Session 13

Using Macros I

Function Macros in Visual Basic for Applications

Last revised: July 6, 2011

Review of Last Time: Service Systems

- A Service System is a facility containing servers, customer entry and exit facilities, and a waiting facility
- The simplest system is single-stage, single-server
- When arrivals are Poisson distributed, and service completion is exponentially distributed, we can predict system performance

What Are Macros?

- Macros are small programs you write to add capability to Excel
- There are two macro languages in Excel:
  - Excel 4 Macro Language (XLM)
  - Visual Basic for Applications (VBA)
- Why two?
  - VBA was introduced in Excel 5.0; XLM was the only language before that
  - Microsoft wants to discontinue XLM
  - Customers have heavy investments in XLM
  - Both are provided for the time being
- Which to use?
  - XLM is easier to learn and write
  - Documentation and help for XLM are not available in Excel 97 or later
  - XLM documents at Microsoft.com
  - VBA is faster and more powerful, and it is documented

Function Macros vs. Command Macros

Command Macros

- Can accept arguments
- Can be written from scratch
- Can be recorded
- Can be attached to menu items, buttons, etc.
- **Cannot** return values
- Can perform actions
  - Set font
  - Open a file

Function Macros

- Can accept arguments
- Can be written from scratch
- **Cannot** be recorded
- **Cannot** be attached to menu items, buttons, etc.
- Can return values
  - Convolve
  - EOQ
- Can perform actions

We'll do only a few simple examples of function macros

Why Write a Command Macro?

- Write a macro to do something you do often
- Modularize knowledge of the technique to save costs
  - With no macro, you must train everyone to perform the needed actions
  - When you train everyone, everyone carries a copy of the technique in their brains
  - If the technique changes, each mental copy must be updated (and stay updated)
  - With a macro, there is only one copy
- Reduce training for people who use your model
  - Automate a complex or error-prone procedure
  - Reduce the volume of what you must tell people
  - Maintain flexibility of the actual technique employed
- Capture a complex technique
  - Encapsulate the procedure by writing a command macro to do it
  - The knowledge of what it does is captured in the macro itself

Why Write a Function Macro?

- Lower maintenance costs
- Improved readability
- Improved performance
- Increased ability to “reuse” structures
  - If you implement a computation in worksheet cells, you must copy and paste to reuse the technique
  - Encapsulating a way of computing a result allows you to reuse the technique without making a copy
  - If you later change the technique slightly, you don’t have to chase around and fix all the copies — modularity.

Use macros to avoid showing intermediate computations. This makes your worksheets easier to read and maintain.

We'll Work with Function Macros Only

- Command macros
  - Require vast knowledge of Excel's user interface
  - Require more understanding of VBA
  - Are less useful in implementing modeling techniques
  - Are more useful for building tools
- Function macros
  - Useful for reducing work of modeling (example: Convolve)
  - Simplify managing the modeling organization
  - Illustrate powerful concept of software engineering: modularity

Terminology

- A scalar macro is a function macro that returns a single value
- An array macro is a function macro that returns a rectangular array of values

This session is about scalar macros – next time we'll tackle array macros

Examples of VBA function macros


```
Function EOQ(demand As Double, unitPrice As Double, _
                        orderCost As Double, carryCost As Double) As Double
   EOQ = Sqr(2 * demand * orderCost / (unitPrice * carryCost))
End Function
```

```
Function GeometricMean(number1 As Double, number2 As Double) As Double
   GeometricMean = Sqr(number1 * number2)
End Function
```

A little bit of notation

- VBA line continuation character is underscore (_)
  - Use a line continuation character whenever you break a statement across lines
- VBA comment character is single-quote (')

Example

- You expect to do lots of depreciation calculations in your company and you want a function macro
- Pattern it after the depreciation formula that supports the Ripple Principle (Session 10)
- Here's the macro:

```
Function Depreciation(firstPeriod As Object, nPeriods As Integer) As Double
    If Application.Caller.Column - firstPeriod.Column + 1 <= nPeriods Then
        Depreciation = 1 / nPeriods
    End If
End Function
```

Now let's learn how to write it.

- Note: You'll need access to the Developer Ribbon. See the reading "Excel Macros in Visual Basic for Applications"

Modules

- VBA macros reside in modules within a workbook
- To write a macro, create a module (or use an existing one):
  - Excel 2007+: Developer>Visual Basic
  - Excel 2000-2004: Open the VB Editor using the menu command Tools>Macro>Visual Basic Editor
  - Choose the menu command View>Project Explorer to open the Project Explorer
  - Select the file to which you want to add a module
  - Choose the menu command Insert>Module
- The module opens; you are now ready to type the macro
- Hint: The VB Editor pesters you every time you make a mistake. This can be annoying.
  - To turn this off in Windows (Mac):
  - Choose the menu command Tools>Options... (Excel>Preferences…)
  - Editor tab of the Options (Preferences) dialog: uncheck Auto Syntax Check
- One last thing: module options

Module options

- Module options are directives to the VB compiler
- They are somewhat technical, but fortunately we need only two
  - Tell the compiler to require variable type declarations
  - Tell the compiler to number array indexes starting at 1 instead of 0

```
Option Explicit
Option Base 1
```

- These statements are inserted at the top of the module

How to build a macro to do depreciation

- Create a module
- Write a function macro in the module.
  - Start with function statement, which includes the function name and its argument list
  - Ends with End Function, which is inserted automatically for you by the editor.

```
Function Depreciation(arguments)
        <Local variable declarations>
        <Computations>
End Function
```

- The body of the function definition contains two parts:
  - Variable declarations
  - Computations

Type declarations

- You get better performance and some error detection if you declare data types:
  - Arguments
  - Return values
  - Variables
- Variable Types
  - Variant (the default)
  - Integer (<32k)
  - Long (>32k)
  - Double (floating point)
  - Boolean (true, false)
  - Object (Excel entity such as a range): Object, Range, Worksheet, Workbook, Chart…
  - String (text)
  - Others: see on-line help

Return value type declaration

- Follows the argument list
- In the form: As <data-type>
- Example:

```
Function Depreciation(arguments) As Double
        <Local variable declarations>
        <Computations>
End Function
```

Argument variable declarations

- In the form: "As <data-type>" following the name of each argument
- Example:

```
Function Depreciation(firstPeriod As Object, nPeriodsAs Integer) As Double
        <Local variable declarations>
        <Computations>
End Function
```

Local variable declarations

- A local variable declaration is of the form

```
Dim Var1 As Type1, Var2 As Type2, ...
```

- Example
  - Dim answerArray, i as integer, j as integer
    declares answerArray as a variant, and i and j as integers.
- You can have as many as you want on a line
- You can have as many lines as you want, but they must be at the beginning of the function body.
- We don't need any for our function "Depreciation"

Returning a value

- To return a value from a function use an assignment statement
- Assign the function's name a value equal to what you want to return

```
Function Depreciation(firstPeriod As Object, nPeriodsAs Integer) As Double
    <Local variable declarations if needed>
    Depreciation = <some expression>
End Function
```

- Although this usually is the last statement, it might not be--it could be anywhere
- When the return value is of type Object, the assignment statement is of the form

```
Set <variable-name> = <object>
```

A simple example

- A simple example of a function macro
  - Takes one argument
  - Returns it

```
Function DoNothing(argRange As Object) As Object
        Set DoNothing= argRange
End Function
```

Where we are so far

- Created a module
- Inserted module options
- Defined the function
- Declared argument types and return types
- Returned the value
- Still to do:
  - Figure out what period number we're in
  - Determine what the value is we want to return

Figure out what period number we are in

- Each cell will contain a call to the function macro Depreciation
- The function will then figure out the period number
- It needs to know:
  - Where the caller is (i.e., what cell invoked the function macro)
  - Where the first period is
  - The period number is: Column(caller)-Column(first period)+1
- To find the location of the cell that invokes a function, ask Excel (it "knows")
- In VBA for Excel, Excel is called "Application"
  - It has a property named "Caller" that holds an object that specifies the caller
  - In this instance, Caller holds the invoking cell object
  - To find the column of the invoking cell, ask for its Column property
- So we need to know how to grab the properties of objects

Methods and properties

- In VBA, an "object" is an entity that represents something in Excel, such as a cell, a row, a range, a number format, etc.
- Objects are instances of classes
- Classes have methods defined for them.
  - Methods can take arguments
  - Example: the Range class has a method Offset that is similar to the worksheet function `OFFSET`, except that it takes only the first two arguments.
- Objects (instances of classes) also have properties
  - Properties cannot take arguments
  - Properties can have a different value for each instance
- Use postfix notation for methods and properties
  - To use the Offset method on FooBar (a Range object): FooBar.Offset(2,3)
  - To get the Value property of FooBar (a Cell object): FooBar.Value

Cells, rows and columns in a range

- To get the (i,j) cell in a range

```
theRange.Cells(i, j)
```

- To get its value

```
theRange.Cells(i, j).Value
```

- To get a collection of all the rows

```
theRange.Rows
```

- To get the ith row in a range

```
theRange.Rows(i)
```

- To get the number of rows

```
theRange.Rows.Count
```

- To get the column number

```
theRange.Column
```

- To get the number of cells

```
theRange.Cells.Count
```

- To get a collection of all the columns

```
theRange.Columns
```

- To get the i^{th} column in a range

```
theRange.Columns(i)
```

- To get the number of columns

```
theRange.Columns.Count
```

The column number of the invoking cell

- The column number is a property of the invoking cell

```
Application.Caller.Column
```

- In a similar way, we can invoke (almost) any Excel worksheet function

```
Application.WorksheetFunction.SUM(theRange)
```

- Some worksheet functions aren't available
  - `SQRT`: use the VBA function Sqr instead
  - Look in VBA on-line help for 'WorksheetFunction' and follow from there to a list of worksheet functions that Excel VBA supports

The If statement

- The syntax of If is

```
If <test condition> Then
        <a sequence of statements>
Else
        <another sequence of statements>
End If
```

- The "Else" and its sequence of statements are optional
- You can chain them together to arbitrary length

```
If <test condition> Then
        <a sequence of statements>
Elseif <another test condition> Then
        <another sequence of statements>
Else
        <another sequence of statements>
End If
```

- Sometimes you need to "flip" the condition with "Not"

```
If Not <test condition> Then
        <a sequence of statements>
End If
```

A formula for period number

- This pretty much does what we want:

```
Application.Caller.Column - firstPeriod.Column + 1
```

- If this value is <= the number of periods of depreciation, then we depreciate
- If it is > the number of periods, then the depreciation is 0

```
If Application.Caller.Column - firstPeriod.Column + 1 <= nPeriods Then
    1 / nPeriods
Else
    0
End if
```

All together now


```
Function Depreciation(firstPeriod As Object, nPeriods As Integer) As Double
    If Application.Caller.Column - firstPeriod.Column + 1 <= nPeriods Then
        Depreciation = 1 / nPeriods
    End If
End Function
```

The main points

- Macros reduce maintenance costs, reduce errors, and speed development
- Two kinds of macros: function macros and command macros
- Two languages — VBA and XLM
- Basic VBA macro structure
  - Variable declarations
  - Computations
  - Returning values
- Objects have properties and methods
- Methods and properties use postfix syntax
  - Caller, Column, Row, Columns, Rows, Count,
  - Application object

Reference readings

- Rob Bovey, Stephen Bullen, John Green, Robert Rosenberg, *Excel 2002 VBA Programmers Reference*. Birmingham, UK: 2001. Wrox Limited.
- Steven Roman. *Writing Excel Macros with VBA, Second Edition*. Sebastopol CA: 2002. O'Reilly.
  - The above are a whole lot more than you need for this course. Don't even think of looking at them unless you want to dive into programming. But if you want to, they're solid references.

- On line help for VB takes some getting used to, but it is serviceable.
- Readings: Excel Macros in Visual Basic for Applications

Preview of next time: Using Macros II

- Iteration
- Dynamic arrays
- Using the Set statement for objects