MS Excel and VBA

Module 3: Visual Basic for Applications (VBA)

Bruno Abreu Calfa

Last Update: April 4, 2012
1 Motivation
2 Recording Macros
3 VBA Programming: Basics
4 Manipulating Objects and Collections
5 Controlling Code Execution
6 Looping Blocks of Instructions
7 Working with Spreadsheets
   Working with Ranges
   Working with Cells
8 Applications
   Calling MATLAB from VBA
Outline

1. Motivation
2. Recording Macros
3. VBA Programming: Basics
4. Manipulating Objects and Collections
5. Controlling Code Execution
6. Looping Blocks of Instructions
7. Working with Spreadsheets
   Working with Ranges
   Working with Cells
8. Applications
   Calling MATLAB from VBA
Why VBA?

- Visual Basic for Applications (VBA) is a programming language available in some of the MS Office packages
Why VBA?

- Visual Basic for Applications (VBA) is a programming language available in some of the MS Office packages.
- VBA enables building user-defined functions, automating processes, and accessing Windows API and other low-level functionality through dynamic-link libraries (DLLs).
Why VBA?

- Visual Basic for Applications (VBA) is a programming language available in some of the MS Office packages.
- VBA enables building user-defined functions, automating processes, and accessing Windows API and other low-level functionality through dynamic-link libraries (DLLs).
- The reason to program with VBA is to make some task easier or more reliable.
Why VBA?

- Visual Basic for Applications (VBA) is a programming language available in some of the MS Office packages.
- VBA enables building user-defined functions, automating processes, and accessing Windows API and other low-level functionality through dynamic-link libraries (DLLs).
- The reason to program with VBA is to make some task easier or more reliable.
- Programming languages make things easier because they are great at performing repetitive operations and following a logical path without getting tired or bored.
Why VBA?

- Visual Basic for Applications (VBA) is a programming language available in some of the MS Office packages.
- VBA enables building user-defined functions, automating processes, and accessing Windows API and other low-level functionality through dynamic-link libraries (DLLs).
- The reason to program with VBA is to make some task easier or more reliable.
- Programming languages make things easier because they are great at performing repetitive operations and following a logical path without getting tired or bored.
- They make things more reliable because they slavishly follow your directions and never, ever get creative.
Outline

1 Motivation

2 Recording Macros

3 VBA Programming: Basics

4 Manipulating Objects and Collections

5 Controlling Code Execution

6 Looping Blocks of Instructions

7 Working with Spreadsheets
   Working with Ranges
   Working with Cells

8 Applications
   Calling MATLAB from VBA
Getting Started

- Macros are pieces of VBA code
Getting Started

- Macros are pieces of VBA code
- Be sure to enable the ‘Developer’ tab in the ribbon
Getting Started

- Macros are pieces of VBA code
- Be sure to enable the ‘Developer’ tab in the ribbon
- Go to: File → Options → Customize Ribbon and select the Developer tab
Automatic VBA Code Generation

- One of the ways of learning how to perform certain tasks in VBA is to “record a macro”
Automatic VBA Code Generation

- One of the ways of learning how to perform certain tasks in VBA is to “record a macro”
- After you are done recording it, you can see what commands correspond to your actions
Automatic VBA Code Generation

- One of the ways of learning how to perform certain tasks in VBA is to “record a macro”
- After you are done recording it, you can see what commands correspond to your actions
- From the Developer tab, click on Record Macro (you can customize its name if you want) and hit OK
Automatic VBA Code Generation

- One of the ways of learning how to perform certain tasks in VBA is to “record a macro”
- After you are done recording it, you can see what commands correspond to your actions
- From the Developer tab, click on Record Macro (you can customize its name if you want) and hit OK
- Do some tasks and then click on Stop Recording
Automatic VBA Code Generation

- One of the ways of learning how to perform certain tasks in VBA is to “record a macro”
- After you are done recording it, you can see what commands correspond to your actions
- From the Developer tab, click on Record Macro (you can customize its name if you want) and hit OK
- Do some tasks and then click on Stop Recording
- To run/edit the macro, click on Macros and click on the respective button
Automatic VBA Code Generation

• One of the ways of learning how to perform certain tasks in VBA is to “record a macro”

• After you are done recording it, you can see what commands correspond to your actions

• From the Developer tab, click on Record Macro (you can customize its name if you want) and hit OK

• Do some tasks and then click on Stop Recording

• To run/edit the macro, click on Macros and click on the respective button

• See file VBA_Examples.xlsm, worksheet “Recording”
Outline

1. Motivation
2. Recording Macros
3. VBA Programming: Basics
   4. Manipulating Objects and Collections
   5. Controlling Code Execution
   6. Looping Blocks of Instructions
   7. Working with Spreadsheets
      - Working with Ranges
      - Working with Cells
   8. Applications
      - Calling MATLAB from VBA
Definitions and Terminology (I)

- **Code:** You perform actions in VBA by executing VBA code. You write (or record) VBA code, which is stored in a VBA module.
Definitions and Terminology (I)

- **Code**: You perform actions in VBA by executing VBA code. You write (or record) VBA code, which is stored in a VBA module.

- **Module**: VBA modules are stored in an Excel workbook file, but you view or edit a module by using the Visual Basic Editor (VBE). A VBA module consists of procedures.
Definitions and Terminology (I)

- **Code**: You perform actions in VBA by executing VBA code. You write (or record) VBA code, which is stored in a VBA module.

- **Module**: VBA modules are stored in an Excel workbook file, but you view or edit a module by using the Visual Basic Editor (VBE). A VBA module consists of procedures.

- **Procedures**: A procedure is basically a unit of computer code that performs some action. VBA supports two types of procedures: `Sub` procedures and `Function` procedures.
Definitions and Terminology (II)

- **Objects**: VBA manipulates objects contained in its host application (Excel in this case). Examples of objects include a workbook, a worksheet, a range on a worksheet, a chart, and a shape.
Definitions and Terminology (II)

- **Objects**: VBA manipulates objects contained in its host application (Excel in this case). Examples of objects include a workbook, a worksheet, a range on a worksheet, a chart, and a shape.

- **Collections**: Like objects form a collection. For example, the Worksheets collection consists of all the worksheets in a particular workbook.
Definitions and Terminology (II)

- **Objects**: VBA manipulates objects contained in its host application (Excel in this case). Examples of objects include a workbook, a worksheet, a range on a worksheet, a chart, and a shape.

- **Collections**: Like objects form a collection. For example, the Worksheets collection consists of all the worksheets in a particular workbook.

- **Object Hierarchy**: When you refer to a contained or member object, you specify its position in the object hierarchy by using a period (also known as a *dot*) as a separator between the container and the member. Example (all in one line):

```vba
Application.Workbooks("Book1.xlsx").Worksheets("Sheet1").Range("A1")
```
Visual Basic Editor (VBE)

• To access the VBE, click on Visual Basic from the Developer tab or use the keyboard shortcut ALT + F11
Visual Basic Editor (VBE)

- To access the VBE, click on Visual Basic from the Developer tab or use the keyboard shortcut ALT + F11
- To insert a new Module, go to: Insert → Module
Visual Basic Editor (VBE)

- To access the VBE, click on Visual Basic from the Developer tab or use the keyboard shortcut ALT + F11
- To insert a new Module, go to: Insert → Module
- Project Explorer Window: Displays a tree diagram that consists of every workbook that is currently open in Excel (including add-ins and hidden workbooks). Each workbook is known as a project.
Visual Basic Editor (VBE)

- To access the VBE, click on Visual Basic from the Developer tab or use the keyboard shortcut ALT + F11.
- To insert a new Module, go to: Insert → Module.
- Project Explorer Window: Displays a tree diagram that consists of every workbook that is currently open in Excel (including add-ins and hidden workbooks). Each workbook is known as a project.
- Code Window: Contains VBA code.
Visual Basic Editor (VBE)

- To access the VBE, click on Visual Basic from the Developer tab or use the keyboard shortcut ALT + F11
- To insert a new Module, go to: Insert → Module
- Project Explorer Window: Displays a tree diagram that consists of every workbook that is currently open in Excel (including add-ins and hidden workbooks). Each workbook is known as a project.
- Code Window: Contains VBA code
- Properties Window: Allows you to change the properties of the item that is selected in the Project Explorer Window. To view it, go to: View → Properties Window or hit F4.
Data Types

- Some of VBA’s data types: Integer, Double (double-precision real numbers), String, Variant
Data Types

- Some of VBA’s data types: Integer, Double (double-precision real numbers), String, Variant
- The Variant data type can store any type of objects
Data Types

- Some of VBA’s data types: Integer, Double (double-precision real numbers), String, Variant
- The Variant data type can store any type of objects
- By default, if you don’t declare the type of a variable it will be Variant
Data Types

- Some of VBA’s data types: Integer, Double (double-precision real numbers), String, Variant
- The Variant data type can store any type of objects
- By default, if you don’t declare the type of a variable it will be Variant
- Good programming practice: use the Option Explicit statement to enforce the declaration of the variables types
Data Types

- Some of VBA’s data types: Integer, Double (double-precision real numbers), String, Variant
- The Variant data type can store any type of objects
- By default, if you don’t declare the type of a variable it will be Variant
- Good programming practice: use the Option Explicit statement to enforce the declaration of the variables types
- Use the keyword Dim to declare variables, for example:

```vba
Dim x as Double, y as Single
```
Operators

- Operators and their precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Order of Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Exponentiation</td>
<td>1</td>
</tr>
<tr>
<td>* and /</td>
<td>Multiplication and division</td>
<td>2</td>
</tr>
<tr>
<td>+ and −</td>
<td>Addition and subtraction</td>
<td>3</td>
</tr>
<tr>
<td>&amp;</td>
<td>Concatenation</td>
<td>4</td>
</tr>
<tr>
<td>=, &lt;, &gt;, &lt;=, &gt;=, &lt;&gt;</td>
<td>Comparison</td>
<td>5</td>
</tr>
</tbody>
</table>
Operators

- Operators and their precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Order of Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Exponentiation</td>
<td>1</td>
</tr>
<tr>
<td>* and /</td>
<td>Multiplication and division</td>
<td>2</td>
</tr>
<tr>
<td>+ and −</td>
<td>Addition and subtraction</td>
<td>3</td>
</tr>
<tr>
<td>&amp;</td>
<td>Concatenation</td>
<td>4</td>
</tr>
<tr>
<td>=, &lt;, &gt;, &lt;=, &gt;=, &lt;&gt;</td>
<td>Comparison</td>
<td>5</td>
</tr>
</tbody>
</table>

- Use parentheses to enforce precedence, for example:

\[
\begin{align*}
    x &= 4 + 3 \times 2 \\
    y &= (4 + 3) \times 2
\end{align*}
\]
**Logical Operators**

- VBA’s logical operators are:

<table>
<thead>
<tr>
<th>Operator</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not</td>
<td>Performs a logical negation on an expression.</td>
</tr>
<tr>
<td>And</td>
<td>Performs a logical conjunction on two expressions.</td>
</tr>
<tr>
<td>Or</td>
<td>Performs a logical disjunction on two expressions.</td>
</tr>
<tr>
<td>Xor</td>
<td>Performs a logical exclusion on two expressions.</td>
</tr>
<tr>
<td>Eqv</td>
<td>Performs a logical equivalence on two expressions.</td>
</tr>
<tr>
<td>Imp</td>
<td>Performs a logical implication on two expressions.</td>
</tr>
</tbody>
</table>
Logical Operators

- VBA’s logical operators are:

<table>
<thead>
<tr>
<th>Operator</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not</td>
<td>Performs a logical negation on an expression.</td>
</tr>
<tr>
<td>And</td>
<td>Performs a logical conjunction on two expressions.</td>
</tr>
<tr>
<td>Or</td>
<td>Performs a logical disjunction on two expressions.</td>
</tr>
<tr>
<td>Xor</td>
<td>Performs a logical exclusion on two expressions.</td>
</tr>
<tr>
<td>Eqv</td>
<td>Performs a logical equivalence on two expressions.</td>
</tr>
<tr>
<td>Imp</td>
<td>Performs a logical implication on two expressions.</td>
</tr>
</tbody>
</table>

- For example, the following statement displays True when either Sheet1 or Sheet2 is the active sheet:

```
MsgBox ActiveSheet.Name = "Sheet1" Or ActiveSheet.Name = "Sheet2"
```
Procedures: **Sub** and **Function**

- **Sub**: Consists of a series of statements and can be executed in a number of ways. Example of a **Sub** called **Test** that displays the string “Hello, world!” in a message box:

```
Sub Test()
    MsgBox "Hello, world!"
End Sub
```

To run it, click on the “play” button in the VBE toolbar or hit **F5**
Procedures: Sub and Function

• **Sub:** Consists of a series of statements and can be executed in a number of ways. Example of a Sub called Test that displays the string “Hello, world!” in a message box:

```vb
Sub Test()
    MsgBox "Hello, world!"
End Sub
```

To run it, click on the “play” button in the VBE toolbar or hit F5.

• **Function:** Returns a single value (or possibly an array) and can be called from another VBA procedure or used in a worksheet formula. Example of a Function named AddTwo:

```vb
Function AddTwo(arg1, arg2)
    AddTwo = arg1 + arg2
End Function
```
Complete Function Definition

- Explicitly define the arguments types and the return type

```vba
Option Explicit

Sub Main()
  Dim a As Double, b As Double
  a = 1
  b = MyFunc(a)
  ' b receives the output of MyFunc and a is modified
End Sub

Function MyFunc(x As Double) As Double
  MyFunc = 2 * x
  x = x + 1
End Function
```
Complete Function Definition

- Explicitly define the arguments types and the return type
- Use the keywords `ByVal` and `ByRef` allow you to send arguments “by value” and “by reference” (default), respectively

```vba
Option Explicit
Sub Main()
    Dim a as Double, b as Double
    a = 1
    b = MyFunc(a)  ' b receives the output of MyFunc and a is modified
End Sub

Function MyFunc(x as Double) as Double
    MyFunc = 2 * x
    x = x + 1
End Function
```
Complete Function Definition

- Explicitly define the arguments types and the return type
- Use the keywords `ByVal` and `ByRef` allow you to send arguments “by value” and “by reference” (default), respectively
- `ByVal` means that only the value of the argument is passed to the procedure
Complete Function Definition

- Explicitly define the arguments types and the return type
- Use the keywords `ByVal` and `ByRef` allow you to send arguments “by value” and “by reference” (default), respectively
- `ByVal` means that only the value of the argument is passed to the procedure
- `ByRef` means that the reference or pointer to the argument is passed to the procedure, and any local changes will affect the passed object. For example (try adding `ByVal` before the argument `x`):

```
Option Explicit
Sub Main()
    Dim a as Double, b as Double
    a = 1
    b = MyFunc(a) ' b receives the output of MyFunc and a is modified
End Sub
Function MyFunc(x as Double) as Double
    MyFunc = 2 * x
    x = x + 1
End Function
```
Declaring Arrays

- An array of integers with 100 elements is declared as follows:

```lang-vba
Dim a(1 To 100) as Integer
```

where 1 is the lower index and 100 is the upper index.
Declaring Arrays

- An array of integers with 100 elements is declared as follows:

\[
\text{Dim a(1 To 100) as Integer}
\]

where 1 is the lower index and 100 is the upper index.

- If you define the array with only the upper index, the lower index is 0 by default, so the following declarations have the same effect:

\[
\begin{align*}
\text{Dim b(0 To 100) as Integer} \\
\text{Dim c(100) as Integer}
\end{align*}
\]
Declaring Arrays

- An array of integers with 100 elements is declared as follows:

```vba
Dim a(1 To 100) as Integer
```

where 1 is the lower index and 100 is the upper index.

- If you define the array with only the upper index, the lower index is 0 by default, so the following declarations have the same effect:

```vba
Dim b(0 To 100) as Integer
Dim c(100) as Integer
```

- Multidimensional arrays follow the same ideas:

```vba
Dim d(1 To 10, 1 To 10) as Integer ' 10-by-10 array

\[d(1, 2) = 0\]
```
Dynamic Arrays

- Useful when the size of the array is unknown *a priori*

```vba
Dim a() as Double
```

suppose the integer variable `n` contains the size, you can do:

```vba
ReDim a(1 To n) ' Destroys array’s values if existent
```

or

```vba
ReDim Preserve a(1 To n) ' Keeps array’s values if existent
```
Outline

1 Motivation
2 Recording Macros
3 VBA Programming: Basics
4 Manipulating Objects and Collections
5 Controlling Code Execution
6 Looping Blocks of Instructions
7 Working with Spreadsheets
   Working with Ranges
   Working with Cells
8 Applications
   Calling MATLAB from VBA
With-End With Constructs

- Enables you to perform multiple operations on a single object
With–End With Constructs

- Enables you to perform multiple operations on a single object
- For example, in the following code we avoided repeating Selection.Font in all statements:

```vba
Sub ChangeFont()
    With Selection.Font
        .Name = "Cambria"
        .Bold = True
        .Italic = True
        .Size = 12
        .Underline = xlUnderlineStyleSingle
        .ThemeColor = xlThemeColorAccent1
    End With
End Sub
```
For Each–Next Constructs

- Enables you to iterate through all objects in a collection and perform some action on them
For Each–Next Constructs

- Enables you to iterate through all objects in a collection and perform some action on them
- For example, in the following code the `MsgBox` function displays each worksheet’s `Name` property:

```vba
Sub CountSheets()
    Dim Item as Worksheet
    For Each Item In Worksheets
        MsgBox Item.Name
    Next Item
End Sub
```
1. Motivation
2. Recording Macros
3. VBA Programming: Basics
4. Manipulating Objects and Collections
5. Controlling Code Execution
6. Looping Blocks of Instructions
7. Working with Spreadsheets
   - Working with Ranges
   - Working with Cells
8. Applications
   - Calling MATLAB from VBA
If–Then **Constructs**

- Used to execute one or more statements conditionally
If–Then Constructs

- Used to execute one or more statements conditionally
- For example, in the following code the `MsgBox` function displays a greeting message according to the time you execute the `Sub`:

```vba
Sub GreetMe()
    If Time < 0.5 Then
        MsgBox "Good Morning"
    ElseIf Time >= 0.5 And Time < 0.75 Then
        MsgBox "Good Afternoon"
    Else
        MsgBox "Good Evening"
    End If
End Sub
```
If–Then Constructs

• Used to execute one or more statements conditionally
• For example, in the following code the `MsgBox` function displays a greeting message according to the time you execute the `Sub`:

```vba
Sub GreetMe()
    If Time < 0.5 Then
        MsgBox "Good Morning"
    ElseIf Time >= 0.5 And Time < 0.75 Then
        MsgBox "Good Afternoon"
    Else
        MsgBox "Good Evening"
    End If
End Sub
```

• The `ElseIf` and `Else` blocks are optional
Select Case Constructs

- Useful for choosing among three or more options
Select Case Constructs

- Useful for choosing among three or more options
- For example, another way of coding the Sub GreetMe():

```vba
Sub GreetMe2()
    Dim Msg As String
    Select Case Time
        Case Is < 0.5
            Msg = "Good Morning"
        Case 0.5 To 0.75
            Msg = "Good Afternoon"
        Case Else
            Msg = "Good Evening"
    End Select
    MsgBox Msg
End Sub
```
Outline

1. Motivation
2. Recording Macros
3. VBA Programming: Basics
4. Manipulating Objects and Collections
5. Controlling Code Execution
6. Looping Blocks of Instructions
7. Working with Spreadsheets
   - Working with Ranges
   - Working with Cells
8. Applications
   - Calling MATLAB from VBA
For-Next Loops

- For example, calculate the sum of the square roots of the first 100 positive integers:

```vba
Sub SumSquareRoots()
    Dim Sum As Double
    Dim Count As Integer
    Sum = 0
    For Count = 1 To 100 Step 1
        Sum = Sum + Sqr(Count)
    Next Count
    MsgBox Sum
End Sub
```
For-Next Loops

- For example, calculate the sum of the square roots of the first 100 positive integers:

```vba
Sub SumSquareRoots()
    Dim Sum As Double
    Dim Count As Integer
    Sum = 0
    For Count = 1 To 100 Step 1
        Sum = Sum + Sqr(Count)
    Next Count
    MsgBox Sum
End Sub
```

- The Step 1 is optional. You can use Step -1 to loop “backward”
Do-While Loops

- For example, open a text file in the current directory and display its contents line-by-line:

```vba
Sub DoWhileFile()
    Dim LineOfText As String
    Open ThisWorkbook.Path & "\file.txt" _
      For Input As #1
    Do While Not EOF(1)
      Line Input #1, LineOfText
      MsgBox LineOfText
    Loop
    Close #1
End Sub
```
Do-While Loops

- For example, open a text file in the current directory and display its contents line-by-line:

```vba
Sub DoWhileFile()
    Dim LineOfText As String
    Open ThisWorkbook.Path & "\file.txt" _
        For Input As #1
    Do While Not EOF(1)
        Line Input #1, LineOfText
        MsgBox LineOfText
    Loop
    Close #1
End Sub
```

- There are also Do-Until loops
Outline

1 Motivation
2 Recording Macros
3 VBA Programming: Basics
4 Manipulating Objects and Collections
5 Controlling Code Execution
6 Looping Blocks of Instructions
7 Working with Spreadsheets
   Working with Ranges
   Working with Cells
8 Applications
   Calling MATLAB from VBA
Ranges: Basics

- A Range may be a single Cell or a collection of Cells
Ranges: Basics

- A **Range** may be a single **Cell** or a collection of **Cells**
- Refer to cells in the same way you would do on a spreadsheet, *i.e.* the cell “A1” can be referred to as **Range(“A1”)** in VBA
Ranges: Basics

- A Range may be a single Cell or a collection of Cells
- Refer to cells in the same way you would do on a spreadsheet, i.e. the cell “A1” can be referred to as Range(“A1”) in VBA
- One of the most useful properties of a Range is its Value
Ranges: Basics

- A **Range** may be a single **Cell** or a collection of **Cells**
- Refer to cells in the same way you would do on a spreadsheet, *i.e.* the cell “A1” can be referred to as `Range(“A1”)` in VBA
- One of the most useful properties of a **Range** is its **Value**
- For example, the statement `Range(“A1”).Value = 1` will set the content of cell A1 in the active workbook and worksheet to 1
Copyng and Moving Ranges

- We can make use of an *object variable* that represents an entire object, such as a range, a worksheet *etc.*
Copying and Moving Ranges

- We can make use of an *object variable* that represents an entire object, such as a range, a worksheet *etc*.
- To create an object variable, use the keyword `Set` after declaring the variable with `Dim`
Copying and Moving Ranges

- We can make use of an *object variable* that represents an entire object, such as a range, a worksheet *etc.*
- To create an object variable, use the keyword *Set* after declaring the variable with *Dim*
- The following example copies a range of cells to another location

```vba
Sub CopyRange()
    Dim Rng1 As Range, Rng2 As Range
    Set Rng1 = Worksheets("Ranges").Range("A1:A3")
    Set Rng2 = Worksheets("Ranges").Range("B1")
    Rng1.Copy Rng2
End Sub
```
Copying and Moving Ranges

• We can make use of an object variable that represents an entire object, such as a range, a worksheet etc.

• To create an object variable, use the keyword Set after declaring the variable with Dim

• The following example copies a range of cells to another location

```vba
Sub CopyRange()
    Dim Rng1 As Range, Rng2 As Range
    Set Rng1 = Worksheets("Ranges").Range("A1:A3")
    Set Rng2 = Worksheets("Ranges").Range("B1")
    Rng1.Copy Rng2
End Sub
```

• Similarly, the method Cut moves a range to another location
Cells objects are useful when reading/writing consecutive cells from/to a worksheet in a loop block.
Cells: Basics

- **Cells objects** are useful when reading/writing consecutive cells from/to a worksheet in a loop block.
- The equivalent to `Range(“A1”)` is `Cells(1, 1)`.
Cells: Basics

- Cells objects are useful when reading/writing consecutive cells from/to a worksheet in a loop block.
- The equivalent to `Range("A1")` is `Cells(1, 1)`.
- Likewise for Range objects, use the Cells’ property Value to read and write values.
Cells: Basics

- **Cells objects** are useful when reading/writing consecutive cells from/to a worksheet in a loop block.
- The equivalent to `Range("A1")` is `Cells(1, 1)`.
- Likewise for `Range` objects, use the `Cells`’ property `Value` to read and write values.
- You can refer to cells relative to other cells by using the `Offset` property. For instance,

  ```vba
  Cells(1, 1).Offset(2, 3)
  ```

  refers to range D3.
Writing Values to Cells Iteratively

- The next code shows how to read integer numbers from a file and write them in the first column of a given worksheet.

```vba
Sub WriteDataToCells()
    Dim count As Integer, number As Integer
    Dim sheetName As String
    sheetName = "Cells"
    count = 1
    Open ThisWorkbook.Path & "\file.txt" For Input As #1
    Do While Not EOF(1)
        Input #1, number
        Worksheets(sheetName).Cells(count, 1).Value = number
        count = count + 1
    Loop
    Close #1
End Sub
```
Writing Values to Cells Iteratively

- The next code shows how to read integer numbers from a file and write them in the first column of a given worksheet

```vba
Sub WriteDataToCells()
    Dim count As Integer, number As Integer
    Dim sheetName As String
    sheetName = "Cells"
    count = 1
    Open ThisWorkbook.Path & "\file.txt" For Input As #1
    Do While Not EOF(1)
        Input #1, number
        Worksheets(sheetName).Cells(count, 1).Value = number
        count = count + 1
    Loop
    Close #1
End Sub
```

- For-Next loops are also useful when reading/writing from/to worksheet cells
Setup and Getting Help

- You can easily call MATLAB functions from VBA
Setup and Getting Help

- You can easily call MATLAB functions from VBA
- First, add the MATLAB reference in: Tools → References... → select “Matlab Application” and hit OK
Setup and Getting Help

- You can easily call MATLAB functions from VBA
- First, add the MATLAB reference in: Tools → References... → select “Matlab Application” and hit OK
- There is *some* documentation with examples in MATLAB’s Help (search for “Visual Basic”)
Setup and Getting Help

- You can easily call MATLAB functions from VBA
- First, add the MATLAB reference in: Tools → References... → select “Matlab Application” and hit OK
- There is some documentation with examples in MATLAB’s Help (search for “Visual Basic”)
- Basic steps:
  - *Create* a “MATLAB object” (VBA function: `CreateObject`)
Setup and Getting Help

- You can easily call MATLAB functions from VBA
- First, add the MATLAB reference in: Tools → References... → select “Matlab Application” and hit OK
- There is some documentation with examples in MATLAB’s Help (search for “Visual Basic”)
- Basic steps:
  - Create a “MATLAB object” (VBA function: CreateObject)
  - Put data in MATLAB’s workspace (VBA functions: PutFullMatrix, Execute)
Setup and Getting Help

- You can easily call MATLAB functions from VBA
- First, add the MATLAB reference in: Tools → References... → select “Matlab Application” and hit OK
- There is some documentation with examples in MATLAB’s Help (search for “Visual Basic”)
- Basic steps:
  - Create a “MATLAB object” (VBA function: CreateObject)
  - Put data in MATLAB’s workspace (VBA functions: PutFullMatrix, Execute)
  - Execute some routine, e.g. solving ODEs (VBA function: Execute)
Setup and Getting Help

- You can easily call MATLAB functions from VBA
- First, add the MATLAB reference in: Tools → References... → select “Matlab Application” and hit OK
- There is some documentation with examples in MATLAB’s Help (search for “Visual Basic”)
- Basic steps:
  - Create a “MATLAB object” (VBA function: CreateObject)
  - Put data in MATLAB’s workspace (VBA functions: PutFullMatrix, Execute)
  - Execute some routine, e.g. solving ODEs (VBA function: Execute)
  - Get data from MATLAB’s workspace (VBA functions: GetVariable, GetFullMatrix, Execute)
Example: Volume of a PFR

- The volume of a Plug Flow Reactor (PFR) is calculated by the following expression:

\[ V = F_{A0} \int_{0}^{X'} \frac{dX}{-r_A(X)} \]

where \( V \) is the reactor volume, \( F_{A0} \) is the inlet molar flow of the limiting reactant \( A \), \( X \) is the reaction conversion, \( r_A(X) \) is the reaction rate.
Example: Volume of a PFR

- The volume of a Plug Flow Reactor (PFR) is calculated by the following expression:

\[ V = F_{A0} \int_0^{X'} \frac{dX}{-r_A(X)} \]

where \( V \) is the reactor volume, \( F_{A0} \) is the inlet molar flow of the limiting reactant \( A \), \( X \) is the reaction conversion, \( r_A(X) \) is the reaction rate.

- The value of \( F_{A0} \) and a table with \(-r_A \) versus \( X \) data are given.
Example: Volume of a PFR

- The volume of a Plug Flow Reactor (PFR) is calculated by the following expression:

\[ V = F_{A0} \int_0^{X'} \frac{dX}{-r_A(X)} \]

where \( V \) is the reactor volume, \( F_{A0} \) is the inlet molar flow of the limiting reactant \( A \), \( X \) is the reaction conversion, \( r_A(X) \) is the reaction rate

- The value of \( F_{A0} \) and a table with \(-r_A\) versus \( X \) data are given

- To compute \( V \), we need to numerically integrate the data
Example: Volume of a PFR

- The volume of a Plug Flow Reactor (PFR) is calculated by the following expression:

\[ V = F_{A0} \int_{0}^{X'} \frac{dX}{-r_A(X)} \]

where \( V \) is the reactor volume, \( F_{A0} \) is the inlet molar flow of the limiting reactant \( A \), \( X \) is the reaction conversion, \( r_A(X) \) is the reaction rate.

- The value of \( F_{A0} \) and a table with \(-r_A\) versus \( X \) data are given.

- To compute \( V \), we need to *numerically* integrate the data.

- You can call MATLAB’s `trapz` function to perform the integral and then retrieve the result to the worksheet.
Example: Volume of a PFR

- The volume of a Plug Flow Reactor (PFR) is calculated by the following expression:

\[ V = F_{A0} \int_0^{X'} \frac{dX}{-r_A(X)} \]

where \( V \) is the reactor volume, \( F_{A0} \) is the inlet molar flow of the limiting reactant \( A \), \( X \) is the reaction conversion, \( r_A(X) \) is the reaction rate.

- The value of \( F_{A0} \) and a table with \(-r_A \) versus \( X \) data are given.

- To compute \( V \), we need to *numerically* integrate the data.

- You can call MATLAB’s \texttt{trapz} function to perform the integral and then retrieve the result to the worksheet.

- See file \texttt{VBA_Examples.xlsm}, worksheet “MATLAB Example”.