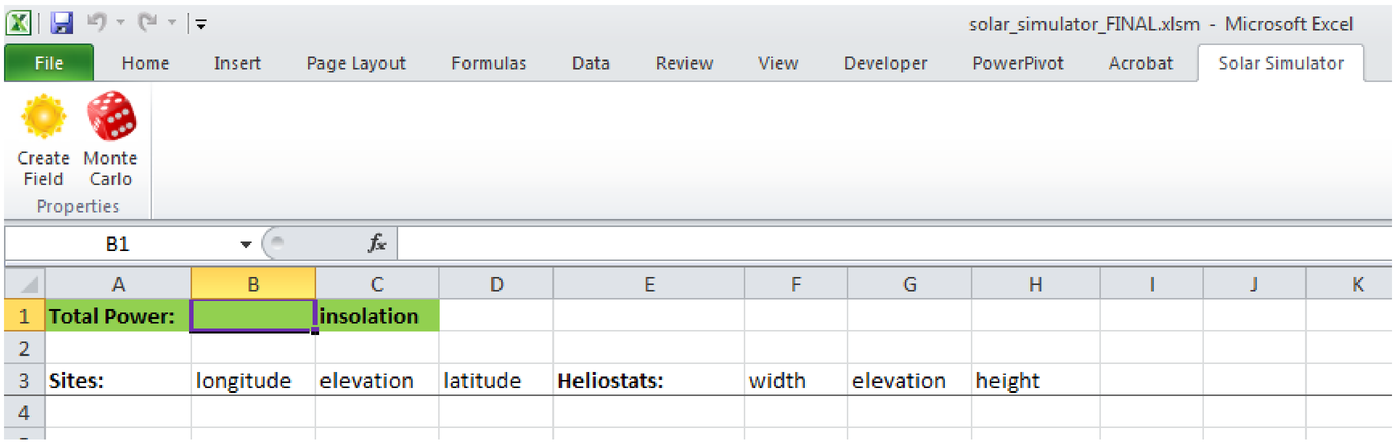
Solar Field Performance Simulator and Optimizer

Preface: my original proposal was to build a data scraper for the European Union patent website, but upon examining robots.txt, I found that this activity was prohibited. So I decided to change my project to this simulation and optimization tool – and I made sure that the scope and difficulty were substantially increased over my original proposal to demonstrate that I wasn't simply copping out.

The Problem: solar power generation can be a very capital expensive process. Most of this capital is deployed in building the solar field which contains hundreds or even thousands of individual heliostats (a heliostat is a mirror or other glass pane attached to a motor and gear that enable it to track the movement of the sun). In a Power-Tower solar configuration, all these individual heliostats track the movement of the sun and position themselves to reflect the sun's rays to a central tower where the heat is gathered and used to generate usable energy (watch the movie *Sahara* to see an example of what this looks like). Obviously, the plant owner wants to know how much energy can be expected from a given solar field. Furthermore, the plant owner also wants to know what the optimum configuration of a solar field should be in order to maximize energy collected for capital deployed on heliostats. The trick is that the wave-particles in each sun ray (called photons) do not always behave as expected. They can bounce in funny angles. They can be blocked by clouds. They can be attenuated by dust. They can do all sorts of unpredictable things on their way from the sun to heliostats to the power tower. Given that building a solar field is a very expensive proposition that requires a plant builder to be born full grown (as in, it won't work unless the entire field is built all at once), the problem is to accurately predict the performance of a theoretical solar field, and then to optimize that field to get maximum performance for a given number of heliostats – and dollars.

The Solution: basically, my project builds various theoretical solar field configurations based on the building sites and heliostat materials available, and then it runs Monte Carlo simulations on those sites on sun positions throughout the year based on the user's inputs for the latitude of the building site. Before I get into the details of how this works, below are the instructions to walk through using this tool (it's really very simple to use) to get an idea of the general function. After going through these quick steps to get familiar with the program, I'll go through them again with detailed explanations on how I did what I did, why I did it, where the limitations are, and what could be done to extend the scope of the project. Here are the simple instructions (if you want to go through it really fast, I promise it's very easy to use):

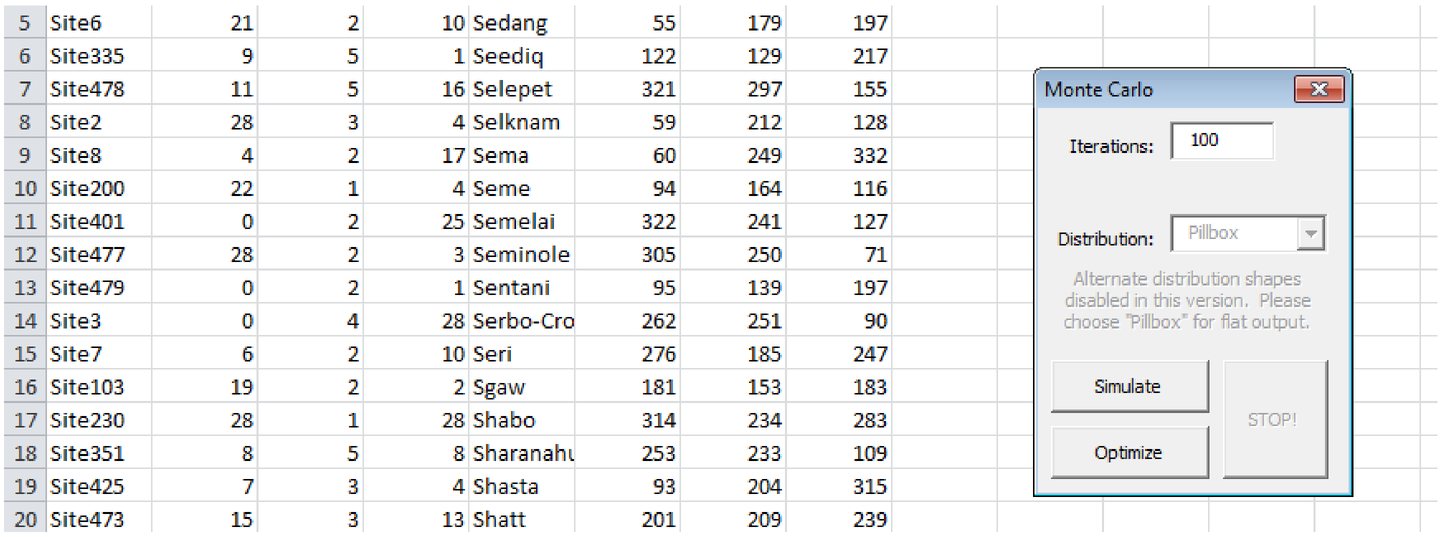
1. **Open the solar_simulator_FINAL.xlsm file and click on the Solar Simulator tab in the Ribbon, and the Current Optimum worksheet. The top of your screen should look like this:**



	A	B	C	D	E	F	G	H	I	J	K
1	**Total Power:**		**insolation**								
2											
3	**Sites:**	longitude	elevation	latitude	**Heliostats:**	width	elevation	height			
4											

2. **Click on the Create Field button (with the picture of the yellow sun above it). This will take you to the Output worksheet, and will show you the solar field that the code has just generated (more details on this later).**

3. **Now click on the Monte Carlo button so that the user form opens. The middle of your screen should now look something like this:**



5	Site6	21	2	10	Sedang	55	179	197
6	Site335	9	5	1	Seediq	122	129	217
7	Site478	11	5	16	Selepet	321	297	155
8	Site2	28	3	4	Selknam	59	212	128
9	Site8	4	2	17	Sema	60	249	332
10	Site200	22	1	4	Seme	94	164	116
11	Site401	0	2	25	Semelai	322	241	127
12	Site477	28	2	3	Seminole	305	250	71
13	Site479	0	2	1	Sentani	95	139	197
14	Site3	0	4	28	Serbo-Cro	262	251	90
15	Site7	6	2	10	Seri	276	185	247
16	Site103	19	2	2	Sgaw	181	153	183
17	Site230	28	1	28	Shabo	314	234	283
18	Site351	8	5	8	Sharanahu	253	233	109
19	Site425	7	3	4	Shasta	93	204	315
20	Site473	15	3	13	Shatt	201	209	239

Monte Carlo

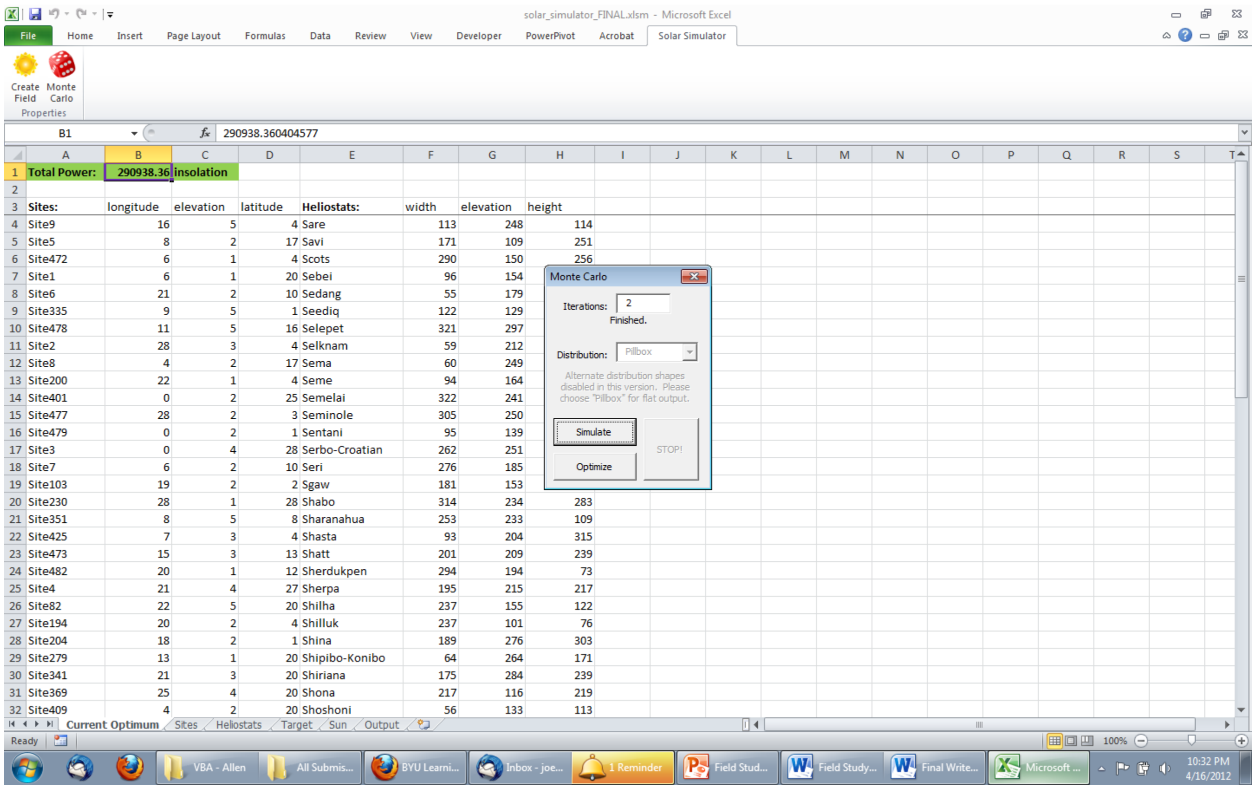
Iterations: 100

Distribution: Pillbox

Alternate distribution shapes disabled in this version. Please choose "Pillbox" for flat output.

4. **Set the iterations to something low (like 1 or 2) so that this doesn't take a long time, and click Simulate.**

5. **This will run the simulation and take you back to the Current Optimum page when you are finished. It will show the expected level of performance (in insolation, or the measure of energy from the sun) for the field configuration you simulated. The screen will look something like this:**



B1 290938.360404577

	A	B	C	D	E	F	G	H
1	**Total Power:**	**290938.36**	**insolation**					
2								
3	**Sites:**	longitude	elevation	latitude	**Heliostats:**	width	elevation	height
4	Site9	16	5	4	Sare	113	248	114
5	Site5	8	2	17	Savi	171	109	251
6	Site472	6	1	4	Scots	290	150	256
7	Site1	6	1	20	Sebei	96	154	
8	Site6	21	2	10	Sedang	55	179	
9	Site335	9	5	1	Seediq	122	129	
10	Site478	11	5	16	Selepet	321	297	
11	Site2	28	3	4	Selknam	59	212	
12	Site8	4	2	17	Sema	60	249	
13	Site200	22	1	4	Seme	94	164	
14	Site401	0	2	25	Semelai	322	241	
15	Site477	28	2	3	Seminole	305	250	
16	Site479	0	2	1	Sentani	95	139	
17	Site3	0	4	28	Serbo-Croatian	262	251	
18	Site7	6	2	10	Seri	276	185	
19	Site103	19	2	2	Sgaw	181	153	
20	Site230	28	1	28	Shabo	314	234	283
21	Site351	8	5	8	Sharanahua	253	233	109
22	Site425	7	3	4	Shasta	93	204	315
23	Site473	15	3	13	Shatt	201	209	239
24	Site482	20	1	12	Sherdukpen	294	194	73
25	Site4	21	4	27	Sherpa	195	215	217
26	Site82	22	5	20	Shilha	237	155	122
27	Site194	20	2	4	Shilluk	237	101	76
28	Site204	18	2	1	Shina	189	276	303
29	Site279	13	1	20	Shipibo-Konibo	64	264	171
30	Site341	21	3	20	Shiriana	175	284	239
31	Site369	25	4	20	Shona	217	116	219
32	Site409	4	2	20	Shoshoni	56	133	113

Monte Carlo

Iterations: 2

Finished.

Distribution: Pillbox

Alternate distribution shapes disabled in this version. Please choose "Pillbox" for flat output.

Simulate

Optimize

STOP!

6. **The foregoing steps will run Monte Carlo simulations of various iterations based on the one solar field configuration and are useful to help calculate total energy coming from the field. However, in order to optimize the field configuration itself, click the Optimize button in the Monte Carlo user form (again, keep the Iterations low to see how it works quickly).**

7. **Notice that the Output worksheet will update and try a new configuration for each Monte Carlo simulation. When the program finds a new maximum output for a given field configuration, it will record the total output on the Current Optimum worksheet and display the current best known field layout below. This optimization will keep running until you click the STOP! button, which is enabled while an optimization is running.**

What I did: now for the details of how this all works. With the exception of building the Ribbon tab and buttons (for which I used the RibbonWizard), I wrote this code and designed these calculations myself. What this really means for practical purposes is that I could spend ages telling you all about it, but I'll try to be brief and cover the main points concisely.

Step 1.

From step 1 above, the Current Optimum worksheet has column headers of **Sites, longitude, elevation, latitude, Heliostats, width, elevation,** and **height.** Line represents a site and heliostat pair (as in one particular heliostat paired up with one particular location where that heliostat can be placed in the solar field building site).

Click on the Sites tab and you will see a list of all the possible sites a heliostat could be installed in a given user field (in this case, I included 488 potential sites). The three numbers represent longitude (an east to west measurement), elevation (the height of the ground from the lowest elevation on the building site), and latitude (a north to south measurement). All these numbers are in meters and Cartesian coordinates. So in summary, for an origin on the Southern edge of the solar field, centered between the Eastern and Western edges, and at the lowest point of elevation for the entire site, these coordinates give the location of every possible building site.

Similarly, the Heliostats tab gives a name and description of each available heliostat. A user could have many duplicates of the same heliostat, or there could be one of each type, or there could be a list of thousands of identical heliostats. The point of this program is to take whatever possibilities the plant owner has to work with and look at possible combinations of all of them until a reasonable optimum is identified. The numbers in the Heliostat tab represent the width of the heliostat, its elevation on the post which holds its motor and gear, and its height (all numbers in centimeters). There are 66 different heliostats for this sample data.

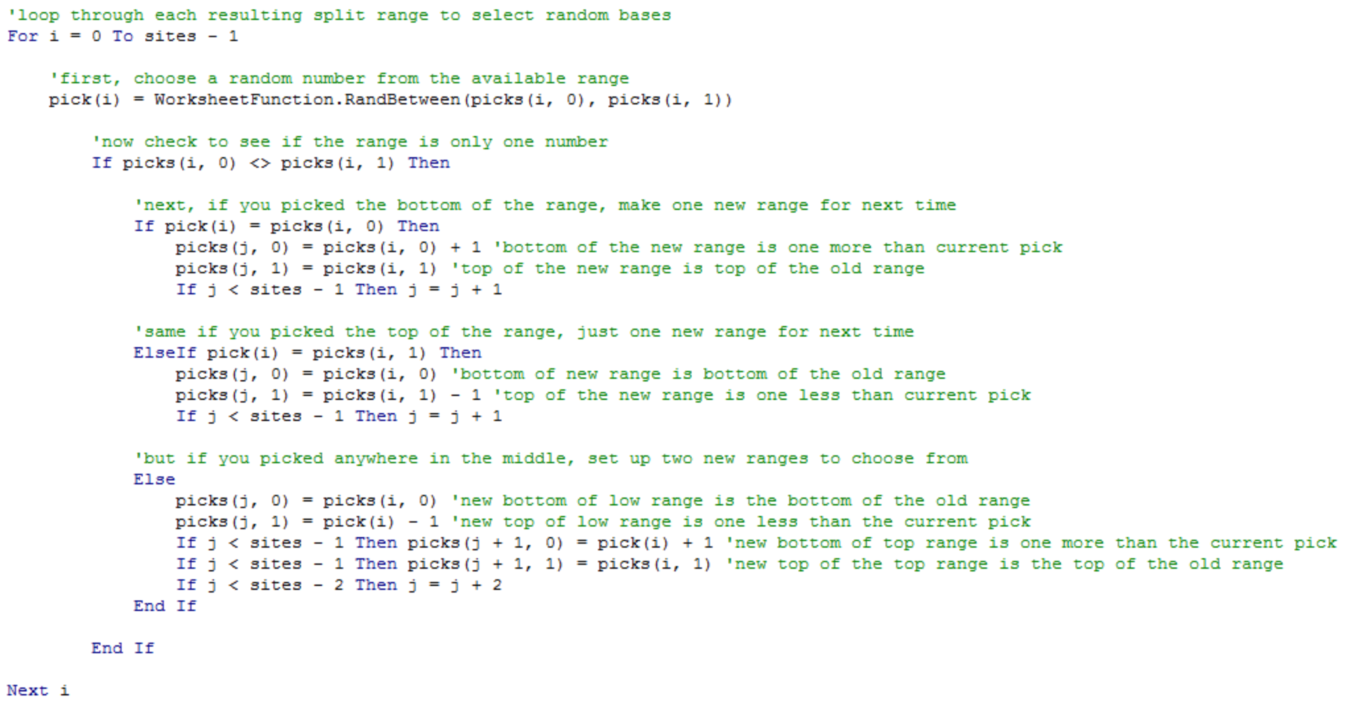
A simulation can have any number of sites and any number of heliostats – the real world doesn't match things up for us naturally, and this code will accommodate that fact.

Step 2.

In step 2, the code automatically generates a possible solar field configuration. If you run this, you see that for the sample data given, it takes a few seconds to generate this field. Consider for a minute what this means. This code is taking 488 possible sites and 66 possible heliostats and pairing them up in such a way that the configuration is random and yet no site or heliostat is used more than once. Furthermore, if there were 488 heliostats and 66 sites, the code would automatically switch the limiting factor in the configuration and run the same random assignment in the same time. If you'd like, go ahead and copy thousands of lines into one or both sheets and run it again – you'll see the speed at which the algorithm runs (even in VBA).

This is an absolutely critical step in solving this problem because it could easily require thousands or even 10s of thousands of possible configurations to identify a stable optimum. Field configurations taking more than a second or two are simply not feasible components of an effective solution. The trick behind this algorithm is the realization that when a calculation selects a random number, what it is really doing is selecting up to two resultant ranges of random numbers yet to be polled. For example, if you pick a random number from 1 to 10, you might pick 7. But what you have really done is identified the yet un-polled ranges of 1 to 6 and 8 to 10. If you keep track of those ranges, then next time your algorithm runs, you can limit your search to the ranges you have not yet examined and thus speed things up dramatically (the more elements you are choosing from, the faster this algorithm becomes in comparison to polling the entire range every single time).

Not only that, but the random selection is interchangeable. In this particular piece of code, I’m using the RandBetween function, but if I were using a particular distribution or following an optimization space routine such as generalized reduced gradient, I could simply swap out the appropriate selection lines and record the new resulting ranges and the algorithm still runs. If you’d like to work through the code to see how it is done, here is a screenshot (it is heavily commented to make it easier to read):



```
'loop through each resulting split range to select random bases
For i = 0 To sites - 1

    'first, choose a random number from the available range
    pick(i) = WorksheetFunction.RandBetween(picks(i, 0), picks(i, 1))

        'now check to see if the range is only one number
        If picks(i, 0) <> picks(i, 1) Then

            'next, if you picked the bottom of the range, make one new range for next time
            If pick(i) = picks(i, 0) Then
                picks(j, 0) = picks(i, 0) + 1 'bottom of the new range is one more than current pick
                picks(j, 1) = picks(i, 1) 'top of the new range is top of the old range
                If j < sites - 1 Then j = j + 1

            'same if you picked the top of the range, just one new range for next time
            ElseIf pick(i) = picks(i, 1) Then
                picks(j, 0) = picks(i, 0) 'bottom of new range is bottom of the old range
                picks(j, 1) = picks(i, 1) - 1 'top of the new range is one less than current pick
                If j < sites - 1 Then j = j + 1

            'but if you picked anywhere in the middle, set up two new ranges to choose from
            Else
                picks(j, 0) = picks(i, 0) 'new bottom of low range is the bottom of the old range
                picks(j, 1) = pick(i) - 1 'new top of low range is one less than the current pick
                If j < sites - 1 Then picks(j + 1, 0) = pick(i) + 1 'new bottom of top range is one more than the current pick
                If j < sites - 1 Then picks(j + 1, 1) = picks(i, 1) 'new top of the top range is the top of the old range
                If j < sites - 2 Then j = j + 2
            End If

        End If

Next i
```

This was a tremendous learning experience through some really interesting mathematically and coding concepts. I began thinking about it with the Boggle project where it occurred to me that it is theoretically possible to randomly poll the 25 possible slots and never find the an unused one. Although the odds of this happening are infinitesimally small with only 25 possibilities, what if you had 10 million possibilities? What if you had to pick a number between one and 10 million, and the only unused number still remaining was 964? Would you keep guessing over and over again only to have the code tell you “Nope, that’s not it” seemingly forever until out of all 10 million numbers you happened to pick 964? It just seemed a little bit inefficient to me for a dataset of any given size, so I wrote the math out on an old notebook page until I could figure out how to retain the random property of selection without ever having to poll the same range twice. Then, for this project, I had to identify which set of inputs was the limiting factor – as in, are there more sites than heliostats or more heliostats than sites – and make the conditionals smart enough to switch between the two while still recursing through the same code. This solution works for one pair – that is, it matches up one input with another – but the algorithm could work in any number of dimensions (you would simply increase the number of dimensions on the arrays and add a nested loop for each one, but identifying the un-polled ranges and continuing the random selection would work the same way it is coded now).

Steps 3 and 4.

The process in step three is fairly simple, but it was extremely tedious to code. This is the actual Monte Carlo simulation, and it uses only a flat distribution (or “Pillbox” as you see in the user form). However, this step is where the actual solar power calculation is performed. What happens is that a square centimeter coordinate on each heliostat is chosen randomly as the place where a theoretical photon will hit. This coordinate is then used in the angular calculation to find the incident angle between the sun, the heliostat and the tower (the tower’s coordinates are input by the user on the Target worksheet). The smaller this angle, the more power will get to the tower because a wide angle means greater dispersion across the total wave. (Imagine shining a flashlight directly at the wall and seeing a relatively small, concentrated and bright spot. Now angle the flashlight and watch the spot turn oblong and dim as the various photons find their path to the target at a wider angle. This is the same thing that happens to sunlight reflecting up to the tower – a wide angle means less total energy hitting the small target required to garner the solar energy.) Here is the calculation for the incident angle of photon impact:

```
angle = (180 / Pi) _
    * WorksheetFunction.Acos(( _
         ((Worksheets("Sun").Cells(k, 2).Value - x) ^ 2 _
           + (Worksheets("Sun").Cells(k, 3).Value - z) ^ 2 _
           + (Worksheets("Sun").Cells(k, 4).Value - y) ^ 2) _
        + ((xT - x) ^ 2 _
           + (zT - z) ^ 2 _
           + (yT - y) ^ 2) _
        - ((Worksheets("Sun").Cells(k, 2).Value - xT) ^ 2 _
           + (Worksheets("Sun").Cells(k, 3).Value - zT) ^ 2 _
           + (Worksheets("Sun").Cells(k, 4).Value - yT) ^ 2) _
                                 ) _
           / (2 _
               * ((Worksheets("Sun").Cells(k, 2).Value - x) ^ 2 _
                   + (Worksheets("Sun").Cells(k, 3).Value - z) ^ 2 _
                   + (Worksheets("Sun").Cells(k, 4).Value - y) ^ 2) ^ (1 / 2) _
               * ((xT - x) ^ 2 _
                   + (zT - z) ^ 2 _
                   + (yT - y) ^ 2) ^ (1 / 2)) _
        )
```

The positions of the sun are also represented in Cartesian coordinates (these are not the literal position of the sun 93 million miles away – only unit vector representations required to calculate the approximate angle of an incoming photon), and they are recorded on the Sun worksheet. This model allows the user to input any number of sun positions. If you check the worksheet, you will see that this sample data includes coordinates for the sun on solstices and equinoxes for each usable solar hour during those days, however, a faster approximation could be made by simply using less solar positions.

Likewise, after rough optimizations have been made, a detailed analysis could be done by using solar positions of each hour of each day throughout the year based on location. In fact, insolation data could be fed into those positions (including modeling the effects of cloud cover, physically blocking and shading of photons by the tower itself or other heliostats, etc.) by simply multiplying the resultant angle against a base insolation level adjusted by each solar position, and the code would still run. This really is a pretty robust little model, at least in some ways (if I may say so, that is).

This step could be made much faster by hiding the Output tab and turning off screen updating (in fact, it could use an array for all the calculations instead of writing to the tab at all), but I wanted to include a visual representation of what the code was doing so it would be easier to follow. Still, given that the vector calculations and Monte Carlo simulation can take some time, it's best to use a low number of iterations when running this section of the code unless you really plan on building an actual solar field.

Steps 5 through 7.

The foregoing steps are the guts of the project. They are designed to run separately so that modules of the code can be used on various datasets. Monte Carlo simulations cannot be interrupted, but any number of external calculations, constraints or configurations can go into the solar field configuration between runs. The Optimize button simply loops these two steps together, but it uses the DoEvents function before and after starting each configuration and simulation so that the user can stop the code once a reasonable optimum is met (or the user is tired and wants to play computer games instead).

If you click Optimize (again, use a low number of iterations, say, 1…) and let it ride, you'll see that the Current Optimum sheet updates periodically. As the code continues to run, the current optimum configuration will update less and less frequently. This is because it is becoming harder and harder to beat the current best-known configuration. Be that as it may, the code will continue running over and over again and do its best to find a new optimum as long as the user lets it run.

Closing Thoughts: before I go, I just wanted to point out that this code is not merely for solar optimizations. It is written to be modular and portable, and to consider any number of inputs or conditions. By swapping out the angular calculation module, any number of conditions could be analyzed. What if you have a housing development with a finite number of lots and only so many types of homes to build? What is the best way to match them up? How about testing virality on a website where a certain number of users will post without knowing that the others have posted, and then that same set of users will react to what the others posted? It's not a random number each time, it's the same number reacting to each other randomly. Or what about a big business initiative or military effort where you have a finite number of team positions to fill and a unique set of discrete individuals to fill them? What's the best way to do that? In each of these examples, the user only needs to swap out the calculation module and populate the input worksheets with the appropriate conditions and inputs – the logic and the speed advantages of the algorithm will still work. That's my project. Thanks for a great class.