

A step by step guide to using the NAG Fortran Library with Microsoft Excel 2003

Michael Croucher, University of Manchester

1. Introduction

Microsoft Excel is a widely available and easy to use spreadsheet package that has applications in many areas of numerical computing but its support for higher-level mathematics is somewhat lacking in various areas. The NAG Fortran Library, on the other hand, is a large set of highly accurate mathematical routines in areas such as the evaluation of special functions, non-linear optimization, statistics, numerical integration, curve fitting and linear algebra.

By using a little Visual Basic for Applications (VBA) it is possible to utilize the full power of the NAG Library from within Excel. This allows investigators to use all of their Excel Macros and templates alongside a highly respected set of advanced numerical routines.

This guide describes how to use the NAG Fortran Library in Excel 2003. It is recognised that some users will want to interface the NAG C Library and Excel. The NAG website gives C users such help <http://www.nag.co.uk/numeric/callingDLLsfromotherlang.asp>.

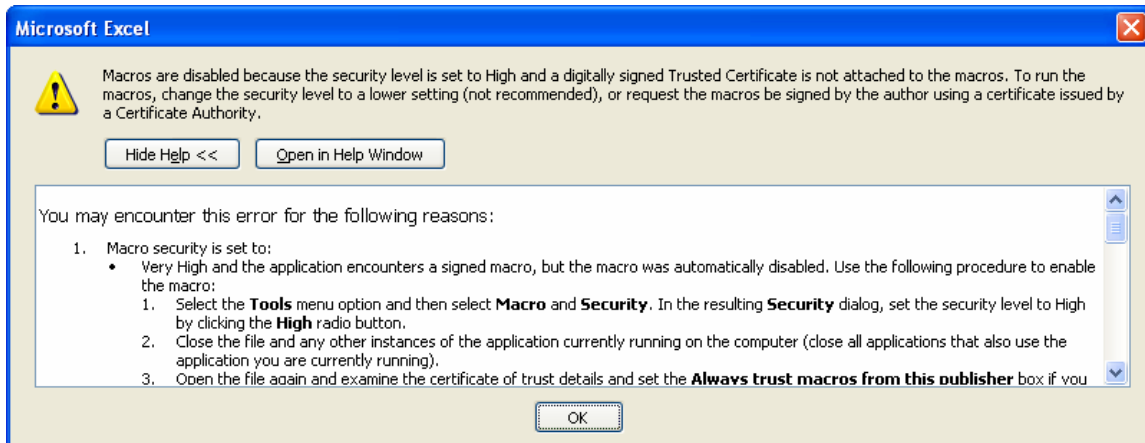
2. Which version of Excel and the NAG Fortran Library do you need?

This document is intended for use with Microsoft Excel 2003.

The NAG Fortran Library is available for a wide range of hardware, compiler and operating system configurations. All of the examples in this document were prepared and tested using **FLDLL214ML** which is a 32 bit Microsoft Windows DLL version of the libraries compiled using the Intel Visual Fortran compiler.

3. Running the Example Programs

All of the example programs discussed in this document are available for download from NAG's website <http://www.nag.co.uk/doc/techrep/index.asp#tr0208> in the file tr0208.zip. If you try to open any of them in Excel then you may receive the following error message:



To run the example programs, you will need to change the Macro security level as follows.

Click on **Tools->Macro->Security** and select the **Medium** security level. In future, whenever you attempt to open an Excel Spreadsheet that contains VBA macros, you will receive a security warning but you will be able to proceed by clicking on 'Enable Macros'.

4. Your first NAG-Excel program – Calling a Bessel Function

The functions in the **S (Approximation of Special Functions)** chapter of the NAG Library are among some of the easiest to use and so we are going to start there with the function **S17AEF** which calculates the Bessel Function $J_0(x)$.

1. After opening a new Excel spreadsheet, press the **Alt** and **F11** keys simultaneously to open the VBA Editor.
2. In the VBA Editor click on **Insert->Module** to open the Module Window – This is where we will construct our code.
3. Copy and paste the following code into the Module Window.

'Listing 1 – basic interface for S17AEF

Option Explicit

Option Base 1

Declare Function S17AEF Lib "FLDLL214M_nag.dll" (_

```
ByRef X As Double, _  
ByRef IFAIL As long _  
) as Double
```

4. Save the file and return to the Spreadsheet Window
5. In Cell A1 type the expression =S17AEF(1.0,1) which evaluates the function at $x=1.0$. The second input argument, **1**, determines how the NAG function will handle errors – see the code walkthrough, below, for more details.

If everything has worked OK then you should see the expression replaced with the result of the expression $J_0(1.0) = 0.765198$.

Let's go through the code a piece at a time:

'Listing 1 - basic interface for S17AEF

This is just a comment that explains what this code does. All comments in VBA start with an apostrophe '

Option Explicit

This statement forces you to declare all variables before they are used which is just good programming practice.

Option Base 1

The NAG Fortran Library is obviously written in Fortran where array variables are usually indexed starting from one by default. In VBA, however, array indices start at 0 by default. This line changes the VBA behaviour to bring it in line with the Fortran convention. It is not actually necessary for this particular example since it does not use arrays in any way – but most non-trivial programs do and so including it is a good habit to get into.

Declare Function S17AEF Lib "FLDLL214M_nag.dll" (_

Here, we are declaring a function called S17AEF and telling VBA that the executable for that function is contained in the DLL file FLDLL214M_nag.dll. The opening bracket (signifies the start of the function definition itself and the _ is a continuation symbol. Without the continuation symbol VBA would think that the (was the end of this particular statement and your code would not run.

The NAG function name S17AEF is case sensitive in this part of the code so using s17aef or S17aef here would fail. However, once declared, you can use either S17AEF or s17aef inside your spreadsheet.

ByRef X As Double, _

This is the first input argument of the function we are declaring – a variable **X** with type **Double**. If you have used VBA before you might not have been expecting at the **ByRef** beginning. This indicates that when we call the function, the variable is passed by reference rather than by value. This is actually the default behaviour of VBA (and

Fortran too for that matter) and so, technically speaking, we could leave it out but it is considered good practice to include it explicitly.

ByRef IFAIL As Long _

This is the second argument of our function – a variable **IFAIL** with type **Long**. **IFAIL** is used for both input and output and is used by the NAG routine for error handling.

When used as an input variable **IFAIL** determines how the NAG routine will handle error messages. If you set **IFAIL** to be 1 inside your spreadsheet (as we did in the example on page 2) then you are asking the routine for a *quiet return*. This means that if the routine encounters an error during calculation then it will not attempt to issue an error message.

You could also pass a value of -1 to **IFAIL** inside Excel which would ask the routine for a *noisy return*. In this mode the routine would output an error message and then attempt to continue execution.

Finally, you could pass a value of 0 to **IFAIL** which would ask the routine for a *hard return*. This reports the error message and then stops execution. This is almost certainly something that you do not want to do when using the libraries inside Excel since it will cause Excel to close. As an example try evaluating **=S17AEF(10E100,0)** inside your Excel spreadsheet and see what happens.

IFAIL is also used as an output variable by the NAG routine. Once the routine has finished its work, it will modify the value of **IFAIL** to indicate success or failure. More information on how to use this feature will be given in section 5. For a more detailed description of the **IFAIL** parameter, the reader is referred to chapter P01 of the NAG Fortran Documentation [1].

) as Double

This is the final line of our function declaration and indicates that the return type of S17ACF is of type **Double**.

5. Writing VBA Function Declarations for the NAG routines

If you have used the NAG Library before then you will know that many of its routines have very long and complicated lists of arguments (**E04CCA** for example) and so the VBA function declarations are going to be tiresome to type out.

Fortunately, NAG has done all of the hard work for you. Included with the Fortran Library is a file called **vb6.txt** that contains VBA function declarations for every single one of the routines in the library. All you need to do is search this file for the routine you want and then copy and paste the declaration into your own code – it's that simple.

If you installed the libraries in the default location then this file is located at **C:\Program Files\NAG\FL21\fldll214m\vb_headers\vb6.txt** but you can also open it from the start menu by clicking on

Start->All Programs->NAG->FL21-> VB6 Declare Statements.lnk

6. Improving the Bessel function code

Although it is a useful illustrative example of how to call a NAG routine from VBA, the code given in listing 1 is a bit raw and not really suitable for production use. A better attempt at writing our own Bessel function is given in the following listing.

```
'Listing 2 - Improved interface to S17AEF
Option Explicit
Option Base 1

Declare Function S17AEF Lib "FLDLL214M_nag.dll" ( _
    ByRef x As Double, _
    ByRef IFAIL As Long _
) As Double

Function Bessel(x As Double) As Variant
Dim IFAIL As Long
IFAIL = 1
Bessel = S17AEF(x, IFAIL)
If (IFAIL <> 0) Then
    Bessel = "Argument too large"
End If
End Function
```

To use this function in Excel all the user has to do is evaluate something like

=Bessel(1.0)

in his spreadsheet which is much more user friendly than S17AEF(1.0,1). Error handling is a little more elegant too which can be seen if you evaluate the following in Excel once you have written the VBA code.

=Bessel(10E100)

As before, let's look at this code piece by piece. The first few lines of listing 2 are identical to listing 1 so we will not discuss that again. The new code starts with the following line

```
Function Bessel(x As Double) As Variant
```

Here we define a function called **Bessel** that only takes one input argument, **x**. Since this function will return a string when the argument is too large we have defined the output of the function as Variant.

```
Dim IFAIL As Long  
IFAIL = 1
```

Here we declare **IFAIL** as a long integer and set it to 1 so that, in the event of an error occurring, the NAG routine will stay quiet about it and not attempt to throw its own error message to the user. This allows us to control exactly how we handle errors using our own code.

```
Bessel = S17AEF(x, IFAIL)
```

This is where the actual calculation is done – simply set the return value of **Bessel** to the output of our original function **S17AEF**.

```
If (IFAIL <> 0) Then  
    Bessel = "Argument too large"  
End If
```

In section 3 it was mentioned that the value of **IFAIL** after a call to a NAG routine indicates the success or failure of a calculation. If it becomes zero then the calculation was successful and you can be sure of the result. If it is any value other than 0 then something has gone wrong and the result is probably meaningless. This section of the code checks to see if **IFAIL** is or not and if it isn't it returns our own error message.

7. Using NAG routines that make use of callback functions

In this section we are going to write some VBA code that evaluates the following integral

$$\int_a^b \frac{4}{x^2 + 1} dx$$

Where *a* and *b* are two real numbers that a user can define inside a spreadsheet. A NAG routine that is suitable for this kind of problem is **D01AHF** which implements a method described by Patterson [2]. The VBA code that we need for this is given in listing 3 below.

```
'Listing 3 - Demonstration of callback functions
```

```
Option Explicit
```

```
Option Base 1
```

```
Declare Function D01AHF Lib "FLDLL214M_nag.dll" ( _  
    ByRef a As Double, _  
    ByRef b As Double, _  
    ByRef epsr As Double, _  
    ByRef npts As Long, _
```

```

ByRef relerr As Double, _
ByVal F As Long, _
ByRef nlimit As Long, _
ByRef IFAIL As Long _
) As Double

```

```

Function funcy(x As Double) As Double
funcy = 4 / (1 + x * x)
End Function

```

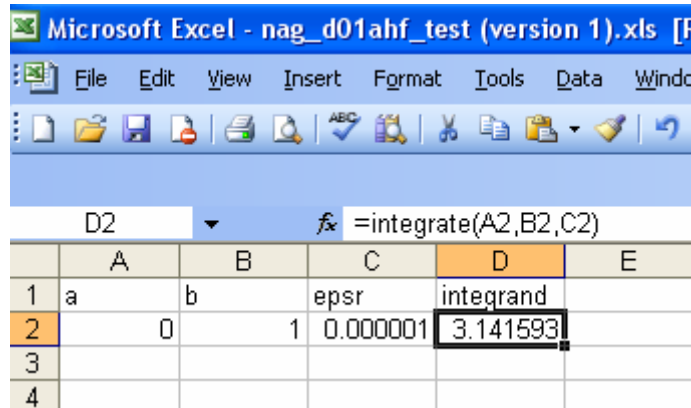
```

Function integrate(a As Double, b As Double, epsr As Double)
Dim relerr As Double
Dim nlimit As Long, IFAIL as Long, npts As Long
nlimit = 0
integrate = D01AHF(a, b, epsr, npts, relerr, AddressOf
funcy, nlimit, IFAIL)
End Function

```

To make use of this in Excel use you should

1. Open up a new Excel spreadsheet and press the **Alt** and **F11** at the same time to open the VBA Editor.
2. In the VBA Editor click on **Insert->Module** to open the Module Window.
3. Copy and paste the above code into the Module Window.
4. Save the file and return to the Spreadsheet Window
5. Set up the spreadsheet so that it looks like the example below. Note that if you change the limits a and b the integral is automatically re-evaluated.



As before, let's break down this program piece by piece. We have seen the two **Option** lines before and the block of code that starts with **Declare Function D01AHF** was just copied from the **vb6.txt** file as described in the previous section so the first piece of new code is

```

Function funcy(x As Double) As Double
funcy = 4 / (1 + x * x)
End Function

```

This is the function that is to be integrated and so if you wanted to integrate a different function then all you need to do is change this block of code.

We don't know in advance the values of x where the function **funcy** is to be evaluated since that is something that will be decided by the NAG routine **D01AHF** at runtime. In order to do this, **D01AHF** needs to be able to evaluate **funcy** wherever it chooses and so we need to pass **funcy** to it somehow. In this context we refer to the function **funcy** as a **callback function**.

```
Function integrate(a As Double, b As Double, epsr As Double)
```

In order to simplify things for the user of our spreadsheet we are wrapping the NAG routine in a function called **integrate**. The only input arguments are the upper and lower integration limits, **a** and **b**, along with the required relative accuracy **epsr**. All of the other arguments required by the NAG routine itself will be taken care of inside this function definition – safely hidden away from the spreadsheet user.

```
Dim relerr As Double  
Dim nlimit As Long, IFAIL as Long, npts As Long  
nlimit = 0
```

This is just a set of declarations for the variables needed by **D01AHF**. For the sake of clarity, all variables have been given the names used in the NAG Fortran library Documentation [1] to which the reader is referred to for details.

```
integrate = D01AHF(a, b, epsr, npts, relerr, AddressOf  
funcy, nlimit, IFAIL)
```

Here we call the NAG routine with all of the required arguments as detailed in the documentation of **D01AHF**. The key thing to notice here is **AddressOf funcy** which passes the address of our callback function to the NAG routine **D01AHF**. This allows **D01AHF** to evaluate **funcy** wherever it likes.

8. Using functions that use arrays as an input argument.

In this section we will learn how to pass arrays to the NAG routines. In order to do so we will use the routine **F03AAF**, which returns the determinant of a real matrix, as an example. Our first example will calculate the determinant of the following 3 x 3 matrix.

$$\begin{pmatrix} 33 & -24 & -8 \\ 16 & -10 & -4 \\ 72 & -57 & -17 \end{pmatrix}$$

'Listing 4 - Array example using F03AAF

Option Explicit

Option Base 1

```
Declare Sub F03AAF Lib "FLDLL214M_nag.dll" ( _  
    ByRef a As Double, _  
    ByRef IA As Long, _  
    ByRef n As Long, _  
    ByRef det As Double, _  
    ByRef WKSPCE As Double, _  
    ByRef IFAIL As Long _  
)
```

```
Function det() As Double  
    Dim a(3, 3) As Double  
    Dim IA As Long  
    Dim n As Long  
    Dim WKSPCE(3) As Double  
    Dim determinant As Double  
    Dim IFAIL As Long  
    IFAIL = 1  
    IA = 3  
    n = 3  
    a(1, 1) = 33  
    a(1, 2) = -24  
    a(1, 3) = -8  
    a(2, 1) = 16  
    a(2, 2) = -10  
    a(2, 3) = -4  
    a(3, 1) = 72  
    a(3, 2) = -57  
    a(3, 3) = -17  
    Call F03AAF(a(1, 1), IA, n, determinant, WKSPCE(1), IFAIL)  
    det = determinant  
  
End Function
```

To use this function inside Excel simply type the following into any cell

=det()

If everything goes according to plan then you will obtain the result of the calculation which is 6. So, let's look at the code itself.

Similar to previous examples, the block of code that starts with **Declare Sub F03AAF** was simply copied from the **vb6.txt** file as described in section 4. The first unfamiliar piece of code to us is

```
Dim a(3, 3) As Double
```

This simply creates a VBA array with the name **a** that contains 3 rows (1st argument) and 3 columns (2nd argument). This will later be used to hold the values of the matrix whose determinant we wish to calculate.

```
Dim WKSPCE(3) As Double
```

As described in the NAG documentation, the function **F03AAF** requires an area of memory to be set aside which it will use as workspace. The dimension of this workspace must be at least N for an N x N matrix and so here it is set to 3.

```
a(1, 1) = 33  
a(1, 2) = -24
```

These, along with the other 7 subsequent lines, define our array element by element. The general syntax is

```
ArrayName(row, col) = value
```

```
Call F03AAF(a(1, 1), IA, n, determinant, WKSPCE(1), IFAIL)
```

Here we call the NAG routine itself. Note that instead of simply using the array names **a** and **WKSPCE**, we must pass the first element of each array **a(1, 1)** and **WKSPCE(1)**. The result of the calculation is placed in the variable **determinant**.

```
det = determinant
```

Here we return the result of our calculation to the calling function.

This is all well and good but it would be much more user friendly if we could produce a function that calculates the determinant of any arbitrary square matrix entered into a spreadsheet. Listing 5 is a skeleton version of just such a function.

'Listing 5 - More advanced array example using F03AAF

Option Explicit

Option Base 1

```
Declare Sub F03AAF Lib "FLDLL214M_nag.dll" ( _  
    ByRef a As Double, _  
    ByRef IA As Long, _  
    ByRef n As Long, _  
    ByRef det As Double, _  
    ByRef WKSPCE As Double, _  
    ByRef IFAIL As Long _  
)
```

Function determinant(a As Range)

Dim myArray() As Double

Dim x As Long

Dim y As Long

Dim result As Double

Dim WKSPCE() As Double

Dim IFAIL As Long

Call Assemble(a, myArray) 'Takes the range a and puts it
into the vba array myArray

'get array dimensions

x = UBound(myArray, 1)

y = UBound(myArray, 2)

If (x <> y) Then

 determinant = "Error: Given range is not square"

 Exit Function

End If

'create workspace

ReDim WKSPCE(x)

IFAIL = 1

Call F03AAF(myArray(1, 1), x, x, result, WKSPCE(1), IFAIL)

determinant = result

End Function

Sub Assemble(x As Range, a As Variant)

' Takes an argument X and form a VB array A (ReDimmed)

Dim i As Long, j As Long

```
Dim m As Long, n As Long
```

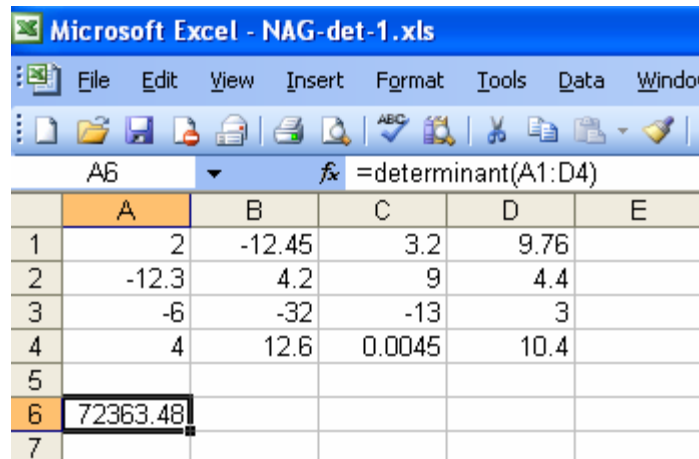
```
'Now get the overall dimensions of the matrix  
m = X.Rows.Count  
n = X.Columns.Count
```

```
ReDim a(m, n) ' VB Array redimensioned
```

```
' Assemble the matrix A  
For j = 1 To n  
  For i = 1 To m  
    a(i, j) = X.Cells(i, j).Value  
  Next i  
Next j
```

```
End Sub
```

The function **determinant** defined in the code above allows the Excel user to calculate the determinant of any square matrix entered into a spreadsheet. An example of its use for a 4x4 matrix is given in the screenshot below



The screenshot shows a Microsoft Excel spreadsheet titled "Microsoft Excel - NAG-det-1.xls". The formula bar shows the formula `=determinant(A1:D4)` entered in cell A6. The spreadsheet contains a 4x4 matrix in cells A1:D4 with the following values:

	A	B	C	D	E
1	2	-12.45	3.2	9.76	
2	-12.3	4.2	9	4.4	
3	-6	-32	-13	3	
4	4	12.6	0.0045	10.4	
5					
6	72363.48				
7					

We have seen most of the concepts in this code before – the first new item concerns the function definition:

Function determinant(a As Range)

This simply lets VBA know that we are going to pass a Range (such as **A1:D4**) to our function, rather than a Double as we have done in our previous functions.

Dim myArray() As Double

This line simply declares an empty VBA array called **myArray**.

The situation we find ourselves in is as follows: An Excel user prefers to work with Ranges such as **A1:D4** but the NAG routines will not accept Ranges – they will only

accept VBA arrays. So, our strategy will be to use a function called **Assemble** to create a temporary VBA array and copy the data specified by the Range into it. We will then pass this array to the NAG routine for processing. The **Assemble** function itself is defined later in our code.

Dim WKSPCE() As Double

As we learned in Listing 4, the NAG routine **F03AAF** requires some memory to use as workspace and will need an array of at least N Doubles when dealing with an N x N matrix. At this stage we have no idea how big the input Range is so we simply create an empty workspace array which we will resize later.

Call Assemble(a, myArray)

Assemble is a function defined later in the program which takes the values in the Range **a** and puts them in the VBA array **myArray**.

```
x = UBound(myArray, 1)
```

```
y = UBound(myArray, 2)
```

UBound is a standard VBA function that returns the largest available subscript for the given array and dimension. For example if **myArray** was an array declared with the command **Dim myArray(3, 4) As Double** then **x** would be 3 and **y** would be 4. Similarly, if the **Assemble** function were to produce a 5 x 5 VBA array then both **x** and **y** would be 5.

If (x <> y) Then

```
    determinant = "Error: Given range is not square"
```

```
    Exit Function
```

End If

The determinant of a matrix is only defined for square matrices so here we return an error message to the user and exit from the function if the two matrix dimensions **x** and **y** are not equal.

ReDim WKSPCE(x)

At this stage in our program we know that **x = y** and that we are going to attempt to calculate the determinant of an **x** by **x** matrix. Here we use the **ReDim** command to set our workspace to the required size.

Call F03AAF(myArray(1, 1), x, x, result, WKSPCE(1), IFAIL)

This is almost exactly the same call to **F03AAF** as we had in Listing 5. Again, note that whenever we pass an array to the routine we must pass its first element.

Sub Assemble(x As Range, a as Variant)

This is the beginning of the code for the **Assemble** subroutine which takes a Range **x** and returns an array **a**. It is fairly self explanatory and commented so we will skip the detail. It is worth noting that, as written, Assemble only works for contiguous selections

in the worksheet. For example if you use the Ctrl key to select columns A and C, then only column A will be used.

9. Using functions that return an array.

Finally, we turn to functions that return an array of values to the calling spreadsheet. As an example we will look at **G01AAF** which returns a set of basic statistics such as mean, standard deviation and kurtosis for a given set of input values.

'Listing 6 - Array output example using F01ABF

Option Explicit

Option Base 1

```
Declare Sub G01AAF Lib "FLDLL214M_nag.dll" ( _  
    ByRef N As Long, _  
    ByRef X As Double, _  
    ByRef IWT As Long, _  
    ByRef WT As Double, _  
    ByRef Xmean As Double, _  
    ByRef S2 As Double, _  
    ByRef S3 As Double, _  
    ByRef S4 As Double, _  
    ByRef Xmin As Double, _  
    ByRef Xmax As Double, _  
    ByRef WTSum As Double, _  
    ByRef IFAIL As Long _  
)
```

Function getstats(A As Range) As Variant

Dim myArray() As Double

Dim N As Long 'The number of observations

Dim IWT As Long 'Are weights supplied by the user?

Dim WT() As Double 'Array that contains weights if used.

Dim IFAIL As Long

Dim results(7) As Double

Dim Xmean As Double 'The mean

Dim S2 As Double 'The standard Deviation

Dim S3 As Double 'The coefficient of Skewness

Dim S4 As Double 'The coefficient of Kurtosis

Dim Xmin As Double 'The smallest value in the sample

Dim Xmax As Double 'The largest value in the sample

Dim WTSum As Double 'The sum of the weights

Call Assemble(A, myArray)

'get array dimension

```

N = UBound(myArray, 1)
ReDim WT(N)

IFAIL = 1
IWT = 0 'No user supplied weights
Call G01AAF(N, myArray(1), IWT, WT(1), Xmean, S2, S3, S4,
Xmin, Xmax, WTSum, IFAIL)

'Populate results array
results(1) = Xmean
results(2) = S2
results(3) = S3
results(4) = S4
results(5) = Xmin
results(6) = Xmax
results(7) = WTSum
'return results
getstats = Application.Transpose(results)
End Function

Sub Assemble(X As Range, A As Variant)
' Takes an argument X and form a VB array A (ReDimmed)
Dim s As String
Dim j As Long
Dim m As Long, N As Long

'Now get the overall dimension of the matrix
N = X.Rows.Count

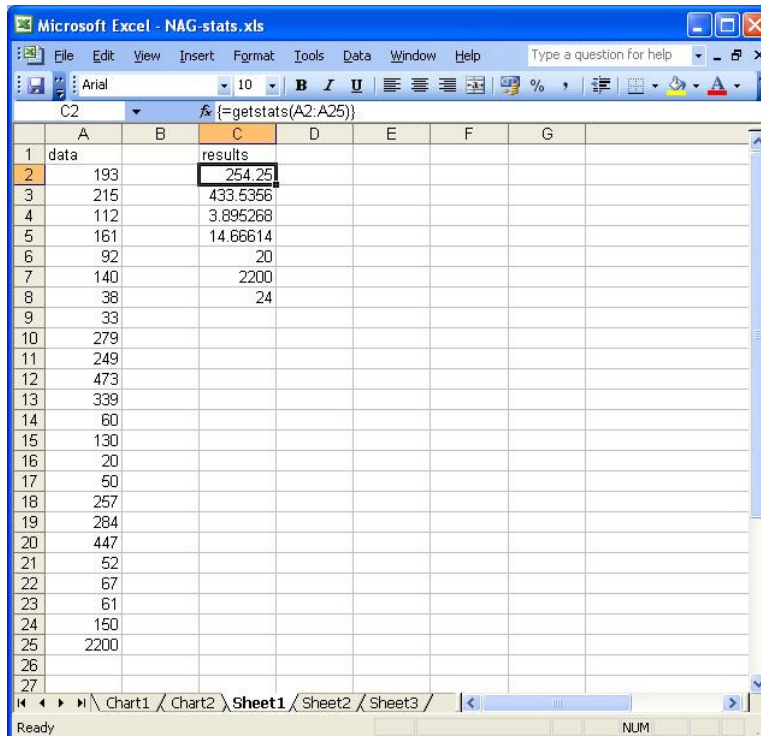
ReDim A(N) ' VB Array redimensioned

'Assemble the matrix A
For j = 1 To N
    A(j) = X.Cells(j).Value
Next j

End Sub

```

The `getstats()` function, defined by the code above, returns an array of 7 values that contains the set of statistics calculated by **G01AAF** and an example of its use is shown in the screenshot below. Notice the curly brackets in the formula `{=getstats(A2:A25)}` which is an example of an *array formula* in Excel. You might be tempted to reproduce the screenshot by actually typing `{=getstats(A2:A25)}` into a cell and pressing Enter but this **will not work!**



The correct way to enter this array formula is to highlight the cells that will contain the results array (in this case a single column of 7 rows), type **=getstats(A2:A25)** and press the **Ctrl, Shift** and **Enter** keys simultaneously. If you highlight less than 7 cells before entering the array formula then only the first few elements of the array will be displayed in the spreadsheet. If you highlight more than 7 cells then the extra cells will contain the value **#N/A**.

This method of entering array formulae into Excel is almost the only new thing we have to learn in this section as much of the code in Listing 6 is familiar to us by now. The only new thing we have done is to set up and return an array, **results**, rather than returning a single value.

getstats = Application.Transpose(results)

Here we return the Transpose of the one dimensional **results** array in order to force it to return a single column of values. If we wanted to return a single row of values instead then we would replace the above line with

getstats = results

Finally, the **Assemble** function is slightly different to that of Listing 5 as it only has to deal with a one dimensional Range

10. What we haven't told you

Although we have covered enough material in this document to enable a typical user to use many of the NAG routines from VBA, there are still several concepts that have not been discussed. Subject to demand these topics may be discussed in a future document and include

Complex Numbers

Callback functions that use arrays (required in functions like E04JYF)

How to deal with non-contiguous ranges

Acknowledgements

Thanks to the staff at NAG for their helpful advice, comments and explanations. Any mistakes that remain are mine alone.

References

1. NAG Fortran Library Documentation (accessed on June 4th 2008),
<http://www.nag.co.uk/numeric/FL/FLdocumentation.asp>
2. Patterson T N L (1968b) The Optimum addition of points to quadrature formulae
Math. Comput. **22** 847–856

If you need more information or assistance about the NAG Libraries and Excel, please contact our support team support@nag.co.uk

NAG Acknowledgement

NAG would like to thank Michael Croucher for his time in developing this article.