Visual Basic for Applications - Programming Excel

Michael Schacht Hansen

October 6, 2002

Contents

1 Introduction

1.1 Purpose

The purpose of this course is to:

- Demonstrate the use of so-called office programs in research.
- Make the participants comfortable with the use of office programs.
- Introduce tools for automating tedious tasks in research.
- Make the participants comfortable with macro programming using VBA.

In short to help the participants overcome some of the boring, tedious tasks and free up time for research.

1.2 Who should participate

Basically anybody who wants to improve their skills with spreadsheets, word processors etc. The course will focus on handling of large amounts of data in an efficient way. It is assumed that the participants have some experience (but not expert level) with spreadsheets and word processors (e.g. MS Word and MS Excel).

1.3 Materials used

The examples in this course were made for Microsoft Office programs. The teachers of this course are not financed by Microsoft in anyway, but realizing that MS Office is the most widely used office suite, the choice seemed obvious. The code (programming) illustrated in this (and other) documents is thus written for MS Office programs in MS Visual Basic for Applications (VBA). Participants working with other systems in their research setting can still profit from the course. We will focus on the general principles, and these should be portable (after reading the documentation on the other system). It has been decided to recommend a book for the course (Excel 2002 Power Programming with VBA by John Walkenbach). It is not absolutely necessaryto purchase and read this book; reading this document and other course notes should suffice, but the books gives a more systematic introduction to the main subject of the course - VBA programming. Materials, examples and presentations used in the course are available from www.intermed.dk/VBA

1.4 Conventions used

To make notes and examples more readable we have adapted the following conventions: All code examples are written with a mono space font (that is the way it looks in the editor). Not all the code examples can work on their own. It may just be a few lines to illustrate a principle. Usually commands should be written on one line, but the width of a normal page would not allow this and sometimes line breaks are needed in the code examples. Line breaks are allowed when actually writing the code if you leave a trailing " _" (space followed by underscore) of the end of the broken line. The same syntax for line breaks are used in the documents. An example:

```
Sub MyMacro()
 'Macro example

 ActiveSheet.Pictures.Insert ( _
  "C:\temp\01valve.jpg")
End Sub
```

Keyboard shortcuts are written in bold face e.g. pressing **ALT+F11** activates the visual basic editor. Menu access is written with arrows indicating the path of the mouse (or sequence of shortcut keys); to save a file click File→Save.

1.5 What one fool can do another can

Participants who have prior programming experience may find some of the material in the course boring. Since no programming skills are required to participate, and since programming is an important part of the course, some basic programming skills must be taught. Please note however that programming languages vary in syntax and suitable programming style, and it might be worth the effort to listen in after all.

If you have no programming experience, do not despair. Even if your basic computer skills are somewhat lacking you should have a very good chance of changing all that. Programming (even macro programming) has been considered a subject for geeks and nerds, and lots of people have solved their problems by spending hours repeating tasks that they could have programmed their way out of, simply because they thought programming was too difficult. The most important thing when you start programming is to be motivated. To be fully motivated you should have the three great virtues of a programmer:

1. Laziness
2. Impatience
3. Hybris

This should not be misunderstood. You should be too lazy to repeat a task when you can have a computer do it for you, you should be interested in setting up a system that lets you analyze results as soon as you are finished measuring (or what ever you do), and most importantly, you should not be afraid to try the impossible.

With the motivation in place you will be amazed how easy it is. You might actually have some fun, and finally feel that you have tamed the beast occupying your desk. Considering how many fools can program, it is surprising that it should be thought either a difficult or a tedious task for any fool to master the same tricks.

2 Getting started

2.1 What you need to follow the examples in this document

All examples in this document were programmed in Microsoft Visual Basic for Applications (VBA) to work in Microsoft Office XP (Excel). It is highly unlikely that they will work without modifications in other office suites, but most of them can be made to work with StarBasic (not for beginners) in the StarOffice suite and earlier versions of Microsoft Office should do fine too. So to follow the examples office you should have Microsoft Office (preferably XP) installed on your system, and you also need the Macro Programming environment Visual Basic (installed by default), and the Visual Basic Editor (VBE) installed. If you have a standard installation of MS Office you should be good to go.

2.2 The macro recorder - the first macro

2.2.1 What is the macro recorder

Unlike other programming environments VBA has a built in recorder that has the ability to record your actions in your spreadsheet or word processor and translate them into VBA. This means that you can actually start programming without knowing a single command or keyword in the programming language. However, as we will see you cannot do everything with the macro recorder. Your actions are limited to the things you are already able to do in excel. This is not enough in the long run, but it will get you started, and it can be a very powerful learning tool.

2.2.2 Recording the first macro

Recording your actions is real simple in MS Office. Try opening excel, and a new workbook. Activate the macro recorder by choosing Tools→Macro→Record new macro...[1] Choose a name for the new macro in the dialog (Macro1 is probably not a suitable name), and press OK. The dialog disapears and a small control panel appears. The control panel has two buttons. Stop (to stop the macro recorder) and a buttons to toggle between absolute and relative references. Forget about that other button for now. We will get back to in in just a few seconds. Try typing something into a few cells and stop the recorder. You have now recorded a new macro. To test if it works, try deleting what you have entered in the cells and run the macro. This is done by choosing Tools→Macro→Macros... or pressing **ALT+F8**. Mark the recorded macro from the list and click run (later on we will see how macros can be assigned to shortcut keys or buttons on the tool bar). If everything is in order the macro will enter what you told it to in the sheet. You might notice that the macro enters the text or numbers in the exact same cells as you did regardless of where you place the cursor prior to running the macro. This is because we recorded the macro with absolute references. The next section illustrates the difference between absolute and relative references.

2.2.3 Recording with absolute and relative references

The absolute and relative reference settings are much easier illustrated than they are explained. Try clearing the sheet from the above example, activate the cell where you want to start filling in information, start the macro recorder and press the relative references button. Now fill in some information in the cells and

[1]Depending on your security settings the Record new macro option might be faded out, and you will not be able to activate it. To correct this choose Tools→Macro→Security..., and set the security level to medium or low. You should restart Excel if the Record new macro option is not available after this.

turn the macro recorder off. If you try to run the macro now, you will notice that it inserts whatever information you have told it to, but it does it in the same position as before relative to your current starting point. That is why it is called relative references. We could write pages about how this works, but when you play around with it you will notice the difference. When we get more comfortable with VBA we will be able to explain the different behavior from the recorded code, and we will appreciate the different macro strategies.

2.2.4 What can be recorded

The macro recorder can record basically any task you can do yourself in Excel. However VBA has lots of other abilities that you cannot do yourself in Excel, some of them being automating tasks (looping), creating dialog boxes and custom menus. It should also be mentioned that the macro recorder does not create the most efficient code in the world (see the following). The macro recorder can be a great learning tool, but to take full advantage of VBA, you should be able to understand the actual VBA code.

2.3 Assigning macros to shortcut keys or buttons

Macros can easily be assigned to a shortcut key or a button on a menu bar or in the sheet. This can be an advantage if you use the macro often or if you want to create a user friendly interface for a macro you have created.

2.3.1 Assign a macro to a shortcut key

Open the macro dialog box (Tools→Macro→Macros… or press **ALT+F8**). Mark the macro you are interested in and click Options… A dialog box opens and you can enter a shortcut. **Ctrl+** is already filled in and you can enter letter in the little box. If you enter an uppercase letter, for instance L, the shortcut would be **CTRL+SHIFT+L** and if you enter a lowercase letter, for instance l, the shortcut would be **CTRL+L**. Remember that you can override built in keyboard shortcuts this way. If you are overriding shortcuts you dont know exist it doesnt really matter!

2.3.2 Assigning macros to menu buttons

Right-click the menu bar an choose customize. Make sure that the menu bar you want to add the macro to is present. Click the Commands tab in the dialog box. Find Macros in the Categories section. Drag the custom button to the menu bar. Click modify selection in the dialog box. Choose assign macro, and assign the macro. Click Modify selection again and choose Edit button image to edit the image in the button. You now have a macro button on your menu bar.

2.3.3 Assign macro to a button on a sheet

Make the Forms menu bar visible by right-clicking the menu bar and choosing Forms. Click the button icon in the menu bar and draw a button on your sheet. A dialog box will automatically ask you what macro you want to assign to the button. Edit the text on the button afterward, and close the forms menu bar. Thats it!

2.4 The Visual Basic Editor

Until now you have been creating macros, and you havent written a single line of code yet. This does not mean that the code has not been written. You have just had the macro recorder do the work for you. This is also a fine solution for some problems, but in the long run you need to be able to modify the code made by the recorder and create your own. The MS Office Suite comes with a built in programming environment, The Visual Basic Editor (VBE). In this section we will just browse through some of the main features of VBE. We will not be able to cover all of them, but we will return to them when it seems appropriate (We will not cover the object browser in this section, but save it for the section on object structure).

2.4.1 Activating VBE and a brief overview

The easiest way to enter VBE is with the shortcut key **ALT+F11**. You can also enter it from the Tools menu, but why should you? It is hard to anticipate what you might meet when you enter VBE, since it is highly customizable, but most likely there is a large area on the right for script windows (thats where the code goes), and on the left side you probably have a window called Project Browser. You might also have a window called Properties, and there might even be a few others. For now we will focus on the Project browser and the code windows. The Project browser gives you a brief overview of the workbooks that are open at the moment and what code they contain. It is actually very simple, and it works much like the normal Windows Explorer, where you browse through your files. Here you browse through different workbooks, sheets and something called modules. All of these can contain code, and most of it you can edit as you see fit. If you minimize all the project in the Project Browser you might be able to locate the workbook you are currently working on. The next thing you will notice is that there are probably other workbooks open at the same time; workbooks that have been opened automatically. Some programs (Reference Manager, Adobe PDF writer etc.) install themselves as macro modules in MS Office, and you might be able to see some of them. Furthermore the Office programs open some on their own. Try expanding the workbook you are working on at the moment. You will be able to see two folders. One contains Excel Object (this means the workbook and the sheets it contains), and the other one contains modules. The Modules are the default place to insert code for the macro recorder, and the macros you have recorded are placed in the modules in this folder. You can place code on the sheets and in the workbook, but for now we will concentrate on the code in the modules.

2.4.2 Taking a first look at the code

Try double-clicking one of the modules in the module folder (there might only be one). A code window should open on the right side of the screen, and you can see the code that you have recorded. The first macro with absolute references should look something like this:

```
Sub Macro1()
'
' Macro1 Macro
' Macro recorded 19/04/2000 by Michael Schacht Hansen
'
'
   Range("D8").Select
   ActiveCell.FormulaR1C1 = "Freedom"
   Range("E8").Select
   ActiveCell.FormulaR1C1 = "to"
   Range("F8").Select
   ActiveCell.FormulaR1C1 = "choose"
   Range("F9").Select
End Sub
```

Depending on where you entered your data and what you entered your macro will look a little bit different, but they should look more or less the same. The first few lines starting with "'" are comments and are not interpreted when the macro is executed, but the remaining lines are. It may seem a bit confusing for you at first, but you will understand the code completely before long. The macro starts with the `Sub` statement followed by the name of the macro. This means a new macro starts here and it continues on to the `End Sub` statement. For now just accept the way things look. If you examine the other macro recorded with relative references you will find something like this:

```
Sub Macro2()
'
' Macro2 Macro
' Macro recorded 19/04/2000 by Michael Schacht Hansen
'
'
   ActiveCell.Select
   ActiveCell.FormulaR1C1 = "Freedom"
   ActiveCell.Offset(0, 1).Range("A1").Select
   ActiveCell.FormulaR1C1 = "to"
   ActiveCell.Offset(0, 1).Range("A1").Select
   ActiveCell.FormulaR1C1 = "choose"
   ActiveCell.Offset(0, 1).Range("A1").Select
End Sub
```

This also might not make sense to you, but for now just notice that the second version makes frequent use of the keyword `Offset` which should tell you that this version works in relative coordinates from the starting point, while the first version have absolute cell references hard-coded into the macro. Hence it works with absolute references.

You should try changing the code for the macro and observe the outcome. Start by changing the text you recognize as the text or formulas you entered when recording the macro. Then continue to change some of the cell references and see what happens.

2.4.3 A few hints for VBE

Below I have listed (in a more or less random order) a few features that I use often in VBE:

- A macro can easily be executed from VBE, when you have made changes and want to observe the effect. This is done by placing the cursor somewhere within the code of the macro you want to execute and pressing **F5**.
- Help on a specific command or keyword in VBA can be found by placing the pointer over the keyword an pressing **F1**.
- The Object Browser (we will get to that in the next section) can be activated by pressing **F2**.
- Cut, copy, paste, and general text navigation works as in normal word processors.

2.5 Does the macro recorder create smooth, efficient code?

It has previously been mentioned that the macro recorder might not be the best programmer in the world. The point of the following example is to illustrate this. You might not be able to appreciate it now, and if the following makes no sense to you just skip it and return to it later when you programming skills have been improved.

The macro recorder records basically everything you do. This means if you open a dialog box, change one setting, and press ok. It will record all the settings in the dialog box. If it is a large dialog this becomes very inefficient. To illustrate this try recording a macro, where you change some page setup settings or similar. This is what gets recorded when you change the page orientation from portrait to landscape:

```
Sub Macro3()
'
' Macro3 Macro
' Macro recorded 06-10-2002 by Michael Schacht Hansen
'

'
    With ActiveSheet.PageSetup
        .PrintTitleRows = ""
        .PrintTitleColumns = ""
    End With
    ActiveSheet.PageSetup.PrintArea = ""
    With ActiveSheet.PageSetup
        .LeftHeader = ""
        .CenterHeader = ""
        .RightHeader = ""
        .LeftFooter = ""
        .CenterFooter = ""
        .RightFooter = ""
        .LeftMargin = Application.InchesToPoints(0.75)
        .RightMargin = Application.InchesToPoints(0.75)
        .TopMargin = Application.InchesToPoints(1)
        .BottomMargin = Application.InchesToPoints(1)
        .HeaderMargin = Application.InchesToPoints(0.5)
        .FooterMargin = Application.InchesToPoints(0.5)
        .PrintHeadings = False
        .PrintGridlines = False
        .PrintComments = xlPrintNoComments
        .PrintQuality = 600
        .CenterHorizontally = False
        .CenterVertically = False
        .Orientation = xlLandscape
        .Draft = False
        .PaperSize = xlPaperLetter
        .FirstPageNumber = xlAutomatic
        .Order = xlDownThenOver
        .BlackAndWhite = False
        .Zoom = 100
        .PrintErrors = xlPrintErrorsDisplayed
    End With
End Sub
```

You would have obtained the same effect with the following macro:

```
Sub ChangeOrientation()
   ActiveSheet.PageSetup.Orientation = xlLandscape
End Sub
```

Obviously the macro recorder is not very efficient in this case, but on the other hand we learned how to change the page orientation by recording the macro, and we were then able to create the compact macro. This is a good illustration of the strength and weakness of the recorder. It is an excellent learning tool, but with a little training you are a much better programmer! Aside from this obvious disadvantage of having the recorder do all the code for you, there is another. There are a lot of features that you simply cannot record. For instance try this macro:

```
Sub DisplayBox()
   MsgBox("This is a message to the user")
End Sub
```

When executed in Excel this should result in a dialog box, with a message to the user. You have no way of recording actions like that. There are lots of other (more important) examples, like creating loops that repeat the same action over and over (do the work for you while you drink coffee). The following chapters will be a short tour of the basic principles needed to write your own macros without use of the recorder, but remember to use the macro recorder when you want to find the command syntax for a specific action in Excel.

3 The Object Model

All macro programming in VBA is based on an object model, and is thus called object orientated programming (OOP). This concept is not unique to VBA, but is a widely adopted programming strategy thats used in almost all the programs released today. The reason why VBA is object orientated is probably because MS Office programs were programmed object orientated, and when creating a programming language for manipulating these programs it seemed obvious to apply an object orientated strategy. If you are already comfortable with OOP and referencing in object structures, you should skip this section, but if you are not quite sure what OOP is, please read on.

3.1 The analogy

All authors of books on object orientated programming have come up with their own analogy to describe OOP. It is one of those subjects that get more and more complicated the more you try to explain, and experience shows, that a description through an analogy does a better job, than a technical explanation of OOP (it might also be that the authors are not quite sure what they are dealing with).

First of all we need a useful definition of and object. Usually objects are considered to be things that you can touch and look at. Here we will define an object as a part of a virtual environment (somewhere in the computers memory) with certain properties and functions (also known as methods). This definition might not seem quite clear, but read on and hopefully things will clear up.

In our analogy we imagine that we have an exact model of the earth in a computers memory (every single atom is accounted for). We could also just have made our analogy on the earth as it is, but to emphasize that objects are not actual objects, but virtual objects, we will use the model of earth. In this model the mother object (the main object, containing all other objects) is called Earth. This Earth object has certain properties (for instance the location of Earth in space) and it has functions (for instance making a storm in southern Europe). The mother object contains other objects. Some of these objects are single objects, like The Pope, others are collections of similar objects like countries (Denmark would be a member of this collection). Each of the objects contained in Earth have properties and methods, and they can also contain objects. The country Denmark would contain another collection of objects called Cars. In such an object model you need a special syntax to identify (or reference) different objects. How do you for instance reference a car in Sweden or a car in Denmark. In OOP, the most widely adapted syntax is known as ”Dot-notation-syntax”. VBA also uses this syntax, and it is actually quite useful. To make an exact reference to the color of a specific car in the country Denmark on the planet Earth we would write:

```
Earth.Countries("Denmark").Cars("XP 38 999").Color
```

If you were to interpret this statement it would translate into something like: ”the color of the car named XP 38 999 in the country called Denmark on the planet Earth”. And here we have actually suggested that Earth might not even be the mother object (it might be part of the collection of Planets in the Universe... part of the collection Universes...). Anyway it seems obvious that the syntax illustrated above is much more useful than actually spelling out what you want.

Now the purpose of programming was to manipulate the model. We now have a model of Earth in a computer memory, and we want to observe what happens if we change the color of a specific car to red. The code would look something like:

```
Earth.Countries("Denmark").Cars("XP 38 999").Color = Red
```

If we wanted to make the car move, the code might look like the following. We are not accessing a property of the object we are asking it so execute a function (use a method) to manipulate the object. We are also supplying coordinates (for the movement):

```
Earth.Countries("Denmark").Cars("XP 38 999").Move(209,99)
```

This could be a movement of 209 units to the right and 99 units down. This is actually pretty simple once you get used to it. Programming Excel is just like manipulating our model of Earth. Excel was build on an object model, and accessing different part of the program should be pretty easy. VBA is actually just a tool to access and manipulate the different objects of Excel.

3.2 Excel objects

The hardest part of learning VBA is to learn what the objects are called and how they act in the model. Once you get the basic idea of how things work in one program (or part of a program) it is easy to apply the same strategies in another program, if you can find a way to learn what the objects, properties and methods are called.

The mother object in Excel is called Application. This a a direct reference to the Excel program itself. An Excel program can contain several open Workbooks (these are the *.xls files). Each of these Workbooks contain a number of sheets, and these contain cells and so on (you are probably beginning to get the idea by now). It would be impossible to list all the objects with their methods and properties here, but read the following sections to get an idea of where you can get that information. The are more than 100 major objects in Excel with the objects they contain, but fortunately you will probably only need to know a few of them.

So from the analogy above we get that referencing a specific sheet in Excel would look something like this:

```
Application.Workbooks("Name_of_wrkbk.xls").Sheets(1)
```

Notice that the reference to the sheet, we are interested in, is made with an index number. We are referencing sheet number 1 in this case. We could also have used the name of that sheet, but we might not know the name of the sheet. We would change the name of the sheet by writing:

```
Application.Workbooks("Name_of_wrkbk.xls").Sheets(1).Name _
  = "My sheet"
```

We could also call a method of the sheet. To activate the sheet (make it the sheet that the user would enter data into) we would write:

```
Application.Workbooks("Name_of_wrkbk.xls"). _
  Sheets(1).Activate
```

Often we are interested in referencing specific cells in a sheet. We have already seen how cells are referenced in the examples recorded with the macro recorder. To change the value of a specific cells we would write:

```
Application.Workbooks("Book1").Sheets(1). _
  Range("A1").Value = 68
```

The above statements is equivalent to instruction somebody to: Set the value of cell A1 on Sheet 1 in Workbook Book1 to 68. The dot notation syntax makes it a little more clear, what you actually mean.

Referencing an object or property of an object with a reference starting with Application. is often referred to as the fully qualified reference. This means that regardless of the context you state the reference in, it will always mean the same. Excel can also deal with implicit references. This may or may not sound complicated, but it is actually quite simple. If you do not state the fully qualified reference, VBA fills in the missing part of the reference based on the context. For instance:

```
Sheets(1).Range("A1").Value = 68
```

means the range A1 in Sheet 1 of the active workbook, and

```
Range("A1").Value = 68
```

means the range A1 in the active sheet of the active workbook. And just to set it straight.

```
Application.Workbooks("Book1").Sheets(1).Activate
Range("A1").Value = 68
```

is equivalent to the statement

```
Application.Workbooks("Book1").Sheets(1). _
  Range("A1").Value = 68
```

It may all seem a bit strange to you at the moment. But after working through a few examples it will all seem more clear.

3.3 Obtaining information about objects and properties

After having understood the basic principle of the object model, all you have to do is obtain some information about the names of all the objects and what properties and methods they have. There are in principle three different ways of obtaining this information. I recommend you use all of them.

1. Read the documentation.
2. Use the macro recorder.
3. Use the object browser in VBE.

3.3.1 VBA documentation

There is plenty of sources for information on the VBA object model. Just try typing "Microsoft Excel Objects" in the search facility of the VBE help files. One of the listed results should be a graphical tutorial of the objects.

When ever you have typed a VBA keyword in a module, you can place the cursor in the keyword and press **F1**. This should bring you to help on that keyword. Try the word "Application". The help screen should give you an option to list all the methods and properties of the object.

3.3.2 The macro recorder

The macro recorder is an excellent way of getting information about objects. If you want to manipulate the properties of a specific object, but you dont know the exact name of the object or the property you want to manipulate, you can record the action, and look at the code created by the recorder. Remember that the recorder is not the best programmer in the world, and you might need to change the code a little (or a lot). The recorder is a great learning tool, when you want to learn the names of specific objects and properties, there are however objects, than cannot be touched by the recorder, so you might still need to check the documentation.

3.3.3 The object browser

VBE has a built-in tool for locating objects and properties - The Object Browser. It is activated by pressing **F2** in the VBE. This object browser is built much like your file browser in windows or the project browser mentioned earlier. You may not find much use for the object browser at first, but when you get more comfortable with VBA you find that it can be a useful tool.

The object browser basically allows you to find information in two ways. By searching or by stepping through a hierarchy. Try searching for a few words like Range or Application or try to locate the word application in the hierarchy at the bottom of the window. Its much harder to explain how the object browser works than it is to actually use it, so just try it out.

4 Basic programming concepts

All programming languages have common features and concepts that must be understood to master programing. These common features might be a little different in the way they are put to use, but the basic concepts are still the same. This section deals with these basic programming features. Most of the information in this section could have been written for another programming language, but the examples are for VBA, and some of the details may be different in another programming language.

The information in this section is important. If you do not understand it, you should read it again or find other sources to obtain the same information.

4.1 Understanding variables and constants

4.1.1 Definitions

Variables (constants are variables that dont change) are an essential to any programming language. Many people have an idea of what a variable might be, and several thousand different definitions have been made to try and make it clear what a variable is. My definition would be: A variable is an identifier (a name) for a storage location (part of the computers memory) which holds a value of some sort. This may sound complicated, but it isnt really. It just means that you can use the name of the variable to reference a value in the computers memory, and you can use the same name to change the value of the variable. Constants are just variables that cannot be changed.

You may choose almost any name for your variables. The rules may differ a little from programming language to programming language, in VBA you must follow these rules:

- The name can consist of letters and numbers, but it must start with a letter.
- You cannot use space or periods in the name.
- VBA variable names are not case sensitive (”MyVar” is the same as ”myvar”’).
- In general do not include any non-letter or non-number characters.
- The name cannot be longer than 256 characters.
- You cannot use words already taken up by VBA. For instance ”Application”.

The following shows very basic use of variables:

```
MyVar = 2
MyVar3 = 101
Result = MyVar + MyVar3
TheText = "Variables can contain text"
```

The variable `Results` ends with the value 103, and `TheText` contains the text within the quotation marks. The variables are really not very useful here, but they will be.

4.1.2 Data types, declarations and scope

We have already seen that a variable is a name that contains a reference to a value. In the above example we also saw that this value does not have to be a number, it could also be some text. This means that the variables have different data types. Some programming languages require that you declare in advance what type of data a certain variable will contain. This is an optional feature in VBA, but I would recommend that you always declare the data types of your variables. This should be done for a number of reasons:

- It is good programming style to declare variables.

- It makes the code easier to read for others.
- If you always declare variables, VBA will tell you if you have misspelled a variable. These are the kind of errors it takes hours to find.
- Your macro will take up less memory.
- You code will run faster. You might not think of this as a problem, but you will be amazed how fast MS Office can run itself into the ground.
- VBA will check if you are putting the right type of data into the variable

VBA has a variety of built in data types. We will just mention a few of the most widely used here. Look in the help files for a complete list of data types. Below is a table of the most common data types. Dont worry if it seems a little confusing.

Data Type	Bytes Used	Value
Bolean	2	True or False
Integer	2	-32768 to 32767
Long	4	-2147483648 to 2147483647
Single (positive)	4	1.401298E-45 to 3.402823E38
Single (negative)	4	-3.402823E38 to -1.401298E-45
Double (positive)	8	4.94065645841247E-324 to 1.79769313486232E308
Double (negative)	8	-1.79769313486231E308 to -4.94065645841247E-324
String	1 per char	Example string
Object	4	Any defined object
Variant	Varies	Varies

If you introduce a variable name in your code without any prior declaration of the variable, VBA will automatically assume that this variable is of type Variant. This means that it can contain any type of data and VBA will automatically change the type of the variable when needed. This may sound like a really good idea. You would not have to worry about data types and VBA would do the work for you. In reality it is a very bad idea. I have already named a number of reasons why, but just imagine this. In the beginning of your code, you do some calculations, and the results end up in a variable. You want to use this result in a later calculation, but at this later point, you make a typing error and the name of the variable is misspelled. VBA would think you are introducing a new variable. Such a new variable would not contain any useful value and your macro would fail. Errors like that are very hard to find, and for that reason alone you should always declare your variables. If it is not clear to you why you should declare your variables, you can either just accept it or go back and read the above section again. The declaration of variables is done with the Dim statement. Dim is short for Dimension, dimensioning is another word for declaring. The following shows examples of variable declaration. The declarations must be made before the code where the variable is used.

```
Dim MyNumber As Integer
Dim theFloat As Double
Dim theName As String
```

Once the variable has been declared it can be used in the code, and VBA will give you an error message if you try to put the wrong type of data into your variable. Constants are declared with a Const statement:

```
Const InterestRate As Single = 0.1453
Const Temperature As Integer = 25
```

You may wonder what constants are good for when you cant change their values. You might as well hard code the values into the macro. But consider this. You are building a macro to do some analysis on some measurements. In the calculations you might use certain constants like temperature. If this value is used many times it is not a good idea to hard code the value. If you want to use the macro for another experiment with another temperature, you would have to change the value in a lot of places and you might forget some of them. Using constants also make your code more readable.

If you dont tell VBA otherwise, it will allow you to implicitly declare variables in your code. This means a new variable name in the code will automatically get the type Variant if it has not been declared. As mentioned this is a bad idea, so to force yourself to declare variables you should include the statement

```
Option Explicit
```

in the beginning of all modules (before declaring any Sub statements). You can also have VBE do this automatically by entering Tools→Options... and checking the box Require variable declaration. This only effects new modules. I can highly recommend this setting!

You may already wonder if the variables keep their values even when the macro has ended. Usually they dont. The computer would eventually run out of memory if variables stayed in the memory. Variables are visible to a certain part of the VBA environment. This part of the environment is know as the variables scope. Unless you tell VBA differently the variable will disappear when the VBA interpreter leaves the variable scope. There are in principle 3 different scopes in VBA:

- Procedure
- Module
- Global

Variables with procedure level scope have their declaration within a procedure (a Sub or macro). They disappear when the macro ends. Module level scope variables are declared at the beginning of the module before any macro code. For instance right after the Option Explicit statement. They are visible to all macros in that module, and they will keep their values even after a certain macro has ended. Dont use these unless you absolutely need them, since they take up memory even when the macros have ended. Global or public variables are visible to all modules in the workbook, and should be declared with a Public statement at the module level:

```
Public Temperature As integer
```

You should also use these with caution or not at all. Personally I have never really needed them, so you should be able to do without them for a while at least. A special case of procedure level variables are the static variable. They are declared in the procedure (or macro) and keep their value even after the macro has ended, but they are only visible to that procedure:

```
Sub myMacro()
Static Counter As integer
Counter = Counter + 1
End Sub
```

They are useful for keeping track of the number of times a certain macro has been called.

4.1.3 Object variables

Variables with an object data type have their values assigned in a little different way, and they deserve special attention. Object variables are variables containing a reference to a certain object. This could be a sheet, a chart, a range etc. The object variable would have the same properties and methods as the object itself, and changing the variable affects the object. To assign an object to an object variable you need to use the Set statement. The following example illustrates the use of object variables. They are also used elsewhere in this document.

```
Sub ObjTest()
  Dim theRange As Range
  Set theRange = ActiveSheet.Range("A1:C25")
  theRange.Value = 19
End Sub
```

4.1.4 Built-in constants

You are allowed to declare constants yourself as mentioned above, but VBA also has a wide range of built-in constants. The purpose of these constants is the same as the constants you declare yourself. They can be used to make the code easier to read, and they allow certain values to be changes, and the code would still work. We have already seen an example of these built-in constants in the example with changing the page orientation. We saw that the page orientation could be changed to landscape with the following statement:

```
ActiveSheet.PageSetup.Orientation = xlLandscape
```

In this case xlLandscape is a built-in constant in Excel. It really holds a value. If you want to know the value of a built in constant try the following sub:

```
Sub ShowValue()
  MsgBox xlLandscape
End Sub
```

Running this sub should illustrate that xlLandscape really holds the value 2. Other built-in constants hold other values. This means that me could also have changed the page orientation to landscape with the macro:

```
ActiveSheet.PageSetup.Orientation = 2
```

The first version of the macro is much easier to read, and the built-in constants should always be preferred over hard coding the numbers.

4.2 Functions and subroutines

The terms functions, procedures, subroutines and macros have already been mentioned, but not properly defined. In principle all of the mentioned terms cover the same concept, but just to make things clear we will define them.

4.2.1 Definitions

A function is a general term used in programming. A function is a structure that requires some sort of data and based on this data, it perform a series of operations and finally returns a value. We know functions from the spreadsheets. For instance the "Average" function takes a series of numbers (or ranges of numbers) as its arguments (we call the data needed by the function arguments) and returns the average of these values. Some functions need several arguments, others can work without arguments (the Rand worksheet function), but usually they return one value and one value only. There are examples of function that dont return values. We have already seen the MsgBox function. This function opens a dialog box and displays a message to the user.

Subroutines, procedures and macros are the same thing. A subroutine is defined with the Sub keyword. A subroutine is a procedure, and the subroutine can be executed as a macro in Excel. So we will use the terms where they seem appropriate but they mean the same thing.

4.2.2 Declarations

Previous examples have illustrate how a subroutine or a macro is defined in a VBA module. The Sub keyword declares the macro and it is followed by the name of the macro. A list of parameters can be supplied in the parenthesis after the name of the macro. The statements that should be executed by the macro follow the sub declaration and the macro ends with an End Sub statement. Arguments cannot be supplied for macros executed from Excel. The following shows the declaration of a macro:

```
Sub MacroName()
  [Statements]
End Sub
```

Functions can also be declared. As mentioneed they need some arguments (in some cases none) and they return one value and one value only. Functions cannot be executed from Excel as macros, but they can be called from running macros or used as worksheet functions in Excel. This allows you to design you own worksheet functions for some calculation that you perform often. The following example show a declaration of a macro, and a function. The macro calls the function, and uses the return value in a display box.

```
Sub TryAdd()
  MsgBox AddNumbers(34, 21)
End Sub
```

```
Function AddNumbers(NumberOne As Integer, NumberTwo As Integer)
  AddNumbers = NumberOne + NumberTwo
End Function
```

The example also illustrate how the functions return their value. An internal variable in the function with the same name as the function holds the return value. The value of this variable is returned when the function exits. After declaration of the function you should be able to use it as a normal worksheet function in your worksheet. Try entering the formula:

```
=AddNumbers(34;21)
```

The function also exists in the function wizard in the user defined category.

4.2.3 Using Worksheet functions

The functions normally used in the worksheets can also be used in macros, unless VBA has a built-in function that does the same job. An example could be calculation the mean of a Range:

```
Sub CalcAvg()
  Dim theRange As Range
  Dim result As Double
  Set theRange = Range("A1:M100")
  result = Application.WorksheetFunction.Average(theRange)
  msgBox result
End Sub
```

You must use the qualified reference Application.WorksheetFunction to access the worksheet function.

4.3 Controlling program flow

One of the advantages of programming your own macros is the ability to automate tasks. You can instruct Excel to do the same task over and over again. To do this you need to be able to control the program flow. When you create macros with the macro recorder you have no control over program flow. If you want Excel to do the same task twice you have to record it twice or execute the macro twice. This might work if you want to do something two or three times, but it becomes annoying quite quickly. You may also be interested in controlling what VBA does based on certain values (a classic ”if something then do something else” problem). This section will illustrate how to deal with these problems in VBA.

4.3.1 GoTo statement

The GoTo statement is the simplest control structure in VBA. It basically allows you to jump to labels you insert in the code. An example:

```
Sub GoToExample()
  GoTo MyLabel
  MsgBox("I have to execute every line of the code")
```

```
MyLabel:
  MsgBox("I have skipped part of the code")
End Sub
```

Running this macro should illustrate how the GoTo statement is used. It should also illustrate why you should avoid using the statement. Your code becomes very messy and difficult to read, and you tend to get unexplainable behavior in you macros, because you loose track of the program flow. The GoTo statement has one useful pupose. This is illustrated in the section on error handling. If you cant find an alternative for the GoTo statement, think harder!

4.3.2 If-Then-Else statements

If statements are the most widely used control statements in VBA. If you only learn one control statement this should be the one. In the simplest form it is very easily understood, but nested if statements can get complicated as we shall see. The basic form of the if statements is:

```
If [Condition] Then
  [Statements]
Else
  [Statements]
End if
```

An example of the use of an If statement could look like this:

```
Sub ExampleIf()
  Dim name As String
  Dim userName As String

  name = Application.userName
  userName = InputBox("Enter you name, please!")
  If name <> userName Then
    MsgBox ("This is not your computer. Username is " & name)
  Else
    MsgBox ("This is your computer")
  End If
End Sub
```

This example also illustrates other features of VBA that cannot be recorded with the macro recorder. The InputBox function is very useful for getting information from the user. It displays a string (some text) to the user and returns the information typed by the user. It also illustrate how to extract the username. It should be mentioned that this macro is not complete. Just try clicking cancel in the dialog box.

If statements can be nested. This means that you can have an If statement inside another if statement. The general form would be:

```
If [Condition] Then
  If [Other condition] Then
    [Statements]
  Else
```

```
    [Statements]
  End If
Else
  [Statements]
End if
```

You can nest as many if statements inside each other as you please, but it gets complicated and you might loose track of what you are doing. Generally avoid more than three or four nested If statements.

You can also modify the If statements with an ElseIf. Its much harder to explain than to understand. The general form is:

```
If [Condition] Then
  [Statements]
ElseIf [Condition] Then
  [Statements]
Else
  [Statements]
End if
```

The following example illustrates the use of the structure:

```
Sub Greeting()
  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
```

Look for other examples of If statements in this documents, and play around with the structure until you feel you are comfortable with it. As mentioned this is a very important control structure.

4.3.3 Select Case structure

The If-Then-Else statements can as mentioned be nested and make is possible to choose between many different possibilities. They tend to get a bit complicated, and the code tends to look messy if you use to many nested If statements. When you need VBA to choose between lots of different options, the Select Case structure is a better solution. The general form is:

```
Select Case [Expression]
  Case [Expression]
    [Statements]
  Case [Expression]
    [Statements]
  Case Else
    [Statements]
End Select
```

In a practical example it could look like this.

```
Sub WhoAreYou()
  Dim username As String
  username = Application.UserName
  Select Case username
    Case "Bill Gates"
      MsgBox "You are rich"
    Case "Michael Schacht Hansen"
      MsgBox "You teach VBA"
    Case Else
      MsgBox "You are unknown, Welcome"
    End Select
End Sub
```

You should use the Select Case structure when making choices based on more than three or four possibilities. It makes the code easier to read, and prevents errors. Select Case Structures can also be nested, and you can have If structures nested inside the Select Case structures.

4.3.4 For-Next loop

VBA provides a number of different loop structures. The loop structures allows you to ask VBA to repeat the same task for a certain number of times. This cannot be done with the recorder. The first and perhaps the most widely used of these loop structures is the For-Next loop. This structures uses a counter to determine the number of times a certain set of statements have been executed. The general form is seen below:

```
For counter = startValue To endValue [Step stepValue]
  [Statements]
Next counter
```

The step value is optional. If no step value is provided, the counter value is incremented by one on each run through the loop. A practical example:

```
Sub FillCells()
  Dim counter As Integer
  For counter = 0 To 100 Step 2
    ActiveCell.Offset(counter, 0).Value = Rnd()
  Next counter
End Sub
```

I think running the example pretty much explains what it does. If statements, Select Case statements etc. can be nested in the loops. This way decisions can be made on each run through the loop. If for some reason you want to exit the loop based on a decision include an ”Exit For” statement.

4.3.5 Do-While and Do-Until loop

Another example of a loop structure is the Do-While:

```
Do While [Condition]
  [Statements]
Loop
```

Again a practical example:

```
Sub ExampleDo()
  Do While ActiveCell.Value
    ActiveCell.Font.Bold = True
    ActiveCell.Offset(1, 0).Select
  Loop
End Sub
```

A variation of the Do-While loop, is the Do-Until. The above example could have been made with a Do-Until structure:

```
Sub ExampleDo()
  Do Until ActiveCell.Value = Empty
    ActiveCell.Font.Bold = True
    ActiveCell.Offset(1, 0).Select
  Loop
End Sub
```

I think that the examples explain themselves. Enter them in a module and try them out. Change some of the statements and observe the effect.

4.3.6 Error handling

It was mentioned that the GoTo statement can be used in a constructive fashion to handle errors. The errors in question are the so-called intentional error. The principle is as follows. You ask VBA to perform a loop, knowing that sooner or later it will fail for some reason. When that happens you use a GoTo statement to exit the loop without the whole macro failing. The code for such an error handling could look like this:

```
Sub ReadFiles()
  Dim x As Integer
  Dim fileName as String

  On Error GoTo NoMoreFiles

  For x = 1 To 1000
    fileName = "File" & x
    Workbooks.OpenText _
    Filename:=fileName, _
    Origin:=xlWindows, StartRow:=1, _
    DataType:=xlDelimited, TextQualifier:= _
    xlDoubleQuote, ConsecutiveDelimiter:=False, _
    Tab:=True, Semicolon:=True, _
    Comma:=False, Space:=False, Other:=False, _
    FieldInfo:=Array(1, 1)
  Next x
```

```
NoMoreFiles:
  MsgBox "There are no more files"

End Sub
```

This example opens a series of text files (named File1, File2...) until there are no more files, in which case the OpenText statement would fail, and cause and error. This error leads to a GoTo statement, and we can handle the error in a suitable way. The OpenText part of the macro may seem complicated, but you can easily make a statement like this. Just record the action, and modify the statement. In this case change the file name to a variable.

5 Working with Ranges

The handling of ranges might not be a large topic in VBA, but most of the macro programming work you will do will somehow involve the use of ranges. For this reason the Range object deserves a section of its own.

5.1 Referencing ranges

Ranges can be referenced in a number of different ways. The simplest way is to use the address of the range. This would allow you to reference a range like this (we have not used the fully qualified reference):

```
Range("A1:B6")
```

Or if the range has a name:

```
Range("RangeName")
```

The above statements reference a range in the active sheet. If you wanted a range in another sheet or even in another workbook, you would need to include the name of the workbook and sheet in the reference. This would look like this:

```
Workbooks("MyWorkbook.xls").Sheets(1).Range("A1:A6")
```

You can reference entire columns or rows with statements like:

```
Range("A:B")
Range("1:3")
```

Ranges can also be non-continuous (some function cannot use non-continuous ranges):

```
Range("A1:B6,D5:M8")
```

As mentioned the Ranges can be assigned to object variables and referenced through these:

```
Dim theRange As Range
Set theRange = Range("A1:B6,D5:M8")
```

There are other ways of referencing Ranges. You might be interested in using row and column numbers instead of actual addresses. This can be accomplished with the Cells property. This property belongs to the worksheet object, and it returns a range object containing one cell. For instance the statement

```
Cells(2,3)
```

Returns the single cell range with row number 2 and column number 3, in other words the Range C2. To use the Cells property to design multi-cell ranges, combine it with the Range keyword like this:

```
Range(Cells(2,3), Cells(4,4))
```

This is equivalent to

```
Range("C2:D4")
```

Another possibility is to use the Offset method of ranges. This allows you to reference Ranges with a specific offset from a defined range. An example:

```
Range("A1").Offset(2,2)
```

This would be equivalent to

```
Range("C3")
```

This should give you an idea of the many different ways to reference Range objects.

5.2 Properties of Range objects

Range objects have many different properties, but some are more useful than others. This is just a quick walk through some of the more important ones.

5.2.1 Value property

The value of a cell can be read with the Value property. You can set the value of several cells at the same time with the value property of a multi-cell range. Try the following statements:

```
MsgBox Range("C3").Value
```

or change the value in several cells with

```
Range("C3:K100").Value = 1000
```

Value is also the default property of Excel Ranges so the above could have been accomplished with

```
Range("C3:K100") = 1000
```

But I think the Value property should be included, since it makes the code easier to read.

5.2.2 Count property

The number of cells in a range can be counted:

```
Range("C3:K100").Count
```

Alternatively the rows or columns can be counted with:

```
Range("C3:K100").Rows.Count
Range("C3:K100").Columns.Count
```

5.2.3 Font and Interior property

Font and Interior properties return Font and Interior objects. These allow you to do a lot of formatting on cells. The following are just examples of what can be done. The are lots of other options. Please check the object browser to see what can done. To manipulate the Font you could write:

```
Range("A1").Font.Bold = True
```

This would set the font to bold. The Interior color of a cell can be set like this:

```
Range("A1").Interior.Color = RGB(255,0,0)
```

This would set the color of the cell interior to red. Try other options.

5.2.4 Entering formulas in cells

Ranges have a Formula property. This property can be manipulated like most other properties. First of all you might be interested in checking if a cell has a formula. This can be done with the HasFormula property:

```
MsgBox Range("A1").HasFormula
```

Based on this test you may or may not be interested in entering a formula:

```
If Range("A1").HasFormula Then
  Range("A1").Formula = "=AVERAGE(A1:C17)"
End If
```

5.2.5 Select method

Select if often used to make some cells the active cells. After this the cells can be referenced with the word Selection. An example:

```
Dim theRange As Range
Range("A1:C4").Select
Set theRange = Selection
```

5.2.6 Copy and Paste

The copy and paste methods are used as they are when you work normally in your spreadsheet. You select a range, copy, put the cursor somewhere else and paste. When you record such a procedure it looks somewhat like this:

```
Sub Macro1()
  Range("A1:C25").Select
  Selection.Copy
  Range("K1").Select
  ActiveSheet.Paste
End Sub
```

As mentioned the macro recorder is not exactly brilliant, and the same operation could been done with the following:

```
Sub Macro1()
  Range("A1:C25").Copy Range("K1")
End Sub
```

That saves you three lines, makes the code easier to read, and you feel smarter than the computer.

5.2.7 NumberFormat property

The NumberFormat property is really simple and very useful. To change the number format of a Range to percent with two decimals, use the statement:

```
Range("A1").NumberFormat = "0.00%"
```

Its easy and useful. If you have difficulty figuring out how to specify the number format in the correct way, just try recording when you change the format for a Range, and use the format specifier that was recorded.

5.3 Range methods

This will also just be a brief summary of the most useful methods of the Range object.

5.3.1 Clear and Delete methods

I have to admit that I almost never use these methods, but they are always mentioned in VBA books, and since there is a difference in the way that they work, they should also be mentioned here. With the statement

```
Range("A1:C25").Clear
```

you remove the contents of the range, with the statement

```
Range("A1:C25").Delete
```

you remove the contents and the cells. Excel shifts the remaining cells to fill in the space. You can specify the direction Excel should move the cells. An example:

```
Range("A1:C25").Delete xlToLeft
```

The built-in constant xlToLeft could be replaced with another of the direction constants (xlToRight, xlUp, xlDown).

6 Chart objects - programming charts

6.1 Introduction

The chart object is the largest most complicated object in excel visual basic programming. It is also very powerful and versatile. Knowing how the chart objects work will help you to create macros that let you visualize your data very quickly, and that can be very useful.

This section is a more systematic description of the most widely used elements of the chart object. It describes how to create a chart from scratch. The features mentioned here are only a fraction of the ones available to you, check the object browser or books on writing excel macros for further information.

6.2 Creating a chart object

A chart can exist in two different ways in excel. The chart can either be a standalone sheet with nothing but a chart on it or it can be a ChartObject embedded in a sheet. This may sound confusing (and it probably is), but to make it really simple you have to choices: The chart has a sheet of its own or it is placed in another sheet. Once created there is not much difference in how they are manipulated. In either case the most elegant way to create charts is to simultaneously create them and store a reference to them in an object variable and then access all the properties of the chart through that variable afterward. To create a standalone start you declare a Chart variable and then create the instance of the chart. The following example creates a new, empty chart on a new sheet and names the sheet My Chart:

```
Dim ch As Chart
Set ch = ThisWorkbook.Charts.Add()
ch.Name = "My Chart"
```

To create the embedded chart you need to create a new ChartObject in the sheet, and access the chart through this ChartObject. The following example creates a ChartObject in the active sheet and places it at the coordinates (0,0,100,100):

```
Dim co As ChartObject
Set co = ActiveSheet.ChartObjects.Add(0,0,100,100)
```

Elements in the chart of the chart object can now be accessed through the ChartObject. The following example sets the chart type to xlLine for both the embedded and the standalone chart:

```
ch.ChartType = xlLine
co.Chart.ChartType = xlLine
```

Everything from here on should be the same for the embedded and the standalone chart.

6.3 Formatting the chart

To illustrate all the formatting features in the Chart object would be beyond the scope of this document, but an illustration of how a chart is created and formatted with its most basic features can be done quite easily within a resonable amount of space. In this example we will create an embedded scatter plot and change some of the features of the plot. First the ChartObject has to be created and a ChartType must be chosen:

```
Dim co As ChartObject
Set co = ActiveSheet.ChartObjects.Add(0,0,300,400)
co.Chart.ChartType = xlXYScatterLines
```

Running this routine will just create a blank area embedded in the active sheet. A blank chart is not of much use, so we should add some data. The following lines will add two data series, and do some formatting afterward:

```
co.Chart.SeriesCollectiOn.Add _
  Source:=ActiveSheet.Range("A1:A25"), _
  Rowcol:=xlColumns

co.Chart.SeriesCollection.Add _
  Source:=ActiveSheet.Range("B1:B25"), _
  Rowcol:=xlColumns

With co.Chart.SeriesCollection(1)
  .XValues = ActiveSheet.Range("C1:C25")
  .Name = "1st Series"
  .MarkerStyle = xlNone
  .Border.Weight = xlMedium
End With

With co.Chart.SeriesCollection(2)
  .XValues = ActiveSheet.Range("C1:C25")
  .Name = "2nd Series"
  .Border.LineStyle = xlNone
  .MarkerForegroundColor = 1
  .MarkerBackgroundColor = 1
End With
```

Actually the code explains itself, but just to set things straight. The first two statements (the statements are broken into 3 lines each) add the two data data series, by telling where two find the data and if the data are in columns or rows of the specified range. This is much better illustrated if you just change it to xlRows and see the difference. The next group of statements formats the first data series. X-values are specified (this is needed with a scatterplot), and the markers are removed, and the line made thinner. The last group of statements formats the last data series by removing the line and setting the marker color to black. The are tons of other attributes to test and set in the data series. Look them up in the object browser and play with them.

The next thing that we should do is to format the chart axes to suit our needs. By default value and category axes are added, but just to illustrate the procedure we will add them in the following example. We will also add titles on the axes illustrating what the axes show. Gridlines are added and formated through the axes property of the chart object and we will in the following example set the grid lines (in this case turn them off).

```
'Add axes
'This is the default setting
With co.Chart
  .HasAxis(xlCategory, xlPrimary) = True
  .HasAxis(xlCategory, xlSecondary) = False
  .HasAxis(xlValue, xlPrimary) = True
  .HasAxis(xlValue, xlSecondary) = False
End With

'Formatting the category axes
With co.Chart.Axes(xlCategory)
  .HasTitle = True
  .AxisTitle.Caption = "Category Title"
  .HasMajorGridlines = False
  .HasMinorGridlines = False
End With

'Format the Value axes
With co.Chart.Axes(xlValue)
  .HasTitle = True
  .HasMajorGridlines = False
  .HasMinorGridlines = False
  With .AxisTitle
    .Caption = "Value title"
    .Font.Size = 6
    .Orientation = xlHorizontal
    .Border.Weight = xlMedium
  End With
End With
```

The last thing we need to do is to format the chart area (the whole area) and the plot area (the area where the chart is actually drawn. In this case we will set both to be white (suitable for printing on a black and white printer). Also we should give the chart a title (at least thats what we will do in this situation).

```
co.Chart.ChartArea.Interior.Color = RGB(255,255,255)
co.Chart.PlotArea.Interior.Color = RGB(255,255,255)

co.Chart.HasTitle = True

With co.Chart.ChartTitle
  .Caption = "My new chart"
  .Font.Size = 14
  .Font.Bold = True
End With
```

This would create a chart and format it to your needs. We have only just scratched the surface in this section, but this standard approach should get you going and hopefully you will feel comfortable enough to venture on and locate other attributes of the chart object on your own. Having ready made macros for creating and formatting charts can save you lots of time.

It is very annoying (at least to me) to do the same formatting over and over because Microsoft didnt format the charts to be printed or viewed for that matter.

7 Examples - Using VBA

VBA is easier understood through practical examples. This sections solves a variety of problems using the features illustrated in the previous sections.

7.1 Statistics Creating Bland-Altman plots

Bland-Altman analysis is used to compare two series of measurements of the same parameter. The difference between the two series illustrates a methods ability to reproduce or repeat a certain measurement. By calculation the 95 percent limits of agreements we are able to get an idea of the change to a given stimulus that could be detected with the method. Bland-Altman analysis includes:

- Calculation of means.
- Calculation of differences.
- Calculation of mean difference.
- Calculation of standard deviations of differences.
- Calculation of 95 percent limits of agreement for the differences.
- Displaying the information in a chart.

We will do all the calculations one step at a time, and save all the calculation results in a separate sheet before drawing the chart. The macro should be designed in such a way that the user should select two columns of data containing the two series and then run the macro for the analysis. The first tasks are then storing the data in a Range object and inserting a new sheet for the calculations:

```
Option Explicit

Sub BlandAltman()
  Dim dataRange As Range
  Set dataRange = Selection
  Sheets.Add after:=Sheets(Sheets.Count)
  ActiveSheet.Name = "Bland-Altman"
```

The next step is to insert some headlines for the calculations in the new sheet:

```
  Range("A2").Value = "Mean"
  Range("B2").Value = "Difference"
  Range("C2").Value = "Mean difference"
  Range("D2").Value = "SD"
  Range("E2").Value = "Mean diff + 2 SD"
  Range("F2").Value = "Mean diff - 2 SD"
  Range("A2", "F2").Font.Bold = True
```

Now we need to calculate the mean, and difference of each data pair in the range, and insert this mean, and difference in the Mean and Difference columns of our result sheet:

```
Dim t As Integer
For t = 1 To dataRange.Rows.Count()
  Cells(t + 2, 1).Value = _
    Application.WorksheetFunction. _
    Average(dataRange.Cells(t, 1).Value, _
    dataRange.Cells(t, 2).Value)
  Cells(t + 2, 2).Value = _
    dataRange.Cells(t, 2).Value - _
    dataRange.Cells(t, 1).Value
Next t
```

The next step is to insert Mean Difference, Standard Deviation and 95 percent limits of agreement in the appropriate rows. The last line adjusts the width of the cells to allow the results to be viewed properly.

```
Range(Cells(3, 3), _
  Cells(dataRange.Rows.Count() + 2, 3)).Value = _
  Application.WorksheetFunction. _
  Average(Range(Cells(3, 2), _
  Cells(dataRange.Rows.Count() + 2, 2)))

Range(Cells(3, 4), _
  Cells(dataRange.Rows.Count() + 2, 4)).Value = _
  Application.WorksheetFunction. _
  StDev(Range(Cells(3, 2), _
  Cells(dataRange.Rows.Count() + 2, 2)))
  Range(Cells(3, 5), _
  Cells(dataRange.Rows.Count() + 2, 5)).Value = _
  Cells(dataRange.Rows.Count() + 2, 3) + _
  2 * Cells(dataRange.Rows.Count() + 2, 4)

Range(Cells(3, 6), _
  Cells(dataRange.Rows.Count() + 2, 6)).Value = _
  Cells(dataRange.Rows.Count() + 2, 3) - _
  2 * Cells(dataRange.Rows.Count() + 2, 4)
  Cells.EntireColumn.AutoFit
```

Now we should visualize our data with a Bland-Altman plot. In this plot, we plot the differences as a function of the mean, and we place a horizontal bar at the mean difference and at the 95 percent limits of agreement. The first step is to define the chart and an object variable to reference it. In this case we embed the chart in the result sheet.

```
Dim ch As ChartObject

Set ch = Worksheets("Bland-Altman"). _
  ChartObjects.Add(100, 30, 400, 250)

ch.Chart.ChartType = xlXYScatterLines
```

Then we need to add all our data series. We will format them later. I admit we could have added the data in a loop and avoided repeating the same statement, but with a little copy and paste this is actually faster than keeping track of a variable, and I know than we would not need to add other series later.

```
ch.Chart.SeriesCollection.Add _
  Source:=ActiveSheet.Range(Cells(3, 2), _
  Cells(dataRange.Rows.Count() + 2, 2)), _
  Rowcol:=xlColumns

ch.Chart.SeriesCollection.Add _
  Source:=ActiveSheet.Range(Cells(3, 3), _
  Cells(dataRange.Rows.Count() + 2, 3)), _
  Rowcol:=xlColumns

ch.Chart.SeriesCollection.Add _
  Source:=ActiveSheet.Range(Cells(3, 5), _
  Cells(dataRange.Rows.Count() + 2, 5)), _
  Rowcol:=xlColumns

ch.Chart.SeriesCollection.Add _
  Source:=ActiveSheet.Range(Cells(3, 6), _
  Cells(dataRange.Rows.Count() + 2, 6)), _
  Rowcol:=xlColumns
```

Now we format the series. In this case each series need individual formating, since some should be lines and some points, and we want different thickness and so on.

```
With ch.Chart.SeriesCollection(1)
  .XValues = ActiveSheet.Range(Cells(3, 1), _
    Cells(dataRange.Rows.Count() + 2, 1))
  .Border.LineStyle = xlNone
  .MarkerForegroundColor = 1
  .MarkerBackgroundColor = 1
  .Name = "Difference"
End With

With ch.Chart.SeriesCollection(2)
  .XValues = ActiveSheet.Range(Cells(3, 1), _
     Cells(dataRange.Rows.Count() + 2, 1))
  .MarkerStyle = xlNone
  .Border.Weight = xlMedium
  .Border.Color = 1
  .Name = "Mean diff."
End With

With ch.Chart.SeriesCollection(3)
  .XValues = ActiveSheet.Range(Cells(3, 1), _
    Cells(dataRange.Rows.Count() + 2, 1))
  .MarkerStyle = xlNone
```

```
    .Border.Weight = xlThick
    .Border.Color = 1
    .Name = "Mean diff + 2SD"
  End With

  With ch.Chart.SeriesCollection(4)
    .XValues = ActiveSheet.Range(Cells(3, 1), _
      Cells(dataRange.Rows.Count() + 2, 1))
    .MarkerStyle = xlNone
    .Border.Weight = xlThick
    .Border.Color = 1
    .Name = "Mean diff - 2SD"
  End With

  ch.Chart.Axes(xlValue).HasMajorGridlines = False
  With ch.Chart.Axes(xlCategory)
    .MinimumScale = Int(Application.WorksheetFunction. _
      Min(Range(Cells(3, 1), _
      Cells(dataRange.Rows.Count() + 2, 1))) - 2)
    .MaximumScale = Int(Application.WorksheetFunction. _
      Max(Range(Cells(3, 1), Cells(dataRange.Rows.Count() + 2, 1))) + 2)
  End With

  ch.Chart.PlotArea.Interior.ColorIndex = xlNone
  ch.Chart.PlotArea.Border.LineStyle = xlNone
End Sub
```

The category axes of the chart is formated to not start with the value 0. This is to avoid funny looking charts if the mean values are very high. Thats about it. Try following the example. When you meet a function or keyword you havent seen before, just enter it in a module, place the cursor over the word and press **F1**. You will then get help on that word.

Visual Basic for Applications (VBA) is a powerful language built on top of popular Microsoft Office applications like Excel, Access, and Outlook. It allows developers to write procedures called macros that perform automated actions. Anything that you can do in Excel, you can automate with VBA! Over the course of more than 18 hours of content, we'll cover VBA from the ground up, beginning with the fundamentals and proceeding to advanced topics including: The Excel Object Model. The Visual Basic Editor. Objects and Methods. Variables and Data Types. Conceptual overviews, programming tasks, samples, and references to help you develop Excel solutions.Â Object model reference: Provides reference materials for the Excel object model. Graph Visual Basic reference. See also. Excel (Office client development). Support and feedback. Have questions or feedback about Office VBA or this documentation? Please see Office VBA support and feedback for guidance about the ways you can receive support and provide feedback. Visual Basic for Applications is simplified impementation of Microsoft's programming language Visual Basic 6. VBA is used for automation operations in applications such as Microsoft Excel and also expands it's capabilities. You can find VBA in all Microsoft Office applications, AutoCAD, WordPerfect and many others. Visual Basic for Application appeared in Excel (Excel version 5) in 1994, earlier (up to version 4) only allow macros to automate those tasks that you can perform using the keyboard. Now