1 Introduction: Hello World

Structured programming can be defined as a sequence of instructions that achieve some objective if followed. We typically think of structured programming as being commands to a computer, but we actually use structured programming in our daily lives. For example, consider the following edited directions from Google Maps:

1. Head east on Church Cir toward Duke of Gloucester St.
2. Take a slight right onto MD-450 E.
3. Turn right onto King George St.
4. Turn left toward King George St.
5. Turn left onto King George St.
6. Slight right onto Boundary Rd.
7. Turn left onto Truxtun Rd.
8. Take the 1st right onto Maryland Ave.
9. Destination will be on the left.

Following these directions would take one from Church Circle to USNA. However, the directions assume some ideas. For example, what does it mean to "Head east" or to "Turn left". Most English-speaking humans know what these directions mean: we have learned what it means to turn left or to head east. Note that directions such as turning left or heading east are independent of where we are. We only need to make sure there is a left turn, or that heading east would not lead us to drive our car off the road. In particular, we have *modularized* the idea of turning left and heading east–the idea of turning left is a *routine*, i.e., a compartmentalized set of directions and actions that are independent of other directions and actions. If you consider the actual mechanics of turning left, explaining how "turning left" works is actually quite complicated. In fact, it is quite time-saving that we can assume people know what it means to "turn left". With structured programming, we will use this concept of modularizing to write readable and effective programs.

Visual Basic (VB) is the language we will be using this semester and is the main *macro language* used by Microsoft products such as Excel and Word. A macro language is a specialized programming language that is used to instruct a different program to perform operations. We will refer to a program in a macro language as a *macro*. The macro language typically has a set of commands that are specific to the overarching program. For example, in VB, typing `MsgBox` will display a dialog box. Note that `MsgBox` is a routine analogous to the human instruction "turn left" in that creating a display box is quite complicated, so having the routine `MsgBox` is quite useful. In VB, we refer to routines as *subroutines*, and *functions*. Macros are the high level subroutines which we, as programmers, create.

Let's create a macro called HelloWorld. To do this on a Mac, click on the Tools menu in Excel and select Macros-Macros... On a PC (using Office 2007 and above), select the Developer Tab, then click on Macros. In the dialog box, enter HelloWorld into the Macro name: input box. Then click the Create button. This should bring up the VB editor which should look like the following.

```
Sub HelloWorld()

End Sub
```

Here, the command `Sub` indicates we have started a subroutine. The first `Sub` in a file indicates we are writing a macro. The next thing written is the name of the macro, in this case `HelloWorld`. The parentheses indicate that there is no *input* to the function. Finally, the `End Sub` command indicates that the macro has ended.

Now within the editor modify the code so that it looks like the following:

```
Sub HelloWorld()
MsgBox "Hello world"
End Sub
```

Save the macro and go back to your Excel worksheet. It is important to point out that when saving the Excel file you are working on, it needs to be a MACRO ENABLED spreadsheet. This is indicated by the file extension `.xlsm`. If you do not save the spreadsheet as a MACRO ENABLED file, all the macros you create will not be saved as part of the file. On a Mac, select your macro from the Tools-Macros-Macros... dialog box and click run. On a PC (using Office 2007 and above), select your macro from the Developer Tab-Macros dialog box and click run. You should get a dialog box that says "Hello world".

In some sense, the code alone describes what it does. But as humans need to understand code too, it helps to have descriptions of the code in English. However, we need to tell the computer to ignore such descriptions as the computer does *not* understand English. To do this, we use a *comment* indicator. In VB, this is a single quote. For example, in your HelloWorld macro, modify it so that it reads the following:

```
'this is a comment
Sub HelloWorld()
'this is another comment
MsgBox "Hello world"
End Sub
```

Save and run your macro again. Note that the same result occurs – the comments have been ignored.

Now do the following.

1. Modify your macro so that it says a greeting with your name.
2. Modify your macro so that the comments reflect what your macro does and who wrote it.

2 Variables

One of the principal characteristics of a computer is the *memory* of a computer. In this section, we will describe how programming languages use computer memory through *variables*. In a programming language, a variable is a name given that represents some data that the programmer controls. In order to use variables, we tell the computer what *type* of variable we want, and what the name of the variable is. For example, in VB, create a new macro (name it what you wish) and modify it so that it looks like the following.

```
'a macro that plays around with some variables
Sub VarIntro()
Dim x As Integer
Dim LongerName As Integer

x = 10
MsgBox "First, x = " & x
LongerName = x + 2
x = LongerName - 2*x

MsgBox "x = " & x & " and LongerName = " & LongerName
End Sub
```

The `Dim` statement indicates that we are creating a variable. In the first statement starting with `Dim`, the name we choose for the variable is `x` and in the second statement starting with `Dim`, the name we choose is `LongerName`. Note that we have considerable flexibility in naming variables, and some caution should be used to make sure variables are chosen so that they are descriptive without being onerously long. The command `As Integer` indicates the variables are integers. We then use a MsgBox to display what the values of the variables are. This is an example of *output* which we describe in more detail in Section 3.

It is often a good idea to require all variables to be declared before they are used. As a rule, Visual Basic does NOT require this to be done. To enforce this as a requirement, you can use the `Option Explicit` statement. If used, the `Option Explicit` statement must appear in a file before any other source code statements. When `Option Explicit` appears in a file, you must explicitly declare all variables using the Dim or ReDim statements. If you attempt to use an undeclared variable name, an error occurs at compile time. Use `Option Explicit` to avoid incorrectly typing the name of an existing variable or to avoid confusion in code where the scope of the variable is not clear. If you do not use the `Option Explicit` statement, all undeclared variables are of `Variant` type.

For the two variables we created, we then set the `x` variable to the value 10, the variable `LongerName` to `x + 2`, i.e., 12, and then reset `x` to `LongerName - 2*x`. Note that the latter statement highlights a difference in how equalities can be used in a programming language. In a programming language, there is equality used to **set** a variable to something and there is equality used to **compare** two quantities. All of this differs from the notion of equality used where we wish to **solve** for unknowns. In particular, a single equals sign indicates, in VB, we are *setting* a variable to a quantity. If the same variable is used on the right-hand side, then the old value is used. For example, in the code above, the statement

```
x = LongerName - 2*x
```

Will cause `x` to be set to -8 as VB will substitute a 12 for `LongerName` and a 10 for `x`. If there is no value to use for a variable on the right-hand side, then VB will use a default value for `x`, which is zero[1]. For example, delete the `x = 10` command from your code and try running the macro. Note that the value for both `x` and `LongerName` is 2.

Of course, integers are not the only types of variables VB uses. As you probably noticed from the autocomplete features of the VB editor, there are several variable types. For now, we will focus on four: integers, booleans, doubles, and strings. We will discuss booleans more in Section 4.

2.1 Numbers: integers and doubles

We now discuss the main variable types we use for numbers which are *integer* or *double*. A variable that is of type integer will be treated as a whole number. For example, if you modify your previous code so that it looks like:

```
'a macro that plays around with some variables
Sub VarIntro()
Dim x As Integer
Dim LongerName As Integer

x = 10.5
MsgBox "First, x = " & x
LongerName = x + 2
x = LongerName - x/2

MsgBox "x = " & x & " and LongerName = " & LongerName
End Sub
```

Note how the variable values were rounded down. To use variables which can be non-integer, we use the type double. Modify your code so that it looks like the following:

```
'a macro that plays around with some variables
Sub VarIntro()
Dim x As Integer
Dim LongerName As Integer
Dim a As Double
Dim b As Double

'integers round down
x = 10.5
MsgBox "First, x = " & x
LongerName = x + 2
x = LongerName - x / 2
MsgBox "x = " & x & " and LongerName = " & LongerName

'double variables are continuous
a = 10.5
MsgBox "First, a = " & a
b = a + 2
a = a - b / 2
```

[1]Different programming languages react differently to not declaring a variable. E.g., matlab would simply crash.

```
MsgBox "a = " & a & " and b = " & b

End Sub
```

Note how the values of `a` and `b` are set correctly with no rounding.

2.2 Strings

A variable of type string can hold variable length alphanumeric characters. Strings in VB are, relative to many other languages, easy to use and manipulate. In fact, we have already been using strings each time we invoked the MsgBox command. The "&" is used to *concatenate* strings. Also, other variable types can be concatenated as well. For example, make your code look like the following:

```
'a macro that plays around with some variables
Sub VarIntro()
Dim x As Integer
Dim LongerName As Integer
Dim a As Double
Dim b As Double
Dim s as String
Dim t as String
Dim u as String

'integers round down
x = 10.5
MsgBox "First, x = " & x
LongerName = x + 2
x = LongerName - x / 2
MsgBox "x = " & x & " and LongerName = " & LongerName

'double variables are continuous
a = 10.5
MsgBox "First, a = " & a
b = a + 2
a = a - b / 2
MsgBox "a = " & a & " and b = " & b

'do not forget the space after First part
s = "First part "
t = "and a second part with"
u = s & t & " x = " & x & " and a = " & a & "."
End Sub
```

2.3 Objects and their Properties

VBA is an *Object Oriented* programming language. There are many prebuilt objects that are specific to Excel VBA. Some important Excel VBA objects are `Range` object, the `Cell` object, and the `Sheet` object. Objects have properties and methods associated with them.

Sheet object: You can access a particular Sheet object by using the code `Sheets(''nameofsheet'')`. `Select` is an important property of the Sheet object. This allows you to specify which sheet is active from within a macro. (The default active sheet is the sheet you were in when the macro was called.) See how the `Select` property of the `Sheet` object is used in the macro in the next section.

Range object: A range is single cell or an array of cells within an Excel Worksheet. You can assign a single cell to a range object, then use the `Cells(row, col)` property to access a particular cell relative to the cell the range object is set to. The `Cell` object has the `Value` property, which allows us to read an write cell values. See an example of these objects and properties in the macro below.

- `Sheets(''Sheet1'')` is a Sheet object.

- `Range(‘‘B3’’)` is a Range object.
- `r.Cells(2,1)` is a Cell object.
- `Value` is a property of a Cell object.

WorksheetFunction object: This useful object is really a wrapper for all of the worksheet functions that are available in Excel. For example, `WorksheetFunction.Max(Range(‘‘A1:B10’’))` returns the maximum value in the provided range of cells.

Object object: This is a general purpose object type. We will use it when we are reading and writing files.

3 Input and Output in Excel

Now that we have variables, we need a way for our programs to gather data and to report the results of any calculations we perform with the variables. *Input* and *output* are how the programs written communicate while running. We have already seen the `MsgBox` subroutine which is a method for output in VB. However, we are most interested in input and output as it relates to Excel. In particular, we are most interested in gathering input from data in an Excel sheet and writing output to the Excel sheet.

Example: Reading and Writing Cell Values

```
Sub HelloWorld2()
Dim r as Range
Dim x As Integer

Sheets("Sheet1").Select                ‘Make Sheet1 active
Set r = Range("B3")                    ‘Set the Range variable r to the range containing cell B3
                                       ‘NOTE: We have to use ‘‘Set’’ when setting values of objects
r.Cells(1,1).Value = "Hello World"     ‘Set the value in cell B3 (row 1 col 1 from B3) to Hello World
r.Cells(2, 10).Value = "Hello World"   ‘Where did this end up?

x = r.Cells(2,1).Value                 ‘Read the value in cell B4 (row 2 col 1 from B3) into x
r.Cells(3,1).Value = x + 1             ‘Set the value below cell B4 to x+1
x = r.Cells(1,1).Value                 ‘Attempt to read the value in B3 into x
End Sub
```

Before running the macro, ensure that cell B4 in Sheet1 has an integer in it. Running the macro generates an error due to the last statement. We have declared `x` as an integer, but tried to set it equal to a string. VB cannot do this, so it reports a bug. Comment out the `x = r.Cells(1,1).Value` line and rerun the macro.

```
Sub HelloWorld2()
Dim r as Range
Dim x As Integer

Sheets("Sheet1").Select                ‘Make Sheet1 active
Set r = Range("B3")                        ‘Set the Range variable r to the range containing cell B3
r.Cells(1,1).Value = "Hello World"     ‘Set the value in cell B3 (row 1 col 1 from B3) to Hello World
x = r.Cells(2,1).Value                 ‘Read the value in cell B4 (row 2 col 1 from B3) into x
r.Cells(3,1).Value = x + 1             ‘Set the value in cell B5 (row 3 col 1 from B3)
‘x = r.Cells(1,1).Value                 ‘Attempt to read the value in B3 into x
End Sub
```

We can use the `Cells(row,col)` property with loops to access the data in an array of values in an Excel spreadsheet.

4 Logical expressions

In order to enable a program to make decisions, we need to use `logical expressions`, i.e., true and false statements. The basic decision structure is the `If-Then-Else`, which acts on a logical expression. The building blocks for logical expressions are the `and`, and `or` commands.

`If-Then-Else` The If-Then-Else structure works as follows:

If a logical expression is true

Then execute a command or set of commands

Else execute a different command or set of commands

For example, write the following code in VB using the previous sheet:

```
Sub TestLogic()
Dim x As Integer
Dim y As Integer

x = Range("b4")
y = Range("b5")
If (x = y - 1) Then MsgBox "worked" Else MsgBox "whoops"

If (x = y + 1) Then
    MsgBox "worked"
Else
    MsgBox "whoops"
End If
End Sub
```

Note that the equals here is used to compare the `x` variable to the `y` variable.

`and, or` : We need to also be able to combine logical expressions. Given two logical expressions, `and` evaluates the two statements to true if both are true, and false otherwise whereas `or` evaluates the two statements to true if at least one is true. For example, modify your code so that it looks like the following.

```
Sub TestLogic()
Dim x As Integer
Dim y As Integer

x = Range("b4")
y = Range("b5")

If (x = y + 1) And (x = y - 1) Then
    MsgBox "worked"
Else
    MsgBox "requires both to be true"
End If

If (x = y + 1) Or (x = y - 1) Then
    MsgBox "requires at least one to be true"
Else
    MsgBox "whoops"
End If
End Sub
```

5 Loops

Computers are extremely useful in their ability to do repetitive tasks. When we wish to have computers repeat an action, we use what are known as *loops*. Here we discuss two loop types, *for* loops and *do...until* loops. When we discuss the latter, we will also describe Boolean variables, i.e., variables that evaluate to true or false.

5.1 For loops

A for loop requires knowing how many actions the program should do. For example, to compute the sum

$$\sum_{i=1}^{5}(i+1)(i^5),$$

we would use the following VB code.

```
Sub WeirdSum()
Dim i As Integer
Dim s As Integer

s = 0
For i = 1 To 5
    s = s + (i + 1) * (i ^ 5)
Next i

MsgBox "The sum is " & s
End Sub
```

5.2 Do...until loops

A do...until loop executes a repetitive action until a logical expression evaluates to true. Often this is easier to write using a Boolean variable. Here is the previous loop written as a Do...until loop.

```
Sub WeirdSum2()
Dim i As Integer
Dim s As Integer
Dim done As Boolean
done = False
s = 0
i = 1
Do
    s = s + (i + 1) * (i ^ 5)
    i = i + 1
    If (i > 5) Then done = True
Loop Until done
MsgBox "The sum is " & s
End Sub
```

6 File System Operations in Excel

6.1 Setting up File Locations

To make your code more "portable" it is often a good idea to let the user specify the locations of important files. For example, if your spreadsheet has a tab labeled "File_Locations" then the following code allows the user to specify the path for the glp solver executable (the software used by GUSEK to solve LP models), model file, and data file, respectively:

```
Dim gus_file As String
Dim mod_file As String
Dim dat_file As String

gus_file = Sheets("File_Locations").Range("A2")
mod_file = Sheets("File_Locations").Range("A4")
dat_file = Sheets("File_Locations").Range("A6")
```

The user would type the path to the file in the correct cell in the spreadsheet, and Excel could then use this file location in the code that you write. For example, the user could specify the path for the data file by typing the following into cell A6 on the "File_Locations" tab:

```
C:\sudoku\sudoku.dat
```

6.2 Writing to a File

In order to use Excel as an interface for external solvers, we need to discuss how to perform file system operations. First, we need a variable type that serves as a pointer to the external file we want to manipulate. In VBA, we use the file type `Object` for this purpose. The example below dimensions `fs` as an `Object` type variable, creates a text file called `sudoku.dat` (where PATH is assumed to be the directory location where you want to write the file) that we can use with GUSEK (the option True means that the file can be overwritten), and writes the text `param xbar :  1 2 3 4 5 6 7 8 9 :=` to the file. (Any string can be used as input to the `.writeline` procedure.) It is important to note that you can use either `fs.write` or `fs.writeline` when writing text to the file. The difference is that `fs.writeline` inserts a carriage return-line feed (or ASCII Chr(13) + Chr(10)) after it has written the final character to the file. It is left to the student to complete the code below to write the remainder of the puzzle to the `sudoku.dat` file.

```
Dim fs As Object

'Write puzzle to file, to be used by GUSEK
Set fs = CreateObject("Scripting.FileSystemObject")
Set fs = fs.CreateTextFile("PATH\sudoku.dat", True)
fs.writeline ("param xbar :          1  2  3  4  5  6  7  8  9 :=")
```

6.3 Reading from a File

Next we discuss how to read data from an external source into Excel. Here we use the example of reading from a `.csv`, or comma-separated value file, but it could be used for other types of files as well. The example below begins by dimensioning the required variables. Next it sets the value of `jay_range` to cells B2 through J10 on the worksheet called Solution. It then erases any previous data stored in these cells. Then the VBA function `FreeFile()` is used to get the next available file number (a file number must be in the range from 1 to 511). It then opens the file `PATH\SudokuSolution.csv` (where PATH is assumed to be the directory location where the file is located). It then writes the header information contained in the file to the one-dimensional array `header`. Finally, the solution data file is closed. It is left to the student to complete the code below to read the remainder of the solution from `PATH\SudokuSolution.csv` and write it to the worksheet Solution.

```
Dim jay_range As Range
Dim dataFN As Integer
Dim header(1) As Integer

'Set up solution range
Set jay_range = wsSolution.Range("B2", "J10")

'Clear solution cell contents
jay_range.ClearContents

'Open solution file
dataFN = FreeFile()
Open "PATH\SudokuSolution.csv" For Input As #dataFN

'Write header information to a dummy variable called header
Input #dataFN, header(1)

'Close solution file
Close #dataFN
```

7 Using an External Solver with Excel

7.1 Using the wsh.Run Command

There is a built-in solver in Excel. However, this solver is limited to a maximum of 200 decision variables and 100 constraints. In addition, formulating problems to use the built-in solver is cumbersome and the results are not always correct (especially when using binary variables). For these reasons it is often useful to have Excel serve as a "front-end" for your optimization model, while the actual solution is obtained using a more robust solver package. The example below assumes that glpsol (the solver behind GUSEK) is used as the solver, but with minor modifications it could be used with other commerical solvers as well.

In Section 6.2 we discussed how to create a data file for use by GUSEK. The examples below contains the VBA code necessary to run GUSEK from the command line (where `mod_file` and `dat_file` are the user defined paths to where the model and data files are located, respectively). The option `/c` terminates the command line window once the program call is complete. The `-m` and `-d` options tell the solver which files to use for the model and data, respectively.

The advantage to using `wsh.Run` over the `Shell` command is that we can specify the option `waitOnReturn`, which tells VBA to wait until the solver has finished before it moves on to the next line of code. The example below is used when the desired solver is `glpsol.exe`, and PATH is the location of `glpsol.exe`. The solver options `--cover --clique --gomory --mir` generate mixed cover cuts, clique cuts, Gomory's mixed integer cuts, and mixed integer rounding cuts, respectively.

```
Dim wsh As Object
Dim waitOnReturn As Boolean: waitOnReturn = True
Dim windowStyle As Integer: windowStyle = 1

gus_file = "C:\PATH\glpsol.exe"

Set wsh = VBA.CreateObject("WScript.Shell")
wsh.Run "cmd /c " & gus_file & " --cover --clique --gomory --mir -m " & mod_file & " -d " & dat_file & "",
windowStyle, waitOnReturn
```

7.2 Output Command in GUSEK

In this section we discuss how to use the built-in table drivers contained in GUSEK to write the output from the solver to a `.csv` file for use in Excel. In the example below, the portion of the command `output{i in I, j in J, k in K:(Y[i,j,k].val = 1)}` instructs GUSEK to only consider output where the values of `Y[i,j,k]` are equal to 1. The portion of the command `OUT "CSV" "PATH\SudokuSolution.csv"` tells GUSEK to write the output to a `.csv` file called `SudokuSolution` at the directory location specified by PATH. Finally, the portion of the command `k~solution` tells GUSEK to make the header of the file `solution`, and then write the values of `k` for each `i in I, j in J, k in K` where `Y[i,j,k]` is equal to 1.

```
table output{i in I, j in J, k in K:(Y[i,j,k].val = 1)} OUT "CSV" "PATH\SudokuSolution.csv" : k~solution;
```