Modeling the Magnus Effect with Baseball Trajectories

Scott Merkley

April/27/2022

Department of Physics, University of Utah, Salt Lake City, 84112, UT, USA

Introduction

In this paper, we will be modeling the trajectories of Minor League Baseball hits from Sports Media Technology data [1]. We will be using Newtons equations of motion along with the Prandtl and Magnus models. For a body in free-fall the Newtonian equations of motion for the acceleration of a body are:

$$F = ma \tag{1}$$

When modeling a body with no air resistance or other forces this equation is commonly written as:

$$F = mg \tag{2}$$

The Prandtl model adds in the effect of air resistance as the body travels through the air. This equation takes the cross-sectional area of the object A, density of air ρ, the objects velocity v, drag coefficient c_d, and unit vector $\hat{e}_v$. This is written as:

$$F_d = -\frac{1}{2} c_d \rho A v^2 \hat{e}_v \tag{3}$$

$$= -m k_d v^2 \hat{e}_v \tag{4}$$

When the ball is hit, it can have differing levels spin causing slower air and higher pressure to be at the top of the ball and faster airflow with lower pressure to be on the bottom or vice versa depending on how it was hit. This causes the balls trajectory to change, either leading it to fly further and stay in the air for longer, or fall faster and hit the ground sooner if it has backspin or forward spin respectively. The equations for the Magnus effect are similar to the Prandtl model but with the change of c_L being the coefficient of lift on the ball and the cross product in place of the unit vector.

$$F_L = \frac{1}{2} c_L \rho A v^2 \frac{\hat{w} \times \hat{v}}{|\hat{w} \times \hat{v}|} \tag{5}$$

$$= m k_L v^2 \frac{\hat{w} \times \hat{v}}{|\hat{w} \times \hat{v}|} \tag{6}$$

Then, using Newton's Second Law , we can write down the equations of motion for a baseball moving in three dimensions as:

$$\frac{dv_x}{dt} = -k_d v_x \sqrt{v_x^2 + v_y^2 + v_z^2} - k_{L_y} v_y \sqrt{v_x^2 + v_y^2 + v_z^2} + k_{L_z} v_z \sqrt{v_x^2 + v_y^2 + v_z^2} + a_x \tag{7}$$

$$\frac{dv_y}{dt} = -k_d v_y \sqrt{v_x^2 + v_y^2 + v_z^2} + k_{L_x} v_x \sqrt{v_x^2 + v_y^2 + v_z^2} + k_{L_z} v_z \sqrt{v_x^2 + v_y^2 + v_z^2} + a_y \tag{8}$$

$$\frac{dv_z}{dt} = -k_d v_z \sqrt{v_x^2 + v_y^2 + v_z^2} + k_{L_x} v_x \sqrt{v_x^2 + v_y^2 + v_z^2} - k_{L_y} v_y \sqrt{v_x^2 + v_y^2 + v_z^2} - g \tag{9}$$

$$\frac{dx}{dt} = v_x \tag{10}$$

$$\frac{dy}{dt} = v_y \tag{11}$$

$$\frac{dz}{dt} = v_z \tag{12}$$

These equations allow us to model the Magnus effect and air resistance in three dimensions.

Data Selection

The events in this report were chosen based off the calculated trajectories average difference from the actual hit and the difference in position between the calculated ground hit and the actual ground hit. For the difference between the paths this was done by taking the actual hit data and subtracting the calculated hit data from each axis, then taking the average. It was found that some of the closest trajectories had an average difference of anywhere from 5 to 50. Some of the lowest differences came from the calculated trajectory not going far enough along the actual hit and so the average difference was lower. Some of the differences around 50 looked to fit the data better and are thought to be off because they follow the actual hit to the ground and are mostly off on the x axis, which will be talked about later. For the end position this was done by using Pythagorean's theorem to find the distance between the two points. We accepted end positions that were found to be less than two meters from the actual end position. Using both of these methods allowed for us to find if the calculated trajectory and end positions were close enough to the actual data to be considered significant.

In our findings, most of the hits tended to stay around 1.5-4 meters in height. When this happens, it is usually called a line drive. Line drives happens because the ball is hit closer to its center and does not spin as much or travel as high in the air. When the ball spins less it is easier to calculate its trajectory, when the ball is hit lower or higher it has much more spin and other forces can take over its trajectory. One of the forces not taken into account here is wind. If the ball is flying higher it will be more likely to get pushed by the wind and cause its flight to curve. Since we were not given data regarding the wind, we are not able to calculate exact trajectories in this case as their difference from the x axis usually makes their trajectory difference too great.

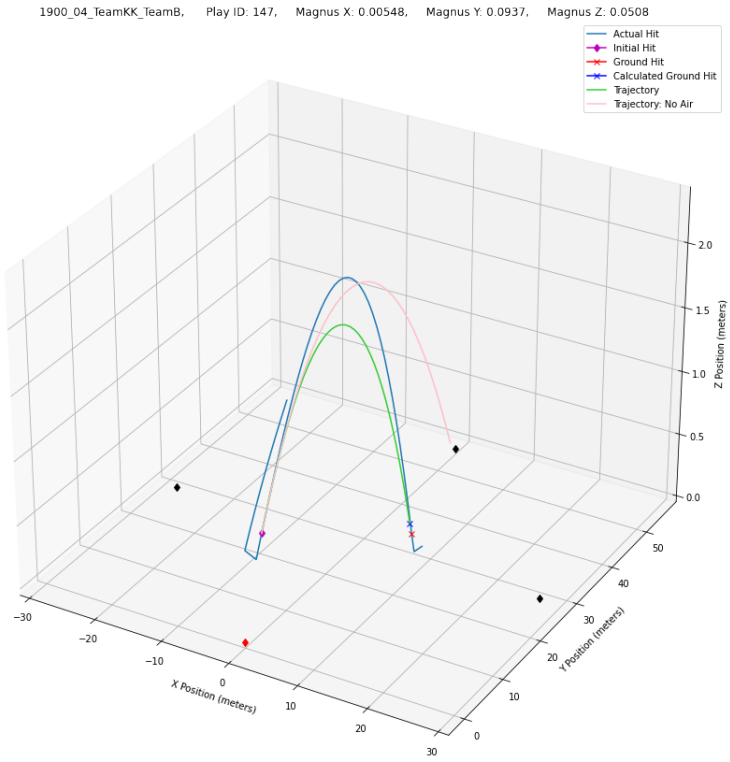

Figure 1: Trajectory Difference: 4.76, Drag Coefficient: 0.25, End Position Difference: 0.55m

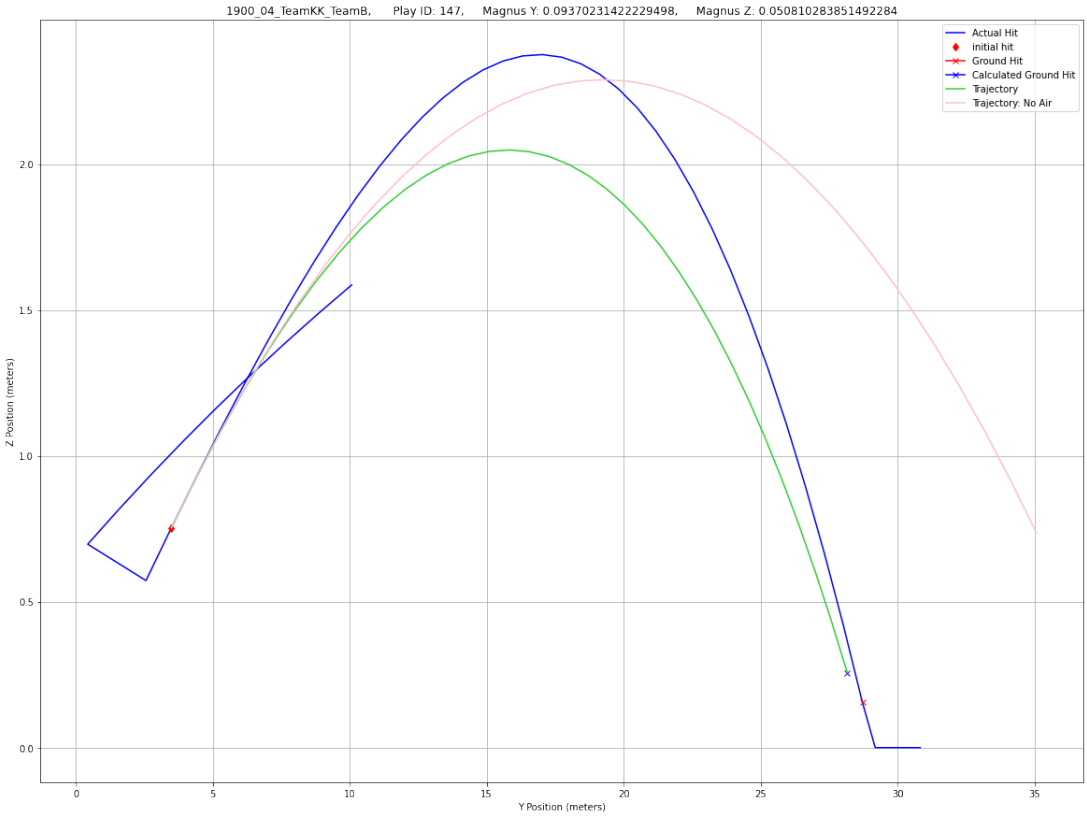

Figure 2: Trajectory Difference: 4.76, Drag Coefficient: 0.25, End Position Difference: 0.55m

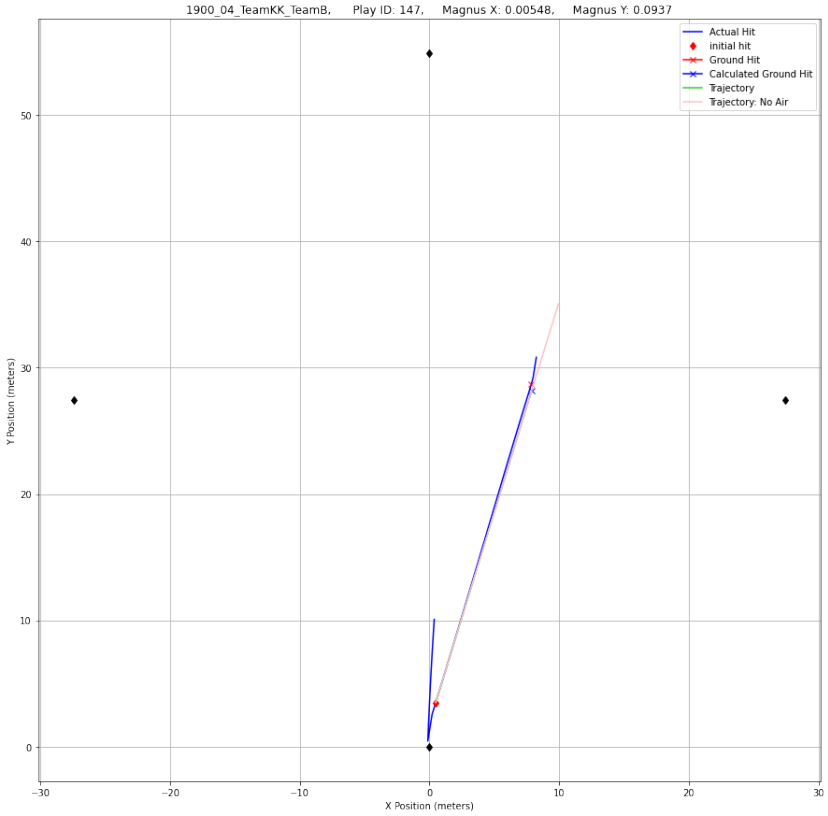

Figure 3: Trajectory Difference: 4.76, Drag Coefficient: 0.25, End Position Difference: 0.55m

Numerical Modeling

For modeling the baseball's trajectory we started out by reading the ball position file using Pandas. Then converted the ball positions from feet to meters and created delta columns for the change in position and time. This allowed us to calculate the velocity as the change in position over the change in time. We also calculated the acceleration as the change in velocity over the change in time. Now that we had all the necessary information for an initial hit and plug it into our integrator, Scipy's odeint. Before calculating we first found all the unique games and plays in the file using Pandas '.unique()' method and grouping the data. This gave us a data frame of all the games and play ID's so that we could examine all the hits. Then we decided that it would be necessary to see the plays so we made a plotGame method that takes a game name, play ID, and a plot method ('xy', 'xz', 'yz', '3D', etc.) and gives a specified plot back to the user. Later, as we were calculating trajectories we added a getTrajectory method that took in the gameData, initialHitIndex, and ballHitsGround parameters, creates an initial hit array, calculates the Magnus force on the ball in the x, y, and z axes, and gives back a trajectory using odeint.

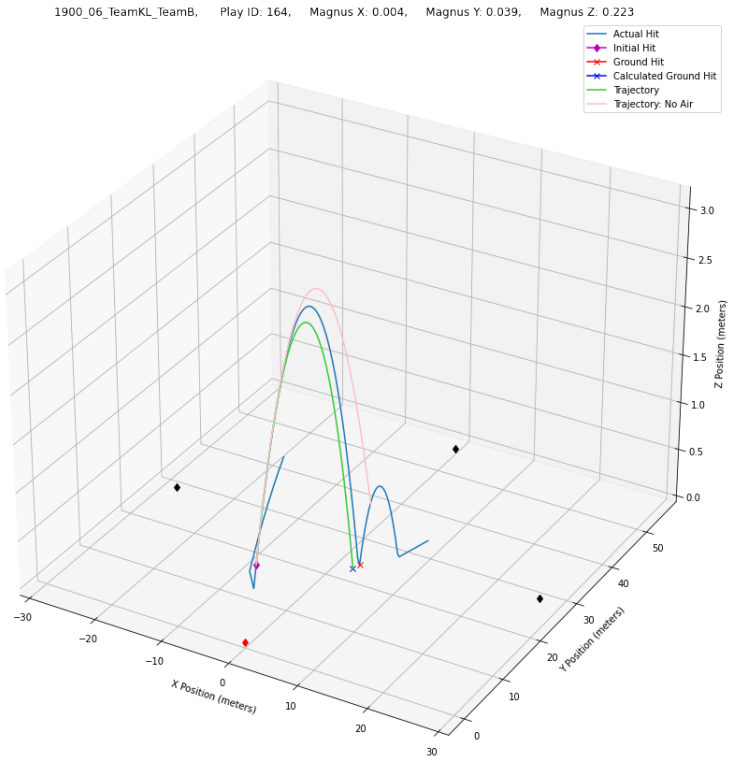

Figure 4: Trajectory Difference: 9.87, Drag Coefficient: 0.25, End Position Difference: 0.97m

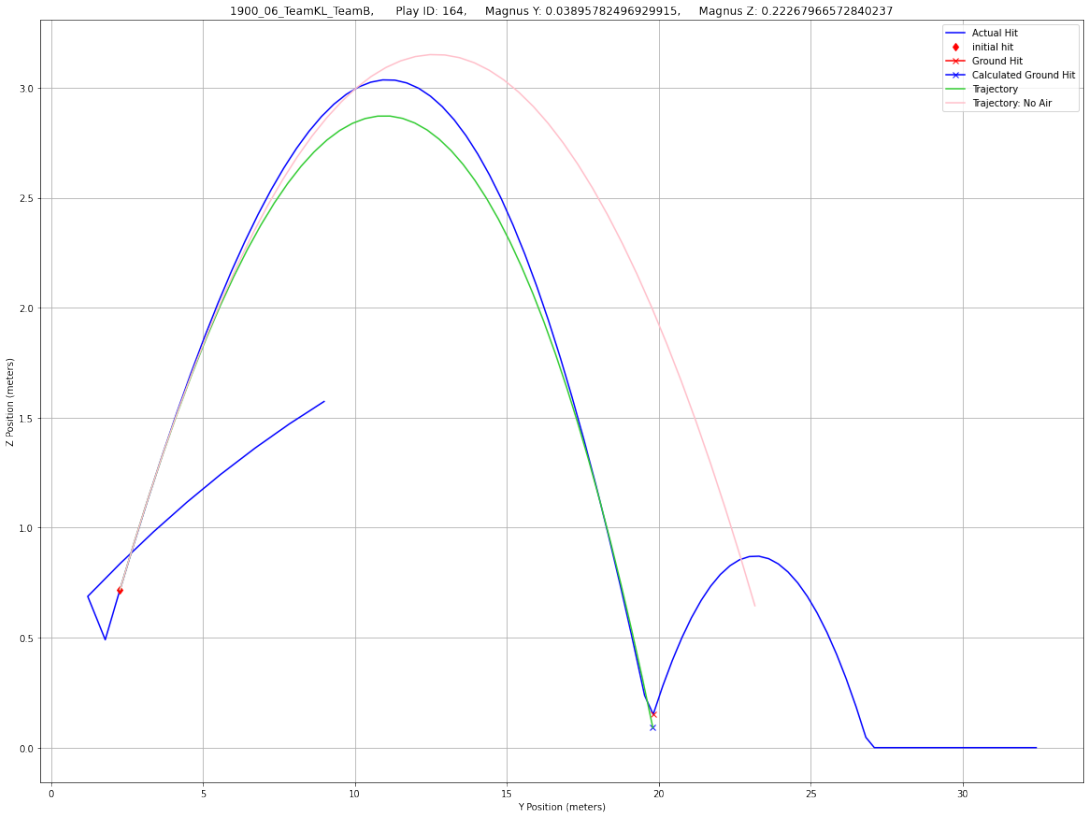

Figure 5: Trajectory Difference: 9.87, Drag Coefficient: 0.25, End Position Difference: 0.97m

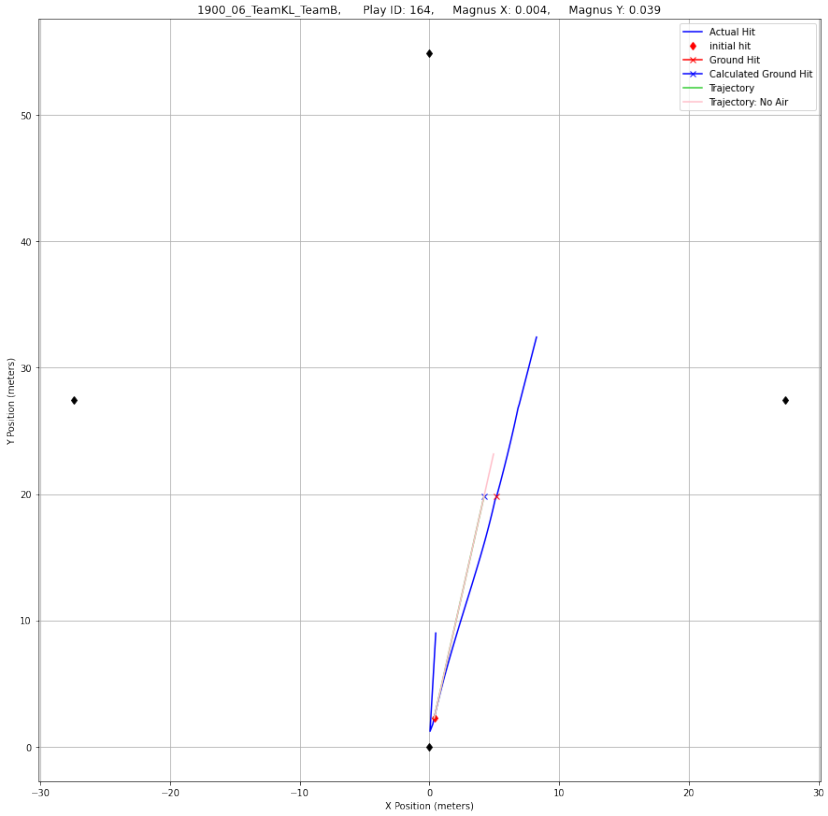

Figure 6: Trajectory Difference: 9.87, Drag Coefficient: 0.25, End Position Difference: 0.97m

We calculated the Magnus force by assuming that the acceleration found using the derivative of the position, a_{real} should be equal to the calculated acceleration, a_{calc}. The magnus force was calculated in 2D for the x, y, and z axes. This calculation was as follows:

$$a_{real} = a_{calc} \tag{13}$$

$$a_{real} = -k_d v_x \sqrt{v_x^2 + v_y^2} - k_{Ly} v_y \sqrt{v_x^2 + v_y^2} + a_{expected} \tag{14}$$

$$a_{real} + k_d v_x \sqrt{v_x^2 + v_y^2} - a_{expected} = -k_{Ly} v_y \sqrt{v_x^2 + v_y^2} \tag{15}$$

$$k_{Ly} = -\frac{a_{real} + k_d v_x \sqrt{v_x^2 + v_y^2} - a_{expected}}{v_y \sqrt{v_x^2 + v_y^2}} \tag{16}$$

Then following the same logic, we were able to solve for the Magnus force in x and z as well.

Discussion

Using the Prandtl and Magnus models consistently gave more accuracy than the trajectory calculated without air resistance. The model was not perfect and could be greatly improved upon by adding in the effects due to wind. This would make it more possible to follow the trajectories on fly balls as they curve. Also, better knowing the drag coefficient would make the calculations more accurate. There is variation from 0.2 to 0.5 and if this was know it would be much more simple to directly calculate the Magnus force. In this paper, a drag coefficient of 0.25 was used.

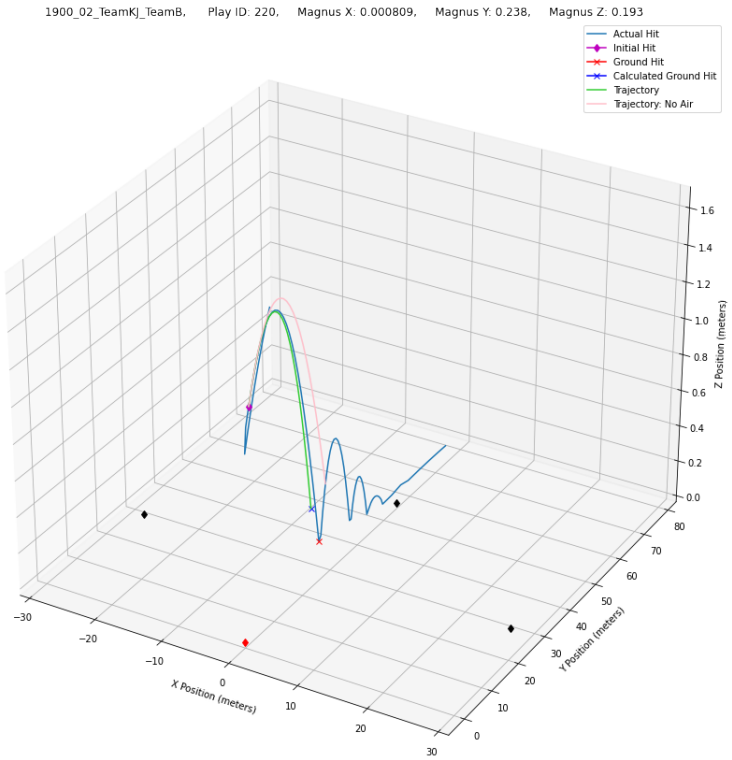

Figure 7: Trajectory Difference: 4.61, Drag Coefficient: 0.25, End Position Difference: 1.97m

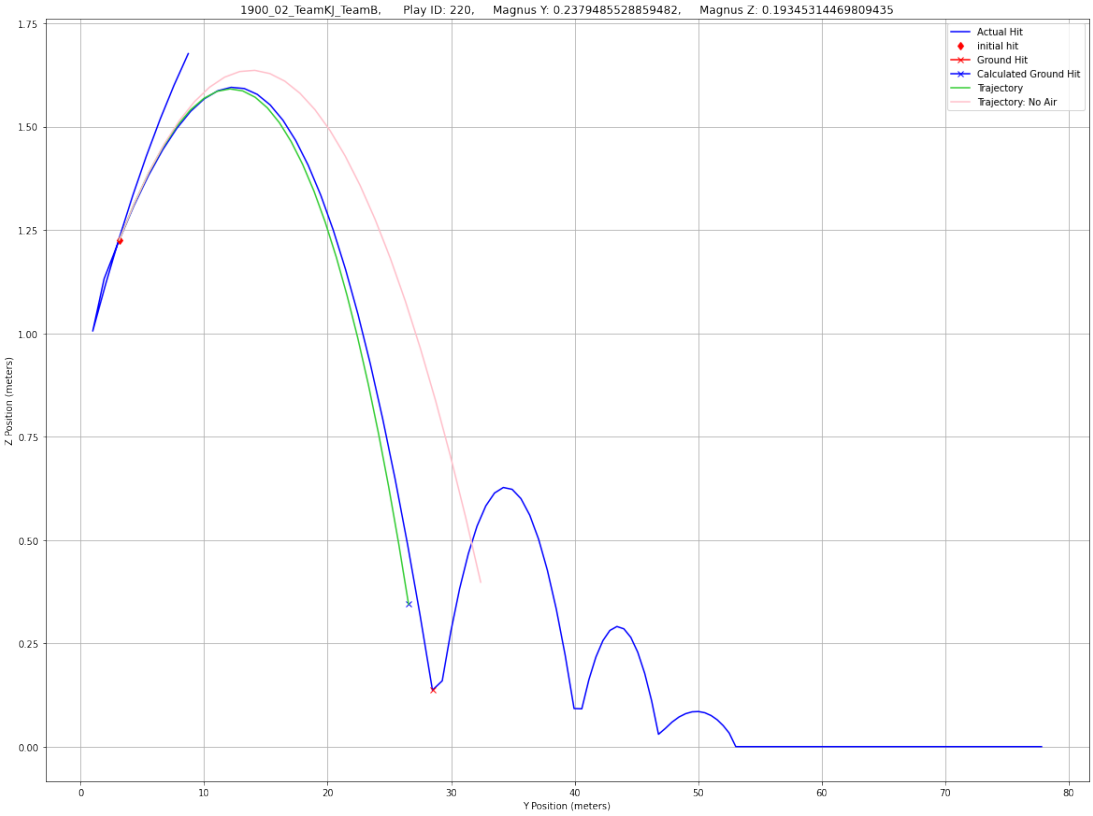

Figure 8: Trajectory Difference: 4.61, Drag Coefficient: 0.25, End Position Difference: 1.97m

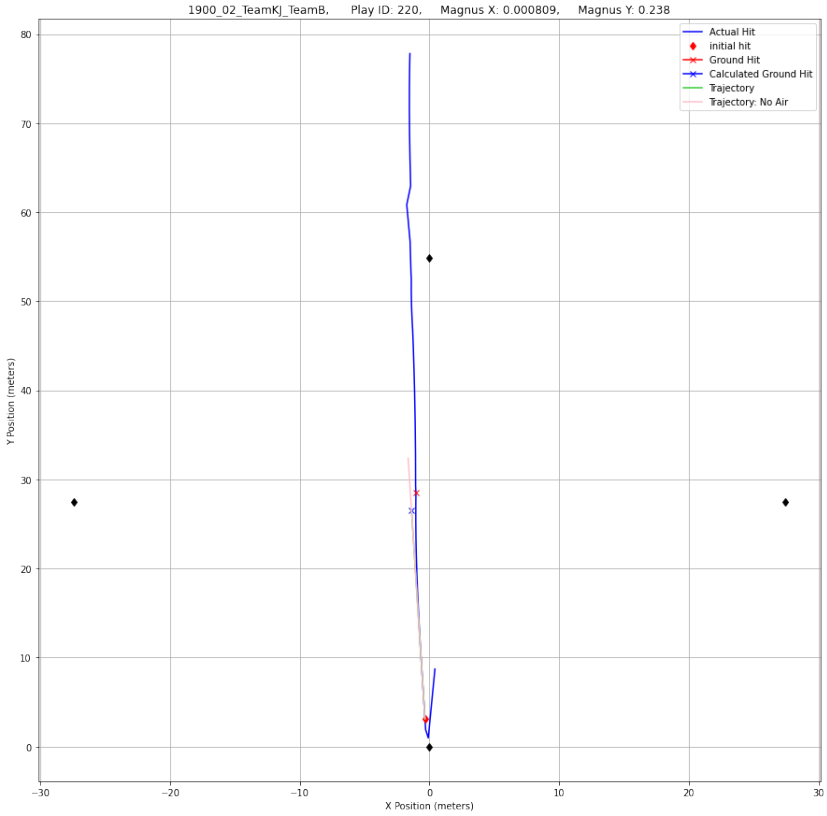

Figure 9: Trajectory Difference: 4.61, Drag Coefficient: 0.25, End Position Difference: 1.97m

There are times where the calculated Magnus force is greater than expected. This might have been caused by the smoothing effects on the data to make them match up during the flight or the effects of wind on the velocity and acceleration. We accounted for there to be no acceleration in the x and y directions after the ball was hit and if there was left over acceleration from the wind it would show in our calculated Magnus coefficient. We tried calculating the Magnus coefficient at both the initial hit and the peak of the trajectory. Both seemed to give respectable answers but could be off sometimes. Again, this is expected to be caused by wind or smoothing of the data. The expected values were used based off this graph [2].

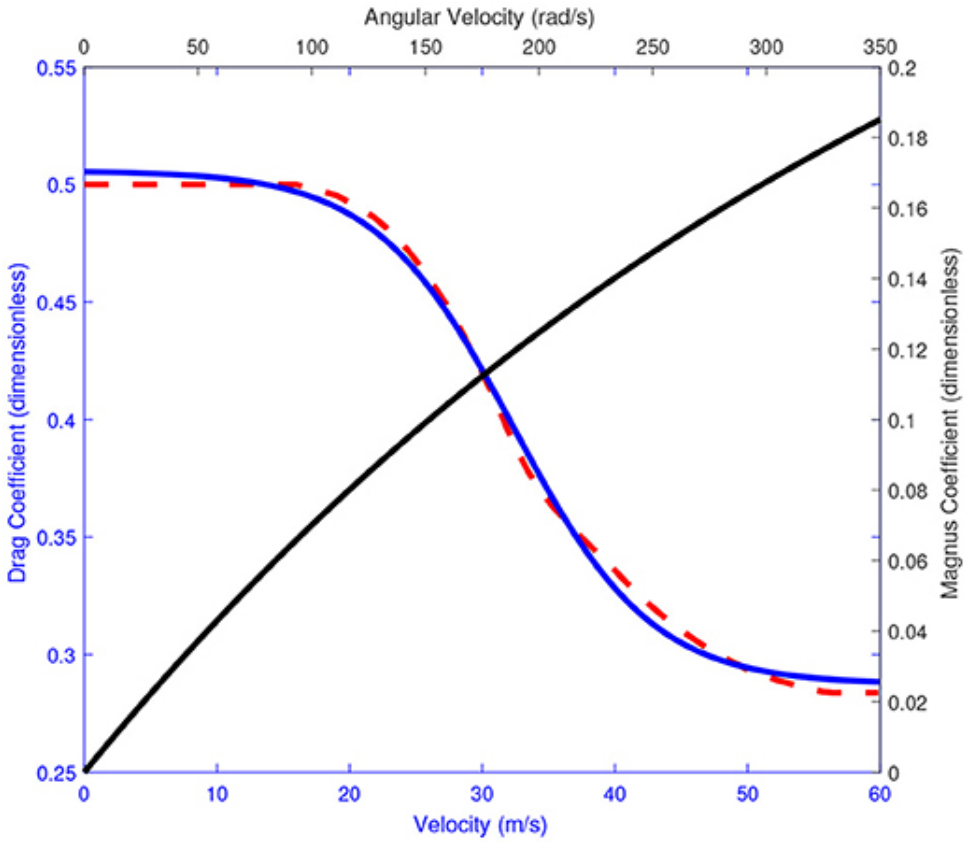

Figure 10: Magnus and Drag Coeffeficients vs Velocity

When calculating the initial velocity for the trajectory it was found that the actual initial hit index would give back bad approximations with negative starting velocities. This is because we calculated the velocity and acceleration using the point behind the current. We found that adding two to the initial hit index gave back much better results since the acceleration and velocity were accounted for. This allowed for more accurate initial hit parameters.

One of the reasons why the trajectory difference can be such a high number but looks to be a great fit is that it is taking all 3 dimensions into account. As stated earlier, the x axis is hard to calculate for since there could be added wind or a varying drag/Magnus coefficient with the speed of the ball. Since this number is calculated using all three dimensions, the y and z graph may look great but then it is found that the x axis is off due to wind or curvature of the balls flight path. This is seen in the game of Team NB vs Team A1, play 178:

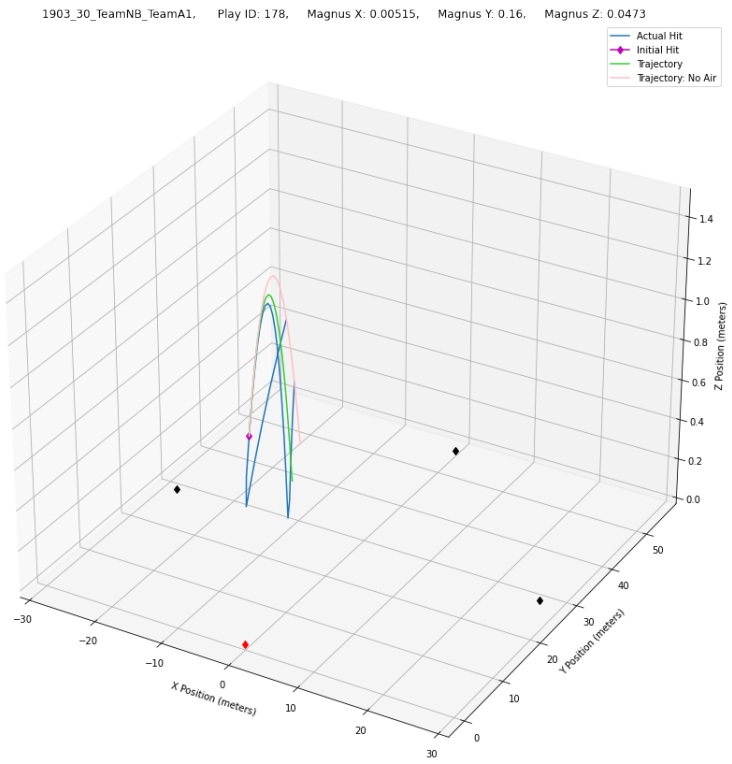

Figure 11: Trajectory Difference: 6.31, Drag Coefficient: 0.25

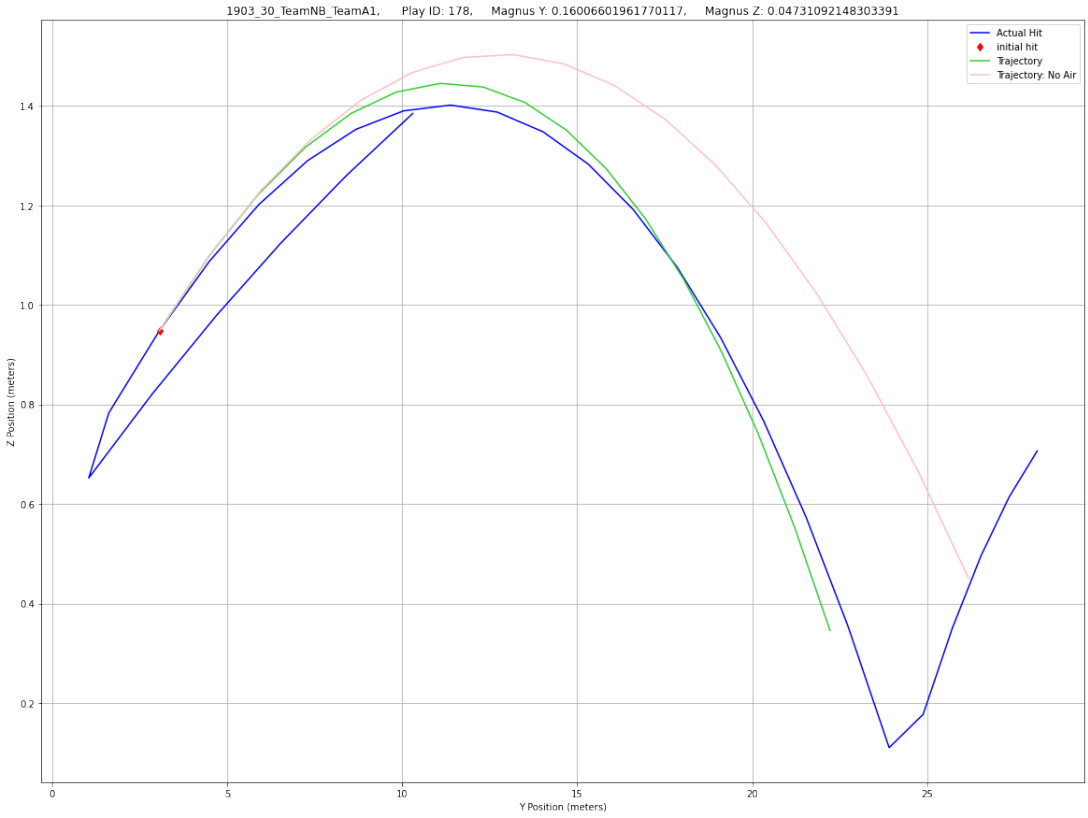

Figure 12: Trajectory Difference: 6.31, Drag Coefficient: 0.25

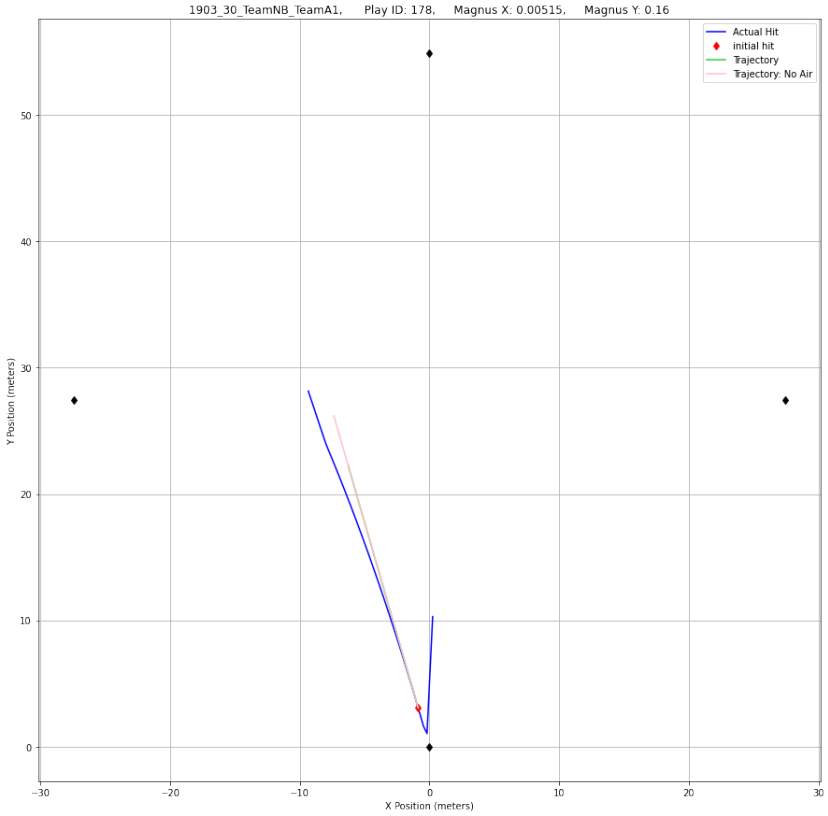

Figure 13: Trajectory Difference: 6.31, Drag Coefficient: 0.25

Conclusion

These calculations support the theoretical model for Prandtl and Magnus to a good degree of accuracy. The calculated Magnus coefficients are shown to be in the expected ranges. If there was information given regarding the wind and drag coefficient of the ball, the degree of accuracy would be greater. Also creating a drag coefficient function instead of a constant value, where the coefficient changed with the velocity would lead to more accurate trajectories. The graphs and calculations made from the given data file show that there is a great level of agreement between the trajectory calculated with air drag and Magnus effect and the actual fight path of the ball. Since the flight path differences are below ten and the end positions are less than two meters from the actual end positions this is proven to be a good model to accurately describe the trajectory of a baseball but has potential to be made better with additions of wind and varying Magnus/drag coefficients.

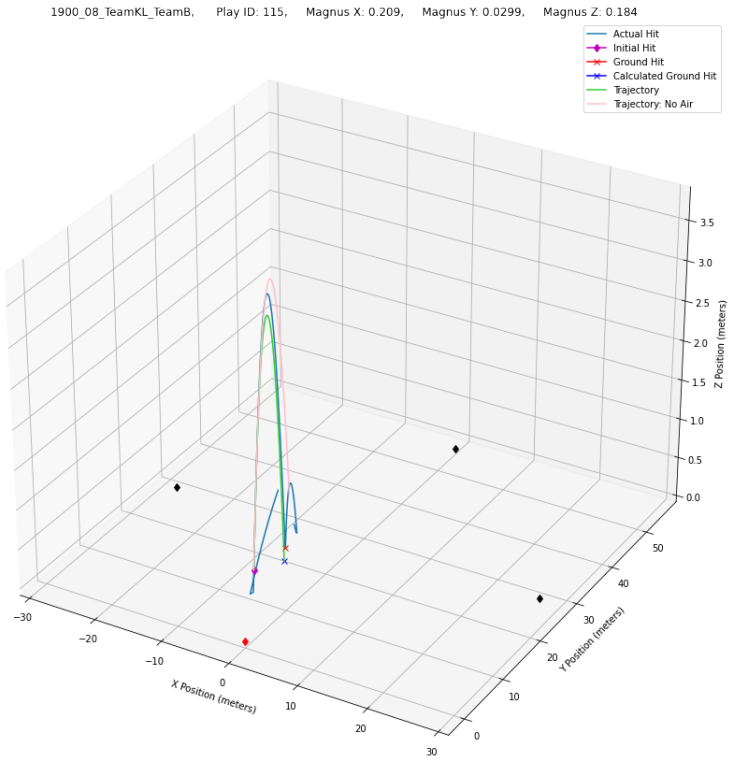

Figure 14: Trajectory Difference: 7.13, Drag Coefficient: 0.25, End Position Difference: 0.82m

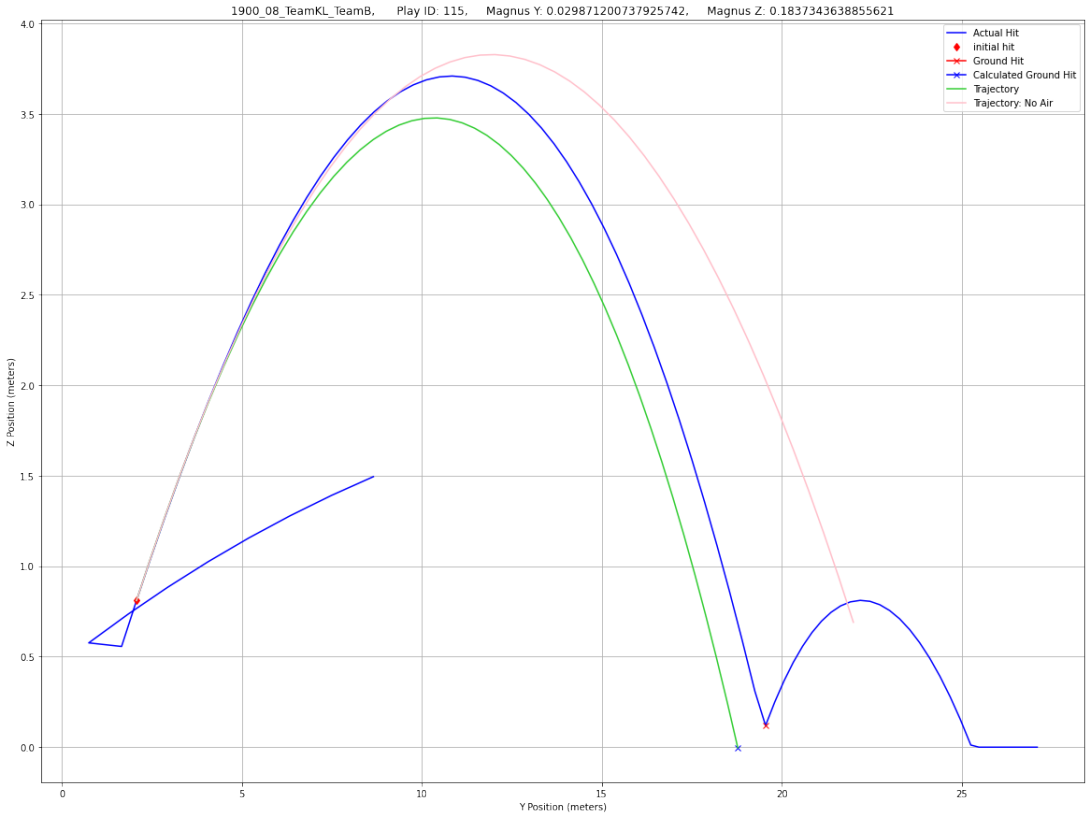

Figure 15: Trajectory Difference: 7.13, Drag Coefficient: 0.25, End Position Difference: 0.82m

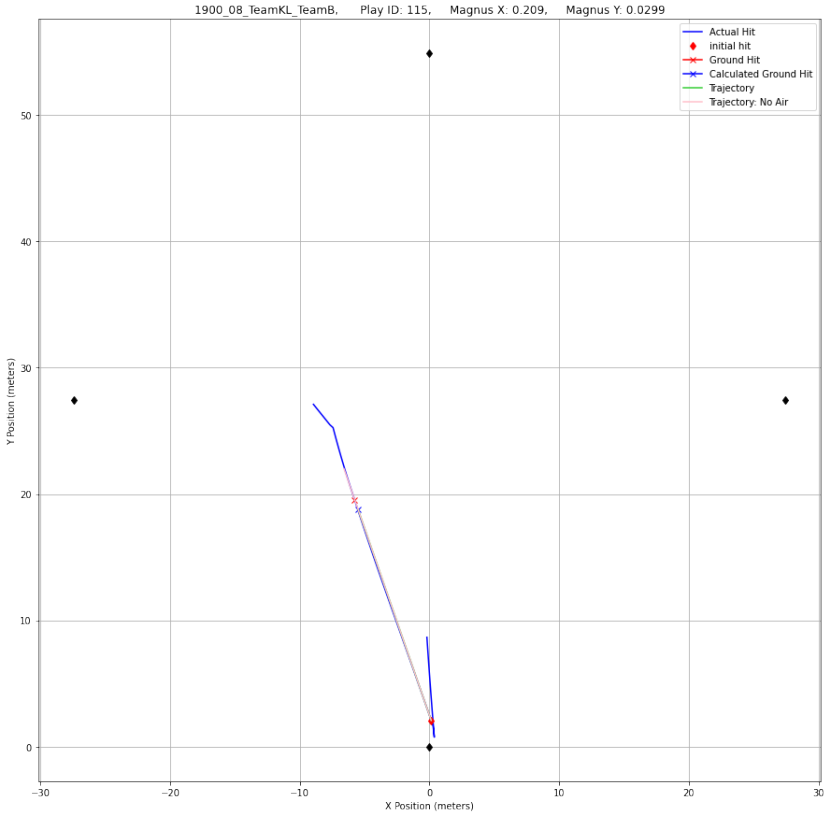

Figure 16: Trajectory Difference: 7.13, Drag Coefficient: 0.25, End Position Difference: 0.82m

References

[1] "SMT": smt.com

[2] "Magnus and Drag Coefficients": https://www.frontiersin.org/articles/10.3389/fams.2018.00066/full

Appendix

```
# Final Project
#
# This program will take the input baseball data and analyze it to conclude if there is an
# accurate way of modeling the baseball using the magnus effect.
#
# @author Scott Merkley
# @version April 15, 2022

%reset -f

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from mpl_toolkits.mplot3d import Axes3D
from scipy.integrate import odeint

# Contstants
feetToMeters = 0.3048
g            = 9.81 * 1 # 0.67 makes some of the graphs look REALLY good
fps          = 1e3
ballMass     = 0.145
ballRadius   = 0.075

# Computing Drag
dragCoeff    = 0.25 # and this at 0.5
rhoAir       = 1.2
airDrag      = (dragCoeff * rhoAir * np.pi * ballRadius**2) / (2 * ballMass)

# Computing Magnus Force (Making Global Variables)
realMagnusX = 0
realMagnusY = 0
realMagnusZ = 0


# Import the data from the file
data = pd.read_csv('fly_ball_ball_positions.dat', low_memory = False)

# Converting the positions to meters
data['ball_position_x'] = data['ball_position_x'] * feetToMeters
data['ball_position_y'] = data['ball_position_y'] * feetToMeters
data['ball_position_z'] = data['ball_position_z'] * feetToMeters

# Creating "delta" position columns
data['delta_x'] = -(data['ball_position_x'].shift(1) - data['ball_position_x'])
data['delta_y'] = -(data['ball_position_y'].shift(1) - data['ball_position_y'])
data['delta_z'] = -(data['ball_position_z'].shift(1) - data['ball_position_z'])
data['delta_t'] = -(data['timestamp'].shift(1) - data['timestamp'])

# Creating velocity columns, then setting all overlapping velocities to 0
threshold = 50
data['ball_velocity_x'] = data['delta_x'] / data['delta_t'] * fps
data['ball_velocity_x'] = np.where((data['ball_velocity_x'] > threshold) | (data['
                                    ball_velocity_x'] < -threshold), 0, data['
                                    ball_velocity_x'])
data['ball_velocity_y'] = data['delta_y'] / data['delta_t'] * fps
data['ball_velocity_y'] = np.where((data['ball_velocity_y'] > threshold) | (data['
                                    ball_velocity_y'] < -threshold), 0, data['
                                    ball_velocity_y'])
data['ball_velocity_z'] = data['delta_z'] / data['delta_t'] * fps
```

```
data['ball_velocity_z'] = np.where((data['ball_velocity_z'] > threshold) | (data['
                                        ball_velocity_z'] < -threshold), 0, data['
                                        ball_velocity_z'])

# Creating "delta" velocity columns
data['delta_vx'] = -(data['ball_velocity_x'].shift(1) - data['ball_velocity_x'])
data['delta_vy'] = -(data['ball_velocity_y'].shift(1) - data['ball_velocity_y'])
data['delta_vz'] = -(data['ball_velocity_z'].shift(1) - data['ball_velocity_z'])

# Creating acceleration columns, then setting all overlapping accelerations to 0
data['ball_acceleration_x'] = data['delta_vx'] / data['delta_t'] * fps
data['ball_acceleration_x'] = np.where((data['ball_acceleration_x'] > threshold) | (data['
                                        ball_acceleration_x'] < -threshold), 0, data['
                                        ball_acceleration_x'])
data['ball_acceleration_y'] = data['delta_vy'] / data['delta_t'] * fps
data['ball_acceleration_y'] = np.where((data['ball_acceleration_y'] > threshold) | (data['
                                        ball_acceleration_y'] < -threshold), 0, data['
                                        ball_acceleration_y'])
data['ball_acceleration_z'] = data['delta_vy'] / data['delta_t'] * fps
data['ball_acceleration_z'] = np.where((data['ball_acceleration_z'] > threshold) | (data['
                                        ball_acceleration_z'] < -threshold), 0, data['
                                        ball_acceleration_z'])

# Creating theta and phi columns
data['ball_theta'] = np.arctan(data['delta_y'] / data['delta_x'])
data['ball_phi']   = np.arctan(data['delta_z'] / data['delta_x'])


# Setting up an array of all the Play ID's per Game
games = data.groupby(['game_str'])['play_id'].unique().reset_index()


# Given a specified game name, play ID, and plot method will plot the game along with a
# two calculated trajectories, one with air resistance and Magnus force, other with no
# air resistance or Magnus force
#
# @param gameName   - name of the game to plot, given as a string
# @param playID     - ID of the game to plot, given as a integer
# @param plotMethod - axes to be plotted, given as a string (ie. xy, yz, 3D, vx, ax, etc.)
# @return           - plot printed to the console
def plotGame(gameName, playID, plotMethod):
    gameData = data[(data['game_str'] == gameName) & (data['play_id'] == playID)]
    pos = 90 * feetToMeters  # Diamond position in meters

    # Trying to find the first non negative point but sometimes all values are negative..
    initialHitIndex = gameData['ball_position_y'].abs().idxmin() + 2

    # Create a new data frame to return to the user with the same columns as before so that
                                            you can test them against each other
    ballHitsGround  = gameData[gameData['ball_position_z'].abs() < 0.2].index.values.astype(
                                            int)[0]

    trajectory, trajectoryNoAir, realMagnusX, realMagnusY, realMagnusZ = getTrajectory(
                                            gameData, initialHitIndex, ballHitsGround)
    trajectoryDF = pd.DataFrame(trajectory, columns = ['ball_position_x', 'ball_velocity_x',
                                             'ball_position_y', 'ball_velocity_y', '
                                            ball_position_z', 'ball_velocity_z'])

    trajectoryDifference = getTrajectoryDifference(gameData, trajectoryDF, initialHitIndex,
                                            ballHitsGround)
    finalPositionDifference = getFinalPositionDifference(gameData['ball_position_x'][
                                            ballHitsGround], gameData['ball_position_y
                                            '][ballHitsGround], gameData['
                                            ball_position_z'][ballHitsGround],\
            trajectoryDF['ball_position_x'].iloc[-1], trajectoryDF['ball_position_y'].iloc[-
                                                    1], trajectoryDF['ball_position_z'
                                                    ].iloc[-1])

    # If the total difference between the trajectory and the actual hit is less than 20 plot
                                             the hit
    if((abs(trajectoryDifference) < 50) & (finalPositionDifference < 2)):
```

```
print(f'trajectoryDifference: {trajectoryDifference}, dragCoeff: {dragCoeff},
                                                finalPositionDifference: {
                                                finalPositionDifference}')

# Trajectory Colors
trajectoryColor = 'limegreen'
trajectoryNoAirColor = 'pink'

if(plotMethod == 'xy'):
    # Plot the X vs Y data
    # Make a nice looking Baseball Diamond Plot
    plt.figure(figsize = (15, 15))
    plt.title(f'{gameName},      Play ID: {playID},     Magnus X: {realMagnusX:0.3},
                                                      Magnus Y: {realMagnusY:0.3}')
    plt.xlabel('X Position (meters)')
    plt.ylabel('Y Position (meters)')
    plt.plot(0, 0, 'kd', -pos, pos, 'kd', pos, pos, 'kd', 0, 2 * pos, 'kd')
    plt.plot(gameData['ball_position_x'], gameData['ball_position_y'], 'b-', label =
                                                'Actual Hit')
    plt.plot(gameData['ball_position_x'][initialHitIndex], gameData['ball_position_y
                                                '][initialHitIndex], 'rd', label =
                                                'initial hit')
    plt.plot(gameData['ball_position_x'][ballHitsGround], gameData['ball_position_y'
                                                ][ballHitsGround], c = 'r', marker
                                                ='x', label = 'Ground Hit')
    plt.plot(trajectoryDF['ball_position_x'].iloc[-1], trajectoryDF['ball_position_y
                                                '].iloc[-1], c = 'b', marker ='x',
                                                label = 'Calculated Ground Hit')
    plt.plot(trajectory[:, 0], trajectory[:, 2], trajectoryColor, label = '
                                                Trajectory')
    plt.plot(trajectoryNoAir[:, 0], trajectoryNoAir[:, 2], trajectoryNoAirColor,
                                                label = 'Trajectory: No Air')
    plt.legend()
    plt.grid()
    plt.show()

if(plotMethod == 'xz'):
    # Plot the X vs Z data
    plt.figure(figsize = (20, 15))
    plt.title(f'{gameName},      Play ID: {playID},     Magnus X: {realMagnusX},
                                                    Magnus Z: {realMagnusZ}')
    plt.xlabel('X Position (meters)')
    plt.ylabel('Z Position (meters)')
    plt.plot(gameData['ball_position_x'], gameData['ball_position_z'], 'b-', label =
                                                'Actual Hit')
    plt.plot(gameData['ball_position_x'][initialHitIndex], gameData['ball_position_z
                                                '][initialHitIndex], 'rd', label =
                                                'initial hit')
    plt.plot(gameData['ball_position_x'][ballHitsGround], gameData['ball_position_z'
                                                ][ballHitsGround], c = 'r', marker
                                                ='x', label = 'Ground Hit')
    plt.plot(trajectoryDF['ball_position_x'].iloc[-1], trajectoryDF['ball_position_z
                                                '].iloc[-1], c = 'b', marker ='x',
                                                label = 'Calculated Ground Hit')
    plt.plot(trajectory[:, 0], trajectory[:, 4], trajectoryColor, label = '
                                                Trajectory')
    plt.plot(trajectoryNoAir[:, 0], trajectoryNoAir[:, 4], trajectoryNoAirColor,
                                                label = 'Trajectory: No Air')
    plt.legend()
    plt.grid()
    plt.show()

if(plotMethod == 'yz'):
    # Plot the Y vs Z data
    plt.figure(figsize = (20, 15))
    plt.title(f'{gameName},      Play ID: {playID},     Magnus Y: {realMagnusY},
                                                    Magnus Z: {realMagnusZ}')
    plt.xlabel('Y Position (meters)')
    plt.ylabel('Z Position (meters)')
    plt.plot(gameData['ball_position_y'], gameData['ball_position_z'], 'b-', label =
                                                'Actual Hit')
```

```
        plt.plot(gameData['ball_position_y'][initialHitIndex], gameData['ball_position_z
                                                    '][initialHitIndex], 'rd', label =
                                                     'initial hit')
        plt.plot(gameData['ball_position_y'][ballHitsGround], gameData['ball_position_z'
                                                    ][ballHitsGround], c = 'r', marker
                                                     ='x', label = 'Ground Hit')
        plt.plot(trajectoryDF['ball_position_y'].iloc[-1], trajectoryDF['ball_position_z
                                                    '].iloc[-1], c = 'b', marker ='x',
                                                     label = 'Calculated Ground Hit')
        plt.plot(trajectory[:, 2], trajectory[:, 4], trajectoryColor, label = '
                                                    Trajectory')
        plt.plot(trajectoryNoAir[:, 2], trajectoryNoAir[:, 4], trajectoryNoAirColor,
                                                    label = 'Trajectory: No Air')
        plt.legend()
        plt.grid()
        plt.show()

    if(plotMethod == '3D'):
        # Plot the 3D data
        plt.figure(figsize = (15, 15))
        ax = plt.axes(projection='3d')
        ax.set_title(f'{gameName},      Play ID: {playID},      Magnus X: {realMagnusX:0.
                                                    3},      Magnus Y: {realMagnusY:0.3
                                                    },      Magnus Z: {realMagnusZ:0.3}
                                                    ')
        ax.plot3D(0, 0, 0, c = 'r', marker = 'd')
        ax.plot3D(-pos, pos, 0, c = 'k', marker = 'd')
        ax.plot3D(pos, pos, 0, c = 'k', marker = 'd')
        ax.plot3D(0, 2 * pos, 0, c = 'k', marker = 'd')
        ax.plot3D(gameData['ball_position_x'], gameData['ball_position_y'], gameData['
                                                    ball_position_z'], label = 'Actual
                                                     Hit')
        ax.plot3D(gameData['ball_position_x'][initialHitIndex], gameData['
                                                    ball_position_y'][initialHitIndex]
                                                    , gameData['ball_position_z'][
                                                    initialHitIndex], c = 'm', marker
                                                    ='d', label = 'Initial Hit')
        ax.plot3D(gameData['ball_position_x'][ballHitsGround], gameData['ball_position_y
                                                    '][ballHitsGround], gameData['
                                                    ball_position_z'][ballHitsGround],
                                                     c = 'r', marker ='x', label = '
                                                    Ground Hit')
        ax.plot3D(trajectoryDF['ball_position_x'].iloc[-1], trajectoryDF['
                                                    ball_position_y'].iloc[-1],
                                                    trajectoryDF['ball_position_z'].
                                                    iloc[-1], c = 'b', marker ='x',
                                                    label = 'Calculated Ground Hit')
        ax.plot3D(trajectory[:, 0], trajectory[:, 2], trajectory[:, 4], trajectoryColor,
                                                     label = 'Trajectory')
        plt.plot(trajectoryNoAir[:, 0], trajectoryNoAir[:, 2], trajectoryNoAir[:, 4],
                                                    trajectoryNoAirColor, label = '
                                                    Trajectory: No Air')
        ax.set_xlabel('X Position (meters)')
        ax.set_ylabel('Y Position (meters)')
        ax.set_zlabel('Z Position (meters)')
        ax.legend()
        plt.show()

    if(plotMethod == 'vx'):
        t = np.linspace(0, len(gameData['ball_velocity_x']) / 60, len(gameData['
                                                    ball_velocity_x']))
        plt.figure(figsize = (15, 15))
        plt.title(f'{gameName},      Play ID: {playID}')
        plt.xlabel('time (sec)')
        plt.ylabel('x velocity (m/s)')
        plt.plot(t, gameData['ball_velocity_x'])
        plt.show()

    if(plotMethod == 'vy'):
        t = np.linspace(0, len(gameData['ball_velocity_y']) / 60, len(gameData['
                                                    ball_velocity_y']))
        plt.figure(figsize = (15, 15))
```

```
            plt.title(f'{gameName},      Play ID: {playID}')
            plt.xlabel('time (sec)')
            plt.ylabel('y velocity (m/s)')
            plt.plot(t, gameData['ball_velocity_y'])
            plt.show()

        if(plotMethod == 'vz'):
            t = np.linspace(0, len(gameData['ball_velocity_z']) / 60, len(gameData['
                                                    ball_velocity_z']))
            plt.figure(figsize = (15, 15))
            plt.title(f'{gameName},      Play ID: {playID}')
            plt.xlabel('time (sec)')
            plt.ylabel('z velocity (m/s)')
            plt.plot(t, gameData['ball_velocity_z'])
            plt.show()

        if(plotMethod == 'ax'):
            t = np.linspace(0, len(gameData['ball_acceleration_x']) / 60, len(gameData['
                                                    ball_acceleration_x']))
            plt.figure(figsize = (15, 15))
            plt.title(f'{gameName},      Play ID: {playID}')
            plt.xlabel('time (sec)')
            plt.ylabel('x acceleration (m/s^2)')
            plt.plot(t, gameData['ball_acceleration_x'])
            plt.show()

        if(plotMethod == 'ay'):
            t = np.linspace(0, len(gameData['ball_acceleration_y']) / 60, len(gameData['
                                                    ball_acceleration_y']))
            plt.figure(figsize = (15, 15))
            plt.title(f'{gameName},      Play ID: {playID}')
            plt.xlabel('time (sec)')
            plt.ylabel('y acceleration (m/s^2)')
            plt.plot(t, gameData['ball_acceleration_y'])
            plt.show()

        if(plotMethod == 'az'):
            t = np.linspace(0, len(gameData['ball_acceleration_z']) / 60, len(gameData['
                                                    ball_acceleration_z']))
            plt.figure(figsize = (15, 15))
            plt.title(f'{gameName},      Play ID: {playID}')
            plt.xlabel('time (sec)')
            plt.ylabel('z acceleration (m/s^2)')
            plt.plot(t, gameData['ball_acceleration_z'])
            plt.show()


# This method calculates the difference between the given trajectory and the game data,
                                            returning
# a number corresponding to the fit of the trajectory. The lower the number, the closer the
                                            fit.
#
# @param gameData                   - the game data to be used, given as a pandas dataframe
                                            with specific columns
# @param trajectory                 - the trajectory to find the difference between, given
                                            as a odeint solution
# @param initialHitIndex            - the index of the first hit to be started at, given as
                                            an integer
# @param ballHitsGround             - the index when the ball hits the ground, given as an
                                            integer
# @return totalTrajectoryDifference - the difference between the hit and trajectory, given
                                            as an integer
def getTrajectoryDifference(gameData, trajectoryDF, initialHitIndex, ballHitsGround):

    # This calculates how close the trajectory is to the actual hit
    trajectoryDifferenceX = np.abs(np.sum(data['ball_position_x'].iloc[initialHitIndex :
                                                ballHitsGround].reset_index(drop = True) -
                                                 trajectoryDF['ball_position_x'].iloc[0 :
                                                ballHitsGround - initialHitIndex]))
    trajectoryDifferenceY = np.abs(np.sum(data['ball_position_y'].iloc[initialHitIndex :
                                                ballHitsGround].reset_index(drop = True) -
                                                 trajectoryDF['ball_position_y'].iloc[0 :
```

```
                                                ballHitsGround - initialHitIndex]))
    trajectoryDifferenceZ = np.abs(np.sum(data['ball_position_z'].iloc[initialHitIndex :
                                                ballHitsGround].reset_index(drop = True) -
                                                 trajectoryDF['ball_position_z'].iloc[0 :
                                                ballHitsGround - initialHitIndex]))
    totalTrajectoryDifference = (trajectoryDifferenceX + trajectoryDifferenceY +
                                                trajectoryDifferenceZ) / 3

    return totalTrajectoryDifference



# This method takesn in the game data, intital hit index, and the index of when the ball
                                                hits the ground
# and calculates the trajectory using Scipy's odeint and returns the trajectory along with
                                                the calculated magnus
# coefficients.
#
# @param gameData        - the game data to be used, given as a pandas dataframe with
                                                specific columns
# @param initialHitIndex - the index of the first hit to be started at, given as an integer
# @param ballHitsGround  - the index when the ball hits the ground, given as an integer
# @return an array with the air resistance trajectory, vacuum trajectory, x magnus
                                                coefficient, y magnus coefficient, z magnus
                                                coefficient
def getTrajectory(gameData, initialHitIndex, ballHitsGround):

    # Create the initial hit array
    initialHit = np.array([gameData['ball_position_x'][initialHitIndex], gameData['
                                                ball_velocity_x'][initialHitIndex],\
                           gameData['ball_position_y'][initialHitIndex], gameData['
                                                                        ball_velocity_y'][
                                                                        initialHitIndex],\
                            gameData['ball_position_z'][initialHitIndex], gameData['
                                                                         ball_velocity_z'][
                                                                         initialHitIndex]])

    # Prints the Initial hit information
#     print(f'x: {initialHit[0]:0.3}, vx: {initialHit[1]:0.3}, \ny: {initialHit[2]:0.3}, vy:
                                                 {initialHit[3]:0.3}, \nz: {initialHit[4]:0.3}
                                                , vz: {initialHit[5]:0.3}')

    # Wanting to try taking the X and Y magnus effects at when the ball reaches its max
                                                height and see if we can get some closer
                                                numbers
#     peakHeightIndex = gameData['ball_position_z'].abs().idxmax()
#     peakVelocity = np.linalg.norm(np.array([gameData['ball_velocity_x'][peakHeightIndex],
                                                gameData['ball_velocity_y'][peakHeightIndex],
                                                gameData['ball_velocity_z'][peakHeightIndex]])
                                                )
#     realMagnusX = (gameData['ball_acceleration_x'][peakHeightIndex] + airDrag *
                                                peakVelocity * gameData['ball_velocity_x'][
                                                peakHeightIndex]) / (peakVelocity * gameData['
                                                ball_velocity_x'][peakHeightIndex])
#     realMagnusY = (gameData['ball_acceleration_y'][peakHeightIndex] + airDrag *
                                                peakVelocity * gameData['ball_velocity_y'][
                                                peakHeightIndex]) / (peakVelocity * gameData['
                                                ball_velocity_y'][peakHeightIndex])
#     realMagnusZ = (gameData['ball_acceleration_z'][peakHeightIndex] + airDrag *
                                                peakVelocity * gameData['ball_velocity_z'][
                                                peakHeightIndex] + g) / (peakVelocity *
                                                gameData['ball_velocity_z'][peakHeightIndex])

    # Trying to calculate the actual magnus effect on the ball and set the global variable
    initialVelocity = np.linalg.norm(np.array([gameData['ball_velocity_x'][initialHitIndex],
                                                 gameData['ball_velocity_y'][
                                                initialHitIndex], gameData['
                                                ball_velocity_z'][initialHitIndex]]))
    realMagnusX = abs((gameData['ball_acceleration_x'][initialHitIndex] + airDrag *
                                                initialVelocity * gameData['
                                                ball_velocity_x'][initialHitIndex]) / (
                                                initialVelocity * gameData['
```

```
                                                ball_velocity_y'][initialHitIndex]))
    realMagnusY = abs((gameData['ball_acceleration_y'][initialHitIndex] + airDrag *
                                                initialVelocity * gameData['
                                                ball_velocity_y'][initialHitIndex]) / (
                                                initialVelocity * gameData['
                                                ball_velocity_z'][initialHitIndex]))
    realMagnusZ = abs((gameData['ball_acceleration_z'][initialHitIndex] + airDrag *
                                                initialVelocity * gameData['
                                                ball_velocity_z'][initialHitIndex] + g) /
                                                (initialVelocity * gameData['
                                                ball_velocity_x'][initialHitIndex]))

    # Create a time array
    t = np.linspace(0, (gameData['timestamp'][ballHitsGround] - gameData['timestamp'][
                                                initialHitIndex]) / fps, ballHitsGround -
                                                initialHitIndex + 1)

    # Make an odeint solver and plug the initial hit array into it with time
    return np.array([odeint(positionDerivative, initialHit, t), odeint(
                                                vacuumPositionDerivative, initialHit, t),
                                                realMagnusX, realMagnusY, realMagnusZ],
                                                dtype = object)


# This method is an air resistance derivative calculator for Scipy's odeint function. It
                                            takes in a specified position array s and
                                            gives back the
# derivates of the positions and velocities.
#
# @param s - an array with specified position and velocities (see below)
# @param t - an array of time values to be calculated against
# @return the derivative of the s array
def positionDerivative(s, t):
    # s is packed like:
    #   s[0] = x velocity
    #   s[1] = x acceleration
    #   s[2] = y velocity
    #   s[3] = y acceleration
    #   s[4] = z velocity
    #   s[5] = z acceleration

    # Unpack the array
    vx    = s[1]
    ax    = 0

    vy    = s[3]
    ay    = 0

    vz    = s[5]
    az    = -g

    vNow = np.array([vx, vy, vz])

    axNow = -airDrag * np.linalg.norm(vNow) * vx - realMagnusY * np.linalg.norm(vNow) * vy +
                                                realMagnusZ * np.linalg.norm(vNow) * vz +
                                                ax
    ayNow = -airDrag * np.linalg.norm(vNow) * vy + realMagnusX * np.linalg.norm(vNow) * vx +
                                                realMagnusZ * np.linalg.norm(vNow) * vz +
                                                ay
    azNow = -airDrag * np.linalg.norm(vNow) * vz + realMagnusX * np.linalg.norm(vNow) * vx -
                                                realMagnusY * np.linalg.norm(vNow) * vy +
                                                az

    dsdt = np.array([vx, axNow, vy, ayNow, vz, azNow])

    return dsdt

# This method is an vacuum derivative calculator for Scipy's odeint function. It takes in a
                                            specified position array s and gives back the
# derivates of the positions and velocities.
#
# @param s - an array with specified position and velocities (see below)
```

```
# @param t - an array of time values to be calculated against
# @return the derivative of the s array
def vacuumPositionDerivative(s, t):
    # s is packed like:
    #   s[0] = x velocity
    #   s[1] = x acceleration
    #   s[2] = y velocity
    #   s[3] = y acceleration
    #   s[4] = z velocity
    #   s[5] = z acceleration

    # Unpack the array
    vx     = s[1]
    ax     = 0

    vy     = s[3]
    ay     = 0

    vz     = s[5]
    az     = -g

    vNow = np.array([vx, vy, vz])

    axNow = ax
    ayNow = ay
    azNow = az

    dsdt = np.array([vx, axNow, vy, ayNow, vz, azNow])

    return dsdt

def getFinalPositionDifference(x0, y0, z0, x1, y1, z1):
    return np.sqrt((x1 - x0)**2 + (y1 - y0)**2 + (z1 - z0)**2)


# Plot all the games in the data file
for game in range(0, len(games['game_str'])):
    for ID in range(0, len(games['play_id'][game])):

        try:
            plotGame(games['game_str'][game], games['play_id'][game][ID], 'yz')
        except Exception as e:
            gameName = games['game_str'][game]
            playID = games['play_id'][game][ID]
#             print(f'Was not able to plot the game: {gameName}, play id: {playID}.')
```