VBA Programming in Excel: Loops – 1-D Arrays - Strings

By Gilberto E. Urroz, July 2013

In chapter 2 we introduced the different types of programming structures: sequence, branching, and loops, and focused particularly in the use of branching structures in VBA. In this chapter, we'll emphasize the use of looping or repetition structures, and applications to one-dimensional arrays.

Looping structures using *If … End If* statements

Consider the evaluation of the summation: $S_n = \sum_{k=1}^{n} \frac{1}{k^2}$. We indicated in Chapter 2 that a summation like this can be calculated by first initializing the summation variable, say, *Sn*, as zero, and then start a loop to add individual terms $1/k^2$ for $k = 1, 2, \ldots, n$. In pseudo-code, this calculation can be summarized as follows, using branching structures (*If ... Else ... End If*, and *GoTo*) :

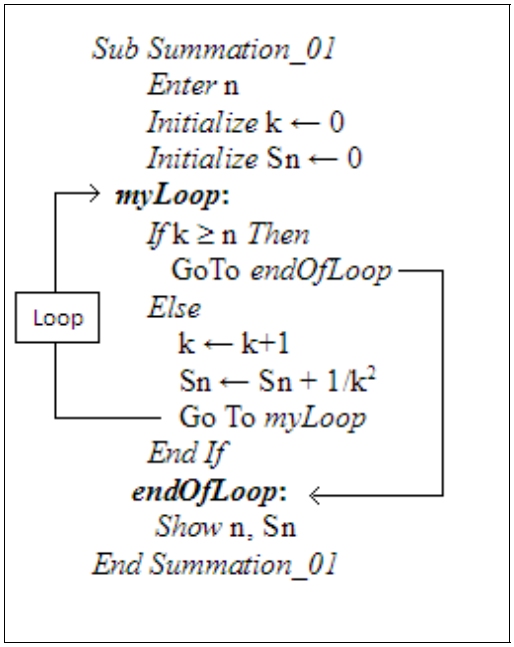


Figure 1. Pseudo-code for calculating the summation.

In this pseudo-code, the loop that calculates the summation starts at the label *myLoop*:. The statement "*If* k ≥ n *Then*" determines whether the loop continues through the *Else* part of the *If … Else ... End If* statement, or if the loop ends by sending to control to label *endOfLoop* (GoTo *endOfLoop*). The *Else* part of the *If ... Else ... End If* statement performs the index increment (k ← k+1) and summation increment (Sn ← Sn + 1/k²), and then sends the program control back to the label *myLoop* to repeat the loop statements. In this pseudo-code the index *k*, thus, takes the values $k = 1, 2, \ldots, n$. As soon as $k = n$ is verified in the statement "*If* k ≥ n *Then*", the control is sent out of the loop through the statement "GoTo *endOfLoop"*. Before that point, however, variable *Sn* accumulates the terms

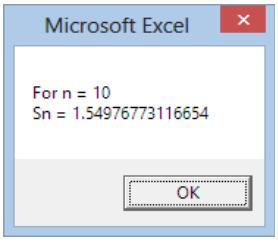
$\frac{1}{1^2}+\frac{1}{2^2}+\ldots+\frac{1}{n^2}=\sum_{k=1}^{n}\frac{1}{k^2}$, which is the required summation.

The pseudo-code shown in Figure 1 can be translated into VBA code as shown in Figure 2.

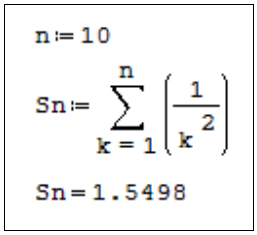

```
Option Explicit

Sub Summation_01()
    Dim n As Integer, k As Integer, Sn As Double
    n = InputBox("Enter n:")
    k = 0: Sn = 0
myLoop:
    If k >= n Then
       GoTo endOfLoop
    Else
       k = k + 1
       Sn = Sn + 1 / k ^ 2
       GoTo myLoop
    End If
endOfLoop:
    MsgBox ("For n = " & CStr(n) & Chr(13) & _
           "Sn = " & CStr(Sn))
End Sub
```

Figure 2. VBA code for the pseudo-code of Figure 1 calculating the summation.

The program of Figure 2 can be run from the *VBA IDE* by pressing the *Run* button. The user is prompted to enter the value of *n* through the statement `n = InputBox("Enter n:")`. Suppose that you enter the value $n = 10$ for this prompt. Then, the program calculates the summation and produces the result:



To verify this result I calculated the summation using the free math program *SMath Studio* which allows the calculation of summations and other calculus operations. The result provided by *SMath Studio* is:



The result provided by *SMath Studio*, 1.5498, is basically the same as calculated with the VBA code.
NOTE: For additional information on the SMath *Studio* software visit:

http://www.neng.usu.edu/cee/faculty/gurro/SMathStudio.html

The program of Figure 2 (based on the pseudo-code of Figure 1) represents a way to code a calculation loop using the branching statements *If ... Else ... Else If* and *GoTo*, as well as labels (*myLoop*: and

outOfLoop:). VBA provides other statements, specifically designed to handle loops, that do not require the use of labels nor the use of the dreaded *GoTo* statement. These looping, or repetition, statements are introduced in a subsequent section.

VBA Looping Statements

VBA offers a number of looping, or repetition, statements that can be classified as follows:

- *While* loops: repeats statements as long as the condition in the *While* statement is true
- *Do* loops: tests a condition at the beginning or end of the loop, and repeats statements either until the condition becomes true or while the condition remains true.
- *For* loops: use for a specific number of repetitions through the use of a counter with beginning, ending, and increment values.

Next, we describe the operation and show examples of the different types of VBA looping statements.

***While* loops**

While loops in VBA use the *While ... Wend* statement, whose general form is:

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

The <condition> is typically a logical statement involving an index, which gets modified within the loop <statements>. The loop <statements> are repeated as long as the <condition> remains true. Once the index is modified in the <statements> in such a way that the <condition> becomes false, the control is sent out of the loop.

As an example, consider the calculation of the summation used in an earlier example, namely, $S_n = \sum_{k=1}^{n} \frac{1}{k^2}$, using a *While ... Wend* statement. A pseudo-code for this program is shown next:

```
Sub Summation_02
      Enter n
      k ← 0
      Sn ← 0
      While k ≤ n
            k ← k+1
            Sn ← Sn + 1/k^2
       Wend
       Show result Sn
End Sub
```

Figure 3. Pseudo-code for calculating the summation $S_n = \sum_{k=1}^{n} \frac{1}{k^2}$ using the *While ... Wend* statement.

This pseudo-code is translated into the following VBA code:

```
Sub Summation_02()
    Dim k As Integer, n As Integer, Sn As Double
    k = 0: Sn = 0
    n = InputBox("Enter n:")
    While k < n
       k = k + 1
       Sn = Sn + 1 / k ^ 2
    Wend
    MsgBox ("For n = " & CStr(n) & Chr(13) & _
           "Sn = " & CStr(Sn))
End Sub
```

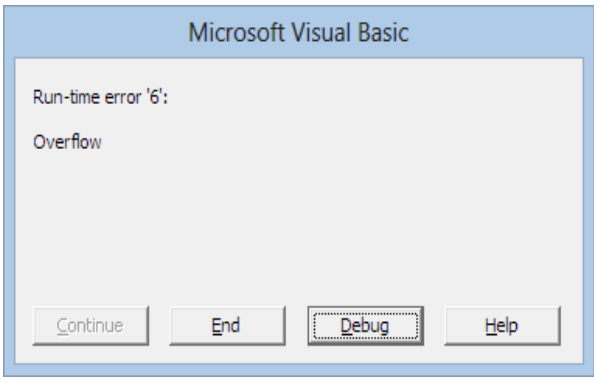
Figure 4. VBA code for calculating the summation $S_n = \sum_{k=1}^{n} \frac{1}{k^2}$ using the *While ... Wend* statement. This code is based on the pseudo-code of Figure 3.

Run this program for $n = 10$ to verify that you get the same result shown in page 2.

Infinite loops

An infinite loop is one which repeats with no possibility of ending it. In the case of a *While ... Wend* statement, an infinite loop will occur if the <condition> in the statement is always true. For example, if you were to, mistakenly, write $k = k - 1$ instead of $k = k + 1$ in the code of Figure 4, the condition $k < n$, for $n > 0$, will always be true and the loop will, theoretically, go on forever.

Try this exercise: change the statement $k = k + 1$ to $k = k - 1$ in the code of Figure 4 and run it for $n = 10$, for example. In this case, your program will get in an infinite loop since we start with $k = 0$, and k then take values of -1, -2, -3, …, etc., so that k is always less than n. In this example, the condition $k < n$, which needs to become false to get the program control out of the loop, is always true, and the loop should go on forever. In practice, however, the absolute value of k (a negative number) ends up growing so large that an *overflow* condition is quickly reached and the program is aborted. This is the message you get in that case:



This is a relatively small program and finding out the logical error in it (i.e., having $k = k - 1$ instead of $k = k + 1$) should be easy. In more complex codes you may have to use debugging (see chapter 2) to follow the behavior of the index variable k to realize that the statement $k = k - 1$ needs to be replaced by $k = k + 1$.

***Do* loops**

The *Do* loops in VBA provide a more flexible way to program loops since the condition to terminate the loop can be located at the beginning or at the end of the loop. Furthermore, the loop termination condition can be associated with a *While* or with an *Until* statement. Thus, there are four different forms of the *Do* loop, namely:

`Do While <condition>`
`  <statements>`
`Loop`
	`Do Until <condition>`
	`  <statements>`
	`Loop`
		`Do`
		`  <statements>`
		`Loop While <condition>`
			`Do`
			`  <statements>`
			`Loop Until <condition>`

(1) Do While … Loop	(2) Do Until … Loop	(3) Do … Loop While	(4) Do … Loop Until

In the statements (1) *Do While ... Loop* and (2) *Do Until ... Loop* the <condition> is checked at the beginning (or top) of the loop, whereas in the statements (3) *Do ... Loop While* and (4) *Do ... Loop Until* the <condition> is checked at the end (or bottom) of the loop.

- In the (1) *Do While ... Loop* statement, the <statements> are repeated as long as the <condition> at the top of the loop remains true. As soon as the <condition> becomes false, control is passed out of the loop.

- In the (2) *Do Until ... Loop* statement, the <statements> are repeated until the <condition> at the top of the loop becomes true, at which point the control is passed out of the loop.

- In the (3) *Do ... Loop While* statement, the <statements> are repeated as long as the <condition> at the bottom of the loop remains true. As soon as the <condition> becomes false, control is passed out of the loop.

- In the (4) *Do ... Loop Until* statement, the <statements> are repeated until the <condition> at the bottom of the loop becomes true, at which point the control is passed out of the loop.

In all four types of the *Do* loop in VBA the condition is typically a logical statement involving an index variable. The index variable needs to be modified in the *Do* loop <statements> so that eventually the <condition> becomes true or false and allow the control to be passed out of the loop.

To show case the different types of *Do* loops, the calculation of the summation $Sn = \sum_{k=1}^{n} \frac{1}{k^2}$ is programmed using each of those cases in the following figure:

```vba
Sub SummationDo_01
    Dim k As Integer, n As Integer, Sn As Double
    n = InputBox("Enter n:")
    k = 0 : Sn = 0.0
    Do While k < n
       k = k + 1
       Sn = Sn + 1/k^2
    Loop
    MsgBox("For n = " & CStr(n) & Chr(13) & _
           "Sn = " & CStr(Sn))
End Sub
```
(1)

```vba
Sub SummationDo_02
    Dim k As Integer, n As Integer, Sn As Double
    n = InputBox("Enter n:")
    k = 0 : Sn = 0.0
    Do Until k >= n
       k = k + 1
       Sn = Sn + 1/k^2
    Loop
    MsgBox("For n = " & CStr(n) & Chr(13) & _
           "Sn = " & CStr(Sn))
End Sub
```
(2)

```vba
Sub SummationDo_03
    Dim k As Integer, n As Integer, Sn As Double
    n = InputBox("Enter n:")
    k = 0 : Sn = 0.0
    Do
       k = k + 1
       Sn = Sn + 1/k^2
    Loop While k < n
    MsgBox("For n = " & CStr(n) & Chr(13) & _
           "Sn = " & CStr(Sn))
End Sub
```
(3)

```vba
Sub SummationDo_04
    Dim k As Integer, n As Integer, Sn As Double
    n = InputBox("Enter n:")
    k = 0 : Sn = 0.0
    Do
       k = k + 1
       Sn = Sn + 1/k^2
    Loop Until k >= n
    MsgBox("For n = " & CStr(n) & Chr(13) & _
           "Sn = " & CStr(Sn))
End Sub
```
(4)

Figure 5. VBA code for calculating the summation $S_n = \sum_{k=1}^{n} \frac{1}{k^2}$ using *Do* loops: (1) *Do While ... Loop*; (2) *Do Until ... Loop*; (3) *Do ... Loop While*; and, (4) *Do ... Loop Until*

You can run these four different Do loop codes for n = 10 to verify that they give the correct result. Care must be given to check that (1) the loop index (k in this case) is properly modified within the loop statements so as to avoid infinite loops, and (2) the condition for terminating the loop represents the proper situation for getting out of the loop.

Breaking out of a Do *loop*

You can use the statement *Exit Do* to unconditionally end the loop. For example, if you want to calculate the summation $S_n = \sum_{k=1}^{n} \frac{1}{k^2}$ for any value $n > 0$, but capping the value of n at 20, you could modify the code of one of the cases in Figure 5 to include an *Exit Do* loop if the user enters a value larger than 20. For example, in the next Figure, the code of case (1) in Figure 5 is modified to end the loop unconditionally if n reaches the value of 20.

```vba
Sub SummationDo_01_Exit
    Dim k As Integer, n As Integer, Sn As Double
    n = InputBox("Enter n:")
    k = 0 : Sn = 0.0
    Do While k < n
       k = k + 1
       Sn = Sn + 1/k^2
       If k >= 20 Then
          Exit Do
       End If
    Loop
    MsgBox("Sn = " & CStr(Sn))
End Sub
```

Figure 6. VBA code of Figure 5, case (1) *Do While … Loop* with an *Exit Do* option.

Test the program for $n = 30$, you should get $Sn = 1.596132....$

***For* loops**

In chapter 2, while discussing the repetition or loop structures, we introduced the *For ... Next* statement as a way to program a loop for a specific number of repetitions. The general form of a *For ... Next* statement is:

```
For k = k0 to kf [Step Δk]
      <statements>
Next k
```

The *For* statement itself includes the following parameters for the *For* loop index k: $k0$ = initial value, kf = upper limit, and Δk = increment. The fact that the *Step* component of the *For* statement is shown between brackets indicates that that the *Step* is an optional component. If the *Step* component is not included, the step Δk is assumed to be equal to the default value of 1.

Assuming $\Delta k>0$ and $k0 \le kf$, the specification *For k = k0 To kf Step Δk* will produce the values:

$$k0,\ k0+\Delta k,\ k0+2\Delta k,\ \ldots,\ ku,$$

where ku is the such that $ku<kf$ and $ku+\Delta k>kf$. For example, the specification *For k = 0.2 To 1 Step 0.25,* will produce the set of values *0.2, 0.45, 0.70, 0.95*. In this example, $ku = 0.95$, which satisfies $ku<1$ and $ku+\Delta k = 1.25>1$.

The number of elements N in the rank generated by the statement *For k = k0 To kf Step Δk* is

$$N = \left[\frac{kf - k0}{\Delta k} \right] + 1$$

where the symbol $[x]$ represents the *floor* function, i.e., the integer value immediately below x. Thus, the number of elements in the range *For k = 0.2 To 1 Step 0.25* is $N = [(1-0.2)/0.25] + 1 = [0.8/0.25] + 1 = [3.2] + 1 = 3+1 = 4$.

When using the *For* statement (*For k = k0 To kf Step Δk*), it is possible to have a negative increment, i.e., $\Delta k<0$, in which case we must have $k0 \ge kf$. For example, the range *For k = -0.25 To -1 Step -0.20* includes the values -0.25, -0.45, -0.65, - 0.85. The number of elements in the range is still calculated by using equation [9], above, i.e., $N = [(-1-(-0.25))/-0.2] + 1 = 4$.

Notes:

(1) If $\Delta k = 1$, the increment can be omitted. Thus, the statement *For k = 1 To n Step 1* can be easily replaced by *For k = 1 To n*.

(2) In a range of the form *For k = k0 To kf Step* Δk when $\Delta k>0$, and $k0 \ge kf$, or when $\Delta k<0$ and $k0 \le kf$, the resulting range is empty. For example, try *For k = 5 To 1 Step 1* or *For k = 1 To 5 Step -1*.

(3) In a range of the form *For k = k0 To kf Step* Δk when $\Delta k<0$, and $k0 \ge kf$, or when $\Delta k>0$ and $k0 \le kf$, the resulting range is infinitely large. Thus, if used to control a loop structure, the program will enter an <u>*infinite loop*</u> (i.e., a loop without ending). You may accidentally create an infinite loop in a program, in which case, you will have to break out of the loop manually.

(4) It is not recommended, not necessary, to modify the index k within the *For* <statements>.

The following Figure shows the VBA code for calculating the summation $Sn = \sum_{k=1}^{n} \frac{1}{k^2}$ using a *For ... Next* statement. Test the code with $n = 10$ to check that you get the result $Sn = 1.54976....$

```
Sub Summation_For
    Dim k As Integer, n As Integer, Sn As Double
    n = InputBox("Enter n:")
    Sn = 0.0
    For k = 1 To n
       Sn = Sn + 1/k^2
    Next
    MsgBox("For n = " & CStr(n) & Chr(13) & _
           "Sn = " & CStr(Sn))
End Sub
```

Figure 7. VBA code for calculating the summation $S_n = \sum_{k=1}^{n} \frac{1}{k^2}$ using a *For ... Next* statement.

The *Step* component of the *For* line in a *For ... Next* statement can be used, for example, in the calculation of the summation of multiples of 3 between 0 and 100, i.e., $S3 = \sum_{k=3, k=k+3}^{100} k$. The following code implements this calculation. The result is 1683.

```
Sub Summation_For_Step
    Dim k As Integer, S3 As Double
    S3 = 0.0
    For k = 3 To 100 Step 3
       S3 = S3 + k
    Next
    MsgBox("S3 = " & CStr(S3))
End Sub
```

Figure 8. VBA code for calculating the summation $S3 = \sum_{k=3, k=k+3}^{100} k$ using a *For ... Next* statement.

Breaking out of a For *loop*

You can use the statement *Exit For* to unconditionally end the loop. For example, if you want to calculate the summation of Figure 7, for any value $n > 0$, but capping the value of n at 20, you could modify the code of Figure 7 to include an *Exit For* loop if the user enters a value larger than 20. For example, in the next Figure, the code of Figure 7 is modified to end the loop unconditionally if n reaches the value of 20.

```
Sub Summation_For
    Dim k As Integer, n As Integer, Sn As Double
    n = InputBox("Enter n:")
    Sn = 0.0
    For k = 1 To n
       Sn = Sn + 1/k^2
       If k >= 20 Then Exit For
    Next
    MsgBox("Sn = " & CStr(Sn))
End Sub
```

Figure 9. VBA code of Figure 7 with an *Exit For* option.

Test the program for n = 30, you should get Sn = 1.596132....

Nested *For* loops

Nested loops are loops controlled by two indices, the outer index varies at a lower pace than the inner index. For every value of the outer index, the inner index changes through its own series of values. Thus, the value of the inner index must be reinitialized for each value of the outer index. Since the index of a *For ... Next* loop statement is initialized automatically, nested *For* loops are preferred to other type of loop statements.

Consider, for example, the calculation of a double summation such as $S_d = \sum_{i=1}^{n} \sum_{j=1}^{m} \frac{1}{i+j}$. In this summation *i* is the outer loop index while *j* is the inner loop index. For each value of *i,* (*i = 1, 2, ..., n*), *j* takes the values *j = 1, 2, ..., m*. Thus, the summation is accumulated a total of *m*×*n* times. The following VBA code can be used to evaluate this double summation:

```
Sub Summation_Double
    Dim i As Integer, j As Integer
    Dim n As Integer, m As Integer, Sd As Double
    n = InputBox("Enter n:")
    m = InputBox("Enter m:")
    Sd = 0.0
    For i = 1 To n
        For j = 1 To m
            Sd = Sd + 1/(i+j)
        Next
    Next
    MsgBox("For n = " & CStr(n) & Chr(13) & _
           "and m = " & CStr(m) & Chr(13) & _
           "Sd = " & CStr(Sd))
End Sub
```

Figure 10. VBA code to calculate the double summation $S_d = \sum_{i=1}^{n} \sum_{j=1}^{m} \frac{1}{i+j}$ using nested *For* loops.

Check that for *n = 3* and *m = 4*, *Sd = 2.992857.*

Summations and other iterative operations in Excel

The examples presented above to illustrate the programming of looping, or repetition, statements in VBA were used to calculate summations. A summation is an example of an iterative operation, i.e., one in which an index controls the value of the terms added up into the summation. Iterated products are sometimes used in calculations. Thus, for a product we could write:

$$Pn = \prod_{k=1}^{n} f(k) \quad ,$$

where the symbol Π (the upper case Greek letter *pi*) represents a product. A product can be programmed in VBA in a similar fashion as we did for the summation, except that the product is initialized to 1.0 rather than to 0.0 as we did for the summation. Here's an example for programming the product $Pn = \prod_{k=1}^{n} \frac{1}{k}$ in VBA.

```
Sub Product_01
    Dim k as Integer, n As integer, Pn As Double
    n = InputBox("Enter n:")
    Pn = 1.0
    For k = 1 To n
        Pn = Pn*(1/k)
    Next k
    MsgBox("For n = " & CStr(n) & Chr(13) & _
           "Pn = " & CStr(Pn))
End Sub
```

Figure 11. VBA code for programming the product $Pn = \prod_{k=1}^{n} \frac{1}{k}$ using a *For ... Next* statement.

A test of the code for $n = 10$ produces the result $Pn = 2.7557319...\times10^{-7}$.

While *For … Next* statements can be used to calculate summations and products, for a relatively small number of elements in the summation or product, you can use the *Excel* worksheet directly to calculate summations and products with the functions *SUM* and *PRODUCT*, as illustrated in Figure 12.

	B	C	D
3	*k*	*1/k^2*	*1/k*
4	1	1	1
5	2	0.25	0.5
6	3	0.111111111	0.333333333
7	4	0.0625	0.25
8	5	0.04	0.2
9	6	0.027777778	0.166666667
10	7	0.020408163	0.142857143
11	8	0.015625	0.125
12	9	0.012345679	0.111111111
13	10	0.01	0.1
14		1.549767731	2.7557E-007
15		SUM	PRODUCT

(a) Values

	B	C	D
3	*k*	*1/k^2*	*1/k*
4	1	=1/B4^2	=1/B4
5	=1+B4	=1/B5^2	=1/B5
6	=1+B5	=1/B6^2	=1/B6
7	=1+B6	=1/B7^2	=1/B7
8	=1+B7	=1/B8^2	=1/B8
9	=1+B8	=1/B9^2	=1/B9
10	=1+B9	=1/B10^2	=1/B10
11	=1+B10	=1/B11^2	=1/B11
12	=1+B11	=1/B12^2	=1/B12
13	=1+B12	=1/B13^2	=1/B13
14		=SUM(C4:C13)	=PRODUCT(D4:D13)
15		SUM	PRODUCT

(b) Formulas

Figure 12. Summation and product example calculated using the Excel spreadsheet functions SUM and PRODUCT.

In the worksheet of Figure 12, values of k = 1, 2, …, 10, are listed in column B. Column C includes the values $1/k^2$ while column D includes the values $1/k$. Cell C14 includes the formula "=SUM(C4:C13)" which basically calculates the summation $Sn = \sum_{k=1}^{n} \frac{1}{k^2}$. On the other hand, cell D14 includes the formula "=PRODUCT(C4:C13)" which calculates the product $Pn = \prod_{k=1}^{n} \frac{1}{k}$.

Summations as Numerical Approximations to Integrals

The integral $I = \int_a^b f(x)\,dx$ can be approximated by a summation as follows. The integration interval [a, b] is divided into n subintervals: [x_1, x_2], [x_2, x_3], …, [x_k,x_{k+1}], …, [x_{n-1}, x_n], [x_n, x_{n+1}], of widths $\Delta x_k = x_{k+1} - x_k$., k = 1, 2, …, n. For each subinterval in the partition, [x_k,x_{k+1}], we select a value in the subinterval ξ_k, with $x_k \le \xi_k \le x_{k+1}$, so that the integral can be approximated by

$$S_n = \sum_{k=1}^{n} f(\xi_k)\,\Delta x_k \quad .$$

For the purpose of programming, we are going to make the partition of the interval [a, b] such that the n subintervals have the same width, Δx, i.e., $n \cdot \Delta x = b - a$. Thus, given the parameters a, b, and n, we can calculate the constant subinterval width as:

$$\Delta x = \frac{b-a}{n} \quad . \qquad [1]$$

The limits of the subintervals can be calculated, systematically, as:

$$x_1 = a \quad ,$$

$$x_2 = x_1 + \Delta x = a + \Delta x \quad ,$$

$$x_3 = x_1 + \Delta x = a + 2 \cdot \Delta x \quad ,$$

$$\ldots.$$

$$x_k = a + (k-1) \cdot \Delta x \quad ,$$

$$\ldots$$

$$x_{n+1} = a + n \cdot \Delta x = b \quad .$$

Thus, the generic k-th interval, [x_k,x_{k+1}], has limits: $x_k = a + (k-1) \cdot \Delta x$, and $x_{k+1} = a + k \cdot \Delta x$.

In the definition of the summation above, we can select the value ξ_k as any value in [x_k,x_{k+1}]. Thus, we could select ξ_k to be either $x_k = a + (k-1) \cdot \Delta x$, or $x_{k+1} = a + k \cdot \Delta x$. Since these values correspond to the left-hand side and right-hand side extremes of the k-th subinterval, we can write the following

summations:

$$S_{nL} = \sum_{k=1}^{n} f(x_k) \cdot \Delta x = \Delta x \cdot \sum_{k=1}^{n} f\left(a + (k-1) \cdot \Delta x\right) \quad , \qquad [2]$$

and

$$S_{nR} = \sum_{k=1}^{n} f(x_{k+1}) \cdot \Delta x = \Delta x \cdot \sum_{k=1}^{n} f\left(a + k \cdot \Delta x\right) \quad , \qquad [3]$$

to approximate the integral $I = \int_a^b f(x)\,dx$. The two summations above, [2] and [3], can now be programmed, together with equation [1], to estimate the desired integral. In the code, we'll define the function $f(x)$ using a user-defined function subprogram. The example code, shown below, calculates the estimates of the integral for $a = 0$, $b = \pi$, $f(x) = \dfrac{1}{1+x^2}$, while let n to be entered by the user.

```
Sub Integrals_01
    Dim a As Double, b As Double, PI As Double
    Dim k As Double, Dx As Double, n As Integer
    Dim SnL As Double, SnR As Double
    PI = 4.0 * Atn(1.0)
    a = 0.0 : b = PI
    n = InputBox("Enter n = ")
    Dx = (b-a)/n
    SnL = 0.0 : SnR = 0.0
    For k = 1 To n
        SnL = SnL + f(a+(k-1)*Dx)
        SnR = SnR + f(a+k*Dx)
    Next
    SnL = SnL*Dx : SnR = SnR*Dx
    MsgBox("For n = " & CStr(n) & Chr(13) & _
           "SnL = " & CStr(SnL) & Chr(13) & _
           "SnR = " & CStr(SnR))
End Sub

Function f(x As Double) As Double
    f = 1/(1+x^2)
End Function
```

Figure 13. VBA code for calculating the approximations of the integral $\int_0^{\pi} \dfrac{dx}{1+x^2} = 1.2626$, using the summations of equations [2] and [3], and Δx calculated from equation [1].

The approximations to the integral get better and better as the value of n increases, as indicated below.

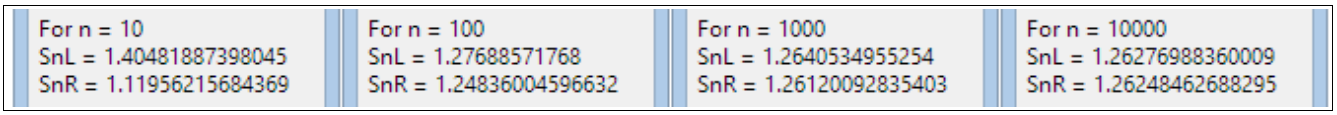


Figure 14. Estimates of the integral coded in Figure 13 for values of n = 10, 100, 1000, and 10000.

Application of Loop Statements for Solving Single Equations – The fixed-point iteration method

A generic equation $f(x) = 0$ can be recast as $x = g(x)$, where $f(x) = x - g(x)$, and solved iteratively by using the expression:

$$x_{k+1} = g(x_k). \qquad [4]$$

An initial guess, x_0, is (somewhat) arbitrarily selected, and then values $x_1 = g(x_0)$, $x_2 = g(x_1)$, …, etc., are calculated. To check if we are approaching, or converging, to a solution, we check the percentage relative error:

$$\epsilon_a = \left| \frac{x_{k+1} - x_k}{x_k} \right| \cdot 100 \quad \%. \qquad [5]$$

If the relative error is less than a certain tolerance $\varepsilon_r > 0$, i.e., if $|\varepsilon_a| < \varepsilon_r$, we declare the equation solved, with the solution being the last value of x calculated. On the other hand, it could happen that the solution instead of converging, diverges (moves away from the solution). To stop the iterative process in such a case, we allow a maximum number of iterations, N_{max}. If the iteration number, k, reaches that maximum number without convergence, i.e., if $k \geq N_{max}$, we stop the process and request a different initial guess, x_0, from the user to start a new attempt to the solution. This equation solution method is called the *fixed-point iteration* method.

To illustrate this approach, we propose to obtain the solution to the equation $f(x) = x^3 - 4.8 \cdot x^2 + 1.55 \cdot x + 10.5 = 0$, by recasting it as: $x = \frac{4.8 \cdot x^2 - 1.55 \cdot x - 10.5}{x^2}$. We'll use a starting value of $x_0 = 2$, a tolerance of $\varepsilon_r = 0.1$ %, and a maximum number of iterations $N_{max} = 100$. The following code can be used to calculate a solution.

```
Sub FixedPointSolution
    Dim xk As Double, xkp1 As Double
    Dim ea As Double, et As Double
    Dim k As Integer, Nmax As Integer
    et = 0.1 : Nmax = 100
    xk = InputBox("Enter x0:")
    ea = 100 : k = 0
    Do
       k = k + 1
       xkp1 = g(xk)
       ea = abs((xkp1-xk)/xkp1)*100
       xk = xkp1
    Loop While abs(ea)>et And k < Nmax
    If abs(ea) <= et Then
       MsgBox("For k = " & CStr(k) & Chr(13) & _
              "x = " & CStr(xk) & Chr(13) & _
              "f(x)=" & CStr(f(xk)))
    ElseIf k>= Nmax Then
       MsgBox("For k = " & CStr(k) & Chr(13) & _
              "No convergence achieved")
    End If
End Sub
```

```
Function g(x As Double) As Double
   g = (4.8*x^2-1.55*x-10.5)/x^2
End Function

Function f(x As Double) As Double
   f = x^3 - 4.8*x^2 + 1.55*x + 10.5
End Function
```

Figure 14. An VBA code implementation of the fixed-point iteration method

A brief description of the operation of *Sub FixedPointSolution* is presented next:
The value of the tolerance, *et* = 0.1, and the maximum number of iterations, *Nmax* = 100, is set at the

start of the program. The initial guess, *xk* (representing x_k), is then requested from the user. Next, the relative error *ea* is initialized to 100%, and *k* to zero. Then a *Do ... Loop While* statement is used to perform the following tasks:

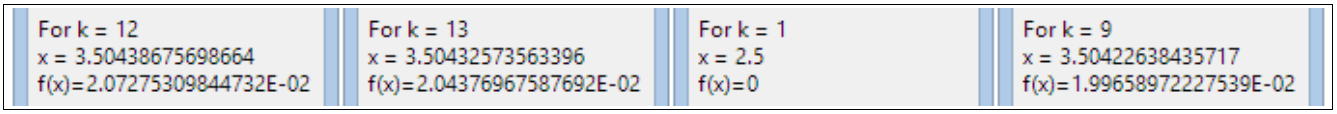
- increment *k, k = k+1*
- calculate *xkp1 = g(xk)*, i.e., $x_{k+1} = g(x_k)$
- calculate relative error, $e_a = \left|\frac{x_{k+1} - x_k}{x_{k+1}}\right| \cdot 100 \quad \%$
- make *xk = xkp1*,i.e., $x_k = x_{k+1}$, for the next iteration
- repeat loop as long as *abs(ea) > et* [i.e., relative error not yet smaller than tolerance] and *k<Nmax* [number of iterations still smaller than maximum number allowed]

As soon as either |*ea*|<*et* [convergence], or *k>Nmax* [no convergence], control is sent out of the loop to an *If ... ElseIf ... End If* statement that checks whether:

- |ea| < et, and shows a result, or
- k >= Nmax, and shows "no convergence"

The code of Figure 14 shows also a couple of user-defined functions, namely, *g*(*x*), – used to perform the fixed-point iteration –, and *f*(*x*), – the original equation to be solved. Function *f*(*x*) is evaluated in the first *MsgBox* within the *If ... ElseIf ... End If* statement to show how close *f*(*x*) is to zero when a solution is found.

The following figure shows the result produced by the code of Figure 14 when the following initial guesses for *x* are used: x_0 = 0.5, 1.5, 2.5, 4.0. The program is run from the VBA IDE.



(a) x_0 = 0.5 (b) x_0 = 1.5 (c) x_0 = 2.5 (d) x_0 = 4.0

Figure 15. Fixed-point iteration solutions using the code of Figure 14 for various initial guesses.

The program outputs shown in Figure 15 show the number of iterations taken to arrive to a solution for the different initial guesses used. For case (c) x_0 = 2.5 it turns out that the initial guess is a solution to the equation, so only one iteration is used (*k* = 1). All other initial guesses converge to a second solution *x* = 3.504..., and the function evaluation at that solution is close to zero, *f*(*x*) ≈ 0.02... . Some conclusions that can be drawn from these observations are the following:

- For the fixed-point iteration expression used, namely, $x = (4.8 \cdot x^2 - 1.55 \cdot x - 10.5)/x^2$, most initial guesses seem to drive the solution towards the value *x* = 3.5
- If you happen to enter one of the solutions as initial guess, the loop is executed only once, since the convergence condition is already fulfilled.

Other issues to keep in mind about this program:

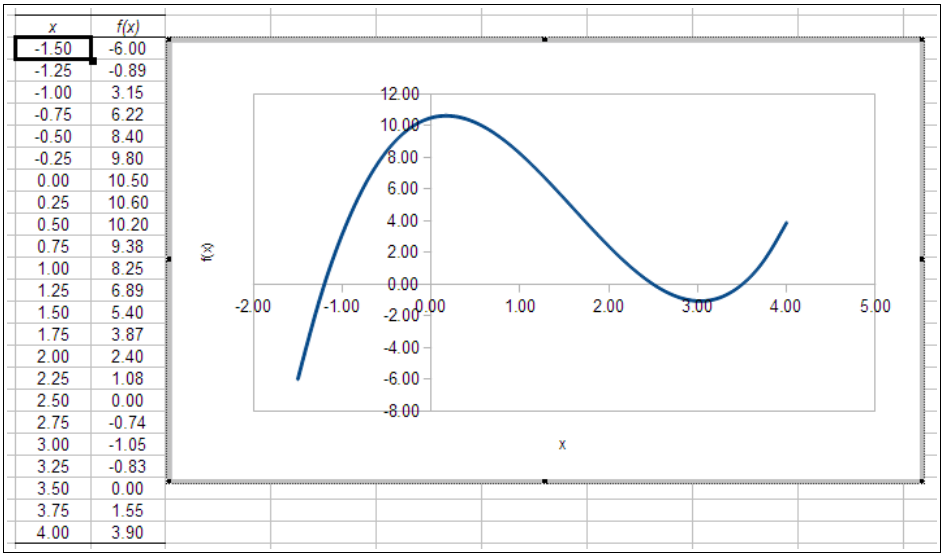
- Given the fixed-point iteration expression used, namely, $x=(4.8\cdot x^2-1.55\cdot x-10.5)/x^2$, an initial guess of zero ($x_0 = 0$) will produce a *division-by-zero* error right away, since there is $^{a\,x2}$ in the denominator. Handling a zero initial guess is not properly addressed by the code in Figure 14, therefore, such an initial guess will crash the program. Additional code can be added to let the program handle this situation. For example, you could adjust the initial guess to a small, non-zero value, say, 0.01, if the user enters an initial guess of zero. Thus, immediately below the line `xk = InputBox("Enter x0:")` you could add the following line of code:

```
If x0 = 0 Then x0 = 0.01
```

- The fixed-point iteration expression used in this code, $x=(4.8\cdot x^2-1.55\cdot x-10.5)/x^2$, is not the only way to define the expression $x_{k+1} = g(x_k)$. Other possibilities you could try are:

$$x=\frac{x^3+1.55\cdot x+10.5}{4.8\cdot x} \quad , \quad x=\frac{-x^3+4.8\cdot x^2-10.5}{1.55\cdot x} \quad \text{, or even,} \quad x=x^3-4.8\cdot x^2+2.55\cdot x+10.5 \quad .$$

- The loop in the code of Figure 14 was written using a *Do ... Loop While* statement. You could re-write this code using one of the other VBA loop statements: *While ... Wend*, *Do ... Loop Until*, *Do While ... Loop, Do Until … Loop*, or even a *For ... Next* statement with an *Exit For* option. When using any of these loop statements, make sure to write the proper logical statement to send the control out of the loop when one of the conditions $|\varepsilon_a| < \varepsilon_r$, or $k \geq N_{max}$, are fulfilled.
- A graph of the function can be used to obtain good initial guesses for the solution. For example, the following figure shows the plot of the function $f(x)=x^3-4.8\cdot x^2+1.55\cdot x+10.5$. From Figure 16 it follows that roots are between -2 and -1, 2 and 3, 3 and 4.



x	f(x)
-1.50	-6.00
-1.25	-0.89
-1.00	3.15
-0.75	6.22
-0.50	8.40
-0.25	9.80
0.00	10.50
0.25	10.60
0.50	10.20
0.75	9.38
1.00	8.25
1.25	6.89
1.50	5.40
1.75	3.87
2.00	2.40
2.25	1.08
2.50	0.00
2.75	-0.74
3.00	-1.05
3.25	-0.83
3.50	0.00
3.75	1.55
4.00	3.90

Figure 16. Graph of the function $f(x)=x^3-4.8\cdot x^2+1.55\cdot x+10.5$.

The *fixed-point method* for solving equations is not necessarily the best approach for finding solutions to equations. Other methods, such as the *bisection,* the *Newton-Raphson*, or the *secant* method are more effective. In the following section we introduce the Newton-Raphson method.

The Newton-Raphson method for solving equations

The Newton-Raphson method uses the following iterative formula:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \qquad [6].$$

for $k = 0, 1, 2, \ldots$ to solve the equation $f(x) = 0$, using not only $f(x)$, but also its first derivative $f'(x)$.

As in the case of the fixed-point iteration method, the Newton-Raphson method requires an initial guess x_0, and the iteration continues until one of the conditions: $|\varepsilon_a| < \varepsilon_t$ (convergence), or $k \geq N_{max}$, (no convergence) are fulfilled. In these expressions, ε_t is a percentage tolerance for convergence, and N_{max} is the maximum number of iterations allowed before declaring that the solution does not converge. The relative error ε_a in each iteration is calculated as in equation [5], above.

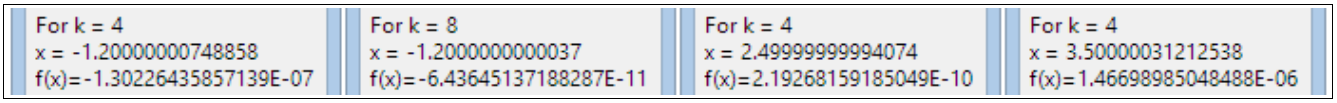
To illustrate the coding of the Newton-Raphson method we use the same equation used earlier, namely, $f(x) = x^3 - 4.8 \cdot x^2 + 1.55 \cdot x + 10.5$, whose derivative is $f'(x) = 3 \cdot x^2 - 9.6 \cdot x + 1.55$. The code, shown in Figure 17, below, is basically the same as in Figure 14, except for replacing the line xkp1 = g(xk) with xkp1 = xk – f(xk)/fp(xk). Also, function $g(x)$ is no longer needed, and is replaced by function $fp(x) = f'(x)$. Thus, the operation of the code is very similar to that in Figure 14.

```
Sub NewtonRaphson
 Dim xk As Double, xkp1 As Double
    Dim ea As Double, et As Double
    Dim k As Integer, Nmax As Integer
    et = 0.1 : Nmax = 100
    xk = InputBox("Enter x0:")
    ea = 100 : k = 0
    Do
       k = k + 1
       xkp1 = xk - f(xk)/fp(xk)
       ea = abs((xkp1-xk)/xkp1)*100
       xk = xkp1
    Loop While abs(ea)>et And k < Nmax
    If abs(ea) <= et Then
       MsgBox("For k = " & CStr(k) & Chr(13) & _
              "x = " & CStr(xk) & Chr(13) & _
              "f(x)=" & CStr(f(xk)))
    ElseIf k>= Nmax Then
       MsgBox("For k = " & CStr(k) & Chr(13) & _
              "No convergence achieved")
    End If
End Sub
```

```
Function f(x As Double) As Double
   f = x^3 - 4.8*x^2 + 1.55*x + 10.5
End Function

Function fp(x As Double) As I
   fp = 3*x^2 - 9.6*x + 1.55
End Function
```

Figure 17. An VBA code implementation of the Newton-Raphson method.

Some solutions are shown below, corresponding to $x_0 = -2$, 0, 2, and 4:



(a) $x_0 = -2.0$ (b) $x_0 = 0.0$ (c) $x_0 = 2.0$ (d) $x_0 = 4.0$

Figure 18. Fixed-point iteration solutions using the code of Figure 17 for various initial guesses.

The results of Figure 18 cover the three possible real roots that the cubic polynomial $f(x)=x^3-4.8\cdot x^2+1.55\cdot x+10.5$, namely, x = -1.2, 2.5, and 3.5.

Newton-Raphson method with *an Excel* interface

The solution of the equation $f(x) = 0$ through the Newton-Raphson method illustrated in the code of Figure 17 is run from the *VBA IDE* and requires the use of two user-defined functions, namely, $f(x)$ and $fp(x) = f'(x)$. Using a worksheet, named "Newton", as interface, we can utilize worksheet cells to enter functions $f(x)$ and $f'(x)$ as formulas, and not have to write those functions in code. A possible form of the interface is shown in Figure 19.

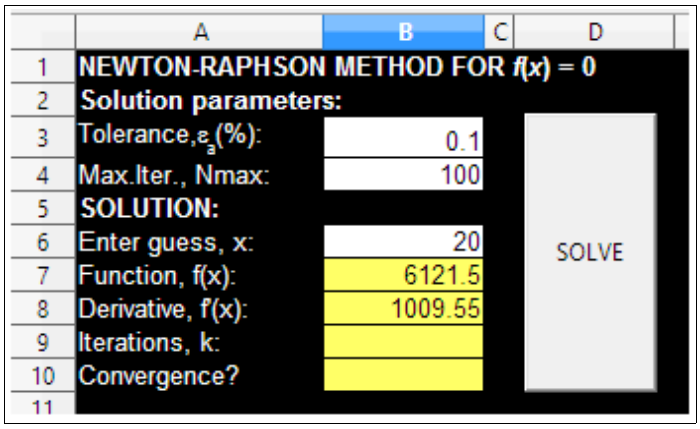


Figure 19. An interface for activating the Newton-Raphson method to solve equations of the form $f(x) =0$, with derivative $f'(x)$. The worksheet is named "Newton".

The user enters the values of the tolerance ε_a (B3), the maximum number of iterations N_{max} (B4), and an initial guess for x (B6), and presses the button [SOLVE], and the program uses cells B7, for $f(x)$, and B8, for $f'(x)$ to attempt a solution. If a solution is found, the value at B7 should be close to zero, and B10 should show the result "Yes". The number of iterations needed to obtain a solution is shown in B9. The code that process this solution is shown in Figure 20.

The user needs to define an expression for $f(x)$ and $f'(x)$ in cells B7 and B8, respectively, by using an expression involving x. The contents of cell *B6* are defined as x by using *Data* > *Define Range*. Make sure that the range *$Newton.$B$6* is selected to define x. As an example, you can enter the following formulas in cells B7 and B8, for $f(x)=x^3-4.8\cdot x^2+1.55\cdot x+10.5$, and $f'(x)=3\cdot x^2-9.6\cdot x+1.55$, respectively:

- cell B7: `= x^3-4.8*x^2+1.55*x+10.5`
- cell B8: `=3*x^2-9.6*x+1.55`

Note that, once the program is activated by pressing the [SOLVE] button, the value of x in cell B6 will be replaced by the updated values calculated by the Newton-Raphson method. The corresponding values of $f(x)$ and $f'(x)$ will be updated automatically by the worksheet.

```
Sub NewtonRaphsonInterface
 Dim xk As Double, xkp1 As Double
    Dim ea As Double, et As Double
    Dim k As Integer, Nmax As Integer
    Dim f As Double, fp As Double
    et = Worksheets("Newton").Range("B3").Value
    Nmax = Worksheets("Newton").Range("B4").Value
    xk = Worksheets("Newton").Range("B6").Value
    ea = 100 : k = 0
    Do
       k = k + 1
       f = Worksheets("Newton").Range("B7").Value
       fp = Worksheets("Newton").Range("B8").Value
       xkp1 = xk - f/fp
       ea = abs((xkp1-xk)/xkp1)*100
       xk = xkp1
       Worksheets("Newton").Range("B6").Value = xk
    Loop While abs(ea)>et And k < Nmax
    Worksheets("Newton").Range("B9").Value = k
    If abs(ea) <= et Then
       Worksheets("Newton").Range("B10").Value = "Yes"
    ElseIf k>= Nmax Then
       Worksheets("Newton").Range("B10").Value = "No"
    End If
End Sub
```

Figure 20. An VBA code implementation of the Newton-Raphson method linked to the interface of Figure 19.

The following figure shows the interface of Figure 19 after a solution has been found starting with initial guesses of x = -2, 2, and 4, which produces the three real roots of the polynomial $f(x)=x^3-4.8\cdot x^2+1.55\cdot x+10.5$, namely, x = -1.2, 2.5, 3.5.

	A	B	C	D	B	B
1	NEWTON-RAPHSON METHOD FOR *f(x)* = 0				N METHOD FO	N METHOD FOR
2	Solution parameters:				rs:	rs:
3	Tolerance, ε_a(%):	0.1			0.1	0.1
4	Max.Iter., Nmax:	100			100	100
5	SOLUTION:					
6	Enter guess, x:	-1.200000007		SOLVE	2.5	3.500000312
7	Function, f(x):	-1.3023E-007			2.1927E-010	0.000001467
8	Derivative, f'(x):	17.39000013			-3.7	4.700003558
9	Iterations, k:	4			4	4
10	Convergence?	Yes			Yes	Yes
11						

(a) x_0 = -2 (b) x_0 = 2 (c) x_0 = 4

Figure 21. The *Excel* interface after solving the equation $f(x)$ = 0 for initial guesses as shown.

A second example of the Newton-Raphson method using an Excel interface

The interface of Figure 19, linked to the code of Figure 20, allows the user to find solutions for an equation of the form $f(x) = 0$ if the derivative $f'(x)$ is known. All the user needs to do is to type expressions for those functions in cells B7 and B8 of the interface. As an example, let's try to find a solution to the equation

$$f(x)=\sqrt{1+x^2}-\sin(x)-1.33=0 \quad ,$$

whose derivative is

$$f'(x)=\frac{x}{\sqrt{1+x^2}}-\cos(x) \quad .$$

Entering the formulas "=Sqrt(1+x^2)-Sin(x)-1.33" in cell B7, and "=x/Sqrt(1+x^2) – cos(x)" in cell B8, and using an initial guess $x_0 = 4$, we find that the solution is $x = 2.00$, $f(x) = 5.58\times10^{-9}$, $k = 4$.

The Newton-Raphson Method with a Excel Interface for Solving an Engineering Equation

An engineering equation, involving a number of known variables, say, *a, b, c*, etc., and one unknown, say, *x*, can be recast into the form $f(x) = 0$. If the derivative of this function, $f'(x)$ can be found, then the interface of Figure 19, with the code of Figure 20, can be used to solve for the unknown. Consider, for example, the equation that describes the trajectory of a projectile in the *x*-*y* plane under the effect of gravity:

$$y=y_0+\tan(\theta_0)\cdot(x-x_0)-\frac{g\cdot(x-x_0)^2}{2\cdot v_0^2\cdot\cos(\theta_0)^2} \quad , \qquad [7]$$

Here, (x,y) are the coordinates of the trajectory, (x_0, y_0) are the coordinates of the launching point, v_0 is the launching speed, θ_0 is the angle of launch, and g is the acceleration of gravity (in vector notation, $\mathbf{g} = -g\mathbf{j}$). Suppose you are given values of $(x_0, y_0) = (2.0 \text{ m}, 5.0 \text{ m})$, $(x,y) = (5.0 \text{ m}, 3.5 \text{ m})$, $v_0 = 5$ m/s , and $g = 9.81$ m/s^2, and you want to calculate the corresponding value of θ_0. Replacing the known values in [7], without including units, and replacing θ_0 with x, we get:

$$3.5=5.0+\tan(x)\cdot(5.0-2.0)-\frac{9.81\cdot(5.0-2.0)^2}{2\cdot5.0^2\cdot\cos^2(x)} \quad ,$$

which results in:

$$f(x)=1.5+3.0\cdot\tan(x)-\frac{1.7658}{\cos^2(x)}=0 \quad .$$

The derivative is:

$$f'(x)=3.0*sec^2(x)-3.5316\cdot\frac{\sin(x)}{\cos^3(x)} \quad .$$

Now, we go back to the interface of Figure 19, and enter the expressions "=1.5+3.0*TAN(x)-1.7658/(COS(x))^2" and "=3.0*(SEC(x))^2 – 3.5316 * SIN(x)/(COS(x))^3" in cells B7 and B8, respectively. Starting with $x = 0$, the solution is $x = 0.0935$, while for a starting value of $x = 1$. the result is $x = 1.013$. These results are in *radians*.

In summary, the interface of Figure 19, with the code of Figure 20, can be used to solve single equations as long you can cast the equation in the form $f(x) = 0$, and know the derivative of the function, $f'(x)$. If the derivative $f'(x)$ is not known or difficult to obtain, you may consider using the so-called *secant method*, in which the derivative $f'(x)$ is replaced by the approximation

$$f'(x) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}} \quad . \qquad [8]$$

The implementation of this method is left as an exercise.

Filling out a table with a *For* loop -Use of the *Cells*() property in VBA

In this example we will produce a table calculating velocities and positions for a projectile motion based on the equations of motion for a projectile launched at time t_0, from position (x_0, y_0) at a launching speed v_0, with θ_0 being the angle of launch, and g being the acceleration of gravity (in vector notation, $\mathbf{g} = -g\mathbf{j}$). The equations that calculate velocities v_x and v_y, and positions (x, y), all as functions of time, t, are given by:

$$v_x = v_0 \cdot \cos(\theta_0) \quad , \quad v_x = v_0 \cdot \cos(\theta_0) - g \cdot (t - t_0) \quad , \qquad [9]$$

$$x = x_0 + v_0 \cdot \cos(\theta_0) \cdot (t - t_0) \quad \text{, and} \quad y = y_0 + v_0 \cdot \sin(\theta_0) \cdot (t - t_0) - \frac{1}{2} \cdot g \cdot (t - t_0)^2 \quad . \qquad [10]$$

The purpose of the program is to produce a table as illustrated in the following Figure.

Column (j) →	2	3	4	5	6	7
Row (i) ↓	B	C	D	E	F	G
1						
2	x0 (m):	2.5	y0 (m):	3.5	t0 (s)	2.5
3	v0 (m/s):	7.5	θ0 (deg):	20	Δt (s):	0.1
4					n	10
5		FILL UP TABLE			Clear Table	
6						
7	k	t (s)	vx (m/s)	vy (m/s)	x (m)	y (m)
8	1	2.50	7.05	2.57	2.50	3.50
9	2	2.60	7.05	1.58	3.20	3.71
10	3	2.70	7.05	0.60	3.91	3.82
11	4	2.80	7.05	-0.38	4.61	3.83
12	5	2.90	7.05	-1.36	5.32	3.74
13	6	3.00	7.05	-2.34	6.02	3.56
14	7	3.10	7.05	-3.32	6.73	3.27
15	8	3.20	7.05	-4.30	7.43	2.89
16	9	3.30	7.05	-5.28	8.14	2.41
17	10	3.40	7.05	-6.26	8.84	1.84
18						

Figure 22. Excel worksheet interface to fill up table of velocities and positions for a projectile motion.

The interface of Figure 22 includes two buttons: [FILL UP TABLE] and [Clear Table]. The [FILL UP TABLE] button will take the data in cells C2 (x_0), C3 (v_0), E2 (y_0), E3 (θ_0), G2 (t_0), G3 (Δt), and G4 (n), and calculate the velocity components (v_x, v_y) and positions (x, y) of the projectile for various times t. Variables x_0, y_0, v_0, θ_0, t_0, v_x, v_y, x, and y were defined earlier. The variable n represents the number of terms to be included in the table ($n > 0$, n integer), and Δt is the time increment for the various times t to be included in the table. The table includes columns k, t, v_x, v_y, x, and y. The values k simply refer to the index in each table entry. Thus, k takes values of 1, 2, …, etc. The times t, in the table, actually represent the values $t_k = t_0 + (k-1)\cdot\Delta t$. For each value of t, thus generated, equations [9] and [10] produce the required values of v_x, v_y, x, and y. The table can be filled by using a *For .. Next* statement, that uses the index k.

In previous input-output operations that use the *Excel* worksheet interface we used the *Range*( ) property to refer to specific cells in the worksheet. For the development of the present program, however, we'll introduce the use of the *Cells*( ) property for input and output, as described next.

The *Cells*( ) property is used to refer to a specific cell in the worksheet. When attached to the *Worksheets*( ) object, the *Cells*( ) property uses two arguments representing the row number (*i*) and the column number (*j*) of the required cell, i.e., *Cells*(*i*,*j*). Figure 22 includes the indices of rows and columns for a worksheet. Thus, referring to Figure 22, *Range*("C2") is equivalent to *Cells*(2,3), while *Range*("E2") is equivalent to *Cells*(2,5), etc. If the worksheet name is, say, "myTable", then the input statements for the values of x_0 and y_0, in code, would be

```
x0 = Worksheets("myTable").Cells(2,3).Value
```

and

```
y0 = Worksheets("myTable").Cells(2,5).Value,
```

respectively.

The fact that the *Cells*( ) property uses numerical indices to refer to cells makes the use of this approach ideal for iteratively filling tables in the worksheet. Thus, after the values of t, v_x, v_y, x, and y have been calculated for each value of k in the example of Figure 22, statements such as :

```
Worksheets("myTable").Cells(7+k,3).Value = t
Worksheets("myTable").Cells(7+k,4).Value = vx
Worksheets("myTable").Cells(7+k,5).Value = vy
Worksheets("myTable").Cells(7+k,6).Value = x
Worksheets("myTable").Cells(7+k,7).Value = y
```

will take care of filling each row in the table. In these statements, the row reference within the *Cells*( ) property calls shown is 7+k. This is so because the first entry in the table (Figure 22), which corresponds to k = 1, is posted in line *8*, i.e., for k = 1, 7 + k = 8. Subsequent values of k, say, k = 2, 3, etc., will fill lines 7+k = 9, 10, etc.

The code associated with button [FILL UP TABLE] is shown in Figure 23. All input from and output to the worksheet, named "myTable", is carried out using the *Cells*( ) property.

```
Sub FillTable
'Variable declarations
    Dim x0 As Double, y0 As Double, x As Double, y As Double
    Dim v0 As Double, theta0 As Double, vx As Double, vy As Double
    Dim t0 As Double, Dt As Double, t As Double
    Dim n As Integer, k As Integer
'Constants
    CONST DegToRad = 1.7453E-2
    CONST g = 9.806
'Input given data
    x0 = Worksheets("myTable").Cells(2,3).Value
    y0 = Worksheets("myTable").Cells(2,5).Value
    t0 = Worksheets("myTable").Cells(2,7).Value
    v0 = Worksheets("myTable").Cells(3,3).Value
    theta0 = Worksheets("myTable").Cells(3,5).Value*DegToRad
    Dt = Worksheets("myTable").Cells(3,7).Value
    n = Worksheets("myTable").Cells(4,7).Value
'calculate values and write table
    For k = 1 To n
        t = t0 + (k-1)*Dt
        vx = v0*cos(theta0)
        vy = v0*sin(theta0)-g*(t-t0)
        x = x0 + v0*cos(theta0)*(t-t0)
         y = y0 + v0*sin(theta0)*(t-t0) - 0.5*g*(t-t0)^2
        Worksheets("myTable").Cells(7+k,2).Value = k
        Worksheets("myTable").Cells(7+k,3).Value = t
        Worksheets("myTable").Cells(7+k,4).Value = vx
        Worksheets("myTable").Cells(7+k,5).Value = vy
        Worksheets("myTable").Cells(7+k,6).Value = x
        Worksheets("myTable").Cells(7+k,7).Value = y
    Next
End Sub
```

Figure 23. VBA code associated with button [FILL UP TABLE] in worksheet "myTable" as shown in Figure 22.

The code of Figure 23 includes the definition of a couple of *CONSTANT* values. The value *DegToRad* is the constant required to convert angles in degrees to angles in radians, i.e., $\pi/180 = 1.7453 \times 10^{-2}$. This is necessary because trigonometric functions in VBA (and most computer languages) are defined exclusively for arguments in radians, rather than degrees. However, the angle in the interface, shown in Figure 22, is in degrees. The second constant defined is the acceleration of gravity, $g = 9.806$ m/s^2.

The results shown in Figure 23 correspond to the table produced by the code of Figure 23 for the input data shown (i.e., $x_0 = 2.5$ m, $y_0 = 3.5$ m, $v_0 = 7.5$ m/s, $\theta_0 = 20^o$, $t_0 = 2.5$ s, $\Delta t = 0.1$ s, and $n = 10$. The [FILL UP TABLE] button is linked to macro *FillTable* (Figure 23). Pressing the button will fill up the table as shown in Figure 22.

The second button in the interface of Figure 22, namely [Clear Table], will clear both the input cells and the table entries, so that a new table, with different values of the parameters, could be created. Basically, clearing the table means to set the values of the required cells to empty, or = "". The code needed to clear the table is shown below. Notice that clearing up the table entries can be accomplished by using two nested *For ... Next* loops.

```
Sub ClearTheTable
'Variable declarations
    Dim n As Integer, k As Integer, j As integer
'Input given data
    n = Worksheets("myTable").Cells(4,7).Value
'Clear input cells
    Worksheets("myTable").Cells(2,3).Value = ""
    Worksheets("myTable").Cells(2,5).Value = ""
    Worksheets("myTable").Cells(2,7).Value = ""
    Worksheets("myTable").Cells(3,3).Value = ""
    Worksheets("myTable").Cells(3,5).Value = ""
    Worksheets("myTable").Cells(3,7).Value = ""
    Worksheets("myTable").Cells(4,7).Value = ""
'Clear table entries
    For k = 1 To n
        For j = 2 to 7
            Worksheets("myTable").Cells(7+k,j).Value = ""
        Next
    Next
End Sub
```

Figure 24. VBA code for clearing up the table and input cells of the interface of Figure 22.

Combining the *Range* ( ) and *Cells* ( ) properties

In the code of Figures 23 and 24, references of the form: Worksheets `("myTable").Cells(i,j)` are related to the cells in worksheet "myTable" with the row (*i*) and column (*j*) counting starting at cell A1. Figure 25 shows the location of the cell represented by *Cells*(*i,j*) for different values of *i* and *j* for a reference of the form Worksheets `("myTable").Cells(i,j)`.

Column (j) ⟶	1	2	3	4	5	6	7	8
Row (i) ↓	A	B	C	D	E	F	G	H
1	Cells(1,1)	Cells(1,2)	Cells(1,3)	Cells(1,4)	Cells(1,5)	Cells(1,6)	Cells(1,7)	Cells(1,8)
2	Cells(2,1)	Cells(2,2)	Cells(2,3)	Cells(2,4)	Cells(2,5)	Cells(2,6)	Cells(2,7)	Cells(2,8)
3	Cells(3,1)	Cells(3,2)	Cells(3,3)	Cells(3,4)	Cells(3,5)	Cells(3,6)	Cells(3,7)	Cells(3,8)
4	Cells(4,1)	Cells(4,2)	Cells(4,3)	Cells(4,4)	Cells(4,5)	Cells(4,6)	Cells(4,7)	Cells(4,8)
5								

Figure 25. Absolute references for the *Cells* ( ) property in a given worksheet. Counting of rows and columns starts at cell A1.

If you want to refer to cells at row *i* and column *j* in relation to a starting cell other than A1, the reference must include a *Range*( ) property call linked to the cell in the upper left corner of a rectangular domain. This upper left corner will correspond to the row *i*=1 and column *j*=1 of the domain.

For example, in the code of Figure 23, we output data in the table whose upper left corner is cell B8 of the worksheet "myTable" by using the lines of code:

```
Worksheets("myTable").Cells(7+k,2).Value = k
Worksheets("myTable").Cells(7+k,3).Value = t
Worksheets("myTable").Cells(7+k,4).Value = vx
Worksheets("myTable").Cells(7+k,5).Value = vy
```

```
Worksheets("myTable").Cells(7+k,6).Value = x
Worksheets("myTable").Cells(7+k,7).Value = y
```

The first index 7+*k* in the *Cells*( ) property call is needed in this case because the references are with respect to the cell A1. Also, the second index in the *Cells*( ) property call uses the values of 2, 3, …, etc., to refer to columns B, C, … etc. An alternative way to write these lines of code would be to use:

```
Worksheets("myTable").Range("B8").Cells(k,1).Value = k
Worksheets("myTable").Range("B8").Cells(k,2).Value = t
Worksheets("myTable").Range("B8").Cells(k,3).Value = vx
Worksheets("myTable").Range("B8").Cells(k,4).Value = vy
Worksheets("myTable").Range("B8").Cells(k,5).Value = x
Worksheets("myTable").Range("B8").Cells(k,6).Value = y
```

Using the `.Range("B8").` specification in the modified lines of code, shown above, simplifies the use of the indices in the *Cells*( ) property call. Once the upper left corner of the table has been identified ("B8", in this case), the indices counting starts in that cell as i=1 (row) and j=1 (column), as illustrated in Figure 26.

	A	B	C	D	E	F	G
7		Column (j) ⟶	1	2	3	4	5
8		Row (i) ↓	1	Cells(1,2)	Cells(1,3)	Cells(1,4)	Cells(1,5)	Cells(1,6)
9			Cells(2,1)	0.60	4.33	1.52	2.43	3.20
10			Cells(3,1)	0.70	4.33	0.54	2.87	3.30
11			Cells(4,1)	0.80	4.33	-0.44	3.30	3.31
12			5	0.90	4.33	-1.42	3.73	3.22

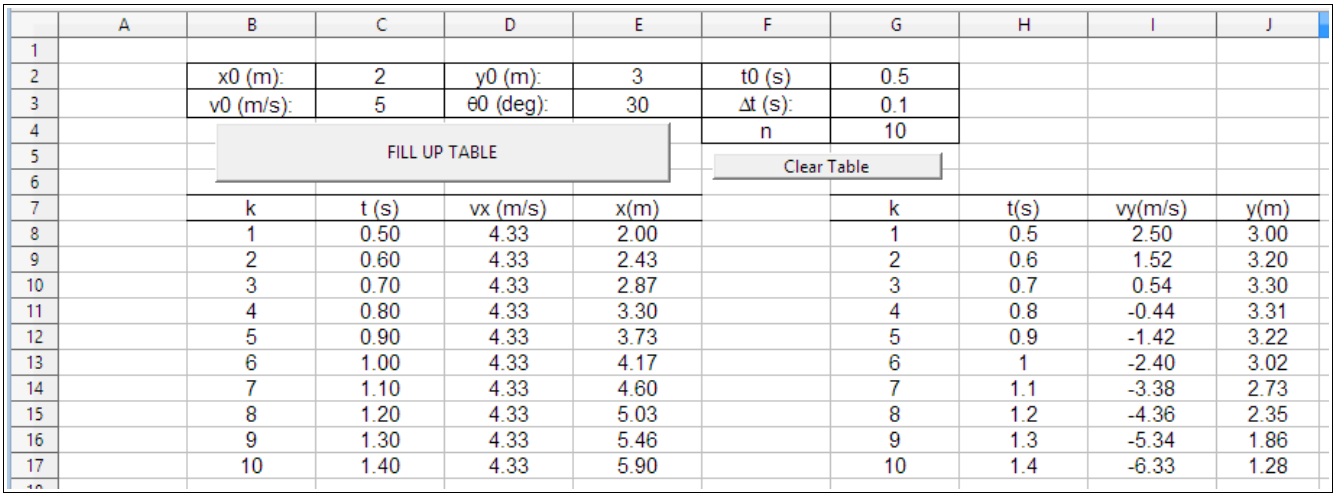
Figure 26. Examples of *Cells*( ) property calls in reference to range B8, rather than to A1, in worksheet "myTable". References to the cells in the rectangular region whose upper-left corner is cell B8 follow the format: `Worksheets("myTable").Range("B8").Cells(i,j).`

Replacing the lines of code, as suggested above, to verify that both sets of lines of codes produce equivalent is left as an exercise for the reader.

NOTE: The creation of this table can be performed without using VBA by simply filling formulas for the equations that calculate the table entries (equations [9] and [10]), and dragging down the formulas until completing the required number of lines in the table. However, the coding of this process could be useful if you have to automate table creation on a routine basis. The programming of the codes of Figures 23 and 24 also allowed us to learn about applications of the *For* ... *Next* loop statements, and about the use of the *Cells*( ) property.

Using Range().Cells() *to fill up two tables in the same worksheet*

In this example we use the combination of properties *Range*( ).*Cells*( ) to fill up two tables with the projectile motion data rather than a single one. Figure 27, below, shows the *Excel* interface for this case. The worksheet name was changed from *myTable* to *TwoTables*. The first table contains columns for the index k, the time t, the x-component of velocity v_x, and position x. The second table, on the other hand, contains columns for the index k, the time t, the y-component of velocity v_y, and the position y.



	x0 (m):	2	y0 (m):	3	t0 (s)	0.5
	v0 (m/s):	5	θ0 (deg):	30	Δt (s):	0.1
					n	10

FILL UP TABLE

Clear Table

k	t (s)	vx (m/s)	x(m)
1	0.50	4.33	2.00
2	0.60	4.33	2.43
3	0.70	4.33	2.87
4	0.80	4.33	3.30
5	0.90	4.33	3.73
6	1.00	4.33	4.17
7	1.10	4.33	4.60
8	1.20	4.33	5.03
9	1.30	4.33	5.46
10	1.40	4.33	5.90

k	t(s)	vy(m/s)	y(m)
1	0.5	2.50	3.00
2	0.6	1.52	3.20
3	0.7	0.54	3.30
4	0.8	-0.44	3.31
5	0.9	-1.42	3.22
6	1	-2.40	3.02
7	1.1	-3.38	2.73
8	1.2	-4.36	2.35
9	1.3	-5.34	1.86
10	1.4	-6.33	1.28

Figure 27. *Excel* worksheet "Two Tables" used as interface for a program that fills up two tables. This interface is similar to that of Figure 22, but producing two tables, rather than one.

The code used to produce the output of Figure 27 is similar to the code in Figure 23, except that the lines

```
Worksheets("myTable").Cells(7+k,2).Value = k
Worksheets("myTable").Cells(7+k,3).Value = t
Worksheets("myTable").Cells(7+k,4).Value = vx
Worksheets("myTable").Cells(7+k,5).Value = vy
Worksheets("myTable").Cells(7+k,7).Value = y
```

in the code of Figure 23, have been replaced by the following lines of code to produce the output of Figure 27:

```
'Fill up table for x values
Worksheets("TwoTables").Range("B8").Cells(k,1).Value = k
Worksheets("TwoTables").Range("B8").Cells(k,2).Value = t
Worksheets("TwoTables").Range("B8").Cells(k,3).Value = vx
Worksheets("TwoTables").Range("B8").Cells(k,4).Value = x
'Fill up table for y values
Worksheets("TwoTables").Range("G8").Cells(k,1).Value = k
Worksheets("TwoTables").Range("G8").Cells(k,2).Value = t
Worksheets("TwoTables").Range("G8").Cells(k,3).Value = vy
Worksheets("TwoTables").Range("G8").Cells(k,4).Value = y
```

Notice the use of the combination of properties *Range*( ).*Cells*( ) to fill up the two different tables whose upper left corners are located at cells "B8" and "G8", respectively, for the tables of the *x* and *y* properties, as shown in Figure 27.

The code associated with the [Clear Table] button in Figure 27 is similar to that shown in Figure 24, except that the two nested *For* ...*Next* statements in Figure 24 get replaced by the following nested loops:

```
For k = 1 To n
    For j = 1 to 4
        Worksheets("TwoTables").Range("B8").Cells(k,j).Value = ""
        Worksheets("TwoTables").Range("G8").Cells(k,j).Value = ""
    Next
Next
```

Filling up tables in different worksheets

Figure 28 shows three worksheets, named "Two Tables", "Table1", and "Table2", in a workbook. The worksheet "TwoTables" acts as a control interface with input data (*x0, y0,* etc.) and two buttons. This interface basically produces the same result as that of Figure 27, except that the tables are now filled up in different worksheets: the *x*-component data are filled up in worksheet "Table1", while the *y*-component data are filled up in worksheet "Table2".

(a)

	B	C	D	E	F	G
2	x0 (m):	2	y0 (m):	3	t0 (s)	0.5
3	v0 (m/s):	5	θ0 (deg):	30	Δt (s):	0.1
4					n	10

FILL UP TABLE

Clear Table

See worksheets "Table1" and "Table2" after pressing the [FILL UP TABLE] button

TwoTables | Table1 | Table2

(b)

	B	C	D	E
1	Table 1: x-components of velocity and position			
2	k	t (s)	vx (m/s)	x(m)
3	1	0.50	4.33	2.00
4	2	0.60	4.33	2.43
5	3	0.70	4.33	2.87
6	4	0.80	4.33	3.30
7	5	0.90	4.33	3.73
8	6	1.00	4.33	4.17
9	7	1.10	4.33	4.60
10	8	1.20	4.33	5.03
11	9	1.30	4.33	5.46
12	10	1.40	4.33	5.90

les | Table1 | Table2

(c)

	B	C	D	E
1	Table 2: y components of velocity and position			
2	k	t (s)	vy (m/s)	y(m)
3	1	0.50	2.50	3.00
4	2	0.60	1.52	3.20
5	3	0.70	0.54	3.30
6	4	0.80	-0.44	3.31
7	5	0.90	-1.42	3.22
8	6	1.00	-2.40	3.02
9	7	1.10	-3.38	2.73
10	8	1.20	-4.36	2.35
11	9	1.30	-5.34	1.86
12	10	1.40	-6.33	1.28

les | Table1 | Table2

Figure 28. Worksheets "TwoTables", "Table1", and "Table2" in a workbook.

The code associated with button [FILL UP TABLE] is shown in Figure 29, below. This code is very similar to that in Figure 23, except for the references to *Worksheets*("Table1") and *Worksheets*("Table2"), as well as the use of the *Range*("B3") property to set the upper left corner of the two tables. The *Cells*( ) property then is used to fill out the individual data cells. After pressing the [FILL UP TABLE] button, the tables will be available in the worksheets "Table1" and "Table2".

```
Sub FillTable
'Variable declarations
    Dim x0 As Double, y0 As Double, x As Double, y As Double
    Dim v0 As Double, theta0 As Double, vx As Double, vy As Double
    Dim t0 As Double, Dt As Double, t As Double
    Dim n As Integer, k As Integer
'Constants
    CONST DegToRad = 1.7453E-2
    CONST g = 9.806
'Input given data
    x0 = Worksheets("TwoTables").Cells(2,3).Value
    y0 = Worksheets("TwoTables").Cells(2,5).Value
    t0 = Worksheets("TwoTables").Cells(2,7).Value
    v0 = Worksheets("TwoTables").Cells(3,3).Value
    theta0 = Worksheets("TwoTables").Cells(3,5).Value*DegToRad
    Dt = Worksheets("TwoTables").Cells(3,7).Value
    n = Worksheets("TwoTables").Cells(4,7).Value
'calculate values and write tables
    For k = 1 To n
        t = t0 + (k-1)*Dt
        vx = v0*cos(theta0)
        vy = v0*sin(theta0)-g*(t-t0)
        x = x0 + v0*cos(theta0)*(t-t0)
        y = y0 + v0*sin(theta0)*(t-t0) - 0.5*g*(t-t0)^2
        'Fill up table for x values (Table1)
        Worksheets("Table1").Range("B3").Cells(k,1).Value = k
        Worksheets("Table1").Range("B3").Cells(k,2).Value = t
        Worksheets("Table1").Range("B3").Cells(k,3).Value = vx
        Worksheets("Table1").Range("B3").Cells(k,4).Value = x
        'Fill up table for y values (Table 2)
        Worksheets("Table2").Range("B3").Cells(k,1).Value = k
        Worksheets("Table2").Range("B3").Cells(k,2).Value = t
        Worksheets("Table2").Range("B3").Cells(k,3).Value = vy
        Worksheets("Table2").Range("B3").Cells(k,4).Value = y
    Next
End Sub
```

Figure 29. Code for filling up tables in worksheets “Table1” and “Table2” activated with the button [FILL UP TABLE] in worksheet “TwoTables” (see Figure 28).

The code associated with the [Clear Table] button is shown next:

```
Sub ClearTheTable
'Variable declarations
    Dim n As Integer, k As Integer, j As integer
'Input given data
    n = Worksheets("TwoTables").Cells(4,7).Value
'Clear input cells
    Worksheets("TwoTables").Cells(2,3).Value = ""
    Worksheets("TwoTables").Cells(2,5).Value = ""
    Worksheets("TwoTables").Cells(2,7).Value = ""
    Worksheets("TwoTables").Cells(3,3).Value = ""
    Worksheets("TwoTables").Cells(3,5).Value = ""
    Worksheets("TwoTables").Cells(3,7).Value = ""
    Worksheets("TwoTables").Cells(4,7).Value = ""
```

```
'Clear table entries
    For k = 1 To n
        For j = 1 to 4
            Worksheets("Table1").Range("B3").Cells(k,j).Value = ""
            Worksheets("Table2").Range("B3").Cells(k,j).Value = ""
        Next
    Next
End Sub
```

Figure 30. Code for clearing up tables in worksheets "Table1" and "Table2" activated with the button [Clear Table] in worksheet "TwoTables" (see Figure 28).

Solution to the differential equation *yd*/*dx* = *f*(*x*,*y*) – The Euler method

A numerical application that requires filling up a table with data is the solution to the first-order ordinary differential equation (ODE) of the form:

$$\frac{dy}{dx} = f(x, y) \quad , \qquad [11]$$

subject to the initial condition $y(x_0) = y_0$. The Euler method consists in replacing the derivative *dy/dx* with the approximation:

$$\frac{dy}{dx} = \frac{y_{k+1} - y_k}{\Delta x} \quad , \qquad [12]$$

where Δx is an increment on the independent variable *x*. The solution consists in producing ordered pairs (x_k, y_k), starting with the initial condition (x_0, y_0), then increasing the values of *x* systematically, by using $x_1 = x_0 + \Delta x$, $x_2 = x_1 + \Delta x = x_0 + 2 \cdot \Delta x$, …,

$$x_k = x_0 + k \cdot \Delta x \quad , \quad \text{for } k = 0, 1, \ldots, n \qquad [13]$$

while calculating the values of *y* from equation [12], i.e.,

$$y_{k+1} = y_k + f(x_k, y_k) \quad , \text{for } k = 0, 1, \ldots, n\text{-}1. \qquad [14]$$

In summary, we calculate x_k and y_k using equations [13] and [14], starting from the initial condition (x_0, y_0), with the index *k* taking values $k = 0, 1, 2, \ldots, n$.

To proceed with the solution, worksheet *ODE1* (for *O*rdinary *D*ifferential *E*quation of the *1*-st order) is used as an interface as shown in Figure 31.

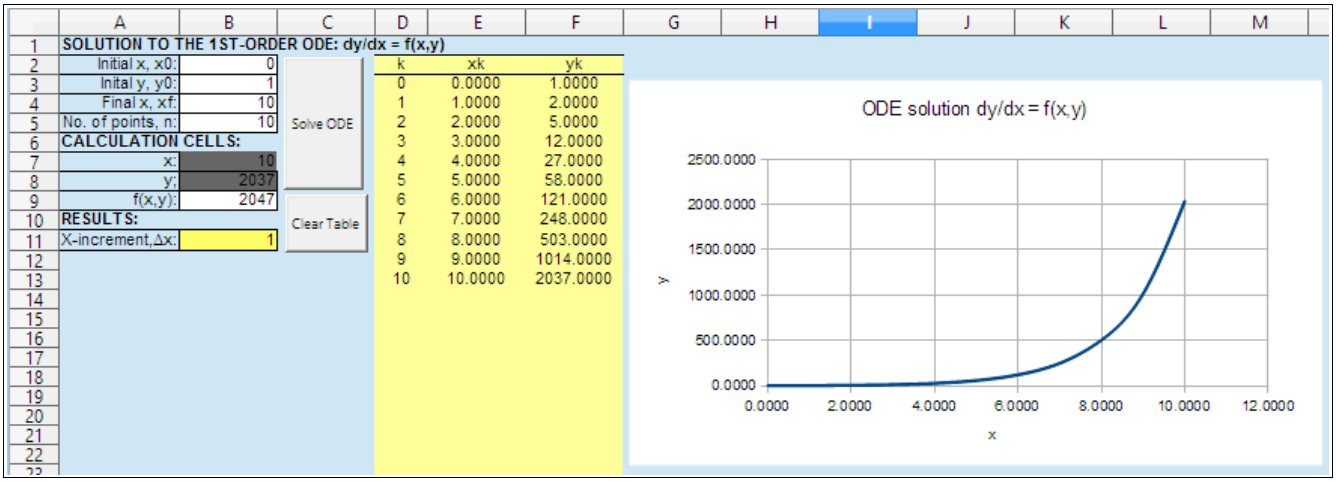


Figure 31. *Excel* interface for solving the ODE $dy/dx = f(x,y)$ subject to the initial condition $y_0 = y(x_0)$ using the Euler method.

The user provides the values of the initial condition (x_0,y_0) in cells B2 and B3. The solution is calculated in the domain $x_0 \le x \le x_f$, thus, the user enters also the value of x_f in cell B4, as well as the number of subintervals n into which the domain $[x_0,x_f]$ is to be divided. The value of n is entered into cell *B5*.

Cells *B7* and *B8* need to be defined as variables x and y using the option *Data* > *Define Range*, so that the definition of $f(x,y)$ in cell B9 can be performed using expressions such as "= x+y" or "= x*sin(y)", etc. [Alternatively, if the user does not want to use the *Data* > *Define Range* option to define variables x and y, the function definition in cell B9 can be given in terms of the cell references B7 for x and B8 for y, e.g., "= B7+B8" or "=B7*SIN(B8)", etc.] Cells B7, B8, and B9 will change during the program execution as new values of x, y, and $f(x,y)$ are calculated. The user only needs to enter the expression for $f(x,y)$ in cell B9.

The program that solves the ODE using the Euler method is activated by pressing the [Solve ODE] button. The program takes care of handling the values of cells B7 and B8. Cell B11 will show the value of $\Delta x = (x_f - x_0)/n$. The program then takes care of filling up the table corresponding to columns D, E, and F. These columns show the values of the index k, and the x and y coordinates of the solution.

For the solution in Figure 31 the value of $f(x,y)$ used is $f(x,y) = x+y$, i.e., we solve the ODE

$$\frac{dy}{dx} = x + y \quad ,$$

with initial condition $y(0) = 1$, in the x interval = [0,10], using n = 10 subintervals. Figure 31 shows that the subinterval size is Δx = 1.0.

The code associated with button [Solve ODE] is shown next.

```
Sub SolveODE1
'Definition of variables
   Dim x As Double, y As Double
   Dim xf As Double, Dx as Double, ff As Double
   Dim k As Integer, n As Integer
'Data input
   x = Worksheets("ODE1").Range("B2").Value
   y = Worksheets("ODE1").Range("B3").Value
   xf = Worksheets("ODE1").Range("B4").Value
   n = Worksheets("ODE1").Range("B5").Value
'calculation of x increment
   Dx = (xf-x)/n
   Worksheets("ODE1").Range("B11").Value = Dx
'Iterative process to calculate ODE numerical solution
   For k = 0 to n
       Worksheets("ODE1").Range("D3").Cells(k+1,1).Value = k
       Worksheets("ODE1").Range("D3").Cells(k+1,2).Value = x
       Worksheets("ODE1").Range("D3").Cells(k+1,3).Value = y
       Worksheets("ODE1").Range("B7").Value = x
       Worksheets("ODE1").Range("B8").Value = y
       ff = Worksheets("ODE1").Range("B9").Value
       x = x + Dx
       y = y + ff*Dx
   Next
End Sub
```

Figure 32. VBA code for calculating the numerical solution of the ODE $dy/dx = f(x,y)$ using the Euler method. This code is associated with the [Solve ODE] button of Figure 31.

In the code of Figure 32, the value of x_0 is entered as x, and that of y_0 is entered as y. Thus, the value of the x increment, referred to as Dx in code, is calculated as $Dx = (xf - x)/n$, and placed in cell B11. The iterative process is contained in the *For ... Next* loop statement. The *For ... Next* loop places the current values of the index k, and the solution x and y, into the table, as well as placing the current values of x and y into cells B7 and B8, respectively. The values in cells B7 and B8 are used to calculate ff which in the interface contains the current value of $f(x,y)$. Then, new values of x and y are calculated using $x = x + Dx$, and $y = y + ff*Dx$, respectively.

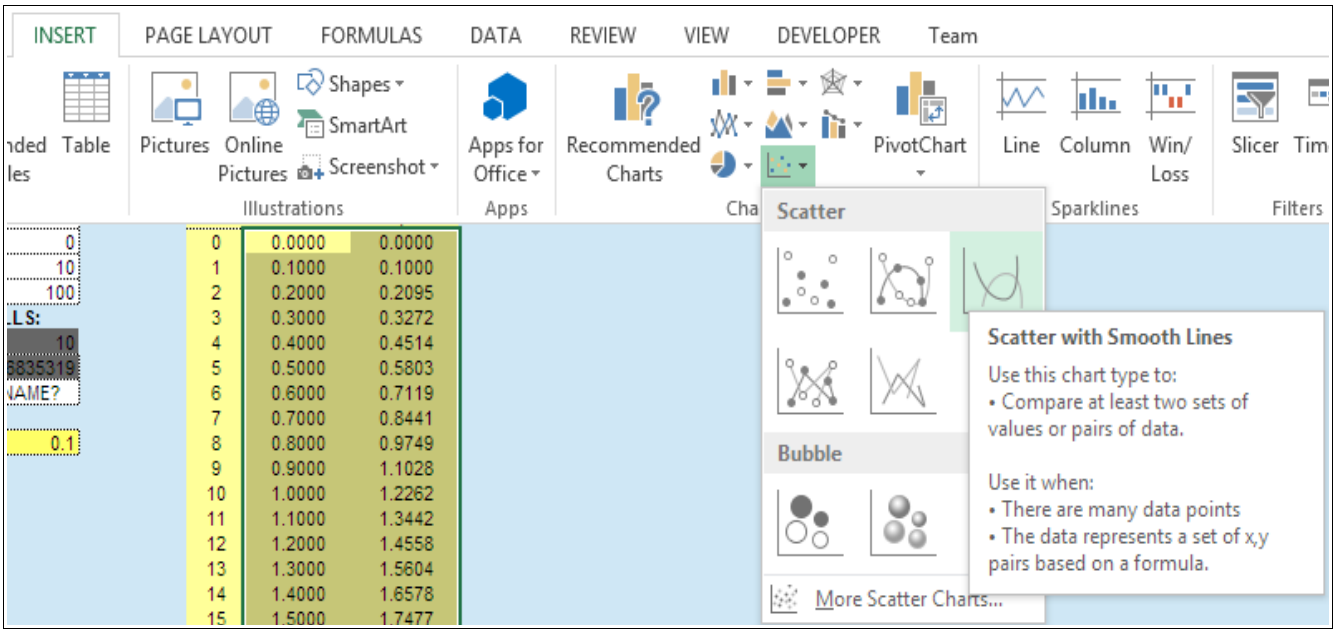
To clear the table, the following code is associated with the [Clear Table] button:

```
Sub ClearTable
    Dim n As Integer, k As Integer, j As Integer
    n = Worksheets("ODE1").Range("B5").Value
    For k = 1 to 4
        Worksheets("ODE1").Range("B2").Cells(k,1).Value = ""
    Next
    For k = 1 to 2
        Worksheets("ODE1").Range("B7").Cells(k,1).Value = ""
    Next
    Worksheets("ODE1").Range("B11").Value = ""
    For k = 1 To n+1
        For j = 1 To 3
            Worksheets("ODE1").Range("D3").Cells(k,j).Value = ""
        Next
    Next
End Sub
```

Figure 33. VBA code for clearing data and table in Figure 31. This code is associated with the [Clear Table] button in Figure 31.

To create the graph, proceed as follows:

- Run the program for any values of x_0, y_0, x_f, and $f(x,y)$ [say, "=x+y"], with n = 1000.
- Next, select the data in the range E3:F1003, and select the option *Insert* > *Scatter* > *Scatter with Smooth Lines*, as shown below. This will produce the required graph.



- Press the [Clear Table] button to clear the interface.

The chart object should be now ready to plot any ODE solution found. Try the solution of Figure 31. The Euler method solution improves as the value of the x increment Δx gets smaller and smaller. For example, if we use the same data of Figure 31, but using n = 1000, the solution does not grow as large as in the case of Figure 31. The solution for n = 1000 is shown in Figure 34, below.

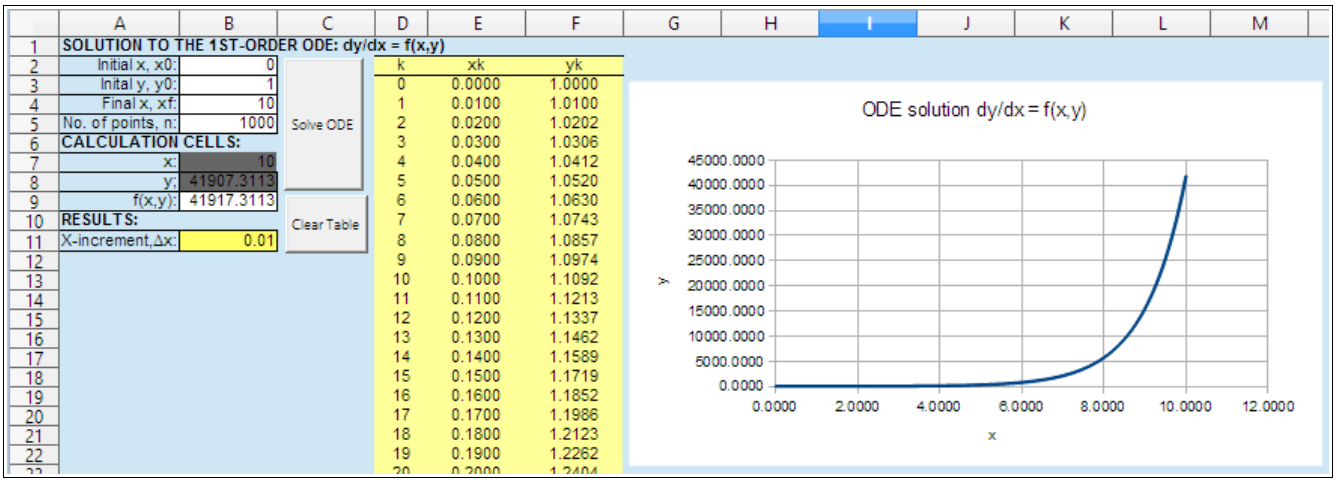


Figure 34. *Excel* interface for solving the ODE $dy/dx = f(x,y)$ subject to the initial condition $y_0 = y(x_0)$ using the Euler method. This solution uses the same parameters of Figure 31, except for n = 1000.

Notice, for example, that the solution of Figure 34 shows y close to 5000 for x = 8 (the table actually shows y = 5729.66 for x = 8), while that of Figure 31 shows y = 503 for x = 8. The actual solution to the ODE $dy/dx = x+y$, with initial conditions $y(0) = 1$, is $y=2\cdot e^{x}-x-1$, which gives y = 5952.92 for x = 8. Thus, the numerical solution for n = 1000 obtained in *Excel* would be closer to the actual solution than the numerical solution for n = 10. These observations point out to the fact that the Euler method is not very reliable in producing accurate numerical solutions unless the value of the x increment, Δx, is relatively small. Other methods, such as the Runge-Kutta methods of orders 2 or 4, would produce better results.

Vectors and One-Dimensional Arrays

In Physics, a vector is typically a mathematical object identified by 2 or 3 components and used to represents quantities that require a magnitude and a direction in space, e.g., position, velocity, acceleration, force, etc. Thus, you may be familiar with vectors such as **F** = (3**i**-5**j**)N, or **v** = (3.2**i**+5.2**j**-1.2**k**) m/s, etc. In these physical vectors, the quantities **i**, **j**, and **k** represent unit vectors in the directions of the x, y, and z axis.

For the purpose of programming, however, we will use the term *vector* to refer to any array of numbers that can be represented as a single row or column of data in a *Excel* worksheet. The number of elements in the array can vary be 2, 3, or any number. The following figure shows examples of row and column vectors of data as would be written in paper:

Examples of *row* vectors:

$$u=(-2\ \ 1\ \ -3\ \ 4)\qquad r=(-2\ \ -1\ \ 0\ \ 0.5\ \ 2)\qquad s=\left(\pi\ \ \frac{\pi}{2}\ \ -\pi\ \ -\frac{\pi}{2}\ \ \frac{3\cdot\pi}{2}\ \ -\frac{3\cdot\pi}{2}\right)$$

Examples of *column* vectors:

$$v=\begin{pmatrix}5\\-2\\0\\3\end{pmatrix}\qquad t=\begin{pmatrix}-2\\-\pi\\0\\5\\3\end{pmatrix}\qquad w=\begin{pmatrix}0\\0.1\\0.2\\0.3\\0.4\\0.5\end{pmatrix}$$

<u>*One-dimensional Arrays in VBA*</u>

In VBA, rows or column vectors are represented as *one-dimensional arrays*. From the point of view of computer applications, an array is a linear or rectangular collection of data memory locations identified by same name and one or two indices. For the representation of vectors, we use arrays with a single index, or one-dimensional arrays. Thus, for example, to represent the vector **X** = (2, 3, 5, -1), we would need memory locations that could be represented as in the following diagram:

X(1)	X(2)	X(3)	X(4)
2	3	5	-1

Thus, while the array name in this case is X, individual members of the array are identified as X(1), X(2), etc. The term between parentheses is the *index* of the one-dimensional array. This notation will be equivalent to referring to the elements of the (mathematical) vector **X** = (2, 3, 5, -1) as $X_1 = 2$, $X_2 = 3$, etc. Thus, sometimes, we would refer to the array index as a *subindex*.

The one-dimensional array X, shown above, can be defined in VBA by using a *Dim* statement such as:

```
Dim X(1 To 4) As Double
```

The expression between parentheses in this declaration identifies the indices to be used with array X as the values 1, 2, 3, and 4, i.e., 1 *To* 4, in increments of 1. The data type for array X is given as *Double,* in this example, but it can be any of the data types available in VBA (*Integer, Long, String, Boolean,* etc.). In general, thus, a one-dimensional array in VBA can be declared using a statement of the form:

```
Dim <array_name>(<lower_index> To <upper_index>) As <data_type>
```

The <array_name> must follow the rules of variable names, i.e., start with a letter, include letters and numbers and the undersign (_), and be no longer than 256 characters. The <lower_index> and <upper_index> can be any integer number, negative, zero, or positive, but the <lower_index> must be lower than the <upper_index>. The <data_type> can be any of the allowed data types in VBA: *Integer, Long, Single, Double, String, Boolean*, etc.

Assigning Values to Elements of a One-dimensional Array

The individual elements of an array can be given values as you would any other variable, for example, using `X(1) = 10 : X(2) = 5 : X(3) = -2`, etc. Values can also be assigned based on an index in a *For ... Next* loop, e.g.,

```
Dim X(1 To 10) As Double, Y(1 To 10) As Double
For k = 1 To 10
       X(k) = 2*k+1
       Y(k) = k^2
Next
```

In some of the examples presented earlier we used expressions that require subindices, for example:

$$x_k = a + (k\text{-}1)\,\Delta x,\ k = 1, 2, \ldots, n$$

This expression can be implemented in code by using:

```
Dim x(1 To 10) As Double, a As Double, Dx As Double
Dim k As Integer, n As Integer
n = InputBox("Enter n (2 < n < 10):")
a = InputBox("Enter a:")
Dx = InputBox("Enter Dx (>0):")
For k = 1 To n
    x(k) = a + (k-1) * Dx
Next
```

Using the Option Base *specification*

Some calculations use vectors where the indices always start as 0 or 1. Thus, a generic vector **X**, of length *n*, whose indices always start with 1, would be expressed as $\mathbf{X} = [X_1, X_2, \ldots, X_n]$. Whereas, a generic vector **Y**, of length *n*, whose indices always start with 0, would be expressed as $\mathbf{Y} = [Y_0, Y_1, \ldots, X_{n+1}]$. If all the vectors (one-dimensional arrays) in use in a particular program will have indices starting, say, with 1, you can use the line:

```
Option Base 1
```

to ensure that the <lower_limit> in the indices of this array is exactly 1. These line is placed above all *Sub*s and *Function*s in the code, at the same location that you would find the *Option Explicit* declaration.

If the *Option Base 1* or the *Option Base 0* declaration is present, any one-dimensional array can be declared using a single value between parentheses, the <upper_index>, which should be a positive number. Thus, the declarations

```
Dim X(5) As Double, K(10) As Integer
```

together with the *Option Base 1* declaration, will create vectors X (double precision) and Y (integer) with elements X(1), X(2), …, X(5), and Y(1), Y(2), …, Y(10). On the other hand, if the declaration *Option Base 0* is active, the declarations

```
Dim X(5) As Double, K(10) As Integer
```

will create vectors X (double precision) and Y (integer) with elements X(0), X(1), …, X(5), and Y(0), Y(1), …, Y(10).

Thus, one-dimensional arrays declared with a single (positive integer) index specification, say *n*, in the *Dim* statement, when one of the declarations *Option Base 1* or *Option Base 0* is active, will have their indices ranging from 1 or 0, respectively, up to *n*. You can still declare one-dimensional arrays with indices starting at values other than 1 or 0 by using the full specification given earlier, namely,

```
Dim <array_name>(<lower_index> To <upper_index>) As <data_type>
```

Thus, assuming that the *Option Base 1* specification is active, the following *Dim* statements

```
Dim X(5) As Double, K(10) As Integer, Z(-2 To 5) As Double
```

will produce the vectors X(double), Y(integer), Z(Double) with elements: X(1), X(2), …, X(5); Y(1), Y(2), …, Y(10); and, Z(-2), Z(-1), …, Z(5).

Generic Array Data Input From a Worksheet – calculating Selected Statistics of a Sample

For the purpose of illustrating the input of data into a one-dimensional array we assume that the data will be available in a column, and that an empty space will be placed at the bottom of the data values, as illustrated in column B of Figure 35.

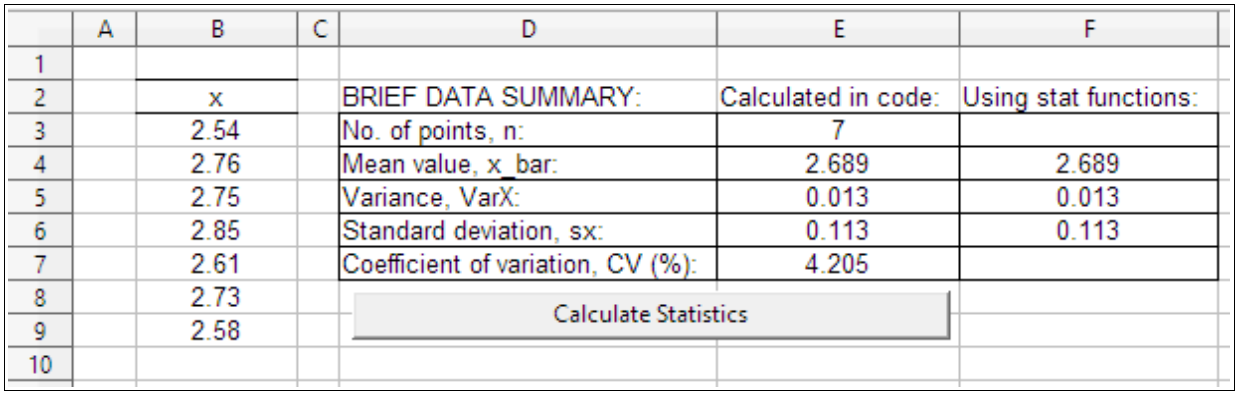


Figure 35. Interface for statistics calculations.

The code will be designed so as to accumulate an index *n* by adding a value of 1 every time a new data value is read from column B in the interface of Figure 35. Thus, as *n* takes values of $n = 1, 2, \ldots$, etc., we will be reading and storing in memory the values of x(1), x(2), …, etc. Furthermore, when we find an empty space at the end of the data set, we will stop reading data, and the current value of *n* becomes the number of data points in the data set.

We will then, have in memory, a collection of numbers $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, i.e., a vector of size *n*. Suppose that this vector of values **x** represent *n* measurements of a certain quantity, say, the diameter (in *mm*) of bearing balls produced by a machine. Then, we refer to the vector **x** as a *sample* of the ball diameters, and *n* is referred to as the *sample size*. In the processing of reading the sample data, as described in the previous paragraph, we have already calculated the sample size, *n*.

The data in the sample $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ can be summarized by calculating certain *sample statistics*, such as the *mean* ($\bar{x}$), the *variance* (*VarX*), and the *standard deviation* (s_x). These quantities are defined as follows:

$$\bar{x} = \frac{1}{n}\sum_{k=1}^{n} x_k \quad , \qquad [15]$$

$$VarX = \frac{1}{n-1}\sum_{k=1}^{n} (x_k - \bar{x})^2 \quad , \qquad [16]$$

and

$$s_x = \sqrt{VarX} \quad . \qquad [17]$$

Finally, the *coefficient of variation* is defined as:

$$C.V. = \frac{s_x}{\bar{x}} \cdot 100 \quad \%. \qquad [18]$$

The mean (Eq. [15]) is calculated by accumulating the sum of the individual elements in the sample, and then dividing by the sample size. The variance (Eq. [16]) requires us to calculate the difference

between each individual value and the mean, square that difference, and accumulate it into a summation, which is, then, divided by (n-1). The standard deviation (Eq. [17]) is just the square root of the variance, and the coefficient of variation is the ratio of the standard deviation to the mean, expressed as a percentage. All these operations can be coded in VBA, and the results then shown in the interface as illustrated in Figure 35.

The code that reads the data, obtains the sample size, and calculates the mean, variance, standard deviation, and coefficient of variation for the data in the interface of Figure 35 is shown in Figure 36, including the *Option* declarations.

```
Option Explicit
Option Base 1

Sub myStats
'Variable declaration
    Dim X(100) As Double, xbar As Double
    Dim VarX As Double, sx As Double, CV As Double
    Dim k As Integer, n As Integer
    Dim SUMX As Double 'Stores summations
'Input data, and find sample size (n):
    n = 1  'Initialize sample size, n
    SUMX = 0 'Initialize sum of X, SUMX
    Do While Worksheets("STATS").Range("B3").Cells(n,1).Value <> ""
        x(n) = Worksheets("STATS").Range("B3").Cells(n,1).Value
        SUMX = SUMX + x(n)
        n = n + 1
    Loop
    n = n - 1   'Value of n adjusted after leaving loop
'calculate mean value
    xbar = SUMX/n
'calculate variance (VarX), standard deviation (sx), coeff. var (CV)
    SUMX = 0 'Re-initialize SUMX to store sum of squared deviations
    For k = 1 To n
        SUMX = SUMX + (x(k)-xbar)^2
    Next
    VarX = SUMX/(n-1)
    sx = Sqr(VarX)
    CV = sx/xbar*100
 'Output results to interface by using the x array:
    x(1) = n : x(2) = xbar : x(3) = VarX : x(4) = sx : x(5) = CV
    For k = 1 to 5
        Worksheets("STATS").Range("E3").Cells(k,1).Value = x(k)
    Next k
End Sub
```

Figure 36. Code for entering data and calculating statistics for the interface of Figure 35.

The code of Figure 36 gets associated with the [calculate Statistics] button. The buttons [Clear Output] and [Clear Data & Output] are associated with the macros of Figures 37 and 38, respectively.

```
Sub ClearOutput
    Dim k As Integer
    For k = 1 to 5
        Worksheets("STATS").Range("E3").Cells(k,1).Value = ""
    Next
End Sub
```

Figure 37. VBA Code associated with the [Clear Output] button.

```
Sub ClearDataAndOutput
    Dim k As Integer
    Call ClearOutput
    For k = 1 To 1000
        Worksheets("STATS").Range("B3").Cells(k,1).Value = ""
    Next
End Sub
```

Figure 38. VBA Code associated with the [Clear Data & Output] button.

The [Clear Output] button clears only the statistics results in the interface of Figure 35, whereas the [Clear Data & Output] button clears both the statistics results as well as the original data in the interface.

Statistics of data in a *Excel* worksheet can be calculated directly by using functions under the *Statistics Function* collection available through the [*fx*] button in the worksheet. For example, cells F4, F5, and F6, contain the following formulas for calculating the mean ("=AVERAGE(B3:B9)"), the variance ("=VAR(B3:B9)"), and the standard deviation ("=STDEV(B3:B9)"). When you change the number of data points whose statistics need to be calculated, you need to change, by hand, the range of values to which functions AVERAGE, VAR, and STDEV are applied. These functions were included in the interface of Figure 35 to verify that the values calculated with the code are the same as those calculated with the worksheet functions.

Obtaining the Maximum and Minimum Values of a Vector

In the example of Figure 35 we learned how to enter data from a column into a vector. We are going to use the same procedure to enter data, but we will be calculating the minimum and maximum values of the sample. The interface used is very similar to that of Figure 35, and is presented in Figure 39.

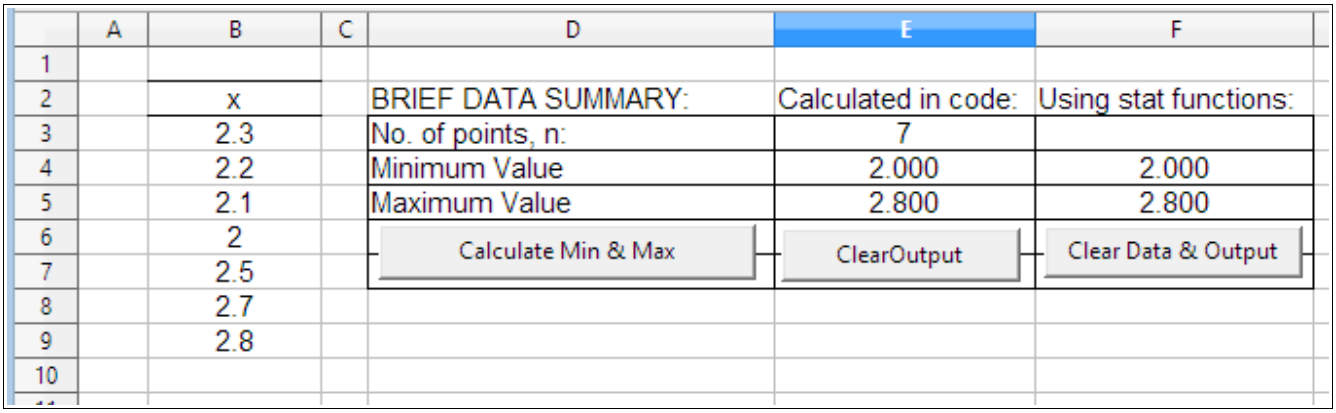


Figure 39. Interface for entering data and obtaining min and max values for the *x* data shown.

Obtaining the maximum value consists in initializing a variable, say, *xMax*, as an exaggeratedly small value, say, *xMax* = -1e100. Then, through *For ... Next* loop, values of the vector, say, *x*(k), are compared to the current value of *xMax*. If *x*(k) > *xMax* then *xMax* is replaced by *x*(k). After comparing *xMax* with each value of *x*(k), the maximum value in vector *x* will be stored in *xMax*. To obtain the minimum value we follow as similar procedure, assigning a variable *xMin* an insanely large value, e.g., *xMin* = 1e100, then, in the *For ... Next* loop, we check whether *x*(k) < *xMin*. If that is the case then make *xMin* = *x*(k), and continue the loop until exhausting the values of k.

The macros associated with the buttons [calculate Min & Max], [Clear Output], and [Clear Data & Output] are shown as Subs *MinMax, ClearOutput*, and *ClearDataAndOutput* in the following code:

```
Option Explicit
Option Base 1

Sub MinMax
'Variable declaration
    Dim X(100) As Double, xMin As Double, xMax As Double
    Dim k As Integer, n As Integer
'Set impossible values of minimum and maximum values
    xMin = 1e100 : xMax = -1e100
'Input data, and find sample size (n):
    n = 1  'Initialize sample size, n
    Do While Worksheets("MinMax").Range("B3").Cells(n,1).Value <> ""
        x(n) = Worksheets("MinMax").Range("B3").Cells(n,1).Value
        n = n + 1
    Loop
    n = n - 1   'Value of n adjusted after leaving loop
'Obtain minimum and maximum values
    For k = 1 To n
        If x(k) < xMin Then xMin = x(k)
        If x(k) > xMax Then xMax = x(k)
    Next k
 'Output results to interface by using the x array:
    x(1) = n : x(2) = xMin : x(3) = xMax
    For k = 1 to 3
        Worksheets("MinMax").Range("E3").Cells(k,1).Value = x(k)
    Next k
End Sub
Sub ClearOutput
    Dim k As Integer
    For k = 1 to 3
        Worksheets("MinMax").Range("E3").Cells(k,1).Value = ""
    Next
End Sub
Sub ClearDataAndOutput
    Dim k As Integer
    Call ClearOutput
    For k = 1 To 1000
        Worksheets("MinMax").Range("B3").Cells(k,1).Value = ""
    Next
End Sub
```

Figure 40. VBA code for the buttons of the interface in Figure 39.

Excel provides worksheets functions *MIN* and *MAX* to obtain the minimum and the maximum values, respectively, of an array. For example, in the interface of Figure 39 we used the following formulas in cells F4 and F5, respectively, "=MIN(B3:B9)" and "=MAX(B3:B9)", to obtain the minimum and maximum values of the vector *x* in the range B3:B9.

Sorting a vector in increasing order

Suppose that you have entered a vector x of size *n*, i.e., you have the values (x(1), x(2), …, x(n)) stored in memory. To order this vector in increasing order we proceed as follows:

- Use an external *For ... Next* loop with an index *i* varying from 1 to *n*-1
- Use an internal *For ... Next* loop with an index *j* varying from *i*+1 to *n*
- Compare x(*i*) with *x*(*j*). If *x*(*j*) < *x*(*i*), then exchange the values of those two elements, by using a temporary storage variable, say, *temp*. At this point, the code needs to do the following:
  - *temp* ← x(*j*) : x(*j*) ← x(*i*) : x(*i*) ← *temp*
- Complete the running of the index *j* for the current value of *i*
- Complete the running of the index *I*

Figure 41 is an interface for entering the original data and showing the data sorted in both increasing and decreasing order. Figure 42 shows the macros associated with the buttons [Sort Data], [Clear Output] and [Clear Data & Output] as the macros *SortData*, *ClearOutput*, and *ClearDataAndOutput*, respectively.

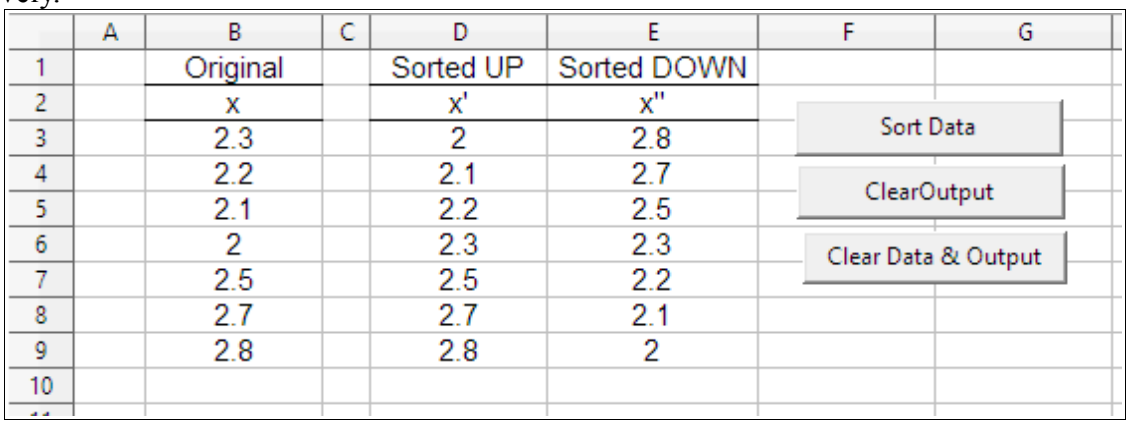


Figure 41. *Excel* interface for sorting a data vector.

```
Option Explicit
Option Base 1

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

Sub ClearOutput
    Dim i As Integer, j As Integer
    For i = 1 to 1000
        For j = 1 to 2
            Worksheets("Sorting").Range("D3").Cells(i,j).Value = ""
        Next
    Next
End Sub

Sub ClearDataAndOutput
    Dim i As Integer
    Call ClearOutput
    For i = 1 To 1000
        Worksheets("Sorting").Range("B3").Cells(i,1).Value = ""
    Next
End Sub
```

Figure 42. VBA code for the buttons of the interface in Figure 41.

Calculating the Median of a Sample – Use of the *MOD* function

Suppose you are given a sample x of size n with elements ($x_1, x_2, \ldots, x_n$). Suppose that we sort the vector x in ascending order so that we end up with vector x' = ($x'_1, x'_2, \ldots, x'_n$). The median x'_m is the value that splits the data so that half of the data is below x'_m and half is above it. Thus, if the vector x' has an odd number n of elements (say, n = 3, 5, 7, etc.), then the median is the data value located at position $\frac{n+1}{2}$, i.e., $x'_m = x'_{\frac{n+1}{2}}$. If, on the other hand, the vector x' has an even number n of elements (say, n = 4, 6, 8, etc.), the median x'_m will be the average of the data points located at positions $\frac{n}{2}$ and $\frac{n}{2}+1$, i.e., $x'_m = \frac{1}{2}\cdot\left(x'_{\frac{n}{2}} + x'_{\frac{n}{2}+1}\right)$. In summary: If n is odd, $x'_m = x'_{\frac{n+1}{2}}$, else

$x'_m = \frac{1}{2}\cdot\left(x'_{\frac{n}{2}} + x'_{\frac{n}{2}+1}\right)$. Or, if n is even, $x'_m = \frac{1}{2}\cdot\left(x'_{\frac{n}{2}} + x'_{\frac{n}{2}+1}\right)$, else $x'_m = x'_{\frac{n+1}{2}}$.

How do we check if n is odd? We use the function MOD(n,d) where n and d are integers. The division of the integers n (*n*umerator) and d (*d*enominator) produces a quotient q and a residual r, so that:

$$\frac{n}{d} = q + \frac{r}{d} \quad .$$

Function MOD(n,d) produces the residual of the division of two integers, i.e., MOD(n,d) = r. Thus, MOD(2,2) = 0, MOD(3,2) =1, MOD(4,2) = 0, MOD(5,2) = 1, etc. A number n is even if MOD(n,2) = 0. Therefore, the pseudo-code for finding the median x'_m is as follows:

- Enter vector x, find its length n
- Copy vector x into vector x'
- Sort vector x' into increasing order as demonstrated in the previous section
- If MOD(n,2) = 0 then the median is $x'_m = \frac{1}{2}\cdot\left(x'_{\frac{n}{2}} + x'_{\frac{n}{2}+1}\right)$, else $x'_m = x'_{\frac{n+1}{2}}$

In the previous section we developed the code for entering vector x, copying into vector xp (i.e., x'), and sorting it in increasing order (see *Sub SortData* in Figure 42). Thus, the piece of code we need to include would be:

```
Sub calculateMedian
… 'Define variables x, xp, n, etc.
   Dim xmedian As Double
… 'Enter x data, find n
… 'Copy x to xp, sort xp (see previous section)
…
       If MOD(n,2) = 0 Then
                    xmedian = (xp(n/2)+xp(n/2+1))/2
       Else
                    xmedian = xp((n+1)/2)
       End If
----
End Sub
```

Figure 43. Code segment to calculate the median of a vector.

Actual implementation of the median algorithm into code is left as an exercise for the reader.

Reversing the order of a vector

Reversing the order of a vector simply means that the vector x of length n with elements ($x_1, x_2, \ldots, x_n$), will be rewritten as the vector xb = ($x_n, x_{n-1}, \ldots, x_2, x_1$), i.e., reversing the order of the subindices. The algorithm for performing this re-ordering depends on whether the size of the vector n is odd or even. If the vector size n is even, i.e., MOD(n,2) = 0, then, we use a *For ... Next* loop to perform the reordering as follows:

```
For k = 1 to n/2
      xb(n-k+1) = x(k)
      xb(k) = x(n-k+1)
Next k
```

On the other hand, if *n* is odd, i.e., MOD(n,2) ≠ 0, then the *For* ... *Next* loop to perform the reordering is the following:

```
For k  = 1 to (n-1)/2
      xb(n-k +1) = x(k)
      xb(k) = x(n-k+1)
Next k
xb((n+1)/2) = x((n+1)/2)
```

This pseudo-code can be implemented in code after the original vector x is loaded and the vector size, *n*, is obtained when entering data into memory (see, for example, the code of Figure 40).

Actual implementation of the backwards reordering algorithm into code is left as an exercise for the reader.

Using MOD for selective, regular output for a long iterative process – ODE Euler method

Consider again the ODE solution through the Euler method presented in Figure 34. The example presented in that figure requires the program to produce $n = 1000$ output values in order to achieve a relatively small x increment Δx. Suppose that you still want to keep n at 1000, but you only want to write out (x,y) values every 100 points. You can modify the code of Figure 32 to include a printing increment, call it, $kp = 100$. This value can be input into the program from the interface, and incorporated in the segment of code that produces the output, as follows:

```
Sub SolveODE1
'Definition of variables
   Dim x As Double, y As Double
   Dim xf As Double, Dx as Double, ff As Double
   Dim k As Integer, n As Integer, kp As Integer
'Data input
   x = Worksheets("ODE1").Range("B2").Value
   y = Worksheets("ODE1").Range("B3").Value
   xf = Worksheets("ODE1").Range("B4").Value
   n = Worksheets("ODE1").Range("B5").Value
   kp = Worksheets("ODE1").Range("??").Value ' cell reference to be decided by programmer
'calculation of x increment
   Dx = (xf-x)/n
   Worksheets("ODE1").Range("B11").Value = Dx
'Iterative process to calculate ODE numerical solution
   For k = 0 to n
       If k = 0 Or Mod(k,kp) = 0 Then
          Worksheets("ODE1").Range("D3").Cells(k+1,1).Value = k
          Worksheets("ODE1").Range("D3").Cells(k+1,2).Value = x
          Worksheets("ODE1").Range("D3").Cells(k+1,3).Value = y
       End If
       Worksheets("ODE1").Range("B7").Value = x
       Worksheets("ODE1").Range("B8").Value = y
       ff = Worksheets("ODE1").Range("B9").Value
       x = x + Dx
       y = y + ff*Dx
   Next
End Sub
```

Figure 44. VBA code for calculating the numerical solution of the ODE $dy/dx = f(x,y)$ using the Euler method. This code is associated with the [Solve ODE] button of Figure 31.

Testing this modification is left as an exercise for the reader.

Use of vectors to calculate a numerical integral when a table (x,y) is given – the trapezoidal rule

The code of Figure 13 was developed to numerically approximate the integral $\int_a^b f(x)\,dx$ if the function $f(x)$ is given. If instead of a function $f(x)$ we are given a data set (x_k,y_k), $k = 1, 2, \ldots, n,$ with $a = x_k$ and $b = x_n$, the approach to be used to estimate the integral will consist in adding the area of the n-1 trapezoids that result for each subinterval $[x_k, x_{k+1}]$, $k = 1, 2, \ldots, n$-1. The area of k-th trapezoid is given by:

$$A_k = \frac{1}{2}\cdot(y_k + y_{k+1})\cdot(x_{k+1} - x_k) \quad ,$$

therefore, the integral is approximated by:

$$\int_a^b f(x)\,dx \approx \sum_{k=1}^{n-1} A_k = \frac{1}{2}\sum_{k=1}^{n-1}(y_k + y_{k+1})\cdot(x_{k+1} - x_k) \quad . \qquad [19]$$

The (x,y) data can be given as a table in the interface, read into the program using combinations of the *Range*( ) and *Cell*( ) properties of the worksheet, and the integral calculated using *For ... Next* loops. The value of the number of points, n, can be obtained from reading the data in as done in the code of Figure 36.

The development of the interface and of the code required for calculating the approximate integral of equation [19] is left as an exercise for the reader.

Dynamic Arrays – Use of *ReDim*

In the code of Figure 42, we declare variables x, xP (representing x'), and xPP (representing x'') with the line of code `Dim x(100) As Double, xP(100) As Double, xPP(100) As Double`. The one-dimensional arrays thus declared are referred to as *static arrays*, meaning that the size of the array is fixed from the start (at 100 elements, in these cases), and cannot be changed as the code is executed. In the particular case of the code of Figure 42, the number of elements n of the arrays x, xP, and xPP can only be less than or equal to 100. If, by any chance, the code was required to process more than 100 elements, the program will crash.

If the number of elements to be used is not known a priori, it is possible to define what are referred to as *dynamic arrays*. The declaration of a dynamic array requires a *Dim* (*Dim*ension) statement with the array's name and an empty set of parentheses. The size of the array is defined afterward in the code execution, and the *ReDim* (*ReDim*ension) statement is used to set the size of the dynamic array.

For example, the code of Figure 42 can be redone using dynamic arrays as illustrated in Figure 45, below.

```
Option Explicit
Option Base 1

Sub SortData
'Variable declaration
    Dim x() As Double, xP() As Double, xPP() As Double 'Declaring dynamic array
    Dim i As Integer, j As Integer, n As Integer, temp As Double

'Find sample size (n):
    n = 1  'Initialize sample size, n
    Do While Worksheets("Sorting").Range("B3").Cells(n,1).Value <> ""
        n = n + 1
    Loop
    n = n - 1   'Value of n adjusted after leaving loop

' Redimensioning the dynamic arrays
    ReDim x(n), xP(n), xPP(n)

'Input data, and copy x to xP and xPP
    For i = 1 To n
        x(i) = Worksheets("Sorting").Range("B3").Cells(i,1).Value
       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 45. The code of Figure 42 for module *SortData* redone using dynamic arrays.

The user is invited to test this code using the interface of Figure 41 as an exercise.

Next, we will use dynamic arrays and random number to simulate a simple algorithm for the random motion of particles. The motion of molecules or small particles in a fluid follows this pattern of motion which is known as *Brownian motion*. First, we introduce random numbers in *Excel* and *VBA*.

Random numbers in *Excel* and *VBA*

Most computer languages include a function (or functions) for producing random numbers. Ideally, a random number is a number associated with a random process, such as throwing a dice, playing

roulette, or measuring the maximum temperature at a weather station. Computers, by its own nature, are deterministic machines. In other words, the algorithms that we program are, basically, predictable. However, mathematicians and computer scientists have come up with algorithms that produce sequences of numbers that behave like random numbers. These sequences of numbers will eventually start repeating themselves. However, if the algorithm is able to produce a large quantity of these numbers before repetition occurs, the numbers thus produced behave, for all practical purposes, as random numbers. Realistically, these sequences are not exactly random numbers, but are referred to as *pseudo-random numbers*.

Excel provides the worksheet function RAND( ) that produces a pseudo-random number in the range (0,1). The equivalent VBA function is Rnd( ). These numbers are uniformly distributed in the range (0,1), meaning that any number in the range has basically the same probability of being selected.

To test function RAND( ) in *Excel*, type the formula "=RAND( )" in any cell, and then click in any other cell. A number between (0,1) will be shown in the cell where the formula was entered. If you click on the cell containing the formula "=RAND( )" once again, and drag this cell down, say, 5 cells, new random numbers between 0 and 1 will be produced.
To test the *Rnd*( ) function in VBA, type the following code in a module, and run it several times from the *VBA IDE*. Different random numbers will be shown in the output.

```
Sub RandomTest
       Dim r As Double
       r = Rnd()
       MsgBox("Random number = " & Cstr(r))
End Sub
```

Instead of the function *Rnd*( ) in the code, we could use the worksheet function *RAND*( ). For example, the code above, can be written as:

```
Sub RandomTest
       Dim r As Double
       r = WorksheetFunction.RAND()
       MsgBox("Random number = " & Cstr(r))
End Sub
```

Try this version of the program to verify that different pseudo-random numbers are produced.

<u>*Uniformly-distributed Random Numbers in the Range* (*a*,*b*)</u>
Whereas worksheet function *RAND*( ) or *VBA* function *Rnd*( ) produce pseudo-random numbers in the range (0,1), to produce a uniformly-distributed pseudo-random number in the range (*a*,*b*) you can use the following formula: `r = a + (b-a)*WorksheetFunction.RAND()`, or `r = a + (b-a)*Rnd()`. These formulas can be useful, for example, to generate uniformly-distributed random angles in the range $(-\pi,\pi)$ or $(0,2\cdot\pi)$ to simulate the Brownian motion of a particle. Test the following code running it from the *VBA IDE*:

```
Sub RandomAngle
       Dim theta As Double, PI As Double
       PI = 4.0 * WorksheetFunction.ATAN(1.0)
       theta = -PI + 2 * PI * WorksheetFunction.RAND()
       MsgBox("Random angle = " & Cstr(theta))
End Sub
```

A model of random motion may require that, in each time step, a particle will move from its current position in a direction determined by a uniformly-distributed angle θ in the range $(-\pi, \pi)$ (as those produced in the code above), and by a distance *R* that is *normally distributed* with a mean value μ_R and a standard deviation σ_R. Thus, before attempting the simulation of Brownian motion we will discuss the probability distributions for the *uniform distribution* and for the *normal distribution* of probabilities. First, we introduce some concepts related to probability distributions.

Random Variables. A *random variable* is a variable representing a quantity that exhibits random behavior. For example, the diameter of ball bearings produced by a factory may be subject to random variations, and it would be, therefore, a random variable. Other examples of random variables include: (i) the number of trucks being serviced at a service station on a given day; (ii) the number of passengers using a bus terminal in a given hour; (iii) the number of hits that a particular web site gets in one minute, or (iv) the number of cars taking a left turn at an intersection in one minute. These four random variables represent quantities that are counted in integer numbers (1, 2, 3, etc.), and are referred to as *discrete random variables*. On the other hand, random variables such as (i) the diameter of a ball bearing, (ii) the speed of a car in a stretch of road, (iii) the current air temperature in a given location, or (iv) the amount of rain received at a meteorological station on a given day, represent quantities that are measures continuously by the corresponding measuring device (a ruler, a speedometer, a thermometer, or a rain Gage, respectively), and are, therefore, referred to as *continuous random variables*.

Probabilities of continuous random variables. In this section we are aiming at simulating the behavior of a particle undergoing Brownian motion, therefore, the two quantities of interest, namely, the direction angle and the distance that the particle travels in a given time step, represent two continuous random variables. When dealing with continuous random variables, we are interested in estimating probabilities of continuous intervals, i.e., we will be interested in estimating $P(x_1 < X < x_2)$ = the probability that the random variable X is located between x_1 and x_2. [NOTE: traditionally, in the study of probability, random variable names are given using upper-case letters, e.g., X, while specific values of the random variable are given using lower-case letters, e.g., *x*]. To calculate this probability we use a function *f*(*x*) referred to as the *probability density function* or *pdf*, such that

$$P(x_1 < X < x_2) = \int_{x_1}^{x_2} f(x)\,dx \quad . \qquad [20]$$

Properties of the probability density function (pdf). (1) Since probability is a quantity between 0 and 1, the pdf *f*(*x*) must be a continuous function such that *f*(*x*) > 0. (2) In principle, a continuous random variable X can take any value in $(-\infty, +\infty)$, and the probability of X being in that interval must be 1.0, thus, another property of the pdf is that $\int_{-\infty}^{+\infty} f(x)\,dx = 1.0$. This can also be interpreted by saying that the area under the curve represented by $y = f(x)$ must add to 1.0.

The pdf *for the uniform distribution*. Let the continuous random variable X be uniformly-distributed in the interval (a, b). The notation "X ~ U(a,b)," interpreted as "X follows the uniform distribution with parameter a and b" is commonly used. The probability density function, or *pdf*, for a uniform distribution is represented by a constant value $f(x) = f_c$ in the interval (a,b), and zero elsewhere, as illustrated in Figure 46, below.

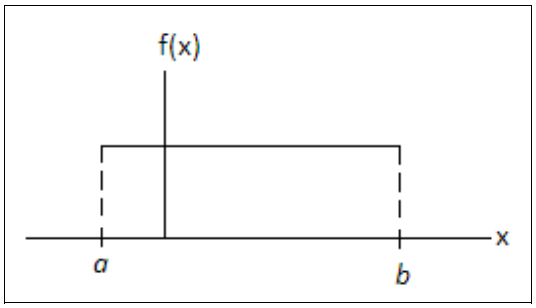


Figure 46. Probability density function for the uniform distribution, X ~ U(a,b).

Since the area of the rectangle limited by $x = a$, $x = b$, and $y = f_c$ must be equal to 1.0, i.e., $(b-a)\cdot f_c = 1.0$, then $f_c = \frac{1}{b-a}$. The probability density function (*pdf*) for the uniform distribution X ~ U(a,b) is, therefore, given by:

$$f(x) = \left\{ \begin{array}{c} \frac{1}{b-a}, for\, a < x < b \\ 0, otherwise \end{array} \right\} \quad . \qquad [21]$$

The cumulative distribution function (CDF) for continuous random variables. The cumulative distribution function $F(x)$ for a continuous random variable is defined by:

$$F(x) = \int_{-\infty}^{x} f(\eta)\, d\eta \quad = P(X \leq x) \qquad [22]$$

Thus, the CDF $F(x)$ is related to a probability. Also, the CDF can be related to the probability calculation of equation [20] as follows:

$$P(x_1 < X < x_2) = F(x_2) - F(x_1) \quad . \qquad [23]$$

It can be shown that, for the uniform distribution X ~ U(a,b), the cumulative distribution function (CDF) is given by

$$F(x) = \left\{ \begin{array}{c} 0, for\, x \leq a \\ \frac{x-a}{b-a}, for\, a < x < b \\ 1.0, for\, x \geq b \end{array} \right\} \quad . \qquad [23].$$

The graph of the uniform-distribution's CDF is shown in the following Figure.

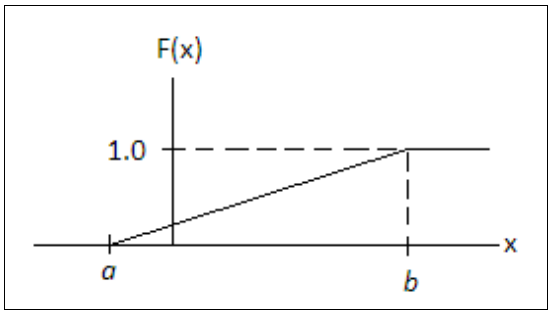


Figure 47. Cumulative distribution function (CDF) for the uniform distribution X ~ U(*a*,*b*).

Thus, having an expression for $F(x)$, as in equation [23] for X ~ U(*a,b*), is useful in terms of calculating probabilities. Furthermore, if an expression for the CDF $F(x)$ exists, it can be used to determine an inverse for the CDF as described next.

The Inverse Cumulative Distribution Function (ICDF). From equation [22], if a probability p is given for a known distribution, i.e., if $p = P(X \le x) = F(x)$ is given, the *inverse cumulative distribution function,* or *ICDF*, will be defined as:

$$x = F^{-1}(p) \quad . \tag{24}$$

For example, from the expression for the CDF for X ~ U(*a*,*b*), it follows that the ICDF for this distribution is given by:

$$x = F^{-1}(p) = a + (b-a) \cdot p \quad . \tag{25}$$

The normal probability distribution. Many physical quantities follow what is known as a normal probability distribution. This distribution depends on two parameters, a mean value (μ_X) and a standard deviation (σ_X). Thus, if the associated random variable is X we say that "X follows the normal distribution with parameters μ_X and σ_X ", or "X ~ N(μ_X , σ_X)." The normal distribution's *pdf* is given by:

$$f(x) = \frac{1}{\sqrt{2 \cdot \pi \cdot \sigma_X}} \exp\left(\frac{(x-\mu_X)^2}{2 \cdot \sigma_X^2} \right) \quad \text{, for} \quad -\infty < x < \infty \quad . \tag{26}$$

There is no closed-form expression for the cumulative distribution function (CDF) of the normal distribution. However, values of the normal CDF can be obtained in *Excel* by using the worksheet function " = $NORMDIST(x, \mu_X, \sigma_X)$." The same function can be used to obtain values of the normal pdf if we add a fourth argument to the call with the value or zero, i.e., " = $NORMDIST(x, \mu_X, \sigma_X, 0)$." The following figure shows the normal distribution's pdf for $\mu_X = 15$ and $\sigma_X = 5$, obtained from a *Excel* worksheet using function *NORMDIST*.

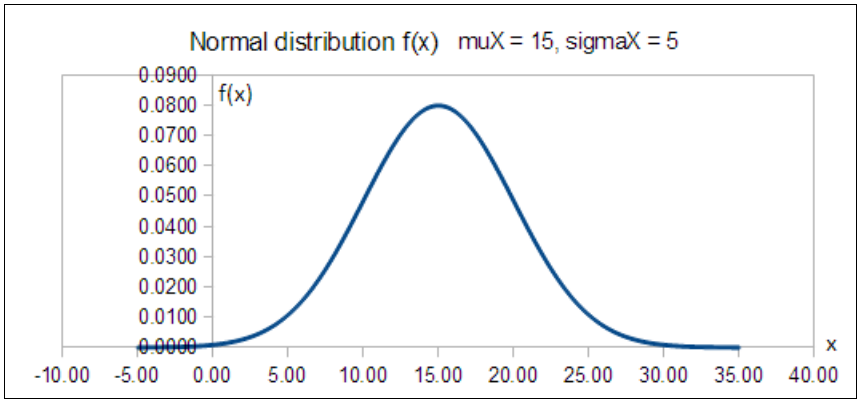


Figure 48. The normal distribution's probability density function (pdf) for $\mu_X = 15$, $\sigma_X = 5$.

The normal distribution's pdf, as illustrated in Figure 48, has a characteristic bell shape, and it's sometimes referred to as the "bell curve," and, also, as the Gaussian distribution. Function *NORMDIST* was also used to produce the following plot of the normal distribution's CDF for the parameters $\mu_X = 15$ and $\sigma_X = 5$. The normal distribution's CDF show a characteristic "elongated S" shape.

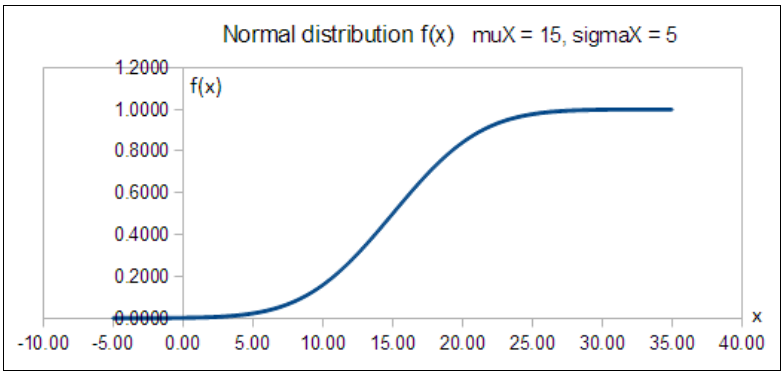


Figure 48. The normal distribution's probability density function (pdf) for $\mu_X = 15$, $\sigma_X = 5$.

To obtain values of the normal distribution's inverse cumulative distribution function (ICDF), given a probability p ($0 < p < 1$), for a normal distribution with parameters μ_X and σ_X , we use *Excel*'s worksheet function "= NORMINV(p , μ_X , σ_X)." This function can be used, in combination with *Excel*'s worksheet function *RAND*( ) or *VBA*'s function *Rnd*( ), to generate random numbers that are normally distributed.

<u>*Generating Normally-distributed Random Numbers*</u>. With *Excel*'s worksheet function *RAND*( ) or *VBA*'s function *Rnd*( ) we can generate a random number between 0 and 1. Since a probability p is also a value between 0 and 1, if we take the random number generated by *RAND*( ), or by *Rnd*( ), to represent a normal probability p, then function *NORMINV*, with the proper parameters, will produce a normally-distributed random number. Thus, to produce a normally-distributed random number for X ~ N(μ_X , σ_X), in code, we can use:

```
r = WorksheetFunction.NORMINV(WorksheetFunction.RAND(), μX , σX )
```

or

```
r = WorksheetFunction.NORMINV(Rnd(), μX , σX ) .
```

In the next section we use what we learned about generating uniformly-distributed and normally-distributed random numbers to simulate the random (Brownian) motion of a single particle.

Brownian Motion – Single Particle

Suppose we release a single particle at the origin of Cartesian coordinate system (x,y) and simulate its motion by producing n time steps. In each time step k the position of the particle is updated by using the equations:

$$x_{k+1} = x_k + R_k \cdot \cos(\theta_k) \quad \text{, and} \quad y_{k+1} = y_k + R_k \cdot \sin(\theta_k) \quad ,$$

where $\theta_k \sim U(-\theta_1, \theta_2)$, and $R_k \sim N(\mu_R, \sigma_R)$. The calculation of positions indicated above is illustrated in the following Figure.

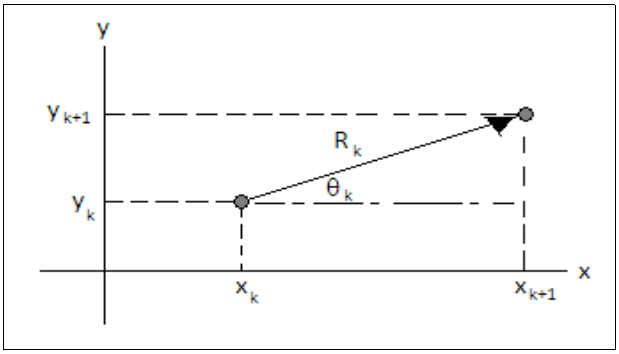


Figure 49. Increment in x and y positions for the motion of a particle.

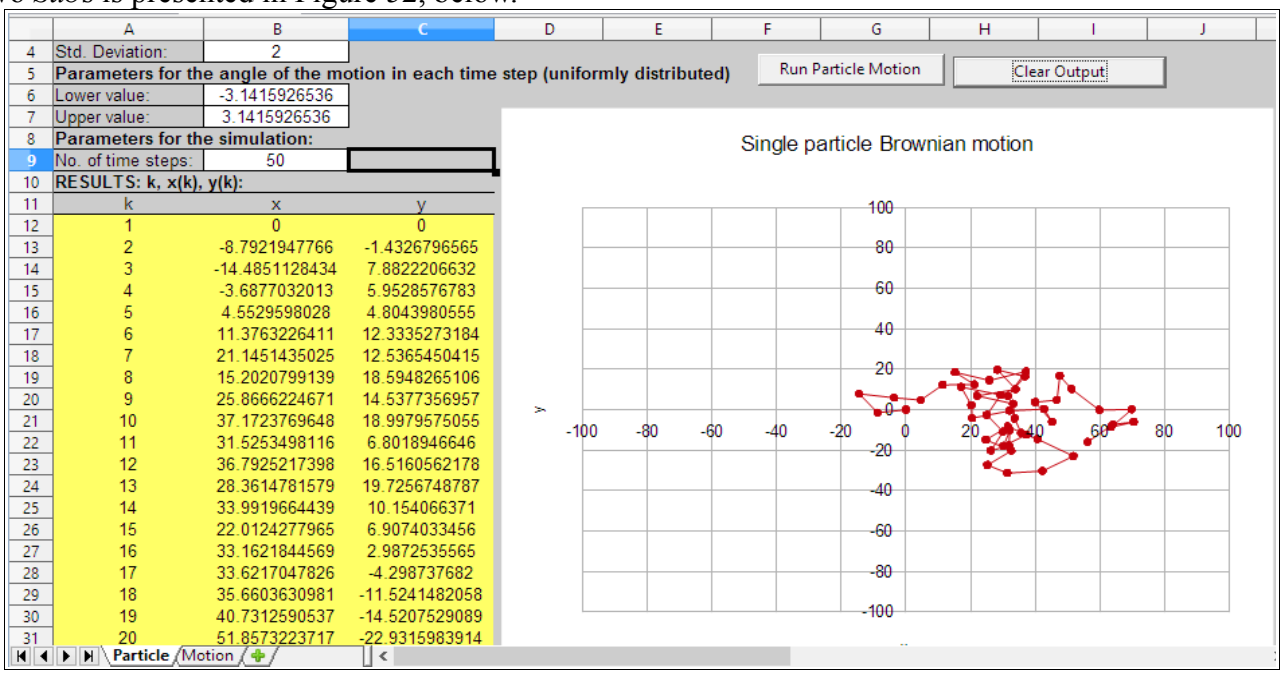
The randomness associated with this motion simulation is incorporated in the facts that the angular position is a uniformly-distributed random number between θ_1 and θ_2 , while the distance is also a random number, normally-distributed with certain parameters μ_R and σ_R .

In this example, we assume that the particle starts at the origin (0,0). A pseudo-code for the simulation of the particle motion through n time steps is shown below.

```
Sub BrownianParticle
    Declare variables: θ1, θ2 ,  μR  ,  σR  , x (), y(), R, θ, k, n
    x(1) ← 0, y(1) ← 0
    Output 1, x(1), y(1)
    Input  values of θ1, θ2 ,  μR  ,  σR  , n
    ReDim x(n), y(n)
    For k  from 1 to n
        θ ← θ1 + (θ2 ,- θ1) * RAND()
        R ← NORMINV(RAND(),  μR  ,  σR  )
        x(k+1) ← x(k) + R*cos(θ) : y(k+1) ← y(k) + R*sin(θ)
        Output k+1, x(k+1), y(k+1)
    Next
End Sub
```

Figure 50. Pseudo-code for simulating the random motion of a single particle.

An interface for the simulation of a single-particle random motion is shown in Figure 51, including a graph showing the motion. Button [Run Particle Motion] is associated with *Sub BrownianParticle* code, while button [Clear Output] is associated with the *Sub ClearOutputParticle*. The code for these two *Sub*s is presented in Figure 52, below.



k	x	y
1	0	0
2	-8.7921947766	-1.4326796565
3	-14.4851128434	7.8822206632
4	-3.6877032013	5.9528576783
5	4.5529598028	4.8043980555
6	11.3763226411	12.3335273184
7	21.1451435025	12.5365450415
8	15.2020799139	18.5948265106
9	25.8666224671	14.5377356957
10	37.1723769648	18.9979575055
11	31.5253498116	6.8018946646
12	36.7925217398	16.5160562178
13	28.3614781579	19.7256748787
14	33.9919664439	10.154066371
15	22.0124277965	6.9074033456
16	33.1621844569	2.9872535565
17	33.6217047826	-4.298737682
18	35.6603630981	-11.5241482058
19	40.7312590537	-14.5207529089
20	51.8573223717	-22.9315983914

Figure 51. *Excel* interface for simulating the random motion of a single particle.

```
Option Explicit
Option Base 1
Sub BrownianParticle
    Dim rbar As Double, sr As Double, Low As Double, High As Double
    Dim uR As Double, rR As Double
    Dim nTimes As Integer
    Dim k As Integer
    Dim x() As Double, y() As Double
    rbar = Worksheets("Particle").Range("B3").Value
    sr = Worksheets("Particle").Range("B4").Value
    Low = Worksheets("Particle").Range("B6").Value
    High = Worksheets("Particle").Range("B7").Value
    nTimes = Worksheets("Particle").Range("B9").Value
    Redim x(nTimes), y(nTimes)
    x(1) = 0.0 : y(1) = 0.0
    Worksheets("Particle").Range("A12").Value = 1
    Worksheets("Particle").Range("B12").Value = x(1)
    Worksheets("Particle").Range("C12").Value = y(1)
    For k = 1 To nTimes-1
         uR = Low + (High-Low) * WorksheetFunction.RAND()
         rR = WorksheetFunction.NORMINV(WorksheetFunction.RAND(),rbar,sr)
         x(k+1) = x(k) + rR*cos(uR) : y(k+1) = y(k) + rR*sin(uR)
         Worksheets("Particle").Range("A12").Cells(k+1,1).Value = k+1
         Worksheets("Particle").Range("A12").Cells(k+1,2).Value = x(k+1)
         Worksheets("Particle").Range("A12").Cells(k+1,3).Value = y(k+1)
    Next
End Sub
```

```
Sub ClearOutputParticle
    Dim n As Integer, i As Integer, j As Integer
    n = Worksheets("Particle").Range("B9").Value
    Worksheets("Particle").Range("B9").Value = ""
    For i = 1 To n
        For j = 1 To 3
            Worksheets("Particle").Range("A12").Cells(i,j).Value = ""
        Next
    Next
End Sub
```

Figure 52. Code for *Sub*s *BrownianParticle* and *ClearOutputParticle* associated with the buttons [Run Particle Motion] and [Clear Output], respectively, in Figure 51.

Brownian Motion – Multiple Particle Animation

Figure 53 shows the interface for simulating the motion of several particles under Brownian motion as described in the previous section. The code is presented in Figure 54. *Sub Brownian* is associated with button [Run Animation] while *Sub ClearOutput* is associated with [ClearOutput]. *Sub Brownian* is a generalization of the code for *Sub BrownianParticle* in Figure 52, producing the coordinates (*x*,*y*) of the various particles modeled, which are then output to the interface, and plotted on *an x-y* graph. When the next time step comes along, the coordinates (*x*,*y*) of the new positions of the particles are output to the table in the interface and the corresponding data points are replaced in the graph, thus producing an animation of the motion of the individual particles.

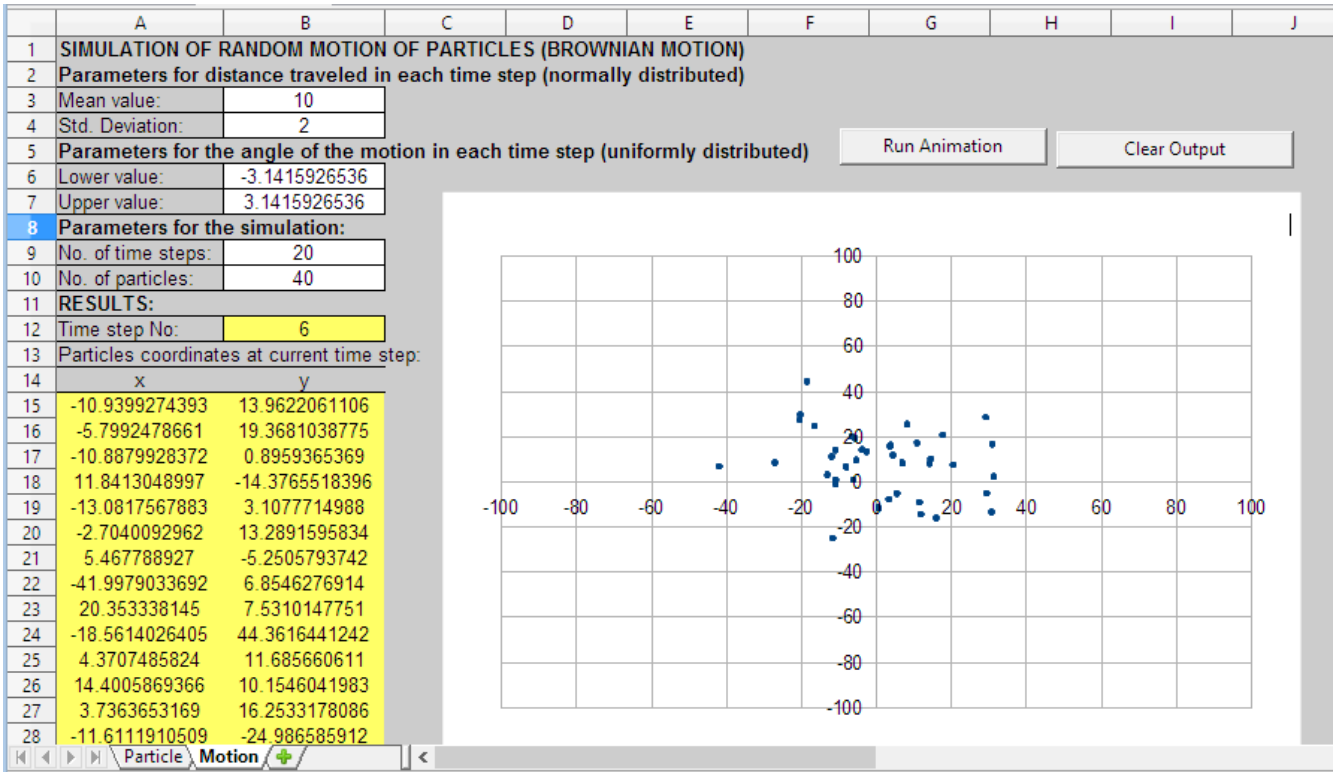


Figure 53. *Excel* interface for simulating the random motion of a several particles.

```
Option Explicit
Option Base 1

Sub BrownianMotion
    Dim rbar As Double, sr As Double
    Dim Low As Double, High As Double
    Dim uR As Double, rR As Double
    Dim nTimes As Integer, nParticles As Integer
    Dim i As Integer, j As Integer, k As Integer, Time As Integer
    Dim x() As Double, y() As Double
    rbar = Worksheets("Motion").Range("B3").Value
    sr = Worksheets("Motion").Range("B4").Value
    Low = Worksheets("Motion").Range("B6").Value
    High = Worksheets("Motion").Range("B7").Value
    nTimes = Worksheets("Motion").Range("B9").Value
    nParticles = Worksheets("Motion").Range("B10").Value
    Redim x(nParticles), y(nParticles)
    Worksheets("Motion").Range("B12").Value = 0
    For j = 1 to nParticles
        x(j) = 0.0 : y(j) = 0.0
        Worksheets("Motion").Range("A15").Cells(j,1).Value = x(j)
        Worksheets("Motion").Range("A15").Cells(j,2).Value = y(j)
    Next
    For i = 1 To nTimes
        Worksheets("Motion").Range("B12").Value = i
        For j = 1 To nParticles
            uR = Low + (High-Low) * WorksheetFunction.RAND()
            rR = WorksheetFunction.NORMINV(WorksheetFunction.RAND(),rbar,sr)
            x(j) = x(j) + rR*cos(uR) : y(j) = y(j) + rR*sin(uR)
            Worksheets("Motion").Range("A15").Cells(j,1).Value = x(j)
            Worksheets("Motion").Range("A15").Cells(j,2).Value = y(j)
        Next
        DoEvents
        Application.Wait DateAdd("s", 1, Now)
    Next
End Sub

Sub ClearOutput
    Dim n As Integer, i As Integer, j As Integer
    n = Worksheets("Motion").Range("B10").Value
    Worksheets("Motion").Range("B12").Value = ""
    For i = 1 To n
        For j = 1 To 2
            Worksheets("Motion").Range("A15").Cells(i,j).Value = ""
        Next
    Next
End Sub
```

Figure 54. Code for *Sub*s *Brownian* and *ClearOutput* associated with the buttons [Run Animation] and [Clear Output], respectively, in Figure 53.

Notice the statements *DoEvents* and *Application.Wait DateAdd*("s", 1, *Now*). The *DoEvents* statement forces the graph to update so that you can see the particles in motion. The statement *Application.Wait DateAdd*("s", 1, *Now*) basically makes the graph updates wait for 1 second ("s", 1, Now).

Strings as Arrays and String Functions in *VBA*

Up to now we have used strings for input and output, learned how to convert numbers to strings (CStr), or string to numbers (CDbl, CInt), how to concatenate strings (&), how to use *Char*(*n*) to produce ASCII characters, and how to use *Asc*(*char*) to find the ASCII code for *char*. A string can be though of as an array with each character in the string being an element of the array in the order in which the string is written. For example, the following string is shown with one character per cell with the cell numbering above each character. You can think of the individual cell numbers as a position in the string.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
H	e	l	p		m	e	,		O	b	i		W	a	n		K	e	n	o	b	i	.

Figure 55. The indexing of characters in a string is similar to that of a one-dimensional array.

Suppose that you assign this string to a variable S, by using, in a VBA program, the following commands:

```
Dim S As String
S = "Help me, Obi Wan Kenobi."
```

The length of the string S can be determined by using function **Len(S).** We can manipulate parts of the string by using a number of string functions such as:

- **Left(S, *k*)** = returns *k* characters of string S counting starting from the leftmost character in string S
- **Mid(S, k_1, k_2)** = returns k_2 characters starting at positions k_1 in string S.
- **Right(S, *k*)** = returns *k* characters of string S counting backwards from the rightmost character in string S

The following program illustrates the use of functions *Len*, *Left*, *Mid*, and *Right* in the string S defined above. Remember that *Chr*(13) produces a new line for the output string in the *MsgBox*.

```
Sub Main
    Dim S As String
    S = "Help me, Obi Wan Kenobi."
    MsgBox("S = " & S & Chr(13) & _
           "Length of S = " & Len(S) & Chr(13) & _
           "Left(S,1) =  " & Left(S,1) & Chr(13) & _
           "Mid(S,6,2) = " & Mid(S,6,2) & Chr(13) & _
           "Right(S,1) = " & Right(S,1) & Chr(13) & _
           "Left(S,3) = " & Left(S,3) & Chr(13) & _
           "Mid(S,10,3) = " & Mid(S,10,3) & Chr(13) & _
           "Right(S,3) = " & Right(S,3))
End Sub
```

Figure 56. Code for testing string functions *Len, Left, Mid*, and *Right*.

The output for this code of Figure 56 is shown in the *MsgBox* of Figure 57. For the positioning of the different characters, refer to Figure 55.

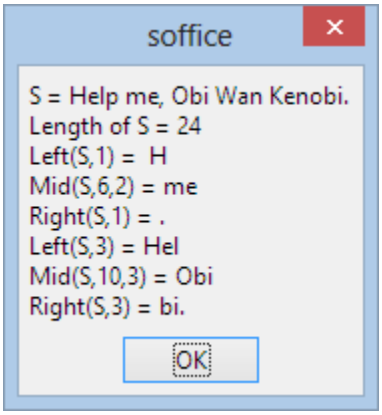


Figure 57. Output of the code of Figure 56 illustrating the use of string functions *Len, Left, Mid*, and *Right*.

Suppose that you want to find the starting position within a string S where a substring S1 is located. The function to use is:

- **InStr(S,S1)** = returns the position within string S where substring S1 starts

To illustrate the use of this function, consider the following code:

```
Sub StringExtractAndCombine
    Dim S As String, SI As String, SL As String, SR As String
    Dim SN As String
    Dim k1 As Integer, k2 As Integer
    S = "Help me, Obi Wan Kenobi."  ' Original string
    SI = "Obi Wan"                  ' Part of string to be removed
    k1 = InStr(S,SI)                ' Find where SI starts in S
    SL = Left(S,k1-1)               ' Left String (SL)
    k2 = k1 + Len(SI)+1             ' Beginning of Right String (SR)
    SR = Mid(S,k2,Len(S)-k2)        ' Right String (SR)
    SN = SL & SR                    ' Combine SL and SR
    MsgBox("Original string:   " & S  & Chr(13) & _
           "Extract string:    " & SI & Chr(13) & _
           "Left String:       " & SL & Chr(13) & _
           "Right String:      " & SR & Chr(13) & _
           "Last 2 combined:   " & SN)
End Sub
```

Figure 58. Code illustrating the use of string function *InStr* together with functions *Len, Left,* and *Mid*, to extract a substring from the original string "Help me, Obi Wan Kenobi."

The original string S = "`Help me, Obi Wan Kenobi.`" is the same used in the code of Figure 56. The string SI = "`Obi Wan`" is to be removed from the original string S. First, we find the location where SI starts within S, by using `k1 = InStr(S,SI).` From the original string S, we extract the "left" string, `SL = Left(S,k1-1).` Then, we calculate index `k2 = k1 + Len(SI) + 1`, which represents the point where the reminding of the original string starts after SI. A "right" string is extracted by using the expression: `SR = Mid(S,k2,Len(S)-k2).` Finally, we combine the "left" string and the "right" string using `SN = SL & SR`. The *MsgBox* command then shows strings S, SI, SL, SR, and SN as output.

The result from the code of Figure 58 is the following *MsgBox*:

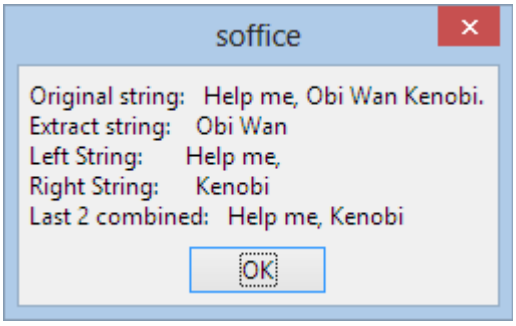


Figure 59. Output of the code of Figure 58 illustrating the use of string function *InStr*, together with *Len, Left,* and *Mid*.

Some strings, particular those read from data files, may have leading and/or trailing blank characters. For the purpose of eliminating those blank characters we can use the following functions:

- **LTrim(S)** = returns a string corresponding to string S, but with leading blank characters removed
- **RTrim(S)** = returns a string corresponding to string S, but with trailing blank characters removed
- **Trim(S)** = returns a string corresponding to string S, but with both leading and trailing blank characters removed

In addition, there are two functions that can change the letter case type to lower-case or upper-case:

- **LCase(S)** = returns the lower-case equivalent of string S
- **UCase(S)** = returns the upper-case equivalent of string S

The examples of Figures 57 and 59 show some of the applications of some functions (*Len, Left, Mid, Right*, and *InStr*) for the manipulation of strings. Also, we briefly described the operation of other string functions, given without examples: *LTrim*, *RTrim*, *Trim, LCase,* and *UCase*. Knowledge and use of these functions is useful if you have to handle text data, for example, in processing data files where some of the fields are strings, or when dealing with files names, and other types of text data. For numerical applications, you probably will not have much use for the functions listed above.

For additional string functions and their operations in *Visual Basic* (not all *Visual Basic* functions are available in *VBA*) visit: *http://msdn.microsoft.com/en-us/library/dd789093.aspx*

Excel also provides a number of functions for manipulating text in the worksheet cells. These functions are available under the category *Text* when pressing the [*fx*] button.

One-dimensional Arrays of Strings

As we did with *Integer* and *Double* data, we can create one-dimensional arrays of strings. For example, we can declare some string variables to be arrays of 10 elements to process a number of last and first names typed in columns B and C of a *Excel* interface as shown in Figure 60.

	A	B	C	D	E	F	G	H
1	No.	First Name	Last Name	Trimmed FN	Trimmed LN	Case fix FN	Case fix LN	Full Name (Last, First)
2	1	Joseph	Nyman					
3	2	Maria	Daines					
4	3	Paul	MCLeLLan					
5	4	MonicA	Lowell					
6	5	Roger	OchoCincO					
7	6	Patrick	Nahmin					
8	7	JakoB	Bush					
9	8	Mario	Rodriguez					
10	9	Luzan	Lozano					
11	10	Trinidad	Laredo					
12								
13								

Figure 60. Interface for string processing before code is activated.

The code shown in Figure 61 is designed to first trim leading and trailing blanks from *Last_Name* and *First_Name,* then correct the case type so that only the first letter of each last name and first name is upper case. Finally, the code will put the corrected last name and first name in the format *last, first*.

```
Sub StringArrayProcessing
    Dim First_Name(1 To 10) As String, Last_Name(1 To 10) As String
    Dim Trimmed_FN(1 To 10) As String, Trimmed_LN(1 To 10) As String
    Dim CaseFix_FN(1 To 10) As String, CaseFix_LN(1 To 10) As String
    Dim NameFixed(1 To 10) As String
    Dim k As Integer, n As Integer
    n= 10
    For k = 1 To n
        First_Name(k) = Worksheets("Sheet2").Range("B2").Cells(k,1).Value
        Last_Name(k)= Worksheets("Sheet2").Range("B2").Cells(k,2).Value
        Trimmed_FN(k) = Trim(First_Name(k))
        Trimmed_LN(k) = Trim(Last_Name(k))
        CaseFix_FN(k) = UCase(Left(Trimmed_FN(k),1)) & _
                        LCase(Mid(Trimmed_FN(k),2,Len(Trimmed_FN(k))))
        CaseFix_LN(k) = UCase(Left(Trimmed_LN(k),1)) & _
                        LCase(Mid(Trimmed_LN(k),2,Len(Trimmed_LN(k))))
        NameFixed(k) = CaseFix_LN(k) & ", " & CaseFix_FN(k)
        Worksheets("Sheet2").Range("B2").Cells(k,3).Value = Trimmed_FN(k)
        Worksheets("Sheet2").Range("B2").Cells(k,4).Value = Trimmed_LN(k)
        Worksheets("Sheet2").Range("B2").Cells(k,5).Value = CaseFix_FN(k)
        Worksheets("Sheet2").Range("B2").Cells(k,6).Value = CaseFix_LN(k)
        Worksheets("Sheet2").Range("B2").Cells(k,7).Value = NameFixed(k)
    Next

End Sub
```

Figure 61. Code for processing the text entries of the *Excel* interface of Figure 60.

The result of running the code of Figure 61 on the interface of Figure 60 (the worksheet is called "Sheet2") is shown in Figure 62.

	A	B	C	D	E	F	G	H
1	**No.**	**First Name**	**Last Name**	**Trimmed FN**	**Trimmed LN**	**Case fix FN**	**Case fix LN**	**Full Name (Last, First)**
2	1	Joseph	Nyman	Joseph	Nyman	Joseph	Nyman	Nyman, Joseph
3	2	Maria	Daines	Maria	Daines	Maria	Daines	Daines, Maria
4	3	Paul	MCLeLLan	Paul	MCLeLLan	Paul	Mclellan	Mclellan, Paul
5	4	MonicA	Lowell	MonicA	Lowell	Monica	Lowell	Lowell, Monica
6	5	Roger	OchoCincO	Roger	OchoCincO	Roger	Ochocinco	Ochocinco, Roger
7	6	Patrick	Nahmin	Patrick	Nahmin	Patrick	Nahmin	Nahmin, Patrick
8	7	JakoB	Bush	JakoB	Bush	Jakob	Bush	Bush, Jakob
9	8	Mario	Rodriguez	Mario	Rodriguez	Mario	Rodriguez	Rodriguez, Mario
10	9	Luzan	Lozano	Luzan	Lozano	Luzan	Lozano	Lozano, Luzan
11	10	Trinidad	Laredo	Trinidad	Laredo	Trinidad	Laredo	Laredo, Trinidad
12								
13								

Figure 62. Output of the code of Figure 61 for processing input text data from columns B and C.