Implementation of a Point-to-Point Ray Tracer

Martin Scott Driggers and Stephen R. Kaeppler

April 30, 2021

Contents

1 Abstract **2**

2 Mathematical Background **2**
2.1 Fermats Principle . . . 2
2.2 Calculating Variables . . . 2

3 Iterative Algorithm **3**
3.1 Newton Raphson Steps . . . 3
3.2 Initial Path Seed . . . 4

4 Path Definitions **5**
4.1 Great Circle Deviation . . . 5
4.2 Calculating Hessian and Gradient with GCD . . . 6

5 Relevant Implementation Details **7**
5.1 Standardized Components . . . 8
5.2 Numpy and Scipy . . . 8
5.3 Multiprocessing . . . 8
5.4 Nonlinear Solvers . . . 9
5.5 Visualizations . . . 9
5.6 Documentation . . . 9

6 Results **10**
6.1 Code Summary . . . 10
6.2 Working examples . . . 10

7 Future Work **13**

8 Conclusions **13**

9 Acknowledgements **13**

10 Source code **13**

1 Abstract

Most ray tracers treat the problem as a Hamiltonian optics problem given an index of refraction, where the index of refraction is derived from atmospheric properties. This direct IVP approach works well when tracing rays with a known start point and launch angle. This method is more less effective when tracing a ray between known start and end points with an unknown launch angle. To overcome this hurdle, some have implemented homing methods that trace numerous rays from the start point with a variety of launch angles until one ray intersects the end point. This task is computationally expensive and difficult to converge especially for high-rays. To solve this issue, Dr. Christopher Coleman developed an iterative approach using Fermat's principle for an aniosotropic medium [Col11]. This approach begins with a initial estimation of the ray's path, and then iteratively mutates this path to find one with a stationary phase angle. In this work, I lay out my implementation of Dr. Coleman's approach. Development of this method may provide a potentially efficient method for determining the linking path between a transmitter receiver pair, given a model ionosphere.

2 Mathematical Background

The detils of the mathematical derivation are presented in Coleman 2011, but I will lay out the groundwork for our computational formulation in this section.

2.1 Fermats Principle

This work relies on Fermat's principle, which states that the total phase angle of any ray path is stationary. We can write this as

$$\delta P = 0. \tag{1}$$

The phase angle in this equation can be written an integral along the ray path. To define this integral, we first need to define some variables. We will call $\mu(\boldsymbol{r})$ the index of refraction of the medium, at point $\boldsymbol{r}$. As we trace out the ray path, we will call $\boldsymbol{r}(t)$ is the position of the ray at time (t), f the wave frequency (or operating frequency), f_p is the plasma frequency, and f_H the gyro frequency.

Then we define $\boldsymbol{Y}$ to be a vector that points along the magnetic field with magnitude $\frac{f_H}{f}$, $X = \frac{f_p^2}{f^2}$, and $\boldsymbol{t}$ is the unit vector tangent to the ray path. We also define $\boldsymbol{p}$ to be a unit vector normal to the wavefront. Note that in an isotropic medium, $\boldsymbol{p} = \boldsymbol{t}$. Now we will finally define $p_t = \boldsymbol{p} \cdot \boldsymbol{t}$, $Y_p = \boldsymbol{Y} \cdot \boldsymbol{p}$, and $Y_t = \boldsymbol{Y} \cdot \boldsymbol{t}$. We can now write P as

$$P = \int \mu(\boldsymbol{r}, Y_p) p_t ds. \tag{2}$$

In his 2011 Radio Science paper, Coleman lays out the details of this derivation of Equation 2 but for this work, we will simply provide the results.

2.2 Calculating Variables

In an isotropic medium, $p_t = 1$ and $Y_p = Y_t$, so our solution to Equation 2 would be straightforward. Unfortunately, the ionosphere with a non-zero magnetic field is an aniosotropic medium. In this medium, we need to derive Y_p and p_t from our knowledge of the interactions between the atmosphere, the magnetic field, and the ray path.

Coleman uses clever differential calculus and assumptions about the relationship between Y_p and μ to calculate these unknown parameters. The end result is two coupled algebraic equations for p_t

and Y_p

$$1 = p_t \left(p_t - \frac{1}{2} \left(Y_t - Y_p p_t \right) \frac{\partial \ln \left(\mu^2 \right)}{\partial Y_p} \right) \tag{3}$$

and

$$Y_t = p_t \left(Y_p - \frac{1}{2} \left(Y^2 - Y_p^2 \right) \frac{\partial \ln \left(\mu^2 \right)}{\partial Y_p} \right). \tag{4}$$

In addition, we use the following equations to solve for μ and its derivative with respect to Y_p

$$\mu^2 = 1 - \frac{2X(1-X)}{2(1-X) - \left(Y^2 - Y_p^2 \right) \pm \sqrt{\left(Y^2 - Y_p^2 \right)^2 + 4(1-X)^2 Y_p^2}} \tag{5}$$

$$\frac{\partial \ln \left(\mu^2 \right)}{\partial Y_p} = \frac{-XY_p \left(\mu^2 - 1 \right)}{A\mu^2 + B}. \tag{6}$$

In Equation 5, we choose the plus sign is for ordinary rays and the minus for extraordinary rays.

Finally, we calculate A, B, and C as follows

$$\begin{aligned} A &= 1 - X - Y^2 + XY_p^2 \\ B &= (X-1)(1-X-Y^2) + \frac{XY^2}{2} - \frac{XY_p^2}{2} \\ C &= (1-X)\left((1-X)^2 - Y^2 \right) \end{aligned}$$

These equations form all the necessary mathematical background we will need to develop numeric point to point ray tracing.

3 Iterative Algorithm

3.1 Newton Raphson Steps

In this optimization, our objective is to find a path with the property

$$\delta P(\alpha_i) = 0 \qquad \text{or} \qquad \frac{\partial P}{\partial \alpha_i} = 0 \ \ \forall \alpha_i,$$

where α_i are the path's parameters. In this formulation, the value P acts as a function from a vector of path parameters α_i to the phase distance of the path described by those parameters. In reality, $P(\alpha_i)$ is calculated using Equation 2 where $\boldsymbol{r}$ and $\boldsymbol{t}$ are functions dependent on both time and α_i.

The Newton Raphson method is an algorithm developed by Joseph Raphson and Isaac Newton to do just that. This method is an iterative method that takes a set of parameters α_i^I at step I and transforms them in to a new set of parameters α_i^{I+1} at step $I+1$. Newton's method works by modling the target function as a linear equation $f(\alpha_i) \to P$ and finding the root of that linear equation

One Newton Raphson step can be written in the form of a matrix equation,

$$\alpha_i^{k+1} = \alpha_i^k - \beta_i^k \qquad \text{where} \qquad \beta_i^k = \mathcal{H}\left(P\right)^{-1} \frac{\partial P}{\partial \alpha_i}.$$

In this system, $\mathcal{H}(P)$ is the hessian matrix of P and $\frac{\partial P}{\partial \alpha_j}$ is the gradient of P.

Each iteration of the Newton-Raphson method will bring the path closer to the correct value. We can continue this iterative method until the path has converged sufficiently. In our implementation, we stop iteration either after a maximum number of iterations, or when the path stops changing significantly at each iteration. For typical paths, this takes anywhere from 2 to 6 newton-raphson steps.

3.2 Initial Path Seed

Our Newton-Raphson solver is an iterative algorithm, so it is important to begin with an appropriate initial path. As suggested by Coleman, we will use the analytic solution assuming a quasi-parabolic atmosphere. This method is outlined in original paper by Croft and Hoogasian [CH68] and further expanded in the follow by Hill [Hil79]. These papers derive formulas for the horizontal distance given the radial position of the arc. Using these formulas, we can can generate an initial path that approximates the correct path.

The exact relationship between Quasi-Parabolic (QP) Atmospheres and Chapman Layers atmospheres was not discussed in any papers, but it is not hard to find a good correspondence between parameters for the Chapman layers and the QP atmosphere. There are three parameters necessary to describe a QP atmosphere, height of maximum electron density, layer semi-width, and critical frequency. Chapman layers atmosphere gives use these three parameters in the form of critical frequency, layer semi-width, and height of maximum electron density. While this coorespondance is not exact, I make the assumption that layer semi-width in the Chapman Layers atmosphere is approximatley equal to the layer semi-width in the QP atmosphere.

To actually generate a path definition from the QP atmosphere, we must first determine the launch angle. In Hill 1979, we see that the following formula can be used to determine launch angle. This formula relates total ground distance D_t to launch angle β_0 and other atmospheric parameters,

$$\frac{D_t}{2} = r_0(\beta_b - \beta_0) + \frac{r_0^2 \cos\beta_0}{\sqrt{C}} \ln\left(\frac{2C + Br_b + 2\sqrt{CX_b}}{r_b\sqrt{B^2 - 4AC}}\right) \tag{7}$$

In this equation, we represent Earth's Radius as r_0, semi-width as y_m, minimum atmosphere height as r_m, operating frequency as f and critical layer frequency as f_c. Other variables can be calculated as follows

$$\begin{aligned}
A &= 1 - \left(\frac{f}{f_c}\right)^2 + \left(\frac{f_c r_b}{f y_m}\right)^2 \\
B &= -2(r_b + y_m)\left(\frac{f_c r_b}{f y_m}\right)^2 \\
C &= \left(\frac{f_c r_b (r_b + y_m)}{f y_m}\right)^2 - r_0^2 \cos^2\beta_0 \\
\beta_b &= \arccos\left(\frac{r_0}{r_b}\cos\beta_0\right) \\
X_b &= r_b^2 - r_0^2\cos^2\beta_0.
\end{aligned}$$

This formula only has one unknown, but is is a non-linear function, so we need to begin with a bounding interval containing the valid $beta_0$ so that we can use iterative methods such as bisection or brent's method.

As a first step, we can calculate the Pedersen ray angle

$$\beta_p = \arccos\left(\frac{\sqrt{-B(r_b + y_m + B/2A)}}{r_0\sqrt{2}}\right)$$

This angle represents the critical launch angle where any angle larger than β_p will never return to the ground.

We now know that $0 < \beta_0 < \beta_p$. Unfortunately, we cannot immediately use an iterative solver because in nearly every system, there are two valid solutions for β_0. We will call these solutions θ_1

and θ_2 where $\theta_1 < \theta_2$. These solutions coorespond to high and low ray paths with θ_1 describing the low ray path and θ_2 describing the high ray path.

To solve for θ_1 and θ_2, we find a critical angle θ_c such that $0 < \theta_1 < \theta_c < \theta_2 < \beta_p$. We can solve for θ_c by realizing that as our launch angle decreases from β_p to 0, the total ground distance decrease from ∞ to some minimum value and then increases again to ∞ when $\beta_0 = 0$. Therefore, θ_c is given by solving the following equation for β_0

$$\frac{\partial D_t}{\partial \beta_0} = 0.$$

This equation is non-linear, but it is immediately solvable by an iterative solver because this derivative is continuous and there is only one root in the interval $[0, \beta_p]$.

Finally, we can now solve for θ_1 and θ_2 by bounding θ_1 in the interval $[0, \theta_c]$ and θ_2 in the interval $[\theta_c, \beta_p]$. We can use iterative solvers to solve Equation 7 in these intervals because there is only one valid launch angle in each interval. In our implementation, we use brent's method because it always converges like the bisection method, but it converges faster in nearly every case.

4 Path Definitions

4.1 Great Circle Deviation

For the purpose of this work, we will define a path to be any function Q defined by parameters α_i such that $Q(\theta) = \boldsymbol{r}$ where θ is a fraction of the path traversed (θ goes from 0 to 1). My tracing algorithm is path agnostic, meaning that any valid function that implements the path protocol (has parameters, a total angle) is valid. In practice, this means that any subclass of the base **Path** class can be used. However, my implementation uses a specific type of path called a deviated great-circle path (GCD)

The GCD path is the path that travels along the great circle connecting a start and an end point. These paths have two types of parameters, radial parameters and angular parameters. Lets call the radial parameters r_i and the angular parameters ϕ_j. For the GCD path, the radial parameters and angular parameters occur at a given path point θ_i or θ_j. While it is not required, my implementation of the GCD path has these radial and angular parameters evenly spaced out over the length of the path. The values of the radial parameters define the altitude of the ray path. The angular parameters represent a lateral deviation from the great circle. Note that these angular deviations are sometimes called normal parameters because they are normal to the primary direction of the path. This formulation is illustrated in the following diagram.

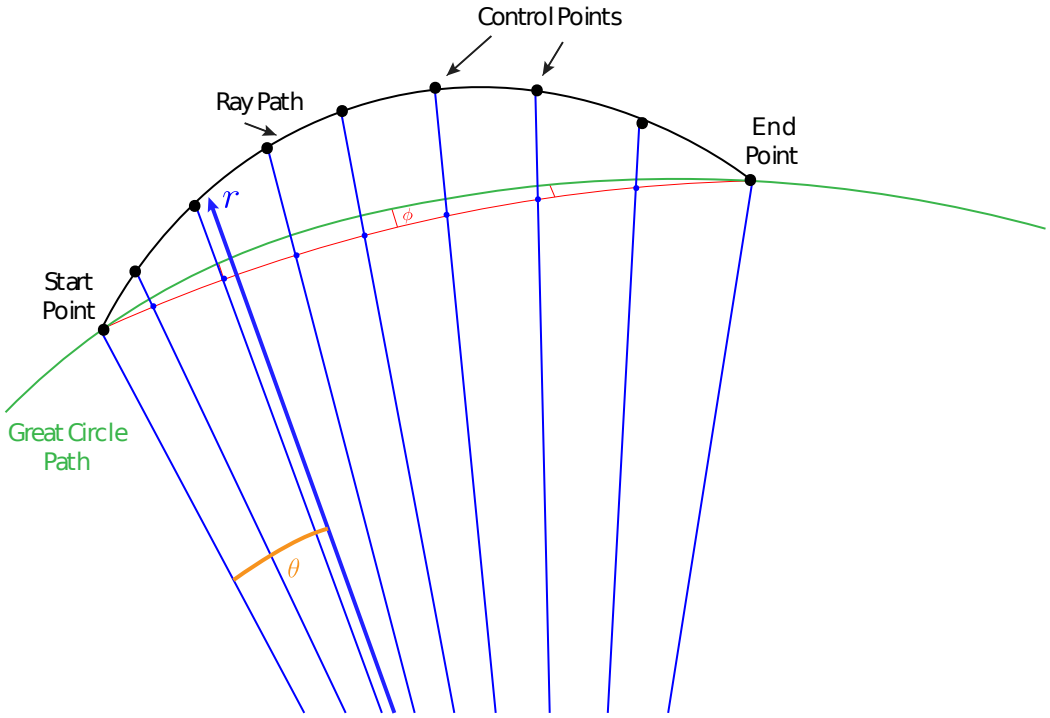


Figure 1: Illustration of Great Circle Deviated Path

In this graph, the angular deviations are represented as the distance from the great circle (given in green) and the red line, while the radial components are given as blue lines.

The number of angular and radial parameters is variable, but there are typically about 10 times more radial parameters than angular parameters, because the path typically varies much more radially than laterally.

Once these parameters are defined, we calculate control points along the path. We begin by creating a cubic spline of angular deviations using Scipy. Call this spline ϕ_S. This spline is shown in red in Figure 1. We then calculate a control point for every radial parameter. These control points are position vectors along the path at even intervals. These control points are calculated by first traveling along the great circle a certain amount θ_i, then moving laterally an amount given by $\phi_S(\theta_i)$, and finally traveling outward a distance r_i from the origin. In the illustration, these control points are shown as black dots along the path.

Now to find the position vector $\boldsymbol{r}$ for any point that is not a control point, we use another cubic spline to find the x, y, and z coordinates for any arbitrary θ along the path. An example of this calculation is given by the blue vector $\boldsymbol{r}$ at orange θ in the illustration.

4.2 Calculating Hessian and Gradient with GCD

Before we begin to calculate the Hessian and Gradient of P, we must first understand how to calculate P itself. P is given by the intergral in Equation 2 over the entire path. To calculate this integral, I reformulate it as

$$P = \int_0^1 \mu(\boldsymbol{r}, Y_p) p_t \frac{ds}{d\theta} d\theta.$$

Note that s is the arclength and $\frac{ds}{d\theta}$ can be calculated by

$$\frac{ds}{d\theta} = \left|\left| \frac{d}{d\theta} \boldsymbol{r}(\theta) \right|\right|$$

Now we can calculate μ, p_t, and $\frac{ds}{d\theta}$ at many values of θ along the path, and integrate through those points using a quadrature method. In my program, I use simpsons method and the implementation given by **scipy.integrate.simps**.

I chose the step size for my integration by trial and error. I ran the algorithm with several different values for step size, and found the one that minimized error while not requiring an obscene computation time.

I also experimented with using romberg integration, but this method wasn't noteably better, so I ended up leaving the simpler simpsons method in my final work.

Now, to calculate the Hessian and the gradient,

To calculate the derivatives of P, the simplest difference method begins with a small number h and then finds $P'(x) = (P(x) - P(x+h))/h$. This formula may remind you of the limit definition of a derivative, and as long as we choose an appropriate h, this result will well approximate the derivative.

For the purpose of our actual algorithm, I know that I will be calculating both the first and second derivative, so this allows me to optimize our choice of difference method. I use the centered derivative for both the first and second derivative. The first derivative calculation is simple,

$$\frac{\partial P}{\partial \alpha_i} = \frac{P(\alpha_i + h) - P(\alpha_i - h)}{2h}. \tag{8}$$

The second derivative calculation is a bit more difficult. We must first define

$$P(\alpha_i - h_i, \alpha_j - h_j) = P_{-1,-1} \qquad P(\alpha_i + h_i, \alpha_j - h_j) = P_{1,-1}$$
$$P(\alpha_i + h_i, \alpha_j + h_j) = P_{1,1} \qquad P(\alpha_i - h_i, \alpha_j + h_j) = P_{-1,1},$$

and then we can calculate the second derivative as

$$\frac{\partial^2 P}{\partial \alpha_i \partial \alpha_j} = \frac{P_{1,1} - P_{1,-1} - P_{-1,1} + P_{-1,-1}}{4 h_i h_j}. \tag{9}$$

I used this method because it's error is $O(h^3)$ while only requiring 3 integrations combined for the first derivative and second derivative where $i = j$. This method also only requires only 4 integrations for second derivative where $i \neq j$. Our function evaluation here for P involves a whole integral (see Equation 2), so we need to limit function evaluations as much as possible.

Our main issue now becomes choosing an appropriate h. I followed some recommendations of the book *Numerical Recipes in C* [Pre+92] to find a value of h that minimizes the combination of truncation error and approximation error. This calculation is simple, but beyond the scope of this report. I will refer the reader to section 5.7 of the book for the full details. I will just leave here that $h = 1$ m for the entirety of this work.

These calculations are the most expensive parts of the entire algorithm, so I have spent the most time optimizing them. First of all I noted that when $|i - j| > 6$, $\frac{\partial^2 P}{\partial \alpha_i \partial \alpha_j} = 0$. This is due to the fact that we are using no splines of higher order than cubic. Additionally I reuse function evaluations whenever I can to avoid reintegrating P. There are still some optimizations I may be able to do here, but I will leave this to future work.

5 Relevant Implementation Details

This computational work will be done in Python. I chose python for its flexibility and advanced scientific computing libraries numpy [Har+20] and scipy [Vir+20].

5.1 Standardized Components

When I developed my code, I was conscious that there are parts to this algorithm that can be modeled in different ways (e.g. QP Atmosphere or Chapman Layers Atmosphere). I did my best to allow for an easy exchange between these models. To achieve this standardization, I created Python base classes for the Atmosphere, Magnetic Field and Path. Each of these classes defines essential features of this type of model. For example a magnetic field model must be able to provide a magnetic field strength and direction for any point in space. This allows the user of this code to replace models with minimal coding. To add a Quasi-Parabolic Atmosphere to the code, all I needed to do was to create a new QP atmosphere class and put that new atmosphere in place of the old one. Even the GCD path is replaceable. Replacement paths need only implement their version of **parameters**, **total_angle**, and **__call__**.

This easy mutability of the code will provide a method for future contributors to easily experiment with different models for fields or atmosphere or even paths themselves.

Currently, I provide implementations of GCD path, QP path, Chapman Layers atmosphere, QP atmosphere, Zero Magnetic field, and Dipole magnetic field.

5.2 Numpy and Scipy

All our vectors and lists of vectors are stored in numpy arrays for speed and efficiency. Additionally, all of our equations will be implemented as vectorized numpy operations. Numpy and scipy operations use compiled C libraries to quickly perform operations on large arrays. This optimization provides significant gains in function evaluation time. Additionally, some core operations like matrix multiply drop down into pure assembly using BLAS or LAPACK functions for greater speed gains.

Numpy provides necessary speed, but its heavy usage means that future developers need to understand how to manipulate arrays. Numpy makes this almost trivial, but there are still some differences in numpy arrays vs regular scalars. For example python can square a variable x like **x**2**. However to square a numpy array element-wise, the code **np.square(x)** should be used.

While numpy provides the core of our functionality, we also use scipy to perform some numeric operations that aren't purely vectorized equations. Our integration uses the scipy routine **integrate.simps**. When we solve nonlinear equations, we use the scipy routine **optimize.brentq**. Lastly, we solve our linear system of equations using either **solve** or **pinvh** from **scipy.linalg**. We use the **solve** method most of the time, but if we are using too many normal component parameters, our matrix can become singular. In this case we switch to using **pinvh** to get a pseudo-inverse of the matrix using the Moore-Penrose pseudo-inverse.

5.3 Multiprocessing

This work requires the repeated calculation of the phase-distance derivative. Each finite-difference derivative requires the calculation of several phase-distance integrals. With even just 50 path components, we end up performing more than 5000 integrations at every Newton Raphson step. Even with all of the optimizations previously discussed, one path solution took 53 seconds on a single core of a Ryzen CPU at 3.7 GHZ. This is not unreasonable, but we can get much greater performance by using multithreading.

Multithreading in Python is challenging because Python uses a Global Interpreter Lock (GIL). This lock prevents concurrent work in any single python instance. Python still provides a multithreading package which can do threaded work, but each thread is forced to pause while the other threads run. These types of threads are useful for I/O applications where one thread may wait for communication while the others work, but they are not useful for our application.

To get around this challenge, python provides a multiprocessing package which allows us to spin up multiple separate python processes to perform concurrent work. While this solves the GIL problem, it presents another challenge because processes can't easily share memory. To perform work in another process, we must transfer all input data into a separate process, perform the calculation, and then return the result to the main process. This interprocess communication isn't very efficient, so we want to find the best way to minimize communication while still balancing the work between a large number of processes effectively. The best choice for our project is to parallelize the derivative calculation. This means passing off each of the 5000 derivatives into our pool of processes and letting each process perform the 2 or 4 integrations before returning the result.

In our code, we use a pool of processes and we call **pool.map_async** to execute our concurrent work. We can pass in functions and input arrays into these pools to perform the necessary work.

Concurrency provides an extra level of complexity to our program, but the performance improvement is significant. We can repeate the path calculation that was previously timed on a 3.7 GHZ, 24 core Ryzen Threadripper and the calculation takes only 2.5 seconds, a factor of 20 speed increase.

The number of parallel processes used for calculating derivatives is controllable, but by default I use 2 less than number of cores in the computer to prevent total gridlock in the computer while running.

5.4 Nonlinear Solvers

Currently, our work contains a tested, viable tracing algorithm for the zero-magnetic-field case. This means it can produce valid traces for any atmosphere as long as the magnetic is zero. This tracing algorithm fails to solve for Y_p and p_t when the magnetic field is non-zero due to challenges with the inconsistency of these nonlinear equations. This issue is the last remaining issue with the full generalized Tracer, and it could be solved with more time.

As a temporary fix, I also developed an approximate solver for the non-zero magnetic field case. This approximate solver addresses the challenges in calculating Y_p and p_t by simply assuming that the medium is isotropic. This way, we avoid the challenges in the nonlinear equations, but we still get a approximate solution to the non-zero magnetic field case. This approximate solution is additionally valuable because we can test extraordinary and ordinary rays using this approximate solver. With a zero-magnetic field, extraordinary and ordinary rays are equivalent.

You can choose which solver to use by toggling the **use_cheater_solver** parameter on the **Tracer** class.

5.5 Visualizations

The primary results in this paper are the path traces. Visualizing these traces and the atmosphere's is important to this project's results. For visualizing the output path, I used matplotlib [Hun07] to perform the relavant visualizations. I chose this library because it is simple and flexible. It also let me overlay the path above the atmosphere color-gradient.

5.6 Documentation

This project does not have a formal Wiki. Instead, please refer to the README and the comments within the code itself.

6 Results

6.1 Code Summary

My work this semester ended with me successfully modeling the path of a ray through the ionosphere using a Quasi-Parabolic atmosphere and zero magnetic field. Additionally, we can model ordinary and extraordinary rays, and both high and low ray paths. We also have an method that approximately solves for ray paths in the non-zero magnetic field case. Any of these solvers can be applied to any atmosphere and any starting path. My code has provided implementations of Zero magnetic field and Dipole magnetic field. It also provides implementations of a Quasi-Paraboilc atmosphere and a Chapman Layers atmosphere with or without gradients. My code has been optimized to run in between 2 and 3 seconds for a single solution. Additionally, it provides the user warnings and errors if the path isn't converging properly or if there are other issues with the trace.

6.2 Working examples

In this section, I will outline several examples of path solutions. These paths use the same start and ending points as Figure 2 in Coleman 2011. I used this path so as to compare with Coleman's approach.

In the diagrams provided the final convergent solution is in black and the previous solutions are in white.

For each of these solutions, I use the same base atmosphere with the following parameters

$$\begin{aligned} y_m &= 100E3 \\ r_m &= 350E3 \\ f_c &= 7E6 \\ f &= 10E6 \end{aligned}$$

where r_m is the height of maximum electron density in the layer, f is the operating frequency, f_c is the maximum plasma frequency in the layer, and y_m is the layer semi-width.

In this section, I will show four variations on this system. Firstly I will show this atmosphere with no gradients, and the zero magnetic field. Secondly I will show the this system with strong gradients of 0.375 MHZ per degree of latitude, which is what Coleman describes in Figure 2 of his 2011 paper. Thirdly, I will show no gradients but with the approximate solution to the dipole magnetic field, and lastly I will show full gradients with dipole magnetic field.

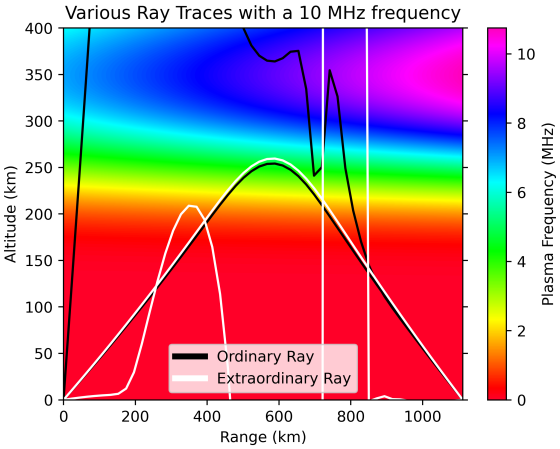


Figure 2: Paths found with Strong Gradients and Dipole Magnetic Field

In this figure you can see that the low ray paths converge to reasonable solutions. To see that these solutions are valid, compoare with Coleman Figure 2 and note the similar shape and structure. These two ray paths are not perfect however. They peak significantly lower than those in Coleman's Figure. Even with this difference however, we can note the similarities. Especially relevant is the fact that our extraordinary path converges slightly higher than its cooresponding ordinary ray path.

These error in these path calculations is due to the fact that the solver assumes an isotropic medium, which is not valid for the non-zero magnetic field case. This error is magnified greatly in the high ray paths, causing them to fail to converge at all. I leave the final path iterations in the graph, but this is not a convergent solution. The iteration only stopped because the error became too large.

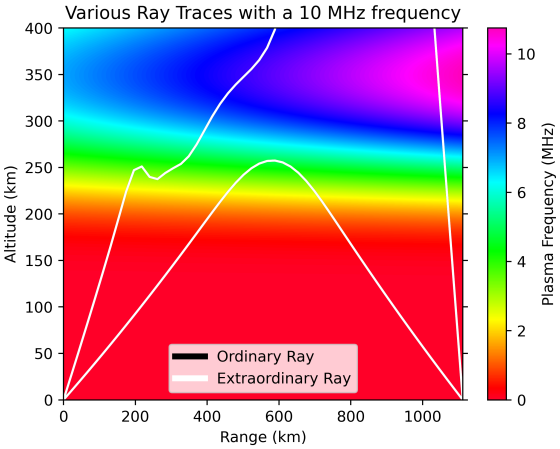
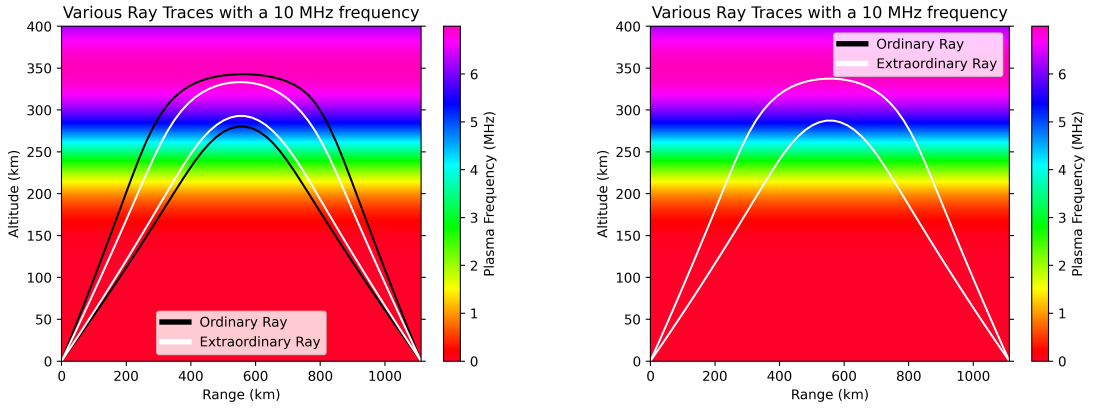


Figure 3: Paths found with Strong Gradients and Zero Magnetic Field

In this figure you can see that the path's are very similar to those in the prevoius figure. This is expected because the atmosphere is not changed. The only change is that we use no magnetic field here.

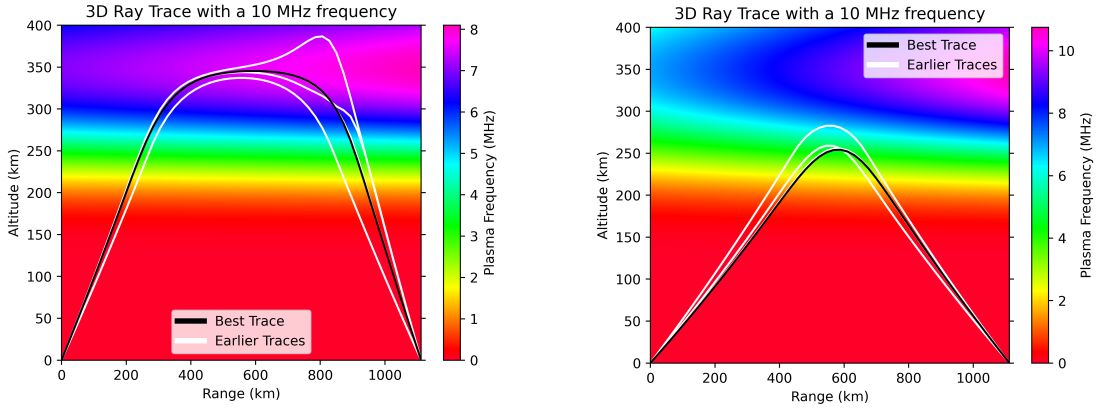
This lack of magnetic field also causes the high and low rays to be equivalent. This equivalence can be see by looking at Equation 5. In the denominator, if $Y = 0$, then $Y_p = Y_t = Y = 0$, and the entire second term goes to zero meaning that the sign of this term is irrelevant.



Dipole Magnetic Field Zero Magnetic Field

Figure 4: Paths found with No Gradients

These solutions to the atmosphere with no gradient exemplify the successes in this method. While we have no figures to directly compare our soultions to, we see every feature we want to see in these paths. First of all, when the field is non-zero, the extrordinary rays are between the ordinary rays. When the field is zero, the ordinary rays overlap with the extraordinary rays. Lastly, the structure of the paths matches our expectations when we compare to the results presented in coleman.



High Path Low Path

Figure 5: Iterative Path Convergence Demonstration with Weaker Gradients of 1.1 MHZ per Degree

In this final figure, I demonstrate path convergence by showing iteration history in white and final path in black. In this figure, it is clear that the final path is truly convergence because the paths around it get closer to the desired result. We also can see in the High path convergence that this convergence is surprisingly robust. Evne though the path veers strongly too high in the first iteration, it can still recover and reach the valid path after 5 more iterations. For reference, the starting is the white trace that is perfectly symmetric in each case. This fact is known because the QP initial path is symmetric. Every other iteration will not be symmetric because the strong gradients break the symmetry.

7 Future Work

The most urgent imrovement needed to this program is a more robust Y_p and p_t solver. The current solver fails when the magnetic field is non-zero. This work will need to take some time to understand where appropriate solutions to Y_p and p_t need to be and find an algorithm that produces valid results and ignores extraneous soultions to Equations 3 and 4.

Future work could also expand the tests of this software to include validation of the algorithms for the non-zero magnetic field case. You could compare the solutions of this iterative method to a direct tracing algorithm and verify that these solutions are valid.

This codebase could also be expanded to explore problems more complex than the simple ground-to-ground problem. For example, this solver could be used to trace rays from ground to satelite. The theory behind the the tracer would extend without adjustment. The only changes needed would be implementations of an appropriate atmosphere model and a new method of path initialization because QP path's wouldn't be valid for this application. Alternatively, this solver could be used to trace 2-hop ray paths. Both of these methods are demonstrated in Coleman 2011, they are not yet implemented in my solver.

8 Conclusions

Over the course of this project, I developed my scientific computing skills and implemented an algorithm laid out in the Coleman 2011 Radio Science paper in a Python program. This software has been developed and tested on various Chapman Layers Atmospheres with both Zero and Dipole magnetic fields. I am leaving this work with a viable codebase, but there are still significant improvements that future researchers could pick up.

9 Acknowledgements

I would like to thank Dr. Kaeppler for his assistance in this research. Without his help and motivation, this would not be possible.

10 Source code

The code for this project can be found on github here.

References

[CH68] Thomas A. Croft and Harry Hoogasian. "Exact ray calculations in a quasi-parabolic ionosphere with no magnetic field". In: *Radio Science* 3.1 (1968), pp. 69–74. DOI: 10.1002/rds19683169.

[Col11] Christopher J. Coleman. "Point-to-point ionospheric ray tracing by a direct variational method". In: *Radio Science* 46.5 (2011). DOI: 10.1029/2011RS004748.

[Har+20] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[Hil79] Jay Roderick Hill. "Exact ray paths in a multisegment quasi-parabolic ionosphere". In: *Radio Science* 14.5 (1979), pp. 855–861. DOI: 10.1029/RS014i005p00855.

[Hun07] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[Pre+92] William H. Press et al. *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing*. USA: Cambridge University Press, 1992. ISBN: 0521431085.

[Vir+20] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.