Assignment 04: Vector Field Visualization

Scott Merkley

March/05/2023

Department of Computer Science, University of Utah, Salt Lake City, 84112, UT, USA

Introduction

This report covers the results and figures from the questions asked in Assignment 04 for CS 5635 at the University of Utah. Assignment 04 overviews vector field visualization of 3D data using Paraview with the different types of seeding and methods of visualizing. The different methods shown in this report are stream lines, cone glyphs, tubes, and colormaps. Lastly, this report also shows the use of the two integration methods Euler's and Runge-Kutta, explaining the pros and cons for each.

Question 1:

First, we were given a data set containing the air flow above a heated disk. We were asked to apply a glyph filter to better visualize the vector field of the temperature. After applying the glyph filter, the scale mode was changed to vector magnitude, and the glyph mode changed to all points. Doing this allowed for the following visualization.

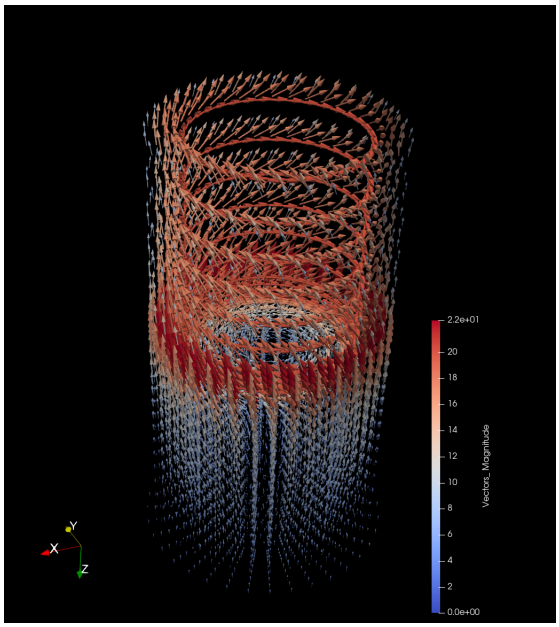

Figure 1: Visualization of the Vector Field by Temperature Above the Heated Disk

We were then asked to extract streamlines using the point source seed type and use tubular surfaces to get a better look at the flow of the air above the heated disk. The following visualization was achieved.

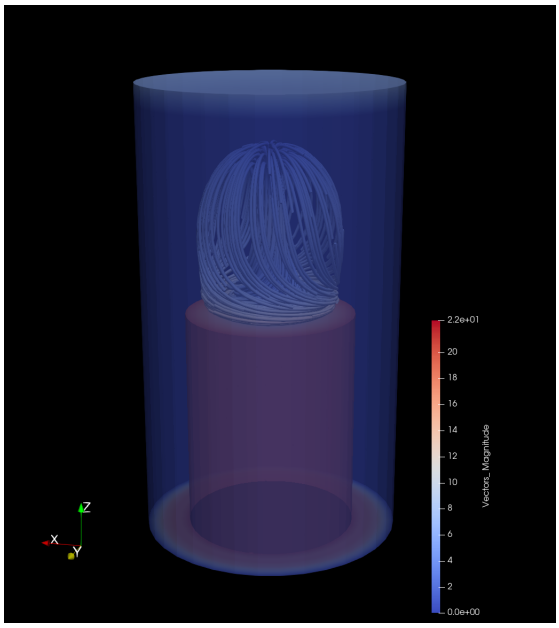

Figure 2: Visualization of the Vector Field by Temperature Above the Heated Disk using Streamlines

We were then left to find a way to properly apply filters to look like the figure in the assignment. A glyph filter was applied to the stream tubes to get the arrows facing in the direction that the airflow is going with a length of the magnitude of the airflow vector. The glyph helps you see that the air is flowing upward from the center of the disk and then reaches a high point, starts to fall outward toward the disk, then spinning counter clockwise as it reaches the disk. This can be seen in the following figure.

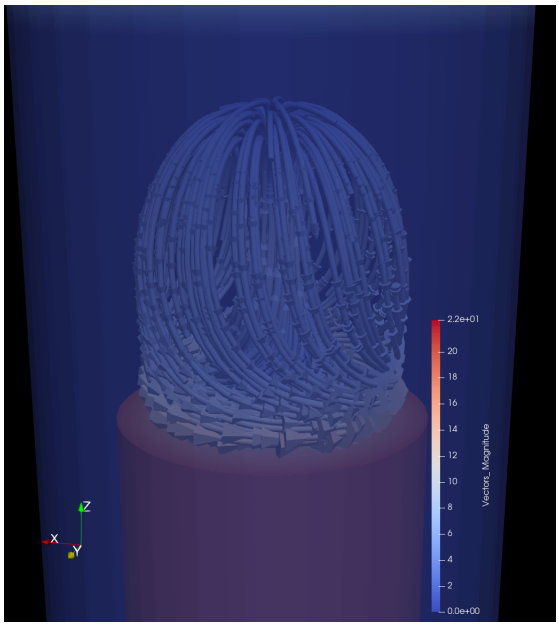

Figure 3: Visualization of the Vector Field by Temperature Above the Heated Disk using Streamlines and Glyphs

Question 2:

For question two we were asked to create a visualization of hurricane Katrina to better understand the wind flow within the storm. After opening the data in Paraview, a stream line filter was used to see the differing wind flow, cone glyphs were then added to the visualization giving a better sense for the direction of the wind. Seeding the data with a streamline was used first and then with a cloud and can be seen in the figures below.

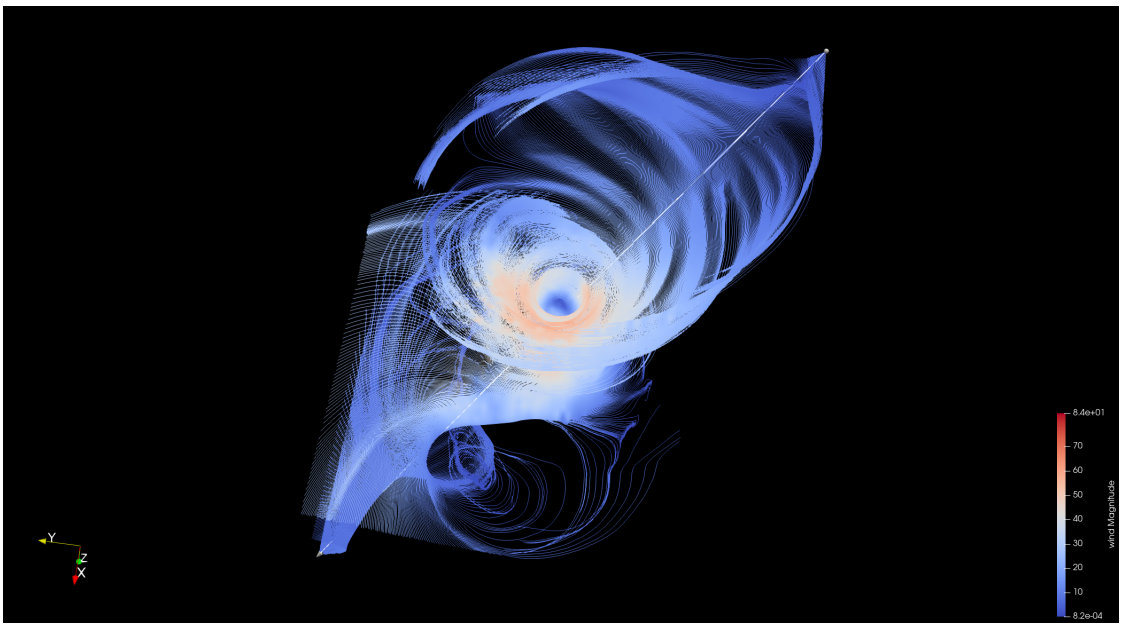

Figure 4: Visualization of the Wind Flow in Hurricane Katrina using Stream Lines and Line Seeding

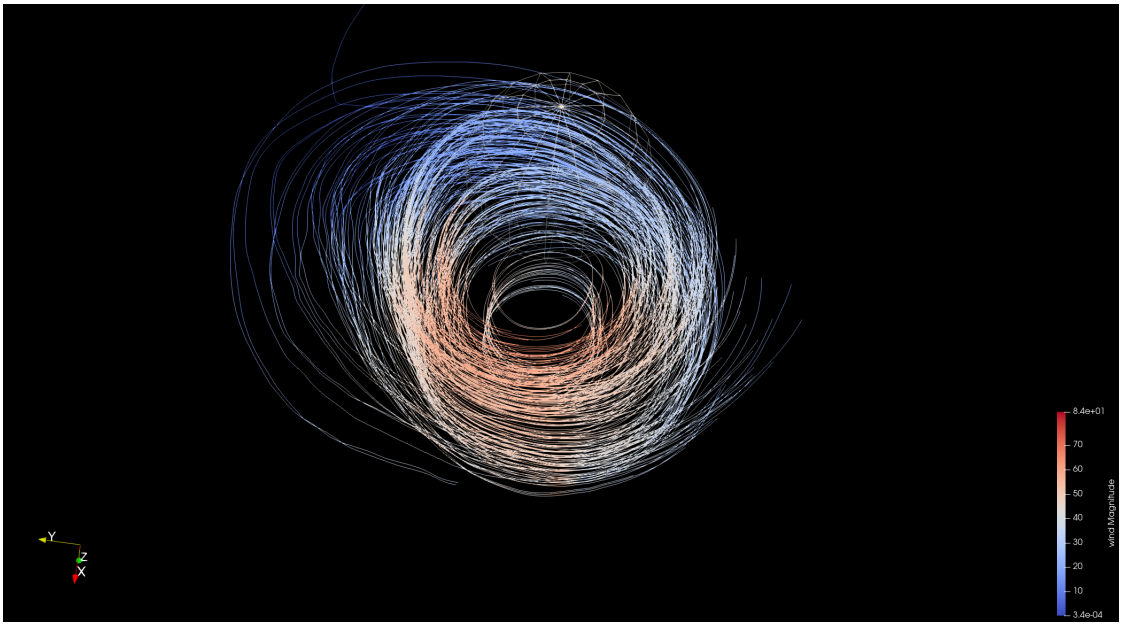

Figure 5: Visualization of the Wind Flow in Hurricane Katrina using Cloud Seeding

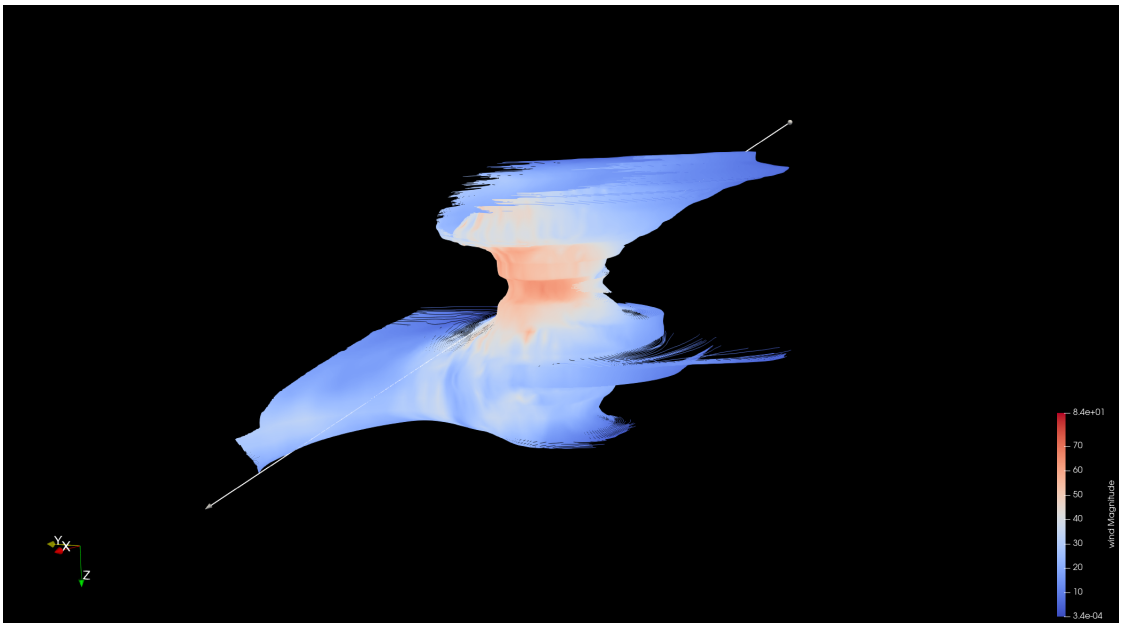

Figure 6: Visualization of the Wind Flow in Hurricane Katrina using Stream Lines from Side View

We were then asked to use a tube filter on with the stream line visualization to get a better understanding for the speed of the wind in the hurricane. After adding the tube filter, cone glyphs were added to show the direction and magnitude of the wind as it flows throughout the storm. This visualization shows that the wind speed is greatly increased near the eye of the storm and slows down near the bottom and top edges. This can best be seen in this visualization.

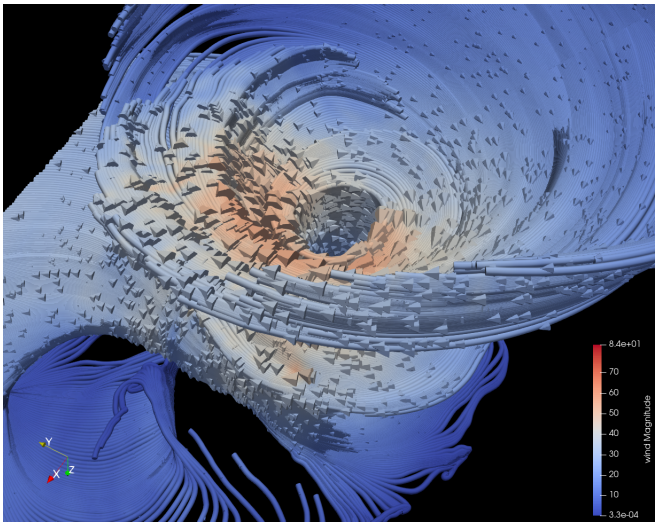

Figure 7: Visualization of the Wind Flow in Hurricane Katrina using Tubes and Cone Glyphs

Cloud seeding was also used to get a better view of only Katrina's eye.

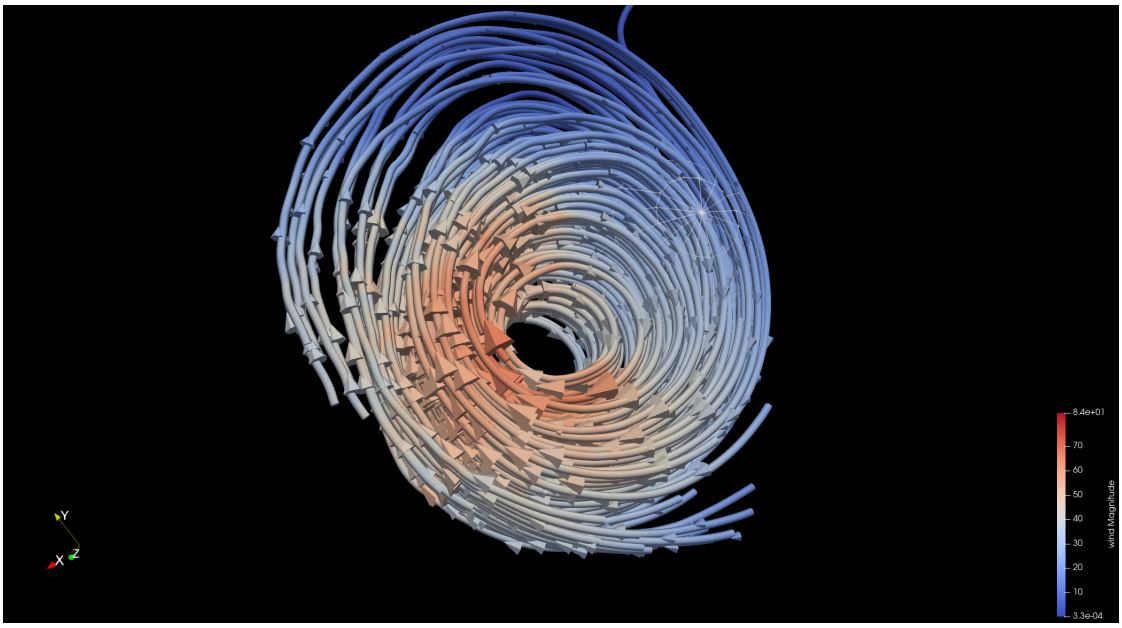

Figure 8: Visualization of the Wind Flow in Hurricane Katrina's Eye using Cloud Seeding

A colormap was used for the visualization because it quickly shows the user where the wind speed is the highest and lowest. Color is one of the best ways of showing a difference in data because a user can quickly see where areas of high and low activity are based off the legend. In this visualization the color shows that at the bottom and top sections of the hurricane the wind speed is slower, as shown by the blue color. In the middle section of the hurricane the wind speed can be seen to speed up as seen by the red color as well as the larger cone glyphs.

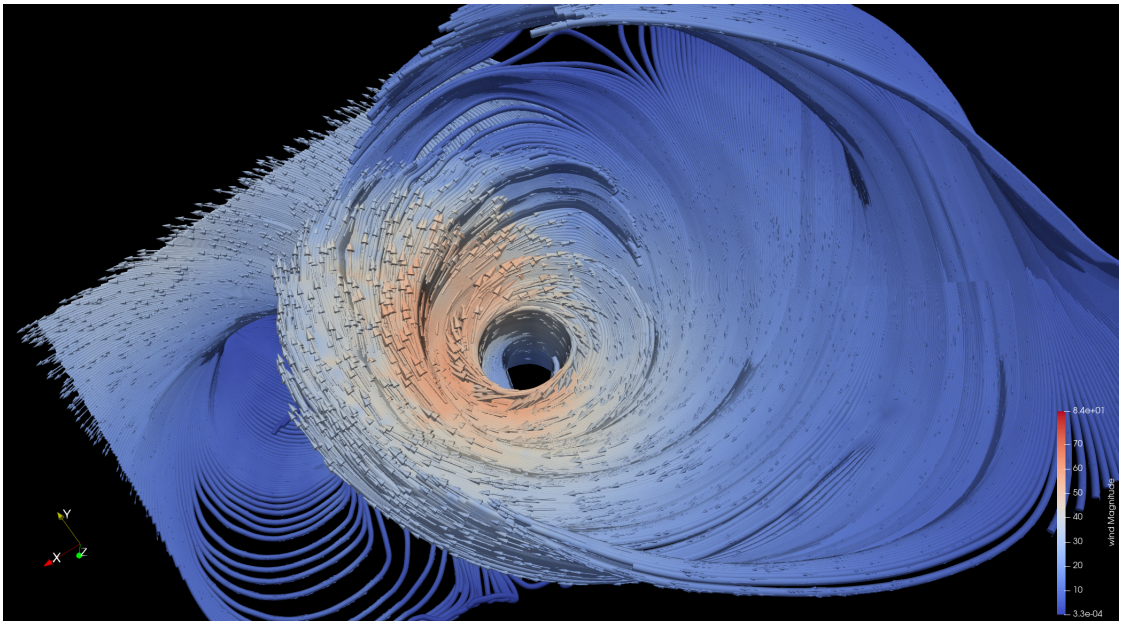

Figure 9: Visualization of the Wind Flow in Hurricane Katrina using Stream Lines, Tubes, and Cone Glyphs

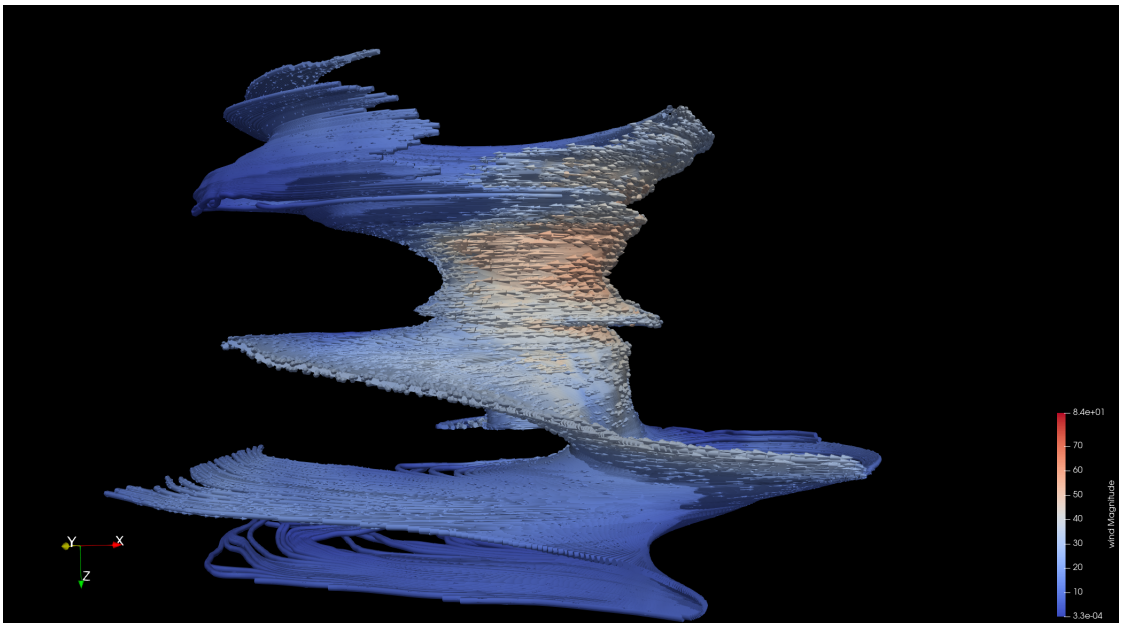

Figure 10: Visualization of the Wind Flow in Hurricane Katrina using Stream Lines, Tubes, and Cone Glyphs

Overall, this form of visualization is very effective because it gives the user multiple different ways of viewing the wind in this hurricane but also any vector in a vector field. Seeing the size of a cone change as well as color allows for more intuition to be formed around the data.

Question 3:

For question three we were given a data set of air flow around a car. We were asked to design a transfer function to show the relatively high and low magnitudes of air flow around the car. At first, a 1D transfer function was used but was not able to visualize much as there was too much changing and the specific regions could not be found. It was then switched to a 2D transfer function quickly allowing for a transfer function that resembled the one shown in the assignment description to be created. This visualizations can be seen in the figures below.

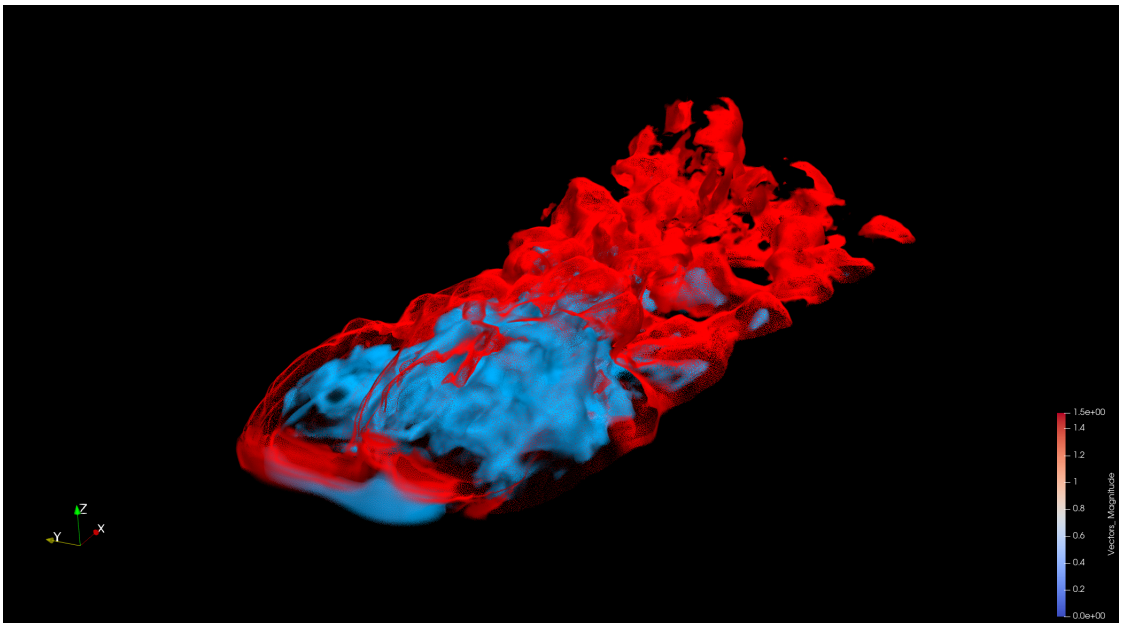

Figure 11: Visualization of Air Flow around a Car

We were then asked to apply the stream tracer filter to the data set and use a line to seed the lines and use the ribbon filter.

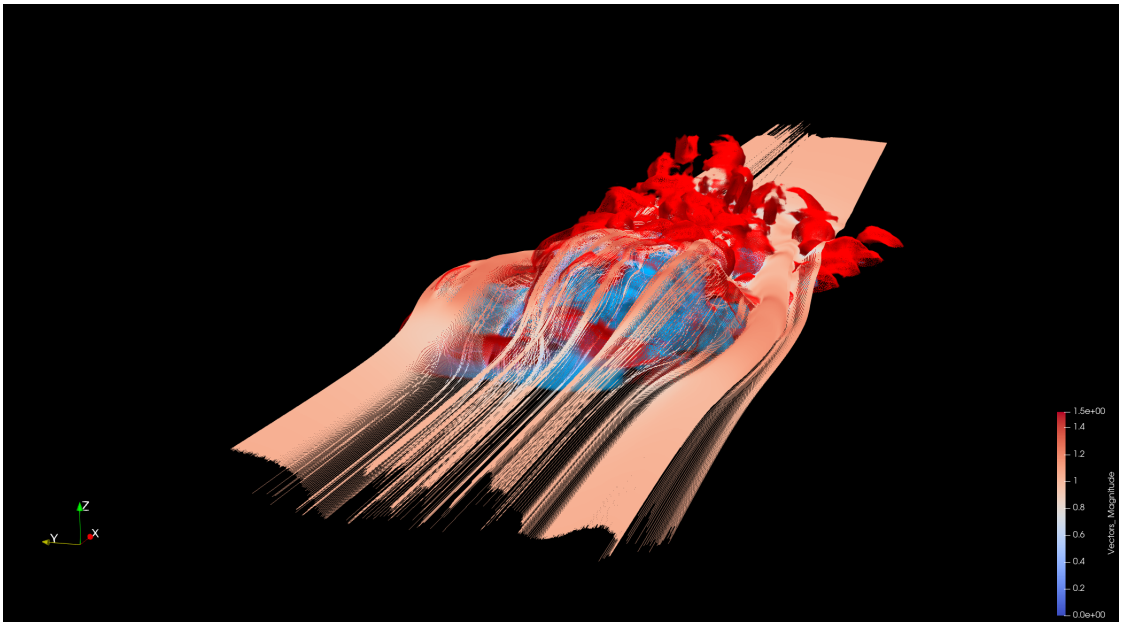

Figure 12: Visualization of Air Flow around a Car with Stream Lines

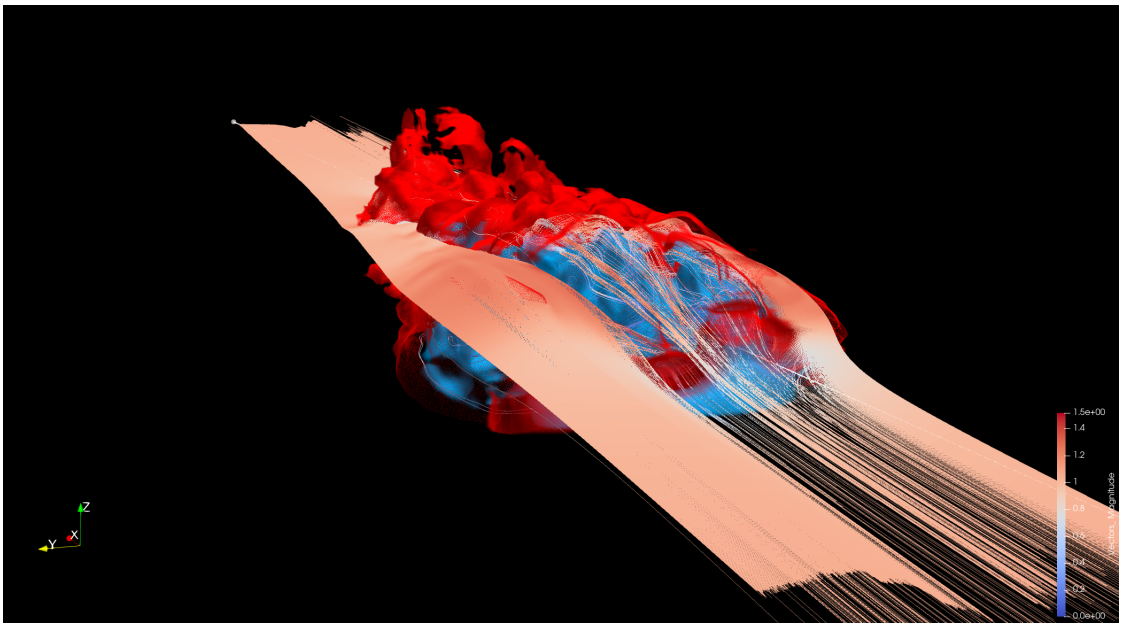

Figure 13: Visualization of Air Flow around a Car with Stream Lines around Tires

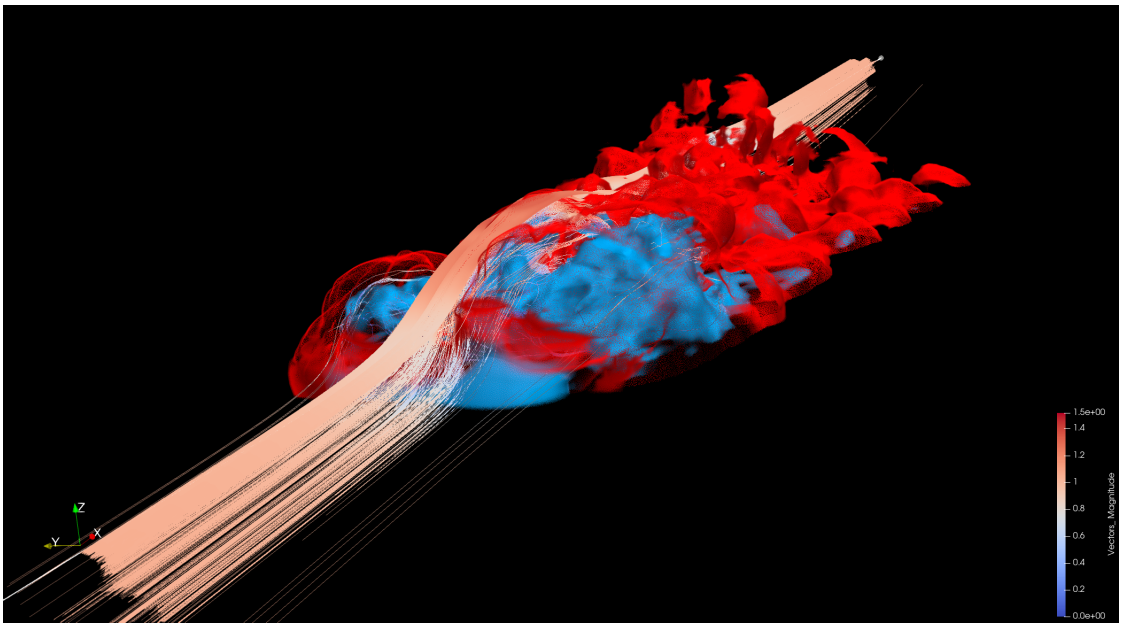

Figure 14: Visualization of Air Flow around a Car with Stream Lines Over Center

Given the flow of air around the car it is thought that the car is a Formula 1 car. The tires look to be very big and on the side of the car. The middle stream going over the top of the car is still very low to the ground and looks as though there is a cockpit in the car. This leads me to think that the car is a Formula 1 racing car.

Question 4:

For question four we were given a data set of wind as a 2D vector field of size 20x20. We were asked to generate 15 randomly seeded points within the range [0, 19]. We were also asked to use bilinear interpolation of the wind vectors on the generated points. This bilinear interpolation was done using the vector data and finding where the randomly point was generated, using its location to find the four wind vectors around the point and then apply them using weighted mean to give the point a different value depending on how close it was to one of the four vectors.

Doing this and then using multiple time steps resulted in stream lines throughout the vector field that were caused by the 'wind' in the field. The method used to move the points was Euler's method and can be seen below.

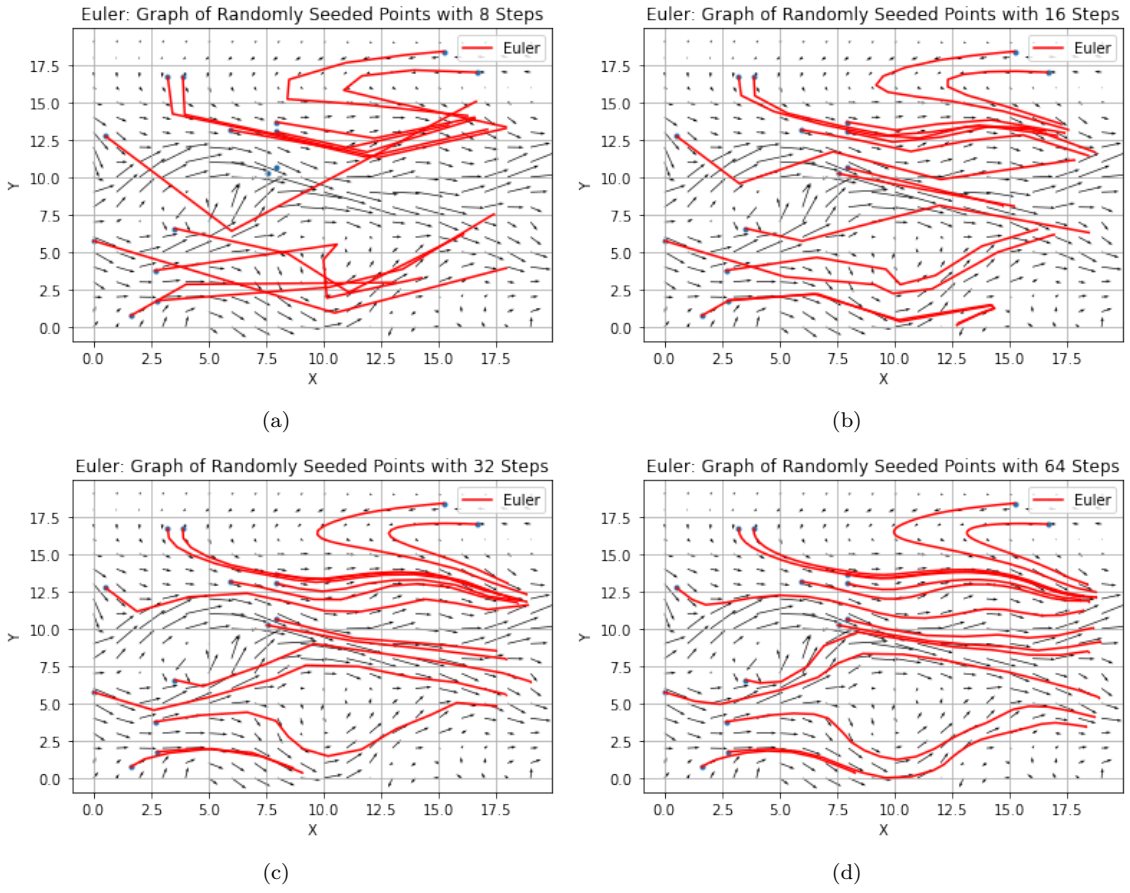

Figure 15: Visualization of Wind Vector Field using Euler's Method

These visualizations show differing time steps of 8, 16, 32, and 64. A scaling factor had to be set for each run and started at 0.3 for 8 and went down to 0.15, 0.075, and 0.0375 for the other steps respectively. At the lower time steps, the movement of each point is very rigid and far. Euler's method is jagged and consistently oversteps, missing the surrounding vectors. As the time steps get larger and each movement gets smaller, the precision and accuracy of each movement gets better and strays from the actual path less. 64 steps is seen to have the most accurate movement throughout the vector field. This looks great but could be done better at smaller time steps using other methods. The error introduced using Euler's method is great but the calculation speed and ease is far better when using Euler's method. The four figures show divergence as the number of time steps increases. Divergence is the tendency to slowly move away from a complete accuracy. Euler's method diverges in the way that if you were to have some vector field that moved

in a circle, Euler's method has the tendency to spiral out as it is slightly off from where it began due to an introduction of error. There are other methods, such as Runge-Kutta, that account for this error and do not diverge.

```
import numpy as np
import matplotlib.pyplot as plt

# Set the random seed
np.random.seed(1)

# Get data
vecs = np.reshape(np.fromfile("wind_vectors.raw"), (20,20,2))
vecs_flat = np.reshape(vecs, (400,2)) # useful for plotting
vecs = vecs.transpose(1,0,2) # needed otherwise vectors don't match with plot

# X and Y coordinates of points where each vector is in space
xx, yy = np.meshgrid(np.arange(0, 20),np.arange(0, 20))

# Generate 15 seed points within the range [ 0, 19 ]
seededPoints = np.random.rand( 15, 2 ) * 19

# Create a streamline from each point using bilinear interpolation
dt = 0.3
numberOfSteps = 8


# bilinearInterpolation calculates the interpolation of the four points around a
# given point returning the weighted mean of the point.
#
# @param point - point to calculate bilinear interpolation around
# @return      -
def bilinearInterpolation( point ):
    x = point[ 0 ]
    y = point[ 1 ]
    x1 = int( np.floor( x ) )
    x2 = x1 + 1
    y1 = int( np.floor( y ) )
    y2 = y1 + 1

    # Equations to calculate the weighted mean were found on:
    # https://en.wikipedia.org/wiki/Bilinear_interpolation
    w11 = ( ( x2 - x ) * ( y2 - y ) ) / ( ( x2 - x1 ) * ( y2 - y1 ) )
    w12 = ( ( x2 - x ) * ( y - y1 ) ) / ( ( x2 - x1 ) * ( y2 - y1 ) )
    w21 = ( ( x - x1 ) * ( y2 - y ) ) / ( ( x2 - x1 ) * ( y2 - y1 ) )
    w22 = ( ( x - x1 ) * ( y - y1 ) ) / ( ( x2 - x1 ) * ( y2 - y1 ) )

    weightedMeanOfPoint =  w11 * vecs[ x1 ][ y1 ] + w12 * vecs[ x1 ][ y2 ] + w21 * vecs[
                                                    x2 ][ y1 ] + w22 * vecs[ x2 ][ y2 ]
    return weightedMeanOfPoint



def computeTimeSteps( method, steps ):

    scaleFactor = 1
    if steps == 8:
        scaleFactor = 0.3
    if steps == 16:
        scaleFactor = 0.15
    if steps == 32:
        scaleFactor = 0.075
    if steps == 64:
        scaleFactor = 0.0375

    # Makes a 3D array for the time steps
    timeSteps = np.zeros( ( len( seededPoints ) , steps, 2 ) )

    for i in range( 0, len(seededPoints) ):
        for j in range( 0, steps ):
            if j == 0:
```

```
            timeSteps[ i ][ j ] = seededPoints[ i ]
        else:
            if( method == "Euler" ):
                timeSteps[ i ][ j ] = timeSteps[ i ][ j - 1 ] + scaleFactor *
                                                                bilinearInterpolation(
                                                                 timeSteps[ i ][ j - 1
                                                                 ] )

                # Fix the Euler Step if it goes out of bounds
                if timeSteps[ i ][ j ][ 0 ] > 19 or timeSteps[ i ][ j ][ 0 ] < 0 or
                                                                timeSteps[ i ][ j ][ 1
                                                                 ] > 19 or timeSteps[
                                                                i ][ j ][ 1 ] < 0:
                    timeSteps[ i ][ j ] = timeSteps[ i ][ j - 1 ]

            elif( method == "RK4" ):
                K1 = bilinearInterpolation( timeSteps[ i ][ j - 1 ] )
                if timeSteps[ i ][ j - 1 ][ 0 ] + scaleFactor * K1[0] / 2 < 19 and
                                                                timeSteps[ i ][ j - 1
                                                                ][ 0 ] + scaleFactor *
                                                                 K1[0] / 2 > 0 and
                                                                timeSteps[ i ][ j - 1
                                                                ][ 1 ] + scaleFactor *
                                                                 K1[1] / 2 < 19 and
                                                                timeSteps[ i ][ j - 1
                                                                ][ 1 ] + scaleFactor *
                                                                 K1[1] / 2 > 0:
                    K2 = bilinearInterpolation( timeSteps[ i ][ j - 1 ] +
                                                                    scaleFactor * K1 /
                                                                     2 )
                    if timeSteps[ i ][ j - 1 ][ 0 ] + scaleFactor * K2[0] / 2 < 19
                                                                and timeSteps[ i ]
                                                                [ j - 1 ][ 0 ] +
                                                                scaleFactor * K2[0
                                                                ] / 2 > 0 and
                                                                timeSteps[ i ][ j
                                                                - 1 ][ 1 ] +
                                                                scaleFactor * K2[1
                                                                ] / 2 < 19 and
                                                                timeSteps[ i ][ j
                                                                - 1 ][ 1 ] +
                                                                scaleFactor * K2[1
                                                                ] / 2 > 0:
                        K3 = bilinearInterpolation( timeSteps[ i ][ j - 1 ] +
                                                                        scaleFactor *
                                                                        K2 / 2 )
                        if timeSteps[ i ][ j - 1 ][ 0 ] + scaleFactor * K3[0] < 19
                                                                        and timeSteps[
                                                                         i ][ j - 1 ][
                                                                         0 ] +
                                                                        scaleFactor *
                                                                        K3[0] > 0 and
                                                                        timeSteps[ i ]
                                                                        [ j - 1 ][ 1 ]
                                                                         + scaleFactor
                                                                         * K3[1] < 19
                                                                        and timeSteps[
                                                                         i ][ j - 1 ][
                                                                         1 ] +
                                                                        scaleFactor *
                                                                        K3[1] > 0:
                            K4 = bilinearInterpolation( timeSteps[ i ][ j - 1 ] +
                                                                            scaleFactor
                                                                             * K3 )
                            timeSteps[ i ][ j ] = timeSteps[ i ][ j - 1 ] + 1 / 6 *
                                                                            ( K1 + 2 *
                                                                             K2 + 2 *
                                                                            K3 + K4 )
                                                                            *
                                                                            scaleFactor
```

```
                if timeSteps[ i ][ j ][ 0 ] > 19 or timeSteps[ i ][ j ][ 0 ] < 0 or
                                                        timeSteps[ i ][ j ][ 1 ] >
                                                         19 or timeSteps[ i ][ j ]
                                                        [ 1 ] < 0 or ( timeSteps[
                                                        i ][ j ][ 0 ] == 0 and
                                                        timeSteps[ i ][ j ][ 1 ] =
                                                        = 0 ):
                    timeSteps[ i ][ j ] = timeSteps[ i ][ j - 1 ]

    return timeSteps


def computeAndPlot( steps ):

    # Compute all the arrays to plot
    eulerTimeSteps = computeTimeSteps( "Euler", steps )

    # Plot the Data
    for i in range( len( eulerTimeSteps ) ):
        plt.plot(eulerTimeSteps[i][:, 0], eulerTimeSteps[i][:, 1], 'r.-', ms = 0.05,
                                                    label = "Euler" if i == 0 else '')


    # Show an image of your plot
    plt.title( f"Euler: Graph of Randomly Seeded Points with { steps } Steps" )
    plt.plot(xx, yy, marker='.', color='b', linestyle='none', ms = 0.01)
    plt.quiver(xx, yy, vecs_flat[:,0], vecs_flat[:,1], width=0.002)
    plt.scatter(seededPoints[:, 0], seededPoints[:, 1], marker = '.')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.legend()
    plt.grid()
    plt.show()


computeAndPlot( 8 )
computeAndPlot( 16 )
computeAndPlot( 32 )
computeAndPlot( 64 )
```

Extra Credit: Runge-Kutta Method

For the extra credit section of this homework we were asked to implement the Runge-Kutta Method (RK4) as the method of integration and recreate the 4 figures from question 4. The RK4 method takes averages with each step, making it far more accurate at lower time steps. This averaging can be seen in the 8 and 16 step figures as the RK4 method gets the general flow and direction correct for the integration of the points where the Euler points start to go everywhere on the plot. As the number of steps gets larger and the step size gets smaller the RK4 method outshines Euler's method by a lot which can be seen when the small spirals begin to be made. Euler's method takes such big steps that the spirals are not made and completely skipped, whereas RK4 begins making spirals at 16 steps and continues to get better as the number of steps increases.

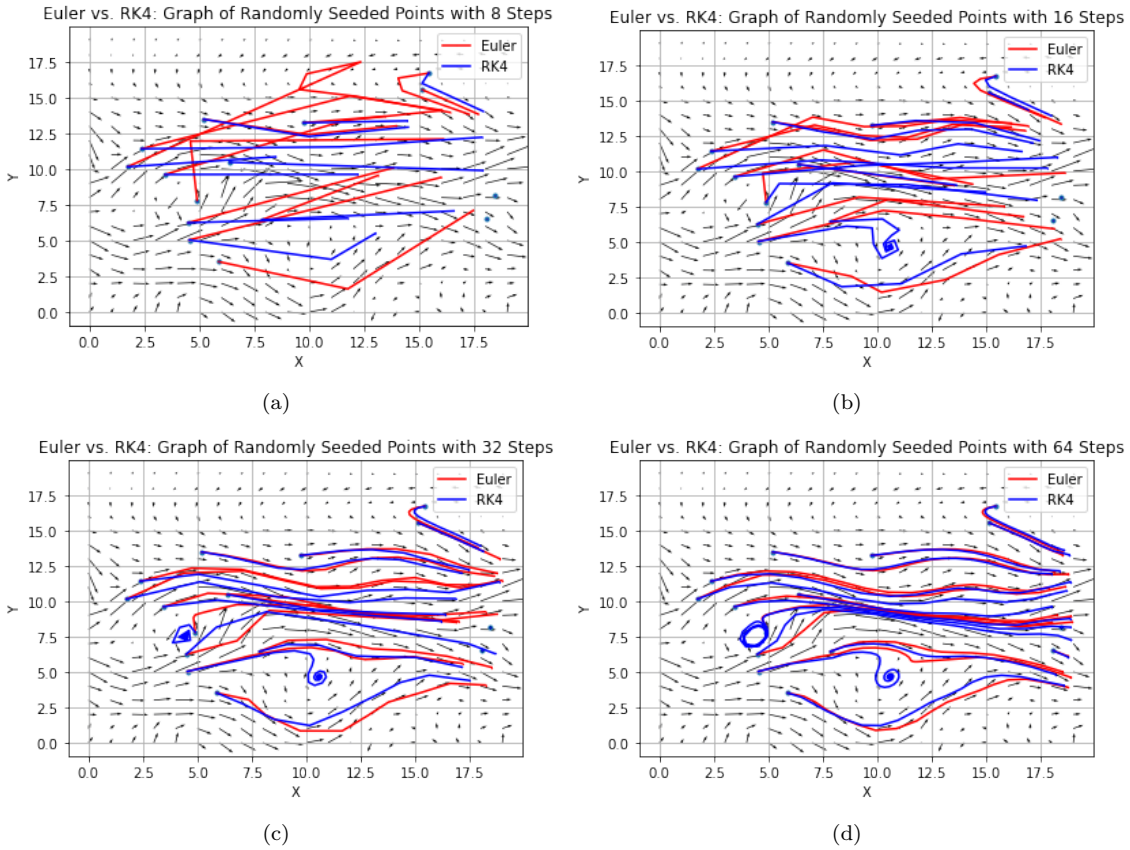

Figure 16: Visualization of Wind Vector Field using Euler's and RK4 Method

The problem with RK4 is that it takes practice to implement it well. It is far easier to use Euler's method and crank up the number of steps as it is usually well within the range of what a computer can do quickly. This allows the user to get most of the desired visualizations out of the data set but not spend a lot of time coding. With RK4, it took me much longer to figure out exactly what my code should look like and what values I should use to do calculations. I also ran into a problem where I had to do a check whether the values I was putting into next step were too large to calculate, making the code less readable. The upside is that you get more accurate visualizations when using smaller time steps. In this case you get most of the data by using 16 time steps and only getting better with 32 and 64. If each calculation is more time intensive to do then having less time steps is better and RK4 will be more useful. If the code is difficult to understand in the first place, using Euler's method is always a good place to start and then implementing RK4 can be done afterwards if needed.

```
import numpy as np
import matplotlib.pyplot as plt

# Set the random seed
np.random.seed(12345)

# Get data
vecs = np.reshape(np.fromfile("wind_vectors.raw"), (20,20,2))
vecs_flat = np.reshape(vecs, (400,2)) # useful for plotting
vecs = vecs.transpose(1,0,2) # needed otherwise vectors don't match with plot

# X and Y coordinates of points where each vector is in space
xx, yy = np.meshgrid(np.arange(0, 20),np.arange(0, 20))

# Generate 15 seed points within the range [ 0, 19 ]
seededPoints = np.random.rand( 15, 2 ) * 19
```

```
# Create a streamline from each point using bilinear interpolation
dt = 0.3
numberOfSteps = 8


# bilinearInterpolation calculates the interpolation of the four points around a
# given point returning the weighted mean of the point.
#
# @param point - point to calculate bilinear interpolation around
# @return      -
def bilinearInterpolation( point ):
    x = point[ 0 ]
    y = point[ 1 ]
    x1 = int( np.floor( x ) )
    x2 = x1 + 1
    y1 = int( np.floor( y ) )
    y2 = y1 + 1

    # Equations to calculate the weighted mean were found on:
    # https://en.wikipedia.org/wiki/Bilinear_interpolation
    w11 = ( ( x2 - x ) * ( y2 - y ) ) / ( ( x2 - x1 ) * ( y2 - y1 ) )
    w12 = ( ( x2 - x ) * ( y - y1 ) ) / ( ( x2 - x1 ) * ( y2 - y1 ) )
    w21 = ( ( x - x1 ) * ( y2 - y ) ) / ( ( x2 - x1 ) * ( y2 - y1 ) )
    w22 = ( ( x - x1 ) * ( y - y1 ) ) / ( ( x2 - x1 ) * ( y2 - y1 ) )

    weightedMeanOfPoint =  w11 * vecs[ x1 ][ y1 ] + w12 * vecs[ x1 ][ y2 ] + w21 * vecs[
                                                x2 ][ y1 ] + w22 * vecs[ x2 ][ y2 ]
    return weightedMeanOfPoint



def computeTimeSteps( method, steps ):

    scaleFactor = 1
    if steps == 8:
        scaleFactor = 0.3
    if steps == 16:
        scaleFactor = 0.15
    if steps == 32:
        scaleFactor = 0.075
    if steps == 64:
        scaleFactor = 0.0375

    # Makes a 3D array for the time steps
    timeSteps = np.zeros( ( len( seededPoints ) , steps, 2 ) )

    for i in range( 0, len(seededPoints) ):
        for j in range( 0, steps ):
            if j == 0:
                timeSteps[ i ][ j ] = seededPoints[ i ]
            else:
                if( method == "Euler" ):
                    timeSteps[ i ][ j ] = timeSteps[ i ][ j - 1 ] + scaleFactor *
                                                                    bilinearInterpolation(
                                                                     timeSteps[ i ][ j - 1
                                                                     ] )

                    # Fix the Euler Step if it goes out of bounds
                    if timeSteps[ i ][ j ][ 0 ] > 19 or timeSteps[ i ][ j ][ 0 ] < 0 or
                                                                    timeSteps[ i ][ j ][ 1
                                                                     ] > 19 or timeSteps[
                                                                    i ][ j ][ 1 ] < 0:
                        timeSteps[ i ][ j ] = timeSteps[ i ][ j - 1 ]

                elif( method == "RK4" ):
                    K1 = bilinearInterpolation( timeSteps[ i ][ j - 1 ] )
                    if timeSteps[ i ][ j - 1 ][ 0 ] + scaleFactor * K1[0] / 2 < 19 and
                                                                    timeSteps[ i ][ j - 1
                                                                    ][ 0 ] + scaleFactor *
                                                                     K1[0] / 2 > 0 and
                                                                    timeSteps[ i ][ j - 1
```

```
                                                                ][ 1 ] + scaleFactor *
                                                                 K1[1] / 2 < 19 and
                                                                timeSteps[ i ][ j - 1
                                                                ][ 1 ] + scaleFactor *
                                                                 K1[1] / 2 > 0:
                                    K2 = bilinearInterpolation( timeSteps[ i ][ j - 1 ] +
                                                                    scaleFactor * K1 /
                                                                     2 )
                                    if timeSteps[ i ][ j - 1 ][ 0 ] + scaleFactor * K2[0] / 2 < 19
                                                                    and timeSteps[ i ]
                                                                    [ j - 1 ][ 0 ] +
                                                                    scaleFactor * K2[0
                                                                    ] / 2 > 0 and
                                                                    timeSteps[ i ][ j
                                                                    - 1 ][ 1 ] +
                                                                    scaleFactor * K2[1
                                                                    ] / 2 < 19 and
                                                                    timeSteps[ i ][ j
                                                                    - 1 ][ 1 ] +
                                                                    scaleFactor * K2[1
                                                                    ] / 2 > 0:
                                        K3 = bilinearInterpolation( timeSteps[ i ][ j - 1 ] +
                                                                        scaleFactor *
                                                                        K2 / 2 )
                                        if timeSteps[ i ][ j - 1 ][ 0 ] + scaleFactor * K3[0] < 19
                                                                        and timeSteps[
                                                                         i ][ j - 1 ][
                                                                         0 ] +
                                                                        scaleFactor *
                                                                        K3[0] > 0 and
                                                                        timeSteps[ i ]
                                                                        [ j - 1 ][ 1 ]
                                                                         + scaleFactor
                                                                         * K3[1] < 19
                                                                        and timeSteps[
                                                                         i ][ j - 1 ][
                                                                         1 ] +
                                                                        scaleFactor *
                                                                        K3[1] > 0:
                                            K4 = bilinearInterpolation( timeSteps[ i ][ j - 1 ] +
                                                                            scaleFactor
                                                                             * K3 )
                                            timeSteps[ i ][ j ] = timeSteps[ i ][ j - 1 ] + 1 / 6 *
                                                                            ( K1 + 2 *
                                                                             K2 + 2 *
                                                                            K3 + K4 )
                                                                            *
                                                                            scaleFactor


                        if timeSteps[ i ][ j ][ 0 ] > 19 or timeSteps[ i ][ j ][ 0 ] < 0 or
                                                                    timeSteps[ i ][ j ][ 1 ] >
                                                                     19 or timeSteps[ i ][ j ]
                                                                    [ 1 ] < 0 or ( timeSteps[
                                                                    i ][ j ][ 0 ] == 0 and
                                                                    timeSteps[ i ][ j ][ 1 ] =
                                                                    = 0 ):
                            timeSteps[ i ][ j ] = timeSteps[ i ][ j - 1 ]

    return timeSteps


def computeAndPlot( steps ):

    # Compute all the arrays to plot
    eulerTimeSteps = computeTimeSteps( "Euler", steps )
    rk4TimeSteps = computeTimeSteps( "RK4", steps )

    # Plot the Data
    for i in range( len( eulerTimeSteps ) ):
        plt.plot(eulerTimeSteps[i][:, 0], eulerTimeSteps[i][:, 1], 'r.-', ms = 0.05,
                                                label = "Euler" if i == 0 else '')
```

```
        for i in range( len( rk4TimeSteps ) ):
            plt.plot(rk4TimeSteps[i][:, 0], rk4TimeSteps[i][:, 1], 'b.-', ms = 0.05, label =
                                                                "RK4" if i == 0 else '')


        # Show an image of your plot
        plt.title( f"Euler vs. RK4: Graph of Randomly Seeded Points with { steps } Steps" )
        plt.plot(xx, yy, marker='.', color='b', linestyle='none', ms = 0.01)
        plt.quiver(xx, yy, vecs_flat[:,0], vecs_flat[:,1], width=0.002)
        plt.scatter(seededPoints[:, 0], seededPoints[:, 1], marker = '.')
        plt.xlabel('X')
        plt.ylabel('Y')
        plt.legend()
        plt.grid()
        plt.show()


    computeAndPlot( 8 )
    computeAndPlot( 16 )
    computeAndPlot( 32 )
    computeAndPlot( 64 )
```

Conclusion:

This assignment showed how Paraview can be used to visualize vector fields using stream lines, tubes, and cone glyphs. It showed the importance of using colormaps with glyphs to show you the different flows in a data set and give you a better intuition for those flows. This assignment also showed different seeding types with line and cloud being the primarily used methods in the questions above. Line seeding can help show a whole plane of a data set which allowed us to see most of hurricane Katrina with a single line. Point cloud seeding allows the user to see specified regions of the data set in smaller details, instead of seeing everything in the data. Lastly, we saw the difference in integration methods used for vector fields. Euler's method is very easy to implement and runs quickly but is falls short on accuracy with small time steps. As the step size gets smaller and the number of steps gets bigger Euler's method gets more accurate but as we saw in the extra credit section above, other integration methods like RK4 have better accuracy at smaller step sizes.

References

[1] "Bilinear Interpolation" : https://en.wikipedia.org/wiki/Bilinear_interpolation

[2] "Euler's Method" : https://en.wikipedia.org/wiki/Euler_method

[3] "Runge-Kutta Methods" : https://en.wikipedia.org/wiki/Runge–Kutta_methods

[4] "Stack Overflow" : https://stackoverflow.com