Implementing Legacy Statistical Algorithms in a Spreadsheet Environment

Stephen W. Liddle and John S. Lawson
Brigham Young University, Provo, UT 84602

Abstract. *Over the years creative researchers in the field of statistics have published many useful and freely available FORTRAN programs implementing novel statistical computations. But it is becoming less common to find a FORTRAN compiler installed on the average workstation, making this prior work less accessible to the masses who rely on personal computers running Windows. We propose a methodology for converting legacy FORTRAN algorithms to run as VBA macros in the Excel environment. Using our approach, moderately literate programmers can effectively migrate legacy algorithms to the ubiquitous Excel spreadsheet tool, where users will again have ready access to past algorithm contributions.*

1. Introduction

Many algorithms for statistical calculations have been published in FORTRAN in journals such as *Applied Statistics* and the *Journal of Quality Technology*. The code for these algorithms and many other programs that have been described in *The American Statistician* and *Journal of the American Statistical Association* can be downloaded for free from STATLIB (`http://lib.stat.cmu.edu/`). These published algorithms have been very valuable to applied statisticians who want to use statistical methods they have read about in current research journals. However, today much of statistical computing is performed on personal computers that do not come with a FORTRAN compiler installed. Therefore the published algorithms are inaccessible to many applied statisticians. Some of the code has been duplicated or incorporated as add-ins for standard statistical packages such as SAS, SPlus, and MINITAB, but the latest algorithms, and many older algorithms, have not. One way to utilize this legacy code on common personal computers is with Microsoft Excel.

Practitioners who use Excel have the option of creating macros using Microsoft Visual Basic for Applications (VBA), a Basic language that is built into Excel. VBA macros perform exactly as Basic programs, but rather than having input from the keyboard and output to the screen, VBA macros typically write output to an Excel worksheet, and take input from a worksheet or through graphical dialog boxes.

This paper shows how published algorithms in FORTRAN can be converted to VBA macros that will run in Excel. Converting FORTRAN algorithms to VBA macros has several advantages. First, conversion allows practitioners to work in an environment where they are comfortable. Second, when tabular output of programs is placed in an Excel worksheet rather than a file or computer screen, users can employ Excel features (with which they are already familiar) to further manipulate or summarize the results. For example, worksheet functions `=SUM()` or `=AVERAGE()` can quickly summarize a column of output, or Excel's Chart Wizard can be used to quickly create a graph from tabular output. Finally, when programs are converted to VBA macros, the spreadsheet environment and graphical dialog boxes can make the programs more user friendly by reducing the need for repetitive typing of inputs when the number of inputs is non-trivial.

Our methodology for converting FORTRAN programs to VBA macros involves the following steps:

1. Understand the original FORTRAN program.
2. Choose suitable input/output (I/O) methods.
    a. What are the data sources and outputs?
    b. What interactive prompts are necessary?
    c. Design an appropriate cell layout and supply labels, borders, etc.
3. Convert the original FORTRAN source code to VBA.
    a. Consider reuse of existing Excel and VBA functions (this might involve adapting and transforming the original program).
    b. Translate individual code statements, transforming code structures where appropriate.
4. Test and use the resulting Excel worksheet.

In the remainder of this paper we give the details of our approach. We start with an overview of the fundamentals of VBA in the Excel environment. Next we describe how the characteristics of a spreadsheet environment affect I/O and other aspects of programming perspective. Then we give specific guidance on converting FORTRAN programs to VBA.

2. Fundamentals of VBA in Excel

Visual Basic for Applications (VBA) is the programming engine used in the Microsoft Office suite (and other similar products) for automating tasks and programming user-specified features. Indeed, when you record a macro in Excel, each individual action is translated into a VBA statement. When you later edit the macro, Excel displays a Visual Basic editor window, and you edit in the VBA language.

VBA is distinguished from the stand-alone version of Visual Basic (VB) primarily by the fact that VBA is embedded within another application (such as Word, PowerPoint, or Excel), whereas VB creates Windows programs that can run independently of a host application. For example, one of the authors has used VBA to create a custom facility that e-mails student grades directly from a grade roll spreadsheet to individual students. The VBA code presents a dialog box, the user selects a first and last student, and the VBA code then e-mails grades to students in the chosen range. This program can only be run within the Excel spreadsheet, and it is tightly bound with the data stored in the spreadsheet. In contrast, a standalone VB program is an application that is launched from the desktop like any other application. In this paper, we restrict our focus to VBA, though the Visual Basic language and many of Microsoft's development tools are common to both VB and VBA.

2.1 The Spreadsheet Object Model

Programming in VBA starts with the host application's *object model*. Once you understand the objects in the VBA host application, you can use those objects to perform complex tasks automatically. The Excel object model is a hierarchy of objects that represent the elements of a spreadsheet. Each object has *properties* that represent data and *methods* that perform specific operations. As shown in Figure 1, `Application` is the top-level object that represents the Excel program. `Application.Caption` is a property that indicates the name that appears in the title bar of the main Excel window. Invoking the `Application.Save` method causes Excel to save changes just as if the user had typed Ctrl+S or selected the File|Save Menu. The `Application` object contains a collection of workbooks, denoted `Application.Workbooks`. This collection represents the Excel workbooks that are currently open. A workbook can hold several worksheets, each represented by `Worksheet` objects. Worksheets contain elements such as cells (represented by `Range` objects) and graphs (`Chart` objects). VBA macros typically run in the context of a particular worksheet, so we often do not need to specify `Application`, `Workbook`,

and `Worksheet` explicitly. For example, to reference the value stored in cell A1 of the current worksheet, we just write `Range("A1").Value`.

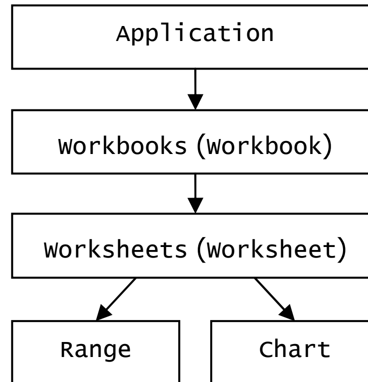


Figure 1. Portion of the Excel Object Model

The complete Excel object model is documented in the Visual Basic Reference section of Excel's help file.

```
' Require explicit variable declaration.  Recommended.
Option Explicit

' Declare constants
Const PI As Double = 3.14159265358979
Const MAX_POP As Integer = 100

' Declare variables
Dim PopCount As Integer
Dim Population(1 To MAX_POP) As Double   ' An array

' AddObservation is a subprocedure that places an
' observation into the Population array if there is room.
Public Sub AddObservation(value As Double)
    If PopCount < MAX_POP Then
        PopCount = PopCount + 1
        Population(PopCount) = value
    End If
End Sub

' SimpleMean returns the average of the observations in
' our population.  A function returns a value.  The
' return value is determined by assigning to the function
' name (SimpleMean, in this case).
Public Function SimpleMean() As Double
    Dim total As Double
    Dim i As Integer

    total = 0
    For i = 1 To MAX_POP
        total = total + Population(i)
    Next i

    SimpleMean = total / PopCount
End Function
```

Figure 2. Example VBA Program

2.2 Language Basics

VBA is a modern descendant of ANSI BASIC. As such, it includes the following major language elements, as demonstrated in Figure 2.

2.2.1 Data Handling. VBA primitive data types include `Boolean`[1] (`True` or `False`), `Integer` (roughly +/- 2 billion), `Double` (IEEE 64-bit floating-point values), and `String`, among others. Variables in VBA are typically typed, but a special `Variant` data type allows programmers to rely on Visual Basic to determine whether to treat the value as numeric or string depending on the context. (We recommend avoiding `Variant` most of the time.) Variables are usually declared by means of the `Dim` statement. Symbolic constants are like variables whose values cannot change during program execution. These constants are introduced by the keyword `Const` (e.g., `PI` and `MAX_POP` in Figure 2). VBA supports arrays of one or more dimensions (e.g., `Population`). VBA also includes many non-primitive components (e.g., `TextBox`, `File`) that provide extended data handling capabilities.

2.2.2 Operators, Expressions, Functions. VBA supports a variety of mathematical, relational, Boolean, and other operators, as summarized in Figure 3. Using these operators and parentheses, we can build arbitrarily complex expressions. VBA supplements these basic operators with an extensive library of functions. Available functions include trigonometric (`Sin`, `Cos`, `Tan`, `Atn`), logarithmic (`Log`, `Exp`), random-value (`Rnd`), and financial (`Rate`, `Pmt`, `NPV`) to name a few. For example, the first form of the quadratic formula $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ can be expressed in VBA as `(-b+Sqr(b^2-4*a*c))/(2*a)`, where `Sqr` is the square root function. Note that VBA observes conventional operator precedence, so that exponentiation precedes multiplication, which precedes addition. Operators with the same precedence are evaluated left-to-right. When in doubt, it is wise to use parentheses to control evaluation order explicitly.

Mathematical	
`^`	exponentiation
`*`, `/`, `+`, `-`	multiplication, division, addition, subtraction
`\`, `Mod`	integer division (discard remainder), modulus (give the remainder)
Relational	
`=`, `<>`	equality, inequality
`<`, `<=`, `>`, `>=`	relative magnitudes
Other	
`And`, `Or`, `Not`	the usual Boolean operators
`&`	string concatenation
`Like`	pattern matching

Figure 3. VBA Operator Summary

2.2.3 Structured Programming Constructs. Structured programming—the idea of organizing code according to the structures of sequence, selection, and iteration—is foundational to modern programming. VBA supports these structures with `If-Then`, `Select-Case` (selection structures), and `For-Next`, `Do-While`, `Do-Until`, and `While-`

[1] We use syntax highlighting throughout this paper. Code is listed in a `monospaced font`. VBA keywords are listed in **`boldface`**. Comments, which begin with a single quote mark, are *`italicized`*.

`Wend` statements (iteration structures). The two most common, `If-Then` and `For-Next`, are illustrated in Figure 2. `If-Then` statements take a Boolean expression (e.g. `PopCount < MAX_POP`) as the condition. `Else` and `ElseIf` clauses can also be used to supply alternative actions in case the condition is false. `For-Next` loops allow us to count through a sequence of numbers. Using the `Step` clause, we can create interesting sequences. For example, the statement `For i = 10 To 2 Step -2` assigns `i` the values 10, 8, 6, 4, and 2 in that order. Sometimes the other selection and iteration structures are more convenient, but in general `If-Then` and `For-Next` are sufficient to implement any computable function.

2.2.4 Procedures and Functions. Another key idea of structured programming is the notion that programs should be decomposed into smaller, more manageable modules. VBA supports user-defined procedures and functions such as `AddObservation` and `SimpleMean` in Figure 2. A procedure (also called "subprocedure" in VBA parlance) is a distinct unit of code we can invoke to accomplish a task. Procedures may receive zero or more parameters, as indicated by a parameter list in parentheses. In Figure 2, `AddObservation` receives one parameter of type `Double`. Functions are procedures that also return a value. VBA uses different keywords to distinguish the two (`Sub` vs. `Function`). Functions specify the type of the result by listing it after the parameter list. Thus, `SimpleMean` receives no parameters (indicated by empty parentheses), but it does return a `Double` value. There is no explicit "return" statement in VBA; instead, you simply assign the result to the function name (e.g. `SimpleMean = total / PopCount`). Best practice is to name procedures with a verb-object phrase (e.g. `AddObservation`) and to name functions with a description of the result (e.g. `SimpleMean`).

3. Retargeting Traditional Algorithms to a Spreadsheet Environment

Embedding a programming language such as VBA in a spreadsheet environment like Excel changes the context in which we implement our computing solutions. Every computer program consists of the three essential components of input, processing, and output. And a spreadsheet environment impacts each of these elements.

3.1 Input/Output

In a traditional programming environment, data input comes from two typical sources: the keyboard (for dynamic data entry) and data files. Similarly, output typically goes either to the system console or to some kind of storage file. In an embedded spreadsheet scenario, we have more possibilities. For example, in addition to regular data files VBA can also access relational databases. But perhaps most commonly, the input and output data can be stored directly in the Excel workbook itself. For example, Figure 4 shows a worksheet designed to implement Lee's (1987) *ORPS* subroutines for optimum ridge parameter selection. Cells A5:A6, B5:B6, C5, and D5 represent the inputs, and cells in columns G and J will receive the output generated by running the *ORPS* algorithms. Command buttons (labeled "Run ORPS1" and "Run ORPS2") provides the event handling. When the user clicks a button, its associated VBA code is executed. (See Appendix D for the original OPRS algorithms as published in *Applied Statistics*, and Appendix E for a listing of our version rewritten in VBA.)

VBA code can reference cells in a worksheet by accessing the `Range` property of a `Worksheet` object. For example, assuming the worksheet in Figure 4 is labeled "ORPS," we could reference the Sigma cell as `Worksheets("ORPS").Range("D5")`. If we know we are already in the context of the "ORPS" worksheet, `Range("D5")` would be sufficient. We could assign the value currently stored in cell D5 to the variable `sigma` with the statement `sigma = Range("D5").Value`. Similarly, we can change the value of a cell by assigning to the `Value` property of a `Range` object. Whereas

`Range` uses the "A1" style of reference, the similar `Cells` property of a `Worksheet` can be used to access cells by numeric index. Thus, `Cells(5, 4)` refers to the same cell as `Range("D5")`.

	A	B	C	D	E	F	G	H	I	J
1–2	AS 223 Optimum Ridge Parameter Selection						Output Region			
3		Inputs								
4	Alpha	Lambda	Delta	Sigma		Optimum k by MSEE			Optimum K by MSPE	
5	5	0.02	0.00001	0.603		$\hat{k}$:	0.014550		$\hat{k}$:	0.015106
6	2.58	1.98				$\sigma^{-2}M_1(\beta)$:	50.505		$\sigma^{-2}M_1(\beta)$:	50.505
7						$\sigma^{-2}M_1(\beta^*(k))$:	29.447		$\sigma^{-2}M_1(\beta^*(k))$:	29.457
8						$\sigma^{-2}M_2(\beta)$:	2.0		$\sigma^{-2}M_2(\beta)$:	2.0
9			Input Region			$\sigma^{-2}M_2(\beta^*(k))$:	1.566		$\sigma^{-2}M_2(\beta^*(k))$:	1.566
10	Clicking these buttons					Iterations:	6		Iterations:	7
11–13	runs the ORPS1 and ORPS2 algorithms.					Run ORPS1			Run ORPS2	

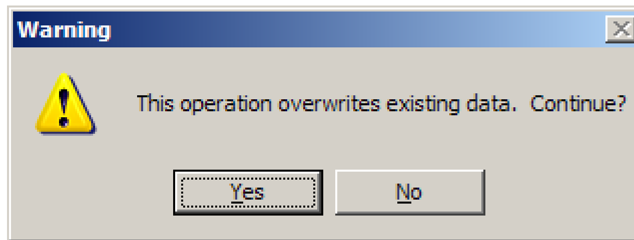
Figure 4. Spreadsheet Template for ORPS Algorithms

Sometimes it is necessary to prompt the user for information dynamically while running an algorithm. Message boxes and input boxes are useful methods for interactive communication. To display information in a pop-up window, use the `MsgBox` function:

`Call MsgBox(`*prompt*`)` or `intResult = MsgBox(`*prompt*`, `*buttons*`, `*title*`)`

Here, *prompt* is a string displayed in the pop-up window. *Buttons* indicates what response buttons are available to the user. *Title* is a string displayed in the title bar of the message box window. If you omit *buttons* and *title*, VBA displays an OK button and a default title. Common choices for buttons include `vbOKOnly` (the default), `vbOKCancel`, `vbYesNo`, and `vbYesNoCancel`. You can add an icon to the message box by adding `vbQuestion`, `vbInformation`, `vbExclamation`, or `vbCritical` to the buttons. For example, the following code displays a message box:[2]

```
intResult = MsgBox( _
       "This operation overwrites existing data.  Continue?", _
                    vbYesNo + vbExclamation, "Warning")
If intResult = vbYes Then
    ' Perform destructive operation here.
End If
```



The value returned from `MsgBox` indicates the button selected by the user. This is typically either `vbOK`, `vbCancel`, `vbYes`, or `vbNo`.

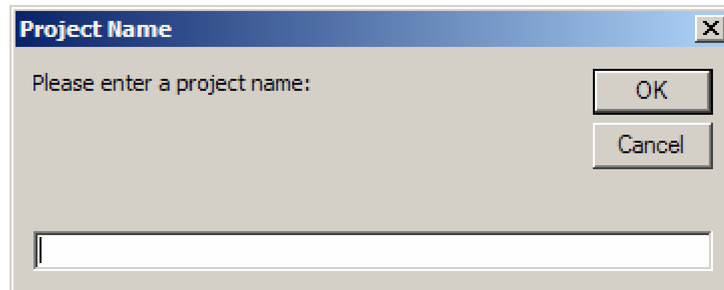
[2] An underscore (_) at the end of a line is the line continuation marker in VBA.

If we want more information from the user than a simple OK/Cancel or Yes/No response, we can use the InputBox function:

```
strAnswer = InputBox(prompt, title, default)
```

where *prompt* is a string telling the user what to enter, *title* is a message displayed in the title bar of the input box, and *default* is the default value shown in the data-entry field of the input box. For example, consider the following code, which displays the input box shown below:

```
Dim strProject As String
strProject = InputBox("Please enter a project name:", _
                      "Project Name")
```



If the user clicks OK, whatever is entered in the text field is stored in the `strProject` variable. If the user clicks Cancel, the `InputBox` function returns the empty string. VBA will perform automatic type conversion, so the results of the call to InputBox can be assigned to numeric variables instead of strings as needed.

3.2 Built-in Functions

One of the strengths of operating in the Excel environment is that many complex functions are already implemented and available. In some cases we may discover that we can replace portions of custom code with calls to the Excel built-in function library. For example, in Montgomery's (1982) FORTRAN program for optimal economic design of an $\overline{X}$ control chart, the function `PNORM(X)`, that evaluates cumulative area under the standard normal density, is programmed in FORTRAN. When we implement this program in VBA, we can replace calls to `PNORM()` with calls to Excel's built-in `NormSDist()` function. Built-in functions not only can save work (including the nontrivial task of debugging), but can also improve precision of the results (`NormSDist()` provides more digits of precision than `PNORM()`). To find the functions available in Excel, look at the `WorksheetFunction` object. Excel provides many dozens of trigonometric, statistical, engineering, financial, and other functions. An extremely abbreviated sample includes the following:

`ChiDist(x,deg_freedom)`	Returns one-tailed probability of the χ^2 distribution.
`Correl(array1,array2)`	Returns the correlation coefficient of two cell ranges.
`Fisher(x)`	Returns the Fisher transformation at a given *x*.
`Pearson(array1,array2)`	Returns the Pearson product moment correlation coefficient for two sets.
`Quartile(array,quart)`	Returns the requested quartile of a data set.
`StDev(array)`	Returns the standard deviation of a data set.
`ZTest(array,x,sigma)`	Returns the two-tailed *P*-value of a *z*-test.

We call the built-in `NormSDist()` function using the `WorksheetFunction` object as follows:

```
p = WorksheetFunction.NormSDist(arg)
```

This directly replaces the similar FORTRAN code, `P=PNORM(ARG)`, and the associated 12-line implementation of `PNORM()`. See the VBA implementation of Montgomery's (1982) program in Appendix C for more context of this example.

Running VBA Code

The final aspect that we address here of repurposing traditional programs for the Excel environment is how to create and invoke algorithms implemented in VBA. Typically, we write VBA code as Excel macros. The source code for each macro is contained in a "module" that is visible when you display Excel's Visual Basic Editor, shown in Figure 5 (choose Tools|Macro|Visual Basic Editor or press Alt+F11 to display the editor). The window on the left side gives access to a hierarchical representation of elements of your Excel workbook. Here we see our single worksheet, named "Sheet1" or "ORPS." We also see a code module, which holds the VBA code we wrote to implement the *ORPS* algorithms. Typically one creates macros by example; that is, we "record" macros (choose Tools|Macros|Record New Macro… and perform the desired operations). Indeed, a good strategy for learning more about VBA programming is to record a macro and inspect the VBA code Excel generates. You can find the recorded macro code by browsing through the modules in the Visual Basic Editor.

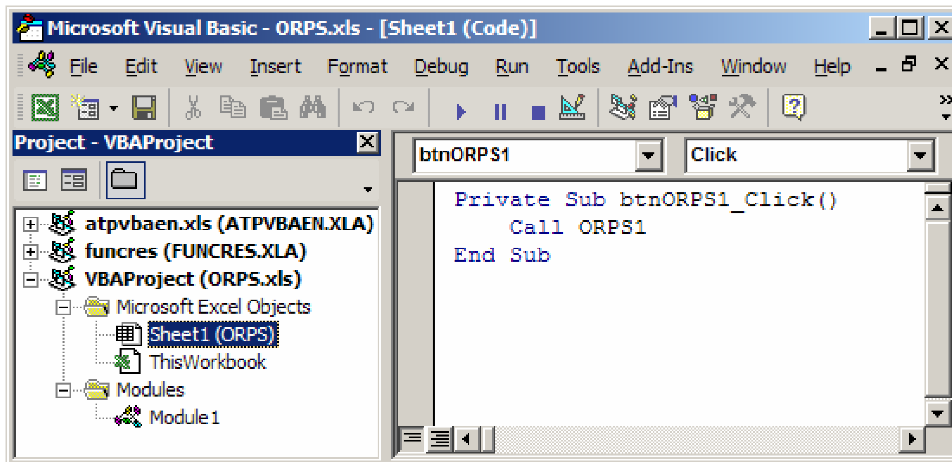


Figure 5. Excel's Visual Basic Editor

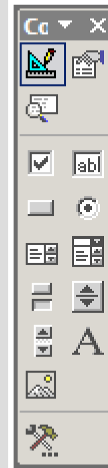
A macro in a code module is defined as a VBA subprocedure. Thus, for example, we implement the *ORPS1* algorithm by declaring a subprocedure in a code module:

```
Sub ORPS1()
    ...
End Sub
```

Each macro needs to be declared in this manner, whether it is a recorded macro or one programmed manually. To implement a new macro, either record a simple macro and then replace the generated code, or manually insert a subprocedure in a code module. If you choose the second method, when you open the Visual Basic Editor, if there are no code modules listed, you can use the Insert menu to add a new code module (choose Insert|Module). Then you can add your subprocedure to the module you just created.

Excel macros can be invoked in a couple of ways. First, we can use the Macro dialog box (choose Tools|Macro|Macros… or press Alt+F8 to display). This dialog box gives a list of the available macros, and allows you to select and run a macro. A second alternative that we sometimes favor is more user-friendly: you can insert a "command button" into your worksheet and connect that button to the macro, so that when the user

Connecting Code to a Command Button



Creating a command button in Excel is straightforward. First, select View|Toolbars|Control Toolbox. This displays a toolbar (shown at right) with graphical components such as command buttons, radio buttons, drop-down lists, and so forth. Select the control that looks like a command button. This places you in "Design Mode" (indicated by the selection around the triangle-ruler-pencil icon). Now click on your worksheet where you want the upper-left corner of the button to be placed. This creates a command button with a default name and caption (e.g. "CommandButton1"). Next, click on the properties button (upper-right corner) in the Control Toolbox. In the Properties window you can rename the button and give it a customized caption. To associate code with the button, just double-click on the button while in Design Mode. This inserts an "event handler" that is invoked when the user clicks this button. Try typing `Call MsgBox("It worked!")` into the code window. Now go back to your worksheet. While in Design Mode, you can move and resize the button. Now leave Design Mode by clicking on the triangle-ruler-pencil icon. Clicking the button now executes the associated code and displays the message box saying "It worked!" To invoke a macro instead, replace `Call MsgBox(...)` with `Call` *macro-name*.

clicks the button, the macro executes. The text box below ("Connecting Code to a Command Button") describes how to create a button and associate VBA code with it. The code to cause your macro to be invoked when the user clicks a button is simple: `Call` *macro-name*. So for example, Figure 5 shows the code to call our *ORPS1* macro. (If the macro were instead named *ORPS2*, we would write `Call ORPS2`.) The name of our command button in this case is `btnORPS1`, so the name of the procedure that runs when the user clicks the button is `btnORPS1_Click`. The VBA code for this procedure is stored in the worksheet where the button is placed (Sheet1 or ORPS in this example).

4. Converting FORTRAN to VBA

We now consider the task of converting FORTRAN code to VBA. The two languages share many common ideas, but they also have unique attributes that are important to understand. We begin by describing what is common to the languages and comparing the syntax of the languages. Then we discuss the differences between data types in the two languages. We also compare control structures for decisions and loops.

Mathematical	
`**`	exponentiation
`*`, `/`, `+`, `-`	multiplication, division, addition, subtraction
`MOD`	modulus (return only the remainder)
Relational	
`.EQ.  .NE.`	equality, inequality
`.LT.  .LE.`	less than, less than or equal
`.GT.  .GE.`	greater than, greater or equal
Other	
`.AND. .OR. .NOT.`	Boolean operators
`//`	string concatenation

Figure 6. FORTRAN Operator Summary (compare with Figure 3)

4.1 Understanding Data-Type Commonalities

Both FORTRAN and VBA are robust languages capable of implementing the complete range of computable functions. Both have structured programming constructs, data types, variables, and rich mechanisms for describing complex expressions. Figure 6 gives a summary of FORTRAN operators similar to the VBA summary shown in Figure 3. Like VBA, FORTRAN has built-in functions for trigonometric (`SIN`, `COS`, `TAN`), logarithmic (`EXP`, `LOG`, `LOG10`), and other operations (`MIN`, `MAX`, `SQRT`). Earlier we gave the VBA expression of the quadratic formula as `(-b+Sqr(b^2 -4*a*c))/(2*a)`. The FORTRAN equivalent is similar: `(-b+SQRT(b**2–4*a*c)) /(2*a)`. The translation of most expressions will be straightforward.

What is more interesting is the translation of data types. FORTRAN has six data types: `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, `LOGICAL` (`.TRUE.` or `.FALSE.`), and `CHARACTER`. Figure 7 shows the correspondence between FORTRAN and VBA. Because VBA is written by a single vendor on a single computing platform, its data types are defined more specifically than FORTRAN's (the FORTRAN language specification does not indicate expected ranges for its numeric data types). So it is important to understand the intended range for numeric values when choosing a VBA data type. Typically, VBA numeric data types are at least as capable as their FORTRAN counterparts. For integers, when in doubt you can use `Long` (32 bits) instead of `Integer` (16 bits) to guarantee that the VBA variable will have a sufficient range.

One special case is that complex numbers are not directly implemented in VBA. Instead, one must use worksheet functions (such as `COMPLEX`, `IMPRODUCT`, `IMSUM`, `IMSQRT`, etc.) to handle complex numbers. See the online VBA help files for details of these functions. (We described how to invoke worksheet functions in a previous section.)

FORTRAN Data Type	VBA Data Type
INTEGER	`Byte` (0-255), `Integer` (-32,768 to 32,767), `Long` (-2^{31} to $2^{31}-1$)
REAL	`Single` (7 digits of precision, exponent of 10^{-45} to 10^{38})
DOUBLE PRECISION	`Double` (15 digits of precision, exponent of 10^{-324} to 10^{308})
COMPLEX	Not a VBA primitive type– use VBA worksheet functions
LOGICAL	`Boolean` (`True` or `False`)
CHARACTER	`String` (variable length up to roughly 2 billion characters)
CHARACTER**length*	`String`**length* (fixed length string, length < about 65,400)

Other notable data types in VBA:

`Currency`	15 digits left of decimal point, 4 to the right
`Decimal`	28-digit floating point value (stored in 14 bytes)
`Date`	1 January 100 to 31 December 9999
`Variant`	Numeric or string data, interpreted by VBA according to context

Figure 7. Comparison of FORTRAN and VBA Data Types

One of the aspects of data types that can be fairly confusing is the use of implicit data types. In FORTRAN, you can declare your variables (e.g. `DOUBLE PRECISION X` declares an double-precision floating-point variable `X`). However, you need not declare variables, and indeed it is common practice to use implict data types. Variables that begin with the letters *I* through *N* are assumed to be of type `INTEGER`, while any other variables are of type `REAL`. If you do not want this default typing, you must either declare all your variables. Or (even worse from a software engineering perspective), you can change the default implicit types using an `IMPLICIT` statement. For example, `IMPLICIT COMPLEX (C, D)` indicates that in this program, all variables starting with *C* or *D* will

be of type COMPLEX. VBA also supports implicit variable declarations (using the Variant data type if none is specified) and type shortcuts (e.g. i% in VBA tags i as an Integer variable, j& indicates j is a Long). Because these shortcuts provide "ample rope with which to hang oneself" we advocate explicit typing as a more sound programming practice. By including Option Explicit at the top of a VBA code module, we ask the compiler to notify us when a variable has not been declared explicitly. It can sometimes be tedious to follow this methodology, but the benefits are sufficient to warrant the effort.

Symbolic constants in FORTRAN are declared as follows: PARAMETER PI=3.14159. The equivalent VBA code is Const PI As Single = 3.14159. Note that in FORTRAN, PI is implicitly assumed to be of type REAL (because it begins with a letter outside the range *I–N*). We could first give an explicit declaration, e.g. DOUBLE PRECISION PI, which would change the underlying data type to double precision instead.

Array declaration is similar in the two languages. FORTRAN's DIMENSION I(1:10) becomes Dim I(1 To 10) As Integer in VBA. Again, I is implicitly of type INTEGER in FORTRAN. It is also possible to declare array types explicitly, so DOUBLE PRECISION X(-5:5) would become Dim X(-5 To 5) As Double. However, note that the similar statements DIMENSION I(7) and Dim I(7) As Integer are actually slightly different. FORTRAN assumes 1 as the lower bound for array subscripts, while VBA assumes 0. So we prefer always to specify upper and lower array bounds explicitly to avoid any possible confusion.

4.2 Subtle Differences

An issue of especial note is that FORTRAN and VBA treat the implicit conversion of real to integer values differently. For example, what happens when we assign 3.7 or -2.3 to an integer variable? The conversion rule in FORTRAN is essentially to truncate the real number (so 3.7 becomes 3 and -2.3 becomes -2). However VBA rounds real numbers to the closest integer (or to the closest even integer if the fractional part is .5). Thus, one must pay special attention to the case where a FORTRAN program assigns real expressions to integer variables. The VBA Fix() function implements the truncation operation used by FORTRAN. Thus, the FORTRAN statement I = X could be coded in VBA as I = Fix(X). If your VBA program is not generating the same values as the original FORTRAN program, it is a good idea to check for real-to-integer conversion problems.

FORTRAN is somewhat more convenient for initializing blocks of data than VBA. The DATA statement is often used in FORTRAN to provide initial values for arrays (see the implementation of Montgomery's(1982) program in Appendix A, and note the use of DATA to initialize the 7-element array C near the end of the program). VBA requires the use of explicit assignment statements to accomplish the same task (observe the corresponding VBA code in Appendix B).

FORTRAN also has a convenient shorthand called the "statement function" which is essentially a one-statement macro. The *ORPS* algorithms declare several statement functions, including FM2(X, Y) = Y * FM1(X, Y). This code looks a lot like an assignment to a two-dimensional array FM2. It is actually the definition of a function FM2 that receives two parameters, X and Y. The translation to VBA is straightforward once you recognize that this is not an assignment statement:

```
Function FM2(x As Double, y As Double) As Double
    FM2 = y * FM1(x, y)
End Function
```

4.3 Comparing Decision and Iteration Structures

Once data types and expressions are properly converted, the other major aspect of FORTRAN-to-VBA translation is rewriting control-flow statements (for decisions and loops).

The two most common variations of the “if” statement are shown here:

	FORTRAN	VBA
Logical if	IF (*expr*) *stmt*	If *expr* Then *stmt*

Block if	IF ($expr_1$) THEN
	    $stmt_1$
	ELSE IF ($expr_2$) THEN
	    $stmt_2$
	...
	ELSE
	    $stmt_n$
	END IF
		If $expr_1$ Then
		    $stmt_1$
		ElseIf $expr_2$ Then
		    $stmt_2$
		...
		Else
		    $stmt_n$
		EndIf

In both languages, the else-if clause can be repeated 0 or more times, and the else clause is optional. VBA does not require parentheses around condition expressions.

Loop statements can be somewhat more challenging to translate, depending on how the FORTRAN programmer originally implemented them. The general form of a loop in FORTRAN is:

DO s i=e_1, e_2 [, e_3]

where s is a statement label, i is a loop variable, e_1 is the initial value assigned to i, e_2 is the terminal value of i, and e_3 is the optional increment value. The effect of this statement is to execute the loop (which includes all statements from DO up to and including the statement labeled s) for each value of i between e_1 and e_2, incrementing by e_3 (if e_3 is not given, the default value is 1). Often we find that s labels a CONTINUE statement, which has no effect except to be a place-holder for the end of a loop. Consider the following example from *ORPS1*:

```
    DO 10 J = 1, IP
        IF (LAMDA(J) .LE. ZERO) RETURN
10  CONTINUE
```

This loop executes the IF statement repeatedly for values of J in the range 1, 2, 3, …, IP. The equivalent form in VBA would be the following:

```
For j = 1 To ip
    If (lambda(j) <= 0) Then Exit Sub
Next j
```

If the increment value had been something other than 1, the For statement would have included a Step clause (see Section 2.2.3 for an example).

4.4 Eliminating Go-To Statements

Older programs tend to rely on the infamous “go-to” statement. Computer science has long since dismissed the construct as overly complex and dangerous from a software engineering perspective. When rewriting FORTRAN code, it is wise to consider how one might transform the code structure to avoid using go-to statements. Consider the following excerpt from the *Econ* algorithm:

```
       DO 8 J=1,3
       ...
6      ...
       IF(OBJFN.GT.BESTFN) GO TO 7
       ...
       GO TO 6
7      IF(J.EQ.3) GO TO 8
       XK=BESTK-STEP
8      CONTINUE
```

This sort of code can get quite messy to understand, but if we peel the onion one layer at a time, it can be managed. If we start with the `DO` loop, we have the following structure:

```
      For j=1 To 3
         ...
6        ...
         IF(OBJFN.GT.BESTFN) GO TO 7
         ...
         GO TO 6
7        IF(J.EQ.3) GO TO 8
         XK=BESTK-STEP
8     Next j
```

In the next step, we remove the third `GO TO` by negating the condition `(J.EQ.3)`:

```
      For j=1 To 3
          ...
6         ...
          IF(OBJFN.GT.BESTFN) GO TO 7
          ...
          GO TO 6
7         If j <> 3 Then
              xk = bestk - step
          End If
      Next j
```

Now we eliminate the remaining `GO TO`'s by instead using a VBA `Do Until` loop:

```
      For j=1 To 3
          ...
          Do Until objfn > bestfn
              ...
          Loop
          If j <> 3 Then
              xk = bestk - step
          End If
      Next j
```

The result is a structured program that is easier to understand and maintain. Sometimes transformations like these will be quite difficult and certainly not obvious. The *sine qua non* of programming is correctness, and in such cases a better strategy is to translate `GO TO` statements directly to VBA (see Appendix B for an example).

5. Conclusion

Contrary to what many assume, it turns out that digital assets are relatively fragile. Physical storage media degrade over time, but so do software formats, protocols, and languages. Just as we periodically migrate our data to newer media and storage formats, so also we should care for our legacy source code base. FORTRAN is by no means a dead or dying programming language. But it is significantly less common to find a FORTRAN compiler on the desktop of an applied statistician in today's PC-dominated world. For this reason, we propose that interested parties participate in an effort to re-write useful statistical algorithms for the user-friendly, robust, and ubiquitous Excel environment. (Note that the authors are not affiliated with Microsoft in any way, and indeed one of us is a participant in the open source software movement.)

To be certain, there are many other ways to manage legacy source code. For example, there is a utility to convert FORTRAN to C code automatically. The Gnu C++ compiler also includes a FORTRAN compiler component, and so FORTRAN is not hard to install on a Linux system. One could translate the algorithms to other languages, like SAS or SPlus. But the fact remains that Microsoft Office is a de facto standard on most personal computers. Moreover, the VBA environment supporting the Office applications is robust

and well suited to statistical calculations. A proficient Excel user employing the vast range of available features can achieve truly remarkable information-related productivity. By learning about VBA, Excel users can be even more productive.

We believe the time has come to look to our legacy of statistical algorithms and think about how we can leverage that work into the future. We can energize this code base by injecting modern user interfaces and updated programming structures. Our proposed methodology will let us share this legacy on a broader scale.

References

Deitel, H. M., Deitel, P., and Nieto, T. R. (1999). *Visual Basic: How to Program*. Prentice Hall, Upper Saddle River, NJ.

Lee, T. (1987) "Algorithm AS 223: Optimum Ridge Parameter Selection", *Applied Statistics*, Vol. 36, No. 1, pp 112-118.

Montgomery, D.C. (1982) "Economic Design of an $\overline{X}$ Control Chart", *Journal of Quality Technology*, Vol. 14, No. 1, pp 40-43.

Schneider, D. I. (2000). *An Introduction to Programming Using Visual Basic 6.0*. Prentice Hall, Upper Saddle River, NJ.

Tidwell, R. and Thomas, N. (2001). *Microsoft Office 2000 with Visual Basic for Applications*. Course Technology, Cambridge, MA.

Tsay, J. J. (2000). *Visual Basic 6 Programming: Business Applications with a Design Perspective*. Prentice Hall, Upper Saddle River, NJ.

Zak, D. (2001). *Microsoft Visual Basic for Applications*. Course Technology, Cambridge, MA.

Appendix A — Montgomery's(1982) Program for the Economic Design of an $\overline{X}$ Chart, as Originally Implemented in FORTRAN

```
      READ(5,2) A1,A2,A3,A3P,A4,XLAM,DELT,G,D
2     FORMAT(9F8.0)
      WRITE(6,3)
3     FORMAT(1H0,3X,'N',5X,'OPTIMUM K',3X,'OPTIMUM H'
     &,6X,'ALPHA',5X,'POWER',4X,'COST')
      AA=DELT**2*A3P/(A2+XLAM*A4*G)
      XK=0.
      DO 4 I = 1,10
      RHS=(1.2826+XK)/ORDN(XK)
      IF(RHS.GT.AA) GO TO 5
4     XK=XK+0.5
5     XK=XK-0.5
      IF(XK.LT.0.) XK=0.
      N=((1.2826+XK)/DELT)**2+0.5
      NMIN=N-10
      IF(NMIN.LE.0) NMIN=1
      NMAX=N+10
      DO 9 I=NMIN,NMAX
      XN=I
      XK=0.5
      STEP=0.5
      DO 8 J=1,3
      BESTFN=1.0E+38
      IF(J.EQ.2) STEP=0.1
      IF(J.EQ.3) STEP=0.01
6     ARG=-1.0*XK
      A=2.0*PNORM(ARG)
      ARG=DELT*SQRT(XN)-XK
```

```
      P=PNORM(ARG)
      H=SQRT((A*A3P+A1+A2*XN)/(XLAM*A4*(1./P-0.5)))
      B=H*(1./P-0.5+XLAM*H/12.)+G*XN+D
      OBJFN=(A4*XLAM*B+A*A3P/H+XLAM*A3)/(XLAM*B+1.0)
     &+(A1+A2*XN)/H
      IF(OBJFN.GT.BESTFN) GO TO 7
      BESTFN=OBJFN
      BESTA=A
      BESTP=P
      BESTK=XK
      BESTH=H
      XK=XK+STEP
      GO TO 6
7     IF(J.EQ.3) GO TO 8
      XK=BESTK-STEP
8     CONTINUE
9     WRITE(6,10) I,BESTK,BESTH,BESTA,BESTP,BESTFN
10    FORMAT(1H ,I4,2(5X,F7.2),3X,2(3X,F7.4),4X,F7.2)
      STOP
      END
      FUNCTION ORDN(Z)
      ORDN=0.39894228*EXP(-Z*Z/2)
      RETURN
      END
      FUNCTION PNORM(X)
      DIMENSION C(7)
      DATA C/.319381530,-.356563782,1.781477937,
     &-1.821255978,1.330274429,.2316419,2.506628725/
      Y=X
      IF(X.LT.0.) Y=-X
      T=1./(1.+C(6)*Y)
      S=((((C(5)*T+C(4))*T+C(3))*T+C(2))*T+C(1))*T
      PNORM=S*EXP(-Y*Y/2)/C(7)
      IF(X.GT.0.) PNORM=1.-PNORM
      RETURN
      END
```

Appendix B — Montgomery's(1982) Program as Implemented Naively in VBA

```
Public Sub Econ()
    Range("E3").Select     ' Select the upper-left output cell
    row = 0                ' Current output row offset
    A1 = Range("C3").Value ' Get data from input cells
    A2 = Range("C4").Value
    A3 = Range("C5").Value
    A3P = Range("C6").Value
    A4 = Range("C7").Value
    XLAM = Range("C8").Value
    DELT = Range("C9").Value
    G = Range("C10").Value
    D = Range("C11").Value
    AA=DELT^2*A3P/(A2+XLAM*A4*G)
    XK=0
    For I=1 To 10
    RHS=(1.2826+XK)/ORDN(XK)
    If RHS > AA Then GoTo 5
    XK=XK+0.5
    Next I
5   XK=XK-0.5
    If XK<0 Then XK=0
    N=Fix(((1.2826+XK)/DELT)^2+0.5)    ' NOTE: truncate expression
    NMIN=N-10
    If NMIN<=0 Then NMIN = 1
    NMAX = N+10
    STEPVAL = 0.5
    For I=NMIN To NMAX
    XK=0.5
```

```
    For J=1 To 3
    BESTFN=1E+38
    If J=2 Then STEPVAL=0.1
    If J=3 Then STEPVAL=0.01
6   ARG=-1*XK
    A=2*PNORM(ARG)
    ARG=DELT*Sqr(I)-XK
    P=PNORM(ARG)
    H=Sqr((A*A3P+A1+A2*I)/(XLAM*A4*(1/P-0.5)))
    B=H*(1/P-0.5+XLAM*H/12)+G*I+D
    OBJFN=(A4*XLAM*B+A*A3P/H+XLAM*A3)/(XLAM*B+1)+(A1+A2*I)/H
    If OBJFN>BESTFN Then GoTo 7
    BESTFN=OBJFN
    BESTA=A
    BESTP=P
    BESTK=XK
    BESTH=H
    XK=XK+STEPVAL
    GoTo 6
7   If J=3 Then GoTo 8
    XK=BESTK-STEPVAL
8   Next J
    ' Write outputs to the worksheet
    ActiveCell.Offset(row, 0).Value = I
    ActiveCell.Offset(row, 1).Value = BESTK
    ActiveCell.Offset(row, 2).Value = BESTH
    ActiveCell.Offset(row, 3).Value = BESTA
    ActiveCell.Offset(row, 4).Value = BESTP
    ActiveCell.Offset(row, 5).Value = BESTFN
    row = row + 1
    Next I
End Sub
Function ORDN(Z)
    ORDN=0.39894228*Exp(-Z*Z/2)
End Function
Function PNORM(X)
    Dim C(1 To 7)
    C(1)=.319381530
    C(2)=-.356563782
    C(3)=1.781477937
    C(4)=-1.821255978
    C(5)=1.330274429
    C(6)=.2316419
    C(7)=2.506628725
    Y=X
    If X<0 Then Y=-X
    T=1/(1+C(6)*Y)
    S=((((C(5)*T+C(4))*T+C(3))*T+C(2))*T+C(1))*T
    PNORM=S*Exp(-Y*Y/2)/C(7)
    If X>0 Then PNORM=1-PNORM
End Function
```

Appendix C — Montgomery's(1982) Program in "Modern" VBA

```
Option Explicit

Sub Econ()
    Dim n As Integer, nmin As Integer, nmax As Integer
    Dim row As Integer, j As Integer, i As Integer

    Dim a1 As Double, a2 As Double, a3 As Double, a3p As Double
    Dim a4 As Double, lambda As Double, delta As Double
    Dim aa As Double, k As Double, rhs As Double, g As Double
    Dim h As Double, p As Double, besth As Double
    Dim besta As Double, bestp As Double, bestfn As Double
    Dim arg As Double, a As Double, d As Double, objfn As Double
    Dim bestk As Double, b As Double, stepval(1 To 3) As Double
```

```
' Clear output cells in case they contain old data
Range("E3", "L200").Clear

' Select the upper-left cell of the output range
Range("E3").Select
row = 0

' Get data from input cells
a1 = Range("C3").Value
a2 = Range("C4").Value
a3 = Range("C5").Value
a3p = Range("C6").Value
a4 = Range("C7").Value
lambda = Range("C8").Value
delta = Range("C9").Value
g = Range("C10").Value
d = Range("C11").Value

aa = delta ^ 2 * a3p / (a2 + lambda * a4 * g)
k = 0
For i = 1 To 10
    rhs = (1.2826 + k) / ordn(k)
    If rhs > aa Then
        Exit For
    End If
    k = k + 0.5
Next i

k = k - 0.5
If k < 0 Then
    k = 0
End If

' FORTRAN converts reals to integers by truncating, but VBA
' rounds.  The VBA Fix function truncates like FORTRAN.
n = Fix(((1.2826 + k) / delta) ^ 2 + 0.5)
nmin = n - 10
If nmin <= 0 Then
    nmin = 1
End If
nmax = n + 10

stepval(1) = 0.5
stepval(2) = 0.1
stepval(3) = 0.01

For i = nmin To nmax
    k = 0.5
    For j = 1 To 3
        bestfn = 1E+38
        Do Until objfn > bestfn
            arg = -k
            a = 2 * WorksheetFunction.NormSDist(arg)
            arg = delta * Sqr(i) - k
            p = WorksheetFunction.NormSDist(arg)
            h = Sqr((a * a3p + a1 + a2 * i) / _
                (lambda * a4 * (1 / p - 0.5)))
            b = h * (1/p - 0.5 + lambda*h/12) + g*i + d
            objfn = (a4*lambda*b + a*a3p/h + lambda*a3) / _
                    (lambda*b + 1) + (a1 + a2*i)/h
            If objfn <= bestfn Then
                bestfn = objfn
                besta = a
                bestp = p
                bestk = k
                besth = h
                k = k + stepval(j)
```

```
                End If
            Loop
            k = bestk - stepval(j)
        Next j

        ' Write outputs to the worksheet
        ActiveCell.Offset(row, 0).Value = i
        ActiveCell.Offset(row, 1).Value = bestk
        ActiveCell.Offset(row, 2).Value = besth
        ActiveCell.Offset(row, 3).Value = besta
        ActiveCell.Offset(row, 4).Value = bestp
        ActiveCell.Offset(row, 5).Value = bestfn

        ' Format numbers with appropriate precision
        ActiveCell.Offset(row, 1).NumberFormat = "0.00"
        ActiveCell.Offset(row, 2).NumberFormat = "0.00"
        ActiveCell.Offset(row, 3).NumberFormat = "0.0000"
        ActiveCell.Offset(row, 4).NumberFormat = "0.0000"
        ActiveCell.Offset(row, 5).NumberFormat = "0.00"

        row = row + 1
    Next i
End Sub

Function ordn(z As Double) As Double
    ordn = 0.39894228 * Exp(-z * z / 2)
End Function
```

Appendix D — AS 223 Subroutines for Optimum Ridge Parameter Selection as Originally Implemented in FORTRAN

```
      SUBROUTINE ORPS1(ALPHA, LAMDA, DELTA, IP, SIGMA, KNEW,
     *                 FM10, FM1K, FM20, FM2K, ITER, IFAULT)
C
C     ALGORITHM AS223  APPL. STATIST. (1987) VOL. 36, NO. 1
C
C     Calculates the optimum ridge parameter under the criterion
C     of minimizing the mean squared error of parameter estimation
C
      INTEGER FM20, IMAX, IP, ITER, IFAULT
      REAL ALPHA(IP), LAMDA(IP), DELTA, K, KNEW, KOLD, DK, FM10,
     *     FM1K, FM2K, DFM1K, DDFM1K, S, SIGMA
      REAL ZERO
C
      FM1(X, Y) = (Y * S**2 + (X * K)**2) / ((Y + K)**2 * S**2)
      FM2(X, Y) = Y * FM1(X, Y)
      DFM1(X, Y) = Y * (X**2 * K - S**2) / (Y + K)**3
      DDFM1(X, Y) = Y * (Y * X**2 - 2.0 * X**2 * K + 3.0 * S**2) /
     *                  (Y + K)**4
      DATA ZERO/0.0/
C
      IFAULT = 1
      IF (DELTA .LE. ZERO) RETURN
      IFAULT = 2
      IF (SIGMA .LE. ZERO) RETURN
      IFAULT = 3
      DO 10 J = 1, IP
        IF (LAMDA(J) .LE. ZERO) RETURN
   10 CONTINUE
      IFAULT = 0
      K = ZERO
      FM10 = ZERO
      S = SIGMA
      DO 20 J = 1, IP
        FM10 = FM10 + FM1(ALPHA(J), LAMDA(J))
   20 CONTINUE
      FM20 = IP
      ITER = 0
```

```
      IMAX = 20
      KNEW = ZERO
      DO 40 I = 1, IMAX
        ITER = ITER + 1
        FM1K = ZERO
        FM2K = ZERO
        DFM1K = ZERO
        DDFM1K = ZERO
        DO 30 J = 1, IP
          FM1K = FM1K + FM1(ALPHA(J), LAMDA(J))
          FM2K = FM2K + FM2(ALPHA(J), LAMDA(J))
          DFM1K = DFM1K + DFM1(ALPHA(J), LAMDA(J))
          DDFM1K = DDFM1K + DDFM1(ALPHA(J), LAMDA(J))
   30   CONTINUE
        KOLD = KNEW
        KNEW = KOLD - DFM1K / DDFM1K
        K = KNEW
        DK = ABS(KNEW - KOLD)
        IF (DK .LE. DELTA) GO TO 50
   40 CONTINUE
   50 RETURN
      END
C
C
C
      SUBROUTINE ORPS2(ALPHA, LAMDA, DELTA, IP, SIGMA, KNEW,
     *                 FM10, FM1K, FM20, FM2K, ITER, IFAULT)
C
C     ALGORITHM AS223  APPL. STATIST. (1987) VOL. 36, NO. 1
C
C     Calculates the optimum ridge parameter under the criterion
C     of minimizing the mean squared error of prediction.
C
      INTEGER FM20, IMAX, IP, ITER, IFAULT
      REAL ALPHA(IP), LAMDA(IP), DELTA, K, KNEW, KOLD, DK, FM10,
     *     FM1K, FM2K, DFM2K, DDFM2K, S, SIGMA
      REAL ZERO
C
      FM1(X, Y) = (Y * S**2 + (X * K)**2) / ((Y + K)**2 * S**2)
      FM2(X, Y) = Y * FM1(X, Y)
      DFM1(X, Y) = Y * (X**2 * K - S**2) / (Y + K)**3
      DFM2(X, Y) = Y * DFM1(X, Y)
      DDFM1(X, Y) = Y * (Y * X**2 - 2.0 * X**2 * K + 3.0 * S**2) /
     *                  (Y + K)**4
      DDFM2(X, Y) = Y * DDFM1(X, Y)
      DATA ZERO/0.0/
C
      IFAULT = 1
      IF (DELTA .LE. ZERO) RETURN
      IFAULT = 2
      IF (SIGMA .LE. ZERO) RETURN
      IFAULT = 3
      DO 10 J = 1, IP
        IF (LAMDA(J) .LE. ZERO) RETURN
   10 CONTINUE
      IFAULT = 0
      K = ZERO
      FM10 = ZERO
      S = SIGMA
      DO 20 J = 1, IP
        FM10 = FM10 + FM1(ALPHA(J), LAMDA(J))
   20 CONTINUE
      FM20 = IP
      ITER = 0
      IMAX = 20
      KNEW = ZERO
      DO 40 I = 1, IMAX
        ITER = ITER + 1
```

```
      FM1K = ZERO
      FM2K = ZERO
      DFM2K = ZERO
      DDFM2K = ZERO
      DO 30 J = 1, IP
        FM1K = FM1K + FM1(ALPHA(J), LAMDA(J))
        FM2K = FM2K + FM2(ALPHA(J), LAMDA(J))
        DFM2K = DFM2K + DFM2(ALPHA(J), LAMDA(J))
        DDFM2K = DDFM2K + DDFM2(ALPHA(J), LAMDA(J))
30    CONTINUE
      KOLD = KNEW
      KNEW = KOLD - DFM2K / DDFM2K
      K = KNEW
      DK = ABS(KNEW - KOLD)
      IF (DK .LE. DELTA) GO TO 50
40  CONTINUE
50  RETURN
    END
```

Appendix E — AS 223 Subroutines for Optimum Ridge Parameter Selection as Re-written in VBA

```
Option Explicit

' Global constants
Const MAX_ARRAY As Integer = 1000
Const BUTTONS As Integer = vbOKOnly + vbExclamation
Const TITLE As String = "Input Error"
Const ERROR1 As String = "Please make sure delta > 0."
Const ERROR2 As String = "Please make sure sigma > 0."
Const ERROR3 As String = "Please make all lambda values > 0."
Const ORPS_ONE As Boolean = True
Const ORPS_TWO As Boolean = False

' Global variables
Dim alpha(1 To MAX_ARRAY) As Double
Dim lambda(1 To MAX_ARRAY) As Double
Dim delta As Double, sigma As Double
Dim k As Double, s As Double
Dim ip As Integer

Function FM1(x As Double, y As Double) As Double
    FM1 = (y * s ^ 2 + (x * k) ^ 2) / ((y + k) ^ 2 * s ^ 2)
End Function

Function FM2(x As Double, y As Double) As Double
    FM2 = y * FM1(x, y)
End Function

Function DFM1(x As Double, y As Double) As Double
    DFM1 = y * (x ^ 2 * k - s ^ 2) / (y + k) ^ 3
End Function

Function DDFM1(x As Double, y As Double) As Double
    DDFM1 = y * (y * x^2 - 2 * x^2 * k + 3 * s^2) / (y + k)^4
End Function

Function DFM2(x As Double, y As Double) As Double
    DFM2 = y * DFM1(x, y)
End Function

Function DDFM2(x As Double, y As Double) As Double
    DDFM2 = y * DDFM1(x, y)
End Function
```

```
Sub ReadORPSInput(column As String)
    Dim i As Integer, j As Integer

    ' Clear output cells which may contain old data
    Range(column & "5", column & "10").Clear

    ' Read input from the worksheet
    ip = 0
    For i = 1 To MAX_ARRAY
        If Trim(Cells(i + 4, 1).Text) = "" Then
            ' Stop reading when we find an empty cell
            Exit For
        End If

        alpha(i) = Cells(i + 4, 1).Value
        lambda(i) = Cells(i + 4, 2).Value
        ip = ip + 1
    Next i

    delta = Range("c5").Value
    sigma = Range("d5").Value

    If (delta <= 0) Then
        Call MsgBox(ERROR1, BUTTONS, TITLE)
        Exit Sub
    End If

    If (sigma <= 0) Then
        Call MsgBox(ERROR2, BUTTONS, TITLE)
        Exit Sub
    End If

    For j = 1 To ip
        If (lambda(j) <= 0) Then
            Call MsgBox(ERROR3, BUTTONS, TITLE)
            Exit Sub
        End If
    Next j

End Sub

Sub WriteORPSResults(kNew As Double, fm10 As Double, _
                     fm1k As Double, fm20 As Integer, _
                     fm2k As Double, iter As Integer, _
                     column As String)
    ' Write output to worksheet
    Range(column & "5").Value = kNew
    Range(column & "5").NumberFormat = "0.000000"
    Range(column & "6").Value = fm10
    Range(column & "6").NumberFormat = "0.000"
    Range(column & "7").Value = fm1k
    Range(column & "7").NumberFormat = "0.000"
    Range(column & "8").Value = fm20
    Range(column & "8").NumberFormat = "0.0"
    Range(column & "9").Value = fm2k
    Range(column & "9").NumberFormat = "0.000"
    Range(column & "10").Value = iter

End Sub

Sub ORPS1()
    Call ORPS("G", ORPS_ONE)
End Sub

Sub ORPS2()
    Call ORPS("J", ORPS_TWO)
End Sub
```

```
Sub ORPS(column As String, algorithm As Boolean)
    Const ZERO As Double = 0
    Dim kNew As Double, kOld As Double, dk As Double
    Dim fm10 As Double, fm1k As Double, fm2k As Double
    Dim fm20 As Integer, iMax As Integer, iter As Integer
    Dim dfmk As Double, ddfmk As Double
    Dim i As Integer, j As Integer, iv As Integer

    Call ReadORPSInput(column)

    k = ZERO
    fm10 = ZERO
    s = sigma
    For j = 1 To ip
        fm10 = fm10 + FM1(alpha(j), lambda(j))
    Next j

    fm20 = ip
    iter = 0
    iMax = 20
    kNew = ZERO
    For i = 1 To iMax
        iter = iter + 1
        fm1k = ZERO
        fm2k = ZERO
        dfmk = ZERO
        ddfmk = ZERO
        For j = 1 To ip
            fm1k = fm1k + FM1(alpha(j), lambda(j))
            fm2k = fm2k + FM2(alpha(j), lambda(j))
            If algorithm = ORPS_ONE Then
                dfmk = dfmk + DFM1(alpha(j), lambda(j))
                ddfmk = ddfmk + DDFM1(alpha(j), lambda(j))
            Else
                dfmk = dfmk + DFM2(alpha(j), lambda(j))
                ddfmk = ddfmk + DDFM2(alpha(j), lambda(j))
            End If
        Next j
        kOld = kNew
        kNew = kOld - dfmk / ddfmk
        k = kNew
        dk = Abs(kNew - kOld)
        If (dk <= delta) Then
            Exit For
        End If
    Next i

    Call WriteORPSResults(kNew,fm10,fm1k,fm20,fm2k,iter,column)

End Sub
```