VBA Programming in Excel: Subprograms

By Gilberto E. Urroz, October 2013

In this chapter we introduce the use of subprograms, an approach that allows us to divide large programs into smaller units. These smaller units, in the form of *Sub* subprograms or *Function* subprograms, can be tested separately, and related to each other to accomplish the program's task. Typically, a *Sub* program, known as the *main program*, acts as the main controlling unit which *calls* other subprograms. Information between subprograms is transferred through the use of subprogram arguments. Dividing a program into smaller units represents an approach known as *modular programming*.

Before describing subprograms, however, we will discuss a couple of issues related to programming style.

Recommended Programming Style 1 – Indentation

As you develop more complex programs in VBA (or any other programming language), it is recommended to write your code with proper indentation of the different programming statements used. Here are some guidelines:

- Let's call the left margin of a code window *indentation level No. 1*. Using the [Tab] key in your keyboard will place your code at subsequent indentation levels (*indentation levels Nos. 2, 3,* etc.).
- Both *Sub*s and *Function*s subprograms should have their *Sub* and *End Sub* statements, or *Function* and *End Function* statements, aligned with *indentation level No.1* and all other statements included in between (except *labels*), should be aligned with *indentation level No. 2*, at least.
- Labels should be placed at *indentation level No. 1*. Thus, only *Sub, EndSub,* or *Function, End Function*, and labels should be placed at *indentation level No. 1*. Labels end with a colon (:).
- Comments can be placed at any level of indentation, since they are not executable statements.
  - A line starting with an apostrophe (`'`) or with the particle *REM* (from *REM*ark) is a whole comment line. No part of it will be interpreted as an executable VBA statement at all.
  - A line that starts with a wholly-contained executable statement can be followed by a comment by placing an apostrophe (`'`) or with the particle *REM* (from *REM*ark) after the executable statement, and before the comment is typed.
- Use the following indentation format for *If ... End If*, *If ... Else ... End If*, *If ... ElseIf ... Else ... End If* statements:

```
If <condition 1> Then
   <statements 1>
ElseIf <condition 2> Then
      <statements 2>
...
Else
   <statements Else>
End If
```

- Use the following indentation format for *nested If statements*

```
If <condition 1> Then
   <statements 1>
   If <condition 2> Then
      <statements 2>
   End If
Else
   <statements Else>
End If
```

- Use the following indentation format for *Select Case … End Select* statements:

```
Select Case <variable>
   Case <case 1>
      <statements 1>
   Case <case 2>
      <statements 2>
   ...
   Case Else
      <statements Else>
End Select
```

- Use the following indentation format for *While … Wend* statements:

```
While <condition>
   <loop statements>
Wend
```

- Use the following indentation format for *Do While … Loop* or *Do Until … Loop* statements:

```
Do While [or, Do Until] <condition>
   <loop statements>
Loop
```

- Use the following indentation format for *Do … Loop While* or *Do … Loop Until* statements:

```
Do
   <loop statements>
Loop While [or, Loop Until] <condition>
```

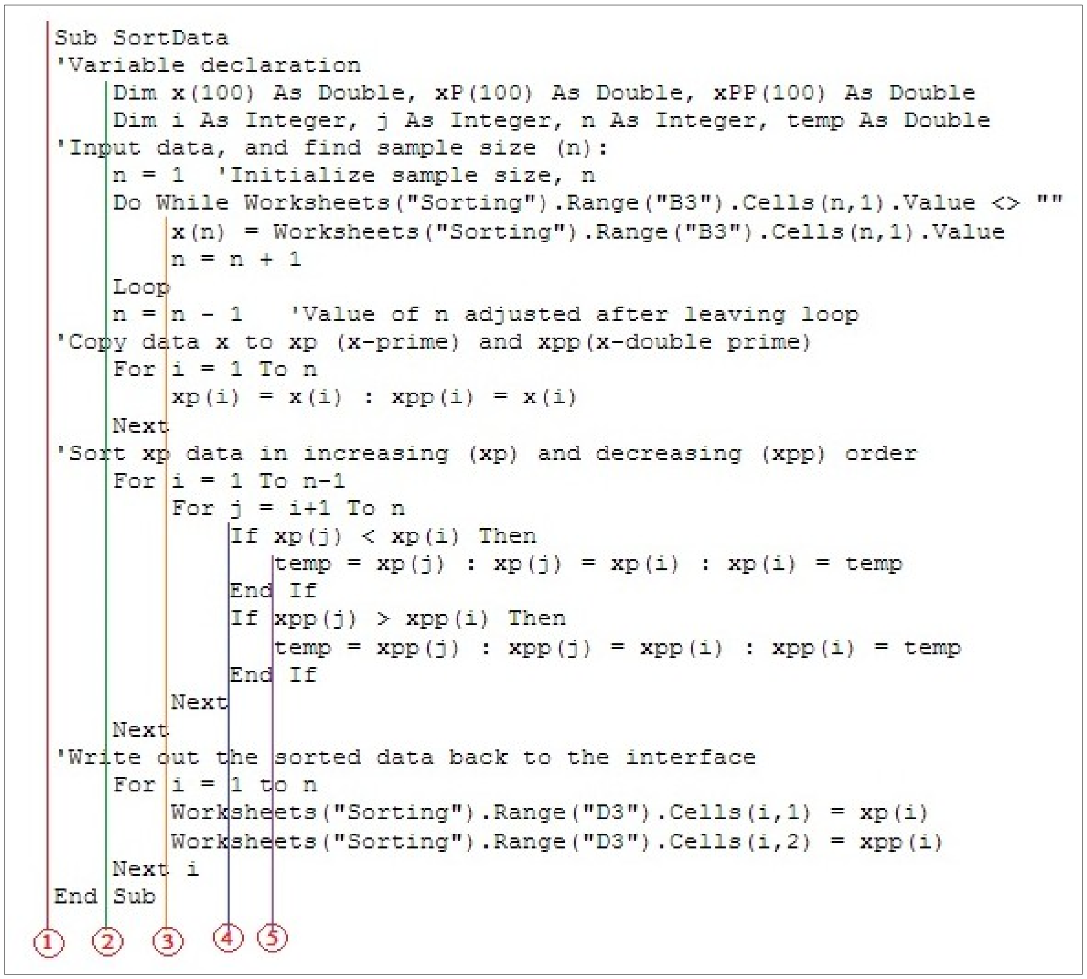
- Use the following indentation format for *For … Next* statements:

```
For <index> = <start> To <end> [Step <increment>]
   <loop statements>
Next [<index>]
```

• Use the following indentation format for *nested For … Next* statements:

```
For <index1> = <start1> To <end1> [Step <increment1>]
   <loop statements 1 before inner loop>
   For <index2> = <start2> To <end2> [Step <increment2>]
      <loop statements2>
   Next [<index2>]
   <loop statements 1 after inner loop>
Next [<index1>]
```

Indentation is recommended so that other programmers or users can understand the operation of the VBA statements you include in your programs. Use of proper indentation, as suggested in the guidelines above, is consider good programming practice. Avoiding indentation is consider lazy programming, and will soon get you the cold shoulder from your fellow programmers.



```
Sub SortData
'Variable declaration
    Dim x(100) As Double, xP(100) As Double, xPP(100) As Double
    Dim i As Integer, j As Integer, n As Integer, temp As Double
'Input data, and find sample size (n):
    n = 1  'Initialize sample size, n
    Do While Worksheets("Sorting").Range("B3").Cells(n,1).Value <> ""
        x(n) = Worksheets("Sorting").Range("B3").Cells(n,1).Value
        n = n + 1
    Loop
    n = n - 1   'Value of n adjusted after leaving loop
'Copy data x to xp (x-prime) and xpp(x-double prime)
    For i = 1 To n
        xp(i) = x(i) : xpp(i) = x(i)
    Next
'Sort xp data in increasing (xp) and decreasing (xpp) order
    For i = 1 To n-1
        For j = i+1 To n
            If xp(j) < xp(i) Then
               temp = xp(j) : xp(j) = xp(i) : xp(i) = temp
            End If
            If xpp(j) > xpp(i) Then
               temp = xpp(j) : xpp(j) = xpp(i) : xpp(i) = temp
            End If
        Next
    Next
'Write out the sorted data back to the interface
    For i = 1 to n
        Worksheets("Sorting").Range("D3").Cells(i,1) = xp(i)
        Worksheets("Sorting").Range("D3").Cells(i,2) = xpp(i)
    Next i
End Sub
```

Figure 1. Example of five levels of indentation on a VBA program.

Figure 1, above, illustrates the use of indentation for a VBA program. Notice that five levels of indentation are used overall. This example illustrates the typical indentation formats for a *Do While ... Loop* statement, a couple of *For ... Next* statements, and a couple of *nested For ... Next* statements, with two *If ... End If* included. Several comment lines are also shown, which, together with the *Sub*, and *End Sub* statements, are started at *indentation level No. 1*. The example includes comments that follow executable statements, e.g., `n = n - 1    'Value of n adjusted after leaving loop`

Recommended Programming Style 2 – Documenting your program in detail

The code shown in Figure 1 uses a few comment lines to document the operation of various parts of the program. However, when developing programs that will be read and interpreted by other users, it is recommended to document your program in more detail, including the program's and programmer's names, date, and version of the program at the top of the code. You may also include a brief description of the program operation and of the variables involved at the top of the code. The following code listing illustrates a detailed program description for the program of Figure 1 after extensive documentation has been added.

```
Sub SortData()
'--------------------------------------------------------------------------|
' Program name:  SortData                                                  |
' Programmer:    Gilberto E. Urroz                                         |
' Date:          September 28, 2013                                        |
' Version:       1.0                                                       |
'--------------------------------------------------------------------------|
' Program description:                                                     |
' ====================                                                     |
' Program SortData reads data from a column in worksheet "Sorting",        |
' while, at the same, counting the number of data points.  The program     |
' then, orders the data read into a column of increasing values, as well   |
' as into a column of decreasing values.  These two columns are then       |
' shown in the "Sorting" worksheet in Excel or CALC.                        |
'--------------------------------------------------------------------------|
' Variables used:                                                          |
' ===============                                                          |
' x( ) : array containing the unsorted data read from the worksheet        |
' n    : size of array read                                                |
' xP( ): array containing data sorted in increasing order                  |
' xPP(): array containing data sorted in decreasing order                  |
'--------------------------------------------------------------------------|
'Variable declaration
    Dim x(100) As Double, xP(100) As Double, xPP(100) As Double
    Dim i As Integer, j As Integer, n As Integer, temp As Double
'Input data, and find sample size (n):
    n = 1  'Initialize sample size, n
    Do While Worksheets("Sorting").Range("B3").Cells(n, 1).Value <> ""
        x(n) = Worksheets("Sorting").Range("B3").Cells(n, 1).Value
        n = n + 1
    Loop
    n = n - 1   'Value of n adjusted after leaving loop
'Copy data x to xp (x-prime) and xpp(x-double prime)
    For i = 1 To n
        xP(i) = x(i): xPP(i) = x(i)
    Next
```

```
'Sort xp data in increasing (xp) and decreasing (xpp) order
    For i = 1 To n - 1
        For j = i + 1 To n
            If xP(j) < xP(i) Then
               temp = xP(j): xP(j) = xP(i): xP(i) = temp
            End If
            If xPP(j) > xPP(i) Then
               temp = xPP(j): xPP(j) = xPP(i): xPP(i) = temp
            End If
        Next
    Next
'Write out the sorted data back to the interface
    For i = 1 To n
        Worksheets("Sorting").Range("D3").Cells(i, 1) = xP(i)
        Worksheets("Sorting").Range("D3").Cells(i, 2) = xPP(i)
    Next i
End Sub
```

Figure 2. Code illustrating detailed program description using comment lines.

VBA subprograms

VBA includes two types of subprograms, *Sub*s and *Function*s. *Function*s typically return a single value to the calling program, whereas *Sub*s do not have to return a particular value at all. Subprograms subordinated to a main program may call other subprograms, thus creating a hierarchy of subprogram links in a particular program.

***Function* subprograms**

Function subprograms start with the particle *Function* followed by the name of the function, followed by parentheses with a list of function parameters with their type declared in the list. The function data type is also declared in the first line of a *Function* subprogram. A *Function* subprogram ends with the line *End Function*. In between the *Function* and *End Function* line you can include any number of variable definitions and VBA statements. The value that the function returns gets assigned to the function's name within the body of the function.

The typical form of a *Function* subprogram is shown next:

```
Function <function_name>(<parameter 1> As <type 1>, <parameter 2> As <type 2>, …, _
                                          <parameter n> As <type n>) _
                        As <data type for function>
   <function statements>
   <function_name> = some result
End Function
```

Example 1: Function with a single argument – A simple example of a *Function* subprogram would be a simple function *f* that can be used as an integrand, to evaluate, for example, a numerical integral:

```
Function f(x As Double) As Double
      f = x^3 - 3*x + 2.5
End Function
```

In this example, *f* is the name of the function and *x* is the only parameter used. Both the function *f* and the parameter *x* are defined as *Double* precision data.

A main *Sub* program that invokes this function *f*(*x*) is illustrated below:

```
Sub NumericalIntegral()
   Dim I As Double, a As Double, b As Double, Dx As Double
   Dim n As Integer, k As Integer
   a = InputBox("Enter a:") : b = InputBox("Enter b:")
   n = InputBox("Enter number of subintervals in [a,b], n:")
   Dx = (b-a)/n : I = 0.0
   For k = 1 To n
       I = I + f(a+(k-1)*Dx)
   Next k
   I = I * Dx
   MsgBox("The integral of f(x) from a = " & CStr(a) & _
          " to b = " & CStr(b) & ", using n = " & CStr(n) & _
          " subintervals is: " & CStr(I))
End Sub
```

The function *f*(*x*) is called in the statement `I = I + f(a+(k-1)*Dx)`. In the function call, just shown, the term `a+(k-1)*Dx` is called the function argument. We say that this argument is *passed* to the function, so that *x*, in the function, takes the value `a+(k-1)*Dx`, and the function *f*(*x*) is thus evaluated. The resulting value, i.e., `f(a+(k-1)*Dx)`, is returned to the calling statement, `I = I + f(a+(k-1)*Dx)`. The value of the numerical integral, *I*, is accumulated as indicated in the calling statement, and the loop is repeated as indicated above. A different evaluation of `f(a+(k-1)*Dx)` results from each pass of the loop controlled by the *For ... Next* statement.

Passing arguments by reference (ByRef) or by value (ByVal)
Arguments can be passed to a *Function* subprogram (or to a *Sub* subprogram) in two ways: *by reference*, or by value. These two approaches are defined below:

- Passing an argument by reference means that the *Function* (or *Sub*) receives not only the value of the argument, but also the address of the memory location where the argument resides in the computer memory. When an argument is passed by reference, any change performed on the argument by the receiving *Function* (or *Sub*) will affect the value of the argument in the calling main program (or subprogram). To pass an argument by reference, the corresponding parameter in the function definition statement must be preceded by the particle *ByRef*. If no specification is made on the parameter in the function definition line, it is assumed, by default, that the arguments in function calls will be passed by reference. In the example of function *f*(*x*), shown above, the arguments of *f* are passed (by default) by reference. Thus, the code of the function *f*(*x*) could have been written as:

```
      Function f(ByRef x As Double) As Double
         f = x^3 - 3*x + 2.5
      End Function
```

Notice, however, that this function doesn't actually perform any changes on the value of the parameter *x*. Therefore, in this particular example, it won't make any difference if you pass arguments to *f*(*x*) by reference or by value.

- Passing an argument by value means that the *Function* (or *Sub*) receives only the value of the argument. You can think of passing an argument by value as making a copy of the value of the argument and sending that copy into the receiving *Function* (or *Sub*). Even if the receiving *Function* (or *Sub*) changes somehow the value of the argument, that change (or changes) will not be reflected in the value of the argument in the calling program (or subprogram). To pass an argument by value, the corresponding parameter in the function definition statement must be preceded by the particle *ByVal*. If we wanted to specify that the argument corresponding to the parameter *x* in function *f*(*x*), shown above, is to be passed by value, the function *f*(*x*) could have been written as:

```
Function f(ByVal x As Double) As Double
   f = x^3 - 3*x + 2.5
End Function
```

Example 2: The Swamee-Jain function for calculating friction factors in pipelines. In pipeline hydraulics, to calculate the energy head loss (energy loss per unit weight), h_f, that a pipe of length, *L*, and diameter, *D*, carrying a fluid of kinematic viscosity[1], ν, at a discharge or volumetric flow rate (volume per unit time), *Q*, we use the Darcy-Weisbach equation:

$$h_f = \frac{8 \cdot f \cdot L \cdot Q^2}{\pi^2 \cdot g \cdot D^2}$$

where *g* is the acceleration of gravity, and *f* is the Darcy-Weisbach friction factor (a dimensionless value in the range 0.001 to 0.100). It can be proven that the friction factor, *f*, is a function of a dimensionless quantity known as the *relative roughness* of the pipe, defined as *RR* = *e/D*, where *e* is the absolute roughness of the pipe (a quantity that depends on the pipe material), and of a second dimensionless quantity known as the *Reynolds number*, defined as $Rey = \frac{4 \cdot Q}{\pi \cdot \nu \cdot D}$. Thus, *f* = *f*(*RR*,*Rey*).

Laboratory experiments indicate that this function can be expressed using the Swamee-Jain equation, namely,

$$fSJ(RR, Rey) = \frac{0.25}{\left(\log_{10}\left(\frac{RR}{3.7} + \frac{5.74}{Rey^{0.9}}\right)\right)^2} .$$

The Swamee-Jain function can be coded into a *Function* subprogram as shown below:

1 The kinematic viscosity is a property of a flowing fluid related to the facility with which the fluid flows in a closed conduit. The kinematic viscosity increases with the fluid temperature. The larger the kinematic viscosity is, the more energy is required to move the same amount of fluid through a given pipeline.

```
Function fSJ(ByVal RelativeRoughness As Double, ByVal ReynoldsNumber As Double) _
         As Double
    fSJ = _
        (WorksheetFunction.Log10(RelativeRoughness/3.7+5.74/ReynoldsNumber^0.9))^2
End Function
```

To code function *fSJ* we borrow the worksheet function *Log10*.

A main *Sub* subprogram to calculate pipe headlosses can be coded as follows:

```
Sub PipeHeadLosses()
   Dim hf As Double, L As Double, D As Double, g As Double
   Dim nu As Double, pi As Double, Q As Double, RR As Double
   Dim Rey As Double, f As Double, e As Double
   pi = WorksheetFunction.PI()
   L = InputBox("Pipe length, L:") : D = InputBox("Pipe diameter, D:")
   e = InputBox("Roughness, e:") : nu = InputBox("Kin. Viscosity, nu:")
   Q = InputBox("Discharge, Q:") : g = InputBox("Gravity, g:")
   RR = e / D : Rey = 4 * Q / (pi * nu * D) : f = fSJ(RR,Rey)
   hf = 8 * f * L * Q^2 / (pi^2 * g * D^5)
   MsgBox("Head loss hf = " & Cstr(hf))
End Sub
```

In this program, function *fSJ* gets called by the statement **`f = fSJ(RR,Rey)`**. The values of *RR* and *Rey* in the main program (*PipeHeadLosses*) are passed on, by value, to function *fSJ*. The order in which the arguments *RR* and *Rey* are listed in the calling statement corresponds to the listing of the corresponding parameters *RelativeRoughness* and *ReynoldsNumber* in the function definition. Thus, the parameters in a *Function* (or *Sub*) call must correspond in order and type to the parameters in the *Function* (or *Sub*) definition statement. The following figure illustrates the correspondence of arguments and parameters for the case of function *fSJ* and program *PipeHeadLosses*.

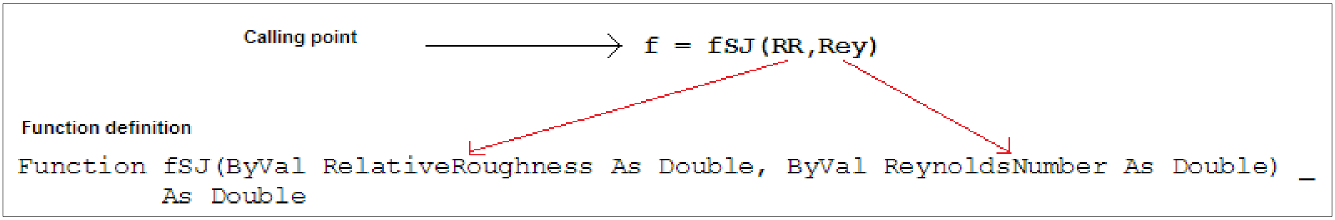

Calling point
f = fSJ(RR,Rey)
Function definition
Function fSJ(ByVal RelativeRoughness As Double, ByVal ReynoldsNumber As Double) _
As Double

Figure 3. Correspondence between function call arguments and function definition parameters for function *fSJ*.

When the statement **`f = fSJ(RR,Rey)`** is executed in the main program *PipeHeadLosses*, the values of the *Double* precision variables *RR* and *Rey* are passed to function *fSJ*. Within the function, the value of *RR* replaces parameter *RelativeRoughness* while the value of *Rey* replaces parameter *ReynoldsNumber*. Calculations are performed within the function, and the resulting value of *fSJ* is then returned to the main program, where that value is assigned to variable *f*. The processing of the program then continues past the point where function *fSJ* was called.

Correspondence between arguments and parameters – In summary, when a *Function* (or *Sub*) is called:

- There should be a one-to-one correspondence between the arguments of the *Function* (or *Sub*) call and the parameters in the *Function* (or *Sub*) definition statement.
- Arguments and parameters must correspond in the order they're listed and in their data types.

Notes on the coding of *Function* subprograms:

- *Function* subprograms can be coded in a VBA *Module* or in the code window associated with individual spreadsheets or with the workbook.
- A *Function* subprogram defined in a VBA *Module* can be called from the worksheet interface by typing a formula that uses the *Function*'s name in a worksheet cell. In this case, the *Function* subprogram is referred to as a *user-defined worksheet function*, or, simply a *user-defined function*.
- You can place a *Function*'s code in a VBA Module with many other *Function*s and/or *Sub*s. However, each *Function* and/or *Sub* must have a different name. Coding any two subprograms with the same name will result in a run-time error.
- The name of a *Function* (or *Sub*) subprogram follows the rules of naming variables: use letters, numbers, or the undersign character (_) only; the name must start with a letter, and it could be up to 256 characters long.
- **Warning**: if you are going to use a *Function* subprogram as a user-defined function, you should avoid using a function name that can be confused with cell references. For example, if you give your *Function* subprogram the name *F11*, there could be a confusion with cell F11, and, most likely, you will get an error in evaluating this particular function. Use, instead, a name that includes only letters and the undersign, e.g., *F_eleven* or *Feleven*.
- Also, when naming your own functions, avoid using names that belong to Excel worksheet functions or VBA functions, e.g., ATN, SIN, PHI, NORMDIST, etc.
- Besides the parameters in a *Function* (or *Sub*) subprogram, you can use additional variables within the body of the subprogram. These *local variables* can be defined through the use of *Dim* statements, and will be active only within the subprogram.

Example 3: Using local variables in a *Function* subprogram**.** Probabilities for the standard normal distribution (mean value = 0.0, standard deviation = 1.0) are calculated using:

$$\Phi(z)=P(Z<z)=\frac{1}{\sqrt{2\cdot\pi}}\cdot\int_{-\infty}^{z} e^{-\frac{\xi^2}{2}}\,d\xi \quad .$$

The integral shown in this formula can only be calculated numerically, however, having negative infinity as the lower limit makes the numerical calculation impossible. Instead, we make use of the symmetry of the integrand, $e^{-\frac{\xi^2}{2}}$, about $z = 0$, to write, for $z > 0$,

$$\Phi(z)=\frac{1}{2}+\frac{1}{\sqrt{2\cdot\pi}}\int_0^z e^{-\frac{\xi^2}{2}}d\xi \quad .$$

Also, because of the symmetry of $e^{-\frac{\xi^2}{2}}$, about $z = 0$, we can write, for $z < 0$:

$$\Phi(z)=\frac{1}{2}-\frac{1}{\sqrt{2\cdot\pi}}\cdot\int_0^{|z|} e^{-\frac{\xi^2}{2}}d\xi \quad .$$

We can introduce a function to account for the integral shown in the two expressions, above, say,

$$\Phi_p(z)=\frac{1}{\sqrt{2\cdot\pi}}\cdot\int_0^z e^{-\frac{\xi^2}{2}}d\xi \quad ,$$

so that the function $\Phi(z)$ can be written as:

$$\Phi(z)=\begin{Bmatrix}\frac{1}{2}-\Phi_p(|z|),\ for\ z<0\\ \frac{1}{2}+\Phi_p(z),\ for\ z\geq 0\end{Bmatrix} \quad .$$

To evaluate $\Phi_p(z)$ we can create a *Function* subprogram as follows:

```
Function PHI_p(z As Double) As Double
   'Numerical integration of exp(-xi^2/2)
   'from 0 to z using 1000 subintervals
   Dim n As Integer, a As Double, b As Double
   Dim Dx As Double, I As Double, k As Integer
   Dim f1 As Double, f2 As Double, C As Double
   C = Sqr(2.0*WorksheetFunction.Pi())
   n = 1000 : a = 0 : b = z : Dx = (b-a)/n : I = 0
   For k = 1 To n-1
       f1 = exp(-(a+(k-1)*Dx)^2/2.0)/C
       f2 = exp(-(a+k*Dx)^2/2.0)/C
       I = I + (f1+f2)/2.0
   Next k
   I = I * Dx
   PHI_p = I
End Function
```

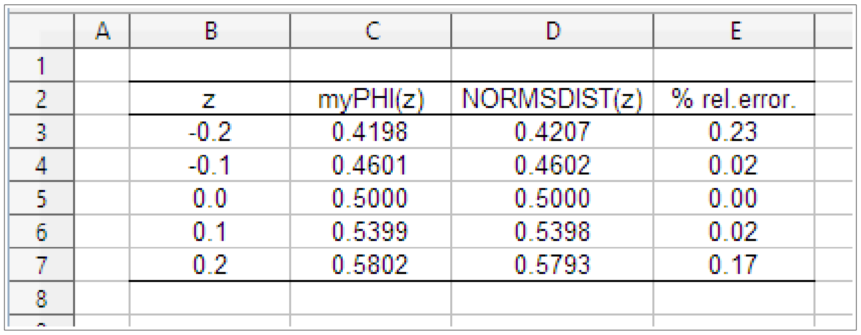
Notice the use of variables *n, a, b, Dx, I, k, f1, f2*, and *C*. These variables are all local variables to function *PHI_p*. These variables are defined, created, and used within function *PHI_p*, alone, and have no meaning outside of the function. It is said that the *scope of these variables* is the function only.

Function $\Phi(z)$ can be coded as follows:

```
Function myPHI(z As Double) As Double
   'Function PHI calculates the cumulative probability
   'function for the standard normal distribution
   'PHI(z) = P(Z<z)
   If z <= 0 Then
       myPHI = 0.5 - PHI_p(Abs(z))
   Else
       myPHI = 0.5 + PHI_p(z)
   End If
End Function
```

It's worth noting that Excel provides function NORMSDIST(z) to calculate the function $\Phi(z)$. Excel also provides a function called *PHI*, that calculates the values of the integrand $\varphi(\xi)=\frac{1}{\sqrt{2\cdot\pi}}\cdot e^{-\frac{\xi^2}{2}}$, which is used in defining $\Phi_p(z)$. To avoid conflict between the Phi function $\Phi(z)$, developed in this section, and Excel's own PHI function, the user-defined function shown above was named *myPHI*.

The following table shows a comparison of values calculated using both user-defined function *myPHI* and worksheet function *NORMSDIST*. The table includes the percent relative error in the value estimated by function *myPHI*.



	A	B	C	D	E
1					
2		z	myPHI(z)	NORMSDIST(z)	% rel.error.
3		-0.2	0.4198	0.4207	0.23
4		-0.1	0.4601	0.4602	0.02
5		0.0	0.5000	0.5000	0.00
6		0.1	0.5399	0.5398	0.02
7		0.2	0.5802	0.5793	0.17
8					

<u>Using arrays as arguments</u>

The examples presented so far in this chapter have used single variables as arguments of *Function* subprograms. To pass arrays as arguments to a *Function* subprogram:

- The array arguments, with a given size and number of dimensions, must exist in the calling program. These array arguments can be defined using a *Dim* or a *ReDim* statement within the calling program. Please notice that redimensioning of an array may occur inside the subprogram.
- The name of an array argument must be listed followed by a blank set of parentheses in the call to the *Function* (or *Sub*) subprogram.
- The name of an array parameter in the line defining a *Function* (or *Sub*) subprogram should also be listed followed by a blank set of parentheses.

- As indicated earlier, there should be a one-to-one correspondence, both in order and data type, between arguments in a *Function* (or *Sub*) call statement and parameters in the *Function* (or *Sub*) definition.
- The lower and upper limits of each dimension of an array that has been passed to a *Function* (or *Sub*) subprogram, can be found using functions *LBound*(<*array_name*>, <*dimension position*>) and *Ubound*(<*array_name*>,<*dimension position*>). These functions can be useful to determine the indices of elements in a particular array argument. If using one-dimensional arrays, the argument <*dimension position*> can only take the value of 1 (there's only one dimension). If using two-dimensional arrays, the argument <*dimension position*> in functions *LBound* and *UBound* could refer to the first or second dimensions in the arrays.

NOTE – Using three-dimensional, and higher-dimension arrays – So far we have used one-dimensional and two-dimensional arrays, e.g., *Dim X*(10) *As Double* defines a double-precision one-dimensional array, while *Dim A*(10,5) *As Double* defines a double-precision two-dimensional array.

As indicated earlier, one-dimensional arrays can be thought of as vectors of data, while two-dimensional arrays can be thought of as matrices.

A three-dimensional array can be defined using, for example, *Dim Z*(5,5,3) *As Double*, etc. Visualizing a three-dimensional array would require thinking of two of the dimensions as rows and columns of a matrix and the third dimension as a page number in a notebook, or a tab in a filing cabinet, where the matrix defined by the first two dimensions is recorded or stored.

We can define arrays of any dimensions, e.g., an array of four dimensions can be defined as *Dim Y*(3,5,2,5) *As Double*, etc.

In a array declaration such as *Dim R*(1 To 5, 0 To 3, -2 To 4), application of the functions *LBound* and *UBound* would produce the following results:

LBound(R,1) → 1; *UBound*(R,1) → 5;
LBound(R,2) → 0; *UBound*(R,2) → 3;
LBound(R,3) → -2; *UBound*(R,3) → 4;

If you use the specification *Option Base 1* at the top of your code window, then, for any array thus defined, e.g., *Dim X*(10) *As Double*, function *LBound*(X,1) → 1.

Example 4: Mean value of a vector – case 1. Given a vector $x = \{x_1, x_2, \dots, x_n\}$ representing a sample of numerical data of size n, the mean value is calculated as $\bar{x} = \frac{1}{n} \cdot \sum_{k=1}^{n} x_k$. This calculation can be coded into a *Function* subprogram with array parameters, as follows:

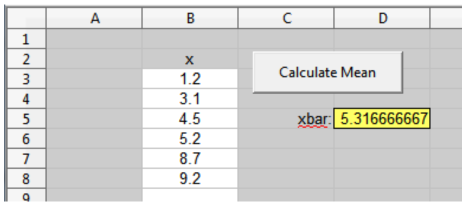
```
Function MeanOfX(ByRef XData As Double) As Double
  'Function for calculating the mean of vector XData()
  'NOTE: Option Base 1 is used for defining arrays
   Dim SumOfX As Double, k As Integer, n As Integer
   n = UBound(XData,1) : SumOfX = 0.0
   For k = 1 To n
      SumOfX = SumOfX  + XData(k)
   Next k
   MeanOfX = SumOfX/n
End Function
```

Notice that *XData* is defined in the *Function* definition line as a *Double* precision variable. The definition *ByRef XData As Double* does not provide any hint that *XData* is an array variable. The *Function* only detects that *XData* is an array when the *UBound* function is activated in the line: **`n = Ubound(XData,1)`**. Later on, the line **`SumOfX = SumOfX  + XData(k)`** shows variable *XData* with a sub-index *k*, which indicates that *XData* is indeed an array variable.

To activate function *MeanOfX* we can use the following program:

```
Sub MeanValueExample01()
    'Main program using Function MeanOfX(...) -- using interface
    Dim y() As Double, n As Integer, yBar As Double, k As Integer
    n = 0
    For k = 1 to 1000
      If Worksheets("XData01").Range("B3").Cells(k,1).Value <> "" Then
         n = n + 1
      Else
         Exit For
      End If
    Next k
    Redim y(n)
    For k = 1 To n
      y(k) = Worksheets("XData01").Range("B3").Cells(k,1).Value
    Next k
    yBar = MeanOfX(y())
    Worksheets("XData01").Range("D5").Value = yBar
End Sub
```

This program is run from an interface in worksheet “XData01”, as shown below:



	A	B	C	D
1				
2		x		
3		1.2		
4		3.1		
5		4.5	xbar:	5.316666667
6		5.2		
7		8.7		
8		9.2		

Figure 4. Interface for the main program *MeanValueExample02* shown above.

Example 5: Mean value of a vector – case 2. In this case, the array of data, *XVData*, is loaded in the code using the function *Array*( ). The variable *XVData* is defined as a *Variant* type, and then function *Array*( ) is used to load a list of numbers separated by commas. Because the variable *XVData* is passed to the function *MeanOfXVData*( ) as a *Variant,* the corresponding parameter of the function is also defined as *Variant.*

```
Option Explicit
Option Base 1
'-----------------------------------------------------------------
Sub MeanValueExample02
   'Main program using Function MeanOfXV(...) -- using Array
   Dim yV As Variant, yVBar As Double
   yV = Array(2.5, 2.3, 3.2, 1.8, 4.7, 5.2, 4.1)
   yVBar = MeanOfXV(yV())
   MsgBox("yBar = " & CStr(yVBar))
End Sub
'-----------------------------------------------------------------
Function MeanOfXV(ByRef XVData() As Variant) As Double
     'Function for calculating the mean of vector XData()
     'NOTE: Option Base 1 is used for defining arrays
   Dim SumOfXV As Double, k As Integer, n As Integer
    n = UBound(XVData,1) : SumOfXV = 0.0
   For k = 1 To n
      SumOfXV = SumOfXV  + XVData(k)
   Next k
   MeanOfXV = SumOfXV/n
End Function
```

The arrangement of subprograms shown above, in which the main *Sub* program is listed above the subordinated *Function* subprogram is typical of modular programs such as this.

NOTE: Use of the function *Array*( ) to load a vector of data, as illustrated in the code above, is not very useful while processing large amounts of data since the data in the vector needs to be typed in the code. Using entries from an interface, as illustrated earlier using the code for main program *MeanValueExample02* and function *MeanOfX*( ), is a more practical way of entering array data for processing into a VBA program for larger arrays. For small arrays, using *Array*( ) is an easier approach.

Example 6: Mean value, variance, and standard deviation of a vector. This example extends the case presented earlier in association with the interface of Figure 4, by adding one more function to the code: a function that calculate the variance of the sample $x = \{x_1, x_2, \dots, x_n\}$ of size n, defined as:

$$s_x^2 = \frac{1}{n-1} \cdot \sum_{k=1}^{n} (x_i - \bar{x})^2 \quad .$$

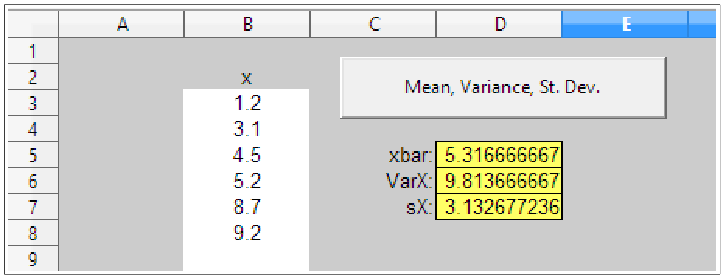
Thus, calculation of the variance requires us to first calculate the mean, $\bar{x}$. Also, the standard deviation, s_x, is simply the square root of the variance. The code for the main program, mean value function, and variance is presented next. The standard deviation is calculated in the main program.

```
Option Base 1
'-----------------------------------------------------------------
Sub MeanValueExample03()
    'Main program using Function MeanX(...) -- using interface
    Dim y() As Double, n As Integer, k As Integer
    Dim yBar As Double, sy2 As Double, sy As Double
    n = 0
    For k = 1 to 1000
      If Worksheets("XData03").Range("B3").Cells(k,1).Value <> "" Then
         n = n + 1
      Else
         Exit For
      End If
    Next k
    Redim y(n)
    For k = 1 To n
      y(k) = Worksheets("XData03").Range("B3").Cells(k,1).Value
    Next k
    yBar = MeanX(y())
    sy2 = VarianceOfX(y(),yBar)
    sy = Sqr(sy2)
    Worksheets("XData03").Range("D5").Value = yBar
    Worksheets("XData03").Range("D6").Value = sy2
    Worksheets("XData03").Range("D7").Value = sy
End Sub
'------------------------------------------------------------------
Function MeanX(ByRef XData() As Double) As Double
    'Function for calculating the mean of vector XData()
    'NOTE: Option Base 1 is used for defining arrays
   Dim SumX As Double, k As Integer, n As Integer
    n = UBound(XData,1) : SumX = 0.0
   For k = 1 To n
      SumX = SumX  + XData(k)
   Next k
   MeanX = SumX/n
End Function
'------------------------------------------------------------------
Function VarianceOfX(ByRef XData() As Double, BarX As Double) As Double
    'Function for calculating the variance of vector XData()
    'NOTE: Option Base 1 is used for defining arrays
   Dim SumOfDX2 As Double, k As Integer, n As Integer
    n = UBound(XData,1) : SumOfDX2 = 0.0
   For k = 1 To n
      SumOfDX2 = SumOfDX2  + (XData(k)-BarX)^2
   Next k
   VarianceOfX = SumOfDX2/(n-1)
End Function
```

In this example, function *VarianceOfX* requires that the mean value, *BarX*, be calculated beforehand. An alternative would be to call the mean value function *MeanX* from within function *VarianceOfX*. This approach would be useful if only the variance is required in the main program. Thus, an alternative form of function *VarianceOfX* is as shown below.

```
Function VarianceOfX_ALT(ByRef XData() As Double) As Double
    'Function for calculating the variance of vector XData()
    'NOTE: Option Base 1 is used for defining arrays
   Dim SumOfDX2 As Double, k As Integer, n As Integer, BarX As Double
   BarX = MeanX(XData)
   n = UBound(XData,1) : SumOfDX2 = 0.0
   For k = 1 To n
      SumOfDX2 = SumOfDX2  + (XData(k)-BarX)^2
   Next k
   VarianceOfX = SumOfDX2/(n-1)
End Function
```

The interface corresponding to the code of Example 6 is shown below:



	A	B	C	D	E
1					
2		x			
3		1.2			
4		3.1			
5		4.5		xbar:	5.316666667
6		5.2		VarX:	9.813666667
7		8.7		sX:	3.132677236
8		9.2			
9					

Figure 4(b). Interface for Example 6.

***Sub* subprograms**

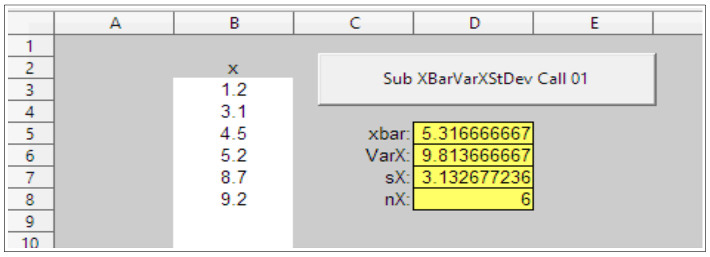
The particle *Sub* in VBA is an abbreviation of the word *Sub*routine, a reference to a generic subprogram linked to a main program. The name *subroutine* originated with FORTRAN, the first-ever high-level computer programming language created in 1957.

Sub subprograms start with the particle *Sub* followed by a name, and end with the *End Sub* statement. A *Sub* can be used as the main program, in which case, no parameters are needed. *Sub* subprograms that are called from a main *Sub* program typically require parameters, so that information can be passed on from the calling *Sub*. If a *Sub* is used specifically for performing changes in a worksheet (or worksheets), rather than for calculations, parameters may not be needed. *Sub* subprogram are typically designed so that they return more than one value to the calling program (unlike *Function* subprograms, which typically return a single value).

Example 7: Mean value, variance, and standard deviation of a vector redone as a *Sub*. For example, in reference to the example of the variance calculation presented above, suppose that we want a subprogram that returns three values to the calling program, namely, the mean, the variance, and the standard deviation. Obviously, function *VarianceOfX* cannot be used for this purpose, as it can only return a single value (i.e., the variance). The solution could be to transform function *VarianceOfX* into a *Sub* subprogram, called *XbarVarSTDev*, that would calculate the three required quantities. This is illustrated below.

```
Option Base 1
'-----------------------------------------------------------------
Sub MeanValueExample04()
    'Main program using Function MeanX(...) -- using interface
    Dim y() As Double, n As Integer, k As Integer
    Dim yBar As Double, VarY As Double, sY As Double
    n = 0
    For k = 1 to 1000
      If Worksheets("XData04").Range("B3").Cells(k,1).Value <> "" Then
         n = n + 1
      Else
         Exit For
      End If
    Next k
    Redim y(n)
    For k = 1 To n
      y(k) = Worksheets("XData04").Range("B3").Cells(k,1).Value
    Next k
    Call XBarVarStDev(y(), yBar, VarY, sY)
    Worksheets("XData04").Range("D5").Value = yBar
    Worksheets("XData04").Range("D6").Value = VarY
    Worksheets("XData04").Range("D7").Value = sY
End Sub
'------------------------------------------------------------------
Sub XBarVarStDev(ByRef XData() As Double, ByRef XBar AS Double, _
                 ByRef VarX As Double, ByRef sX As Double)
    'Sub subprogram to calculate mean (XBar), variance (VarX),
    'and standard deviation (sX) of a vector of data XData
    Dim k As Integer, SumX As Double, n As Integer
    n = UBound(XData,1)
    SumX = 0.0
    For k = 1 To n
        SumX = SumX + XData(k)
    Next k
    XBar = SumX/n
    SumX = 0.0
    For k = 1 To n
      SumX = SumX + (XData(k)-XBar)^2
    Next k
    VarX = SumX/(n-1)
    sX = Sqr(VarX)
End Sub
```

The interface for this code is shown below.



	A	B	C	D	E
1					
2		x			
3		1.2			
4		3.1			
5		4.5	xbar:	5.316666667	
6		5.2	VarX:	9.813666667	
7		8.7	sX:	3.132677236	
8		9.2	nX:	6	
9					
10					

Figure 4(c). Interface for Example 7.

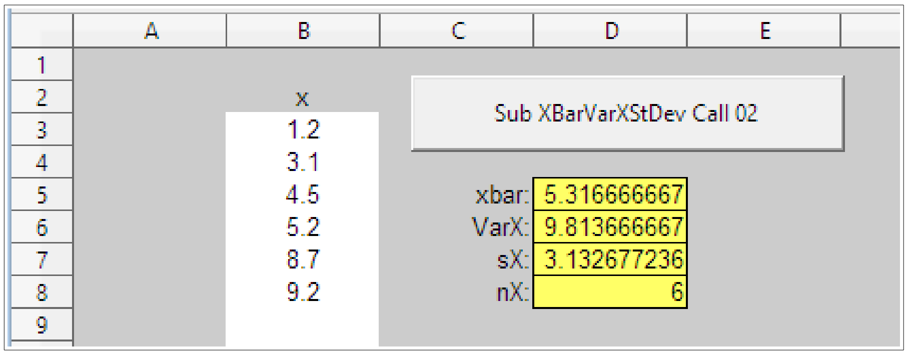
Example 8: Mean value, variance, and standard deviation calculated in a modular fashion. The code shown above, can be modified even further turning it into a purely modular code, where the main program simply calls *Sub* subprograms to (i) input data from the interface; (ii) calculate mean, variance, standard deviation, and sample size; and (iii) write the result to the interface. The *Sub* that calculates the mean, variance, and standard deviation, in turn, calls two functions: (i) one to calculate the mean; and (ii) one to calculate the variance. The standard deviation is just the square root of the variance, therefore, no separate *Function* subprogram is required to calculate it. The complete code is shown below.

```
Option Base 1

'----------------------------------------------------------------
Sub MeanValueExample05()
    'Main program calling three different Subs
    Dim y() As Double, n As Integer, k As Integer
    Dim yBar As Double, VarY As Double, sY As Double
    Dim nY As Integer
    Call ReadMyData(y())
    Call XBarVarStDev(y(), yBar, VarY, sY, nY)
    Call ShowMyResults(yBar, VarY, sY, nY)
End Sub
'-----------------------------------------------------------------
Sub ReadMyData(ByRef myData() As Double)
'This Sub subprogram inputs the data myData from the interface "XData05"
    Dim k As Integer, nData As Integer
   nData = 0
    For k = 1 to 1000
      If Worksheets("XData05").Range("B3").Cells(k,1).Value <> "" Then
         nData = nData + 1
      Else
         Exit For
      End If
    Next k
    Redim myData(nData)
    For k = 1 To nData
      myData(k) = Worksheets("XData05").Range("B3").Cells(k,1).Value
    Next k
End Sub
'-----------------------------------------------------------------
Sub XBarVarStDev(ByRef XData() As Double, ByRef XBar AS Double, _
                 ByRef VarX As Double, ByRef sX As Double, _
                 ByRef nDataPoints As Integer)
    'Sub subprogram to calculate mean (XBar), variance (VarX),
    'and standard deviation (sX) of a vector of data XData
    Dim k As Integer, SumX As Double, n As Integer
    nDataPoints = UBound(XData,1)
    XBar = XMean(XData())
    VarX = XVariance(XData())
    sX = Sqr(VarX)
End Sub
'-------------------------------------------------------------------
```

```
Function XMean(ByRef XXData() As Double) As Double
    'Function XMean calculates the mean of vector XXData
   Dim nPoints As Integer, k As Integer, SumXX As Double
   nPoints = UBound(XXData,1)
   SumXX = 0.0
   For k = 1 To nPoints
      SumXX = SumXX + XXData(k)
   Next k
   XMean = SumXX/nPoints
End Function
'---------------------------------------------------------------------
Function XVariance(ByRef DataX() As Double) As Double
    'Function XVariance calculate the variance of vector DataX
   Dim xBar As Double, SumX2 As Double
   Dim k As Integer, nDataX As Integer
   nDataX = UBound(DataX,1)
   xBar = XMean(DataX())
   SumX2 = 0.0
   For k = 1 To nDataX
      SumX2 = SumX2 + (DataX(k)-xBar)^2
   Next k
   XVariance = SumX2/(nDataX-1)
End Function
'---------------------------------------------------------------------
Sub ShowMyResults(ByRef Value1 As Double, ByRef Value2 As Double, _
                  ByRef Value3 As Double, ByRef Value4 As Integer)
'This Sub outputs the values of mean, variance, std. deviation, and sample size
    Worksheets("XData05").Range("D5").Value = Value1
    Worksheets("XData05").Range("D6").Value = Value2
    Worksheets("XData05").Range("D7").Value = Value3
    Worksheets("XData05").Range("D8").Value = Value4
End Sub
```

The interface for Example 8 is shown next.



	A	B	C	D	E
1					
2		x	Sub XBarVarXStDev Call 02		
3		1.2			
4		3.1			
5		4.5	xbar:	5.316666667	
6		5.2	VarX:	9.813666667	
7		8.7	sX:	3.132677236	
8		9.2	nX:	6	
9					

Figure 4(d). Interface for Example 8.

NOTE: In all the *Function* and *Sub* subprograms of the example shown above the arguments were passed to the subprograms by reference (*ByRef*). This is required in this case because, from the start, we called a *Sub* subprogram to input the data (*Sub ReadMyData*), which is then sent to *Sub XbarVarStDev* to calculate the mean, variance, and standard deviation, and later on to *Sub ShowMyResults* for output. *Sub XbarVarStDev*, in turn, calls *Functions XMean* and *XVariance*. In other words, the original data is not defined in the main program, but in a subroutine, which basically forces us to use *ByRef* in all *Sub* and *Function* calls, to link these data values properly.

If you try, for example, to define the parameters of function *XMean* by value, e.g.,

```
Function XMean(ByVal XXData() As Double) As Double
```

when you run the program you will find an error message indicating that an *Object variable* [was] *not set*, and pointing to the definition of function *XMean*. Changing the definition of this function to

```
Function XMean(ByRef XXData() As Double) As Double
```

resolves the situation.

Thus, if you decide to use *ByVal* in the definition of a *Sub* or *Function*, and if you run into a "*Object variable not set*" error message pointing to the definition of that *Sub* or *Function*, change the particle *ByVal* to *ByRef* to fix this problem. As indicated earlier, passing arguments by reference (i.e., using *ByRef*) is the default option for VBA *Sub* and *Function* subprograms. Therefore, the use of the particle *ByRef* can be omitted, but keep in mind that all the data transfer between subprograms will be done by reference in such case.

Figure 5, below, illustrates the relationship between arguments in *Sub* and *Function* calls and their corresponding subprogram parameters for the code presented in this example. It is obvious from the code, shown above, and from Figure , that arguments and parameters do not need to have the same name, however, the order in which they're defined and their corresponding data types need to be preserved.

Remember that, when passing arguments by reference (i.e., using *ByRef*), any change performed on the argument values within a subprogram will be reflected in the main program. For example, if you send a one-dimensional array, using *ByRef*, to a *Sub* subprogram to be sorted, the original vector will be lost. One way to avoid loosing your original array in a situation like this is to copy your original array to an auxiliary array where the sorting gets performed.

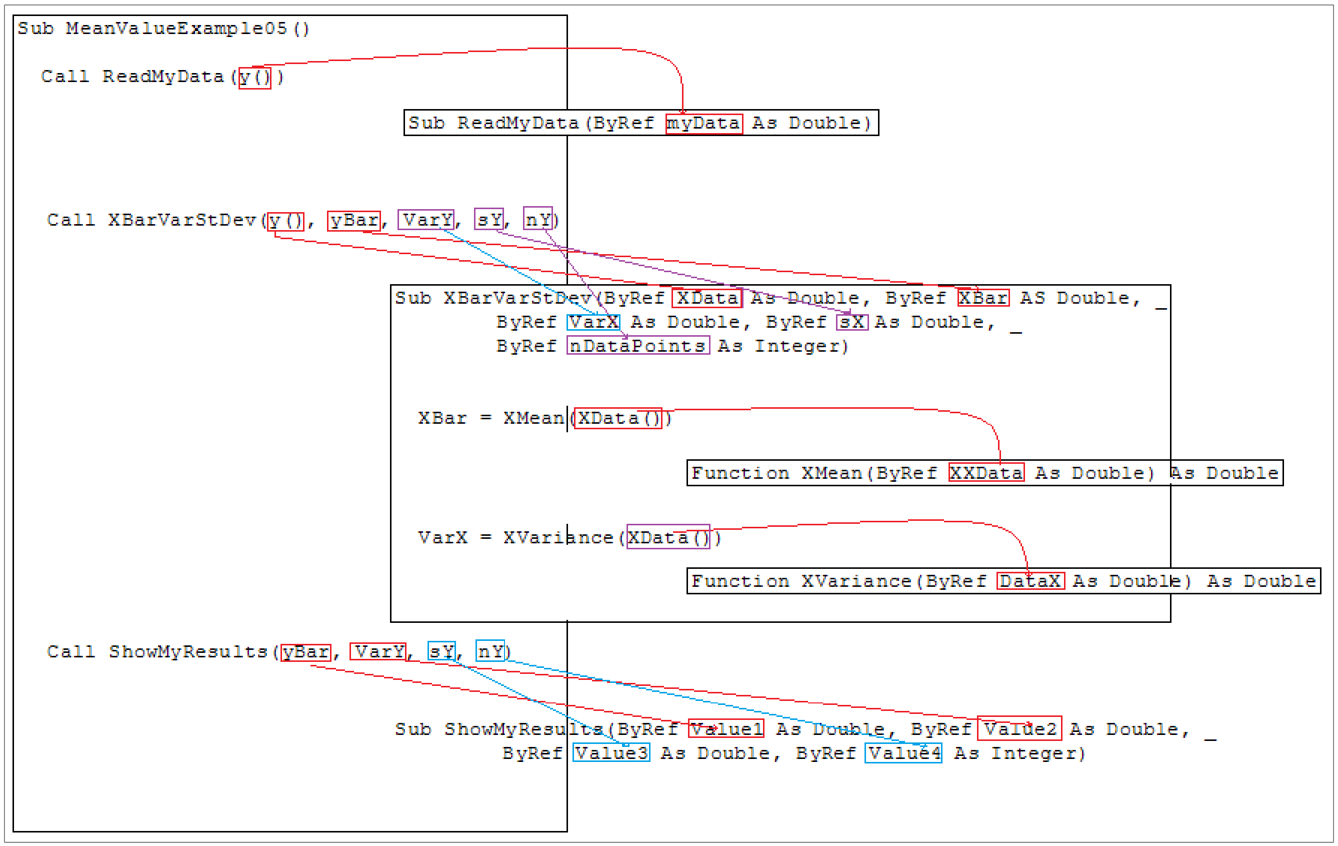

Sub MeanValueExample05()
Call ReadMyData(y())
Sub ReadMyData(ByRef myData As Double)
Call XBarVarStDev(y(), yBar, VarY, sY, nY)
Sub XBarVarStDev(ByRef XData As Double, ByRef XBar AS Double, _
ByRef VarX As Double, ByRef sX As Double, _
ByRef nDataPoints As Integer)
XBar = XMean(XData())
Function XMean(ByRef XXData As Double) As Double
VarX = XVariance(XData())
Function XVariance(ByRef DataX As Double) As Double
Call ShowMyResults(yBar, VarY, sY, nY)
Sub ShowMyResults(ByRef Value1 As Double, ByRef Value2 As Double, _
ByRef Value3 As Double, ByRef Value4 As Integer)

Figure 5. Relationships between arguments in *Sub* and *Function* calls and the corresponding parameters for the case of the code shown above.

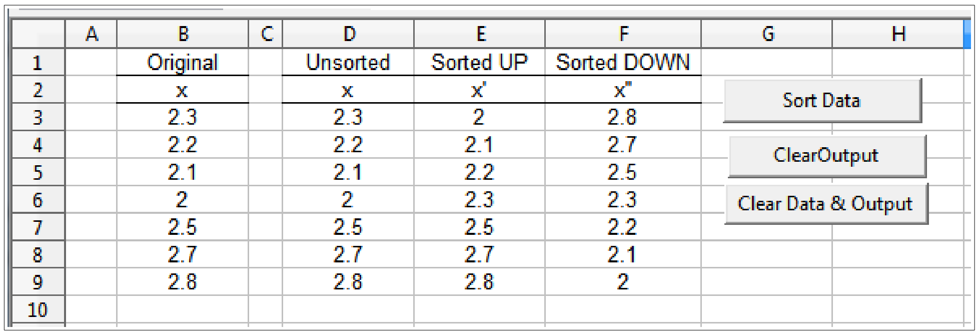
__**Example 9** – Sorting an array in a subprogram__ – In this example a main program (*MainProgramSortData*) calls three subroutines: one to read the data from a worksheet (*ReadMyData*), one to copy the data into two other arrays and sort those arrays in increasing and decreasing order, respectively (*SortMyData*), and one to print the original array and the sorted arrays (*WriteOutMyData*). By copying the original array into two other arrays, we avoid affecting the original array in the sorting subroutine. The code corresponding to the main program and the associated subroutines is presented in next page.

```
Option Explicit
Option Base 1
'------------------------------------------------------------------------
Sub MainProgramSortData()
   Dim x() As Double, xP() As Double, xPP() As Double
   Call ReadMyData(x())
   Call SortMyData(x(),xP(),xPP())
   Call WriteOutMyData(x(), xP(), xPP())
End Sub
'---------------------------------------------------------------------------
```

```
Sub ReadMyData(ByRef xData() As Double)
   Dim nData As Integer, i As Integer
   'Find sample size (n):
    nData = 1  'Initialize sample size, n
    Do While Worksheets("SortMyData").Range("B3").Cells(nData,1).Value <> ""
        nData = nData + 1
    Loop
    nData = nData - 1   'Value of n adjusted after leaving loop
' Redimensioning the dynamic arrays
    ReDim xData(nData)
'Input data, and copy x to xP and xPP
    For i = 1 To nData
        xData(i) = Worksheets("SortMyData").Range("B3").Cells(i,1).Value
    Next
End Sub
'--------------------------------------------------------------------------------
Sub SortMyData(ByRef myData() As Double, ByRef myDataPrime() As Double, _
               ByRef myDataDoublePrime() As Double)
   Dim nPoints As Integer, k As Integer, j As Integer, temp As Double
   nPoints = UBound(myData,1)           'find array size
   ReDim myDataPrime(nPoints), myDataDoublePrime(nPoints)
   For k = 1 To nPoints                 'copy data to new vectors
      myDataPrime(k) = myData(k)
      myDataDoublePrime(k) = myData(k)
   Next k
   For k = 1 To nPoints-1               'sort data
      For j = k+1 To nPoints
         'Sort in increasing order
         If myDataPrime(k) > myDataPrime(j) Then
            temp = myDataPrime(k)
            myDataPrime(k) = myDataPrime(j)
            myDataPrime(j) = temp
         End If
         'Sort in decreasing order
         If myDataDoublePrime(k) < myDataDoublePrime(j) Then
            temp = myDataDoublePrime(k)
            myDataDoublePrime(k) = myDataDoublePrime(j)
            myDataDoublePrime(j) = temp
         End If
      Next j
   Next k
End Sub
'-------------------------------------------------------------------------------
Sub WriteOutMyData(ByRef y() As Double, ByRef yP() As Double, _
                   ByRef yPP() As Double)
   Dim nP As Integer, k As Integer
   nP = UBound(y,1) 'find array size
   For k = 1 To nP  'write out arrays
      Worksheets("SortMyData").Range("D3").Cells(k,1).Value = y(k)
      Worksheets("SortMyData").Range("D3").Cells(k,2).Value = yP(k)
      Worksheets("SortMyData").Range("D3").Cells(k,3).Value = yPP(k)
   Next k
End Sub
'----------------------------------------------------------------------
```

```
Sub ClearOutput()
   Dim k As Integer, j As Integer
   For k = 1 To 1000
      For j = 1 To 3
         Worksheets("SortMyData").Range("D3").Cells(k,j).Value = ""
      Next j
    Next k
End Sub
'---------------------------------------------------------------------
Sub ClearOutputAndData()
   Dim k As Integer
   Call ClearOutput()
   For k = 1 To 1000
      Worksheets("SortMyData").Range("B3").Cells(k,1).Value = ""
    Next k
End Sub
```

The interface for this code is shown in Figure 5, below.



	A	B	C	D	E	F	G	H
1		Original		Unsorted	Sorted UP	Sorted DOWN		
2		x		x	x'	x''		
3		2.3		2.3	2	2.8		
4		2.2		2.2	2.1	2.7		
5		2.1		2.1	2.2	2.5		
6		2		2	2.3	2.3		
7		2.5		2.5	2.5	2.2		
8		2.7		2.7	2.7	2.1		
9		2.8		2.8	2.8	2		
10								

Figure 6. Interface for the sorting program coded above (Example 9).

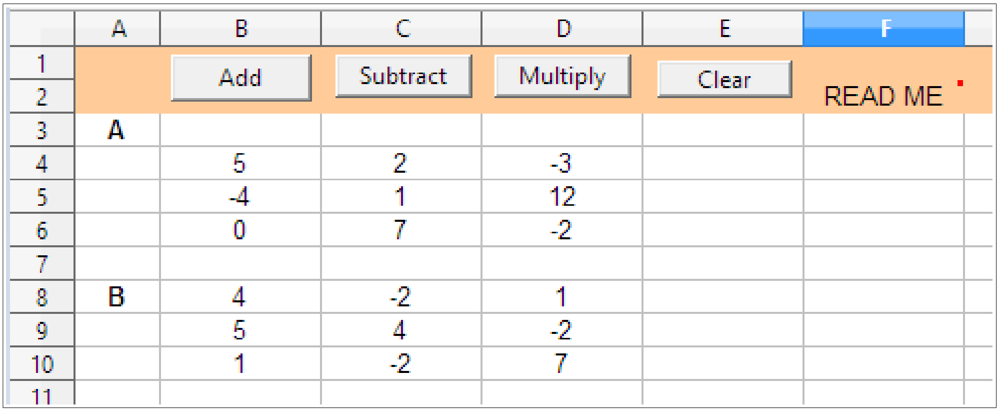
Example 10 – Rewrite the *AddTwoMatrices*( ) program of Chapter 4 in a modular fashion. The code for the original program is available in Figure 7 of Chapter 4 and reproduced next.

```
Option Explicit
Option Base 1
'=================================================================================
Sub AddTwoMatrices()
    Dim A() As Double, B() As Double, C() As Double
    Dim k As Integer, i As Integer, j As Integer
    Dim nA As Integer, mA As Integer
    Dim nB As Integer, mB As Integer
    Dim nC As Integer, mC As Integer
    ' Check size of matrix A(mA x nA)
    nA = 0 : mA = 0
    For k = 1 To 100
        If Worksheets("Matrix01").Range("B3").Cells(1,k).Value <> "" Then
           nA = nA + 1
        Else
           Exit For
        End If
    Next k
    For k = 1 To 100
        If Worksheets("Matrix01").Range("B3").Cells(k,1).Value <> "" Then
           mA = mA + 1
        Else
           Exit For
        End If
    Next k
    ' Check size of matrix B(mB x nB)
    nB = 0 : mB = 0
    For k = 1 To 100
        If Worksheets("Matrix01").Range("B3").Cells(mA+2,k).Value <> "" Then
           nB = nB + 1
        Else
           Exit For
        End If
    Next k
    For k = 1 To 100
        If Worksheets("Matrix01").Range("B3").Cells(mA+1+k,1).Value <> "" Then
           mB = mB + 1
        Else
           Exit For
        End If
    Next k
    If ((mA = mB) And (nA = nB)) Then 'Check that A and B have same size
       mC = mA : nC = nA

       'Redimension matrices
       Redim A(mA, nA)
       Redim B(mB, nB)
       Redim C(mC, nC)

       'Read data for matrix A
       For i = 1 to mA
           for j = 1 to nA
               A(i,j) = _
               Worksheets("Matrix01").Range("B3").Cells(i,j).Value
           Next
       Next
       'Read data for matrix B
       For i = 1 to mB
           for j = 1 to nB
               B(i,j) = _
               Worksheets("Matrix01").Range("B3").Cells(mA+1+i,j).Value
           Next
       Next
       'Add matrices A + B = C, show results
       Worksheets("Matrix01").Range("A3").Cells(mA+mB+3,1).Value = "C"
       For i = 1 to mC
           for j = 1 to nC
               C(i,j) = A(i,j) + B(i,j)
               Worksheets("Matrix01").Range("B3").Cells(mA+mB+2+i,j).Value = _
                    C(i,j)
           Next
       Next
       'Redo A and B labels -- not needed, but just in case
       Worksheets("Matrix01").Range("A3").Cells(1,1).Value = "A"
       Worksheets("Matrix01").Range("A3").Cells(nA+2,1).Value = "B"
    Else
       MsgBox("Matrices are of different size. " & Chr(13) & _
              "Addition cannot be performed.")
    End If
End Sub
```

The interface used is shown below. The *AddTwoMatrices*( ) macro is associated with the [Add] button.



	A	B	C	D	E	F
1		Add	Subtract	Multiply	Clear	
2						READ ME
3	A					
4		5	2	-3		
5		-4	1	12		
6		0	7	-2		
7						
8	B	4	-2	1		
9		5	4	-2		
10		1	-2	7		
11						

Figure 7. Interface for macro *AddTwoMatrices* (see code above).

The main program for performing the matrix addition can be designed to perform the following tasks:

- Call a *Sub* called, for example, *EnterMatrix* to enter the matrix A. This *Sub* should return the two-dimensional array containing the matrix A. By design, the upper left corner of this matrix will be in cell B4. Thus, this location can be passed on to the *Sub*. For example, if we pass the row and column values of the upper left corner of matrix A as *row* = 4, and *column* = 2, the *Cells*( ) function for the worksheet will point to that particular cell. The sub should return the location of the upper left corner of matrix B, which should be easy to figure out since it is in the same column as the upper left corner of matrix A, and at a row whose position can be obtained by adding the number of rows of A to the position of the upper left corner of A, plus 1.
- Call the *Sub EnterMatrix* again to enter matrix B. The location of the upper left corner of matrix B, obtained from the previous call to *Sub EnterMatrix*, is passed on to the *Sub*. This call to *Sub EnterMatrix* returns the position where the resulting matrix, C = A + B, will be written.
- Call a *Sub* called, for example, *AddMatrices* that would take in the arrays containing *matrices* A and B and calculate matrix C = A + B, which gets returned to the main program.
- Call a *Sub* called, for example, *WriteMatrix* that takes the position of the upper left corner of matrix C (from the last call to *Sub EnterMatrix*) and write matrix C to the interface.

The following code accomplishes the required tasks for the main program and subroutines. The main program also checks that the dimensions of matrices A and B are compatible.

```
Option Explicit
Option Base 1
'---------------------------------------------------------------------------
Sub MainProgramMatrixAddition()
   Dim rowULC As Integer, colULC As Integer
   Dim A() As Double, B() As Double, C() As Double
   Dim nA As Integer, mA As Integer, nB As Integer, mB As Integer
   rowULC = 4 : colULC = 2 'Upper-left corner of matrix A
    Call EnterMatrix(rowULC, colULC, A()) 'Input matrix A
    nA = UBound(A,1) : mA = UBound(A,2)
    Call EnterMatrix(rowULC, colULC, B()) 'Input matrix B
    nB = UBound(B,1) : mB = UBound(B,2)
    If nA <> nB Or mA <> mB Then
      MsgBox("Matrices A and B have different dimensions." & Chr(13) & _
             "Addition C = A + B cannot be calculated.")
    Else
      Call AddMatrices(A(), B(), C())       'Calculate C = A+B
      Call WriteMatrix(rowULC, colULC, C()) 'Write out matrix C
   End If
End Sub
'---------------------------------------------------------------------------
Sub EnterMatrix(ByRef ULC_row As Integer, _
                ByRef ULC_col As Integer, _
                ByRef MatrixData() As Double)
    Dim nRows As Integer, nCols As Integer
    Dim i As Integer, j As Integer
    'Search for end of matrix by rows
    nRows = 0
    Do While Worksheets("MatrixOperations").Cells(ULC_row+nRows, ULC_col).Value <> ""
       nRows = nRows + 1
    Loop
    'Search for end of matrix by columns
    nCols = 0
    Do While Worksheets("MatrixOperations").Cells(ULC_row,ULC_col+nCols).Value <> ""
       nCols = nCols + 1
    Loop
    ReDim MatrixData(nRows,nCols) 'Redimension matrix MatrixData
    For i = 1 To nRows            'Read values of MatrixData(i,j)
      For j = 1 To nCols
         MatrixData(i,j) = _
         Worksheets("MatrixOperations").Cells(ULC_row+i-1,ULC_col+j-1).Value
      Next j
    Next i
    ULC_row = ULC_row + nRows + 1
End Sub
'---------------------------------------------------------------------------
Sub AddMatrices(ByRef Matrix1() As Double, _
                ByRef Matrix2() As Double, _
                ByRef Matrix3() As Double)
    Dim nMatrix As Integer, mMatrix As Integer
    Dim i As Integer, j As Integer
    nMatrix = UBound(Matrix1,1)
    mMatrix = UBound(Matrix1,2)
    ReDim Matrix3(nMatrix, mMatrix)
    For i = 1 to nMatrix
      For j = 1 To mMatrix
         Matrix3(i,j) = Matrix1(i,j) + Matrix2(i,j)
      Next j
    Next i
End Sub
```

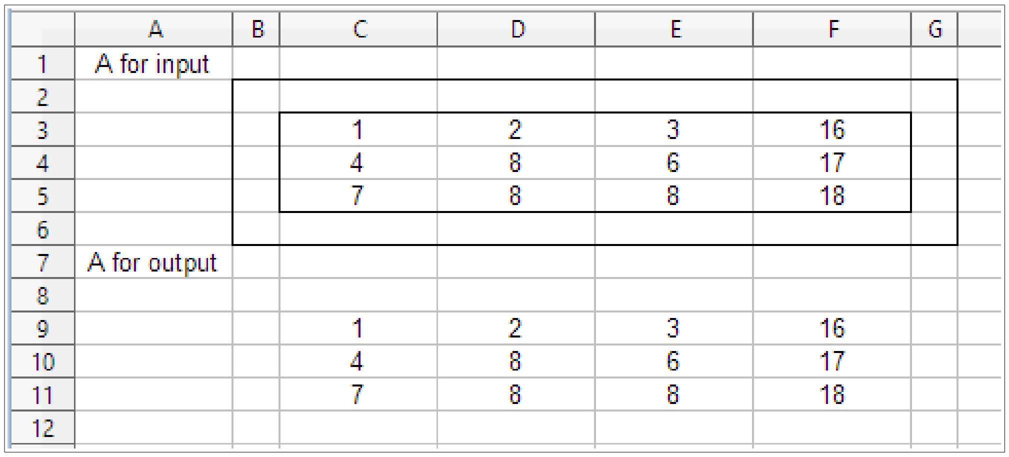
```
'----------------------------------------------------------------------------
Sub WriteMatrix(ByRef ULC_row As Integer, _
              ByRef ULC_col As Integer, _
              ByREf MatrixData() As Double)
    Dim nRows As Integer, nCols As Integer
    Dim i As Integer, j As Integer
    nRows = UBound(MatrixData, 1)
    nCols = UBound(MatrixDAta, 2)
    For i = 1 To nRows
       For j = 1 To nCols
          Worksheets("MatrixOperations").Cells(ULC_row+i-1,ULC_col+j-1).Value = _
          MatrixData(i,j)
       Next j
    Next i
End Sub
'-----------------------------------------------------------------------------
```

NOTE: The original worksheet with the interface of Figure 7 includes also buttons for subtraction and matrix multiplication. The original codes for those operations is available in Chapter 4. It is left as an exercise converting those original codes for subtraction and multiplication into modular codes similar to the one shown above.

Example 11 – Modular version of matrix input and output using *Range* variables – The following code was presented in Chapter 04. This code reads data from worksheet "`Sheet10`" in the form of a matrix whose upper left corner is located in cell "C3". The data thus entered is then converted internally into matrix A, and the matrix is outputted (unchanged) into the worksheet. The upper left corner of the outputted matrix is located in *Cells*(*m*+3+*i*,j) of the range defined by "C3". The input matrix needs to have clean rows and columns surrounding the data.

```
Sub InputOutputVariableSizeMatrix()
   Dim i As Integer, j As Integer, n As Integer, m As Integer
   Dim A() As Double
   Dim RangeA As Range
   Set RangeA = Worksheets("Sheet10").Range("C3")
   Set RangeA = RangeA.CurrentRegion.SpecialCells(xlCellTypeConstants, xlNumbers)
   m = RangeA.Rows.Count:  n = RangeA.Columns.Count
   MsgBox("No. of rows (m) = " & CStr(m) & _
          ", No. of columns (n) = " & CStr(n))
   ReDim A(m,n)
   For i = 1 To m
       For j = 1 To n
           A(i, j) = RangeA(i,j)
           RangeA.Cells(m+3+i,j).Value = A(i,j)
       Next j
   Next i
End Sub
```

The interface used to run this program is shown next:



	A	B	C	D	E	F	G
1	A for input						
2							
3			1	2	3	16	
4			4	8	6	17	
5			7	8	8	18	
6							
7	A for output						
8							
9			1	2	3	16	
10			4	8	6	17	
11			7	8	8	18	
12							

Figure 8. Interface for the original code of Example 11 (see above).

The code presented above includes three basic operations:

- Given the upper left corner of the matrix, which defines *RangeA*, the extent of the matrix and its contents are identified. The number of rows, *n*, and columns*, m*, are obtained.
- The two-dimensional array A is redimensioned, and loaded with values from *RangeA.* This is done with the *For* loop shown.
- The contents of matrix A are then outputted to the worksheet, an operation that is also performed within the *For* loop shown.

What we would like to do to convert this code into a modular code is to have a main program which will call three subroutines, each performing each of the tasks listed above. Let's call these subroutines, *InputMatrixWithRange*, *ConvertRangeToMatrix*, and *OutputMatrix*. Code for the main program and the associated subroutines is shown next.

```
Option Explicit
Option Base 1
'-------------------------------------------------------------------
Sub InputOutputMatrixMainProgram()
   'Read matrix A with its upper left corner in range "RangeA" and
   'output it to range "RangeOut" as its upper left corner
   Dim RangeA As Range, RangeOut As Range
   Dim nA As Integer, mA As Integer, A() As Double
    Set RangeA = Worksheets("MatrixRangeIO").Range("C3")
   Call InputMatrixWithRange(RangeA)
   Call ConvertRangeToMatrix(RangeA, A(), nA, mA)
   RangeOut = RangeA.Cells(nA+4,1)
   Call OutputMatrix(A(), RangeOut)
End Sub
'--------------------------------------------------------------------
Sub InputMatrixWithRange(ByRef myRange As Range)
   'The Range specification "myRange" must be passed as the location of
   'the upper left corner of the matrix A in the interface
   Set myRange = _
       myRange.CurrentRegion.SpecialCells(xlCellTypeConstants, xlNumbers)
End Sub
```

```
'-------------------------------------------------------------------------
Sub ConvertRangeToMatrix(ByRef TheRange As Range, ByRef TheMatrix() As Double, _
                         ByRef TheRows As Integer, TheCols As Integer)
    'The Range variable "TheRange" is sent from the main progam
    'The matrix "TheMatrix", its number of rows and columns are sent back
   Dim k As Integer, j As Integer
   TheRows = TheRange.Rows.Count : TheCols = TheRange.Columns.Count
   Redim TheMatrix(TheRows, TheCols)
   For k = 1 to TheRows
      For j = 1 To TheCols
         TheMatrix(k,j) = TheRange(k,j)
      Next j
   Next k
End Sub
'---------------------------------------------------------------------------
Sub OutputMatrix(ByRef myMatrix() As Double, ByRef myRange As Range)
   'The range "myRange" is sent from the main progam as the upper left
   'corner where matrix "myMatrix" is to be printed
   Dim i As Integer, j As Integer, nRows As Integer, nCols As Integer
   nRows = UBound(myMatrix,1) : nCols = UBound(myMatrix,2)
   For i = 1 To nRows
      For j = 1 To nCols
         myRange.Cells(i,j).Value = MyMatrix(i,j)
      Next j
   Next i
End Sub
```

A collection of subprograms for matrix operations
The code shown above, which is linked to the interface of Figure 8, performs the same operations from the original code listed earlier by linking the main program to the three *Sub* subprograms described in Example 11. The advantage of using this modular approach, over the original single-program *Sub* `InputOutputVariableSizeMatrix`, is that we can produce a variety of *Sub* subprograms (or *Function* subprograms) that can be reused in other programs. This is particular important for matrix and linear algebra solutions since you can maintain a good collection of subprograms (e.g., input, output, conversion of range to matrix, addition, subtraction, multiplication, inversion, transpose, determinant, etc.), some of which you can select to link to different main programs for a variety of applications.

Thus, in Example 11 we started to collect matrix-related subprograms that can be reused for a variety of applications. The first three subprograms are, therefore:

- `Sub InputMatrixWithRange`
- `Sub ConvertRangeToMatrix`
- `Sub OutputMatrix`

Next, code listings for subprograms that perform transposition, addition, subtraction, multiplication by a scalar, linear combination, and matrix multiplication are presented as part of the collection of matrix-related subprograms. To this list, we can also a add a subprogram to copy a matrix. Thus, we add to our collection the following subprograms:

- `Sub TransposeMatrix`
- `Sub AddMatrices`
- `Sub SubtractMatrices`
- `Sub MatrixTimesScalar`
- `Sub MatrixLinearCombination`
- `Sub MultiplyMatrices`
- `Sub CopyMatrix`

These programs have been developed assuming that the specification *Option Base* 1 is active.

```
Sub TransposeMatrix(ByRef matrixA() As Double, ByRef matrixB() As Double)
   'Produces the transpose of matrixA into matrixB
   'Dimensions of matrices: matrixA(nxm), matrixB(mxn)
   'Declare matrixB as a dynamic array
   Dim n As Integer, m As Integer, i As Integer, j As Integer
   n = UBound(matrixA, 1) : m = UBound(matrixA, 2)
   ReDim matrixB(m,n)
   For i = 1 To n
      For j = 1 To m
         matrixB(j,i) = matrixA(i,j)
      Next j
   Next i
End Sub

'-----------------------------------------------------------------------------------
Sub AddMatrices(ByRef matrixA() As Double, ByRef MatrixB() As Double, _
                ByRef matrixC() As Double)
    'Calculates sum: matrixC = matrixA + matrixB, all matrices must
    'have the same dimensions  -- declare matrixC as a dynamic array
    Dim n As Integer, m As Integer, i As Integer, j As Integer
    n = UBound(matrixA, 1) : m = UBound(matrixB, 2)
    ReDim matrixC(n,m)
    For i = 1 To n
      For j = 1 To m
         matrixC(i,j) = matrixA(i,j) + matrixB(i,j)
      Next j
    Next i
End Sub
'----------------------------------------------------------------------------------
Sub SubtractMatrices(ByRef matrixA As Double, ByRef MatrixB As Double, _
                     ByRef matrixC As Double)
    'Calculates difference: matrixC = matrixA - matrixB, all matrices must
    'have the same dimensions  -- declare matrixC as a dynamic array
    Dim n As Integer, m As Integer, i As Integer, j As Integer
    n = UBound(matrixA, 1) : m = UBound(matrixB, 2)
    Redim matrixC(n,m)
    For i = 1 To n
      For j = 1 To m
         matrixC(i,j) = matrixA(i,j) - matrixB(i,j)
      Next j
    Next i
End Sub
'-----------------------------------------------------------------------------------
```

```
Sub MatrixTimesScalar(ByRef matrixA() As Double, ByRef r As Double, _
                  ByRef matrixB() As Double)
    'Calculates multiplication of matrixA with scalar r
    'Declare matrixB as a dynamic array
    Dim n As Integer, m As Integer, i As Integer, j As Integer
    n = UBound(matrixA, 1) : m = UBound(matrixA, 2)
    ReDim matrixB(n,m)
    For i = 1 To n
      For j = 1 To m
         matrixB(i,j) = r * matrixA(i,j)
      Next j
    Next i
End Sub
'-----------------------------------------------------------------------------------
Sub MatrixLinearCombination(ByRef matrixA() As Double,ByRef MatrixB() As Double, _
                            ByRef r As Double, ByRef s As Double, _
                            ByRef matrixC() As Double)
    'Calculates the linear combination of matrices given by:
    '          matrixC = r * matrixA + s * matrixB
    'All matrices must have the same dimensions - declare matrixC as dynamic array
    Dim n As Integer, m As Integer, i As Integer, j As Integer
    n = UBound(matrixA, 1) : m = UBound(matrixB, 2)
    ReDim matrixC(n,m)
    For i = 1 To n
      For j = 1 To m
         matrixC(i,j) = r * matrixA(i,j) + s * matrixB(i,j)
      Next j
    Next i
End Sub


'-----------------------------------------------------------------------------------
Sub MultiplyMatrices(ByRef matrixA() As Double, ByRef MatrixB() As Double, _
                     ByRef matrixC() As Double)
    'Calculates product: matrixC = matrixA * matrixB
    'Matrix dimensions are: matrixA(nxp), matrixB(pxm), matrixC(nxm)
    'Declare matrixC as a dynamic array
    Dim n As Integer, m As Integer, p As Integer
    Dim i As Integer, j As Integer, k As Integer
    n = UBound(matrixA, 1) : p = UBound(matrixA, 2) : n = UBound(matrixB, 2)
    ReDim matrixC(n,m)
    For i = 1 To n
      For j = 1 To m
         matrixC(i,j) = 0.0
         For k = 1 to p
            matrixC(i,j) = matrixC(i,j) + matrixA(i,k) * matrixB(k,j)
         Next k
      Next j
    Next i
End Sub

'-----------------------------------------------------------------------------------
```

```
Sub CopyMatrix(ByRef matrixA() As Double, ByRef matrixB() As Double)
   'Copies matrixA into matrixB
   Dim n As Integer, m As Integer, i As Integer, j As Integer
   n = UBound(matrixA, 1) : m = UBound(matrixA, 2)
   ReDim matrixB(n, m)
   For i = 1 To n
      For j = 1 To m
         matrixB(i,j) = matrixA(i,j)
      Next j
   Next i
End Sub
```

Notice that checking matrix compatibility was not included in these codes. That process would have to be performed in the main program. Once the matrices (and/or scalars, if needed) are passed onto the subprograms, the compatibility is assumed to exist. The *Sub*s, however, could be modified to perform the check for compatibility inside the *Sub*s.

The subprograms developed above for transposition, matrix addition, subtraction, multiplication by a scalar, linear combination, and matrix multiplication did not require use to use any *Range* variables. Other operations such as calculating the determinant or the inverse of a matrix, operations that can be performed using Excel array operations, will need the use of *Range* variables. Subprograms for calculating determinants and inverse matrices are shown below. With these two subprograms, namely,

- `Sub MatrixDeterminant`
- `Sub MatrixInverse`

we complete a basic collection of matrix-related subprograms to be incorporated in more complex programs for matrix operations and linear algebra.

NOTES:

1 – Matrix transposition, addition, subtraction, multiplication by a scalar, linear combination, and matrix multiplication can also be performed using array operations in Excel. However, their coding is relatively simple (as demonstrated above), and we do not need to use array operations for those subprograms.

2 – Matrix inversion and calculating determinants can be performed without using array operations in Excel. Such operations can be performed using specific algorithms based on Gaussian and Gauss-Jordan elimination. However, the algorithms for such methods will be left for a course on numerical methods. For lack of better resources, therefore, we utilize the array operation approach to calculate determinants and inverse matrices.

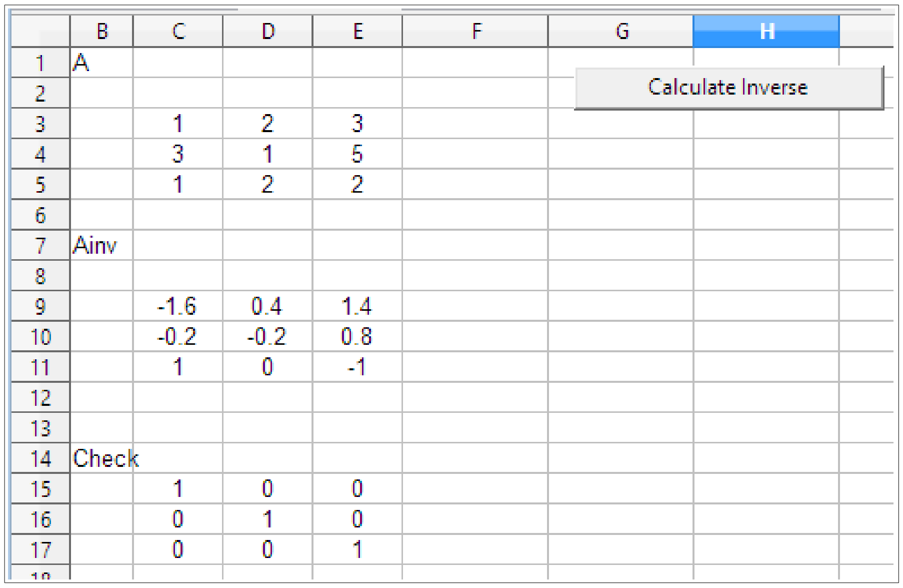
Example 12 – Inverse of a matrix using modular programming – The code for determinant and inverse matrix calculations using *Range* variables is presented next. To test the code, a main program called *MainProgramInverseMatrix*.

```
Option Explicit
Option Base 1
'-------------------------------------------------------------------
Sub MainProgramInverseMatrix()
   Dim RangeA As Range, RangeAI As Range, RangeOut As Range
   Dim detA As Double, epsilon As Double
   Dim nA As Integer, mA As Integer, nAI As Integer, mAI As Integer
   Dim A() As Double, AI() As Double
   epsilon = 0.00001
   Set RangeA = Worksheets("MatrixInverse").Range("C3")
   Call InputMatrixWithRange(RangeA)
   Call ConvertRangeToMatrix(RangeA, A(), nA, mA)
   Call MatrixDeterminant(A, detA)
   MsgBox("Determinant of A = " & CStr(detA))
   If Abs(detA) >= epsilon Then
      Call MatrixInverse(A(), AI())
      Set RangeOut = RangeA.Cells(nA+4,1)
      Call OutputMatrix(AI(), RangeOut)
   Else
      MsgBox("det(A) = 0, no inverse possible")
   End If
End Sub

'------------------------------------------------------------------

Sub MatrixDeterminant(ByRef matrixA() As Double, ByRef detA As Double)
   'Calculates the determinant of matrixA, detA = det(matrixA)
   'Matrix A must be square
    Dim n As Integer, m As Integer, RangeA As Variant
    n = UBound(matrixA, 1) : m = UBound(matrixA, 2)
    RangeA = matrixA
    detA = Application.MDETERM(RangeA)
End Sub
'-------------------------------------------------------------------------------
Sub MatrixInverse(ByRef matrixA() As Double, ByRef matrixAI() As Double)
   'Calculates the inverse of matrixA, i.e., matrixAI = matrixA^(-1)
   'Matrix A must be square
   Dim n As Integer, m As Integer, i As Integer, j As Integer
   Dim RangeA As Variant
   n = UBound(matrixA, 1) : m = UBound(matrixA, 2)
   ReDim matrixAI(n,m)
    RangeA = Application.MINVERSE(MatrixA)
    For i = 1 To n
      For j = 1 To m
         matrixAI(i,j) = RangeA(i,j)
      Next j
    Next i
End Sub
'-------------------------------------------------------------------------------
```

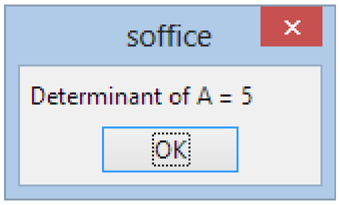
To test the inverse and determinant code, we run the program with the data shown in the following Figure. The product of the matrix *A* and its inverse *Ainv* is shown in the interface to verify that it is indeed equal to the identity matrix.



	B	C	D	E	F	G	H
1	A						
2						Calculate Inverse	
3		1	2	3			
4		3	1	5			
5		1	2	2			
6							
7	Ainv						
8							
9		-1.6	0.4	1.4			
10		-0.2	-0.2	0.8			
11		1	0	-1			
12							
13							
14	Check						
15		1	0	0			
16		0	1	0			
17		0	0	1			

Figure 9. Interface for calculating the inverse of a matrix, based on the code shown above.

The value of the determinant is given in a message box, as shown below.



soffice

Determinant of A = 5

Summary of subprograms for matrix operations

The following list summarizes the 12 subprograms defined above to deal with matrix operations:

`Call InputMatrixWithRange(RangeA)`

- **Input**: RangeA – a specific upper left corner of a matrix in an interface, e.g., RangeA = Worksheets("Sheet1").Range("C3")
- **Output**: RangeA – the entire matrix read from the interface passed as a range variable

`Call ConvertRangeToMatrix(RangeA, A, n, m)` – used typically immediately after *InputMatrixWithRange*

- **Input**: RangeA – a range to which a matrix from the interface has been loaded
- **Output**: A – matrix A extracted from RangeA, n – number of rows of A, m – number of columns of A

`Call OutputMatrix(A(), RangeA)`

- **Input**: A - a matrix, RangeA – position of the upper left corner of the matrix for output in an interface, e.g., RangeA = Worksheets("Sheet1").Range("H3")
- **Output**: none – subprogram simply writes out matrix to the interface

`Call TransposeMatrix(A, AT)`

- **Input**: A – a matrix
- **Output**: AT – the transposed of A

`Call AddMatrices(A(),B(),C())`

- **Input**: A and B – two matrices of the same dimensions
- **Output**: C – a matrix of the same dimensions as A and B, so that C = A + B

`Call SubtractMatrices(A(),B(),C())`

- **Input**: A and B – two matrices of the same dimensions
- **Output**: C – a matrix of the same dimensions as A and B, so that C = A - B

`Call MatrixTimesScalar(A(), r, B())`

- **Input**: A – a matrix, r – a scalar number
  Output: B – a matrix of the same dimensions as A, so that B = r A

`Call MatrixLinearCombination(A(),B(),r,s,C())`

- **Input**: A,B – matrices of the same dimensions, r, s – scalar values
- **Output**: C – a matrix, such that C = r A + s B

`Call MultiplyMatrices(A(),B(),C())`

- **Input**: A,B – matrices of dimensions A(nxp) and B(pxm)
- **Output**: C – a matrix of dimensions C(nxm), such that C = A B

`Call CopyMatrix(A(), B())`

- **Input**: A – a matrix
- **Output**: B – a matrix, an exact copy of A

`Call MatrixDeterminant(A(), detA)`

- **Input**: A – a square matrix
- **Output**: detA – a scalar value representing the determinant of matrix A, i.e., det(A) = |A|

`Call MatrixInverse(A(), Ainv())`

- **Input**: A – a square, non-singular matrix (i.e., det(A) should not be zero)
- **Output**: Ainv – the inverse of matrix A

The way the twelve subprograms were defined earlier, the variables A, Ainv, B, and C are *Double*-precision, floating-point, dynamic arrays; r, s, and detA are all *Double* precision, floating point non-array variables, while *RangeA* is a *range* variable. The matrices used as input for most of the subprograms get redimensioned within the subprograms, therefore the requirement that in the main program the corresponding array variables be defined as dynamic arrays. See the main programs in the examples associated with Figures 8 and 9, above, for illustration of this requirement.

You can place all of these subprograms into one (or more) modules in your workbook, and then link them through the proper *Call* statements to a main program that you would design to carry out specific matrix operations. Keep in mind that the main program will need to check that the matrices involved in the operations are of the same dimensions (for addition, subtraction, or linear combination), of compatible dimensions (for multiplication), a square matrix (for calculating the determinant), or a non-singular square matrix (for calculating the inverse). No check is required for calculating matrix transposes or copying a matrix, other than the fact that the corresponding arrays must be declared in the main programs.

The subprograms listed above were developed assuming that *Option Base 1* for array subindices is active. If you ever need to use *Option Base 0* for array subindices, you will need to change the starting index in the *For* loops, and make other changes in the use of integer index to conform to that requirement.

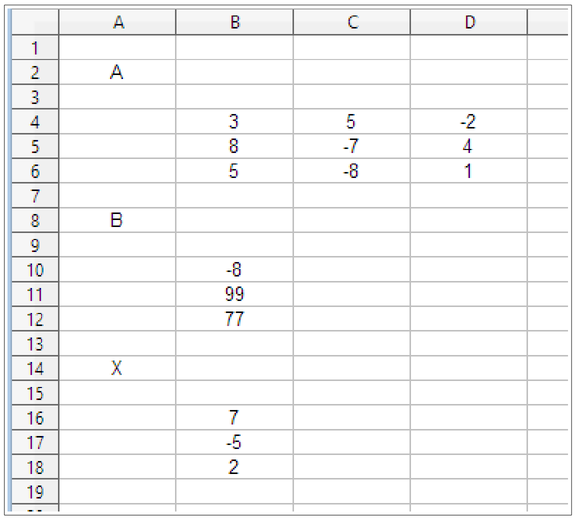
NOTE: A *Function* subprogram for calculating the determinant of a matrix.
We used the *Sub MatrixDeterminant* subprogram, listed above, to calculate the determinant of a matrix. Since the value of the determinant calculated therein is a single scalar value we could have used a *Function* subprogram to calculate such value. If we select to go that way, a possible code for the *Function* subprogram would be as listed below.

```
Function MatrixDeterminant(ByRef matrixA() As Double) As Double
   'Calculates the determinant of matrixA, det(matrixA)
   'Matrix A must be square
    Dim n As Integer, m As Integer, RangeA As Range
    n = UBound(matrixA, 1) : m = UBound(matrixA, 2)
    Set RangeA = matrixA
    detA = Application.MDETERM(RangeA)
End Sub
```

Using this *Function* subprogram for calculating the determinant in a main program would entail an assignment statement as the following: `detA = MatrixDeterminant(A())`

For all other matrix operation subprograms, the use of *Sub* subprogram would be required since they return matrices back to the calling main program.

Example 13 – Solution of a system of linear equation using modular programming. In this example we take advantage of the collection of matrix operation subprograms, listed above, to solve a matrix equation. The matrix of coefficients A and the right-hand side vector B, as well as the solution (calculated by the program) are shown in the interface below.



	A	B	C	D
1				
2	A			
3				
4		3	5	-2
5		8	-7	4
6		5	-8	1
7				
8	B			
9				
10		-8		
11		99		
12		77		
13				
14	X			
15				
16		7		
17		-5		
18		2		
19				

Figure 10. Interface for solving a matrix equation AX = B.

The VBA code for the main program to solve the matrix equation AX = B is shown next. Notice the use of comments to document the program's operation, and the use of proper indentation to facilitate reading the code. A number of nested *If ... Then ... Else* statements are used to check the following conditions:

- Matrix A must be square (nA = mA)
- Matrix A and Matrix B must have the same number of rows (nA = nB)
- The determinant of A, detA, must be nonzero, so that matrix A is nonsingular.

If any of these conditions is not satisfied, the program reports the problem to the user through the use of Moneyboxes and terminates itself before continuing with the calculations.

```
Option Explicit
Option Base 1

'-------------------------------------------------------------------
'
Sub MatrixEquationSolutionMainProgram()
'
'-------------------------------------------------------------------
' Programmer: Gilberto E. Urroz - Date: 10/17/2013
'-------------------------------------------------------------------
' This program solves the matrix equation A*X = B, with matrices
' A and B read from the interface Worksheets("MatEqSol").  The
' solution uses some of the matrix operation subprograms developed
' in Chapter 05 of the class notes.
'-------------------------------------------------------------------
' Variables used:
'  A() = matrix of coefficients, of size nA x mA
'  B() = right-hand side matrix, of size nB x mB (typically mB = 1)
'  X() = solution matrix, of size nA x mB
'  Ainv() = inverse of matrix A
'  epsilon = tolerance for the determinant of matrix A
'  detA = determinant of matrix A
'  RangeA, RangeB, RangeX = upper left corner of matrices A, B, and
'                           X, respectively, in the interface
'---------------------------------------------------------------------
' DECLARATION OF VARIABLES INVOLVED IN THE SOLUTION:
   Dim A() As Double, RangeA As Range
   Dim Ainv() As Double, X() As Double, RangeX As Range
   Dim B() As Double, RangeB As Range
   Dim nA As Integer, mA As Integer
   Dim nB As Integer, mB As Integer
   Dim detA As Double, epsilon As Double
   epsilon = 0.000001
'INPUT MATRIX A
   Set RangeA = Worksheets("MatEqSol").Range("B4")
   Call InputMatrixWithRange(RangeA)
   Call ConvertRangeToMatrix(RangeA, A(), nA, mA)

'CHECK THAT MATRIX A IS SQUARE
```

```
   If nA = mA Then  'Matrix is square
   'INPUT MATRIX (VECTOR) B
      Set RangeB = RangeA.Cells(nA+4,1)
      Call InputMatrixWithRange(RangeB)
      Call ConvertRangeToMatrix(RangeB, B(), nb, mb
   'CHECK THAT MATRIX B HAS THE SAME NUMBER OF ROWS AS MATRIX A
      If nA = nB Then  'A and B have the same number of rows
      'CALCULATE DETERMINANT OF A TO MAKE SURE IT'S NOT A SINGULAR MATRIX
           Call MatrixDeterminant(A(), detA)
           If Abs(detA) > 0 Then                        'Matrix A is not singular
             Call MatrixInverse(A(), Ainv())           'Calculate inverse of A
             Call MultiplyMatrices(Ainv(), B(), X())   'Calculate X = Ainv*B
           'OUTPUT SOLUTION (MATRIX X)
             RangeX = RangeB.Cells(nA+4,1)
             Call OutputMatrix(X(), RangeX)
           Else
             MsgBox("Matrix A is singular - no solution exists") 'IF det = 0
           End If
      Else
           MsgBox("Matrices A and B do not have the same number of rows")
      End If
   Else
      MsgBox("Matrix A is not square - program terminated")
   End If
End Sub
```

A list of additional subprograms for linear algebra
The collection of subprograms listed above constitute a basic set of code that can be used for matrix operations. However, more extensive applications in linear algebra would require additional subprograms to perform operations such as:

- Calculating the trace of a matrix
- Calculating the different norms of a matrix
- Calculating the rank of a matrix
- Creating a diagonal matrix out of a one-dimensional vector
- Generating an identity matrix of a given dimension
- Extracting rows and columns from a matrix
- Extracting a submatrix out of a matrix
- Augmenting a matrix by adding rows or columns
- Obtaining the characteristic equation out of a square matrix (for calculating eigenvalues)
- Finding the eigenvalues and eigenvectors of a square matrix
- Finding the eigenvalues and eigenvectors of the generalized eigenvector problem involving two square matrices of the same dimensions
- Performing the forward elimination process in a Gaussian elimination
- Introducing partial and full pivoting in a Gaussian elimination
- Solving a matrix equation by backward substitution in a Gaussian elimination
- Performing a Gauss-Jordan elimination to produce a row-reduced echelon form of a square matrix

- Solving a matrix equation using Gauss-Jordan elimination
- Calculating the inverse of a matrix by applying Gauss-Jordan elimination to an augmented matrix
- Calculating the determinant of a matrix by using Gaussian or Gauss-Jordan elimination
- Calculating a quadratic form from a matrix
- Finding the singular values of a matrix
- Solving matrix equations for a tri-diagonal matrix
- Matrix operations for sparse matrices (i.e., matrices with a large number of zero elements)

Most of these matrix and linear algebra operations belong in an advanced course in numerical methods, therefore, we will not address the programming of these operations in this book. For extensive linear algebra programming, however, it would be convenient to build a library of these type of subprograms to keep handy in developing specific solutions.

Scope of a variable

The term *scope of a variable* refers to the extent of the code operation where a variable is active. In a *Sub* that constitutes a main program, without subprograms, a variable is activated by a *Dim* declaration and remains active as long as the program operates. Once an *End Sub* statement is reached, all active variables are de-activated, and, therefore, the *End Sub* statement signifies the extend of the scope of those variables.

In programs with a main program and attached subprograms, the scope of a variable is the extent of the main program, or a particular subprogram, where the variable is active. If you use the debugger and add a watch for a particular variable name in a subprogram, you will see the variable acquire a value when the control is passed on to the subprogram. When the control is sent out of the subprogram, the *Watch* window will report the status of a variable contained within the subprogram, as *Out of Scope*.

In general, therefore, the scope of a variable is the subprogram where the variable is defined. If a variable is defined using a *Dim* statement within a subprogram, that variable is said to be a *local variable* to the subprogram. If a variable is defined as a parameter of a subprogram, the scope of the parameter is the subprogram, but the associated variable name used in the calling main program or subprogram is referred to as a *global variable*.

Consider the following code, which was used earlier in Example 5.

```
Sub MeanValueExample02
   'Main program using Function MeanOfXV(...) -- using Array
   Dim yV As Variant, yVBar As Double
   yV = Array(2.5, 2.3, 3.2, 1.8, 4.7, 5.2, 4.1)
   yVBar = MeanOfXV(yV())
   MsgBox("yBar = " & CStr(yVBar))
End Sub
'-------------------------------------------------------------------
Function MeanOfXV(ByRef XVData As Variant) As Double
    'Function for calculating the mean of vector XData()
    'NOTE: Option Base 1 is used for defining arrays
   Dim SumOfXV As Double, k As Integer, n As Integer
    n = UBound(XVData,1) : SumOfXV = 0.0
   For k = 1 To n
      SumOfXV = SumOfXV  + XVData(k)
   Next k
   MeanOfXV = SumOfXV/n
End Function
```

The scope of variables *yV* and *yVBar* is the main program *MeanValueExample02*. The scope of variables *XVData*, *SumOfXV*, *k*, and *n* is the *Function* subprogram *MeanOfXV*. Variable *yV*, passed on into the *Function MeanOfXV* as parameter *XVData*, is a *global variable* to that function. On the other hand, variables *SumOfXV, k,* and *n* are *local variables* to *Function MeanOfXV.*

Levels of variable's scopes in the Excel application
The following list represents the technical terms of scopes within the Excel application:

- *Procedure-level scope*: variables within subprograms
- *Module-level scope*: variables used through a module
- *Project-level, workbook-level, or Public Module-level scope*: variables active through the various worksheets of a workbook – Use *Public*, rather than *Dim*, in their definitions.

The term *procedure* in this context means any subprogram. Thus, the variables *yV, yVBar,* etc., referred above, are variables whose scope is at the *procedure-level*.

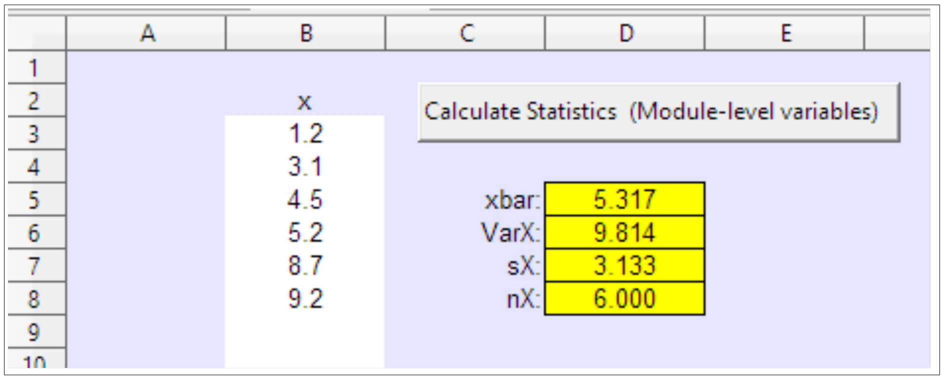
Example 14 - Module-level variables – If a variable is defined, through the use of a *Dim* statement, at the top of a module, then the scope of that variable is at the *module-level*. Consider, for example, the following code, modified from the code of Example 8, above. In this code, variables *y*( ), *nY, yBar, VarY*, and *sY* are declared as module-level variables and used throughout the various *Sub* and *Function* subprograms in the code. Notice that, by using these variables in each *Sub* and *Function*, there is no need to pass arguments to those subprograms. The only procedure-level variables used in these code are auxiliary variables such as *k*, *SumXX*, and *SumX2* used in the various *Sub* and *Function* subprograms listed below.

```
Option Explicit
Option Base 1
'------------------------------------------------------------
'Module-level variables
Dim y() As Double, nY As Integer
Dim yBar As Double, VarY As Double, sY As Double
'------------------------------------------------------------
Sub MeanValueExample06()   ' Main program
    'Main program calling three different Subs
    Call ReadData()
    Call yBarVarStDev()
    Call ShowMyResults()
End Sub
'-------------------------------------------------------------
```

```
Sub ReadData()
'This Sub subprogram inputs the data y from the interface "Sheet1"
    Dim k As Integer
    nY = 0
    For k = 1 to 1000
      If Worksheets("Sheet1").Range("B3").Cells(k,1).Value <> "" Then
         nY = nY + 1
      Else
         Exit For
      End If
    Next k
    Redim y(nY)
    For k = 1 To nY
      y(k) = Worksheets("Sheet1").Range("B3").Cells(k,1).Value
    Next k
End Sub
'------------------------------------------------------------------
Sub yBarVarStDev()
    'Sub subprogram to calculate mean (yBar), variance (VarY),
    'and standard deviation (sY) of a vector of data y
    yBar = XMean()
    VarY = VarData()
    sY = Sqr(VarY)
End Sub

'--------------------------------------------------------------------
Function XMean() As Double
    'Function XMean calculates the mean of vector y
   Dim k As Integer, SumXX As Double
   SumXX = 0.0
   For k = 1 To nY
      SumXX = SumXX + y(k)
   Next k
   XMean = SumXX/nY
End Function
'---------------------------------------------------------------------
Function VarData() As Double
    'Function VarY calculate the variance of vector DataX
   Dim SumX2 As Double, k As Integer
   SumX2 = 0.0
   For k = 1 To nY
      SumX2 = SumX2 + (y(k)-yBar)^2
   Next k
   VarData = SumX2/(nY-1)
End Function
'---------------------------------------------------------------------
Sub ShowMyResults()
'This Sub outputs the values of mean, variance, std. deviation, and sample size
    Worksheets("Sheet1").Range("D5").Value = yBar
    Worksheets("Sheet1").Range("D6").Value = VarY
    Worksheets("Sheet1").Range("D7").Value = sY
    Worksheets("Sheet1").Range("D8").Value = nY
End Sub
```

The interface used to run this code is shown below. It is not much different as that used in Figure4(d) for Example 8.



	A	B	C	D	E
1					
2		x			
3		1.2			
4		3.1			
5		4.5	xbar:	5.317	
6		5.2	VarX:	9.814	
7		8.7	sX:	3.133	
8		9.2	nX:	6.000	
9					
10					

Figure 11. Interface for the code for Example 12, which uses module-level variables.

Chapter summary
In this chapter we introduced the concepts of *Function* and *Sub* subprograms with the aim of organizing complex programs into smaller units. This programming approach is referred to as *modular programming*. Examples were provided to illustrate the use of subprograms, arguments, and parameters. The idea of the scope of a variable was also introduced.