Save a City from an Asteroid

EEE4119F: Mechatronics

Milestone 4 Report

Author: Safiya Mia

Student Number: MXXSAF002

May 2025

Contents

1 Introduction

In the not-so-distant future, an asteroid was hurtling toward Earth with the potential to annihilate an entire city. With limited time and resources, humanity had only one shot: to launch an interceptor rocket capable of precisely striking and neutralising the threat. The stakes of this task were high: failure meant death and destruction (and not meeting a GA requirement); success meant salvation for planet Earth and all its inhabitants!
This report presents the denouement of a three-part project involving the modelling, simulation, and control of a rocket tasked with intercepting a tumbling asteroid. Across the milestones, the complexity of the challenge increased significantly, with each phase targeting specific learning and technical outcomes:

- **Milestone 1** focused on physical modelling. The rocket's dynamics were derived using symbolic methods in MATLAB, including thrust, gravity, and torque due to thrust offset. The asteroid was modelled under gravity and non-linear air resistance. The asteroid's drag coefficient was also estimated from noisy simulation data.
- **Milestone 2** introduced control scenarios. The rocket had to detonate within 150 meters of the asteroid's centre of mass, before the asteroid descended below a critical altitude threshold (200m). One scenario permitted vertical launch, while the second required trajectory prediction and interception due to a displaced asteroid path.
- **Milestone 3** added an angular constraint: the rocket had to strike within a $\pm 30°$ cone on either the front or rear face of the asteroid, relative to its orientation. This required precise control of both the approach trajectory and impact angle. Additionally, the asteroid began with a non-zero angular velocity, further complicating timing and control.

Testing and validation were conducted in simulation, under varying initial conditions and dynamic profiles (randomised seeds), to evaluate performance and reliability.

2 Milestone 1

2.1 Rocket Modelling

My modelling process began by identifying the rocket's motion in terms of three generalised coordinates: its position in the x and y directions, and its orientation angle about the z-axis. These were defined symbolically in MATLAB (see Listing 26).
To account for the rocket's thrust vectoring, I needed to transform vectors between the body frame (attached to the rocket) and the inertial frame (fixed to the ground). I created a rotation matrix function about the z-axis (see Listing 27). The body-to-inertial transformation was applied using:

Listing 1: Assigning body-to-inertial rotation matrices

```
Ri_b = RotZ(pth);
Rb_i = transpose(Ri_b);
```

The next step was to define where the thrust force is applied. Since the thruster is offset from the rocket's center of mass, it induces both translational and rotational motion. The offset position in the body frame was defined as:

Listing 2: Position of force application in body frame

```
p_F_b = [0; -OFFSET; 0];
```

I then converted this point to the inertial frame by rotating it and adding it to the rocket's center of mass:

Listing 3: Position of rocket COM and thrust point in inertial frame

```
p_COM_i = [px; py; 0];
p_F_i = p_COM_i + Rb_i * p_F_b;
```

The thrust force itself was applied at an angle α in the body frame, with a symbolic magnitude F. The force vector was defined and rotated into the inertial frame:

Listing 4: Thrust vector in body and inertial frames

```
syms F alph
F_vec_b = [-F*sin(alph); F*cos(alph); 0];
F_vec_i = Rb_i * F_vec_b;
```

With the physical setup complete, I computed the linear velocity of the center of mass and the angular velocity of the rocket. These were used to calculate the kinetic energy:

Listing 5: Linear and angular velocities, and kinetic energy

```
v = simplify(jacobian(p_COM_i, q) * dq);
w = [0; 0; dpth];
T = simplify(0.5*MASS*transpose(v)*v + 0.5*I*transpose(w)*w);
```

The potential energy was derived directly from the vertical position of the rocket:

Listing 6: Potential energy from vertical position

```
V = simplify(MASS * g * p_COM_i(2));
```

From here, I constructed the system matrices that form the manipulator equation. The mass matrix $M(q)$ was computed as the Hessian of the kinetic energy with respect to $\dot{q}$:

Listing 7: Computation of mass matrix from kinetic energy

```
M = simplify(hessian(T, dq));
```

I then derived the Coriolis matrix $C(q, \dot{q})$ using the time derivatives of $M(q)$ (see Listing 28) and the Jacobian of T:

Listing 8: Computation of Coriolis and centrifugal matrix

```
C = simplify(dM * dq - transpose(jacobian(T, q)));
```

The gravity vector was computed directly from the potential energy:

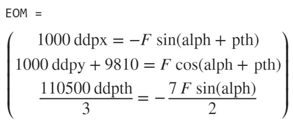
Listing 9: Gravity vector from potential energy gradient

```
G = simplify(jacobian(V, q));
```

Lastly, I computed the generalised force vector Q by projecting the thrust force along the directions of the generalised coordinates (Code viewable in Listing 29).
All components were brought together to form the final manipulator equation of motion:

$$M(q)\ddot{q} + C(q, \dot{q}) + G(q)^T = Q$$

The following result was obtained after running the script:



```
EOM =
```

$$\begin{pmatrix} 1000\,\mathrm{ddpx} = -F\,\sin(\mathrm{alph} + \mathrm{pth}) \\ 1000\,\mathrm{ddpy} + 9810 = F\,\cos(\mathrm{alph} + \mathrm{pth}) \\ \frac{110500\,\mathrm{ddpth}}{3} = -\frac{7\,F\,\sin(\mathrm{alph})}{2} \end{pmatrix}$$

Figure 1: Final Manipulator Equation Matrix

This derivation confirmed that the model accurately captured the interaction between thrust direction, offset torque, and gravitational effects. The resulting equations provided a solid analytical foundation for trajectory planning and controller design in later stages of the project.

2.2 Asteroid Modelling

The asteroid model was based on a realistic physical scenario where the body falls toward Earth under the influence of gravity and nonlinear air resistance. The first step was to simulate the asteroid's motion using the provided Simulink file. This simulation generated velocity data in the x and y directions ($\dot{x}$ and $\dot{y}$) and a corresponding time vector.
To estimate the asteroid's drag coefficient c, I needed to compute the drag force, which required knowledge of the asteroid's acceleration. Since only velocity data were available, I used finite differencing to approximate acceleration (see Listing 30).
Due to noise in the simulation, direct differentiation of velocity introduced significant fluctuations in the acceleration data. To mitigate this, I applied a 1D median filter with a window size of 50 using MATLAB's `medfilt1` function. This approach replaces each point with the median of 50 neighbouring points (25 before and 25 after), effectively smoothing out transient spikes while preserving the shape of the curve.

Listing 10: Median filtering to reduce noise in acceleration

```
ast_ddx_filtered = medfilt1(ast_ddx, 50);
ast_ddy_filtered = medfilt1(ast_ddy, 50);
```

With smoothed acceleration estimates, I then applied Newton's second law to recover the drag force in each direction:

Listing 11: Solving for drag forces using Newton's second law

```
Fdrag_x = ast_ddx_filtered * MASS_AST;
Fdrag_y = (g + ast_ddy_filtered) * MASS_AST;
```

The magnitude of the total drag force vector was computed, and since drag force was also given by $F_{\text{drag}} = c\sqrt{\dot{x}^2 + \dot{y}^2}$, I solved for c at each time step:

Listing 12: Estimation of drag coefficient c

```
c = Fdrag ./ sqrt(ast_dx.^2 + ast_dy.^2);
c = mean(c);
```

By taking the average value of c over the simulation, I obtained a robust estimate of the drag coefficient. Overall, this approach demonstrated how noisy simulation data can be processed, differentiated, and interpreted to extract physically meaningful parameters.

3 Milestone 2

3.1 Control Design Strategy

The control design process began with the fundamental requirement of guiding the rocket to detonate within a narrow positional tolerance relative to the asteroid. My approach to passing this milestone was to design a controller that worked for Scenario 1's dynamics, followed by a separate controller that worked for Scenario 2's more complex dynamics. I then combined the two controllers using Simulink logic and the scenario-specific simulation parameters to automatically run the appropriate control algorithm.
I chose to implement **LQR** controllers because LQR offers an optimal trade-off between control performance and effort, making it suitable for a system where both fuel efficiency and tracking accuracy are critical (in real life). It allows systematic tuning using weighting matrices (Q and R) to prioritise specific state variables. The resulting feedback law is simple and fast to compute, making it practical for real-time simulation.
For each scenario, I identified relevant state variables that could be isolated, linearised, and used in a state-space model.

3.1.1 Scenario 1: Vertical Interception

State-Space Model and LQR Controller Design

The primary goal of Scenario 1 was to ensure that the rocket intercepted the asteroid at a safe vertical offset, detonating within 150 meters of its centre of mass. For this purpose, I modelled the rocket's vertical dynamics based on the y-axis equation from the manipulator equation derived in Milestone 1, i.e. the second row of the Matrix in in Figure 1.
From this, I constructed the state-space representation. The chosen states were vertical position (p_y) and vertical velocity ($\dot{p}_y$), forming the vector:

$$x = \begin{bmatrix} p_y \\ \dot{p}_y \end{bmatrix}$$

The continuous-time state-space matrices were defined as:

Listing 13: State-space model for vertical motion

```
A1 = [0, 1; 0, 0];
B1 = [0; 1/1000];
C1 = [1, 0];
D1 = 0;
```

To prioritise accurate position tracking while allowing some tolerance in control effort, I selected a high Q of 10000 and a low R of 1. The full code for determining the controller gains `Kr1` and `Klqr1` is viewable in Listing 31.
Setting the Controller's Reference Position

In Scenario 1, the rocket must ascend vertically to intercept the asteroid at the point where the asteroid's horizontal (x) position reaches zero. To achieve this, a predicted reference y-position is calculated based on the asteroid's current trajectory and estimated time to reach $x = 0$.
The time to intercept is computed using the asteroid's current x-position and horizontal velocity:

Listing 14: Time to intercept at $x = 0$

```
t = (x_target - ast_x)/ast_dx; % Time to intercept at x = 0
```

Assuming motion under constant gravity only, the asteroid's vertical position at this time is predicted using basic kinematic equations. The predicted reference point for the rocket is then:

Listing 15: Reference y-position at intercept

```
ref = ast_y + ast_dy * t - 0.5 * g * t^2; % Predicted y-position at intercept
```

This reference is passed to the LQR controller to compute the required thrust:

Listing 16: LQR thrust computation

```
F = ref * Kr1 - Klqr1 * [y; dy];
```

The thrust is then clamped between 0 and 1000000N to ensure it stays within realistic bounds.

Listing 17: Thrust saturation and detonation logic

```
F = max(min(F, 1000000), 0); % Thrust saturation
```

3.1.2 Scenario 2: Guided Interception via Angular Control

State-Space Model and LQR Controller Design

Scenario 2 introduced a more complex problem: the asteroid's trajectory no longer permitted a vertical launch. The rocket's orientation needed to be controlled precisely to allow redirection and interception at a desired point.
To address this, I extracted the angular motion from the third row of the Matrix in in Figure 1. I chose to fix the rocket's thrust at its maximum allowable value throughout the simulation, ensuring sufficient force was always available to perform aggressive orientation adjustments as required. Here, θ is the rocket's tilt angle, and α is the thrust vector angle. I defined the state vector as:

$$x = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix}$$

and constructed the following linear system:

Listing 18: State-space model for angular motion

```
A2 = [0, 1; 0, 0];
B2 = [0; (-21*1000000)/221000];
C2 = [1, 0];
D2 = 0;
```

Since aggressive angular manoeuvring could destabilise the rocket, I chose a relatively balanced tuning with low Q and R values of 1. The full code for determining the controller gains `Kr2` and `Klqr2` is alike to Listing 31.
Setting the Controller's Reference Angle

To set the reference position which I wanted the LQR controller to track, I first estimated the time of collision, based on proximity between the asteroid and the rocket. The value of 4610 was obtained from viewing the underlying Scenario 2 generation code, and represents the maximum possible distance between the rocket and asteroid. The idea is that the time of collision estimate will dynamically adjust, as the proximity between the rocket and asteroid varies.

Listing 19: Collision time estimation

```
time_factor = proximity / 4610;
t = time_factor * 11;
```

Next, I predicted the intercept coordinates:

Listing 20: Predicted target point for intercept

```
target_x = ast_x + ast_dx * t;
target_y = ast_y + ast_dy * t - 0.5 * g * t^2;
```

Because the LQR controller was designed to control the rocket's angular position, I needed to compute the required heading to be the controller's reference.

Listing 21: LQR for orientation control

```
dx = target_x - x;
dy = target_y - y;
theta_ref = atan2(dy, dx) - pi/2;
```

The conditional thrust logic in Listing 34 ensured that, in a realistic scenario where fuel would be limited, fuel was conserved until the rocket was correctly oriented. It also prevented premature launches that could lead to missed interceptions.

3.2 Controller Implementation in Simulink

The above LQR controller design algorithms were implemented as a MATLAB Function block in Simulink, named `fcn`, with the following function signature describing its inputs and outputs:

Listing 22: Function signature for custom control block

```
function [F, alpha, detonate] = fcn(x, y, dth, ast_y, dy, ...
  Klqr1, Kr1, Klqr2, Kr2, ast_dx, ast_x, ast_dy, th, scenario)
```

The controller block uses real-time state information to make dynamic decisions about thrust and orientation, based on which scenario is active.

3.2.1 Automatic Controller Selection via the `scenario` Input Variable

While configuring my control system, I noticed a consistent pattern in the underlying simulation code: in Scenario 1, the asteroid always started at a height of approximately 5000 m on the y-axis, whereas in Scenario 2 it consistently started closer to 4000 m. This observation allowed me to differentiate between the two scenarios automatically at runtime. To exploit this, I added a conditional logic block in Simulink that monitored the initial vertical position of the asteroid (`ast_y`). Specifically, I inserted an `If` block that compared `ast_y` to a threshold value of 4500, i.e If `ast_y` $\geq$ 4500, the scenario was classified as Scenario 1, else the scenario was classified as Scenario 2.
The `If` block was connected to an `If Action` subsystem, set to a default output value of 2, but outputted a scalar value of 1 if Scenario 1 was detected. The result was wired into the `scenario` input of my unified controller block. This setup allowed both control strategies to be embedded within the single `fcn` block. Internally, the function would check the scenario value and run the appropriate logic:

Listing 23: Scenario check inside controller

```
if scenario == 1
    % Run Scenario 1 logic
elseif scenario == 2
    % Run Scenario 2 logic
end
```

3.2.2 The `Detonate` Output Variable and Logic

The detonation logic is implemented to determine whether the rocket should initiate its payload detonation based on its proximity to the asteroid. This is achieved by calculating the distance between the rocket's current position (x, y) and the asteroid's position (`ast_x`, `ast_y`):

Listing 24: Proximity calculation

```
proximity = norm([ast_x - x, ast_y - y]); % Distance between rocket and asteroid
```

If the rocket is within a predefined threshold of 150 meters, detonation is triggered by setting the `detonate` flag to 1. Otherwise, it remains 0:

Listing 25: Detonation condition

```
detonate = 0; % Default detonation status

if proximity <= 150
    detonate = 1; % Trigger detonation if close enough
end
```

The `detonate` variable is a binary output flag that serves as the control signal for the detonation logic block. This flag is connected to the subsystem responsible for initiating the explosion sequence. When `detonate` is set to 1, the detonation logic block is activated, ensuring the system only initiates detonation under safe and intentional conditions.

4 Milestone 3

4.1 Control Design Strategy

For the final milestone, the challenge evolved further: not only did the rocket have to intercept the asteroid, but it also had to strike within a specific angular arc aligned with either end of a visible blue strip on the asteroid's surface. This introduced a critical requirement: precise control over the rocket's orientation during its final approach.
Fortunately, since Milestone 3's requirements were still rooted in orientation-based targeting, I was able to recycle the same LQR gain matrices `Klqr2` and `Kr2` without any structural modification to the control law and minimal modification to the controller architecture.

4.1.1 Updating the Controller's Reference Angle and Logic

To meet the angular targeting requirement in Milestone 3, I introduced a new layer of logic designed to estimate the optimal time-to-impact (t) more accurately, based on the asteroid's angular velocity at launch. This parameter, `ast_dth`, while not constant, remained relatively stable during each simulation run and strongly influenced the asteroid's orientation during the rocket's approach. Rather than use a single time estimation formula for all cases, I partitioned the possible range of `ast_dth` values into 12 subintervals. Each interval corresponded to a distinct behaviour pattern observed during trial-and-error tuning, and was associated with a different multiplier for the time factor (see Listing 35). Additionally, when I struggled to find an accurate time factor for some intervals, I included fallback logic: for example, when the proximity between the rocket and asteroid dropped below a certain threshold (e.g., 2900 m), the controller would override the standard prediction and instead aim for the asteroid's current target point. This proximity-aware refinement made the system more robust to fluctuations in `ast_dth` and increased the reliability of successful strikes across all test cases.
Originally, the rocket was programmed to target the asteroid's center of mass. However, upon further analysis of the simulation environment, it was observed that a blue line (300 meters in length) is drawn across the asteroid to represent its orientation. To ensure realistic impact targeting and increase the likelihood of success, the aiming logic was modified to aim for either end of this blue line, rather than the COM. The approach shown in 36 involves computing the distance from the rocket to both ends of the line, which are offset by ± 150 meters along the asteroid's orientation angle. The rocket then aims for the closer of these two points.

4.2 Final Controller Implementation in Simulink

Aside from the addition of the `interval` input variable which is explained below, the only new wired input to the controller block, was the `ast_th` variable which was needed to compute the new reference angle.

4.2.1 Automatic Time Estimate Adjustment via the `interval` Input Variable

In the Simulink model, I implemented the interval logic using: an `If` block that compared `ast_dth` against the predefined interval boundaries, 12 corresponding `If Action Subsystems`, each outputting a fixed scalar (1 to 12) based on the condition, and a `Merge` block that consolidated the outputs into a single signal called `interval`, which was fed into the controller.
This structure allowed the controller to dynamically select one of 12 tuned time estimation behaviours based on the starting angular velocity of the asteroid. As explained in 4.1.1, within the controller itself, the `interval` variable was used to branch into a series of conditional statements, each applying a different time multiplier to the proximity-based time factor.

5 Results and Discussion

5.1 Milestone 2

To validate the robustness of both controllers in this Milestone, extensive testing was conducted using a wide range of random seeds. Many different seeds were tried, and both scenarios passed all test cases. Confidence in the solution is high, as the controller design directly leverages insights from the underlying simulation code that governs asteroid movement and seed generation. This foundational understanding ensures that the control logic generalises well to all possible generated cases for each scenario.

5.1.1 Scenario 1

The LQR-based thrust control ensured a smooth and efficient ascent. The detonation logic triggered correctly once the rocket reached within 150 meters of the asteroid. An example of this can be viewed in Figure 2.

5.1.2 Scenario 2

The rocket consistently adjusted its flight path and detonated as expected within the proximity threshold. An example of this can be viewed in Figure 3.

5.2 Milestone 3

The same testing approach used for Scenarios 1 and 2 was applied to Scenario 3 to evaluate the robustness of the controller.

5.2.1 Scenario 3

While the controller performed well in most cases, two key limitations were identified. Firstly, due to time constraints, I was unable to accurately determine the correct time factor for middle intervals, specifically when the asteroid's angular velocity (`ast_dth`) was near zero. This resulted in inaccurate predictions of the asteroid's orientation during interception, which reduced the effectiveness of the control strategy in these cases. Secondly, the controller occasionally failed when there was a large and rapidly changing variation in `ast_dth` throughout the simulation (observed using a scope). Although such cases were rare and not representative of typical behaviour, they introduced significant uncertainty in the aim point prediction logic. Aside from these edge cases, the controller demonstrated reliable performance across the remaining seed tests, successfully adjusting its aim and triggering detonation when within the required proximity. An example of this can be viewed in Figure 4.

6 Conclusion

This project demonstrated the successful modelling, simulation, and control of a rocket intercepting a tumbling asteroid across three increasingly complex milestones. Through symbolic dynamics, state-space modelling, and LQR control, robust solutions were developed and validated for Scenarios 1 and 2 via extensive seed-based testing. While Scenario 3 posed challenges due to time constraints and unpredictable angular velocity variations, the controller performed reliably in most cases. The design was grounded in a strong understanding of the simulation code, supporting its generalisability. In future work, I would focus on improving edge-case performance and explore Proportional Navigation (PN) as a potential alternative control strategy.

A Appendix

Listing 26: Definition of generalised coordinates

```
syms px py pth dpx dpy dpth ddpx ddpy ddpth
q = [px; py; pth];
dq = [dpx; dpy; dpth];
ddq = [ddpx; ddpy; ddpth];
```

Listing 27: Rotation matrix function about the z-axis

```
function rot = RotZ(th)
    rot = [ cos(th)  sin(th) 0;
           -sin(th)  cos(th) 0;
                 0        0  1];
end
```

Listing 28: Matrix Derivative

```
dM = sym(zeros(length(M)));
for i = 1:length(M)
    for j = 1:length(M)
        dM(i,j) = jacobian(M(i,j), q) * dq;
    end
end
dM = simplify(dM);
```

Listing 29: Generalised force vector due to thrust

```
Qx = sum(F_vec_i .* diff(p_F_i, px), 'all');
Qy = sum(F_vec_i .* diff(p_F_i, py), 'all');
Qth = sum(F_vec_i .* diff(p_F_i, pth), 'all');
Q = simplify([Qx; Qy; Qth]);
```

Listing 30: Numerical differentiation of velocity to estimate acceleration

```
ast_ddx = diff(ast_dx)./diff(simulation_time);
ast_ddx(end+1) = ast_ddx(end); % Match array size
ast_ddy = diff(ast_dy)./diff(simulation_time);
ast_ddy(end+1) = ast_ddy(end);
```

Listing 31: LQR tuning and gain computation for vertical control

```
Q1 = 10000 * eye(2);
R1 = 1;

Klqr1 = lqr(A1, B1, Q1, R1);      % Optimal feedback gain
Acl1 = A1 - B1 * Klqr1;           % Closed-loop system matrix
Kdc1 = D1 - C1 * inv(Acl1) * B1; % Output scaling term
Kr1 = 1 / Kdc1;                   % Reference scaling factor
```

Listing 32: LQR tuning and gain computation for angular control

```
Q2 = 1 * eye(2);
R2 = 1;

Klqr2 = lqr(A2, B2, Q2, R2);      % Optimal feedback gain
Acl2 = A2 - B2 * Klqr2;           % Closed-loop system matrix
Kdc2 = D2 - C2 * inv(Acl2) * B2; % Output scaling term
Kr2= 1 / Kdc2;                   % Reference scaling factor
```

Listing 33: LQR for orientation control

```
dx = target_x - x;
dy = target_y - y;
theta_ref = atan2(dy, dx) - pi/2;
ref = [theta_ref; 0];
state = [th; dth];
alpha = -Klqr2 * (state - ref) + Kr2 * theta_ref;
alpha = max(min(alpha, pi/2), -pi/2);
```

Listing 34: Angle-based thrust gating

```
angle_error = wrapToPi(theta_ref - th);
if abs(angle_error) < deg2rad(10) && cos(th) > 0
    F = F_max;
else
    F = F_hover;
end
```

Listing 35: Time-to-impact estimation based on `interval` value and proximity

```
if interval == 1 % ast_dx between -0.349066 and -0.290888 rad/s
    t = time_factor * 11.8;
elseif interval == 2 % ast_dx between -0.290888 and -0.232711 rad/s
    t = time_factor * 12.6;
elseif interval == 3 % ast_dx between -0.232711 and -0.174533 rad/s
    if proximity > 2900
        t = time_factor * 16;
    elseif proximity <= 2900
        t = time_factor * 0;
    end
elseif interval == 4 % ast_dx between -0.174533 and -0.116355 rad/s
    t = time_factor * 12.5;
elseif interval == 5 % ast_dx between -0.116355 and -0.058178 rad/s
    t = time_factor * 13;
elseif interval == 6 % ast_dx between -0.058178 and 0 rad/s
    t = time_factor * 13.7;
elseif interval == 7 % ast_dx between 0 and 0.058178 rad/s
    t = time_factor * 13.6;
elseif interval == 8 % ast_dx between 0.058178 and 0.116355 rad/s
    t = time_factor * 12.5;
elseif interval == 9 % ast_dx between 0.116355 and 0.174533 rad/s
    t = time_factor * 12.4;
elseif interval == 10 % ast_dx between 0.174533 and 0.232711 rad/s
    if proximity > 2900
        t = time_factor * 12.3;
    elseif proximity < 800
        t = time_factor * 0;
    elseif proximity <= 2900
        t = time_factor * 16;
    end
elseif interval == 11 % ast_dx between 0.232711 and 0.290888 rad/s
    if proximity > 2900
        t = time_factor * 12.2;
    elseif proximity < 900
        t = time_factor * 0;
    elseif proximity <= 2900
        t = time_factor * 11;
    end
elseif interval == 12 % ast_dx between 0.290888 and 0.349066 rad/s
    t = time_factor * 12.5;
end
```

Listing 36: Aiming for the nearest end of the asteroid's visual marker

```
if norm([ast_x + 150*cos(ast_th) - x, ast_y + 150*sin(ast_th) - y]) >= ...
   norm([ast_x - 150*cos(ast_th) - x, ast_y - 150*sin(ast_th) - y])

    x_aim = ast_x + 150*cos(ast_th);
    y_aim = ast_y + 150*sin(ast_th);
else
    x_aim = ast_x - 150*cos(ast_th);
    y_aim = ast_y - 150*sin(ast_th);
end
```

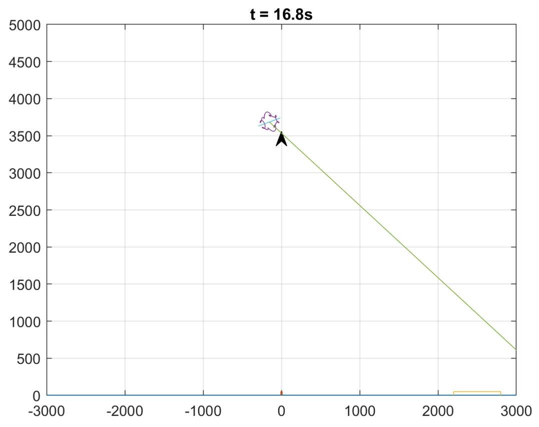


Figure 2: Successful vertical interception in Scenario 1

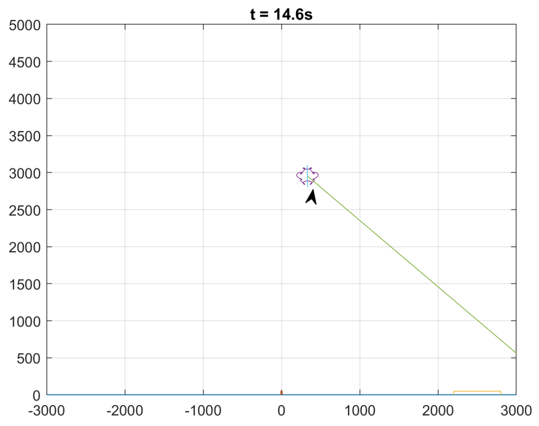


Figure 3: Successful angular interception in Scenario 2

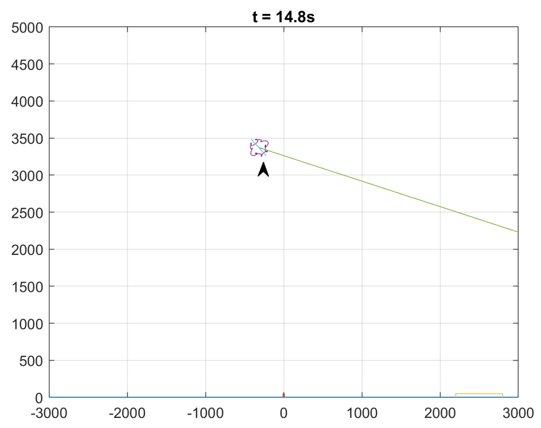


Figure 4: Successful angular interception in Scenario 3