## Appendix 1 Haikus

A **haiku** is "a form of Japanese poetry with 17 syllables in three unrhymed lines of five, seven, and five syllables, often describing nature or a season." (Encarta World English Dictionary)

I found the following in the college photocopier room.

---

In Japan, Sony-Vaio machines have replaced the impersonal and unhelpful Microsoft error messages with their own 17 syllable haikus.

Yesterday it worked.
Today it is not working.
Windows is like that.

Serious error.
All shortcuts have disappeared.
Screen. Mind. Both are blank.

A file that big ?
It might be very useful.
But now it is gone.

Chaos reigns within.
Reflect, repent, and reboot.
Order shall return.

First snow, then silence.
This thousand dollar screen dies
So beautifully.

Aborted effort:
Close all that you have worked on.
You ask way too much.

With searching comes loss
And the presence of absence :
"My Novel" not found.

A crash reduces
Your expensive computer
To a simple stone.

Having been erased,
This document you're seeking
Must now be retyped.

Stay the patient course.
Of little worth is your ire.
The network is down.

Windows NT crashed.
I am the Blue Screen of Death.
No one hears your screams.

Out of memory.
We wish to hold the whole sky,
But we never will.

Three things are certain :
Death, taxes, and lost data.
Guess which has occurred.

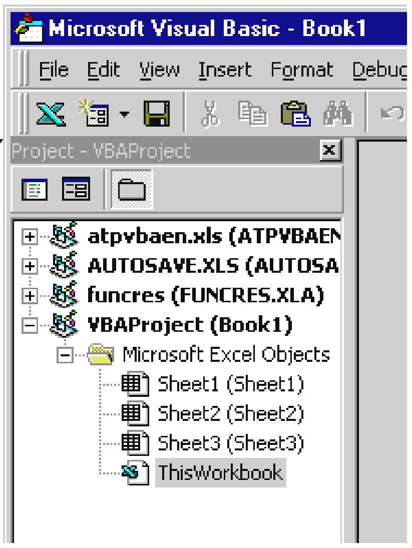
The Tao that is seen
Is not the true Tao, until
You bring fresh toner.

You step in the stream,
But the water has moved on.
This page is not here.

## Appendix 2 Docking & Restoring the Windows

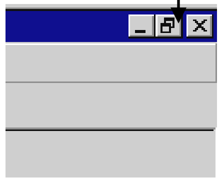
Windows in the **VB Editor** are sometimes "docked" to the sides of the main window. [The expression "docking' is used in the context of a boat which is docked, or attached, to a pier.] To turn off the docking, right click on a blank area of the window concerned, then de-select the **Dockable** option.

**Project Explorer window** docked to top of **VB Editor window**.



If one of the windows, usually the **Code** window, becomes too big it has usually been inadvertently maximized. Restore it to the proper size with the **Restore** button at the top right of the window.

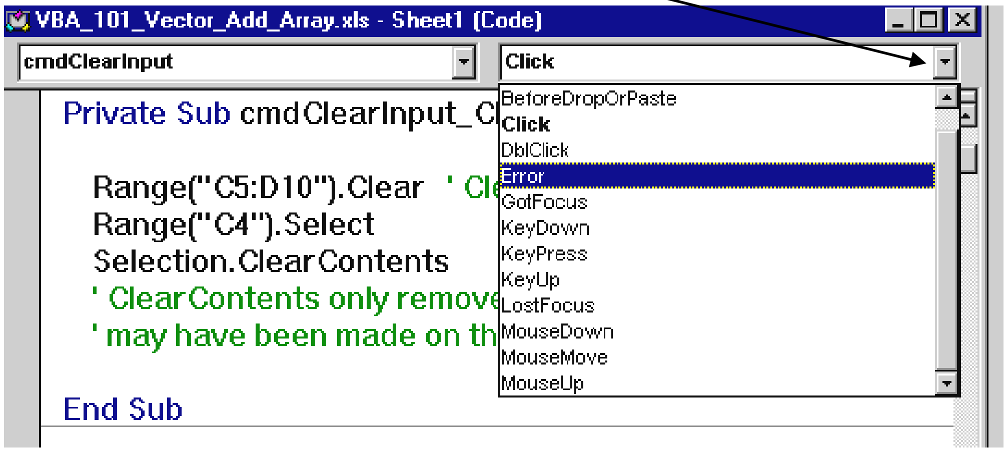
**Restore button**



## Appendix 3 Events

Apart from scroll bar Change events, most of our programs are being activated by what are known as **Click** events : every time a button on the worksheet is clicked the code in the associated subroutine is executed. There are many other events that can be used to trigger the execution of program code. Enter the **Code window** and click the cursor into any of the subroutines associated with the buttons. Now click on the drop down list arrow at the top right of the code window. A list of other events related to the command button will appear as shown on the right. Scroll through the list.. Each control in the Toolbox will have a variety of related events. One of the events that you commonly encounter is the **mouse move** event. If your screen saver is password protected, and you move your mouse, you will trigger a dialog box that asks you to enter a password. If not, moving the mouse will remove the screen saver by triggering an event that returns the window of the software that you were using.

**Drop Down List Box** activation arrow



```
Private Sub cmdClearInput_Cl

    Range("C5:D10").Clear  ' Cl
    Range("C4").Select
    Selection.ClearContents
    ' ClearContents only remove
    ' may have been made on th

End Sub
```

## Appendix 4 Variable Prefixes

Many programmers, particularly business programmers, use a **naming convention** involving prefixes that identify the variable type. In scientific programming there is generally not the same variety of variable types and so prefixes are not often used. In our programs we'll usually follow the convention with strings and occasionally Boolean variables.

| Data Type | Prefix | Data Type | Prefix |
|---|---|---|---|
| Boolean | bln | Integer | int |
| Byte | byt | Long | lng |
| Collection Object | col | Object | obj |
| Currency | cur | Single | sng |
| Date/Time | dtm | String | str |
| Double | dbl | User-defined Type | udt |
| Error | err | Variant | vnt |

## Appendix 5 Control Prefixes

Including prefixes with controls is much more useful than with variables, particularly when many different types of controls are used in a given application.

| Control | Prefix | Control | Prefix |
|---|---|---|---|
| 3D Panel | pnl | MAPI session | mps |
| Animated button | ani | MCI | mci |
| Checkbox | chk | MDI child form | mdi |
| Combobox | cbo | Menu | mnu |
| Command button | cmd | MS Flex grid | msg |
| Common dialog | dlg | MS Tab | mst |
| Communications | com | OLE | ole |
| Data control | dat | Option Button | opt |
| Data-bound combo box | dbcbo | Outline | out |
| Data-bound grid | dbgrd | Pen Bedit | bed |
| Data-bound list box | dblst | Pen Hedit | hed |
| Directory list box | dir | Pen ink | ink |
| Drive list box | drv | Picture | pic |
| File list box | fil | Picture clip | clp |
| Form | frm | ProgressBar | prg |
| Frame | fra | Report | rpt |
| Gauge | gau | RichTextBox | rtf |
| Graph | gra | Shape | shp |
| Grid | grd | Slider | sld |
| Horizontal scroll bar | hsb | Spin | spn |
| Image | img | StatusBar | st |
| Imagelist | ils | TabStrip | tab |
| Key status | keya | Text box | txt |
| Label | lbl | Timer | tmr |
| Line | lin | Tool bar | tlb |
| List box | lst | TreeView | tre |
| ListView | lvw | UpDown | upd |
| MAPI message | mpm | Vertical scroll bar | vsb |

## Appendix 6 Data Types

A list of the available data types, their storage requirements, and range of values is given below. Computers use binary codes ; each **binary digit** (or **bit**) has a value of either **1** or **0**. A collection of **8** bits is known as a **byte**.

| Data Type | Storage Requirement | Range of Values |
|---|---|---|
| **Byte** | 1 byte | 0 to 255 ($2^8$ values available in an 8 bit binary number) |
| **Boolean** | 2 bytes | True or False |
| **Integer** | 2 bytes | -32,768 to + 32,76**7** (16 bits : 1 bit used for + or - leaving $2^{15}$ = 32, 768 values. Zero uses one of the positive values) |
| **Long** (Big Integers) | 4 bytes | -2,147,483,648 to 2,147,483,647 (32 bits : 1 bit used for + or - leaving $2^{31}$ = 2,147,483,648 ) |
| **Single** (Decimal Numbers) | 4 bytes | **-** 3**.**402823E38 to **-**1**.**401298E-45 (negative values) 1**.**401298E-45 to 3**.**402823E38 (positive values) |
| **Double** (Decimal Numbers) | 8 bytes | really big numbers !!!! |
| **Currency** | 8 bytes | Big bucks !!! |
| **Date** | 8 Bytes | January 1, 0100 to December 31, 9999 |
| **String** (variable length) | 10 bytes + string length | 0 to about 2 billion |
| **String** (fixed length) | Length of string | 1 to about 65,400. |
| **Variant** (with numbers) | 16 bytes | Any numeric number within range of double type |
| **Variant** (with characters) | 22 bytes + string length | 0 to about 2 billion |
| **Object** | 4 bytes | Any object reference |
| **User defined** | Varies | Varies by element |

**Excel 2000** also has a **Decimal** data type that provides 28 places to the right of the decimal point.

A **string** is a combination of numeric, alphabetic, and any other keyboard characters (except reserved characters).

## <u>Appendix 7</u>    <u>Calculation Statements</u>

Arithmetic operations are built-in to the **VB Editor** and are made using the operator symbols listed below in order of their <u>precedence</u> ( that is, in order of which operations occur before which other operations).

| **Operator** | **Type** | **Use** | **Description** |
|---|---|---|---|
| ^ | Exponentiation | x^y | Raises x to the power y |
| – | Negation | – x | Negates x |
| * | Multiplication | x*y | Multiplies x and y |
| / | Division | x/ y | Divides x by y |
| \ | Integer division | x\ y | Divides x by y and returns the integer result |
| **Mod** | Modulo operator | x **Mod** y | Divides x by y and returns the remainder |
| + | Addition | x + y | adds x and y |
| – | Subtraction | x – y | Subtracts y from x |

[ The **Int()** function also performs integer division by returning the integer part of the argument.]

Special attention must be given to the precedence, or hierarchy, of the operations in order that the expressions are evaluated in the desired way. It is usually quotients that can lead to an incorrect sequence of operations. For example, suppose you wished to evaluate the quotient $z = \frac{4 + 6}{2 + 3}$.

Writing the code line z = 4 + 6 / 2 + 3 would give the incorrect result of 10 since the division 6/ 2 would be evaluated first, and then added to 4 and to 3. <u>Parentheses can be used to force the evalation of whatever is inside the parentheses first</u> ( of course inside the parentheses the hierarchy of operations is still in effect unless additional parentheses are included). Adding parentheses <u>only to the numerator</u> of the quotient in this example, however, would still produce an incorrect answer :
z = (4 + 6)/ 2 + 3 ⇒ 8 here the 4 and 6 are added first because of the parentheses to give 10, which is then divided by 2, and finally added to 3.

The correct approach for this example is to write z = (4 + 6)/ (2 + 3) which gives 2 .

**Note that on the Worksheet the <u>negation occurs before exponentiation</u>. This means that if you wish to evaluate exp(-x$^2$) then entering the formula =Exp(-(x^2)) into a cell produces a correct answer while entering = Exp(-x^2) does not ! Check this out in Excel Help : Operator Precedence ⇒ About calculation operators ⇒ The order in which Excel performs operations in formulas. Compare with VBA Help under Operator Precedence.**

**Appendix 8** **Implicit Declaration** :

When a variable is not assigned a specific data type VBA automatically assigns it as a **variant** type, which means it can be any of the built-in data types listed in the table of Chapter 1. The variant type is sometimes referred to as a chameleon type as it takes on the type of whatever data is entered. Unfortunately this flexibility, like all flexibility, comes with a price : execution is slower and memory is not efficiently used (each character in the string requires 1 byte !). Slower execution arises from the time consuming checks that VBA must make to determine the current type of the variant. When a specific data type is declared no checks need to be made. Execution can in some cases take twice as long when the variables are by default assigned as variants.

The **TypeName** function can be used to determine the data type of any variable; it returns a string identifying the type. The following code illustrates the chameleon-like nature of the variant data type :

```
Debug.Print  TypeName(XX)
XX =  16
Debug.Print  TypeName(XX)
XX = 16.82
Debug.Print  TypeName(XX)

XX = "Sixteen point 82"
Debug.Print  TypeName(XX)
```

The program output is :

```
Empty
Integer
Double
String
```

The first print statement returns **Empty** because although **XX** is implicitly declared a variant when first encountered in the Debug.Print statement, there has been no value yet assigned to it (it has not been initialized). When the second print statement calls the **TypeName** function, variable **XX** had been assigned an integer value which changed the variant from **Empty** to **Integer**. Prior to the third print statement **XX** has been assigned the **Double** value 16**.**82 (although we might have expected the smaller **Single** data type assignment, the default is **Double**). Finally, entering a string into **XX** changes the variant type to **String**. [Observe again that a string is entered into a variable by enclosing it in quotation marks].

**Note:** We'll later consider **Null** values of variants. **Null** is not the same as **Empty**. It indicates that the variant is intended not to contain any data.

**Appendix 9** **Explicit Declaration Using the Option Explicit Statement**

The mismatch in type of two or more variables in an algebraic expression results in a common programming error that can be easily avoided by forcing the declaration all variable types using the **Option Explicit** statement. The **Option Explicit** is placed in the so-called **General Declarations** section of a module which is right at the top of the code window or module. **Option Explicit** can be added automatically to each project by changing one of the overall settings. In the **VB Editor: Tools menu** ⇒ **Options** ⇒ **Editor Tab** ⇒check off **Require Variable Declaration** in the. {There is no "grandfather clause" here, it is not retroactively applied to old projects, only to new ones.}

## Appendix 10 Public Versus Private Variables, Constants, Functions and Subroutines

Keywords **Public** or **Private** set the scope of the variable, constant, function, or subroutine. The scope of something is loosely the range over which it is recognized, or defined, or can be applied.

### Public or Private Variables and Constants

There are three places where a variable, or a constant can be declared Public or Private :

a) A **Dim** , **Private**, or **Static** statement within a procedure (that is, a function, or subroutine) means that the scope of the variable is within that single procedure. Variables declared within a function or subroutine are automatically private and consequently cannot be declared public.

When a variable is declared within a procedure it is known as a ***local variable*** and it ceases to exist when the procedure (function or subroutine) ends. The usefulness of such a limited scope is that memory is freed up when the variable no longer exists. In certain applications, though, it may be desirable to have a local variable within a procedure retain its value once the procedure ends so that the value is available the next time the procedure is called -- you may want to keep incrementing a certain index, for example. Declaring a local variable as **Static** ensures that its value will be retained once the procedure ends. Eg. **Static Index As Integer**

Constants declared within a procedure similarly become ***local constants***.
Eg. **Const InterestRate As Double = 0.065**

b) A **Dim** statement before the first procedure in a particular module or code window makes the variable available to functions and subroutines throughout that module.

Declaring a constant using the **Const** keyword prior to the first procedure in a module makes it available to all procedures in that module.

c) A **Public** declaration before the first procedure in a module or code window makes the variable available to **all modules**. This declaration must appear in a module, and **not** in the code window for a Sheet.

A constant can be made available to all modules and code windows in a workbook by using the **Public** keyword and declaring it before the first procedure in any module of the project.

Eg. **Public Const FileName1 As String = “Crane_Application”**

### Public or Private Functions and Subroutines

Public functions and subroutines can be called from any module or code window within a project, and, as the adjective implies, have the greatest possible access. Functions and subroutines are public by default making the Public declaration redundant. A private function or subroutine can only be called from within the module in which they are declared.

## Appendix 11 Message Box Constants

**Constants to Set Button Type**

| Constant | Description |
|---|---|
| vbOKOnly (default) | OK button only |
| 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) |

## <u>Appendix 12 Comparison Operators</u>

Comparisons operators compare the value of two variables, or expressions, and return the result of the comparison as either being <u>true</u> or <u>false</u>. They are usually found in **If** statements, but can also be useful in **Do Loops**. A list of the comparison operators in order of precedence is given below.

**If x > 4 Then Vox = - Vox**
**(from the Ball-In-A-Box application)**

```
Do While I < 7
….
I = I + 1
Loop
```

Comparison Operators

| Operator | Type |
|---|---|
| = | Equality |
| <> | Inequality |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |

Notice that an equal sign can serve as either a comparison operator or an assignment operator (when a value is being assigned to a variable).

## <u>Appendix 6 Logical Operators</u>

Logical operators are used in **If** statements and <u>return true or false values</u>. The expressions used with the logical operators involve <u>comparison operators</u> that are <u>first evaluated as true or false</u> before being acted on by the logical operator.

Logical Operators **{Exp is abbreviation of expression}**

| Operator | Syntax | Outcome |
|---|---|---|
| **And** | Exp1 **And** Exp2 | Returns true only if both expressions are true |
| **Or** | Exp1 **Or** Exp2 | Returns true if either Exp1, or Exp2, or both are true |
| **Not** | **Not** Exp | Returns true if Exp is false and false if Exp is true |
| **Xor** {Exclusive Or} | Exp1 **Xor** Exp2 | Returns true if either Exp1 or Exp2 are true (but not both true) |
| **Eqv** {Equivalence} | Exp1 **Eqv** Exp2 | Returns true if both expressions are the same |
| **Imp** {Implication} | Exp1 **Imp** Exp2 | Returns true <u>except when</u> Exp1 is true and Exp2 is false |

## Appendix 13 Truth Tables for Logic Operators

**The *And* Operator**

| | | Exp1 | Exp2 |
|---|---|---|---|
| | | Outcome | |
| True | ***And*** | True | True |
| True | ***And*** | False | False |
| False | ***And*** | True | False |

**The *Or* Operator**

| | | Exp1 | Exp2 |
|---|---|---|---|
| | | Outcome | |
| True | ***Or*** | True | True |
| True | ***Or*** | False | True |
| False | ***Or*** | True | True |

**The *Not* Operator**

| Expr1 | | Outcome |
|---|---|---|
| ***Not*** | True | False |
| ***Not*** | False | True |

**The *Xor* Operator**

| | | Exp1 | Exp2 |
|---|---|---|---|
| | | Outcome | |
| True | ***Xor*** | True | False |
| True | ***Xor*** | False | True |
| False | ***Xor*** | True | True |

**The *Equ* Operator**

| | | Exp1 | Exp2 |
|---|---|---|---|
| | | Outcome | |
| True | ***Equ*** | True | True |
| True | ***Equ*** | False | False |
| False | ***Equ*** | True | False |

**The *Imp* Operator**

| | | Exp1 | Exp2 |
|---|---|---|---|
| | | Outcome | |
| True | ***Imp*** | True | True |
| True | ***Imp*** | False | False |
| False | ***Imp*** | True | True |

## <u>Appendix 14</u>  <u>Looping  Statements</u> :   *Do …Until / While* Loops

With these loops the looping condition can be placed either at the beginning or the end. Loop index I must be initialized, **<u>but does not have to be an integer value</u>**.

| **Do Loop<br>Until Condition** | **Do Until Condition<br>Loop** | **Do Loop<br>While Condition** | **Do While Condition<br>Loop** |
|---|---|---|---|
| **Sum = 0<br>I = 1<br><br>Do<br><br>Sum = Sum + I<br>I = I + 1<br>Loop Until I = 11** | **Sum = 0<br>I = 1<br><br>Do Until I = 11<br><br>Sum = Sum + I<br>I = I + 1<br>Loop** | **Sum = 0<br>I = 1<br><br>Do<br><br>Sum = Sum + I<br>I = I + 1<br>Loop While I < 11** | **Sum = 0<br>I = 1<br><br>Do While I <11<br><br>Sum = Sum + I<br>I = I + 1<br>Loop** |
| **Always executes loop once.  Loops if condition is False** | **Enters loop if condition is False** | **Always executes loop once.  Loops if condition is True** | **Enters loop if condition is True** |

The preceding loops all produce a final sum of 10. Notice that the value of Sum and the loop index I must be initialized before entering the loop  The **Until** and **While** loops both operate by evaluating Boolean expressions to decide whether to continue or terminate the loop. For example, the **Do Until I =11** statement determines the Boolean value of **I =11** . When **I** has a value of , say, 5, the expression **I =11** evaluates to **False**, and the loop continues as long as **I =11** is **False**.

It is not necessary to use an index in the loop condition **;** the looping condition can be based on the value of some some variable. The loop of the previous example can constructed as shown on the right.

```
Sum = 0
Do While Sum < 10
Sum = Sum + 1
Loop
```

The looping condition can involve a Boolean variable :
**Do Until bStop = True**
where **bStop** has been dimensioned as a Boolean variable and initialized as **False**.

## Appendix 15 *For …Next* Loops

Syntax : **For *CounterIndex* = *Start* To *End* [ Step *stepvalue*]**
**……… code statements**
**Next [*CounterIndex* ]**

With this type loop there is no need to initialize the value of the loop index I or to increment it. In addition, a **Step** keyword can be added to increment the index by amounts greater than 1; steps can also be negative.

```
Sum = 0

For I = 1 To 10
Sum = Sum + I
Next I
```

```
For I = 1 To 10  Step 2

For I = 5 To 1 Step -1
```

The loop index can be a variable and, contrary to the examples in many texts, do not have to be integers. Furthermore, the *Start* and *End* values can be any numeric expression that evaluates to a number of the same type as the *CounterIndex*. Consider the following code fragments :

| | |
|---|---|
| Dim Counter As Double<br>Dim AA As Double, BB As Double<br><br>AA = 2.4<br>BB = 5.3<br><br>For Counter = AA To BB Step 0.7<br>  Debug.Print Counter<br>Next | Dim I As Integer, N As Integer, M As Integer<br><br>N = 2<br>M = 12<br><br>For I = N ^ 2 + 1 To 2 * M Step 3<br>  Debug.Print I<br>Next |
| The output is | The output in the Immediate window is |
| **2.4 3.1 3.8 4.5 5.2** | **5 8 11 14 17 20 23** |

**Exiting For … Next Loops Before the CounterIndex Reaches the End Value**

If a For … Next loop is being used to search for something, say a telephone number in a telephone directory or a string of characters within a larger body of text, it is only logical to exit the loop once the item has been found. The loop can be terminated using the **Exit For** command. Typically an **If** statement is used to check a condition which if true directs the program to an Exit For command.

```
For I = 1 To 10
……
If Value > 16  Then
Exit For
End If
…..
Next I
```

## Appendix 16  Custom Functions  (User Defined Functions)

It is often convenient to define custom functions.  In **Exercise # 7** of Chapter 3 (Bacteria Growth) the absence of a built-in base 10 logarithmic function led us to program a derived function using base "e" logs.   The basic structure of defining a function is to type in the keyword **Function** followed by the function name (including arguments).  Pressing **ENTER** to move to the next line should cause the **End Function** part of the function shell to automatically appear. In the actual  defining statement  only the function name, and not the brackets and arguments,  must appear on the left side.  Other code statements can be included as req-uired in the function routine.

```
Function Exptal_Position(t)
Exptal_Position = 2 * t + 3 * t ^ 2
End Function
```

Functions that are to be  used only in the VBA code can be entered anywhere in the **Code window** (after the **Option Explicit** general declarations).  If the function is also,  or only,  to be used on the Worksheet,  it must appear in the separate **Module window** of the VB Editor.  To open a **Module window :  Insert menu** (in VB Editor)  ⇒  **Module**  or right-click on a blank area of the **Project Explorer ⇒ Insert ⇒ Mod**ule**.**

The function arguments are passed by value meaning that whatever variable is present in the main program has its value placed in the argument,  and this value is then transferred to the function defining routine where it becomes the value of whatever variable is being used.  The argument name(s) in the call statement (where the function is being referred to) does/do not have to be the same as the name(s) in the defining routine.  In a later application the position function defining an object's location under constant acceleration motion is defined as :

```
Function Position(Xo, Vox, ax, t)
Position = Xo + Vox * t + 0.5 * ax * t ^ 2
End Function
```

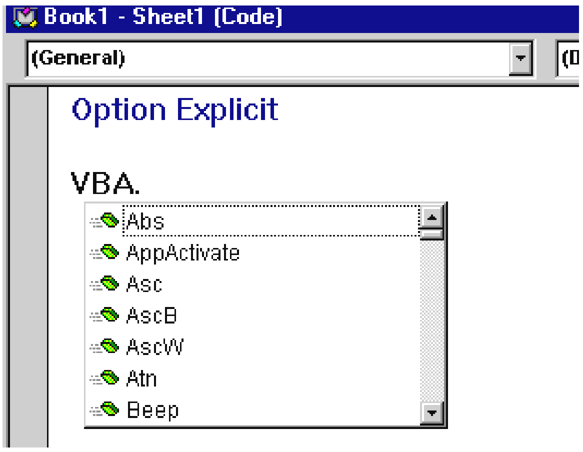
This position function can be used to determine both the X and Y locations under constant acceleration provided the proper data is passed through the arguments. The fact that the argument names are different in the two statements below in which the function is used is of no consequence.  It is not the variable names  that are passed to the function routine,  but the value of the variables.

```
X = Position(Xo, Vox, ax, t)
Y = Position(Yo, Voy, ay, t)
```

Functions can be used with only values included in the arguments :

```
X = Position(0, 30, 0, 1.2)
Y = Position(20, 40, -10, 1.2)
```

## Appendix 17   Built-in & Derived Functions



```
Option Explicit

VBA.
    Abs
    AppActivate
    Asc
    AscB
    AscW
    Atn
    Beep
```

A number of built-in functions, such as **Sin( ), Cos( ), Sqr( )**, etc. are available for use in algebraic expressions. The list of all built-in VBA functions can be displayed by going into any **code window** and typing **VBA.** -- (**VBA** followed by a period). The functions are indicated by the book (brick ??) symbol. If this list doesn't appear go into the **VB Editor**, enter the **Tools menu** ⇒ **Options** ⇒ **Editor tab** and check off **Auto List Members** (while you're there, also check off **Require Variable Declaration** which automatically adds the **Option Explicit** statement). Also note that some functions have different spelling depending on where they're being used. For example, **Sqrt()** is used to take square roots on the <u>worksheet</u> while **Sqr()** is used in <u>code statements</u>.

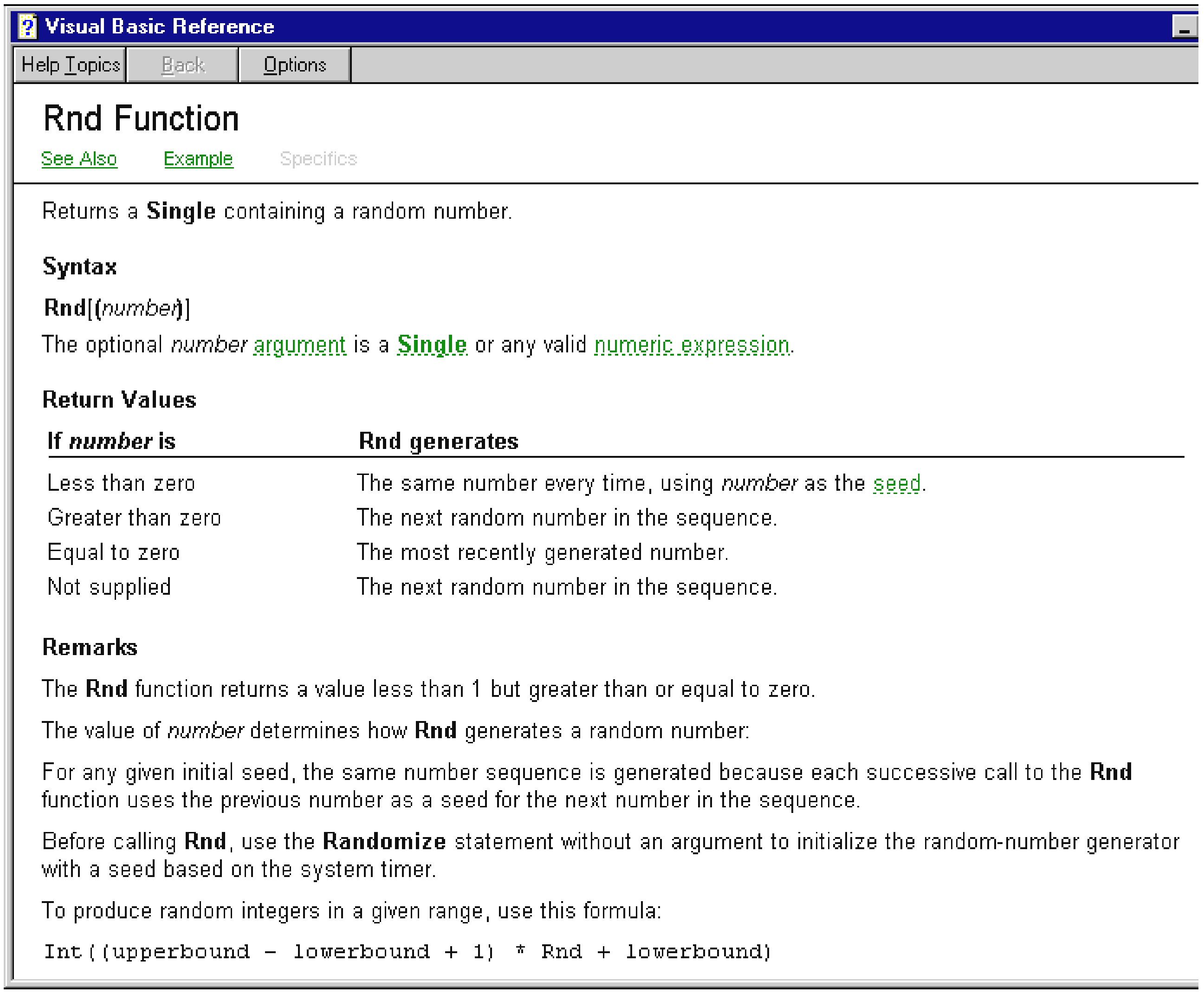
**<u>Derived functions</u>**. Some standard mathematical functions are not provided and must be constructed out of other functions which are given. These are known as <u>derived functions</u> since they're derived from other functions. A list of functions which can be derived from the built-in functions, along with the expressions for doing so, can be found via the **Help** in **VB Editor** (note that **Help** on the <u>worksheet</u> differs from **Help** in the **VB Editor**).

**<u>Accessing worksheet functions.</u>** Fortunately, it is possible to access the larger number of worksheet functions by using either of the prefixes **Application.WorksheetFunction.** or **Excel.WorksheetFunction.** :

**Angle = Application.WorksheetFunction.Atan2(-3, 4)**

**pi = Excel.WorksheetFunction.Pi()**

## Appendix 18 Random Number Generating Function



Rnd Function

Returns a **Single** containing a random number.

**Syntax**

**Rnd**[(*number*)]

The optional *number* argument is a **Single** or any valid numeric expression.

**Return Values**

| **If *number* is** | **Rnd generates** |
|---|---|
| Less than zero | The same number every time, using *number* as the seed. |
| Greater than zero | The next random number in the sequence. |
| Equal to zero | The most recently generated number. |
| Not supplied | The next random number in the sequence. |

**Remarks**

The **Rnd** function returns a value less than 1 but greater than or equal to zero.

The value of *number* determines how **Rnd** generates a random number:

For any given initial seed, the same number sequence is generated because each successive call to the **Rnd** function uses the previous number as a seed for the next number in the sequence.

Before calling **Rnd**, use the **Randomize** statement without an argument to initialize the random-number generator with a seed based on the system timer.

To produce random integers in a given range, use this formula:

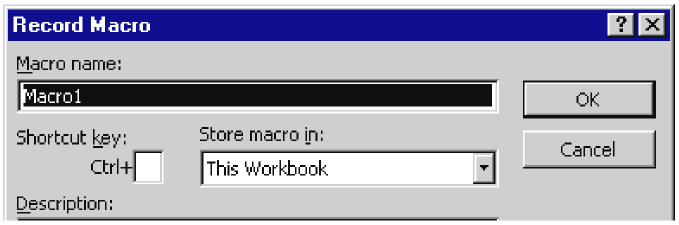
```
Int((upperbound - lowerbound + 1) * Rnd + lowerbound)
```

## Appendix 19 Legal Plagiarism Using the Macro Recorder

The **Macro recorder** allows you to record the code associated with whatever actions you make in your Excel Worksheet or Word Document; it is extremely useful in generating the code statements used by the analytical and processing tools that are available. Once this code is known, actions on the work-sheet, and analysis tools such as the **Chart Wizard**, can be controlled programmatically by VB code. In chapter 5 we'll examine how to invoke the Chart Wizard and select various options with code state-ments. As an example let's see how the code to change the colour of cells, to center text, and to bold text was obtained for **Application #2**.

Open, but don't save, a new workbook. Start the **Macro Recorder** as follows : click on the thick circle beside the play button on the **VB Toolbar**, or go into the **Tools menu** ⇒ **Macro** ⇒ **Record New Macro.** A **Record Macro** dialog box will appear. Use the default macro name, **Macro1**, and press **OK**.

A small box with a **Stop** button may appear on your worksheet which provides one method to stop recording the macro. If no stop button pops up the **Stop** button on the **VB Toolbar** can be used, or, you can go back into the **Tools menu** ⇒ **Macro** ⇒ **Stop Recording**). When the macro recorder is running you must be careful as every action you take will be recorded, and all the extraneous code can be confusing !



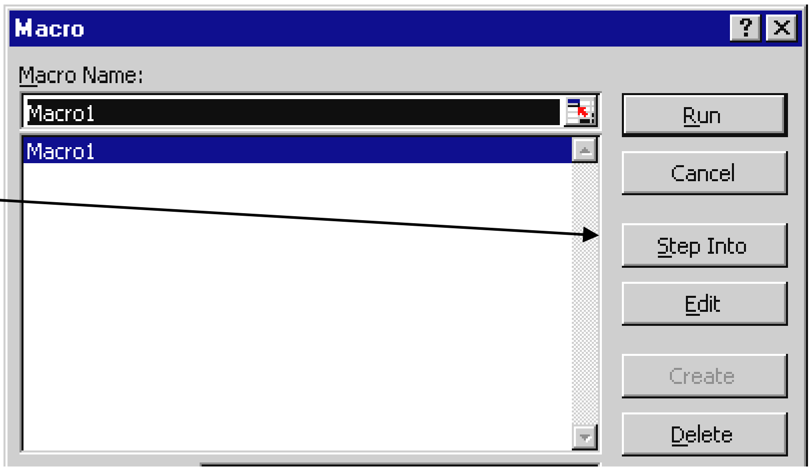
Macro name: Macro1

Shortcut key: Ctrl+

Store macro in: This Workbook

Description:

With the **Macro Recorder** running, highlight cells **C9** and **D9**. Right click, select **Format Cells**, choose the **Patterns** tab, select the yellow to the right of the hot pink (the selection should appear in the sample window), click **OK**. Find the **Bold** button at the top of your spreadsheet, click on it, then click on the **Center** text icon, then **Stop** recording. To access all the code that has been recorded either click on the **Play** button on the **VB toolbar**, or go into **Tools menu** ⇒ **Macro** ⇒ **Macros.** A dialog box similar to that displayed on the right should appear. Select the **Macro1** name and press the **Step Into** button. The code shown below should have been generated.



Macro Name: Macro1

Macro1

```
Sub Macro1()

Range("C9:D9").Select
With Selection.Interior
    .ColorIndex = 44
    .Pattern = xlSolid
    .PatternColorIndex = xlAutomatic
End With

Selection.Font.Bold = True

With Selection
    .HorizontalAlignment = xlCenter
    .VerticalAlignment = xlBottom
    .WrapText = False
    .Orientation = 0
    .ShrinkToFit = False
    .MergeCells = False
 End With

End Sub
```

Much of the code obtained with macro recording gives **default values** that can be deleted (and which I've indicated in ***italics***). Compare this code with the code that was actually used on pg 2-4.

The simile that you might want to make when you're using the macro recorder is that of a fishermen out trolling/trawling for code (or cod !)

## Appendix 20 "With…End With" Statements

The **With …. End With** structure is used to replace long statements with abbreviated ones and serves to un-clutter the program making it more readable as well as reducing the amount of typing. It also makes the program run faster since the object to which the **With** code statements refer only has to be looked up once (this is known as **dereferencing**). The syntax is simple :

**With** ***object***
    **……..**
    **……..**
**End With**

Code statements <u>related to the object</u>.
Often indented for the sake of readability.

The first **With** statement in **A2.2** replaced the code lines :

```
Selection.Interior.ColorIndex = 44
Selection.Interior.Pattern = xlSolid
Selection.Interior.PatternColorIndex = xlAutomatic
```

The following might alternatively have been used if the **Range("C9:D9").Select** statement had been eliminated:

```
Range("C9:D9").Interior. ColorIndex = 44
Range("C9:D9").Interior.Pattern = xlSolid
Range("C9:D9"). PatternColorIndex = xlAutomatic
```

Although the simplification is not that great in this case it can be significant where many, or complex, attributes are being changed.

## Appendix 21  Methods of Selecting More Than 1  Cell

```
Range("C9:D9").Select  ←  Could be replaced by the single line :
Selection.Clear           Range("C9:D9").Clear

Range(Cells(9, 5), Cells(9, 6)).ClearContents
```

The **.Clear** and **.ClearContents** methods do different things :  **ClearContents** only eliminates any number or text entered in the cell  --  it leaves the formatting that may have been applied to the cell, such as colour, centering, font,  bold, etc.  The **Clear** command completely "cleanses" the cell and returns it back to its original default condition.

## Appendix 22  Referencing non-contiguous data columns

In one of the assignment exercises you will be asked to also plot graphs of X versus t ,  $V_y$ versus t, and Y versus X.  Clearly it would be efficient to place the data in only four columns which from left to right would be t,  X,  Y,  and $V_y$ .  When two columns of data that are to be plotted are not adjacent (non-contiguous,  or not touching) the left column is first highlighted and then the **Ctrl** key is pressed and held, the mouse button is released,  the right column is highlighted,  and finally the **Ctrl** key is released  (and then the Chart Wizard is activated).  Using **Record Macro** to plagiarize the associated code shows that the **Range** method used to set the data source would be **ActiveChart.SetSourceData Source:=Sheets ("Sheet1").Range("A2:A9,C2:C9")**. Unfortunately I have yet to find a way of adapting this to a variable size data set using either the city-map type addresses **Range("L" & 4, "M" & Index)**  or the matrix type addresses **Range(Cells(4, 12), Cells(Index, 13))** .

## Appendix 23  Select Case Statement

The **Select Case** statement (such as used in **cmdSolve** of Application #6)  allows different segments of code to be executed depending on the value of some expression,  or some variable.  It is more flexible than using **If** statements to identify a specific situation since **If** statements use true and false evaluation. The syntax of **Select Case** is :

```
Select Case  Expression

        Case Expression Value 1
        …. code lines
        Case Expression Value 2
        …. code lines
        Case Expression Value N
        …. code lines
        { Case Else}        optional
        …. code lines
End Select
```

**Case Else** is an optional choice if none of other selections have been made and it is desired to do something. If no selection has been made, and there is no **Case Else**, the program simply flows through to **End Select** without executing any code. Case Select statements can be nested inside one another.

**<u>Example 1</u>:** From **Two Body Dynamics Application (Double Inclined Plane)**
(may not be included in the current set of notes – speak to me if you're interested)

The string **strSelection** was set to the value of one of the situation choices found in the **Combo Box**. The **Select Case** statement compares the current value of **strSelection** with the 4 possible cases, and then executes the code associated with the particular case.

```
Select Case strSelection
   Case "Initially Moving to Right"
         strSituation = " System is initially moving to the right"
   Case "Initially Moving to Left"
         Uk1 = -Uk1: Uk2 = -Uk2: strSituation = " System is initially moving to the left"
   Case "Initially at Rest,  Accel. to Right"
         strSituation = " System initially at rest;  assumed to accelerate to the right"
         Flag1 = True
   Case "Initially at Rest,  Accel. to Left"
         Uk1 = -Uk1: Uk2 = -Uk2: Flag2 = True
         strSituation = " System initially at rest;  assumed to accelerate to the left"
End Select
```

[ Note : Once the specific case has been detected, and the code executed, the program flow jumps to **End Select**. This would not be the situation if a series of **If** statements were used. Each successive **If** would be examined, even after the correct situation had been identified. ]

**<u>Example 2</u> :** Ranges of values can be specified in the Case choices.

```
Dim  J As Integer
…..
Select Case J
   Case 1 To 10
      …..
   Case 11 To 20
      …..
   Case 22,  27,  52
      …..
   Case Else
      ….
End Select
```

**<u>Example 3</u>** : Comparison operators can be used to define the cases by using the **Is** keyword.

```
Dim  JJ As Integer
…..
Select Case JJ
   Case  Is < 6
   …..
   Case Is >= 7 And  JJ < 11
   …..
   Case Is >= 11
   …..
End Select
```

**<u>Example 4</u>** :

```
Dim  KK As Integer
…..
Select Case KK
   Case Is > 200 Or KK <= 100
      …..
   Case Is >100 And KK < 150
      …..
   Case Else
      …..
End Select
```

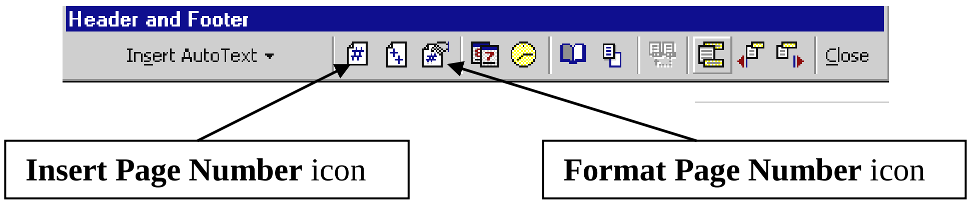
Observe that the case expression must be repeated when making more than one comparison.

## Appendix 24  Automatic Numbering of Pages That Includes Chapter Numbers

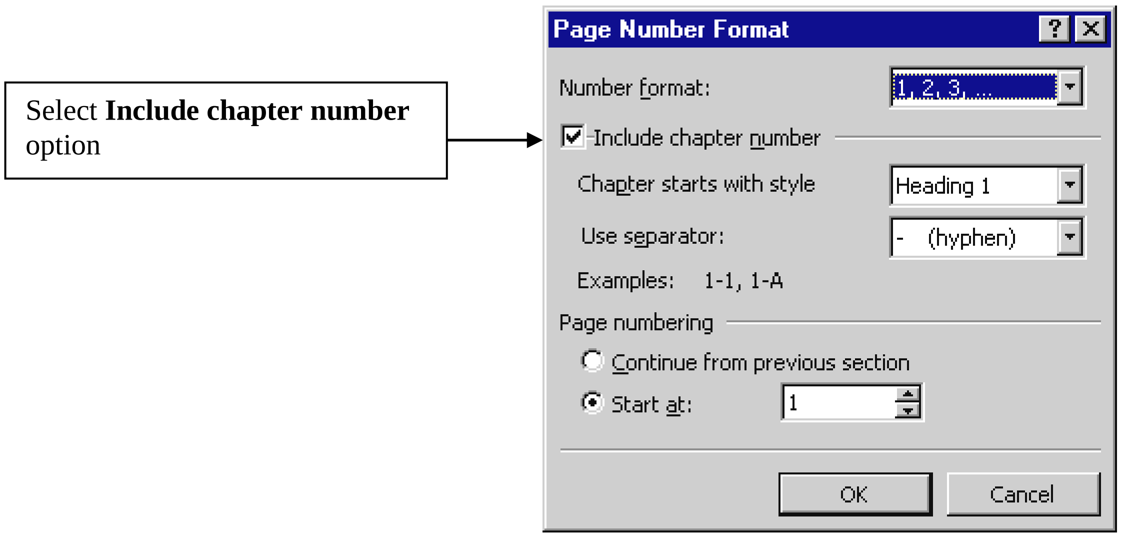
I've included the following description of how to automatically number pages with a chapter number also included more for my own reference, although you may find it useful with certain of your longer multi-chapter essays. The benefit of this approach is that modifying a certain chapter requires making adjustments only to the page numbers of that particular chapter. This is useful when the chapters are contained in separate files.

**View Menu** ⇒ **Header/Footer** . Use the keyboard Tab key to center cursor in Header box at the top of your page, type the chapter number followed by a space and a dash {for example: **3 –** would form the basic shell of chapter 3}. On the **Header/Footer toolbar** select the **Insert Page Number** icon { **#** } , then press **Format Page Number** icon { **# with the hand**}. Check off the **Include chapter number** option, press the **OK** button. An information message box will probably appear, press **OK**, then press the **Close** button on the **Header/Footer toolbar**.

### Header/Footer toolbar



### Dialog Box produced by selecting Format Page Number icon



Page Number Format

Number format: 1, 2, 3, ...

[✓] Include chapter number

Chapter starts with style: Heading 1

Use separator: - (hyphen)

Examples: 1-1, 1-A

Page numbering

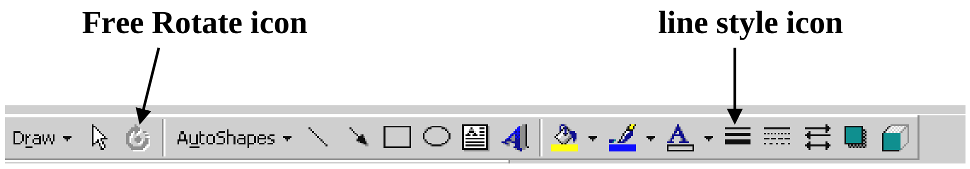
( ) Continue from previous section

(●) Start at: 1

OK | Cancel

## <u>Appendix 25</u> <u>Adding Diagrams on the Worksheet</u>

Display the **Drawing Toolbar** : **View menu ⇒ Toolbars ⇒ Drawing** . For Excel 97 you should obtain :



To draw a block on an inclined plane, for example, click on the **line icon** in the toolbar and then click and drag over the desired length and orientation. Add a block by clicking on the **square icon** and then click and expand the block to some reasonable size. The block can be resized by dragging on a hot spot when the cursor has become a double headed arrow. Hotspots are any of the white squares visible when the block is active (click inside the block to make it active). The block can be shifted (translated) by clicking any point on the periphery that is <u>not</u> a hot spot, and then dragging it wherever desired. To rotate the block, first make it active, and then click on the **Free Rotate icon**. The corners of the block region should now become circles. Click on any of the circles and rotate. Force vectors can be added by using the **arrow icon**. The thickness of the lines comprising any object can be altered using the **Line style icon**. Pulleys can be represented by clicking on the **AutoShapes button ⇒ Basic Shapes ⇒ Donut** , or, you can try resizing an **Oval**. [Making the pulley small enough for the diagram is sometimes challenging ! ] <u>Right-click</u> inside the block to see what other options are available.

## <u>Appendix 26</u> <u>Using an Array to Avoid Dumping the Data Back on the Worksheet</u>

The following code can be used with a special chart control **Microsoft Chart Control Version 6.0 (OLEDB)** which, <u>unfortunately</u>, is available only with the **Professional version** of VB. I have yet to find the equivalent code for use with Excel based VBA. The strategy here is to store the data directly into array **arrXYData** so that there is no need to display the values on the worksheet. The statement **MSChart1.ChartData = arrXYData** turns the array data into the chart data.

```
Dim arrXYData(1 To 100, 1 To 2) As Double

For X = 0 To 4 Step 0.04
   Y = -10 + 20 * X + 30 * X ^ 2 - 10 * X ^ 3
   arrXYData(Row, 3) = X
   arrXYData(Row, 4) = Y
Next X

   MSChart1.ChartData = arrXYData
```

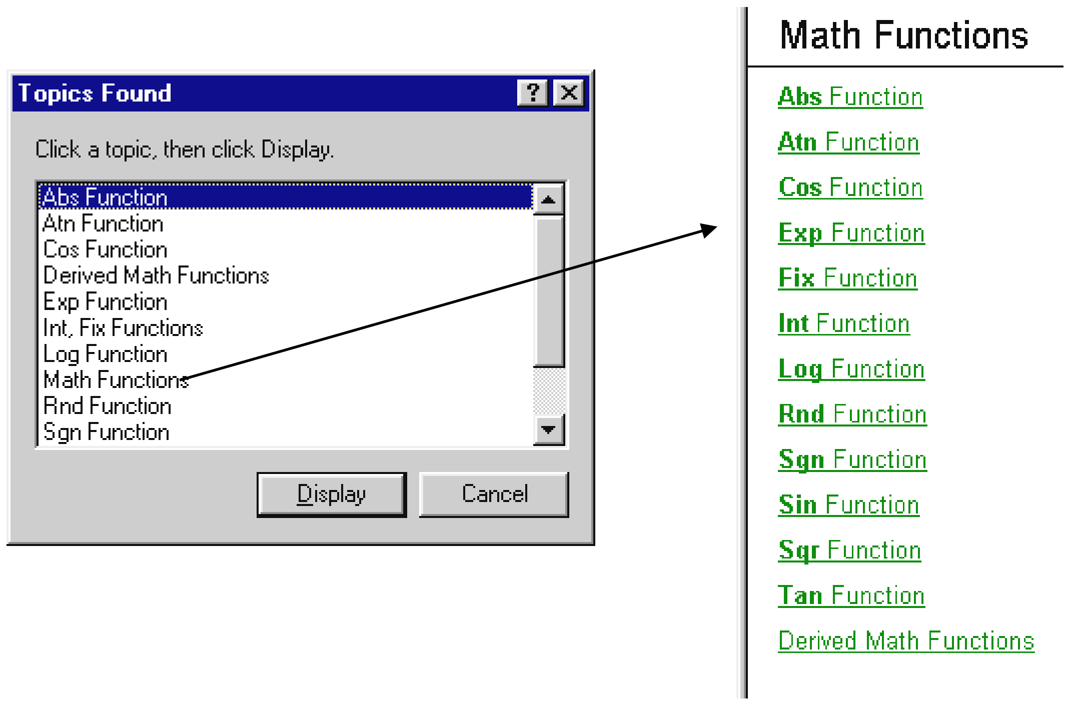
## <u>Appendix 27</u>  <u>Using arrays to store the data sets</u>

It would have been easier to store the position-time data in an array rather than unnecessarily dumping it back onto the worksheet. The problem is to communicate the array location to the **ActiveChart.SetSourceData Source:=** statement, which so far I haven't been able to do. In the professional version of Visual Basic (that is, the Big $$$ version) there is a Chart control that allows its data to be set from an array using **MSChart1.ChartData = arrXYData** where **arrXYData** is the name of an array containing the desired data.

## <u>Appendix 28</u>  <u>Built-in and Derived Math Functions</u>

As with **Excel**, certain built-in functions are provided although, surprisingly, the set is vastly smaller and, inconveniently, the function names are sometimes different !

A search for **Math Functions** from the **VB Editor** (<u>not from **Excel**</u>) produced the following :



Topics Found

Click a topic, then click Display.

- Abs Function
- Atn Function
- Cos Function
- Derived Math Functions
- Exp Function
- Int, Fix Functions
- Log Function
- Math Functions
- Rnd Function
- Sgn Function

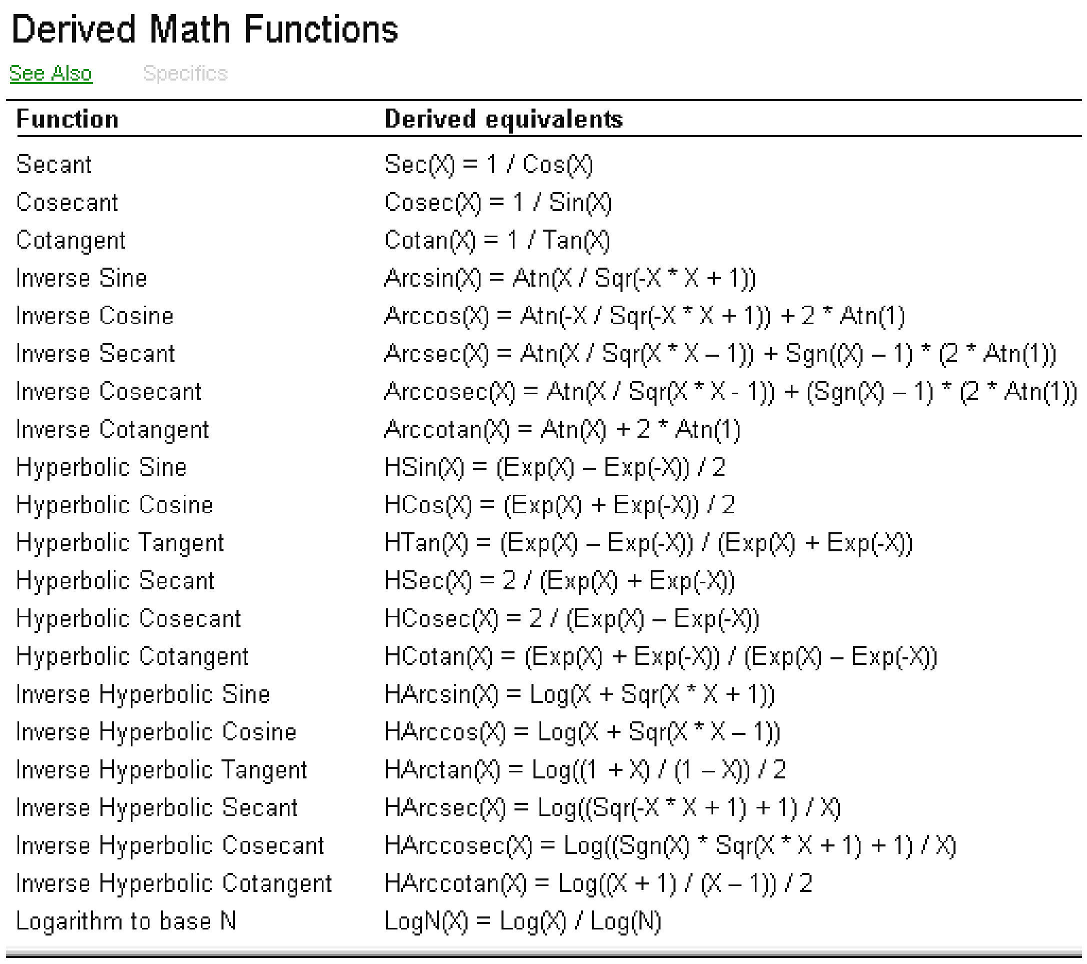
Math Functions

- Abs Function
- Atn Function
- Cos Function
- Exp Function
- Fix Function
- Int Function
- Log Function
- Rnd Function
- Sgn Function
- Sin Function
- Sqr Function
- Tan Function
- Derived Math Functions

**Accessing Worksheet Functions :** All worksheet functions can be used in VBA code by attaching the prefix : **Application.WorksheetFunction.** For example, the 4 quadrant inverse tangent function **ATAN2()** is available using :

**Theta = Application.WorksheetFunction.Atan2(X, Y)**

Derived math functions are those which can be derived (or obtained) from the built-in functions.



## Derived Math Functions

| Function | Derived equivalents |
|---|---|
| Secant | Sec(X) = 1 / Cos(X) |
| Cosecant | Cosec(X) = 1 / Sin(X) |
| Cotangent | Cotan(X) = 1 / Tan(X) |
| Inverse Sine | Arcsin(X) = Atn(X / Sqr(-X * X + 1)) |
| Inverse Cosine | Arccos(X) = Atn(-X / Sqr(-X * X + 1)) + 2 * Atn(1) |
| Inverse Secant | Arcsec(X) = Atn(X / Sqr(X * X – 1)) + Sgn((X) – 1) * (2 * Atn(1)) |
| Inverse Cosecant | Arccosec(X) = Atn(X / Sqr(X * X - 1)) + (Sgn(X) – 1) * (2 * Atn(1)) |
| Inverse Cotangent | Arccotan(X) = Atn(X) + 2 * Atn(1) |
| Hyperbolic Sine | HSin(X) = (Exp(X) – Exp(-X)) / 2 |
| Hyperbolic Cosine | HCos(X) = (Exp(X) + Exp(-X)) / 2 |
| Hyperbolic Tangent | HTan(X) = (Exp(X) – Exp(-X)) / (Exp(X) + Exp(-X)) |
| Hyperbolic Secant | HSec(X) = 2 / (Exp(X) + Exp(-X)) |
| Hyperbolic Cosecant | HCosec(X) = 2 / (Exp(X) – Exp(-X)) |
| Hyperbolic Cotangent | HCotan(X) = (Exp(X) + Exp(-X)) / (Exp(X) – Exp(-X)) |
| Inverse Hyperbolic Sine | HArcsin(X) = Log(X + Sqr(X * X + 1)) |
| Inverse Hyperbolic Cosine | HArccos(X) = Log(X + Sqr(X * X – 1)) |
| Inverse Hyperbolic Tangent | HArctan(X) = Log((1 + X) / (1 – X)) / 2 |
| Inverse Hyperbolic Secant | HArcsec(X) = Log((Sqr(-X * X + 1) + 1) / X) |
| Inverse Hyperbolic Cosecant | HArccosec(X) = Log((Sgn(X) * Sqr(X * X + 1) + 1) / X) |
| Inverse Hyperbolic Cotangent | HArccotan(X) = Log((X + 1) / (X – 1)) / 2 |
| Logarithm to base N | LogN(X) = Log(X) / Log(N) |

## Appendix 29  Adding Series to Graphs & Setting Data Source Locations (Step 2 -- Chart Wizard)

Highlighting data prior to activating the **Chart Wizard** is simple but has a number of serious limitations : the X data must be in the left column, and each data series must be contained in a separate column. Remember that if data columns are not contiguous (touching, or adjacent) you will need to press and hold the **Ctrl** key at the end of highlighting the first column (but here again, the X data must be in the left column).

The most flexible approach to dealing with data that is scattered about the worksheet, and which does not necessarily have the X data in the left column, is to use **Step 2** of the **Chart Wizard**. In the case of an existing graph hover the cursor over a blank region on the graph region (not the borders) until "**Plot Area**" appears and then right click. Select **Source Data** from the menu that appears; the dialog box given below on the left will appear. Alternatively, if the graph cannot be created because the X data is not in the left column, activate the **Chart Wizard** (without highlighting any data), select **XY (Scatter)**, and click on the **Next** button; the following dialog box (on the left) should appear (or right click on a blank region of the graph to give the right window directly):

The **Chart Wizard** is sometimes temperamental and may not provide a graph without some data. If this happens, simply provide a little and adjust the source later (or remove the series entirely).

Click on the **Series** tab at the top of the window to give the dialog box shown on the right :

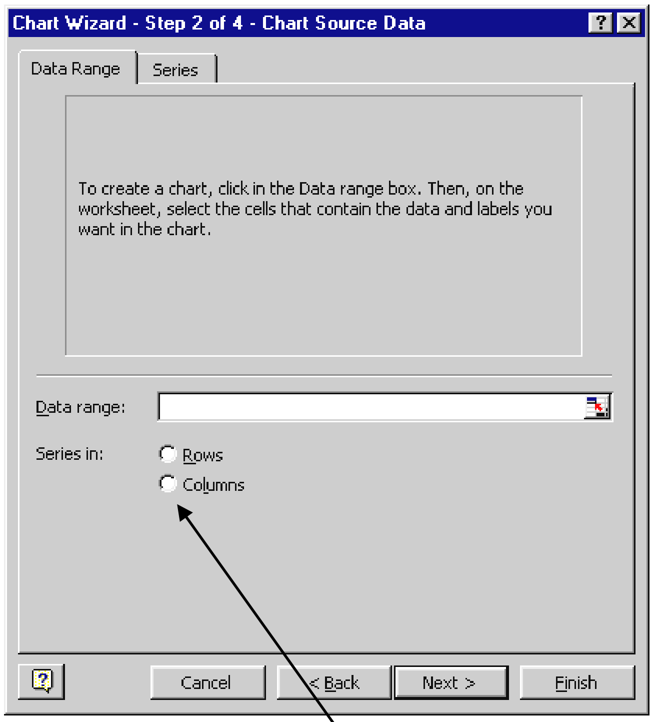
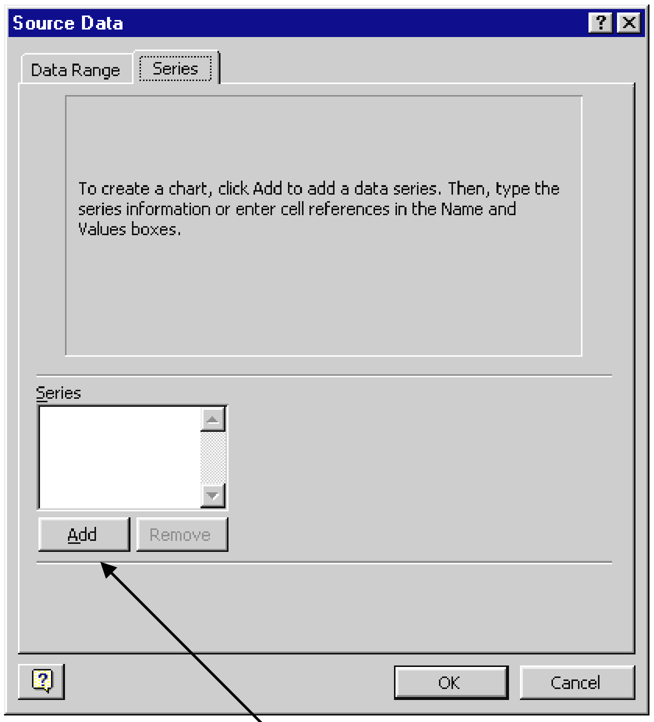


Chart Wizard - Step 2 of 4 - Chart Source Data

To create a chart, click in the Data range box. Then, on the worksheet, select the cells that contain the data and labels you want in the chart.

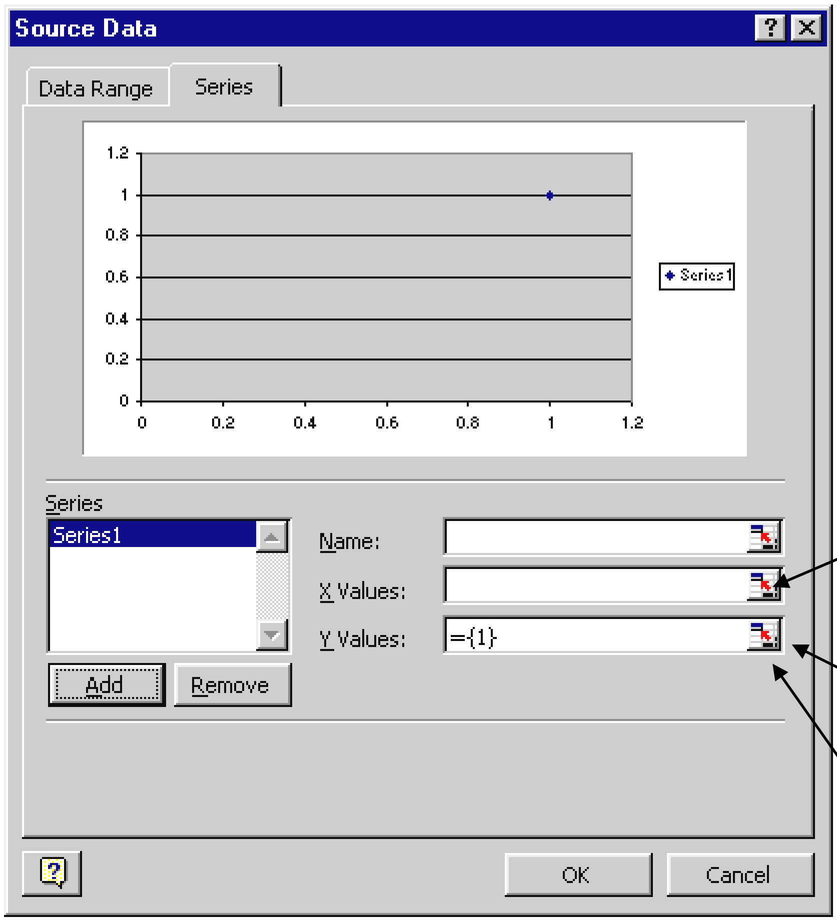

Source Data

To create a chart, click Add to add a data series. Then, type the series information or enter cell references in the Name and Values boxes.

Although good practice, it is not necessary to specify **Rows** or **Columns**. This should occur automatically as you specify the data range (although occasional mis-specifications may occur depending on who used the computer before you). We'll generally want our data in columns.

Click here.

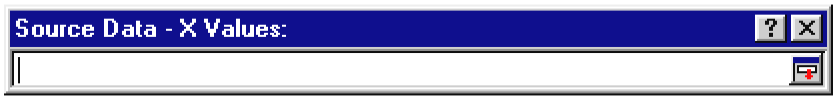
Click on the **Add** button produces the window below (which has a dummy point at (1, 1) :



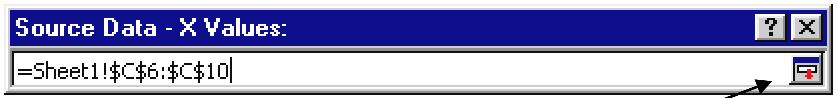
Click here to specify **X** data.

Click here to specify **Y** data.

The X and Y data are added by clicking on the icons at the right end of the input boxes. Clicking the X icon gives the input box :



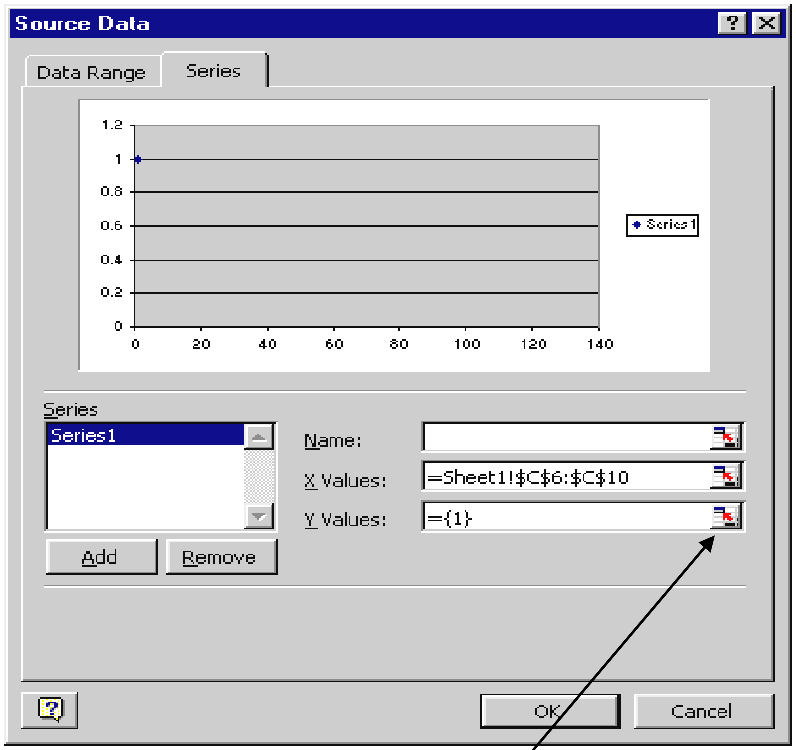
Drag the cursor over the cells containing the X data



=Sheet1!$C$6:$C$10

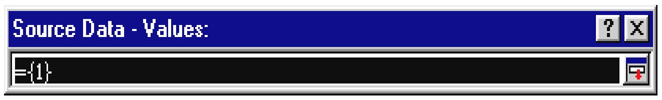
and then click on the icon at the end of the input box.

The X data range is now complete.

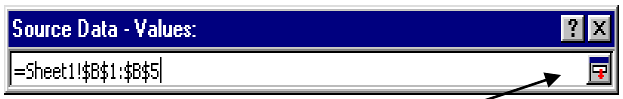


Name:
X Values: =Sheet1!$C$6:$C$10
Y Values: ={1}

Click on the icon at the right of the Y data input box

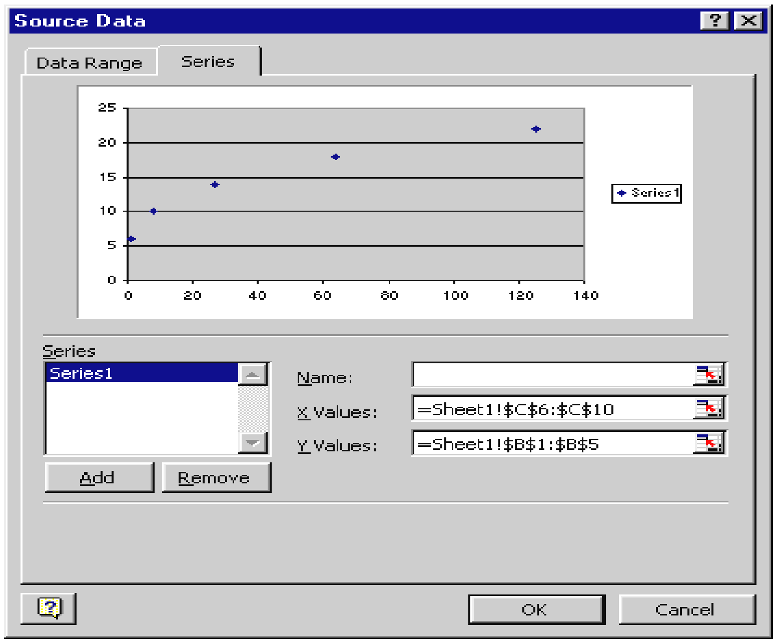


={1}



=Sheet1!$B$1:$B$5

Drag the cursor over the cells containing the Y data;  click on the icon on the right to close the box.



Name:
X Values: =Sheet1!$C$6:$C$10
Y Values: =Sheet1!$B$1:$B$5

Additional data series can be added by clicking on the **Add** button,  and then repeating the process.

**APPENDICES**