ENPH 213 Project 2020

Final Report – Projectile Game Theatre

2020-04-15

Csaba Nemeth

Queen's University, Kingston ON

Student ID: 20090753

Section 1 Background – Projectile Modelling

The modelling of a projectile in a flight is most commonly described using the analytical equations of kinematic motion, namely that, for a given initial velocity v_0 and initial position s_0, the position of a particle undergoing constant acceleration a is given as a function of time t.

$$\vec{s} = \overrightarrow{s_0} + \overrightarrow{v_0}t + \frac{1}{2}\vec{a}t^2 \tag{1}$$

Equation (1) describes the shape of a parabola, and while accurate within a vacuum, it diverges from the true projectile path in the presence of aerodynamic drag and buoyancy. This divergence typically occurs at a location between 10% and 70% of the total path length, but can vary greatly depending on the characteristics of the projectile in question [1]. Examples of such trajectories are shown in Figure 1.

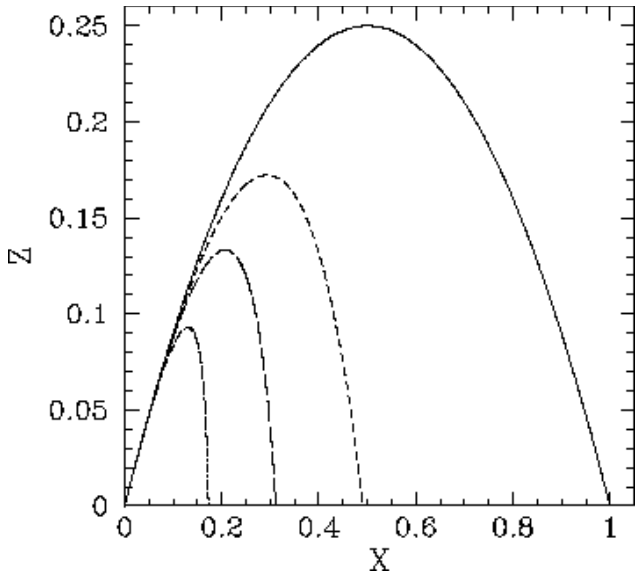

Figure 1: Various projectile trajectories, indicated with a dashed line, plotted against a trajectory calculated using Equation (1). Height Z is plotted against horizontal distance X, relative to the total path length. Citation: [2].

This figure indicates that another approach is needed to describe the path of a projectile for the game theatre simulation.

Section 1.2 Modelling with Drag and Buoyancy

The force of drag F_D of an object traveling through a fluid is given by:

$$F_D = \frac{1}{2}\rho|v|^2 C_D A \tag{2}$$

where ρ is the density of the fluid, $|v|$ is the magnitude of the projectile velocity relative to the fluid, C_D is the coefficient of drag, and A is the cross-sectional area of the projectile [3]. The acceleration due to drag in a cartesian component x, y, z, is given by:

$$a_{D\,x,y,z} = \frac{F_D}{m}\frac{v_{x,y,z}}{|v|} \tag{3}$$

where m is the mass of the projectile. Finally, the force of buoyancy acting on a particle is given in Equation (4):

$$F_B = \rho g V_f \, [+\hat{z}] \tag{4}$$

where g is the acceleration due to gravity, and V_f is the volume of the displaced fluid. In this case the buoyancy force acts in the positive z direction, opposing the force of gravity. This model can be constructed analytically, as Seungyeop Han outlines in a journal article published to the International Federation of Automatic Control journal [4]. These equations of motions are accurate to within 0.01% of the true projectile position but require precise knowledge of environmental conditions and projectile parameters [4]. This is not a viable option for the simulation because the model must be able to accurately predict positions of projectiles purely from positional, velocity, and acceleration data.

Figure 1 and Equation (1) indicate that while divergence occurs for relatively large time-intervals, over a small finite interval dt, Equation (1) is accurate and reduces to a linear form. This means that the projectile path can be found computationally by solving for the buoyant and drag force at each timestep and using the resultant velocity and acceleration vectors in Equation (1) over an interval dt. As shown in the following section, this also allows the effects of drag and buoyancy to be deduced from purely positional data and was therefore the selected method.

Section 1.2.1 Deriving Acceleration and Velocity from Position Data

Observations from the game server regarding positional data are taken once every second. To be able to accurately predict the path of a projectile, a descriptor is needed of the unknown projectile parameters that encompass the mass, density, cross sectional area and drag coefficient. Let us call this descriptor A_{proj}. To derive A_{proj} from the positional data, the velocity and acceleration of the projectile is needed at a given point. The velocity at the central observation can be found simply by subtracting the relative position vectors and dividing by the time between observations. An average of the two vectors gives a reasonable estimation. The acceleration can be found in the same way with the two calculated velocity vectors. While this method is crude, it will be shown later that it is accurate enough for the purposes of the simulation.

Without considering the effects of drag and buoyancy, the expected vectors of a ballistic projectile are:

$$\overrightarrow{v_{exp}} = \langle v_x, v_y, v_z \rangle$$

$$\overrightarrow{a_{exp}} = \langle 0, 0, -9.81 \rangle$$

Whereas the actual acceleration vector a_0 will be:

$$\overrightarrow{a_0} = \langle \frac{F_D}{m}\frac{v_x}{|v_0|}, \frac{F_D}{m}\frac{v_y}{|v_0|}, -9.81 + F_B + \frac{F_D}{m}\frac{v_z}{|v_0|} \rangle$$

$$F_D = \frac{1}{2}\rho |v|^2 C_D A$$

$$F_B = \rho g V_f$$

which considers the drag and buoyancy effects. We can solve these equations for A_{proj} independently through the x and y components of acceleration, and taking their average gives a best estimate:

$$A_{proj} = -\frac{a_{0x,y}}{v_{0_{x,y}}|v|} = \frac{1}{2}\frac{\rho C_D A}{m} \tag{5}$$

This means that equation (3), the acceleration due to drag is then simply:

$$a_{Dx,y,z} = -A_{proj} * v_{x,y,z} * |v| \tag{6}$$

This can be used to solve for the acceleration for a certain time within our time-step model. This also leads to the observation that the buoyancy can simply be represented as an effective gravitational acceleration, assuming that the density of air does not change throughout the simulation. The effective gravitational acceleration is therefore the addition of the buoyant force and g:

$$g_{eff} = -a_{0z} - A_{proj} * |v_0| * v_{0_z}$$

$$g_{eff} = -9.81 + \rho g V_f$$

These constants can now be used to model the trajectory of target projectiles using three position observations at n, and the description of motion Equation (1), for each timestep n+1 is:

$$\overrightarrow{s_{n+1}} = \overrightarrow{s_n} + \overrightarrow{v_{n+1}}(dt) + \frac{1}{2}\overrightarrow{a_{n+1}}(dt)^2 \tag{7}$$

$$\overrightarrow{v_{n+1}} = v_n + \overrightarrow{a_{n+1}}(dt) \tag{8}$$

$$\overrightarrow{a_{n+1}} = \langle -A_{proj} * (\vec{v}_{x_n,y_n,z_n} - \overrightarrow{v_{wind}}) * |\vec{v} - \vec{v}_{wind}| \rangle - \langle 0, 0, g_{eff} \rangle \tag{9}$$

Where a correcting term has been added for the wind velocity v_{wind}.

Section 2 Method – Determining Launch Solution

Now that the path of the target projectile can be modelled, it is necessary to determine the conditions of the interception projectile needed to successfully collide with the target. Of these, the largest impact on the projectile motion is the initial launch velocity [2]. This is done through a bisection search, a root-finding method which is discussed in detail in Section 2.2 Launch Velocity. The following focuses on the other projectile parameters: mass, density, cross-sectional area, and drag-coefficient.

Section 2.1 Projectile Parameters

The model developed in Section 1.2.1 is most accurate for projectiles experiencing low drag and low buoyancy. This is due to the finite time step necessary for the solve. As can be seen in Equation (7), velocity is propagated linearly with the timestep dt, while the acceleration vector (containing the components of buoyancy and drag) is propagated with the square of dt. This has the implication that a high-drag model will diverge quicker from the true path with a fixed dt, leading to a more accurate prediction for low drag projectiles. This concept is explored in depth in Section 3 Analysis – Determining Launch Solution, but for the time being, it is enough to know that for an accurate interception, the non-kinematic effects of the projectile should be reduced.

This minimization can be achieved for both buoyancy and drag as they are monotonic functions for the projectile conditions, as can be seen in Equation (3) and Equation (4).The bisection search breaks down in higher dimension searches, as only one variable can be varied [5]. This was chosen to be the velocity, as variations in its value correspond to the largest freedom for the bisection search algorithm [1]. It is

evident that a minimal coefficient of drag corresponds to the smallest drag force, therefore the smallest allowable value of 0.05 was chosen.

While not directly optimizing for velocity, the launched projectile has maximum initial kinetic energy of 5000 [J], therefore, there is a tradeoff between the speed of the initial launch and the mass of the projectile. Furthermore, an increased mass increases the cross-sectional area, which increases the drag on the projectile. Because there is a finite amount of projectile mass available, a reasonable upper cap on the limit is 5kg, which corresponds to 8 available projectiles throughout the simulation. Because we want to maximize the range of the interceptor projectile, it makes sense to plot distance as a function of mass for a 5000 [J] launch, as seen in Figure (2).

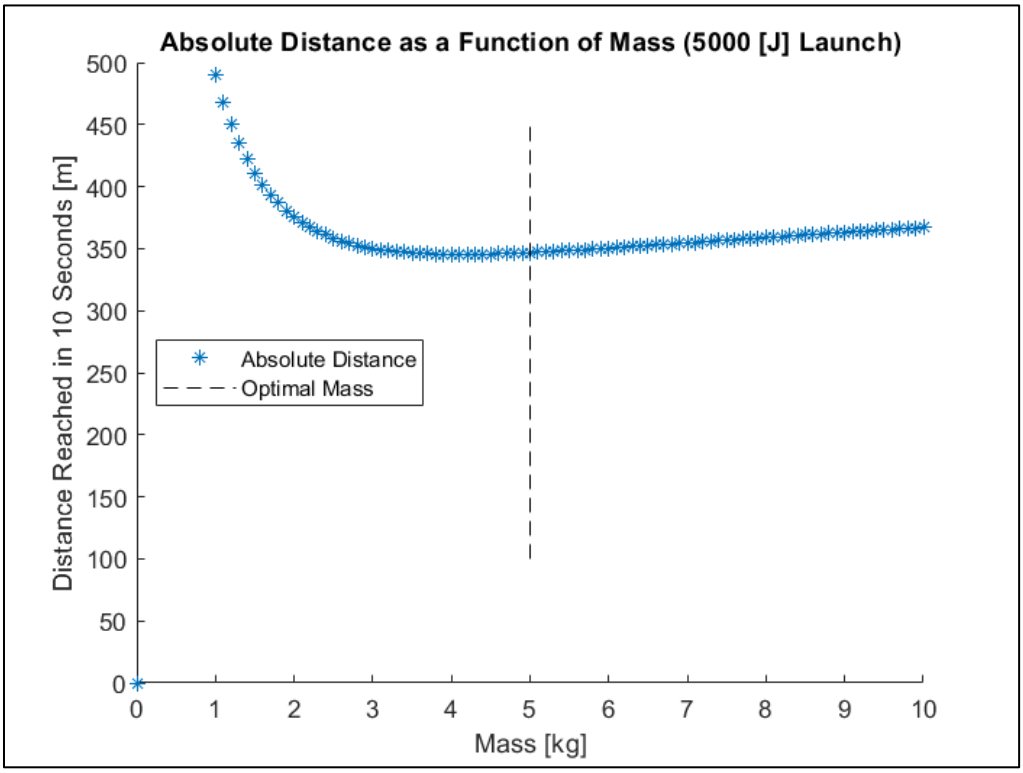

Figure 2: Absolute distance reached for a lead projectile for various masses. The optimal/chosen mass is indicated with a dashed line. The projectile path was solved computationally with a time-step of 0.01 [s].

Lead was chosen as the material of choice as it mimics the real world and is relatively dense. The shape of this curve indicates that for masses greater than 2 [kg], the absolute distance reached mostly levels out. For a spherical projectile, the cross-sectional area A is proportional to the mass m by: $A \propto m^{2/3}$, therefore, an increase in mass will decrease the drag by a larger factor than the cross-sectional area. This stipulates that we should choose the largest mass within the level area, as at that point the velocity is low enough to be reasonably efficient ($F_D \propto v^2$) and the mass is large enough to be significantly decreasing the drag. Therefore, a good choice would be the upper cap of 5 [kg], which corresponds to a cross sectional area of 0.007 [m^2] for lead. This also gives a volume of 0.0005 [m^3], which is small enough to only change the effective gravity by 0.06% away from the accepted value of 9.81 [m/s^2]. These choices minimize the non-kinematic effects of the projectile motion.

Section 2.2 Launch Velocity

To determine the launch velocity $\overrightarrow{v_0}$, a bisection search was implemented to find an intersection between a target projectile path and a projectile fired from the position of the launcher. By rewriting the intersection of two non-linear curves as a single curve intersecting the x-axis, the problem can be reduced to a root finding problem [5]. An issue that arises from using a bisection search is that an intersection point must be specified, which in the case of the simulation corresponds to some collision time t_C. Choosing this collision time depends on many factors such as the current location of the target and the velocity. The choice of this time t_C is explained in Section 3 Analysis – Determining Launch Solution

Because computations had to be completed while projectiles were detected mid-air, it was crucial that time was not wasted either when modelling a projectile or completing the bisection search. An analysis was completed on both methods to determine reasonable values for iteration.

Section 3.1 Analysis of Time-Step

The method developed in Section 1.2.1 was to write a function *modelTrajectory.m* that returned data sets of x, y, z positions at each interval dt, and a third order polynomial fit describing the path for each direction. Two methods were used to compare the relative accuracy of using a polynomial fit against a straight data set. To characterize the behavior of high and low drag projectiles, Figure 6 and Figure 7 were created, which show various projectile paths computed at values of dt. They can be found in Appendix A: Time-Step Plots. Here, it is important to notice that a low-drag projectile will converge upon a solution at a larger time-interval, while a high-drag model will require a smaller time-step and, thus more computing time to achieve an accurate result. This is the reasoning behind designing the projectile parameters in Section 2.1 Projectile Parameters to reduce the non-kinematic effects of drag-and buoyancy.

To choose a time-step, *modelTrajectory.m* was run on a dataset containing 3 azimuth and elevation data points for 84 projectiles across 3 seconds. All 84 projectiles also had a known modelled position 26 seconds after the initial data point. The returned x, y, z positions, as well as the polynomial fits for the data were evaluated at 26 seconds, and the absolute distance between the known positions and the modelled position was averaged for all 84 projectiles. This was repeated for various time-steps, and the results are plotted in Figure 3. Figure 3 shows that a time step of 0.05 is the most accurate, which was surprising considering it is quite large, and would not be computationally heavy. This meant that a value of 0.05 was chosen for the time-step.

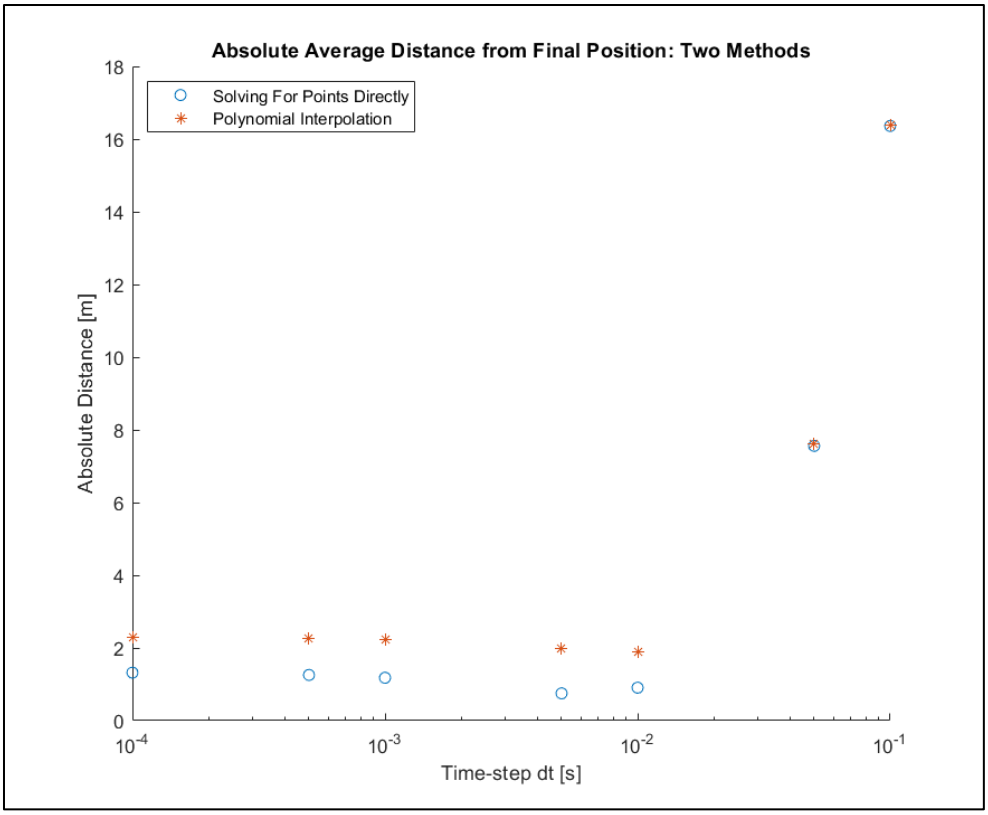

Figure 3: Average absolute distance from target projectile for various time steps, for both polynomial interpolation and direct data solve.

Section 3.2 Analysis of Bisection Search Run-Time

To analyze the effect of added accuracy on run-time, the bisection search algorithm *bisectionSearch.m* was timed and repeated for various iterations *N*. The bisection search was completed for various projectile configurations, and the values were averaged to produce the plot seen in Figure 4. The plot shows the relative accuracy of the final root velocity to the initial bisection, and the accuracy starts to level out at about 0.4 seconds of runtime, which corresponds to 12 bisections. For 12 iterations, the relative accuracy between the initial guess is 0.02%, which was too high considering it could contribute to approximately 10 [m] of uncertainty. To achieve a relative accuracy of 0.002%, or 1 [m], 16 iterations were needed which produced an average run-time of 0.6 seconds. This length was acceptable, because after the bisection search was completed the rest of the code was not computationally heavy to launch the projectile and could be completed in under 0.2 seconds. Polynomial interpolation performed worse than not interpolating the data and was this discarded.

.

The bisection search works by continuously halving an interval $[x_a, x_b]$ which contains a function $F(x)$ with a root x_0. The interval is halved, and the function is evaluated at the new location x_m. If $F(x_a)$ and $F(x_m)$ are opposite in sign, then the root x_0 must be to the left of x_m and the interval is reduced to $[x_a, x_m]$. If they are equal in sign, then the interval is reduced to $[x_m, x_b]$. This is repeated for *N* iterations until a desired accuracy is reached.

This process can be applied to a projectile search by defining $\overrightarrow{s_L}$ as the position of the launcher and $\overrightarrow{s_P}$ as the position of the target projectile at time t_C. The function $F(v)$ is defined using the model presented in Section 1.2.1 Deriving Acceleration and Velocity from Position Data, which returns the position of a theoretical projectile launched with some velocity at time t_C. The interval $[v_a, v_b]$ is defined by the velocity in the x, y, z component that corresponds to an initial kinetic energy of 5000 [J]. The midpoint of the interval v_m is evaluated in the projectile modelling function $F(v)$, and the resultant position $\overrightarrow{s_m}$ is compared with $\overrightarrow{s_P}$. The velocity interval is adjusted, and the process is repeated until $\overrightarrow{s_m}$ and $\overrightarrow{s_P}$ agree within some error, or the maximum number of iterations is reached. The error that $\overrightarrow{s_m}$ and $\overrightarrow{s_P}$ had to agree within was set as 2 [m]. This value was arrived at by looking at the average error on the modelling function $F(v)$, which was $\simeq 1$ [m]. Because this function had to be used twice, once for determining the position of the target and once for determining the position of the intercept projectile (during the bisection search), the errors were added to reach a maximum. The velocity interval was capped at ± 44.7 [m/s], which corresponds to a 5000 [J] launch.

Section 3 Analysis – Determining Launch Solution

Because computations had to be completed while projectiles were detected mid-air, it was crucial that time was not wasted either when modelling a projectile or completing the bisection search. An analysis was completed on both methods to determine reasonable values for iteration.

Section 3.1 Analysis of Time-Step

The method developed in Section 1.2.1 was to write a function *modelTrajectory.m* that returned data sets of x, y, z positions at each interval dt, and a third order polynomial fit describing the path for each direction. Two methods were used to compare the relative accuracy of using a polynomial fit against a straight data set. To characterize the behavior of high and low drag projectiles, Figure 6 and Figure 7 were created, which show various projectile paths computed at values of dt. They can be found in Appendix A: Time-Step Plots. Here, it is important to notice that a low-drag projectile will converge upon a solution at a larger time-interval, while a high-drag model will require a smaller time-step and, thus more computing time to achieve an accurate result. This is the reasoning behind designing the projectile parameters in Section 2.1 Projectile Parameters to reduce the non-kinematic effects of drag-and buoyancy.

To choose a time-step, *modelTrajectory.m* was run on a dataset containing 3 azimuth and elevation data points for 84 projectiles across 3 seconds. All 84 projectiles also had a known modelled position 26 seconds after the initial data point. The returned x, y, z positions, as well as the polynomial fits for the data were evaluated at 26 seconds, and the absolute distance between the known positions and the modelled position was averaged for all 84 projectiles. This was repeated for various time-steps, and the results are plotted in Figure 3. Figure 3 shows that a time step of 0.05 is the most accurate, which was surprising considering it is quite large, and would not be computationally heavy. This meant that a value of 0.05 was chosen for the time-step.

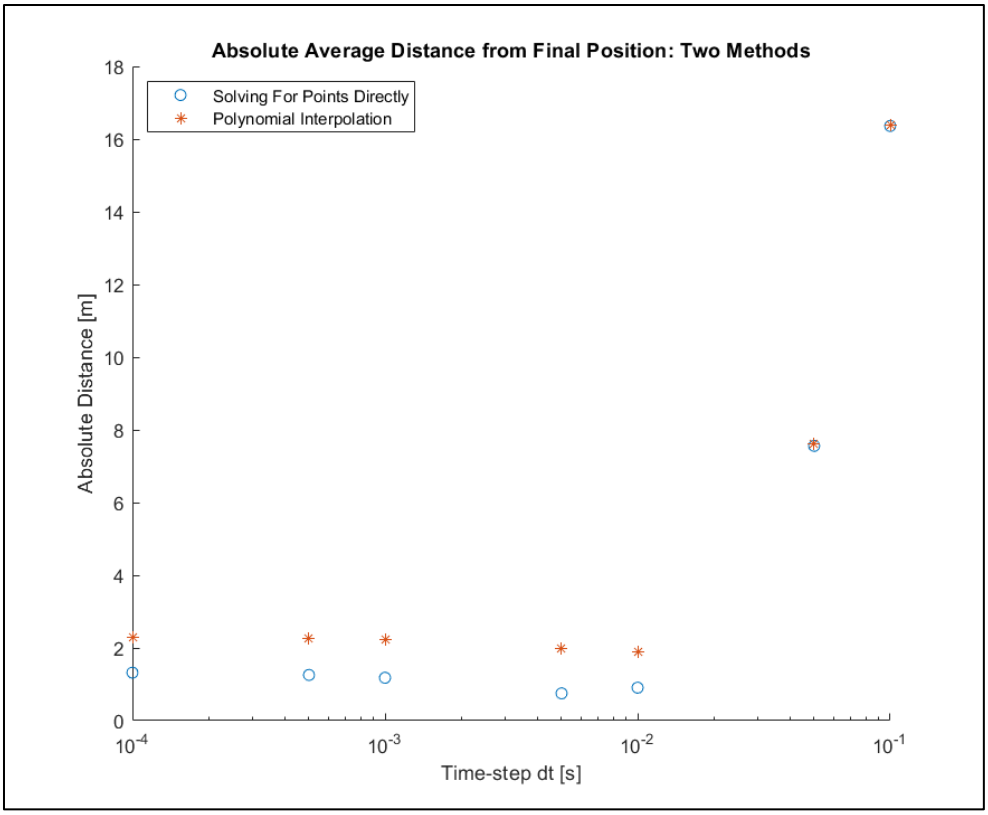

Figure 3: Average absolute distance from target projectile for various time steps, for both polynomial interpolation and direct data solve.

Section 3.2 Analysis of Bisection Search Run-Time

To analyze the effect of added accuracy on run-time, the bisection search algorithm *bisectionSearch.m* was timed and repeated for various iterations *N*. The bisection search was completed for various projectile configurations, and the values were averaged to produce the plot seen in Figure 4. The plot shows the relative accuracy of the final root velocity to the initial bisection, and the accuracy starts to level out at about 0.4 seconds of runtime, which corresponds to 12 bisections. For 12 iterations, the relative accuracy between the initial guess is 0.02%, which was too high considering it could contribute to approximately 10 [m] of uncertainty. To achieve a relative accuracy of 0.002%, or 1 [m], 16 iterations were needed which produced an average run-time of 0.6 seconds. This length was acceptable, because after the bisection search was completed the rest of the code was not computationally heavy to launch the projectile and could be completed in under 0.2 seconds. Polynomial interpolation performed worse than not interpolating the data and was this discarded.

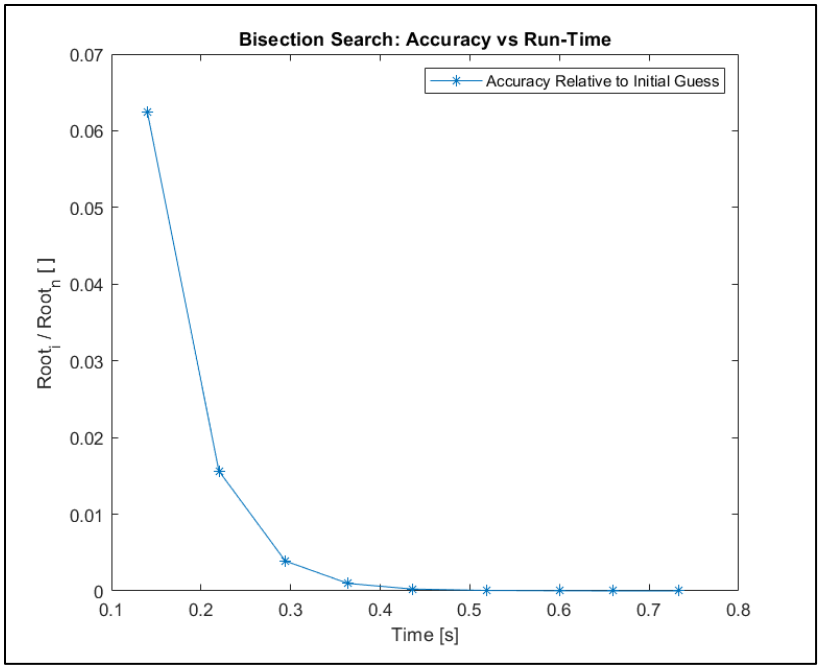

Figure 4: A plot of relative accuracy against the time taken to run bisectionSearchVelocity.m. The relative accuracy is between the initial bisection and the bisection on which the program finishes on after N iterations.

Section 4 Game Strategy and Code Design

The code was fully automated to perform everything away from human interaction and is controlled through a master function *master.m*. The decision was made to automate everything as a challenge, and to mimic real life interception programs that rely on continual observation. This function runs a continuous while loop that contains 4 stages triggered by flag variables, which call on both internal and external functions. A logical flow schematic is presented in Figure 5.

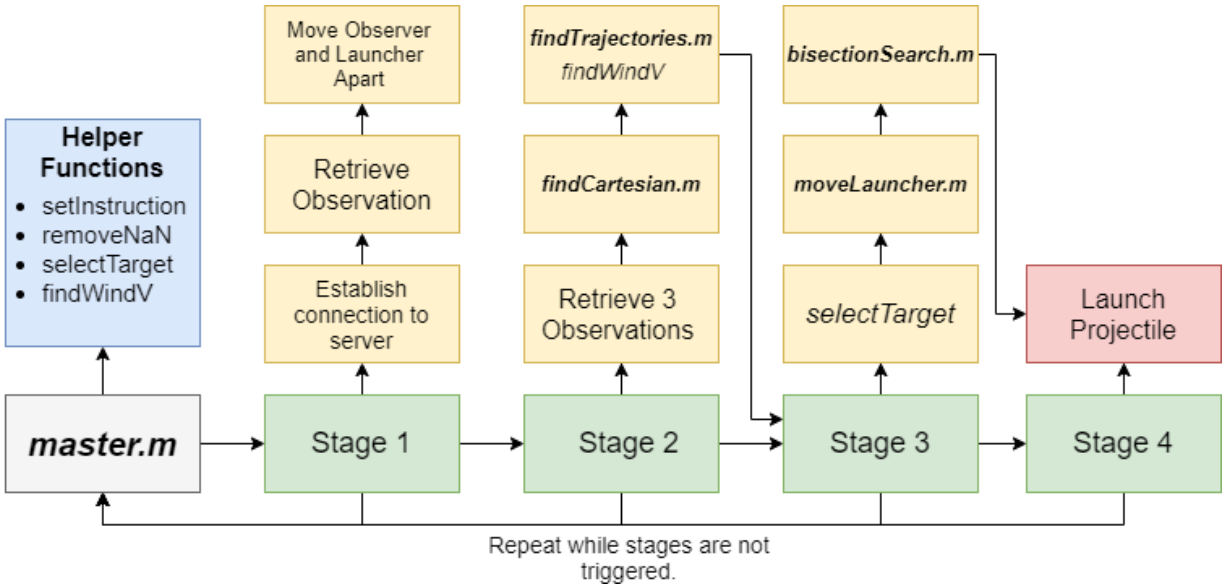

Figure 5: Software schematic of the process used to launch an interception projectile.

The first thing done in the program is to move the launcher and observer apart. This is done so that separate azimuth and elevation data from the projectiles currently in the air can be read into *findCartesian.m* and converted to (x, y, z) coordinates. After three observations spanning three seconds have been recorded, the wind velocity is computed using the observation balloons, and the trajectories of all projectiles are modelled for 10 seconds. Using these models, the target is selected, and the path is recalculated until a selected collision time t_C.

Section 4.1 Target Selection

Target selection is completed using a scoring system applied to a set of projectiles. The system was tested against a human who intuitively was choosing the best projectile to fire at and was able to agree with the human for 80% of the rounds. While this is not the best way to accomplish target selection, it factors in the three most important aspects: absolute distance from launcher, current height, and current velocity.

Consider a list of N projectiles given by the set $\{P_1, P_2, \dots, P_{N-1}, P_N\}$. The list is sorted in in ascending order by absolute distance from the launcher at 10 seconds in the model, this transformed list (A) is given by $\{P_1', P_2', \dots, P_{N-1}', P_N'\}$. At the same time, another list (B) is sorted by the current height of the projectile in the z-coordinate, given by the list $\{P_1'', P_2'', \dots, P_{N-1}'', P_N''\}$. The index position of the projectile is now counted as its rank. The initial rank is given by the index position in list A. The vertical velocity of each projectile is calculated. If the velocity if the projectile is currently upwards, then the score is given by:

$$score = rank_A - (N - rank_B) \tag{10}$$

If the velocity if the projectile is currently downwards, then the score is given by:

$$score = rank_A - rank_B \tag{11}$$

The projectile with the lowest score is chosen. Preference is given to those projectiles that are currently moving upwards and are at the earliest stage (lowest z) of their path and a scoring system ensures a projectile will always be chosen.

Section 4.2 Launcher Movement

Launcher movement is one area that needs to be further developed. Currently, after the target is selected the launcher is moved at maximum velocity towards the landing location. This can only be done for a few seconds before the bisection search takes over and the launcher must be halted. A better approach would to be start moving the launcher as soon as the first observation is taken and keep changing directions as the target is progressively narrowed down.

Section 4.3 Collision Time Selection

Without the use of an optimization function to select the best time of impact, it is difficult to properly select a time of collision t_C due to the multitude possible conditions that may be experienced. To give the interception projectile as much time as possible, a time is always chosen that is 3 seconds prior to the time when the projectile will impact the ground. Through discussion with other members working on the project, and through testing using an offline dataset, 3 seconds was usually a good choice for hitting a projectile.

Section 5 Results and Discussion

The program was unable to undergo formal completion scenarios, and due to bugs and server connection issues the code was unable to complete a full round of live testing. Due to this, a numerical table of results is not presented in this report. Instead, the components of the program were tested individually to ensure they were functioning effectively and were ready to be integrated into the program. The function *findTrajectories.m* was able to predict the path of observed projectiles consistent within 0.1-0.3 meters of the actual target, although this varied greatly by time of flight. This final discrepancy is most likely an effect of the initial velocity and acceleration calculations, which were assumed to be perfectly symmetrical about the center observation. To improve this method, it would make sense to run a bisection search on the time of validity of the first difference and the time of validity of the second difference until the simulation of the first three points yields less error. While less computationally efficient, the parameters of the projectile could more easily be determined.

Connection with the server was successfully reached, live observations were modelled, and the launcher was positioned successfully ready for launch. The largest issue in the program was the synchronization of the internal program run-time and the gamer server time. This would mean launch conditions are calculated and sent at the wrong time, and the program was not fixed in time for the completion of the report. This synchronization issue is most likely a cause of the while loop. It might make more sense to have the program work in a linear fashion, and have the user re-start the program every-time a new launch is to be made.

Target selection was tested by loading observations from the server into an offline file and running the selection algorithm to choose a target based on various locations within the game theatre. The choice was analyzed, and a point was given to the program if it made a reasonable choice. Defining reasonable was up to user but was loosely based on hand calculations that confirmed if the projectile could be reached in the given time. The *bisectionSearch.m* function was tested by integrating the code into another student's program that had a successful prototype. The function performed equally well and was able to make accurate initial velocity guesses. This was expected as the process of a bisection-search is relatively straightforward, with not much variation in the algorithm.

References

[1] B. V. Liengme, SMath for Physics, Morgan & Claypool Publishers, 2015.

[2] R. Fitzpatrick, "Projectile Motion with Air Resistance," University of Texas, 2011. [Online]. [Accessed 4 April 2020].

[3] S. Maxemow, "That's a Drag: The Effects of Drag Forces," *Undergraduate Journal of Mathematical Modelling,* vol. 2, no. 1, 2009.

[4] M.-C. H. Seungyeop Han, "Analytic Solution of Projectile Motion with Quadratic Drag and Unity Thrust," *ScienceDirect,* vol. 49, no. 17, pp. 40-45, 2016.

[5] D. F. Richard Burden, Numerical Analysis: 2.1 The Bisection Algorithm, PWS Publishers, 1985.

Appendix A: Time-Step Plots

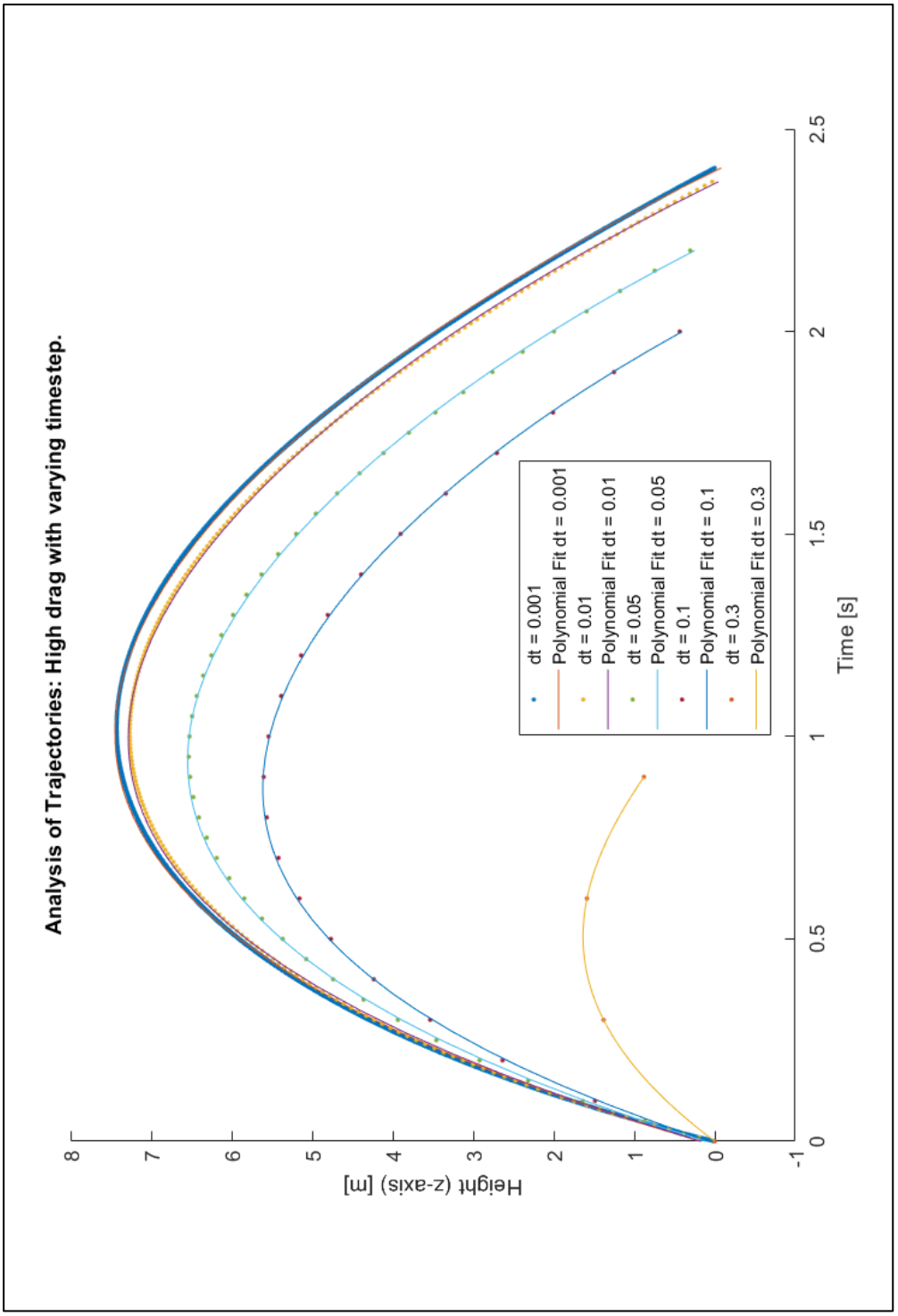

Figure 6: Flight trajectories for a projectile with high drag parameters, solved using different time-steps and the modelTrajectory.m code.

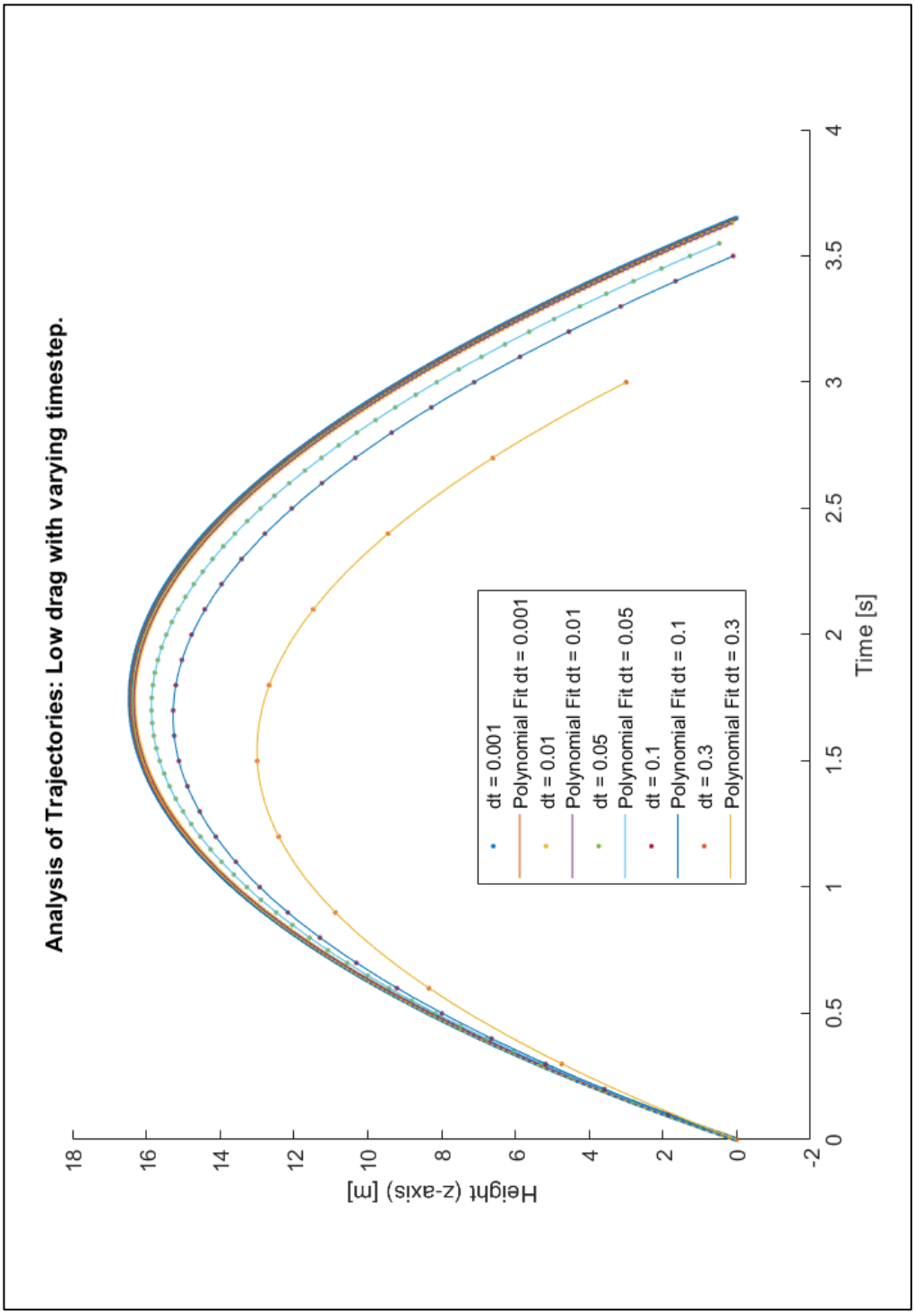

Figure 7: Flight trajectories for a projectile with low drag parameters, solved using different time-steps and the modelTrajectory.m code.

Appendix B: Projectile Prediction

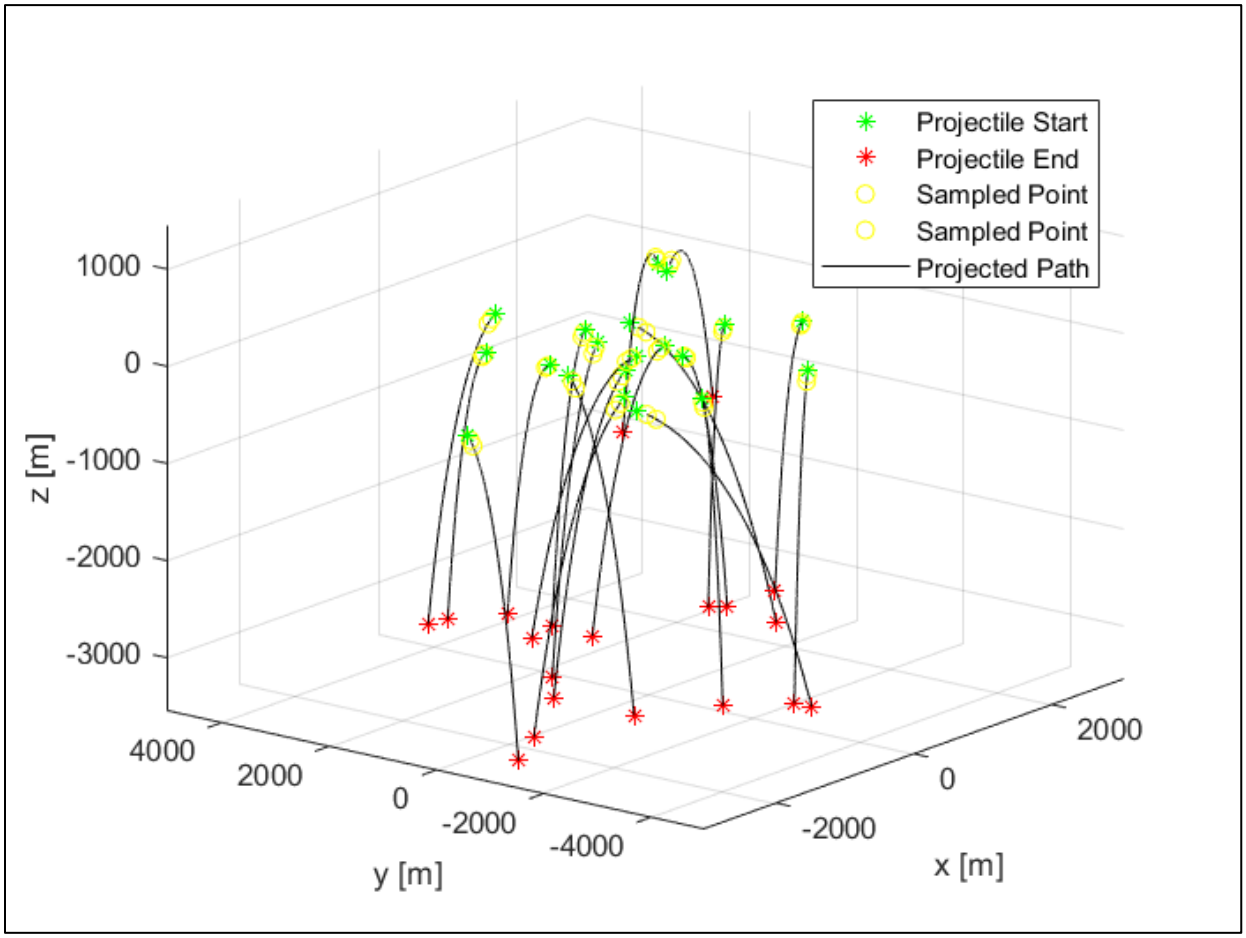

Figure 8:Modelled projectile paths plotted against sampled points, and the true of the position at 26 seconds. The average absolute distance was 0.46 [m] for this simulation.

Appendix C: Code

master.m

```
function master()
% Controls all high level operation for the succesful implementation of a
% projectile intercept system. This is to be used in conjunction with the
% ENPH 213 2020 Game Theatre Project.
%
% Written By: Csaba Nemeth
% Version Since: 2020

    %Always true -> Used to keep the while loop running until exited by
    %user.
    game_is_live = true;

    %--Stage Flags--
    %Stage 1 -> Move observer and launcher vehicles apart.
    stage1 = true;
    %Stage 2 -> Retrieve three observations, use observations to calculate
    %the wind speed and model the paths for 10 seconds.
    stage2 = false;
    collected_data = 0;
    %Stage 3 -> Select a target projectile and remodel the target until
    %ground impact point. Move the launcher towards the impact point and
    %complete a bisection search to find the initial veloecity.
    stage3 = false;
    %Stage 4 -> Launch the projectile.
    stage4 = false;

    %Initalize ID for Server Connection
    SERVER.id_number = 20090753;
    SERVER.net_id = '17cmn  ';
    %Retrieve Address
    SERVER.address = get_game_server_address;
    %Initialize Connection to Game Server
    [SERVER] = game_client(SERVER, 1);
    %Collect the inital observation
    [SERVER, OBSERVATION] = game_client(SERVER, 2);

    %Initialize game-time loop
    while game_is_live == true

        %STAGE 1
        if (stage1 == true)
            disp("Stage 1")

            %Collect an observation
            [SERVER, OBSERVATION] = game_client(SERVER, 2);

            %Retrieve the current positions of vehicles
            obs_pos = transpose(OBSERVATION.pos_launch);
            laun_pos = transpose(OBSERVATION.pos_observ);

            %If the position of the vehicles is the same, move the
```

```
            %vehicles apart.
            if isequal(obs_pos, laun_pos)
                %Send instruction to move the observer.
                instruction = setInstruction(NaN, [10, 10], NaN, NaN, NaN,
NaN, NaN, NaN);
                game_client(SERVER, 3, instruction);
            else
                %Send instruction to stop the observer.
                instruction = setInstruction(NaN, [0, 0], NaN, NaN, NaN, NaN,
NaN, NaN);
                game_client(SERVER, 3, instruction);

                %Display the final vehicle locations and confirm
                %completion of stage.
                disp("Stage 1 has been completed. The final vehicle locations
are:");
                disp(obs_pos);
                disp(laun_pos);

                %Set flag variables to next stage.
                stage1 = false;
                stage2 = true;

                %Break to next iteration of the while loop.
                continue;
            end
        end %END STAGE 1

        %STAGE 2
        if (stage2 == true)
            disp("Stage 2")
            disp(collected_data);
            if (collected_data < 3)
                %Collect an observation
                [SERVER, OBSERVATION] = game_client(SERVER, 2);

                %Concatenate azimuth and elevation data
                [az_observ, indices, onlyBalloons] =
removeNaN(OBSERVATION.az_observ);
                azimuths = [az_observ, removeNaN(OBSERVATION.az_launch)];
                elevations = [removeNaN(OBSERVATION.el_observ),
removeNaN(OBSERVATION.el_launch)];

                %Convert the azimuth and elevation data to cartesian
                [X, Y, Z] = findCartesian(azimuths, elevations, obs_pos,
laun_pos);

                %Switch stament to collect three sets of data.
                switch collected_data
                    case 0
                        %Save the observation into the first time-slot.
                        coords1 = [X, Y, Z];
                        collected_data = 1;
                        disp("First Collection");
                    case 1
                        disp("Second Collection");
                        %Save the observation into the second time-slot.
```

```
                        coords2 = [X, Y, Z];

                        %Reset collection if the amount of projectiles
                        %chnages throughout the collection.
                        if (isequal(size(coords1),size(coords2)) == 0)
                            collected_data = 0;
                            disp("Case 1 Not Equal");
                            continue;
                        end
                        collected_data = 2;

                    case 2
                        disp("Third Collection");
                        %Save the observation into the last time-slot.
                        coords3 = [X, Y, Z];

                        %Reset collection if the amount of projectiles
                        %changes throughout the collection.
                        if (isequal(size(coords2),size(coords3)) == 0)
                            collected_data = 0;
                            disp("Case 2 Not Equal");
                            continue;
                        end
                        collected_data = 3;
                        continue;
                end %End data collection switch statement.

            else
                if (collected_data == 3)
                    disp("Analyzing Data");
                    if ( isequal(size(coords1), size(coords2), size(coords3))
== 1 )
                        %This if statement should always be true.

                        %Use helper function to calculate the wind-speed.
                        wind = findWindV(coords2, coords1);

                        %Create single matrix holding all of the
                        %observations.
                        coords = cat(3, coords1, coords2, coords3);

                        %Predict the PATHS of all projectiles for the next
                        %10 seconds.
                        [PATHS, ~] = findTrajectories(coords, 0.005,
OBSERVATION.game_time , OBSERVATION.game_time + 10, wind);
                        stage2 = false;

                        %Initialize stage 3.
                        stage3 = true;
                    else
                        collected_data = 0;
                        continue;
                    end
                else
                    continue;
                end
            end
```

```
        end %END STAGE 2

        %STAGE 3
        if (stage3 == true)

            %Select the target projectile and read into array.
            [target_index, game_target_index] = selectTarget(PATHS, laun_pos,
indices, onlyBalloons);
            target_coords = coords(target_index, :, :);

            %Remodel the projectile data until impact point on ground.
            [target_path, time_of_impact] = findTrajectories(target_coords,
0.005, OBSERVATION.game_time, 0, wind);

            %Move the launcher towards the target impact point.
            target_position = [target_path(1, end, 1), target_path(2, end,
1)];
            [launcher_vx, launcher_vy] = moveLauncher(target_position,
laun_pos);

            %Send instruction to move launcher.
            instruction = setInstruction([launcher_vx, launcher_vy], NaN,
NaN, NaN, NaN, NaN, NaN, NaN);
            game_client(SERVER, 3, instruction);

            %Wait for 3 seconds to allow movement of launcher.
            pause(3);

            %Send instruction to stop the launcer.
            instruction = setInstruction(NaN, NaN, NaN, NaN, NaN, NaN, NaN,
NaN);
            game_client(SERVER, 3, instruction);

            %Retrieve location of target at chosen time.
            target_x = target_path(1, end - floor(3/0.005),1);
            target_y = target_path(2, end - floor(3/0.005),1);
            target_z = target_path(3, end - floor(3/0.005),1);

            %Run bisection search on target path for that time.
            [proj_g, proj_A] = getAandG();
            target_location = [target_x, target_y, target_z];
            launch_velocity = bisectionSearch(target_location, laun_pos,
OBSERVATION.game_time,...
                                OBSERVATION.game_time + time_of_impact - 3,
proj_g, proj_A, wind);
            %Run next stage
            stage3 = false;
            stage4 = true;

        end %END STAGE 3

        %STAGE 4
        if (stage4 == true)
            instruction = setInstruction(NaN, NaN, launch_velocity, NaN,
time_of_impact, game_target_index, proj_g, proj_A);
            game_client(SERVER, 3, instruction);
```

```
            %Restart detection and launch sequence.
            stage4 = false;
            stage1 = true;

        end %END STAGE 4
    end %END MASTER WHILE LOOP
end %End Function master(..)

%HELPER FUNCTIONS

%Classifies and cleans the obersvation data. Removes NaN values, finds the
%correct indices and specifies if only balloons exist in the simulation.
function [arr_out, indices, onlyBalloons] = removeNaN(arr_in)

 onlyBalloons = true; %boolean
 j = 1;
 indices = zeros(1, length(arr_in));

 %Loop through inputed array.
 for i = 1:size(arr_in, 1)
     if isnan(arr_in(i)) == 0
         indices(j) = i;
         j = j + 1;
         onlyBalloons = false;
     end
 end

 %Remove NaN values and reutrn specified indices.
 arr_out = arr_in(~isnan(arr_in));
 indices = indices(1:j);

end %End function.

%Returns A-coefficiant and the effective gravitational constant of the
%interceptor projectile. These justification for these values is found in
%the final report.
function [g, a] = getAandG()

%Define projectile constants.
CD = 0.05;
mass = 5;
area = 0.007;
volume = 0.0005;

%Calculate a and g values.
a = (1/2)*1.225*CD*area/mass;
g = 9.81*(1 - 1.225*volume/mass);

end %End Function

%Sets and returns an instruction object based on input values. If the
%instriction is not specified (NaN value specified) it is defualted to 0
value.
function instruction = setInstruction(vel_launch, vel_observ, vel_projectile,
next, t, c, g, a)

%Initialize instruction object with defualts.
```

```
instruction.vel_launch = [0, 0];
instruction.vel_observ = [0, 0];
instruction.vel_projectile = [0, 0, 0];
instruction.t = 0;
instruction.c = 0;
instruction.g = 0;
instruction.a = 0;

%Change instruction if the input value exists.
if ~isnan(vel_launch)
    instruction.vel_launch = vel_launch;
end
if ~isnan(vel_observ)
    instruction.vel_observ = vel_observ;
end
if ~isnan(vel_projectile)
    instruction.vel_projectile = vel_projectile;
end
if ~isnan(next)
    instruction.next = next;
end
if ~isnan(t)
    instruction.t = t;
end
if ~isnan(c)
    instruction.c = c;
end
if ~isnan(g)
    instruction.g = g;
end
if ~isnan(a)
    instruction.a = a;
end

end %End function.

%Finds the wind velocity assuming the last coordinate describes a balloon.
function wind = findWindV(coords1, coords2)

%Subtract positional data across one seocnd to find x, y componenet.
windX = coords2(end, 1) - coords1(end, 1);
windY = coords2(end, 2) - coords1(end, 2);
wind = [windX, windY];

end %End function.

%Chooses the target projectile by sorting the projected distances, heights
%and velocities. Returns the best projectile as an index in the input
%matrix paths, and also returns the corresponding game_index. If only
%balloons exist, then a balloon is chosen for the target.
function [best_index, game_index] = selectTarget(paths, launcher_position,
indices, onlyBalloons)

    %Initialize variables
    closest_absolute_distance = 10000;
    best_index = 1;
    game_index = 1;
```

```
    %Choose a balloon based on absolute distance to current observer.
    if (onlyBalloons == true)
        for i = 1:size(paths, 3)

            %Retrieve x, y, and z position at 10 seconds into flight of
            %balloon.
            x = paths(1, end, i);
            y = paths(2, end, i);
            z = paths(3, end, i);

            %Calculate absolute distance from launcher.
            distance = sqrt((x - launcher_position(1))^2 + (y -
launcher_position(2))^2 + (z)^2);

            %If the current distance is better than previously saved, save
the current
            %indices.
            if (distance < closest_absolute_distance)
                best_index = i;
                game_index = indices(i);
                closest_absolute_distance = distance;
            end
        end
    else %If other targets exist other than balloons...
        %Loop through all targets except for the ballons

        %If there is only one projectile, choose that one.
        number_of_targets = (size(paths, 3) - 5);
        if ( number_of_targets == 1 )
            game_index = indices(1);
            best_index = 1;
            return;
        end

        %If there are more projectiles, loop through them. Want to choose
        %projectiles that are currently moving upwards, at the currently
        %lowest z position with the closest absolute distance.

        %An array that contains a 1 if the projectile is currently
        %moving upwards and a 0 if it is not.
        going_upwards = zeros(1, number_of_targets);

        %The current z position of the projectile.
        z_positions = zeros(1, number_of_targets);

        %The absolute distance from the projectile to the launcher
        %after a projected 10 seconds.
        absolute_distance = zeros(1, number_of_targets);

        %Start loop.
        for i = 1:number_of_targets

            %If velocity upwards in z at observation time, save into array
            if (paths(3,3,i) - paths(3, 2, i)) > 0
                going_upwards(i) = 1;
            end
```

```
            %Save current z position
            z_positions(i) = paths(3, 1, i);

            %Save absolute distance at t = 10s
            x = paths(1, end, i);
            y = paths(2, end, i);
            z = paths(3, end, i);
            absolute_distance(i) = sqrt((x - launcher_position(1))^2 + (y -
launcher_position(2))^2 + (z)^2);

        end

        %Declare sorted index arrays
        sorted_indices_absolute_distance = 1:1:number_of_targets;
        sorted_indices_height = 1:1:number_of_targets;

        %Sort indices by absolute distance from launcher, the closer, the
        %more towards the front the projectile is. Also sort by current
        %height.
        sorted_absolute_distance = sort(absolute_distance);
        sorted_z_position = sort(z_positions);

        %Save indices of projectiles in the order of the sort.
        for i = 1:length(absolute_distance)
            sorted_indices_absolute_distance(i) =
find(sorted_absolute_distance == absolute_distance(i));
            sorted_indices_height(i) = find(sorted_z_position ==
z_positions(i));
        end

        %Apply scoring based on z velocity and height ranking.
        scores = 1:number_of_targets;
        for i = 1:number_of_targets
            if going_upwards(i) == 1
                scores(i) = scores(i) - (number_of_targets -
find(sorted_indices_height == i));
            else
                scores(i) = scores(i) - find(sorted_indices_height == i);
            end
        end

        %Compute final best targets.
        winning_index = min(scores);
        best_index = sorted_indices_absolute_distance(winning_index);
        game_index = indices(best_index);
    end
end %End function
```

findCartesian.m

```
function [x, y, z] = findCartesian(az, el, pos_observer, pos_launcher)
% [x, y, z] = findCartesian(az, el, pos_observer, pos_launcher)
% Written By: Csaba Nemeth
% Version: 1.2 Since 2020-03-03
%
% Takes a 2D array of azimuth data observations for two locations, and a 2D
% array of elevation observations for both locatations. These two locations
% are defined by pos_observer and pos_launcher respectively. These are
% arrays in cartesian coordinates of (x, y).

    %Import observer and azimuth data.
    az_observer = az(:,1);
    az_launcher = az(:,2);
    el_observer = el(:,1);

    %Find the slopes of the observer and launcher data points.
    m_observer = tan(az_observer);
    m_launcher = tan(az_launcher);

    %Find the intercept of the slopes in the x,y plane.
    b_observer = pos_observer(2) - m_observer .* pos_observer(1);
    b_launcher = pos_launcher(2) - m_launcher .* pos_launcher(1);

    %Find the intersection of lines in cartesian coordinates.
    x = (b_observer - b_launcher)./(m_launcher - m_observer);
    y = m_observer .* x + b_observer;

    %Find the z locations
    rxy_observer = sqrt((x - pos_observer(1)).^2 + (y - pos_observer(2)).^2);
    z = tan(el_observer).*rxy_observer;

end %End Function findCartesian(az, el, pos_observer, pos_launcher)
```

modelTrajectory.m

```
function [data_sx, data_sy, data_sz, start_time, collision_time] = ...
    modelTrajectory(pos_0, vel_0, start_time, collision_time, dt, A_coeff, g,
wind)
%Written By: Csaba Nemeth
%Version: 2.0 since 2020-03-27
%
%Models the the trajectory of the specified projectile and returns three
%data sets containing the x, y, z position at each interval of dt. Also
%returns the start time and the collision time, if no collision time is
%pre-specified.
%
%Takes inputs the inital position pos_0, the inital velocity vel_0, the
%start time, the collision time (if specified as 0 then the trajectory is
%computed until the ground is hit), the desired timestep dt, the A_coeff of
%the projectile
%(0.5*density_air*drag_coefficient*frontal_area/mass_projectile), the
%effective gravity g (9.81* (1 -
density_air*volume_projectile/mass_projectile)
%and a vector defining the current wind.

%Import initial conditions into vectors.
s_vector = pos_0;
v_vector = vel_0;

%Initialize arrays to store data. The assumption is that 1000 points will
%be enough using the current time step.
data_sx = zeros(1, 10000);
data_sy = zeros(1, 10000);
data_sz = zeros(1, 10000);
T = zeros(1, 1000);

%Loop across the time interval. Terminating condition is either the passed
%time of collision or the moment that the ground is hit.
index = 1;
if collision_time ~= 0
    for time = start_time:dt:collision_time
        %Append current time to array
        T(index) = time;

        %Append positions to arrays
        data_sx(index) = s_vector(1);
        data_sy(index) = s_vector(2);
        data_sz(index) = s_vector(3);

        %Update Values
        [v_vector, a_vector] = getNewValues(v_vector, wind, g, A_coeff, dt);
        s_vector = s_vector + v_vector.*dt + (1/2).*a_vector.*(dt.^2);

        %Update index value
        index = index + 1;

    end %End find posiitons loop.
else
```

```
    time = start_time;
    while s_vector(3) >= 0
        %Append current time to array
        T(index) = time;

        %Append positions to arrays
        data_sx(index) = s_vector(1);
        data_sy(index) = s_vector(2);
        data_sz(index) = s_vector(3);

        %Update Values
        [v_vector, a_vector] = getNewValues(v_vector, wind, g, A_coeff, dt);
        s_vector = s_vector + v_vector.*dt + (1/2).*a_vector.*(dt.^2);

        %Update index value
        index = index + 1;

        %Update time step
        time = time + dt;
    end
    collision_time = floor(time);
end

%Trim trailing zeros on data.
data_sx = data_sx(1:find(data_sx,1,'last'));
data_sy = data_sy(1:find(data_sy,1,'last'));
data_sz = data_sz(1:find(data_sz,1,'last'));

end %End function modelTrajectory(..)

%Helper function to update the velocity and acceleration values.
function [velocity, acceleration] = getNewValues(velocity, wind, g, A_coeff,
dt)

%Correct velocity based on wind
v_corrected = [velocity(1) - wind(1), velocity(2) - wind(2), velocity(3)];

%Update acceleration value
acceleration = getAcceleration(v_corrected, g, A_coeff);

%Update velocities
velocity = velocity + acceleration.*dt;

end %End function getNewValues(..)

%Helper function to find the current acceleration
function acceleration = getAcceleration(velocity, g, A_coeff)

%Construction acceleration vector.
acceleration = [0, 0, 0];

%Find magnitude of the velocity.
v_mag = sqrt(sum(velocity.^2));

%Compute the acceleration due to drag.
drag_acceleration = findDragAccel(A_coeff, v_mag);
```

```
%Update the acceleration vector with the drag components and g.
acceleration(1) = - drag_acceleration.*velocity(1)./v_mag;
acceleration(2) = - drag_acceleration.*velocity(2)./v_mag;
acceleration(3) = - g - drag_acceleration.*velocity(3)./v_mag;

end %End function getAcceleration(..)

%Helper function to find the drag due to acceleration.
function acceleration = findDragAccel(A_coeff, v_mag)
%Compute using the A_coeff coefficiant.
acceleration = A_coeff.*v_mag.^2;
end %End function findDragAccel(..)
```

findTrajectories.m

```
function [PATHS, collision_time] = findTrajectories(coords, dt, start_time,
end_time, wind)
% Written By: Csaba Nemeth
% Version: 2.1 since 2020-03-27
%
% Takes a 3D array coords of the form:
% -> Each row represents a projectile in motion;
% -> Each column representes x, y, z location at indices 1, 2, 3
% -> Each page represents the same projectile information 1 second later
%    than the previous page.
%
% RETURNS:
%
% PATHS -> A 3D-Matrix where:
% x| x(start_time), x(start_time + dt),..., x(end_time)
% y|                          ...
% z|                          ...
% And each page represents a seperate projectile.
%
% FUNCTIONS -> A 3D-Cell Array where:
% x| fx(t) -> A third order polynomial
% y| fy(t) -> A third order polynomial
% z| fz(t) -> A third order polynomial
% And each page represents a seperate projectile.

    %Create Matrix to hold Functions. If no end time is given, then only
    %one projectile is being tracked until impact.
    if (end_time ~= 0)
        num_points = length(start_time:dt:end_time);
        PATHS = zeros(3, num_points, 100);
    else
        PATHS = zeros(3, 10000, 1);
    end

    %Loop through all the projectile data
    for projectile = 1:size(coords,1)

            %Read points at time t = 1, 2, 3 into arrays
            x = coords(projectile, 1, :);
            y = coords(projectile, 2, :);
            z = coords(projectile, 3, :);

            %Calculate the difference between points to find the
            %approximate velocity.
            v1 = [x(2) - x(1), y(2) - y(1), z(2) - z(1)];
            v2 = [x(3) - x(2), y(3) - y(2), z(3) - z(2)];

            %Average of these velocities is the approximate velocity at the
            %center measurement.
            velocity = (v1+v2)./2;

            %Calculate the acceleration as the difference between the
            %velociites.
```

```
            accel = v2 - v1;

            %Find magnitude of velocity at that instant:
            velocity_mag = sqrt(sum(velocity.^2));

            %Use both x and y accelerations to solve for the A coefficiant.
            A_coeff_x = -accel(1)/(velocity(1)*velocity_mag);
            A_coeff_y = -accel(2)/(velocity(2)*velocity_mag);

            %Average indiviudal components to achieve a best estimate of
            %the A coefficiant
            A_coeff = abs((1/2)*(A_coeff_x + A_coeff_y));

            %Solve for apparent acceleration due to gravity. Add a
            %correction factor of 0.002 to bring the estimate closer.
            g = -accel(3) - A_coeff.*velocity_mag.*velocity(3) + 0.002;

            %Call modelTrajectory to find the trajectory of the targets.
            [x_data, y_data, z_data, ~, collision_time] = ...
                modelTrajectory([x(2), y(2), z(2)], velocity, start_time,
end_time, dt, A_coeff, g, wind);

            %Save points into the 3d Array.
            PATHS(1,:,projectile) = x_data;
            PATHS(2,:,projectile) = y_data;
            PATHS(3,:,projectile) = z_data;

    end %End For Loop

end %End Function findTrajectories(coords)
```

moveLauncher.m

```
function velocity = moveLauncher(targetPosition, currentPosition)
%Target VELOCITY velocity = moveLauncher(targetPosition, currentPosition)
%Written By: Csaba Nemeth
%Version Since: 2020-04-05
%
% Returns the maximum velocity of the launcher ot move in the direction of
% the target position. Takes an input of the current position
% currentPosition and the target position targetPosition. Returns the
% velocity vector velocity.

%Find direction and declare maximum total velocty.
direction_vector = targetPosition - currentPosition;

%Keep adding velocities in the direction until a total velocity of 30 m/s
%is reached.
for vel = 1:30
    v = vel.*direction_vector;
    v_mag = sqrt(sum(v.^2));
    if (v_mag > 29)
        break;
    end
end

%Set final velocity.
velocity = v;

end
```

bisectionSearch.m

```
function velocity = bisectionSearch(pos_target, pos_launcher, t_initial,
t_collision, g, A_coeff, wind)
%TARGET VELOCITY velocity = bisectionSearch(pos_target, pos_launcher,
t_initial, t_collision, g, A_coeff, wind)
%Written By: Csaba Nemeth
%Version Since: 2020-04-05
%
% Uses a bisection search to find the initial velocity needed to intersect
% a position pa_target at a time t_collision. The function takes an input of
% the target position pos_target, the position of the launcher
% pos_launcher, the initial time t_initial, the desired time of collision
% t_collision, the effective gravity g, the A coefficiant of the launch
% projectile and the wind. The Function completes 16 iterations for an
% absolute accuracy of 0.002%.

%Define anonymous function as modelTrajectory.m
F = @(V) modelTrajectory(pos_launcher, V, t_initial, t_collision, A_coeff, g,
wind);

%Declare solution.
velocity = zeros(1, 3);
V_a_init = zeros(1, 3);
V_b_init = zeros(1, 3);

%Set the inital boundary on condition for the interval. The velocity is the
%calculaed velocity for a maximum 5000 J launch.
for i = 1:3
    if pos_target(i) - pos_launhcer(i) > 0
        V_a_init(i) = 44.7;
        V_b_init(i) = 0;
    else
        V_a_init(i) = 0;
        V_b_init(i) = -44.7;
    end
end

%Complete initial bisection, here we force the first bisection to occur for
%a velocity of 5 m/s. This is an arbitrary value, but for safety, do not
%want to pass 0 m/s to modelTrajectory.
V_m_init = 5;

%Run bisection search for x, y, z, coordinates.
for dim = 1:3

    %Set current interval and bisection as starting values.
    V_a = V_a_init;
    V_b = V_b_init;
    V_m = V_m_init;

    for N = 1:16
        value_a = F(V_a).*F(V_m);
        value_b = F(V_b).*F(V_b);
```

```
        %Check bisection and adjust [a, b] interval.
        if (value_a(dim) > 0)
            V_a = V_m;
            V_m = (V_a + V_b)/2;
        elseif (value_b(dim) > 0)
            V_b = V_m;
            V_m = (V_a + V_b)/2;
        else
            V_a = V_m;
            V_b = V_m;
            break;
        end
    end
    %Set velocity solution to current bisection.
    velocity(dim) = V_m(dim);
end
```