ASCII
0	□	64	@	128	€	191	¿
1		65	A	129	◆	192	À
2		66	B	130	,	193	Á
3		67	C	131	f	194	Â
4		68	D	132	"	195	Ã
5		69	E	133	...	196	Ä
6		70	F	134	†	197	Å
7		71	G	135	‡	198	Æ
8		72	H	136	^	199	Ç
9		73	I	137	‰	200	È
10		74	J	138	Š	201	É
11		75	K	139	<	202	Ê
12		76	L	140	Œ	203	Ë
13		77	M	141	◆	204	Ì
14		78	N	142	Ž	205	Í
15		79	O	143	◆	206	Î
16		80	P	144	◆	207	Ï
17		81	Q	145	,	208	Ð
18		82	R	146	,	209	Ñ
19		83	S	147	"	210	Ò
20		84	T	148	"	211	Ó
21		85	U	149	•	212	Ô
22		86	V	150	—	213	Õ
23		87	W	151	—	214	Ö
24		88	X	152	~	215	×
25		89	Y	153	™	216	Ø
26		90	Z	154	§	217	Ù
27		91	[155	>	218	Ú
28		92	\	156	œ	219	Û
29		93]	157	◆	220	Ü
30		94	^	158	ž	221	Ý
31		95	˘	159	ÿ	222	Þ
32		96	`	160		223	ß
33	!	97	a	161	ı	224	à
34	"	98	b	162	¢	225	á
35	#	99	c	163	£	226	â
36	\$	100	d	164	¤	227	ã
37	%	101	e	165	¥	228	ä
38	&	102	f	166		229	å
39	'	103	g	167	§	230	æ
40	(104	h	168	"	231	ç
41)	105	i	169	©	232	è
42	*	106	j	170	ª	233	é
43	+	107	k	171	«	234	ê
44	,	108	l	172	¬	235	ë
45	-	109	m	173	-	236	ì
46	.	110	n	174	®	237	í
47	/	111	o	175	—	238	î
48	0	112	p	176	°	239	ï
49	1	113	q	177	±	240	ð
50	2	114	r	178	²	241	ñ
51	3	115	s	179	³	242	ò
52	4	116	t	180	,	243	ó
53	5	117	u	181	µ	244	ô
54	6	118	v	182	¶	245	õ
55	7	119	w	183	·	246	ö
56	8	120	x	184	¸	247	÷
57	9	121	y	185	¹	248	ø
58	:	122	z	186	º	249	ù
59	;	123	{	187	»	250	ú
60	<	124		188	¼	251	û
61	=	125	}	189	½	252	ü
62	>	126	~	190	¾	253	ý
63	?	127	□	191	¿	254	þ
64	@	128	€	192	À	255	ÿ

Some of the ASCII characters, particularly those between the values of 0 and 32 do not show in the list because they represent control characters, such as line feed, tab, etc., that do not produce printable characters, but produce some control of the lines produced, particularly for output. The decimal digits (0-9) correspond to ASCII characters between 48 and 57, while lower case English letters are characters numbers 97 through 122. Upper case English letter correspond to ASCII numbers 65 through 90. For more details on ASCII characters visit <http://en.wikipedia.org/wiki/ASCII>.

Kilobytes, Megabytes, Gigabytes, Terabytes, etc.

Thus, *bytes* (= 8 *bits*) are used to store characters. Collections of bytes can be used to store characters and numbers. The prefix *kilo* in the International System of Units (S.I., formerly known as the *metric system*) represents 1000 units. Since binary data increases naturally in power of 10, and the number $2^{10} = 1024$ is close to 1000, the term *kilobyte* is used to represent 1024 *bytes*. The following multiples of bytes are commonly used in references to modern computers:

1 Kilobyte (Kb, KB) = 1024 bytes
1 Megabyte (Mb, MB) = 1024 KB
1 Gigabyte (Gb, GB) = 1024 MB
1 Terabyte (Tb, TB) = 1024 GB
1 Yottabyte (Yb, YB) = 1024 TB

To give you an idea of what you can store in those memory units, consider the following:

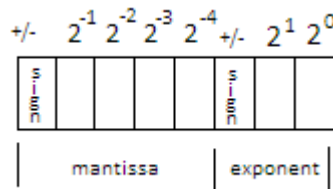
- To store 1 page of text you need about 3 KB
- To store 500 pages of text (a book) you need about 1.5 MB
- To store a typical black-and-white picture (1280 × 1024 pixels) you may need about 164 KB
- To store a typical color picture (1280 × 1024 pixels) you may need about 4 MB
- A CD ROM can store about 600 MB, or 400 books, or 150 color pictures
- A DVD ROM can store about 4.7 GB, or 3133 books, or about 1175 color pictures
- Laptop computer hard disks run in the order of 250 GB to 750 GB
- External hard disks could contain from 500 GB to 2 TB
- iPads and other pads are sold with memories of 16 GB and 32 GB, some other pads can accommodate up to 64 GB
- Older MP3 players (e.g., the original huge iPods, or the Microsoft ZUNE, now deceased) had memories of 32, 40, 80, and up to 120 GB
- More recently, MP3 players, such as the mini iPod, provide 16 GB or 32 GB
- SD Cards for your cameras come in sizes of 32 or 64 GB
- USB thumbdrives can be found with capacities of 1, 2, 4, 8, and 16 GB

Laptop and desktop personal computers use what is called *RAM* (*Random Access Memory*) memory chips to temporarily store data used by the computer *Central Processing Unit* (*CPU*, i.e., the “brain” of the computer). RAM memory is expensive compared to storage device memory (hard disk, thumbdrives), therefore, the typical sizes of RAM memory in most current personal computers is limited to 2, 4, 6, or 8 GB, although, for the right price you can get 16 GB of RAM installed in your computer.

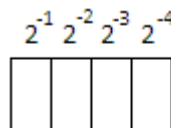
Storing floating-point numbers

A floating-point number, as indicated earlier, contains a fractional part. Thus, it can easily be converted into scientific, or power-of-ten notation, e.g., $1254.35 = 0.1254 \times 10^6$. In this scientific-notation number, the number 0.1254 is called the *mantissa* and the value of 6 is called the *exponent* (it is understood that is the exponent of an integer power of ten). A relatively small negative floating-point number would require, in addition, space for two signs, one for the mantissa and one for the exponent, e.g., the number $-0.0002356 = -0.2356 \times 10^{-3}$. Thus, to store a floating-point point number we need a computer word that includes one bit for the sign of the mantissa, some bits to store the mantissa, one bit to store the sign of the exponent, and some bits to store the exponent. The exponent is always an integer number, therefore, storing the exponent is similar to storing a signed integer. The mantissa, on the other hand, is typically a decimal number between 0 and 1. Therefore, the bits assigned to store the mantissa are associated always with negative powers of 2.

Consider, as an example, a word with a total of 8 bits (i.e., a byte), with one bit for the sign of the mantissa, one bit for the sign of the exponent, 2 bits for the exponent, and 4 bits to store the digits in the mantissa. Such an arrangement is shown in the following sketch:



Since we know that the mantissa is converted to a decimal number between 0 and 1 (the 1 is not included), we need to store only the decimal numbers to the right of the decimal point. But, how many numbers can be store? If there are only 4 bits to store the mantissa, the positions of these bits correspond to the powers of 2 illustrated in the following sketch:



With 4 bits, we can store a total of $2^4 = 16$ numbers, resulting from storing the binary numbers 0000_2 (i.e., 0.0) to 1111_2 , where the subindex -2 means we are dealing with negative powers of two. Thus, the mantissa alone allows us to store 16 numbers only. If we add the fact that the exponent, including the sign bit, contains three bits, i.e., it can store $2^3 = 8$ numbers, we realize that we can, in practice, store a total of $16 \times 8 = 128$ numbers. Actually, accounting for the bit containing the sign of the mantissa the total number of floating-point numbers that we can store in this byte of memory is 256. Thus, this very limited-size computer word can only store 256 of the infinitely large number of floating-point (real numbers) that exist. The results of this calculations point out the fact that computer memory is of a finite nature, while floating-points are essentially infinity in number. Therefore, regardless of the computer word size, we cannot possible store all floating-point numbers that we may encounter. In other words, some floating-point numbers, when stored in a computer word, must necessarily be truncated or rounded off. This is a *loss of precision* in memory storage of floating-point numbers that is inherent to the digital nature of computer memory.

To improve precision in the storage of computer numbers, modern computers use computer words that are 32 and 64 bit long. Most modern laptop computers, for example, are now provided with 64 bit long words, each one of which could store, in principle, up to 18446744073709551616 numbers: much better than the 256 numbers that we can store in a single byte.

Storing a floating-point number in the given size of your computer's word, is referred to as storing a *single-precision* number. Actually, some computer languages (such as FORTRAN 2013) lets you code the size of the memory word to be used in your programs. Computer programs, such as *VBA*, allow the user to select *single-precision* or *double-precision* storage. In double-precision storage, two computer words are used to store each floating-point data value. This way, the precision of the stored numbers is improved, by carrying much more decimal digits in a single memory location. Single-precision and double-precision floating-point data will be discussed in more detail in a subsequent section.

Basic VBA data types: *Integer* and *Long*

The VBA's *Integer* data type was originally intended to store signed 16-bit (2-byte) integer numbers that may range in value from between -32,768 and 32,767, while the *Long* data type could store 32-bit (4-byte) data values ranging from -2,147,483,648 to 2,147,483,647. The most recent versions of VBA, however, have done away with the distinction. Thus, whether you declare your data type as *Integer* or *Long*, the result will be the same: to store signed integer values in a 32-bit (4-byte) format for values ranging between -2,147,483,648 to 2,147,483,647. That represents a very large range of integers indeed, and it should be enough for most applications. If you need to use values with larger absolute values, you may need to switch to *Double* precision data (see below).

NOTE: since there is no longer a real distinction between *Integer* and *Long* data types, I suggest you declare all your integer values as *Integer* (a more descriptive term, than *Long*).

To declare *Integer* or *Long* data, use the *Dim* statement as in this examples:

```
Dim iCount As Integer
Dim jCount As Integer, kCount As Integer
Dim myCount As Long
Dim myCount01 As Long, myCount02 As Long
```

When you declare an *Integer* (or *Long*) variable, its value gets initialized to 0. Operations with *Integer* (or *Long*) data types (addition, subtraction, multiplication, division, power) will produce *Integer* (or *Long*) data. If you mix *Integer* (or *Long*) data with *Single* or *Double* data, the result will be *Single* or *Double* data.

Basic VBA data types: *Single* and *Double*

VBA's *Single* data type is intended for holding signed IEEE¹ 32-bit (4-byte) single-precision floating-point numbers ranging in value from $-3.4028235 \times 10^{38}$ through $-1.401298 \times 10^{-45}$ for negative values and from 1.401298×10^{-45} through 3.4028235×10^{38} for positive values. Single-precision numbers store an approximation of a real (floating-point) number.

¹ IEEE (pronounced "I triple E") stands for the *Institute of Electrical and Electronic Engineers* (<http://www.ieeeusa.org/>). The IEEE writes many of the specifications used in computer data storage. Thus, the term *IEEE 32-bit single-precision floating point numbers* refers to the IEEE specification for that data type.

My advice is to never use single-precision data, since there is an inherent loss of precision when storing floating-point data in binary format, anyway, which increases as more and more operations are performed with those data values. Thus, you want to have the largest amount of precision in your floating-point data storage. Single precision used to be justified in order to save memory space in earlier computers. For most, except numerical-intensive computations (typically performed in parallel computers and super-computers), most modern personal computers would have plenty of memory space to allow using all floating-point values involved as *Double*-precision data.

The *Double-precision*, or simply, *Double*, data type is intended to store signed IEEE 64-bit (8-byte) double-precision floating-point numbers that range in value from $-1.79769313486231570 \times 10^{308}$ through $-4.94065645841246544 \times 10^{-324}$ for negative values and from $4.94065645841246544 \times 10^{-324}$ through $1.79769313486231570 \times 10^{308}$ for positive values. As you can see, the range of floating-point data that can be stored in *Double* data memory locations is much larger than that allowed for *Single* data types.

To type in or enter an integer value as *Double* data, add a decimal point to the right of the integer value, or, even better, add .0, e.g., 323. or 323.0. To enter numbers in power-of-ten notation type an *E* instead of using powers of ten. For example, the number 5645.84×10^{-12} can be entered as 5.64584E-15, or 56.458E-15, or 564.58E-14, etc.

Combining *Integer* or *Single* data with *Double* data in arithmetic operations will produce a *Double* result. The *Integer* and/or *Single* data will be converted to *Double* internally, in order to perform the calculations.

The example of Figure 15 shows the use of two *Double* variables:

```
Dim temp_C As Double, temp_F As Double
```

The same example shows one operation involving *Double* data:

```
temp_F = 9#/5C#*temp_C+32
```

In this assignment statement, I typed the constants 9.0 and 5.0 as floating-point data, and VBA then translated them as 9#/5#. This is a notation originally used in BASIC to emphasize that these numbers, although, in principle, integers, are actually entered as floating-point numbers.. Since they're combined with *temp_C*, which is a *Double* variable, the result of the operation '9#/5#*temp_C' will be *Double*. Adding the integer constant, 32, to the *Double* result found above, will produce a *Double* value for the right-hand side of the assignment statement, and such result will then be stored in the *Double* variable *temp_F*.

NOTE: If I had written the division '9#/5#' as '9/5', VBA would have interpreted this division as that of two integers, and truncation of the result may have been effected. In VBA, most likely $9/5 = 1$, while, $9.0/5.0 = 1.8$. Thus, my recommendation is to use a decimal point, or even add .0, to any integer value used in a floating-point operation.

Hierarchy of operations

Consider the following equation:

$$C = 1.32 + \frac{m}{d} - \frac{k \cdot r^2}{5}$$

To calculate the value of C in VBA, you would declare all variables (C , m , d , k , and r) as *Double*, assign values to them, and write out the proper VBA assignment statement in code:

```
C = 1.32 + m/d - k*r^2/5.0
```

The term *hierarchy of operations* refer to the order in which arithmetic operations (addition, subtraction, multiplication, division, and power) get performed when evaluating an expression in VBA. In order to calculate a VBA arithmetic expression, such as in the right-hand side of the expression immediately above, VBA scans the expression from left to right, and perform arithmetic operations in this order:

1. Power, from the leftmost to the rightmost term
2. Multiplication or division, whichever comes first, from the leftmost to the rightmost term
3. Addition and subtraction, whichever comes first, from the leftmost to the rightmost term

For the right-hand side in the assignment statement above the order of operations will be as follows:

1. r^2
2. m/d
3. $k \cdot (r^2)$
4. $k \cdot (r^2) / 5.0$
5. $1.32 + (m/d)$
6. $1.32 + (m/d) - (k \cdot r^2 / 5.0)$
7. Assign result to C

Parentheses can be used to clarify operations if needed, for example, the expression above can be re-written as:

```
C = 1.32 + (m/d) - (k*r^2)/5.0
```

or,

```
C = 1.32 + (m/d) - ((k*r^2)/5.0),
```

or, even

```
C = 1.32 + (m/d) - (k*(r^2))/5.0
```

If parentheses are used, then operations in parentheses are performed first, from the innermost to the outermost parentheses, with operations within parentheses following the hierarchy of operations listed above.

Built-in functions in VBA

A built-in function in VBA is a function that is readily available to be used in expressions for calculations. A number of numerical built-in functions are available in VBA as detailed in Table 3, below. NOTE: *method* is also a name used to refer to *functions*.

Table 3. Mathematical functions (methods) available in VBA

.NET Framework method	Description
Abs	Returns the absolute value of a specified number.
Atan	Returns a Double value containing the angle whose tangent is the specified number.
Cos	Returns a Double value containing the cosine of the specified angle.
Exp	Returns a Double value containing e (the base of natural logarithms) raised to the specified power.
Log	Returns a Double value containing the logarithm of a specified number. This method is overloaded and can return either the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Round	Returns a Double value containing the number nearest the specified value. Additional round functions are available as methods of the intrinsic types such as Round .
Sign	Returns an Integer value indicating the sign of a number.
Sin	Returns a Double value specifying the sine of an angle.
Sqrt	Returns a Double value specifying the square root of a number.
Tan	Returns a Double value containing the tangent of an angle.

To return the value of a function, use the function name followed by parentheses with an argument value in between. Trigonometric functions (*Cos*, *Sin*, *Tan*) require that their arguments be given in radians. Function *Atan* returns the arctangent (\tan^{-1}) of a real number. It is known, for example, that $\tan(\pi/4) = 1.0$, thus, the value of π can be calculated in code as follows:

```
Dim Pi As Double
Pi = 4.0 * Atan(1.0)
```

The conversion factor between radians (θ^r) and degrees (θ°) is:

$$\theta^r = \frac{\pi}{180} \cdot \theta^\circ \quad ,$$

or, vice versa,

$$\theta^\circ = \frac{180}{\pi} \cdot \theta^r$$

Function *Log(x)* represents the *natural logarithmic* function, which in some math textbooks is written as *ln(x)*. *Log(x)* is the inverse of the *exponential* function, $\text{Exp}(x) = e^x$. To obtain the number *e*, base of the natural logarithms, you can use:

```
Dim myE As Double
myE = Exp(1.0)
```


Some other useful functions in math, are the *square root* function, $Sqr(x)$, the *sign* function, $Sign(x)$, and the *absolute value* function, $Abs(x)$.

An example using mathematical functions

In the VBA IDE shown below, I have a macro called *Sinusoidal* (), located within the module *MyMathFunctions*, within a macro-enabled EXCEL file called *MathFunctions.xlsm*.

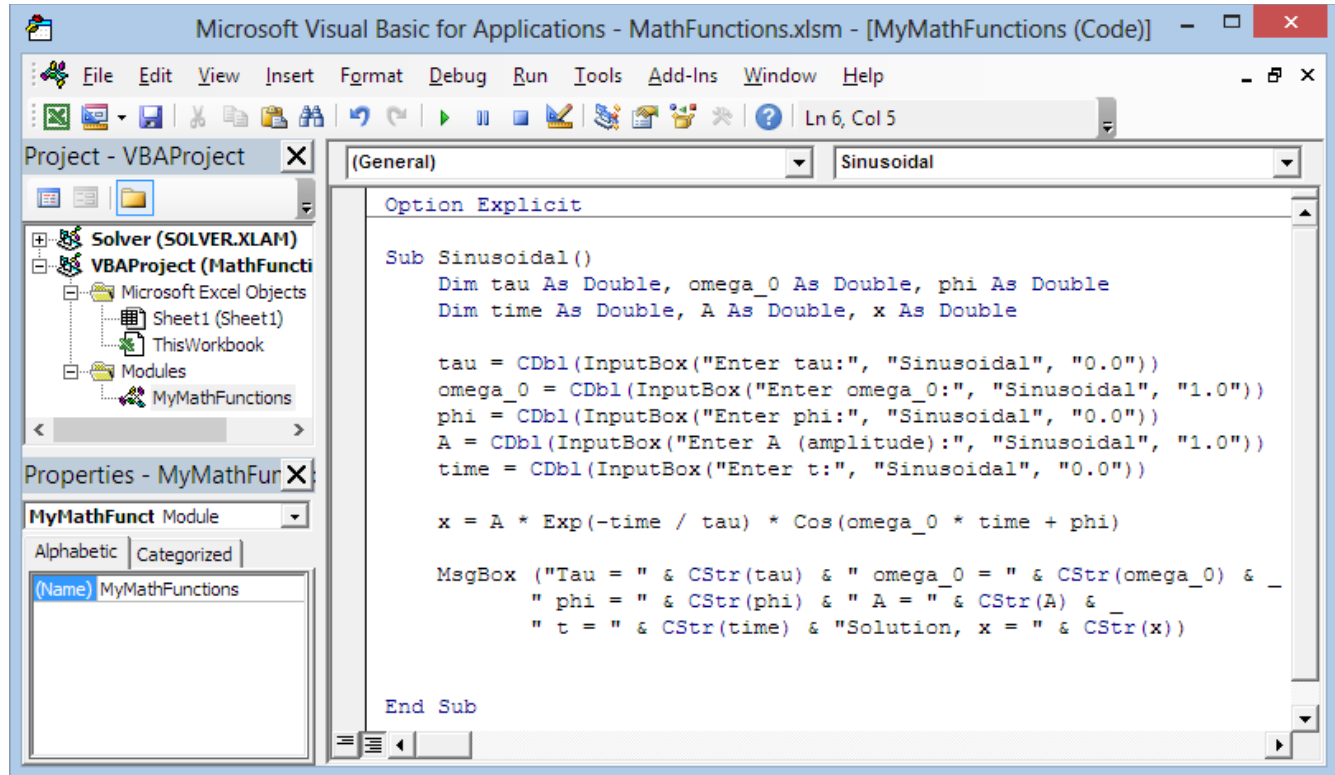


Figure 16. Example of VBA program using mathematical functions.

This program takes as input the following variables: τ (*tau*), ω_0 (*omega_0*), A (A = Amplitude), ϕ (*phi*), t (*time*), and calculates position x for a damped harmonic motion. The value of x is calculated using:

$$x = A \cdot e^{-\frac{t}{\tau}} \cdot \cos(\omega_0 \cdot t + \phi) ,$$

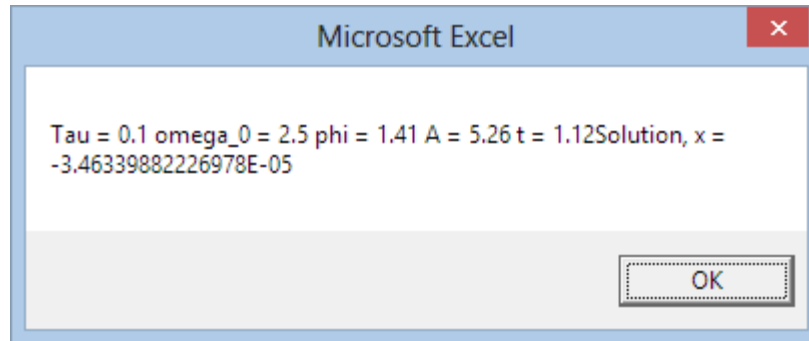
which translates into: $x = A * \text{Exp}(-\text{time} / \text{tau}) * \cos(\text{omega_0} * \text{time} + \text{phi})$, in VBA. Input takes place through a number of *InputBox* statements, preceded by the function *CDBl* (Convert to Double). Function *CDBl* is needed because the value we enter in *InputBox*, at the prompt, is in *String* format. Function *CDBl* converts the input string into a *Double*-precision data value. After calculating x , the output to this program is listed in a single *MsgBox* whose prompt (or output message) is formed by concatenating a number of string constants, e.g., "Tau =", etc., with numerical values converted to strings by using function *CStr* (Convert to String).

Using the continuation character

The *MsgBox* statement of Figure 16 also includes what is known as the *continuation character*, a simple underscore (_), placed at the end of a line, which indicates to VBA that the statement started in

a particular line of code does not end in that line, but continues in the next line. VBA then treats the multiple-line statement as a single-line long statement. The example shown in the *MsgBox* of Figure 16 shows a line of code divided into three lines, with two continuation characters used.

A sample output from the program of Figure 16 is shown right below:



This program can be run from the *VBA IDE* by pressing the *Run Macro* button (see button (10), in Figure 8, above).

Other data type conversion functions

The functions *Cdbl* and *CStr* are probably some of the most useful for input (using *InputBox*) and output (using *MsgBox*). Their use was illustrated by the example of Figure 16. VBA include a number of other data type conversion functions to convert between the different type of data available to the programmer. A complete listing of those data type conversion functions for *VBA 2012* is given in the following address: <http://msdn.microsoft.com/en-US/library/s2dy91zy%28v=vs.110%29.aspx>

Using a *EXCEL* worksheet as an interface for input-output

The example of Figure shows the use of *InputBox* for input, and *MsgBox* for output. While the *InputBox* is convenient for entering data into a program, an easier way to do it is by using a *EXCEL* spreadsheet at the input-output interface. Thus, for the program of Figure 16, instead of using *InputBoxes*, we could create the following interface in a spreadsheet:

	A	B	C	D	E
1	CALCULATION OF DAMPED HARMONIC MOTION POSITION				
2	Use Program "Sinusoidal" in the VBA IDE				
3					
4	INPUT DATA				
5	$\tau(s):$	0.1			
6	$\omega_0 (rad/s):$	2.5			
7	$\phi (rad):$	1.41			
8	A (cm):	5.26			
9	time, t(s):	1.12			
10	OUTPUT DATA:				
11	x(cm):	-3.4634E-05			
12					

Figure 17. Spreadsheet interface for input-output for an alternative version of the *Sinusoidal* program of Figure 25 (see also Figure 27).

The code in Figure 16 is modified to read as shown below:

```
Option Explicit

Sub Sinusoidal()
    Dim tau As Double, omega_0 As Double, phi As Double
    Dim time As Double, A As Double, x As Double

    tau = Range("B5").Value
    omega_0 = Range("B6").Value
    phi = Range("B7").Value
    A = Range("B8").Value
    time = Range("B9").Value

    x = A * Exp(-time / tau) * Cos(omega_0 * time + phi)

    Range("B11").Value = x
End Sub
```

Figure 18. Code of program *Sinusoidal*, modified from Figure 16 to use the spreadsheet of Figure 17 as input-output interface

After modifying program *Sinusoidal*, from Figure 16, to the version shown in Figure 18, the program can be run from the *VBA IDE* by pressing the *BASIC run* button (see button (10), in Figure 8, above). This time, the program gets the values of τ (*tau*), ω_0 (*omega_0*), A (A = Amplitude), ϕ (*phi*), and t (*time*), from the spreadsheet cells in Figure 26 by using the *Range* function. The argument to *Range* is the cell specification where the corresponding data value is entered, enclosed in double quotes. Thus, the value of *tau* is copied from cell *B5*, while that of A comes from cell *B8*, etc. The calls to function *Range* are followed by the property *.Value* to indicate that the value of that particular cell is what is passed onto VBA for processing. After entering the values of τ (*tau*), ω_0 (*omega_0*), A (A = Amplitude), ϕ (*phi*), and t (*time*), the value of x is calculated as shown in the corresponding assignment statement in Figure 18, and the resulting value is returned as the *.Value* property of cell *B11* through the last programming statement of Figure 18. Figure 17 shows the calculation for the data entered in the yellow-colored cell *B11*.

This example, therefore, illustrates the use of function *Range(" ").Value* both for input and output. Using the spreadsheet interface facilitates entering the data. I find this approach more user friendly than using the *InputBoxes* and the *MsgBox* used in the program version presented in Figure 16.

Notice from Figure 16 that there is only one spreadsheet in the workbook (*Sheet1*), therefore, calls to *Range("B7").Value* are unambiguous. Since only spreadsheet *Sheet1* is currently available, we know that all references in the *Range* function refer to that spreadsheet. If more than one spreadsheet were to be available, however, it is a good idea to refer to the cell(s) of interest by using its full reference, i.e., including the spreadsheet name in the form of a call to function *Worksheets("Sheet1")*, followed by the *Range* range reference, followed by *.Value*. See Figure 19, for details of the modified code.

```

Option Explicit

Sub Sinusoidal()
    Dim tau As Double, omega_0 As Double, phi As Double
    Dim time As Double, A As Double, x As Double

    tau = Worksheets("Sheet1").Range("B5").Value
    omega_0 = Worksheets("Sheet1").Range("B6").Value
    phi = Worksheets("Sheet1").Range("B7").Value
    A = Worksheets("Sheet1").Range("B8").Value
    time = Worksheets("Sheet1").Range("B9").Value

    x = A * Exp(-time / tau) * Cos(omega_0 * time + phi)

    Worksheets("Sheet1").Range("B11").Value = x
End Sub

```

Figure 19. Modifying the code of Figure 27 to include an explicit reference to the *Sheet1* worksheet in input from and output to *Sheet1*'s cells.

Adding a command button to the spreadsheet interface

In this section, we'll show you how to add a command button to the spreadsheet interface so that you can run the program by pressing that command button, instead of having to run the program from the *VBA IDE*.

In the *DEVELOPER* tab, (1) click to select the *Design Mode* option. Then, (2) press the *Insert* button to see the following tools:

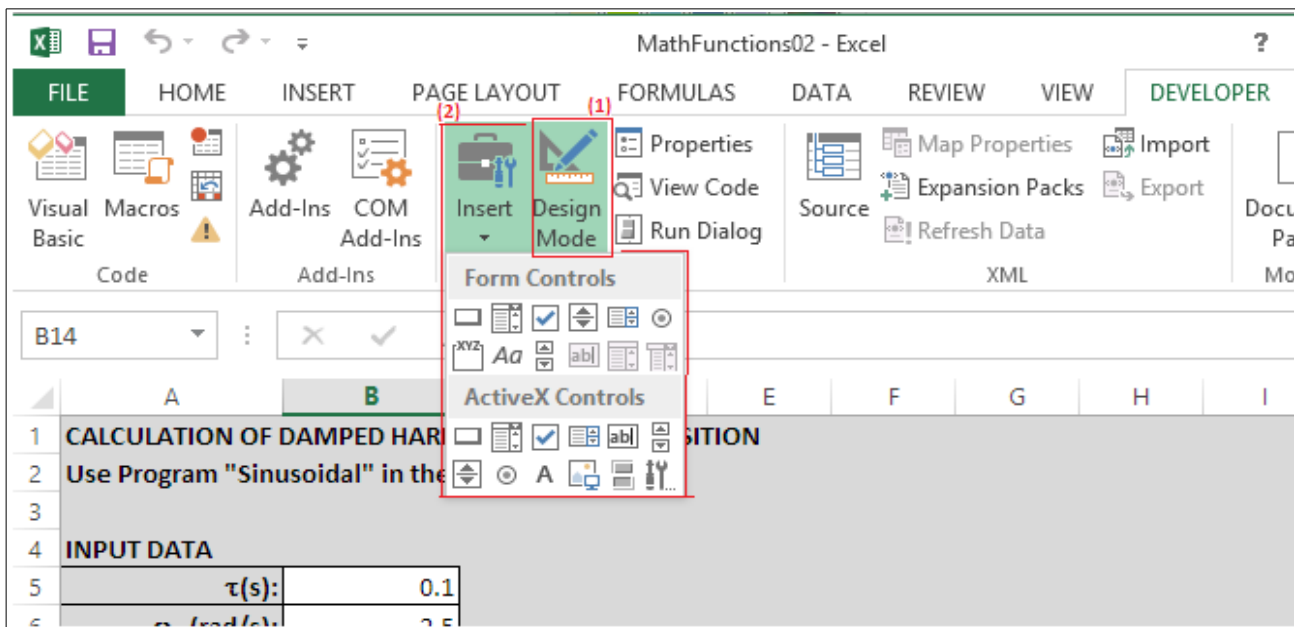


Figure 20. The *Design Mode* and the *Insert* (insert commands) button in the *DEVELOPER* tab in *EXCEL*.

The *Insert (insert command)* button produces a toolbox consisting of two collections of commands, namely, *Form Controls* and *ActiveX Controls*. The following figure shows those collection of commands, as well as a diagram referencing the buttons with the numbers 1 through 12 for each collection. The diagram is used later to identify the various components of the command collections from the *Insert* button.

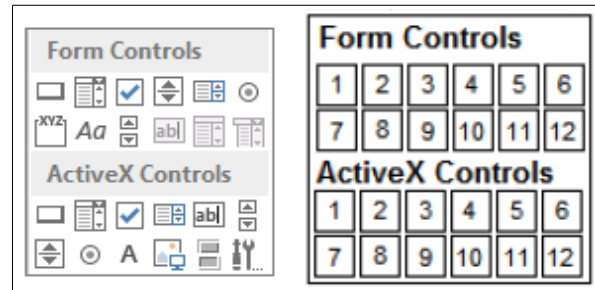


Figure 21. *Form Controls* and *ActiveX Controls* available through the *Insert* button in the *DEVELOPER* tab in *EXCEL*.

The various controls available under the *Form Controls* collection are (see Figure 21):

(1) Button; (2) Combo Box; (3) Check Box; (4) Spin Button; (5) List Box; (6) Option Button; (7) Group Box; (8) Label; (9) Scroll Bar; (10) Text Field; (11) Combo List – Edit; and, (12) Combo Drop-Down.

On the other hand, the controls available under the *ActiveX Controls* collection are (see Figure 21):

(1) Command Button; (2) Combo Box; (3) Check Box; (4) List Box; (5) Text Box; (6) Scroll Bar; (7) Spin Button; (8) Option Button; (9) Label; (10) Image; (11) Toggle Button; (12) More Controls.

In the present example, a command button will be added. The question to answer is: should it be a *Form Control* button or an *ActiveX Control* button? In general, either one of the will perform the task required in this example, i.e., to activate a program from an *EXCEL* worksheet. However, they have different ways of formatting and activating events. For this example, we will add a *Form Control* command button as indicated below.

Procedure for adding a Form Control command button to an EXCEL worksheet:

- Click on the *Form Control* button
- Click at the location in the *EXCEL* worksheet where the upper left corner of the button will be located (see Figure 23 for this example)
- Drag the cursor to the position where the lower right corner of the button will be located and click to finish drawing the button. This will open up the following *Assign Macro* dialog form:

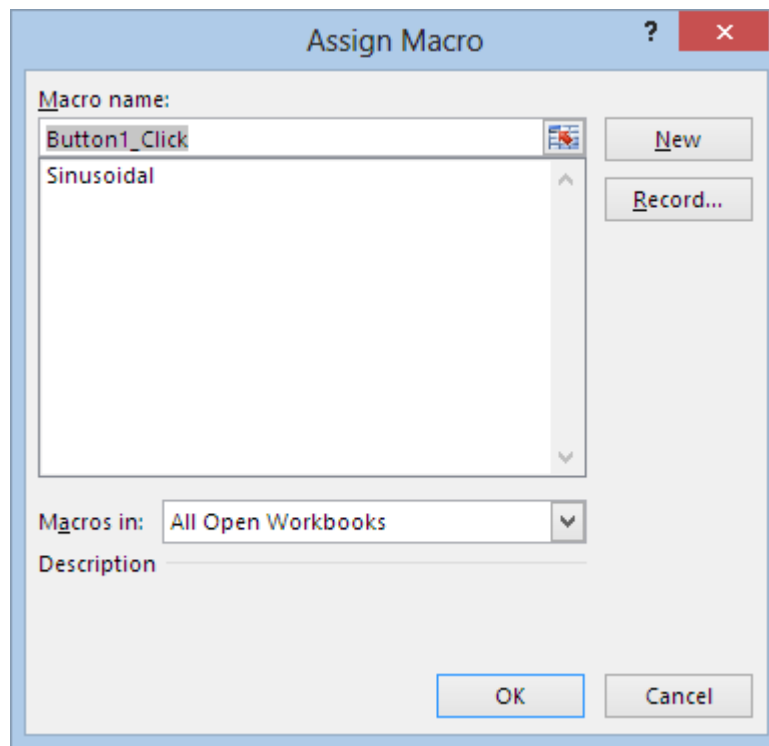


Figure 22. The *Assign Macro* dialog form for assigning a macro to a command button

Figure 22 shows the default name of *Button1_Click* in the field *Macro name*. This indicates that the button has been given the default name *Button1*, and that, consequently, the name assigned to the macro associated with this button is *Button1_Click*. This particular macro name implies that the macro will be activated by a click to *Button1*. However, instead of the default macro (*Button1_Click*), we actually want to activate the macro entitled *Sinusoidal*, which is conveniently listed in the *Assign Macro* form of Figure 22. Thus,

- Click, once, on the name *Sinusoidal*, in the *Assign Macro* form of Figure 22. The default macro name *Button1_Click* is now replaced by *Sinusoidal*.
- Press [OK].

The button is now associated with the *Sinusoidal* macro, whose last version is shown in Figure 19. The worksheet interface should now show the same labels as those in Figure 17, plus the button labeled “Button 1” (see Figure 23). At this point, while still in *Design Mode*, it is convenient to change the label in the button to, say, “Calculate position x”. This can be accomplished as follows:

- Click to the right of the button label, i.e., to the right of the “Button 1” text
- Delete the current label (delete “Button 1”)
- Type “Calculate position x”

To activate the button proceed as follows:

- Click on a worksheet cell so that the focus moves away from the button

- Click off the *Design Mode* button
- Clear cell *B11* by clicking on it, and pressing the *Backspace* key in your keyboard, then click somewhere else in the worksheet
- Press the “Calculate position x” button

The result is shown in Figure 23, below.

	A	B	C	D	E
1	CALCULATION OF DAMPED HARMONIC MOTION POSITION				
2	Use Program "Sinusoidal" in the VBA IDE				
3					
4	INPUT DATA				
5	τ (s):	0.1	Calculate position x		
6	ω_0 (rad/s):	2.5			
7	ϕ (rad):	1.41			
8	A (cm):	5.26			
9	time, t(s):	1.12			
10	OUTPUT DATA:				
11	x(cm):	-3.4634E-05			
12					

Figure 23. Adding a *Form Command Button* to the spreadsheet interface.

Recording a macro to clear the output cell

Suppose we want to have a button that would clear cell *B11*, the output field. We know how to do that by hand: click on *B11*, press [Backspace] in our keyboard, click in other cell, say *E12*. And that's it. *EXCEL* will let you record those steps in a macro by using the *Record Macro* button, available within the *Code* frame in the *DEVELOPER* tab's *Ribbon* (see below):



Figure 24. The *Record Macro* button in the *Code* frame in the *DEVELOPER* tab in *EXCEL*.

Thus, to record the macro to clear cell *B11*, first click the *Record Macro* button in the *DEVELOPER* tab. This opens up the following *Record Macro* dialog form, which we modify to read as shown in the following Figure.

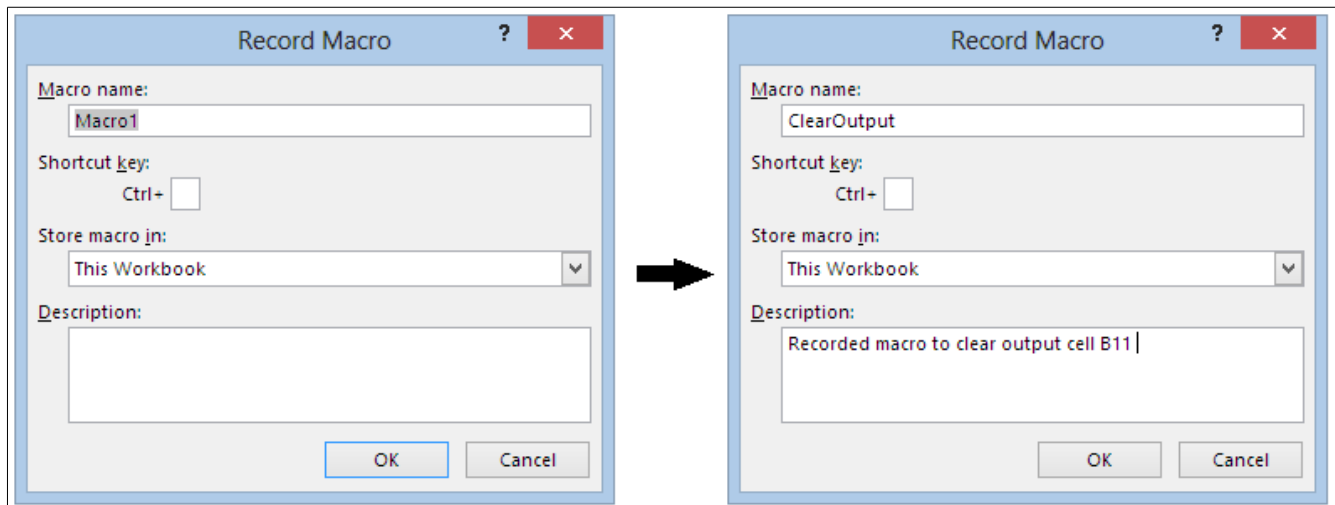


Figure 25. Naming a recorded macro using the *Record Macro* dialog form.

When done giving the macro a name and selecting where to store it (default: *This Workbook*), press the [OK] button in the *Record Macro* form. At this point *EXCEL* will be recording any change in the worksheet as part of the macro *ClearOutput*. Notice that the *Record Macro* button in the *DEVELOPER* tab changes to a blue rectangle (reminiscent of a *stop* button in a recorder). As long as the blue rectangle remains present, the macro is being recorded. Proceed as follows:

- Click on *B11*
- Press [Backspace] in your keyboard
- Click in another cell, say *E12*.
- Then, press the *Stop Recording* button.

Next, open the *VBA IDE*. In the *Object Explorer* you will see a new module, whose (default) name is *Module1*. To see the code corresponding to the macro recently recorded click on the *Module1* object in the *Object Explorer*. The code for macro *ClearOutput* will now be shown as follows:

```
Sub ClearOutput ()
'
' ClearOutput Macro
' Recorded macro to clear ouptut cell B11
'
'
    Range("B11").Select
    ActiveCell.FormulaR1C1 = ""
    Range("E12").Select
End Sub
```

Figure 25. Code for the *ClearOutput* macro recorded using the *Record Macro* button in the *DEVELOPER* tab.

The next step is to create a command button to be called *Clear Output*, and assign to it the macro *ClearOutput*. To do this, follow a procedure similar to that described in pages 29 through 31, above. The resulting interface is shown below (Figure 26).

	A	B	C	D	E
1	CALCULATION OF DAMPED HARMONIC MOTION POSITION				
2	Use Program "Sinusoidal" in the VBA IDE				
3					
4	INPUT DATA				
5	τ (s):	0.1	Calculate position x		
6	ω_0 (rad/s):	2.5			
7	ϕ (rad):	1.41			
8	A (cm):	5.26			
9	time, t(s):	1.12			
10	OUTPUT DATA:				
11	x(cm):		Clear Output		
12					
13					

Figure 26. Spreadsheet interface for calculating damped harmonic motion position including a *Clear Output* button associated with the recorded macro of Figure 25.

Press the [Clear Output] button to clear cell B11, then, change the data in cells B5 through B9 as needed, and press [Calculate position x] to recalculate the value of x. Try it.

User-defined functions

The code of Figures 11, 15, 16, 18, and 19 represent one type of programs available in VBA, the *Sub* (or subroutine) programs. A *Sub* program typically can be run on its own, and may have its own input and output statements. A *Function* program, on the other hand, typically has one or more parameters, performs calculations or other processes on the data passed on to it, and returns a single value to the program from which it was called. While subroutines can also be called from other programs, for the time being we'll describe only how to use *Function* programs that can be called from the *EXCEL* spreadsheet as *user-defined functions*.

For example, suppose you want to program the following expression as a user-defined function:

$$f(a, b, c, x) = \sin(a + c \cdot x) \cdot \sqrt{a + b \cdot x} + \cos(a + c \cdot x) \cdot \ln(a + b \cdot x) \quad .$$

Since the quantities $a + c \cdot x$ and $a + b \cdot x$ are repeated twice, they can be calculated into two variables, say $t_1 = a + c \cdot x$, and $t_2 = a + b \cdot x$, so that we can write the function in pseudo-code as:

```
Function f(a, b, c, x)
    t1 = a + c * x
    t2 = a + b * x
    f = sin(t1) * sqrt(t2) + cos(t1) * ln(t2)
End Function
```

Figure 27 shows the coding of this function in the *VBA IDE*. The function name was changed from *f* in the pseudo-code to *myFunction* in the code.

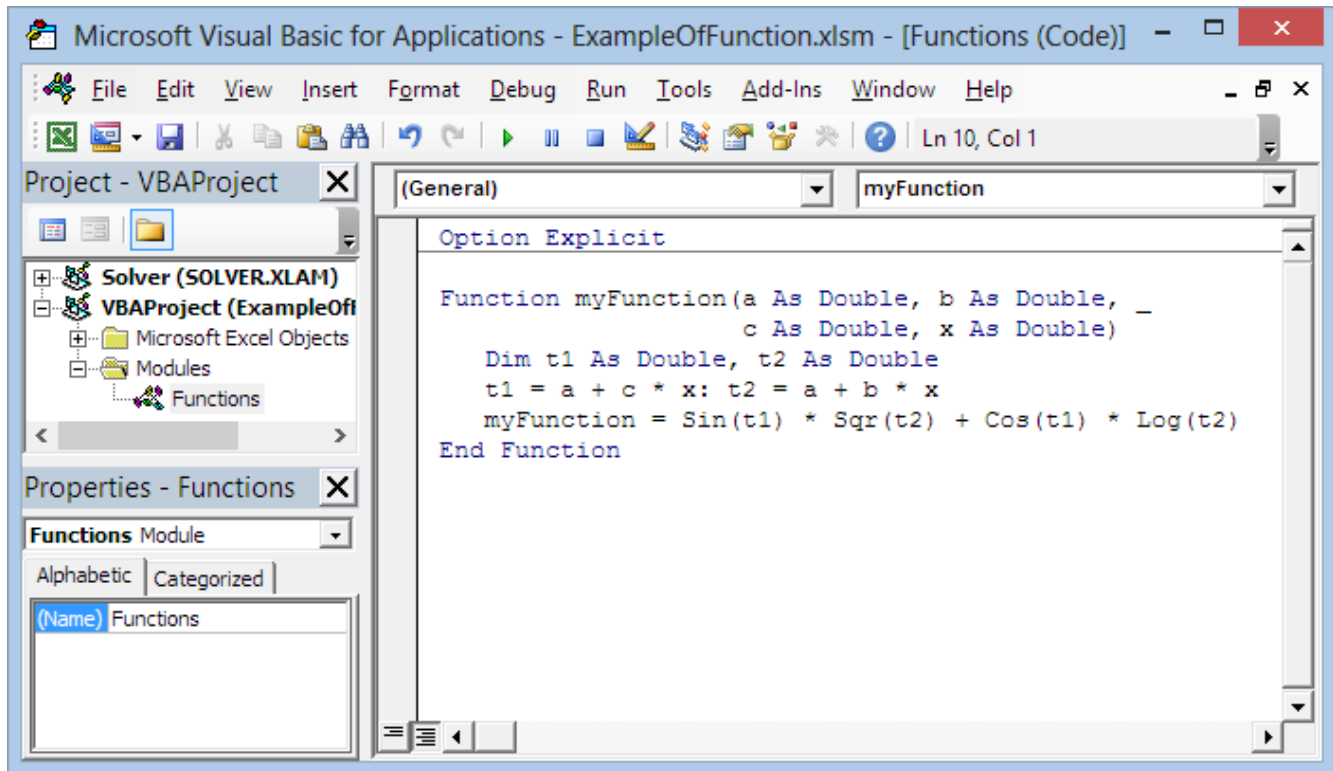


Figure 27. Code for function *myFunction*.

The code of function *myFunction* in Figure 27 includes some programming features worth mentioning:

- (1) There is a continuation line (`_`) in the *Function* statement.
- (2) The function *parameters* are the variables *a*, *b*, *c*, and *x*. Notice that they are declared in the *Function* statement using, *a As Double*, *b As Double*, etc., instead of using a *Dim* statement in the body of the function.
- (3) Since a *Function* subprogram returns a value, there is a data type associated with it. Thus, at the very end of the *Function* statement, we added *As Double* to declare function *myFunction* as a *Double*-precision value. (Note: *Sub* programs do not return a single value, like *Functions* do, therefore, there is no data type associated with their names).
- (4) Variables *t1* and *t2* are *local variables* to the *Function* subprogram, i.e., they're defined within the body of the *Function* and used there only. Local variables are used to hold intermediate values in the calculation of the function name.
- (5) The line `t1 = a + c * x: t2 = a + b * x` in this *Function*'s code illustrates the fact that you can place two or more VBA statements in a single line if you separate them with a colon (:).
- (6) Somewhere in the body function (typically, in the last statement) the function name (in this case, *myFunction*) gets assigned the value that the function will return to the point where it is called. Thus, the value of function *myFunction*, in this example, gets assigned in the last line of code in the function's body, namely, `myFunction = Sin(t1) * Sqr(t2) + Cos(t1) * Log(t2)`.

The function *myFunction* is now available to be called from the *EXCEL* spreadsheet. For example, click on cell *B1*, type "`= myFunction(2.3,5.2,1.8,0.5)`", and press [Enter]. The result, shown in cell *B1*, is -1.71574.

As a second example of application, consider the creating of a table as shown in Figure 28. The values of *a*, *b*, and *c* are entered in cells *C4*, *C5*, and *C6*, respectively. A table of *x* and *f(x)* is created in the range *B8:C28*. Start by entering a 0.0 in cell *B8*, then, enter the formula: `= 0.5+B8` in cell *B9*. Click on *B9* and drag the formula down all the way to cell *B28*. In cell *C8* enter the formula: `=myFunction(C4,C5,C6,B8)`, and then drag it out down to cell *C28* to complete the table.

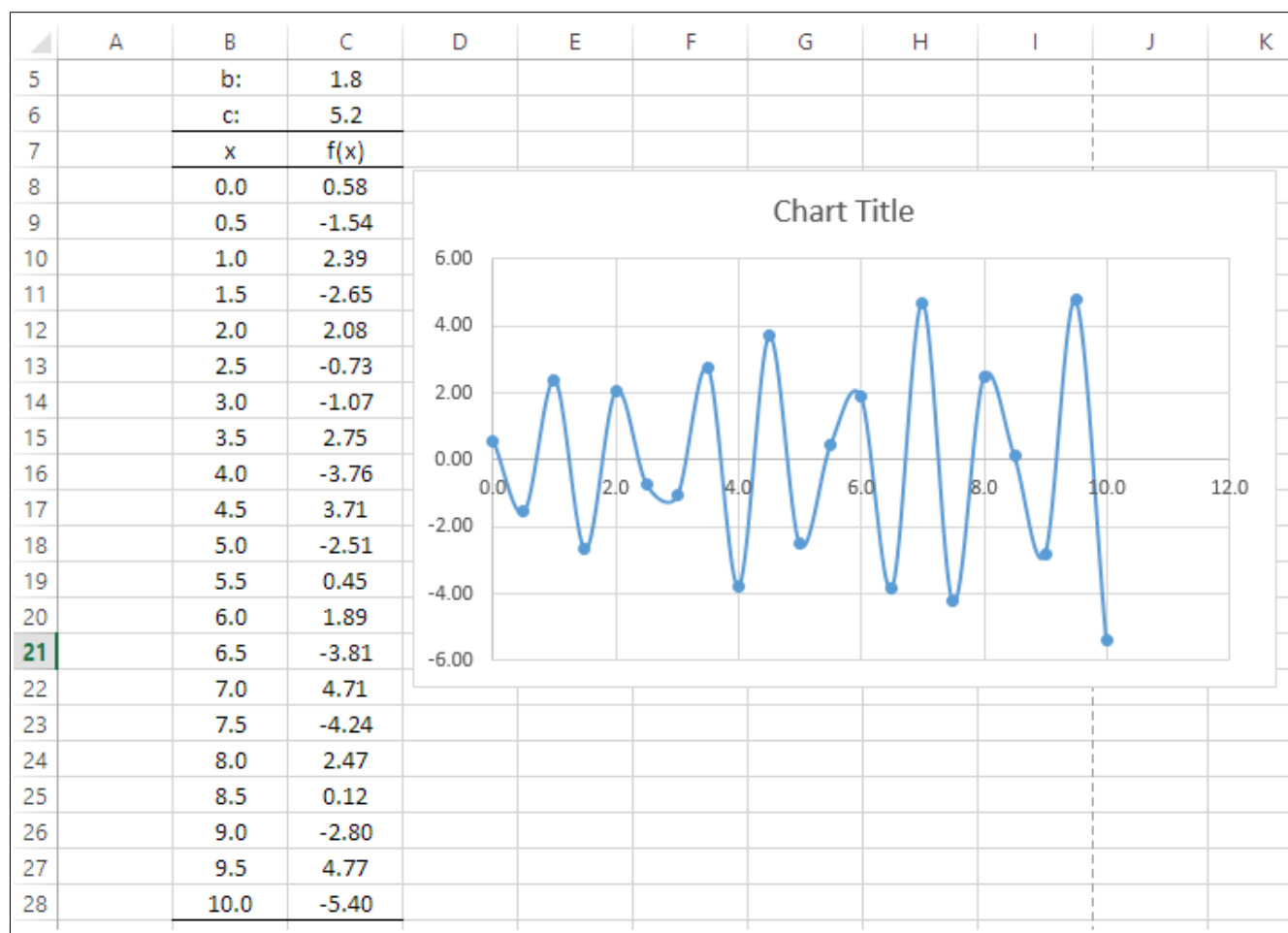


Figure 28. Evaluation of the function *myFunction* using the parameters *a*, *b*, and *c*, as shown, while creating a table of *x* and *f(x)*.

To create the plot, select the cells in the range *B8:C28*, then click on the *INSERT* tab. In the *INSERT* tab, select the *Insert Scatter (X,Y) or Bubble Chart* button as illustrated in Figure 29, below.

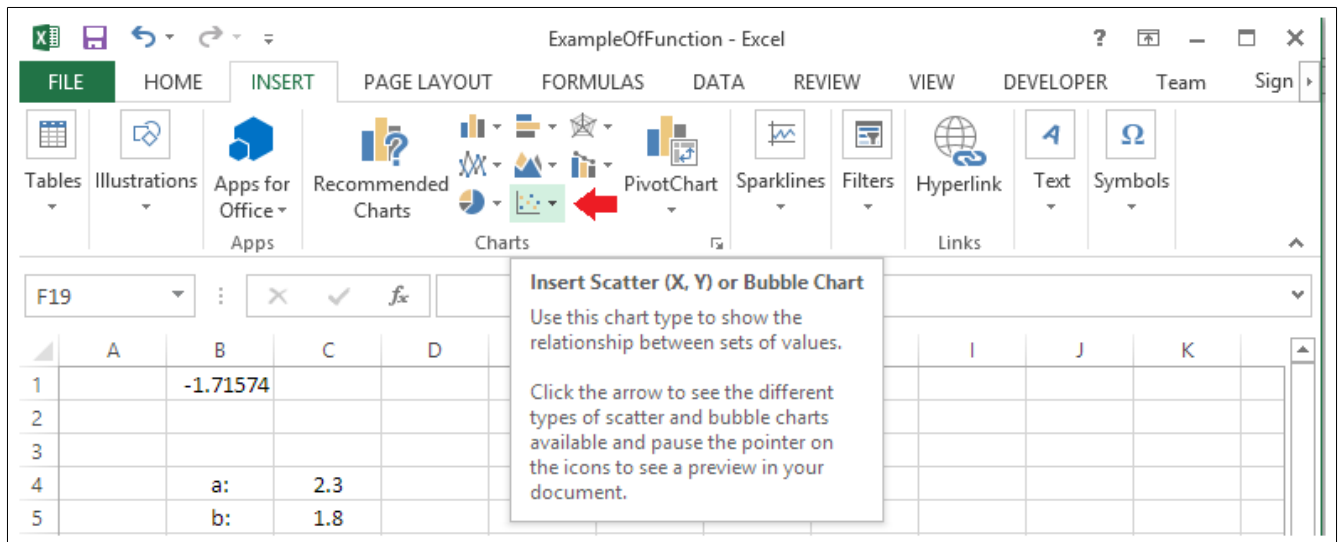


Figure 29. Select the *Insert Scatter (X,Y) or Bubble Chart* button in the *INSERT* tab to insert a (x,y) scatter plot.

The button of Figure 29 provides several options, as illustrated in Figure 30. Select the option *Scatter with Smooth Lines and Markers* in the *Scatter* chart group as shown in Figure 30. The graph of Figure 28 will now be shown.

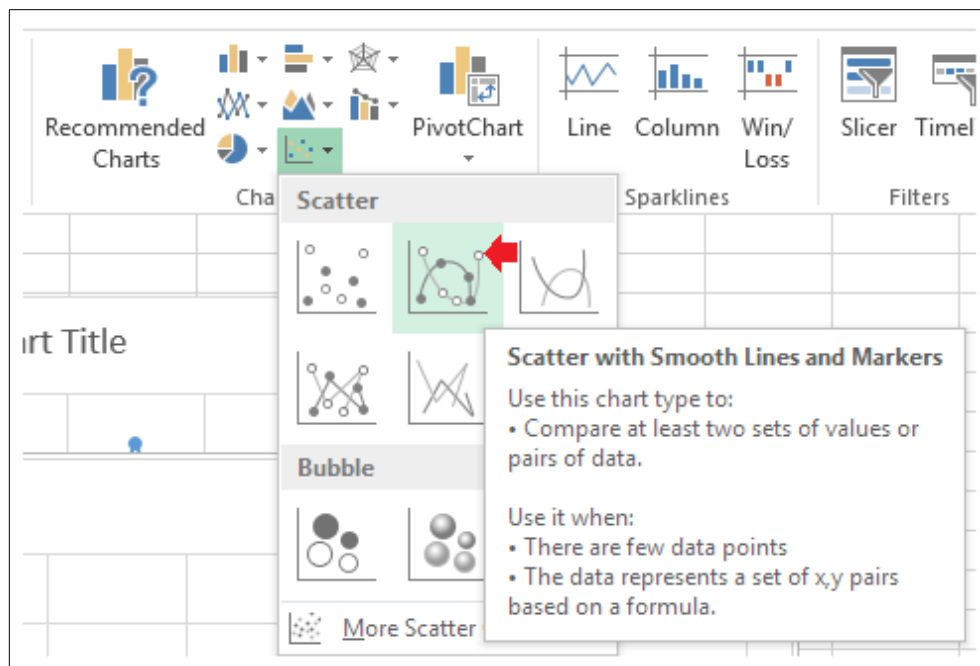


Figure 30. Select the *Scatter with Smooth Lines and Markers* option within the *Scatter* chart group in the *INSERT* tab.

Using worksheet functions – The *Insert Function* button

EXCEL includes a number of functions that you can use directly in evaluating cells in the spreadsheet. To see the listing of functions available, try the following:

1. Click on a spreadsheet cell (e.g., cell *E1*), and type “=”
2. Click the *Insert Function* button in the entry bar of the EXCEL spreadsheet (see the *fx* button in Figure 31).

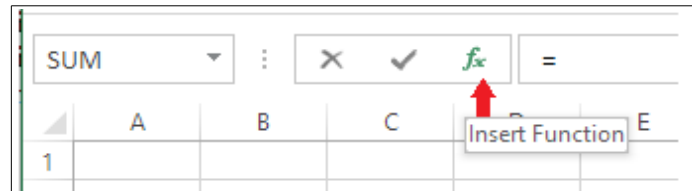


Figure 31. Location of the *Insert Function* button in the EXCEL spreadsheet.

3. This will open up a *Insert Function* window as shown in Figure 32.

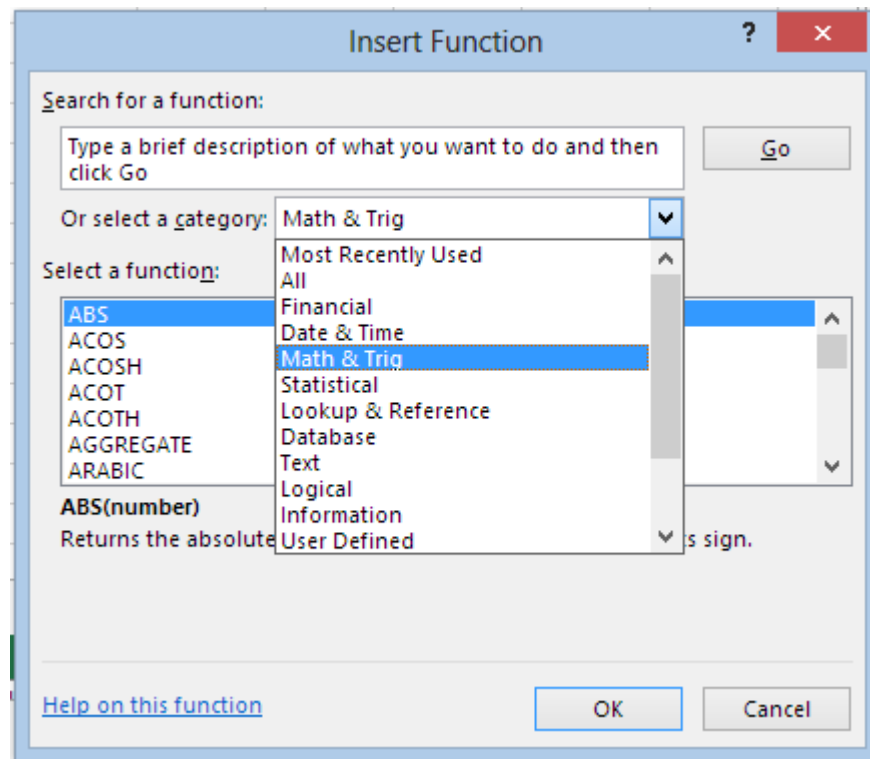


Figure 32. The *Insert Function* window in EXCEL.

The *Insert Function* window of Figure 32 includes a field where you can search for the name of a specific function, or you can select a function category from the drop-down menu shown. Once a category of functions is selected (e.g., the *Math & Trig* category shown in Figure 32), the functions contained within that category will be listed in the *Select a function:* field. Clicking on a function name will then copy the function name in the cell that you had previously selected.

For example, select the function *ABS* (*ABS*olute value) within the *Math & Trig* category. This opens up the following form:

Figure 33. The *Function Arguments* window corresponding to the *ABS* function from the *Math & Trig* category of functions in the *Insert Function* button.

The *Function Arguments* form of Figure 33 specifies the nature of the selected function's argument(s). For example, for function *ABS*, as shown in Figure 33, the argument is a *Number*. In the field shown in front of the *Number* label in Figure 33 you can enter a number and have the form provide the result of applying the selected function. In this example, I entered the number -23 as argument to *ABS* and found that the *Formula result* is 23. This means $ABS(-23) = 23$.

If you now press [OK], the application of function *ABS* to the argument -23, as done in Figure 33, will show up in the cell where the function assignment was initially set. In this example, as shown in step 1 in page 37, we had selected cell *E1*, to enter the function *ABS*. For this case, then the result of the calculation $=ABS(-23)$ is shown below.

<div> ✕ ✓ <i>f_x</i> </div>			=ABS(-23)
	C	D	E
			23

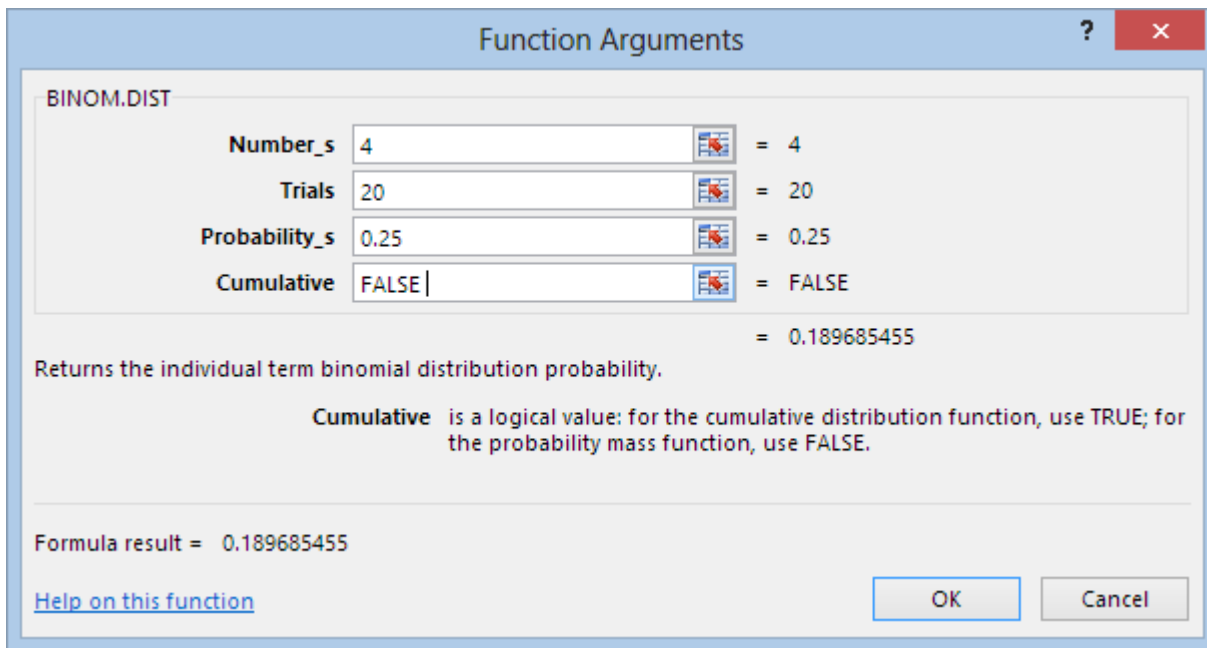
Figure 34. Use of the pre-defined function *ABS* in cell *E1*.

Entering a worksheet function directly

If you know the name of the function to enter, you can just type it in a cell with the proper argument(s), after an equal sign, then press [ENTER], without having to use the *Insert Function* button. The function evaluation will be shown in the selected cell. For example, type “=ATAN(5.5)” in cell *C5*, and press [ENTER]. Cell *C5* then will show the result: 1.390943.

Worksheet functions with more than one argument

To enter a worksheet function with more than one argument, you can use the *Insert Function* button, or type it directly into a cell. Make sure to separate function arguments with commas (.). Figure 35 shows the *Insert Function* corresponding to function *BINOM.DIST* (the binomial distribution) within the *Statistical* category of functions. When you click in each argument's field, a brief description of the argument's nature is presented. In Figure 35, the last argument (*Cumulative*) is currently selected. The value *FALSE* was entered in this field to calculate the *probability mass function* of the binomial distribution.



The screenshot shows the 'Function Arguments' dialog box for the *BINOM.DIST* function. The arguments are: *Number_s* (4), *Trials* (20), *Probability_s* (0.25), and *Cumulative* (FALSE). The result is 0.189685455. A description for the *Cumulative* argument is provided: 'Cumulative is a logical value: for the cumulative distribution function, use TRUE; for the probability mass function, use FALSE.' The formula result is displayed as '= 0.189685455'.

Figure 35. *Function Wizard* for the *BINOM.DIST* function

The function call corresponding to the case of Figure 44, once placed on a cell, will simply look as follows:

= BINOM.DIST (4, 20, 0.25, FALSE)

Using worksheet functions in VBA code

As indicated in Table 3, above, VBA provides a limited number of mathematical functions, namely, *Abs*, *Atan*, *Cos*, *Exp*, *Log*, *Round*, *Sign*, *Sin*, *Sqrt*, and *Tan* for use in coding. For some VBA programs in *EXCEL*, however, you may need the use of other worksheet functions as those listed in Figure 32, above. In order to be able to access worksheet functions in VBA code you need to refer to the function name preceding it with the word *WorksheetFunction*. Thus, to use function *BINOM.DIST* (see Figure 44) in VBA code you could use, for example:

```
px = WorksheetFunction.BINOM.DIST(x, n, p, FALSE)
```

To illustrate the use of worksheet functions in VBA code, let's modify the code in the macro *Sinusoidal*, of Figure 19, to the one shown in Figure 36.

```

Option Explicit

Sub Sinusoidal()
    Dim tau As Double, omega_0 As Double, phi As Double
    Dim time As Double, A As Double, x As Double

    tau = Worksheets("Sheet1").Range("B5").Value
    omega_0 = Worksheets("Sheet1").Range("B6").Value
    phi = Worksheets("Sheet1").Range("B7").Value
    A = Worksheets("Sheet1").Range("B8").Value
    time = Worksheets("Sheet1").Range("B9").Value

    x = A * WorksheetFunction.BesselI(time, 2) * Cos(omega_0 * time + phi)

    Worksheets("Sheet1").Range("B11").Value = x
End Sub

```

Figure 36. Modifying macro *Sinusoidal* from Figure 19 to include the worksheet function *BesselI*(x, n).

Compare the result of this version of the *Sinusoidal* macro (Figure 36) with that of Figure 26.

	A	B	C	D	E
1	CALCULATION OF DAMPED HARMONIC MOTION POSITION				
2	Use Program "Sinusoidal" in the VBA IDE				
3					
4	INPUT DATA				
5	τ (s):	0.1	Calculate position x		
6	ω_0 (rad/s):	2.5			
7	ϕ (rad):	1.41			
8	A (cm):	5.26			
9	time, t(s):	0.5			
10	OUTPUT DATA:				
11	x(cm):	-0.148737495	Clear Output		
12					
13					

Figure 46. Calculating position x for harmonic function using the macro of Figure 36.