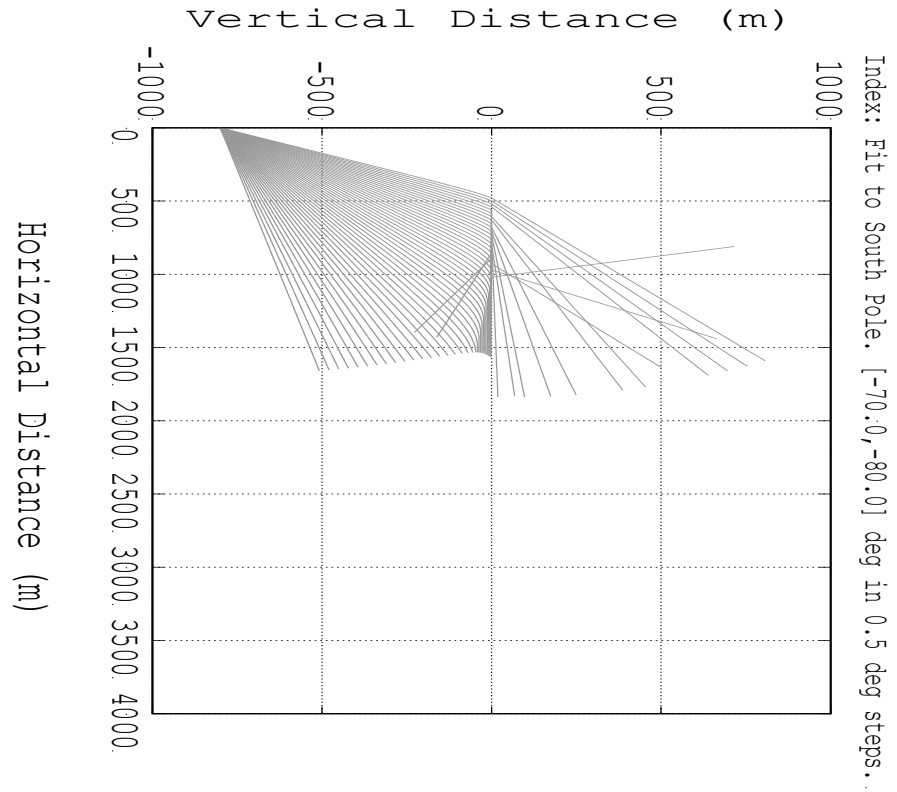
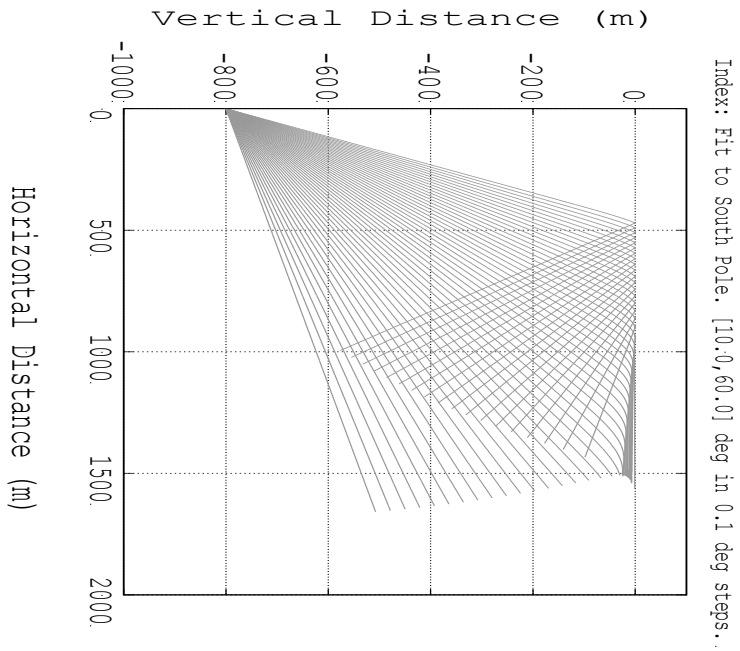
What Jordan's current ray tracing code does?

So I have been using Jordan's code and adding some of my own edits to it the last few months. Its written in c++ and has many classes which inherit from each other. What it mainly does is trace different possibilities for rays with different initial angles. The range of angles, the increment size of the angles and the position of the transmitter can all be adjusted in the RunPropagator.cc class. Here is an example plot of his original code.



As you can see, there are rays that are refracted off the ice surface. So I changed the reflection coefficient to 1.0 to force all the rays to reflect. There are also a few rays in the above plot that are unphysical. This was fixed by changing the TIR criteria, more specifically by modifying the if condition and also removing the break statement in the if loop. The resulting plot is shown below.

P.S. Ignore the title.



So now we do not have any solutions that get refracted and our TIR problem is also fixed.

Note: I added just one reflector at the ice surface.
P.S. Ignore the title.

What AraSim's current ray tracer does?

No fancy plots like Jordan's code. But it does solve ray tracing differential equations to different transmitter and antenna positions. The positions are set inside the code (I don't know where) and they can be accessed by a function called Earth_To_Flat_Same_Depth in RaySolver.cc class. The main ray tracer function in this class is the Solve_Ray function. What I did first is to assess what the main "output" of the ray tracer was. It turned out it is an array of the initial angles, receipt angles, arrival times, pathlengths and surface reflection angles of all the solutions. This can be found in the Report.cc class.

What my code does?

Once I understood what Arasim's ray tracer was doing, I decided to go ahead and add another function within RaySolver.cc and called it Solve_Ray_new. The goal is to use this new function to output the exact same thing that the previous function did, but now with Jordan's code.

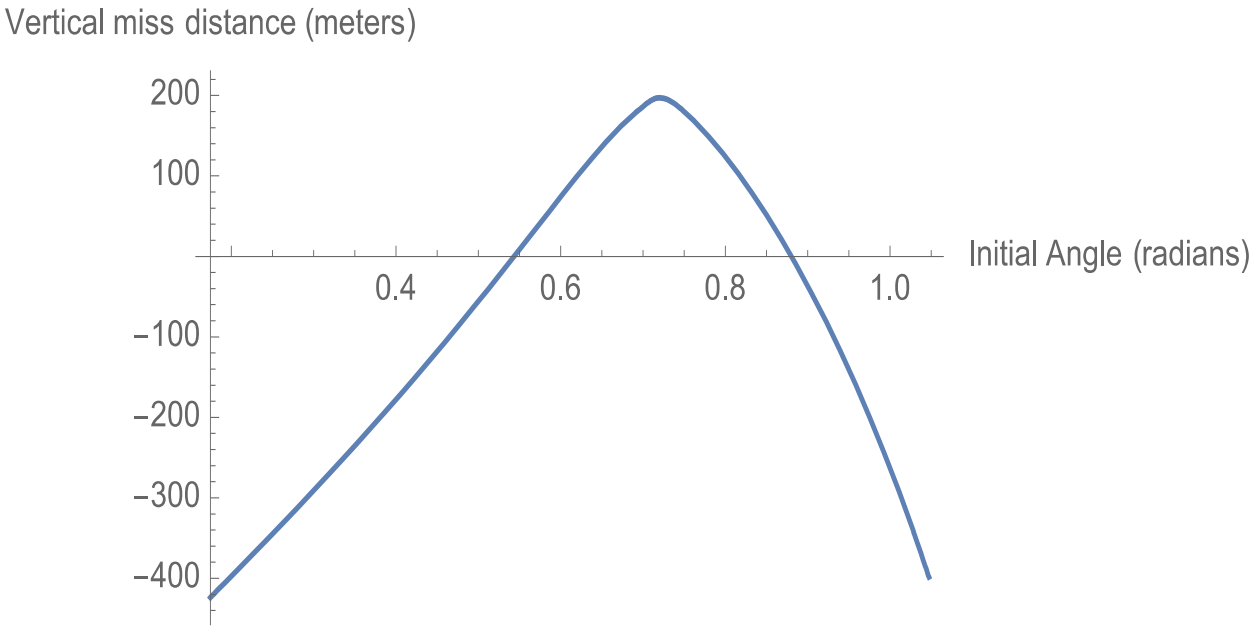
Step 1: Modifying Jordan's code

So I had to modify Jordan's code a little bit to do what AraSim can handle. First off, I added a variable to include the antenna position. I initialized it in the InitializePropagator function in the

Propagator.cc class. Next, what I did was to find the "z distance" at the point where a ray crosses the detector's "x position" using a simple linear interpolation. Using this, I calculated something called the vertical miss distance or the distance between the "z distance" of the ray and the z position of the detector. Then, I go on to calculate the outputs that AraSim produces i.e. the initial angle of the ray, receipt angle, reflection angle (if there are no reflections, I output "No reflections"), arrival time and pathlength.

Step 2: Chris Weaver's Method

I did some research on how Chris Weaver was optimizing the solutions in AraSim, and found a nice way to do it using a simple relationship between the initial angle and the vertical miss distance. So I wrote a new function in the code called Solve_Final_Ray which is called in RaySolver.cc at the end. In this function, I stored the vertical miss distances in one array and their corresponding initial angles in another array. The goal was to find at which point the vertical miss distance's value changed from negative to positive or vice-versa. This indicated that there should be a point in between where the vertical miss distance crossed zero. Then, I did a simple linear interpolation between the points to find at which initial angle this occurred. I've plotted a graph of the vertical miss distance versus the initial angle from the values I got from my code in Mathematica just to give you all a visual representation of this procedure.



As you can probably see from the plot, there are two solutions for this particular ray tracing problem. The first angle which is slightly smaller corresponds to a direct ray solution and the second higher angle to a reflected one. The values that I got using the c++ code were similar to this. The way we know that this technique actually worked is by calculating their vertical miss distances. I found them to be on the order of centimeters for each kind of solution.

Step 3: Integrating within AraSim

The last thing I did was to integrate my code into AraSim's RaySolver.cc class. I essentially created a new function called Solve_Ray_new in it, which has the new ray tracing code. You can switch between this function and the previous ray tracing function by changing the variable RAY_TRACE_MODE in Settings.cc class. The new ray tracer runs and gives solutions now!