Raytracing using Runge Kutta Method

Uzair Latif

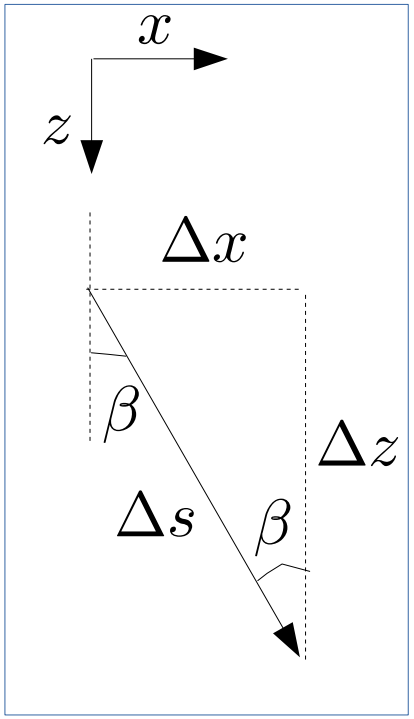
Introduction

- I will basically go into detail of how my raytracing code works.
- I have implemented it in several different ways.
- Today I will discuss in detail the implementation I have done to Tx position scans for ARA and ARIANNA.
- I will also briefly show 2 other implementations where I have used it:
  - Radiation (radiowaves) from cosmic ray showers entering ice and then refracted towards ARA stations
  - Just a basic raytracer with transmits rays until you reach the critical angle of the medium and total internal reflection takes places
- In all of them the fundamental RK4 raytracing code is the same though the complexity varies of how you are using it.

How the Launch angle function works-I

- Used for calculating Launch angles to Direct rays.
- The way I get the angle from that expression is that since I have x0,z0 and x1,z1 therefore I just run a for loop over all values of Beta angle on the function and select the one that gives me the closest value to zero.
- I basically define the function as TF1 and evaluate it until for all values of theta until I get the closest value to zero I can get.
- The step size for the varying the angle is 0.00001 rad.
- There is this weird thing with function (blue box previous slide) that if u(z0)>u(z1) then I have to swap them.
  - Meaning make if u(z0)=a and u(z1)=b then make u(z0)=b and u(z1)=a
  - I essentially took the functional form from AraSim and over there we have the same issue.
  - I think it has something to do with the way the function was derived. Essentially it prevents the square roots from becoming complex.
  - Although at the end this function just gives me a guess at the end and I find the angle on my own so it does not really matter.
- This is implemented in:
  - Double_t *getDirAng(Double_t src, Double_t rec, Double_t x0, Double_t x1);

Raytracing code



- Semi-Analytical approach
  - Solve the following equations with Runge-Kutta (RK4) method:

$$\frac{dx}{ds} = \sin(\beta) \qquad \frac{d\beta}{ds} = -\frac{\sin(\beta)}{n(z)}\frac{dn}{dz}\bigg|_z$$

From Snell's Law

$$\frac{dz}{ds} = \cos(\beta) \qquad \frac{dt}{ds} = \frac{n(z)}{c} \qquad \frac{dA_0}{ds} = \frac{A_0}{L(z,\omega)}$$

Amplitude

Attenuation Length

Need initial conditions . Can provide with x_0, z_0 and x, z

What about the launch angle? β_0

Can use an analytical expression to give us the value of β_0.

Impelementation for Tx position scans for ARA and ARIANNA

Expression for the Launch angle for Direct rays

$$u = A + Be^{Cz} \qquad \sigma_0 = \sin(\beta_0)$$

$$u_0 = A + Be^{Cz_0}$$

$$\frac{C\sqrt{A^2-(\sigma_0 u_0)^2}}{\sigma_0 u_0}(x_0 - x) = \ln\left(\frac{\sqrt{u^2-(\sigma_0 u_0)^2}+\sqrt{A^2-(\sigma_0 u_0)^2}}{u-A} + \frac{A}{\sqrt{A^2+(\sigma_0 u_0)^2}}\right)$$

$$- \ln\left(\frac{\sqrt{u_0^2-(\sigma_0 u_0)^2}+\sqrt{A^2-(\sigma_0 u_0)^2}}{u_0-A} + \frac{A}{\sqrt{A^2+(\sigma_0 u_0)^2}}\right)$$

- Cannot be solved for u
  - So cannot be solved for z as function of x.
- Cannot be solved for launch angle (σ_0) to find the ray between and start and stop points.
- Can be rearranged to be solved numerically for launch angle

$$0 = -\frac{C\sqrt{A^2-(\sigma_0 u_0)^2}}{\sigma_0 u_0}(x_0 - x) + \ln\left(\frac{\sqrt{u^2-(\sigma_0 u_0)^2}+\sqrt{A^2-(\sigma_0 u_0)^2}}{u-A} + \frac{A}{\sqrt{A^2+(\sigma_0 u_0)^2}}\right)$$

$$- \ln\left(\frac{\sqrt{u_0^2-(\sigma_0 u_0)^2}+\sqrt{A^2-(\sigma_0 u_0)^2}}{u_0-A} + \frac{A}{\sqrt{A^2+({}_0 u_0)^2}}\right)$$

Raytracing Parameters

- My values for the refractive index parameters are:
  - A=1.78, B=-0.43, C=0.0132
- I use a fixed step size of 1 mm for my RK4 method to solve the differential equations for raytracing.
- There is basically one script which calculates all 3 types of solutions (independent of each other) for each Tx position.

How the Launch angle function works-I

- Used for calculating Launch angles to Direct rays.
- The way I get the angle from that expression is that since I have x0,z0 and x1,z1 therefore I just run a for loop over all values of Beta angle on the function and select the one that gives me the closest value to zero.
- I basically define the function as TF1 and evaluate it until for all values of theta until I get the closest value to zero I can get.
- The step size for the varying the angle is 0.00001 rad.
- There is this weird thing with function (blue box previous slide) that if u(z0)>u(z1) then I have to swap them.
  - Meaning make if u(z0)=a and u(z1)=b then make u(z0)=b and u(z1)=a
  - I essentially took the functional form from AraSim and over there we have the same issue.
  - I think it has something to do with the way the function was derived. Essentially it prevents the square roots from becoming complex.
  - Although at the end this function just gives me a guess at the end and I find the angle on my own so it does not really matter.
- This is implemented in:
  - Double_t *getDirAng(Double_t src, Double_t rec, Double_t x0, Double_t x1);

How the Launch angle function works-II

- This function gives me 2 solutions to the angle
  - For z0>z1 I select the first one in my code (Tx is shallower than the Rx)
    - The second one is zero.
  - For z0<z1 I select the second one (Rx is shallower than the Tx)
    - The first one is zero
    - For this case if the solution angle is greater than 90 deg. Than I have to to do “soln_ang=180 – soln_ang”
- If the function gives me zero for both angles OR abs(z0-z1)<0.15 m then:
  - I calculate the direct straight line launch angle between the Tx and Rx.
  - Subtract 2 deg from it so as to push it back a bit to account for bending ray launch angle.
- If somehow everything else fails (more details on slide 10) and I still get a zero or below zero angle value then I just force my code to pick a launch angle of 3 deg.
  - This can often happen when my launch angle is too small and I subtract too much from it .
  - Starting at 3 my code will automatically catch the right launch angle since there is very little room to miss it.

Calculate Launch angle for Direct ray-I

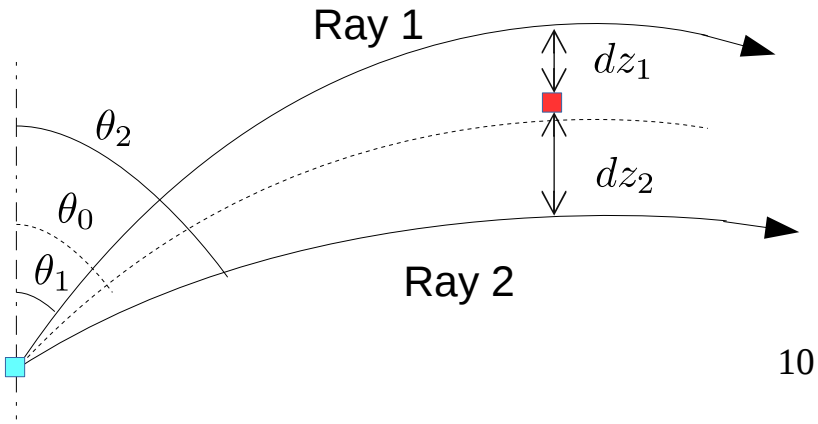
- The expression for the launch angle only gives me an estimate of the possible ray angle.
- I use interpolation to narrow down on the exact ray angle needed to hit the Rx.
  - Take the estimated angle θ_0
  - Select 2 angles around that angle:
    - $\theta_1 = \theta_0 + 2\ ,\ \theta_2 = \theta_0 - 2$

Note: Subtraction in degrees

  - I select my target to be a box of ± 10 cm around the Rx coordinates in both z and x axes.
  - Launch 2 rays on θ_1, θ_2
  - Using dz_1, dz_2 calculate a new launch angle which gives a ray closer to the target:

$$\text{if } dz_2 > dz_1\text{:}$$
$$\theta' = \theta_1 + \frac{|dz_1|}{dz_2 - dz_1} \times (\theta_2 - \theta_1)$$
$$\text{if } dz_1 > dz_2\text{:}$$
$$\theta' = \theta_2 + \frac{|dz_2|}{dz_1 - dz_2} \times (\theta_1 - \theta_2)$$

  - Then using the new angle θ' calculate the two angles for the next iteration. This time the angle width will be 0.6 of the width of the previous iteration.
  - Repeat the whole procedure until you narrow down and hit the target.



Calculate Launch angle for Direct ray-II

- I get the launch angle from:
  - Double_t *getDirAng(Double_t src, Double_t rec, Double_t x0, Double_t x1);
- Sometimes the expression for the launch angle does not give me any solution.
  - This happens mostly when we are very close the shadow region boundaries.
- However I can try to force-find the solution through brute force and there is a sufficient amount of times where I can get a solution when the expression gave me none.
- This happens because the expression assumes that Tx and Rx are both point sources although in our case the Rx is not.
  - Box target in my case.
- So technically the expression is right but then we have to find a solution manually.
- In this case I calculate:
  - The direct straight path angle from source to target.
  - Select 2 angles around that angle:
    - $\theta_1 = \theta_0 + 2\ ,\ \theta_2 = \theta_0 - 2$
    - This is the same angle range as in the previous case
  - The repeat the same procedure as shown in the previous slide.

Limits on iterations

- Everything below is true for all cases: Direct, Reflected, Refracted
- I define one iteration as a pair of 2 rays.
- I let the code run for a total of 17 “complete” iterations (or 35 rays) before it gives up.
- Complete: where the rays hit the target and their launch angles did not have to be adjusted or altered to bring them in position.
- Adjustments need to be made since we are essentially making guesses about the angle.
  - If there is a ray that has to have its angle adjusted then it needs to be thrown multiple times towards the target.
- I stop counting the iterations when these adjustments are happening. These adjustments will be explained in the following slides now.
- In total I let the code run for 32 iterations (or 64 rays) complete or not before it gives up.
  - This allows some room for adjustments before the code stops.

What happens when the Direct ray is being calculated-I

- Now all of this (below) is implemented in:
  - Double_t *getStartAngles(Double_t z0, Double_t x0, Double_t x1, Double_t z1, Int_t solnum);

- At the start of each iteration after the launch angles for the next 2 rays have been calculated I always check for:
  - If the dz's are zero:
    - This would indicate that the target was not passed at all.
    - Then launch angle for rays 20 ± 2 deg.
  - If launch angle of the 1st ray is below zero:
    - Happens generally when launch angle is too close to 0 deg.
    - Then launch angle for rays 2.5 ± 2 deg.
  - If for some weird reason launch angle >180 deg.
    - Then launch angle for rays 100 ± 2 deg.

- The code stops looking for solution if:
  - Even after the application of all these conditions the launch angles are still above >180 deg or
  - Smaller than 0.1 deg
  - This never happens now. I put these conditions in initially when I was playing around with the interpolation method. I have still kept them in as a fail safe to stop the code when it goes crazy.

What happens when the Direct ray is being calculated-II

- The if conditions (a), (b), (c), (d) are implemented in:
  - Double_t *getAngleCorrections(Int_t num, Int_t antco, Int_t D56co,Double_t test0, Double_t test1, Int_t psns, Int_t in,Int_t solnum)

- (a) If the surface is hit by the ray 1st ray of the iteration:
  - Essentially move the 'cone' forwards by increasing both the angles.
  - Increment size depends on
    - how close the Tx and Rx are.
    - What is the number of ray that is being thrown?
    - Generally it takes around 20 rays (or 10 full iterations)to hit the target.

```
If(num>8 && TxRxdist<201){
  test0=test0+0.1*(pi/180);
  test1=test1+0.1*(pi/180);
 }
if(num<9 && TxRxdist<201){
  test0=test0+0.4*(pi/180);
  test1=test1+0.4*(pi/180);
  }
if(num>8 && TxRxdist>200){
  test0=test0+1*(pi/180);
  test1=test1+1*(pi/180);
  }
if(num<9 && TxRxdist>200){
  test0=test0+2*(pi/180);
  test1=test1+2*(pi/180);
 }
```

Num= nth ray
test0=Launch angle of 1st ray
test1=Launch angle of 2nd ray
200 & 201=Tx Rx horizontal distance in m.

What happens when the Direct ray is being calculated-III

- (b) If the surface is hit only by the 2nd ray of the iteration (although this is never the case as both hit together):
  - Do the same as you did in (a). Move the cone forward by increasing the launch angles.

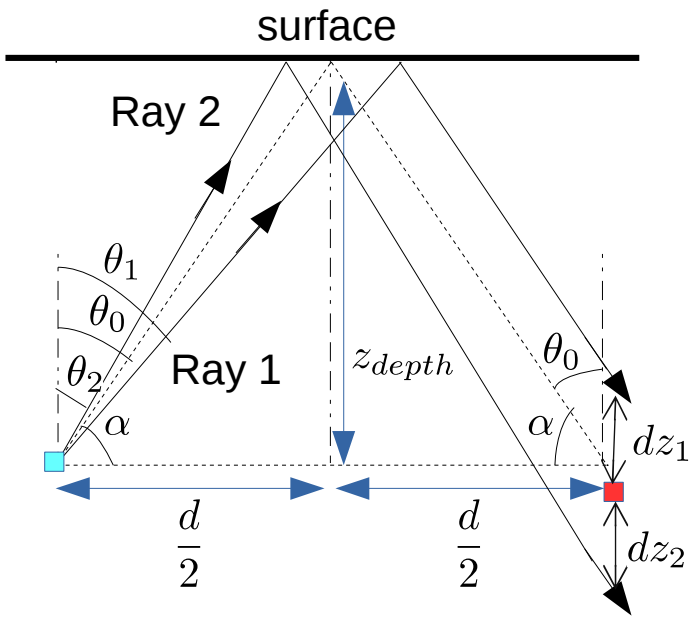
```
if(psns<0 && in==1){
   test0=test0-fabs(test1-test0)*0.6;
   test1=test1-fabs(test1-test0)*0.6;
 }

 if(psns<0 && in==2){
   test1=test1-fabs(test1-test0)*0.6;
 }
```

- (c) If the 1st ray hits the bottom of the scan region and has not passed target (the 2nd will ofcourse do the same):
  - I move the cone backward by decreasing both the launch angles.
  - I decrease them by 60% of the cone width. This is done so very small angle variations can be achieved to hit the target.

- (d) If only the 2nd ray hits the bottom of the scan region and has not passed the target:
  - I just decrease the launch angle of the 2nd ray.

Calculate Launch angle for Reflected ray-I

- For reflected ray I use a similar approach.
  - Make an equilateral triangle (dotted line) using Tx depth (z_depth) and distance between Tx and Rx (d).
  - Estimate angle θ_0
  - Launch 2 rays around that angle.
  - Use the interpolation process (the same one used earlier) until I zone in on the launch angle to hit Rx.



$$\theta_0 = \frac{\pi}{2} - \alpha$$

$$\alpha = \arctan\left(\left|\frac{2z_{depth}}{d}\right|\right)$$

$$\theta_1 = \theta_0 - 2\ ,\ \theta_2 = \theta_0 + 2$$

Old method now demoted to secondary method.

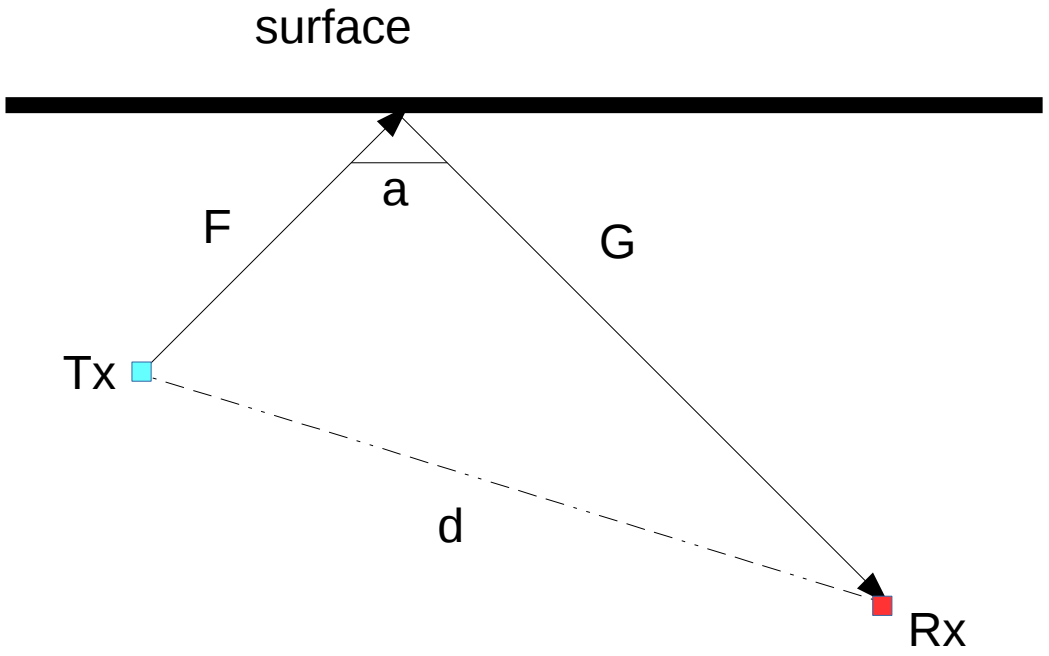
Note: Subtraction in degrees

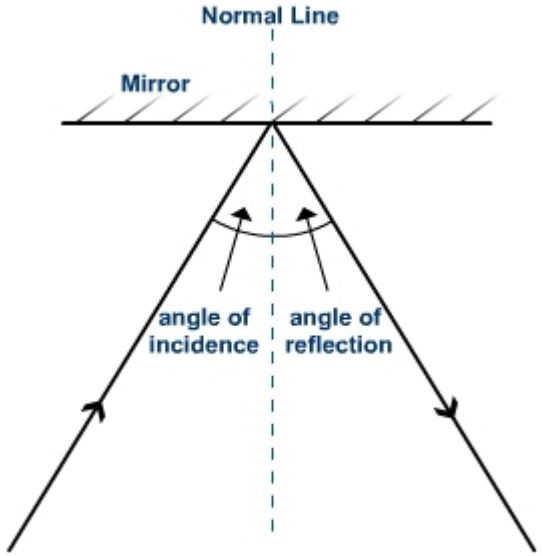
d: distance between Tx & Rx

z_depth: Tx depth

Calculate Launch angle for Reflected ray-II

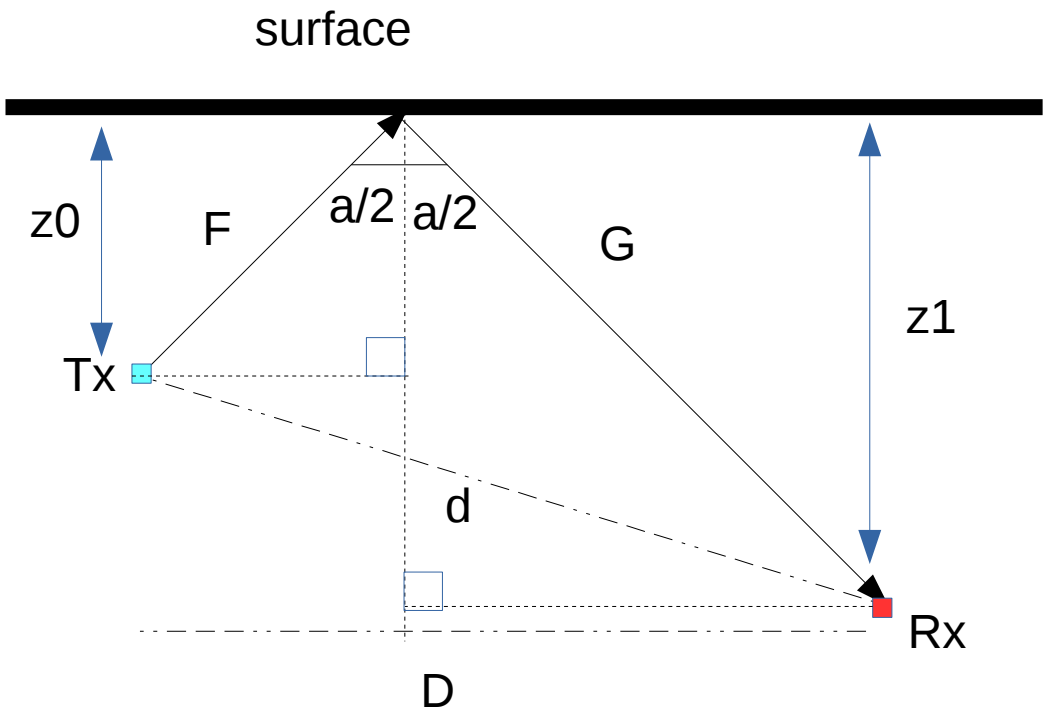
- Now we talk about the new method
- We already know for reflection that the angle of incidence equals the angle of reflection.





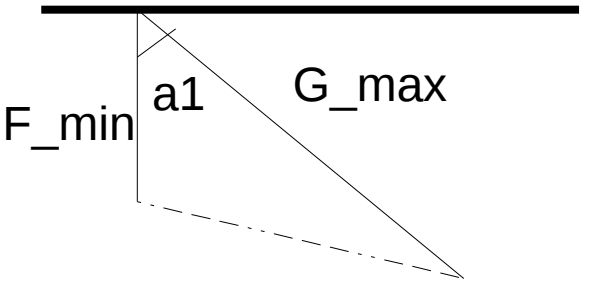
Calculate Launch angle for Reflected ray-III

- The FGd triangle should obey 2 conditions
  - **Condition 1**:
    - abs(F/z0)-abs(G/z1)=0
  - **Condition 2**:
    - Cosine rule: F*F+G*G-2*G*F*cos(a)-d*d=0

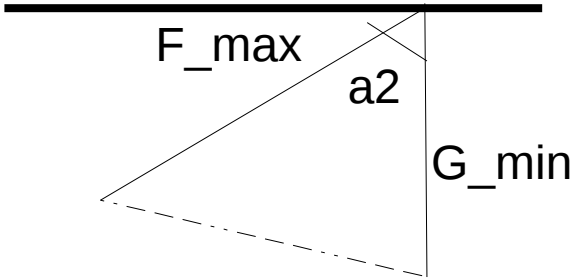


-Here we can consider 2 extreme cases:

Case 1



Case 2



-If a1<a2 then a1=a_min

-If a2<a1 then a2=a_min

Calculate Launch angle for Reflected ray-IV

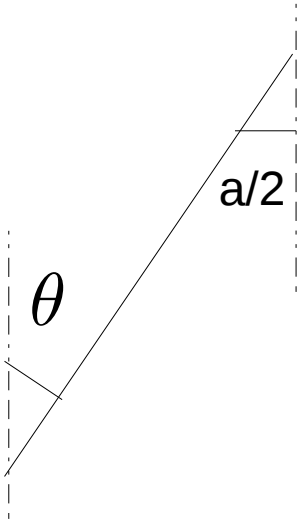
- So using the 2 conditions given in the previous slide and the minimum and maximum cases for F,G and a:
  - I run a for loop for each of the three variables over the cosine rule until I get value less than 1.
  - This gives me a value of a which approximately satisfies all of the required conditions.

- Using the value of a/2 I can calculate the value of the launch angle.
  - Launch angle= a/2 – 2 (in deg)
  - I subtract 2 in order to try to account for the curvature of the bending ray.

- This whole thing take about a second and can give me a value for the launch angle for about ~95% of time.

- When this method fails I move back to the secondary method.

- **Note:** Primary method gives me a value of launch angle which is always less than the actual value. Secondary often gives me a value above the required value and I have to make big adjustments to correct for it.



What happens when the Reflected ray is being calculated-I

- I make sure I successfully got the launch angle from the primary method from the function:
  - Double_t getReflAng(Double_t z0, Double_t x0, Double_t x1, Double_t z1);

- Now all of this (below) is implemented in:
  - Double_t *getStartAngles(Double_t z0, Double_t x0, Double_t x1, Double_t z1, Int_t solnum);

- If not then I move to the secondary method.
  - After getting the launch angle from the secondary method I apply these two corrections:
  - midang=((pi/2)-atan(fabs((z0*2.0)/(x1-x0))));
  - if(midang>49*(pi/180)){
    midang=midang-15*(pi/180);
    }
  - if(midang<50*(pi/180)){
    midang=midang-0.5*(pi/180);
    }

Midang= Just a code name for the main launch angle.

- Then after subtracting ± 2 and getting my launch for my 2 rays I check for
  - if the launch angle of my 1st ray is less than 0
    - If yes then assign a new main launch of 3 deg and then redefine the rays around it.
    - Again this happens when the actual launch angle is actually very close to 0 deg.

  - If the 2nd ray's launch angle is greater than 60 deg
    - Subtract 2 deg from launch angles of both the rays. To move the cone inside 60 deg angle range.
    - The number 60 comes from after looking at data from my analysis for deep (200 m) and shallow (2 m) antennas. In those cases I have noticed that the reflection launch angle actually never goes above 60 deg. However this may change for different cases.

What happens when the Reflected ray is being calculated-I

- The if conditions (a), (b), (c), (d) are implemented in:
  - Double_t *getAngleCorrections(Int_t num, Int_t antco, Int_t D56co,Double_t test0, Double_t test1, Int_t psns, Int_t in,Int_t solnum)

- (a) If the bottom of the scan region is hit by the ray 1st ray of the iteration and it does not pass the target:
  - Essentially move the 'cone' forwards by increasing both the angles.
  - Increment size depends on
    - how close the Tx and Rx are.
    - What is the number of ray that is being thrown?
    - Generally it takes around 20 rays (or 10 full iterations)to hit the target.

```
If(num>8 && TxRxdist<201){
  test0=test0+0.1*(pi/180);
  test1=test1+0.1*(pi/180);
 }
if(num<9 && TxRxdist<201){
  test0=test0+0.4*(pi/180);
  test1=test1+0.4*(pi/180);
  }
if(num>8 && TxRxdist>200){
  test0=test0+1*(pi/180);
  test1=test1+1*(pi/180);
  }
if(num<9 && TxRxdist>200){
  test0=test0+2*(pi/180);
  test1=test1+2*(pi/180);
 }
```

Num= nth ray
test0=Launch angle of 1st ray
test1=Launch angle of 2nd ray
200 & 201=Tx Rx horizontal distance in m.

What happens when the Reflected ray is being calculated-II

- (b) If the bottom of the scan region is hit only by the 2nd ray of the iteration (the 1st ray will also hit it):
  - Do the same as you did in (a). Move the cone forward by increasing the launch angles.

```
if(psns<0 && in==1){
   test0=test0-fabs(test1-test0)*0.6;
   test1=test1-fabs(test1-test0)*0.6;
  }

  if(psns<0 && in==2){
   test1=test1-fabs(test1-test0)*0.6;
  }
```

- (c) If the 1st ray does NOT hit the surface (the 2nd will ofcourse do the same):
  - I move the cone backwards by decreasing both the launch angles.
  - I decrease them by 60% of the cone width. This is done so very small angle variations can be achieved to hit the target.

- (d) If only the 2nd ray does NOT hit the surface and 1st does it means that target is actually present in that region:
  - So I just decrease the launch angle of the 2nd ray.

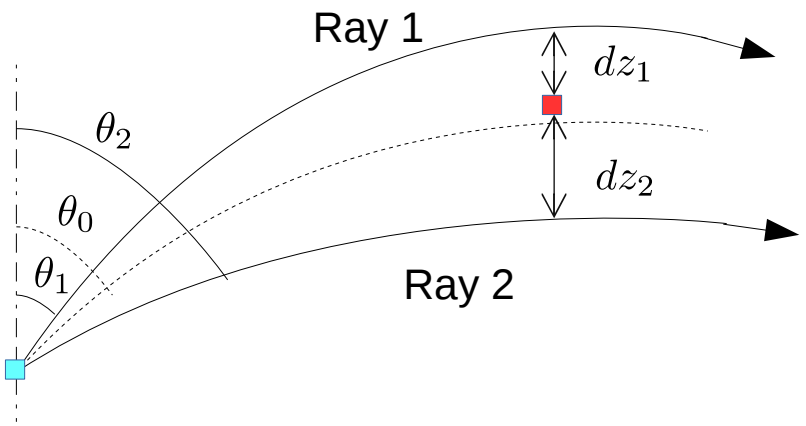
What happens when the Reflected ray is being calculated-III

- Also between each iteration after the launch angles for the next 2 rays have been calculated I always check for:
  - If the dz's are zero or less than zero:
    - Then launch angle for rays 20 ± 2 deg.
  - If launch angle of the 1st ray is below zero:
    - Happens generally when launch angle is too close to 0 deg.
    - Then launch angle for rays 2.5 ± 2 deg.
  - If the launch angle is above 65 deg:
    - Again this comes from looking at past results fro deep (200 m) and shallow (2m) Rx.
    - Start the launch angles from 5 ± 2 deg.
  - If for some weird reason launch angle >180 deg.
    - Then launch angle for rays 50 ± 2 deg.
- The code stops looking for solution if:
  - Even after the application of all these conditions the launch angles are still above >180 deg or
  - Smaller than 0.1 deg
  - This never happens now. I put these conditions in initially when I was playing around with the interpolation method. I have still kept them in as a fail safe to stop the code when it goes crazy.

Calculate Launch angle for Refracted ray-I

- Refracted rays are basically direct rays and take longer times to reach target as compared to Direct rays.
  - They are the hardest ones to make using my raytracing code.
- Having multiple 'direct' rays is possible because I am treating Rx as a box and not a point target.
- For refracted ray I use the approach of the reflected rays to calculate initial angle but then the rays themselves follow a direct path.
  - Make an equilateral triangle (dotted line) using Tx depth (z_depth) and distance between Tx and Rx (d).
  - Estimate angle θ_0 using the Reflected ray formula.
  - Launch 2 rays around that angle.
  - Use the interpolation process (the same one used earlier) until I zone in on the launch angle to hit Rx.

Note: For shallow stations for ARIANNA all refracted ray solutions were the same as direct ones so I did not include them in my plots.



$$\theta_0 = \frac{\pi}{2} - \alpha$$

$$\alpha = \arctan\left(\left|\frac{2z_{depth}}{d}\right|\right)$$

$$\theta_1 = \theta_0 - 2\ ,\ \theta_2 = \theta_0 + 2$$

Old method now demoted to secondary method.

Note: Subtraction in degrees

Calculate Launch angle for Refracted ray-II

- Refracted rays are the hardest ones to make and my code still struggles with them as they lie somehwere between reflected and direct rays.

- For the refracted rays I just calculate the launch angle the same way I do for the reflected rays but the only caveat is that 15 deg to what I get suggested reflected launch angle.

- This is implemented in:
  - Double_t *getStartAngles(Double_t z0, Double_t x0, Double_t x1, Double_t z1, Int_t solnum);

What happens when the Refracted ray is being calculated?

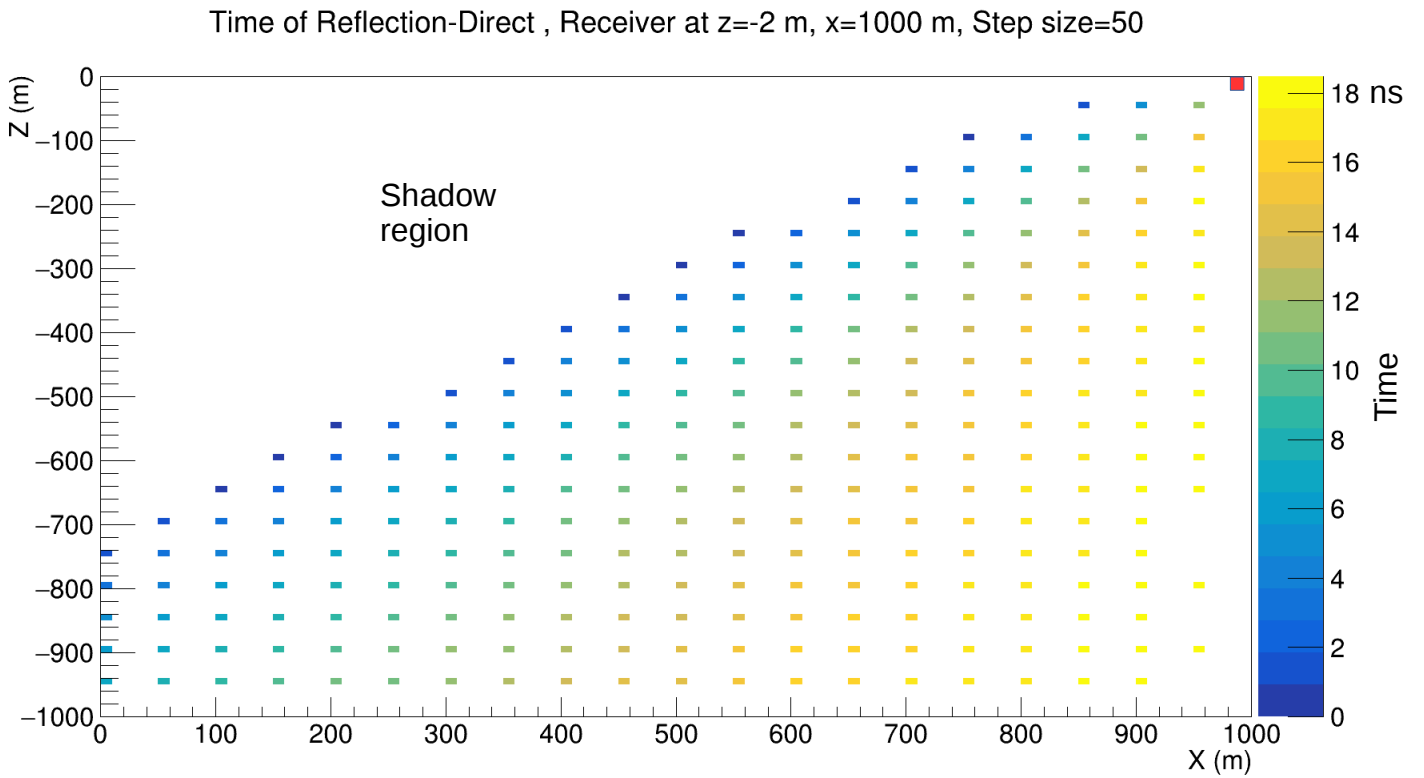
- Pretty much the same process is followed as if you are calculating a direct ray.

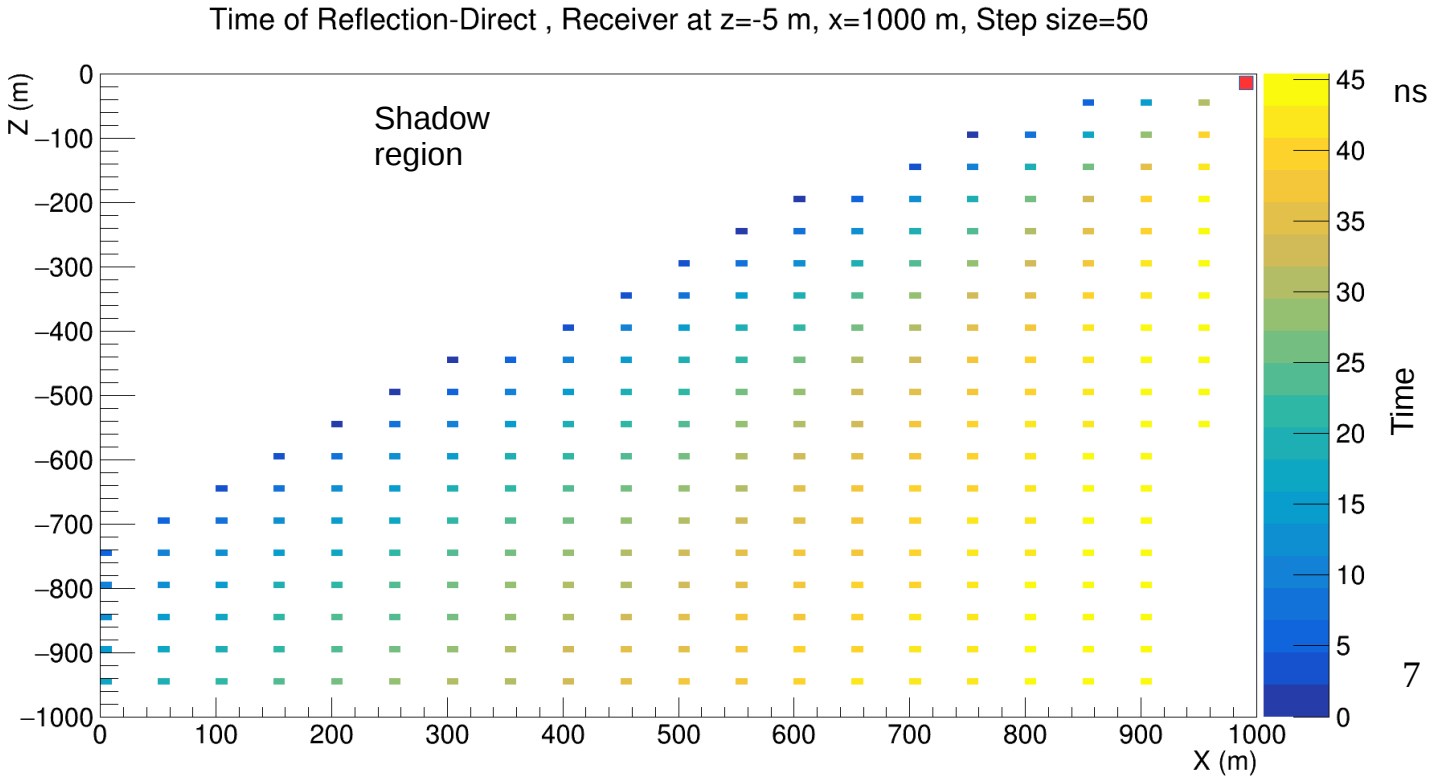
- The two differences are that:
  - when you are starting off an iteration and the launch angles come up to be <0 deg then:
    - Then launch angle for rays 65 ± 2 deg.
    - This is because refracted rays really cant have that small launch angles. They essentially come into play launch at much bigger launch angles than these. At these small launch angles you only get a direct ray. (refer to backup slide)
  - I add additional adjustments to refracted ray angles to make them vary faster at closer distances.:
    -
```
if((psns>0 && solnum==2)){
if(num>8 && fabs(antco-D56co)<201){
 test0=test0+1*(pi/180)*psns;
 test1=test1+1*(pi/180)*psns;
 }
 if(num<9 && fabs(antco-D56co)<201){
 test0=test0+2*(pi/180)*psns;
 test1=test1+2*(pi/180)*psns;
 }
}
```

psns= just a sign indicating whether the angles should be added or subtracted.
solnum=telling what type ray you are working with.

R-D Time Difference

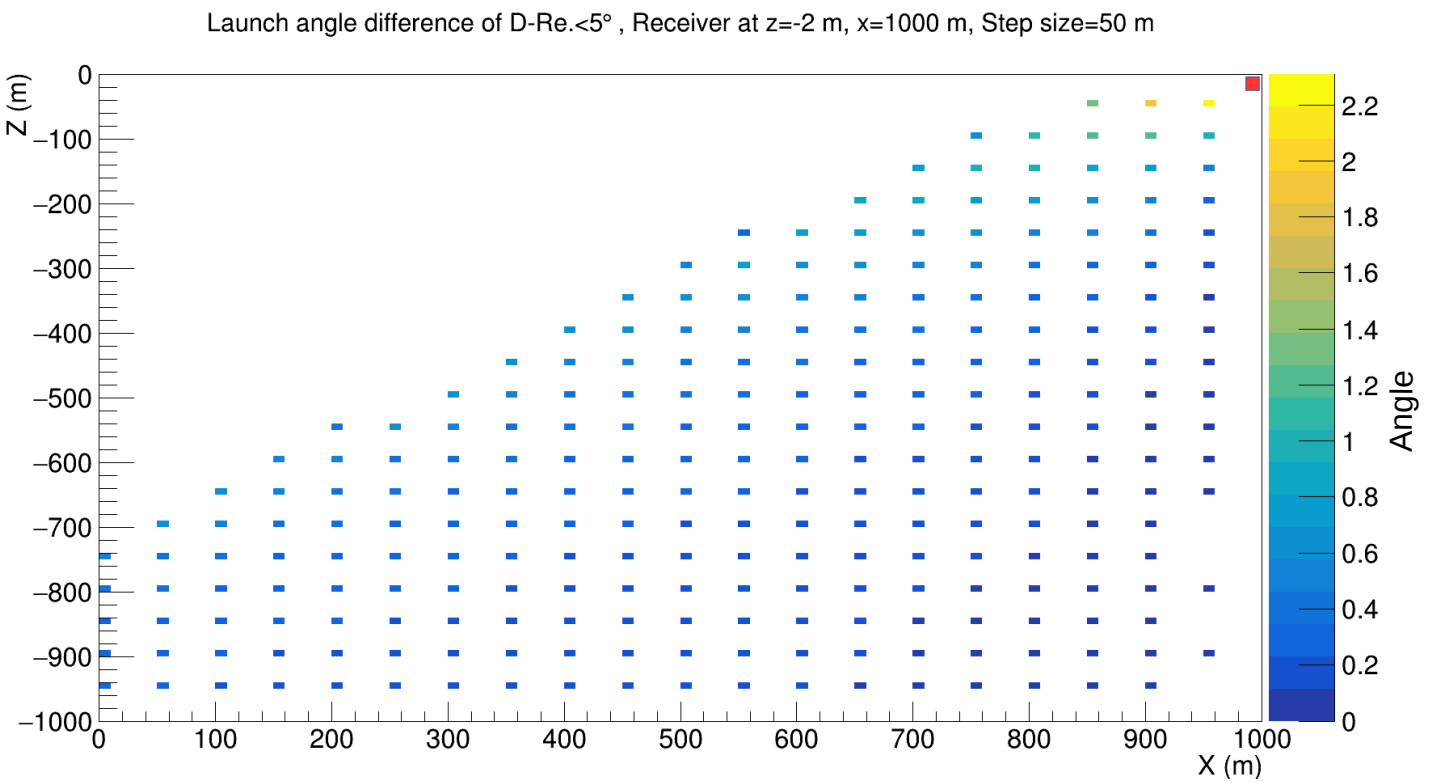
- This plot shows the time difference between the direct and the reflected rays to reach the target.
- As you go further in Rx depth the time difference tends to increase overall.
- The time differences are of much lower magnitude for shallow Rx depths as compared to large depths.

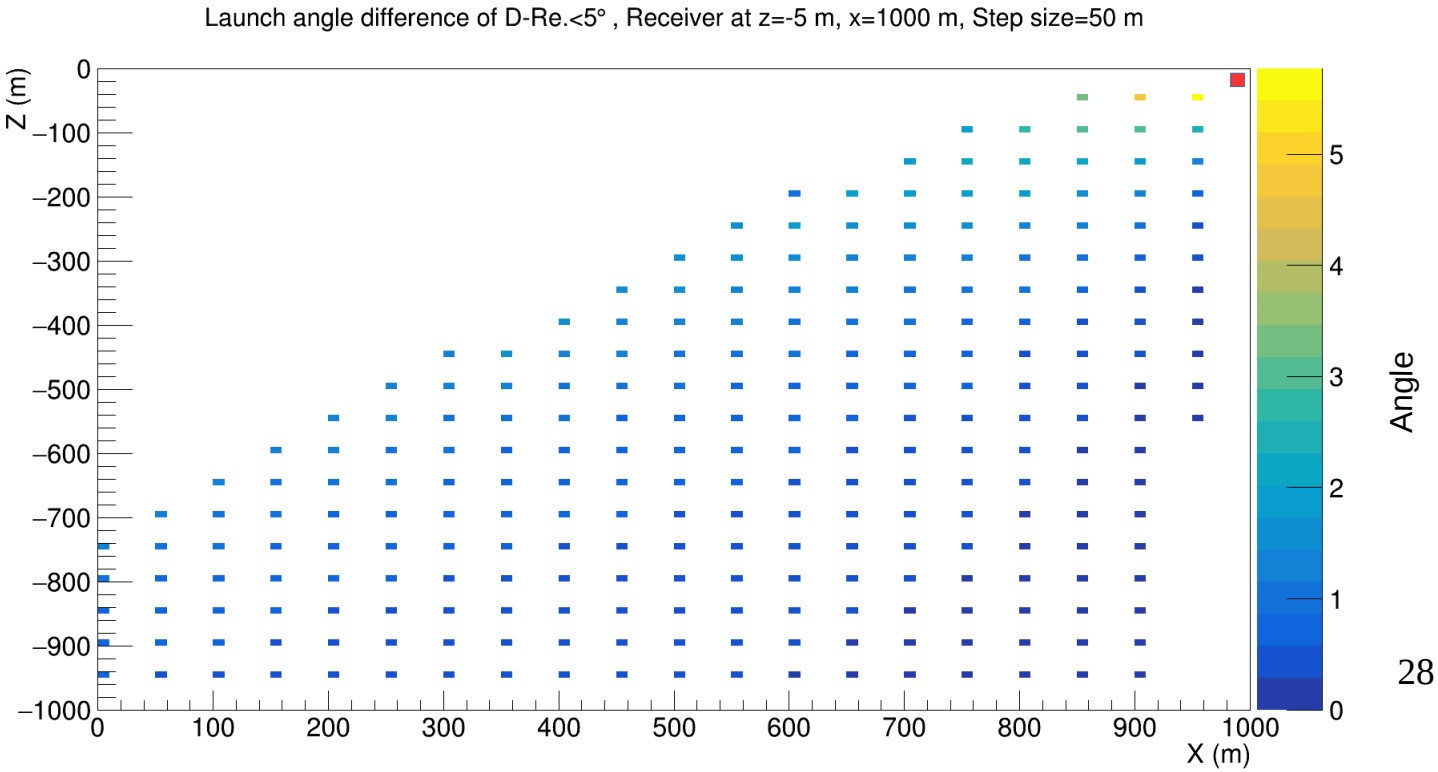




D-R Launch angle

- This plot shows the launch angle difference between the direct and the reflected rays to reach the target.
- The Cherenkov cone produced by the neutrino has width of ~3 degrees.
- So I am just focusing on events which have angular difference of <5 degrees.

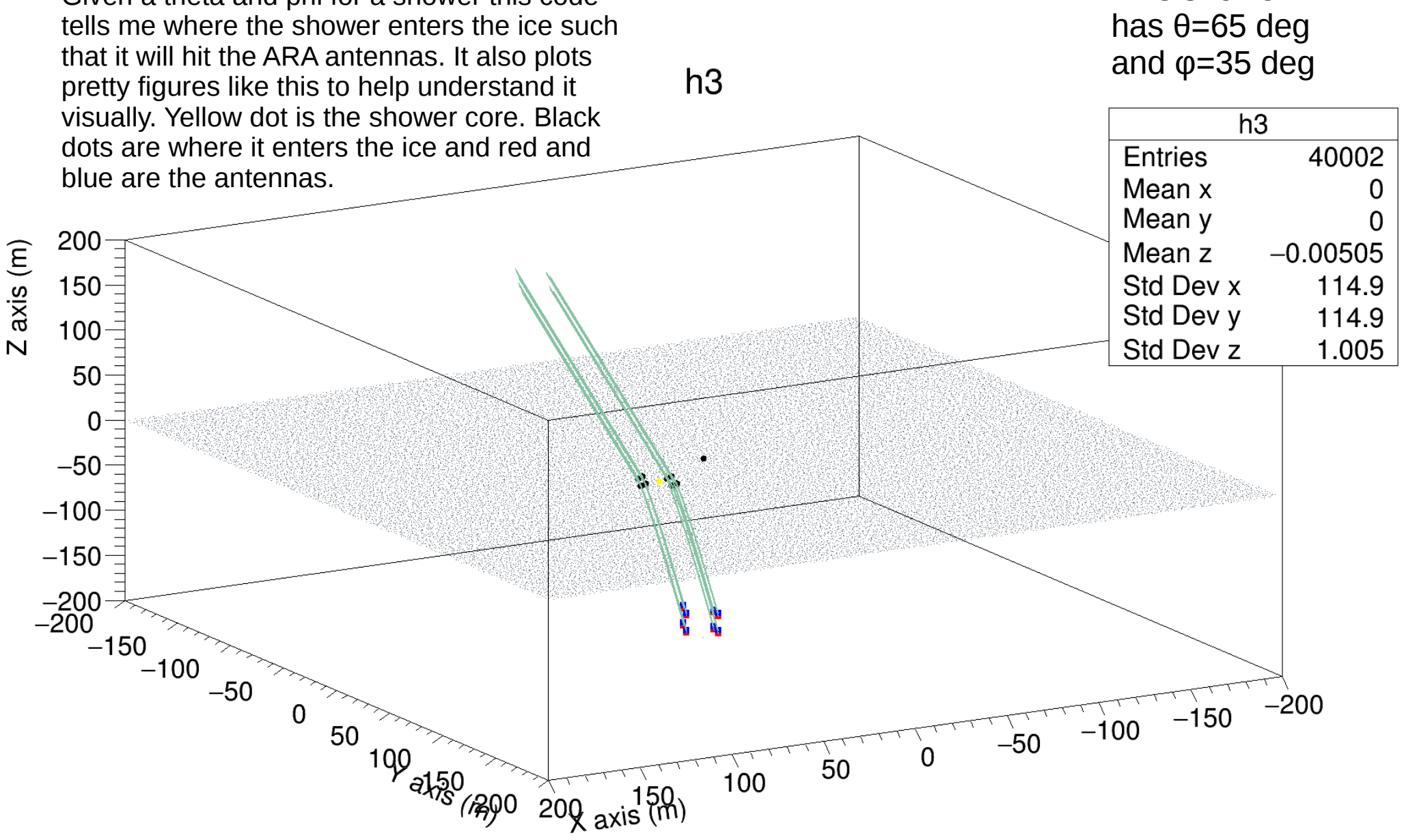




Implementation for radiation from showers entering ice

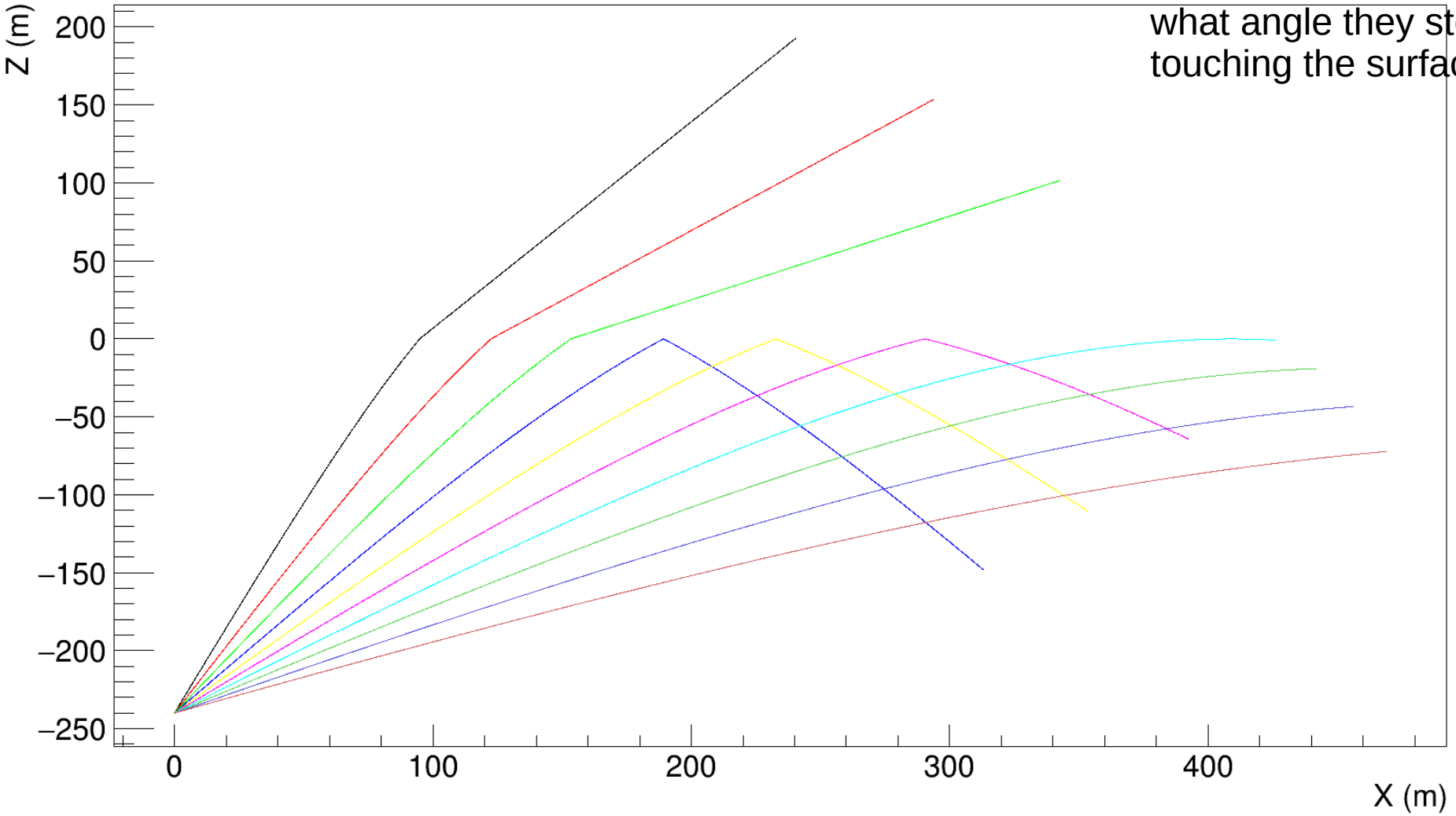
Given a theta and phi for a shower this code tells me where the shower enters the ice such that it will hit the ARA antennas. It also plots pretty figures like this to help understand it visually. Yellow dot is the shower core. Black dots are where it enters the ice and red and blue are the antennas.

This shower has θ=65 deg and φ=35 deg



Implementation for Transmission of rays

This code just tells me at what angles the rays stop transmitting and at what angle they stop touching the surface.



Backup

Refracted Times

- This plot shows the time taken **only** by the refracted rays to reach the target.
- Refraction mostly seems to happen in the region next to the shadow region (left side of the figure).
- The number of refracted solutions also increases:
  - As you go further away from Rx in distance.
  - As you go deeper in Rx depth.
- In places where the angular difference between a reflected and a refracted ray have very small angular difference my code finds it hard to zone in one of them.
  - In these cases it either gives me no solution or gives me the same solution as the direct case.