SOME USEFUL "ENGINEERING" FEATURES IN EXCEL

INTRODUCTION

This spreadsheet details some Excel features that can be useful when you are developing spreadsheets for the solution of engineering problems. It began life as the screen "notes" for a short talk I had been requested to give, and has gradually grown in scope since then.

The features chosen to be discussed here are not merely those that I believe to be useful. They are also those that I have found engineers tend not to use, either because they (the engineers) are unaware of them, or because they believe the features are "difficult".

The spreadsheet is organised into several worksheets, with different classes of feature on each. These worksheets are as follows.

(1) **Introduction**: The sheet you are presently viewing.
(2) **Presentation features**: Features to make your spreadsheet more "user friendly", more robust, or more readable.
(3) **Array formulas**: Array formulas and array functions seem to be among the best-kept secrets of Excel.
(4) **User defined functions**: Using VBA, you can extend Excel's extensive function library to include functions you develop yourself.
(5) **Solver**: Excel's "solver" add-in is intended for use in optimisation problems, but can also be used to solve nonlinear simultaneous equations.
(6) **Miscellaneous**: As the name implies. Items discussed are:
   - **DATEVALUE** — to convert text to dates.
   - **INDIRECT** — to convert text to a usable cell reference.
   - **SUMIF** — to add up values that satisfy a single condition.
   - **SUMPRODUCT** — to add up values that satisfy multiple conditions.
   - **SUMPRODUCT** — to retrieve a value from a closed spreadsheet.
   - Dynamic ranges — to define a range of cells that will expand as more data is added to it.
   - A minor warning regarding the use of the **OFFSET** function — relevant only if your spreadsheet is set for manual recalculation.
   - **Recalculation** — a web site for information on how Excel recalculates a spreadsheet.
   - **TABLE** — to tabulate the effect of changing one "input" variable on the value of an "output" variable, a process generally known as sensitivity analysis or parametric analysis.

Suggestions for further inclusions in this spreadsheet are always welcome (but will not necessarily be implemented).

REVISION HISTORY

<table>
<thead>
<tr>
<th>Ver</th>
<th>Description</th>
<th>Release date</th>
<th>Released by</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>Initially developed.</td>
<td>Dec 2003</td>
<td>Rob Niall</td>
</tr>
<tr>
<td></td>
<td>— Various additions and improvements over the years.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.00</td>
<td>First &quot;public&quot; release.</td>
<td>May 2015</td>
<td>Rob Niall</td>
</tr>
</tbody>
</table>
**PRESENTATION FEATURES**

If you are writing a spreadsheet for later re-use, either by yourself or by others, you can probably afford to invest a bit more time in it to make it more friendly and more bulletproof.

Some of the features described below might help in this process.

**PROTECTING CELLS**

By default all cells in Excel are locked but the protection is set OFF. Most engineering-type spreadsheets, once they have been developed, will have only a small number of cells into which the user will enter the problem's description and its defining numerical value: We do not want a fumble-fingered user to clobber a formula when entering the problem's data. It is a good idea to unlock the individual cells into which the user is allowed/expected to enter the problem's descriptive data, then turn the overall worksheet protection ON.

A password can be used to inhibit subsequent unprotecting, but is not compulsory. If you do use a password make sure you remember it, and if it is not a personal spreadsheet make sure someone with project authority also knows it and records it. (Be aware that password protection is very easily cracked by a knowledgable person, and so it offers no real protection against such behaviours as the theft of any intellectual property embedded in the spreadsheet.)

If you do unlock the input cells, it can be helpful if you also add a faint colour to them. This aids the user to see where he is expected to enter some data, and it also helps any subsequent checker to know where to look (assuming that this brave checker is prepared to accept that the spreadsheet logic and construction are correct).

Why a "faint" colour? Because you have to allow for the spreadsheet to remain readable after it has been printed out on a black & white printer and then faxed to someone.

**DATA VALIDATION**

Excel allows you to set up a cell so that it will only accept contents of a certain kind. You can restrict entry to numbers, dates, text strings of constrained lengths, or integer numbers. Even more usefully, you can restrict entry to an item that appears on a list, where that list can be specified within the feature or in a set of contiguous cells elsewhere on the spreadsheet (or on another worksheet within the same workbook).

Data validation is invoked from the menu via Data / Validation.

**Example** with the list specified within the feature:

This cell must contain Mon, Tue, Wed, Thur or Fri → Tue

**Example** with the list specified in the spreadsheet:

This cell must contain an item from the list to the right → Elliott

List of alternative valid entries

- Utt
- Connor
- Elliott
- Niall

You can customise the message that appears for an invalid entry, as you will see if you put your own name in the cell concerned.

To see how the two above examples work, this worksheet needs to be unprotected.

**CONDITIONAL FORMATTING**

Excel also allows you to set up a cell to that its format changes according to what it contains, or according to what another cell contains, or according to almost any condition.
or set of conditions.  
This feature is most useful for highlighting errors or other strange conditions.

Conditional formatting is invoked from the menu via Format / Conditional Formatting.

**Example** with the format dependent on the contents of the cell itself:
Text in this cell will be red if it contains a negative number → **-1**

**Example** with the format depending upon more complex criteria:
Any contents in this cell → **Hello**
will be red if this cell contains an even number → **3**

### SUPPRESSING ZEROS

Excel allows you to specify different presentations for a number according to whether the number is positive, negative or zero.

This feature can be used when you want a cell to appear blank if it contains a number that is zero, although it offers a huge number of further capabilities.

It is invoked from the menu via Format / Cells / Number / Custom. You then enter a "formatting string" into the space headed "Type:"
The formatting string has up to four components, separated by semicolons.
The first component is the numeric formatting string for a positive number, the second is the formatting string for a negative number, and the third is the one for a zero number.
The fourth component (seldom needed) is for if the cell content is text.

**Examples** using **General;-General;** as the overall entry.
Note that the third formatting string, the one after the second semicolon, is deliberately null.
This is the way you specify that a zero number is to appear blank.

<table>
<thead>
<tr>
<th>Number</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>-1.074E+11</td>
<td>###</td>
</tr>
</tbody>
</table>

**Examples** using **00;-00;"·"** as the overall entry.
Here the third formatting string will show a zero as a small dot, for placeholding purposes.
This dot character is ANSI 0183. It could just as easily have been a dash character (-).
Non-zero numbers will display at least two digits, with leading zeros if necessary.

<table>
<thead>
<tr>
<th>Number</th>
<th>Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>-1</td>
<td>-01</td>
</tr>
<tr>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>-5.4</td>
<td>-05</td>
</tr>
<tr>
<td>0</td>
<td>.</td>
</tr>
<tr>
<td>-1.074E+11</td>
<td>###</td>
</tr>
</tbody>
</table>

Take great care that you include a minus sign in your formatting string for negative numbers, or your negative numbers will be displayed as if they are positive. **Not a good look.**
For a comprehensive discussion of custom formats, see the web site http://www.ozgrid.com/Excel/CustomFormats.htm

ADDING "COMMENTS" TO CELLS

If you wish to provide your users with a readily accessible description of what they should enter into a cell, or how to interpret a particular calculated result, consider adding a cell comment. This is done using RightClick>InsertComment. The resulting comment can be edited using RightClick>EditComment, and deleted using RightClick>DeleteComment.

The presence of a comment "behind" a cell is indicated by the cell having a small red triangle in its top right corner. The comment itself will appear on the screen when you "hover" your cursor over the cell.

In the case of an input value, it can sometimes be better to place your comment on the cell that describes the actual input cell.

Bandersnatch factor 1.2
Most engineering-type spreadsheets, once they have been developed, will have only a small number of cells into which the user will enter the problem's description and its defining numerical values.
ARRAY FORMULAS

DEFINITION  (Extracted from the depths of Excel's Help system)

A formula that performs multiple calculations on one or more sets of values, and then returns either a single result or multiple results. Array formulas are displayed enclosed between braces { }, and are entered by pressing Ctrl+Shift+Enter. (Excel automatically adds the braces.)

BUILT-IN ARRAY FUNCTIONS

Numerous of Excel's built-in functions are Array Functions, in that they return multiple values.

An example is the linear regression function LINEST(y-vals, x-vals, ..., ...)

This returns two (or more) values in a row, as demonstrated below:

<table>
<thead>
<tr>
<th>Data</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.95</td>
<td>2.89</td>
<td></td>
</tr>
<tr>
<td>1.9</td>
<td>4.85</td>
<td></td>
</tr>
<tr>
<td>3.04</td>
<td>6.78</td>
<td></td>
</tr>
<tr>
<td>3.97</td>
<td>9.08</td>
<td></td>
</tr>
</tbody>
</table>

Results

2.004818  0.958125 ← The two values returned when the array function is properly specified.
2.004818   ← If you forget the Ctrl+Shift bit, you only get the first value (the slope).
0.958125   ← If you only want the 2nd value (the intercept), use the INDEX function.

ARRAY FORMULAS YOU CONSTRUCT YOURSELF

You enter these into cells the same way as you enter "normal" formulas. The key difference is that you enter a RANGE of cells rather than a single cell as the operand. The range can be either a column of contiguous cells or a row of contiguous cells. When you are entering multiple ranges into the same formula, each must be the same size.

You will find more discussion of array formulas on the "Miscellaneous" worksheet of this workbook, under the discussion of the extended capabilities of the SUMPRODUCT function.

Example simply copying one range to another:

<table>
<thead>
<tr>
<th>Range</th>
<th>Copy</th>
<th>&quot;Transposed&quot; copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1 2 3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Example involving two input ranges, plus some inputs that are NOT ranges:

<table>
<thead>
<tr>
<th>Range A</th>
<th>Range B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mult X</th>
<th>Mult Y</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>2</td>
<td>X<em>A + Y</em>B</td>
</tr>
</tbody>
</table>

Examples where the resulting array is itself the input to a subsequent operation within the formula:
(These summations just happen to be the ones required for linear regression.)

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>$\Sigma(X)$</th>
<th>$\Sigma(Y)$</th>
<th>$\Sigma(X^2)$</th>
<th>$\Sigma(XY)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.95</td>
<td>2.89</td>
<td>9.86</td>
<td>A normal formula</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.9</td>
<td>4.85</td>
<td>23.6</td>
<td>A normal formula</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.04</td>
<td>6.78</td>
<td>29.515</td>
<td>An array formula</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.97</td>
<td>9.08</td>
<td>68.6193</td>
<td>An array formula</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Incorrect attempt at $\Sigma(XY)$ → #VALUE! ← Entered without the Ctrl+Shift t

**Example** of a less-trivial task. We have two columns of numbers, and we wish to keep a count of how many rows have the same value in each of the two columns.

Think about how you might achieve this in a single formula before you look at the answer.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>11</td>
</tr>
<tr>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
</tr>
</tbody>
</table>

2 ← Number of identical rows

There are several other ways of achieving this result. However this particular approach has the advantage that it continues to work if the columns contain non-numeric data.

**WARNINGS**

If your range arguments are not of equal length you will get unpredictable & wrong results.
If your range arguments are not either all horizontal or all vertical, ditto.
If you forget the Ctrl+Shift bit, ditto again. (See above for an example of this problem.)

**CONCLUSION**

Array formulas can be tricky to learn and use at first, and take some getting used to. But once you understand how they work and become comfortable with them, you will find that they are an extremely powerful tool for writing useful and concise formulæ.

For some more examples, and the associated mental gymnastics, see

http://www.cpearson.com/excel/array.htm
The two values returned when the array function is properly specified. If you forget the Ctrl+Shift bit, you only get the first value (the slope). If you only want the 2nd value (the intercept), use the INDEX function.
USER DEFINED FUNCTIONS

INTRODUCTION

Functions are the key building blocks for spreadsheets, offering the twin benefits of capability and flexibility. Although Excel is shipped with a large repertoire of in-built functions, these are mainly general in nature. User defined functions (UDFs) allow you to achieve these same benefits in specific disciplines. UDFs are written in a programming language "Visual Basic for Applications" (VBA), which is in effect a cut-down version of Microsoft's "Visual Basic" language. Programming is done using the "Visual Basic Editor" (VBE), which is an integral part of Excel.

Once a UDF has been programmed (and tested) it forms an integral part of the spreadsheet, and behaves almost identically to the in-built functions. If you want your UDFs to be available beyond the spreadsheet in which they were developed, they can be formed into an "add-in". This add-in can then be distributed to all and sundry. Even if you have no intention of distributing the add-in to other people, the capabilities it contains are available to all your spreadsheets.

CREATING A SIMPLE UDF

UDFs live in what are called "modules" within the VBE, where they are associated with a workbook. To create a simple UDF, do the following:

1. If the workbook already has a module, you can add to it, so open it. Then scroll to its end.
2. Begin typing code for your UDF by entering: `Function SomeName(Argument1, Argument2, ...)`
3. Enter the lines of VBA programming, culminating in assigning your answer to an internal variable.
4. The UDF is now complete.
5. Return to the workbook by closing the VBE or by the shortcut Alt-F11.
6. Test the UDF for a variety of input conditions. Then test it again.

Here is a totally trivial example that merely adds up two numbers and returns the result of the summation.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A+B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>#VALUE!</td>
</tr>
<tr>
<td>-11</td>
<td>12</td>
<td>#VALUE!</td>
</tr>
<tr>
<td>3.141593</td>
<td>2.718282</td>
<td>#VALUE!</td>
</tr>
<tr>
<td>15</td>
<td>Hello</td>
<td>#VALUE!</td>
</tr>
</tbody>
</table>

A MORE COMPLICATED EXAMPLE

Suppose you are building a spreadsheet that requires quadratic equations to be solved in numerous places. It will be advantageous to develop a UDF to do this, then invoke that UDF wherever it is required.

The general quadratic equation is A*X² + B*X + C = 0 and its solution is (–B ± √(B²– 4AC))/(2A)

So our UDF will have to return two values, ie it will have to be an Array Function.

To be completely general, it will also have to trap two special conditions:

1. The case when the input values A, B and C do not define a quadratic equation;
2. The case when the quadratic equation is defined but has no solution.
There is a UDF to do this already written, and associated with this workbook. It is called `SolveQuad(A, B, C)`. Open the VBE to see it. Feed it some sets of data:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>1st ans</th>
<th>2nd ans</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>-6</td>
<td>#VALUE!</td>
<td>#VALUE!</td>
</tr>
<tr>
<td>1</td>
<td>-4</td>
<td>4</td>
<td>#VALUE!</td>
<td>#VALUE!</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>#VALUE!</td>
<td>#VALUE!</td>
</tr>
<tr>
<td>1</td>
<td>-6</td>
<td>11</td>
<td>#VALUE!</td>
<td>#VALUE!</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>-7</td>
<td>#VALUE!</td>
<td>#VALUE!</td>
</tr>
</tbody>
</table>

**USING A UDF TO SOLVE AN ITERATIVE PROCEDURE**

Many engineering procedures involve solving implicit equations. Such equations often require iterative techniques for their solutions, and iterative solutions are not readily solvable in the spreadsheet environment. However they are able to be attacked using VBA programming, and are ideal candidates for a UDF.

An example of an iterative procedure is the Colebrook-White equation for partially-full flow of a liquid along a pipe. A solution has been developed for this problem in the case of a pipe whose cross-section is elliptical, and it is included with this workbook. Note that it does not simply return the flow depth, but it also returns the wetted perimeter, the width of the surface of the liquid, the cross-sectional area of the void above the liquid, and the hydraulic radius of the flow. (The user is under no obligation to use all of these returned values.)

**Inputs:**

- `3.5`
- `0.6`
- `1.5`
- `0.001`
- `0.0008`
- `1.00E-06`

**Outputs:**

- `#VALUE!`  
- `#VALUE!`  
- `#VALUE!`  
- `#VALUE!`  
- `#VALUE!`  

**SOME HINTS FOR WRITING USER DEFINED FUNCTIONS**

UDFs can call other UDFs from within themselves. VBA has an extensive number of its own in-built functions, such as the `Sqr()` function used in `SolveQuad`. In addition to VBA's own in-built functions, VBA allows a UDF to call many of Excel's in-built functions.
UDFs can only obtain information from the spreadsheet via the contents of the cells that are passed to them as arguments. They have no access to any other information on the spreadsheet, and no access to any formatting information etc applied to the cells they are passed.

UDFs can only pass information to the spreadsheet in the form of the content of the cells from which they are invoked. They cannot change the formatting etc of those cells. They cannot influence any other cells in any way whatsoever.

In these two regards (getting information from the spreadsheet and passing results back to the spreadsheet), VBA's UDFs are quite different from its "subroutines". Subroutines are more powerful in that they can do just about anything, but they are less flexible in that they require knowledge of the layout of the spreadsheet. They also need to be triggered in some way before they will run, whereas a UDF (like all Excel's functions) will run automatically whenever a change in the spreadsheet data requires that it be run. UDFs do have access to information on the general environment, such as the time of day. They can put messages up on the screen, and they can ask for responses from the user. These messages can include intermediate results, a capability which is extremely useful when you are trying to get a UDF working correctly.

A REFERENCE BOOK

The best introductory reference book on Excel (IMHO) is "Excel 97 Programming for Windows for Dummies", by John Walkenbach. Unfortunately it is out of print.
Functions are the key building blocks for spreadsheets, offering the twin benefits of capability and flexibility. Although Excel is shipped with a large repertoire of in-built functions, these are mainly general in nature. UDFs are written in a programming language "Visual Basic for Applications" (VBA), which is in effect a

Once a UDF has been programmed (and tested) it forms an integral part of the spreadsheet, and behaves

If you want your UDFs to be available beyond the spreadsheet in which they were developed, they can be

If the workbook already has a module, you can add to it, so open it. Then scroll to its end.

Enter the lines of VBA programming, culminating in assigning your answer to an internal variable.

Here is a totally trivial example that merely adds up two numbers and returns the result of the summation.

Suppose you are building a spreadsheet that requires quadratic equations to be solved in numerous places.

If there is no module associated with your workbook, you must create one with

_I

aves

n be

_Jule.

variable.

e

ion.

laces.
VBA has an extensive number of its own in-built functions, such as the Sqr() function used in SolveQuad.

In addition to VBA's own in-built functions, VBA allows a UDF to call many of Excel's in-built functions.

No solution (negative discriminant)

Kinematic viscosity (m²/s)
o tell
P, Q).
**SOME USES FOR THE "SOLVER" ADD-IN**

**GENERAL DESCRIPTION**

Solver adjusts the values in a group of cells so as to "optimise" the value in one other cell. It is a standard part of Excel, but must be activated (once only) before it can be used. This activation is done via Tools>Add-Ins, or Developer>Add-Ins for more recent versions of Excel.

Solver is much more powerful and versatile than the "Goal Seek" tool:
- It can minimise or maximise the target cell as well as set it to a value;
- It can adjust many cells in order to achieve its aim;
- It can accommodate "constraints".

**Warnings:**
Solver draws no distinction between a local optimum and the global optimum. The Solver with Excel-2003 (and earlier) left solution variables totally unconstrained by default. In Excel-2010 this had changed, and solution variables were constrained to being non-negative by default. I am uncertain of the situation with Excel-2007. If you are trying to make your spreadsheets work on as many versions of Excel as possible you should always explicitly specify the acceptibility of negative solution values. (This is achieved by ensuring the appropriate check box is UNCHECKED.)

If your problem has multiple answers you cannot predict which solution Solver will home in on. If you start too far away from the answer, Solver might trigger an error condition (such as taking the square root of a negative number) as it searches over a wide range of possible scenarios. This will abort the solution.

Solver will not run on a worksheet that is protected, even if the cells it is to change are unlocked. (Therefore this worksheet is not protected.)

**SIMPLE EXAMPLE (WITHOUT CONSTRAINTS)**

Want to find the value of X that minimises the expression \((X-1)^2 + \sin(6X)\)

\[
X \rightarrow 3.2 \quad \text{← This is the cell whose starting value Solver should adjust}
\]

\[
(X-1)^2 + \sin(6X) \rightarrow 5.183315 \quad \text{← This is the "target" cell, whose value we wish to minimise.}
\]

The expression \((X-1)^2 + \sin(6X)\) is shown as the solid line on the graph below, whilst the current X point is shown as a small solid diamond.
Use the Solver on this problem.  
Try it with different starting points (1.5 will give a false local optimum, 1.0 will give the true answer).  
Try it with an outrageous starting point such as 1000.  What happens?  Why?  
Note how the Solver remembers the details of its last usage on the worksheet, thereby making it easier to make minor modifications and then re-run it.

NOW ADD ONE CONSTRAINT TO THE ABOVE PROBLEM

Add a requirement that X*cos(X^2) be positive.

\[ X \cdot \cos(X^2) \rightarrow -2.194262 \leftarrow \text{This cell must be positive.} \]

The expression X*cos(X^2) is shown as the dashed line on the graph above.

USE OF SOLVER FOR SIMULTANEOUS EQUATIONS

Consider the three simultaneous equations:

\[
\begin{align*}
3X^2 - 5\cos(Y) + 1.5^2 &= 26 \\
\log_{10}(X) - Z^{1/2} &= -1 \\
2X + Y\tan(Z) &= 8
\end{align*}
\]

Can the Solver be used to obtain an answer to this problem?  
We have 3 targets to meet, but the solver can handle only one target cell.

Do we have to resort to trial-and-error?

Set up a calculation structure as follows. The equations are expressed in "=0" form, and the expressions on the resulting left-hand sides are evaluated for the trial X,Y,Z values.  
When we have found our solution the results of these evaluations will all be zero, and the amount by which they differ from zero is a measure of the error associated with any set of trial values.

\[
\begin{align*}
\text{Trial value for } X &\rightarrow 3 \leftarrow \\
\text{Trial value for } Y &\rightarrow -1 \leftarrow \text{These three cells are the ones to change as we seek} \\
\text{Trial value for } Z &\rightarrow 2 \leftarrow \text{An adequate starting point is (3,-} \\
3X^2 - 5\cos(Y) + 1.5^2 - 26 &\rightarrow 0.548488 \leftarrow \\
\log_{10}(X) - Z^{1/2} + 1 &\rightarrow 0.062908 \leftarrow \text{These three cells are the ones we want to be zero.} \\
2X + Y\tan(Z) - 8 &\rightarrow 0.18504 \leftarrow
\end{align*}
\]

Think about it. Then scroll down for an answer.
Since we are trying to make all three target cells equal to zero, we can construct a single composite target cell whose value is the sum of the squares of the three actual targets.

\[ 0.339037 \leftarrow \text{This is the "composite target cell" we wish to be zero.} \]

If this composite target cell is then minimised, the deviations from zero of the individual target cells will also have been minimised.

However, we need to give Solver an initial trial answer that is reasonably close to correct. Luckily, in most "engineering" applications this is not too difficult to achieve.

Repeat the exercise with an excessively distant starting point such as (20,20,20). Oops.

Note that it is possible (but less elegant and less amenable to generalisation) to formulate and solve this problem by regarding two of the equations as constraints whilst seeking a zero value for the third equation. Try this approach as a further exercise.
to being non-negative by default. I am uncertain of the situation with Excel-2007. If you
you should always explicitly specify the acceptibility of negative solution values. (This is
If your problem has multiple answers you cannot predict which solution Solver will home in on.
taking the square root of a negative number) as it searches over a wide range of possible

This is the cell whose starting value Solver should adjust.

This is the "target" cell, whose value we wish to minimise.

+ sin(6X) is shown as the solid line on the graph below, whilst the current X point
Set up a calculation structure as follows. The equations are expressed in "=0" form, and the expressions.

An adequate starting point is (3, -1, 2).

These three cells are the ones to change as we seek a solution. How?
<table>
<thead>
<tr>
<th>X</th>
<th>F(X)</th>
<th>X*cos(X^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>9.536573</td>
<td>1.307287</td>
</tr>
<tr>
<td>-1.875</td>
<td>9.233433</td>
<td>1.745366</td>
</tr>
<tr>
<td>-1.75</td>
<td>8.442196</td>
<td>1.744529</td>
</tr>
<tr>
<td>-1.625</td>
<td>7.210144</td>
<td>1.425317</td>
</tr>
<tr>
<td>-1.5</td>
<td>5.837882</td>
<td>0.94226</td>
</tr>
</tbody>
</table>

This is the "composite target cell" we wish to be zero.

\[ \text{X} \cos(X^2) \]
**MISCELLANEOUS FEATURES**

This page lists various features in Excel that (in this author's opinion) are firstly very useful and secondly either inadequately publicised or confusingly documented.

**DATEVALUE (text)**

Converts a text string in the form of a date into an actual date that Excel can operate upon. If the text string cannot be interpreted as a date, an error is returned. An error is also returned if the argument is not actually a text string.

This is often required if you are importing dates into your spreadsheet from external sources.

<table>
<thead>
<tr>
<th>Text</th>
<th>DATEVALUE(text) (unformatted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>31Dec48</td>
<td>Err:502</td>
</tr>
<tr>
<td>2 Jan 1949</td>
<td>Err:502</td>
</tr>
<tr>
<td>Garbage</td>
<td>Err:502</td>
</tr>
<tr>
<td>2/1/49</td>
<td>17930</td>
</tr>
<tr>
<td>1/2/1949</td>
<td>Err:502</td>
</tr>
</tbody>
</table>

**INDIRECT (text, a1style)**

Converts a text string in the form of a cell reference into an actual cell reference that can be used in formulae. The *a1style* argument is optional: if it is TRUE or omitted the function attempts to interpret the text string as an "A1-style" reference; otherwise it assumes the "R1C1-style" reference. The text can also be a "defined name".

This function is invaluable under certain unusual circumstances, such as if we want the arguments of a function to depend upon the contents of a cell or upon the location of a cell.

Simple examples:

1. Put a cell address here (as text in A1-style) 
   Here is the contents of cell G17          G17
   Err:502

2. Put a cell address here (as text in R1C1-style) 
   Here is the contents of cell R17C7        R17C7
   Err:502

Complicated example:

Put a cell address here (as text in A1-style) 
Here is the sum of all the cells above cell G17   Err:502

How does this work? Let's look at it step by step:

1. Get the row number of cell G17 17
2. Get the column number of cell G17 7
3. Form the R1C1-style address for the start of the range we wish to sum (noting the need for the TEXT function) R1C7
4. Form the R1C1-style address for the end of the range we wish to sum R16C7
5. Form the full range to be summed R1C7:R16C7
6. Do the summation Err:502
SUMIF (test_range, criterion, sum_range)

Adds those numbers in the range sum_range whose corresponding entries in the range test_range satisfy criterion. If the sum_range argument is omitted it is taken as being the same as test_range.

This function and its companion COUNTIF provide an easy way to achieve some limited database capability. (More substantial database capabilities are available using the “database functions” such as DCOUNT, DSUM etc.)

As an example, consider the following table:

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Sold cases</th>
<th>Received money</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oranges</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>Apples</td>
<td>10</td>
<td>98</td>
</tr>
<tr>
<td>Pears</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>apples</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>oraNGes</td>
<td>3</td>
<td>35</td>
</tr>
<tr>
<td>Oranges</td>
<td>6</td>
<td>50</td>
</tr>
</tbody>
</table>

Number of cases of oranges sold 14
Money rec'd from sales of <5 cases 88

The criterion can be in a cell rather than embedded in the SUMIF formula:

Number of cases of oranges sold 14
Money from sales of <5 cases 88

Multiple criteria cannot be handled by SUMIF, and will probably need DSUM:

Cases Fruit
Money from sales of <=5 oranges 80

An alternative and more flexible way of accommodating multiple criteria can o be achieved by creative use of the SUMPRODUCT function, as described bel

Using SUMPRODUCT to add up values that satisfy multiple conditions

The intention behind the SUMPRODUCT function is to take a series of arguments each of which is a range (or array) of the same size, multiply corresponding terms together, then return the sum the resulting products. It thus expects each of its arguments to be an array or to resolve to an array, and it further expects these arrays to be (or to resolve to) numbers rather than text.

As a consequence of these expectations, if an argument is some sort of formula rather th simply a directly-entered range of cell addresses then Excel will attempt to evaluate this formula in its "array mode”. With this "array mode” (which is also relevant to the discussi array formulas on the “Array formulas” worksheet of this workbook) the designated opera are performed element by corresponding element, with the intermediate results being sto as the corresponding element in yet another (temporary and internal) array.

Under the rules of Excel's array-mode arithmetic, the following examples apply.
If A1:A3 contains the array {2,3,4} and B11:B13 contains the array {10,100,1000} then A1:A3*B11:B13 will resolve to the (column) array {20,300,4000} and TRANSPOSE(A1:A3)+TRANSPOSE(B11:B13) will resolve to the (row) array {12,103,1004} and MOD(B11:B13,A1:A3) will resolve to the (column) array {0,1,0}. However MAX(MOD(B11:B13,A1:A3)) will resolve to the single value 3 only if it is entered as an "array function". This point is discussed in a bit more detail below.

Note that only the very brave will attempt to invoke array-mode calculations between arrays that are not of the same size and shape. If you breach this guidance you will usually still get results, but the evaluation rules that will be applied defy ready description.

How can array mode calculations be extended to evaluate a condition? Let's continue with the above examples for a bit longer. Now A1:A3>2 will resolve to the array of logical values {FALSE,TRUE,TRUE} where each element is the logical result of a corresponding logical test and -(A1:A3>2) will resolve to the array of numerical values {0,1,1} because the minus sign (the "unary negation" operator) converts the logical value FALSE to the numeric value 0, and converts TRUE to 1 then negates it to -1. By applying the operator twice, TRUE ends up as 1 rather than -1. Note that there are other ways to convert a logical array to its numerical equivalent. All that is necessary is the application of any arithmetic operator (other than a naked plus sign). Thus (A1:A3>2)*1 or 0+(A1:A3>2) would also work. This author prefers the use of "--" because its alarming obscurity makes its intention clearer to the cognoscenti, whilst forcing thought upon the uninitiated.

Now extend this to the evaluation of multiple conditions. The easier case is that where the multiple conditions must all be met (known in set theory as intersection, and in Boolean algebra as the AND'ing of conditions). Multiplying together a series of arrays of TRUE/FALSE logical values produces an array of 1/0 numerical values, where a 1 value will be produced if and only if all the corresponding multiplied values are TRUE. Thus (A1:A3>2)*(B11:B13<=100) will resolve to the column array {0,1,0} and so SUMPRODUCT((A1:A3>2)*(B11:B13<=100)) will yield the number of instances where both the value in column A is greater and the corresponding value in column B is less than or equal to 100. Note that in these formulations the "--" is not required, because the "*" operator induces the required conversion from logical to numeric.

To see how these capabilities can be further extended we shall return to the earlier exam of sales of cases of fruit, reproduced below for convenience.

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Sold cases</th>
<th>Received money</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oranges</td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td>Apples</td>
<td>10</td>
<td>98</td>
</tr>
<tr>
<td>Pears</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>apples</td>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>oranGes</td>
<td>3</td>
<td>35</td>
</tr>
<tr>
<td>Oranges</td>
<td>6</td>
<td>50</td>
</tr>
</tbody>
</table>

The money received from the sales of five or less cases of oranges can be obtained from SUMPRODUCT( (C147:C152="oranges")*(D147:D152<=5) , E147:E152 ) which multiplies the elements of the resulting 1/0 array by the array of dollar values.
The more difficult case is where only one, or more, of the specified multiple conditions need to be met (known in set theory as "union", and in Boolean algebra as OR'ing the conditions). Adding together a series of arrays of TRUE/FALSE logical values might superficially seem to do the job, but entries that meet more than one of the conditions will be counted more than once. The neatest way this author has found around this problem is to use the SIGN function on the resulting array.

Then \[ \text{SUMPRODUCT( SIGN((C147:C152="oranges")+(D147:D152<=5)) ) } \]

will give the number of instances where **either** oranges or less than six cases were sold (regardless of the contents of the cases sold)

while \[ \text{SUMPRODUCT( SIGN((C147:C152="oranges")+(D147:D152<=5)) , E147:E152) } \]

will give the total money received from those instances.

However \[ \text{SUM( SIGN((C147:C152="oranges")+(D147:D152<=5)) ) } \]

will work correctly only if it is entered as an array function (see below).

Further notes and caveats for this use of the SUMPRODUCT function and arrays.

1. For both AND'ing and OR'ing, the approach is readily scalable to more than two conditions.
2. When using SUMPRODUCT the arrays being evaluated must all be columns or all be rows. If you ignore this requirement, you will still get an answer but it will be the wrong answer.
3. You can use the TRANPOSE function to meet the requirement of point 2 above. However if you do, you will need to enter the resulting SUMPRODUCT as an array function. If you ignore this requirement you will again still get an answer, but again it will be the wrong answer.
4. Point 3 above is an example of a more general phenomenon. When you use array-mode calculations inside a function that returns a single-cell result, you cannot readily predict whether you will need to enter it as a normal function or as an array function. This means that you must rely on detailed checking. Such checking would best involve both an example where the correct form produces the right answer and an example where the incorrect form produces the wrong answer.
5. For a more detailed exposition on this subject, see www.xldynamic.com/source/xld.SUMPRODUCT.html which was the principal source used in producing this material.
6. A new function, SUMIFS, was introduced in Excel-2007. This function is aimed specifically at summing values that satisfy multiple conditions. You will be better off using SUMIFS possible, but only if you are sure that nobody will want to use your spreadsheet on earlier versions of Excel.

**Using SUMPRODUCT to retrieve a value from a closed spreadsheet**

It is widely believed that Excel cannot retrieve the contents of a cell from another spreadsheet if that spreadsheet is closed. There is, however, a method to retrieve the value of a numeric cell from a closed spreadsheet.

The method uses the SUMPRODUCT function, and the format for the argument(s) is as follows:

\[ \text{SUMPRODUCT( 'path\{file.xls\}sheet'!cell ) } \]

This will retrieve the contents of cells that resolve to numbers or dates, but not text or logicals.

I do not understand why this capability exists. It is not documented anywhere by Microsoft and so it should be regarded as an unsupported feature. This means that whenever you use it you should test it thoroughly (even more thoroughly than you test all the other aspects of any spreadsheet you develop). It also means that you must re-test it whenever you down grade to a new version of Excel.
An Excel guru named Harlan Grove has written a macro (a "user defined function") that will retrieve the contents of a cell in a closed spreadsheet, be it text or numeric. It is called PULL, and an Internet search should be able to locate this for you. However, it is quite slow to run, so you will be better off using SUMPRODUCT wherever possible.

With either method, you should explore the circumstances under which the retrieved cell contents are updated. This will always happen when you open your spreadsheet, but it probably will not happen under a normal recalculation. You can force a complete recalculation with the Ctrl-Alt-F9 key combination, (or Ctrl-Alt-Shift-F9 if you also want to rebuild the dependency table).

**Defining "dynamic ranges" (whose sizes automatically adjust to match the amount of data)**

*Names* are ranges of cells that are given a name, with the cells subsequently being able referred to by that *name* rather than by their addresses. It is often possible to create a *name* that refers to a range whose size will vary automatically according to the amount of data embraced. This *dynamic name* can then be used to drive graphs, or as a validation list, etc.

It might help you understand how this can be done if you think of a *name* as a *named for* rather than a *named range*. In most contexts the *name* evaluates directly as a range because it is a range, but to achieve a dynamic range requires that the *named formula* be a bit more complicated. It will normally involve the use of the OFFSET function.

The OFFSET function returns a range based on five arguments. Its calling sequence is

```
OFFSET (CellAddress, RowOffset, ColOffset, NumbRows, NumbCols)
```

where: *CellAddress* is a reference address on the spreadsheet; *RowOffset* and *ColOffset* locate the top left hand corner of the range relative to the reference address (with positive numbers indicating a location downwards or to the right respectively); and *NumbRows* and *NumbCols* are the dimensions of the (rectangular) range that is to be returned.

Here is a relatively simple example of a dynamic range that uses the OFFSET function:

```
=OFFSET( Sheet!$C$1, 0, 0, COUNTA(Sheet!$C:$C), 1 )
```

This will return a range that is 1 column wide, starts at cell C1, and extends downwards by a number of rows that equals the total number of non-empty cells in column C. Provided a data you wish the range to embrace is at the top of the row (and with no embedded empt a *name* containing this formula will do the trick.

If cell C6 contains a heading and the real data begins in cell C7, we could use

```
=OFFSET( Sheet!$C$6, 1, 0, COUNTA(Sheet!$C:$C)-1, 1 )
```

which would be fine provided cells C1 through C5 were empty.

To allow for any amount of emptiness/nonemptiness in cells C1 through C5, we could use

```
=OFFSET( Sheet!$C$6, 1, 0, COUNTA(Sheet!$C:$C)-COUNTA(Sheet!$C$1:$C$6), 1 )
```

This still requires that the data the range is to embrace be contiguous from C7 downward but this will usually be the case.

Can we further extend this formula to allow for some embedded empty cells in the range cell C6? I cannot think of a way, yet I feel sure that there will be one. Suggestions anyor

### A minor warning regarding use of the OFFSET function

If you have a large spreadsheet that takes quite a long time to recalculate, you might dec
to set Excel’s recalculation mode to "manual" so that you can control when to have Excel spend time on calculation. Normally when you do this, Excel tells you when a recalculation is required by putting the word "Calculate" in its status bar, and you can then initiate a recalculation by hitting the <F9> key at a time of your choosing.

However if you use the OFFSET function in a spreadsheet whose recalculation mode is set to "manual" Excel will not tell you when recalculation is required. (This behaviour is logical but is complicated to explain. It stems from the fact that the OFFSET function is so powerful that it has the potential to return the value of any cell on the worksheet. This makes it almost impossible for Excel to maintain its usual "dependency tree", the device it uses to keep track of which cells depend on which other cells.)

This behaviour is mentioned here because the usual way to create "dynamic arrays", discussed above, involves the OFFSET function. Some of the functionality of the OFFSET function can be achieved through the INDEX function. The INDEX function does not upset the reliability of the "Calculate" message in spreadsheets being recalculated manually.

As an example relevant to dynamic arrays, the formula

```
=OFFSET($B$5,0,0,COUNTA($B:$B)-COUNTA($B$1:$B$5)+1,1)
```

can be replaced with

```
=$B$5:INDEX($B:$B,ROW($B$5)+COUNTA($B:$B)-COUNTA($B$1:$B$5))
```

to achieve the same end result.

How Excel recalculates a spreadsheet

Normally you do not need to know how Excel recalculates a spreadsheet — it just happens quickly and accurately. However if you have a particularly large or complex spreadsheet recalculation time might become an issue. Understanding how Excel performs its calculations can sometimes help you to speed the process up.

The web site [http://www.decisionmodels.com/calcsecrets.htm](http://www.decisionmodels.com/calcsecrets.htm) contains a lot of very useful information on all aspects of Excel’s calculation engine. Anyone creating large spreadsheets should at least scan this site before beginning their task.

Creating "tables" for sensitivity analysis

Suppose you have set up a complicated sequence of calculations that takes a number of input values and produces an output value from them. You now want to see how change in one of the input values affect the output value. This process is known as sensitivity analysis or parametric analysis, and Excel provides an automated way to assemble the results of an analysis in tabular form. (Once in tabular form the results can readily be graphed.)

This feature is accessed via the Data>Table menu item with earlier versions of Excel, or via Data>DataTools>What-if>DataTable with versions 2007 and later.

1. Create a columnar range that contains the successive values for the input variable.
2. In the cell one row above the top of this range and one column to its right, put a formula that returns the value of the output variable. The easiest way to do this is usually to enter a formula along the lines of "=OutputCellAddress".
3. Select the resulting (N+1)row × (2)column range.
4. Invoke the feature’s menu as described above.
5. Enter as the column input cell the address for the input cell in your calculation sequence leaving the row input cell box blank.
(6) Hit the "OK" button. The table will be filled out with an array formula that calculates the appropriate values of your calculation sequence.

Points to note.
(1) The results in the table are not static: they are dynamic and will change as you change other parts of the spreadsheet.
(2) The table can be constructed in row form rather than in column form if you prefer.
(3) Other output cells can be included in the table by adding additional columns to the right; these new columns must share the same set of values for the input variable.
(4) There is a two-dimensional version of the Table feature, which allows you to investigate variations in two input variables simultaneously. One varies column-wise, the other row-wise.

An extremely simple example:

<table>
<thead>
<tr>
<th>Input values</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>-0.37</td>
</tr>
<tr>
<td>A</td>
<td>2.5</td>
</tr>
<tr>
<td>πX</td>
<td>-1.162389</td>
</tr>
<tr>
<td>A*sin(πX)</td>
<td>-2.294387</td>
</tr>
</tbody>
</table>

Enter here the values of X you wish to have evaluated in the tabulation. The corresponding output values are tabulated here.
Converts a text string in the form of a date into an actual date that Excel can operate upon. This is often required if you are importing dates into your spreadsheet from external sources.

`=VALUE(text)`
(formatted)

Err:502
Err:502
Err:502
01-Feb-49
Err:502
An alternative and more flexible way of accommodating multiple criteria can often be achieved by creative use of the SUMPRODUCT function, as described below. The intention behind the SUMPRODUCT function is to take a series of arguments each of return the sum the resulting products. It thus expects each of its arguments to be an array, or to resolve to an array, and it further expects these arrays to be (or to resolve to) numbers. As a consequence of these expectations, if an argument is some sort of formula rather than formula in its "array mode". With this "array mode" (which is also relevant to the discussion of array formulas on the "Array formulas" worksheet of this workbook) the designated operations are performed element by corresponding element, with the intermediate results being stored...
A1:A3 contains the array \{2,3,4\} and B11:B13 contains the array \{10,100,1000\}. 

\text{MAX(MOD(B11:B13,A1:A3))} will resolve to the single value 3 only if it is entered into a single cell.

Note that only the very brave will attempt to invoke array-mode calculations between arrays that are not of the same size and shape. If you breach this guidance you will usually still get strange results.

How can array mode calculations be extended to evaluate a condition? Let’s continue with the example:

A1:A3 \gt 2 will resolve to the array of logical values \{FALSE,TRUE,TRUE\} where each element indicates whether the condition is true or false.

\text{--(A1:A3 > 2)} will resolve to the array of numerical values \{0,1,1\} because the use of the minus sign (the “unary negation” operator) converts the logical value FALSE to the numeric value 0, and converts TRUE to 1, and then negates it to -1.

By applying the logical operator, the condition is evaluated for each element of the array.

Note that there are other ways to convert a logical array to its numerical equivalent. All that is needed is to multiply each element by 1, or add 0 to each element.

Thus \text{ (A1:A3 > 2) * 1 } or \text{ 0 + (A1:A3 > 2) } would also work. This author prefers the use of \text{ -- } because its alarming obscurity makes its intention clearer to the cognoscenti, whilst forcing the reader to think about the implications of their actions.

Now extend this to the evaluation of multiple conditions. The easier case is that where the conditions are ANDed together.

Multiple conditions must all be met (known in set theory as intersection, and in Boolean algebra as the logical AND).

The AND operation on arrays produces an array of 1/0 numerical values, where a 1 value will be produced if all conditions are true.

Note that in these formulations the \text{ -- } is not required, because the \text{ * } operator induces the AND operation on the logical arrays.

To see how these capabilities can be further extended we shall return to the earlier example.

Working cells used to create flexible “text”:
C147:C152  D147:D152  E147:E152
The more difficult case is where only one, or more, of the specified multiple conditions needs to be met (known in set theory as “union”, and in Boolean algebra as OR’ing the conditions). Adding together a series of arrays of TRUE/FALSE logical values might superficially seem to do the job, but entries that meet more than one of the conditions will be counted more than one time. The neatest way this author has found around this problem is to use the SIGN function:

\[
\text{SUMPRODUCT( SIGN((C147:C152="oranges")+(D147:D152<=5)) , E147:E152 )}
\]

\[
\text{SUM( SIGN((C147:C152="oranges")+(D147:D152<=5)) )}
\]

will work correctly only if:

1. For both AND’ing and OR’ing, the approach is readily scalable to more than two conditions.
2. When using SUMPRODUCT the arrays being evaluated must all be columns or all be rows. If you ignore this requirement, you will still get an answer but it will be the wrong answer.
3. However if you do, you will need to enter the resulting SUMPRODUCT as an array function.
4. Point 3 above is an example of a more general phenomenon. When you use array-mode calculations inside a function that returns a single-cell result, you cannot readily predict whether you will need to enter it as a normal function or as an array function. This means that you must rely on detailed checking. Such checking would best involve both an example where the correct form produces the right answer and an example where the incorrect form.
5. A new function, SUMIFS, was introduced in Excel-2007. This function is aimed specifically at summing values that satisfy multiple conditions. You will be better off using SUMIFS where possible, but only if you are sure that nobody will want to use your spreadsheet on earlier.

It is widely believed that Excel cannot retrieve the contents of a cell from another spreadsheet. The method uses the SUMPRODUCT function, and the format for the argument(s) is as follows:

\[
\text{SUMPRODUCT( array1, array2, ..., arrayN )}
\]

This will retrieve the contents of cells that resolve to numbers or dates, but not text or logicals. I do not understand why this capability exists. It is not documented anywhere by Microsoft, and so it should be regarded as an unsupported feature. This means that whenever you use it you should test it thoroughly (even more thoroughly than you test all the other aspects of any spreadsheet you develop). It also means that you must re-test it whenever you downgrade.
An Excel guru named Harlan Grove has written a macro (a "user defined function") that will retrieve the contents of a cell in a closed spreadsheet, be it text or numeric. It is called PULL, and an Internet search should be able to locate this for you. However, it is quite slow to run.

With either method, you should explore the circumstances under which the retrieved cell that refers to a range whose size will vary automatically according to the amount of data to be located the top left hand corner of the range relative to the reference address (with positive number).

This will return a range that is 1 column wide, starts at cell C1, and extends downwards by a number of rows that equals the total number of non-empty cells in column C. Provided all the data you wish the range to embrace is at the top of the row (and with no embedded empty cells),

to allow for any amount of emptiness/nonemptiness in cells C1 through C5, we could use

This still requires that the data the range is to embrace be contiguous from C7 downwards.

Can we further extend this formula to allow for some embedded empty cells in the range below cell C6? I cannot think of a way, yet I feel sure that there will be one. Suggestions anyone?

If you have a large spreadsheet that takes quite a long time to recalculate, you might decide to be some to be etc.

* mula *cause *ore

* et

* e

* y a
Il the *y cells),

* e

* ls,

below *e?

* ide
to set Excel’s recalculation mode to “manual” so that you can control when to have Excel spend time on calculation. Normally when you do this, Excel tells you when a recalculation is required. However, if you use the OFFSET function in a spreadsheet whose recalculation mode is set to “manual,” Excel will not tell you when recalculation is required. (This behaviour is logical, but is complicated to explain. It stems from the fact that the OFFSET function is so powerful that it has the potential to return the value of any cell on the worksheet. This makes it almost impossible for Excel to maintain its usual “dependency tree,” the device it uses to keep track of what cells depend on what other cells.)

This behaviour is mentioned here because the usual way to create “dynamic arrays,” discussed above, involves the OFFSET function. Some of the functionality of the OFFSET function is mentioned here because the usual way to create “dynamic arrays,” discussed above, involves the OFFSET function.

Normally you do not need to know how Excel recalculates a spreadsheet—it just happens quickly and accurately. However, if you have a particularly large or complex spreadsheet its recalculation time might become an issue. Understanding how Excel performs its calculations is important for anyone creating large spreadsheets. Information on all aspects of Excel’s calculation engine is available. Anyone creating large spreadsheets should read this information.

Suppose you have set up a complicated sequence of calculations that takes a number of input values and produces an output value from them. You now want to see how changes in each of the input values will affect the output value. To do this, you can use Excel’s sensitivity analysis feature.

In the cell one row above the top of this range and one column to its right, put a formula that returns the value of the output variable. The easiest way to do this is usually to enter a formula that refers to the cell containing the output value. This formula should include a function that returns the value of the output variable. For example, if the output variable is in cell B2, you could enter the formula `=B2` in the cell above and one column to the right of the output variable. This formula would return the value of the output variable each time the spreadsheet is recalculated.
Hit the "OK" button. The table will be filled out with an array formula that calculates the results in the table are not static: they are dynamic and will change as you change other output cells can be included in the table by adding additional columns to the right, but there is a two-dimensional version of the Table feature, which allows you to investigate variations in two input variables simultaneously. One varies column-wise, the other row-wise.

Table:

<table>
<thead>
<tr>
<th></th>
<th>-2.29438656</th>
<th>0.77254249</th>
<th>1.46946313</th>
<th>2.02254249</th>
<th>2.37764129</th>
<th>2.5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>