

# VBA Programming in EXCEL: Programming Structures – Branching Structures

By Gilberto E. Urroz, June 2013

In this chapter we introduce the different types of programming structures: sequence, branching, and loops, and study branching structures in detail. However, before getting to those subjects we will discuss some practical applications of spreadsheets: formula evaluation, equation solution, and the use of multiple worksheets. After discussing these issues we go back to programming in VBA.

## Formula evaluation in EXCEL

The solution of equations from various disciplines is an important task in engineering practice. In this section we provide some useful ideas for solving equations using the EXCEL spreadsheet. If the variable you are solving for can be written explicitly, calculating that variable is as simple as evaluating a formula in the spreadsheet. For example, out of the equation of position in a uniformly-accelerated motion, namely,

$$y = y_0 + v_0 t + \frac{1}{2} a t^2, \quad [1]$$

we can solve for  $t$  by using the quadratic equation:

$$a t^2 + 2 v_0 t + 2 (y_0 - y) = 0. \quad [2]$$

Comparing with the generic quadratic equation:

$$\alpha x^2 + \beta x + \delta = 0, \quad [3]$$

whose solutions are:

$$x = \frac{-\beta \pm \sqrt{\beta^2 - 4 \alpha \delta}}{2 \alpha}, \quad [4]$$

allows to write:  $\alpha = a$ ,  $\beta = 2 v_0$ , and  $\delta = 2 (y_0 - y)$ , so that the solutions to [2] are:

$$t = \frac{-2 v_0 \pm \sqrt{4 v_0^2 - 8 a (y_0 - y)}}{2 a} = \frac{-v_0 \pm \sqrt{v_0^2 - 2 a (y_0 - y)}}{a}. \quad [5]$$

Actually, equation [5] represents two results, one for the plus sign, and one for the minus sign, thus, we could actually calculate the following two values:

$$t_1 = \frac{-v_0 + \sqrt{v_0^2 - 2 a (y_0 - y)}}{a}, \quad [6]$$

and

$$t_2 = \frac{-v_0 - \sqrt{v_0^2 - 2 a (y_0 - y)}}{a}. \quad [7]$$

Given values of the known variables, say,  $v_0 = 5.2$ ,  $a = 2.3$ ,  $y_0 = 1.2$ , and  $y = 15.2$ , we can simply enter the formulas in EXCEL cells, as illustrated in the figure below.

	A	B	C	D	E		A	B
1						1		
2	t1:	=(-5.2+sqrt(5.2^2-2*2.3*(1.2-15.2)))/2.3				2	t1:	1.9
3	t2:	=(-5.2-sqrt(5.2^2-2*2.3*(1.2-15.2)))/2.3				3	t2:	-6.42
4						4		

Figure 1. Formulas and values for calculating  $t_1$  and  $t_2$  from equations [6] and [7].

If we enter the known values in the spreadsheet cells before evaluating the formula, we can refer to the cell specifications in evaluating the formula. Consider the following spreadsheet for evaluating the formulas of equations [6] and [7].

	A	B	C	D	E
1	KNOW VALUES:				
2	v0:	5.2			
3	a:	2.3			
4	y0:	1.2			
5	y:	15.2			
6	CALCULATED VALUES:				
7	t1:	1.9	← =(-B2+SQRT(B2^2-2*B3*(B4-B5)))/B3		
8	t2:	-6.42	← =(-B2-SQRT(B2^2-2*B3*(B4-B5)))/B3		

Figure 2. Using relative cell references to calculate a formula.

Cells  $B7$  and  $B8$  contain the calculations of  $t_1$  and  $t_2$  from equations [6] and [7], respectively, with the known values, namely,  $v_0 = 5.2$ ,  $a = 2.3$ ,  $y_0 = 1.2$ , and  $y = 15.2$ , contained in cells  $B2$ ,  $B3$ ,  $B4$ , and  $B5$ , respectively. To build the formulas in cells  $B7$  and  $B8$ , you can start typing the equation and, then, click on the cell where the known value is contained as needed to build the formula for calculation. The resulting formula, for example, that in cell  $B7$ , will then contain references to the cells  $B2$ ,  $B3$ ,  $B4$ , and  $B5$ , e.g.,

$$= (-B2+SQRT (B2^2-2*B3*(B4-B5) ) ) /B3$$

References such as  $B2$ ,  $B3$ , etc., are called *relative references* to the cells.

#### *Relative references and table filling*

Suppose that we want to calculate the value  $t_1$ , from equation [6], for different values of  $y$ , while keeping  $v_0$ ,  $a$ , and  $y_0$ , constant. This will produce a table of values of  $y$  versus some values of  $t$ . Such a table can be built as in the spreadsheet of Figure 3. To build the table, proceed as follows:

- Enter a value of 2 in  $D3$ .
- In  $D4$ , enter " $=1+D3$ ".
- In  $E3$ , enter: " $=(-\$B\$2+SQRT(\$B\$2^2-2*\$B\$3*(\$B\$4-D3)))/\$B\$3$ "
- Click on  $E3$ , and drag it down to  $E4$
- Select  $D4:E4$ , and drag it down to  $D11:E11$

The references  $\$B\$2$ ,  $\$B\$3$ , and  $\$B\$4$ , in the formula entered above in  $E3$ , are *absolute references*,

thus, always referring to those particular cells. The reason being that they represent the known values  $v_0$ ,  $a$ , and  $y_0$ , which remain constant through the filling of the table.

If you were to check the formula in, say, cell  $E7$ , you will find the following:

$$=(-\$B\$2+\text{SQRT}(\$B\$2^2-2*\$B\$3*($B\$4-D7)))/\$B\$3$$

Comparing it to the original formula in  $B3$ , namely,

$$=(-\$B\$2+\text{SQRT}(\$B\$2^2-2*\$B\$3*($B\$4-D3)))/\$B\$3$$

you can see that the values of  $y$  have changed from  $D3$  to  $D7$ , as the table progresses downwards. The absolute references remain unchanged.

	A	B	C	D	E
1	KNOWN VALUES:			TABLE	
2	$v_0$ :	5.2		$y$	$t_1$
3	$a$ :	2.3		2	0.15
4	$y_0$ :	1.2		3	0.32
5	$y$ :	15.2		4	0.49
6	CALCULATED VALUES:			5	0.64
7	$t_1$ :	1.9		6	0.79
8	$t_2$ :	-6.42		7	0.93
9				8	1.06
10				9	1.19
11				10	1.31

Figure 3. Using relative and absolute references to fill out a table.

If you want to see the formulas used in a spreadsheet, all at once, do the following:

- Click on the *FORMULAS* tab
- Click on the *Show Formulas* button
- Back in the spreadsheet you may need to widen the columns where formulas appear, to be able to fit them – For example, for the spreadsheet of Figure 3, formulas are shown below:

	A	B	C	D	E
1	KNOWN VALUES:			TABLE	
2	$v_0$ :	5.2		$y$	$t_1$
3	$a$ :	2.3		2	$=(-\$B\$2+\text{SQRT}(\$B\$2^2-2*\$B\$3*($B\$4-D3)))/\$B\$3$
4	$y_0$ :	1.2		$=1+D3$	$=(-\$B\$2+\text{SQRT}(\$B\$2^2-2*\$B\$3*($B\$4-D4)))/\$B\$3$
5	$y$ :	15.2		$=1+D4$	$=(-\$B\$2+\text{SQRT}(\$B\$2^2-2*\$B\$3*($B\$4-D5)))/\$B\$3$
6	CALCULATED VALUES:			$=1+D5$	$=(-\$B\$2+\text{SQRT}(\$B\$2^2-2*\$B\$3*($B\$4-D6)))/\$B\$3$
7	$t_1$ :	$=(-B2+\text{SQRT}(B2^2-2*B3*(B4-B5)))/B3$		$=1+D6$	$=(-\$B\$2+\text{SQRT}(\$B\$2^2-2*\$B\$3*($B\$4-D7)))/\$B\$3$
8	$t_2$ :	$=(-B2-\text{SQRT}(B2^2-2*B3*(B4-B5)))/B3$		$=1+D7$	$=(-\$B\$2+\text{SQRT}(\$B\$2^2-2*\$B\$3*($B\$4-D8)))/\$B\$3$
9				$=1+D8$	$=(-\$B\$2+\text{SQRT}(\$B\$2^2-2*\$B\$3*($B\$4-D9)))/\$B\$3$
10				$=1+D9$	$=(-\$B\$2+\text{SQRT}(\$B\$2^2-2*\$B\$3*($B\$4-D10)))/\$B\$3$
11				$=1+D10$	$=(-\$B\$2+\text{SQRT}(\$B\$2^2-2*\$B\$3*($B\$4-D11)))/\$B\$3$

Figure 4. Formulas for the table of Figure 3.

To return to showing values, use the following:

- Click on the *FORMULAS* tab (if you moved to other tab in the meantime)
- Click on the *Show Formulas* button
- Back in the spreadsheet you may need to reduce the width of the columns that were widened earlier.

### Solving equations in EXCEL numerically with *Goal Seek* and *Solve*

Suppose that you want to solve for  $t$  from equation [1], but you don't want to algebraically isolate  $t$  as we did earlier. What is called for at this point is a *numerical solution*. We start from the original equation [1], and make it into an equation equal zero, i.e.,

$$y_0 + v_0 t + \frac{1}{2} a t^2 - y = 0 \quad . \quad [8]$$

You can either type the values of the known quantities in the expression that is to be made equal to zero, or enter those known quantities in specific cells and refer to those cells in the expression. For this example, we follow the latter approach and prepare a spreadsheet as follows:

B8		:	X	✓	$f_x$	=B4+B2*B7+1/2*B3*B7^2-B5				
	A	B	C	D	E					
1	KNOWN VALUES:									
2	v0:	5.2								
3	a:	2.3								
4	y0:	1.2								
5	y:	15.2								
6	SOLVING FOR:									
7	t:	1								
8	EQUATION:	-7.65								

Figure 5. Spreadsheet set up for numerical solution of an equation.

Cells  $B2$ ,  $B3$ ,  $B4$ , and  $B5$  contain the known values. Cell  $B7$  contains a guess for the solution,  $t$ , and cell  $B8$  contains the expression of equation [8], which we are trying to drive to zero. The expression for equation [8] is shown in the entry line in Figure 5, namely, " $=B4+B2*B7+1/2*B3*B7^2-B5$ ".

The idea behind a numerical solution as that set up in Figure 5 is to drive the equation to zero by changing the value of the unknown. This can be done systematically in EXCEL by using either the function *Goal Seek* or the *Solver* application.

#### *Solving an equation using Goal Seek*

Referring to the equation solution set up of Figure 5, to use *Goal Seek* for solving the equation, proceed as follows:

- Click on the *DATA* tab
- Click on the *What-If Analysis* button
- In the resulting drop-down menu (see Figure 6A, below), select *Goal Seek...*

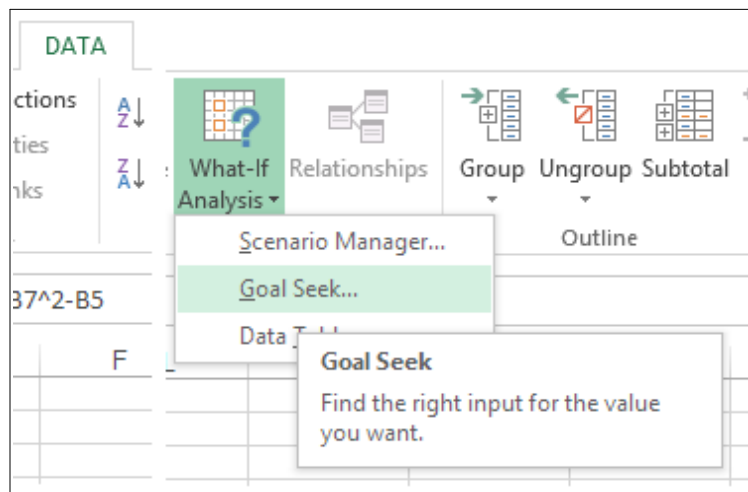


Figure 6A. Activating the *Goal Seek* option in the *DATA* tab

Next  
 Net, fill out the following in the resulting form:

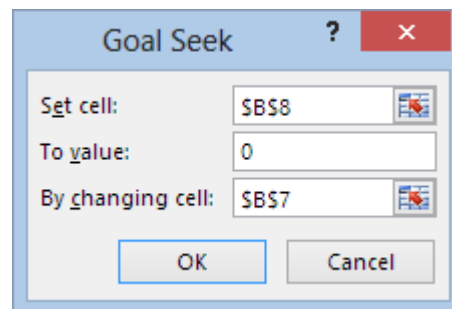


Figure 6. *Goal Seek* settings for the equation solution of Figure 5.

The settings in Figure 6 indicate that the formula in cell  $B8$  seeks a target value of zero by changing the value of cell  $B7$ . Press [ OK ] to see a solution. For the starting value of  $t = 1$ , *Goal Seek* finds the following solution:

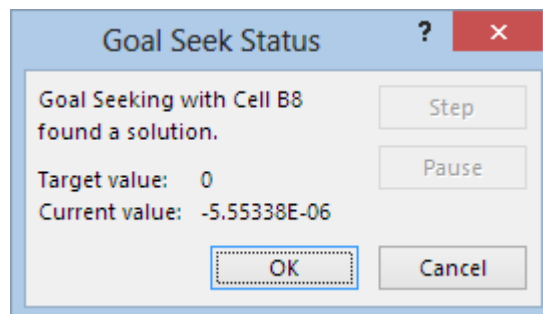


Figure 7. Solution found by *Goal Seek* using the set up of Figure 5 and the settings of Figure 6.

After pressing [ OK ], you get the following output in the spreadsheet.

	A	B
1	KNOWN VALUES:	
2	v0:	5.2
3	a:	2.3
4	y0:	1.2
5	y:	15.2
6	SOLVING FOR:	
7	t:	1.89670658
8	EQUATION:	-5.55338E-06
9		

Figure 8. Equation solution using *Goal Seek* for the settings of Figure 7 and initial guess as in Figure 5.

Notice, in Figure 8, that for the value of  $t$  found, namely, 1.89670658, the equation evaluates to  $-5.55338 \times 10^{-6}$ , which is very close to zero. For this equation, there is a second root that can be found by starting with an initial guess for  $t$  of  $-10$ . Try this solution on your own.

### Naming cells and ranges

In the examples shown above, formulas have been build using relative and absolute references to cells. It is possible to assign a name to a cell and use those names to build formulas. To assign a name to a cell, click on the cell, and then, in the *FORMULAS* tab, select the *Define Names* button. In the resulting drop-down list, as shown in Figure 7A, select the *Define Name* option.

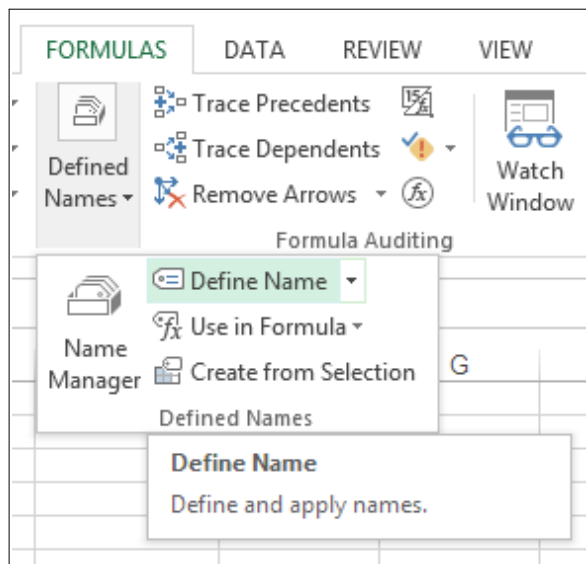


Figure 7A. Using the *Define Name* option in the *FORMULAS* tab.

In the resulting form, enter the name to assign to the cell. For example, using the spreadsheet shown in Figure 5, we can assign the names  $v0$ ,  $a$ ,  $y0$ ,  $y$ ,  $t$ , and  $EQUATION$  to cells  $B2$ ,  $B3$ ,  $B4$ ,  $B5$ ,  $B7$ , and  $B8$ , respectively. The following figure shows the form used to define  $B4$  as  $y0$ .

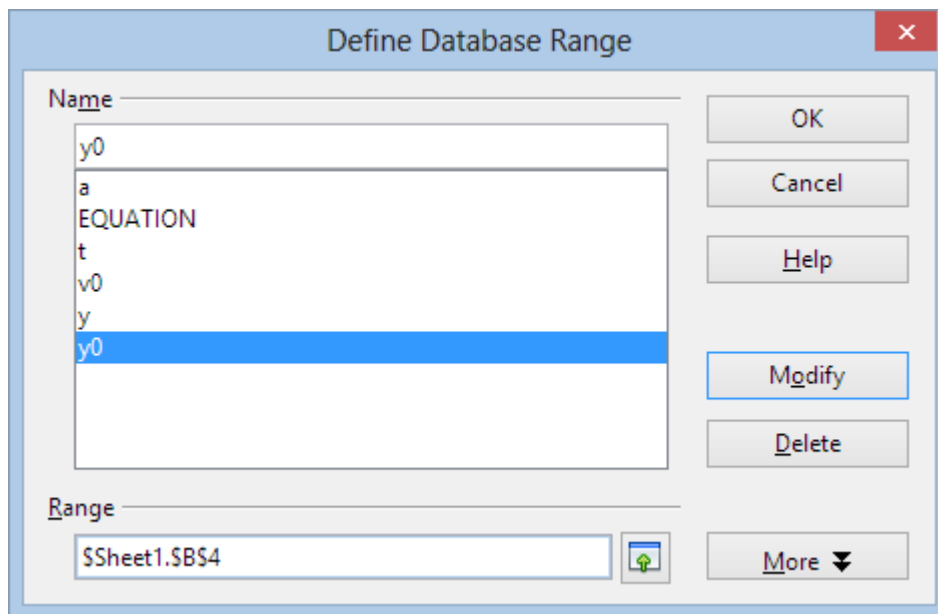


Figure 9. The *Define Database Range* form to assign names to cells or ranges.

The spreadsheet of Figure 5 is next shown using the names defined through the *Define Database Range* form of Figure 9.

EQUATION		:	X	✓	$f_x$	$=y0+v0*t+1/2*a*t^2-y$				
	A	B	C	D	E					
1	KNOWN VALUES:									
2	v0:	5.2								
3	a:	2.3								
4	y0:	1.2								
5	y:	15.2								
6	SOLVING FOR:									
7	t:	2.5								
8	EQUATION:	6.1875								

Figure 10. The spreadsheet of Figure 5 using named cells – see the equation in the entry line.

Giving names to individual cells facilitate writing formulas, as shown in Figure 10. Using the cell names can be used instead of cell references when activating *Goal Seek*, as shown in Figure 11A, below.

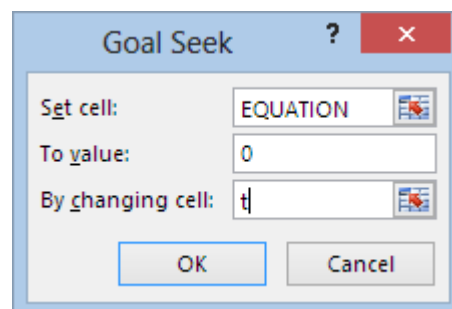


Figure 11A. The *Goal Seek* window using named cells rather than cell references.

## Using *Solver* for solving equations

The *Solver* application operates similar to the *Goal Seek* function for solving equations. Instructions for adding the *Solver* application were given in Chapter 1. After adding it, the *Solver* button should be available in the *DATA* tab. For the data of Figure 5, for example, the *Solver* interface, ready to solve the equation in cell *\$B\$8*, is shown below.

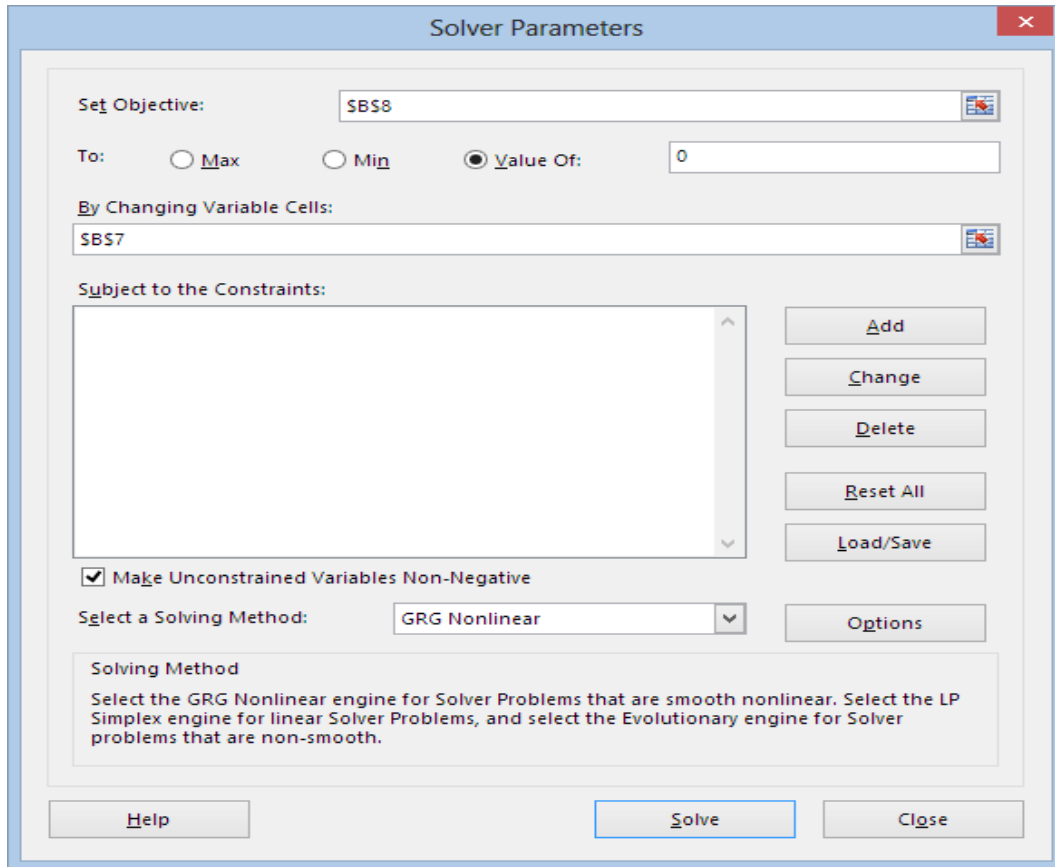


Figure 11. The *Solver* interface for the problem of Figure 5.

Pressing [ Solve ] at this point, produces the following result:

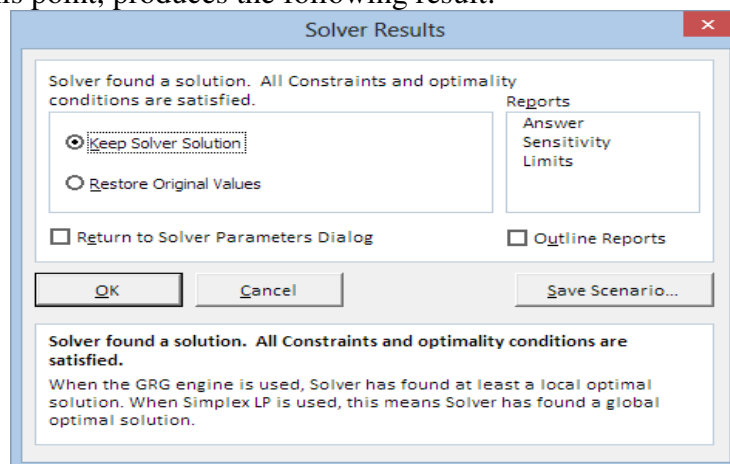


Figure 12. The *Solver Status* window for the case of Figure 5.



Press [ OK ] to show the solution in the worksheet.

As with *Goal Seek*, you can use cell names, rather than cell references, in the *Solver* application. Figure 13 shows the *Solver* form using the cell names from Figure 10 to solve the *EQUATION*.

Figure 13. The *Solver* form using cell names rather than cell references. Compare with Figure 12.

### Simple optimization example with *Solver*

In optimization problems, the idea is to minimize or maximize an *objective function* subject to a number of constraints. For example, an optimization problem can be expressed as follows:

$$\text{Max } P = 2900x_1 + 2300x_2 \text{ (this is the objective function)}$$

subject to (these are the constraints):

$$\begin{aligned} 9x_1 + 6x_2 &\leq 180 \\ 21x_1 + 18x_2 &\leq 504 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

An interface for solving this optimization is shown in the following spreadsheet:

	A	B		A	B
1	Maximize:	12700	1	Maximize:	=2900*B3+2300*B4
2	Variables:		2	Variables:	
3	x1:	2	3	x1:	6
4	x2:	3	4	x2:	21
5	LHS of constraints:	RHS of constraints:	5	LHS of constraints:	RHS of constraints:
6	36	180	6	=9*B3+6*B4	180
7	96	504	7	=21*B3+18*B4	504
8			8		
	Values			Formulas	

Figure 14. Values and Formulas for a spreadsheet setting up the optimization problem listed above.

The following figure shows the *Solver* interface for solving this problem:

Figure 15. *Solver* interface for solving the optimization problem of Figure 14.

To add the constraints, press the [ Add ] button of the form shown in Figure 15. This produces a new entry form to define individual constraints. Figure 15B, below, shows the constraints used for this case. Press [Add] after the first constraint, add the second one, then press [OK].

Figure 15B. Adding constraints to the *Solver* form to solve an optimization problem.

Since we are trying to solve for  $x_1$  and  $x_2$  as non-negative numbers, make sure that the option: ☐ *Make Unconstrained Variables Non-Negative* is selected in the *Solver* input form of Figure 15. After entering all the constraints, therefore, the *Solver* input form should contain the following information:

Subject to the Constraints:

SAS6 <= SBS6  
SAS7 <= SBS7

☒ Make Unconstrained Variables Non-Negative

Add  
Change  
Delete  
Reset All  
Load/Save

Figure 15C. Constraints in the *Solver* input form for the case of Figure 14.

Using the *Solver* input form of Figure 15, you can also select one of the various methods available to solve the optimization problem, as illustrated in Figure 15D. For a simple linear problem, as that set up in Figure 14, the *Simplex LP* method.

Select a Solving Method:

GRG Nonlinear  
Simplex LP  
Evolutionary

Options

Solving Method

Select the GRG Nonlinear engine for Solver Problems that are smooth nonlinear. Select the LP Simplex engine for linear Solver Problems, and select the Evolutionary engine for Solver problems that are non-smooth.

Figure 15D. Selecting a solving method for optimization using the *Solver* application in *EXCEL*.

After entering the problem set up (Figure 15), the constraints (Figures 15B and 15C), and the solving method (Figure 15D), press [Solve] (see Figure 15) to obtain a solution to the problem. You will get a report of a solution (see Figure 15E, below). Press the [ OK ] button to show the solution

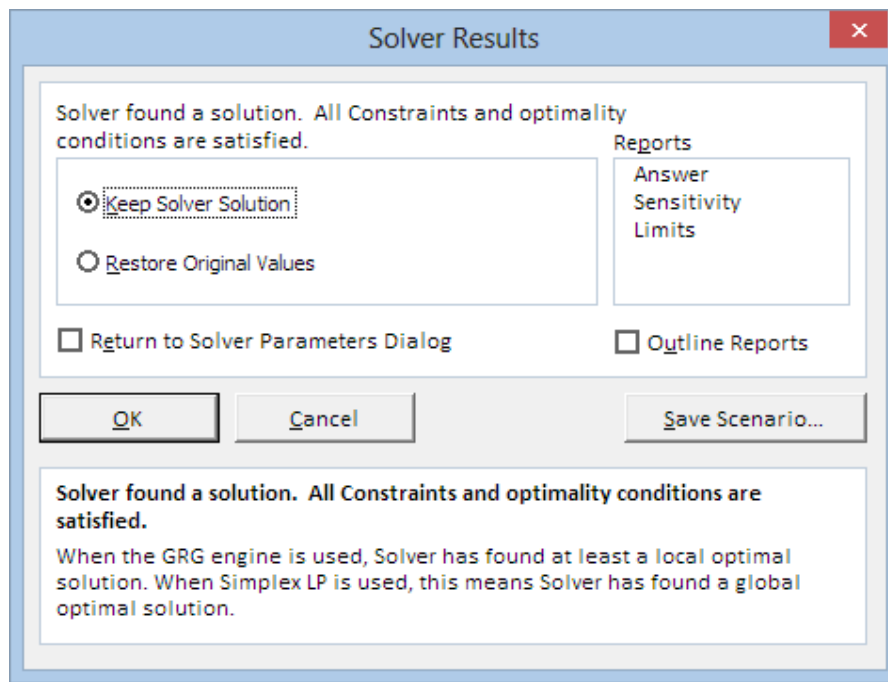


Figure 15E. Report of solution found using the *Solver* to solve the optimization problem of Figure 14.

The user can change parameters of the three different solution methods available by clicking on the [Options] button in the *Solver* input form (see Figure 15). The different options are shown in the following figure. For solving the problem of Figure 14 we used the default parameters for the Simplex LP (Linear Programming) method.

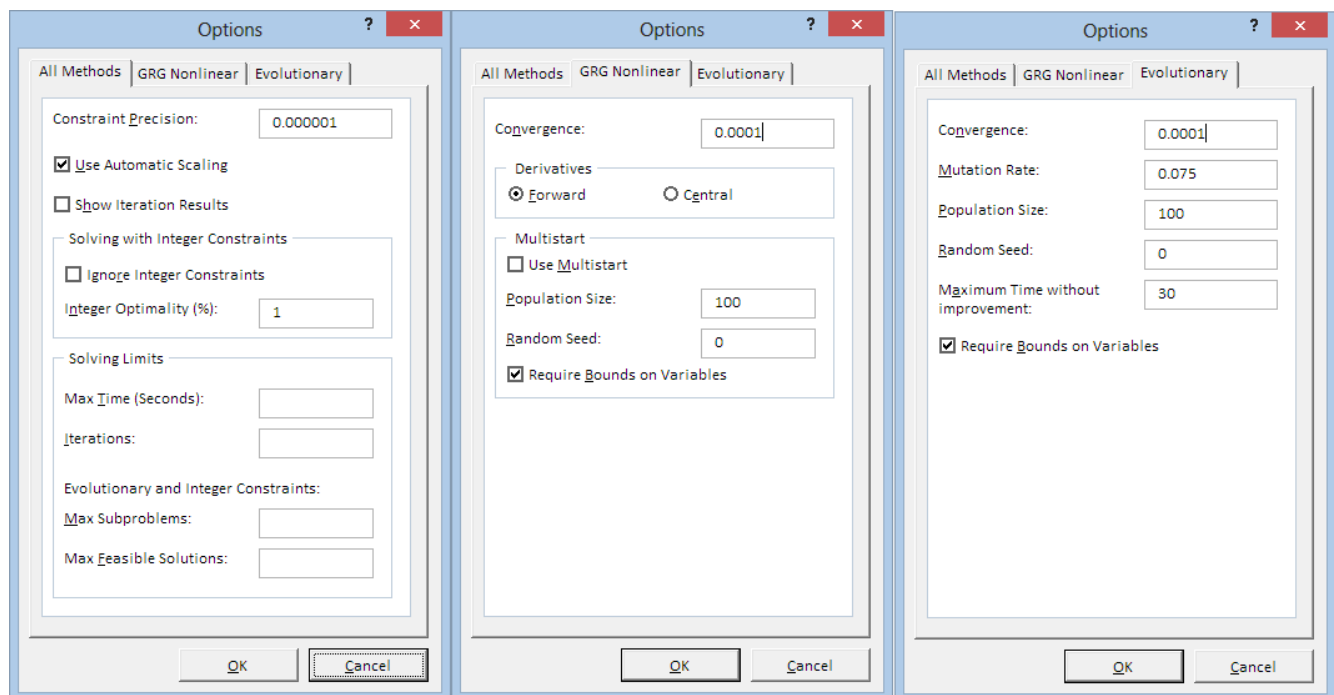


Figure 16. The *Options* windows for the *Solver* solution methods for optimization problems.

The solution of the optimization problem of Figure 14 will be shown in the worksheet, with the solution being  $x_1 = 6$  and  $x_2 = 21$ , as illustrated in the Figure below.

	A	B
1	Maximize:	65700
2	Variables:	
3	x1:	6
4	x2:	21
5	LHS of constraints:	RHS of constraints:
6	180	180
7	504	504
8		

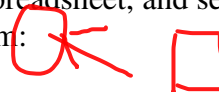
Figure 17. Solution to the optimization problem of Figure 14.

For more details on using the *Solver* feature in *EXCEL* for optimization solutions, see the following references:

- Online reference: <http://office.microsoft.com/en-us/excel-help/introduction-to-optimization-with-the-excel-solver-tool-HA001124595.aspx>
- Video example: <http://office.microsoft.com/en-us/excel-help/introduction-to-optimization-with-the-excel-solver-tool-HA001124595.aspx>

### Renaming and Adding Worksheets

Up to now, the examples presented have used a single worksheet per workbook to illustrate programming and other spreadsheet applications. The default name of the single worksheet that is available when you open a new workbook is *Sheet1*. If needed, you can change the name of the worksheet by doing a right-click on the *Sheet1* tab at the bottom of the spreadsheet, and selecting the option *Rename*. Enter the new name of the worksheet in the resulting form:



In some spreadsheet applications, we may want to use more than one worksheet in our workbook. To add a new worksheet simply click on the plus (+) sign to the right of the worksheet name tab located at the bottom of the spreadsheet. The left and right arrow buttons located in the lower left corner of a worksheet allow the user to navigate between worksheets.

### One program per worksheet: a collection of programs

Consider a workbook with 4 worksheets renamed as *INDEX*, *Triangle*, *Rectangle*, and *Circular*. The workbook with the *INDEX* worksheet is shown in Figure 18. The buttons are added by using the option *Insert > Form Controls > Button* in the *DEVELOPER* tab in *EXCEL*. Notice the three dots (...) to the right of the *INDEX* worksheet name at the bottom of the worksheet. This indicates that other worksheets exist. Clicking the right-arrow button will show the other worksheets (Figures 19-21).

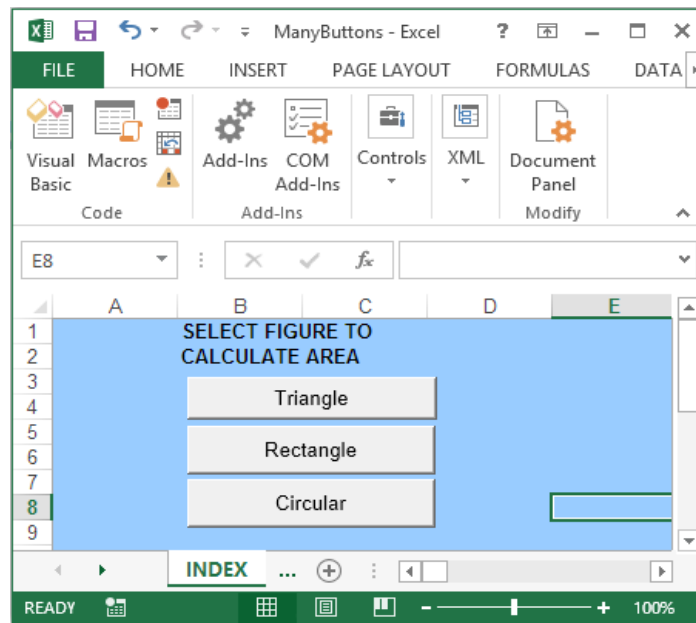


Figure 18. A workbook with 4 worksheets. The first worksheet is called *INDEX*.

Navigation buttons can be seen in Figure 18 to the left of the worksheet name *INDEX*. To move the focus to any worksheet, click on the right-arrow or left-arrow navigation buttons. When the name of the worksheet of interest shows up, click on the worksheet name to select it.

The workbook of Figure 18 is used to illustrate the use of buttons to navigate from worksheet to worksheet, as well as to illustrate some additional VBA programming. To begin with, Figures 19 through 21 show the contents of the worksheets *Triangle*, *Rectangle*, and *Circle*.

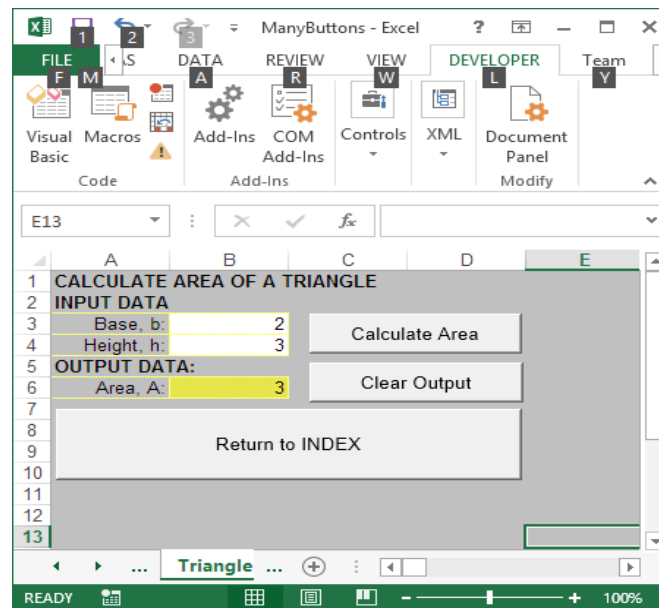


Figure 19. Contents of the *Triangle* worksheet.

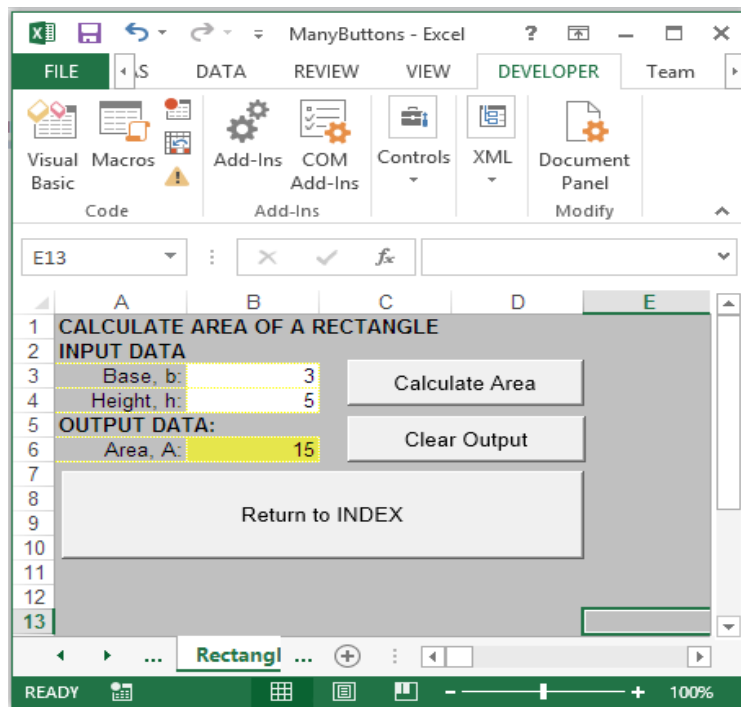


Figure 20. Contents of the *Rectangle* worksheet.

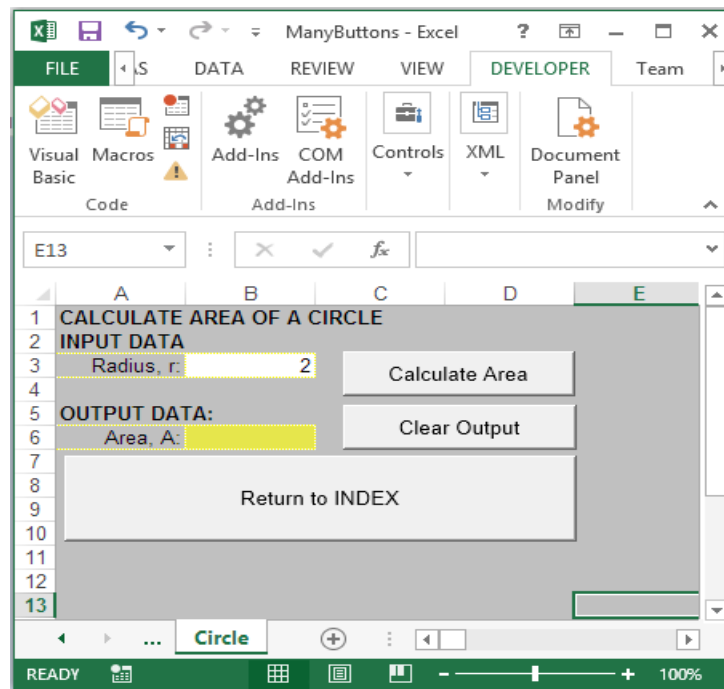


Figure 21. Contents of the *Circle* worksheet.

The operation of this workbook is as follows:

- From the *INDEX* worksheet, the buttons shown will take the user to one of the other worksheets: *Triangle*, *Rectangle*, or *Circle*.
- In each of the *Triangle*, *Rectangle*, and *Circle* worksheets you can recognize the following elements:
  - Input cells to enter values of geometric elements required to calculate an area
  - A [ *Calculate Area* ] button to read the input data, calculate the area, and show the result
  - An output cell to show the calculated area
  - A [ *Clear Output* ] button to clear the output cell
  - A [ *Return to INDEX* ] button to send the control back to the *INDEX* worksheet

ch. 1  
The buttons in each worksheet can be created as detailed in the example presented in the *Introduction to Programming EXCEL with VBA* document. For example, to enter the [ *Triangle* ] button in the *INDEX* worksheet (see Figure 18), proceed as follows:

- Open the *DEVELOPER* tab, and select the option *Insert > Form Control > Button*
- Draw the control button on the *INDEX* worksheet
- Click somewhere else on the worksheet. A form will appear asking to assign a macro to the button. At this point, simply press [CANCEL]. This cancels the macro assignment.
- Click on the button to the left of the default name, then click to the right of the name. Use the [Backspace] key to clear that default name. Finally, type the button's new name, e.g., *Triangular*.
- Repeat the steps above to draw all the buttons in the *INDEX* worksheet
- Make sure to change the labels of the buttons as shown in Figure 18.

Next, move to each of the other worksheets, and create the buttons as shown in Figures 19 through 21. Change their labels accordingly.

The next step is to create the macros associated with each button. For example, the macro to be associated with the *Triangle* button in the *INDEX* worksheet (see Figure 18), can be recorded using the *Record Macro* button in the *DEVELOPER* tab. The steps to record are as follows:

- After pressing the *Record Macro* button, enter the name *gotoTriangle* for the macro, and press [OK]
- Starting from the *INDEX* worksheet, find and select the *Triangle* worksheet
- Click on the cell *A1* in the *Triangle* worksheet
- Press the *Stop Recording* button
- Follow a similar procedure for the [ *Rectangle* ] and [ *Circle* ] buttons in the *INDEX* worksheet, saving the corresponding macros with the names *gotoRectangle* and *gotoCircle*, respectively

In each of the *Triangle*, *Rectangle*, and *Circle* worksheets, create macros to clear the output cell using the *Macro Recorder*. Basically all you have to do, while recording the macro, is:



- Click on the *Record Macro* button, and name the macro as *ClearOutputTriangle*, *ClearOutputRectangle*, or *ClearOutputCircle*, according of the worksheet name
- Click on the output cell in each worksheet while recording the macro
- Press the [*Backspace*] button in your keyboard
- Click on cell *A1*
- Press *Stop Recording*

In each of the *Triangle*, *Rectangle*, and *Circle* worksheets, create one, and only one, macro to move the control to the *INDEX* worksheet using the *Macro Recorder*. Basically all you have to do is the following:

- From either the *Triangle*, *Rectangle*, or *Circle* worksheet, and using the *DEVELOPER* tab, press the *Record Macro* button
- Name the macro *gotoIndex* and press [ OK ]
- Select the *INDEX* worksheet, and click on cell *A1* in that worksheet
- Click the *Stop Recording* button

By default, all the macros recorded will be listed in a module called *Module1* in the VBA IDE, which you can access by clicking on the *Visual Basic* button in the *DEVELOPER* tab. I added dividing lines between *Sub* to the code shown below, and cleaned up the code by eliminating extra blank lines. The code that results from recording all the macros above is shown next:

```
Sub gotoTriangle()
' gotoTriangle Macro
    Sheets("Triangle").Select
    Range("A1").Select
End Sub
'=====
Sub gotoRectangle()
' gotoRectangle Macro
    Sheets("Rectangle").Select
    Range("A1").Select
End Sub
'=====
Sub gotoCircular()
' gotoCircular Macro
    Sheets("Circle").Select
    Range("A1").Select
End Sub
'=====
Sub ClearOutputTriangle()
' ClearOutputTriangle Macro
    Range("B6").Select
    ActiveCell.FormulaR1C1 = ""
    Range("A1").Select
End Sub
'=====
Sub ClearOutputRectangle()
' ClearOutputRectangle Macro
    Range("B6").Select
```

```

        Selection.ClearContents
        Range("A1").Select
End Sub
'=====
Sub ClearOutputCircle()
' ClearOutputCircle Macro
    Range("B6").Select
    Selection.ClearContents
    Range("A1").Select
End Sub
'=====
Sub gotoIndex()
' gotoIndex Macro
    Sheets("INDEX").Select
    Range("A1").Select
End Sub

```

Next, type the VBA code for calculating the area in each of the three worksheets: *Triangle*, *Rectangle*, and *Circle* in the same *Module1* as the recorded macros. The code corresponding to these calculation macros are listed next:

```

Sub TriangleArea()
    Dim b As Double, h As Double, A As Double
    b = Worksheets("Triangle").Range("B3").Value
    h = Worksheets("Triangle").Range("B4").Value
    A = 0.5 * b * h
    Worksheets("Triangle").Range("B6").Value = A
End Sub
'=====
Sub RectangleArea()
    Dim b As Double, h As Double, A As Double
    b = Worksheets("Rectangle").Range("B3").Value
    h = Worksheets("Rectangle").Range("B4").Value
    A = b * h
    Worksheets("Rectangle").Range("B6").Value = A
End Sub
'=====
Sub CircleArea()
    Dim R As Double, A As Double, PI As Double
    PI = 4# * Atn(1#)
    R = Worksheets("Circle").Range("B3").Value
    A = PI * R ^ 2
    Worksheets("Circle").Range("B6").Value = A
End Sub
'=====

```

Press the *Macros* button in the *DEVELOPER* tab to verify that all the macros recorded and typed are available. The listing of macros is shown in Figure 22.

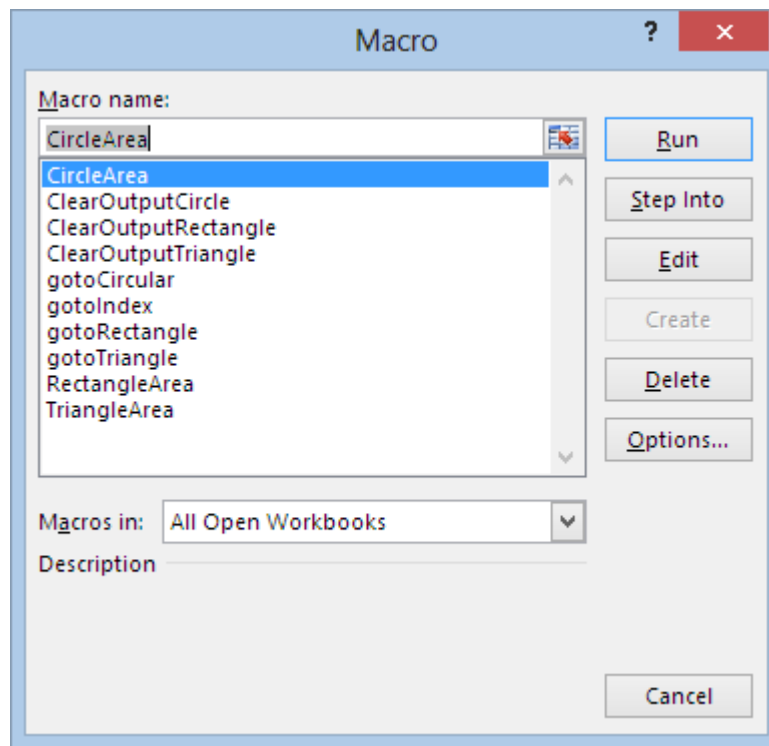


Figure 22. Macro organizer in the workbook of Figure 18.

Click [Cancel] to close the *Macro* organizer of Figure 22. The next step is to associate each button in each worksheet with a macro. Start in the *INDEX* worksheet, with the *DEVELOPER* tab active. Make sure that the button *Design Mode* is selected. Click on the [Triangle] button to select it. Then, do a right click on the button, and select the option *Assign Macro*. Click on the *gotoTriangle* to assign it to the button, and press [ OK ]. Next, repeat assigning macros *gotoRectangle*, and *gotoCircular* to the [Rectangle] and [Circular] buttons, respectively.

After assigning the macros to the buttons of the *INDEX* worksheet, proceed to assign macros to the buttons of the *Triangle*, *Rectangular*, and *Circular* worksheets. For example, for the *Triangle* worksheet, associate the buttons [Calculate Area], [Clear Output], and [Return to INDEX] with the macros *TriangleArea*, *ClearOutputTriangle*, and *gotoIndex* ~~macros~~, respectively. For the *Rectangle* worksheet, associate the buttons [Calculate Area], [Clear Output], and [Return to INDEX] with the macros *RectangleArea*, *ClearOutputRectangle*, and *gotoIndex* macros, respectively. Finally, for the *Circle* worksheet, associate the buttons [Calculate Area], [Clear Output], and [Return to INDEX] with the macros *CircleArea*, *ClearOutputCircle*, and *gotoIndex* macros, respectively. When done, make sure the *Design Mode* button in the *DEVELOPER* tab is deselected, and try the workbook buttons.

The design and operation of this workbook provides an idea for putting together workbooks where the user can select different worksheets, and each worksheet has a number of buttons to perform calculations, clear output, or move to other worksheets. This workbook illustrates the fact that we can create quite complex workbooks for programming, by utilizing multiple worksheets, and buttons linking them.

Typically, to write a program, the programmer selects or designs an *algorithm*, i.e., a plan for performing the action required from the computer. An algorithm can be simply a series of sequential steps that the computer must perform to produce a result. For example, if we intent to use the computer to add two numbers (a relatively simple operation, mind you), we can describe the corresponding algorithm in words as follows: *input the first number, input the second number, add the two numbers, store the resulting number into a memory location, show the number in the screen.*

```

graph LR
    Start((start)) --> InputA[/input a/]
    InputA --> InputB[/input b/]
    InputB --> Calculate[calculate  
c ← a + b]
    Calculate --> Display[/display c/]
    Display --> End((end))
  
```

```
graph LR
    Start((start)) --> Input[/input  
a, b/]
    Input --> Process[calculate  
c ← a + b]
    Process --> Output[/display  
c/]
    Output --> End((end))
```

```

graph TD
    Start((start)) --> Input[/input  
a, b/]
    Input --> Process[calculate  
c ← a + b]
    Process --> Output[/display  
c/]
    Output --> End((end))

```



The listing of a program written in a high-level language is also referred to as code. For example, the following figure shows the code corresponding to the flowchart shown above written as a *VBA Sub-procedure*:

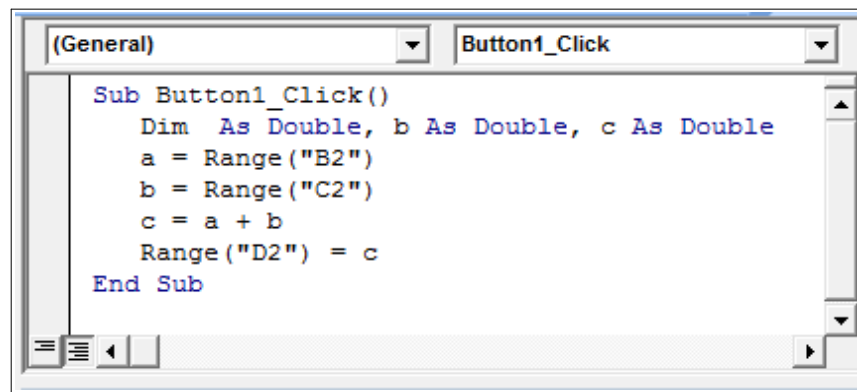


Figure 23. Example of a simple sequential program in VBA.

Notice that the *Start* and *End* symbols are represented by the *Sub* and *End Sub* statements in VBA code. The *declaration* statement *Dim a As Double*, etc., is not specifically listed in the flow chart.

While flowcharts are useful guides for describing an algorithm, producing a flowchart can become quite complicated, particularly if done by hand. Also, modifications of a hand-made flowchart can be quite involved. Luckily, flowcharting software are now available that can simplify the process (e.g., *Dia*, a free software: [http://dia-installer.de/index\\_en.html](http://dia-installer.de/index_en.html)). If one doesn't want to get involved in producing a flowchart, one can use another technique known as *pseudo-code*.

Pseudo-code simply means writing the algorithm in brief English-like sentences that can be understood by any programmer. For example, a pseudo-code corresponding to the algorithm described in the flowchart shown earlier is presented next:

*Start*  
*Input a, b*  
 $a \leftarrow b + c$   
*Display c*  
*End*

Notice the use of the algorithmic sentence  $a \leftarrow b + c$ , in both the flowchart and the pseudo-code, to indicate that the addition of  $a$  and  $b$  is to be stored in variable  $c$ . This algorithmic sentence was translated in VBA code (see Figure 23) as  $c = a + b$  because, in that computer language, the equal sign represents an *assignment* operation (i.e., the value  $a+b$  is assigned, or stored into,  $c$ ).

Since the equal sign ( $=$ ) represents assignment in most high-level languages, it is possible to write the sentence

$$n = n + 1$$

This sentence, rather than representing an algebraic equality that will result in the wrong result  $0 = 1$ , indicates that the value contained in variable  $n$  is to be incremented by  $1$ , and the resulting value is to be stored back into  $n$ .

## Basic programming structures

There are three basic programming structures: *sequence*, *decision*, and *loops*.

### Sequential structure

A *sequential structure* was used in previous sections to illustrate the use of flowcharts and pseudo-code. Sequential structures have a single entry point and a single output point, and consist of a number of steps executed one after the other. Sequential structures can be useful in simple operations such as the addition of two numbers as illustrated earlier. The following pseudo-code illustrates a sequential structure consisting of entering a number and evaluating a function given by a single expression:

```
start
  request x
  calculate  $y \leftarrow 3 \sin(x)/(\sin(x)+\cos(x))$ 
  display x, y
end
```

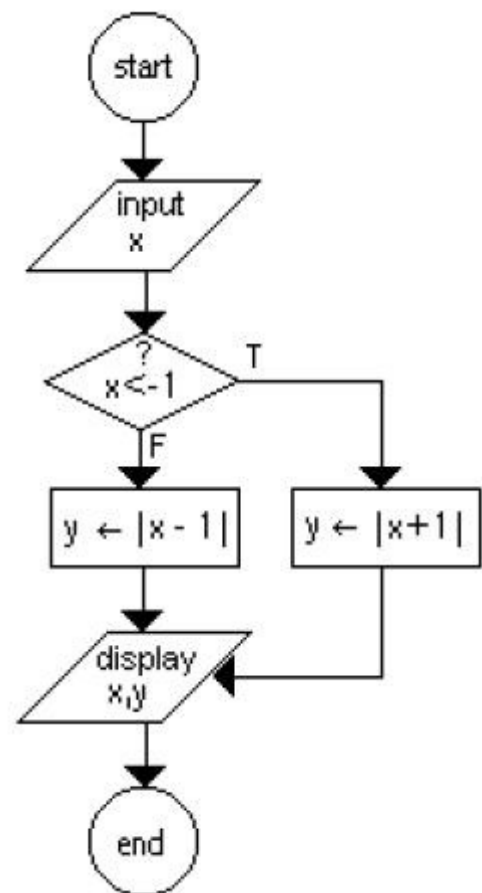
### Decision structure

A *decision structure* provides for an alternative path to the program process flow based on whether a logical statement is true or false. For example, suppose that you want to evaluate the piecewise function

$$f(x) = \begin{cases} |x+1|, & \text{if } x < -1 \\ |x-1|, & \text{if } x \geq -1 \end{cases}$$

The flowchart shown to the right indicates the process flow of a program that requests a value of  $x$  and evaluates  $y = f(x)$ . The diamond contains the logical statement that needs to be checked to determine which path (T - true, or F - false) to follow. Regardless of which path is followed from the diamond, the control is returned to the *display* statement.

Notice that the *input* statement and the decision statement form a sequence structure in this flowchart. As in this example, the three types of structures under consideration (sequence, decision, loop) do not appear alone, but two or three of them are commonly combined in many algorithms.



The algorithm illustrated above can be written in pseudo-code as follows:

```

start
  input x
  if x < -1 then
    y ← |x+1|
  else
    y ← |x-1|
  display x,y
end

```

The following function represents a possible translation of this pseudo-code into VBA code:

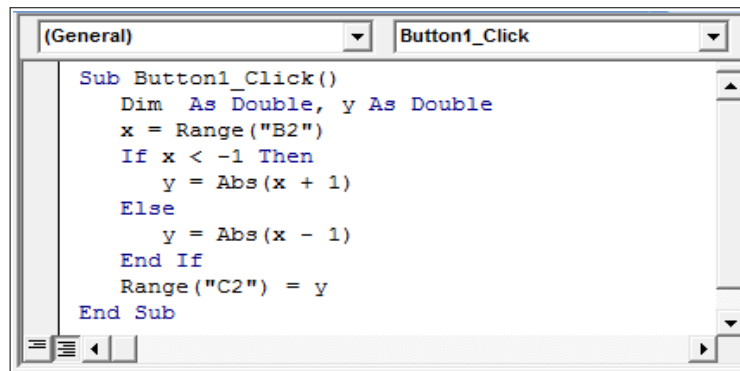


Figure 24. VBA code for a decision structure using an *IF ... Else ... End If* statement.

The decision structure shown above is such that an action is taken whether the condition tested is true or false. In some cases, if the condition tested is false, no action is taken. This is illustrated in the following flowchart that describes the definition of the function *absolute value*:

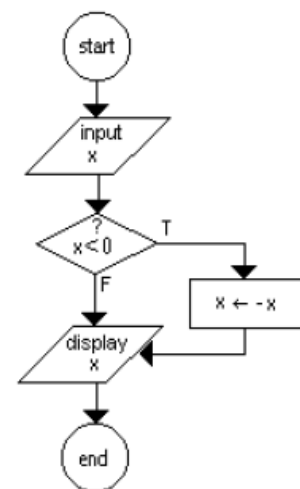
$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ -x, & \text{if } x < 0 \end{cases}$$

Notice that the value of  $x$  is redefined as  $-x$  if  $x < 0$ , but it does not change if  $x > 0$ . Thus, action is only taken if the condition  $x < 0$  is true (T). In pseudo-code, this algorithm will be written as:

```

start
  input x
  if x < 0 then
    x ← -x
  display x
end

```



A decision structure may include more than one or two possible paths. For example, if we define a function  $f(x)$  by

$$f(x) = \begin{cases} -x, & \text{if } x < 0 \\ x, & \text{if } 0 \leq x < 1 \\ x+1, & \text{if } 1 \leq x < 2 \\ 0, & \text{elsewhere} \end{cases}$$

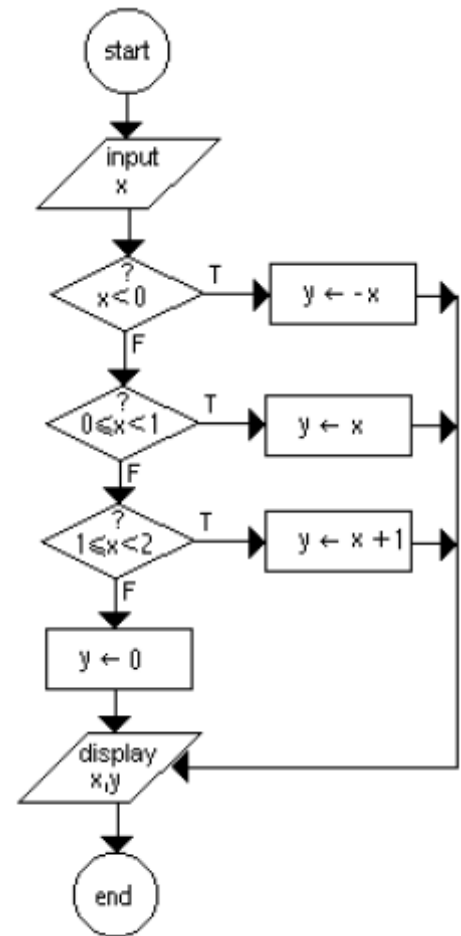
depending on the different conditions listed (e.g.,  $x < 0$ ,  $0 \leq x < 1$ , etc.), one of four actions will be taken. This decision structure is represented by the flowchart to the right. Notice that only three conditions are shown in this flowchart, with a default assignment ( $y \leftarrow 0$ ) that takes place if none of the three conditions tested is true.

In pseudo-code, such a multiple-decision structure will be written as:

```

start
  input x
  if x < 0 then
    y ← -x
  else if 0 ≤ x < 1 then
    y ← x
  else if 1 ≤ x < 2 then
    y ← x + 1
  else
    y ← 0
  display x,y
end

```



Notice that the use of the particle *else if* implies that the condition following is one of many for a specific decision structure. If we re-write this pseudo-code as follows:



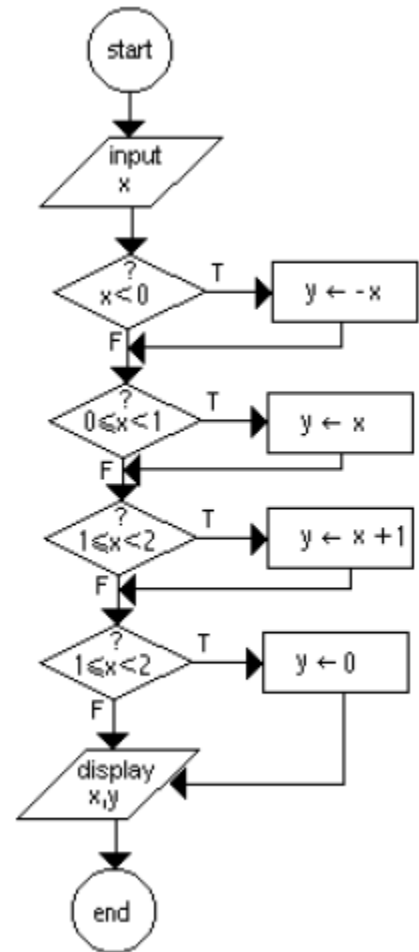
```

start
  input x
  if  $x < 0$  then
     $y \leftarrow -x$ 
  if  $0 \leq x < 1$  then
     $y \leftarrow x$ 
  if  $1 \leq x < 2$  then
     $y \leftarrow x + 1$ 
  if  $x \geq 2$  then
     $y \leftarrow 0$ 
  display x,y
end

```

In this case, the meaning of the decision structure changes, even though the result may not change. The corresponding flowchart is shown to the right.

From this flowchart, it is clear that the algorithm shown represents four simple decision structures, rather than a single decision structure with four possible outcomes as depicted earlier.



VBA codes for the two algorithms presented above are shown in the following Figure.

<pre> Option VBASupport 1 Option Explicit  Sub Decision01   Dim x As Double, y As Double   x = InputBox("Enter x")   If x &lt; 0 Then     y = -x   ElseIf ((0 &lt;= x) And (x &lt; 1)) Then     y = x   ElseIf ((1 &lt;= x) And (x &lt; 2)) Then     y = x + 1   Else     y = 0   End If   MsgBox("f(" &amp; CStr(x) &amp; ") = " &amp; CStr(y)) End Sub </pre>	<pre> Option VBASupport 1 Option Explicit  Sub Decision02   Dim x As Double, y As Double   x = InputBox("Enter x")   If x &lt; 0 Then     y = -x   EndIf   If ((0 &lt;= x) And (x &lt; 1)) Then     y = x   EndIf   If ((1 &lt;= x) And (x &lt; 2)) Then     y = x + 1   EndIf   MsgBox("f(" &amp; CStr(x) &amp; ") = " &amp; CStr(y)) End Sub </pre>
---	---

Figure 25. Two decision-structure codes based on the algorithms shown above.

Most high-level computer languages include a *switch-case* or *case-select* structure to code a multiple-decision structure as described above. VBA provides a *case-select* structure, in which one out of many possible outcomes is selected depending on the value of a selector variable. An example of a *case-select* structure is shown in VBA code below as *Sub Decision 03*

```
Option VBASupport 1
Option Explicit

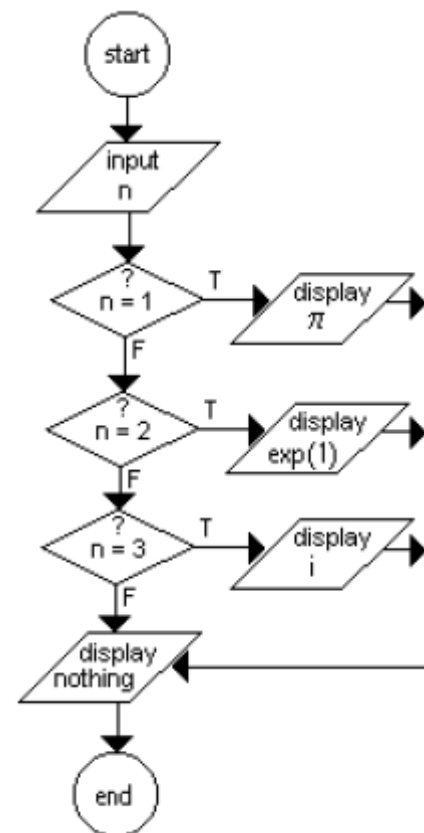
Sub Decision03
    Dim n As Integer, PI As Double
    PI = 4.0 * Atn(1.0)
    n = InputBox("Select a case: 1 - pi, 2 - e, 3 - i")
    Select Case n
        Case 1
            MsgBox("You selected pi = " & CStr(PI))
        Case 2
            MsgBox("You selected e = " & Exp(1.0))
        Case 3
            MsgBox("You selected i = 0 + 1*i")
        Case Else
            MsgBox("Wrong selection")
    End Select
End Sub
```

Figure 26. Example of a VBA *Case-Select* decision structure.

A flowchart for this case-select structure is shown to the right. Notice that this flow chart is not different from a multiple-decision structure. In pseudo-code, this flowchart could be translated as follows:

```
start
input n
if n = 1 then
    display p
if n = 2 then
    display exp(1)
if n = 3 then
    display i
else
    display nothing
end
```

The *case-select* structure can be incorporated into the pseudo-code as shown next:



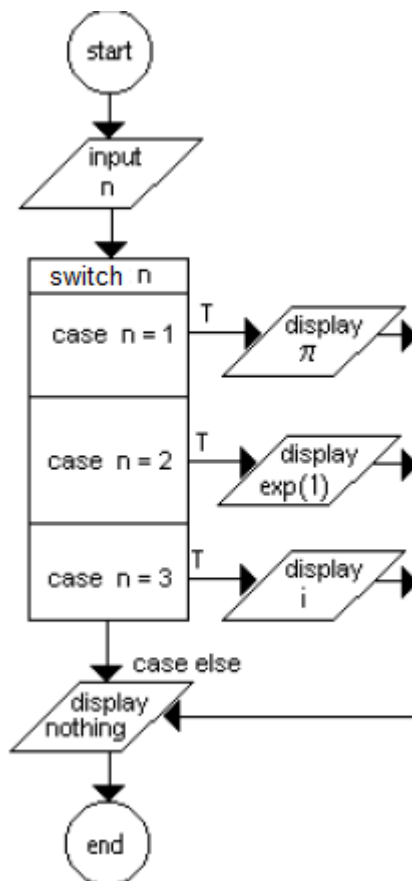
```

start
  select case n
    case n = 1, display p
    case n = 2, display exp(1)
    case n = 3, display i
    else, display nothing
  end

```

Thus, the *case-select* structure is a way to code a multiple decision structure and not a different programming structure.

If the programmer decides to use the *case-select* structure from the start, it is possible to create a *case-select* flowchart symbol as illustrated in the flowchart below. Notice that this is not a standard flowchart symbol, but a made-up one to incorporate a *case-select* structure in the flowchart.



### Loop structure

A *loop structure* represents a repetition of a statement or statements a finite number of times. For example, suppose that you want to calculate the following summation:

$$S_n = \sum_{k=1}^n \frac{1}{n}.$$

The algorithm for the calculation is illustrated in the flowchart shown to the right. Notice that the loop structure is part of a sequence structure and that it contains a decision structure within, thus, re-emphasizing the fact that the three basic structures (sequence, decision and loops) commonly appear together in many algorithms. Notice also that the loop structure requires an index variable, in this case  $k$ , to control when the process will leave the loop.

The summation  $S_n$  and the index variable  $k$  are both initialized to zero before the control is passed to the loop structure. The first action within the loop structure is to increment the index variable  $k$  by 1 ( $k \leftarrow k + 1$ ). Next, we check if the value of  $k$  has not grown beyond that of  $n$  ( $k > n$ ?). If  $k$  is still less than  $n$ , the control is passed to incrementing the summation ( $S_n \leftarrow S_n + 1/k$ ), and back to the first step in the loop. The process is then repeated until the condition  $k > n$  is satisfied. At this point, the control is passed on to reporting the results  $n$ ,  $S_n$ .

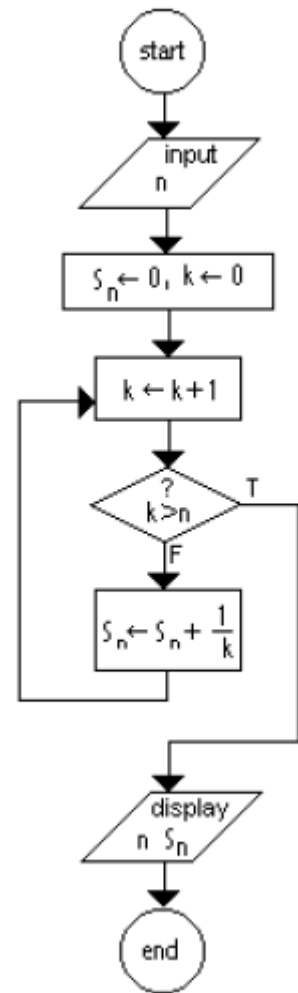
The pseudo-code corresponding to the flowchart shown above is the following:

```

start
  input n
   $S_n \leftarrow 0$ 
   $k \leftarrow 0$ 
  do while  $\sim(k > n)$ 
     $k \leftarrow k + 1$ 
     $S_n \leftarrow S_n + 1/k$ 
  end loop

  display n,  $S_n$ 
end

```



Notice that instead of translating the loop structure in the flowchart with an *if* statement, we used the statement *do while*. (*Do-while* statements are commonly available in most high-level computer programming languages). Notice that a condition, namely  $\sim(k > n)$ , is attached to the *do while* statement in the pseudo-code. The statement is to be read as “do (the statements in the loop) while  $k$  is not larger than  $n$ ”. After the condition in the loop structure is no longer satisfied, i.e., when  $k > n$ , then the loop ends and the control is sent to the statement following the end of the loop.

**[Note:** The symbol  $\sim$  represents negation in mathematical logic notation, thus, if  $p$  stands for the statement  $x > 0$ ,  $\sim p$  stands for  $\sim(x > 0)$  or  $x \leq 0$ ].

A possible VBA coding of this process that uses VBA's *while* statement is shown next. The end of the loop is signaled by the particle *end*:

```

Option VBASupport 1
Option Explicit

Sub Loop01
    Dim n As Integer, k As Integer, Sn As Double
    n = InputBox("Enter n")
    Sn = 0 : k = 0
    Do While k <= n
        k = k + 1
        Sn = Sn + 1/k
    Loop
    MsgBox("For n = " & CStr(n) & ", Sn = " & CStr(Sn))
End Sub

```

Figure 27. Example of repetition structure using a VBA *Do While ... Loop* statement.

Most high-level computer programming languages provide a *for* loop structure by which an index variable is initialized, an increment to the index variable is provided, and a maximum value of the index variable is specified. For example, in VBA the *For ... Next* structure that we would use to calculate the summation  $S_n$  indicated above is shown next:

```

Option VBASupport 1
Option Explicit

Sub Loop02
    Dim n As Integer, k As Integer, Sn As Double
    n = InputBox("Enter n")
    Sn = 0
    For k = 1 To n
        Sn = Sn + 1/k
    Next
    MsgBox("For n = " & CStr(n) & ", Sn = " & CStr(Sn))
End Sub

```

Figure 28. Example of repetition structure using a *For ... Next* statement.

The *For* statement in this case uses the specification  $k = 1 \text{ To } n$  to produce a range of values  $1, 2, \dots, n$ . [If the increment were to be different than 1, say, 2, the *For* statement would read: *For k = 1 To n Step 2*]. The general form of the range specified in a *For* statement is, therefore:

$$\text{For } k = k_0 \text{ To } k_f \text{ Step } \Delta k$$

where  $k_0$  = initial value,  $k_f$  = upper limit, and  $\Delta k$  = increment. Assuming  $\Delta k > 0$  and  $k_0 \leq k_f$ , the specification *For k = k<sub>0</sub> To k<sub>f</sub> Step  $\Delta k$*  will produce the values:

$$k_0, k_0 + \Delta k, k_0 + 2\Delta k, \dots, k_u,$$

where  $k_u$  is the such that  $k_u < k_f$  and  $k_u + \Delta k > k_f$ . 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,  $k_u = 0.95$ , which satisfies  $k_u < 1$  and  $k_u + \Delta k = 1.25 > 1$ .

The number of elements  $N$  in the range generated by the statement  
*For*  $k = k_0$  *To*  $k_f$  *Step*  $\Delta k$  is

$$N = \left\lfloor \frac{k_f - k_0}{\Delta k} \right\rfloor + 1 \quad [9]$$

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*  $k = k_0$  *To*  $k_f$  *Step*  $\Delta k$ ), it is possible to have a negative increment, i.e.,  $\Delta k < 0$ , in which case we must have  $k_0 \geq k_f$ . For example, the range *For*  $k = -0.25$  *To*  $-1$  *Step*  $-0.20$  includes the values  $-0.25$ ,  $-0.45$ ,  $-0.65$ ,  $-0.85$ .

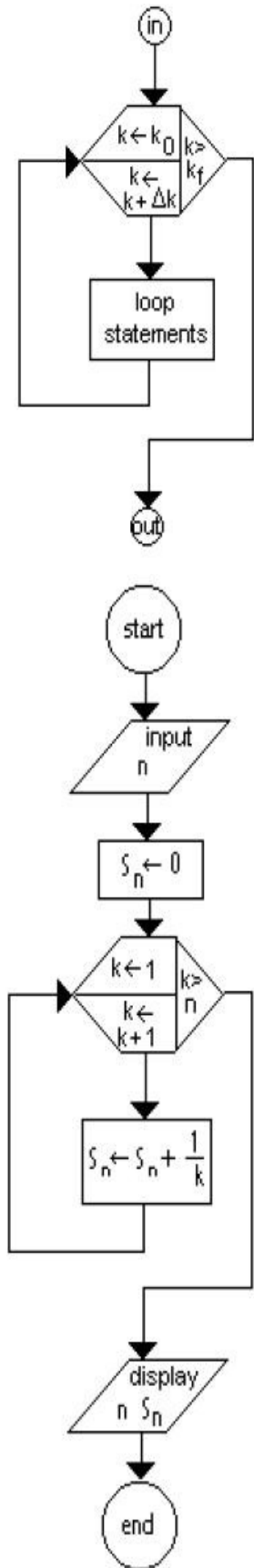
The number of elements in the range is still calculated by using equation [9], above, i.e.,

$$N = [(-1 - (-0.25))/-0.2] + 1 = 4.$$

#### 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*  $k = k_0$  *To*  $k_f$  *Step*  $\Delta k$  when  $\Delta k > 0$ , and  $k_0 \geq k_f$ , or when  $\Delta k < 0$  and  $k_0 \leq k_f$ , 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*  $k = k_0$  *To*  $k_f$  *Step*  $\Delta k$  when  $\Delta k < 0$ , and  $k_0 \geq k_f$ , or when  $\Delta k > 0$  and  $k_0 \leq k_f$ , the resulting range is infinitely large. Thus, if used to control a loop structure, the program will enter an infinite loop (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.

As we did with the *case-select* structure in flowcharts, we can create our own flowchart symbol for a *For* loop. One possibility is the hexagonal flowchart symbol shown in the flowchart at the top-right. Right below the *in* symbol the hexagonal symbol shows the initialization of the index variable ( $k \leftarrow k_0$ ). Right below this initialization, within the hexagonal symbol, the increment of the index variable is presented ( $k \leftarrow k + \Delta k$ ). The increment step connects to the loop statements, indicating that these statements are repeated as long as the condition  $k > k_f$  is not reached. This condition is shown in the right section of the hexagonal symbol indicating that, when it is satisfied, the control is sent *out* of the loop.



Thus, for the case of the summation program presented earlier, we can use the flowchart shown at the bottom right of last page. The corresponding pseudo-code will look like this:

```

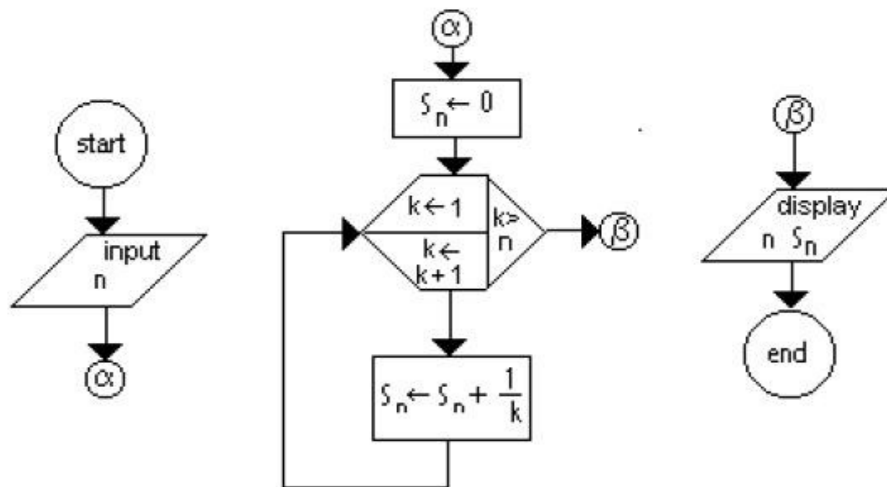
start
  input n
   $S_n \leftarrow 0$ 
  for  $k = 1$  To  $k > n$  Step  $Dk$ 
     $S_n \leftarrow S_n + 1/k$ 
  end For loop
  display n,  $S_n$ 
end

```

Recall that the hexagonal *For* loop symbol is not a standard flowchart symbol, but one made up to represent the *For* programming structures that are available in most high-level computer programming languages.

### Splitting the flowchart

Sometimes, a complete flowchart will not fit in the paper. In such cases, you can split the flowchart by creating sub-flowcharts linked by circles identified with numbers or letters. The flowchart shown in next page is the same flowchart to the right, but split into three parts. The links are identified with Greek letters.



### Hierarchy charts

Hierarchy charts are charts that show the different components of a program identified by tasks. A hierarchy chart is not as detailed as a flowchart or pseudo-code, instead, it shows the relationship between different components of a program in a concise manner. The hierarchy chart is used to strategize the coding of a program by splitting the program into self-contained, but interrelated, components. We will learn that such components can be coded as *Sub* or *Function* subprograms in *VBA*. Once a program is divided into its components, flowcharts or pseudo-code can be developed for the individual components before coding the program.

The following is an example of a hierarchy chart developed to solve the problem of uniform flow in a trapezoidal open channel (a common problem in civil engineering hydraulics). The solution was produced in Visual Basic 6.0 back in the early 2000's. The chart shows all the Sub procedures and Function procedures (one only, *fManning*) included in a Visual Basic 6.0 project called *prjManning*.

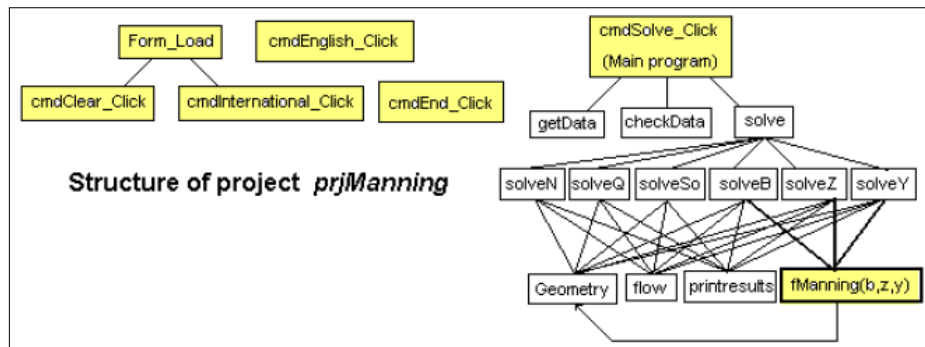


Figure 29. An example of a hierarchy chart for a Visual Basic 6.0 project.

### Subroutines or sub-functions

A sub-routine is a program that is called from another program. In VBA there are two types of programs, which can act as subprograms, namely, *Subs* and *Functions*. For example, in Figure 30, the Sub *MyMain*, below, which we can refer to as the *main program*, calls function *addTwo* to calculate the sum of variables *x* and *y*. Thus, function *addTwo* is a *sub-function* of *MyMain*.

```
Option VBASupport 1
Option Explicit

Sub MyMain()
    Dim x As Double, y As Double, z As Double
    x = InputBox("Enter x")
    y = InputBox("Enter y")
    z = addTwo(x,y)
    MsgBox("The sum of " & CStr(x) & " + " & _
        CStr(y) & " is equal to " & CStr(z))
End Sub

Function addTwo(r As Double, t As Double) As Double
    addTwo = r + t
End Function
```

Figure 30. Example of a main program (Sub *MyMain*) with a subprogram (Function *addTwo*).

A main program and subprograms can be coded in the same module, or in separate modules. In the example in Figure 30, both the main program and the subprogram were coded in the same module. The links in the diagram of Figure 30 indicates how function *addTwo* is called from within function *MyMain*.

The parameters of Function *addTwo* are the variables *r* and *t*. In the call to Function *addTwo*, within Sub *MyMain*, the function arguments are *x* and *y*. Thus, *x* takes the place of *r* and *y* takes the place of *t* in the function. We say that *x* and *y* are *passed on to the function*. The name of the function, *addTwo*,



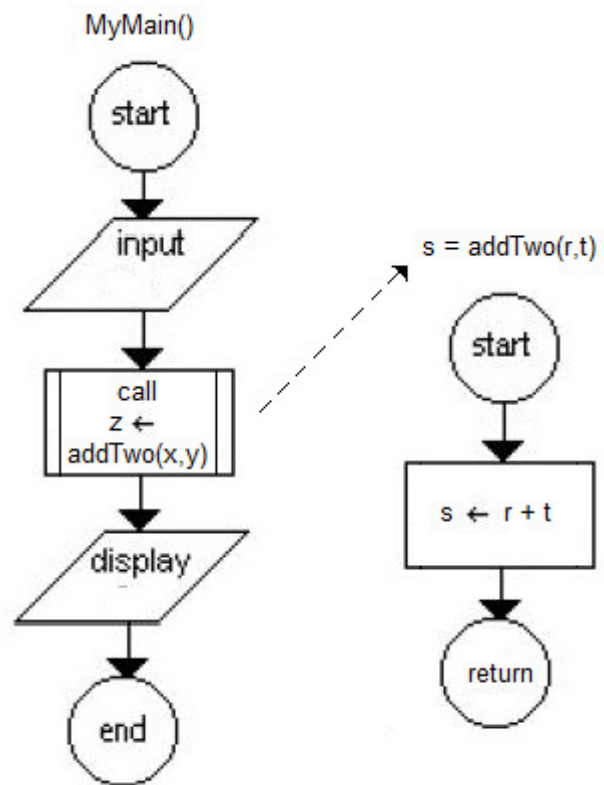
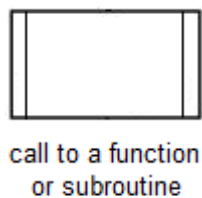
gets assigned the value  $s + t$ , within the function body. This is the value that *gets returned back to the main program*, and assigned to variable  $z$  in the statement  $z = \text{addTwo}(x,y)$ . Variables  $r$  and  $t$  are referred to as *dummy variables* since they can be replaced by any other variables or values in evaluating the function.

A flow chart illustrating the relationship between Sub *MyMain* and Function *addTwo* is presented below. The corresponding pseudo-code is:

```
Sub MyMain
  input x, y
  z ← addTwo(x,y)
  display x, y, z
end
```

```
Function addTwo(r,t)
  s = r+t
  return s
```

In the diagram to the right, we introduced, therefore, the symbol for a subroutine or sub-function call, namely:



### Comparison Statements in VBA

Decision statements (and some repetition statements) in VBA require the use of *Boolean* (or *logical*) *statements*. These are statements that evaluate to *TRUE* or *FALSE*. The simplest type of Boolean (or logical) statements are comparison statements between numerical data. These comparison statements can be used to compare the numerical values of two constants, or two variables, or a variable and a constant. Comparison statements use the following comparison symbols:

- *is equal to:* = (same as the assignment sign)
- *is not equal to:* < >
- *is greater than:* >
- *is greater than or equal to:* >=
- *is less than:* <
- *is less than or equal to:* <=

Some examples of comparison statements follow:

$2 == 3$  (F),  $2 < 3$  (T),  $2 <= 3$  (T),  $2 > 3$  (F),  $2 >= 3$  (F),  $x == 2$ ,  $x < 2$ ,  $x <= 2$ ,  $x > 2$ ,  $x >= 2$ ,  $x == y$ ,  $x < y$ ,  $x <= y$ ,  $x > y$ ,  $x >= y$ ,  $z == x+y$ ,  $z + y < x$ , etc.

An example of a simple comparison statement used in an *If* statement in VBA was shown earlier in Figure 24.

### Boolean (or logical) statements

Comparison statements, since they can be evaluated as *TRUE* or *FALSE*, are also *Boolean* (or *logical*) *statements*. However, more complex Boolean statements can be constructed by affecting a logical statement through a *Negation* (*Not*), or joining two comparison statements through one of the following logical particles:

- *Conjunction*: And
- *Disjunction*: Or
- *Exclusive Or*: Xor

To understand the effect of the logical particles *Not*, *And*, *Or*, and *Xor*, we need to use some concepts from mathematical logic:

- Let  $p$  and  $q$  be two logical statements (these could be, for example, comparison statements), which could be *TRUE* (T) or *FALSE* (F).
- In mathematical logic notation, the following symbols are used for logical operations:
  - Negation:  $\sim$  (not) example:  $\sim p$  (not p)
  - Conjunction:  $\wedge$  (and) example:  $p \wedge q$  (p and q)
  - Disjunction:  $\vee$  (or) example:  $p \vee q$  (p or q)
  - Exclusive or:  $\otimes$  (xor) example:  $p \otimes q$  (p xor q)
- The logical value (T or F) of a logical statement depends on the logical value of the statement components (p, q) and on the operation involved. The logical values of the various logical operations shown above can be shown in *true tables*, as follows:

Not		And			Or			Xor		
p	$\sim p$	p	q	$p \wedge q$	p	q	$p \vee q$	p	q	$p \otimes q$
T	F	T	T	T	T	T	T	T	T	F
F	T	T	F	F	T	F	T	T	F	T
		F	T	F	F	T	T	F	T	T
		F	F	F	F	F	F	F	F	F

Thus, negation ( $\sim$ , not) simply changes the logical value of a statement, a conjunction ( $\wedge$ , and) is true only if both components are true, a disjunction ( $\vee$ , or) is true as long as one of the components is true, and an exclusive or ( $\otimes$ , xor) is true only if both components have different logical value.

Logical statements formed by combining simple comparison statements using *Not*, *And*, *Or*, and *Xor*, can be used in control statements for branching and looping structures in VBA programming. For

example, the condition  $0 < x < 5$ , can be coded as:  $(0 < x) \text{ And } (x < 5)$ . The condition,  $|x| \geq 5$ , on the other hand, can be coded as  $(x \leq -5) \text{ Or } (x \geq 5)$ .

### Basic VBA Data Types – Boolean Variables

A variable can be defined as *Boolean* by using a *Dim* statement, e.g., *Dim myAnswer As Boolean*. A *Boolean* variable can only take the value of *TRUE* (T) or *FALSE* (F). A variable that has been defined as *Boolean* can be assigned a value as illustrated below:

```
myAnswer = TRUE  
myAnswer = FALSE
```

### Branching structures in VBA

The simplest branching structure in VBA is a simple If... End If statement, whose general form is:

```
If <logical statement> Then  
    <statement(s)>  
End If
```

In this programming structure, if <logical statement> is true, then the <statement(s)> in the body of the *If... End If* structure are executed, otherwise, no action is taken.

Example: The absolute value function (*Abs*) is a built-in mathematical function in VBA. However, you could define your own absolute, according to the pseudo-code shown at the bottom of page 19, using a simple *If... End If* statement:

```
Function myAbs(x As Double) As Double  
    If x < 0 Then  
        x = -x  
    End If  
    myAbs = x  
End Function
```

Figure 31. Example of a simple *If... End If* statement.

The If... Else ... End If statement allows for two possible actions. The general form of this structure is:

```
If <logical statement> Then  
    <statement(s) 1>  
Else  
    <statement(s) 2>  
End If
```

In this programming structure, if <logical statement> is true, then the <statement(s) 1> are executed, otherwise, the <statement(s) 2> are executed.

Examples: (1) Figure 24 shows an example of a *If ... Else ... End If* statement used to evaluate a piecewise function. (2) The following figure shows an example of a *If ... Else ... End If* statement used for classifying a flow as subsonic or supersonic based on the value of the Mach number,  $M$ :

```
Sub ClassifyMach()
    Dim M As Double
    M = CDbl(InputBox("Enter Mach number:"))
    If M <= 1 Then
        MsgBox("For M = " & CStr(M) & ", the flow is Subsonic")
    Else
        MsgBox("For M = " & CStr(M) & ", the flow is Supersonic")
    End If
End Function
```

Figure 32. Example of a *If ... Else ... End If* statement.

For a branching structure with more than 2 possible outcomes you can use the *If ... Else If ... Else ... End If* statement:

```
If <logical statement 1> Then
    <statement(s) 1>
Else If <logical statement 2> Then
    <statement(s) 2>
Else If <logical statement 3> Then
    <statement(s) 3>
...
Else
    <statement(s) default>
End If
```

In this programming structure, if <logical statement 1> is true, then the <statement(s) 1> are executed, otherwise, if <logical statement 2> is true, then the <statement(s) 2> are executed, and so on through the various *Else If* statements. If none of the logical statements in the *If* or in the *Else If* statements are true, then the <statement(s) default> are executed.

Example: The classification of Figure 32 is incomplete. In such case, we arbitrarily grouped the option  $M=1$  with the *subsonic* classification, while, in reality, that should be a separate option called *sonic* flow. Thus, we can expand the *If .. Else ... End If* statement into a *If ... Else If .. Else ... End If* statement as follows:

```
Sub ClassifyMach()
    Dim M As Double
    M = CDbl(InputBox("Enter Mach number:"))
    If M <= 1 Then
        MsgBox("For M = " & CStr(M) & ", the flow is Subsonic")
    Else If M = 1 Then
        MsgBox("For M = " & CStr(M) & ", the flow is Sonic")
    Else
        MsgBox("For M = " & CStr(M) & ", the flow is Supersonic")
    End If
End Function
```

Figure 33. Example of a *If ... Else If .. Else ... End If* statement

### Nested If statements

If ... Else ... End If statements can be nested as illustrated in the following example. In this case we produce a classification of flows based on their values of a couple of parameters, their Reynolds number (range from 0 to  $10^8$ ) and their Mach number (typical range, 0 to 20). The classification is performed according to the following criteria:

- If the Reynolds number is less than 2000, the flow is *laminar*, otherwise is classified as *non-laminar*.
- If the Mach number is less than 1, the flow is *subsonic*; if it is equal to 1, the flow is *sonic*, and if it is larger than 1, the flow is *supersonic*.

In the example of nested loops shown below, the outer If ... Else ... End If statement decides on the Reynolds-number-based classification, and the interior If ... Else ... End If statement decides on the Mach-number-based classification. The large brackets enclosing the If statements were drawn to emphasize the *nesting* of the If statements.

```
Sub NestedIf
    Dim Ry As Double, Mc As Double, OutString As String
    OutString = "This flow is classified as "
    Ry = InputBox("Enter a Reynolds number (0 to 10^8):")
    Mc = InputBox("Enter a Mach number (0.01 to 10):")
    If Ry < 2000 Then
        OutString = OutString & "laminar and "
        If Mc < 1 Then
            OutString = OutString & " subsonic."
        ElseIf Mc = 1 Then
            OutString = OutString & " sonic."
        Else
            OutString = OutString & " supersonic."
        End If
    Else
        OutString = OutString & "non-laminar and "
        If Mc < 1 Then
            OutString = OutString & " subsonic."
        ElseIf Mc = 1 Then
            OutString = OutString & " sonic."
        Else
            OutString = OutString & " supersonic."
        End If
    End If
    MsgBox(OutString)
End Sub
```

Figure 34. Example of nested If ... Else ... End If statements.

Notice that the structure of the nested If ... Else ... End If statements of Figure 34 is such that the following cases are present:

- $Ry < 2000$  and  $Mc < 1$ : laminar and subsonic
- $Ry < 2000$  and  $Mc = 1$ : laminar and sonic

- $Ry < 2000$  and  $Mc > 1$ : laminar and supersonic
- $Ry > 2000$  and  $Mc < 1$ : non-laminar and subsonic
- $Ry > 2000$  and  $Mc = 1$ : non-laminar and sonic
- $Ry > 2000$  and  $Mc > 1$ : non-laminar and supersonic

When listed in this manner, the different cases can be handled using a single, but complex, *If... ElseIf... Else... End* statement, e.g.,

```
Sub NestedIfModified
    Dim Ry As Double, Mc As Double, OutString As String
    OutString = "This flow is classified as "
    Ry = InputBox("Enter a Reynolds number (0 to 10^8):")
    Mc = InputBox("Enter a Mach number (0.01 to 10):")
    If Ry < 2000 And Mc < 1 Then
        OutString = OutString & "laminar and subsonic."
    ElseIf Ry < 2000 And Mc = 1 Then
        OutString = OutString & "laminar and sonic."
    ElseIf Ry < 2000 And Mc > 1 Then
        OutString = OutString & "laminar and supersonic."
    ElseIf Ry > 2000 And Mc < 1 Then
        OutString = OutString & "non-laminar and subsonic."
    ElseIf Ry > 2000 And Mc = 1 Then
        OutString = OutString & "non-laminar and sonic."
    Else
        OutString = OutString & "non-laminar and supersonic."
    End If
    MsgBox(OutString)
End Sub
```

Figure 35. Single *If... ElseIf... Else... End If* statement producing the same results as the nested *If... Else... End If* statements of Figure 34.

Notice that the *If* and *ElseIf* statements in Figure 35 are associated with logical statements, such as  $Ry < 2000$  And  $Mc = 1$ , etc., rather than the simple comparison statements of Figure 34. Logical statements that combine two or more comparison statements can use parenthesis to isolate the component statements, as shown in the example of Figure 25.

### Back to EXCEL: Using the EXCEL IF function

An *IF* function is provided with *EXCEL* that can be used to make a decision based on a logical statement that may be related to the contents of cells or to an arithmetic expression. The general form of the *IF* statement is:

$IF(logical\_statement, statement\ 01, statement\ 02)$

The *IF* function works similar to an *If... Else... End If* statement in VBA, i.e., if *logical\_statement* is true, then *statement 01* gets executed, otherwise *statement 02* gets executed. Here's an application of the *IF* function used to evaluate the function first shown in page 18, namely:

$$f(x) = \begin{cases} |x+1|, & \text{if } x < -1 \\ |x-1|, & \text{if } x \geq -1 \end{cases}$$

The following figure shows applications of the *IF* function to evaluate the function  $f(x)$  shown above.

	B	C		B	C
13	x	f(x)	13	x	f(x)
14	-3	2	14	-3	=IF(B14<-1,ABS(B14+1),ABS(B14-1))
15	5	4	15	5	=IF(B15<-1,ABS(B15+1),ABS(B15-1))

(a) Evaluation (b) Formulas

Figure 36. Use of the *IF* function in EXCEL to evaluate at piecewise function

For a more complex piece-wise function, such as the following,

$$f(x) = \begin{cases} -x, & \text{if } x < 0 \\ x, & \text{if } 0 \leq x < 1 \\ x+1, & \text{if } 1 \leq x < 2 \\ 0, & \text{elsewhere} \end{cases}$$

you can use nested *IF* functions, e.g.,

`IF(x<0, -x, IF((0<=x) And (x<1), x, IF((1<=x) And (x<2), x+1, 0)))`

Replacing  $x$  with the proper cell reference, the following figure shows various evaluations of this piecewise function using nested *IF* functions:

	B	C		B	C
13	x	f(x)	13	x	f(x)
14	-3	3	14	-3	=IF(B14<0, -B14, IF((0<=B14) AND (B14<1), B14, IF((1<=B14) AND (B14<2), B14+1, 0)))
15	0.5	0.5	15	0.5	=IF(B15<0, -B15, IF((0<=B15) AND (B15<1), B15, IF((1<=B15) AND (B15<2), B15+1, 0)))
16	1.5	2.5	16	1.5	=IF(B16<0, -B16, IF((0<=B16) AND (B16<1), B16, IF((1<=B16) AND (B16<2), B16+1, 0)))
17	3.5	0	17	3.5	=IF(B17<0, -B17, IF((0<=B17) AND (B17<1), B17, IF((1<=B17) AND (B17<2), B17+1, 0)))

(a) evaluations (b) formulas

Figure 37. Use of nested *IF* functions to evaluate a piecewise function in EXCEL.

The following figure shows the equivalence between the nested *IF* function in EXCEL and the corresponding *If ... ElseIf ... Else ... End If* statement in VBA.

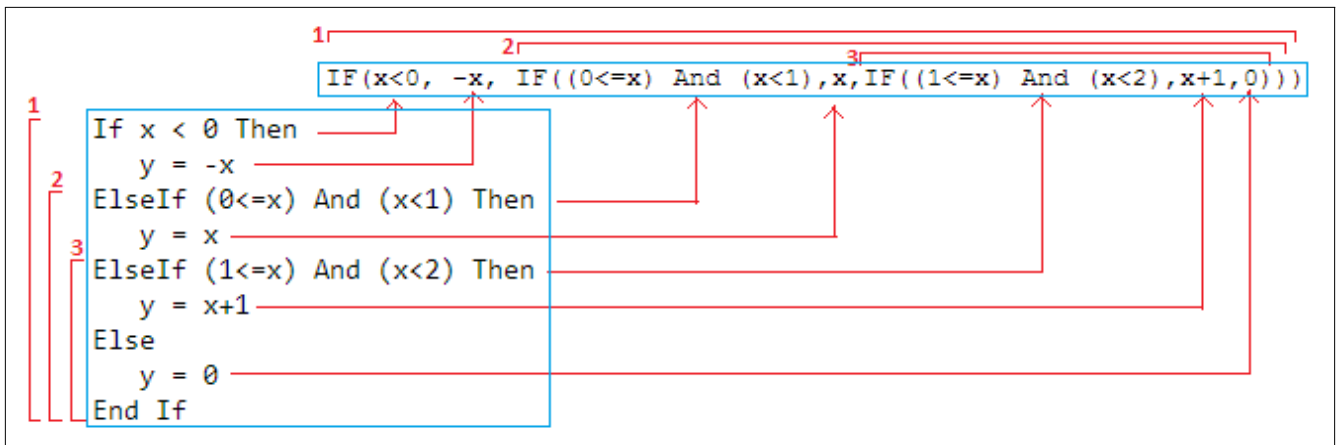


Figure 38. Equivalence between a VBA *If ... ElseIf ... Else ... End If* statement and a *EXCEL* nested *IF* function.

### Back to VBA: the *Case ... Select ... End Case* statement

The *If... ElseIf... Else... End If* statement of Figure 38 can be re-written as follows:

```

If x < 0 Then
    y = - x
ElseIf x < 1 Then
    y = x
ElseIf x < 2 Then
    y = x+1
Else
    y = 0
End If

```

This selection statement can be re-written furthermore using a *Select Case ... End Case* statement as follows:

```

Select Case x
    Case Is < 0
        y = - x
    Case Is < 1
        y = x
    Case Is < 2
        y = x+1
    Case Else
        y = 0
End Select

```

This example works for a continuous *Case* variable *x*. An alternative for this *Select Case ... End Case* statement, where the *Case* variable is continuous, is the following:



```

Select Case x
  Case Is < 0
    y = - x
  Case 0 To 1
    y = x
  Case 1 To 2
    y = x+1
  Case Else
    y = 0
End Select

```

Consider next a discrete *Case* variable, say,  $k$ , which can take positive integer values. Study the following *Select Case ... End Select* statement:

```

Select Case k
  Case 1
    r = 2*k + 1
  Case 2, 3, 5, 7
    r = 4*(k-1)
  Case 4, 6
    r = 3*k - 1
  Case 7 to 10
    r = Sq(k^2+1)
  Case Is > 10
    r = k
  Case Else
    r = 0
End Select

```

These three examples allows to write the general expression for the *Select Case ... End Select* statement as follows:

```

Select Case <expression>
  Case <case expression 01>
    <statements 01>
  Case <case expression 02>
    <statements 02>
  ...
  Case Else
    <statements Else>
End Select

```

In this general expression for the *Select Case ... End Select* statement, *<expression>* is an expression that involves a *Case* variable (typically, it's just a variable name). For each *Case* line the *<case expression xx>* defines the values of the *Case* variable that determine that particular case. If the *Case* variable takes on one of the values defined by the corresponding *<case expression xx>*, then the *<statements xx>* get executed, and control is sent out of the *Select Case ... End Select* statement. If all *<case expressions >* are exhausted, then the *<statements Else>* get executed.

The *<case expression xx >* may be one of the following:

- A single variable name, e.g.,  $k$
- An expression involving a variable, e.g.,  $2*k+1$
- A list of values of a variable, e.g., 2, 3, 5, 7

- A list of expressions involving a variable, e.g.,  $2*k+1$ ,  $2*k+3$ ,  $2*k+5$
- A list involving both values and expressions, e.g., 2, 5,  $2*k+1$
- A range of values determined by an expression of the form  $a$  To  $b$ , where  $a < b$ , e.g., 2 To 5
- A list of range of values, e.g., 2 To 5, 10 To 12
- A range of value determined by the *Is* statement, e.g.,  $Is < 5$ ,  $Is > 10$

Thus, by using combinations or the *Case* expressions, you can create complex *Select Case ... End Select* statements. The following example illustrates a simple *Select Case ... End Select* statement.

In this example we use two *Select Case ... End Select* statements to calculate the area of some simple geometric figures. The code for this example is shown below.

```
Sub SelectGeometricFigure
    Dim iType As Integer, Pi As Double
    Dim b As Double, h As Double, A As Double, r As Double
    Pi = 4.0 * Atn(1.0)
    iType = InputBox("Select a geometric figure:" & Chr(13) & _
        "1 - Triangle      2 - Rectangle" & Chr(13) & _
        "3 - Circle")
    Select Case iType 'Select Case for input data
        Case 1,2
            b = InputBox("Enter base, b = ", "Triangle or Rectangle")
            h = InputBox("Enter height, h = ", "Triangle or Rectangle")
        Case 3
            r = InputBox("Enter radius, r = ", "Circle")
    End Select
    Select Case iType 'Select Case for calculating area
        Case 1
            A = 1./2.*b*h
        Case 2
            A = b*h
        Case 3
            A = Pi*r^2
    End Select
    MsgBox("Area = " & CStr(A))
End Sub
```

Figure 39. Example of two *Select Case ... End Select* statements used in a program that calculates areas of simple geometric figures

Some programming issues of interest in the code of Figure 39 are the following:

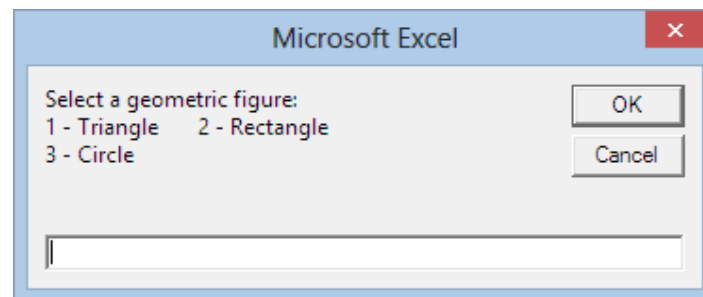
- The value of  $\pi$  is calculated using  $Pi = 4.0 * Atn(1.0)$
- The integer variable *iType* is used to select among three different geometric figures. Thus,  $iType = 1$  represents a triangle,  $iType = 2$  represents a rectangle, and  $iType = 3$  represents a circle. The value of *iType* is entered through an *InputBox*.
- The string in the *InputBox* requesting the value of *iType* includes the function *Chr*(13). Function *Chr*(*n*) is used to insert an ASCII character based on the value of *n*. As indicated earlier, individual ASCII characters are stored in a *byte* (= 8 bits), which can store up to 256 ( $=2^8$ )

values. Thus,  $n$  can take values from 0 to 255. `Chr(13)` is a non-printable character representing a carrier feed (CF), i.e., the beginning of a new line in output. The use of `Chr(13)` allows you to create an output string in different lines.

- The `InputBox` statement requesting the value of `iType` includes two continuation characters ( \_ ), which allow the user to write a long command statement into several lines. The `InputBox` statement used to request the value of `iType` in the program of Figure 36 is the following:

```
iType = InputBox("Select a geometric figure:" & Chr(13) & _
                "1 - Triangle      2 - Rectangle" & Chr(13) & _
                "3 - Circle")
```

- The string that results out of the `InputBox` statement shown above is shown below (you can use more than three lines of output in the string):



- The code of Figure 36 shows two `Select Case ... End Select` statements, the first one is used to enter the input data for the geometric figure selected, while the second one is used to calculate the area of the selected figure.

An alternative way to code the program of Figure 35, using a single `Case Select`, is shown below.

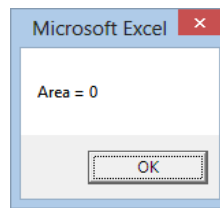
```
Sub SelectGeometricFigure2
    Dim iType As Integer, Pi As Double
    Dim b As Double, h As Double, A As Double, r As Double
    Pi = 4.0 * Atn(1.0)
    iType = InputBox("Select a geometric figure:" & Chr(13) & _
                    "1 - Triangle      2 - Rectangle" & Chr(13) & _
                    "3 - Circle")
    Select Case iType 'Select Case for calculating area
        Case 1
            b = InputBox("Enter base, b = ", "Triangle or Rectangle")
            h = InputBox("Enter height, h = ", "Triangle or Rectangle")
            A = 1./2.*b*h
        Case 2
            b = InputBox("Enter base, b = ", "Triangle or Rectangle")
            h = InputBox("Enter height, h = ", "Triangle or Rectangle")
            A = b*h
        Case 3
            r = InputBox("Enter radius, r = ", "Circle")
            A = Pi*r^2
    End Select
    MsgBox("Area = " & CStr(A))
End Sub
```

Figure 37. Alternative code for the program of Figure 36 using a single `Select Case` statement.

## Labels and the *Go To* statement

A *label* in a VBA program is a string, ending in a colon (:), that specifies a location in the program. Typically, a label is typed in a single line, starting on the left margin of the code window. You can send the program control to a particular label by using the *Go To* statement. The use of labels and the *Go To* statement will be illustrated next by modifying the code of Figure 37.

In the code of Figure 37 the user is asked to select a value of the integer variable *iType* through an *InputBox* corresponding to *Type* = 1 for a triangle, *iType* = 2 for a rectangle, or *iType* = 3 for a circle. If the user enters a value other than 1, 2, or 3, the *Select Case ... End Select* statement (which lacks a *Case Else* statement) will be skipped, and the *MsgBox* at the end of the program will be executed. However, since the value of *A* (the area) is initialized as zero when the variable is declared, then a value of  $A = 0$  is reported in this case:



One alternative way to handle the case when the user enters a value of *iType* other than 1, 2, or 3, is to add a *Case Else* statement to the *Select Case ... End Select* letting the user know that they made the wrong choice. The modified code for this situation is shown below:

```
Option VBASupport 1
Option Explicit

Sub SelectGeometricFigure2
    Dim iType As Integer, Pi As Double
    Dim b As Double, h As Double, A As Double, r As Double
    Pi = 4.0 * Atn(1.0)
    iType = InputBox("Select a geometric figure:" & Chr(13) & _
                    "1 - Triangle      2 - Rectangle" & Chr(13) & _
                    "3 - Circle")

    Select Case iType 'Select Case for calculating area
        Case 1
            b = InputBox("Enter base, b = ", "Triangle or Rectangle")
            h = InputBox("Enter height, h = ", "Triangle or Rectangle")
            A = 1./2.*b*h : MsgBox("Area = " & CStr(A))
        Case 2
            b = InputBox("Enter base, b = ", "Triangle or Rectangle")
            h = InputBox("Enter height, h = ", "Triangle or Rectangle")
            A = b*h : MsgBox("Area = " & CStr(A))
        Case 3
            r = InputBox("Enter radius, r = ", "Circle")
            A = Pi*r^2 : MsgBox("Area = " & CStr(A))
        Case Else
            MsgBox("Wrong selection - Try again")
    End Select
End Sub
```

Figure 38. Code of Figure 37 modified to account for wrong selection of *iType*.

Another way to modify the code of Figure 37 to force the user to make an acceptable selection when entering the value of *iType* is to use a label (*SelectGeometricFigure*, in this case) and a *Go To* statement, as illustrated next:

```
Sub SelectGeometricFigure2
    Dim iType As Integer, Pi As Double
    Dim b As Double, h As Double, A As Double, r As Double
    Pi = 4.0 * Atn(1.0)
SelectGeometricFigure:
    iType = InputBox("Select a geometric figure:" & Chr(13) & _
        "1 - Triangle          2 - Rectangle" & Chr(13) & _
        "3 - Circle")
    If ((iType <> 1) And (iType <> 2) And (iType <> 3)) Then
        MsgBox("Wrong selection - try again")
        GoTo SelectGeometricFigure
    End If
    Select Case iType 'Select Case for calculating area
        Case 1
            b = InputBox("Enter base, b = ", "Triangle or Rectangle")
            h = InputBox("Enter height, h = ", "Triangle or Rectangle")
            A = 1./2.*b*h
        Case 2
            b = InputBox("Enter base, b = ", "Triangle or Rectangle")
            h = InputBox("Enter height, h = ", "Triangle or Rectangle")
            A = b*h
        Case 3
            r = InputBox("Enter radius, r = ", "Circle")
            A = Pi*r^2
    End Select
    MsgBox("Area = " & CStr(A))
End Sub
```

Figure 39. Modifying the code of Figure 37 using the label *SelectGeometricFigure* and a *Go To* statement to force the user to make a selection for variable *iType* to be 1, 2, or 3, only.

Notice the following programming issues of interest in the code of Figure 39:

- An *If ... End If* statement checks that the value of *iType* is not 1, 2, or 3, by using the logical statement:  $((iType \neq 1) \text{ And } (iType \neq 2) \text{ And } (iType \neq 3))$ , or, using mathematical logical notation:  $((iType \neq 1) \wedge (iType \neq 2) \wedge (iType \neq 3))$ .
- This logical statement is true only if the three comparisons involved are true, i.e., if *iType* is not 1, 2, or 3.
- The *If ... End If* statement, therefore, lets the user know they got the wrong choice if *iType* is not 1, 2, or 3, and then sends the control to label *SelectGeometricFigure*.
- Since the first statement following label *SelectGeometricFigure* is the *InputBox* requesting the value of *iType*, the user is prompted again to enter a value for *iType*.
- If the user does not select 1, 2, or 3, then the *If ... End If* statement repeats the message and sends the user back to enter the value of *iType* again, until a value of 1, 2, or 3 is selected.

Thus, the code of Figure 39 forces the user to make the right choice of *iType*, or the program would just continue repeating the prompt until the user selects *iType* to be 1, 2, or 3.

### **Warning: Avoid using the *GoTo* statement – Use Structured Programming**

The code of Figure 39 is the only situation where I would ever recommend using a label and *Go To* statement. The reason for this is that the use of *Go To* statements, where the control of the code can be sent to any labeled location in a program, may create what is known as "*spaghetti code*," i.e., a convoluted code which is difficult to follow by other programmers. The production of "*spaghetti*" code was common in the earlier years of programming with *Fortran* and other high-level languages (1957-1975) thanks to the heavy use of the *Go To* statement. To avoid such problematic programming style, in the 1970's the idea of *structured programming* was developed. Structured Programming introduced the three types of basic programming structures presented earlier in this document: sequence, decision (or branching), and repetition (or looping), and suggested the use of blocks of these structures to build programs, rather than sending the control to labeled locations through the *Go To* statement. Even though the *Go To* statement exists in most high-level programming languages, its use is discouraged. Once again, use it only in a situation as that illustrated by Figure 39, namely, when you need to force the user to enter the proper selection by sending the control back to the input statement if the wrong value is entered. Otherwise, *do not use labels and the Go To statement at all*.

### **Note: Low- and High-Level Languages**

It was indicated earlier that computers, at their core, handle data and instructions in the form of binary data, i.e., combinations of bits: zeros and ones, only. This is the lowest level of programming possible: *binary language*. Binary language, however, is not easy to program by most humans. Therefore, forms of higher-level language were developed which are called *Assembly*, or *Assembler Language*. Instructions in assembly, or assembler, language may look as these:

```
001 RCL
002 RCL
ADD
003 STO
```

In this (made up) example, the number 001, 002, and 003 represent memory locations, while the commands *RCL*, *ADD*, and *STO* are abbreviations for *recall* (to recall values from a memory location), *add* (to add values in the register or stack), and *store* (to store a value in a memory location). The assembly instruction above thus recall values of memory locations 001 and 002, place them in a register or stack (a location where data is placed for operations), add those two values, and store the result in memory location 003.

Obviously, assembly (or assembler) language is an improvement over using binary language, but it's still a long way from the easier commands of VBA or other *high-level languages*. Thus, in 1957, the first ever high-level language was developed at IBM: *Fortran*. The word *Fortran* stands for *Formula translation*, i.e., it was intended to code mathematical calculations, although, the ability to handle text data (strings) was incorporated in the language. A high-level language, like *Fortran* or VBA, includes commands that look somewhat like human language, e.g., *IF*, *FOR*, *InputBox*, *MsgBox*, etc.

The creation of high-level computer languages, starting with *Fortran* in 1957, opened up access to computer programming to everyone. *Fortran* was the programming language preferred by scientists and engineers, pretty much until the 1980's, when the availability of personal computers made *BASIC* a very popular alternative. Many other high-level computer languages have been developed since 1957. A list of many of them is available in the following web site, in alphabetical order:

<http://www.99-bottles-of-beer.net/abc.html>

The list of programming languages in that web site include also several versions of the *Assembler* language (listed in the first page, under A). Therefore, the list is not only for high-level languages.

The purpose of the web site linked above is to let people program the lyrics of the traditional song "99 bottles of beer in the wall" in their favorite programming language. This song is a classical example of a repetition or loop in programming, since the lyrics refer to starting with "99 bottles of beer in the wall" and then executing the procedure of taking "one down" and passing "it around", so you end up with "98 bottles of beer on the wall." The song continues repeating similar lyrics with "97 bottles of beer on the wall", "96 bottles of beer on the wall", etc., until you get "zero bottles of beer on the wall", at which point the loop ends. Although started as a joke by some programmers, the site is now a nice reference for listing about 1500 versions of the program, which implies that the number of high-level computer languages is very large.

### **Note: Calculators and programming**

Calculators commonly used by science and engineering students, such as the *TI 89 Titanium*, the *TI Nspire*, or the *HP 50G* calculators, include programming languages with which you can create your own programs to use in the calculators. The *TI* calculators can be programmed in their own version of BASIC, called *TI BASIC*, while the *HP 50G* uses a language called *RPL (Reverse Polish LISP)*. For information on programming calculators check the *programming* links in my *calculators* web page:

[http://www.neng.usu.edu/cee/faculty/gurro/Software\\_Calculators/Calculators.htm](http://www.neng.usu.edu/cee/faculty/gurro/Software_Calculators/Calculators.htm)

If you have an *HP 50G* calculator, and are interested in programming the *HP 50G* calculator in *Assembler* language, you can learn the basics by reading "*Introduction to Saturn Assembler Language*" by *Gilbert Fernandes* (2005), available at this link:

<http://www.hpcalc.org/hp48/docs/programming/asm-pdf.zip>

Useful links for programs and programming for the *TI 89 Titanium* calculator are available at the following location: [www.tiEXCEL.org/](http://www.tiEXCEL.org/), while for the *HP 50G* calculator you should use: [www.hpEXCEL.org/](http://www.hpEXCEL.org/)

### **Back to VBA programming – An example using *If*, *Select Case*, etc.**

The code shown in this section, starting in page 49, uses almost all concepts presented so far for programming in VBA. The program is aimed at calculating a number of quantities for open channel flow for different types of channel cross-sections. The program lets the user select first the system of units to use by assigning a value to variable *iUnits* (*iUnits* = 1 for the S.I., or *iUnits* = 2 for the E.S.) through an *InputBox*. Based on this selection, and through the use of a *Select Case ... End Select* statement, the program assigns values to variables *Cu* and *g*. Thus, for the S.I., *Cu* = 1.0, and *g* = 9.806, while for the E.S., *Cu* = 1.486, and *g* = 32.2. Also, based on the type of units some string variables (*sL*, *sA*, *sV*, and *sQ*) are loaded. These variables hold the units of length (*sL* = " m", or *sL* = " ft"), area (*sA* = " m^2", or *sA* = " ft^2"), velocity (*sV* = " m/s", or *sV* = " fps"), and discharge (*sQ* = " m^3/s", or *sQ* = " cfs"), which will be built into an output string (*outString*) using the proper units for the system of units selected. The *Select Case ... End Select* statement includes a *Case Else* option where, if the value of *iUnits* is not 1 or 2, a message is sent to the user indicating that the selection of the system of units is wrong. The control then is sent to label *SelectUnits* so that the user must select either *iUnits* = 1 or *iUnits* = 2.



After a correct selection for *iUnits* is performed, the program offers a second *InputBox* where the user selects the type of cross-section in the open channel with options: 1 – *Trapezoidal*, 2 – *Rectangular*, 3 – *Triangular*, and 4 – *Circular*. A *Select Case ... End Select* statement, afterward, will handle the input of geometric variables such as bottom width *b*, side slope *m*, and flow depth *y* for a trapezoidal channel, or bottom width *b* and flow depth *y* for a rectangular channel, or side slope *m* and flow depth *y* for a triangular channel, or diameter *D* and flow depth *y* for a circular channel. Each of the *Cases* in the *Select Case ... End Select* statement proceed to calculate the cross-sectional area *A*, wetted perimeter *P*, and top width of the cross section *T*, and starts to build the output string (*outString*) by listing the type of cross section into the string.

The *Select Case ... End Select* statement that uses *iType* as its *Case* variable offers a *Case Else* option that will send the user back to the label *SelectType* in case a value different from 1, 2, 3, or 4 is entered by the user. Thus, the user will have to select one of those four cases, or the program will continue repeating the request for a cross-section type to be entered. Within this *Select Case ... End Select* statement, the *Case* option corresponding to the circular case (i.e., *iType* = 4), includes an error trap for the case that the user mistakenly enters a flow depth *y* that is larger than the diameter of the circular cross-section *D*. If that situation is detected, the control is sent back to label *ReEnterD*, where the user can rectify the error.

Past this *Select Case ... End Select* statement, the program uses *InputBoxes* to enter values of the bed slope *S0* and the Manning's resistance coefficient *n*. Then, the program proceeds to calculate the hydraulic radius *R*, the hydraulic depth *Dh*, the flow velocity *V*, the discharge *Q*, and the Froude number *Fr*. (These calculations are shown in a single line, with the individual calculation statements separated by colons, :).

After the calculation line, the program proceeds to put together the output string *outString* to show all the calculation results, with units, in several lines of output. Several calls to function *Char(13)* are used to produce the new lines of output. Once the output string *outString* is ready, the string is shown in a *MsgBox*.

Next, the program checks if other problems need to be calculated by using an *InputBox* to enter the value of the string variable *iAnswer* as "Y" (or "y") or "N" (or "n"). If *iAnswer* = "Y" (or "y"), then the control is sent to label *SelectUnits*, i.e., the program starts all over again. Any other value of *iAnswer* will simply let the control go down to the *End Sub* line, thus ending the program.

Thus, in this example we use integer, double, and string variables, all defined with *Dim*; we used *If* statements, *Select Case* statements, *InputBoxes*, *MsgBoxes*, assignment statements, continuation lines, *Char(13)* for new output lines, string concatenation, built-in functions (*Sin*, *Cos*, etc.), arithmetic operations, comparison statements (for the *If* statements), labels, the *Go To* statement, and followed proper indentation for the code. The code is shown next. The program can be run from the *EXCEL IDE*.



Option Explicit

Sub SampleSelectCase

```
Dim iUnits As Integer, iType As Integer
Dim iAnswer As String
Dim outString As String
Dim Cu As Double, n As Double, S0 As Double
Dim b As Double, m As Double, y As Double
Dim D As Double, theta As Double, g As Double
Dim A As Double, P As Double, R As Double, Dh As Double
Dim T As Double, V As Double, Q As Double, Fr As Double
Dim sL As String, sA As String, sV As String, sQ As String
```

SelectUnits:

```
iUnits = InputBox("Select the system of units:" & _
    Chr(13) & "1 - International System (S.I.)" & _
    Chr(13) & "2 - English System")
```

Select Case iUnits

Case 1

Cu = 1.0 : g = 9.806

sL = " m" : sA = " m^2" : sV = " m/s" : sQ = " m^3/s"

outString = "Using the International System of Units" & Chr(13)

Case 2

Cu = 1.486 CC: g = 32.2

sL = " ft" : sA = " ft^2" : sV = " fps" : sQ = " cfs"

outString = "Using the English System of Units" & Chr(13)

Case Else

MsgBox("Wrong type of units selected")

Go To SelectUnits

End Select

SelectType:

```
iType = InputBox("Select the type of cross-section to calculate" & _
    Chr(13) & "1 - Trapezoidal      2 - Rectangular" & _
    Chr(13) & "3 - Triangular      4 - Circular ", "Open Channels")
```

Select Case iType

Case 1 'Trapezoidal

b = InputBox("Enter channel bottom width:", "Trapezoidal")

m = InputBox("Enter side slope:", "Trapezoidal")

y = InputBox("Enter flow depth:", "Trapezoidal")

outString = outString & "Trapezoidal open channel" & Chr(13)

A = (b+m\*y)\*y : P = b + 2\*y\*Sqr(1+m^2) : T = b+2\*m\*y

Case 2 'Rectangular

b = InputBox("Enter channel bottom width:", "Rectangular")

y = InputBox("Enter flow depth:", "Rectangular")

outString = outString & "Rectangular open channel" & Chr(13)

A = b\*y : P = b + 2\*y : T = b

Case 3 'Triangular

m = InputBox("Enter side Slope:", "Triangular")

y = InputBox("Enter flow depth:", "Triangular")

outString = outString & "Triangular open channel" & Chr(13)

A = 1./2.\*m^2\*y : P = 2\*y\*Sqr(1+m^2) : T = 2\*m\*y

Case 4 'Circular

```

ReEnterD:
    D = InputBox("Enter channel diameter:", "Circular")
    y = InputBox("Enter flow depth:", "Circular")
    If D < y Then
        MsgBox("You enter D = " & CStr(D) & " < y = " & CStr(y) & ". Try _
            again")
        Go To ReEnterD
    End If
    outString = outString & "Trapezoidal open channel" & Chr(13)
    theta = WorksheetFunction.Acos(1-2*y/D)
    A = D^2/4*(theta - Sin(theta)*Cos(theta))
    P = theta*D : T = D*sin(theta)
Case Else
    MsgBox("Wrong selection")
    Go To SelectType
End Select

S0 = InputBox("Enter bed slope:")
n = InputBox("Enter Manning's n:")
R=A/P : Dh=A/T : V=Cu*Sqr(S0)/n*R^(2./3.)*Sqr(S0) : Q=V*A : Fr=V/Sqr(g*Dh)

outString = outString & "Area = " & CStr(A) & sA & Chr(13)
outString = outString & "Wetted Perimeter = " & CStr(P) & sL & Chr(13)
outString = outString & "Hydraulic Radius = " & CStr(R) & sL & Chr(13)
outString = outString & "Hydraulic Depth = " & CStr(Dh) & sL & Chr(13)
outString = outString & "Flow Velocity = " & CStr(V) & sV & Chr(13)
outString = outString & "Discharge = " & CStr(Q) & sQ & Chr(13)
outString = outString & "Froude Number = " & CStr(Fr) & Chr(13)

MsgBox(outString)

iAnswer = InputBox("Would you like to solve another problem? Y or N:", _
    "Open Channel Flow")

If iAnswer = "y" Or iAnswer = "Y" Then
    Go To SelectUnits
End If

End Sub

```

The geometric calculations presented in this program correspond to calculations of area  $A$ , wetted perimeter (length of the cross-sectional boundary in contact with water)  $P$ , and top width of the cross-section  $T$ . The basic geometric elements required for the four different types of cross-sections used are illustrated in the figure below. Other geometric elements calculated in the program are the hydraulic radius  $R = A/P$ , and the hydraulic depth (the depth of a rectangular cross-section with the same width as the top width of the cross section)  $Dh = A/T$ . With the constant  $Cu$ , which depends on the system of units, and the acceleration of gravity  $g$ , both selected by the program, and with the bed slope  $S0$ , and the Manning's resistance coefficient  $n$ , given by the user, we can calculate the flow velocity using the Manning's equation,  $V = \frac{Cu}{n} R^{2/3} \sqrt{S0}$ , the flow discharge using the continuity equation,  $Q = A*V$ , and the Froude number (a dimensionless number) as  $Fr = \frac{V}{\sqrt{g Dh}}$ .

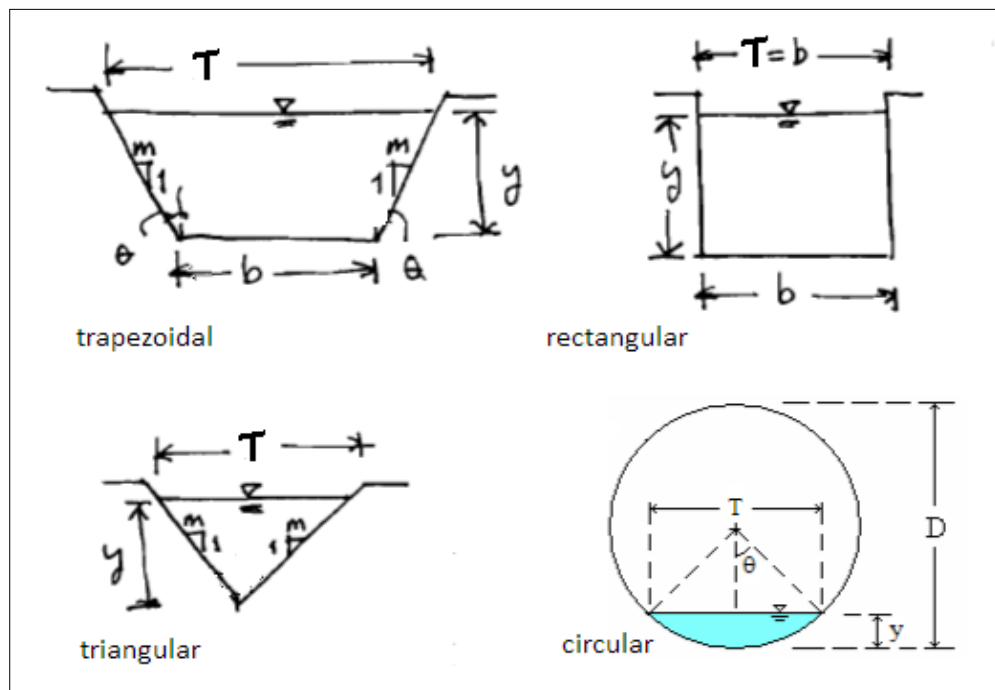


Figure 40. The geometry of various open channel cross sections.

Formulas for the area  $A$ , wetted perimeter  $P$ , and top width  $T$ , of the four cross-sections of Figure 40 are given by:

- Trapezoidal:  $A = (b + m y) y$  ,  $P = b + 2 y \sqrt{1 + m^2}$  ,  $T = b + 2 m y$
- Rectangular:  $A = b y$  ,  $P = b + 2 y$  ,  $T = b$
- Triangular:  $A = \frac{1}{2} m y^2$  ,  $P = 2 y \sqrt{1 + m^2}$  ,  $T = 2 m y$
- Circular:  $A = \frac{D^2}{4} (\theta - \sin(\theta) \cos(\theta))$  ,  $P = \theta D$  ,  $T = D \sin(\theta)$   
where  $\theta = \cos^{-1} \left( 1 - \frac{2 y}{D} \right)$

To test the program, enter the data for the following Examples:

Example No.	Units	Type	bottom width $b$	side slope $m$	flow depth $y$	diameter $D$	bed slope $S_0$	Manning's $n$
1	3	(Use to test program error trap for units (must be 1 or 2))						
2	1	5	(Use to test program error trap for cross-sectional type (must be 1, 2, 3, or 4))					
3	1	1	2.5 m	0.75	3.0 m	-	0.0001	0.01
4	2	1	10.0 ft	1.5	2.5 ft	-	0.000025	0.012
5	1	2	5.0 m	-	4.2 m	-	0.000326	0.008
6	2	2	15.0 ft	-	7.5 ft	-	0.00185	0.023
7	1	3	-	1.5	1.5 m	-	0.0000345	0.018
8	2	3	-	2	3.0 ft	-	0.000567	0.025
9	1	4	-	-	2.7 m	3.0 m	0.000835	0.015
10	2	4	-	-	3.5 ft	10.0 ft	0.000112	0.032
11	1	4	-	-	6.0 ft	5.0 m	(Test case $y > D$ error trap)	

## Debugging

Debugging is the process of analyzing the behavior of a program by using breakpoints, step-by-step processing, and variable viewing, in order to figure out problems in the program logic that are producing unexpected results. As indicated in an earlier section, there are typically three types of errors in programming: (1) *syntax errors*: easily caught by the IDE; (2) *run-time errors*: found when running the program, diagnosed during running, and relatively easy to fix (e.g., division by error, etc.); and, (3) *logical errors*: errors in the algorithm. Logical errors are the most difficult to figure out because the program seems to run properly, yet producing the wrong results. Thus, the only way to know that there may be logical errors is by comparing the results of your program with some known results calculated separately. Debugging is the way to find logical errors.

In an earlier section we introduced the different components of the *EXCEL VBA IDE*. The IDE includes some buttons and menu options that are useful for debugging, namely:

(a) Buttons for running, pausing, and stopping macros (see Figure 8, Chapter 1):



The operation of each of these three buttons is as follows:

- (1) *Run Sub/User Form*: to run the code currently in the code window
- (2) *Break* (Cntl+Break): to pause code execution
- (3) *Reset*: stops code execution and sends control to top of code

These buttons have equivalent options in the VBA IDE's *Run* menu (see Figure 40B, below).

(b) Debugging menu options (see Figure 7, Chapter 1):

Three of the VBA IDE methods in *EXCEL* contain items useful for debugging as illustrated in Figure 40B, below.

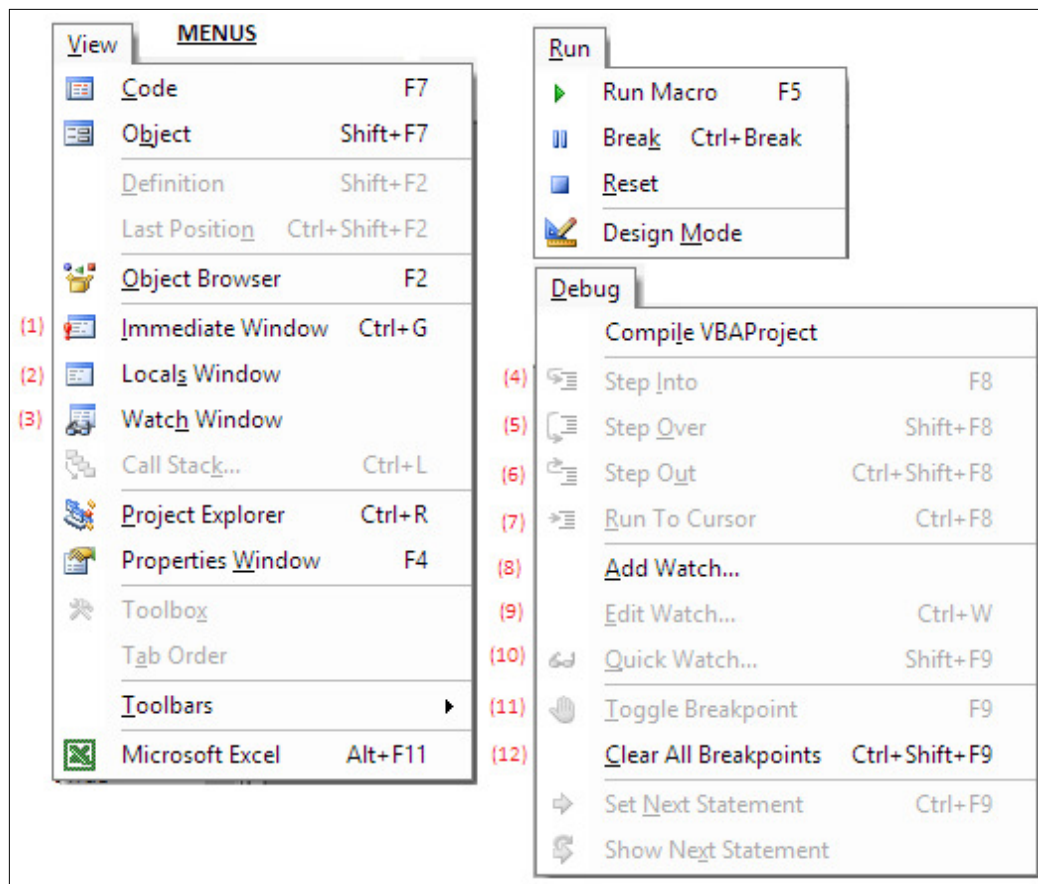


Figure 40B. The *View*, *Run*, and *Debug* menus in the *EXCEL*'s VBA IDE.

The items available in the *View* and *Debug* menus in the *EXCEL*'s VBA IDE that concern debugging are:

In the *View* menu - important windows:

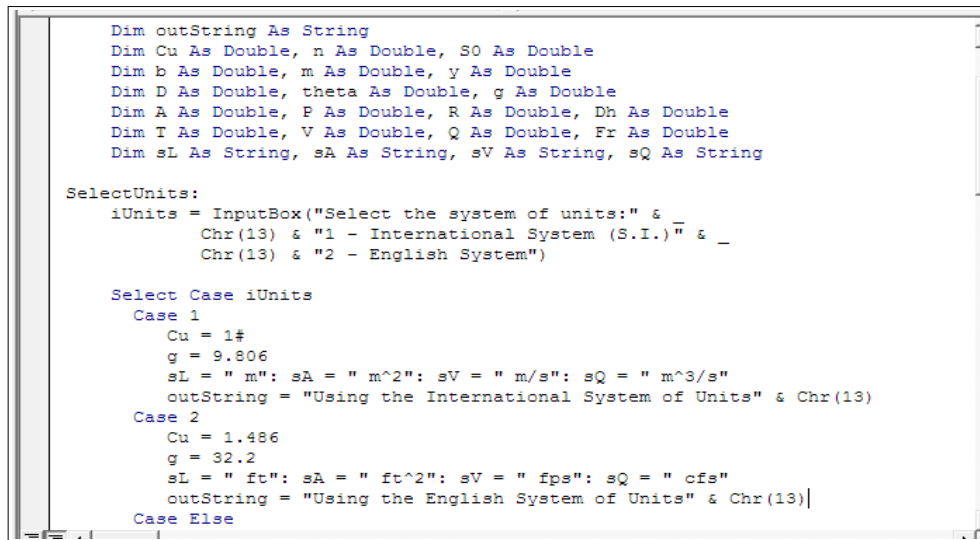
- (1) *Immediate Window*: used for printing current results (not used in this section)
- (2) *Locals Window*: opens a window to observe variables local to the Sub under consideration
- (3) *Watch Window*: opens a window to observe variables selected by the user

In the *Debug* menu:

- (4) *Step Into*: Begin debugging code
- (5) *Step Over*: Step over code in a called subroutine.
- (6) *Step Out*: Out of debugging mode, run program normally
- (7) *Run To Cursor*: Click on a line, let code run from current position to the line clicked
- (8) *Add Watch*: add variable to be observed
- (9) *Edit Watch*: modify definition of variable(s) to be observed
- (10) *Quick Watch*: select a variable, *Quick Watch* shows current value
- (11) *Toggle Breakpoints*: Click on a line, select this option to enter a breakpoints
- (12) *Clear All Breakpoints*: As the option indicates, all breakpoints are removed

## A Debugging exercise

Let's use the code of pages 49 and 50 to practice debugging. Open the *EXCEL BASIC IDE* to show the code to be debugged.

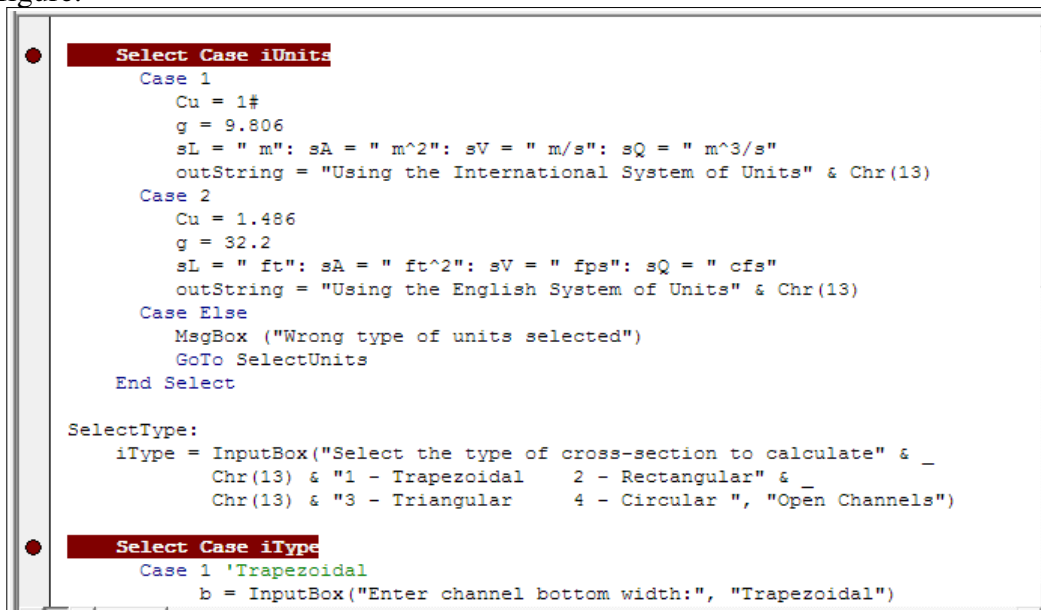


```
1 Dim outString As String
2 Dim Cu As Double, n As Double, S0 As Double
3 Dim b As Double, m As Double, y As Double
4 Dim D As Double, theta As Double, g As Double
5 Dim A As Double, P As Double, R As Double, Dh As Double
6 Dim T As Double, V As Double, Q As Double, Fr As Double
7 Dim sL As String, sA As String, sV As String, sQ As String
8
9 SelectUnits:
10 iUnits = InputBox("Select the system of units:" & _
11 Chr(13) & "1 - International System (S.I.)" & _
12 Chr(13) & "2 - English System")
13
14 Select Case iUnits
15 Case 1
16 Cu = 1#
17 g = 9.806
18 sL = " m": sA = " m^2": sV = " m/s": sQ = " m^3/s"
19 outString = "Using the International System of Units" & Chr(13)
20 Case 2
21 Cu = 1.486
22 g = 32.2
23 sL = " ft": sA = " ft^2": sV = " fps": sQ = " cfs"
24 outString = "Using the English System of Units" & Chr(13)|
25 Case Else
```

Figure 41. The *EXCEL BASIC IDE* showing line numbers in the code window.

### Adding a Breakpoint

Click on the line where you want to insert the breakpoint, then do one the following to insert a breakpoint: (1) select the option *Debug > Toggle breakpoint*; (2) Press [F9]; or, simply click on the gray vertical bar to the left of the code. This will add a small maroon round mark to the left of the line number of the selected line, and will highlight the line in the same color, indicating that a breakpoint is set and active at that point. For the present exercise, add breakpoints at the lines shown in the following figure.



```
14 Select Case iUnits
15 Case 1
16 Cu = 1#
17 g = 9.806
18 sL = " m": sA = " m^2": sV = " m/s": sQ = " m^3/s"
19 outString = "Using the International System of Units" & Chr(13)
20 Case 2
21 Cu = 1.486
22 g = 32.2
23 sL = " ft": sA = " ft^2": sV = " fps": sQ = " cfs"
24 outString = "Using the English System of Units" & Chr(13)
25 Case Else
26 MsgBox ("Wrong type of units selected")
27 GoTo SelectUnits
28 End Select
29
30 SelectType:
31 iType = InputBox("Select the type of cross-section to calculate" & _
32 Chr(13) & "1 - Trapezoidal    2 - Rectangular" & _
33 Chr(13) & "3 - Triangular    4 - Circular ", "Open Channels")
34
35 Select Case iType
36 Case 1 'Trapezoidal
37 b = InputBox("Enter channel bottom width:", "Trapezoidal")
```

Figure 43. Breakpoints shown at selected location of the code of pages 49 and 50.

The first breakpoint in Figure 43 will allow us to check on the value of *iUnits*, while the second breakpoint will allow us to check on the values of *Cu*, *sL*, *sA*, *sV*, *SQ*, *outString*, and *iType*. To proceed with the debugging process, let's use any arbitrary values of the input data, and get the process started by pressing the *Run* button in the *EXCEL VBA IDE*. The program will get started and stop at the first breakpoint to allow the programmer to view variables, or perform other actions. Get started and watch as the control stops at the first breakpoint as illustrated below. To run this exercise, I use *iUnits* = 1. Notice that a yellow arrow is now located above the breakpoint marker (red circle).

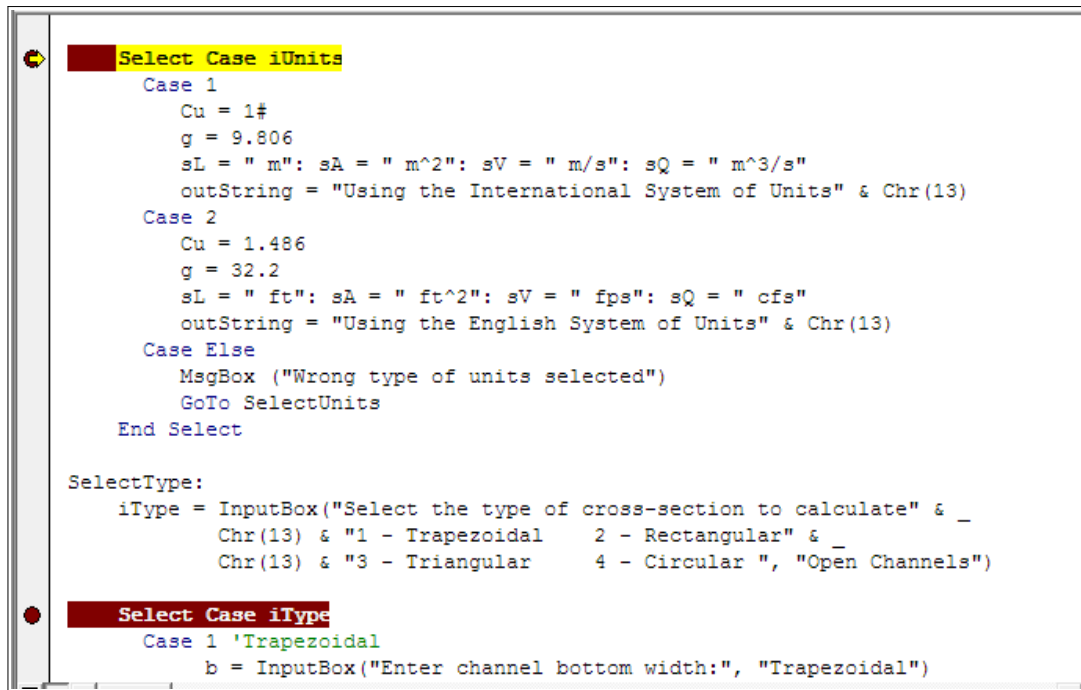


Figure 44. Program control stopped at first breakpoint in the program of pages 49 and 50.

#### Adding a variable watch

At this point, let's add a *Watch* for the variable *iUnits*. First, if you don't see a window called *WATCHES* at the bottom of the IDE, select the option *View > Watch Window*. Next, use the option *Debug > Add Watch* to add a variable name to watch. This action produces the following entry form:

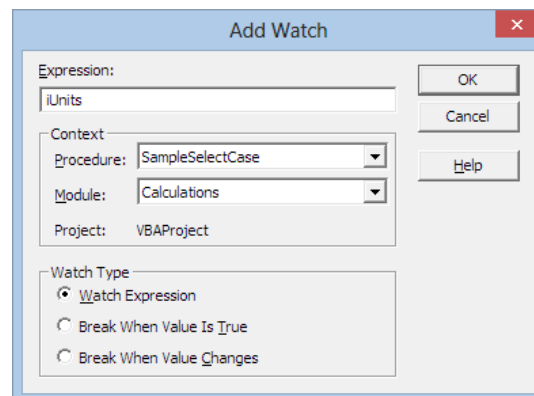


Figure 45A. The *Add Watch* window to add variable *iUnits* to the watch list.

The entry form of Figure 45A shows that we entered the variable *iUnits* in procedure *SampleSelectCase* in module *Calculations* (The module name can be changed in the *Properties* window in the *VBA IDE*. The procedure name is the name of the *Sub* within which variable *iUnits* is contained). The default *Watch Type* is “Watch Expression”, meaning we will be looking at the value of *iUnits* in the *WATCHES* window as the program is being debugged. At this point, press [ OK ] to add the variable *iUnits* to the watch list. The current value of *iUnits* (I used *iUnits* = 1) will be shown in the *WATCHES* window as displayed in the following figure:

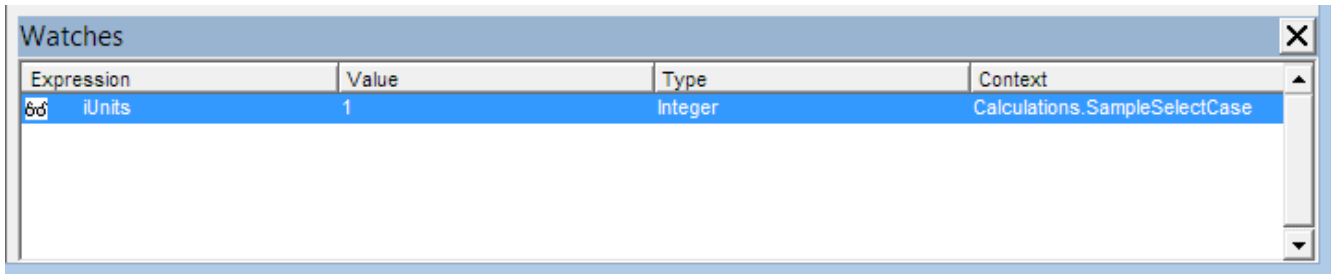


Figure 45. The *Watch* window in the *EXCEL VBA IDE* showing the value of variable *iUnits* for the program of pages 49-50 when the control is at the breakpoint as shown in Figure 44.

The *Watch* window shows that variable *iUnits* has a value of 1, that its type is *Integer*, and that it is defined in the context of the object *Calculations.SampleSelectCase*. The latter reference means the procedure (a *Sub*, in this case) named *SampleSelectCase* within the *Calculations* module.

For the next breakpoint stop in Figure 44, we would like to watch the variables *Cu*, *sL*, *sA*, *sV*, *SQ*, *outString*, and *iType*. You could proceed to enter each variable name through the option *Debug > Add Watch* as shown for variable *iUnits* in the case of Figure 45. An alternative way to enter watches is to use *Debug > Quick Watch...*[Shift+F9], as follows:

- In the code window, place the cursor next to or within the variable name you want to add to the watch list, say, *Cu*. This means, you can click on any instance of the variable *Cu* in the code, to select it.
- Then, select the option *Debug > Quick Watch....* This produces the following form:

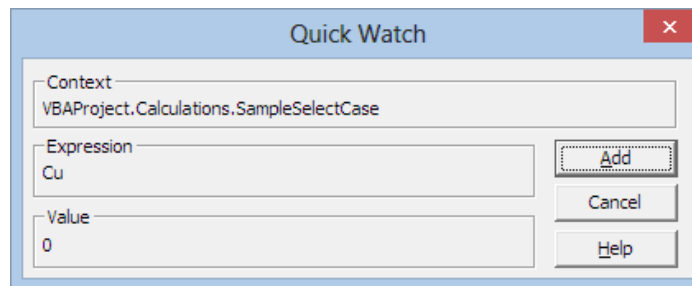


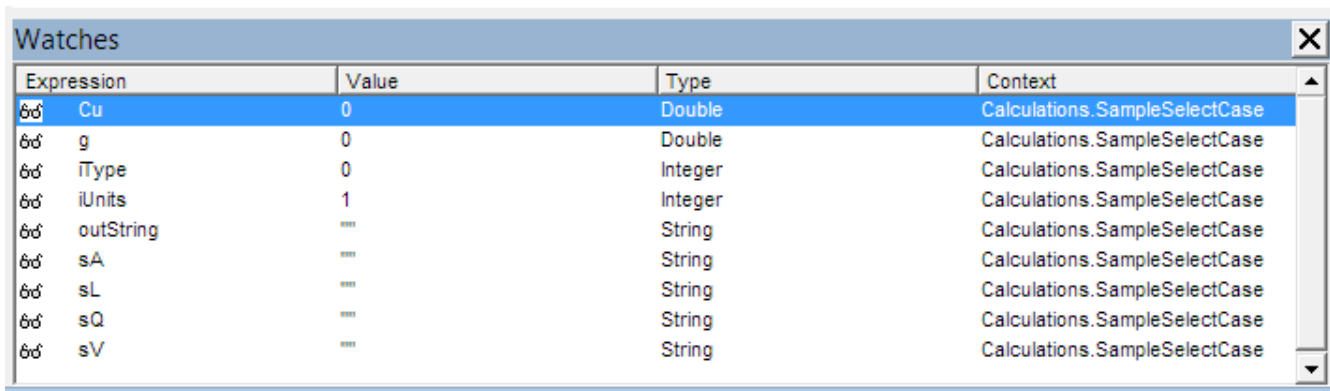
Figure 46A. The *Quick Watch* window for variable *Cu* in the code of Figure 44

- The form of Figure 46A shows that the variable *Cu* (referred to as *Expression* in the form) has a *Value* of zero (default initialization value), and belongs in the object *VBAProject.Calculations.SampleSelectCase*. The latter references means that variable *Cu* is contained in the procedure *SampleSelectCase*, contained within the module *Calculations* in a *VBAProject* (generic name).



- Press the [ Add ] button in the form of Figure 46A to add variable *Cu* to the watch list.

Continue adding variables to the watch list by using either *Debug > Add Watch* or *Debug > Quick Watch* (Shift+F9). After finishing adding variables, the *Watch window* will look as follows:



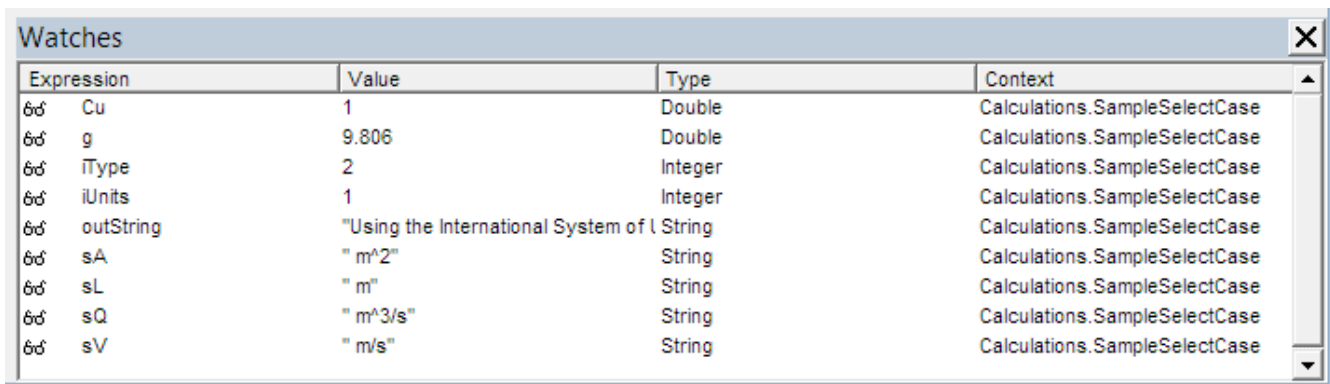
Expression	Value	Type	Context
Cu	0	Double	Calculations.SampleSelectCase
g	0	Double	Calculations.SampleSelectCase
iType	0	Integer	Calculations.SampleSelectCase
iUnits	1	Integer	Calculations.SampleSelectCase
outString	""	String	Calculations.SampleSelectCase
sA	""	String	Calculations.SampleSelectCase
sL	""	String	Calculations.SampleSelectCase
sQ	""	String	Calculations.SampleSelectCase
sV	""	String	Calculations.SampleSelectCase

Figure 46. The *WATCHES* window in the EXCEL VBA IDE showing the value of selected variables for the program of pages 49-50 when the control is at the first breakpoint of Figure 44.

While the control is still at the first breakpoint in Figure 44, only variable *iUnits* has been given a value by the user (*iUnits* = 1, for the S.I.), while the other values have been initialized to their default values, namely: zero for *Integer* and *Double* variables (e.g., *Cu*, *g*, and *iType*), and an empty string "" for *String* variables (e.g., *outString*, *sA*, *sL*, *sQ*, and *sV*).

#### *Proceeding Step-by-Step*

Starting at the first breakpoint of Figure 44, until we reach the second breakpoint in that Figure, we will proceed step-by-step. This is accomplished by using the option *Debug>Step Into* (the tedious way), or by simply pressing the [F8] key (the quick and easy way). Every time the [F8] key is pressed, the yellow arrow in the window code moves by one step. For the present program that means that you may have to enter data in *InputBoxes*. Proceed by pressing the [F8] key, until you reach the second breakpoint. Watch the contents of the variables in the *Watch window* as you move step-by-step in the program. When you reach the second breakpoint, the *Watch window* will show the following values:

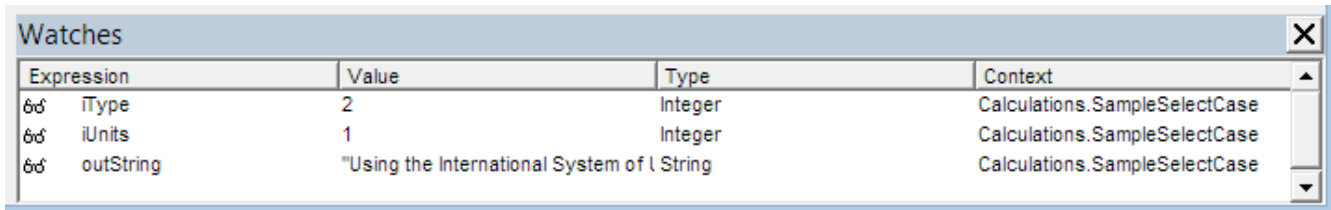


Expression	Value	Type	Context
Cu	1	Double	Calculations.SampleSelectCase
g	9.806	Double	Calculations.SampleSelectCase
iType	2	Integer	Calculations.SampleSelectCase
iUnits	1	Integer	Calculations.SampleSelectCase
outString	"Using the International System of l	String	Calculations.SampleSelectCase
sA	" m^2"	String	Calculations.SampleSelectCase
sL	" m"	String	Calculations.SampleSelectCase
sQ	" m^3/s"	String	Calculations.SampleSelectCase
sV	" m/s"	String	Calculations.SampleSelectCase

Figure 47. The *Watch window* in the EXCEL VBA IDE showing the contents of selected variables when debugging control reaches the second breakpoint in Figure 44.

### Removing variables from the WATCHES window

To remove a variable from the *WATCHES* window, right-click on the variable name in the *WATCHES* window, and select the option *Delete Watch* in the resulting menu. For example, after removing the variables *Cu*, *g*, *sA*, *sL*, *sQ*, and *sV*, the *WATCHES* window in the *EXCEL VBA IDE* will look as follows:

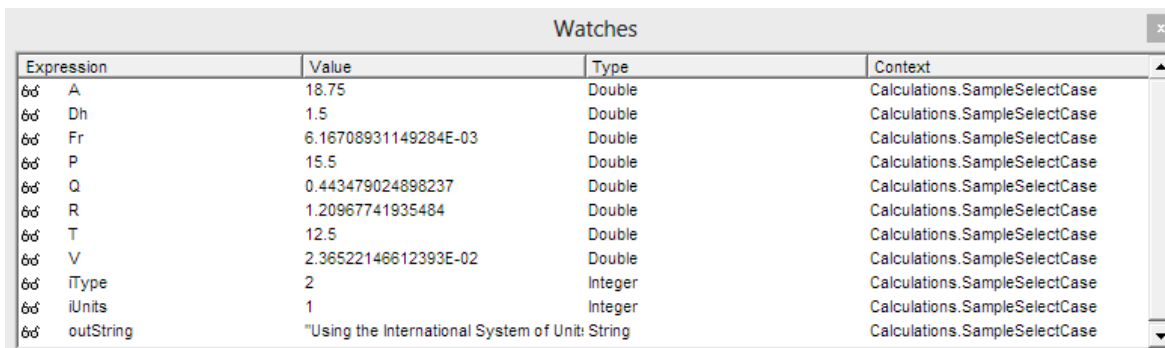


Expression	Value	Type	Context
!Type	2	Integer	Calculations.SampleSelectCase
iUnits	1	Integer	Calculations.SampleSelectCase
outString	"Using the International System of l String		Calculations.SampleSelectCase

Figure 48. The *Watch* window in the *EXCEL IDE* after removing variables *Cu*, *g*, *sL*, *sA*, *sV*, and *sQ*.

### Continue the debugging exercise

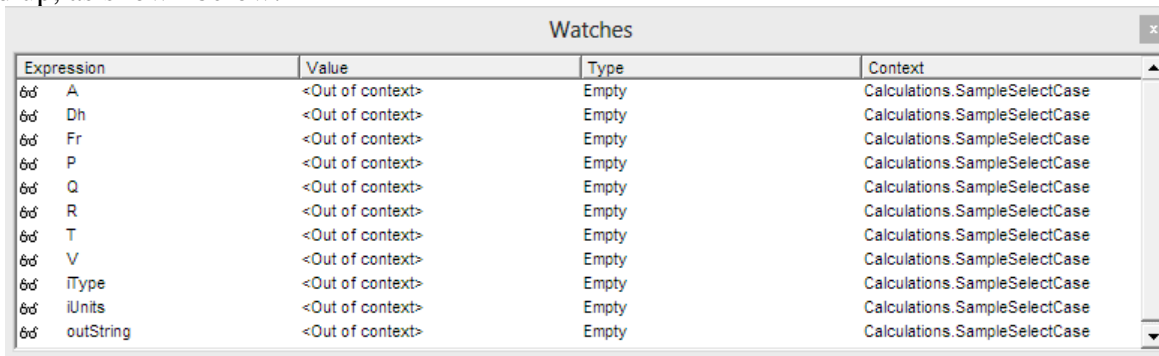
Continuing with our debugging exercise, add watches for variables *A*, *P*, *T*, *R*, *Dh*, *V*, *Q*, and *Fr*. Then proceed step-by-step until the end of the program. Watch how all the variables in the *WATCHES* window change as you step into the program a step at a time. Using some arbitrary values for the variables requested by the *InputBoxes*, the *WATCHES* window after calculating all variables in the program, would show something along these lines:



Expression	Value	Type	Context
A	18.75	Double	Calculations.SampleSelectCase
Dh	1.5	Double	Calculations.SampleSelectCase
Fr	6.16708931149284E-03	Double	Calculations.SampleSelectCase
P	15.5	Double	Calculations.SampleSelectCase
Q	0.443479024898237	Double	Calculations.SampleSelectCase
R	1.20967741935484	Double	Calculations.SampleSelectCase
T	12.5	Double	Calculations.SampleSelectCase
V	2.36522146612393E-02	Double	Calculations.SampleSelectCase
!Type	2	Integer	Calculations.SampleSelectCase
iUnits	1	Integer	Calculations.SampleSelectCase
outString	"Using the International System of Unit: String		Calculations.SampleSelectCase

Figure 49. Showing a number of variables in the program of pages 49-50 after performing all calculations

When you reach the end of the program, and the program ends, all variables in the *Watch* window are cleared up, as shown below:



Expression	Value	Type	Context
A	<Out of context>	Empty	Calculations.SampleSelectCase
Dh	<Out of context>	Empty	Calculations.SampleSelectCase
Fr	<Out of context>	Empty	Calculations.SampleSelectCase
P	<Out of context>	Empty	Calculations.SampleSelectCase
Q	<Out of context>	Empty	Calculations.SampleSelectCase
R	<Out of context>	Empty	Calculations.SampleSelectCase
T	<Out of context>	Empty	Calculations.SampleSelectCase
V	<Out of context>	Empty	Calculations.SampleSelectCase
!Type	<Out of context>	Empty	Calculations.SampleSelectCase
iUnits	<Out of context>	Empty	Calculations.SampleSelectCase
outString	<Out of context>	Empty	Calculations.SampleSelectCase

Figure 50. Empty references for variables in the *Watch* window after a program ends.

### *Variable watching without the WATCHES window*

When the code processing is stopped by a breakpoint, you can look up the value of any variable, without using the *WATCHES* windows, by simply placing the cursor over any instance of the variable of interest in the code. The figure below shows the code for the program of pages 49 and 50 after stopping at a breakpoint. Placing the cursor over the variable *iUnits*, 4 lines above the breakpoint, for example, shows a box with the current value of *iUnits* [*iUnits* = 1] in the following Figure.

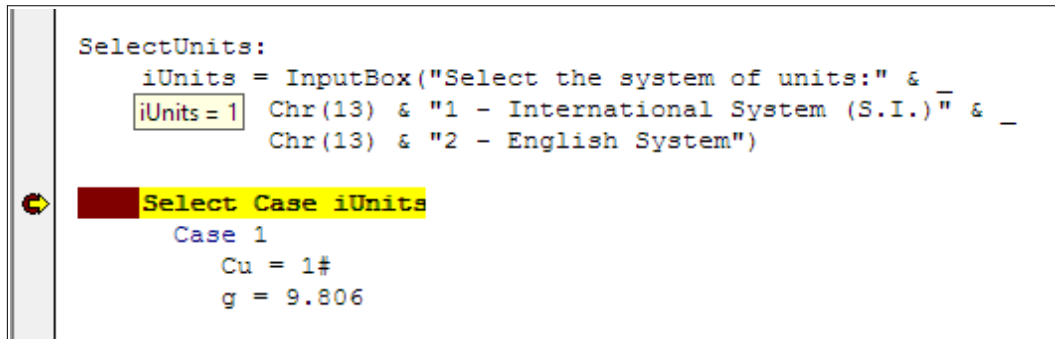


Figure 51. The *Manage Breakpoints* window showing existing breakpoints.

### *The Step Over and Step Out options*

The *Debug>Step Over* (Shift+F8) option is used to skip a procedure call. The program of pages 49-50 do not include a situation where this button can be demonstrated.

The *Debug > Step Out* (Ctrl+Shift+F8) option allows you to resume normal execution of the program after the control has stopped at a breakpoint. For the breakpoints of Figure 44, you could try the following to see the *Step Out* button in action:

- Start the program by using the *Run* button. It will stop at the first breakpoint.
- Press the *Step Out* button, in which case, the program execution will resume up to the next breakpoint.
- Press the *Step Out* button again to reach the end of the program.

### *Removing breakpoints*

After checking that your program is running properly, you will need to remove the breakpoints from your program so it can run uninterrupted when needed. To remove a single breakpoint you can simply click on the line where the breakpoint is located, and use the option *Debug > Toggle Breakpoint*. If you want to remove all the breakpoints in the code, use the option *Debug > Clear All Breakpoints*.

### *In Summary*

Debugging basically consists in using breakpoints and variable watch to step into a program, run it step-by-step, if needed, to determine the reasons why the program is not producing the expected results.

### Note: Hierarchy of logical and arithmetic operations

In an earlier section, in Chapter 1, we addressed the issue of the hierarchy of arithmetic operations, i.e., the priority in which arithmetic operations are performed in a formula. Parentheses are executed first, and then, scanning from left to right, the operations are executed in this order: (1) power (^); (2) multiplication (\*) and division (/); and (3) addition (+) and subtraction (-). If built-in functions, e.g., *Sin*, *Cos*, etc., are used, the hierarchy of arithmetic operations referred to above apply to the argument(s) of those functions if they include arithmetic operations, and then the function gets evaluated. Since the arguments are enclosed in parentheses, the function evaluation would have priority over other operations in which the function is involved, based on the order indicated above. Consider the following example:

$$y = A * \text{Cos}(2*x^2 + x*y + 5) + \text{Sin}(x*y*z + 3)$$

The operations will be calculated as follows (current operations are shown in **italic bold** letters):

*not bold*

$$\begin{aligned} y &= A * \text{Cos}(2*x^2 + x*y + 5) + \text{Sin}(x*y*z + 3) \\ y &= A * \text{Cos}(2*x^2 + x*y + 5) + \text{Sin}(x*y*z + 3) \\ y &= A * \text{Cos}(2*x^2 + x*y + 5) + \text{Sin}(x*y*z + 3) \\ y &= A * \text{Cos}(2*x^2 + x*y + 5) + \text{Sin}(x*y*z + 3) \\ y &= A * \text{Cos}(2*x^2 + x*y + 5) + \text{Sin}(x*y*z + 3) \end{aligned}$$

Logical operations also have a hierarchy. Except for the case in which three comparison statements were linked by *And* particles in Figure 39, most of the examples of logical statements used so far have involved one logical statement affected by *Not* or two logical statements linked by *And*, *Or*, or *Xor*. With one or two logical statements combined, we are not so concerned on hierarchy of operations. All what need to now in these cases are the truth tables shown in page 34.

For three or more logical operations combined, it is necessary to follow the hierarchy of operations (also referred to as *operator precedence*) shown in the following list. Logical operators have precedence over arithmetic operations, since the later may be involved in the comparison expressions that constitute the elementary logical operations. However, since comparisons need to be evaluated first to determine if the statements involved are *TRUE* or *FALSE*, comparison operators (=, <, >, >=, <=, <=) are evaluated before any of the logical operators. As with arithmetic operators, parentheses have higher priority, and operators of the same priority are evaluated from left to right. Here is the list of the logical operator priority:

1. *Not*
2. *And*
3. *Or*
4. *Xor*

Thus, in the example of Figure 39, namely,

`((iType <> 1) And (iType <> 2) And (iType <> 3))`

The expression is evaluated as follows (current operation shown in **italic bold** letters):

```
((iType < > 1) And (iType < > 2) And (iType < > 3))
((iType < > 1) And (iType < > 2) And (iType < > 3))
((iType < > 1) And (iType < > 2) And (iType < > 3))
```

Other examples follow:

- [1].
- ```
y > 0 And x < 0 Or y > x
y > 0 And x < 0 Or y > x
y > 0 And x < 0 Or y > x
```
- [2].
- ```
x > y+1 And Not y>0 And x>1 Or t < 1 Xor y>t And t > 10
x > y+1 And Not y>0 And x>1 Or t < 1 Xor y>t And t > 10
x > y+1 And Not y>0 And x>1 Or t < 1 Xor y>t And t > 10
x > y+1 And Not y>0 And x>1 Or t < 1 Xor y>t And t > 10
x > y+1 And Not y>0 And x>1 Or t < 1 Xor y>t And t > 10
x > y+1 And Not y>0 And x>1 Or t < 1 Xor y>t And t > 10
x > y+1 And Not y>0 And x>1 Or t < 1 Xor y>t And t > 10
```

Keep in mind the hierarchy of arithmetic, comparison, and logical operators when using logical statements in decision statements (*If ... End If*, *If ... ElseIf ... End If*), so that your logical statements will get the correct logical (or Boolean) value in the program process.

### Using a *EXCEL* worksheet as program interface for the open-channel flow program

The program of pages 49-50 utilizes *InputBoxes* and *MsgBoxes* for input and output, respectively. An alternative approach to activate the program is to use a *EXCEL* worksheet as an interface, so that the input data can be placed in worksheets cells and the program activated by one or more push buttons. There is no unique way that this interface can be designed. One possible layout is shown in Figure 52, in next page.

The code associated with the *Calculate Flow* button is the macro entitled *UniformFlowCalculations*. The code for macro *UniformFlowCalculations* is thoroughly commented to explain its operation. In comparison to the code of pages 49-50, we removed the *InputBoxes* and all but a couple of warning *MsgBoxes*. Input and output of data is performed through the cells in worksheet *UniformFlow*. The code includes several lines whose only purpose is to change the contents of cells in the worksheet. Access to individual cells is done by using the *Range* object within the *Worksheets* object. The references to specific cells are similar to this one: `Worksheets (UniformFlow) .Range ("F8") .Value`. As in the case of the code of pages 49-50, macro *UniformFlowCalculations* uses integer variables *iUnits* and *iType* to decide the system of units and the type of cross-section to use in the calculation. There is no error trap for the case where the user misses entering one of the geometric elements, but there are error traps in case the user chooses values other than 1 and 2 for *iUnits*, or other than 1, 2, 3, and 4 for *iType*. If *iUnits* = 2 (E.S.) is selected for the system of units, the program calls another subroutine with the statement *Call ESLabelsReset*. The later, macro *ESLabelsReset*, simply changes all unit references from S.I. to E.S. units. The code for this macro is shown in Figure 54. The case in which the data enter produces the unrealistic condition that the depth of flow is larger than the diameter in a circular cross-section is handled by sending the control to a label called *EndCalculation*, which simply links to the very last statement in the code, namely, *End Sub*.

	A	B	C	D	E	F	G	H	I	J	
1											
2											
3			<b>UNIFORM FLOW IN OPEN CHANNELS</b>								
4			<b>Select system of Units: 1 – S.I. 2 – E.S.:</b>					<input type="text" value="1"/>			
5			<b>Select type of cross-section: → → → →</b>					<input type="text" value="1"/>			
6			1 – Trapezoidal 2 – Rectangular 3 – Triangular 4 – Circular								
7			<b>Trapezoidal, Rectangular, or Triangular Geometry:</b>								
8			Bottom Width of Cross-Section, b(m):					<input type="text" value="1"/>			
9			Side Slope, m:					<input type="text" value="1"/>			
10			<b>Circular Geometry:</b>								
11			Diameter, D(m):					<input type="text"/>			
12			<b>Flow Depth for All Types of Cross-Sections:</b>								
13			Flow Depth, y(m):					<input type="text" value="1"/>			
14			<b>Channel Bed Properties:</b>								
15			Channel Bed Slope, S0:					<input type="text" value="0.0001"/>			
16			Manning's resistance coefficient, n:					<input type="text" value="0.012"/>			
17			<b>UNIFORM FLOW CALCULATIONS</b>								
18			Cross-sectional area, A(m <sup>2</sup> ):					<input type="text"/>			
19			Wetted Perimeter, P(m):					<input type="text"/>			
20			Top Width of Cross-section, T(m):					<input type="text"/>			
21			Hydraulic Radius, R(m):					<input type="text"/>			
22			Hydraulic Depth, Dh(m):					<input type="text"/>			
23			Flow Velocity, V(m/s):					<input type="text"/>			
24			Flow Discharge, Q(m <sup>3</sup> /s):					<input type="text"/>			
25			Froude Number, Fr:					<input type="text"/>			
26											

Calculate Flow

Clear Output

Reset Worksheet

Figure 52. A possible worksheet interface for the open-channel calculations program first introduced in pages 49-50.

```

Sub UniformFlowCalculations
    Dim iUnits As Integer, iType As Integer
    Dim Cu As Double, n As Double, S0 As Double
    Dim b As Double, m As Double, y As Double
    Dim D As Double, theta As Double, g As Double
    Dim A As Double, P As Double, R As Double, Dh As Double
    Dim T As Double, V As Double, Q As Double, Fr As Double

    iUnits = CInt(Worksheets("UniformFlow").Range("F4").Value)
    Select Case iUnits
        Case 1 'S.I. Units
            Cu = 1.0 : g = 9.806
        Case 2 'E.S. Units
            Cu = 1.486 : g = 32.2
            Call ESLabelsReset
        Case Else 'iUnits not 1 or 2 - S.I. used by default
            MsgBox("Wrong selection for system of units." & Chr(13) & _
                "S.I. system of units will be used.")
            Worksheets("UniformFlow").Range("F4").Value = 1
            Cu = 1.486 : g = 9.806
    End Select

    iType = CInt(Worksheets("UniformFlow").Range("F5").Value)
    Select Case iType
        Case 1 'Trapezoidal

```



```

        b = CDBl(Worksheets("UniformFlow").Range("F8").Value) 'Read bottom width
        m = CDBl(Worksheets("UniformFlow").Range("F9").Value) 'Read side slope
        y = CDBl(Worksheets("UniformFlow").Range("F13").Value) 'Read flow depth
        Worksheets("UniformFlow").Range("F11").Value = "" 'Diameter cell is emptied
        A = (b+m*y)*y : P = b + 2*y*Sqr(1+m^2) : T = b+2*m*y 'Calculate geometry
Case 2 'Rectangular
        b = CDBl(Worksheets("UniformFlow").Range("F8").Value) 'Read bottom width
        y = CDBl(Worksheets("UniformFlow").Range("F13").Value) 'Read flow depth
        Worksheets("UniformFlow").Range("F9").Value = "" 'Side slope cell is emptied
        Worksheets("UniformFlow").Range("F11").Value = "" 'Diameter cell is emptied
        A = b*y : P = b + 2*y : T = b 'Calculate geometry
Case 3 'Triangular
        m = CDBl(Worksheets("UniformFlow").Range("F9").Value) 'Read side slope
        y = CDBl(Worksheets("UniformFlow").Range("F13").Value) 'Read flow depth
        Worksheets("UniformFlow").Range("F8").Value = "" 'Bottom width cell is emptied
        Worksheets("UniformFlow").Range("F11").Value = "" 'Diameter cell is emptied
        A = 1./2.*m^2*y : P = 2*y*Sqr(1+m^2) : T = 2*m*y 'Calculate geometry
Case 4 'Circular
        Worksheets("UniformFlow").Range("F8").Value = "" 'Bottom width cell is emptied
        Worksheets("UniformFlow").Range("F9").Value = "" 'Side slope cell is emptied
        D = CDBl(Worksheets("UniformFlow").Range("F11").Value) 'Read diameter
        y = CDBl(Worksheets("UniformFlow").Range("F13").Value) 'Read flow depth

        If D<y Then 'If diameter is less than depth, end calculation
            MsgBox("You enter D = " & CStr(D) & " < y = " & CStr(y) & ". Try again")
            Go To EndCalculation
        End If
        theta = WorksheetFunction.Acos(1 - 2*y/D) 'Calculate central half angle
        A = D^2/4*(theta - Sin(theta)*Cos(theta)) 'Calculate x-sectional area
        P = theta*D : T = D*sin(theta) 'Calculate P and T
Case Else 'If wrong value of iType is entered, default is trapezoidal
        MsgBox("Wrong selection for cross-section type" & Chr(13) & _
            "Trapezoidal cross-section with default values will be used")
        Worksheets("UniformFlow").Range("F5").Value = 1 'default iType = 1
        Worksheets("UniformFlow").Range("F8").Value = 1.0 'default b = 1.0
        Worksheets("UniformFlow").Range("F9").Value = 1.0 'default m = 1.0
        Worksheets("UniformFlow").Range("F11").Value = "" 'default D = empty
        Worksheets("UniformFlow").Range("F13").Value = 1.0 'default y = 1.0
        Worksheets("UniformFlow").Range("F15").Value = 0.0001 'default S0 = 0.0001
        Worksheets("UniformFlow").Range("F16").Value = 0.012 'default n = 0.012
        b = CDBl(Worksheets("UniformFlow").Range("F8").Value) 'Read default b
        m = CDBl(Worksheets("UniformFlow").Range("F9").Value) 'Read default m
        y = CDBl(Worksheets("UniformFlow").Range("F13").Value) 'Read default
        A = (b+m*y)*y : P = b + 2*y*Sqr(1+m^2) : T = b+2*m*y 'Calculate geometry
End Select

S0 = CDBl(Worksheets("UniformFlow").Range("F15").Value) 'Enter bed slope
n = CDBl(Worksheets("UniformFlow").Range("F16").Value) 'Enter Manning's n, calculate:
R = A/P : Dh = A/T : V = Cu*Sqr(S0)/n * R^(2./3.)*Sqr(S0) : Q = V*A : Fr = V/Sqr(g*Dh)

Worksheets("UniformFlow").Range("F18").Value = A 'Output Area
Worksheets("UniformFlow").Range("F19").Value = P 'Output Wetted Perimeter
Worksheets("UniformFlow").Range("F20").Value = T 'Output Top Width
Worksheets("UniformFlow").Range("F21").Value = R 'Output Hydraulic Radius
Worksheets("UniformFlow").Range("F22").Value = Dh 'Output Hydraulic Depth
Worksheets("UniformFlow").Range("F23").Value = V 'Output Flow Velocity
Worksheets("UniformFlow").Range("F24").Value = Q 'Output Flow Discharge
Worksheets("UniformFlow").Range("F25").Value = Fr 'Output Froude Number
EndCalculation: 'Label connecting to End Sub
End Sub

```

Figure 53. Code for the macro entitled *UniformFlowCalculations* associated with the *Calculate Flow* button in Figure 52.

```

Sub ESLabelsReset
    'All labels are reset to the English System of Units
    Worksheets("UniformFlow").Range("E8").Value = _
        "Bottom Width of Cross-Section, b(ft):"
    Worksheets("UniformFlow").Range("E11").Value = _
        "Diameter, D(ft):"
    Worksheets("UniformFlow").Range("E13").Value = _
        "Flow Depth, y(ft):"
    Worksheets("UniformFlow").Range("E18").Value = _
        "Cross-sectional area, A(ft^2)"
    Worksheets("UniformFlow").Range("E19").Value = _
        "Wetted Perimeter, P(ft):"
    Worksheets("UniformFlow").Range("E20").Value = _
        "Top Width of Cross-section, T(ft):"
    Worksheets("UniformFlow").Range("E21").Value = _
        "Hydraulic Radius, R(ft):"
    Worksheets("UniformFlow").Range("E22").Value = _
        "Hydraulic Depth, Dh(ft):"
    Worksheets("UniformFlow").Range("E23").Value = _
        "Flow Velocity, V(fps)"
    Worksheets("UniformFlow").Range("E24").Value = _
        "Flow Discharge, Q(cfs)"
    Worksheets("UniformFlow").Range("E25").Value = _
        "Froude Number, Fr:"
End Sub

```

Figure 54. Code for the macro entitled *ESLabelsReset*, which is called from the method *UniformFlowCalculations* of Figure 52 to adjust labels in the interface for units of the E.S..

The *Clear Output* button in Figure 52 is associated with the macro *ClearOutput* shown in Figure 55. The macro *ClearOutput* simply places empty strings in the output cells of the interface of Figure 52. These cells correspond to the range "F18:F26" in the interface. Thus, the code for macro *ClearOutput* includes only one line as shown in Figure 55.

```

Sub ClearOutput
    'Output cells, in F18:F26 are emptied
    Worksheets("UniformFlow").Range("F18:F26").Value = ""
End Sub

```

Figure 55. Code for the macro entitled *ClearOutput*, which places blank strings in the output cells of the interface of Figure 52.

The *Reset Worksheet* button is associated with the *ResetWorksheet* macro. This macro resets the system of units to, and the cell labels to units of, the S.I.; resets the type of cross-section to trapezoidal, and provides default values of the geometry and properties of the channel button for a trapezoidal cross-section. The macro *ResetWorksheet* also calls macro *ClearOutput* to clear the output cells in the interface of Figure 52. A listing of the *ResetWorksheet* macro is presented in Figure 56.

```

Sub ResetWorksheet
    'Default values for all variables: S.I., Trapezoidal X-section
    Worksheets("UniformFlow").Range("F4").Value = 1          'Default iUnits
    Worksheets("UniformFlow").Range("F5").Value = 1          'Default iType
    Worksheets("UniformFlow").Range("F8").Value = 1.0        'Default b
    Worksheets("UniformFlow").Range("F9").Value = 1.0        'Default m
    Worksheets("UniformFlow").Range("F11").Value = ""        'Default D
    Worksheets("UniformFlow").Range("F13").Value = 1.0        'Default y
    Worksheets("UniformFlow").Range("F15").Value = 0.0001     'Default S0
    Worksheets("UniformFlow").Range("F16").Value = 0.012      'Default n

```



```

'All labels are reset to S.I. units
Worksheets("UniformFlow").Range("E8").Value = _
    "Bottom Width of Cross-Section, b(m):"
Worksheets("UniformFlow").Range("E9").Value = _
    "Side Slope, m:"
Worksheets("UniformFlow").Range("E11").Value = _
    "Diameter, D(m):"
Worksheets("UniformFlow").Range("E13").Value = _
    "Flow Depth, y(m):"
Worksheets("UniformFlow").Range("E18").Value = _
    "Cross-sectional area, A(m^2)"
Worksheets("UniformFlow").Range("E19").Value = _
    "Wetted Perimeter, P(m):"
Worksheets("UniformFlow").Range("E20").Value = _
    "Top Width of Cross-section, T(m):"
Worksheets("UniformFlow").Range("E21").Value = _
    "Hydraulic Radius, R(m):"
Worksheets("UniformFlow").Range("E22").Value = _
    "Hydraulic Depth, Dh(m):"
Worksheets("UniformFlow").Range("E23").Value = _
    "Flow Velocity, V(m/s)"
Worksheets("UniformFlow").Range("E24").Value = _
    "Flow Discharge, Q(m^3/s)"
Worksheets("UniformFlow").Range("E25").Value = _
    "Froude Number, Fr:"
Call ClearOutput
End Sub

```

Figure 56. Code for the macro entitled *ResetWorksheet* which resets the interface to the S.I. and provides default values of the input variables.

One important note in using the interface of Figure 52: if you enter 2 in cell F4 (i.e., selecting units of the E.S.), the labels in the worksheet will not change to units of the E.S. until the *Calculate Flow* button is pressed. This may mislead the user as the interface will be showing all units in S.I. Thus, the interface could be improved by creating a *Select Units* button that uses an *InputBox* to enter the value of *iUnits*, and a *Select Cross-Section* button that uses another *InputBox* to enter the value of *iType*. The *Select Units* button will include a macro that would instruct the changing of unit labels to the proper units. The *Select Cross-Section* button can be used to label the input cells properly for the selected cross-section.

It is obvious, therefore, that there is no unique design for a *EXCEL* interface to activate calculations that solve a particular problem. The development of an interface typically includes providing enough information for the user to understand the input and output from the program, and a personal touch from the designer in terms of the selection of font type and size, frames, and cell colors used in the design. The design of the interface of Figure 52 involved using 4 different colors for the cell backgrounds, two different colors and two sizes of fonts, as well as selecting the size and labels for the buttons that activate the macros.

### Programming numerical algorithms vs. programming "bells-and-whistles"

The code of Figure 53 includes mostly statements ~~listed~~ <sup>that</sup> are strictly necessary for the input, processing, and output of the numerical calculations involved. On the other hand, the code of Figures 54, 55, and 56 deal mostly with updating labels and clearing cells in the interface. Programming the interface details and the behavior of the interface in response to an action is referred to as *event-driven programming*. Pressing a button, or effecting a change in other form controls, constitutes event-driven programming. We could refer to as "bells-and-whistles" programming, which is an integral part of modern day computing.

On the other hand, in the original version of the open-channel program, shown in pages 49-50, we used *InputBoxes* and *MsgBoxes* for input and output, respectively, while the rest of the code was purely numerical programming and string manipulation. *InputBoxes* and *MsgBoxes* use the computer's graphical interface, thus, in the program of pages 49-50, these two commands are the only "bells-and-whistles" programming component used.

In an earlier section, in Chapter 1, we indicated that to insert a button (or command button) we use the *Form Controls > Button* option in the *Developer* tab, and showed how to link push buttons to macros. The *Form Controls* menu includes other form controls such as *combo boxes*, *check boxes*, *spin buttons*, etc. So far we have used only command buttons, and will keep our instructions on the use of form controls to those buttons only, so that we can concentrate on the numerical processing aspects of VBA programming. If you are interested in learning the programming of other objects in the *Form Controls* menu, see the following link:

<http://support.microsoft.com/kb/291073>