**Building a Basic, Understandable Sudoku Solver Using Excel Iterative Calculation – Part 1/2**

*Today’s author, Charlie Ellis, a Program Manager on the Excel team, shares a spreadsheet he built in Excel for solving Sudoku puzzles.  The spreadsheet can be found in the attachments at the bottom of this post.*

For those of you who don’t already know, Sudoku is a type of logic puzzle (that I was completely addicted to about three years ago) that requires you to place the numbers 1-9 into a grid obeying certain rules ([lots more information on Sudoku is available on the web](http://search.live.com/results.aspx?q=how+to+play+sudoku)).

A while back, a fellow PM on the Excel team, Dan Cory, wrote a spreadsheet for solving Sudoku puzzles using Excel formulas and made it available on Office Online ([here](http://office.microsoft.com/en-us/templates/TC100809721033.aspx)). Dan’s spreadsheet was great in that, unlike many of the Sudoku solving spreadsheets out there, it didn’t use any VBA or other scripting to do the work of solving the puzzles, and relied instead on the iterative calculation feature of Excel. It’s quite cool and has been a popular download, but one thing about the spreadsheet that I wanted to see if I couldn’t improve upon was just how complicated it is. In fact, Dan made every single cell its own different formula, and he ended up having to use VBA to create the formulas because maintaining and debugging it without VBA to write all those different formulas in an automated way was impossible.

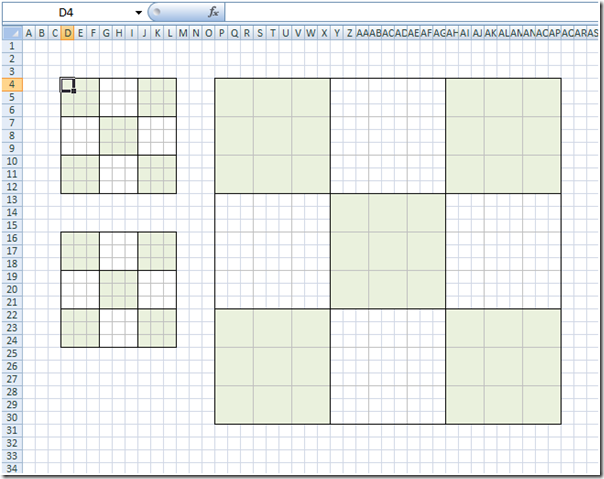
As soon as I saw Dan’s spreadsheet, I wanted to make my own version of a Sudoku solver that not only used only formulas, but also one where the formulas were relatively understandable and there were a small number of distinct formulas. It turned out to not be that tough to build, but I think I learned a fair amount trying different approaches to the problems of making an iterative model like this one perform well and at the same time be reasonably maintainable and understandable. I think it might even have turned up a reasonably useful way at looking at abstraction within formulas given the Excel formula language. I’ve always wanted to blog about the process of creating this spreadsheet and about how iterative formulas work to show the power of Excel’s formula language, because it illustrates the usefulness of circular references and iterative calculation, and because I just think it’s an incredible amount of fun so here goes. Lots of people have created more powerful solvers, many as spreadsheets, some using just formulas, but I wanted to try to explain how you can go about creating a solver and hopefully share some formula tricks that people find useful.

**Pre-reqs**

Creating a spreadsheet for solving a Sudoku isn’t entry-level spreadsheeting. In addition to being pretty good with formulas, you’ll need to understand the concept of iteration. Chris Rae did a great job of explaining the topic in his earlier post on [Iteration & Conway’s Game of Life](http://blogs.msdn.com/excel/archive/2007/11/02/iteration-conway-s-game-of-life.aspx), so I’m not going to repeat that, and I’ll simply assume you already understand iteration. Second, we’re going to make extremely heavy use of named ranges, and for the stuff I’m doing, the new name manager is very helpful (see [Formula building improvements Part 4: Defined Names](http://blogs.msdn.com/excel/archive/2005/10/21/483661.aspx) for some information about this) and I’m going to assume working knowledge of it and of named ranges generally (though I’m going to show some tricks which may be new to even experienced formula users). Finally, you’ll need to at least be familiar with array notation in Excel.

**Setting up the boards**

For those I haven’t lost already, I’m going to start by creating a series of boards very much like the ones that Dan Cory used: one 9×9 board for my input, one 9×9 board for the solution, and a 27×27 board for the possible values in each box. I do this by changing the row height, column width, font, and zoom such that all the cells are small squares and then applying borders and fills to get the following:

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_13F2D/image_2.png)

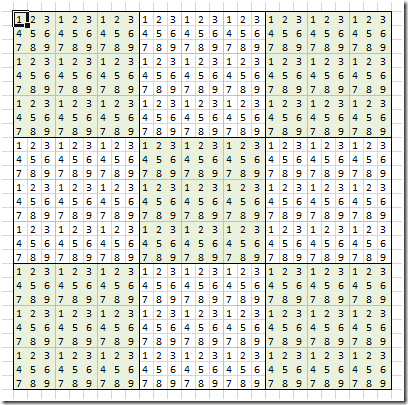
The input and solution boards are reasonably straightforward (the input board is the one in the top left where you’ll type in a puzzle to be solved, the solution board is where the correct answer hopefully shows up). The board with possible values, which I’ll call the valid values board, is a bit trickier. It is 27×27 because each box in the input and solution boards is represented by a 3×3 set of cells in the valid values board. Each of these nine cells represents whether one of the numbers 1-9 is still in the running to be the actual value for the corresponding box of the solution board and the set of possible values for a given cell in the input/solution cell is the set of all the numbers in a single 3×3 “big cell” that are not blank. If it isn’t already, the purpose/use of this board should become clear later. For now, let’s fill in all the possible values from one to nine in each of these big cells.

**Filling in the valid value board**

We want to do this by creating a single formula that will fill in the various numbers 1-9 based on which row and column the formula sits in, and then we’ll later add logic to blank out the numbers that aren’t valid. This formula is a little more complicated than the average spreadsheet formula, so I’ll first give the whole formula and then break it down. This looks like the following:

= MOD(COLUMN(A1)-1,3) + 1 + MOD(ROW(A1)-1,3)\*3

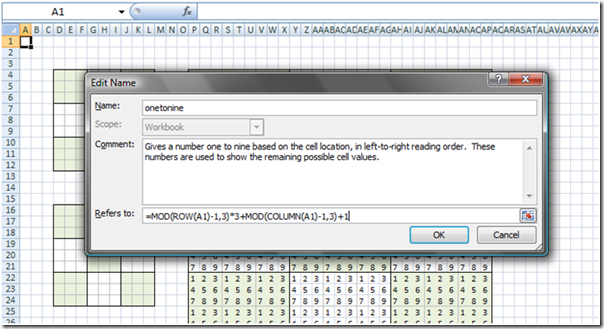
When this is entered into the top-left cell of the valid values board and then filled into the entire valid values board, it gives the following results:

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_13F2D/image_4.png)

Note that you’ll want to do the filling in with either Paste Special | Formulas or CTRL-Enter because otherwise you’ll mess up all the pretty formatting.

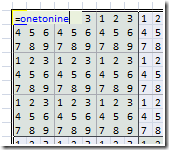
Breaking this formula down, ROW and COLUMN return (duh) the row or column of the reference passed to them as a number. Passing these functions A1, as in this formula, means they’ll give us a number that starts at one and goes up. The first part of the formula uses the modulus function to transform the column numbers given by COLUMN into the numbers 0-2, and then adds one to get 1-3. To this we add a 0, 3, or 6, depending on the row number by using the modulo function on the result of the ROW function.

Next, because that’s a bit of a gnarly formula to have sitting around, and we’re going to have to use it all over the place, we’re going to take this formula and move it out of the cell and into a named range. This allows us to abstract away all of the logic for this formula into a single, understandable name. For lack of a better name, I’m going to call it “onetonine” and it will have the same exact formula we just created. Because the context for the relative references (i.e. what they take as being the current cell) is determined by what cell you’re in when you create the named range, it’s critical that you start off by selecting cell A1, then create the new named range, so that your formula works everywhere within the sheet.

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_13F2D/image_6.png)

This is also why we allow gutters of three rows and three columns around all the boards.

Now we can take our new name and test it out in the board, like so:

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_13F2D/image_8.png)

Here CTRL+Enter is by far the easiest way to set the formula for all the cells in the valid values board. First select the whole board, then type in the formula, and instead of pressing Enter, just hit CTRL+Enter to fill the formula you just typed into all the cells (without messing up their formatting).

**Setting up the solution board**

We’re going to want to base what valid values are left for a given box on what our current solution looks like (as opposed to the input), but in order to do that, we need something in the solution board. To begin with, at least, the solution will definitely contain all of the numbers in boxes from the input board. Let’s start off by doing this in the simplest possible way, while catching the case of blanks. In the solution board, let’s make the cells there all simply equal the corresponding cell in the input board using relative references unless the input cell is blank. The absolute easiest way to do this is with the following formula (shown in the form in which it would be entered into cell D16):

=IF(D4,D4,””)

Again, use CTRL+Enter to fill this into the appropriate cells. Now that we have the base thing working, let’s make it more re-usable and meaningful by using named ranges.

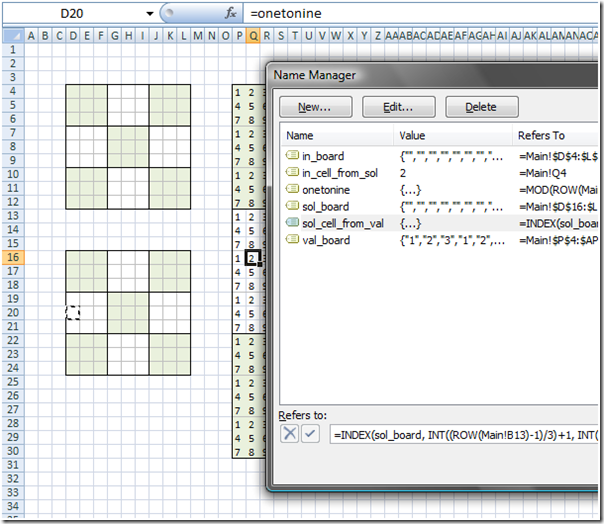
As we did with the name *onetonine*, let’s abstract the concept of referring to the correct input cell from any cell in the solution board and turn that into a name. We’ll need to do something similar for all the boards at some point, so we’ll start by making named ranges for each of the boards (I chose *in\_board*,*sol\_board*, and*val\_board*)and then a name to go from the solution board to the input board (*in\_cell\_from\_sol*) which is simply =Main!D4, then use this to change the formula to be =IF(in\_cell\_from\_sol, in\_cell\_from\_sol, “”). Note that this needs to be input from D16.

OK, so far we just made our formula longer, but trust me, this concept becomes a life saver. Doing the same for valid value cells from solution board cells is only a bit trickier. The name *sol\_cell\_from\_val* is:

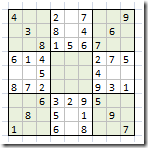
=INDEX(sol\_board, INT((ROW(Main!A1)-1)/3)+1, INT((COLUMN(Main!A1)-1)/3)+1)

This must be created from cell P4. This formula uses ROW and COLUMN together with the division operator and INT to convert from the coordinates of the current cell in the 27×27 board to their coordinates in a 9×9 board, then uses INDEX to get the cell out of the *sol\_board* corresponding to those coordinates.

A neat way of testing this formula is to click into the “Refers to” box of the name manager from different cells in the valid values board. Depending on what cell you’re in you’ll see “dancing ants” (a moving highlight) for a different cell – hopefully the corresponding cell in the solution board.

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_13F2D/image_12.png)

Now that we have some basics, let’s put in an actual puzzle and see about getting the inputs to propagate to the solution board and the valid value board. Here’s the puzzle we’ll use:

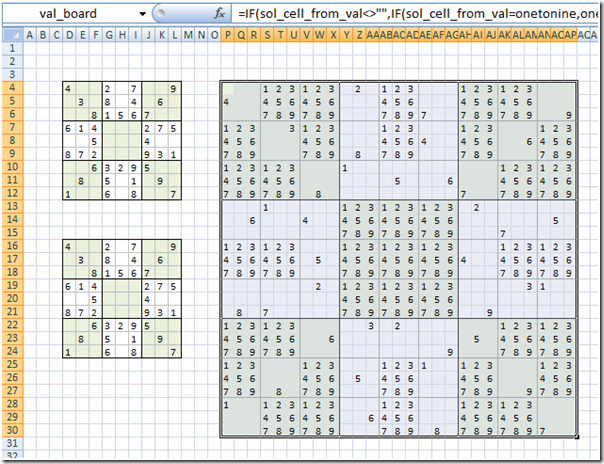
[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_13F2D/image_14.png)

After entering it, the solution board should look like the input board. To make the valid value board work, we use this formula for all valid board cells:

=IF(sol\_cell\_from\_val<>””,IF(sol\_cell\_from\_val=onetonine, onetonine,””), onetonine)

This means that the current cell is blanked out if a value exists in the solution cell and that value isn’t the current*onetonine* value.

This should give you:

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_13F2D/image_16.png)

Now we’re ready to do the stuff that will actually help come up with solutions based on the rules and strategies of Sudoku.

**Checking for a number in the rows of the solution board**

The main rule of Sudoku is that you can’t have two of the same number in any row, column, or 3×3 big box. We will start by adding the rule that there can’t be more than one of a number in any row and then working in columns and big boxes. For example, in the second big cell in the first row, none of the numbers 4, 2, 7 or 9 are possible as a result of this rule. We can do this by turning blank any cells in the valid values board for which 1) a solution for the box doesn’t exist (which is precisely when we get to the final *onetonine*in the formula for valid value cells) and 2) the row of the solution board contains the number equal to the current value of *onetonine*. Note that condition #1 is precisely where the last *onetonine*shows up (i.e. no solution exists for the current big cell), so all we have to do is put the logic for #2 there. This logic can be expressed as:

=IF(COUNTIF(sol\_row\_from\_val, onetonine)>0, “”, onetonine)

Where *sol\_row\_from\_val* is:

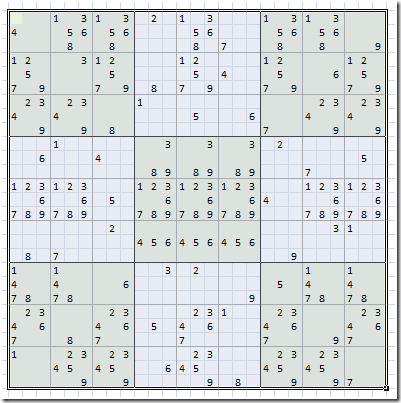
=INDEX(sol\_board, INT((ROW(Main!A1)-1)/3)+1, 0)

Again, this must be entered from P4.

So combining these we get:

=IF(sol\_cell\_from\_val<>””,IF(sol\_cell\_from\_val=onetonine, onetonine,””), IF(COUNTIF(sol\_row\_from\_val, onetonine)>0, “”,onetonine))

Which, while not simple, is at least understandable and gives you a valid values board that looks like this:

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_13F2D/image_18.png)

**Extending to columns and (3×3) big boxes**

When we go to add the rules for “no two of the same number in a column” and “no two of the same number in a big box” in this same way, we will run into two problems: 1. you can’t create *sol\_bigbox\_from\_val* directly using INDEX because INDEX only returns a cell, row, or column from a range or the whole range and 2. it will start to get unwieldy to have all three of the COUNTIFs OR’d together at the end of this formula.

To solve the first problem, you can use OFFSET – as you could use OFFSET to create any of the other references here – but because OFFSET is volatile this will lead to performance problems down the road. A better solution is to take the union of two references that you get from INDEX (using the union operator in Excel – the colon) in order to make a 3×3 range. This gives us a *sol\_bigbox\_from\_val* with the following formula (entered from P4):

=INDEX(sol\_board, INT((ROW(Main!A1)-1)/9)\*3+1, INT((COLUMN(Main!A1)-1)/9)\*3+1):INDEX(sol\_board, INT((ROW(Main!A1)-1)/9)\*3+3, INT((COLUMN(Main!A1)-1)/9)\*3+3)

By now we can pick this formula apart more easily. The INT, ROW, division part says that for every nine rows you move in the valid value board, move down by a block of three rows in the solution board. There’s a similar expression for columns that accomplishes much the same thing moving across. The second reference is precisely the first reference, but offset two rows down and two columns across, giving you a 3×3 box.

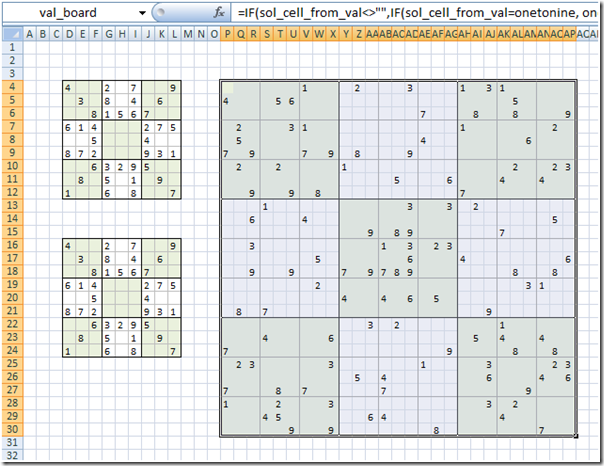
Now that we have this, we could write one big formula that covers whether the current *onetonine* value already exists in any of the row, column or big box, but let’s use abstraction again here to keep the fundamental formula of the valid value board more simple. Instead of putting it directly into the formula, let’s invent a new name called *solution\_in\_rcb*for “does there exist a solution cell with my number in any of the row, column, or bigbox?” This name only ever has to return true or false (doing the test part of the condition #2 does above) and despite not being short, is actually really simple to write:

=OR(COUNTIF(sol\_row\_from\_val, onetonine)>0, COUNTIF(sol\_col\_from\_val, onetonine)>0, COUNTIF(sol\_bigbox\_from\_val, onetonine)>0)

Taking advantage of this new name makes our new formula for valid value cells:

=IF(sol\_cell\_from\_val<>””,IF(sol\_cell\_from\_val=onetonine, onetonine,””), IF(solution\_in\_rcb, “”,onetonine))

Which is not only shorter than this formula had been and much more understandable, it also results in some clear places where there’s only one possible solution:

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_13F2D/image_20.png)

So we can eyeball some solutions, but the trick now is to feed those into the solution board. This is where iteration comes in. Next time we’ll use iteration and a few more formula tricks to solve some Sudokus.

**Edit:** Updated the sol\_bigbox\_from\_val formula to reflect what it looks like when entered from the starting cell in P4.  Also clarified in a couple other places that the starting cell should be P4.

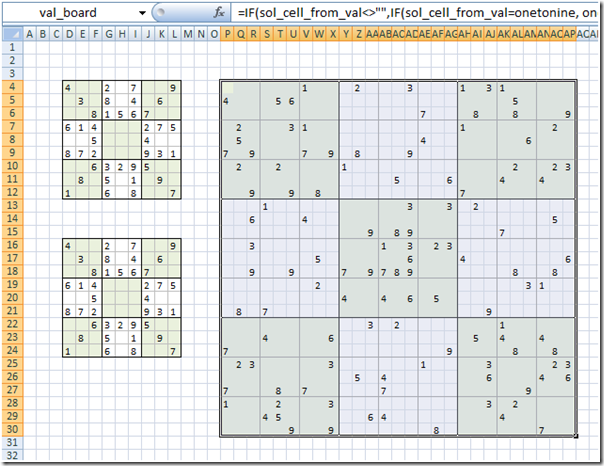
[Products](https://products.office.com/)

**Building a Basic, Understandable Sudoku Solver Using Excel Iterative Calculation – Part 2/2**

In my [previous post](http://blogs.msdn.com/excel/archive/2008/09/29/building-a-basic-understandable-sudoku-solver-using-excel-iterative-calculation-part-1-2.aspx), I walked through a number of formulas for setting up the valid values and solution board. In this post we’ll cover using iteration and other formula tricks to help solve the puzzle.

**Using valid values to drive solutions**

Looking at cells P22:R24 below, we can clearly see that the only solution here is 7:

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_14798/image_2.png)

Let’s reflect this in the solution board.

We’re going to modify the “else” argument of the IF function here to make it check to see if the *val\_bigcell\_from\_sol* has only a single answer, and if so, return that. First we need to make *val\_bigcell\_from\_sol*:

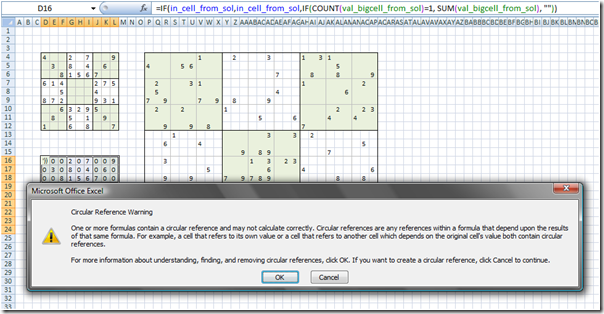
=INDEX(val\_board, ROW(Main!A1)\*3-2, COLUMN(Main!A1)\*3-2):INDEX(val\_board, ROW(Main!A1)\*3, COLUMN(Main!A1)\*3)

Making sure to enter this from the first cell in the solution board (cell D16).  Next we use this to make the else clause:

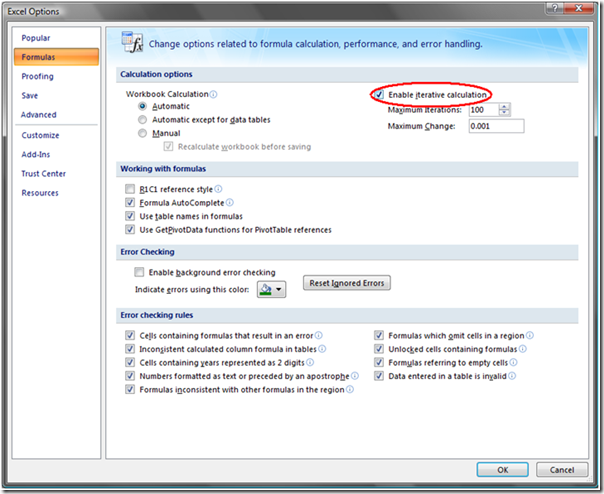
=IF(in\_cell\_from\_sol,in\_cell\_from\_sol,IF(COUNT(val\_bigcell\_from\_sol)=1, SUM(val\_bigcell\_from\_sol), “”))

Note the trick here to get a value from a 3×3 big cell; we can’t just get its value, but since we know there’s only once cell with a value, we can just use SUM to get the value of the one cell within the big cell that has a value.

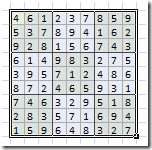
And then we enter it and. whoops, we get a big scary warning:

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_14798/image_4.png)

Click Cancel, and then go to the tools options and make sure that “Enable iterative calculation” is checked:

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_14798/image_6.png)

Now, when you go back to the spreadsheet, it’s worked, and we’ve completely solved this (really easy) Sudoku:

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_14798/image_8.png)

To see what it’s doing, you can dial down the maximum iterations to 1 and add a reset to both the solution board and the valid values board. The way I like to do this is with a cell that tracks state, and either has a string that either says “reset” or “solve”. Adding this to a cell (I chose D21) and naming it *state*, you can modify the valid values board formula and the solution board formula to be:

=IF(state=”reset”,onetonine,IF(sol\_cell\_from\_val<>””,IF(sol\_cell\_from\_val=onetonine, onetonine,””), IF(solution\_in\_rcb, “”,onetonine)))

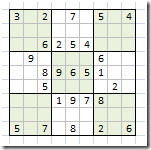
And

=IF(in\_cell\_from\_sol, in\_cell\_from\_sol, IF(state=”Reset”, “”, IF(COUNT(val\_bigcell\_from\_sol)=1, SUM(val\_bigcell\_from\_sol), “”)))

Respectively.

**Finding the only cell in a row that can take a given value**

Let’s take a slightly tougher puzzle now, and see how we can extend this very rudimentary solver.

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_14798/image_10.png)

When you try to solve this puzzle with our existing solver, it gets some answers, but then it bogs down. To solve this puzzle, we need to make use of another rule of solving Sudoku puzzles which is that there must be at least one of each number in every row, column and big box. This rule, which is almost the first rule flipped around helps you get to a solution when there is only one possible place in a whole row, column or big box that a particular number can go.

Before we get to how to actually check for this, let’s talk about the right place to put this logic. Because it is helping us to find a solution as opposed to eliminate a possibility, it should go in the solution board. And because it covers a case that is the non-reset-state case where a solution can’t be found (and the empty string is returned), the right place is where that normally would be. Furthermore, we have to both find a solution and return that solution, so it makes sense to collapse both of those into one name instead of testing for a solution then doing the same work we did to find the solution in order to return it. That means we’re going to change our formula to something like:

=IF(in\_cell\_from\_sol, in\_cell\_from\_sol, IF(state=”Reset”, “”,IF(COUNT(val\_bigcell\_from\_sol)=1, SUM(val\_bigcell\_from\_sol), single\_sol\_in\_rcb)))

Where *single\_sol\_in\_rcb* is either a solution, if one exists, or the empty string.

To implement this, we’ll need *val\_bigcell\_from\_sol*, but also *val\_bigrow\_from\_sol*,*val\_bigcol\_from\_sol*,and*val\_bigbox\_from\_sol*. These are all fairly straightforward extensions on the concepts from *sol\_row\_from\_val* and*val\_bigrow\_from\_sol*, so I won’t go into detail here, but here’s the formula for *val\_bigbox\_from\_sol* as entered/viewed from cell D16:

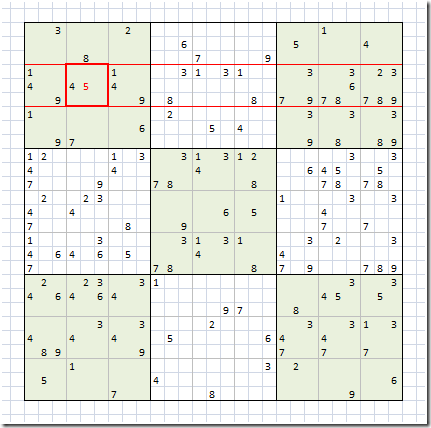
=INDEX(val\_board, INT((ROW(Main!A1)- 1) / 3) \* 9 + 1, INT((COLUMN(Main!A1)- 1 ) / 3)\*9+1):INDEX(val\_board, INT((ROW(Main!A1)-1) / 3) \* 9 + 9, INT((COLUMN(Main!A1)-1) / 3) \* 9 + 9 )

For now, we’re going to start out just checking the rows for cases where there’s only one cell that can contain a given value. Thus we’ll make *single\_sol\_in\_rcb* just be:

=single\_sol\_in\_row

And then eventually make it so that we can detect a single solution in any of rows, columns, or boxes. BTW, it’s fine to create names that make use of these as-yet-undefined names, but if you put these into the sheet, Excel annoyingly resizes your columns to accommodate the #NAME errors that inevitably result.

Here’s what this looks like on the valid value board:

[](https://blogs.msdn.com/blogfiles/excel/WindowsLiveWriter/BuildingaBasicUnderstandableSudokuSolver_14798/image_12.png)

Here the five in the big cell highlighted in red is the only five in the entire row, so the corresponding solution cell must be five. We can break this down into two pieces:

1. Checking that a value exists within the valid values board big cell corresponding to the current solution cell
2. Checking that there’s only one of that value in the valid value board big row corresponding to the current solution cell

The most efficient way I’ve gotten to work, is to do these checks and comparisons by creating an array of the values in the current big cell and multiplying that array by which of those values only occur once in a given row. Here’s the formula for this:

=MAX((COUNTIF(val\_bigrow\_from\_sol,array1to9)=1)\*(COUNTIF(val\_bigcell\_from\_sol,array1to9)=1)\*array1to9)

Where *array1to9* is the array {1;2;3;4;5;6;7;8;9}, which is easiest to get using =ROW($A$1:$A$9). The first COUNTIF and equality operator check whether, for each number 1-9, there’s only one of the number in the row. In the above example, this becomes {0;1;0;0;1;1;0;0;0} because each of 2, 5, and 6 occur only once in the highlighted row. The second COUNTIF and equality operator check which values are present in the current cell. In the case of the highlighted cell, this is {0;0;0;1;1;0;0;0;0}. Multiplying them together gets the intersection of those two conditions ({0;0;0;0;1;0;0;0;0}), and multiplying this by *array1to9*gives you {0;0;0;0;5;0;0;0;0}, which can be reduced to just the desired value by taking the MAX.

Then, in *single\_sol\_in\_rcb*, in order to not have to call *single\_sol\_in\_row* more than once to both return an answer and turn the 0 condition into an empty string, you can use the CHOOSE function like so:

=CHOOSE(single\_sol\_in\_row+1, “”, 1, 2, 3, 4, 5, 6, 7, 8, 9)

This is enough to solve the second sample puzzle, but extending this to columns and boxes is actually really easy at this point. All you have to do is make a *single\_sol\_in\_col*and*single\_sol\_in\_box*, which are identical to *single\_sol\_in\_row* except for the first named range. Then you have to adjust *single\_sol\_in\_rcb* to include the solutions for column and box like so:

=CHOOSE(MAX(single\_sol\_in\_row, single\_sol\_in\_col, single\_sol\_in\_box)+1, “”, 1, 2, 3, 4, 5, 6, 7, 8, 9)

Well, that’s it. The actual workbook itself is attached to my [original post](http://blogs.msdn.com/excel/archive/2008/09/29/building-a-basic-understandable-sudoku-solver-using-excel-iterative-calculation-part-1-2.aspx). I think it’s neat in that there are only really three different formulas on the actual sheet and another 18 in the names. If you think about it as lines of code, there aren’t a lot of places outside of obfuscated code contests where you get 21 lines of code to do as much complicated stuff. I think you’ll have to be the judge of how clear the formulas are, but by using the commenting on names and further cleaning up naming conventions, I think this could be very understandable. Certainly there’s a lot more I could walk through in a future post – primarily how to add in more complicated strategies that I’ve gotten to work in this framework such as pairs, triples, hidden pairs, hidden triples, and box-and-line, but also debugging strategies, pitfalls for performance or maintainability, etc. – but that’s the basic idea.