CHAPTER 2 MESSAGE BOX FUNCTIONS, VECTOR ADDITION, INTRODUCTION TO LOOPS,

This chapter contains a number of disparate topics. First, a more detailed consideration is given to **Message Boxes** and **Message Box functions** which are used in an interactive version of the Chapter 1 **Temperature Converter**. Next, a vector application is presented that will add up to six vectors. One of the purposes of this application is to demonstrate the tediousness of coding repetitive operations. The concept of using loops to manage repetitive operations is examined and applied to a simplified version of the vector addition application. Finally, an application is presented that solves 1st Condition of Equilibrium problems having up to six forces where two of the forces have unknown magnitudes.

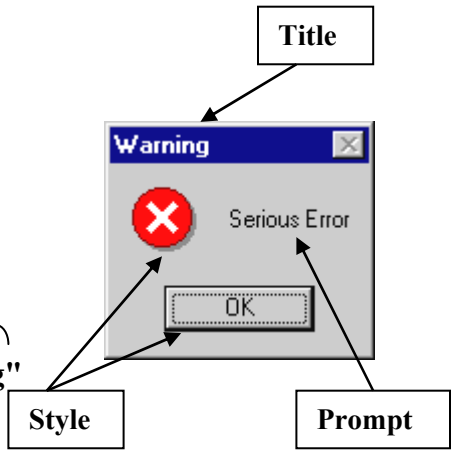
2.1 The Message Box and Message Box Function

2.1.1 Message Box Syntax

The simplified version of the message box given in Chapter 1 contained only the prompt; the complete version consists of 3 components : **Prompt**, **Style**, and **Title** separated by commas.

MsgBox *Prompt, Style, Title*

MsgBox "Serious Error", vbCritical + vbOKOnly, "Warning"



The **prompt** holds the message to be displayed in the box. Anything contained inside quotes, or defined by a string, will appear as text; anything not enclosed in quotes will be interpreted as a variable. Variables and text can be combined together in the prompt by linking them with ampersands **&** (refer to the next example). A plus + cannot be used.

The **style** consists of two terms : a constant that adds an emphasis symbol such as a question mark, exclamation mark, etc. (**vbCritical** in this example), and a constant that sets the button type (**vbOKOnly** in this example). The two terms must be joined by a plus + sign or an ampersand **&**. The button type constant can be omitted in which case the **OKOnly** button becomes the default choice. The emphasis term is optional; if it's omitted no symbol is displayed.

The **title** contains whatever is desired in the title bar of the message box and must be enclosed in quotes, or defined by a string. As with the prompt, variable values can be added provided they're linked to the text via an ampersand.

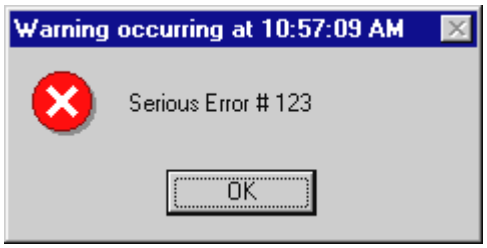
Visual Basic Constants (vbConstants) In math and physics many numerical constants are symbolically designated with letters or names (for example, pi, e, G, g, k, μ_o, ε_o, etc) as it is much more convenient to refer to the value symbolically rather than numerically. **VB** also uses a number of predefined constants to describe certain things such as characteristics and properties, and to use with certain commands and functions. The white X in the previous message box is numerically represented by the number 16, but is also represented by the vbConstant **vbCritical** (which is probably easier to remember). All **vbConstants** begin with the prefix "vb". A number of **vbConstants** are listed on page 2-4. The numerical value of any vbConstant can be found using the **Val() function** . For example, the following code line returns the value 3 into cell **H10** : **Worksheets("Sheet1") .Range("H10") = Val(vbYesNoCancel)** .

Examples of message boxes :

MsgBox "Serious Error # " & 123, vbCritical, "Warning occurring at " & Time

The ampersand & joins the text in quotes to the number 123 (which could also be represented by a variable)

Time is a built-in function that displays the value of the internal computer clock. Any other variable may be included.



The format of the prompt message can be controlled using various **vbConstants** :

MsgBox "Excel became unstable at " & Time & vbCr & "and is shutting down" _
& vbTab & "Oh No !!", vbCritical, "Not Again !!"

vbTab shifts ("tabs") the text over a number of spaces.

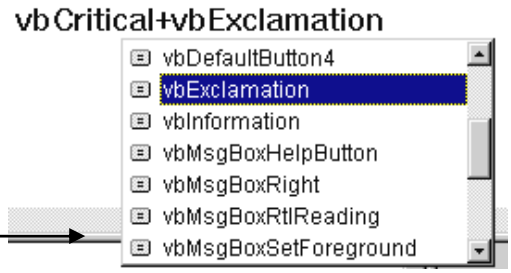
Standard continuation to next line character

vbCr is a carriage return constant that harks back to the era of typewriters and early printers. The paper was wrapped around a cylindrical drum known as a carriage; to start a new line the carriage had to be returned to a position where the left end was positioned in front of the key strike position.

A common mistake is to attempt to specify a **Message Box** using only prompt and title components (which *is* valid for an **Input Box**). For a **Message Box**, the choice is between the simplified version consisting of the prompt only, and the complete version composed of the prompt, style, and title.

Caution: If you do not respond to a message box and instead enter the **VB Editor** an apparent screen freeze will occur. Click on the **Excel Task Bar** button at the top or bottom of your screen (outside of the **Excel** window) to return to the worksheet and then click **OK** on the message box.

Note : Two emphasis conditions cannot be combined. For example, you can present a question mark symbol , or the critical symbol, but not both together.



A more complete list of constants will automatically appear as you type in any constant.

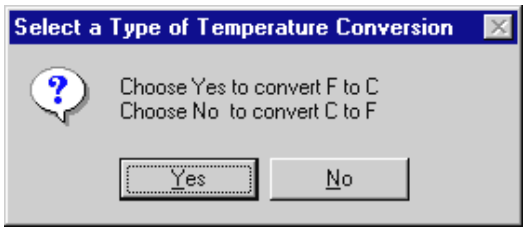
2.1.2 Message Box *Function* Syntax

The message box described so far is passive as it only presents the user with a message. In many programs it is necessary to query the user as to the situation, or how to proceed. This can be accomplished with the **Message Box *Function*** that includes a number of button options (such as **Yes**, **No**, **Cancel**, **Retry**, etc); the user's response is then returned as the value of the function. This interactive capability allows the program to quiz the user as to certain choices and decisions and then use the response to guide subsequent program execution. [The syntax for the message box function described below is a subset of the complete syntax which is available in the **Help** menu accessed from **the VB Editor** (not the worksheet).

A **Message Box *Function*** has three components : **Prompt** , **Style** , and **Title** ; however, these components must now be enclosed in parentheses. The function returns an integer value that describes the user's response and which is assigned to a variable using an equal sign. The variable that receives the integer response must be **Dim**-med as an **Integer**. For example, if the user clicks on a **Yes** button the message box function returns a value of 6. The integer value associated with any button also corresponds to a **vbConstant** that describes the button : a **Yes** button is represented by the constant **vbYes** which has the numerical value of 6.

Example : This example has been taken from **Example # 1** pg 2-5. The following code line produces the message box shown below and stores the user's reply in the variable **Response**.

```
Response = MsgBox("Choose Yes to convert F to C" & vbCr & "Choose No  to convert C to F", _
vbYesNo + vbQuestion,  "Select a Type of Temperature Conversion")
```



For the sake of clarity, it is recommended that the prompt, style, and title be defined by specific variables. In the code given below variables **Prompt** and **Title** are declared as strings, and variable **Style** must be **Dim**-med as an **Integer** since it receives the sum of two **vbConstants** which have integer values.

```
Prompt = "Choose Yes to convert F to C" & vbCr & "Choose No  to convert C to F"
Style = vbYesNo + vbQuestion
Title = "Select a Type of Temperature Conversion"

Response = MsgBox(Prompt, Style, Title)
```

Message Box Constants

Constants to Set Button Type

Constant	Description
vbOKOnly	OK button only (default)
vbOKCancel	OK and Cancel buttons
vbYesNo	Yes and No buttons
vbYesNoCancel	Yes, No, and Cancel
vbRetryCancel	Retry and Cancel buttons
vbAbortRetryIgnore	Abort, Retry, and Ignore

Constants for MsgBox Function Return Value

Constant	User Clicked
vbOK	OK
vbCancel	Cancel
vbYes	Yes
vbNo	No
vbRetry	Retry
vbIgnore	Ignore

Constants to Add Emphasis

Constant	Description
vbCritical	Critical message (red circle & white X)
vbExclamation	Warning message
vbQuestion	Warning query (question mark)
vbInformation	Information message

Miscellaneous Constants

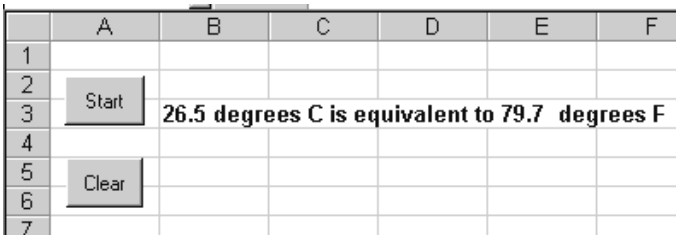
Constant	Description
vbDefaultButton1	1st button is default
vbDefaultButton2	2nd button is default
vbDefaultButton3	3rd button is default
vbSystemModal	System modal msg box
vbApplicationModal	Applicat. modal msg box

Constants that Control the Display Format

Constant	Description	Chr Equivalent
vbTab	Tab	Chr$(9)
vbCr	Carriage return	Chr(13)
vbLf	Linefeed (skips a line)	Chr(10)
vbCrLf	Carriage return & Linefeed	Chr(13)+Chr(10)

Example # 1 : Temperature Converter (If statement & MsgBox/ InputBox version) This version of the Chapter 1 exercise queries the user as to the type of conversion desired (**C** to **F**, or **F** to **C**), and then uses an **If** statement to identify the user response and cause execution of the relevant code. Open and save a new Workbook and add two buttons :

Name = cmdStart, **Caption = Start** ;
Name = cmdClear, **Caption = Clear** .



Enter the following code :

```
Option Explicit

Private Sub cmdStart_Click()
Dim Response As Integer, F As Double, C As Double
Dim Prompt As String, Title As String, Style As Integer

MsgBox "The time now is " & Time, vbExclamation + vbOKOnly, "Computer Time"

Prompt = "Choose Yes to convert F to C" & vbCr & "Choose No  to convert C to F"
Title = "Select a Type of Temperature Conversion"
Style = vbYesNo + vbQuestion
Response = MsgBox(Prompt, Style, Title)

If Response = vbYes Then
   F = InputBox("Enter temperature in Fahrenheit", "Fahrenheit to Centigrade Conversion")
   C = (5 / 9) * (F - 32)
    Range("B3") = F & " degrees F is equivalent to " & C & "  degrees C"
Else
   C = InputBox("Enter temperature in Centigrade", "Centigrade to Fahrenheit Conversion")
   F = (9 / 5) * C + 32
   Range("B3") = C & " degrees C is equivalent to " & F & "  degrees F"
End If

End Sub

Private Sub cmdClear_Click()
Range("B3").ClearContents
End Sub
```

Observe that **Prompt** and **Title** are **String**s, while **Style** and **Response** are **Integers**.

The numerical value **6** could have replaced **vbYes**.

As was noted in Chapter 1, **Example # 2b)**, an error message will be generated if the user inadvertently enters non-numeric data into the **Input Box**, or if the **Cancel** button is selected, or if the **Input Box** is closed. The **IsNumeric** function can be used to program around this problem as in **Example # 2b)** .

There are three parts to the main program. The first part begins with a superfluous **Message Box** which uses the built-in **Time** function to display the current time. While this action part no practical value it will serve to illustrate the difference between a **Message Box** and a **Message Box *function*** .

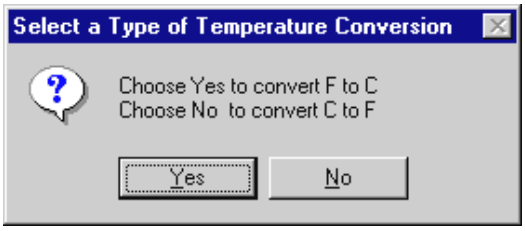
MsgBox "The time now is " & Time, vbExclamation + vbOKOnly, "Computer Time"

Prompt **Style** **Title**

A **Message Box *function*** then queries the user as to the type of temperature conversion desired : **F to C** or **C to F**. The user response is stored in integer variable **Response** : a **Yes** response has the value **6**, a **No** response the value **7** (the response is equivalently represented by either of the constants **vbYes** or **vbNo**). The **MsgBox function** with the information for **Prompt**, **Style**, and **Title** entered directly is :

Response = MsgBox("Choose Yes to convert F to C" & vbCr & "Choose No to convert C to F", _
vbYesNo + vbQuestion, "Select a Type of Temperature Conversion")

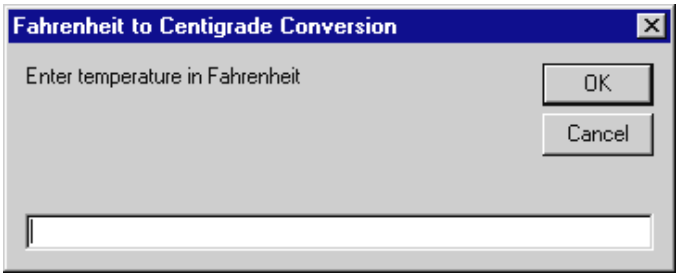
Constant **vbYesNo** has a value of 4, and **vbQuestion** a value of 32. The total of 36 could replace the sum **vbYesNo + vbQuestion** in the **MsgBox function**.



The third part of the program provides an **If** statement to detect which response the user has made and then an **Input Box** is used to obtain the temperature value to be converted. Finally, the relevant calculation is made and the result is written into cell **B3**.

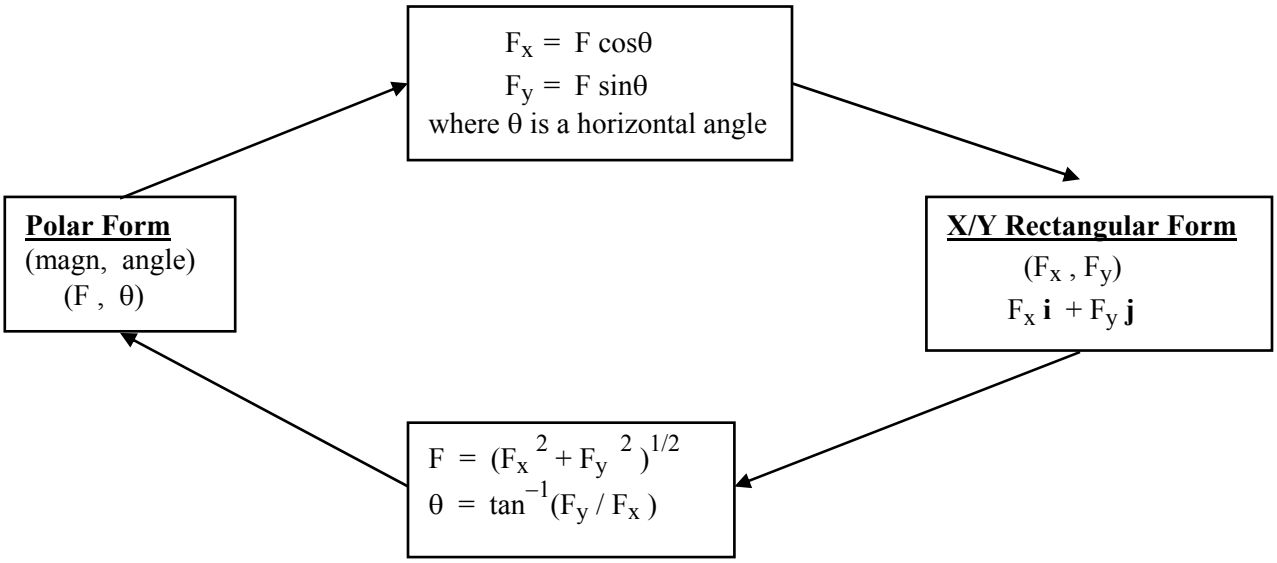
F = InputBox("Enter temperature in Fahrenheit", "Fahrenheit to Centigrade Conversion")

Prompt Title



2.2 Vector Addition: Background Theory

As the physical constraints[1] that govern the interaction of objects must consider the effect of all vectors present it is clearly essential to be able to add vectors together. The most useful form of vector addition is the **analytical** or **component method** which requires that the vectors be in **X/ Y component** form and then separately adds up all the X and Y components. Unfortunately, vectors associated with physical phenomena are often presented in **polar form** consisting of a **magnitude** and an **angle**. The chart below summarizes the process for converting between polar form and X/Y components and presents the mathematical description of component addition.



Component Addition

$$Rx = \Sigma F_x$$
$$Ry = \Sigma F_y$$

Problem # 1 at the end of Chapter 1 dealt with converting the polar form of a vector into X & Y components. **Exercise # 1** that follows demonstrates the inverse process of obtaining the polar from the rectangular components. **Application # 1a)** then creates a component addition program that can take up to 6 vectors in polar form, find their X and Y components, and then add the components to producethe resultant. **Application 1b)** uses a more efficient loop based method that is capable of adding any number of vectors.

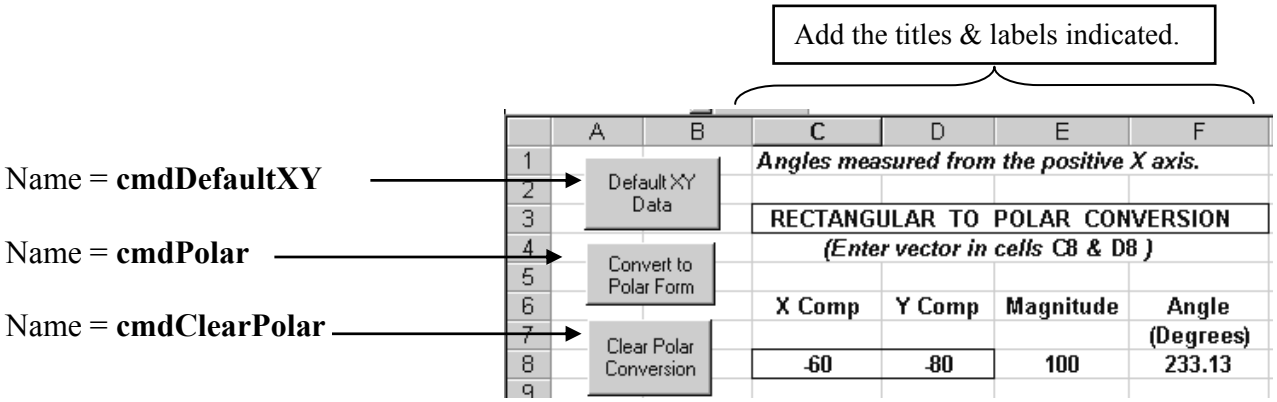
---

[1] such as Newton's Laws. Although Conservation of Momentum is a fundamental relationship that is always valid for an entire system of interacting objects, it simply reflects the 3rd Law from which it may be derived. [The 3rd Law indicates that there are no isolated forces in nature; forces arise due to the simultaneous interaction of two objects. The basic constraint is that when two objects interact the forces which occur must have the same magnitude, act for identical periods of time, and act in opposite directions.]

Example # 2 : Rectangular to Polar Conversion

The process of converting from X/ Y components to the polar form can become complicated by the limitations of the built-in inverse tangent function. As with most calculators, the inverse tangent function **Atn(Y/X)** that comes with VBA only provides angles over the range of -90^{o} to $+90^{o}$ which means that second and third quadrant angles are simply not produced (they're erroneously reflected into the fourth and first quadrants respectively). In this exercise a series of **If** statements are used to identify which quadrant the vector is in and program around the limitations. Fortunately the **Excel** worksheet does have a built-in function **ATAN2(X, Y)** that gives angles in all four quadrants and which can be accessed via the **Application.WorksheetFunction** approach as described in conjunction with the Snell's Law problem of Chapter 1.

Add 3 command buttons giving them the names and captions as shown below. Type in titles **RECTANGULAR TO POLAR CONVERSION**, **X Comp**, etc. as shown. Make certain that cells **C8** through **F8** are empty; this is where the inputs and outputs will appear.



Add the following code to the appropriate subroutines remembering that **Option Explicit** must appear at the very top of the code window.

It is common practice to document (or describe) programs by means of comment statements, which are lines preceded by an apostrophe or the letters **Rem** (short for **remark**). Comment statements allow anyone reading the code to obtain an idea of purpose of the individual code fragments; they are ignored by the compiler. Although I'll generally use textboxes in these notes, I've included comments here.

```
Option Explicit

Private Sub cmdPolar_Click()
Dim Magnitude As Double, Angle As Double
Dim Xcomp As Double, Ycomp As Double

' Factor converts radians to degrees
Const Factor As Double = 180 / 3.1415926535897

' Read in the components from the worksheet
Xcomp = Worksheets("Sheet1").Range("C8").Value
Ycomp = Worksheets("Sheet1").Range("D8").Value
```

```
' Calculate the magnitude and return it directly to the desired cell
Worksheets("Sheet1").Range("E8") = Sqr(Xcomp ^ 2 + Ycomp ^ 2)

' The If statements identify which quadrant vector is in,  or whether it lies along the Y axis.
' The angle is determined using the inverse tangent function Atn(),  except for the Y axis.

If Xcomp = 0 And Ycomp > 0 Then Angle = 1.570796        ' vector on positive Y axis
If Xcomp = 0 And Ycomp < 0 Then Angle = -1.570796       ' vector on negative Y axis

' vector in 3rd quadrant
If Xcomp < 0 And Ycomp < 0 Then Angle = (3.14159265 + Atn(Ycomp / Xcomp))

' vector in 2nd quadrant
If Xcomp < 0 And Ycomp > 0 Then Angle = (3.14159265 – Abs(Atn(Ycomp / Xcomp)))
      ' Note:  we could also omit the Abs function and use (3.14159265 + (Atn(Ycomp / Xcomp)))

' vector in 1st or 4th quadrant;  Atn( ) gives correct angle
If (Xcomp > 0 And Ycomp > 0) Or (Xcomp > 0 And Ycomp < 0) Then Angle = Atn(Ycomp / Xcomp)
```

The five **If** statements above can be replaced by accessing the worksheet function **ATAN2( )** :

```
Angle = Application.WorksheetFunction.Atan2(Xcomp, Ycomp)
```
or
```
Angle = Excel.WorksheetFunction.Atan2(Xcomp, Ycomp)
```

```
' Convert the angle from radians to degrees using Factor
Angle = Angle * Factor

' Write the solution into the output cell
Worksheets("Sheet1").Cells(8, 6) = Format(Angle, "###.00")
End Sub

Private Sub cmdClearPolar_Click()
Range(Cells(8, 3), Cells(8, 6)).ClearContents
End Sub

Private Sub cmdDefaultXY_Click()
Worksheets("Sheet1").Range("C8").Value = -60
Worksheets("Sheet1").Range("D8").Value = -80
End Sub
```

Worksheet & VBA "idiosyncrasies" (to be polite) :
For some reason the team that developed **VB** & **VBA** decided to avoid the conventional nomenclature used in **FORTRAN** and generally used for worksheet functions, and instead invent new names for a greatly reduced set of built-in functions !!!
For example : **VBA** uses **Sqr()** instead of **SQRT()** and **Atn()** instead of. **ATAN() .**

Appendix 28 gives the list of built-in VBA math functions -- which is less than impressive.
Also note the different order of operator precedence in VBA (refer to the box on page 1-13), which is more conventional than on the worksheet

If you understood the logic behind the **If** statements on the previous page you can safely skip the following remarks :

If Xcomp = 0 And Ycomp > 0 Then Angle = 1.570796
If Xcomp = 0 And Ycomp < 0 Then Angle = –1.570796

These two **If**s check for Y axis situations where the ratio **Ycomp/Xcomp** would be infinite and would produce an **overflow**.

Angle must be in radians since the built-in trig functions work in radians : 1.57 rads ≡ 90 degrees, 3.14 rads ≡ 180 degrees

If Xcomp < 0 And Ycomp < 0 Then Angle = (3.14159265 + Atn(Ycomp / Xcomp))

In the 3rd quadrant the quotient **Ycomp/Xcomp** is positive, so that the positive first quadrant value of **Atn()** value is positive and may be added to 180 degrees (or 3.14 radians)

For 2nd quadrant vectors Atn() returns 4th quadrant angle values (which are negative). The 2nd quadrant value can be obtained using 3.1415 – Abs(Atn( )) (which is of the form 180^o - θ). Alternatively, since the Atn() value is negative 3.1415 + Atn( ) also works.

If Xcomp < 0 And Ycomp > 0 Then Angle = (3.14159265 – Abs(Atn(Ycomp / Xcomp)))

' Note: we could also omit the **Abs** function and use **(3.14159265 + (Atn(Ycomp / Xcomp)))**

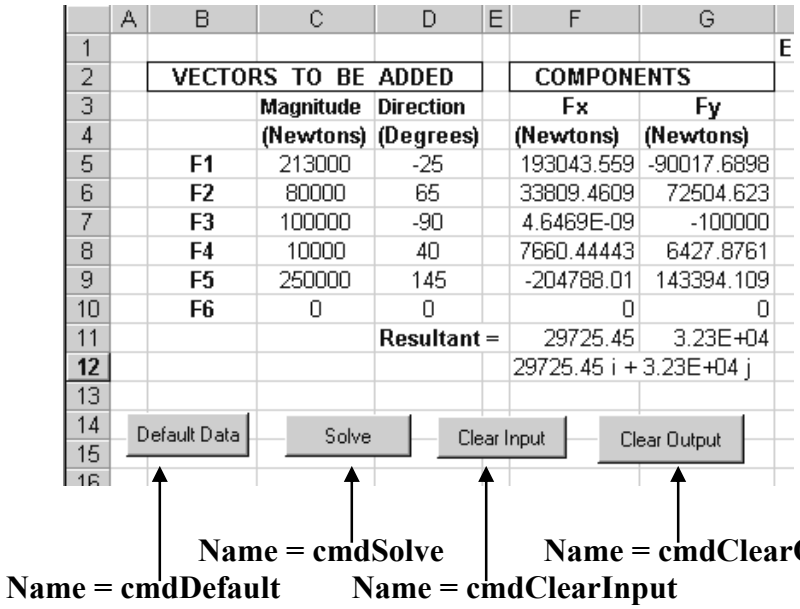
In the 1st and 4th quadrants the angle can be determined directly using **Atn().**

If (Xcomp > 0 And Ycomp > 0) Or (Xcomp > 0 And Ycomp < 0) Then Angle = Atn(Ycomp / Xcomp)

Application # 1a) : Component Addition of Up to 6 Vectors :

This application reads in up to six vectors (not necessarily forces), calculates their components, and then performs the component additions Σ Fx and Σ Fy . A default data button has been included which is good practice if the program will be made available to other users (that being said, in the interests of space and development time, very few of later programs include default buttons). As you will quickly appreciate the program contains an enormous amount of repetitive code which can be avoided by using the loops that are introduced at the end of the chapter.

1. Add the 4 buttons with names and labels as given below :



Add all the **labels** shown. **Do not** add any of the numbers or the i & j expression in cells **F12** and **G12** .

Name = cmdDefault **Name = cmdSolve** **Name = cmdClearInput** **Name = cmdClearOutput**

2. Double click on the **Solve** button and enter the following code .

```
Option Explicit

Private Sub cmdSolve_Click()
Dim F1 As Double, Theta1 As Double, F2 As Double, Theta2 As Double
Dim F3 As Double, Theta3 As Double, F4 As Double, Theta4 As Double
Dim F5 As Double, Theta5 As Double, F6 As Double, Theta6 As Double
Dim Rx As Double, Ry As Double
Dim F1x As Double, F1y As Double, F2x As Double, F2y As Double
Dim F3x As Double, F3y As Double, F4x As Double, F4y As Double
Dim F5x As Double, F5y As Double, F6x As Double, F6y As Double

' Factor converts degrees to radians
Const Factor As Double = 3.1415926535897 / 180

' Read in the magnitude and angle from the worksheet
F1 = Range("C5")
Theta1 = Cells(5, 4) * Factor
```

Note the two different ways of referencing cell addresses.

```
F2 = Cells(6, 3)
Theta2 = Range("D6") * Factor

F3 = Range("C7")
Theta3 = Range("D7") * Factor

F4 = Range("C8")
Theta4 = Range("D8") * Factor

F5 = Range("C9")
Theta5 = Range("D9") * Factor

F6 = Range("C10")
Theta6 = Range("D10") * Factor
```

Remember that it's prudent to add **Worksheets("Sheet1").** before each cell location.

You may want to locate the X and Y components on a single line separated by a colon to make the code more compact.

```
' Calculate the components
F1x = F1 * Cos(Theta1):  F1y = F1 * Sin(Theta1)

F2x = F2 * Cos(Theta2):  F2y = F2 * Sin(Theta2)

F3x = F3 * Cos(Theta3):  F3y = F3 * Sin(Theta3)

F4x = F4 * Cos(Theta4):  F4y = F4 * Sin(Theta4)

F5x = F5 * Cos(Theta5):  F5y = F5 * Sin(Theta5)

F6x = F6 * Cos(Theta6):  F6y = F6 * Sin(Theta6)

' Return the components to the worksheet
Worksheets("Sheet1"). Range("F5") = F1x:  Worksheets("Sheet1"). Range("G5") = F1y

Worksheets("Sheet1"). Range("F6") = F2x:  Worksheets("Sheet1"). Range("G6") = F2y

Worksheets("Sheet1"). Range("F7") = F3x:  Worksheets("Sheet1"). Range("G7") = F3y

Worksheets("Sheet1"). Range("F8") = F4x:  Worksheets("Sheet1"). Range("G8") = F4y

Worksheets("Sheet1"). Range("F9") = F5x:  Worksheets("Sheet1"). Range("G9") = F5y

Worksheets("Sheet1"). Range("F10") = F6x:  Worksheets("Sheet1"). Range("G10") = F6y

' Calculate the resultant of the known vectors F1 through F6

Rx = F1x + F2x + F3x + F4x + F5x + F6x
Ry = F1y + F2y + F3y + F4y + F5y + F6y

Range("F11") = Format(Rx, "0.00")
Range("G11") = Format(Ry, "0.0E+00")
Range("F12") = Format(Rx, "0.00") & " i" & " + " & Format(Ry, "Scientific") & " j"

End Sub
```

Two ways of obtaining scientific notation. The "**E**" format allows the number of decimal places to be specified.

3. Double click on the **Default Data** button and add the following code :

Default data is generally provided to avoid a user selecting physically inappropriate data, and to build confidence in the program; all that is required of the user is to press buttons. **Caution** : Unless specifically requested, providing default data should be avoided in quizzes. The objective of a quiz is to assess your intellectual analysis of some situation; adding a default button is non-essential and wastes time.

```
Private Sub cmdDefault_Click()

' Set default values (Airplane problem in class notes)
Range("C5") = 213000 : Range("C6")  = 80000
Range("D5") = -25 : Range("D6")  = 65

Range("C7") = 100000 : Range("C8") = 10000
Range("D7") = -90 : Range("D8") = 40

Range("C9") = 250000 : Range("C10") = 0
Range("D9") = 145 : Range("D10") = 0

    End Sub
```

4. Double click on the **Clear Input** button and enter the code below :

```
Private Sub cmdClearInput_Click()
Range("C5:D10").Clear
Range("C4").Select
Selection.ClearContents
End Sub
```

Clear eliminates everything related to the cell : its contents as well as any formatting such as **bold**, *italics*, centered, etc. **ClearContents** only removes the contents of the cells.

5. Double click on the **Clear Output** button and enter the code below :

```
Private Sub cmdClearOutput_Click()
Range("F5:G11").Select
Selection.Clear
Range("F4:G4").ClearContents
Range("F12").Select
Selection.ClearContents
End Sub
```

Test your program. The output that you should obtain is shown on the right. Any bugs that you may encounter will almost certainly be due to typing mistakes.

	A	B	C	D	E	F	G
1							
2		VECTORS TO BE ADDED				COMPONENTS	
3			Magnitude	Direction		Fx	Fy
4			(Newtons)	(Degrees)		(Newtons)	(Newtons)
5		F1	213000	-25		193043.559	-90017.6898
6		F2	80000	65		33809.4609	72504.623
7		F3	100000	-90		4.6469E-09	-100000
8		F4	10000	40		7660.44443	6427.8761
9		F5	250000	145		-204788.01	143394.109
10		F6	0	0		0	0
11				Resultant =		29725.45	3.23E+04
12						29725.45 i + 3.23E+04 j	

2.3 Using Loops to Reduce the Volume of Code

Each of the six vectors in the previous application required separate code statements to read it in, take its components, and then write out the components. Repetitive operations such as these can be simplified by using **loops** and **arrays** (the discussion of arrays will be deferred until Chapter 4). A **loop** is a portion of code that is repeated a number of times. Do-it-yourself loops can be created using an **If** statement, a **GoTo** statement and a loop **Index**. Chapter 3 introduces two built-in loops that come with **VB**.

In order to construct a loop we'll need to be able to name a code line and use a **GoTo** statement. Any line of code can be named by entering a name followed by a colon. The name can be constructed out of any combination of keyboard characters (except reserved characters) but, obviously, cannot be the same as any variable name that's being used. Regular code statements may be added to the right of the colon.

A blank line of code has been named **Work**; the other two lines have been named **Home** and **22**.

```
Work:
Home:  y = 2*x^2 - 3.2*x + 4.1
22:  Worksheets("Sheet1").Range("C4") = Magnitude*cos(Angle)
```

A **GoTo** statement directs the program to go to a named code line, and can be used to bypass portions of code (usually if certain conditions are met), or to create loops as will be demonstrated here.

I must be initialized to some convenient starting value.

```
Dim I As Integer
I = 0

Beginning:
I = I + 1
If I <= 10 Then GoTo Beginning
```

Bizarre Algebra : the statement **I = I + 1** is meaningless from the point of view of normal algebra. The equal sign in computing is referred to as an assignment statement since the value of the right hand side is calculated and assigned to the storage location indicated on the left hand side. In this case the right hand side is evaluated by reading the value of **I** from the storage location named **I**, and adding **1** (one) to that value. The new value is of **I** is then stored in the location indicated on the left, which means that it is stored back in location **I**.

Program Execution : The program begins by setting the value of **I** to **0**, and then execution moves through the line named **Beginning** (which does nothing). The statement **I = I + 1** adds **1** to the current value of **I** (which is **0**) so that the new value is **1**. The **If** statement contains a condition to detect if the current value of **I** is less than or equal to **10**. [Remember, all **If** conditions are evaluated as **True** or **False.**] In this case the condition is **True** (since **1** is **<= 10**) and program execution loops back to the blank line named **Beginning**; then **1** is added to the current value of **I** (which is now **1**) to produce a new value of **2**. The **If** condition again evaluates to **True**, and execution is again looped back to **Beginning** , and so on. The program jumps from the **If** back to **Beginning** ten times at which point the value of **I** will then reaches **11**. When this happens the **If** condition evaluates to **False** since **11 <= 10** is **not True** and program execution proceeds to whatever code is placed after the **If** statement. [Note that although program execution jumps from the **If** back to **Beginning** ten times, the code between **Beginning** and the **If** is executed one last time as **I** becomes **11**; we thus might say that it's looped 10 times but passed through the code in the loop 11 times. What would happen if **I** were initialized to 1 instead of 0 ?] The code could have been compacted by combining the **I = I + 1** statement with the line labeled **Beginning** to give the single statement **Beginning: I = I + 1**

The variable **I** is referred to as the **loop index** since it controls the number of loops (or iterations) that are made. In this example it could also be described as a **dummy index** since it has no physical significance (it's not a time, or a location, or a force, or a number of dollars, etc); it simply keeps track of the number of loops made.

Moving Read and Moving Write Statements

The read and write statements encountered so far have been static, meaning that they work with cells at specific and fixed locations. However, the repetitive operations in a loop frequently require different data to be read-in for each iteration and the results of calculations written back onto the worksheet. The input data is usually arranged in columns so that the read statements need to move down the columns and read values from a different row each time. Similarly, the output values often need to be written down another column such that the values for a particular iteration are written in the same row as the input data. [Data is occasionally read-in and written across rows but is limited by a maximum of 256 columns.] The varying locations for reading and writing in loops makes it logical to refer to them as ***moving* read** and ***moving* write** statements. From time to time the output of loops may even be written into random cells (see **the Random Colour Generator** Exercise, and **Random Walk** Exercise) or cell locations that spiral inward (******Future*** *-- add* ***Spiral*** *Exercise*).

Moving read and write statements are most easily constructed using the **Cells** property and a **Row** index which is used to control the row from which the data is being read (or into which it's being written). The value of the **Row** index is increased by **1** with each iteration using the statement **Row = Row + 1** to cause the read and write locations to move down the column of data. For example :

MOVING READ **X = Worksheets("sheet1").Cells(Row, 2)** reads a value of **X** from different rows of column **2** with each new value of index **Row**, while

MOVING WRITE **Worksheets("sheet1"). Cells(Row, 4) = Z** writes a value of **Z** into different rows of column **4** with each new value of index **Row**.

Alternative :
Moving read & write statements can also be constructed using the **Range** property :

X = Worksheets("sheet1").Range("B" & Row)

The statement **Worksheets("sheet1").Range("A2: B" & N).ClearContents** could be used to clear a variable set of N pairs of data from two columns (where N was read in off the worksheet).

Example #3 : Moving Write / Simple Loop Using If Statement. The loop index used in the code fragment given on the previous page will be written into successive cells of column **3** starting at row **4** .

```
Option Explicit
Private Sub cmdLoop_Click()
Dim I As Integer, Row As Integer
I = 0
Row = 4

Beginning: I = I + 1
   Worksheets("Sheet1").Cells(Row, 3) = I
   Row = Row + 1
If I <= 10 Then GoTo Beginning

End Sub
```

*** **Row** must be **Dim**-med as **Integer** since cell locations only come in integer values.

Row is initialized to whichever row will receive the first value; the 4th row in this example.

MOVING WRITE STATEMENT

Index **Row** is incremented by **1** so that the next cell to be written into is one row down.

```
Private Sub cmdClear_Click()
Range(Cells(4, 3), Cells(14, 3)).ClearContents
End Sub
```

Although **Range("C4: C14").ClearContents** is a simpler alternative, the **Cells** approach will be useful with variable length data sets.

Example # 4: Moving Read & Write Statements. This program reads values of X and Y from columns **B** and **C** on the worksheet, then squares and adds them to produce Z, and finally writes the value of Z back onto the worksheet.

```
Private Sub cmdCalculate_Click()
Dim X As Double, Y As Double, Z As Double
Dim I As Integer, Row As Integer

Row = 4
I = 1

Beginning:
   X = Worksheets("sheet1").Cells(Row, 2)
   Y = Worksheets("sheet1").Cells(Row, 3)
   Z = X ^ 2 + Y ^ 2
   Worksheets("sheet1").Cells(Row, 4) = Z

   I = I + 1
   Row = Row + 1
If I <= 5 Then GoTo Beginning

End Sub
```

	A	B	C	D
1				
2		X	Y	Z
3				(= X^2 + Y^2)
4		1	2.2	5.84
5		2	3.1	13.61
6		3	4.6	30.16
7		4	4.7	38.09
8		5	5.1	51.01

The initial row to be read from & written into must be specified.

MOVING READ STATEMENTS

MOVING WRITE STATEMENT

The **Row** index is incremented by **1** to move the read & write statements down the columns to the next row.

Example # 5 : Adding Colour to a Cell Programmatically. The previous exercise can be enhanced by adding colour to the cells, and by bolding and centering the values of **I**. Unfortunately a bug in the **VBA** program that I've been using (**Excel 97**) requires that two **Select** commands, which don't really do anything, be included. This bug appears to have been corrected in some of the **Windows 2000** versions making the **Select** commands unnecessary. Make a copy of **Example #1** and add the indicated code. New code statements are presented in **bold**; existing code statements are *italicized*.

```
Option Explicit

Private Sub cmdLoop_Click()
Dim I As Integer, Row As Integer
I = 0
Row = 4
Cells(1, 1).Select

Beginning: I = I + 1
    Worksheets("Sheet1").Cells(Row, 3) = I
    Cells(Row, 3).Interior.ColorIndex = 3
    Cells(Row, 3).Font.Bold = True
    Cells(Row, 3).HorizontalAlignment = xlCenter
    Row = Row + 1
If I <= 10 Then GoTo Beginning

End Sub

Private Sub cmdClear_Click()
Range(Cells(4, 3), Cells(14, 3)).Clear
Range("A2").Select
Worksheets("sheet1").Range("C4:C14").Interior.ColorIndex = xlNone
End Sub
```

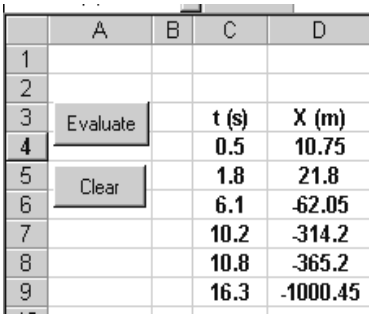
Excel 97 VBA Bug : **Cells(1, 1).Select** must be included to allow the **.Interior.ColorIndex** , **.Font.Bold** , and **.HorizontalAlignment** statements to execute without producing an error message. Try removing it by means of a single quote and then run the program. Any cell could have been selected.

The **VBA** constants **xlCenter** and **xlNone** (found below in *cmdClear*) contain an "**x**" followed by a lower case "**L**" -- **xl** is an abbreviation of **Excel**.

The **.Clear** method removes the bold and centering effects without having to resort to using **.Font.Bold** , and **.HorizontalAlignment** commands. **Range("A2").Select** must be present in order that the line containing **.Interior.ColorIndex** does not produce an error message (**Excel 97**).

The code for centering, bolding, and changing the interior colour of a cell was obtained by using the **Macro Recorder**. Refer to the **Appendix 19 : Legal Plagiarism Using the macro Recorder.**.

Example # 6 : Evaluating an X(t) Position Function.
The following program uses a **moving read statement** to read–in different values of time which are then used in a position-time function to calculate the location of an object. The positions are written back onto the worksheet using a **moving write statement**. The object is initially at $x_o = 2$ meters, travelling at $V_{ox} = 20$ m/s, and is experiencing an acceleration of -10 m/s^2 . Recall that the general position-time function has the form :
$x(t) = x_o + V_{ox}\, t + (1/2)\, a_x\, t^2$.




```
Option Explicit

Private Sub cmdEvaluate_Click()
Dim I As Integer, Row As Integer, t As Double, x As Double
I = 0

Row = 4

Beginning:  I = I + 1
   t = Worksheets("sheet1").Cells(Row, 3)
    x = 2 + 20 * t - 5 * t ^ 2
    Worksheets("sheet1").Cells(Row, 4) = x
    Row = Row + 1
If I < 6 Then GoTo Beginning

End Sub

Private Sub cmdClear_Click()
Worksheets("sheet1").Range("D4:D9").ClearContents
End Sub
```

Moving Read statement

Notice that the moving read & write statements must be inside the loop.

Moving Write statement

I = I + 1 can be located anywhere inside the loop. **Row = Row + 1 n**eeds to be after the write statement if we wish to start writing in row **4**.

Alternatively, the variable **x** might have been omitted altogether and the following single statement used instead of the two statements given above :

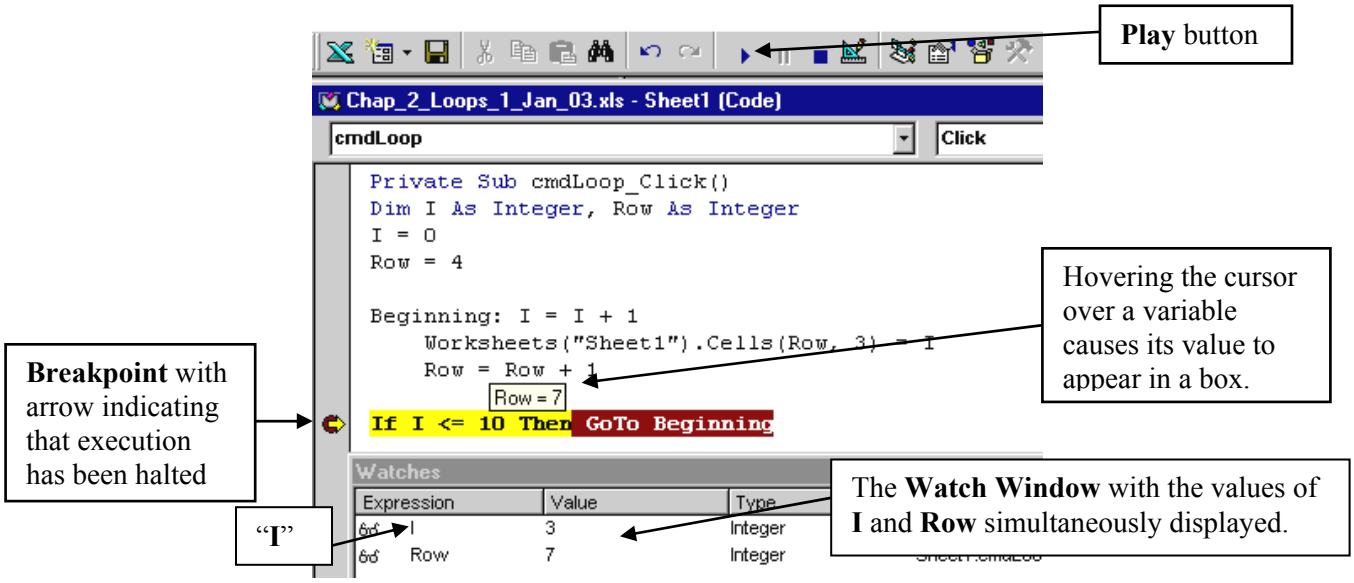
```
Worksheets("sheet1").Cells(Row, 4) = 2 + 20 * t - 5 * t ^ 2
```

Programming Mistakes : Rookie programmers often misunderstand the importance of **correctly locating** code statements, and creating the **proper sequence** of statements. A typical mistake by a novice writing this program would be to place the read statement before the loop is entered. No error would occur, but the program would use the first value of time over and over (the **Row** index is only changed inside the loop). A common sequencing error would be to locate the function evaluation statement before the read statement which would mean that no **t** value would be available during the first iteration (a value of **0** would be assumed), and thereafter, the **t** value would always be one loop behind.

Both the read and write statements must be inside the loop with the read placed before the calculation statement (so that there is a current value of time to work with) and the write located after the calculation (so that **x** reflects the latest calculated value). If the **I = 0** initialization is inadvertently included inside the loop the value of **I** will never exceed **1** so that the loop cannot be exited; an infinite loop occurs which appears like a screen freeze (use **Ctrl + Break** to interrupt an infinite loop). If the **Row = 4** initialization is placed inside the loop, the write statement will always write into the same cell. What happens if **Row = Row + 1** is moved to before the write statement ?

Break Points and the Watch Window

Program execution can be followed by adding **breakpoints** and displaying variable values in the **Watch Window**. A breakpoint is set by clicking in the gray border on the left side of the **Code Window** which causes a coloured circle to appear (certain lines may not allow a breakpoint to be set). When the program is run, execution will be halted at the breakpoint. To resume execution press on the VCR-type **Play** button. Breakpoints are removed by clicking on the dot when the program has been reset. When program execution is stopped at a breakpoint variable values can be observed by hovering the cursor over a variable; its value will then appear in a small box. Any number of breakpoints can be set.



Alternatively, multiple variable values can be observed at the same time by writing their value in the **Watch Window**. The **Watch Window** is opened from the **View Menu** (in the **VB Editor**) by selecting **Watch Window**. Highlight the variable you wish to "watch", hold the cursor over the highlighted area until it becomes an arrow, then click and drag the variable into the **Watch Window** and release.

Programs are run from the **VB Editor** by clicking the cursor anywhere inside the subroutine, and then pressing the VCR-type **Play** button. The program will stop at the breakpoint (an arrow will appear inside the breakpoint). Program execution can be resumed by clicking on the **Play** button; execution will proceed until the next breakpoint is encountered. If only one breakpoint is set, a loop is made each time the **Play** button is pressed. If more than one breakpoint is present the program will move to the next breakpoint each time the **Play** button is pressed. To re-start the program, press the **Reset** button followed by the **Play** button.

Application # 1 b) : Component Addition of Up to 6 Vectors (using Loops)

A significant amount of repetitious typing was required in **Application # 1a)** as individual read, calculation, and write statements were required for each of the six vectors; imagine if 12 vectors were to be added. A far more efficient approach is to use a **loop** to handle the repetitive operations and a **running total** to keep track of the resultant sum. Save **Application # 1a)** under a new name and modify the **cmdSolve_Click()** subroutine so that it contains the code given below. New code is indicated by **bolded** statements, existing code is shown in *italics*. Approximately 43 lines of old code must be deleted and replaced by 16 lines of new code. The "homemade" loop used here can be replaced by one of the built-in loops that are available in **VBA** and discussed in Chapter 3; the necessary modifications are indicated with asterisks ***.

```
Private Sub cmdSolve_Click()
Dim Magnitude As Double, Angle As Double, Xcomp As Double, Ycomp As Double
Dim I As Integer, Row As Integer
Dim Rx As Double, Ry As Double
' Factor converts degrees to radians
Const Factor As Double = 3.1415926535897 / 180

Rx = 0: Ry = 0
Row = 5                                   <- Row location of the first vector to be read-in.
I = 0                                     <- ***Delete I = 0 if built-in For loop is used (see Chap 3).

Beginning:  I = I + 1                     <- ***Replace by For I = 1 To 6 to use the built-in For loop.

    ' Read in polar form of vector
    Magnitude = Cells(Row, 3)
    Angle = Cells(Row, 4) * Factor

    ' Calculate components and add to existing (running) total
    Xcomp = Magnitude * Cos(Angle)
    Ycomp = Magnitude * Sin(Angle)
    Rx = Rx + Xcomp
    Ry = Ry + Ycomp                       <- Running totals are used to hold the cumulative value of the component

    ' Write components back on spreadsheet
    Cells(Row, 6) = Xcomp  ' write in columns F & G
    Cells(Row, 7) = Ycomp

    ' Increase the row index to reference the next vector in the list
    Row = Row + 1
If I < 6 Then GoTo Beginning              <- ***Replace by Next I to use built-in For loop.

Range("F11") = Format(Rx, "0.00")
Range("G11") = Format(Ry, "00.0E+00")
Range("F12") = Format(Rx, "0.00") & " i" & " + " &  Format(Ry, "Scientific") & " j"

End Sub
```

Two running total statements : **Rx = Rx + Xcomp** and **Ry = Ry + Ycomp** have been used to keep track of the cumulative sum of **Rx** and **Ry** . As each successive vector is read-in, its components **Xcomp** and **Ycomp** are determined and added to the previous total. Recall that from a computing point of view an equal sign is an assignment statement that takes the value on the right side and stores it in the storage location identified on the left side. The latest value of **Xcomp** is added to the existing total of the previous X components currently stored in location **Rx** . The result of this updating is then returned back into the storage location for **Rx** .

The efficiency of this technique becomes apparent when it is realized that the single modification necessary to allow an arbitrary number of vectors **N** to be added is to change the **If** statement :

If I < N Then GoTo Beginning

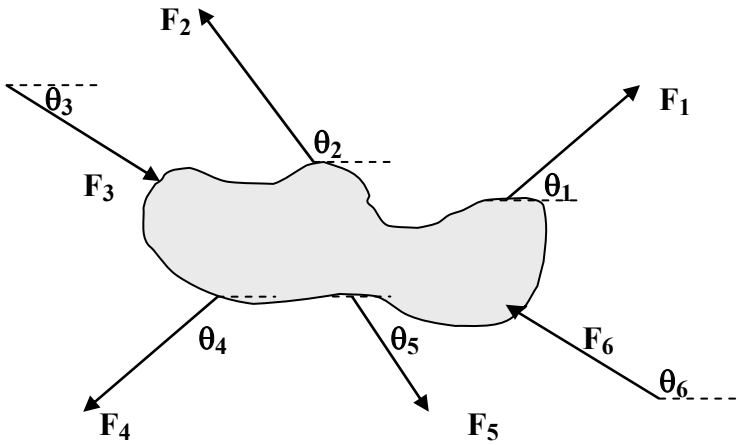
and, of course, to specify the value of **N**.

2.4 The 1st Condition of Equilibrium (Translational Equilibrium) (Newton's 1st & 2nd Laws Describe Change)

Application # 1 constructed the sum of up to 6 vectors and was simply a problem in vector mathematics. The branch of physics referred to as mechanics concerns the study of forces and their resultant impact on the motion of the objects affected. Newton's Laws provide the framework to analyze the mechanics of physical systems. According to Newton's 1st Law there will be no change in the motion of a system if the net force acting on it is zero. It will continue doing whatever it was doing : if it was at rest, it will remain at rest; if it was in motion, it will continue to move in the same direction with the same speed. [The 1st Law might thus be referred to as the Law of the Status Quo.] Systems for which there is no change are referred to as being in a state of **equilibrium**[2]. A system at rest is described as being in **static equilibrium**; if the system is moving there is no special term, however, a useful description is **"kinetic" equilibrium**. The 2nd Law indicates that if there is a net force present then a change in motion will occur which depends on both the net force and the inherent resistance that the system displays to change (or its **inertia**, which is measured by its mass). The change in motion that results is a rate of change of velocity as described by the acceleration. For a system of given mass, the greater the net force, the greater the subsequent rate of change of velocity.

The first step of any analysis is to decide if change is occurring and then to apply either the 1st or 2nd Law. In this section we'll be considering exclusively 1st Law situations. The mathematical description of the 1st Law situation is simple: the sum of the forces is zero so that $\Sigma \mathbf{F} = \mathbf{0}$. Since vectors are added analytically via the component method, the force components must satisfy the X and Y equations : $\Sigma F_x = 0$ and $\Sigma F_y = 0$. With these two equations it follows that we can have at most two unknowns. In the application that follows the unknowns will be restricted to two unknown magnitudes [3].

A system consisting of 6 concurrent forces can be imagined in general as :



---

[2] There are actually two types of equilibrium : Translational equilibrium in which there is no change in the orientation of a system, and rotational equilibrium.

[3] The other possible sets of unknowns are 1 magnitude and 1 angle of different vectors, 1 magnitude and 1 angle of the same vector (or both components unknown), and 2 unknown angles.

Applying the equilibrium equations to this problem and assuming that the unknown magnitudes are F_1 and F_2 gives :

$$\Sigma F_x = F_1 \cos\boldsymbol{\theta_1} + F_2 \cos\boldsymbol{\theta_2} + \overbrace{F_3 \cos\boldsymbol{\theta_3} + F_4 \cos\boldsymbol{\theta_4} + F_5 \cos\boldsymbol{\theta_5} + F_6 \cos\boldsymbol{\theta_6}}^{R_x^*} = 0$$

$$\Sigma F_y = F_1 \sin\boldsymbol{\theta_1} + F_2 \sin\boldsymbol{\theta_2} + \underbrace{F_3 \sin\boldsymbol{\theta_3} + F_4 \sin\boldsymbol{\theta_4} + F_5 \sin\boldsymbol{\theta_5} + F_6 \sin\boldsymbol{\theta_6}}_{R_y^*} = 0$$

The components of vectors $\mathbf{F_3}$ through $\mathbf{F_6}$ are all known quantities and will be replaced by their resultant R_x^* and R_y^* .

The simultaneous equations in the two unknowns F_1 and F_2 become :

$$F_1 \cos\boldsymbol{\theta_1} + F_2 \cos\boldsymbol{\theta_2} = -R_x^* \qquad \ldots (\mathbf{1})$$

$$F_1 \sin\boldsymbol{\theta_1} + F_2 \sin\boldsymbol{\theta_2} = -R_y^* \qquad \ldots (\mathbf{2})$$

where the cosines, sines, R_x^*, and R_y^* are all known. The values obtained from the data of the upcoming application produce the following equations :

$$0.342\, F_1 + 0.643\, F_2 = 137.5$$

$$0.940\, F_1 + 0.766\, F_2 = 311.8$$

Using the substitutions : $b_1 = -R_x^*$, $b_2 = -R_y^*$, $a_{11} = \text{Cos}\theta_1$, $a_{12} = \text{Cos}\theta_2$, $a_{21} = \text{Sin}\theta_1$, $a_{22} = \text{Sin}\theta_2$ reduces equations (1) and (2) to the form :

$$a_{11} F_1 + a_{12} F_2 = b_1 \qquad \ldots (\mathbf{3})$$

$$a_{21} F_1 + a_{22} F_2 = b_2 \qquad \ldots (\mathbf{4})$$

The simultaneous solution can be shown to be :

$$F_2 = (b_2 - (a_{21} / a_{11})* b_1)/(a_{22} - (a_{21} / a_{11}) * a_{12}) \qquad \ldots(\mathbf{5})$$

$$F_1 = (b_1 - a_{12} * F_2) / a_{11} \qquad \ldots(\mathbf{6})$$

and it is this solution that will form the basis of our programmed solution.

Equations (3) & (4) are simultaneous linear equations in F1 and F2 which could be solved graphically, albeit with more work and less precision. Nevertheless, a computer analysis provides a good opportunity for emphasizing the underlying graphical interpretation and demonstrating how certain situations may be vulnerable to rounding error if the two equations intersect as small angles.

Although a variety of more elegant methods might be used to plot equations (3) & (4), we'll use the direct approach of determining two points on each line and then manually adding a trendline using the **Chart Wizard**. Since certain equilibrium situations (such as simple hanging weight problems) result in one of the equations passing through the origin the use of axis intercepts will be avoided. Instead, the points to be used correspond to the values of F_2 found by substituting $0.25\ F_1^*$ and $1.75\ F_1^*$ into the two equations where F_1^* is the simultaneous solution for F_1. Solving X equation (3) for F2 assuming that the value of F1 is given produces :

$$F_2 = (b_1 - a_{11} F_1)/ a_{12} \quad \ldots(7)$$

and similarly for Y equation (4) :

$$F_2 = (b_2 - a_{21} F_1)/ a_{22} \quad \ldots(8)$$

The following code computes the two points to be plotted for the X and Y equations :

```
' X equation solution points
Worksheets("Sheet1").Range("H4") = 0.25 * F1
Worksheets("Sheet1").Range("I4") = (b1 - a11 * (0.25 * F1)) / a12

Worksheets("Sheet1").Range("H5") = 1.75 * F1
Worksheets("Sheet1").Range("I5") = (b1 - a11 * (1.75 * F1)) / a12
```

Value of F1.

Corresponding value of F2 from eqn (**7**)

Ditto.

The brackets around **0.25 * F1** and **1.75 * F1** are unnecessary ; they've been included to emphasize that the equations are being evaluated at these two values of **F1** and **F2** .

```
' Y equation solution points
Worksheets("Sheet1").Range("H6") = 0.25 * F1
Worksheets("Sheet1").Range("J6") = (b2 - a21 * (0.25 * F1)) / a22

Worksheets("Sheet1").Range("H7") = 1.75 * F1
Worksheets("Sheet1").Range("J7") = (b2 - a21 * (1.75 * F1)) / a22
```

Value of F1.

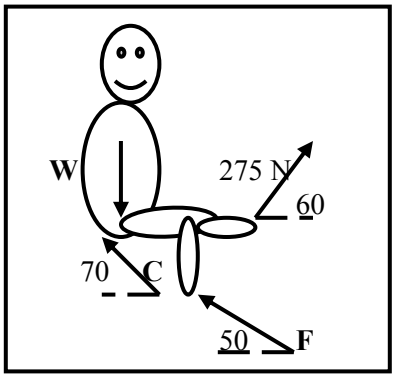
Corresponding value of F2 from eqn (**8**).

Ditto.

Application #2 : Equilibrium Problems With Two Unknown Magnitudes

The application developed here gives the solution to a general translational equilibrium problem with up to six forces present for which two of the forces have unknown magnitudes. [The program is easily extended to N forces with two unknown magnitudes using the modification given at the end of Application #1b).] Rotational effects are neglected as the vectors are assumed to be concurrent. The X and Y equilibrium equations are plotted on a single graph so that the simultaneous solution is evident.

The problem situation is taken from pg 2-12 of your mechanics class notes (if you were in my section). A woman (weight W = 550 Newtons) with a broken leg is sitting on a chair that exerts a force **C** on her "rear" acting at 70^o . Her cast is supported by a pulley mechanism (not shown) that creates a tension of 275 Newtons at 60^o on her cast. Her other leg rests on the floor which causes a contact force **F** acting at 50^o on the bottom of her foot. The objective is to find the unknown magnitudes of **F** and **C**.



The contents of cells **C13**, **C14**, **(G4: H5)**, and **(G8: H9)** are produced by the program. The code in *italics* and not bolded was taken directly from **Application # 1b)** :

	G	H
Plot Points (F2 vs F1)		
X equation points		
	16.48	371.04
	115.39	185.17
Y equation points		
	16.48	318.42
	115.39	237.79

Angles must be measured from the positive X axis.

	B	C	D
		Magnitude	Angle
		(Newtons)	(Degrees)
	F1	F	130
	F2	C	110
	F3	550	-90
	F4	275	60
	F5	0	0
	F6	0	0

Solve

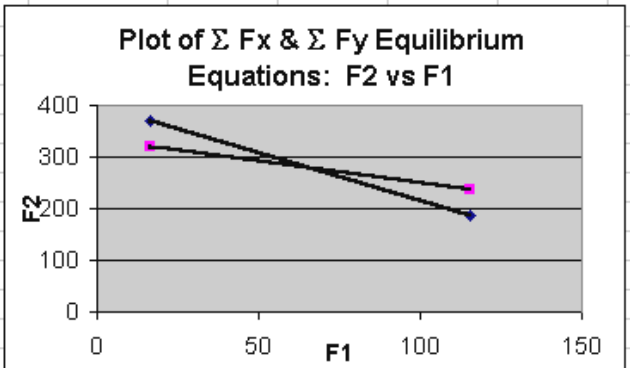
Clear Output

Simultaneous Solution

F = 65.94

C = 278.11

The vectors with unknown magnitudes must be entered first in rows 5 & 6. Enter the unknown magnitude names in C5 & C6



Option Explicit

Private Sub cmdClear_Click()
Worksheets("Sheet1").Range("G4:H5").ClearContents
Worksheets("Sheet1").Range("G8:H9").ClearContents
Worksheets("Sheet1").Range("D13:D14").ClearContents
End Sub

```
Private Sub cmdSolve_Click()
Dim Magnitude As Double, Angle As Double, Xcomp As Double, Ycomp As Double
Dim I As Integer, Row As Integer
Dim Rx As Double, Ry As Double
Dim Theta1 As Double, Theta2 As Double, F1 As Double, F2 As Double
Dim b1 As Double, b2 As Double
Dim a11 As Double, a12 As Double, a21 As Double, a22 As Double
Const Factor As Double = 3.1415926535897 / 180

Rx = 0: Ry = 0
Row = 7
I = 0

'  Read in the angles of the 2 vectors with unknown magnitudes
Theta1 = Worksheets("Sheet1").Range("D5") * Factor
Theta2 = Worksheets("Sheet1").Range("D6") * Factor

Beginning:   I = I + 1

Magnitude = Cells(Row, 3)
Angle = Cells(Row, 4) * Factor

Xcomp = Magnitude * Cos(Angle)
Ycomp = Magnitude * Sin(Angle)
Rx = Rx + Xcomp
Ry = Ry + Ycomp

Row = Row + 1
If I < 6 Then GoTo Beginning

' Define the coefficients and constants of the two equations to be solved simultaneously.
b1 = -Rx
b2 = -Ry
a11 = Cos(Theta1)
a12 = Cos(Theta2)
a21 = Sin(Theta1)
a22 = Sin(Theta2)

' Calculate the simultaneous solution.
F2 = (b2 - (a21 / a11) * b1) / (a22 - (a21 / a11) * a12)
F1 = (b1 - a12 * F2) / a11

Worksheets("Sheet1").Range("D13") = F1
Worksheets("Sheet1").Range("D14") = F2
Worksheets("Sheet1").Range("C13") = Worksheets("Sheet1").Range("C5") & "  ="
Worksheets("Sheet1").Range("C14") = Worksheets("Sheet1").Range("C6") & "  ="
```

Equations (**5**) & (**6**) produce the simultaneous solution for F1 & F2.

The last 2 write statements are “fluff” that put **F =** and **C =** in cells **C13** and **C14**.

```
' Plot two points on each of the X and Y equations
' X equation solution points
Worksheets("Sheet1").Range("G4") = 0.25 * F1
Worksheets("Sheet1").Range("H4") = (b1 - a11 * (0.25 * F1)) / a12

Worksheets("Sheet1").Range("G5") = 1.75 * F1
Worksheets("Sheet1").Range("H5") = (b1 - a11 * (1.75 * F1)) / a12

' Y equation solution points
Worksheets("Sheet1").Range("G8") = 0.25 * F1
Worksheets("Sheet1").Range("H8") = (b2 - a21 * (0.25 * F1)) / a22

Worksheets("Sheet1").Range("G9") = 1.75 * F1
Worksheets("Sheet1").Range("H9") = (b2 - a21 * (1.75 * F1)) / a22

End Sub
```

Use the **Chart Wizard** to create 2 series of data each consisting of 2 points [cells (**G4:H5**) & (**G8:H9**) -- don't forget to specify the data as in columns]. Add a linear trendline to each series (do not display the equation). You will only have to fit the trendline once even though the graph becomes empty when the **Clear** button is activated.

Rounding Errors Simultaneous equations are vulnerable to rounding errors when they do not intersect orthogonally (at right angles). In the sample graph given the equations are skewed at quite a small angle to one another. This means that region of intersection is not precisely defined, which in turn means that there is a relatively large region of values of F1 and F2 which tend to satisfy both equations. Under these conditions, even the smallest rounding errors in the computations will move the solution away from the true solution. Methods are available for expressing each equation in terms of an orthogonal set of basis vectors to minimize the effects of rounding errors (but this is something that you may only encounter in your last year of university).

Problems [Chapter 2]

1. **Conversion of chapter 1 problems**. All the problems of chapter 1 can be converted (using a simple **If…Then GoTo** loop and **moving read and write statements**) to read-in multiple sets of input data. Along with introducing a simple loop, it would be useful to eliminate both **Input Box** and the **Message Box** as these become time consuming and are made redundant by the read and write statements.. Create about 8 different sets of input data.

2. Modify problem # 3 of chapter # **1** by introducing an **If…Then…GoTo** loop and moving write statement. However, rather than reading in the values of **r**, make **r** the loop variable and increment it by 1000 km (10^6 meters) every iteration. Use a starting value of **r** of 10^6 meters and stop calculating when **r** is greater 30 x 10^6 meters. Write the values of **r** and the gravitational field onto the worksheet and plot the data using the **Chart Wizard**.

3. Modify ***Example # 3*** pg 2-16 so that the values are written across a single row. The first value of **I** printed out should be 6 and the last 38. Increment **I** by 2 each loop. [The width of the columns can be reduced by highlighting the column letters across the top of the worksheet, right-clicking, and setting to column width to around 3.5.] Add a **Clear** button to clear the values. Once the program is working use index **I** to set the colour of each cell (refer to ***Example # 4***, chapter **2, pg 2-17** -- use the same approach to delete the colour in the **Clear** button). An error will occur if the colour index exceeds 55.

4. Construct a loop that writes out down a single column an index which starts at a value read-in from the worksheet. The index, which is a floating point (decimal value) number, should be increased by an amount that is also a floating point value and is read-in off the worksheet . The number of values to be produced should be entered via an **Input Box**. A running total (example pg 2-20, or any index increased in a loop) of the values produced should be printed out beside each number. Use the **Row** index involved in the moving write statements as the colour index (***Example # 4***, pg 2-17) of each cell where the running total is written. Note : colour index values must be integer and should not exceed 55.

5. The objective of this problem is to calculate values of x(t) and $v_x(t)$ for two moving vehicles. The range of values of time should be between 0 and 10 seconds in steps of 0.4 s. Use **t** as the loop index, and increment it by 0.4 instead of 1 : **t = t + 0.4**

   Read the following motion parameters off the worksheet.

   Car A : X_{oA} = 100 meters, V_{oA} = 40 m/s, a_A = −1.4 m/s^2

   Car B : X_{oB} = 300 meters, V_{oB} = −30 m/s, a_B = −2.0 m/s^2

   The general position-time and velocity-time functions for constant acceleration are :

   $$x(t) = x_o + V_{ox}\, t + 0.5\, a_x\, t^2 \qquad \text{and} \qquad v_x(t) = V_{ox} + a_x\, t$$

   Use the **Chart Wizard** to plot the data with both sets of positions on the same graph, and both sets of velocities on the same graph. [Clearly, the values of **t** also need to be written onto the worksheet.]

Save your problem under a different name (so that you're protected with a back-up version before making modifications). The constraint that defines the meeting point is that the difference in locations is small. Add a single outcome **If** that models this constraint and determines when and where the two objects meet (if they do at all; for certain input parameters the closest value may occur for the initial positions).

If Abs((X_B – X_A) < Small Then ……..

The value of **Small** should be read-in off the worksheet. The **Abs()** function, which takes the magnitude of the expression in parentheses, should be used to avoid to avoid the situation where large negative values are perceived as being small. In physics, negative position values are not small, but refer to a location in a negative region. Similarly, negative velocities simply mean that the object is moving in the negative direction, while negative accelerations indicate that the object has a negatively directed net force acting on it.

6. In problem # 5, simply writing out the position and time for which the **If** condition is satisfied may not produce the best answer (instead, it gives the last value for which the condition is true). A better approach is to search for the smallest difference by continually updating when a better smallest value has been found : **If Abs((X_B – X_A) < Small Then Small = Abs(X_B - X_A) : $X_B$small = X_B : etc. ……..** The starting value of **Small** would be selected reasonably large. The best solution is written after the loop is exited. Defining the "step size" (the amount by which **t** is incremented each iteration) as a variable read-in from the worksheet allows the user to control the precision of the answer. A smaller step size results in positions that are evaluated more often over closer times leading to smaller differences in **(X_B – X_A)** at the meeting point. The downside is that more points must be added to the graph. Modify problem # 5 to search for the optimum solution with a step size as small as 0.1 seconds (so that 100 values are plotted). [Note : do not use **Step** as the variable name to describe the step size; **Step** is a reserved name.]

7. The lens/ mirror equation $\frac{1}{d_o} + \frac{1}{d_i} = \frac{1}{f}$ relates the object location d_o , image location d_i , and focal length **f** for convex and concave mirrors and lenses. [Check the sign conventions in any physics textbook.] The equation is "symmetric" in the unknowns $\mathbf{d_o}$ and $\mathbf{d_i}$ meaning that the solution has the same structure for either of these unknowns.

$$d_o = \frac{f\ d_i}{d_i - f} \qquad d_i = \frac{f\ d_o}{d_o - f}$$ where both expressions have the form $$y = \frac{f\ x}{x - f}$$

A single expression can be used to solve for either value provided the input variable **x** is properly defined. Construct a program (similar to **Exercise # 1**, chapter **2**) which uses **message boxes** and **input boxes** to query the user as to the type of device - lens or mirror, whether its convex or concave, and which unknown is present: $\mathbf{d_o}$ or $\mathbf{d_i}$. The calculation should be made using the general solution **y = (f x)/ (x – f)** , and this expression should only appear once in the program. Display the answer in a message box that describes the situation (eg. **Concave lens**, given the **do = ?? cm** and **f = ?? cm**) In addition, the answer should be written on the worksheet with informative labels and/or titles.