Numerical Modeling of Atmospheric Vortices

New Mexico

Supercomputing Challenge

Final Report

April 3, 2013

Team 56

Los Alamos High School

Team Members:

Sudeep Dasari

David Murphy

Colin Redman

Mentors

Venkat Dasari

Teachers:

Lee Goodwin

Contents

Executive Summary

Tornadoes such as Joplin, Mo (2011), Spencer, SD (1998), and Dallas, Texas (1957) induce extremely high wind velocities that devastate structures and lift off large objects in their path. Typically, a tornado takes up to an hour to materialize in the form of a narrow axisymmetric vortex and sustains that structure for 10-20 minutes after which the tornado dies down. Near real time simulation of tornado core flow could significantly improve accuracy of Dopplar on Wheels (DOW) radar measurements which in turn would enable more effective warning and evacuation strategies. Presently very simplified vortex models, such as the algebraic Wood-White model and the modified Rankine model, are being used to simulate radar signatures necessary to reconstruct real world tornadoes with limited success. Other alternatives, such as commercial CFD tools, may provide more accurate results but take long periods of time to compute. We believe that mathematical frameworks developed by Drs. Sullivan and Kuo provide an optimum approach to bridge this gap because they are based on fluid-mechanically and thermodynamically self-consistent physics to model atmospheric vortices with sufficient accuracy. Implementation of their models required the solving of multiple and coupled non-linear boundary value fourth order ordinary differential equations (ODEs) using a fourth-order iterative Runge-Kutta method (RK4). Because certain sub-models consisted of differential-integrals, we had to effectively implement Euler integration and Newton's finite differentiation algorithms, as well as create a coordinate transformation system to easily move data from polar coordinates used for tornado computations and the Cartesian system for all other operations. Combining all these methods and building on the works of scientists such as Kuo and Sullivan we created a series of programs that accurately simulated vortex flow, examined dynamics of particles with pre-defined mass and drag coefficients when subjected to said vortex, and exported the data for use in visualization with Paraview, a multi-platform scientific visualization software being developed by Sandia and Los Alamos National Labs. Furthermore, we validated the models by comparing their predictions for wind velocities with measurements reported in the literature for the 1998-Spencer and 1957-Dallas tornadoes. A surprisingly good comparison was observed: which established validity of our implementation of Kuo's models. Once these computer models were built, it was quickly realized that they can be very time consuming by modern standards and at the same time had tremendous potential for successful optimization work. As such, we conducted a series of optimization tasks designed to speed up the execution of our code including: (a) utilized the graphics processing unit (GPU) to accelerate the RK4 and Euler integration computations, and finally the entire Kuo model, (b) used CPU multi-threading feature to accelerate the particle tracing algorithm; and (c) implemented a new data format system to reduce the size of the exported file and decrease write times. The optimization tasks proved to be extraordinarily effective, as the GPU optimization dropped computing time by over 2000 percent, CPU multithreading proved to shave significant amounts of time from the particle

computing, and the data formats allowed the writing of large volumes of data while taking smaller amounts of disk space as well as write time.

Introduction

Phenomenological description of an atmospheric vortex

There exists much complex physics behind the behavior and physical characteristics that make up atmospheric vortices, such as hurricanes, tornadoes and dust devils. At a basic level, such vortices form when heavier colder Arctic air finds itself atop of lighter warm air from the Gulf of Mexico, and as a result is forced to sink toward the ground, while the warm air is forced to rise, thus causing instability. This situation is akin to a bath tub filled to the brim with water with a stopper preventing the water from draining into the pipes. When the stopper is pulled the water begins to drain, but since the water is attempting to enter the drain from multiple different angles a circular motion, or vortex, is observed, as it represents the simplest way for all the water in the tub to exit. In an atmospheric vortex the colder air at a higher altitude begins to fall rapidly while the warm air rises, and for the same reason the water in the bathtub began to spin, the column of air begins to spin as the warmer air makes its ascent. As described by Neuenshwander, tornadoes form when an exceptionally powerful thunderstorm collects enough heat and moisture to form a supercell, which tips over to a horizontal orientation when it hits sufficient horizontal drafts, thus forming a mesocyclone. The mesocyclone then tumbles through a series of not fully understood transformations and at the other end emerges a tornado [1]. This transformation, aka tornado-genesis, might take several hours, but the tornado-vortex itself only lasts tens of minutes during which time it causes immense destruction. It is this intense phase of tornado that we intend to simulate.

Once a tornado is formed other factors affect it as well, such as pressure differences throughout the vortex which in turn are a reflection of the buoyancy forces induced by temperature differences. Generally, a more powerful tornado will have a larger difference between temperature of the cold air up top and the warm air closer toward the bottom. This heat gradient, referred to as the suction strength or buoyancy loading, generally describes how powerful the forces driving the tornado will be and the greater the gradient the more powerful the tornado. Other forces that determine the intensity of the tornado include the amount of energy a block of air would have when it's raised skyward, or its convective available potential energy (CAPE) which is also related to the circulation intensity, and finally the eddy viscosity which is an artificial construct to simulate the 'solid-like' core region of a tornado. Despite these complex phenomena measured wind velocities exhibit a rather simple structure. Many twisters tend to be axisymmetric even when extraordinarily high intensity tangential winds are involved. Further velocities in the core region tend to increase steadily as the radius from the center of the tornado increases until a maximum point is reached, after which the velocities decrease steadily until they return to a baseline. Often meteorologists such as storm chasers collect important data about the tornado such as; the CAPE, suction strength, and wind speeds. Such measurements are crucial, not just to for assessing the power of tornadoes, but for the purpose of understanding the advanced mechanics present in a tornado through media such as a computer model.

Tornado Morphology: One Cell vs. Two Cell Tornadoes

The tornadoes we analyzed vary between two basic forms: one cell and two cell tornadoes. A one-cell tornado has a single column of upward flowing air in which the most powerful winds occur, and is surrounded by an area of downward flowing air with slower wind speeds. In a two cell tornado, there is a core of downward flowing cold air surrounded by a ring of upward flowing air. Thus there is a center in which the velocities in the z direction are negative (that is flowing downwards), and a ring outside the center in which the z velocities are positive (that is, flowing upwards). Just like in the one-cell tornado, the velocities in two-cell tornado also die down the further away you go from the core of the twister [2]. Figure below illustrates these phenomena. Note that foci of our simulations are the three drawings on the left side of the figure.

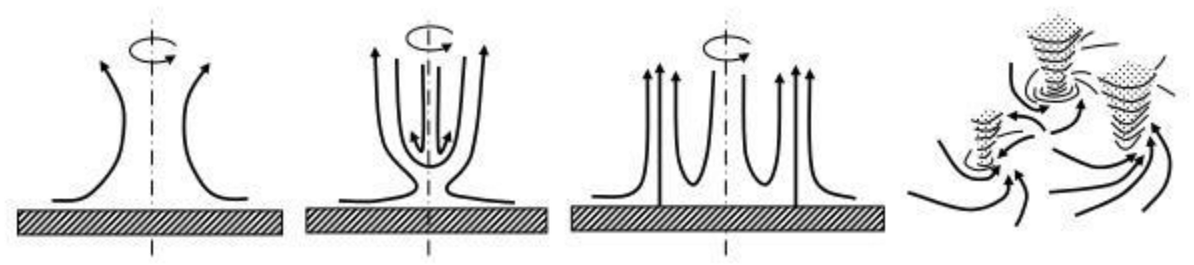


Figure 1: Diagram of one cell tornado on far left, and multiple cell tornadoes in middle. Far right describes multiple tornadoes in a single super cell – like those occurred in Texas in 1957. Diagram is from Church et al [2]

Mathematical Models

Although theoretical models of tornadoes have existed since the 1950s they were not accurate enough to yield any useful results when parsed into a computer model. Experimental work conducted by Ward [3] provided fundamental understanding into the underlying physics behind a tornado and helped pave the way for new and improved theoretical work and models. Improved measurements coupled with modeling resulted in a definitive and scientific work by Davies-Jones [4]. We rely on this and similar publications by scientist such as Drs. Kuo and Sullivan to develop our computer model.

For the most part, modern computer modeling and simulation of tornadoes has taken two parallel paths. The first technique relies on the computational fluid dynamics (CFD) codes to simulate 3-D aspects of a developing tornado which was first demonstrated by Harlow and Stein from Los Alamos National Laboratory [13]. Recently Sarkar et al [5] utilized commercially available CFD code Fluent 6.0 to create a model twister and then they compared their predictions to data from the 1998-Spencer tornado. Their approach, while accurate, was incredibly time consuming taking in excess of a day to run on a powerful workstation. It was also extremely sensitive to the boundary conditions and did not capture the storm thermodynamics accurately.

As described by Thakar et al, [6] another alternative is to take advantage of the axisymmetric nature of a vortex flow which allows one to reduce the governing Navier-Stokes equations into a set of non-linear coupled ODEs. This is the approach followed by Sullivan and Kuo who generated models upon which our own computer simulations are based around. Yet another approach that has been gaining ground recently due to its flexibility is that developed by Wood and White [10]. This particular approach is referred to as a parametric rational function and is developed for the sole purpose of real-time simulation of radar signatures generated by a tornado and to reconstruct tornado structure from the radar measurements. But what it gains in flexibility it loses in accuracy, for example predicted velocities tend towards infinity within the core of the tornado. It is notable that Kuo's, Sullivan's, and Wood-White's models take advantage of the axisymmetry and are written using a polar coordinate scheme shown to the left in figure 2. The boundary conditions of the models are also shown as well.

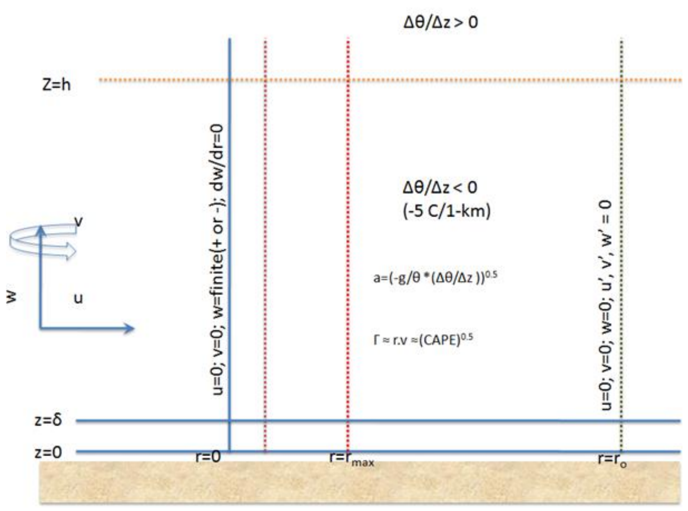


Figure 2: Displays diagram of polar coordinates and boundary cond.

Kuo's Model

In 1966, Dr. Kuo proposed a tornado model that is directly solved from the Navier Stokes equations. As described by Kuo his model is based on "a system of simplified yet sufficiently accurate equations adequate by expanding flow variables". These variables come in the form of "two atmospheric parameters that are related to driving energies that are caused by the unstable stratification and the vorticity that determines tornado radius". Unlike his contemporary scientists, Dr. Kuo [8] modeled not only the fluid mechanics of the tornado but their overall thermodynamics through the use of an energy conservation equation. It is notable to mention that the Kuo's model transitions from the two-cell vortex to a single-cell vortex by simply changing the boundary condition. The underlying ODEs are as follows:

$$x\frac{d^3F}{dx^3}+(F+1)\frac{d^2F}{dx^2}-(\frac{dF}{dx})^2+\sigma=0$$

$$\frac{K1}{v1}*\frac{d}{dx}\left(x\frac{d\sigma}{dx}\right)+F\frac{d\sigma}{dx}-(\sigma-1)\frac{dF}{dx}=0$$

$$x\frac{d^2m}{dx^2} + F\frac{dm}{dx} = 0$$

Where, x, F, σ, and m are non-dimensional parameters that are related to the physical parameters as follows:

Radius, r, is related to x as $r = \sqrt{\frac{4\nu x}{\beta}}$

Tangential Velocity (v) = $\Gamma \cdot \frac{m}{r}$

Radial Velocity (u) = $-10 \cdot \nu \cdot \frac{F}{r}$

Axial Velocity (w) =$\beta \cdot z \cdot (\frac{df}{dx})$

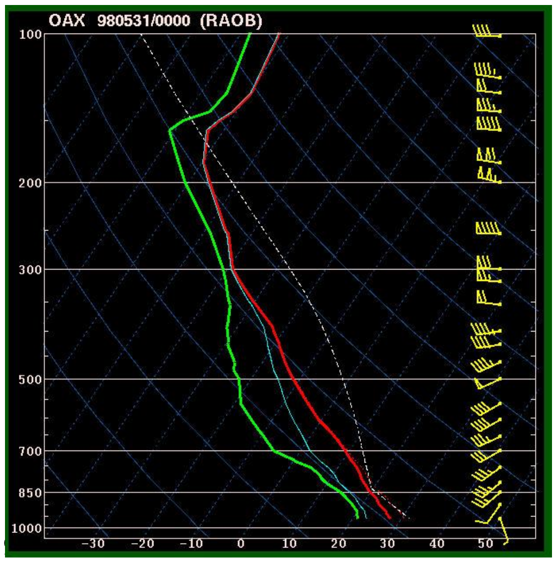
Where,

- ν is eddy viscosity (m^2/s)
- β is buoyancy related suction strength (#/s)
- Pressure =$P_a + \frac{\beta \cdot \rho \cdot \Gamma \cdot r}{8\nu} \cdot \int \frac{m^2}{(\frac{\beta \cdot r \cdot r}{4 \cdot \nu})^2} dr$
- ρ is the density of the air (kg/m^3)
- P_a is the ambient pressure (Pa)
- Γ is the circulation strength (m^2/s)

Suction strength, β, and circulation intensity, Γ, are related to temperature (θ) and ceiling height (h) as follows

$$\beta = [-\frac{g}{\theta}\frac{d\theta}{dz}]^{1/2} \text{ and } \Gamma = 28.91\sqrt[3]{\frac{\beta \cdot h^2 \cdot \nu^2}{4}}$$

Weather stations across the mid-western states monitor conditions in the super cell, especially Pa, ρ, θ and h regularly. Below in figure 3 is an example of a sounding graph used by meteorologists during 1998-Spencer tornado. Recorded readings illustrate the instability as shown by the slop of the red line, which is negative initially up to a pressure of 157 milli-bar (also 10061 ft). Negative pressure indicates that density is increasing with elevation which is inherently unstable. The stable layer exists above 10,0061 feet. For this storm the estimated β values ranged between 0.01 and 0.02 and Γ values ranged between



7500 and 9000 m^2/s. Those values were used for validating flow velocity predictions by our model.

Figure 3. Omaha, Nb, sounding graph
on 31/May/98 Spencer Tornado

After proposing his ODEs, Kuo found an approximate closed form solution to the equations above to facilitate easier computation of velocities. His analytical solutions are as follows.

$$m = \frac{\int_0^x e^F\,dx}{\int_0^\infty e^F\,dx};\ w = \beta z(1 - ze^{-x});\ v = \Gamma \cdot \frac{m}{\sqrt{2}\cdot r} \text{ and } u = -\beta r\left[0.5 - \frac{1}{x}(1 - e^{-x})\right]$$

It is easy to see that these equations can give rise to non-physical predictions as x→0 u, v→∞. While less accurate than the ODEs, closed form solutions offer an easier way to craft a similar prediction using less computing time and allowing for a simpler code.

Sullivan's model

R.D. Sullivan, a contemporary of Kuo, followed a similar procedure to simplify the Navier-Stokes equations into a set of coupled ODEs, and then derived a set of analytical expressions. Sullivan ODEs were the subject of a 2009 Master's Thesis by Mr. Baker [7], who concluded that Sullivan's equations can be extremely unstable, stiff, and cannot be generalized easily at low eddy viscosities [7]. We independently concluded that Sullivan ODE equations become unstable during the shooting method implementation of RK4. So we chose to use his closed form solution instead, which offers near similar results but is less accurate than the ODEs. These equations also give rise to non-physical predictions as x→0 u, v→∞.Following, are the closed form solutions [9].

$$u(r) = -\beta r + \frac{6\nu}{r}\left(1 - e^{\frac{\beta r^2}{2\nu}}\right);\ v(r) = \frac{\gamma}{2\pi r}\frac{H\left(\frac{\beta r^2}{2\nu}\right)}{H(\infty)};\text{ and } w(z,r) = 2\beta z\left(1 - 3e^{\frac{-\beta r^2}{2\nu}}\right)$$

$$P(r,z) = P_a + \rho\int_0^r \frac{v^2}{r}\,dr - \frac{\rho\beta^2}{2}(r^2 + 4z^2) - \frac{18\rho\nu^2}{r^2}\left(1 - e^{\frac{-\beta r^2}{2\nu}}\right)^2$$

$$H(x) = \int_0^x e^{\int_0^x 3e^x\,dx}\,dx$$

Wood-White Model

The Wood-White model, written in 2010, is a parametric model that can be fit to the actual measured tangential velocities and then inversed to estimate other parameters of interest including u, w, pressure, circulation strength, and buoyancy related to suction strength.

$$v(r) = \frac{r^k}{(1+r^{\frac{n}{\lambda}})^{\lambda}};\ \in = \frac{dv}{dr} + \frac{v}{r};\ u = \nu\left(\frac{\frac{d\in}{dr}}{\in}\right);\ \text{and } w = -\left(\frac{du}{dr} + \frac{u}{r}\right) * \in$$

Where the parameter ν represents the eddy viscosity of the vortex at hand, and n, λ , k are parameters changed by the user to fit the v vs r graph the model generates to one observed in real life. The u, and w are then solvable once the v value is derived [10].

Problem Statement

Multiple fundamental problems still exist with the modeling of atmospheric vortices, such as tornadoes and dust devils, that this projects attempts to address. Several modeling options exist for simulating tornados, as shown in Figure 4. The primary models in use currently for the modeling of such phenomenon are the Rankine models originally developed in 1800's. This simplified model does not account for two-cell vortices and was eliminated from further consideration in our study. On the other hand, commercial CFD codes could be used to obtain accurate predictions, but they take long periods of computational time, on the scale of hours or even days. Further, utilizing CFD codes would not give this team the opportunity to explore the physics behind a tornado or engage in optimization tasks to the extent with which we desire.

The dynamics of a fully developed tornado – as confirmed by most recent Doppler on Wheels radar imaging – to a first-order approximation are very similar to steady axisymmetric vortices. Under such an assumption the governing Navier Stokes equations have been reduced into a set of coupled non-linear ordinary differential equations by Drs. Kuo and Sullivan. We have chosen mathematical framework of these two researchers in our study. Though they were originally postulated in 1960s, as demonstrated by Baker in 2010 [7] solutions to these ODEs are not straight forward, and are complicated by the fact that the boundary conditions are ill defined, the ODEs are stiff, and the geometry is extremely large. Our objective is to employ modern computation algorithms to solve ODEs that include Eulerian fluid mechanics equation, as well as Lagragian particle transport equations, which come out to around 20 equations altogether. These equations formed the basis for our computer model that can: (a) in near real time simulate an atmospheric vortex with sufficient accuracy, (b) utilize its predictions to predict how debris or other objects would move through said vortex, and (c) then export the data for visualization in the state-of-the-art scientific visualization software Paraview. Upon verifying code predictions by comparing with the approximate solutions presented above, we extended the code to simulate the 1998-Spencer tornado and 1957-Dalls tornado. These simulation results were compared with

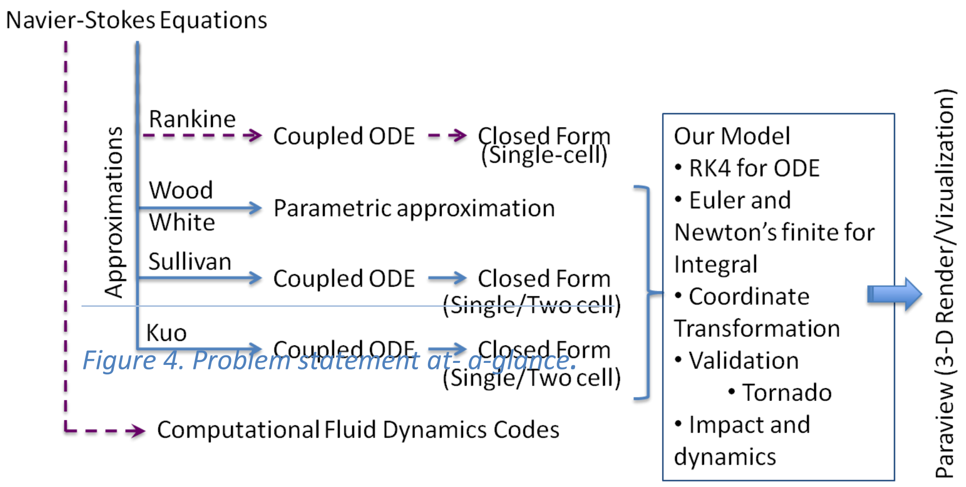

Navier-Stokes Equations
Approximations
Rankine
Coupled ODE
Closed Form (Single-cell)
Wood
White
Parametric approximation
Sullivan
Coupled ODE
Closed Form (Single/Two cell)
Kuo
Coupled ODE
Closed Form (Single/Two cell)
Computational Fluid Dynamics Codes
Our Model
• RK4 for ODE
• Euler and Newton's finite for Integral
• Coordinate Transformation
• Validation
• Tornado
• Impact and dynamics
Paraview (3-D Render/Vizualization)

Figure 4. Problem statement at-a-glance.

the actual measurements as a step towards validating the code. Then we optimized the created computer code through the use of the GPU, CPU multithreading, and data format optimization.

Problem Solution

Overview of Methodology

As shown in Figure 4, the tasks necessary to create a model that would fulfill our expectations were broken into four main categories: the RK4 loop that would be used to solve ODEs that describe the fluid flow in an Eulerian frame, Euler's method for integration especially to compute pressure once velocities are known, Newton's method for differentiation used to compute axial and radial velocities using the Wood-White model, and of course a coordinate transformation system that switches between polar and Cartesian coordinates, and a Lagrangian particle tracer program. We employed most if not all of these steps in the writing of separate programs, each built to utilize either Kuo's, Sullivan's, or Wood-White's methodology. We utilized a combination of Python and Java for most of the implementation of the solution. C was utilized mainly during the optimization phase of our program, which will be elaborated on in the optimization section of the report.

Fourth Order Runge-Kutta Loop and Shooting method

The fourth order Runge-Kutta method (RK4), also known as the classic Runge-Kutta method, is used extensively throughout our various programs, be it solving the ODEs behind Kuo's equations or handling the calculations necessary for the Lagrangian particle trace system. Assuming y_n is a scalar or vector value, t is a scalar to correspond to a time or x value with which we are approximating, and h is the step with which we are sampling, then given a function f which represents an ODE, we can utilize the RK4 methodology to solve for y_{n+h} and t_{n+h}. using the following equations.

$$y_{n+h} = y_n + h\frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$t_{n+h} = t_n + h$$

$$k_1 = f(t_n, y_n)$$

$$k_2 = f(t_n + \frac{1}{2}h, y_n + h\frac{1}{2}k_1)$$

$$k_3 = f(t_n + \frac{1}{2}h, y_n + h\frac{1}{2}k_2)$$

$$k_4 = f(t_n + h, y_n + hk_3)$$

We adopted this method on multiple occasions into Java, Python, and C most notably for solving the ODEs behind Kuo's equation. First, in order to solve Kuo's model the ODEs described in the introduction under Kuo's model were re-written as follows.

$$F' = \frac{dF}{dx}, F'' = \frac{dF'}{dx}, F''' = \frac{-(F+1)F'' + F'^2 - \sigma}{x}$$

$$\sigma' = \frac{d\sigma}{dx}, \sigma'' = \frac{-\left(F + \frac{k}{v}\right)\sigma' + F'(\sigma - 1)}{x},$$

$$m' = \frac{dm}{dx}, m'' = \frac{-Fm'}{x}$$

The variables F, σ, and m were solved after being subjected to the following boundary conditions.

$$@x = 0\ F(0) = 0.0\ F'(0) is\ finite\ such\ that\ F'(\infty) \rightarrow 0\ and\ F''(0) = F'(0)[F'(0) - 1]$$
$$@x = 0\ \ m(0) = 0\ and\ m'(0) is\ finite\ such\ that\ m'(\infty) \rightarrow 0$$

Because the RK4 loop is not capable of solving such a boundary value problem, we utilized a shooting method that iterates over various guesses for F'(0) and m'(0) until the boundary conditions are satisfied. We used a self implemented bisect root finding algorithm to find the appropriate initial values, that were then stored into an array of type double and size five, and filled in the order of [F,F',F'',m,m']. At each gridpoint, the first step is to compute dimensionless distance *x* as given by $\frac{r^2\beta}{4v}$ where, r is distance the gridpoint is from the center (that is, $\sqrt{x^2 + y^2}$). Then the vector [F, F', F'', σ, σ'', m, m'] is computed at *x* using the RK4 loop. The F, F', and m elements of the data set are then used to calculate the u, v, w, and pressure values for the grid point at hand; σ is used to compute logarithmic temperature (which is not used much in our analyses, hence not shown in our code). Since, as one may imagine, the RK4 loop is at times quite computationally intensive we took advantage of the axisymmetric nature of the problem yet again. Instead of calculating the vector at each gridpoint, we split the physical grid into four quadrants and only calculated the points at quadrant one, before any other values were calculated. As the F, F', and m values are the only ones used later in calculations only they were stored into their own two dimensional arrays (i.e. all F values were stored in an array named F) and then were reflected across the axis. After all the RK4 calculations were finished we continued to calculated the u, v, w, and pressure values at each gridpoint, and the program accessed the needed F, F', and m through the array they were contained in. The actual code that compromised the RK4 loop and shooting method can be found in Appendix A, under the Java section. Either KuoTest.java or KuoTestMP.java will contain it.

Lagrangian Particle Tracing Method

The particle tracer method is designed to determine how a vortex as modeled by our simulator would affect its surroundings, such as debris or other objects in its path. This feature is important because Doppler on Wheels radars often rely on reflections from the debris to resolve flow fields within the core region. Essentially, the method sets up a particle with a certain mass, area, and drag coefficients, and starting (x,y,z) coordinate as well as a starting <Vx,Vy,Vz> velocity vector. The path the particle would follow when subjected to a flow field is described by Newton's equations of motion, namely F=ma. Mathematically then, force is expressed in vector notation as:

$$\vec{F_x} = M_p\left(\frac{d\vec{V}_{oxp}}{dt}\right) = \left(\frac{d^2\vec{x}_{op}}{dt^2}\right)$$

Force is caused by the drag and further expressed as:

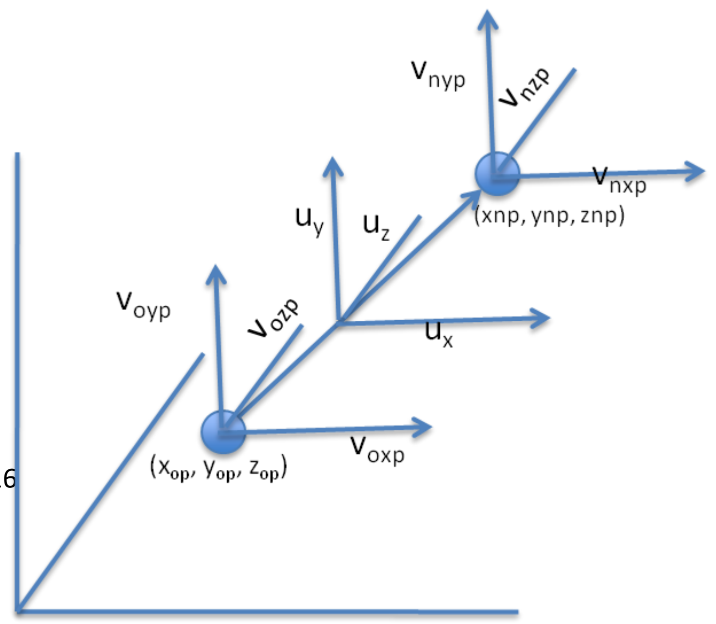
$$\vec{F} = \frac{1}{M_p}\cdot\rho_{air}\cdot C_d\cdot A_f * \left(\vec{V}-\vec{V}p\right) * \left|\vec{V}-\vec{V}p\right|$$

Resulting in over all force balanced equations of:

$$F_x = \frac{1}{M_p}\cdot\rho_{air}\cdot C_{dx}\cdot A_f\cdot(Vx - Vxp)\cdot|Vx - Vxp|$$
$$F_y = \frac{1}{M_p}\cdot\rho_{air}\cdot C_{dy}\cdot A_f\cdot(Vy - Vyp)\cdot|Vy - Vyp|$$
$$F_z = \frac{1}{M_p}\cdot\rho_{air}\cdot C_{dz}\cdot A_f\cdot(Vz - Vzp)\cdot|Vz - Vzp|) - 9.81$$

Where:

- $Vx, Vy, and\ Vz\ are\ the\ x, y, z\ components\ of\ the\ air's\ velocity\ respectivly$
- $Vpx, Vpy, and\ Vpz\ are\ the\ x, y, z\ components\ of\ the\ particle's\ velocity\ respectivly$
- $CD_x, CD_y, CD_z\ are\ the\ drag\ cofficients\ of\ particle\ in\ x, y, z\ components\ respectivly$
- $M_p\ is\ the\ mass\ of\ the\ particle, and\ A_f\ is\ the\ area\ of\ the\ particle\ normal\ to\ flow$



The following three equations are used with the previous three in the RK4 loop to trace the path of the

Figure 5: Describes the various forces that affect movement of a particle in a flow field

$$vxp = \frac{}{dt}, \qquad vyp = \frac{}{dt}, \qquad vzp = \frac{dzp}{dt}$$

The initial conditions employed by the methods are the initial (x,y,z) coordinates of the particle and its initial <Vx,Vy,Vz> velocity values. Then an initial time value of 0 is set and the maximum time value of (in this case) 250 seconds is initialized. Then the RK4 method moves from the initial time value to the maximum at an interval of 0.01 seconds, while employing the previous six equations to determine the particles (x,y,z) and <Vx,Vy,Vz> values at each timestep. The values are then stored into their own array so that the [x, y, z, Vx, Vy, Vz] values for the particle at each timestep can then be exported into a .csv file and visualized in Paraview. This particular method was only written in Java and a full listing of the code utilized in the particle trace can be found in Appendix A, under the Java section. Sullivan.java, KuoTest.java, or KuoTestMP.java will contain the method but only KouTestMP.java will contain a multithreaded variant.The area of the particle was subsumed into the drag coefficient in our code.

Euler and Newton's Finite Differentiation

Throughout all three models; Kuo, Sullivan, and Wood-White, we were often forced to integrate functions. In Kuo's and Sullivan's models the Euler integration method was used to solve the function necessary to find the pressure in each grid cell. Furthermore, the H function that is used throughout Sullivan's model depends on Euler integration in order to find a valid solution. The Wood-White model depends on Newton's finite differentiation methods heavily in order to invert the algebraic approximation and yield results other than the v unit vector value.

Coordinate Mapping and Matrix Transformations

In order to yield results that could be successfully used for 3d visualization in a program such as Paraview, we had to first create a grid system that would serve as a framework for literally everything that followed. The most logical choice was to utilize a set of three dimensional arrays of type float to store the data for each gridpoint. However, since we aspired to simulate an entire vortex, not just the core or outskirts, we would have to set up a rectangular grid space around the size of 2 km by 2km by 1 km, in our program, with the center of the tornado being positioned in the middle of the grid at z=0. As one might imagine such a large array (2000x2000x1000 elements) would be prohibitively expensive, are far as computing time is concerned, so we came up with a compromise that would allow us to maintain a large world area while still keeping a manageable array size. We created an Rmax variable, a resolution variable, and a deltaR variable, the first of which dictates the maximum distance away from the center of the tornado our gridworld world will evaluate to in real world units, the second dictates how many gridspaces that our program will have from the center of the tornado to the Rmax, and the third is represented by Rmax/resolution. For example, say that you want to evaluate the vortex to 1000 meters away from you origin (i.e. Rmax=1000) in the x direction, 1000 in the y, and 1000 in the z, at a resolution of 50. The program would create an array of 100x100x50 elements and divide the Rmax by the resolution to obtain a deltaR of 20. Each element of the array symbolizes a 3d grid space in the simulated world. However, if the coordinates of the gridspace say (1,1,1) must be multiplied by the deltaR value to translate them into real life values that can then be inputted into the various models.

Yet another hurdle that must be overcome before our array system can actually symbolize a real three dimensional grid is the fact that the coordinates (0,0,0) are not the same as the array element [0][0][0]. Since array elements can never go negative, assuming the resolution is 50, the element [0][0][0], is actually symbolized by the coordinates (-50,-50,0). In order to get around this you must subtract the resolution from the elements x,y,z values in the array, and then multiply by deltaR to get its x,y,z values that would correspond to real life. Thus element [0][0][0] would correspond to (-1000,-1000,0) in real life. (-1000,-1000,0) would then be fed into the various subroutines in the program to find the u,v,w and pressure the simulated world would experience at that point. It is noteworthy that z never goes below zero in real life, so the z coordinate need not go through any correction arithmetic other than being multiplied by deltaR.

Finally, the models we depend on utilize a polar coordinate scheme while the Paraview visualization software, and really most programs, utilize a Cartesian scheme. Thus the array we construct to store our values utilizes a Cartesian scheme, and utilizes polar to Cartesian translations to allow the data to yield meaningful results. Equations used for translation are as follows: $r = \sqrt{x^2 + y^2}, \theta = \tan^{-1}\left(\frac{y}{x}\right), Vz = w, Vx = ucos\theta - vsin\theta, Vy = usin\theta - vcos\theta.$ Relationship among these variables is shown in Figure 6.

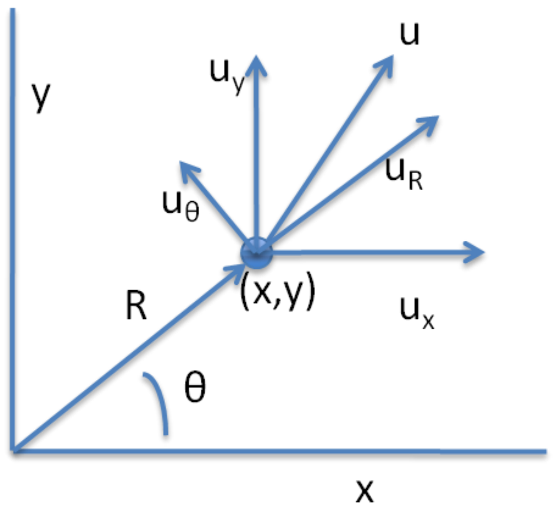


Figure 6: Illustration of coordinate transformation used in our model.

Once all these elements are put together, the following process is constructed as to how the program iterates through points inside its grid. The program starts out at the array element [0][0][0] and then iterates to the elements [resolution*2][resolution*2][resolution]. At each array element the x,y,z values corresponding the array indices are subtracted by resolution and multiplied by deltaR to get their real life values. Then these real life values are used to determine the u, v, and w unit vector values at each of the real life gridspaces. The Cartesian to polar coordinate transformations are necessary in the above step to translate the x,y,z values r and theta values which will then be used by the equations outlined by Kuo and Sullivan, as well as the transformations back from polar to Cartesian coordinate schemes. Then all the derived data is exported out under a Cartesian coordinate scheme for use in visualization programs such as Paraview. The polar unit vectors and r values are exported to in case one wishes to compare them to other polar data. The array system is seen throughout the code in the Sullivan and Kuo models written both in Java and C, listings of the full code can be found in Appendix A under the Java nd CUDA sections.

Exporting and Visualization

We utilized two programs to handle the bulk of our visualization work, Microsoft Excel and Paraview. The first of which is a basic spreadsheet application useful for doing basic visualization and graphing work, the second is an open-source, multi-platform, interactive 3D visualization program developed by Sandia National Laboratories, Los Alamos National Laboratory, the US Army Research Lab, and Kitware. Using Excel we created a series of v unit vector vs. r graphs to be used for comparing the various models as well as validation with real life tornadoes. Paraview was used for the bulk of the visualization including some graphing of the pressure, u, v, and w variables vs r, generating streamlines and glyphs to provide a rich and graphical way to visualize our results, as well as offering a simple way to show the path our particles would have travelled in 3D.

For a good deal of time, the program exported the data utilizing the comma separated value (CSV) format which was slow and took large amounts of disk space. To conserve time we switched over to a MySQL database system that both sped up write time, and reduced disk space. We will elaborate more in the optimization section.

Real World Dust Devil Simulator

In order to validate the output of the computer model, small-scale dust devils were formed in a vortex generator. The vortex generator is an apparatus designed to mimic the natural formation of dust devils in a controlled environment. The updraft within the generator was held constant across all trials. The circulation of air within the generator was manipulated to determine relationships between vorticity and tangential velocity.

Tangential velocity within the generator was determined with a process known as particle image velocimetry. A small Styrofoam ball was dropped into the apparatus and, using a strobe light and a long-exposure camera, pictures were taken which showed the ball’s location over a regular timestep. These images were analyzed with a basic computer program to determine the tangential velocity of the dust devil. We ran out of time before we could utilize this vortex generator as further validation of our computer model. We plan to integrate this into the presentation we will give at the expo.

Optimization

We conducted three main optimization tasks with the goal of significantly decreasing the execution time, as well as the disk space and general computing footprint our program utilized. The tasks include, GPU optimization of the Euler integration loops as well as the RK4 method and Kuos model, CPU multithreading of the particle trace system used throughout our program, and optimization of the storage format used to store our data, namely through the use of a database rather than CSV files. While all the modes of optimization did in fact utilize available resources more effectively and resulted in faster run times, and smaller data footprints by far the most successful was the GPU experimentation judging by nothing other than the huge amounts of times saved, while returning the exact same results.

GPU optimization

A GPU unlike a CPU has massive opportunities for parallelization that makes porting over a program such as our own especially enticing. While it may seem to be a difficult task to parallelize our model, it is in fact quite easy due to the lack of interdependencies between the points in our model. As far as our program is concerned the gridspace (1,1,1) and the gridspace (2,2,2) are completely separate from each other. Taking advantage of this trait we utilized the Compute Unified Device Architecture (CUDA) framework developed by the company Nvidia in an attempt to speed up our code using GPU (it is noteworthy to mention that CUDA code will only work on an Nvidia GPU with CUDA cores). On the most basic level, our program is iterating through an array, performing mathematical operations on each element of the array, and then storing the results back into the array to be exported. CUDA offers a block and thread framework that allows us to take advantage of this trait. Essentially a block is a group of threads that run on one multiprocessor on the GPU. Each multiprocessor contains a set of stream processors upon which one or more threads run on. Blocks are completely independent from one another and threads have limited synchronization abilities which are useful in only certain situations. The program to be run on the GPU is referred to as the kernel, which should be written in such a way that the task contained in each can be split across multiple blocks and threads which are then executed simultaneously thus achieving a major speed up from a CPU implementation of the same code. We take advantage of this feature by first initializing a set of arrays using the same coordinate mapping system described in the problem solution. Then, a set of pointers are created on the CPU and space is allocated on the GPU using the cudaMalloc method. The kernel is then called with the pointers as parameters and a block size of resolution+1, and thread size of resolution +1. (note that in the GPU version of the code RES=2*resolution). On the kernel the x, and y indices that each thread will be tasked with handling corresponds to the blockId and threadId the thread is assigned. The z indices are accounted for through the use of a for loop that each thread handles. Then the same calculations, once conducted on the CPU, are done so on the GPU, and once again taking advantage of the axi-symmetric nature of our problem the final values are translated across the x and y axis. Once all the values are computed in parallel by the GPU, the values, now stored inside the pointers on the GPU's memory, are copied into the arrays previously defined through the use of the cudaMemcpy method. Then using a similar method as defined on the CPU the values are exported to a CSV.

The GPU optimization resulted in massive speedups in all scenarios it was tested under. Even though the GPU would have been theoretically doing more calculations than the CPU, as far as the Euler integration loop was concerned, the GPU did in hundreds of milliseconds what took the CPU tens of thousands, resulting in a peak 13156.05 percent speed up. Figure 7 below shows the time it took in milliseconds for the CPU to conduct calculations vs. the GPU at varying resolutions utilizing a logarithmic scale (lower on y axis is better). GPU optimization of the RK4 loop resulted in similarly impressive gains, with the GPU doing the same calculations that took the CPU hundreds of thousands of milliseconds in just thousands of milliseconds. At its peak the GPU cut down execution time from the CPU by 4591.552 percent. Figure 8 below shows the time it took in milliseconds for the CPU to conduct the calculations for the RK4 loop vs. the GPU at varying resolutions utilizing a logarithmic scale (lower on y axis is better).

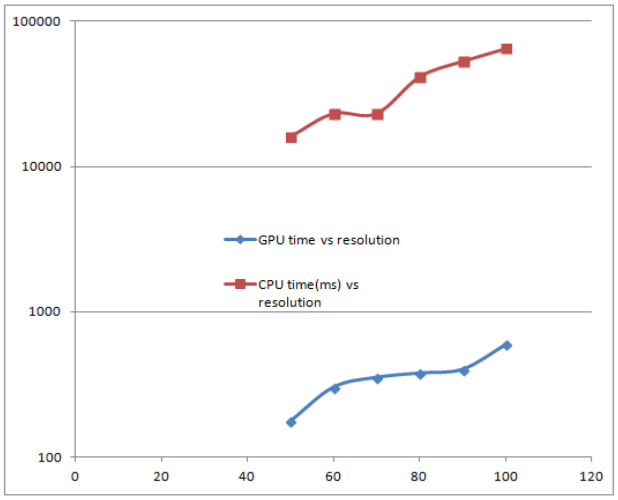


Figure 7. CPU vs. CUDA Euler's Integration

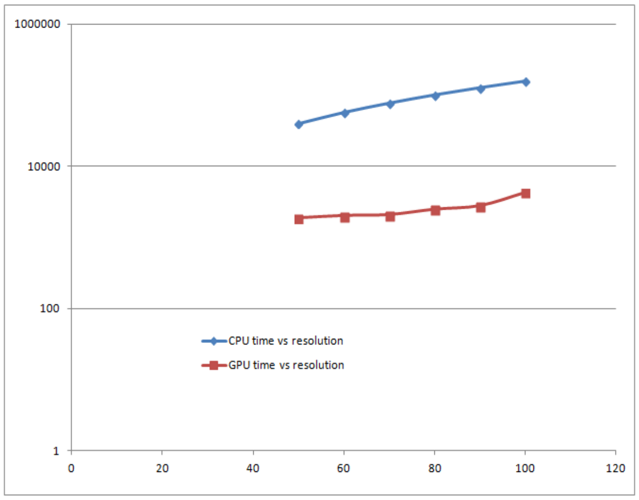


Figure 8. CPU vs. CUDA for RK4 computations.

Following the same trend of the RK4 loop and the Euler integration loop, the GPU implementation of Kuo's model achieved significant performance improvements over its CPU counterparts, doing once again in thousands of milliseconds what took the CPU hundreds of thousands. At its peak the GPU achieved a percent decrease of 4056.718 percent from the CPU. Figure 9 below shows the time it took in milliseconds for the CPU to conduct the calculations for Kuo's model vs. the GPU at varying resolutions utilizing a logarithmic scale (lower on y axis is better).

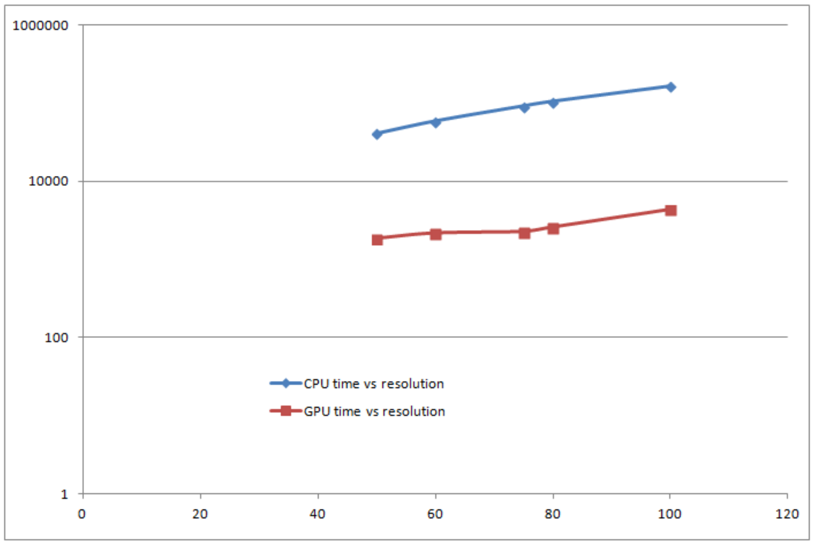


Figure 9. CPU vs. CUDA for the entire Kuo's model

With such impressive gains it's understandable for one to question if the GPU is returning the same results as the CPU. Below in Figure 10 is a set of graphs taken after running Kuo's model both on the GPU and on the CPU and comparing their results. As you can see the v, u, w vs r graphs, and values, are the same despite the fact that the GPU completed much faster than the CPU.

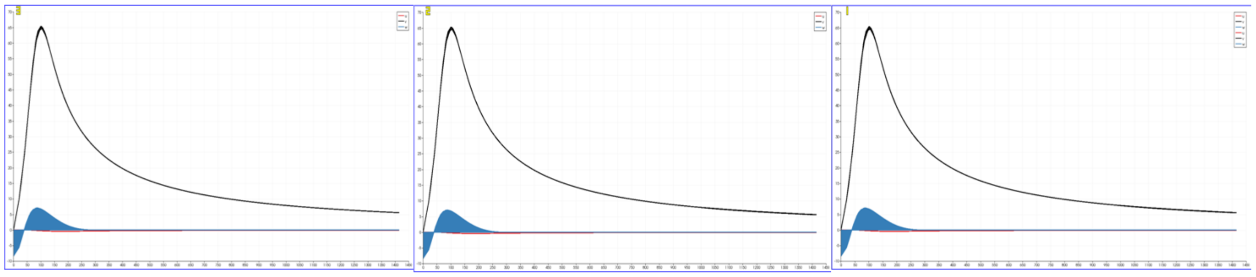


Figure 10, to the left is the graph of the CPU's results, in the middles the GPUS, and to the left the two graphs placed over each other. X-axis is radius and y axis is velocity in m/s

The CUDA code was all written in C, and a full listing of the code can be found under Appendix A in the CUDA section.

CPU Multithreading

Although less optimization tasks were performed with the CPU than with the GPU, we still conducted work that decreased the time the particle tracer program took to run. Rather, than just tracing one particle at a time by calling the appropriate, waiting for it to finish, then repeating the process. We took advantage of Java's built in multithreading functionality. We created a new Thread object for each of the particles we wanted to run, and then ran them simultaneously in order to improve efficiency. We found that without employing multithreading it took 55 milliseconds for the particle tracer to run one particle, but while running 3 particles simultaneously we allowed each one to complete in 35, 40 and 44 ms respectively. Thus we were able to reduce the time each particle took to run separately and quite effectively reduced the total time running 3 particles would have taken if they were not multithreaded.

Data Output Optimization

To store the results of our program, we chose to use a MySQL database. The program exports results of every run to a server that stores the results for future use. In the database, runs are stored as individual tables that are given a name with a universally unique ID assigned by the program that writes them. Currently manual access to the database is possible, while a proof-of-concept python script exists for loading directly from the database to Paraview (we plan to complete this script so it will be fully functional shortly).

Exporting program results to a database instead of a text based file has multiple benefits. First and most importantly is that results in a database are much easier to track and store over longer periods of time. Due to the increased accessibly of the results, they are easier to compare and analyze. In addition, MySQL uses relatively little disk space. It uses a more space-efficient binary data format, and because of the differences in how a database can be loaded into Paraview it should be possible to entirely remove one of the columns that needs to be stored. The reason for this is that the velocity on the Z Axis must be stored redundantly in order to load a file properly directly into Paraview, but a database would be loaded by a script that could simply correct the table as the values are read in. Next, the nature of how queries function in the sequel language that MySQL uses means that only portions of the data that need to be used can be read, limiting the time that it takes to access specific pieces of information. This increases efficiency greatly for many types of data-access because it is often not necessary to acquire all the data that is stored. Finally, MySQL is designed so that it can be used on a remote server. Our current server is connected to the network, but is only accessible on the local area network. Still, the potential exists to find a dedicated server and use it to store our runs, which has the benefit of allowing global access to our data and allowing distant computers effectively to run the program in parallel. The currently functioning Java code can be found under Appendix A as the last entry in the Java section.

Results

This section provides a summary of important predictions of the model including a comparison and validation with wind velocities measured in real world tornadoes, in real time. Results presented include velocity, stream tracers, glyphs, and debris transport predictions visualized using the Paraview software.

Model Validation

The 1998-Spencer and 1957-Dallas tornadoes were two Fujita Class 4 tornadoes for scientific measurements are widely published (de facto these two tornados are used as yard-sticks for comparison). We simulated these two tornados, using our models and employed them as a form of validation. Table 1 provides important parameters measured during these tornadoes that were used in our simulation, and references to the data. Also note that these values closely align with the range of values provided by Church et al [2]. Table 1 β and $\boldsymbol{\Gamma}$ were estimated in previous studies using meteorological data[11, 12] comparison of the model results with the measured data are presented below in Figures 11 and 12.

Table 1. Meteorological conditions of the tornados used for validation purpose.

Tornado	$\boldsymbol{\Gamma}$ (m^2/s)	β (#/s)	$\boldsymbol{\nu}$ (m^2/s)	References
Spencer, SD	8750 and 7990	0.01	5	Wurman et al [11]
Cleveland, TX	7500 and 6700	0.01 - 0.02	5	Hoechem et al[12]

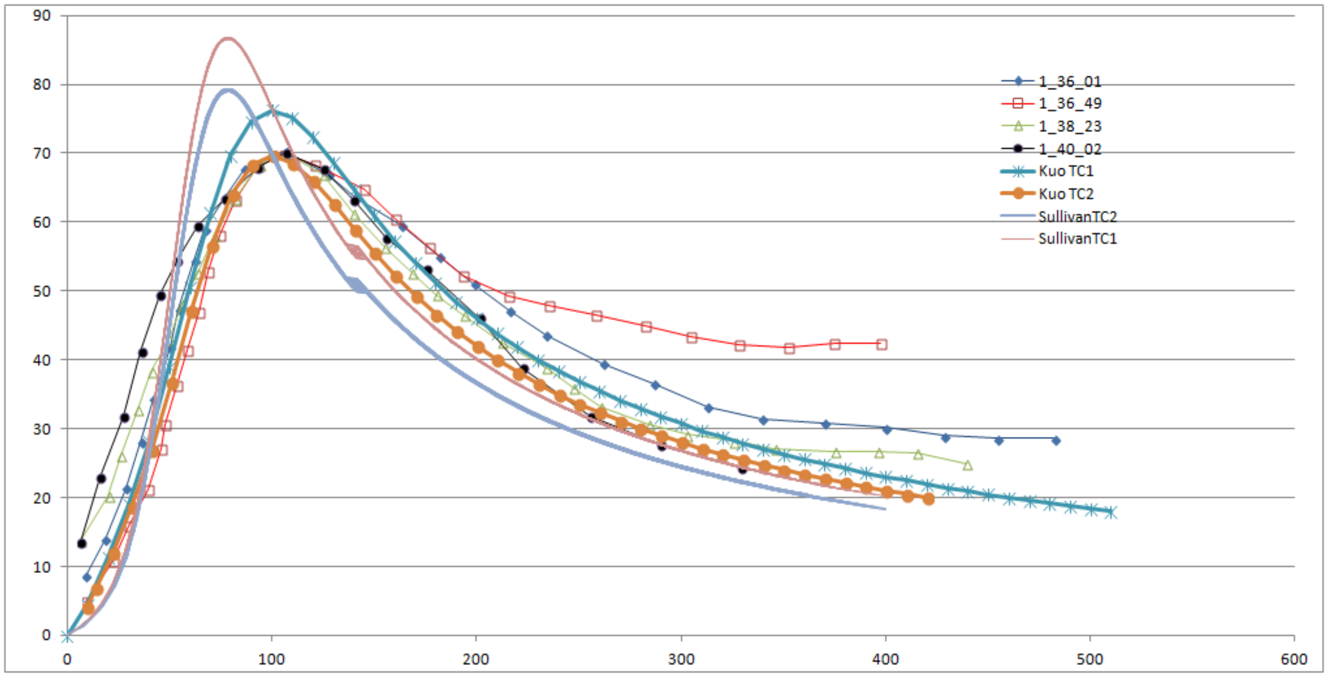


Figure 11 (Spencer, SD tornado) : In test case 1 (TC1) Γ was 8750, in test case 2 (TC2) Γ was at 7990. Also, velocities are 2-m averages. Depending on the reference peak values of measured values changed slightly as ground velocities are subtracted.

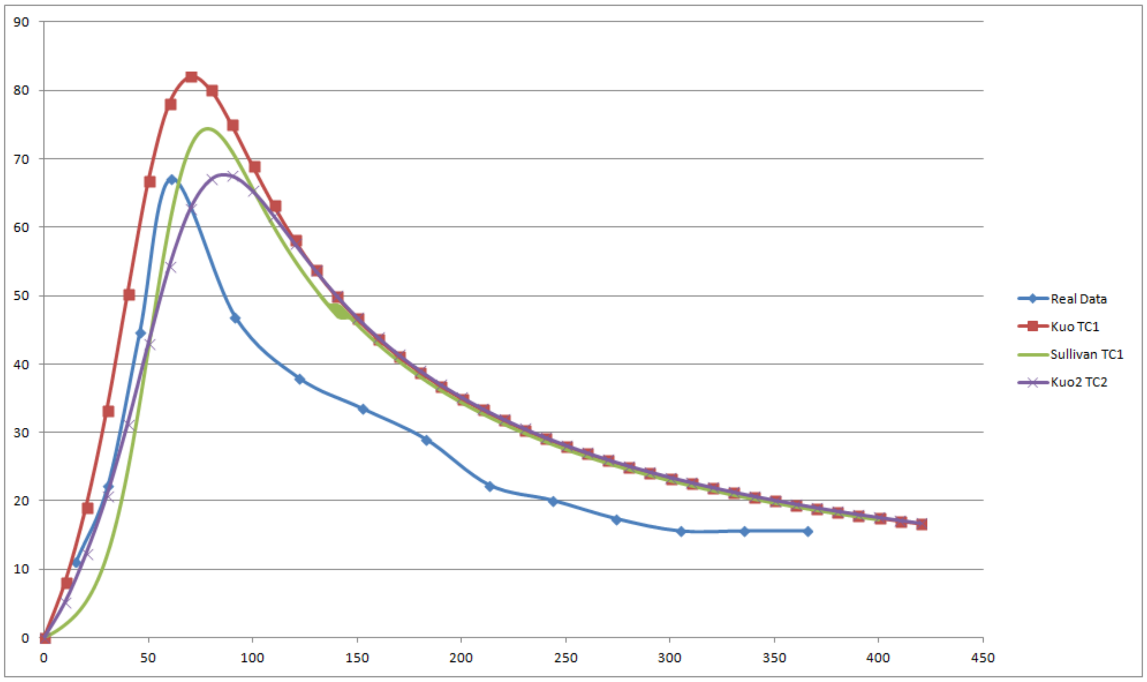


Figure 13 (Cleveland TX): In test case 1(TC1) Γ was 7500 and β was 0.01, in test case 2 (TC2) was 6700 and β was 0.0135.

Our models provided very good agreement with the real data. It is worth noting that this agreement is as good as the comparison presented in Wurman et al [11] using the Wood-White models, and those presented in Sarkar et al [5] which utilized a full scale CFD. Finally, agreement between Sullivan's equations and the measurements are not as good as our adaptation of Kuo's model. It should however be noted that there is considerable uncertainty in the measurements, especially those for the Cleveland TX tornado. 3D Doppler on wheels technology used in measurements of the Spencer tornado have reduced the error considerably, but still uncertainties as large as 25 percent are to be expected.

Predictions

1D Velocity Fields

After validation, Kuo's model was run for three cases as described in Table 2. The first case is a one cell tornado, the last two are two cells.

Table 2. Meteorological conditions of the tornados used for predictions.

	$\boldsymbol{\Gamma}$ (m^2/s)	β (#/s)	$\boldsymbol{\nu}$ (m^2/s)
Case 1	5000	0.01	5
Case 2	10000	0.02	5
Case 3	10000	0.01	5

Results are shown in Figures 13-15. No numerical instabilities were observed in our simulations. As shown here maximum wind velocities can reach as high as 123 m/s or 275 MPH. The radius of such an eye could be as large as 100's of meters. In addition sustained high winds exist at distances reaching a kilometer and beyond. These figures also show complicated flow fields that exist in two-cell tornadoes. At the very center of the tornado, cold air is "piped in" from above at high velocities. Adjacent to these velocities are fast upward moving velocities that carry warm air to high elevations. Radial velocities vary correspondingly outwards in the narrow central region and inwards in the outward region. This phenomenon is much less complex than in the case of a single cell tornado where it acts like a single pipe only moving air upwards.

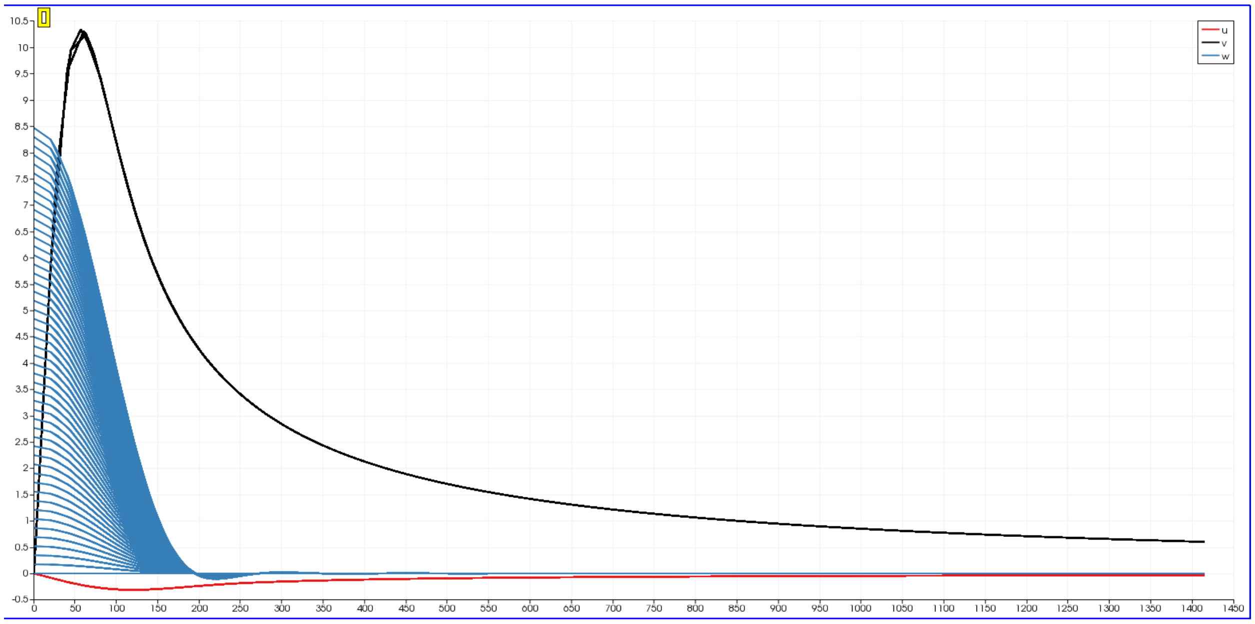


Figure 13 u, v and w vs r graphs for Case 1

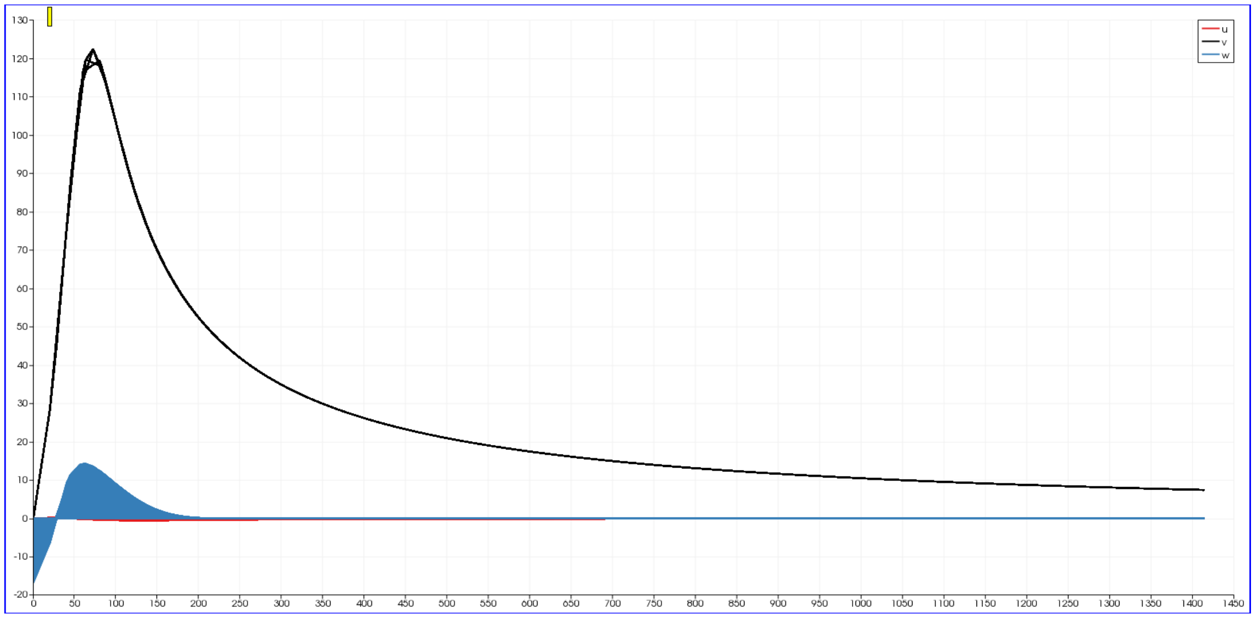


Figure 14: u, v, and w vs r graphs for Case 2

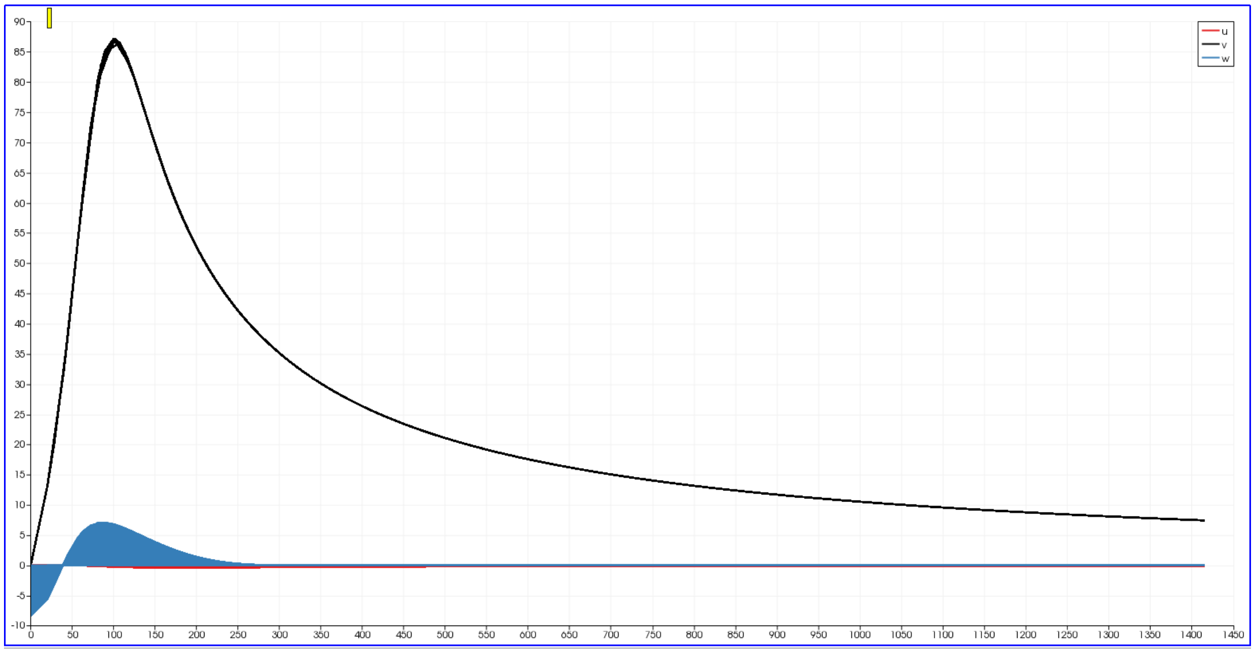


Figure 15: u, v and w vs r graphs for Case 3

3D Visualization

As previously discussed the Paraview software was used to visualize model predictions. Figures 16-19 illustrates these fields. Figure 16 displays vectors glyphs that clearly establish inward and radial movement of the flow. The stream tracers illustrate 3D movement of an imaginary air packet dragged upwards by the tornado. As expected the stream tracers exhibit axi-symmetry which was also observed in the Spencer tornado. Note the central core region where the flow is downward, in the two cell tornado and how the pressure of the air is consistently higher towards the core of the tornado. Also, note that the radial velocities are strongest towards the core of the tornado and weaker on the outskirts.

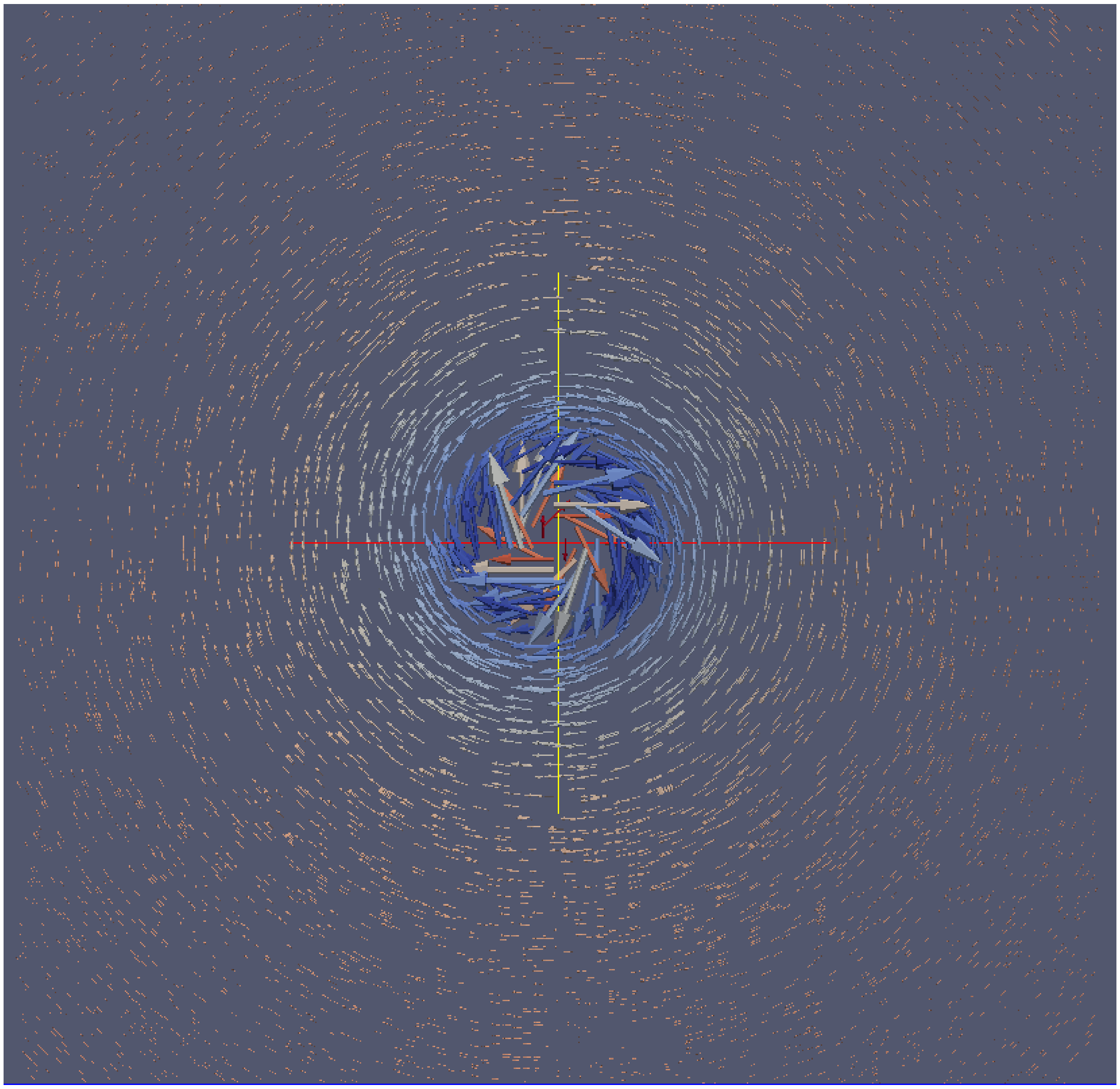


Figure 16. Glyph map of case 3, colored by u velocities

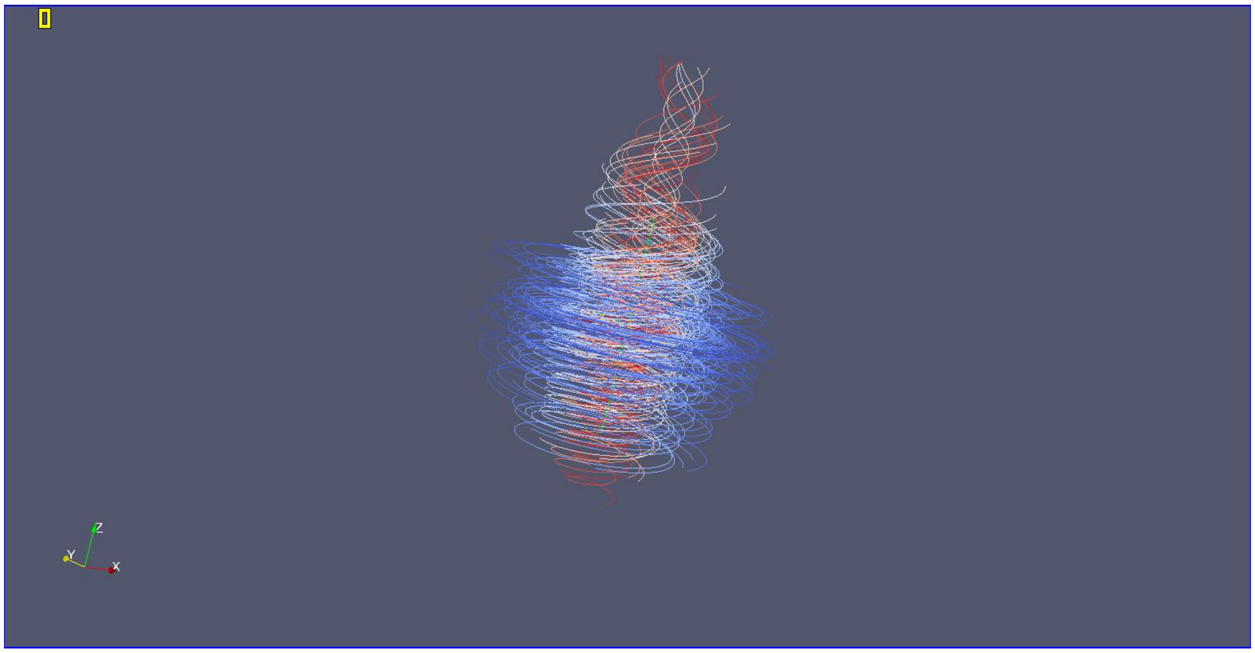


Figure 17 stream tracer of case 1 colored by pressure

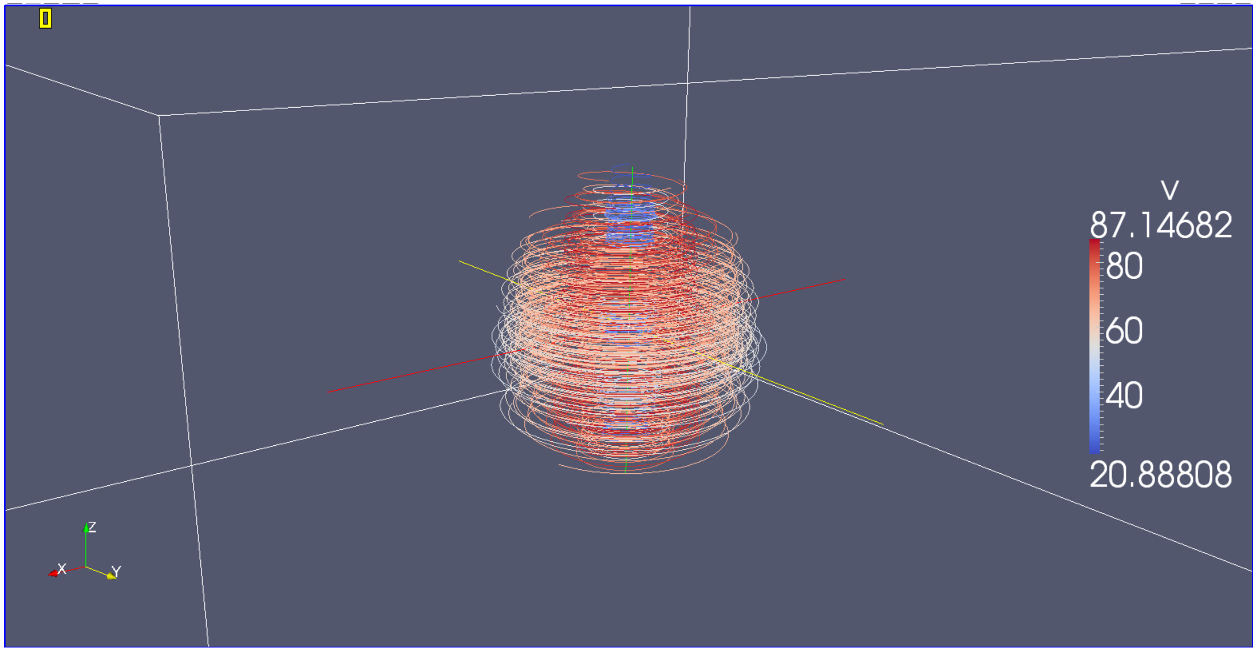


Figure 18 stream tracer of case 2 colored by v velocities

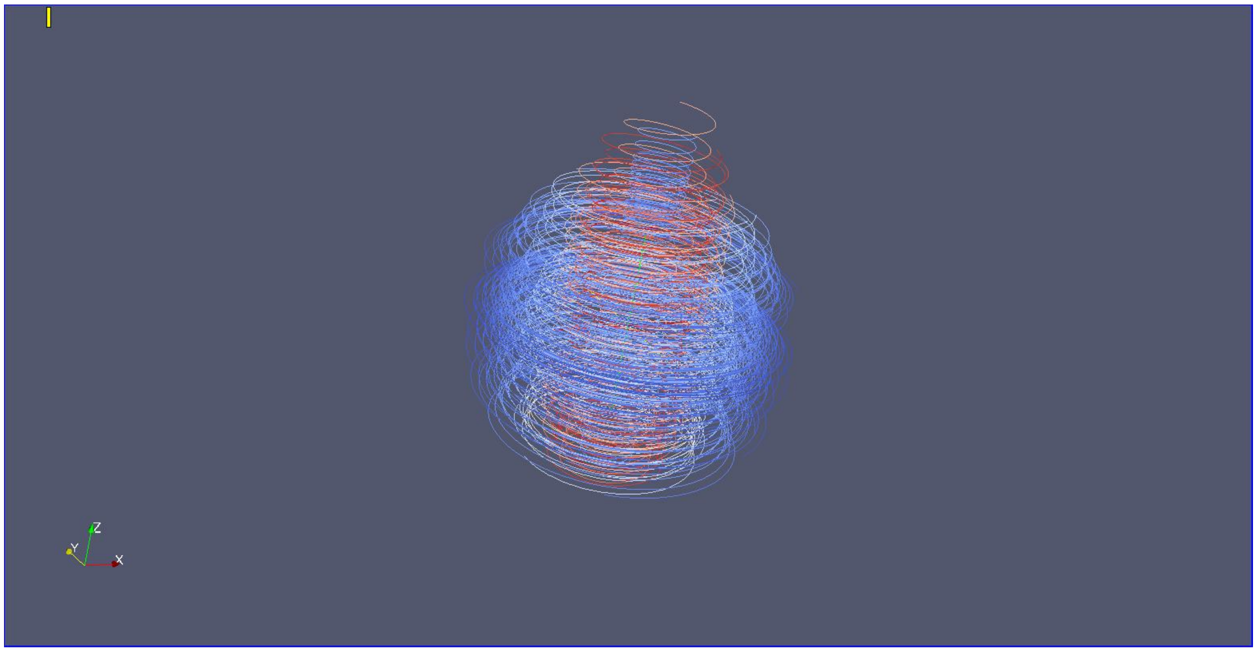


Figure 19 stream tracer of case 3 colored by pressure

Debris transport maps

As previously discussed, these winds are capable of lifting off and carrying large debris. We have used five arbitrarily chosen "particles" to simulate this phenomenon, and modeled the path they would follow using our particle tracer program. Table 3 provides drag characteristics, as well as mass of the particle.

	Mass of particle (kg)	Drag Area ($C_d \cdot A_f$)
Particle 1	1.0	0.1
Particle 2	1.0	0.25
Particle 3	10	0.5
Particle 4	10	1
Particle 5	100	3.2

Figures 20-22 below present visualized model predictions as to the path all five particles will follow within each case. Each particle was started off at a high altitude, near the center of the tornado.

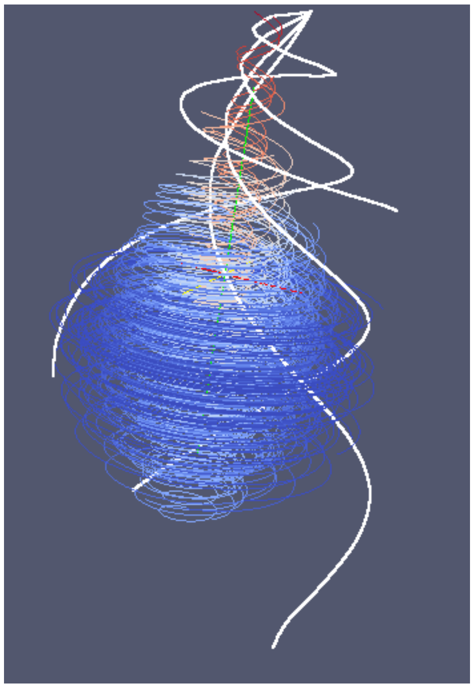


Figure 20 Paths for case 1 overlaid atop a stream tracer colored by w

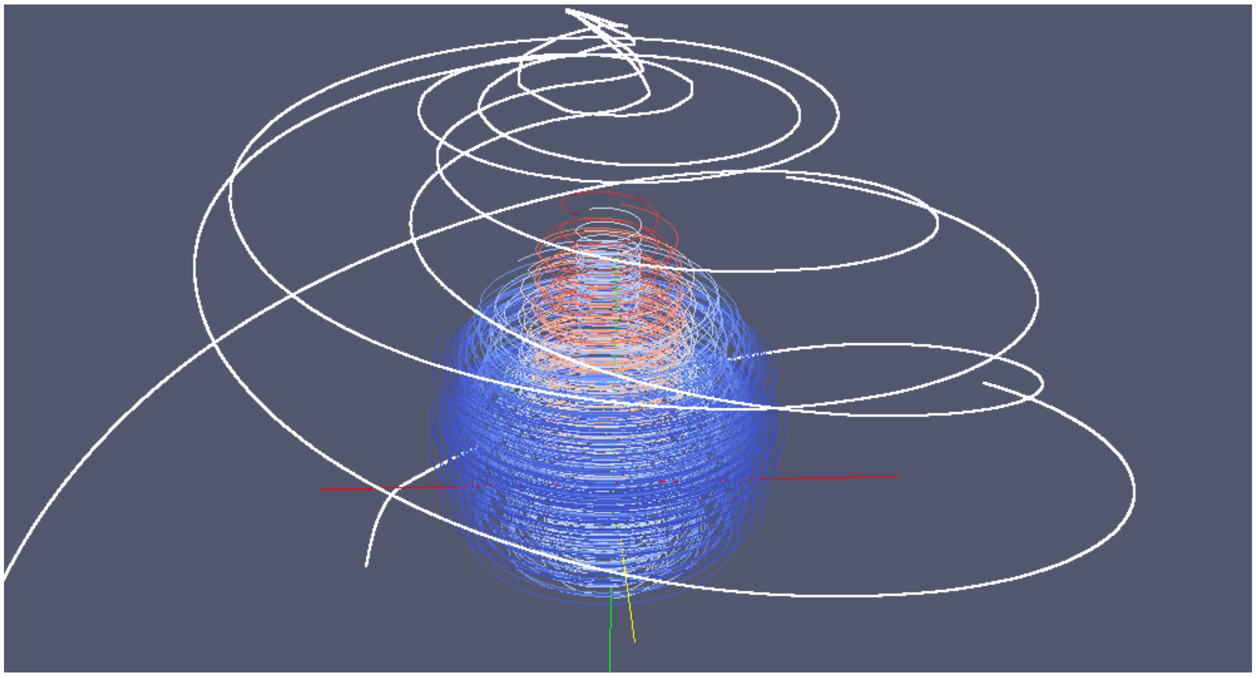


Figure 21 Paths for case 2 overlaid atop a stream tracer colored by v

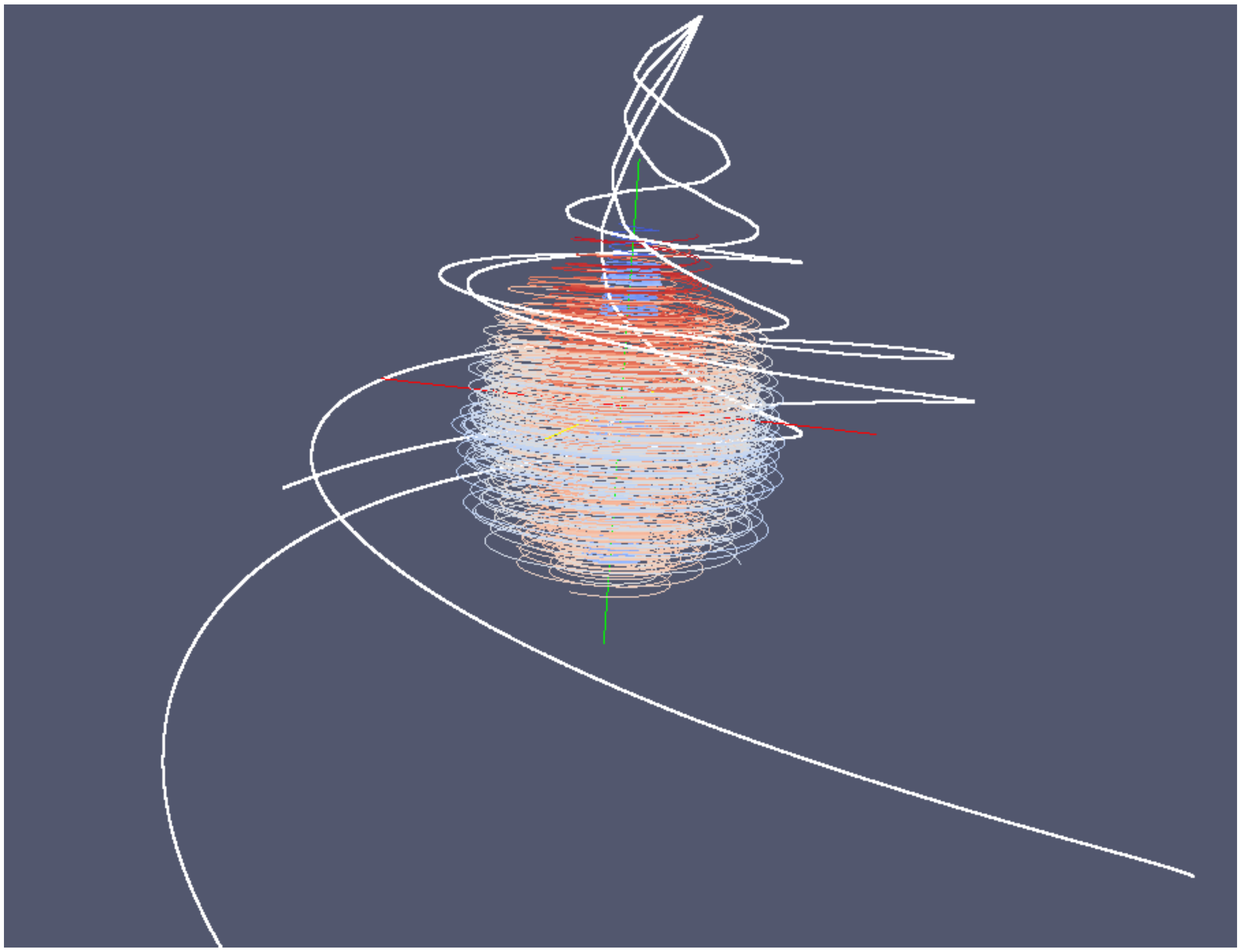


Figure 22: Paths for case 3 overlaid atop a stream tracer colored by w

Conclusions:

It is our conclusion that the complex dynamics of a tornado can be simulated with sufficient accuracy using the mathematical framework developed by Dr. Kuo of University of Chicago. Scientists have known that models such as Kuo’s are superior to simple algebraic models that are in use right now but they have settled for real-time computation enabled by simple models over accuracy. Other’s have utilized commercial CFD codes which generate similar results, but take hours or even days to run. By implementing the ODEs on GPU and multi-threading on CPU, we have demonstrated that scientifically superior models can also be computed in real time, and that their data can also be visualized in 3-D in real-time. As GPUs continue to improve, it is our belief that algorithms such as these could be used by the meteorologists to reconstruct radar images in real-time.

Two areas of further research in this general area should be explored: first extending the Kuo’s ODEs to simulate unsteady effects of a tornado and secondly, linking our model to radar signature model to simulate Doppler velocity signatures of evolving tornados as described by Davies –Jones and Wood[14].

Significant Original Accomplishments:

Throughout the extensive research we conducted as part of creating a working implementation of the various components of this project we found that some goals that we had set out to achieve had never been done before. While Kuos' model was originally developed in 1960-70, it is our belief that we are the first to have compared data generated by Kuo's models to data collected in the field using most recent Dopplar on Wheels technology; previous comparison of Kuo's models were with legacy data with large uncertainties. Thus, our results comparisons between Kuo's model and Sullivan's model and measured data are one of a kind. Based on this good comparison we believe meteorologists are more likely to use such phys-cs-based models rather than parametric models of the kind being in use right now. The particle tracer system we developed in itself may not be new, but its application to tornado conditions is relatively new. We have seen one other instance where particle transport was studied using Rankine and modified Rankine methods which are only capable of simulating tangential velocities, but not the vertical and axial velocities. Unlike that previous study, our model accounts for all three components of velocity and thus provides more accurate (at least theoretically) particle dynamics. We did not find a single reported scientific study where multi-threading of particle computations on the CPU were used to study their dynamics in real time – which is an original contribution as well. More importantly this feature could enable storm chasers to compute simulated radar tracks and also to invert radar tracks to reconstruct flow fields. Finally, the GPU code optimization we ran utilizing the CUDA framework was the first ever such experimentation conducted as far as Kuo's model is concerned. While we were able to find certain implementations of the RK4 loop on a GPU, none of the papers found discussed solving non-linear, coupled, ODEs or a boundary value problem. Further, none of the current RK4 optimization work done on the GPU has been done in such a way to speed up RK4 calculations over a coordinate grid (our ODEs had to be solved at each unique r value over the entire coordinate grid), instead focusing on creating a method that can solve an ODE one at a time. Certainly we do not claim that previous scientists could not solve such a problem, but only that none of these previous optimization works had a problem of our type in mind during the writing of their papers.

While uniqueness of individual accomplishments could be debated, even disagreed, it is a fact that computational approaches demonstrated by us – such as, use of physics-based ODEs to compute flow and temperature fields and Lagrangian particle transport in near-real time, their acceleration using GPU-based computation and CPU-multi-threading – have the potential to transform the way computation is used in researching tornados and in developing more accurate advanced warning technologies. Its significance is further underlined by the fact that two recent Master's Theses in Mathematics are focused on application of ODE's to simulate tornadoes.

Discussion of our Team and Experience:

Our team is composed of three sophomore students from Los Alamos High School. All the members have extensive experience with the Supercomputing Challenge. This is Colin Redman's 6th year competing, Sudeep Dasari's 5th, and David Murphy's 3rd year. Prior experience was extraordinarily useful in planning and implementing this year's project. Normally, this team has done work utilizing agent based modeling software such as Madkit and Greenfoot to optimize the likes of shipping facilities. This is the first time we have ever conducted such a physics based project, and we found it to be an extraordinary learning experience, and most importantly fun. Throughout the course of the year we learned in depth the complex physics that compose a tornado, as well as how they relate to the world around them. For the first time we were able to take our understanding of the physics behind a situation and translate that into a code that could then be used to provide a realistic model of said situation. Such a unique experience proved to be extremely educational and stimulating. Furthermore, we were for the first time given a computational task that begged for optimization through use of; the GPU, CPU multithreading, and data optimization. This is the first time we were able to conduct such experimentation in a Supercomputing project and the opportunity proved to be useful.

Both Sudeep and Colin have worked together for the past four Supercomputing challenges and this is the third year David has been part of the team. Sudeep and Colin both contributed their programming talents, and while Sudeep handled most the mathematical and GPU optimization work, Colin was more involved in data optimization and writing of the final report. David's physical model, while not implemented in the final report, provided some interesting validation options and will be included in our expo presentation. He also used his artistic talent to make graphics and did a good job with data analysis.

Acknowledgements

First, our team would like to thank the New Mexico Supercomputing Challenge for making this project possible. Next, we would like to thank Venkat Dasari who was our mentor this year. He helped us with; research, understanding the physics behind a tornado, understanding the equations, and proofreading our final report. The project would not have been a success without his valuable insight. Our team would also like to thank our teacher Lee Goodwin who provided encouragement throughout the year. We would like to thank those who evaluated our project at the Northern New Mexico Community College, as well as all those who reviewed the various reports we submitted. Their suggestions as well as constructive criticism helped guide us through the last stages of the report. Finally, we would like to thank Mr.Robey for all the valuable insight he gave to us as to the current tornado modeling conventions early in the year.

Bibliography

1. D. Neuenshwander, The Physics of Tornadoes, SPS Observer (2011)

2. C. Church, J.T. Snow, G.L. Baker and E. Agee, "Characteristics of Tornado like Vortices as a Function of Swirl Ratio," J. Atm. Sciences, Vol. 36 (1979).

3. N. ward, The exploration of certain features of tornado dynamics, J. Atm. Sciences, 29: 1194-1204 (1972)

4. R.P. Davies-Jones, Tornado dynamics, thunderstorm morphology and dynamics, US Govt Printing Office (1986)

5. K.P. Sarkar, Likuai, F.L. Haan, W. Galkes, CFD Simulations of the flow field of a laboratory simulated tornado and comparison with field measurements, Wind and Structures, Vol. 11, No. 2 (2008)

6. H.S. Thakar, Mathematical Models of the Geophysical Vortices, Manchester university (1969)

7. J.T. Baker, High Viscosity Solutions of Navier-Stokes Equations Modeling a Tornado vortex, Master of Science in Mathematics, Texas Tech University

8. H.L. Kuo, On the dynamics of convective atmospheric vortices, J. Atm. Sciences, Vol. 23 (1965)

9 . R. Sullivan, A two-cell Vortex Solution of the Navier-Stokes Equations, J. Aero. Space Science, 26, 767 (1969)

10. V.T. Wood and L.W. White, A parametric Wind-Pressure relationship for Rankine and Non-Rankine Cyclostropic Vortices, American Meteorological Conference, (2011)

11. C. Alexandar and J. Wurman, The May 1998 Spencer, South Dakota Storm, Part 1: The evolution and environment of the tornadoes, Mon. Wea. Rev., 133, 72-76 (2005)

12. N.H. Hoechem, Wind Speed and airflow patterns in the Dallas Tornado of 1957, Monthly Weather Review, 88, 167-180, (1960)

13. Harlow and L.R. Stein, Structural Analysis of tornado like vortices, J. Atm. Sci, Vol. 31, 1974

14. Davies-Jones and V.T. Wood, Simulated Doppler Velocity Signatures of Evolving Tornado-like

Vortices, J. of Atm. And Oceanic Technology, Page 1029, (2006)

Appendix A

The following section contains the final versions of the code developed throughout the course of this project. While there was more written, those programs were alpha or beta versions of the following program and thus it would serve little purpose to include them. This section has been split into three sections, Java, Python, and CUDA. The Java code makes up the backbone of our project, and the final programs used for visualization of results in Paraview were written in Java. Some Python was used throughout the project as a means to develop alpha versions of all models and easily check the validity of generated results through use of matplotlib. The CUDA code was written in C, and contains only the code written for optimization with the GPU.

Java

SullivanTest.java

```java
import java.io.FileWriter;
import java.io.IOException;

import javax.swing.JFileChooser;
//java implementation of Sullivan's equations and Lagrangian Particle Tracer
public class SullivanTest {

    private static final float eV = 5;// eddy viscosity
    private static JFileChooser chooser = new JFileChooser();;
    private static final float R = 7500;// circulation strength of vortex
    private static final float a = (float) 0.01;// strength of suction
    private static final float aP = 10000;// ambient pressure
    private static final float p = (float) 1;// density of air
    // limit of H function
    private static float limitH = calcXInteg(20, 500, 500);
    // value of H function as it approaches infinity
    private static float infiniteH = calcXInteg(calcX(42), 500, 500);
    // Rmax divided by resolution
    private static float deltaR;
    // matrix translation variables
    private static float Xmin, Xmax, Ymin, Ymax, Zmax;
    // maximum range function will be evaluated to
    private static float RMax;
    // variable dictating how many gridpoints Rmax will be split into
    private static final int resolution = 50;
    // 3d array containing v,u,w,pressure,H,Vx,Vy values at each gridpoint
    private static float[][][] V;
    private static float[][][] U;
    private static float[][][] W;
    private static float[][][] P;
    private static float[][][] Vx;
    private static float[][][] Vy;
    private static float[][] H;
    // array in which the particle tracer will store particles' X,Y,Z,i,j,k
    // values at each time step
    private static float[] traceVx;
    private static float[] traceVy;
    private static float[] traceVz;
    private static float[] traceX;
    private static float[] traceY;
    private static float[] traceZ;
    // maximum time the particle tracer program will iterate to
    private static float T = 70;
    // time step used by the particle tracer program
    private static float dT = (float) 0.01;
    // the mass of the particle
    private static float Mp = 1;
    // the drag coefficients for the particle in the x,y,z direction
    // respectively
    private static float Cdz = 1;
    private static float Cdx = 1;
    private static float Cdy = 1;

    public static void main(String[] args) {
        // calculates the deltaR for program
        deltaR = (RMax) / resolution;
        System.out.println(deltaR + " " + RMax);
        // sets up the Xmin,Xmax,Ymin,Ymax,Zmax variables for matrix
        // calculations
        Xmin = -resolution;
        Xmax = resolution;
        Ymin = -resolution;
        Ymax = resolution;
        Zmax = resolution;
        // initializes the size for each of the above described arrays
        // When resolution=50 there will be -50 to 50 grid points in x and y
        // axis and 0-50 in z
        // The matrix and set up so that each grid point has a slot in the
array
        System.out.println(infiniteH);
        V = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int) Zmax];
        U = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int) Zmax];
        W = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int) Zmax];
        P = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int) Zmax];
        H = new float[(int) (Xmax * 2)][(int) (Ymax * 2)];
        Vx = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int) Zmax];
        Vy = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int) Zmax];
                //evaluates the integ and stores in H arrays
                genHInteg();
                //for loops calculate the V U W unit vector and pressure
values for each grid point
                //Since [0][0][0] corresponds to (-resolution,-resolution,0)
Xmin is added to the X coordinate and
                //Ymin to the Y coordinate to achieve the neccesary matrix
corrections
                for (int zCor = 0; zCor < Zmax; zCor++) {
                    for (int yCor = 0; yCor < (2 * Ymax); yCor++) {
                        for (int xCor = 0; xCor < (2 * Xmax); xCor++) {
                            V[xCor][yCor][zCor] = calcV(xCor + Xmin, yCor +
Ymin);
                            U[xCor][yCor][zCor] = calcU(xCor + Xmin, yCor +
Ymin, zCor);
```

```
                            W[xCor][yCor][zCor] = calcW(xCor + Xmin, yCor +
Ymin, zCor);
                            //pressure has an additional resolution variable
for use in Euler Integration
                            P[xCor][yCor][zCor] = calcP(xCor + Xmin, yCor +
Ymin, zCor);
                        }
                    }
                }
                try {
                    System.out.println("Starting data export");
                    //calls function that exports data into .csv format
                    exportData();
                } catch (IOException e) {
                    e.printStackTrace();
                }
                //initilializes the arrays used in particle tracer
                traceX = new float[(int) (T / dT)+1];
                traceY = new float[(int) (T / dT)+1];
                traceZ = new float[(int) (T / dT)+1];
                traceVx = new float[(int) (T / dT)+1];
                traceVy = new float[(int) (T / dT)+1];
                traceVz = new float[(int) (T / dT)+1];
                //starts timer for tracer
                long sT=System.currentTimeMillis();
                //begins particle trace using starting position of
(400,400,1000) and starting velocity of <0,0,0>
                partTrace(400,400,1000,0,0,0);
                //calculates and prints how long the "particle trace" method
took
                System.out.println("Parts Took:
"+(System.currentTimeMillis()-sT));
                try {
                    //exports the particle data in .csv format
                    System.out.println("Starting particle export");
                    exportPartData();
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                System.out.println("Done");
    }

    public static void exportPartData() throws IOException {
        //exports data for particles in .csv
        chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
        chooser.showSaveDialog(null);
        String filename = chooser.getSelectedFile().getPath();
        FileWriter writer = new FileWriter(filename + "\\PartData.csv");
        writer.append("Time");
        writer.append(",");
        writer.append("X0");
        writer.append(",");
        writer.append("Y0");
        writer.append(",");
        writer.append("Z0");
        writer.append(",");
        writer.append("Vpx");
        writer.append(",");
        writer.append("Vpy");
        writer.append(",");
        writer.append("Vpz");
        writer.append(",");
        writer.append("R");
        writer.append("\n");
        for (float t = 0; t < (100*T); t++) {
            Float calc=t/100;
            writer.append(""+calc);
            writer.append(",");
            calc=traceX[(int) t];
            writer.append(""+calc);
            writer.append(",");
            calc=traceY[(int) t];
            writer.append(""+calc);
            writer.append(",");
            calc=traceZ[(int) t];
            writer.append(""+calc);
            writer.append(",");
            calc=traceVx[(int) t];
            writer.append(""+calc);
            writer.append(",");
            calc=traceVy[(int) t];
            writer.append(""+calc);
            writer.append(",");
            calc=traceVz[(int) t];
            writer.append(""+calc);
            writer.append(",");
            calc=(float) Math.sqrt(traceX[(int)
t]*traceX[(int)t]+traceY[(int) t]*traceY[(int)t]);
            writer.append(""+calc);
            writer.append("\n");
            writer.flush();
        }

        writer.close();
    }
    public static void partTrace(float x, float y, float z, float vx, float
vy ,float vz) {
        //sets initial values for the particle
        float x0 = x, y0 = y, z0 = z, Vpx = vx, Vpy = vy, Vpz = vz;
        //initiliazes current time value
        double t = 0;
        //cT, when multiplied by t, calculates the correct array index for
the t value
        int cT=(int) (1/dT);
        //fills 0 slot in arrays with starting conditions
        traceX[(int) (t * cT)] = x0;
        traceY[(int) (t * cT)] = y0;
        traceZ[(int) (t * cT)] = z0;
        traceVx[(int) (t * cT)] = Vpx;
        traceVy[(int) (t * cT)] = Vpy;
        traceVz[(int) (t * cT)] = Vpz;
        //fills starting conditions into an array to be used in RK4
        double stats[]={x0,y0,z0,Vpx,Vpy,Vpz};
        //iterates through T values from 0 to T
        do {
            t += dT;
            //uses an fourth order Runge Kutta method to calculate the
x,y,z,vx,vy,vz at the next time step
            stats=rk41(T,dT,stats);
            //stores new x,y,z,Vx,Vy,Vz values into corresponding arrays at
correct index at restarts process till t=T
            traceX[(int) (t * cT)]=(float) stats[0];
            traceY[(int) (t * cT)]=(float) stats[1];
            traceZ[(int) (t * cT)]=(float) stats[2];
            traceVx[(int) (t * cT)]=(float) stats[3];
            traceVy[(int) (t * cT)]=(float) stats[4];
            traceVz[(int) (t * cT)]=(float) stats[5];
        } while (t <= T);
    }

    public static double[] rk41(double x, double h, double[] y) {
        //implements Fourth Order Runge Kutta with derivpd as f(),returns
array with all
        //timestep values x,y,z,Vx,Vy,Vz moved up a step
        double[] f = derivpd(x, y);
        double[] k = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4], h * f[5] };
        double[] a = new double[] { y[0] + 0.5 * k[0], y[1] + 0.5 * k[1],
                y[2] + 0.5 * k[2], y[3] + 0.5 * k[3], y[4] + 0.5 * k[4],
                y[5] + 0.5 * k[5] };
        f = derivpd(x + 0.5 * h, a);
        double[] k2 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4], h * f[5] };
        a = new double[] { y[0] + 0.5 * k2[0], y[1] + 0.5 * k2[1],
                y[2] + 0.5 * k2[2], y[3] + 0.5 * k[3], y[4] + 0.5 * k[4],
                y[5] + 0.5 * k[5] };
        f = derivpd(x + 0.5 * h, a);
        double[] k3 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4], h * f[5] };
        a = new double[] { y[0] + k3[0], y[1] + k3[1], y[2] + k3[2],
                y[3] + k3[3], y[4] + k3[4], y[5] + k3[5] };
        f = derivpd(x + h, a);
        double[] k4 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4], h * f[5] };
        a = new double[] { y[0] + (k[0] + 2 * (k2[0] + k3[0]) + k4[0]) / 6,
                y[1] + (k[1] + 2 * (k2[1] + k3[1]) + k4[1]) / 6,
                y[2] + (k[2] + 2 * (k2[2] + k3[2]) + k4[2]) / 6,
                y[3] + (k[3] + 2 * (k2[3] + k3[3]) + k4[3]) / 6,
                y[4] + (k[4] + 2 * (k2[4] + k3[4]) + k4[4]) / 6,
                y[5] + (k[5] + 2 * (k2[5] + k3[5]) + k4[5]) / 6 };
        return a;
    }
    public static double[] derivpd(double t,double[]y){
        //gets the i,j,k unit vector values calculated by Kuos model at the
particles current postion
        int ix=(int) ((y[0]/resolution)+Xmax);
        int iy=(int) ((y[1]/resolution)+Ymax);
        int iz=(int) ((y[2]/resolution));
        float Vfx=0,Vfy=0,Vfz=0;
        Vfx=Vx[ix][iy][iz];
        Vfy=Vy[ix][iy][iz];
        Vfz=W[ix][iy][iz];
        //calculates the particles next Vx,Vy,Vz
        double Fx=(Cdx*Math.abs(Vfx-y[3])*(Vfx-y[3]))/Mp;
        double Fy=(Cdy*Math.abs(Vfy-y[4])*(Vfy-y[4]))/Mp;
        double Fz=(Cdz*Math.abs(Vfz-y[5])*(Vfz-y[5]))/Mp-9.81;
        //returns new x,y,z,Vx,Vy,Vz values for next time step
        double a[]= {y[3],y[4],y[5],Fx,Fy,Fz};
        return a;
    }

    public static void exportData() throws IOException {
        //exports data in .csv
        chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
        chooser.showSaveDialog(null);
        String filename = chooser.getSelectedFile().getPath();
        float v, u, w, angle, pr;
        FileWriter writer = new FileWriter(filename + "\\VectData.csv");
        int counter = 0;
        writer.append("XCor");
        writer.append(",");
        writer.append("YCor");
        writer.append(",");
        writer.append("ZCor");
        writer.append(",");
        writer.append("r");
        writer.append(",");
        writer.append("Vx");
        writer.append(",");
        writer.append("Vy");
        writer.append(",");
        writer.append("Vz");
        writer.append(",");
        writer.append("v");
        writer.append(",");
        writer.append("u");
        writer.append(",");
        writer.append("w");
        writer.append(",");
        writer.append("Pressure");
        writer.append("\n");
        for (int zCor = 0; zCor < Zmax; zCor++) {
            for (int yCor = 0; yCor < (2 * Ymax); yCor++) {
                for (int xCor = 0; xCor < (2 * Xmax); xCor++) {
                    v = V[xCor][yCor][zCor];
                    w = W[xCor][yCor][zCor];
                    u = U[xCor][yCor][zCor];
                    pr = P[xCor][yCor][zCor];
                    if ((xCor + Xmin) * deltaR == 0
                            && (yCor + Ymin) * deltaR == 0)
                        angle = 0;
                    else {
                        angle = (float) Math.atan2((yCor + Ymin) * deltaR,
                                (xCor + Xmin) * deltaR);
                        if (angle < 0)
                            angle = (float) ((2 * Math.PI) + angle);
                    }
                    writer.write("" + ((xCor + Xmin) * deltaR));
                    writer.write(",");
                    writer.write("" + ((yCor + Ymin) * deltaR));
                    writer.write(",");
                    writer.write("" + (zCor * deltaR));
                    writer.write(",");
                    float x = (xCor + Xmin) * deltaR;
                    float y = (yCor + Ymin) * deltaR;
                    Float calc = (float) Math.sqrt(x * x + y * y);
                    writer.write("" + calc);
                    writer.write(",");
                    calc = (float) ((u * Math.cos(angle)) - (v * Math
                            .sin(angle)));
                    if (calc.equals(Float.NaN))
                        System.out.println(u + " " + v + " " + xCor + " "
                                + yCor + " " + H[0][0]);

                    Vx[xCor][yCor][zCor] = calc;
                    writer.write("" + calc);
                    writer.write(",");
                    calc = (float) ((u * Math.sin(angle)) + (v * Math
                            .cos(angle)));
                    if (calc.equals(Float.NaN))
                        System.out.println(u + " " + v + " " + xCor + " "
                                + yCor + " " + H[0][0]);

                    Vy[xCor][yCor][zCor] = calc;
                    writer.write("" + calc);
                    writer.write(",");
                    writer.write("" + w);
                    writer.write(",");
                    writer.write("" + v);
                    writer.write(",");
                    writer.write("" + u);
                    writer.write(",");
                    writer.write("" + w);
                    writer.write(",");
                    writer.write("" + pr);
                    writer.write("\n");
                    counter++;
                    writer.flush();
                }
            }
        }

        writer.close();
        System.out.println(counter);
    }

    public static float calcRMax() {
        //returns RMax value for certain eddy viscosity and suction strength
        return (float) Math.sqrt((20 * 2 * eV) / a);
    }

    public static float calcV(float x1, float y1) {
        //returns V value at gridpoint
        float x = x1 * deltaR;
        float y = y1 * deltaR;
        float r = (float) Math.sqrt((x * x) + (y * y));
        r++;
        return (float) ((R / (2 * Math.PI * r)) * (getHInteg(x1 + Xmax, y1
                + Ymax) / infiniteH));
    }

    public static float calcU(float x1, float y1, float z) {
        //returns U value at gridpoint
        float x = x1 * deltaR;
        float y = y1 * deltaR;
        float r = (float) Math.sqrt((x * x) + (y * y));
        r++;
        return (float) ((-a * r) + (((6 * eV) / r) * (1 - Math.pow(Math.E,
                -calcX(r)))));
    }

    public static float calcW(float z1, float x1, float y1) {
        //returns W value at gridpoint
        float z = z1 * deltaR;
        float x = x1 * deltaR;
        float y = y1 * deltaR;
        float r = (float) Math.sqrt((x * x) + (y * y));
        float value = (float) (2 * a * z * (1 - (3 * Math
                .pow(Math.E, -calcX(r)))));
        r++;
        return value;
    }

    public static float calcP(float x1, float y1, float z) {
        //returns pressure value at gridpoint
        float x = x1 * deltaR;
        float y = y1 * deltaR;
        float r = (float) Math.sqrt((x * x) + (y * y));
        r++;
        float val1 = (float) (1 - Math.pow(Math.E, -calcX(r)));

        return aP + calcPresInteg(200, 2000)
                - (((a * a * p) / 2) * ((r * r) + (4 * z * z)))
                - (((18 * p * eV * eV) / (r * r)) * (val1 * val1));
    }

    public static float getHInteg(float xCor, float yCor) {
        //returns value of H function at specified gridpoint
        return H[(int) xCor][(int) yCor];
    }

    public static void genHInteg() {
        //calculates the various ODE values at each gridpoint in quadrant 1
        //takes advantage of axi-symettry to translate to other quandrants
        //stores into necessary arrays
        long sT = System.currentTimeMillis();
        for (int x = 0; x <= resolution; x++) {
            for (int y = 0; y <= resolution; y++) {
                float x1 = (x + Xmin) * deltaR;
                float y1 = (y + Ymin) * deltaR;
                float r = (float) Math.sqrt((x1 * x1) + (y1 * y1));
                float X = calcX(r);
                if (X > 20) {
                    H[x][y] = limitH;
                } else {
                    H[x][y] = calcXInteg(X, 200, 200);
                }
            }
        }
        for (int x = 0; x < resolution; x++) {
            for (int y = (resolution * 2) - 1; y >= resolution; y--) {
                H[x][y] = H[x][(resolution * 2) - y];
            }
        }
        for (int x = (resolution * 2) - 1; x >= resolution; x--) {
            for (int y = (resolution * 2) - 1; y >= resolution; y--) {

                H[x][y] = H[(resolution * 2) - x][(resolution * 2) - y];
            }
        }
        for (int x = (resolution * 2) - 1; x > resolution; x--) {
            for (int y = 0; y < resolution; y++) {

                H[x][y] = H[(resolution * 2) - x][y];
            }
        }
        System.out.println("H took: " + (System.currentTimeMillis() - sT));
    }

    public static float calcXInteg(float x, float resolution, float secRes) {
        //integrate equation at given x and uses resolution variables to
determine size of Euler integration blocks
        if (x == 0)
            return 0;
        float total = 0;
        float dR1 = x / resolution;
        for (int n = 2; n <= resolution; n++) {
            float r = (n) * dR1;
            float rn = ((n - 1)) * dR1;
            float integ = (float) (0.5 * ((Math.pow(Math.E,
                    FunctionT(r, secRes))) + (Math.pow(Math.E,
                    FunctionT(rn, secRes)))) * dR1);
            total += integ;
        }
        return total;
    }

    private static float FunctionT(float t, float resolution) {
        //integrate equation at given x and uses resolution variables to
determine size of Euler integration blocks
        float total = 0;
        float dR = t / resolution;
        for (int n = 2; n <= resolution; n++) {
            float r = (n) * dR;
            float rn = ((n - 1)) * dR;
            float integ = (float) (0.5 * (((1 - Math.pow(Math.E, -r)) / r) +
((1 - Math
                    .pow(Math.E, -rn)) / rn)) * dR);
            total += (3. * integ);
        }
        total = total - t;
        return total;
    }

    public static float calcPresInteg(float R, int resolution) {
        //integrate equation at given R and uses resolution variables to
determine size of Euler integration blocks
        //returns integral for use in calculating pressure
        float total = 0;
        float DR = R / resolution;
        for (int n = 2; n < resolution; n++) {
            float r = n * DR;
            float rn = (n - 1) * DR;
            float integ = (float) (0.5 * (((5 * 5) / r) + ((5 * 5) / rn)) *
DR);
            total += integ;

        }
        return (float) (1.5 * total);
    }

    public static float calcX(float r) {
        //returns X for given r
        return (a * r * r) / (2 * eV);
    }

}
```

KuoTest.java

```
import java.io.FileWriter;
import java.io.IOException;
import java.math.*;
import javax.swing.JFileChooser;
//java implentation of Kuo's equations and Lagrangian particle tracer
public class KouTest {
    // eddy viscosity
    private static final float eV = 5;
    private static JFileChooser chooser = new JFileChooser();
    // circulation strength of vortex
    private static final float R = 7500;
    // suction strength of vortex
    private static final float b = (float) 0.01;
    // ambient pressure
    private static final float aP = 10000;
    //density
    private static final float p = (float) 1;
    //Program will calculate x,y,z values between -RMax to RMax
    private static final float RMax=1000;
    //number of grid points between origin and maximum
    private static final int resolution = 50;
    //distance between each grid point ie. RMax/resolution
    private static float deltaR;
    //symbolizes -resolution,resolution,-resolution,resolution,resolution
respectively
    //acts as syntacticly friendly replacer for resolution during matrix
corrections
    private static float Xmin, Xmax, Ymin, Ymax, Zmax;
    //3d array containing all the V unit vector values at each grid point
    private static float[][][] V;
    //3d array containing all the U unit vector values at each grid point
    private static float[][][] U;
    //3d array containing all the W unit vector values at each grid point
    private static float[][][] W;
    //3d array containing all the i unit vector values at each grid point
    private static float[][][] Vx;
    //3d array containing all the j unit vector values at each grid point
    private static float[][][] Vy;
    //3d array containing all the pressure values at each grid point
    private static float[][][] P;
    //2d arrays containing all values of the F, F', and M ODEs at each grid
point
    private static float[][] F, dF, M;
    //array in which our shooting method will store initial values for RK4
method
    private static double[] shoot;
    //array in which the particle tracer will store particles' X,Y,Z,i,j,k
values at each time step
    private static float[] traceX;
    private static float[] traceY;
    private static float[] traceZ;
    private static float[] traceVx;
    private static float[] traceVy;
    private static float[] traceVz;
    //maximum time the particle tracer program will iterate to
    private static float T = 70;
    //time step used by the particle tracer program
    private static float dT = (float) 0.01;
    //the mass of the particle
    private static float Mp=1;
    //the drag coefficients for the particle in the x,y,z direction
respectively
    private static float Cdz=1;
    private static float Cdx=1;
    private static float Cdy=1;
    public static void main(String[] args) {
        //calculates the deltaR for program
        deltaR = (RMax) / resolution;
        System.out.println(deltaR + " " + RMax);
        //sets up the Xmin,Xmax,Ymin,Ymax,Zmax variables for matrix
calculations
        Xmin = -resolution;
        Xmax = resolution;
        Ymin = -resolution;
        Ymax = resolution;
        Zmax = resolution;
        //initialializes the size for each of the above described arrays
        //When resolution=50 there will be -50 to 50 grid points in x and y
axis and 0-50 in z
```

```
        //The matrix and set up so that each grid point has a slot in the
array
        V = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int) Zmax];
        U = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int) Zmax];
        W = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int) Zmax];
        Vx = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int) Zmax];
        Vy = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int) Zmax];
        P = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int) Zmax];
        F = new float[(int) (Xmax * 2)][(int) (Ymax * 2)];
        dF = new float[(int) (Xmax * 2)][(int) (Ymax * 2)];
        M = new float[(int) (Xmax * 2)][(int) (Ymax * 2)];
        System.out.println("generating velocity ODEs...");
        //uses shooting method to initialize initial value which will be
stored in shoot
        genShoot();
        //evaluates the ODE and stores in F,dF,M arrays
        calcODE();
        //for loops calculate the V U W unit vector and pressure values for
each grid point
        //Since [0][0][0] corresponds to (-resolution,-resolution,0) Xmin is
added to the X coordinate and
        //Ymin to the Y coordinate to achieve the neccesary matrix
corrections
        for (int zCor = 0; zCor < Zmax; zCor++) {
            for (int yCor = 0; yCor < (2 * Ymax); yCor++) {
                for (int xCor = 0; xCor < (2 * Xmax); xCor++) {
                    V[xCor][yCor][zCor] = calcV(xCor + Xmin, yCor + Ymin,
zCor);
                    U[xCor][yCor][zCor] = calcU(xCor + Xmin, yCor + Ymin,
zCor);
                    W[xCor][yCor][zCor] = calcW(xCor + Xmin, yCor + Ymin,
zCor);
                    //pressure has an additional resolution variable for use
in Euler Integration
                    P[xCor][yCor][zCor] = calcP(xCor + Xmin, yCor + Ymin,
zCor,200);
                }
            }
        }
        try {
            System.out.println("Starting data export");
            //calls function that exports data into .csv format
            exportData();
        } catch (IOException e) {
            e.printStackTrace();
        }
        //initilializes the arrays used in particle tracer
        traceX = new float[(int) (T / dT)+1];
        traceY = new float[(int) (T / dT)+1];
        traceZ = new float[(int) (T / dT)+1];
        traceVx = new float[(int) (T / dT)+1];
        traceVy = new float[(int) (T / dT)+1];
        traceVz = new float[(int) (T / dT)+1];
        //starts timer for tracer
        long sT=System.currentTimeMillis();
        //begins particle trace using starting position of (400,400,1000) and
starting velocity of <0,0,0>
```

```
        partTrace(400,400,1000,0,0,0);
        //calculates and prints how long the "particle trace" method took
        System.out.println("Parts Took: "+(System.currentTimeMillis()-sT));
        try {
            //exports the particle data in .csv format
            System.out.println("Starting particle export");
            exportPartData();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("Done");
    }
    public static void exportPartData() throws IOException {
        //exports data for particles in .csv
        chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
        chooser.showSaveDialog(null);
        String filename = chooser.getSelectedFile().getPath();
        FileWriter writer = new FileWriter(filename + "\\PartData.csv");
        writer.append("Time");
        writer.append(",");
        writer.append("X0");
        writer.append(",");
        writer.append("Y0");
        writer.append(",");
        writer.append("Z0");
        writer.append(",");
        writer.append("Vpx");
        writer.append(",");
        writer.append("Vpy");
        writer.append(",");
        writer.append("Vpz");
        writer.append(",");
        writer.append("R");
        writer.append("\n");
        for (float t = 0; t < (100*T); t++) {
            Float calc=t/100;
            writer.append(""+calc);
            writer.append(",");
            calc=traceX[(int) t];
            writer.append(""+calc);
            writer.append(",");
            calc=traceY[(int) t];
            writer.append(""+calc);
            writer.append(",");
            calc=traceZ[(int) t];
            writer.append(""+calc);
            writer.append(",");
            calc=traceVx[(int) t];
            writer.append(""+calc);
            writer.append(",");
            calc=traceVy[(int) t];
            writer.append(""+calc);
            writer.append(",");
            calc=traceVz[(int) t];
            writer.append(""+calc);
            writer.append(",");
            calc=(float) Math.sqrt(traceX[(int)
t]*traceX[(int)t]+traceY[(int) t]*traceY[(int)t]);
            writer.append(""+calc);
            writer.append("\n");
            writer.flush();
        }

        writer.close();
    }
    public static void partTrace(float x, float y, float z, float vx, float
vy ,float vz) {
        //sets initial values for the particle
        float x0 = x, y0 = y, z0 = z, Vpx = vx, Vpy = vy, Vpz = vz;
        //initiliazes current time value
        double t = 0;
        //cT, when multiplied by t, calculates the correct array index for
the t value
        int cT=(int) (1/dT);
        //fills 0 slot in arrays with starting conditions
        traceX[(int) (t * cT)] = x0;
        traceY[(int) (t * cT)] = y0;
        traceZ[(int) (t * cT)] = z0;
        traceVx[(int) (t * cT)] = Vpx;
        traceVy[(int) (t * cT)] = Vpy;
        traceVz[(int) (t * cT)] = Vpz;
        //fills starting conditions into an array to be used in RK4
        double stats[]={x0,y0,z0,Vpx,Vpy,Vpz};
        //iterates through T values from 0 to T
        do {
            t += dT;
            //uses an fourth order Runge Kutta method to calculate the
x,y,z,vx,vy,vz at the next time step
            stats=rk41(T,dT,stats);
            //stores new x,y,z,Vx,Vy,Vz values into corresponding arrays at
correct index at restarts process till t=T
            traceX[(int) (t * cT)]=(float) stats[0];
            traceY[(int) (t * cT)]=(float) stats[1];
            traceZ[(int) (t * cT)]=(float) stats[2];
            traceVx[(int) (t * cT)]=(float) stats[3];
            traceVy[(int) (t * cT)]=(float) stats[4];
            traceVz[(int) (t * cT)]=(float) stats[5];
        } while (t <= T);
    }

    public static double[] rk41(double x, double h, double[] y) {
        //implements Fourth Order Runge Kutta with derivpd as f(),returns
array with all
        //timestep values x,y,z,Vx,Vy,Vz moved up a step
        double[] f = derivpd(x, y);
        double[] k = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4], h * f[5] };
        double[] a = new double[] { y[0] + 0.5 * k[0], y[1] + 0.5 * k[1],
                y[2] + 0.5 * k[2], y[3] + 0.5 * k[3], y[4] + 0.5 * k[4],
                y[5] + 0.5 * k[5] };
        f = derivpd(x + 0.5 * h, a);
        double[] k2 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4], h * f[5] };
        a = new double[] { y[0] + 0.5 * k2[0], y[1] + 0.5 * k2[1],
                y[2] + 0.5 * k2[2], y[3] + 0.5 * k[3], y[4] + 0.5 * k[4],
                y[5] + 0.5 * k[5] };
        f = derivpd(x + 0.5 * h, a);
        double[] k3 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4], h * f[5] };
        a = new double[] { y[0] + k3[0], y[1] + k3[1], y[2] + k3[2],
                y[3] + k3[3], y[4] + k3[4], y[5] + k3[5] };
        f = derivpd(x + h, a);
        double[] k4 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4], h * f[5] };
        a = new double[] { y[0] + (k[0] + 2 * (k2[0] + k3[0]) + k4[0]) / 6,
                y[1] + (k[1] + 2 * (k2[1] + k3[1]) + k4[1]) / 6,
                y[2] + (k[2] + 2 * (k2[2] + k3[2]) + k4[2]) / 6,
                y[3] + (k[3] + 2 * (k2[3] + k3[3]) + k4[3]) / 6,
                y[4] + (k[4] + 2 * (k2[4] + k3[4]) + k4[4]) / 6,
                y[5] + (k[5] + 2 * (k2[5] + k3[5]) + k4[5]) / 6 };
        return a;
    }
    public static double[] derivpd(double t,double[]y){
        //gets the i,j,k unit vector values calculated by Kuos model at the
particles current postion
        int ix=(int) ((y[0]/resolution)+Xmax);
        int iy=(int) ((y[1]/resolution)+Ymax);
        int iz=(int) ((y[2]/resolution));
        float Vfx=0,Vfy=0,Vfz=0;
        Vfx=Vx[ix][iy][iz];
        Vfy=Vy[ix][iy][iz];
        Vfz=W[ix][iy][iz];
        //calculates the particles next Vx,Vy,Vz
        double Fx=(Cdx*Math.abs(Vfx-y[3])*(Vfx-y[3]))/Mp;
        double Fy=(Cdy*Math.abs(Vfy-y[4])*(Vfy-y[4]))/Mp;
        double Fz=(Cdz*Math.abs(Vfz-y[5])*(Vfz-y[5]))/Mp-9.81;
        //returns new x,y,z,Vx,Vy,Vz values for next time step
        double a[]= {y[3],y[4],y[5],Fx,Fy,Fz};
        return a;
    }
    public static void exportData() throws IOException {
        //exports data using the .csv file format
        chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
        chooser.showSaveDialog(null);
        String filename = chooser.getSelectedFile().getPath();
        float v, u, w, angle, pr, f, df, m;
        FileWriter writer = new FileWriter(filename + "\\VectData.csv");
        int counter = 0;
        writer.append("XCor");
        writer.append(",");
        writer.append("YCor");
        writer.append(",");
        writer.append("ZCor");
        writer.append(",");
        writer.append("r");
        writer.append(",");
        writer.append("Vx");
        writer.append(",");
        writer.append("Vy");
        writer.append(",");
        writer.append("Vz");
        writer.append(",");
        writer.append("v");
        writer.append(",");
        writer.append("u");
        writer.append(",");
        writer.append("w");
        writer.append(",");
        writer.append("f");
        writer.append(",");
        writer.append("df");
        writer.append(",");
        writer.append("m");
        writer.append(",");
        writer.append("Pressure");
        writer.append("\n");
        for (int zCor = 0; zCor < Zmax; zCor++) {
            for (int yCor = 0; yCor < (2 * Ymax); yCor++) {
                for (int xCor = 0; xCor < (2 * Xmax); xCor++) {
                    v = V[xCor][yCor][zCor];
                    w = W[xCor][yCor][zCor];
                    u = U[xCor][yCor][zCor];
                    pr = P[xCor][yCor][zCor];
                    f = F[xCor][yCor];
                    df = dF[xCor][yCor];
                    m = M[xCor][yCor];
                    if ((xCor + Xmin) * deltaR == 0
                            && (yCor + Ymin) * deltaR == 0)
                        angle = 0;
                    else {
                        angle = (float) Math.atan2((yCor + Ymin) * deltaR,
                                (xCor + Xmin) * deltaR);
                        if (angle < 0)
                            angle = (float) ((2 * Math.PI) + angle);
                    }
                    writer.write("" + ((xCor + Xmin) * deltaR));
                    writer.write(",");
                    writer.write("" + ((yCor + Ymin) * deltaR));
                    writer.write(",");
                    writer.write("" + (zCor * deltaR));
                    writer.write(",");
                    float x = (xCor + Xmin) * deltaR;
                    float y = (yCor + Ymin) * deltaR;
                    Float calc = (float) Math.sqrt(x * x + y * y);
                    writer.write("" + calc);
                    writer.write(",");
                    calc = (float) ((u * Math.cos(angle)) - (v * Math
                            .sin(angle)));
                    Vx[xCor][yCor][zCor] = calc;
                    writer.write("" + calc);
                    writer.write(",");
                    calc = (float) ((u * Math.sin(angle)) + (v * Math
                            .cos(angle)));
                    Vy[xCor][yCor][zCor] = calc;
                    writer.write("" + calc);
                    writer.write(",");
                    writer.write("" + w);
```

```
                    writer.write(",");
                    writer.write("" + v);
                    writer.write(",");
                    writer.write("" + u);
                    writer.write(",");
                    writer.write("" + w);
                    writer.write(",");
                    writer.append("" + f);
                    writer.append(",");
                    writer.append("" + df);
                    writer.append(",");
                    writer.append("" + m);
                    writer.append(",");
                    writer.write("" + pr);
                    writer.write("\n");
                    counter++;
                    writer.flush();
                }
            }
        }

        writer.close();
        System.out.println(counter);
    }


    public static float calcP(float x1, float y1, float z1, int res) {
        //calculates pressure at each x,y,z gridpoint, res dictates
"resolution" of Euler integration
        float x = deltaR * x1, y = deltaR * y1, z = deltaR * z1;
        float r = (float) Math.sqrt(x * x + y * y);
        if (r == 0)
            r += 0.1;
        float X = calcX(r);
        float total = 0;
        float dR = X / res;
        for (int x2 = 2; x2 < res + 1; x2++) {
            float b = x2 * dR;
            float bn = (x2 - 1) * dR;
            float m = M[(int) (x1 + Xmax)][(int) (y1 + Ymax)];
            float integ = (float) (0.5 * ((m * m / (b * b)) + (m * m / (bn *
bn))) * dR);
            total += integ;
        }
        return aP + ((b * p * R * R) / (8 * eV)) * total;
    }

    public static float calcV(float x1, float y1, float z1) {
        //returns V value at each gridpoint
        float x = deltaR * x1, y = deltaR * y1;
        float r = (float) Math.sqrt(x * x + y * y);
        float m = M[(int) (x1 + Xmax)][(int) (y1 + Ymax)];
        if (r == 0)
            r += 0.1;
        if (x == 0 && y == -1000 && z1 == 0)
            System.out.println(m + " " + x1 + " " + y1 + " " + ((R / r) *
m));
```

```
        return (R / r) * m;
    }

    public static float calcU(float x1, float y1, float z1) {
        //returns U value at each gridpoint
        float x = deltaR * x1, y = deltaR * y1;
        float r = (float) Math.sqrt(x * x + y * y);
        float f = F[(int) (x1 + Xmax)][(int) (y1 + Ymax)];
        if (r == 0)
            r += 0.1;
        return (-10 * eV / r) * f;
    }

    public static float calcW(float x1, float y1, float z1) {
        //returns W value at each gridpoint
        float z = deltaR * z1;
        float df = dF[(int) (x1 + Xmax)][(int) (y1 + Ymax)];
        return b * z * df;
    }

    public static void calcODE() {
        //calculates the various ODE values at each gridpoint in quadrant 1
        //takes advantage of axi-symettry to translate to other quandrants
        //stores into necessary arrays
        for (int x = 0; x <= resolution; x++) {
            for (int y = 0; y <= resolution; y++) {
                float x1 = (x + Xmin) * deltaR;
                float y1 = (y + Ymin) * deltaR;
                float r = (float) Math.sqrt((x1 * x1) + (y1 * y1));
                float[] a = calcODE(calcX(r));
                F[x][y] = a[0];
                dF[x][y] = a[1];
                M[x][y] = a[3];
            }
        }
        for (int x = 0; x <= resolution; x++) {
            for (int y = (resolution * 2) - 1; y >= resolution; y--) {
                F[x][y] = F[x][(resolution * 2) - y];
                dF[x][y] = dF[x][(resolution * 2) - y];
                M[x][y] = M[x][(resolution * 2) - y];
            }
        }
        for (int x = (resolution * 2) - 1; x >= resolution; x--) {
            for (int y = (resolution * 2) - 1; y >= resolution; y--) {

                F[x][y] = F[(resolution * 2) - x][(resolution * 2) - y];
                dF[x][y] = dF[(resolution * 2) - x][(resolution * 2) - y];
                M[x][y] = M[(resolution * 2) - x][(resolution * 2) - y];
            }
        }
        for (int x = (resolution * 2) - 1; x >= resolution; x--) {
            for (int y = 0; y <= resolution; y++) {

                F[x][y] = F[(resolution * 2) - x][y];
                dF[x][y] = dF[(resolution * 2) - x][y];
                M[x][y] = M[(resolution * 2) - x][y];
            }
        }
    }

    public static float calcX(float r) {
        //returns the x value when given r
        return b * r * r / (4 * eV);
    }

    public static float[] calcODE(float X) {
        //iterates through X and uses RK4 loop to get correct ODE values and
return array in [F,F',F'',m,m'] format
        double x, dx;
        x = 0.01;
        dx = b / eV;
        double[] y = shoot;

        while (x < X) {
            x += dx;
            y = rk4(x, dx, y);

        }
        float[] a = new float[y.length];
        for (int i = 0; i < y.length; i++) {
            a[i] = (float) y[i];
        }
        return a;
    }

    public static double[] rk4(double x, double h, double[] y) {
        //solves ODE used in Kuo's equation through RK4 methodology
        double[] f = integ(x, y);
        double[] k = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4] };
        double[] a = new double[] { y[0] + 0.5 * k[0], y[1] + 0.5 * k[1],
                y[2] + 0.5 * k[2], y[3] + 0.5 * k[3], y[4] + 0.5 * k[4] };
        f = integ(x + 0.5 * h, a);
        double[] k2 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4] };
        a = new double[] { y[0] + 0.5 * k2[0], y[1] + 0.5 * k2[1],
                y[2] + 0.5 * k2[2], y[3] + 0.5 * k[3], y[4] + 0.5 * k[4] };
        f = integ(x + 0.5 * h, a);
        double[] k3 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4] };
        a = new double[] { y[0] + k3[0], y[1] + k3[1], y[2] + k3[2],
                y[3] + k3[3], y[4] + k3[4] };
        f = integ(x + h, a);
        double[] k4 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4] };
        a = new double[] { y[0] + (k[0] + 2 * (k2[0] + k3[0]) + k4[0]) / 6,
                y[1] + (k[1] + 2 * (k2[1] + k3[1]) + k4[1]) / 6,
                y[2] + (k[2] + 2 * (k2[2] + k3[2]) + k4[2]) / 6,
                y[3] + (k[3] + 2 * (k2[3] + k3[3]) + k4[3]) / 6,
                y[4] + (k[4] + 2 * (k2[4] + k3[4]) + k4[4]) / 6 };
        return a;
    }
    public static double[] integ(double x, double[] y) {
        //represents ODE fed as f() into RK4
        double F = y[0];
        double dF = y[1];
        double ddF = y[2];
        double dddF = -(((F + 1) * ddF) - (dF * (dF - 1))) / x;
        double dM = y[4];
        double ddM = (-F * dM) / x;
        return new double[] { dF, ddF, dddF, dM, ddM };
    }

    public static void genShoot() {
        //uses various shooting methods to find correct initial values to
store in shoot array
        //uses bisect algorithm
        double o = bisect(-0.9, -0.8);
        double o1 = o * (o - 1);
        double o2 = bisect1(0.1, 0.13, o, o1);
        double yinit[] = { 0, o, o1, 0, o2 };
        shoot = yinit;
        System.out.println("Shooter: " + o + " " + o1 + " " + o2);
    }

    public static double bisect(double minVal, double maxVal) {
        //bisection algorithm specially configured to approximate F value
        double x = minVal;
        double fmin = F(minVal);
        double tol = 0.003;
        int maxiter = 100;
        int k = 0;
        while (k <= maxiter) {
            x = (minVal + maxVal) / 2.0;
            double fx = F(x);
            k += 1;
            if ((Math.abs(maxVal - minVal) / 2.0) < tol)
                break;
            if (sign(fmin) == sign(fx)) {
                minVal = x;
                fmin = fx;
            } else
                maxVal = x;
        }
        if (k > maxiter)
            System.err.println("Exceeded Maxiter");
        return x;
    }

    public static double bisect1(double minVal, double maxVal, double o1,
            double o2) {
        //bisection algorithm specially configured to approximate m value
        double x = minVal;
        double fmin = M(minVal, o1, o2);
        double tol = 0.003;
        int maxiter = 100;
        int k = 0;
        while (k <= maxiter) {
            x = (minVal + maxVal) / 2.0;
            double fx = M(x, o1, o2);
            k += 1;
            if ((Math.abs(maxVal - minVal) / 2.0) < tol)
                break;
            if (sign(fmin) == sign(fx)) {
                minVal = x;
                fmin = fx;
            } else
                maxVal = x;
        }
        if (k > maxiter)
            System.err.println("Exceeded Maxiter");
        return x;
    }

    public static double F(double e) {
        //returns F as requested by first bisect method
        double f1 = e;
        double f2 = f1 * (f1 - 1);
        double yinit[] = { 0.0, f1, f2, 0.0, 0.12 };
        yinit = calcODE1(yinit, 200);
        return yinit[1];
    }

    public static double M(double e, double o1, double o2) {
        //returns m as requested by second bisect method
        double m1 = e;
        double yinit[] = { 0.0, o1, o2, 0.0, m1 };
        yinit = calcODE1(yinit, 200);
        return yinit[4];
    }

    public static double[] calcODE1(double y[], double X) {
        //modified method of previous calcODE method specifically made to
work with bisect methods
        double x = 0.01;
        double dx = 0.01;
        while (x < X) {
            x += dx;
            y = rk4(x, dx, y);

        }
        return y;
    }

    public static int sign(double x) {
        //returns int depending on sign of x value
        if (x < 0)
            return -1;
        else if (x > 0)
            return 1;
        else
            return 0;
    }
}
```

KuoTestMP.java

```
import java.io.FileWriter;
import java.io.IOException;
import java.math.*;
import javax.swing.JFileChooser;
//implementation of Kuo's equations in Java utilizing Multithraeding with
particles
public class KuoTestMP {

    // eddy viscosity
    private static final float eV = 5;
    private static JFileChooser chooser = new JFileChooser();
    // circulation strength of vortex
    private static final float R = 7500;
    // suction strength of vortex
    private static final float b = (float) 0.01;
    // ambient pressure
    private static final float aP = 10000;
    //density
    private static final float p = (float) 1;
    //Program will calculate x,y,z values between -RMax to RMax
    private static final float RMax=1000;
    //number of grid points between origin and maximum
    private static final int resolution = 50;
    //distance between each grid point ie. RMax/resolution
    private static float deltaR;
    //symbolizes -resolution,resolution,-resolution,resolution,resolution
respectively
    //acts as syntacticly friendly replacer for resolution during matrix
corrections
    private static float Xmin, Xmax, Ymin, Ymax, Zmax;
    //3d array containing all the V unit vector values at each grid point
    private static float[][][] V;
    //3d array containing all the U unit vector values at each grid point
    private static float[][][] U;
    //3d array containing all the W unit vector values at each grid point
    private static float[][][] W;
    //3d array containing all the i unit vector values at each grid point
    private static float[][][] Vx;
    //3d array containing all the j unit vector values at each grid point
    private static float[][][] Vy;
    //3d array containing all the pressure values at each grid point
    private static float[][][] P;
    //2d arrays containing all values of the F, F', and M ODEs at each grid
point
    private static float[][] F, dF, M;
    //array in which our shooting method will store initial values for RK4
method
    private static double[] shoot;
    //maximum time the particle tracer program will iterate to
    private static float T = 250;
    //time step used by the particle tracer program
    private static float dT = (float) 0.01;
    //filepath name for use in exports
    private static String file;
    public static void main(String[] args) {
                //calculates the deltaR for program
                deltaR = (RMax) / resolution;
                System.out.println(deltaR + " " + RMax);
                //sets up the Xmin,Xmax,Ymin,Ymax,Zmax variables for matrix
calculations
                Xmin = -resolution;
                Xmax = resolution;
                Ymin = -resolution;
                Ymax = resolution;
                Zmax = resolution;
                //initialializes the size for each of the above described
arrays
                //When resolution=50 there will be -50 to 50 grid points in x
and y axis and 0-50 in z
                //The matrix and set up so that each grid point has a slot in
the array
                V = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int)
Zmax];
                U = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int)
Zmax];
                W = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int)
Zmax];
                Vx = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int)
Zmax];
                Vy = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int)
Zmax];
                P = new float[(int) (Xmax * 2)][(int) (Ymax * 2)][(int)
Zmax];
                F = new float[(int) (Xmax * 2)][(int) (Ymax * 2)];
                dF = new float[(int) (Xmax * 2)][(int) (Ymax * 2)];
                M = new float[(int) (Xmax * 2)][(int) (Ymax * 2)];
                System.out.println("generating velocity integrals...");
                //uses shooting method to initialize initial value which will
be stored in shoot
                genShoot();
                //evaluates the ODE and stores in F,dF,M arrays
                genODE();
                //for loops calculate the V U W unit vector and pressure
values for each grid point
                //Since [0][0][0] corresponds to (-resolution,-resolution,0)
Xmin is added to the X coordinate and
                //Ymin to the Y coordinate to achieve the neccesary matrix
corrections
                for (int zCor = 0; zCor < Zmax; zCor++) {
                    for (int yCor = 0; yCor < (2 * Ymax); yCor++) {
                        for (int xCor = 0; xCor < (2 * Xmax); xCor++) {
                            V[xCor][yCor][zCor] = calcV(xCor + Xmin, yCor +
Ymin, zCor);
                            U[xCor][yCor][zCor] = calcU(xCor + Xmin, yCor +
Ymin, zCor);
                            W[xCor][yCor][zCor] = calcW(xCor + Xmin, yCor +
Ymin, zCor);
                            //pressure has an additional resolution variable
for use in Euler Integration
```

```
                            P[xCor][yCor][zCor] = calcP(xCor + Xmin, yCor +
Ymin, zCor,200);
                        }
                    }
                }
                try {
                    System.out.println("Starting data export");
                    //calls function that exports data into .csv format
                    exportData();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            //each of the following initializes a new thread object with a
runnable that will execute the operations
            //to trace one individual particle
            //they run simultaneously resulting in speed up
            Thread t = new Thread(new Runnable() {
                public void run() {
                    try {
                        partTrace(50, 50, 990, 0, 0,
0,1,(float)0.1,(float)0.1,(float)0.1,1);
                    } catch (IOException e) {

                    }
                }
            });
            t.start();
            Thread t1 = new Thread(new Runnable() {
                public void run() {
                    try {
                        partTrace(50, 50, 990, 0, 0,
0,1,(float)0.25,(float)0.25,(float)0.25,2);
                    } catch (IOException e) {

                    }
                }
            });
            t1.start();
            Thread t2 = new Thread(new Runnable() {
                public void run() {
                    try {
                        partTrace(50, 50, 990, 0, 0,
0,10,(float)0.5,(float)0.5,(float)0.5,3);
                    } catch (IOException e) {

                    }
                }
            });
            t2.start();
            Thread t3 = new Thread(new Runnable() {
                public void run() {
                    try {
                        partTrace(50, 50, 990, 0, 0,
0,10,(float)1,(float)1,(float)1,4);
                    } catch (IOException e) {

                    }
                }
            });
            t3.start();
            Thread t4 = new Thread(new Runnable() {
                public void run() {
                    try {
                        partTrace(50, 50, 990, 0, 0,
0,100,(float)3.2,(float)3.2,(float)3.2,5);
                    } catch (IOException e) {

                    }
                }
            });
            t4.start();
    }
    public static void partTrace(float x, float y, float z, float vx, float
vz,
            float vy,float Mp, float Cdx,float Cdy,float Cdz, int
partCounter) throws IOException {
        //multithreaded version of partTrace method used in KuoTest
        //uses Lagrangian particle equations to calculate motion of particle
given surrounding volecities
        //cheif difference is this loop uses compartmentalized variables
rather than global to allow for multithreading
        float[] traceX = new float[(int) (T / dT) + 1];
        float[] traceY = new float[(int) (T / dT) + 1];
        float[] traceZ = new float[(int) (T / dT) + 1];
        float[] traceVx = new float[(int) (T / dT) + 1];
        float[] traceVy = new float[(int) (T / dT) + 1];
        float[] traceVz = new float[(int) (T / dT) + 1];
        long sT = System.currentTimeMillis();
        //sets initial values for the particle
                float x0 = x, y0 = y, z0 = z, Vpx = vx, Vpy = vy, Vpz = vz;
                //initiliazes current time value
                double t = 0;
                //cT, when multiplied by t, calculates the correct array
index for the t value
                int cT=(int) (1/dT);
                //fills 0 slot in arrays with starting conditions
                traceX[(int) (t * cT)] = x0;
                traceY[(int) (t * cT)] = y0;
                traceZ[(int) (t * cT)] = z0;
                traceVx[(int) (t * cT)] = Vpx;
                traceVy[(int) (t * cT)] = Vpy;
                traceVz[(int) (t * cT)] = Vpz;
                //fills starting conditions into an array to be used in RK4
                double stats[]={x0,y0,z0,Vpx,Vpy,Vpz};
                //iterates through T values from 0 to T
                do {
                    t += dT;
                    //uses an fourth order Runge Kutta method to calculate
the x,y,z,vx,vy,vz at the next time step
                    stats=rk41(T,dT,stats,Mp,Cdx,Cdy,Cdz);
                    //stores new x,y,z,Vx,Vy,Vz values into corresponding
arrays at correct index at restarts process till t=T
                    traceX[(int) (t * cT)]=(float) stats[0];
                    traceY[(int) (t * cT)]=(float) stats[1];
                    traceZ[(int) (t * cT)]=(float) stats[2];
                    traceVx[(int) (t * cT)]=(float) stats[3];
                    traceVy[(int) (t * cT)]=(float) stats[4];
                    traceVz[(int) (t * cT)]=(float) stats[5];
                } while (t <= T);

        System.out.println("Part trace took: "
                + (System.currentTimeMillis() - sT)+" "+partCounter);
        FileWriter writer = new FileWriter(file + "\\PartData" + partCounter
                + ".csv");
        writer.append("Time");
        writer.append(",");
        writer.append("X0");
        writer.append(",");
        writer.append("Y0");
        writer.append(",");
        writer.append("Z0");
        writer.append(",");
        writer.append("Vpx");
        writer.append(",");
        writer.append("Vpy");
        writer.append(",");
        writer.append("Vpz");
        writer.append(",");
        writer.append("R");
        writer.append("\n");
        for (float i = 0; i < (100 * T); i++) {
            Float calc = i / 100;
            writer.append("" + calc);
            writer.append(",");
            calc = traceX[(int) i];
            writer.append("" + calc);
            writer.append(",");
            calc = traceY[(int) i];
            writer.append("" + calc);
            writer.append(",");
            calc = traceZ[(int) i];
            writer.append("" + calc);
            writer.append(",");
            calc = traceVx[(int) i];
            writer.append("" + calc);
            writer.append(",");
            calc = traceVy[(int) i];
            writer.append("" + calc);
            writer.append(",");
            calc = traceVz[(int) i];
            writer.append("" + calc);
            writer.append(",");
            calc = (float) Math.sqrt(traceVx[(int) i] * traceVx[(int) i]
                    + traceVy[(int) i] * traceVy[(int) i]);
            writer.append("" + calc);
            writer.append("\n");
            writer.flush();
        }
        writer.close();
    }
    public static double[] rk41(double x, double h, double[] y,float Mp,
float Cdx, float Cdy, float Cdz) {
        //implements Fourth Order Runge Kutta with derivpd as f(),returns
array with all
        //timestep values x,y,z,Vx,Vy,Vz moved up a step
        double[] f = derivpd(x, y,Mp,Cdx,Cdy,Cdz);
        double[] k = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4], h * f[5] };
        double[] a = new double[] { y[0] + 0.5 * k[0], y[1] + 0.5 * k[1],
                y[2] + 0.5 * k[2], y[3] + 0.5 * k[3], y[4] + 0.5 * k[4],
                y[5] + 0.5 * k[5] };
        f = derivpd(x + 0.5 * h, a,Mp,Cdx,Cdy,Cdz);
        double[] k2 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4], h * f[5] };
        a = new double[] { y[0] + 0.5 * k2[0], y[1] + 0.5 * k2[1],
                y[2] + 0.5 * k2[2], y[3] + 0.5 * k[3], y[4] + 0.5 * k[4],
                y[5] + 0.5 * k[5] };
        f = derivpd(x + 0.5 * h, a,Mp,Cdx,Cdy,Cdz);
        double[] k3 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4], h * f[5] };
        a = new double[] { y[0] + k3[0], y[1] + k3[1], y[2] + k3[2],
                y[3] + k3[3], y[4] + k3[4], y[5] + k3[5] };
        f = derivpd(x + h, a,Mp,Cdx,Cdy,Cdz);
        double[] k4 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4], h * f[5] };
        a = new double[] { y[0] + (k[0] + 2 * (k2[0] + k3[0]) + k4[0]) / 6,
                y[1] + (k[1] + 2 * (k2[1] + k3[1]) + k4[1]) / 6,
                y[2] + (k[2] + 2 * (k2[2] + k3[2]) + k4[2]) / 6,
                y[3] + (k[3] + 2 * (k2[3] + k3[3]) + k4[3]) / 6,
                y[4] + (k[4] + 2 * (k2[4] + k3[4]) + k4[4]) / 6,
                y[5] + (k[5] + 2 * (k2[5] + k3[5]) + k4[5]) / 6 };
        return a;
    }

    public static double[] derivpd(double t, double[] y,float Mp, float
Cdx,float Cdy, float Cdz) {
        //gets the i,j,k unit vector values calculated by Kuos model at the
particles current postion
        int ix=(int) ((y[0]/resolution)+Xmax);
        int iy=(int) ((y[1]/resolution)+Ymax);
        int iz=(int) ((y[2]/resolution));
        float Vfx=0,Vfy=0,Vfz=0;
        Vfx=Vx[ix][iy][iz];
        Vfy=Vy[ix][iy][iz];
        Vfz=W[ix][iy][iz];
        //calculates the particles next Vx,Vy,Vz
        double Fx=(Cdx*Math.abs(Vfx-y[3])*(Vfx-y[3]))/Mp;
        double Fy=(Cdy*Math.abs(Vfy-y[4])*(Vfy-y[4]))/Mp;
        double Fz=(Cdz*Math.abs(Vfz-y[5])*(Vfz-y[5]))/Mp-9.81;
        //returns new x,y,z,Vx,Vy,Vz values for next time step
        double a[]= {y[3],y[4],y[5],Fx,Fy,Fz};
        return a;
    }

    public static void exportData() throws IOException {
        //exports data using the .csv file format
        chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
        chooser.showSaveDialog(null);
        file = chooser.getSelectedFile().getPath();
        float v, u, w, angle, pr, f, df, m;
        FileWriter writer = new FileWriter(file + "\\VectData.csv");
        int counter = 0;
        writer.append("XCor");
        writer.append(",");
        writer.append("YCor");
        writer.append(",");
        writer.append("ZCor");
        writer.append(",");
        writer.append("r");
        writer.append(",");
        writer.append("Vx");
        writer.append(",");
        writer.append("Vy");
        writer.append(",");
        writer.append("Vz");
        writer.append(",");
        writer.append("v");
        writer.append(",");
        writer.append("u");
        writer.append(",");
        writer.append("w");
        writer.append(",");
        writer.append("f");
        writer.append(",");
        writer.append("df");
        writer.append(",");
        writer.append("m");
        writer.append(",");
        writer.append("Pressure");
        writer.append("\n");
        for (int zCor = 0; zCor < Zmax; zCor++) {
            for (int yCor = 0; yCor < (2 * Ymax); yCor++) {
                for (int xCor = 0; xCor < (2 * Xmax); xCor++) {
                    v = V[xCor][yCor][zCor];
                    w = W[xCor][yCor][zCor];
                    u = U[xCor][yCor][zCor];
                    pr = P[xCor][yCor][zCor];
                    f = F[xCor][yCor];
                    df = dF[xCor][yCor];
                    m = M[xCor][yCor];
                    if ((xCor + Xmin) * deltaR == 0
                            && (yCor + Ymin) * deltaR == 0)
                        angle = 0;
                    else {
                        angle = (float) Math.atan2((yCor + Ymin) * deltaR,
                                (xCor + Xmin) * deltaR);
                        if (angle < 0)
                            angle = (float) ((2 * Math.PI) + angle);
                    }
                    writer.write("" + ((xCor + Xmin) * deltaR));
                    writer.write(",");
                    writer.write("" + ((yCor + Ymin) * deltaR));
                    writer.write(",");
                    writer.write("" + (zCor * deltaR));
```

```
                        writer.write(",");
                        float x = (xCor + Xmin) * deltaR;
                        float y = (yCor + Ymin) * deltaR;
                        Float calc = (float) Math.sqrt(x * x + y * y);
                        writer.write("" + calc);
                        writer.write(",");
                        calc = (float) ((u * Math.cos(angle)) - (v * Math
                                .sin(angle)));
                        Vx[xCor][yCor][zCor] = calc;
                        writer.write("" + calc);
                        writer.write(",");
                        calc = (float) ((u * Math.sin(angle)) + (v * Math
                                .cos(angle)));
                        Vy[xCor][yCor][zCor] = calc;
                        writer.write("" + calc);
                        writer.write(",");
                        writer.write("" + w);
                        writer.write(",");
                        writer.write("" + v);
                        writer.write(",");
                        writer.write("" + u);
                        writer.write(",");
                        writer.write("" + w);
                        writer.write(",");
                        writer.append("" + f);
                        writer.append(",");
                        writer.append("" + df);
                        writer.append(",");
                        writer.append("" + m);
                        writer.append(",");
                        writer.write("" + pr);
                        writer.write("\n");
                        counter++;
                        writer.flush();
                    }
                }
            }

            writer.close();
            System.out.println(counter);
        }


    public static float calcP(float x1, float y1, float z1, int res) {
        //calculates pressure at each x,y,z gridpoint, res dictates
"resolution" of Euler integration
        float x = deltaR * x1, y = deltaR * y1, z = deltaR * z1;
        float r = (float) Math.sqrt(x * x + y * y);
        if (r == 0)
            r += 0.1;
        float X = calcX(r);
        float total = 0;
        float dR = X / res;
        for (int x2 = 2; x2 < res + 1; x2++) {
            float b = x2 * dR;
            float bn = (x2 - 1) * dR;
            float m = M[(int) (x1 + Xmax)][(int) (y1 + Ymax)];
            float integ = (float) (0.5 * ((m * m / (b * b)) + (m * m / (bn *
bn))) * dR);
            total += integ;
        }
        return aP + ((b * p * R * R) / (8 * eV)) * total;
    }

    public static float calcV(float x1, float y1, float z1) {
        //returns V value at each gridpoint
        float x = deltaR * x1, y = deltaR * y1;
        float r = (float) Math.sqrt(x * x + y * y);
        float m = M[(int) (x1 + Xmax)][(int) (y1 + Ymax)];
        if (r == 0)
            r += 0.1;
        if (x == 0 && y == -1000 && z1 == 0)
            System.out.println(m + " " + x1 + " " + y1 + " " + ((R / r) *
m));

        return (R / r) * m;
    }

    public static float calcU(float x1, float y1, float z1) {
        //returns U value at each gridpoint
        float x = deltaR * x1, y = deltaR * y1;
        float r = (float) Math.sqrt(x * x + y * y);
        float f = F[(int) (x1 + Xmax)][(int) (y1 + Ymax)];
        if (r == 0)
            r += 0.1;
        return (-10 * eV / r) * f;
    }

    public static float calcW(float x1, float y1, float z1) {
        //returns W value at each gridpoint
        float z = deltaR * z1;
        float df = dF[(int) (x1 + Xmax)][(int) (y1 + Ymax)];
        return b * z * df;
    }

    public static void genODE() {
        //calculates the various ODE values at each gridpoint in quadrant 1
        //takes advantage of axi-symettry to translate to other quandrants
        //stores into necessary arrays
        for (int x = 0; x <= resolution; x++) {
            for (int y = 0; y <= resolution; y++) {
                float x1 = (x + Xmin) * deltaR;
                float y1 = (y + Ymin) * deltaR;
                float r = (float) Math.sqrt((x1 * x1) + (y1 * y1));
                float[] a = calcODE(calcX(r));
                F[x][y] = a[0];
                dF[x][y] = a[1];
                M[x][y] = a[3];
                // if(x==y)
                // System.out.println("X "+x1+" Y ="+y1+" "+F[x][y]+"
"+calcX(r)+" "+r);
            }
        }
        for (int x = 0; x <= resolution; x++) {
            for (int y = (resolution * 2) - 1; y >= resolution; y--) {
                F[x][y] = F[x][(resolution * 2) - y];
                dF[x][y] = dF[x][(resolution * 2) - y];
                M[x][y] = M[x][(resolution * 2) - y];
            }
        }
        for (int x = (resolution * 2) - 1; x >= resolution; x--) {
            for (int y = (resolution * 2) - 1; y >= resolution; y--) {

                F[x][y] = F[(resolution * 2) - x][(resolution * 2) - y];
                dF[x][y] = dF[(resolution * 2) - x][(resolution * 2) - y];
                M[x][y] = M[(resolution * 2) - x][(resolution * 2) - y];
            }
        }
        for (int x = (resolution * 2) - 1; x >= resolution; x--) {
            for (int y = 0; y <= resolution; y++) {

                F[x][y] = F[(resolution * 2) - x][y];
                dF[x][y] = dF[(resolution * 2) - x][y];
                M[x][y] = M[(resolution * 2) - x][y];
            }
        }
    }

    public static float calcX(float r) {
        //returns the x value when given r
        return b * r * r / (4 * eV);
    }

    public static float[] calcODE(float X) {
        //iterates through X and uses RK4 loop to get correct ODE values and
return array in [F,F',F'',m,m'] format
        double x, dx;
        x = 0.01;
        dx = b / eV;
        double[] y = shoot;

        while (x < X) {
            x += dx;
            y = rk4(x, dx, y);

        }
        float[] a = new float[y.length];
        for (int i = 0; i < y.length; i++) {
            a[i] = (float) y[i];
        }
        return a;
    }

    public static double[] rk4(double x, double h, double[] y) {
        //solves ODE used in Kuo's equation through RK4 methodology
        double[] f = integ(x, y);
        double[] k = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4] };
        double[] a = new double[] { y[0] + 0.5 * k[0], y[1] + 0.5 * k[1],
                y[2] + 0.5 * k[2], y[3] + 0.5 * k[3], y[4] + 0.5 * k[4] };
        f = integ(x + 0.5 * h, a);
        double[] k2 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4] };
        a = new double[] { y[0] + 0.5 * k2[0], y[1] + 0.5 * k2[1],
                y[2] + 0.5 * k2[2], y[3] + 0.5 * k[3], y[4] + 0.5 * k[4] };
        f = integ(x + 0.5 * h, a);
        double[] k3 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4] };
        a = new double[] { y[0] + k3[0], y[1] + k3[1], y[2] + k3[2],
                y[3] + k3[3], y[4] + k3[4] };
        f = integ(x + h, a);
        double[] k4 = new double[] { h * f[0], h * f[1], h * f[2], h * f[3],
                h * f[4] };
        a = new double[] { y[0] + (k[0] + 2 * (k2[0] + k3[0]) + k4[0]) / 6,
                y[1] + (k[1] + 2 * (k2[1] + k3[1]) + k4[1]) / 6,
                y[2] + (k[2] + 2 * (k2[2] + k3[2]) + k4[2]) / 6,
                y[3] + (k[3] + 2 * (k2[3] + k3[3]) + k4[3]) / 6,
                y[4] + (k[4] + 2 * (k2[4] + k3[4]) + k4[4]) / 6 };
        return a;
    }

    public static double[] integ(double x, double[] y) {
        //represents ODE fed as f() into RK4
        double F = y[0];
        double dF = y[1];
        double ddF = y[2];
        double dddF = -(((F + 1) * ddF) - (dF * (dF - 1))) / x;
        double dM = y[4];
        double ddM = (-F * dM) / x;
        return new double[] { dF, ddF, dddF, dM, ddM };
    }

    public static void genShoot() {
        //uses various shooting methods to find correct initial values to
store in shoot array
        //uses bisect algorithm
        double o = bisect(0.8,0.9);
        double o1 = o * (o - 1);
        double o2 = bisect1(0.1, 0.13, o, o1);
        double yinit[] = { 0, o, o1, 0, o2 };
        shoot = yinit;
        System.out.println("Shooter: " + o + " " + o1 + " " + o2);
    }

    public static double bisect(double minVal, double maxVal) {
        //bisection algorithm specially configured to approximate F value
        double x = minVal;
        double fmin = F(minVal);
        double tol = 0.003;
        int maxiter = 100;
        int k = 0;
        while (k <= maxiter) {
            x = (minVal + maxVal) / 2.0;
            double fx = F(x);
            k += 1;
            if ((Math.abs(maxVal - minVal) / 2.0) < tol)
                break;
            if (sign(fmin) == sign(fx)) {
                minVal = x;
                fmin = fx;
            } else
                maxVal = x;
        }
        if (k > maxiter)
            System.err.println("Exceeded Maxiter");
        return x;
    }

    public static double bisect1(double minVal, double maxVal, double o1,
            double o2) {
        //bisection algorithm specially configured to approximate m value
        double x = minVal;
        double fmin = M(minVal, o1, o2);
        double tol = 0.003;
        int maxiter = 100;
        int k = 0;
        while (k <= maxiter) {
            x = (minVal + maxVal) / 2.0;
            double fx = M(x, o1, o2);
            k += 1;
            if ((Math.abs(maxVal - minVal) / 2.0) < tol)
                break;
            if (sign(fmin) == sign(fx)) {
                minVal = x;
                fmin = fx;
            } else
                maxVal = x;
        }
        if (k > maxiter)
            System.err.println("Exceeded Maxiter");
        return x;
    }

    public static double F(double e) {
        //returns F as requested by first bisect method
        double f1 = e;
        double f2 = f1 * (f1 - 1);
        double yinit[] = { 0.0, f1, f2, 0.0, 0.12 };
        yinit = calcInteg1(yinit, 200);
        return yinit[1];
    }

    public static double M(double e, double o1, double o2) {
        //returns m as requested by second bisect method
        double m1 = e;
        double yinit[] = { 0.0, o1, o2, 0.0, m1 };
        yinit = calcInteg1(yinit, 200);
        return yinit[4];
    }

    public static double[] calcInteg1(double y[], double X) {
        //modified method of previous calcODE method specifically made to
work with bisect methods
        double x = 0.01;
        double dx = 0.01;
        while (x < X) {
            x += dx;
            y = rk4(x, dx, y);

        }
        return y;
    }

    public static int sign(double x) {
        //returns int depending on sign of x value
        if (x < 0)
            return -1;
        else if (x > 0)
            return 1;
        else
            return 0;
    }
}
```

WoodWhite.java

```
import java.io.FileWriter;
import java.io.IOException;

import javax.swing.JFileChooser;
//java implementation of Wood-White Algebraic Powerlaw Function
public class WoodWhite {
    //k l n used to approximate powerlaw function
    private static float k = (float) 0.93;
    private static float l = (float) 0.96;
    private static float n = (float) 1.6;
    //eddy viscity
    private static float eV=5;
    //z at which the function will be evaluated at
    private static float z=20;
    //arrays containing inversed values of powerlaw
    private static float[] O;
    private static float[] E;
    private static float[] V1;
    private static JFileChooser chooser = new JFileChooser();
    private static float[] W1;
    //radius value at which the maximum velocity occured
    private static float Rmax = (float) 119.4774;
    //step r(radius) will iterate at
    private static float dR = (float) 0.001;
    //to be used for index correction
    private static int ic = (int) (1 / dR);

    public static void main(String[] args) {
        //initializes arrays
        E = new float[(int) (Rmax / dR)];
        V1 = new float[(int) (Rmax / dR)];
        O = new float[(int) (Rmax / dR)];
        W1 = new float[(int) (Rmax / dR)];
        //iterates through r values getting and storing value of powerlaw
function at each
        for (float R = 0; R < Rmax; R += dR) {
            int i = (int) (ic * R);
            O[i] = calcO(R);
            E[i]=calcE(R);
            V1[i]=calcV1(R);
            W1[i]=calcW1(R);
        }

        try{
            //exports in .csv
            exportData();
        }
        catch(Exception e){

        }
        System.out.println("Done");
    }
    public static void exportData() throws IOException{
        //exports data into .csv file
        chooser.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
        chooser.showSaveDialog(null);
        String filename = chooser.getSelectedFile().getPath();
        FileWriter writer = new FileWriter(filename + "\\WWData.csv");
        writer.append("R");
        writer.append(",");
        writer.append("v");
        writer.append(",");
        writer.append("u");
        writer.append(",");
        writer.append("w");
        writer.append("\n");
        for (int i = 0; i < O.length; i++) {
            float R=i*dR;
            writer.append(""+R);
            writer.append(",");
            //O value is v value
            writer.append(""+O[i]);
            writer.append(",");
            //u value is eddy viscosity * dE
            float U=eV*V1[i];
            writer.append(""+U);
            writer.append(",");
            //w value is eddy viscosity dW*z, should change with z but in
order to save computing time
            //was only calculated at one z
            float W=z*W1[i];
            writer.append(""+W);
            writer.append("\n");
            writer.flush();
        }

        writer.close();
    }
    public static float calcO(float R){
        //uses powerlaw function to approximate x value
        return (float) (Math.pow(R, k) / Math.pow(
                1 + (k / n) * (Math.pow(R, n / l) - 1), l));
    }
    public static float calcE(float R){
        //uses newton's finite differntiation to get dO
        float d=0;
        if(R==0)
            d=(calcO(R+dR)-calcO(R))/dR;
        else if(R==Rmax)
            d=(calcO(R)-calcO(R-dR))/dR;
        else
            d=(calcO(R+dR)-calcO(R-dR)/(dR*2));
        if(R==0)
            return (float) (d+(calcO(R)/0.01));
        return d+(calcO(R)/R);
    }
public static float calcV1(float R){
    //uses newtons finite differntiation to get dE
    float d=0;
    if(R==0)
        d=(calcE(R+dR)-calcE(R))/dR;
    else if(R==Rmax)
        d=(calcE(R)-calcE(R-dR))/dR;
    else
        d=(calcE(R+dR)-calcE(R-dR)/(dR*2));
    return d/calcE(R);
}
public static float calcW1(float R){
    //uses newtons finite differentiation to get dW
    float d=0;
    if(R==0)
        d=(calcV1(R+dR)-calcV1(R))/dR;
    else if(R==Rmax)
        d=(calcV1(R)-calcV1(R-dR))/dR;
    else
        d=(calcV1(R+dR)-calcV1(R-dR)/(dR*2));
    if(R==0)
        return -(d+(calcV1(R)/dR));
    return -(d+(calcV1(R)/R));
}
}
```

Database Export Method

```
public static void recordTest() {
               try {
                       // System.out.println(
Class.forName("org.hsqldb.jdbcDriver"));
                       Class.forName("com.mysql.jdbc.Driver").newInstance();
                       conn = DriverManager.getConnection(dburl, user,
password);
                       // System.out.println("Database connection
established");
               } catch (Exception ex) {
                       ex.printStackTrace();
                       JOptionPane
                                       .showMessageDialog(
                                                       null,
                                                      "Did you forget to start
the database?. The program will stop and you can restart.",
                                                      "Database Error",
JOptionPane.ERROR_MESSAGE);
                       System.exit(99);
               }
               Statement st;
               ResultSet rs = null;
               float v, u, w, angle, pr;
               // store start step in the database
               // store the timestamp and get the runID from the database
               long timeMillis = System.currentTimeMillis();

               Timestamp startTime = new Timestamp(timeMillis);
               UUID uuid = UUID.randomUUID();
               long timeNano = System.nanoTime();
               uuid.toString();
               System.out.println("Program took "
                               + (System.nanoTime() - timeNano)
                               + " to do the tostringing");
               String stringUUID = uuid.toString();
               try {
                       st = conn.createStatement();
                       st.execute("CREATE  TABLE IF NOT EXISTS
`Vortexes`.`RUN"
                                       + stringUUID
                                       + "` (`XCOR` FLOAT NULL , `YCOR` FLOAT
NULL ,"
                                       + "`ZCOR` FLOAT NULL ,`R` FLOAT NULL
,`VX` FLOAT NULL ,`VY` FLOAT NULL ,`VZ` FLOAT NULL ,"
                                       + "`V` FLOAT NULL ,`U` FLOAT NULL ,`W`
FLOAT NULL ,`PRESSURE` FLOAT NULL )");

                       for (int zCor = 0; zCor < Zmax; zCor++) {
                               for (int yCor = 0; yCor < (2 * Ymax); yCor++) {
                                       timeNano = System.nanoTime();
                                       for (int xCor = 0; xCor < (2 * Xmax);
xCor++) {
                                               v = V[xCor][yCor][zCor];
                                               w = W[xCor][yCor][zCor];
                                               u = U[xCor][yCor][zCor];
                                               pr = P[xCor][yCor][zCor];
                                               if ((xCor + Xmin) * deltaR == 0
                                                               && (yCor + Ymin)
* deltaR == 0)
                                                       angle = 0;
                                               else {
                                                       angle = (float)
Math.atan2((yCor + Ymin) * deltaR,
                                                                       (xCor +
Xmin) * deltaR);
                                                       if (angle < 0)
                                                               angle = (float)
((2 * Math.PI) + angle);
                                               }
                                               float x = (xCor + Xmin) *
deltaR;
```

```
                                                      float y = (yCor + Ymin) *
deltaR;
                                                      Float r = (float) Math.sqrt(x *
x + y * y);
                                                      Float vX = (float) ((u *
Math.cos(angle)) - (v * Math
                                                                    .sin(angle)));
                                                      if (vX.equals(Float.NaN))
                                                             System.out.println(u + "
" + v + " " + xCor + " "
                                                                                   + yCor +
" " + H[0][0]);
                                                      Float vY = (float) ((u *
Math.sin(angle)) + (v * Math
                                                                    .cos(angle)));
                                                      if (vY.equals(Float.NaN))
                                                             System.out.println(u + "
" + v + " " + xCor + " "
                                                                                   + yCor +
" " + H[0][0]);

                                                      String sql = "insert into
`Vortexes`.`run"
                                                                    + stringUUID
                                                                    + "`
(XCOR,YCOR,ZCOR,R,VX,VY,VZ,V,U,W,PRESSURE) values ("
                                                                    + x + "," + y +
"," + zCor * deltaR + "," + r
                                                                    + "," + vX + ","
+ vY + "," + w + "," + v + ","
                                                                    + u + "," + w +
"," + pr + ");";
                                                      System.out.println(sql);
                                                      st.execute(sql);


                                               }
                                               /*System.out.println("Program took "
                                                              + (System.nanoTime() -
timeNano)
                                                              + " to do the single
loop");*/

                                        }

                                 }
                                 System.out.println("Program took "
                                               + (System.currentTimeMillis() -
timeMillis)
                                               + " to do the sqling");
                                 //System.out.println(wholeStatement.toString());
                                 System.out.println("Program took "
                                               + (System.currentTimeMillis() -
timeMillis)
                                               + " to do the exporting");
                                 // st.executeQuery("CHECKPOINT");
```

```
                } catch (SQLException e) {

                        e.printStackTrace();
                }
        }
```

Python

kuoTest.py

```
import numpy
import matplotlib.pyplot as plt
#graphs values of U,V,W, F, F', and M versus R using matplotlib
#served as testbed for java application
#implements Kuo's equations
lu=[]
lv=[]
lw=[]
lx=[]
lf=[]
lf1=[]
lm=[]
lp=[]
lt=[]
b=0.01
R=7500
eV=5
r=0.1
dR=4
X=0
t=0.01
dt=b/eV
aP=100000
p=1
global o
global o1
def integ(x, state):
    F,dF,ddF,M,dM = state
    dddF=-(((F+1)*ddF)-(dF*(dF-1)))/x
    ddM = (-F * dM) / x
    return numpy.array([dF,ddF,dddF,dM,ddM])
def rk4(x, h, y, f):
    k1 = h * f(x, y)
    k2 = h * f(x + 0.5*h, y + 0.5*k1)
    k3 = h * f(x + 0.5*h, y + 0.5*k2)
    k4 = h * f(x + h, y + k3)
    return x + h, y + (k1 + 2*(k2 + k3) + k4)/6.0
def calcX(r):
    return b*r*r/(4*eV)
def calcU(r,f):
    return (-2*eV/r)*f
def calcV(r,m):
    return (R/r)*m
def calcW(z,dF):
    return b*z*dF
def calcInteg(X):
    t=0.01
    dt=b/eV
    y= numpy.array([0, -0.85, 1.5725, 0, 0.12])
    while t<X:
        t,y= rk4(t,dt,y,integ)
    return y
def calcPres(X, m,res):
    total=0
    deltaR=X/res
    for x in range (2,res+1):
        r=x*deltaR
        rn=(x-1)*deltaR
        integ=0.5*((m*m/(r*r))+(m*m/(rn*rn)))*deltaR
        total+=integ
    print total
    return total,aP+((b*p*R*R)/(8*eV))*total
def calcInteg1(y,X):
    t=0.01
    dt=0.01
    while t<X:
        t,y= rk4(t,dt,y,integ)
    return y
def F(e):
    f1=e
    f2=f1*(f1-1.)
    yinit=numpy.array([0.,f1,f2,0.0,0.12])
    yinit=calcInteg1(yinit,200)
    #print yinit[0]," ",yinit[1]," ",yinit[2]," ",yinit[3]," ",yinit[4]
    return yinit[1]
def M(e):
    m1=e
    yinit=numpy.array([0.,o,o1,0.,m1])
    yinit=calcInteg1(yinit,200)
    #print yinit[0]," ",yinit[1]," ",yinit[2]," ",yinit[3]," ",yinit[4]
    return yinit[4]
def sign( x ):
    if x < 0:
        return -1
    elif x > 0:
        return 1
    else:
        return 0

def bisect( f, minVal, maxVal):
    x  = minVal
    fmin = f(minVal)
    tol=0.003
    maxiter=100
    k = 0
    while k <= maxiter:
        x = (minVal + maxVal) / 2.0
        fx = f(x)
        k +=1
        if (abs(maxVal - minVal) / 2.0) < tol:
            break
        if sign(fmin) == sign( fx ):
            minVal, fmin = ( x, fx )
        else:
            maxVal = x
    if k > maxiter:
        print "Error: exceeded %d iterations" % maxiter
    else:
        return x
o=bisect(F,-0.9,-0.8)
o1=o*(o-1.)
o2=bisect(M,0.1,0.13)
print "F' %10f F'' %10f M' %10f"%(o,o1,o2)
g= numpy.array([0, o, o1, 0, o2])
print g
while r<500:
    X=calcX(r)
    while t<X:
        t,g= rk4(t,dt,g,integ)
    lx.append(r)
    u=calcU(r,g[0])
    w=calcW(1000,g[1])
    v=calcV(r,g[3])
    total,pS=calcPres(X,g[3],200)

    lf.append(g[0])
    lf1.append(g[1])
    lm.append(g[3])
    lu.append(u)
    lw.append(w)
    lv.append(v)
    lp.append(pS)
    lt.append(total)
    print("%10f %10f %10f %10f %10f %10f" %(r,u,v,w,pS,g[0]))
    r+=dR
plt.figure(1)
plt.plot(lx,lf,color="g",label="F vs r")
plt.plot(lx,lf1,color="r",label="F' vs r")
plt.plot(lx,lm,color="b",label="M vs r")
plt.legend()
plt.figure(2)
plt.plot(lx,lu,color="g",label="U vs r")
plt.plot(lx,lv,color="r",label="V vs r")
plt.plot(lx,lw,color="b",label="W vs r")
plt.legend()
plt.figure(3)
plt.plot(lx,lp,color="y",label="P vs r")
plt.legend()
plt.show()
```

wwTest2.py

```
import numpy
import matplotlib.pyplot as plt
#graphs values of U,V,W, and powerlaw values versus R using matplotlib
```

```
#served as testbed for java application
#implements Wood-White equations
n=1.6
k=0.93
l=0.96
ic=1/0.01
Rmax=2
dR=0.001
cor=1./dR
R=numpy.arange(0,Rmax,dR)
O=numpy.arange(0,Rmax,dR)
E=numpy.arange(0,Rmax,dR)
V=numpy.arange(0,Rmax,dR)
W=numpy.arange(0,Rmax,dR)
def calcO(r):
    return numpy.power(r,k)/numpy.power(1+(k/n)*(numpy.power(r,n/l)-1),l)
def calcE(O):
    E=numpy.arange(0,Rmax,dR)
    a=len(O)
    for x in range(0,a):
        if(x==0):
            d=(O[x+1]-O[x])/dR
            E[x]=d+(O[x]/cor)
        elif(x==a-1):
            #print x
            d=(O[x]-O[x-1])/dR
            c=float(x)/cor
            E[x]=d+(O[x]/c)
        else:
            d=(O[x+1]-O[x-1])/(dR*2)
            c=float(x)/cor
            E[x]=d+(O[x]/c)
    return E
def calcV(E):
    V=numpy.arange(0,Rmax,dR)
    dE=numpy.arange(0,Rmax,dR)
    a=len(E)
    for x in range(0,a):
        if(x==0):
            d=(E[x+1]-E[x])/dR
            V[x]=d/E[x]
            dE[x]=d
        elif(x==a-1):
            #print x
            d=(E[x]-E[x-1])/dR
            V[x]=d/E[x]
            dE[x]=d
        else:
            d=(E[x+1]-E[x-1])/(dR*2)
            V[x]=d/E[x]
            dE[x]=d
    return dE,V
def calcW(V):
    W=numpy.arange(0,Rmax,dR)
    dde=numpy.arange(0,Rmax,dR)
    a=len(V)
    for x in range(0,a):
        if(x==0):
            d=(V[x+1]-V[x])/dR
            W[x]=-(d+(V[x]/dR))
            dde[x]=d
        elif(x==a-1):
            #print x
            d=(V[x]-V[x-1])/dR
            c=float(x)/cor
            W[x]=-(d+(V[x]/c))
            dde[x]=d
        else:
            d=(V[x+1]-V[x-1])/(dR*2)
            c=float(x)/cor
            W[x]=-(d+(V[x]/c))
            dde[x]=d
    return dde,W
O=calcO(R)
E=calcE(O)
dE,V=calcV(E)
ddE,W=calcW(V)
plt.figure(1)
plt.plot(R,O,color="r",label="O vs r")
plt.legend()
plt.figure(2)
plt.plot(R,E,color="r",label="E vs r")
plt.legend()
plt.figure(3)
plt.plot(R,10*V,color="r",label="V vs r")
plt.legend()
plt.figure(4)
plt.plot(R,V*E,color="r",label="W vs r")
plt.legend()
plt.figure(5)
plt.plot(R,dE,color="r",label="dE vs r")
plt.legend()
plt.figure(6)
plt.plot(R,ddE,color="r",label="ddE vs r")
plt.legend()
plt.show()
```

Cuda

For those unfamiliar with CUDA, __device__ methods can only be called by the GPU, and __global__ methods can be called by both and are often the kernel. The kernel is called by a <<<blocksize,threadsize>>>kernalName() statement.

tests_cuda3.cu (Euler Integration)

```
//uses GPU to solve Sullivan's integrals
#include <stdio.h>
//RES is resolution*2
#define RES 100
#define E 2.71828182846
__device__ float FunctionT(float t, int resolution){
        //integrates Sullivan H function at each t value split accros
resolution
        float total = 0;
        float dR = t / resolution;
        int n=0;
        for (n = 2; n <= resolution; n++) {
                float r = (n) * dR;
                float rn = ((n - 1)) * dR;
                float e=E;
                float integ = (float) (0.5 * (((1 - pow(e, -r)) / r) + ((1 -
pow(e, -rn)) / rn)) * dR);
                total += 3. * integ;
        }
        total = total - t;
        return total;
}
__global__ void calcInteg(float *h,int resolution,int secRes){
                //grid split accross gpu x= blockId y = threadID for each
process
                //calculates the value of Sullivans integral using Euler
integration at each gridpoint
                float x=blockIdx.x;
                float y=threadIdx.x;
                int i=(RES*y)+x;
                float x1 = (x -RES) * 9;
                float y1 = (y -RES) * 9;
                float r1 = (float) sqrt((x1 * x1) + (y1 * y1));
                float X=(float) (0.04 * r1 * r1) / (2 * 50);
                float total = 0;
                float dR1 = X / resolution;
                int n;
                for (n = 2; n <= resolution; n++) {
                        float r = (n) * dR1;
                        float rn = ((n - 1)) * dR1;
                        float e=E;
                        float integ = (float) (0.5 * (pow(e,FunctionT(r,
secRes)) + pow(e,FunctionT(rn, secRes))) * dR1);
                        total += integ;
                }
                h[i]=total;
}


int main(){
        //initializes cudaEvents for timing
        cudaEvent_t start,stop;
        float elapsedTime;
        //array to contain all H values
        float h[RES][RES];
        //pointer for h for GPU use
        float *ph;
        //creates Events
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        cudaEventRecord(start,0);
        //allocates necessary space on GPU
        cudaMalloc((void**) &ph, RES*RES*sizeof(float));
        //calls the Kernel with RES blocks and RES threads to split grid over
processes
        calcInteg<<<RES,RES>>>(ph,200,200);
        //copies values of now filled array on GPU to CPU counterparts
        cudaMemcpy(h,ph,RES*RES*sizeof(float),cudaMemcpyDeviceToHost);
        //stop time recorded
        cudaEventRecord(stop);
        cudaEventSynchronize(stop);
        //elapsed time recorded and printed
        cudaEventElapsedTime(&elapsedTime,start,stop);
        printf("That took %f ms\n",elapsedTime);
        //sampling of results printed into console
        int x, y;
        for(x=0;x<RES;++x){
                for(y=0;y<RES;++y){
                        if(x==y)
                                printf("H at x=%d y=%d is %f\n",x,y,h[x][y]);
                }
        }
        }
```

KuoCuda3.cu

```
#include <stdio.h>
#include <math.h>
//amount of loops RK4 will traverse
#define N 5
//this variable is the equivalent of resolution *2
#define RES 100
//maximum radius problem will be evaluated to
#define Rmax 1000
//divides Rmax by Res/2 to get distance between each gridpoint
#define deltaR 1000/(RES/2)
//suction strength of vortex
#define B 0.01
//density of air
#define p 1
//ambient pressure of air
#define aP 10000
//eddy viscosity of vortex
#define eV 5
//circulation strength of vortex
#define R 7500
#define PI 3.14159265359
//sets up arrays containing F, F', m values of ODE and U,V,W and Pressure
values at
//each gridpoint
static float f[RES][RES];
static float dF[RES][RES];
static float M[RES][RES];
static float U[RES*RES*RES/2];
static float V[RES*RES*RES/2];
static float W[RES*RES*RES/2];
static float P[RES*RES*RES/2];
```

```
__device__ float integ(float x, float y[], int i) {
        //represents Kuo's ODEs to be solved with RK4
        //returns either F, F', F'', m, or m' depending on what was requested
by RK4 loop

        float ans;
        if (i == 0)
                ans = y[1]; /* derivative of first equation */
        if (i == 1)
                ans = y[2]; /* derivative of second equation */
        if(i==2)
                ans=-(((y[0] + 1) * y[2]) - (y[1] * (y[1] - 1))) / x;
        if(i==3)
                ans=y[4];
        if(i==4)
                ans=(-y[0]*y[4])/x;
        return ans;
}
__device__ void RK4(float x, float y[], float h) {
        //solves ODE used in Kuo's equation through RK4 methodology
        float t1[N], t2[N], t3[N], /* temporary storage arrays */
        k1[N], k2[N], k3[N], k4[N]; /* for Runge-Kutta */
        int i;

        for (i = 0; i < N; i++) {
                double m=k1[i] = h * integ(x, y, i);
                t1[i] = y[i] + 0.5 *m;
        }

        for (i = 0; i < N; i++) {
                double m=k2[i] = h * integ(x + 0.5*h, t1, i);
                t2[i] = y[i] + 0.5 * m;
        }

        for (i = 0; i < N; i++) {
                double m=k3[i] = h * integ(x + 0.5*h, t2, i);
                t3[i] = y[i] + m;
        }

        for (i = 0; i < N; i++) {
                k4[i] = h * integ(x + h, t3, i);
        }

        for (i = 0; i < N; i++) {
                y[i] += (k1[i] + 2 * k2[i] + 2 * k3[i] + k4[i]) / 6.0;
        }
}
__device__ float calcX(float r){
        //returns X value for each r
        return  (B * r * r) / (4 * eV);
}
__device__ void calcODE(float a[], float X){
        //calculates ODE values for each grid, stores newest values into array
a
        float x=0.01, dx=B/eV;
        while(x<X){
                x+=dx;
                RK4(x,a,dx);
        }
}
__device__ float calcU(float r1,float f){
                //returns U value at each gridpoint
                float r=r1;
                if (r == 0)
                        r += 0.01;
                return (-10 * eV / r) * f;
}
__device__ float calcV(float r1,float m){
                //returns V value at each gridpoint
                float r=r1;
                if (r == 0)
                        r += 0.01;
                return (R / r) * m;
}
__device__ float calcW(float z,float dF){
                //returns W value at each gridopint
                return B * z * dF;
}
__device__ float calcP(float X,float m, int res) {
                //uses Euler integration to find pressure at each gridpoint
                float total = 0;
                float dR = X / res;
                int x2=0;
                for (x2 = 2; x2 < res + 1; ++x2) {
                        float b = x2 * dR;
                        float bn = (x2 - 1) * dR;
                        float integ = (float) (0.5 * ((m * m / (b * b)) + (m *
m / (bn * bn))) * dR);
                        total += integ;
                }
                return aP + ((B * p * R * R) / (8 * eV)) * total;
        }
__global__ void calcRK4(float *f, float *dF, float *M,float *u,float *v,
float *w, float *pres){
                //initiates calculations for each gridpoint
                //grid split accross gpu x= blockId y = threadID for each
process
                //uses axisymettry to cut down calculations through use of
reflections
                float x=blockIdx.x;
                float y=threadIdx.x;
                int i=(RES*y)+x;
                int z;
                float x1 = (x -(RES/2)) *deltaR;
                float y1 = (y -(RES/2)) * deltaR;
                float r1 = (float) sqrt((x1 * x1) + (y1 * y1));
                float a[5];
                //initial values copied from Java implemented shooting code
                a[0]=0;
                a[1]=-0.857;
                a[2]=1.593;
                a[3]=0;
                a[4]=0.128;
                calcODE(a,calcX(r1));
                float f1=a[0];
                float df=a[1];
                float m=a[3];
                float U=calcU(r1,f1);
                float V=calcV(r1,m);
                float P=calcP(calcX(r1),m,500);
                f[i]=f1;
                dF[i]=df;
                M[i]=m;
                for(z=0;z<=(RES/2);++z){
                        int i2=x + y * RES + z * RES * RES;
                        u[i2]=U;
                        v[i2]=V;
                        w[i2]=calcW(z*deltaR,df);
                        pres[i2]=P;
                }
                float x2=x;
                float y2=RES-y;
                i=(RES*y2)+x2;
                f[i]=f1;
                dF[i]=df;
                M[i]=m;
                for(z=0;z<(RES/2);++z){
                        int i2=x2 + y2 * RES + z * RES * RES;
                        u[i2]=U;
                        v[i2]=V;
                        w[i2]=calcW(z*deltaR,df);
                        pres[i2]=P;
                }
                x2=RES-x;
                y2=y;
                i=(RES*y2)+x2;
                f[i]=f1;
                dF[i]=df;
                M[i]=m;
                for(z=0;z<(RES/2);++z){
                        int i2=x2 + y2 * RES + z * RES * RES;
                        u[i2]=U;
                        v[i2]=V;
                        w[i2]=calcW(z*deltaR,df);
                        pres[i2]=P;
                }
                x2=RES-x;
                y2=RES-y;
                i=(RES*y2)+x2;
                f[i]=f1;
                dF[i]=df;
                M[i]=m;
                for(z=0;z<(RES/2);++z){
                        int i2=x2 + y2 * RES + z * RES * RES;
                        u[i2]=U;
                        v[i2]=V;
                        w[i2]=calcW(z*deltaR,df);
                        pres[i2]=P;
                }
}
```

```
int main(){
        //initializes variables, cudaEvents store time each was recorded at
        cudaEvent_t start,stop;
        //elapsedTime in ms will be calced and stored in here
        float elapsedTime;
        //creates the Cuda events
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        //pointers for use in mem operations
        float *pf,*pdF,*pm,*u,*v,*w,*pres;
        //start event recorded
        cudaEventRecord(start,0);
        //allocates sufficient memory for arrays in GPU ram
        cudaMalloc((void**) &pf, RES*RES*sizeof(float));
        cudaMalloc((void**) &pdF, RES*RES*sizeof(float));
        cudaMalloc((void**) &pm, RES*RES*sizeof(float));
        cudaMalloc((void**) &u, RES*RES*(RES/2)*sizeof(float));
        cudaMalloc((void**) &v, RES*RES*(RES/2)*sizeof(float));
        cudaMalloc((void**) &w, RES*RES*(RES/2)*sizeof(float));
        cudaMalloc((void**) &pres, RES*RES*(RES/2)*sizeof(float));
        //calls the Kernel with Res/2+1 blocks and Res/2+1 threads
        //this allows the calculations to be properly split accross the GPU
        calcRK4<<<(RES/2)+1,(RES/2)+1>>>(pf,pdF,pm,u,v,w,pres);
        //checks for errors in execution
        cudaError_t error = cudaGetLastError();
        printf("%s\n",cudaGetErrorString(error));
        //copies now filled arrays on GPU into CPU counterparts
        cudaMemcpy(f,pf,RES*RES*sizeof(float),cudaMemcpyDeviceToHost);
        cudaMemcpy(dF,pdF,RES*RES*sizeof(float),cudaMemcpyDeviceToHost);
        cudaMemcpy(M,pm,RES*RES*sizeof(float),cudaMemcpyDeviceToHost);
        cudaMemcpy(U,u,RES*RES*(RES/2)*sizeof(float),cudaMemcpyDeviceToHost);
        cudaMemcpy(V,v,RES*RES*(RES/2)*sizeof(float),cudaMemcpyDeviceToHost);
        cudaMemcpy(W,w,RES*RES*(RES/2)*sizeof(float),cudaMemcpyDeviceToHost);
        cudaMemcpy(P,pres,RES*RES*(RES/2)*sizeof(float),cudaMemcpyDeviceToHost
);
        //end event recorded
        cudaEventRecord(stop);
        cudaEventSynchronize(stop);
        //ellapsed time recorded and printed
        cudaEventElapsedTime(&elapsedTime,start,stop);
        printf("That took %f ms\n",elapsedTime);
        //results randomly sampled and printed to console
        int o=0;
        for(o=0;o<RES/2;++o){
                int i=o + o * RES + o * RES * RES;
                printf("%f %f %f\n",V[i],W[i],U[i]);

}
        //results exported to .csv file
        int x, y,z;
        printf("Beggining Data Export\n");
        FILE *f= fopen("VectDataCuda.csv","w");
        if(f==NULL){
                printf("Error opening file\n");
                exit(1);
        }
```

```
                    fprintf(f,"XCor");
                    fprintf(f,",");
                    fprintf(f,"YCor");
                    fprintf(f,",");
                    fprintf(f,"ZCor");
                    fprintf(f,",");
                    fprintf(f,"r");
                    fprintf(f,",");
                    fprintf(f,"Vx");
                    fprintf(f,",");
                    fprintf(f,"Vy");
                    fprintf(f,",");
                    fprintf(f,"Vz");
                    fprintf(f,",");
                    fprintf(f,"v");
                    fprintf(f,",");
                    fprintf(f,"u");
                    fprintf(f,",");
                    fprintf(f,"w");
                    fprintf(f,",");
                    fprintf(f,"Pressure");
                    fprintf(f,"\n");
                    for (z = 0; z < RES/2; ++z) {
                            for (y = 0; y < RES; ++y) {
                                    for (x = 0; x < RES; ++x) {
                                            int i=x + y * RES + z * RES * RES;
                                            float v = V[i];
                                            float w = W[i];
                                            float u = U[i];
                                            float p1=P[i];
                                            float xCor=(x -(RES/2)) * deltaR;
                                            float yCor=(y -(RES/2)) * deltaR;
                                            float zCor=z * deltaR;
                                            float r=sqrt((xCor*xCor)+(yCor*yCor));
                                            float angle;
                                            if (xCor == 0 && yCor == 0)
                                                    angle = 0;
                                            else {
                                                    angle = (float)
atan2(yCor,xCor);

                                                    if (angle < 0)
                                                            angle = (float) ((2 *
PI) + angle);
                                            }
                                            float Vx = (float) ((u * cos(angle)) -
(v * sin(angle)));
                                            float Vy= (float) ((u * sin(angle)) +
(v *cos(angle)));
                                            fprintf(f,
"%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f\n", xCor,yCor,zCor,r,Vx,Vy,w,v,u,w,p1);
                                                    }
                                            }
                                    }
                                    fclose(f);
}
```

RK4cuda.cu

```
#include <stdio.h>
#define N 5
//RES is resolution *2
#define RES 200
//finds space between gridopoints with Rmax of 1000
#define deltaR 1000/(RES/2)
//defines suction strength of vortex
#define B 0.01
//denisty of air
#define p 1
//ambient pressure of air
#define aP 10000
//eddy viscosity of vortex
#define eV 5
//circulation strenght of vortex
#define R 7500
#define PI 3.14159265359
__device__ float integ(float x, float y[], int i) {
        //gets appropriated calculated ODE value based on query from RK4 loop
        float ans;
        if (i == 0)
                ans = y[1]; /* derivative of first equation */
        if (i == 1)
                ans = y[2]; /* derivative of second equation */
        if(i==2)
                ans=-(((y[0] + 1) * y[2]) - (y[1] * (y[1] - 1))) / x;
        if(i==3)
                ans=y[4];
        if(i==4)
                ans=(-y[0]*y[4])/x;
        return ans;
}
__device__ void RK4(float x, float y[], float h) {
        //uses RK4 methodology to solve ODE
        float t1[N], t2[N], t3[N], /* temporary storage arrays */
        k1[N], k2[N], k3[N], k4[N]; /* for Runge-Kutta */
        int i;

        for (i = 0; i < N; i++) {
                double m=k1[i] = h * integ(x, y, i);
                t1[i] = y[i] + 0.5 *m;
        }

        for (i = 0; i < N; i++) {
                double m=k2[i] = h * integ(x + 0.5*h, t1, i);
                t2[i] = y[i] + 0.5 * m;
        }

        for (i = 0; i < N; i++) {
                double m=k3[i] = h * integ(x + 0.5*h, t2, i);
                t3[i] = y[i] + m;
        }

        for (i = 0; i < N; i++) {
                k4[i] = h * integ(x + h, t3, i);
        }

        for (i = 0; i < N; i++) {
                y[i] += (k1[i] + 2 * k2[i] + 2 * k3[i] + k4[i]) / 6.0;
        }
}
__device__ float calcX(float r){
        //returns X value for each r
        return  (B * r * r) / (4 * eV);
}
__device__ void calcODE(float a[], float X){
        //runs the loop for RK4 solving of ODE
        float x=0.01, dx=B/eV;
        while(x<X){
                x+=dx;
                RK4(x,a,dx);
        }
}
__global__ void calcRK4(float *f, float *dF, float *M){
                //grid split accross gpu x= blockId y = threadID for each
process
                float x=blockIdx.x;
                float y=threadIdx.x;
                //array index calculated
                int i=(RES*y)+x;
                //real x values calculated
                float x1 = (x -(RES/2)) *deltaR;
                float y1 = (y -(RES/2)) * deltaR;
                float r1 = (float) sqrt((x1 * x1) + (y1 * y1));
                float a[5];
                //initial values copied from Java implemented shooting code
                a[0]=0;
                a[1]=-0.857;
                a[2]=1.593;
                a[3]=0;
                a[4]=0.128;
                //calculated ODE values for each gridpoint
                calcODE(a,calcX(r1));
                //stores into arrays and takes advantage of axi-symmetry to
shorten calcuations
                f[i]=a[0];
                dF[i]=a[1];
                M[i]=a[3];
                float x2=x;
                float y2=RES-y;
                i=(RES*y2)+x2;
                f[i]=a[0];
                dF[i]=a[1];
                M[i]=a[3];
                x2=RES-x;
                y2=y;
                i=(RES*y2)+x2;
                f[i]=a[0];
                dF[i]=a[1];
                M[i]=a[3];
                x2=RES-x;
                y2=RES-y;
                i=(RES*y2)+x2;
                f[i]=a[0];
                dF[i]=a[1];
                M[i]=a[3];
}


int main(){
        //initializes variables to measure time in ms
        cudaEvent_t start,stop;
        float elapsedTime;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        //creates arrays contained ODE values at each point
        float f[RES][RES],dF[RES][RES],M[RES][RES];
        //pointers for use in GPU
        float *pf,*pdF,*pm;
        //records start event
        cudaEventRecord(start,0);
        //allocates space in GPU
        cudaMalloc((void**) &pf, RES*RES*sizeof(float));
        cudaMalloc((void**) &pdF, RES*RES*sizeof(float));
        cudaMalloc((void**) &pm, RES*RES*sizeof(float));
        //runs kernel with RES/2+1 blocks and RES/2+1 threads to calculate
        //ODE values at each gridpoint in parallel
        calcRK4<<<RES/2+1,RES/2+1>>>(pf,pdF,pm);
        //copies filled arrays in GPU to CPU counterparts
        cudaMemcpy(f,pf,RES*RES*sizeof(float),cudaMemcpyDeviceToHost);
        cudaMemcpy(dF,pdF,RES*RES*sizeof(float),cudaMemcpyDeviceToHost);
        cudaMemcpy(M,pm,RES*RES*sizeof(float),cudaMemcpyDeviceToHost);
        //records stop event and calculates and prints elapsed time
        cudaEventRecord(stop);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&elapsedTime,start,stop);
        printf("That took %f\n",elapsedTime);
        //prints random sampling of resuls to console
        int x, y;
        for(x=0;x<RES;++x){
                for(y=0;y<RES;++y){
                                if(x==y)
                                        printf("R at x=%d y=%d F is:
%f\n",x,y,f[x][y]);
                }
        }
}
```