\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

SUMPRODUCT

A very powerful tool for selective summation of tabular data

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

Multiple Condition Tests

One of the most basic functions in any spreadsheet is to return an answer based upon some condition. This becomes especially useful when counting or summing based upon that condition. One condition is useful, and is easily achieved using COUNTIF or SUMIF. These are incredibly useful and flexible functions, but limited as they are to single conditions, they can be lacking.

Multiple conditions, such as counting the number of items sold by part number AND by month, greatly extends the functionality of our solution. There are a number of ways that this can be achieved within Excel, but this paper is focusing on one particular function, the SUMPRODUCT function, which by creative use has evolved to a flexibility undreamt of by its originators in Microsoft. Because this usage has been driven outside of Microsoft, by real-world Excel users, you will not see it documented within Excel help, or in MSDN.

SUMPRODUCT is one of the most versatile functions provided in Excel. In its most basic form, SUMPRODUCT multiplies corresponding members in given arrays, and returns the sum of those products. This page discusses the classic use of SUMPRODUCT, how creativity and inbuilt flexibility has enabled it to evolve into a far more useful function, and explains some of the techniques being deployed.

## Standard Use of SUMPRODUCT

In it's classic form, SUMPRODUCT multiplies each value in one array by the corresponding value in another array, and returns the summed result. As an example, if cells A9:A11 contain the values 1,2,3 and B9:B11 contain 10,20,30, then  
  
=SUMPRODUCT(A9:A11,B9:B11)  
  
returns 140, or (1\*10)+(2\*20)+(3\*30)=10+40+90=140.

This is a useful function, but nothing more than that. A further, more 'creative' use of SUMPRODUCT has evolved, and is still evolving, driven as far as I can see mainly by the regular contributors of the Microsoft Excel newsgroups. This has been a creative and productive process that has significantly increased the useability of SUMPRODUCT, but in a way that you will not find documented in Excel's Help.

## 

## Evolving Use of SUMPRODUCT

Within Excel, there are two very useful functions that support conditional counting and summing, namely COUNTIF and SUMIF. Very useful functions, but limited in that they can only evaluate a single test range.

In certain instances, a very simple double conditional test between two values can be emulated by testing for the lower condition and subtracting anything that is beyond the upper condition. For instance, the formula =COUNTIF(A1:A10,>=10) COUNTIF(A1:A10,>20) calculates how many items in A1:A10 that fall between 10 and 20, by counting all items greater than 10, which also includes those items greater than 20, and then subtracting the count of those items in A1:A10 that are greater than 20. Whilst this emulates a double conditional test, it is very limited, it cannot work on different ranges, or more conditions.

Multiple conditions are so useful to test ranges (say between two dates), and double tests (one array = A and another = B), and whilst this can be managed using array functions  
=SUM(IF(test\_A,IF(test\_B, etc.,  
this is somewhat unwieldy, and is an array formula. And there is a better way, using SUMPRODUCT.

Note that in this section, all formulae given are using the **'\*'** (multiply) operator format, but this in itself is one of the biggest discussion points around the SUMPRODUCT function, one which is discussed [below](http://www.xldynamic.com/source/xld.SUMPRODUCT.html#format).

To understand how SUMPRODUCT can be used, first consider the following data.

|  |  |  |  |
| --- | --- | --- | --- |
|  | A | B | C |
| 1 | Make | Month | Price |
| 2 | Ford | June | 7,500 |
| 3 | Ford | June | 8,300 |
| 4 | Ford | May | 6,873 |
| 5 | Ford | June | 11,200 |
| 6 | Renault | June | 13,200 |
| 7 | Renault | June | 14,999 |
| 8 | BMW | June | 17,500 |
| 9 | BMW | May | 23,500 |
| 10 | BMW | June | 18,000 |
| **Table 1.** | | | |

We can easily count the number of Fords with  
=COUNTIF(A1:A10,"Ford")  
which returns 4.

Similalrly, it is straight-forward to get the value of Fords sold, using  
=SUMIF(A1:A10,"Ford",C1:C10)  
which gives 33,873.

But supposing that we want a count of how many Fords are sold in June, or the value of them? The number can be calculated with

=SUM(IF(A1:A10="Ford",IF(B1:B10="June",1,0),0))

which is an array formula so is committed with Ctrl-Shift-Enter, not just Enter. Similarly, the value is obtained with

=SUM(IF(A1:A10="Ford",IF(B1:B10="June",C1:C10,0),0))

which is also an array formula.

But as this page is about SUMPRODUCT, you would expect that we could use that function in this case, and we can. The solution for the number of Fords sold in June using this function is

=SUMPRODUCT((A1:A10="Ford")\*(B1:B10="June")).

The value is obtained with

=SUMPRODUCT((A1:A10="Ford")\*(B1:B10="June")\*(C1:C10))

The \* is being used as the AND operator, the formula is saying, where A1:A10 = Ford AND B1:B10 = June, and where A1:A10 = Ford AND B1:B10= June, multiplied by C1:C10.

In my view, this formula more readily shows what the author's objective is, and of course, as it is not an array formula it is simply committed with Enter.

We can see that the \* is equivalent to AND in the formula, how this works is explained later, but supposing we want an OR condition. As a further extension of its use, we use the '+' (plus) operator to count OR conditions, such as how many cars sold were either Fords or Renaults. The formula for this is

=SUMPRODUCT((A1:A10="Ford")+(A1:A10="Renault"))

which returns the result 6 as expected[[1]](http://www.xldynamic.com/source/xld.SUMPRODUCT.html#note1).

So far, so good, in that we have a versatile function that can do any number of conditional tests, and has an inbuilt flexibility that provides extensibility. Its power is augmented when combined with other functions, such as can be found in the [examples](http://www.xldynamic.com/source/xld.SUMPRODUCT%20Part%202.html) page[[2]](http://www.xldynamic.com/source/xld.SUMPRODUCT.html#note2).

## Advantages of SUMPRODUCT

Multiple conditional tests are a major advantage of the SUMPRODUCT function as descibed above, but it has two other considerable advantages. The first is that it can function with closed workbooks, and the second is that the handling of text values can be tailored to the requirement.

In the case of another workbook, the SUMIF function can be used to calculate a value, such as in   
=SUMIF('[Nowfal Rates.xls]RATES'!$K$11:$K$13,"gt;1")

This is fine in itself, and the value remains if the other workbook is closed, but as soon as the sheet is re-calculated, the formula returns **#VALUE**. Similarly, if the formula is entered with the other workbook already closed, a **#VALUE** is immediately returned.

SUMPRODUCT, however, overcomes this problem. The formula  
=SUMPRODUCT(--('[Nowfal Rates.xls]RATES'!$K$11:$K$13>1),--('[Nowfal Rates.xls]RATES'!$K$11:$K$13))  
returns the same result, but it will still work when the other workbook is closed and the sheet is re-calculated, and can be initially entered referencing the closed workbook, without a **#VALUE** error.

The second major advantage is being able to handle text in numeric columns differently. Consider the follwoing dataset, as shown in **Table 2.**

|  |  |  |
| --- | --- | --- |
|  | A | B |
| 1 | Item | Number |
| 2 | X | 1 |
| 3 | Y | 2 |
| 4 | X | 3 |
| **Table 2.** | | | |

If we are looking at rows 1:4. we can see that we have a text value in B1 In this case it is simply a heading row, but the principle applies to a text value in any row.

Using SUMPRODUCT, we can either return an error, or ignore the text. This can be useful if we want to ignore errors, or if we want to trap the error (and presumably correct it later).

Errors will be returned if we use this version  
=SUMPRODUCT((A1:A4="x")\*(B1:B4))

To ignore errors, use this amended version which uses the double unary operator (see [SUMPRODUCT Explained](http://www.xldynamic.com/source/xld.SUMPRODUCT.html#explain) below for details)  
=SUMPRODUCT(--(A1:A4="x"),(B1:B4))

And a third, most significant advantage, is that the conditional test range or the condition can be constructed in a huge number of ways to facilitate the requirement, such as   
LEFT(A1:A10),   
ISNUMBER(MATCH(A1:A10,{"apples","pears"},0),or  
ISNUMBER(MATCH(K2:K30,ROW(INDIRECT(TODAY()&":"&TODAY()+10)),0))

But how does it work?

## SUMPRODUCT Explained

Understanding how SUMPRODUCT works helps to determine where to use it, how to can construct thus formula, and thus how it can be extended.

**Table 3.** below shows an example data set that we will use.

|  |  |  |  |
| --- | --- | --- | --- |
|  | A | B | C |
| 9 | Ford | B | 3 |
| 10 | Vauxhall | C | 4 |
| 11 | Ford | A | 2 |
| 12 | Ford | A | 1 |
| 13 | Ford | D | 4 |
| 14 | Ford | A | 3 |
| `5 | Ford | A | 2 |
| 16 | Renault | A | 8 |
| 17 | Ford | A | 6 |
| 18 | Ford | A | 8 |
| 19 | Ford | A | 7 |
| 20 | Ford | A | 6 |
| **Table 3.** | | | |

In this example, the problem is to find how many Fords with a category of "A" were sold. A9:A20 holds the make, B9:B20 has the category, and C9:C20 has the number sold. The formula to get this result is  
=SUMPRODUCT((A9:A20="Ford")\*(B9:B20="A")\*(C9:C20)).

The first part of the formula (A9:A20="Ford") checks the array of makes for a value of **Ford**. This returns an array of TRUE/FALSE, in this case it is  
**{TRUE,FALSE,TRUE,TRUE,TRUE,TRUE,TRUE,FALSE,TRUE,TRUE,TRUE,TRUE}**

Similarly, the categories are checked for the vale **A** with (B9:B20="A"). Again, this returns an array of TRUE/FALSE, or   
**{FALSE,FALSE,TRUE,TRUE,FALSE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE,TRUE}**

And finally, the numbers are not checked but taken as is, that is (C9:C20), which returns an array of numbers  
**{3,4,2,1,4,3,2,8,6,8,7,6}**

So now we have three arrays, two of TRUE/FALSE values, one of numbers. This is showm in **Table 4.**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A |  | B |  | C |
| 9 | TRUE | \* | FALSE | \* | 3 |
| 10 | FALSE | \* | FALSE | \* | 4 |
| 11 | TRUE | \* | TRUE | \* | 2 |
| 12 | TRUE | \* | TRUE | \* | 1 |
| 13 | TRUE | \* | FALSE | \* | 4 |
| 14 | TRUE | \* | TRUE | \* | 3 |
| 15 | TRUE | \* | TRUE | \* | 2 |
| 16 | FALSE | \* | TRUE | \* | 8 |
| 17 | TRUE | \* | TRUE | \* | 6 |
| 18 | TRUE | \* | TRUE | \* | 8 |
| 19 | TRUE | \* | TRUE | \* | 7 |
| 20 | TRUE | \* | TRUE | \* | 6 |
| **Table 4.** | | | | | |

And this is where it gets interesting.

SUMPRODUCT usually works on arrays of numbers, but we have arrays of TRUE/FALSE values as well as an array of numbers. By using the **'\*'** (multiply) operator, we can get numeric values that can be summed. **'\*'** has the effect of coercing these two arrays into a single array of 1/0 values. Multiplying TRUE by TRUE returns 1 (try it, enter =TRUE\*TRUE in a cell and see the result), any other combination returns 0. Therefore, when both conditions are satisfied, we get a 1, whereas if any or both conditions are not satisfied, we get a 0. Multiplying the first array of TRUE/FALSE values by the second array of TRUE/FALSE values returns a composite array of 1/0 values, or   
**{0,0,1,1,0,1,1,0,1,1,1,1}**.

This subsequent array of 1/0 values is then multiplied by the array of numbers sold to give a further array, an array of numbers sold that satisfy the two test conditions. SUMPRODUCT then sums the members of this array to give the count.

**Table 4.** shows the values that the conditional tests break down to before being acted upon by the **'\*'** operator.

**Table 5.** shows a virtual representation of those TRUE/FALSE values as their numerical equivalents of 1/0 and the individual multiplication results. From this, you should be able to see how SUMPRODUCT arrives at its result, namely 35.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | A |  | B |  | C |  |
| 9 | 1 | \* | 0 | \* | 3 | 0 |
| 10 | 0 | \* | 0 | \* | 4 | 0 |
| 11 | 1 | \* | 1 | \* | 2 | 2 |
| 12 | 1 | \* | 1 | \* | 1 | 1 |
| 13 | 1 | \* | 0 | \* | 4 | 0 |
| 14 | 1 | \* | 1 | \* | 3 | 3 |
| 15 | 1 | \* | 1 | \* | 2 | 2 |
| 16 | 0 | \* | 1 | \* | 8 | 0 |
| 17 | 1 | \* | 1 | \* | 6 | 6 |
| 18 | 1 | \* | 1 | \* | 8 | 8 |
| 19 | 1 | \* | 1 | \* | 7 | 7 |
| 20 | 1 | \* | 1 | \* | 6 | 6 |
|  |  |  |  |  |  | **35** |
| **Table 5.** | | | | | | |

**Table 6.** shows you the same virtual representation of 1/0 numerical values without the numbers sold column, that is using SUMPRODUCT to count the number of rows satisfying the two conditions, or   
=SUMPRODUCT((A9:A20=A1)\*(B9:B20="A"))

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A |  | B |  |  |
| 9 | 1 | \* | 0 | = | 0 |
| 10 | 0 | \* | 0 | = | 0 |
| 11 | 1 | \* | 1 | = | 1 |
| 12 | 1 | \* | 1 | = | 1 |
| 13 | 1 | \* | 0 | = | 0 |
| 14 | 1 | \* | 1 | = | 1 |
| 15 | 1 | \* | 1 | = | 1 |
| 16 | 0 | \* | 1 | = | 0 |
| 17 | 1 | \* | 1 | = | 1 |
| 18 | 1 | \* | 1 | = | 1 |
| 19 | 1 | \* | 1 | = | 1 |
| 20 | 1 | \* | 1 | = | 1 |
|  |  |  |  |  | **8** |
| **Table 6.** | | | | | |

If you have been able to follow this explanation all of the way through, it may have occurred to you that although we are using the SUMPRODUCT function, the **'\*'** operators have resolved the multiple arrays into a single composite array, leaving SUMPRODUCT to simply sum the members of that composite array, that is, there is no product. This is perfectly correct, and perfectly valid, SUMPRODUCT can work on a single array (put 1,2,3 in cells A1,A2,A3, and insert =SUMPRODUCT(A1:A3) in a cell, it returns 6 correctly). In reality, we only need the **'\*'** to coerce the arrays that are being tested for a particular condition, we do not need it for the array that is not subject to a conditional test. So we could also use  
=SUMPRODUCT((A9:A20="Ford")\*(B9:B20="A"),(C9:C20)),   
which does use the product aspect (see more on this in the [next section](http://www.xldynamic.com/source/xld.SUMPRODUCT.html#format)).

When using the SUMPRODUCT function, all arrays must be the same size, as corresponding members of each array are multiplied by each other.

When using the SUMPRODUCT function, no array can be a whole column (A:A), the array must be for a range within a column (although the best part of a column could be defined with A1:A65535 if so desired). Whole rows (1:1) are acceptable[[3]](http://www.xldynamic.com/source/xld.SUMPRODUCT.html#note3).

In a SUMPRODUCT function, the arrays being evaluated cannot be a mix of column and row ranges, they must all be columns, or all rows. However, the row data can be transposed to present it to SUMPRODUCT as columnar - see the [Using TRANSPOSE to test against values in a column not row](http://www.xldynamic.com/source/xld.SUMPRODUCT%20Part%202.html#transpose) example.

## Format of SUMPRODUCT

In the examples presented so far, the format has been  
=SUMPRODUCT((array1=condition1)\*(array2=condition2)\*(array3))

As mentioned above, we could also use  
=SUMPRODUCT((array1=condition1)\*(array2=condition2),(array3))  
which works as the **'\*'** operator is only required to coerce the conditional arrays that resolve to TRUE/FALSE into numeric values.

As it the use of a arithmetic operator that coreces the TRUE/FALSE values to 1/0, we could use many different operators and achieve the same result. Thus, it is also possible to coerce each of the conditional arrays individually by multiplying them by 1,   
=SUMPRODUCT((array1=condition1)\*1,(array2=condition2)\*1,(array3))  
or  
=SUMPRODUCT(1\*(array1=condition1),1\*(array2=condition2),(array3))

or by raising to the power of 1,   
=SUMPRODUCT((array1=condition1)^1,(array2=condition2)^1,(array3))

or by adding 0,   
=SUMPRODUCT((array1=condition1)+0,(array2=condition2)+0,(array3))  
or  
=SUMPRODUCT(0+(array1=condition1),0+(array2=condition2),(array3))

or even by using the N function,   
=SUMPRODUCT(N(array1=condition1),N(array2=condition2),(array3))

These methods differ from the **'\*'** operator in that they are applied to individual arrays, **'\*'** operates on two arrays.

All of these methods work, when there is more than one conditional array, so it is really a matter of preference as to which to use. If there is a single conditional array, then the **'\*'** operator cannot be used (there are not two to multiply), so one of the other above methods has to be used.

Yet another method is to use the double unary operator, **--**, in this way   
=SUMPRODUCT(--(array1=condition1),--(array2=condition2),(array3))  
The double unary operator also coerces the indivual array(s), which then acts more akin to classic SUMPRODUCT.

There has been much discussion that one way is faster than another, or is more of a 'standard' than another, but in reality there will be few instances where one method will gain a noticeable performance advantage over another, and as for standards, this is all new territory, and will mainly be used by people who have never been involved in using these standards, and who care even less.

For me, I believe it is a matter of preference. Personally, I am being swayed to the double unary **--** notation, because it avoids a function call, it works in all situations (the **'\*'** operator won't work on a single array), and I don't like the **'1\*'**, **'\*1'**, **'^1'**, or **'+0'** variations. So my preference is for   
=SUMPRODUCT(--(array1=condition1),--(array2=condition2),(array3))  
which also has more similarity to classic SUMPRODUCT,

There is one other varitaion which has been promoted recently, which is the single unary operator, **'-'**, such as   
=SUMPRODUCT(-(array1=condition1),-(array2=condition2),(array3))  
but I would not encourage this as it has no real merit that I can see, and has to be paired off, otherwise it will return a negative result.

**So, to sum up ...**  
Tests, like A=10 normally resolve to TRUE or FALSE, and any operator is only needed if you want to coerce an array of TRUE/FALSE values to 1/0 integers, such as  
=SUMPRODUCT(--(B5:B1953=101))

SUMPRODUCT arrays are normally separated by the comma. So, to preserve this format, if you have multiple conditions, you can use the -- on both conditions like so  
=SUMPRODUCT(--(B5:B1953=101),--(C5:C1953=7))

But, if you simply multiply two arrays of TRUE/FALSE, that implicitly resolves to 1/0 values that are then summed, you don;t need comma, so you could then use  
=SUMPRODUCT((B5:B193=101)\*(C5:C193=7))

Any further, final, array of values can use the same operator, or could revert to comma. So your formula can be written as  
=SUMPRODUCT(--(B5:B1953=101),--(C5:C1953=7),(D5:D1953))  
or  
=SUMPRODUCT((B5:B1953=101)\*(C5:C1953=7),(D5:D1953))  
or  
=SUMPRODUCT(--(B5:B1953=101),--(C5:C1953=7),--(D5:D1953))  
or  
=SUMPRODUCT((B5:B1953=101)\*(C5:C1953=7)\*(D5:D1953))  
or  
=SUMPRODUCT(--(B5:B1953=101),--(C5:C1953=7)\*(D5:D1953))

If the result is the product of two conditions being multiplied, it is fine to multiply them together as this will coerce the True/False values to 1/0 values to allow the summing  
=SUMPRODUCT((condition1)\*(condition2))

However, if there is only one condition, you can coerce to 1/0 with the double unary --  
=SUMPRODUCT(--(condition1))

You could achieve this equally as well with  
=SUMPRODUCT((1\*(condition1)))

and equally the first could be represented as  
=SUMPRODUCT(--(condition1),--(condition2))

There is no situation that I know of whereby a solution using -- could not be achieved somehow with a **'\*'**. Conversely, if using the TRANSPOSE function within SUMPRODUCT, then the **'\*'** has to be used.

So, as you can see there are a number of possibilities, and you make your own choice. I leave the final word to Harlan Grove, who once wrote this paragraph on why he prefers the double unary operator ...

*....as I've written before, it's not the speed of double unary minuses I like, it's the fact that due to Excel's operator precedence it's harder to screw up double unary minuses with typos than it is to screw up the alternatives ^1, \*1, +0. Also, since I read left to right, I prefer my number type coercions on the left rather than the right of my Boolean expressions, and -- looks nicer than 1\* or 0+. Wrapping Boolean expressions inside N() is another alternative, possibly clearer, but it eats a nested function call level, so I don't use it.*

## Conditional Counting and Summing in VBA

All of the discussion so far has been about conditional formulae, that is directly within Excel worksheets. It is often necessary to count or sum conditionally some worksheet ranges within a VBA routine. In these instances, we could code a simple loop to go through all of the data and check if it matches the condition, summing the matching items as we go.

Excel VBA has a method that allows a call out from VBA routines to a built-in worksheet function, saving ourselves having to build that functionality, and greatly improving the power of our VBA code. Whilst there is an overhead to calling an Excel function from within VBA, any performance impact should be minimal if not over-used, and the usefulness of this facility is clear. We can utilise this facility to achieve conditional counting and summing in VBA with little effort, but there are a few things to be aware of.

As an example, consider the data in Table 1. above. If we needed to know how many Fords were in the range A1:A10from within a VBA procedure, we could simply use the following code

Dim mModel As String

Dim mCount As Long

mModel = "Ford"

mCount = Application.WorksheetFunction.Countif( \_

Range("A1:A10"), mModel)

This will load the mCount variable with the number of Fords, 4 in this instance.

Similalry, we can use SUMIF to calculate the value

Dim mModel As String

Dim mCount As Long

mModel = "Ford"

mValue = Application.WorksheetFunction.SumIf( \_

Range("A1:A10"), mModel, Range("C1:C10"))

This will load the mCount variable with the value of the Fords, 33873 in this instance. The natural next step is to assume that we can extend this technique to our multiple condition test formulae discussed above. If we are using COUNTIFS and SUMIFS in Excel 2007 (see [SUMPRODUCT and Excel 2007](http://www.xldynamic.com/source/xld.SUMPRODUCT.html#2007)) then this is correct. For example, we can count how many Fords were sold in June using

Dim mModel As String

Dim mMonth As String

Dim mCount As Long

mModel = "Ford"

mMonth = "June"

mCount = Application.WorksheetFunction.CountIfs( \_

Range("A1:A10"), mModel, \_

Range("B1:B10"), mMonth)

We get a result of 3 here in our mCount variable. Unfortunately, this technique cannot be extended to array formulae, or conditional testing SUMPRODUCT formulae.

For example, a simple formula to count how many Fords were sold in Feb might be

=SUMPRODUCT((A2:A10="Ford")\*(B2:B10="Feb"))

(none, as it happens), and you might think that we could use the following VBA to get the same result

Dim mModel As String

Dim mMonth As String

Dim mCount As Long

mModel = "Ford"

mMonth = "Feb"

mCount = Application.WorksheetFunction.Sumproduct( \_

Range("A1:A10") = mModel , Range("C1:C10") = mMonth))

But this fails to compile, never mind getting the correct result. In this case, VBA is trying to make a simple call to the worksheet function, but when array and these type of SUMPRODUCT formulae are resolved in Excel each item is within the array is resolved and then passed to the main function for SUMming, AVERAGEing, or whatever is being actioned. As VBA doesnt evaluate the ranges, it is not passing correct information to the worksheet function, so we get the error[[4]](http://www.xldynamic.com/source/xld.SUMPRODUCT.html#note1).

There is a solution to this problem, and that is to evaluate the function call within VBA, using the VBA Evaluate method, which converts a Microsoft Excel name to an a value. The code here is

Dim mModel As String

Dim mMonth As String

Dim mFormula As String

Dim mCount As Long

mModel = "Ford"

mMonth = "Feb"

mFormula = "SUMPRODUCT((A1:A10=""" & mModel & \_

""")\*(B1:B10=""" & mMonth & """))"

mCount = Application.Evaluate(mFormula)

Although there is more effort required to ensure that the syntax of the function call is properly constructed, and that strings tested against are properly formed with quotes around them[[5]](http://www.xldynamic.com/source/xld.SUMPRODUCT.html#note5), it is still a useful technique to have, and provides the capability to use SUMPRODUCT (and by association, array formulae) within VBA.

## SUMPRODUCT and Excel 2007

When Microsoft introduced Excel 2007, the main focus was on ease of use, and improved business analysis functionality. Unfortunately, the worksheet functions did not get much attention, but there were a few new functions. Two of the new functions, COUNTIFS and

SUMIFS, support multiple conditional tests. For instance, in our previous examples ,

=SUMPRODUCT((A1:A10="Ford")\*(B1:B10="June"))

=SUMPRODUCT((A1:A10="Ford")\*(B1:B10="June")\*(C1:C10))

where we count those items where A1:A10 is = Ford AND B1:B10 = June, and where A1:A10 = Ford AND B1: B10 = June multiplied by C1:C10. In Excel 2007, COUNTIFS and SUMIFS can be used in place of SUMPRODUCT. The Excel 2007 formulae would be

=COUNTIFS(A1:A10,"Ford",B1:B10,"June")

=SUMIFS(C1:C10,A1:A10,"Ford",B1:B10,"June")

A further improvement is that in Excel 2007, SUMPRODUCT can address a whole column, which is a helpful change.

So, with Excel 2007 supporting multiple conditional tests, does this mean that the special use of SUMPRODUCT is now redundant, and that it is relegated to its original, simple array multiplication role?

Whilst this may seem to be the case at first sight, a little thought shows that SUMPRODUCT retains its unique position in the Excel developers toolkit. Why? Because COUNTIFS and SUMIFS are still unable to calculate values in closed workbooks just as their predecessors could not; and the Excel 2007 functions are still not able to accommodate the complex extra functions that can be added to the conditional ranges in SUMPRODUCT.

## Performance Considerations

### Double Unary v \* Operator

In most circumstances, either the '\*' or -- versions of SUMPRODUCT can be used, and both will function correctly. There are some exceptions to this. Consider a table of names and amounts in A1:B10, where row 1 is a text heading of 'Name' and 'Amount'. The formula

=SUMPRODUCT(--(A1:A10="Bob"),--(B1:B10>0),B1:B10)

will correctly sum the positive values in column B where the value in column is Bob. However, this formula

=SUMPRODUCT((A1:A10="Bob")\*(B1:B10>0)\*(B1:B10))

returns a #VALUE! Error. The reason for the error is the text in B1, multiplying a text value is causing the error. To overcome it with the second formula, the ranges need to start beyond the heading, in A2 and B2[[6]](http://www.xldynamic.com/source/xld.SUMPRODUCT.html#note6).

### Using Transpose

If using the TRANSPOSE function within SUMPRODUCT, then the '\*' has to be used.

### Formula Efficiency

Most people will be familiar with the fact that array formulas can be very expensive, and if over-used can significantly impair the recalculation of a worksheet/workbook.

Whilst SUMPRODUCT is not an array formula per se, it suffers from the same problem. Although SUMPRODUCT is often faster than an equivalent array formula, it is marginal. And like array formula, SUMPRODUCT is much slower than COUNTIF/SUMIF,thus it is better to use these if appropriate.

So, never use SUMPRODUCT in this situation

=SUMPRODUCT((A1:A10="Ford")\*(C1:C10))

Use the equivalent SUMIF

=SUMIF(A1:A10,"Ford",C1:C10)

Even two COUNTIF /SUMIF functions are quicker than one SUMPRODUCT, so this

=COUNTIF(A1:A10,>=10) COUNTIF(A1:A10,>20)

will be more efficient than this,

=SUMPRODUCT((A1:A10>=10)\*(A1:A10<=20))

by a factor of roughly 20%.

## Notes

[1] We can also use =SUMPRODUCT(--(A1:A10={"Ford","Renault"})) in this instance as we have a single range being tested for two (Or more) values, the -- is to coerce the Booleans to numbers that can be counted - see later.

[2] Although array formulae are mentioned here, they are not explained. For a detailed discussion, see Chip Pearson's [Array Formulas](http://www.cpearson.com/excel/array.htm) web page.

[3] Excel 2007 has now broken this constraint, SUMPRODUCT can now use whole columns, as can any array formula - see [SUMPRODUCT and Excel 2007](http://www.xldynamic.com/source/xld.SUMPRODUCT.html#2007)

[4] Note that the simple form of SUMPRODUCT, =SUMPRODUCT(rng1,rng2) works perfectly well in VBA as Application.WorksheetFunction.SUMPRODUCT(rng1,rng2), as VBA is conforming to the functions call criteria

[5] When embedding quotes within a string, the quotes have to be doubled up, otherwise the single quote is taken as the start or end of the string. This gets more complex if the quotes are just after or just before an opening/closing quote, as we then have three quotes, i.e. one to tell VBA that the next quotes is part of the string, one for the embedded quotes, and one to close the string

[6] The error is not caused because the text field is being summed, SUM happily ignores text, but rather because the value in column B is multiplied by the result of the conditional tests, it is multiplying text by a number that causes the #VALUE!

## References

As mentioned above, a detailed discussion on arrays and array formulae, by Microsoft Excel MVP Chip Pearson, can be found [here](http://www.cpearson.com/excel/array.htm).

Another real-world example of SUMPRODUCT is given in [Processing Coloured Cells](http://www.xldynamic.com/source/xld.ColourCounter.html) which, in conjunction with a custom UDF, shows how to count colour instances.

Felipe Gualberto has translated the majority of this page into Portuguese, which can be found [here](http://www.ambienteoffice.com.br/).

# Why use -- in SUMPRODUCT formula

SUMPRODUCT() ignores non-numeric entries. A comparison returns a boolean (TRUE/FALSE) value, which is non-numeric. XL automatically coerces boolean values to numeric values (1/0, respectively) in arithmetic operations (e.g., TRUE + 0 = 1).

The most efficient way to coerce the value is first to apply the unary minus operator, coercing TRUE/FALSE to -1/0, then applying it again to negate the value, e.g., +1/0.

The array is then numeric and will be evaluated by SUMPRODUCT().

## An example

Say you have the formula

=SUMPRODUCT(--(A1:A5>10),B1:B5))

and the following values in A1:B5:

A B

1 20 2

2 5 7

3 10 5

4 8 3

5 11 6

The first term (A1:A5>10) evaluates to

{TRUE,FALSE,FALSE,FALSE,TRUE}

The first unary minus coerces the array to

{-1, 0, 0, 0, -1}

The second negates the first

{1, 0, 0, 0, 1}

So the SUMPRODUCT() evaluates

=SUMPRODUCT({1,0,0,0,1},{2,7,5,3,6})

returning (1 \* 2) + (0 \* 7) + (0 \* 5) + (0 \* 3) + (1 \* 6) = 8.

## SUMPRODUCT with Sub-Strings

In this example we want to search cells for a substring and report values/products for those cells that contain the substring. The trick is to combine the ISNUMBER() function which evaluates True/False with the SEARCH() function as in the following example which counts the number of matches:

=SUMPRODUCT(--ISNUMBER(SEARCH(“Test String”,A1:AZ999,1)))

In the usual way we can expand SUMPRODUCT() to report the total of corresponding cells as follows:

=SUMPRODUCT(--ISNUMBER(SEARCH("TEST",D3:D14,1)),(E3:E14))

I believe this same approach of nesting functions to yield a true/false result can be used to effect with SUMPRODUCT() in many different ways.

## SUMPRODUCT for counting unique values

In this example we want to search cells and report the number of unique values found in those cells. The trick is to combine the FREQUENCY(range, range) function over the same range as input and output. This returns the frequency of each unique entry as an array. We turn this into true/false by testing that the frequency is greater than zero as follows:

=SUMPRODUCT(--FREQUENCY(A1:AZ999, A1:AZ999)>0))

The result is the number of unique values found in the range A1:AZ999.