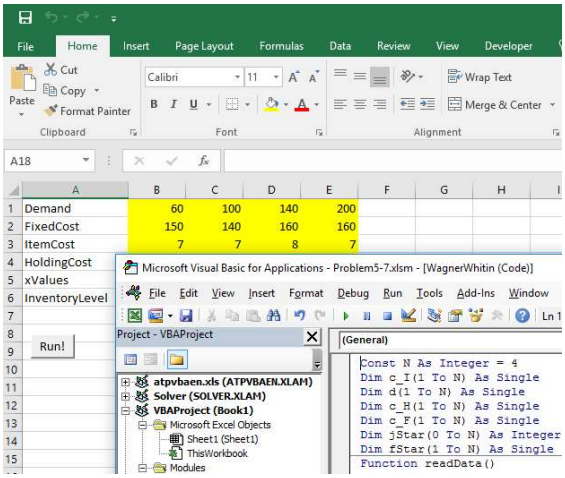
Using Excel in TPK5115, TPK4161, PK6021

By Jørn Vatn
Date 2017-09-05



1.Comma, semicolon and VBA strings

MS Excel treats numbers and lists according to definitions given in "Region and Language". Below we discuss the following

1. Decimal symbol
2. List separator
3. Syntax for VBA-strings

The discussion below relates to MS Excel. There might be different approaches in Excel for the Mac.

1.1Decimal symbol

In many languages the comma (,) is the symbol used to separate the decimal part of a number from the integer. For example we write $\pi \approx 3{,}14$. In the English language a period (.) is used, and one writes $\pi \approx 3.14$. I this course a period is used as the decimal symbol in slides, course compendium and in MS Excel demonstrations. It is recommended to edit the “Regional settings” on your computer so that the period is used as the decimal symbol. This is explained later down.

1.2List separator

Many functions in MS Excel require two or more arguments. For example to find the larges value fo two A1 and A2 we may use the Max() function, by e.g., typing in cell A3: =Max(A1,A2). Note that the English name of the maximum function is used. If your MS Excel is set up with a national language, the Max() function has to be replaced by a national language variant, e.g., Størst() in Norwegian. The separator between the two arguments is here the comma (,), i.e., the list separator corresponding to a standard English configuration in the Regional settings. In Norwegian and many other European countries the standard list separator is the semi colon (;). It is recommended to edit the “Regional settings” on your computer so that the comma is used as the list separator. This is explained later down.

1.3Syntax for VBA-strings

In the routine for numerical integration in pRisk.xlsm (supported in some NTNU courses), the integrand is processed by visual (VBA). The integrand is enclosed in (") in pRisk.xls, and this text string has to be stated in English syntax, i.e., if numbers are specified they have to have the period as the decimal symbol, and when arguments are to be separated, the comma has to be used as list separator. Not that this applies independent on your Regional setting.

1.4Editing the Region and Language

To change the standard configuration of your PC, choose the **Control panel** from the Start button on the lower left corner of your screen. Then choose **Region and Language**. At the bottom of this menu **Additional settings....** From this menu you may edit the symbols used for the decimal symbol and the list.

2.Introduction problems

Problem 1
We are considering the maintenance of an emergency shutdown valve (ESDV). The ESDV has a hidden function, and it is considered appropriate to perform a functional test of the valve at regular intervals of length τ. The cost of performing such a test is NOK 10 000. If the ESDV is demanded in a critical situation, the total (accident) cost is NOK 10 000 000. The rate of demands for the ESDV is one every 5 year. The failure rate of the ESDV is 2×10^{-6} (hrs^{-1}). Determine the optimum value of τ by:

- Finding an analytical solution
- Plotting the total cost as a function of τ
- Minimising the cost function by means of numerical methods

Problem 2
In order to reduce testing it is proposed to install a redundant ESDV. The extra yearly cost of such an ESDV is NOK 15 000. Determine the optimum test interval if we assume that the second ESDV has the same failure rate, but that there is a common cause failure situation, with $\beta = 0.1$. Will you recommend the installation of this redundant ESDV?

3.Basic worksheet operations

3.1Using “variable names” in the excel sheet

When using Excel from the worksheet windows, cells could be referred to by the row/column name of the cell, e.g. A1 is the name of the upper left cell. The cell name is shown in the Name Box pointed to by the arrow in Figure 1.

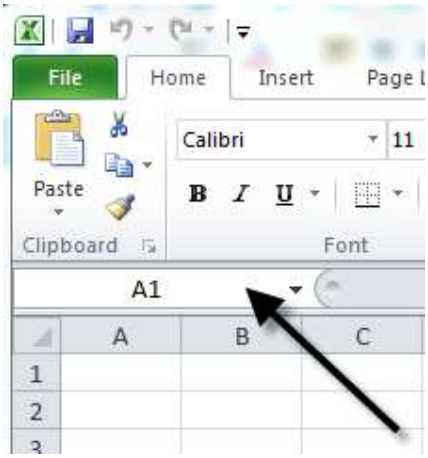


Figure 1 Name Box

If we want to refer to the value within this cell in an expression, we could just refer to the cell by its name. For example, if we in cell B1 want to calculate the square of the value in A1, we just type **`=A1^2`** in cell B1.

When a lot of variables are defined in a worksheet, it will not be easy to read the formulas used if we are always referring to variables by more or less arbitrary names. Hence, we would like to give the variables more meaningful names. For example, if the cell A1 represents temperature, we would rather refer to e.g. Temp, rather than A1. In order to accomplish this, we just type **`Temp`** in the Name Box input filed. Now, the expression we would like to type in cell B1 would be more easy to read, .e.g. we type **`=Temp^2`**.

Note, that if you want to use a variable like x_2, it would be natural to write **`X2`** in the cell window. However, since X2 is a cell name already defined by Excel, this will not work (the result would be that the active cell changes to X2). In order to prevent confusion with the predefined cell names in Excel, we rather specify **`X_2`**.

Further note that we could specify more than one name for each cell. This will however be difficult to trace, and should be avoided. If we by accident give one cell a wrong name, we should first delete the cell name, before we give a new cell name. To delete the cell name, choose Name Manager from the Formulas tab. Then search for the cell name to delete.

If we have given a cell on one worksheet a name, we could refer to this cell from an arbitrary other work sheet just by specifying the cell name we have given. If we have not given a cell a name, and want to refer to it from another worksheet we have to prefix the cell name with the worksheet name, e.g. **`=Sheet1!A1`**.
Often we label the variables to use in one column, and then insert their values in the column to the right. To easy give the corresponding cells the names given in the label cells then:

1. Give variable names in a column of cells.
2. Mark these cells and the column to the right.
3. On the **Formulas** tab, in the **Defined names** group, click **Create from Selection**.
4. Tick Left column, and press OK in Figure 2.

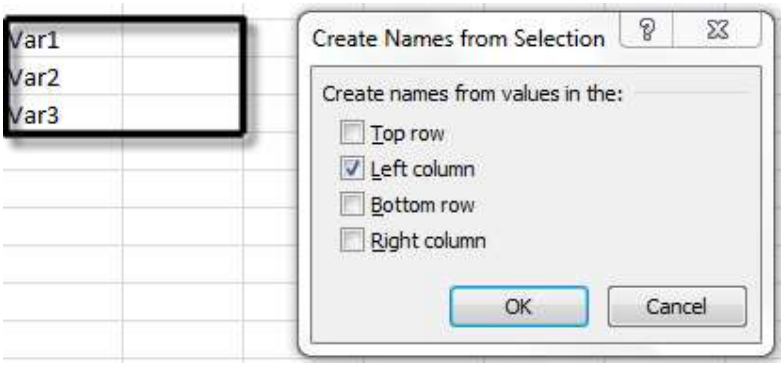


Figure 2 Create cell names from selection

Ensure that the labels for the variables are legal cell names, i.e., do not contain blanks or special symbols. If predefine cell names like A1 is used, Excel will add a "_" to the variable name, e.g., A1_.

3.2Cell operations

To sum a range of cells and put the result in another cell, move the cell selector to the cell you want to store the result in, and type **`=Sum(`**, then drag the cell selector over the cells you want to sum, release the mouse button, and click Enter. Excel automatically completes the formula, e.g. the resulting formula will be e.g. **`=SUM(A1:A5)`**. Note that national versions of Excel require national function names when used from the worksheets. Other functions that often are used to manipulate cells would be **`=AVERAGE(A1:A5)`**, **`=STDEV(A1:A5)`**, **`=MIN(A1:A5)`** and **`=MAX(A1:A5)`**. For more advanced functions, we refer to the Excel Help function.

3.3Plotting the results

Excel provides a wide range of possibilities for visualising the data stored in the worksheets. Very often we have data where one column represent x-values, and subsequent columns represent y-values. To draw graphs representing the various y-values:

1. Mark the corresponding cells (were the x-values are stored in the first column and the y-values are stored in the second column)
2. On the **Insert** tab, in the **Chart** group, click **Scatter**
3. Select an appropriate chart sub-type.

Example
Consider Problem 1. The cost per time unit would be

C(tau) = PMCost/tau + tau*lambda*dRate*HCost

where we have specified the parameters like:

Parameter	Value
PMCost	10000
Hcost	1E+07
lambda	0.01752
dRate	0.2

Figure 3 Parameters for Problem 1

Where the legend for each variable in Figure 3 corresponds to the cell name we have specified. To plot the cost function, we specify the tau values in one column, then the PM cost, the Accident (Risk) cost and the total cost in subsequent columns, e.g.

	Tau	PM	Risk	Tot
13	0.4	25 000	7 008	32 008
14	0.5	20 000	8 760	28 760
15	0.6	16 667	10 512	27 179

Figure 4 Section of data to plot

In cell A13 we specify **`0.4`**, then in cell A14 we write **`=A13+0.1`**. We may now copy the formula in cell A14 by first selecting cell A14, pressing <Ctrl>C, then selecting the cell A14 to A34, and pressing <Ctrl>V. In cell B13 we specify **`=PMCost/A13`**. In column C13 we specify **`=lambda*A13/2*Hcost*dRate`**. The total cost are now entered in cell D13 by **`=B13+C13`**. Cells B13-D13 are then copied to subsequent rows. To find the minimum cost graphically, we create the plot in Figure 5.

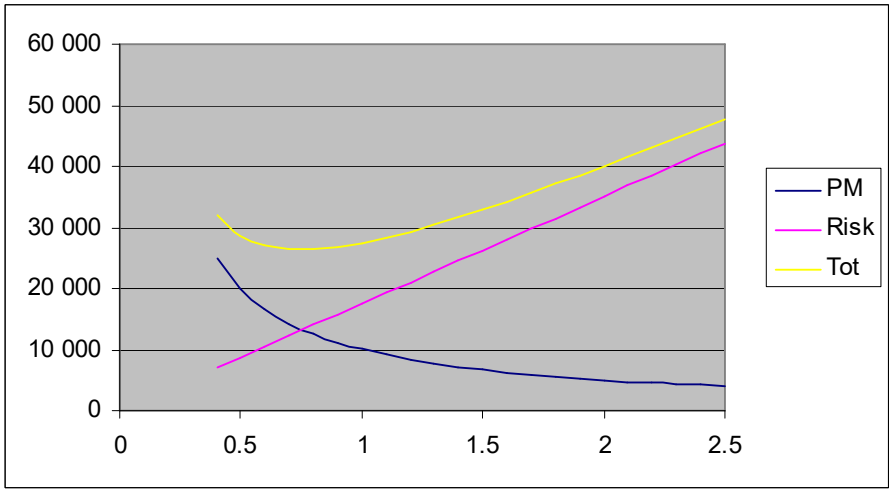


Figure 5 Plot of data in problem 1

3.4Using the Solver (Problemløser) Add-In to minimize the value of a cell

In problem 2 we might calculate the total cost for a given inspection interval tau by a step of calculations, see Figure 6.

Parameter	Value	Formula
PMCost	10 000	
Hcost	10 000 000	
lambda	0.01752	
dRate	0.2	
beta	0.1	
n	2	
k	1	
tau	0.7	
PDFC	0.0006132	=beta*lambda*tau/2
PDFI	4.061E-05	=COMBIN(n,n-k+1)*((1-beta)*lambda*tau)^(n-k+1)/(n-k+2)
PFD	0.0006538	=PDFC+PDFI
PM	14285.714	=PMCost/tau
Accident	1307.6191	=PDF*dRate*HCost
TotCost	15593.333	=PM+Accident

Figure 6 Calculation of total cost in Problem 2

We now use the solver to minimize the total cost:

1. On the **Data** tab, in the **Analysis** group, click **Solver**[1].
2. In the **Set Objective** box, enter a cell reference or name for the objective cell. The objective cell must contain a formula. In the example specify **TotCost.**
3. Do one of the following:
    a. If we want the value of the objective cell to be as large as possible, click **Max**.
    b. If we want the value of the objective cell to be as small as possible, click **Min**. Since we will like to minimize cost, click **Min**.
    c. If we want the objective cell to be a certain value, click **Value of**, and then type the value in the box.
4. In the **By Changing Variable Cells** box, enter a name or reference for each decision variable cell range. Separate the nonadjacent references with commas. The variable cells must be related directly or indirectly to the objective cell. In the example, specify **tau**.
5. In the **Subject to the Constraints** box, enter any constraints that you want to apply. See Excel Help for more instructions.

[1] If the **Solver** command or the **Analysis** group is not available, you need to load the Solver Add-in program (In Norwegian Solver = Problemløser).

1. Click the **File** tab, click **Options**, and then click the **Add-Ins** category.
2. In the **Manage** box, click **Excel Add-ins**, and then click **Go**.
3. In the **Add-ins available** box, select the **Solver Add-in** check box, and then click **OK**.

3.5What if analysis

Another way to find this minimum would be to use so-called What-If analysis.

1. Create the tau-values in one column. This is accomplished similar to what was done in Section 3.3
2. Lave the column to the right for Excel to fill in
3. In the cell just above the upper destination cell for the total cost values, specify **`=TotCost`**.
4. Mark two columns, the first column represent the tau values, and the second column represent the destination cells for the total cost. When marking these cells, also include the row *above* the data cells, i.e. the row containing the **`=TotCost`** cell
5. On the **Data** tab, in the **Data tools** group, click **What-If-Analysis** .
6. Select **Data Table.**
7. In the Column input cell, specify **tau**, see Figure 7.

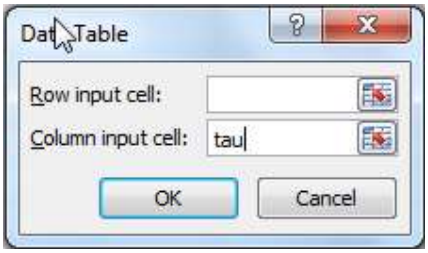


Figure 7 Specification of Column input cell

Excel will now recalculate the total cost by changing the value of the **tau** cell according to the list of tau values stored in the first column of the selected area. Then Excel store the result in the second column of the selection.

4.Creating and using VBA functions

4.1Introduction

VBA is the programming language offered by Microsoft Office programs (Word, Excel etc). The basic principles and syntax is similar for all VBA's independent of which program they are used in. However, the way we access data are quite different. In e.g. Excel data from the worksheets are specified with the Range() function, whereas in Access stronger database functions are available. Note also that Excel provides a very nice set of worksheet functions that also are available from the VBA code. These functions are generally not available from other Office programs, meaning that using these functions cause problems if you want to copy the code to e.g. Access.

Note that Excel files containing VBA code need to be **Saved As** an Excel Macro-enabled Workbook (*.xlsm).

To invoke the VBA editor, just press <Alt>F11. In the VBA editor, choose Insert and Module to create a module holding the declarations and other stuff defining functions and procedures.

In Excel for Mac things are slightly different. First of all the VBA editor is only available if the **`Developer tab`** on the Excel Ribbon has been activated. On the Ribbon dialog box, under **`Show or hide tabs`** there is a checking box for the **`Developer`**. Then going to the **`Developer tab`** and clicking **`Editor`** opens the VBA Editor.

The different VBA functions and procedures are stored in so-called modules. The modules are default given name by Excel, i.e. the first one starts with Module 1, the second Module 2 and so on. It is however, possible to give the modules new names that are more informative, e.g. NumIntLib for a library of procedures for numerical integration. It is a good idea to collect procedures and functions that relates to each other in one module. A module comprises two main parts:

- Common declarations
- The functions and procedures

In the declaration part you typically define variables that are common to all functions in one module, or that should be common to all modules. In the declaration part we specify either variables that should be available only from the actual module. These variables are specified by the Private statement, e.g.

```
Private xValue As Single
```

Note that this statement should be given in the top of the module before any declaration of functions or procedures. Later on, in a function or a procedure you may use the variable xValue. Note that the xValue will be available in all procedures and functions in the module where it is defined. This means that you might give a value to **`xValue`** in one function, and then use this value in another function.

Sometimes you want variables to be available from all functions in all modules. You then use the Public statement, e.g.

```
Public TimeUnit As String
```

You might in one module, e.g. the InitModule write a function that set the TimeUnit, e.g.

```
TimeUnit = "Hours"
```

and you then access the value from another module, e.g.,:

```
MsgBox "The time unit is " & TimeUnit
```

Most variables you need should however, only be defined within one function. E.g. if you need a counter, you define it within one function, e.g.

```
Function Sum1To10()
Dim x as integer
Dim s as single
s = 0
For x = 1 To 10
      s = s + x
Next x
Sum1To10 = s
End Function
```

Figure 8 Using the Dim statement

The Sum1to10() function is now available any spreadsheet cell by typing **`=Sum1To10()`** in the cell. While debugging your code it is convenient to test it from the VBA editor. Figure 1 shows the set-up for this. Click on the bar left to the code and a red bullet is shown in the bar. Another click removes the red bullet. You can create as many bullet as you will. When the module runs it stops at each line with a bullet associated to it. To run the module put cursor some place in the module, and press the F5 key. The code execution now stops at each line with a bullet. You may then move the cursor over any variable available in within this function. The F5 key is also the key to use on the Mac.

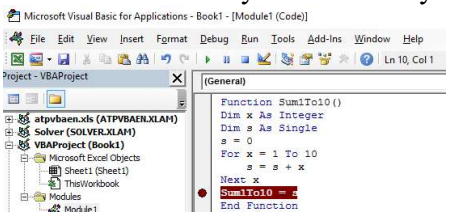


Figure 9 Running the function from the VBA editor

Note that if a function or procedure has arguments it is not possible to just run the function because the arguments are not defined. To test such a function it may be run from the **`Immediate`** window. To open the Immediate window press ^G. From this window you may type **`debug.print Sum1To10()`**. The value (55) is now shown in the Immediate window. For example if we had written the function **`Sum1ToN(N as integer)`**, we type from the Immediate window: **`debug.print Sum1ToN(10)`** to produce the same result. In Mac the Immediate window is opened by ^⌘G.

In Figure 8 we have used the **Dim** statement to define the variables **x** and **s**. Note that **x** and **s** would not be available from other functions. But, you may define **x** in another function and use **x** in the same way in that other function. Note all **Dim** statements should be specified before any executable code.

Sometimes you want do define constants rather than variables. For example you might want to specify a constant for gravity, and you write:

```
Const gravity As Single = 9.81
```

A constant statement could be specified either in the declarations part of the module (top of module), or the declaration part of the function (i.e. before any executable code)

Passing arguments to a function or a procedure is accomplished by the statements in the header of the function. When you define the function, you also define the variable types to be passed, whereas when you call the function you only pass the variables, or values, e.g.

```
Function MySum(x As Single, y As Single)
MySum = x + y
End Function
```

Figure 10 Specification of arguments

You might then later from another function call MySum, e.g. **x = MySum(3, 4)**. Note that if you define an argument as e.g. Single, you cannot call the function with a variable of e.g. Integer type. See the reference manual for variable types to use within VBA.

Loops are programming constructions that you will need. The simples loops are accomplished either by the **For** statement, or the **Do** statement. In the following function two loop constructions are used to count the numbers from 1 to 10:

```
Function Sum1To10()
Dim x As Integer
Dim s As Single
s = 0
For x = 1 To 10
    s = s + x
Next x
Debug.Print s
s = 0
x = 0
Do While x < 10
    x = x + 1
    s = s + x
Loop
Debug.Print s
End Function
```

Figure 11 Loop constructions

In the first construction we use the **For** statement to specify the start, end, and optional the increment of the counter variable x. Here we might have specified

```
For x = 2 To 10 Step 2
```

If we wanted to count only even numbers. The statements to be executed for each step are specified before the **Next** statement.

In the **Do While** construction we instruct the computer to repeat as long a logical expression is true. The statements to be executed for each step are specified before the **Loop** statement. You might jump out of the loop by an **Exit For**, or **Exit Do** statement within the loop construction.

Note that we want a function to return a value, this is done by assigning an expression to the function name at the end of the function, e.g.

```
Sum1To10 = s
```

If a function should not return a value, you could alternatively use a procedure construction, see the Excel reference for further information.

4.2Simple functions

In the previous example we calculated the probability of failure on demand (PFD) for the safety valve by a number of steps. A more elegant approach would be to crate a function accomplishing these calculations. In Figure 12 we have shown the VBA code for the PFDb function, where "b" indicate that the beta-factor model is assumed. VBA (Visual Basic for Applications) is the MS Office programming language.

```
Function PFD(lambda As Single, tau As Single, beta As Single, _
k As Integer, n As Integer)
' Beta factor model
' PFD = Probability of failure of demand
' PFD = Dependent part + Independent part
' Independent part = Combin(n,n-k+1)(lambda*(1-beta)*tau)^(n-k+1)/(n-k+2)
' lambda = total failure rate = lambda_commoncause + lambda_independent
'
PFD = lambda * tau * beta / 2 + _
((1 - beta) * lambda * tau) ^ (n - k + 1) / (n - k + 2) * _
Application.WorksheetFunction.Combin(n, n - k + 1)
End Function
```

Figure 12 VBA code for the PFD function

To invoke the VBA editor, just press <Alt>F11. In the VBA editor, choose Insert and Module to create a module for storing the function.

From the Excel worksheet you might now call the function you have crated, e.g. in cell B17 you could specify **=PFD(lambda,tau,beta,k,n)**.

When you crate functions in VBA you might want to get data stored in cells without passing these values as arguments to the function. To accomplish this, use the Range() function in VBA: The Range() function is specified as e.g., **Range("lambda")**, where **lambda** is a cell name.

4.3Loops

VBA provides three types of loops

- For loop
- For Each loop
- Do While loop

The for loop is used when there is a natural variable to use as a counter in the calculations, for example:

```
For x = 1 To 10
      s = s + x
Next x
```

It is possible to define the increment in x by each step, for example counting only odd numbers:

```
For x = 1 To 10 Step 2
      s = s + x
Next x
```

The increment could also be negative, and even fractions if **x** is defined as a **Single** type. In some situations we would like to loop through all elements of an array, a range of worksheet cells etc. We then use the **For Each** construct:

```
For Each c in Range("xValues")
      s = s + c
Next x
```

where **xValues** is the name of a range of cells in the worksheet. The **Do While** construct is used when the there is no "counter" or set of values to follow:

```
x = 1
Do While x <= 10
    s = s + x
    x = x + 1
Loop
```

The condition `x <= 10` could be replaced by any condition calculated inside the **Do Loop**. Note that the condition needs to be initialized before the execution enters into the **Do Loop**. In some situations we do not have an initial condition, and we could use an alternative construct:

```
x = 1
Do
    s = s + x
    x = x + 1
Loop Until x > 10
```

In many cases we need to escape from the construct, and we may use the statements **Exit For** and **Exit Do,** causing the execution to continue just after the **Next** or **Loop** statement.

4.4Advanced VBA functions

Sometimes you might want to create more complicated functions, e.g. a function for numerical integration. Such a function would require the following elements:

- The name of the function to integrate
- Parameters used in this function
- Limits for the integration

The VBA language is not optimal for such programming because we could not pass a function name as an argument to a function as we could do in e.g. FORTRAN or C++. A work around approach would be to create a general purpose "Exec-function", which takes two arguments, an integer representing the function name (or number), and the argument, e.g.

```
Function execFunc(f As Integer, x As Single)
Select Case f
Case 1
execFunc = Sin(x)
Case 2
execFunc = Cos(x)
End Select
End Function
```

Figure 13 Simple Exec-function

If we in a program system need first to integrate the sin function, then the cos function we could write a general purpose numerical integration function, which we first call with argument **`1`**, and the **`2.`** To make the code more readable, we typically define constant like: **`Public Const eFuncSin = 1`**, and so on. In the example above, we did not pass any parameters to the execFunc. In some situations we would like to pas an argument, for example we would like to have a more general cos function like **`a*Cos(b*x+c)`**. One way to accomplish this would be to store a, b and c in a variant variable. For example before calling the numInt function we specify **`PassPar=ARRAY(1,2,3)`** to pass the parameters a=1, b=2 and c=3. We could then call the NumInt function by **`NumInt(eFuncCos,PassPar,0,3.141)`**.

It would then be the task of the NumInt procedure to pass further the variant PassPar to the execFunc.

Another simpler way to pass arguments would be to create Public variables which could be accessed from any module.

4.5Recursive functional calls

VBA allows recursive functional calls. This is very elegant, but not necessarily efficient. The function calls itself typically with a new argument where the following example is self-explanatory:

```
Function factorial(n As Integer)
If n = 1 Then
    factorial = 1
Else
    factorial = n * factorial(n - 1)
End If
End Function
```

Figure 14 Factorial function - Recursive functional call

Note that variables declared with a **Dim** statement within a recursive function is created each time the function is called, so several instances of these variables exist. If a variable is declared as **Static** there is only one copy of the variable, and care should be taken in order to not mix things up.

4.6Interaction between VBA and the Worksheets

In some cases VBA is used to simplify worksheet operations. For example to calculate a complicated expression a VBA function is written one times, and could then be used for several purposes. In such situations it is recommended to pass all arguments to the function through the function header, for example **=factorial(B7)** if B7 contains an integer value. We should avoid that the VBA function needs to read more data from the spreadsheet, because then it will not be a general purpose function.

In some situations we would like a VBA function to read data from the spreadsheet. The **Range(<Name>)** function is used. Here **<Name>** is either a cell name, or a name of a range of cells. If a range of cells is to be treated, the **For Each c in Range(<Name>)** is used. For example to load a range of cells from the spreadsheet into an array we use:

```
Dim xValues(1 To 10) as Single
i = 1
For Each c In Range("xValues")
    xValues(i) = c
    i = i + 1
Next c
```

The function should typically return a calculated value:

```
Function SumAB(a as Single, b as Single)
SumAB = a + b
End Function
```

Figure 15 Returning a value from a function

An alternative way to send calculated values back to the spreadsheet is to use the **Range(<Name>)** function:

```
Function SumAB()
Range("AB") = Range("a") + Range("b")
End Function
```

Figure 16 Reading and writing from the spreadsheet

where the values of cells **a** and **b** are added together and put in the cell **AB**. Note that if a function or procedure writes back to the spread sheet in this manner in cannot be called directly from a cell by the **=<functionName>**. In some cases it is appropriate to read date from the spreadsheet, process the data, and then write back. The fastest way to do that is to write a function and then run it from the Immediate window. A more professional way is to create a Button on the appropriate spreadsheet, and assign a function to the button. This requires the **Developer tab** to be available from the Excel Ribbon.

4.7Built in functions

In VBA you could call a set of standard build in functions like sin(), cos(), log(), exp() etc. These functions are available in all VBA settings (Word, Excel, Access etc). One strength of Excel is that a number of Worksheet functions are also available from the VBA code. For example the Normdist() function. However, to use these worksheet functions their name has to be preceded by **`Application.WorksheetFunction.<FuncName>`**. For example we might find the probability that a normally distributed variable is less than 2, when the mean and standard deviation is 0 and 1 respectively by:

```
p = Application.WorksheetFunction.NormDist(2, 0, 1, True)
```

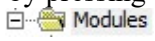
Note that from VBA the worksheet functions are always specified by their English name, whereas from the worksheet you need to specify them by their national language (i.e. depending on you Excel installation).

4.8Importing and exporting modules

Modules with VBA code are specific for each Excel file (workbook). If you create a module in one Excel file it will not be available in another Excel file. There are two ways you may copy VBA code from one file to another. The easiest way is to use the Clipboard to copy code from one Excel file and paste into another Excel file. A more structured way is to establish libraries of modules. To Export a module from Excel do the following:

1. Open the VBA editor by pressing <Alt F11>
2. Right click the module you will export, see Figure 17
3. Click **Export File…**, and select a folder to keep VBA modules

VBA modules are saved with the extension **.bas** and may be opened with a text editor. From another Excel file the VBA code may be imported:

1. Open the VBA editor by pressing <Alt F11>
2. Right click **modules,** 

3. Click **Import File…**, and select a file from the folder keeping VBA modules

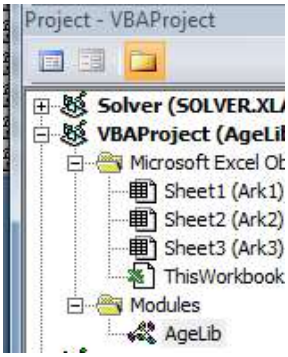


Figure 17 Module library

5.VBA Examples

5.1The effective failure rate in age related models

In age based maintenance models an approximation for the effective failure rate could be specified in Excel by:

```
lambdaE = EXP(GAMMALN(1+1/alpha))/MTTF)^alpha*tau^(alpha-1)
```

where **tau**, **alpha** and **MTTF** are variables defined in cells with corresponding names. This function is often required in maintenance optimization problems, and it would be more convenient to specify a VBA function, and then call this function each time we need to find the effective failure rate:

```
Function LambdaEWApproxSimple(Tau As Single, Alpha As Single, MTTF As
Single)
If Alpha > 1 Then
    LambdaEWApproxSimple = Gamma(1# / Alpha + 1#) ^ Alpha * _
       Tau ^ (Alpha - 1#) / MTTF ^ Alpha
Else
    LambdaEWApproxSimple = 1 / MTTF
End If
End Function
```

Figure 18 First approximation of the effective failure rate

Note that this function makes a call to the Gamma-function which could be implemented by:

```
Function Gamma(x As Single)
Gamma = Exp(Application.WorksheetFunction.GammaLn(x))
End Function
```

Figure 19 The Gamma() function

From the spread sheet in Excel, we may now call the **LambdaEWApproxSimple()** function with the arguments **tau**, **alpha** and **MTTF.**

This approximation function is not very accurate when tau is higher than 10% of the MTTF. A better approximation may be achieved by implementing a correction term.

```
Function LambdaEW(Tau As Single, Alpha As Single, _
    MTTF As Single)
Dim lambda As Single
If MTTF > 0 Then
    If Tau < 0.1 * MTTF Then
        LambdaEW = LambdaEWApproxSimple(Tau, Alpha, MTTF)
    Else
        LambdaEW = LambdaEWApproxSimple(Tau, Alpha, MTTF) * _
          (1# - 0.1 * Alpha * (Tau / MTTF) ^ 2 + _
               (0.09 * Alpha - 0.2) * (Tau / MTTF))
    End If
Else
    LambdaEW = 1E+30
End If
End Function
```

Figure 20 A better approximation to the effective failure rate

Where the correction term ensures reasonable precision whenever **tau** < ½ **MTTF**. An even better implementation would be to use an iterative procedure to find the renewal function whenever when **tau** > ½ **MTTF.**

6.Monte Carlo Simulation

Monte Carlo simulation is a technique to do probability calculus when it is not straight forward to use analytical techniques. The basic idea is that rather than working with the distribution functions, or probability density function of random variables, we work with these variables directly in either the work sheet, or the VBA code. To assign a (random) value to a such variable we use the RAND() function from the worksheet, or the rnd() function from the VBA modules. These two functions return a uniformly distributed variable in the interval from 0 to 1. In the worksheet, we might then define a cell with name Duration, and specify `=RAND()` as the formula for that cell. Excel will then generate a random number. To generate a new number in the cell we ask Excel to do this by pressing the F9 key. If we want to see e.g. the average duration, we could press F9 1000 times and write down the output, and then take the average, of we could press F9 another 1000 times and count the number of times when the Duration cell exceeds 0.9 to find Pr(Duration > 0.9) and so on.

It is, however, rather tedious to do this manually. We could write a VBA code that update the field e.g. 1000 times, and for each time records the number, and then finally take the average of the values that have been generated.

Note that the RAND() function generates a uniformly distributed variable. If you want variables from other distributions you could write your own code.

7.Feedback

Feedback to this document could be emailed to jorn.vatn@ntnu.no

8.Index