(Part 1) – It ain't what you do it's the way that you do it

Posted on May 25, 2011by fastexcel

Its fairly easy to write a User Defined Excel function using VBA:
Suppose you want to write a function that calculates the average of a range of cells, but exclude from the average anything that is not a number or is less than a tolerance.
Lets call the Function AverageTol
Start Excel
Alt-F11 gets you to the Visual Basic Editor (VBE)
Insert–>Module
Enter the following VBA Code

```
1           Function AverageTol(theRange, dTol)
2           For Each Thing In theRange
3           If IsNumeric(Thing) Then
4           If Abs(Thing) > dTol Then
5           AverageTol = AverageTol + Thing
6           lCount = lCount + 1
7           End If
8           End If
9           Next Thing
10          AverageTol = AverageTol / lCount
11          End Function
```

The function loops through every cell in the range and, if the cell is a number greater than the tolerance, adds it to the total and increments a count. Finally it divides the total by the count and returns the result.
Now go back to the Excel worksheet, enter some data in cells A1:A10 and in B1 enter =AverageTol(A1:A10 , 5)
That was pretty easy, and works well for 10 cells.
But if you have a lot of data, say 32000 cells, then 10 formulas using this UDF takes over 5 seconds to calculate on my fast PC (Intel I7 870 2.9 GHZ).
One major reason this is so slow is that I used all the defaults: I was lazy and did not declare any of the variables so they all defaulted to Variants.
Thats SLOW … but I can easily improve it: here is version A of AverageTol:

```
1           Function AverageTolA(theRange As Range, dTol As Double)
2           Dim oCell As Range
3           Dim lCount As Long
4           For Each oCell In theRange
5           If IsNumeric(oCell) Then
6           If Abs(oCell) > dTol Then
7           AverageTolA = AverageTolA + oCell
8           lCount = lCount + 1
9           End If
10          End If
11          Next oCell
12          AverageTolA = AverageTolA / lCount
13          End Function
```

This is the same function but with each variable declared as a sensible Type. This is good programming practice, and considerably faster.
10 formulas using this UDF on 32000 cells now calculates in 1.4 seconds, thats an improvement factor of 3.5 but still SLOW.
One reason its slow is that there is a large overhead each time a VBA program transfers data from an Excel cell to a VBA variable
And this function does that lots of times (3 times 32000).

If you transfer the data in one large block you can avoid much of this overhead:

```
Function AverageTolC(theRange As Range, dTol As Double)
Dim vArr As Variant
Dim v As Variant
Dim lCount As Long
'
On Error GoTo FuncFail
'
' get Range into a variant array
'
vArr = theRange
'
For Each v In vArr
If IsNumeric(v) Then
If Abs(v) > dTol Then
AverageTolC = AverageTolC + v
lCount = lCount + 1
End If
End If
Next v
AverageTolC = AverageTolC / lCount
Exit Function
FuncFail:
AverageTolC = CVErr(xlErrNA)
End Function
```

The statement vArr = theRange takes the values from all the cells in the Range and transfers it to a 2-dimensional Array of Variants. Then the UDF loops on each element of the Variant array. I also added an error handling trap that makes the UDF return #N/A if any unexpected error occurs.
Now the 10 formulas calculate in less than 0.1 seconds: thats an additional improvement factor of 14.
But we have'nt finished yet! Another speedup trick is to replace
vArr = theRange
with
vArr = theRange.Value2
That reduces the calculation time from 98 milliseconds (thousandths of a second) to 62 milliseconds. Using .Value2 rather than the default property (.Value) makes Excel do less processing (.Value checks to see if cells are formatted as Currency or Date, whereas .Value2 just treats all numbers including dates and currency as Doubles).
We can also make another small speedup by using Doubles rather than Variants wherever possible. Change the For Each v ... Next v loop to:

```
Dim d as Double
Dim r as Double
On Error GoTo skip
For Each v In vArr
d = CDbl(v)
If Abs(d) > dTol Then
r = r + d
lCount = lCount + 1
End If
skip:
Next v
```

Now the calculation time has come down to 47 milliseconds.
So a series of small changes has improved the calculation speed of this simple UDF from 5.4 seconds to 0.047 seconds, 115 times faster!

(Part 2) – using Excel Functions inside a UDF

Posted on June 6, 2011by fastexcel

In part 1 of Writing efficient VBA UDFs I looked at more efficient ways for the UDF to process a Range of data by reading it all into a Variant array. In this post I look at a case (using Excel Functions from within the UDF) where the most efficient way is somewhat different.

Linear interpolation is a commonly used technique for finding missing values or calculating a value that lies between the values given in a table.

Suppose you have a table of values like this:

Level	Flow1	Flow2
64.00	0.10	2.59
64.50	0.77	3.18
65.00	2.19	3.73
65.50	4.02	4.28
66.00	6.19	6.88
66.50	8.64	12.04
67.00	11.36	13.85
67.50	13.45	14.84
68.00	15.00	16.37
68.50	16.41	21.12
69.00	17.71	21.68

And you want to find out what the value of Flow1 would be for a level of 66.25. Assuming that the value is on a straight line between the Flow for 66.0, which is 6.19, and the flow for 66.5, which is 8.64, you can calculate it like this:

The difference between 8.64 and 6.19 is 2.45, and 66.25 is half-way between 66.0 and 66.5, so add half of 2.45 to 6.19=7.415.

As a formula this becomes =6.19+(8.64-6.19)*(66.25-66.0)/(66.5-66.0)

So writing a UDF the same way as in Writing VBA UDFs Efficiently Part 1 you get this (ignoring error handling etc. for the sake of simplicity):

```
1  Function VINTERPOLATEA(Lookup_Value as Variant, Table_Array as Range,
   Col_Num as long)
2  Dim vArr As Variant
3  Dim j As Long
4  ' get values
5  vArr = Table_Array.Value2
6  ' find first value greater than lookup value
7  For j = 1 To UBound(vArr)
8  If vArr(j, 1) &gt; Lookup_Value Then
9  Exit For
10 End If
11 Next j
12 ' Interpolate
13 VINTERPOLATEA = (vArr(j - 1, Col_Num) + _
14 (vArr(j, Col_Num) - vArr(j - 1, Col_Num)) * _
15 (Lookup_Value - vArr(j - 1, 1)) / (vArr(j, 1) - vArr(j - 1, 1)))
16 End Function
```

Where the Lookup_value is the value to find in the first column of the range Table_Array and Col_Num gives the column number index of the data to be interpolated (in this example 2).
This is reasonably efficient: 20 formulas interpolating on a table 10000 rows long takes 323 millisecs.
But of course we can do better!
When you look at this UDF you can see that the actual calculation only uses 2 rows of data, but to get that 2 rows it has to:

- import 10000 rows by 3 columns of data into an array
- do a linear search on the first column.

This sounds suspiciously like a LOOKUP or MATCH.
So lets try using Excel's MATCH function instead: you can call MATCH from inside your VBA UDF using Application.WorksheetFunction.MATCH. And since the data is sorted we can use approximate MATCH. Once we have got the row number from MATCH we can get the 2 rows we are interested in.
The new UDF looks like this (again I am ignoring error handling for the sake of simplicity):

```
1  Function VINTERPOLATEB(Lookup_Value As Variant, Table_Array As Range,
   Col_Num As Long)
2  Dim jRow As Long
3  Dim rng As Range
4  Dim vArr As Variant
5  '
6  ' create range for column 1 of input
7  '
8  Set rng = Table_Array.Columns(1)
9  '
10 ' lookup Row number using MATCH
11 ' Approx Match finds row for the largest value &lt; the lookup value
12 jRow = Application.WorksheetFunction.Match(Lookup_Value, rng, 1)
13 '
14 ' get 2 rows of data
15 '
16 vArr = Table_Array.Resize(2)<code>.Offset(jRow - 1, 0)</code>.Value2
17 '
18 ' Interpolate
19 '
20 VINTERPOLATEB = (vArr(1, Col_Num) + _
21 (vArr(2, Col_Num) - vArr(1, Col_Num)) * _
22 (Lookup_Value - vArr(1, 1)) / (vArr(2, 1) - vArr(1, 1)))
23 End Function
```

Once MATCH has found the row we can use Resize and Offset to subset the range to just the 2 required rows. (Thanks to Jim Cone for reminding me that you need to do the Resize before the Offset to make sure that the Offset does not cause an error by moving off the boundaries of the Worksheet).
This UDF takes 2 milliseconds on the test data, 160 times faster!
Update: As Peter Sestoft points out, much of the improvement is due to approximate Match using binary search rather than the linear search used in VINTERPOLATEA. In fact if you program the binary search in VBA inside the UDF it takes 3.3 milliseconds which is only 1.7 times slower than using MATCH.
Note: there are 2 ways of calling Excel Functions such as MATCH from VBA: Application.Match and Application.WorksheetFunction.Match. The differences are mostly in error handling (for instance when no match is found for the exact match option):

- Application.Match returns a Variant containing an error, which allows the use of IsError: If IsError(Application.Match …)
- Application.WorksheetFunction.Match raises a VBA error which requires an On Error handler.

Also WorksheetFunction.Match is somewhat faster.

So we need to add some error handling and boundary case handling:

- Use On Error to trap non-numeric data
- Check for the lookup value being outside the range of data in the table
- Check if the lookup value is the last value in the table

Then the UDF looks like this:

```
<span style="color:#008000;">Function VINTERPOLATEC(Lookup_Value As
Variant, Table_Array As Range, Col_Num As Long)
Dim jRow As Long
Dim rng As Range
Dim vArr As Variant
Dim vValue As Variant
On Error GoTo FuncFail
Set rng = Table_Array.Columns(1)
' Check for case if val = last value in rng
vValue = rng.Cells(rng.Rows.Count, 1).Value2
If Lookup_Value = vValue Then
VINTERPOLATEC = Table_Array.Cells(rng.Rows.Count, Col_Num).Value2
Exit Function
End If
' Return an error if lookup_value is not within rng
If Lookup_Value &gt; vValue Or Lookup_Value &lt; rng.Cells(1).Value2 Then
VINTERPOLATEC = CVErr(xlErrNA)
Exit Function
End If
' lookup Row number using MATCH
jRow = Application.WorksheetFunction.Match(Lookup_Value, rng, 1)
' get 2 rows of data
vArr = Table_Array.Resize(2)<code>.Offset(jRow - 1, 0)</code>.Value2
' Interpolate
VINTERPOLATEC = (vArr(1, Col_Num) + _
(vArr(2, Col_Num) - vArr(1, Col_Num)) * _
(Lookup_Value - vArr(1, 1)) / (vArr(2, 1) - vArr(1, 1)))
Exit Function
FuncFail:
VINTERPOLATEC = CVErr(xlErrValue)</span>
End Function
```

Conclusion:
The only thing faster than bringing all the data across to VBA in one lump is to use Excel functions to bring across only the minimum data your function needs.

(Part 3) – Avoiding the VBE refresh bug – Updated

Posted on June 13, 2011by fastexcel

In Part 1 and Part 2 of Writing Efficient VBA UDFs I looked at some simple ways of changing the VBA code you write to make it run massively faster. In this post I look at a bug in Excel that slows down your UDFs and show you how to avoid it.
A few years back I was having trouble working out why VBA UDFs ran so much faster on my PC than on someone else's system (I think it was Jan Karel Pieterse's). Eventually I tracked it down to the fact that I had FastExcel installed. And further investigation showed that FastExcel made the VBA UDFs ran faster because the calculation was initiated by FastExcel using VBA.
It turned out that the underlying reason was a little bug in the EXCEL Visual Basic Editor (the VBE) : each formula that contains a UDF changes the VBE title bar to say "Running" whenever a UDF is being executed during the calculation, and then switches it back again when the UDF has finished.

The VBE Title Bar

The word [Running] gets inserted into the title bar after the name of the workbook. To make this happen requires the VBE to send a message to the Windows screen handler and the window containing the VBE gets refreshed:

This is a CPU intensive operation: on my system I created a volatile do-nothing UDF (using Excel 2010 32-bit):

```
Function VolUDF()
Application.Volatile
End Function
```

And entered it A1:A10000 (ten thousand cells). Then I resized the windows so that I could see both the Excel window and the VBE windows at the same time and triggered a calculation: I could see the VBE title bar flashing.

The calculation took 7.3 seconds.

Then I closed the VBE window and triggered another calculation.

This time it took 3.6 seconds.

Then I saved the workbook with the VBE window closed, closed Excel, reopened Excel and the workbook and triggered another calculation:

This time it took 1.1 seconds.

Then I initiated the calculation from VBA using Application.Calculate:

This time it took 0.058 seconds. That's an improvement factor of 125.

And it still takes 0.058 seconds using Application.Calculate even with the VBE window open and visible.

Of course if you only have a few formulae using UDFs you will never notice this slowdown in calculation time, but on my fast system even 1000 formulas using UDFs will take an additional 0.7 seconds to calculate.

Bypassing the refresh bug

OK, so now you know what's happening how do you avoid this problem? (By the way this bug is present in all versions of Excel from Excel 97 to Excel 2013 (both 32 bit and 64 bit).)

Trapping Calculation in Manual calculation mode.

If Excel is in Manual Calculation mode you can trap all the keystrokes that trigger a calculation, and initiate the calculation from your VBA code.

You need calculation subs and keytraps for

- Shift/F9 – Activesheet.Calculate
- F9 – Application.Calculate
- Ctrl/Alt/F9 – Application.CalculateFull
- Ctrl/Alt/Shift/F9 – Application.CalculateFullRebuild

In the ThisWorkbook module add the key trapping subs:

```
1          Private Sub Workbook_Open()
2          Application.OnKey "+{F9}", "SheetCalc"
3          Application.OnKey "{F9}", "ReCalc"
4          Application.OnKey "^%{F9}", "FullCalc"
5          Application.OnKey "+^%{F9}", "FullDependCalc"
6          End Sub
```

and in an ordinary module add the corresponding calculation subs:

```
Sub SheetCalc()
ActiveSheet.Calculate
End Sub
Sub ReCalc()
Application.Calculate
End Sub
Sub FullCalc()
Application.CalculateFull
End Sub
Sub FullDependCalc()
Application.CalculateFullRebuild
End Sub
```

(The equivalent of these procedures is built-in to the FastExcel add-in)

In Automatic Calculation Mode

Unfortunately I do not know of a way of bypassing the problem using VBA in Automatic Calculation Mode.

Closing the VBE window and reopening Excel will improve things substantially, but if you are creating workbooks with UDFs for other people to use you cannot control whether they will have the VBE open or not.

The only solution seems to be to use some non-vba technology such as a VB6 Automation Addin, .NET addin, or XLL addin, but each of these approaches has their own difficulties, which I plan to cover in a future post comparing Excel UDF Technologies.

Update

A useful idea about handling the problem in Automatic Mode comes from Francisco Aller Labandeira.

He suggests adding this code to the Workbook module.

```
Private Sub Workbook_SheetChange(ByVal Sh As Object, ByVal Target As Range)
If Application.Calculation = xlCalculationAutomatic Then
Application.Calculation = xlCalculationManual
Exit Sub
End If
Calculate
End Sub
```

This has the drawback that it won't stop the first automatic calculation, but will catch subsequent calculations. You could also perhaps add some code to the Workbook BeforeSave event to switch back to Automatic if appropriate.

Conclusion:

If you need to use a large number (>1000) of formulas referencing VBA UDFs you will need to use Manual Calculation mode and add calculation key-trappers and handlers to your workbooks.

(Part 4) – Variants, References, Arrays, Calculated Expressions, Scalars

Posted on June 20, 2011by fastexcel

In part 1 and part 2 of "Writing efficient UDFs" I used parameters defined as Range to get data from Excel.

```
Function VINTERPOLATEB(Lookup_Value As Variant, Table_Array As Range,
Col_Num As Long)
```

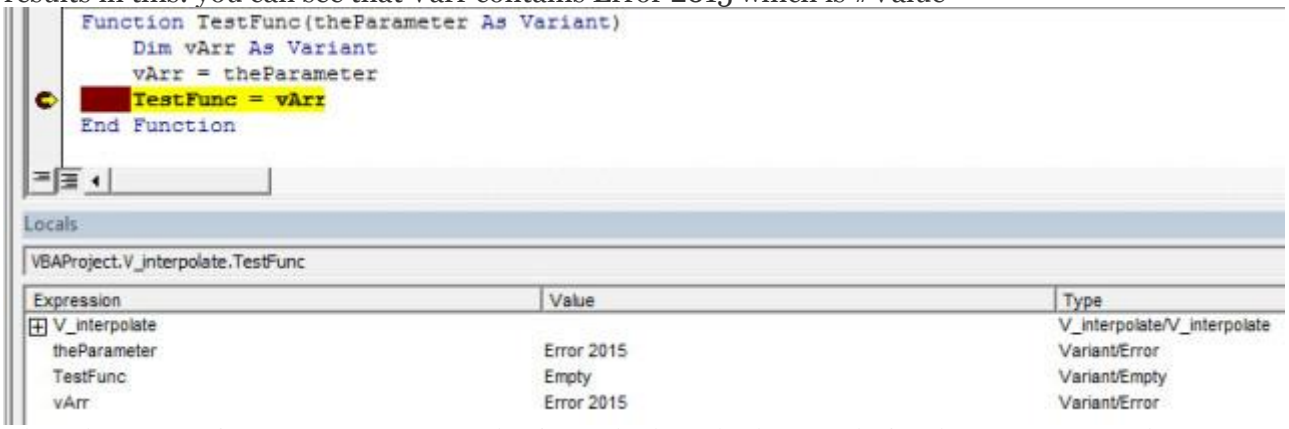
This works OK if the function is called from a formula using a range:

```
=VINTERPOLATEB($E5,$A$10:$C$10200,2)
```

but results in #Value if you use a calculated expression or an array of constants:
`{=VINTERPOLATEB($E5,($A$10:$C$10200*1),2)}`
This formula has to be entered as an array formula using Control/Shift/Enter (don't enter the { … } , Excel will add them).
`=VINTERPOLATEB(4.5,{1,3,3.5;4,4,4.5;5,4.5,5},2)`
This uses a 3 column 3 row array constant. You do have to enter the { … } surrounding the constants, but it does NOT have to be entered as an array formula. The , separates the columns and the ; separates the rows.
Excel detects that these parameters are not Ranges before even calling the function.
You can fix this by defining the parameter as a variant rather than a range: a variant parameter can hold virtually anything! But the UDF now has to handle all the different types of data that the Variant might contain.
One simple approach is to assign the parameter to a Variant: this will coerce everything to values:

```
1           Function TestFunc(theParameter As Variant)
2           Dim vArr As Variant
3           vArr = theParameter
4           TestFunc = vArr
5           End Function
```

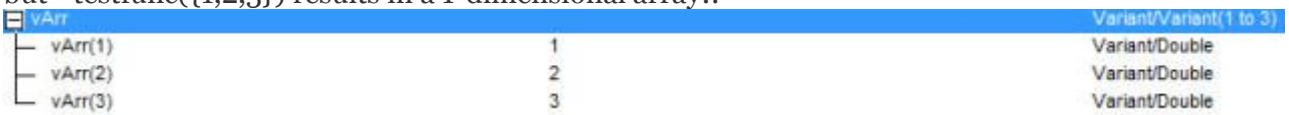
`=TestFunc($A$10:$A$15*1)`
In the VBE putting a breakpoint (use F9) on the return line and showing the Locals window results in this: you can see that Varr contains Error 2015 which is #Value



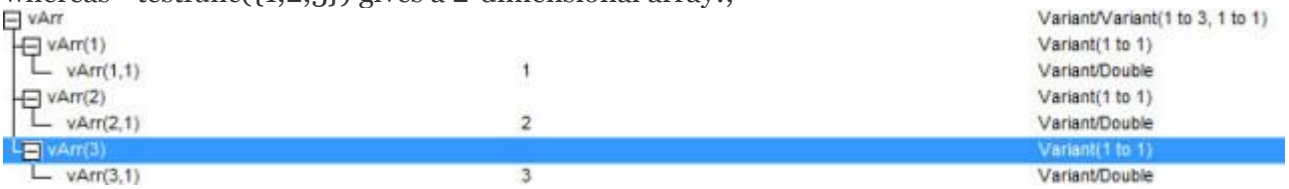
Thats because I forgot to array-enter the formula, here is the Locals for the array-entered formula:

Now you can see in the Locals window that the vArr variant contains a 2-dimensional array of variants with a sub-type of double.
Entering =testfunc({1,2,3;5,6,7}) also results in a 2 dimensional array:

but =testfunc({1,2,3}) results in a 1-dimensional array!:

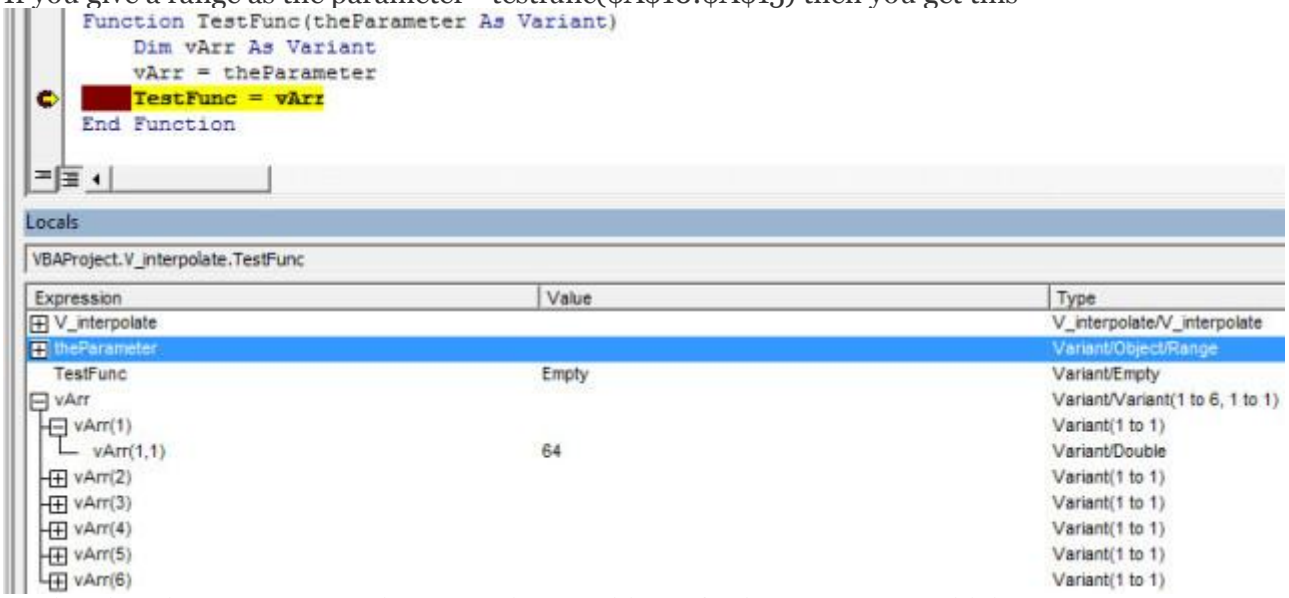


whereas =testfunc({1;2;3}) gives a 2-dimensional array!;



and =testfunc(45) gives a scalar, not an array;

If you give a range as the parameter =testfunc($A$10:$A$15) then you get this



Notice that theParameter variant contains an object of sub-type Range, which means you have to treat it as a Range Variable, whereas the vArr contains the values extracted from the Range.

Determining Type and Dimensions for a Variant parameter

So in a general purpose UDF you want to use Variant parameters, and you often need to determine the type and upper and lower bounds of the variant.

For maximum efficiency you cannot just use vArr=theVariant, because:

- You cannot use .Value2 because it might not be a range.
- In many cases you want to manipulate the Range object before/instead of just coercing all its values.

So here is a function to determine what has been passed, and how large it is:

```
1       Function Variant_Type(theVariant As Variant)
2       Dim jRowL As Long
3       Dim jRowU As Long
4       Dim jColL As Long
5       Dim jColU As Long
6       Dim jType As Long
7       Dim varr As Variant
8       '
9       ' theVariant could contain a scalar, an array, or a range
10      ' find the upper and lower bounds and type
11      ' type=1 range, 2 2-d variant array, 3 1-d variant array (single row of
12      columns), 4 scalar
13      '
14      On Error GoTo FuncFail
15      jType = 0
```

```
16      jRowL = 0
17      jColL = 0
18      jRowU = -1
19      jColU = -1
20      If TypeName(theVariant) = "Range" Then
21      jRowL = 1
22      jColL = 1
23      jRowU = theVariant.Rows.Count
24      jColU = theVariant.Columns.Count
25      jType = 1
26      ElseIf IsArray(theVariant) Then
27      jRowL = LBound(theVariant, 1)
28      jRowU = UBound(theVariant, 1)
29      On Error Resume Next
30      jColL = LBound(theVariant, 2)
31      jColU = UBound(theVariant, 2)
32      On Error GoTo FuncFail
33      If jColU &lt; 0 Then
34      jType = 3
35      jColL = jRowL
36      jColU = jRowU
37      jRowL = 0
38      jRowU = -1
39      Else
40      jType = 2
41      End If
42      Else
43      jRowL = 1
44      jRowU = 1
45      jColL = 1
46      jColU = 1
47      jType = 4
48      End If
49      Variant_Type = jType
50      Exit Function
51      FuncFail:
52      Variant_Type = CVErr(xlErrValue)
53      jType = 0
54      jRowU = -1
55      jColU = -1
        End Function
```

Note that the first test is whether the variant contains a Range. This is to avoid inadvertently coercing a Range to its values. Also there are several ways in VBA in determining the sub-type of a variant:

- If TypeOf theVariant Is Range Then
- If TypeName(theVariant) = “Range” Then

Beware of trying to use VarType(theVariant) : this does an under-the-covers coerce of a Range and then throws the resulting values away! (Expensive for large ranges).

Conclusion:
In a general purpose UDF you have to use Variant type parameters rather than Range type. You can handle this efficiently by determining what the variant contains before processing it.

(Part5) – UDF Array Formulas go faster!

Posted on June 20, 2011by fastexcel

Just in case you thought the previous posts on writing efficient VBA UDFs (Part1, Part2, Part3, Part 4) meant we had finished making UDFs run faster, think again – its time to explore UDF Array Formulas.

Single and Multi-Cell Array Formulas

Excel array formulas can do amazing things. They are like ordinary formulas except that you enter them with Control/Shift/Enter rather than just enter.

There are two kinds of array formulae:

- Single cell array formulae are entered into a single cell, loop through their arguments (which are often calculated arguments) and return a single answer.
- Multi-cell array formulae are entered into multiple cells and return an answer to each of the cells.

With this power comes a cost: because array formulae are doing a lot of work they can be slow to calculate (particularly single-cell array formulas).

UDF Multi-cell Array Formulas go Faster!

You can break down the time taken by a VBA UDF into these components:

- Overhead time to call the UDF.
- Time to fetch the data thats going to be used by the UDF.
- Time to do the calculations.
- Overhead time to return the answer(s).

In the post on Excel VBA Read/Write timeings you could see that there was quite a significant overhead on each VBA read and write call, so that its usually much faster to read and write large blocks of data at a time.

So it sounds like a good idea to make your VBA UDF read as much data as possible in a single block and return data to Excel in as large a block as possible.

Enter the Multi-cell array formula – it does exactly that – and also minimises the calling overhead – and often it can read the data once and re-use it lots of times.

So how do you make a Multi-Cell Array formula?

Lets create an array version of the AverageTolE function shown in the first Writing Efficient VBA UDFs post.

The scenario is that you want to find the Averages of the data excluding a number of different tolerances rather just one tolerance.

To keep things simple I am assuming that

- the tolerances are all in one row
- both the data and the tolerances will be supplied as ranges
- error-handling is largely ignored
- the function returns a row of answers that correspond to the row of tolerances.

```
1       Public Function AverageTolM(theRange As Range, theTols As Range) As Variant
2       Dim vArr As Variant
3       Dim vArrTols As Variant
4       Dim v As Variant
5       Dim d As Double
6       Dim r As Double
7       Dim k As Long
8       Dim vOut() As Variant
9       Dim dTol As Double
10      Dim lCount As Long
11      On Error GoTo FuncFail
12      vArr = theRange.Value2
13      vArrTols = theTols.Value2
14      ReDim vOut(1 To 1, 1 To UBound(vArrTols, 2))
15      On Error GoTo skip
16      For k = 1 To UBound(vArrTols, 2)
17      dTol = CDbl(vArrTols(1, k))
```

```
18        r = 0#
19        lCount = 0
20        For Each v In vArr
21        d = CDbl(v)
22        If Abs(d) &gt; dTol Then
23        r = r + d
24        lCount = lCount + 1
25        End If
26        skip:
27        Next v
28        vOut(1, k) = r / lCount
29        Next k
30        AverageTolM = vOut
31        Exit Function
32        FuncFail:
33        AverageTolM = CVErr(xlErrNA)
34        End Function
```

The changes to the UDF are quite simple:

- theTols range is coerced into a variant array: vArrTols = theTols.Value2
- an output array of the same size is created: ReDim vOut(1 To 1, 1 To UBound(vArrTols, 2))
- The UDF loops on the tolerance array and populates the output array
- The output array is assigned to the function variable: AverageTolM = vOut

Note that the Function is declared as returning a variant (which will contain an array) rather than being declared as returning an array of variants.
Assuming that the data is in H27:AA27 then enter the array function with Ctrl/Shift/Enter as {=AVERAGETOLM(Data!$A$1:$A$32000,$H$27:$AA$27)} into 20 rows (so we will get 20 x 20 = 400 cells of answers).
Calculating this 20 formulas takes 975 milliseconds.
Using the original AVERAGETOLE formula for the 400 cells takes 1660 milliseconds, an improvement factor of 1.7

Summary

- In many real-life cases using multi-cell array UDFs can be the fastest way to calculate.
- Converting a conventional UDF to a multi-cell array UDf is straightforward

(Part 6) – Faster string handling and Byte arrays

Posted on October 18, 2011by fastexcel

None of the previous posts on writing efficient VBA UDFs (Part1,Part2,Part3,Part4,Part5) talked about handling strings in VBA.
This could be a major omission since string-handling is one of VBAs slowest “features”.
Suppose you want to find the position of the first capital letter in a string.

Array Formula

You could use an array formula like this:
{=MATCH(TRUE,ISERR(FIND(MID(A5,ROW($1:$255),1), LOWER(A5))),0)}
My test data is 2000 rows, each containing 25 lower-case characters and one randomly placed upper-case character.
2000 calls to this array formula takes 250 milliseconds.
So lets try some VBA UDFs.

Using LIKE

One way is to use the VBA LIKE statement:

```
1         Function FirstCap2(Cell As Range)
2         For FirstCap2 = 1 To Len(Cell.Value)
3         If Mid(Cell.Value, FirstCap2, 1) Like "[A-Z]" Then
```

```
4           Exit For
5           End If
6           Next FirstCap2
7           End Function
```

The code loops across the string using Mid to look at each character in turn, and then uses LIKE to see if the character is one of upper-case A to upper-case Z.
2000 calls to this UDF takes 50 milliseconds – a factor of 5 faster, but we can make it faster (of course).

```
1           Function FirstCap3(Rng As Range) As Long
2           Dim theString As String
3           theString = Rng.Value2
4           For FirstCap3 = 1 To Len(theString)
5           If Mid$(theString, FirstCap3, 1) Like "[A-Z]" Then
6           Exit For
7           End If
8           Next FirstCap3
9           End Function
```

I changed the code to only get the string out of the cell once, and to use Mid$ rather than Mid. All the VBA string handling functions have 2 versions: versions without the $ work with variant arguments, whereas versions with the $ suffix only work on string arguments, but are slightly faster.
2000 calls to this version of the UDF takes 17 milliseconds, nearly 3 times faster.

Using MID$

But maybe using LIKE is slow? Lets try comparing a lower-case version of the string and stopping when the characters don't match:

```
1           Function FirstCap4(strInp As String) As Long
2           Dim tmp As String
3           Dim i As Long
4           Dim pos As Long
5           tmp = LCase$(strInp)
6           pos = -1
7           For i = 1 To Len(tmp)
8           If Mid$(tmp, i, 1) &lt;&gt; Mid$(strInp, i, 1) Then
9           pos = i
10          Exit For
11          End If
12          Next
13          FirstCap4 = pos
14          End Function
```

Well surprisingly this is **slower** than the optimised version using LIKE:
2000 calls to this version of the UDF takes 36 milliseconds.

Using Byte Arrays

Using Byte arrays with strings is one of VBAs less well known secrets, but its often an efficient way of handling strings when you need to inspect each character in turn.

```
1           Public Function FirstCap5(theRange As Range) As Long
2           Dim aByte() As Byte
3           Dim j As Long
4           FirstCap5 = -1
5           aByte = theRange.Value2
6           For j = 0 To UBound(aByte, 1) Step 2
7           If aByte(j) &lt; 91 Then
8           If aByte(j) &gt; 64 Then
9           FirstCap5 = (j + 2) / 2
10          Exit For
11          End If
12          End If
```

```
13          Next j
14          End Function
```

This version of the UDF is slightly faster: 2000 calls takes 15 milliseconds.

So how does this work?

First create an undimensioned array of Bytes : Dim aByte() as Byte
Then assign a string to it: aByte=”abEfg”
You can use the Locals window to see what the resulting Byte array looks like:

aByte		Byte(0 to 9)
aByte(0)	97	Byte
aByte(1)	0	Byte
aByte(2)	98	Byte
aByte(3)	0	Byte
aByte(4)	69	Byte
aByte(5)	0	Byte
aByte(6)	102	Byte
aByte(7)	0	Byte
aByte(8)	103	Byte
aByte(9)	0	Byte

Each character in the string has resulted in 2 bytes which are the Unicode code points for the character. Since I am working in a UK English Locale using the Windows Latin-1 codepage the first byte is the ANSI number for the character and the second byte is always zero. Unaccented english upper-case characters are ANSI numbers 65 to 90, so I can loop down the byte array, looking at every other byte, and do a numeric test directly on the character to see if it is upper-case. You can see that only the third character is upper-case.
Another surprising feature of this kind of Byte array is that you can assign a byte array directly back to a string:

```
Dim str1 as string
str1=aByte
```

Str1 now contains “abEfg”

Array version of the Byte UDF

As discussed in Part 5 of writing efficient UDFs, Array Formulae go faster. So here is an array formula version of the Byte UDF.

```
1           Public Function AFirstCap(theRange As Range) As Variant
2           Dim aByte() As Byte
3           Dim j As Long
4           Dim L As Long
5           Dim vRange As Variant
6           Dim jAnsa() As Long
7           Dim NumCells As Long
8           vRange = theRange.Value2
9           NumCells = UBound(vRange, 1)
10          ReDim jAnsa(NumCells - 1, 0)
11          For L = 0 To NumCells - 1
12          jAnsa(L, 0) = -1
13          aByte = vRange(L + 1, 1)
14          For j = 0 To UBound(aByte, 1) Step 2
15          If aByte(j) &lt; 91 Then
16          If aByte(j) &gt; 64 Then
17          jAnsa(L, 0) = (j + 2) / 2
18          Exit For
19          End If
20          End If
21          Next j
22          Next L
23          AFirstCap = jAnsa
24          End Function
25
```

This version, entered into 2000 rows as an array formula using Control/Shift/Enter, takes just 4.8 milliseconds.

Conclusion

Here is a table comparing the speed of these different approaches.

Method	Milliseconds
Array Formula	250
LIKE UDF	50
Optimised LIKE UDF	17
MID$ UDF	36
Byte Array UDF	15
Array Formula version of Byte Array UDF	4.8

So the fastest VBA is just over 10 times faster than the slowest VBA solution, and a whopping 52 times faster than the array formula solution.

Using Byte arrays for strings can be a good solution for string handling where you need to inspect or manipulate many individual characters.

So what do you use Byte arrays for?

(Part 7) – UDFs calculated multiple times

Posted on November 25, 2011by fastexcel

There are several circumstances where Excel will calculate a UDF multiple times when you would expect it to only be calculated once. This can be a significant problem if your UDF takes a long time to execute.

Multiple UDF recalcs caused by uncalculated cells

When Excel recalcs a workbook after changes have been made the calculation engine starts by calculating the most recently changed formulas, and then uses the most recent calculation sequence for the remaining formulas.

If the calculation engine finds a formula that depends on a cell that has been dirtied/changed (or is volatile) but has not yet been calculated, *it reschedules the formula to the end of the calculation chain so that it can be recalculated again after the uncalculated cell.*

The problem is that the calculation engine only does this rescheduling **after the formula/UDF has been calculated**, so a formula containing a UDF can be calculated many times in each recalculation.

Here is a very simple example:

1. Set Calculation to Manual so that its easier to see whats happening.
2. Enter this UDF into a standard Module in the VBE.

```
1        Public Function Tracer(theCell As Range)
2        Tracer = theCell.Value
3        Debug.Print Application.Caller.Address &amp;amp; "-" &amp;amp; Tracer
4        End Function
```

3. Show the Immediate Window (Ctrl G)
4. Enter 1 in Cell A1
5. Enter =Tracer(A1)+1 in cell A2
6. Enter =Tracer(A2)+1 in cell A3

The Immediate window shows

$A$2-1

$A$3-2

because the formulas were calculated as they were entered.

Now clear the immediate window, return to Excel and press F9 to recalculate. The immediate Window now shows
$A$3-
$A$2-1
$A$3-2
Which shows that cell A3 was calculated first (with the value of its parameter range A2 showing as empty), followed by A2, followed by A3 again, this time with the correct value for its parameter A2.
Now if you clear the Immediate window and calculate the formulas again without changing anything (use Ctrl/Alt/F9). This time A3 only gets recalculated once, because Excel is reusing the final calculation sequence from the previous recalculation.

Handling uncalculated cells

Fortunately its fairly easy for the UDF to detect when its being passed an uncalculated cell because the cell will be empty:

```
1       Public Function Tracer2(theCell As Range)
2       If IsEmpty(theCell) Then Exit Function
3       Tracer2 = theCell.Value
4       Debug.Print Application.Caller.Address &amp;amp; "-" &amp;amp; Tracer2
5       End Function
```

This version of the UDF checks if the cell is empty and exits immediately. If you need to distinguish between genuinely empty cells and uncalculated cells you can check that the cell contains a formula using:

```
=IsEmpty(theCell.Value) and Len(theCell.formula)>0 Then Exit Function
```

or

```
=IsEmpty(theCell.Value) and theCell.HasFormula Then Exit Function
```

If the parameter is a range of cells containing formulae then you need something a bit more complex:

```
1       Public Function IsCalced(theParameter As Variant) As Boolean
2       '
3       ' Charles Williams 9/Jan/2009
4       '
5       ' Return False if the parameter refers to as-yet uncalculated cells
6       '
7       Dim vHasFormula As Variant
8       IsCalced = True
9       On Error GoTo Fail
10      If TypeOf theParameter Is Excel.Range Then
11      vHasFormula = theParameter.HasFormula
12      '
13      ' HasFormula can be True, False or Null:
14      ' Null if the range contains a mix of Formulas and data
15      '
16
17      If IsNull(vHasFormula) Then vHasFormula = True
18      If vHasFormula Then
19      '
20      ' CountA returns 0 if any of the cells are not yet calculated
21      '
22      If Application.WorksheetFunction.CountA(theParameter) = 0 Then IsCalced =
23      False
24      End If
25      ElseIf VarType(theParameter) = vbEmpty Then
26      '
27      ' a calculated parameter is Empty if it references uncalculated cells
28      '
29      IsCalced = False
30      End If
31      Exit Function
32      Fail:
```

```
        IsCalced = False
        End Function
```

This function handles both range references and calculated ranges (array formula expressions etc) and checks if ALL the cells in the parameter contain formulas and ANY of the cells are uncalculated.

Only Variant and Range Parameters can be uncalculated.

Only a UDF parameter defined as a Range or a Variant can be uncalculated. If all your paramters are defined as, for instance, Double then Excel will attempt to coerce the parameter to Double before passing it to the UDF, and if the Paramter actually refers to an uncalculated cell then the UDF will not be called.

Multiple UDF Recalcs caused by the Function Wizard

Whenever you use the Function Wizard with a UDF the UDF gets called lots of times, because the Function Wizard uses Evaluate to dynamically show you the result of the function as you enter the parameters for the function. This is not good if your UDF is slow to execute!

You can detect that the UDF has been called by the function wizard by checking if the Standard commandbar is enabled (this works in Excel 2007 and Excel 2010 even though the commandbars are not visible).

```
If Not Application.CommandBars("Standard").Controls(1).Enabled Then Exit Function
```

Multiple UDF Recalcs with multi-cell Array Formula UDFs

Useing an array UDF that returns results to multiple cells can be a very good way of speeding up UDF execution (see Part 5 – Array UDFs go faster), but there is a nasty slowdown bug you should be aware of:

When a multi-cell UDF is entered or modified *and depends on a volatile formula*: **the UDF is evaluated once for each cell it occupies.** This does not happen when the UDF is recalculated, only when it is entered or changed.

UDFs in Conditional Formatting formulas.

Formulas in Conditional Formatting rules get evaluated each time the portion of the screen containing the conditional format gets redrawn or recalculated (you can demonstrate this by using a Debug.Print statement in a UDF being used in a conditional formatting rule). So on the whole using UDFs in Conditional formats is probably not a great idea.

Conclusion

If you have UDFs which take a long time to execute it makes sense to add code to check for both uncalculated cells and the UDF being called by the function wizard.

(Part 8) – Getting the previously calculated value from the calling cells

Posted on January 8, 2012by fastexcel

If you have a UDF that depends on some slow-calculating resource you may want the UDF to mostly just return the values obtained at the last calculation from the cells the UDF occupies, and only occasionally go and use the slow-calculating resource.

In Excel 2010 you can create efficient C++ XLL UDFs that execute asynchronously and multithreaded, which can be a very effective solution.

But how do you get the previous value from the cells calling a VBA UDF?

Lets suppose that you want to pass the UDF a parameter for the slow-calculating resource and a switch to tell it when to use the slow resource. You can set the switch (I am using a defined name called “RefreshSlow”) and refresh the UDFs in a VBA sub like this:

```
1           Sub RefreshUDFs()
2           Dim lCalcMode As Long
3           lCalcMode = Application.Calculation
4           Application.Calculation = xlCalculationManual
5           Names("RefreshSlow").RefersTo = True
6           Calculate
7           Names("RefreshSlow").RefersTo = False
```

```
8           Application.Calculation = lCalcMode
9           End Sub
```

I will use a dummy function to simulate getting a slow resource:

```
1           Function GetSlowResource(vParam As Variant) As Variant
2           Dim j As Long
3           For j = 1 To 10000000
4           Next j
5           GetSlowResource = Rnd()
6           End Function
```

This function (ignores the parameter) and just (slowly) returns a random number.
There are several ways of getting the previously calculated value for a UDF: they each have advantages and disadvantages.

Application.Caller.Value

You can use Application.Caller.Value, but this causes a circular reference that you have to switch on Iteration to solve. This is slow and can mask other unintentional circular refs, so its not recommended.

```
1           Function UDF1(vParam, Refresh)
2           If Not Refresh Then
3           UDF1 = Val(Application.Caller.Value2)
4           Else
5           UDF1 = GetSlowResource(vParam)
6           End If
7           End Function
```

Application.Caller.Text

If you use Application.Caller.Text you don't get the circular reference, but it retrieves the formatted value that is displayed in the cell as a string. So if the cell is formatted as a number with 2 decimal places the retrieved value will be truncated to 2 decimal places.

```
1           Function UDF2(vParam, Refresh)
2           If Not Refresh Then
3           UDF2 = Val(Application.Caller.Text)
4           Else
5           UDF2 = GetSlowResource(vParam)
6           End If
7           End Function
```

This solution will work OK if you can control the formatting or the function returns a string.

Application.Caller.ID

You can use the Range.ID property to store and retrieve a string value within the UDF.

```
1             Function UDF3(vParam, Refresh)
2             Dim var As Variant
3             If Not Refresh Then
4             UDF3 = Val(Application.Caller.ID)
5             Else
6             var = GetSlowResource(vParam)
7             UDF3 = var
8             Application.Caller.ID = var
9             End If
10            End Function
```

This works well, except that the Range.ID property is not stored in a Saved workbook, so the next time you open the workbook the retrieved value will be Blank/Zero.

Using an XLM or XLL function to pass the Previous value to the UDF

Using XLM or XLL technology it is possible to create a non-multi-threaded command-equivalent function to retrieve the previous value.
Here is the code for an XLL+ function called PREVIOUS which has a parameter to make it Volatile or not Volatile.

(Command-equivalent functions default to Volatile but when using it to pass the previous value to a VBA UDF you generally want it to be non-volatile).
This function also works for multi-celled array formulae.
Edited following Keith Lewis comment.

```
CXlOper* PREVIOUS_Impl(CXlOper&amp; xloResult, const CXlOper* Volatile_op)
{
// Input buffers
bool Volatile;
// Validate and translate inputs
static CScalarConvertParams Volatile__params(L"Volatile",
XLA_DEFAULT_ZERO|XLA_DEFAULT_EMPTY|XLA_DEFAULT_NONNUMERIC|
XLA_DEFAULT_BLANK, 0, -1, true);
XlReadScalar(*Volatile_op, Volatile, Volatile__params);
// End of generated code
//}}XLP_SRC
// defined as a macro function defer recalc so that the func gets
previous results
CXlOper xloCaller,xlo;
CXlOper arg;
arg=true;
if (!Volatile) arg=false;
// set volatility of this function: 237 is the function number for
volatile
xlo.Excel(237,1,&amp;arg);
// Get caller. Fail if it is not a range of cells
if ( ( xloCaller.GetCaller() != 0 ) || !xloCaller.IsRef() ) return
CXlOper::RetError(xlerrNA);
//coerce the caller ref
xloResult.Coerce(xloCaller);
return xloResult.Ret();
}
```

Then you can use this to pass the previous value to the UDF.

```
Function UDF4(vParam, Refresh, Previous)
Dim var As Variant
If Not Refresh Then
UDF4 = Previous
Else
var = GetSlowResource(vParam)
UDF4 = var
End If
End Function
```

The UDF is called from a formula like this =UDF4("AAPL",RefreshSlow,PREVIOUS(False))
This works well, but requires access to the XLL PREVIOUS function (Laurent Longre's MOREFUNC addin has a similar function).
Its supposed to be possible to write a similar function using the old XLM Macro language, but I have not tried it.

Conclusion

There are several ways of getting the previous value from the last calculation for a VBA UDF. But the best solution requires using a C++ XLL.
Special prize to the first person to write the XLM Macro Previous UDF!

Part 9 – An Example – Updated

Posted on January 31, 2012by fastexcel

Pedro wants to know how to speed up his UDF, which needs to calculate results for 35040 cells the minimum difference between the cell and a column of values of unknown length.

Pedro's UDF

```
Function MinofDiff(r1 As Long) As Variant
Dim r2 As Range
Dim TempDif As Variant
Dim TempDif1 As Variant
Dim j As Long
Dim LastRow As Long
On Error GoTo FuncFail
If r1 = 0 Then GoTo skip
With Sheets("Dados")
LastRow = .Cells(.Rows.Count, "P").End(xlUp).Row
Set r2 = .Range("P8", "P" & LastRow)
End With
TempDif1 = Application.Max(r2)
For j = 1 To LastRow - 7
If r1 >= r2(j) Then
TempDif = r1 - r2(j)
Else
TempDif = r1
End If
MinofDiff = Application.Min(TempDif, TempDif1)
TempDif1 = MinofDiff
Next j
skip:
Exit Function
FuncFail:
MinofDiff = CVErr(xlErrNA)
End Function</code>
```

There is a fundamental problem with Pedro's UDF: it is referencing a range in column P without passing it in as a parameter, so if anything changes in column P the UDF could give the wrong answer because Excel will not recalculate it. Pedro has done this so that the UDF can dynamically adjust to the number of entries in column P.

On test data with 60000 entries in column P 20 calls to the UDF take 18.5 seconds on my laptop, so 34K calls would take about 9 hours to calculate! So why is it so slow?

- Every time the function is called (35K times) it finds the last row and the MAX value in column P: but this only needs to be done once.
- 35040 calls will hit the VBE refresh slowdown bug: so we need to bypass that.
- The For loop is referencing each cell value in column P (using R2(j)) twice. Each reference to a cell is slow because there is a large overhead for each call out to the Excel object model.
- The UDF uses Worksheetfunction.Min to find out which of 2 values is smaller: its much quicker to compare the values using VBA If than invoking a worksheet function.

The revised UDF

To solve the fundamental problem with the UDF I will pass it an additional parameter: a whole column reference to column P. Then the UDF can resize the range to the last cell containing data. (Another alternative would be to create a Dynamic Named Range for column P and pass that as a parameter.

To solve the first 2 slowdown problems the UDF will be made into an array formula UDF that returns an array of 35040 results.

To avoid referencing each cell in column P twice inside the loop, the UDF will get all the values from column P once, into a variant array and then loop on the variant array.

```
Function MinofDiff2(R1 As Range, R2 As Range) As Variant
Dim R2Used As Range
Dim vArr2 As Variant
Dim vArr1 As Variant
Dim vOut() As Double
Dim TempDif As Double
Dim TempDif1 As Double
Dim D1 As Double
Dim D2 As Double
Dim TMax As Double
Dim j1 As Long
Dim j2 As Long
Dim LastRow As Long
'
On Error GoTo FuncFail
'
' handle full column
'
LastRow = R2.Cells(R2.Rows.Count, 1).End(xlUp).Row
Set R2Used = R2.Resize(LastRow - 7, 1).Offset(7, 0)
'
' get values into arrays
'
vArr2 = R2Used.Value2
vArr1 = R1.Value2
'
' find max
'
TMax = Application.Max(R2Used)
'
' set output array to same size as R1
'
ReDim vOut(1 To UBound(vArr1), 1)
'
' loop on R1
'
For j1 = 1 To UBound(vArr1)
TempDif1 = TMax
D1 = vArr1(j1, 1)
'
' loop on R2
'
For j2 = 1 To (LastRow - 7)
D2 = vArr2(j2, 1)
If D1 >= D2 Then
TempDif = D1 - D2
Else
TempDif = D1
End If
If TempDif < TempDif1 Then
vOut(j1, 1) = TempDif
Else
vOut(j1, 1) = TempDif1
End If
TempDif1 = vOut(j1, 1)
Next j2
Next j1
MinofDiff2 = vOut
skip:
Exit Function
```

```
62          FuncFail:
63          MinofDiff2 = CVErr(xlErrNA)
            End Function
```

Because this is an array function you need to select the 35040 cells that you want to contain the answer, then type the formula into the formula bar =MinofDiff2(A1:A35040,P:P) and then press Ctrl/Shift/Enter to enter the formula as an array formula into the 35040 cells. This revised UDF takes .222 seconds for 20 values, and completes the 35040 UDF calculations in 6.25 minutes, a speedup factor of over 80.

Updated with Harlan Grove's suggestions

Harlan Grove has pointed out several ways of speeding up the UDF. Here is a revised version implementing most of his suggestions. It is about 17% faster than my original version.

```
1           Function MinofDiff3(R1 As Range, R2 As Range) As Variant
2           Dim R2Used As Range
3           Dim vArr2 As Variant
4           Dim vArr1 As Variant
5           Dim vOut() As Double
6           Dim TempDif As Double
7           Dim TempDif1 As Double
8           Dim D1 As Double
9           Dim D2 As Double
10          Dim TMax As Double
11          Dim TMin As Double
12          Dim j1 As Long
13          Dim j2 As Long
14          Dim LastRow As Long
15          '
16          On Error GoTo FuncFail
17          '
18          ' handle full column
19          '
20          LastRow = R2.Cells(R2.Rows.Count, 1).End(xlUp).Row - 7
21          Set R2Used = R2.Resize(LastRow, 1).Offset(7, 0)
22          '
23          ' get values into arrays
24          '
25          vArr2 = R2Used.Value2
26          vArr1 = R1.Value2
27          '
28          ' find max & Min
29          '
30          TMax = Application.Max(R2Used)
31          TMin = Application.Min(R2Used)
32          '
33          ' set output array to same size as R1
34          '
35          ReDim vOut(1 To UBound(vArr1), 1)
36          '
37          ' loop on R1
38          '
39          For j1 = 1 To UBound(vArr1)
40          TempDif1 = TMax
41          D1 = vArr1(j1, 1)
42          TempDif = D1 - TMax
43          If D1 > TMax Then
44          If TempDif < TMax Then
45          vOut(j1, 1) = TempDif
46          Else
47          vOut(j1, 1) = TMax
48
49
```

```
50          End If
51          Else
52          If D1 < TMin Then
53          vOut(j1, 1) = D1
54          Else
55          '
56          ' loop on R2
57          '
58          For j2 = 1 To LastRow
59          D2 = vArr2(j2, 1)
60          If D1 >= D2 Then
61          TempDif = D1 - D2
62          Else
63          TempDif = D1
64          End If
65          If TempDif < TempDif1 Then TempDif1 = TempDif
66          vOut(j1, 1) = TempDif1
67          Next j2
68          End If
69          End If
70          Next j1
71          MinofDiff3 = vOut
72          skip:
73          Exit Function
74          FuncFail:
            MinofDiff3 = CVErr(xlErrNA)
            End Function
```

Harlan also points out that a version using QuickSort to sort R2 and Binary Search instead of the loop would be an order of magnitude faster!

Part 10 – Volatile Functions and Function Arguments

Posted on February 2, 2012by fastexcel

I just realised that none of my previous 9 posts on writing efficient VBA UDFs has discussed when and why you should make Functions Volatile. Since that's fairly fundamental I really should have covered the topic early in the series … but anyway here goes.

What does Volatile mean?

Normally Excel's smart recalculation engine only recalculates formulas that either have been changed/entered or depend on a cell or formula that has been changed somewhere higher up the chain of precedents for the formula.

This makes for very efficient calculation speed since in a typical workbook only a small faction of the formulas will be dependent on any particular cell or piece of data.

But some functions need to recalculate at every recalculation. For example NOW() should always give you the current time at the last calculation, and RAND() should give you a different random number each time it is calculated. These functions are called Volatile Functions, and any formula that uses one of them is a Volatile formula.

You can see more discussion of Excel's built-in volatile functions and the volatile actions that trigger a recalculation at http://www.decisionmodels.com/calcsecretsi.htm.

How does Excel's smart recalc engine know when to recalculate a function or a formula?

Excel maintains its dependency trees by looking at what other cells a function or a formula refers to, and the smart recalc engine uses these dependency trees to work out which formulas to recalculate.

For Functions Excel only looks at the arguments to the function to determine what the function depends on. So if you write a function like this:

```
Function Depends(theCell as range)
Depends=ActiveSheet.range("Z9")+theCell + _
theCell.Offset(0,1)
End Function
```

and call it in a formula =Depends("A1")
then Excel will only recalculate your function when A1 changes, and not when B1 or Z9 changes.
This could give you incorrect results.
Note: During a recalculation if Excel **does** evaluate the UDF it determines which cell references are actually being used inside the function to affect the function result, and if those cells have not yet been finally calculated it will reschedule the Function for later calculation. This is required to make the UDF be finally calculated in the correct dependency sequence.

How to fix this problem

There are several ways to fix this problem, but only one good one!

Make the function Volatile

If you add Application.Volatile to the function it will always recalculate:

```
Function Depends(theCell as range)
Application.Volatile
Depends=ActiveSheet.range("Z9")+theCell+ _
theCell.Offset(0,1)
End Function
```

But this will slow down the calculation, so generally its a bad idea unless, like RAND() or NOW() the function really needs to be Volatile.

Use Ctrl/Alt/F9 to trigger a full calculation

If you press Ctrl/Alt/F9 then Excel will recalculate every single formula in all the open workbooks, regardless of what has changed or is volatile.
Of course this can be very slow.

Make sure the Arguments to the UDF refers to ALL the cells the UDF uses.

Change the UDF to

```
Function Depends(theCell1 as range, theCell2 as range)
Depends = theCell1.Resize(1, 1)+ _
theCell1.Resize(1, 1).Offset(0, 1) + theCell2
End Function
```

This is the best solution.
Call it using **=Depends(A1:B1,Z9)** so that Excel knows that B1 is being referenced by theCell1.Offset(0,1).
Now Excel knows all the cells that the function depends on and it will be recalculated correctly and efficiently.

Detecting whether a Function or Formula is Volatile

You can download VolatileFuncs.zip from
http://www.DecisionModels.com/Downloads/VolatileFuncs.zip
This contains tests for the volatile Excel built-in functions, using a function to increment a counter each time the referenced cell changes.

```
Public jCalcSeq As Long ''' calculation sequence counter
Public Function CalcSeqCountRef(theRange As Range) As Variant
' COPYRIGHT © DECISION MODELS LIMITED 2000. All rights reserved
' increment calculation sequence counter at Full Recalc or when theRange changes
' fixed for false dependency
jCalcSeq = jCalcSeq + 1
CalcSeqCountRef = jCalcSeq + (theRange = theRange) + 1
End Function
```

Summary
Make sure that the arguments to your UDF always directly refer to ALL the cells that the UDF uses.

Part 11 – Full-Column References in UDFs: Used Range is Slow

Posted on December 2, 2012by fastexcel

Excel users often find it convenient to use full-column references in formulas to avoid having to adjust the formulas every time new data is added. So when you write a User Defined Function (UDF) you can expect that sooner or later someone will try to use it with a full-column reference:

```
=MyUDF(A:A,42)
```

When Excel 2007 introduced the "Big Grid" with just over 1 million rows it became even more important to handle these full-column references efficiently. The standard way to handle this in a VBA UDF is to get the INTERSECT of the full-column reference and the used-range so that the UDF only has to process the part of the full-column that has actually been used. The example VBA code below does this intersection and then returns the smaller of the number of rows in the input range and the number of rows in the used range.

```
1          Public Function GetUsedRows(theRng As Range)
2          Dim oRng As Range
3          Set oRng = Intersect(theRng, theRng.Parent.UsedRange)
4          GetUsedRows = oRng.Rows.Count
5          End Function
```

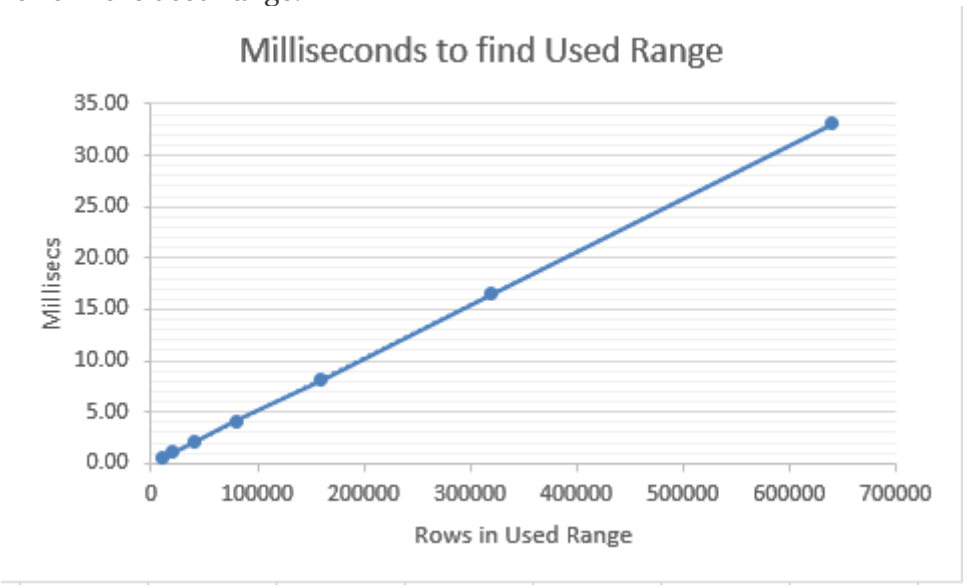
The parent of theRng is the worksheet that contains it, so `theRng.Parent.UsedRange` gets the used range of the worksheet you want.
Two problems with this technique are:

- Getting the Used Range can be slow.
- The XLL interface does not have a direct way to access the Used Range, so you have to get it via a single-thread-locked COM call. (More on this later).

So just how slow is it to get the used Range?
I created a very simple UDF and timed the calculation of 1000 calls to this UDF for filled used ranges of between 10K rows and 640K rows.

```
1          Public Function CountUsedRows()
2          CountUsedRows = ActiveSheet.UsedRange.Rows.Count
3          End Function
```

It turns out that the time taken to execute this UDF is a linear function of the number of used rows in the used range.



And its quite slow, 1000 calls to this UDF with 640K rows of data takes 33 seconds!

When the used range is small you won't notice the time taken, but for large used ranges with the big grid you certainly will. And the problem is that your UDF will do this check on every range that is passed to the UDF, even if its not really needed.

Colin points out that what affects the time is actually the number of cells containing data or formatting (or that previously contained data or formatting) rather than the last cell in the used range.

Speeding up finding the used range.

So you could start by only doing the used-range check when theRng parameter has a large number of rows:

```
1         Public Function GetUsedRows2(theRng As Range)
2         Dim oRng As Range
3         If theRng.Rows.Count > 500000 Then
4         Set oRng = Intersect(theRng, theRng.Parent.UsedRange)
5         GetUsedRows = oRng.Rows.Count
6         Else
7         GetUsedRows = theRng.Rows.Count
8         End If
9         End Function
```

This example only does the check if the user gives the UDF a range referring to more than half a million rows.

Another, more complicated, way of minimising the time is to store the number of rows in the used range in a cache somewhere and retrieve it from the cache when needed. The tricky part of this is to make sure that the used-range row cache always is either empty (in which case go and get the number) or contains an up-to-date number.

One way of doing this would be to use the Application AfterCalculate event (which was introduced in Excel 2007) to empty the cache. Then only the first UDF that requested the used range for each worksheet would use time to find the used range, and (assuming that the calculation itself did nothing to alter the used range) the correct number would always be retrieved.

The equivalent for Excel versions before Excel 2007 would be to use the Application SheetCalculate event to empty the cache for that particular worksheet. This technique would be less efficient since a worsheet may well be calculated several times in each calculation cycle.

As Colin points out, if you want to find the last row containing data it is faster to use Range.Find when you have many cells containing data.

Note that you can only use Range.Find in UDFS in Excel 2002 and later, and you cannot use the Find method at all from an XLL except in a command macro or via COM.

```
1         Public Function CountUsedRows2()
2         CountUsedRows2 = ActiveSheet.Cells.Find(What:="*", LookIn:=xlFormulas,
3         SearchOrder:=xlByRows, SearchDirection:=xlPrevious).Row
          End Function
```

So have you got any better ideas on how to process full-column references efficiently?

Part 12: Getting Used Range Fast using Application Events and a Cache

Posted on December 5, 2012by fastexcel

In the previous post I suggested that one good way to speed up retrieval of the Used Range last row would be to use a Cache and the AfterCalculate Application event.
I have now tested this approach and it works well: here is the code for the demo function GetUsedRows3:

```
Option Explicit
'
' create module level array for cache
'
Dim UsedRows(1 To 1000, 1 To 2) As Variant
Public Function GetUsedRows3(theRng As Range)
' store & retrieve used range rows if Excel 2007 & later
Dim strBookSheet As String
Dim j As Long
Dim nFilled As Long
Dim nRows As Long
' create label for this workbook & sheet
strBookSheet = Application.Caller.Parent.Parent.Name & "_" &
Application.Caller.Parent.Name
If Val(Application.Version) >= 12 Then
' look in cache
For j = LBound(UsedRows) To UBound(UsedRows)
If Len(UsedRows(j, 1)) > 0 Then
nFilled = nFilled + 1
If UsedRows(j, 1) = strBookSheet Then
' found
GetUsedRows3 = UsedRows(j, 2)
Exit Function
End If
Else
' exit loop at first empty row
Exit For
End If
Next j
End If
' find used rows
nRows = theRng.Parent.UsedRange.Rows.Count
'
If Val(Application.Version) >= 12 Then
' store in cache
nFilled = nFilled + 1
If nFilled <= UBound(UsedRows) Then
UsedRows(nFilled, 1) = strBookSheet
UsedRows(nFilled, 2) = nRows
End If
End If
'
GetUsedRows3 = nRows
End Function
Sub ClearCache()
'
' empty the first row of the used-range cache
'
UsedRows(1, 1) = ""
End Sub
```

Note: there is no error handling in this code!
Start by defining a module level array (UsedRows) with 1000 rows and 2 columns. Each row will hold a key in column 1 (book name and sheet name) and the number of rows in the used range for that sheet in that book in column 2. I have assumed that we will only cache the first 1000 worksheets containing these UDFs!
The key or label is created by concatenating the name of the parent of the calling cell (which is the worksheet) to the name of the parent of the parent of the calling cell (which is the workbook containing the sheet).
Then loop down the UsedRows array looking for the key, but exit the loop at the first empty row.
If the key is found, retrieve the number of rows in the used range from column 2, return it as the result of the function and exit the function.
Otherwise find the number of rows in the used range, store it in the next row of the UsedRange cache and return it as the result of the function.
Only for Excel 2007 or later
You can see that the function only operates the cache for Excel 2007 and later versions. There are two reasons for this:

- Excel 2003 and earlier have a maximum of 64K rows so finding the used range is relatively fast anyway.
- Only Excel 2007 and later have the AfterCalculate event which will be used to empty the cache after each calculate.

We need to empty the cache after each calculate because the user might alter the used range and so the safe thing to do is to recreate the cache at each calculation.
AfterCalculate is an Application Level event which is triggered after completion of a a calculation and associated queries and refreshes. (A BeforeCalculate event would be even more useful but does not exist!)
Using the AfterCalculate Application Event.
Chip Pearson has an excellent page on Application Events. I always consult it when I need application events because I can never remember exactly how to do it!
First I added a Class Module called AppEvents with code like this:

```
Option Explicit
Private WithEvents App As Application
Private Sub Class_Initialize()
Set App = Application
End Sub
Private Sub App_AfterCalculate()
ClearCache
End Sub
```

Then I added some code to the ThisWorkbook module:

```
Option Explicit
Private XLAppEvents As AppEvents
Private Sub Workbook_Open()
Set XLAppEvents = New AppEvents
End Sub
```

This sets up the hooks that are needed for Application level events. Quite a lot of code just to run the ClearCache sub after each calculation!
ClearCache just empties the first key in the Cache so that the find loop in GetUsedRows3 exits straight away.
This code is ignored in Excel 2003 and earlier: since the AfterCalculate event does not exist it never gets called but still compiles OK.
Performance of GetUsedRows3
For 640K rows of data 1000 calls to GetUsedRows3 takes 66 milliseconds. The original CountUsedRows function took 33 seconds.
Thats a speedup factor of 500!