Article from

The Modeling Platform

November 2017
Issue 6

Making Spreadsheets Great Again

By Bob Crompton

One of the most important innovations affecting actuarial work has been the electronic spreadsheet. Spreadsheets are now so ubiquitous that it is hard to realize that there was a time when actuaries did their work without them.

Spreadsheets have been used for almost every conceivable aspect of actuarial work, and actuaries have demonstrated considerable ingenuity, insight and skill in developing spreadsheet solutions in a fraction of the time needed to develop corresponding solutions through more formal channels.

But spreadsheets have no enforced structure or control. Much of what we see in spreadsheet-land is ad hoc and chaotic, put together in the heat of the moment. Many spreadsheets are large and unwieldy, difficult to control and subject to bouts of unexplainable behavior—much like a St. Bernard or a teenager just learning to drive.

This bad behavior results in recriminations and finger-pointing, with actuaries bearing the burden of blame for mistakes attributable to spreadsheets. Owners of actuarial spreadsheets need to be more proactive in ensuring the accuracy of spreadsheet results.

This is not an article on best practices; rather, it is an article on how to deal with worst practices and still end up with verifiable results. I discuss some of the ways we have been able to apply structure, identify anomalies, determine architecture and purpose, fix errors and generally make users of spreadsheets comfortable with results.

For spreadsheets of more than a trivial size, manual inspection is a fool's game. There is simply not enough time nor enough human concentration to effectively manually inspect a typical actuarial spreadsheet. This article, therefore, is limited to techniques that address the structure and form of spreadsheets rather than techniques that directly address spreadsheet results. Techniques addressing results are adequately discussed elsewhere. The techniques discussed in this article use the often-dormant power of spreadsheets to analyze themselves and make spreadsheets great again!

FORMULA LISTING

Since the heart and soul (and maybe the pancreas, too) of a spreadsheet are the formulas that are used to determine the results, it is important to have a sense of formulas used. Excel provides a special range of all the formulas in each worksheet. This range can be used to display information about the formulas. In Figure 1, basic formula metrics are displayed. While giving complete sample code to get such analytics is beyond the scope of this article, the sidebar "Notes and Observations About the Code" on pages 10–11 gives a starting point for developing these tools.

Figure 1
Basic Formula Metrics

Worksheet Name	Formula Count
Reserve Summary	497
Trad	49
Acquired	15,867
BOLI	156
Bank	103
Hybrid	4,325
Group Life	552
International	973
Total	**22,522**

It is clear from the formula count that the heavy lifting in this spreadsheet is performed in "Acquired" and "Hybrid." Any spreadsheet review or audit would naturally focus on these two worksheets. But there are more metrics we can create with the range of formulas. A simple listing of formulas, as shown in Figure 2, can reveal important characteristics of the spreadsheet.

This display shows the location of each formula, along with the formula presented as a string and the current value for each formula. We can glean several important observations just by scanning the formula list:

Figure 2
Sample Formula List

	Worksheet Name	Cell	Formula	Current Value
1	Reserve Summary	C9	='Trad'!D45	197,103,261
2	Reserve Summary	D9	='Trad'!E45	193,331,261
3	Reserve Summary	E9	='Trad'!F45	193,943,838
4	Reserve Summary	C10	='Bank'!N72-'Bank'!N83	1,327,193,384
5	Reserve Summary	D10	='Bank'!O72-'Bank'!O83	1,394,889,616
6	Reserve Summary	C11	='Acquired'!N15	41,374,518
7	Reserve Summary	D11	='Acquired'!O15	35,436,540
8	Reserve Summary	S9	=AVERAGE(OFFSET(OFFSET(Q9,0,0,1,-2),0,0,1,-S7))	180,336,812
9	Reserve Summary	S10	=AVERAGE(OFFSET(OFFSET(Q10,0,0,1,-2),0,0,1,-S7))	1,978,348,388
10	Trad	Q8	=16394*0.98*0.98*0.98*0.98*0.98*0.98*0.995	14,450
11	Trad	Q11	=38832*0.98*0.98*0.98*0.98*0.98*0.98*0.98*0.98*0.98*0.98*0.98*0.98*0.98*0.98*0.995	29,119
12	Trad	Q24	=P24+900000	22,500,000
13	Acquired	C26	=#REF!	#REF!
14	Acquired	D26	=#REF!	#REF!

- Some formulas are simple and need little or no review. For example, the first seven formulas listed in Figure 2 are simple references to other cells in the spreadsheet, with or without a simple arithmetic operation.
- Some formulas are complex and may need thorough review. The eighth and ninth formulas in Figure 2 fall into this category. One easy way to spot complexity is to *autofit* the column width of the Formula column, then scan to find the formulas that take up all the space.
- Sometimes formulas hide constants. The 10th, 11th and 12th formulas in Figure 2 are of this nature. These items may also require thorough review to determine if they are truly reflective of the intent of the spreadsheet.
- Sometimes there are broken formulas, such as the last two in Figure 2. Broken formulas typically—but not always—are formulas that are no longer used. But even if broken formulas are not material to spreadsheet results, they signal a casual attitude toward spreadsheet maintenance and should be investigated.

The formula listing provides an overview of spreadsheet complexity as well as where potential issues lie.

LISTING LINKS TO OTHER SPREADSHEETS

One of the common attributes of most actuarial spreadsheets is links to other spreadsheets. Links provide a quick and easy way to update data. Links mean that we can create a beautiful cascading waterfall of data transfer, moving massive amounts of data effortlessly downstream to all dependent spreadsheets.

But links can create an extensive update burden if the data sources of the links are updated and given new names every period. Such a burden also creates concern that link updates are performed accurately.

One way to address this concern is to list all of the spreadsheet's links and compare them to the prior period. Excel will provide a list of your spreadsheet links, but there is limited functionality included in the native listing. Instead, we can construct a routine that will compile all links in the spreadsheet.

In Figure 3, we show the current period link address versus the prior period link address. Our expectation is that most of the links will be updated to use a new file consistent with the convention of naming files with the period end date included in the file name. Because some of the links access the same data source as in the prior period, they are flagged with a contrasting cell fill.

Figure 3
Current Period Versus Prior Period Link Addresses

Current Period vs. Prior Period				
Worksheet	**Cell**	**Current Link**	**Prior Link**	**Status**
Tab 1	Q7	=+'C:\Users\Bob\AppData\Local\Microsoft\	=+'C:\Users\Bob\AppData\Local\Microsoft\	Same File
Tab 1	Q14	=+'C:\Users\Bob\AppData\Local\Microsoft\	=+'C:\Users\Bob\AppData\Local\Microsoft\	Different Date
Tab 1	Q21	=+'C:\Users\Bob\AppData\Local\Microsoft\	=+'C:\Users\Bob\AppData\Local\Microsoft\	Different Date
Tab 1	Q22	=+'C:\Users\Bob\AppData\Local\Microsoft\	=+'C:\Users\Bob\AppData\Local\Microsoft\	Different Date
Tab 1	Q26	=+'C:\2017\03\Wealth\Polysystems\Results\	=+'C:\2017\03\Wealth\Polysystems\Results\	Different Date
Tab 1	Q27	=+'C:\2017\03\Wealth\Polysystems\Results\	=+'C:\2017\03\Wealth\Polysystems\Results\	Different Date
Tab 1	Q33	=+'C:\Users\Bob\AppData\Local\Microsoft\	=+'C:\Users\Bob\AppData\Local\Microsoft\	Different Date
Tab 1	Q39	='C:\2017\03\Life\GWLA Reinsurance\[qry	='C:\2017\03\Life\GWLA Reinsurance\[qry	Same File
Tab 1	Q40	='C:\2017\03\Life\GWLA Reinsurance\[qry	='C:\2017\03\Life\GWLA Reinsurance\[qry	Same File
Tab 1	Q41	='C:\2017\03\Life\GWLA Reinsurance\[qry	='C:\2017\03\Life\GWLA Reinsurance\[qry	Same File
Tab 1	Q51	='C:\2017\03\Wealth\Polysystems\Results\	='C:\2017\03\Wealth\Polysystems\Results\	Same File

A word of warning about this technique—it does not scale well. (This is a common problem with Visual Basic for Applications, or "VBA"). We ran into one spreadsheet that had 20,000 links, and it took well over an hour to list the links with our VBA automation tool.

FORMULA LOCKDOWN

Formula lockdown is an approach we see in a number of end-user computing standards. While we understand the intent behind these types of standards, we believe that any standard that materially impedes workflow will not be successfully implemented. Spreadsheet users and owners, being more vested in the operation of the spreadsheets, will always be able to circumvent standards that make their lives more difficult.

One approach we have taken is to identify formulas that are nonvolatile—that is, we expect these formulas will not be updated except in unusual cases. We then apply formula locking selectively to only these nonvolatile formulas.

If nonvolatile formulas are identifiable in some way, such as special formatting, it is possible to programmatically perform the selective lockdown.

Lockdown can be implemented without a password. This is usually preferable to lockdowns that have passwords. If the proverbial milk truck runs over the spreadsheet owner and no one can find where she wrote down the password, the spreadsheet may become unusable.

Lockdowns with no passwords can also be unlocked programmatically without passwords. This means that unlocking for necessary changes can be accomplished with minimal interruption of workflow.

Such an approach will not stop deliberate maliciousness, nor will it stop determined stupidity. However, the implementation of formula locking will mean that any changes require intentionality on the part of the one updating the spreadsheet, so at least some casual errors would be prevented.

HIGHLIGHTING CONSTANTS (AKA HARD-CODED NUMBERS)

For some reason, almost all spreadsheets of any size have constants. Perhaps spreadsheet gnomes sprinkle these throughout the spreadsheet while the owner is asleep.

We have seen constants put into the middle of a column or row of formulas as test values, then not changed. We have seen constants put in as manual adjustments, then not changed. We have seen constants put in as true-up values, then not changed. Excel allows you to be as boneheaded as you truly are.

Figure 4
Highlighting Cells With Constants

	6/30/16	9/30/16	12/31/16	3/31/17
Traditional	61,610,504	62,220,911	62,830,632	63,448,005
UL	152,868,104	159,319,432	162,543,593	176,438,180
VUL	318,894	333,619	383,290	492,108
Total	**214,797,502**	**221,873,962**	**225,757,515**	**240,378,293**

However, Excel also provides a special range of all the constants in each worksheet, so it is a straightforward exercise to programmatically highlight cells with constants. In Figure 4, constant cells are highlighted with gray fill so that they contrast with cells containing formulas.

But we can do even better than this! We can write a routine (macro) that allows the spreadsheet user to select a range of columns or rows and inspect all constants in the range that exceed a specified threshold. This improves efficiency, especially when the spreadsheet user knows where the critical cells are.

FINDING CONSTANTS HIDDEN IN FORMULA CELLS

As the formula-listing technique demonstrated, sometimes constants are included in formula cells. Because Excel does not have a native method of identifying this type of formula, such cells are not easy to find. In fact, this is the second most difficult technique discussed in this article. We had to jump through several hoops in order to automate locating these cells.

However, once the automation is in place, we can treat these cells like constant cells. We can apply contrasting formatting if we wish. We can also incorporate a routine that will allow the spreadsheet user to inspect all such cells and update them as required.

FINDING BROKEN FORMULAS

Like constants, but unlike constants hidden in formulas, broken formulas can be discovered using native Excel capabilities. This means that anything we implement for constants we can implement with equal facility for broken formulas.

We can put special formatting on broken formulas to highlight their locations. We can implement routines that will inspect all broken formulas. And we can implement routines to inspect broken formula dependencies.

VALIDATION CONTROLS FOR MANUAL ENTRIES

A typical requirement for many end-user computing standards is some form of validation applied to keypunched data entries in a spreadsheet. Although this form of data entry is not widespread in most actuarial spreadsheets, we have heard of instances of some outrageous spreadsheet results generated from such things as the entry of "No" rather than "N" or vice versa.

Once again we can use Excel's native abilities to help out. Figure 5 shows a listing of all constants with dependencies. We can use this chart to determine if any of our manual entries need validation controls. By examining the form of the dependency formulas, we can construct appropriate validation limits. Manual entries without dependencies need no validation controls.

This was the most difficult of all the routines discussed in this article. However, we followed the process used by astute programmers: we searched the internet to see if anyone had done this before. A BIG THANKS to the obsessive souls who take the time to put this sort of thing out on the web for the public.[1]

A final note on this routine: The Range.Dependents property shows dependencies only on the same worksheet. In order to capture dependencies existing on other worksheets, the ActiveCell.ShowDependents method must be used.

FINDING DEPENDENT WORKBOOKS

With the extensive use of spreadsheet links, an additional risk that spreadsheet users face is that changes to "upstream" workbooks will break the links in a "downstream" workbook, or—even worse—cause the links to access unintended data.

There is no direct way to detect downstream connections from a workbook. However, if the directory or directories of all (or nearly all) potential downstream connections can be enumerated, it is possible to construct a routine that will check for downstream dependencies.

This approach involves opening all workbooks in the indicated directories and checking for links to the upstream file. Since the search must look at each formula in each workbook in the search directories, this can be a time-consuming process just for one upstream file. Perhaps this approach should be considered a just-in-time process whenever spreadsheet restructuring is undertaken.

Figure 5
Sample List of Constants With Dependencies

Constant Worksheet	Constant Cell	Constant Value	Dependent Worksheet	Dependent Cell(s) →						
BOLI/COLI	D1	26504	BOLI/COLI	D4						
BOLI/COLI	O1	26504	BOLI/COLI	O33						
Acquired	AB7	spwl	Acquired	AA96	AA97	AA98	AA100			
Acquired	AC7	spwl	Acquired	AA82	AA83	AA84	AA86	AA89		
Acquired	AB8	term	Acquired	AA96	AA97	AA98	AA100			
Acquired	AC8	term	Acquired	AA82	AA83	AA84	AA86	AA89		
Acquired	AB9	term	Acquired	AA96	AA97	AA98	AA100			
Acquired	AC9	term	Acquired	AA82	AA83	AA84	AA86	AA89		
Summary	Z13	31488.33	Acquired	AA13	AA29	AA82	AA83	AA84	AA96	AA97
Summary	AA22	-277000	Acquired	AA29	AA82	AA83	AA84	AA96	AA97	AA98
Summary	AA23	1300000	Acquired	AA29	AA82	AA83	AA84	AA96	AA97	AA98

CONCLUSION

Many actuarial spreadsheets are created using worst practices, resulting in spreadsheets that do not always behave as intended and are difficult to control. However, several techniques allow us to understand the structure and complexity of these spreadsheets and spot areas where mistakes are likely to occur. We can then address the problems so that we have confidence in the results.

Much of what is discussed in this article grew out of having to perform model validation on spreadsheet models. We have implemented all of these techniques in one form or another. Most of them are general enough that they can be applied to almost all spreadsheets. One or two—notably spotting exceptions in spreadsheet link updates—are highly dependent on spreadsheet context. ■

Bob Crompton, FSA, MAAA, is a vice president of Actuarial Resources Corporation of Georgia, located in Alpharetta, Ga. He can be reached at *bob.crompton@arcga.com*.

ENDNOTE

1 Some of the sites that I found to be helpful are listed here. Although there are many other sites, these are the ones I wound up using the most. The Microsoft Developers Network site is indispensable (*https://msdn.microsoft.com/vba/vba-excel*). Stack Overflow has many good worked examples to questions (*https://stackoverflow.com*). Ozgrid is another site with worked responses to questions (*www.ozgrid.com*). Code Project has articles as well as answers to questions (*www.codeproject.com*).

Notes and Observations About the Code

I am not an expert on VBA; however, my Google-fu is strong! This may be even more important than being an expert, since nearly everything you could want, or even imagine, in Excel macros has been done and posted on the internet. The ability to locate specimen macros on the internet is your best bet to becoming proficient in VBA.

Specimen macros not only give insight into techniques that might otherwise take a long time to track down, but they also show good style. After reviewing a number of macros, you learn that good code is succinct and easier to read than poor code.

One of the benefits to using macros is that they are self-documenting in the sense that they fully describe the actions and calculations they perform.

THE FORMULA-LISTING MACRO

The formula-listing macro is the first of these tools that was assembled, and for that reason it has evolved more than the others. The key to the formula listing is getting the range object of all formulas in a worksheet. The code for this is:

```
Set FormulaCells = Range("A1"). _
SpecialCells(xlCellTypeFormulas, 23)
```

Excel contains a number of special ranges. You can get a sense of these by hitting the F5 key while in Excel. This brings up the Go-To dialogue. If you click the button labeled "Special...," you will see all of the special ranges that Excel can easily create for you.

Once this range is created, it is easy to loop through each cell and extract the pertinent information. For example:

.Cells.Count contains the formula count

Use a "For Each" loop through the range as follows:

```
For Each fCell In FormulaCells
```

Then the appropriate attributes can be accessed. For example,

- *fCell.Address* contains the cell address
- *fCell.Formula* contains the cell formula
- *fCell.Value* contains the current value to which the formula evaluates

Because most actuarial spreadsheets contain multiple worksheets, it is important to loop through each of the worksheets. I typically use a *For Each* loop like the following:

```
For Each ws In ActiveWorkbook.Worksheets
  If ws.name <> TabName Then
  ws.Activate
```

The worksheet named TabName is the worksheet I add to contain the formula listing, so I exclude this from the processing.

ALTERNATE SORTING OF RESULTS

The results in the formula listing are not arranged by column. The *natural* order seems to me to be one in which the results start from column A, then proceed column-wise to the right-most column, since this is how spreadsheet logic typically proceeds. But sorting by cell doesn't work since cell AA1000 precedes cell A1.

In order to address this, I have written user-defined functions that allow me to sort results by column. There is no end to the possibilities!

HIDDEN WORKSHEETS AND COLUMNS

One of my serendipitous discoveries was that when using a *For Each* loop through each worksheet, it operates on hidden worksheets as well as visible worksheets. Sometimes in spreadsheet review, it is easy to forget that there may be hidden worksheets. Running the formula listing reminds me of this whenever the formula count tableau shows names of worksheets that I can't see.

Hidden columns are likewise displayed whenever we list any of the special ranges. Although hidden columns are usually more apparent than hidden worksheets, it is still convenient to have listings that do not require un-hiding columns or rows.

ERROR TRAPPING

Because VBA contains only rudimentary error-trapping abilities, the macro may contain some odd "GoTo" statements. For example, in the formula-listing macro, if there are no formulas in a worksheet, an error is generated when attempting to set *FormulaCells*. In this case, error trapping

sends the code to the part that lists the number of formulas (zero in this case), but skips the attempt to list the formulas themselves, since there are none.

THE CONSTANTS SEARCH-AND-REPLACE MACRO

The constants search-and-replace macro and the formula-listing macro are very similar, but a few additional details may prove useful when using the constants search-and-replace routine.

First we create a range of constants:

```
Set ConstantCells = Range("A1"). _
SpecialCells(xlCellTypeConstants, 23)
```

Then we allow the user to select some arbitrary range for the search-and-replace operation:

```
Set SearchRange = Application.InputBox( _
"Click Rows or Columns", _
"SEARCH RANGE SELECTION",,,,,, 8)
```

We then reduce our review-and-replace operation to the intersection of the ConstantCells range and the SearchRange range:

```
Set SearchCells = Application. _
Intersect(SearchRange, ConstantCells)
```

The *Intersect* function is one of the most powerful and useful functions when dealing with Excel ranges.

WEAPONS OF MASS IMPLEMENTATION

In certain situations you may want the routines outlined here installed on a number of spreadsheets. Manually copying code from one spreadsheet to another quickly becomes tedious. It also creates another point of potential error. Any tedious manual process is a process that is ripe for automation. You can construct a macro that will perform the implementations. It is a straightforward process to programmatically select a directory and install the VBA code in all (or only some) of the spreadsheets in the directory. If you like buttons for your macros, you can put all of your buttons on a single worksheet and copy it to all of the target spreadsheets. You can then programmatically connect the buttons to the macros.

DEFAULT ACTIONS FOR USER INPUT

When we originally developed this macro, the default action was to enter the threshold value as the new value for cell. One client told us that this resulted in unintentional changes in the spreadsheet. We changed the macro so that updating the spreadsheet with new values required intentional clicking.

Users are the ultimate arbiters of usability!

That's it for macros. And now, as The Dude would say, let's go bowling.