1.1 GETTING STARTED

To access the VBA editor, point to the Developer (VBA menu on the ribbon). In case this menu is not visible, proceed like this:

**Office 2007**  Click the Microsoft Office Button then click on Excel Options (bottom right). Point to Popular and mark Show Developer tab in the Ribbon check box, then OK: the Developer tab is now displayed on the ribbon

**Office 2010 and beyond**  Point the File menu and select Options Click on Customize Ribbon, and mark Developer

When you click on Developer, Excel displays a minimum of three *groups*:

- **Code**: this is the group from which you open the VBA editor
- **Controls**: user interface components, to create Windows-like applications
- **XML**: converts XML files into Excel files and vice versa.
From the Code group, click on “Visual basic” icon (far left). If you work on a newly created file, the VBA editor looks like this:

The bar on the top provides three especially useful menus:

- **View**: a menu that lists windows editing information regarding the code execution. Also hides or shows the project explorer (on the left). At creation, the project reduces to three open sheets and an empty **module** named **ThisWorkbook**. As new contents are added in your project, they appear in one of the following items:
  - **Module**
  - **Class Module**
  - **UserForm**
- **Insert**: from this menu, you can add modules to edit your procedures, or class modules if you need to create customized objects. You can also build UserForms from this menu.
- **Debug**: from this menu, you can toggle breakpoints where the execution of the code is interrupted. To track algorithmic mistakes, you can also activate the execution step by step (shortcut F8) and watch the changes in some variables, displayed in the **Immediate window** or **Local variables window** (see View menu).

You cannot write any code on the central gray colored zone: all the procedures must be coded within a module.

The specific **ThisWorkbook** component contains macros that are executed automatically at the opening of the workbook or when special events, such as the updating of one sheet, come up.

### 1.2 VBA objects and syntax

VBA objects are essentially visible pieces of Excel applications. This definition includes sheets, ranges of cells, icons on the ribbon, and custom interfaces.

To handle objects, one must specify their **types**. For instance, a range of cells is given the type **Range**. To locate a Range in the Excel application, we use its physical coordinates on the sheet. For instance the cell “D5” is identified by

\[
\text{Range("D5") or [D5]}
\]
The Basics of VBA Programming

A range of cells (e.g., D5:F8) is identified as

\[ \text{Range}("D5:F8") \text{ or } [D5:F8] \]

Objects of the same type can also be pulled together into a **Collection**, a set of indexed elements. We list below some common collections of objects in VBA:

- **Worksheets**: collection of worksheet-type in a file
- **Sheets**: collection of worksheets + charts + dialog sheets
- **Cells**: collection of cells within a range
- **Workbooks**: collection of Excel application files open at the same time.

How do you access an element of a collection?

- either through the **name** of the element (e.g., `Worksheets("VAT")`)
- or its **index** (e.g., `Worksheets(2): 2nd Excel sheet tab`)

We highly recommend designating the cells of a sheet through their coordinates \((i,j)\). For example

\[ \text{Cells}(i,j) \]

denotes the cell at the intersection of the \(i\)-th row and the \(j\)-th column of the sheet.

You must be aware that charts are members of the sheets collection, just like worksheets: if your project includes charts, you must be aware that the order has changed in the collection.

### 1.2.1 The object-oriented basic syntax

Between objects, there exists some kind of hierarchy, or ties of belonging. By nature, an Excel sheet belongs to the Worksheets collection, and itself contains a collection of cells. The collection of worksheets itself belongs to a Workbook instance.

Therefore, the comprehensive identification of an object should mention its "pedigree". In other words,

\[ \text{Range}("D5") \]

denotes an object that is not completely defined since its location remains vague.

Which worksheet does this cell belong to?

In which workbook (more than one Excel file can be open at the same time)?

To link one object and its parent, we use the **dot** character "." For instance

\[ \text{Workbooks}("Bonds").Worksheets(3).Range("D5") \]

provides a more exhaustive identification of the cell.

Also, cells are somewhat complex objects that have numerous properties: color, borders, values, etc. To access one property of an object, we similarly use the dot character. For instance

\[ \text{Range}("D5").Column \]

denotes the number of column D, i.e., 4.

### 1.2.2 Using objects

When populating an Excel sheet with data and formulas, you are probably used to performing actions such as activating a worksheet, or copying and pasting a range of cells. In VBA, the piece of code needed to copy values displayed in \([A3:D8]\) is, for instance,

\[ \text{Range}("A3:D8").Copy \]
This statement does the work, and no return value is expected. However, things are generally
different when manipulating objects: when you change anything in an object, you generate a
new instance of this object. The modifications operated can involve two kinds of attributes:
**members** and **methods**.

**Members** Members are used to describe an object. For instance, borders, background, or font
type are members of a Range object. These members are themselves objects, but, in contrast
to their **parent**, they are **not visible**, and are thus somehow abstract. Therefore, to access these
members, we again use the dot symbol.

For instance, the **Interior** and **Font** members aim to describe the background aspect and
the font properties of a Range.

To color in red the font of the characters displayed in "A1"

```
Range("A1").Font.Color=vbRed
```

To turn the background color of "C1" blue

```
Range("C1").Interior.Color=vbBlue
```

VBA provides a kind of **code assistant**: A list of relevant properties and actions pops up as you
start coding the ID of an object followed by a dot. For instance, if you start writing

```
Range("A1")
```

the following list will pop up:

- **Activate**
- **AddComment**
- **AddIndent**
- **Address**
- **AddressLocal**
- **AdvancedFilter**
- **AllowEdit**

Members are identified by the icon 🌐.

When you nominate a cell and want to perform some arithmetic operation on its value, you
may omit the member **Value**, for the sake of conciseness, probably about developers’ demand.
The statement

```
Range("A2")=Range("A3")+1
```

is therefore valid, and is equivalent to

```
Range("A2").Value=Range("A3").Value+1
```

**Methods** Methods are actions carried out on objects (iconized by 📢 in the code assistant
relevant list):

- Some do not return any value, such as

```
Range("B3:D8").ClearContents (clears all values in the range B3:D8)
```

```
Range("B3:D8").Select (puts the focus on B3:D8)
```
The Basics of VBA Programming

- Others return instances of the same object type (objects with specific properties): among them, we can quote Offset and End:
  - Offset shifts the location of one cell to another position on the sheet. This method returns a Range-type object.
    \[ \text{Range("B3").offset(n,m)} \]
    [B3] indicating Cells(3,2), Range("B3").offset(n,m) points to Cells(3+n,2+m)
  - End locates the last cell which is not empty in some direction. The arguments of this method are xlDown, xlUp, xlToLeft.
    \[ \text{Range("B3").End(xlDown)} \]
    finds the last non-empty cell in column B, from B3

1.3 VARIABLES

As an experienced Excel user, you have probably already given names to cells or ranges of cells. You can do the same with any type of object in VBA, including RANGE or WORKSHEET.

This section will explore how to handle variables in accordance with the category they belong to, i.e., either basic or Object type. Thereafter, we will address the specific cases of Variant type variables and Constants.

1.3.1 Variable declaration

Basic variables

The VBA developer can ignore matters regarding memory management. All you need to know is that the name of the variable is a reference to the physical location of your variable in the memory, or, in short, its address. One thing that you must be aware of is that the memory allocation depends on the precision required when handling the variables: A decimal value needs more memory than an integer. To mention the variable type at creation is a good practice.

To declare a new variable and specify its type, write

\[ \text{Dim [VariableName] As [VariableType]} \]

Below is a list of some commonly used types:

- Byte
- Integer
- Long (Integer > 2*10^9)
- Single (decimal)
- Double (decimal double precision)
- String (chain of characters)
- String*n (bounded to n characters)
- Date
- Boolean

For instance:

\[ \text{Dim yield As Single} \]
\[ \text{Dim cclientName As String} \]
To make the code more concise, use shortcut notations:

\$
\!\n\%

and group declarations in one single line:

```
Dim vol!, rate!, spot!
Dim client$
```

It is recommended that you initialize the variable (or **instantiate**) when it is created. For instance:

```
Dim vol!: vol = 0.3
Dim client$: client = Sheets("clients").Range("B2")
```

It is possible to code several short statements on the same line, separated by `;`.

**Scope of variables**  When declared inside a procedure, a variable is **local**. For instance the variable `varName` declared as follows:

```
Sub ProcName()
    Dim [varName] As [Type]
End Sub
```

is not visible outside `ProcName`. When the execution of `ProcName` terminates, `varName` is ignored. If you declare it at the top of this module, as in the following example:

```
Dim [VarName] As [Type]
Sub ProcName1()
    [statements]
End Sub
Sub ProcName2()
    [statements]
End Sub
```

`varName` in that case is seen from all the procedures **within the same module**. Dim, alone, is equivalent to **Private**. If you need this variable to be **Global**, i.e., visible to all code in the project, not only one module, add the directive **Public**.

You may omit to specify the type of variable (shame!) when creating it: the default type given by VBA is **Variant**, a kind of catch-all type. It can store any kind of data, except bounded strings. Variant will be introduced shortly after Arrays, since it is more or less related to matrices.

**Object-type variables**  Object variables, unlike data, have different properties, not only values. Declare them with Dim, but initialize them with the directive **Set**. For instance:

```
Dim zoneYields As Range: Set zoneYields = Sheets("yields").Range("B2:B11")
```

You can also declare several object variables on one single line:

```
Dim zoneYields, zoneVols, zoneSpots As Range
```

**Constants**  Unlike variables, constants store values that cannot be changed in the course of the program. The objective is obvious when considering the number PI, for example: an explicit Id, such as **PI**, is surely more concise and explicit than 3.14159265. They are declared such that
The Basics of VBA Programming

Const PI=3.14159265
Const VAT=0.205

Besides custom constants created by the developer, VBA provides native constants: they are passed as arguments of VBA/Excel functions, generally values of type Long. VBA assigns explicit names to these values.

In the following examples:

[B5].Interior.Color=vbRed
[B5].end(xlDown).Select

the hidden values of vbRed and xlDown are vbRed=255 and xlDown=-4121

**The Variant type**  By default, the Variant type is assigned to any variable not declared explicitly. In practice, you can store any kind of objects in a variable declared as Variant, even an array of values (see §1.3.3 below).

The declaration statement is similar to other basic types, i.e.,

```
Dim v As Variant
```

A Variant type variable can even be assigned a Range-type object, which is convenient when you need to use VBA functions (in fact, only Range-type values can be passed as arguments to VBA functions).

For instance, this declaration:

```
Dim w As Variant:w = [B3:B8]
```

is correct.

### 1.3.2 Some usual objects

Some objects are unavoidable: Worksheet and Range. In this section, we outline the way they are commonly handled.

**Worksheet**  For instance, to initialize a worksheet named ”wk” and assign it the 2nd sheet of the active Excel Workbook, write:

```
Dim wk As Worksheet: Set wk = ActiveWorkbook.Worksheets(2)
```

You can change dynamically (although this is not recommended) the name of the worksheet that appears at the bottom of your Excel workbook:

```
ActiveWorkbook.Worksheets(2).Name = "Bonds"
```

Once a worksheet is activated, you can refer to it using ActiveSheet:

```
ActiveSheet.Columns(1).Interior.Color = vbBlue
```

**Worksheet collections**  The Worksheet object is a member of the Worksheets collection, that itself contains a collection of child objects:

- Cells
- Comments (all the comments in a sheet)
- ChartObjects (charts in a given sheet).
Some current members and methods

Members:

- Name
- Visible
- Password

Methods:

- Activate (select the whole sheet or the top right cell of a range)
- Copy
- Paste
- Protect

Range  Range type may refer to:

- one cell
- a range of cells
- a collection of several ranges of cells.

For instance, if you want to create a range named “SwapRates” containing

- a range of maturities displayed on [A3:A12]
- a list of currencies on [B1:F1]
- some swap rates on [B3:F12]

the relevant code will look like this:

Dim SwapRates As Range
Set SwapRates = Range(“A3:A12, B1:F1, B3:F12”)  
SwapRates.Select

When selecting SwapRates, pay attention that Selection is to designate [A1] alone, although the whole range is highlighted:
The Basics of VBA Programming

Range collections  A Range type object is the parent of four collections:

- Cells, of course
- Columns
- Rows
- Areas.

In our example

```
Areas(1) = [A3:A12]
Areas(2) = [B1:F1]
Areas(3) = [F3:F12]
```

Actually, the number of rows and columns in SwapRates are those of the first element in the collection, i.e., Areas(1) (as an illustration, SwapRates.Rows.Count = Areas(1).Rows.Count).

Some current members and methods

Members:

- Interior
- Font
- Borders
- Formula
- Count (provides the number of cells inside a Range).

Methods:

- Copy
- Paste
- Activate
- Resize
- Offset
- End.

1.3.3 Arrays

An array is a set of objects of the same type, ranked with a numerical index. The size of an array can be:

1. fixed once and for all when it is declared, or
2. omitted.

In any case, the objects’ type must be declared.

1. When the size of the array has been specified at creation, such as in the example below:

```
Dim tabYields(10) As Single 'or
Dim tabYields!(10)
```

it cannot be resized in the course of the program.
10 HOW TO IMPLEMENT MARKET MODELS USING VBA

If omitted, the size of an array can be modified dynamically: use `ReDim` to modify it:

```vba
Dim tabYields!()
Dim n1%; n1=10
ReDim tabYields(n1)
Dim n2%; n2=15
ReDim tabYields(n2)
```

Before populating an array, it is imperative that you redimension it (`ReDim` directive): if not, the VBA error message emitted is unfortunately insufficiently explicit.

When using the `ReDim` statement, all the elements of the array are erased. To avoid this, add `Preserve` to the `ReDim` statement.

```vba
Dim tabYields!()
ReDim tabYields(n1)
ReDim Preserve tabYields(n2) ' the n1 first elements remain if n2>n1
```

The lower and upper bound of an array are given by:

```vba
LBound([Name of the array])
UBound([Name of the array])
```

By default, the number 0 is the first element in an array. You can force the lower and upper bound in the array declaration:

```vba
Dim n1%, n2%
n1 = 10: n2 = 15
ReDim tb!(n1 To n2) ' first element is number 10
```

You can also set the lower bound of every array in a module to 1: type:

```vba
Option Base 1
```

on top of the module, outside and before any procedure.

NB: Omitting the size of an array at creation is almost automatic since the settings of financial applications need to be dynamically adjusted, as a general rule.

An array can have more than one dimension: a two-dimensional array is called a matrix.

For instance, to declare a dynamically resizeable matrix, write:

```vba
Dim MatrixName!()
[Statements]
ReDim MatrixName(n,n)
```

Another example: to declare a three-dimensional fixed-size array where the first element of the second coordinate is indexed by 1:

```vba
Dim MatrixName!(5,1 to 10,10)
```

1.4 ARITHMETIC

Math operations Numerical operations are standard. The code for

\[ M \times (1 + r)^n \]

is

\[ M^* (1+r)^n \]
Attention must be paid to precedences. Brackets have to be added to avoid miscalculations. For instance,

\[
\frac{M}{1+r_1)/(1+r_2) \text{ is equivalent to } M/(1+r_1)*(1+r_2)
\]

while

\[
\frac{M}{1+r_1)\ast(1+r_2) \text{ equals } (M/(1+r_1))*\ast(1+r_2)
\]

Some operators are specific to integer operands:

\[
7\!\div 3 \text{ ' returns the integer result of the division i.e. 2 }
7 \text{ Mod 3 ' returns the integer remainder of the division i.e. 1 }
\]

**Comparison operators**

\[
=, \geq, <, \leq, =\lt, \neq \text{ (not equal to)}
\]

These operators compare two data and return a boolean value (True or False). For instance,

\[
\text{Dim bool As Boolean: bool=(45<3) ' bool=False}
\]

The comparison operators are widely used in conditional statements (see below).

**Binary logical operators**

Binary logical operators also perform comparisons, but uniquely between boolean values:

\[
\text{And}, \text{ Or}, \text{ Xor}
\]

\[
X \text{ And Y ' returns True if and only if X and Y are true}
4>3 \text{ And 8>9 = False}
X \text{ Or Y ' returns True if one at least of the expressions is true}
4>3 \text{ Or 8>9 = True}
X \text{ Xor Y ' returns True if one exactly of the expressions is true}
4>3 \text{ Xor 8>9 = True}
4>3 \text{ Xor 9>8 = False}
\]

**Unary logical operator**

This operator takes one single operand:

\[
\text{Not}
\]

\[
\text{Not X ' returns False if X is true,}
\text{In that case, Not(Not X) returns True}
\]

\[
\text{Not (Not 3 > 2) ' returns True}
\]

**Conditional statements**

\[
\text{if...then}
\]

The if...then...(else) statement evaluates a condition. If this condition is met, some code is executed. Alternative actions can be carried out if the condition is not met:

\[
\text{If [Condition] Then}
\text{[statements]}
\text{End If}
\]

Additional conditions may be inserted:

\[
\text{Dim TaxInc As Boolean}
\text{Dim Total!, DutyFree!, VAT!}
\]
If TaxInc Then
    Total = DutyFree
ElseIf VAT = 7.5 / 100 Then
    Total = (1 + VAT) * DutyFree
Else
    Total = (1 + 12.5 / 100) * DutyFree
End If

Select Case...
The Select...Case statement can run several blocks of statements, contingent on the value taken by one single variable:

Select Case Variable
    Case Value1
        [statements if Variable=Value1]
    Case Value2
        [statements if Variable=Value2]
    .
    .
    Case Else
        [Statements if Value does not fit conditions above]
End Select

Intervals may also be evaluated:

Dim AgeClass$, Age%
Select Case Age
    Case 1
        AgeClass = "baby"
    Case 2, 3
        AgeClass = "SmallInfant"
    Case 4 To 7
        AgeClass = "Infant"
    Case 8 To 17
        AgeClass = "Young"
    Case 18 To 64
        AgeClass = "Adult"
    Case Is >= 65
        AgeClass = "Senior"
End Select

Loops
A loop is a block of statements aimed at being executed repeatedly. The number of iterations can be predetermined or contingent upon the evaluation of one or more conditions. Three situations may come about:

- the number of iterations N is known:
  For k=1 to N
  [code]
  .
  Next k
  ' the code is executed N times
The Basics of VBA Programming

- the statements are repeated as many times as the number of elements in a collection. As an example, dealing with a collection of cells:
  
  For each Cell in [Name of the collection]
  [code]
  ...
  Next Cell

- the following block repeats an action as long as a condition (boolean value) is met:
  
  While [condition]
  [code]
  ...
  Wend

  or the block repeats an action until a condition is met

  Do
  [code]
  ...
  Loop Until (While) [condition]

  Do While...Loop allows you to insert the Exit Do statement, in case the sequence of iterations needs to be interrupted: For instance:

  Dim d As Date, n%
  d=CDate("01/01/2020")
  Do
    n=n+1
    if d>Date() then exit do
  Loop Until DateAdd("yyyy",d,-n)-Date()>0

Most of the time, different loops can achieve the same task: the more concise the better.

1.5 SUBROUTINES AND FUNCTIONS

Subroutines and functions are both procedures, i.e., blocks of code that complete a specific task. In addition, functions return a result while subroutines don’t. The following issues deserve some consideration:

- the scope of the procedure. When a subroutine or a function is created, you must decide in which module this procedure is supposed to be run (main procedure), or called from another procedure. For instance, the following statements

  Sub ProcName([Arg]), etc.
  
or
  Private ProcName([Arg])

  create procedures only visible in the module in which they are edited. To be called from other modules of the active workbook, add a Public directive:

  Public ProcName([Arg])
the **passing mechanism** Unless otherwise specified, arguments are normally passed to subroutines *by reference* (syntax **ByRef**):

Sub ProcName(Arg)

is equivalent to

Sub ProcName( ByRef Arg)

When an argument is passed by reference, any change made in its value within the called procedure remains effective outside this procedure. Reference here designates equally the address and the data registered at this address.

When an argument is passed by value (**ByVal**), a *copy* of the variable is made and passed to the procedure, so that any change in the copy will not impact the value of the passed variable.

The example which follows will underline the difference: given the procedures `invSingle1` and `invSingle2` performing both the inversion of Single type variables,

```vba
Sub invSingle1(s!)
    s = -s
End Sub
Sub invSingle2(ByVal s!)
    s = -s
End Sub
```

Running the procedure `testPass` below

```vba
Sub testPass()
    Dim x!: x = 1
    Dim y!: y = 1
    Call invSingle1(x)
    Debug.Print x
    Call invSingle2(y)
    Debug.Print y
End Sub
```

we get (in **immediate window**)

-1
1

VBA does not allow us to pass arrays by value.

### 1.5.1 Subroutines

To create a subroutine, just type

`Sub [Name of the procedure]`

VBA completes the code, adding

`End Sub`

For instance, a procedure that scans through a range of cells in sheets(1), and fills the empty ones with the next cell up, could look like this:

```vba
Sub CheckData()
    Sheets(1).Activate
    Dim zoneData As Range: Set zoneData = Range([B3], [B3].end(xlDown))
```
The Basics of VBA Programming

For Each cell in zoneData
if isEmpty(Cell) Then Cell=Cell.offset(-1,0)
Next Cell
End Sub

Subroutines may accept arguments: in that case, these arguments may be regarded as parameters of the procedure. For instance, in the code below
Sub RetrieveData(Market As String)
Worksheets(Market).Activate
End Sub

‘Market’ denotes the name of the sheet where the bulk of data to be retrieved (and dealing with Liffe exchange) is stored. We guess that the same block of statements can apply to other markets (different sheets). To call a subroutine from an external procedure:

■ use Call if some arguments are to be passed, or
■ just write the name of the subroutine should this not be the case.

In the example below, a main program calls successively RetrieveData and another procedure OtherProc taking no argument:
Sub MainProc()
Call RetrieveData("Liffe")
OtherProc
End Sub

NB: Arguments of a subroutine need not to be enclosed in round brackets.
It is highly recommended to mention not only the type of argument, but also the return type. Obviously, the type of argument must not be omitted.

1.5.2 Functions
To create a function, use the keyword Function and specify the type of the returned value:

Function [FunctionName](list of arguments) As [Type of returned value]
[Statements]
FunctionName= value to be returned
End Sub

For the function to return a value, add the statement

FunctionName = result to return

The value returned can be a single value or an array of values: In the second case, add parentheses to the type declaration.
Here is an example of a daily changes calculation. The argument is an array of market quotes, the value returned is an array of daily variations.

Function DailyVar(Quotes!()) As Single()
ReDim t!(UBound(Quotes))
For i = 1 To UBound(t)
    t(i) = (Quotes(i) - Quotes(i - 1)) / Quotes(i - 1)
Next i
DailyVar = t
End Function
1.5.3 Operations on one-dimensional arrays

In this section, we display some examples of procedures where arrays are passed as arguments:

■ function deleting one element in a list:

```vba
Public Function DelFromList(t!(), e!) As Single()
    ReDim result!(UBound(t) - 1)
    i = LBound(t)
    While t(i) <> e
        result(i) = t(i)
        i = i + 1
    Wend
    For j = i To UBound(t) - 1
        result(j) = t(j + 1)
    Next j
    DelFromList = result
End Function
```

■ function merging two sorted arrays of single values:

```vba
Function Merge2List(a!(), b!) As Single()
    Dim la%, ua%, lb%, ub%
    la = LBound(a): ua = UBound(a): lb = LBound(b): ub = UBound(b)
    ReDim t!(ua - la + ub - lb + 1)
    Dim i%, j%, k%
    i = la: j = lb: k = 0
    While i < ua And j < ub
        If (a(i) < b(j)) Then
            t(k) = a(i): i = i + 1
        Else
            t(k) = b(j): j = j + 1
        End If
        k = k + 1
    Wend
    While (i <= ua)
        t(k) = a(i): i = i + 1: k = k + 1
    Wend
    While (j <= ub)
        t(k) = b(j): j = j + 1: k = k + 1
    Wend
    Merge2List = t
End Function
```

1.5.4 Operations on two-dimensional arrays (matrices)

First of all, let us point out that Excel can perform some basic matrix operations: multiplication, transpose, inversion, etc. So can VBA proprietary functions.
However, these functions reject two-dimensionnal arrays of numerical values for performing the product of two matrices. As a consequence, working with Excel/VBA proprietary functions binds you to process solely data stored in an Excel sheet. This is definitely not compatible with standard programming rules.

Nevertheless, since you may be concerned to check your self-developed functions (e.g., inverting a matrix, or calculating a pseudo-square inverse), it is useful to become familiar with Excel functionalities.

**The proprietary Excel function**  Let us consider two ranges of cells named A and B to be multiplied:

1. Select with the mouse the area where you want your solution to be located in the Excel sheet and choose the relevant function in the Excel drop-down list, here **MMULT**: first, ensure that
   
   Number of columns (A) = Number of rows (B)

   The format of the selected range C must be (given C = A.B)

   Number of rows (A) * Number of columns (B)

2. Provide ranges A and B as inputs of the function MMULT in the pop-up dialog box
3. Press Ctrl + Shift + Enter: if you click on "ok", the result is the top left element of the matrix alone.

**VBA customized functions**  Some of Excel functions have their equivalent in VBA. More precisely, these functions are *methods* of a wide VBA object called **WorksheetFunction**. Therefore, you must mention the parent object when using one of these functions, as in the example below:

```vba
Dim A, B, C As Variant
A = Range("B25:C26"): B = Range("E25:E26")
C = WorksheetFunction.MMult(A, B)
```

WorksheetFunction is itself a member of **Application**, the patriarch of all existing VBA objects.

**Multiplication**  The following code shows how to multiply two arrays of numerical values (of type Single in our example). First of all, we have to check the dimensions of the arguments, which is achieved by **Mdim(m#())**. As a matter of fact, VBA functions do not perform dot products, for instance.

```vba
Public Function Mdim(m#()) As Integer
On Error GoTo Dimension

    For DimNum = 1 To 3
        ErrorCheck = LBound(m, DimNum)
    Next DimNum

Dimension:
Mdim = DimNum - 1

End Function
```
Next, the function **MatMult**\( (m1\#(), m2\#()) \) performs the appropriate multiplication, taking the dimensions of \( m1 \) and \( m2 \) into account:

\[
\text{Public Function MatMult}(m1\#(), m2\#()) \text{ As Double()}
\]

\[
\text{Dim dim1\%, dim2\%}
\text{dim1 = Mdim(m1); dim2 = Mdim(m2)}
\text{Dim res\#()}
\text{Select Case dim1 * dim2}
\text{Case 2}
\text{If dim1 = 2 Then}
\text{ReDim res(UBound(m1, 1))}
\text{For i = 1 To UBound(m1, 1)}
\text{res(i) = 0}
\text{For K = 1 To UBound(m1, 2)}
\text{res(i) = res(i) + m1(i, K) * m2(K)}
\text{Next K}
\text{Next i}
\text{Else}
\text{ReDim res(UBound(m2, 2), UBound(m2, 2))}
\text{For i = 1 To UBound(m1, 1)}
\text{For j = 1 To UBound(m2, 2)}
\text{res(i, j) = m1(i) * m2(1, j)}
\text{Next j}
\text{Next i}
\text{End If}
\text{Case 4}
\text{ReDim res(UBound(m1, 1), UBound(m2, 2))}
\text{For i = 1 To UBound(m1, 1)}
\text{For j = 1 To UBound(m2, 2)}
\text{res(i, j) = 0}
\text{For K = 1 To UBound(m1, 2)}
\text{res(i, j) = res(i, j) + m1(i, K) * m2(K, j)}
\text{Next K}
\text{Next j}
\text{Next i}
\text{End Select}
\text{MatMult = res}
\text{End Function}

**Transposition of a matrix**

\[
\text{Public Function matTranspose(t\#()) \text{ As Double()}}
\]

\[
\text{ReDim res\#(UBound(t, 1), UBound(t, 2))}
\text{For i = 1 To UBound(t, 1)}
\text{For j = 1 To UBound(t, 2)}
\text{res(i, j) = t(j, i)}
\text{Next}
\]
1.5.5 Operations with dates

**Date** type variables are created just like basic variables. The reference *Date* points to today’s date:

```vba
Dim Today As Date: Today = Date
```

Setting payment schedules is a permanent concern in financial activities. Here are some functions specifically devoted to these tasks:

- **DateAdd**
- **DateDiff**
- **Weekday().**

**DateAdd** adds a number of periods (days, months, quarters...) to a date passed as argument and returns the resulting date. The first argument is a string value figuring the period. For instance, to add 6 months to today’s date:

```vba
Dim TodayAnd6M As Date: TodayAnd6M = DateAdd("m", 6, Date)
```

<table>
<thead>
<tr>
<th>Value</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>yyyy</td>
<td>year</td>
</tr>
<tr>
<td>q</td>
<td>quarter</td>
</tr>
<tr>
<td>m</td>
<td>month</td>
</tr>
<tr>
<td>d</td>
<td>day</td>
</tr>
<tr>
<td>ww</td>
<td>week</td>
</tr>
</tbody>
</table>

**DateDiff** returns the number of time periods between two dates. The periods are those of the DateAdd function:

```vba
Dim nb_of_months%: nb_of_months = DateDiff("m", Date, TodayAnd6M) ' returns 6
```

**Weekday** returns an integer representing the day of the week. By default, the first day of the week is Sunday:

```vba
Dim DayOfWeek%
DayOfWeek = Weekday("01/01/2012") ' returns 1 for Sunday
```

To modify the convention, use the optional argument **FirstDayOfWeek** in the function definition:

```vba
Weekday(ArgDate, Optional [FirstDayOfWeek])
```

```vba
Weekday("01/01/2012", vbMonday) ' returns 7
```

The function **PayDates** below sets up a cash payments schedule, given

- the expiry date **tenor**
- the periodicity of the cash flows period in number of months ("1m","2m",...)
Prior to the setting of the schedule, you must merge the Target Holiday Calendar and the week-end days in order to determine the first business day following:

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target holidays</td>
</tr>
<tr>
<td>12/25/2012</td>
</tr>
<tr>
<td>12/26/2012</td>
</tr>
<tr>
<td>1/1/2013</td>
</tr>
<tr>
<td>3/29/2013</td>
</tr>
<tr>
<td>4/1/2013</td>
</tr>
<tr>
<td>5/1/2013</td>
</tr>
<tr>
<td>12/25/2013</td>
</tr>
<tr>
<td>12/26/2013</td>
</tr>
<tr>
<td>1/1/2014</td>
</tr>
<tr>
<td>4/18/2014</td>
</tr>
<tr>
<td>4/21/2014</td>
</tr>
<tr>
<td>5/1/2014</td>
</tr>
<tr>
<td>12/25/2014</td>
</tr>
<tr>
<td>12/26/2014</td>
</tr>
<tr>
<td>1/1/2015</td>
</tr>
<tr>
<td>4/3/2015</td>
</tr>
<tr>
<td>4/6/2015</td>
</tr>
<tr>
<td>5/1/2015</td>
</tr>
<tr>
<td>12/25/2015</td>
</tr>
<tr>
<td>12/26/2015</td>
</tr>
<tr>
<td>1/1/2016</td>
</tr>
<tr>
<td>3/25/2016</td>
</tr>
<tr>
<td>3/28/2016</td>
</tr>
<tr>
<td>5/1/2016</td>
</tr>
<tr>
<td>12/25/2016</td>
</tr>
<tr>
<td>12/26/2016</td>
</tr>
<tr>
<td>1/1/2017</td>
</tr>
</tbody>
</table>

Public Function PayDates(tenor As Date, period$) As Date()
Dim d As Date: d = tenor
Dim n As Long: n = 0
Dim i As Long

Dim tempDates() As Date

Dim f%: f = CInt(Left(period, Len(period) - 1))
Dim m%: m = Fix(DateDiff("m", Date, tenor) / f)
ReDim tempDates(m) As Date

While d > Date
    tempDates(m - n) = d
    n = n + 1
    d = DateAdd("m", -f * n, tenor)
Wend
The Basics of VBA Programming

ReDim targetDays(Range([A3], [A3].End(xlDown)).Count - 1) As Date

i = 0
For Each cell In Range([A3], [A3].End(xlDown))
    targetDays(i) = cell
    i = i + 1
Next cell

ReDim WEdays((tenor - Date) / 3) As Date ' size by default 1/3 > 2/7

Dim temp As Date: temp = Date
i = 0
While temp < tenor
    temp = temp + 1
    If Weekday(temp) = 7 Or Weekday(temp) = 1 Then WEdays(i) = temp: i = i + 1
Wend

ReDim Preserve WEdays(i)

Dim holidays() As Date
holidays = MergeDateList(targetDays, WEdays)

For i = 0 To m
    While testList(tempDates(i), holidays)
        tempDates(i) = tempDates(i) + 1
    Wend
Next i

PayDates = tempDates
End Function

1.6 CUSTOM OBJECTS

1.6.1 Types

A type is a user-defined type of structure that stores closely associated data together in a single object. For instance, if you intend to price instruments like bonds that have more than one term in common, such as maturity, nominal, coupon, or frequency of interest payments, you should be interested in creating a custom object named, e.g., bond and declare it as follows (you may notice the absence of Dim statement):

Type Bond
    Nominal As Single
    Market As String
    FixedRate As Single
    Period As String
    Maturity As Date
End Type

To create a variable named Tbond1 of type Bond, write:

Dim Tbond1 As Bond
To access an element of a Bond variable, you must relate to the name of the parent variable: in our example, to place 1000000 in Nominal data, write

```
Tbond1.Nominal=1000000
```

You can also declare arrays into your Type definition and even nest other custom types into it. For instance, suppose that:

1. You have created a custom type Nominal, with two elements, the amount and the currency name:

   ```
   Type Nominal
   Amount As Single
   Market As String
   End Type
   ```

2. Then, you can declare an interest payments schedule CoupDates, more detailed than a unique maturity in the body of the type definition: your Bond definition may become:

   ```
   Type Bond
   Capital As Nominal
   FixedRate As Single
   Period As String
   CoupDates() As Date
   End Type
   ```

### 1.6.2 Classes

Types are custom objects with limited abilities. Since they handle no custom methods and provide no control over what values are allotted to their elements, they are like a static data structure. For instance, you would be interested to verify that a cash payments schedule complies with the target calendar: this has to be done in an outside procedure. In this section, we introduce **class modules** where you can edit the definitions of new objects that are somewhat similar to Java or C++ classes.

**Defining a new class**  From the **Insert** menu in the VBA editor, add a new class module to your project. Change the default name in the properties window into something more explicit. As an example, let us create a new class, named Bond:

```
Dim Nominal As Single
Dim Market As String
Dim FixedRate As Single
Dim Period As String
Dim Maturity As Date
```
By default, the class is declared **Private**, thus accessible only in the project where it has been defined. To use this class in another project, we will select **PublicNotCreatable**. To create a new instance of the Bond class, i.e., define a variable called **BOAapr2017** for instance, just type

```
Dim BOAapr2017 As New Bond
```

You may want to make the elements available throughout the project and also give some of them default values when an instance is created:

1. Declare the elements as **Public**
2. Hardcode the default values in the **Class_Initialize()** procedure. The definition of your class in the editor looks like this:

```vba
Class

Public Nominal As Single
Public Market As String
Public FixedRate As Single
Public Period As String
Public Maturity As Date

Private Sub Class_Initialize()

Nominal = 100
Market = "GBF"

End Sub

End Class
```

To handle the values set or retrieved from an object, we use **Property** procedures:

1. Get Property to retrieve an element
2. Let Property to initialize an element
3. Set Property to assign a reference to an object.

Some code can be embedded into those properties, allowing appropriate actions to be taken when assigning values to the object elements, such as raising an error if values are invalid. The piece of code below creates a new Bond object named **oblig** and uses the Let Property to instantiate oblig. Maturity while checking the date relevance:

```vba
Public Property Get Expiry() As Date
    Expiry = Maturity
End Property

Public Property Let Expiry(tenor As Date)

ReDim target(Range([A3], [A3].End(xlDown)).Count - 1) As Date
i = 0
```
For Each cell In Range([A3], [A3].End(xlDown))
    target(i) = cell
    i = i + 1
Next cell
If testList(tenor, target) Then
   MsgBox ("invalid maturity: not a business day")
Else
   Maturity = tenor
End If
End Property

1.7 DEBUGGING

1.7.1 Error handling

In this section, we will focus on runtime errors, and discard banal omissions such as "for without next" or variable declaration omissions. A runtime error occurs when a statement of the program cannot be executed properly. The origins of such interruptions are usually:

- a division by zero
- an overflow error (e.g., when an algorithm diverges)
- a call to an element of an array whose index is beyond the length of that array
- a call to a local variable or procedure out of the module where it has been declared.

When it happens, the program stops and the faulty statement is highlighted. Moving the cursor over the edited code shows you the current value of variables at the time of error. If you wish the execution to continue normally, then type On Error.

On Error Resume Next

When an error is encountered, VBA resumes the execution on the next line. An error number is recorded as a value of the Err object.

After testing Err.Number, an appropriate action can be taken:

On Error Resume Next
...  
  x=Sqrt(y)  ' runtime error when y<0
  If Err.Number <> 11 Then
     y=-y
  End If

On Error Goto <line label>:  The execution is transferred to the line following the specified line label, ignoring all the lines in between.

Dim price!
On Error GoTo DefaultVol:
   vol = sqrt(Var)  ' can cause an error
   price=' some code involving vol'
DefaultVol:
   vol = 0.3  ' the default value in case var<0
Resume Next
1.7.2 Tracking the code execution

Activate **Debug** on the Menu bar. VBA displays four groups of items:

<table>
<thead>
<tr>
<th>Immediate Window</th>
<th>F7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Window</td>
<td>Shift+F7</td>
</tr>
<tr>
<td>Definition</td>
<td>Shift+F2</td>
</tr>
<tr>
<td>Last Position</td>
<td>Ctrl+Shift+F2</td>
</tr>
<tr>
<td>Object Browser</td>
<td>F2</td>
</tr>
<tr>
<td>Immediate Window</td>
<td>Ctrl+G</td>
</tr>
<tr>
<td>Locals Window</td>
<td>Ctrl+Shift+F3</td>
</tr>
<tr>
<td>Watch Window</td>
<td>Ctrl+L</td>
</tr>
<tr>
<td>Call Stack</td>
<td>Ctrl+L</td>
</tr>
</tbody>
</table>

These items can be activated in conjunction with windows that aim to change some lines of code at the very moment of runtime or display information in the course of the program execution:

**Immediate window**  You can use this window to display the value of some key variables at some point in the program execution. Type

```
Debug.Print <Name of variable>
```

You can also use this window to add commands when the execution is in break mode. To place a break point on a line, put the cursor on that line and press F9 or choose "Toggle Breakpoint":
the line appears with a brick background. You cannot place a breakpoint on variable declaration lines (you can if you redimension one array). In the course of runtime, VBA pauses immediately before processing this line, which is then highlighted in yellow.

To illustrate this, the small piece of code below calculates the variance of a series of data, which is displayed in the immediate window, just before the breakpoint:

```vbnet
Sub test()

    Dim n% 
    Dim zoneVal As Range: Set zoneVal = Range([A1], [A1].End(xlDown))
    n = zoneVal.Count
    ReDim v!(n)
    variance = 0

    For i = 1 To n
        v(i) = zoneVal(i) 
        variance = variance + v(i) ^ 2
    Next i

    vol = Sqr(variance / n) * 16 
    Debug.Print vol

End Sub
```

In break mode, we type

```
n=n-1
```

inside the immediate window, following the value of the variance. To resume the code execution, we can press F5 or choose "Continue" to complete the runtime from the Run menu, or stepping through the code line by line (see below): the change in the value of n will be taken into account!

```vbnet
Sub test()

    Dim n% 
    Dim zoneVal As Range: Set zoneVal = Range([A1], [A1].End(xlDown))
    n = zoneVal.Count
    ReDim v!(n)
    variance = 0

    For i = 1 To n
        v(i) = zoneVal(i) 
        variance = variance + v(i) ^ 2
    Next i

    vol = Sqr(variance / n) * 16 
    Debug.Print vol

    .vol = Sqr(variance / n) * 16 
    Debug.Print vol

End Sub
```
Stepping through the code  From the beginning or after a breakpoint, it is possible to step through the program line by line. You can activate this by pressing the F8 key from the start of the procedure execution, or from a toggle breakpoint. Pressing F8 causes VBA to execute each line one at a time, highlighting the next line of code in yellow.

If your (main) procedure calls another one, pressing F8 will cause VBA to step inside the called procedure and step through it line by line. You can use SHIFT+F8 to “Step Over” the procedure call, for the sake of time saving (for instance, if this external procedure populates a big matrix).

Locals Window  When running a step-by-step execution of a procedure, it is very helpful to follow how the variables change in the course of the program execution. For this purpose, you can display the Locals Window by choosing it from the View menu. The Locals Window displays the local variables in a procedure and their values. To see the values taken by Global variables, you must:

- either display them with the Debug.Print statement into the immediate window
- or move the cursor over the variable you want to check.