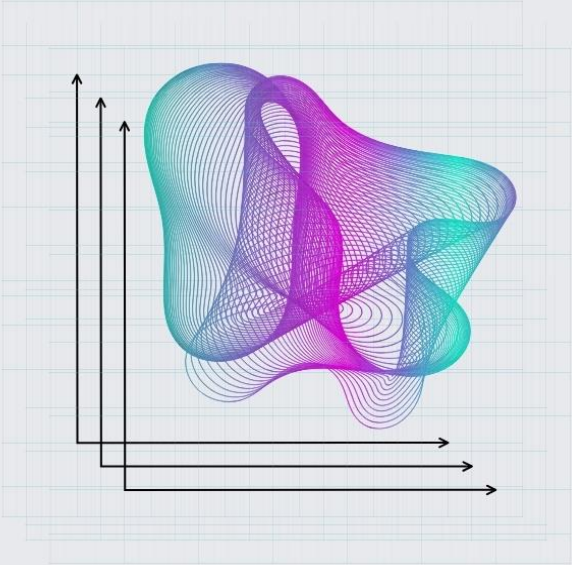
Ahsanullah University of Science and Technology

Numerical Technique Laboratory

Report: Open Ended Lab Project



Submitted by

Name: Syed Zarir Sabit || Mahadi Amin Khan || Tanvir Ahmed Semon
Student ID: 20220105126, 131, 134
Semester: 3rd Year 1st Semester
Section : C-1
Group: 01

Submitted to:

Mr. Kazi Tauseef Mohammad
Dr. Dewan Monzurul Islam

EEE-3110

JApril 8. 2025

Project Name: Skypath-Interceptor UAV Path Simulation using MATLAB

1. Project Overview

In the evolving field of UAV technology, one of the most critical challenges is the interception of moving targets. This report delves into the design and implementation of a simulation for intercepting a moving UAV using a second UAV, referred to as the "interceptor." The goal of the project is to create a realistic model where the interceptor UAV can track and intercept the target UAV by adjusting its trajectory in real time.

The simulation leverages the principles of proportional navigation, spline interpolation for smooth path planning, and real-time visualization to assess the effectiveness of the interception strategy. By simulating this interaction, we aim to explore potential applications in defense, autonomous drone operations, and airspace management.

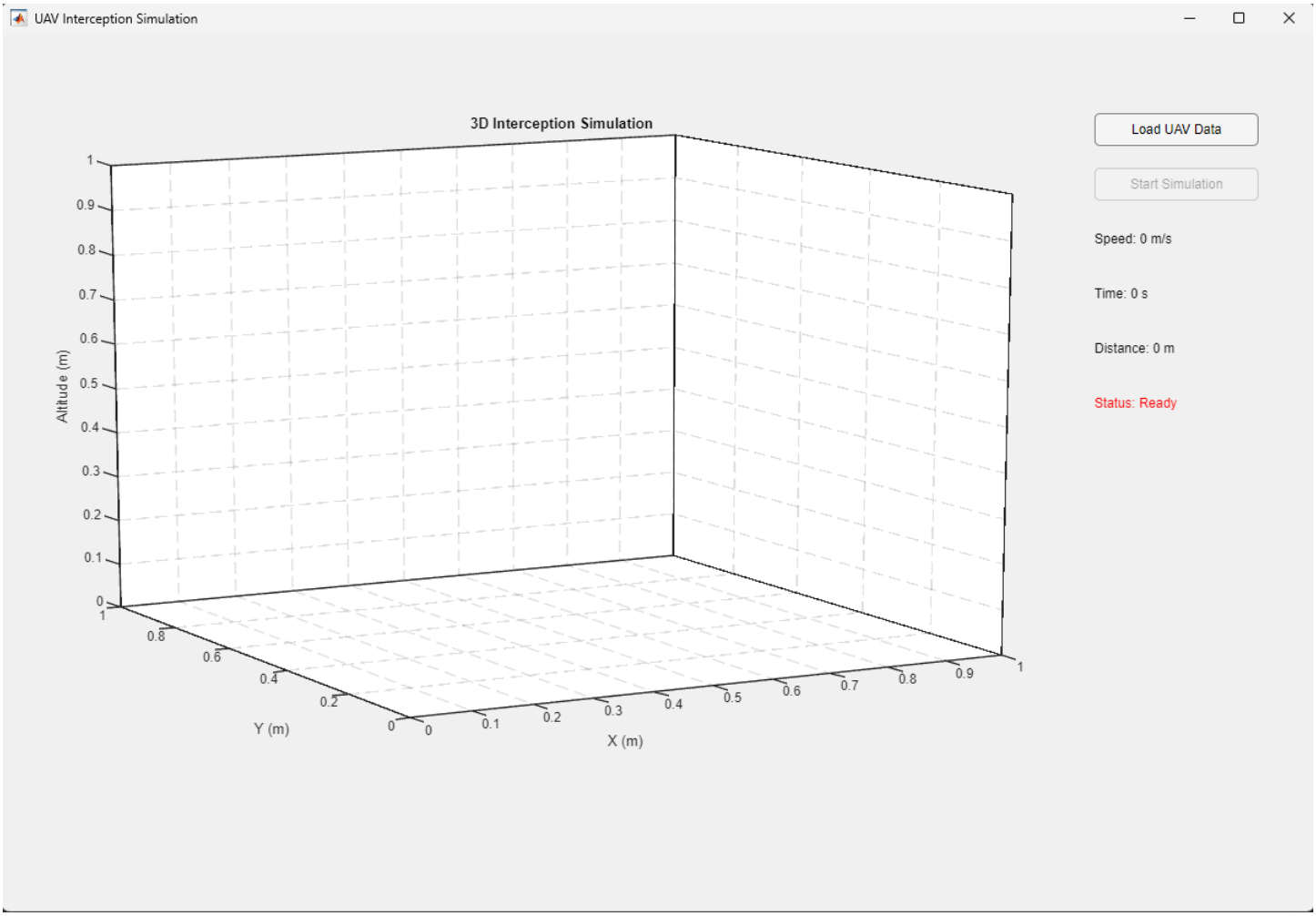
2. Project Objectives

- **Read UAV Data:** Read the latitude and altitude of a target UAV from a file.
- **Coordinate Conversion:** Convert latitude and altitude to Cartesian coordinates.
- **Path Plotting:** Plot the trajectory of the target UAV using curve fitting or interpolation techniques.
- **Interception Path Calculation:** Calculate the path of the interceptor UAV, considering constant velocity.
- **Interception Points:** Determine the points at which the interceptor UAV will intersect the target UAV's path.
- **Speed and Navigation:** Compute the required speed of the interceptor UAV using proportional navigation and Euler's method.

3. Project Outline

The project consists of several key stages, from reading data to simulating the flight path of the interceptor UAV. Below is a detailed explanation of each section with object-oriented code snippets.

3.1 mlapp UI Layout



4. **MATLAB Code Explanation**

To maintain clarity, modularity, and scalability, the entire project is structured using **Object-Oriented Programming (OOP)** in MATLAB. OOP allows for encapsulating different functionalities into reusable and isolated components — ideal for modeling UAV behavior and extending to more complex swarm or multi-agent systems in the future.

We structured the simulation into **four distinct classes**, each with a dedicated role in the system:

Class Name	Responsibility
UAVdata	Reads the target UAV's latitude and altitude, and converts them into Cartesian coordinates.
PathPlanner	Interpolates and plots the smooth trajectory of the target UAV.
Interceptor	Implements proportional navigation and Euler's method to calculate interceptor UAV dynamics.
Simulation	Integrates all modules and runs the complete real-time interception simulation.

This modular structure makes the system:

- **Readable** — each class has a focused role.
- **Maintainable** — easy to test and upgrade components individually.
- **Extendable** — future support for 3D navigation, wind modeling, swarm logic, etc.

4.1 UAVdata Class

Purpose

Reads the raw data from a csv file (sensor_data.csv) containing latitude and altitude of a target UAV. Converts the data into a Cartesian (x, y) format, calculates velocities, and prepares it for further processing.

```
longitude,latitude,altitude,time
90.41249520825761,23.810304967141533,10.079103194704304,0
90.41249335166785,23.81030358449852,9.988164449224831,1
90.4124822883181,23.810310061383902,10.128443880318441,2
90.41247032625186,23.810325291682464,9.988258774039213,3
90.4124784515101,23.810322950148716,10.04694448341924,4
90.41249201391038,23.810320608779147,10.265990046000237,5
90.41249129380917,23.810336400907303,10.166936413487168,6
90.41250132913814,23.810344075254594,10.110306640526892,7
90.4125049454984,23.810339380510737,10.120271777035656,8
90.41249849430085,23.81034480611117,10.069924211624036,9
90.4125021082569,23.810340171934243,9.914857868517423,10
90.41251748862257,23.81033551463671,9.921714165998026,11
90.41251713036218,23.810337934259422,9.815483794625415,12
90.41253277679873,23.810318801456976,9.862843037688933,13
90.41250657934769,23.810301552278652,9.770900614265553,14
```

Photo: **Raw Sensor Data of the Targeted UAV's GPS co-ordinates**

Design Highlights

- Uses a simple flat Earth model with radius R=6371000R = 6371000R=6371000 meters.
- Assumes constant time intervals between readings (1 second).
- Returns interpolatable data arrays.

Key Methods

- readData(): Loads and converts data.
- getData(): Returns the x, y, vx, vy, t arrays for external use.

Code

```
classdef UAVdata
    properties
        filepath;
        ref_longitude;
        ref_latitude;
        ref_altitude;
        cartesian;
        velocity;
        time;
    end


    methods
        function obj= UAVdata(FilePath)
            obj.filepath= FilePath;
            obj.ref_longitude= [];
            obj.ref_latitude= [];
            obj.ref_altitude= [];
            obj.cartesian= [];
```

```
    obj.velocity=[];
    obj.time=[];
end

function obj= readData(obj)
    rawData=readmatrix(obj.filepath);

    if size(rawData,2) <4
        error('File not formated correctly!!!');
    end

   longitudes= rawData(:,1);
   latitudes= rawData(:,2);
   altitudes= rawData(:,3);
   obj.time=rawData(:,4);



   obj.ref_longitude= longitudes(1);
   obj.ref_latitude= latitudes(1);
   obj.ref_altitude= altitudes(1);

   %formula
   delta_longitude= longitudes-obj.ref_longitude;
   delta_latitude= latitudes-obj.ref_latitude;
   delta_altitude= altitudes-obj.ref_altitude;
```

```
    R=6378137; %earth radius

    x= deg2rad(delta_longitude)*R*cosd(obj.ref_latitude);
    y= deg2rad(delta_latitude)*R;
    z= delta_altitude;

    obj.cartesian= [x , y, z];

    dt=diff(obj.time);
    if any(dt<=0)
        error('Time must be monotonically increasing!');
    end
    vx=diff(x)./dt;
    vy=diff(y)./dt;
    vz=diff(z)./dt;

    vx=[vx; vx(end)];
    vy=[vy; vy(end)];
    vz=[vz; vz(end)];

    obj.velocity=[vx,vy,vz];

end

function [x,y,z,vx,vy,vz, time]= getData(obj)
```

```
        x=obj.cartesian(:,1);
        y=obj.cartesian(:,2);
        z=obj.cartesian(:,3);
        vx=obj.velocity(:,1);
        vy=obj.velocity(:,2);
        vz=obj.velocity(:,3);
        time=obj.time;
    end
  end
end
```

4.2 Simulation Class

Purpose

Handles cubic spline interpolation for the UAV's position and velocity in 3D space. This allows smooth trajectory generation from discrete data points, which is essential for accurate simulation and guidance.

Design Highlights

- Takes time-series data for position (x, y, z) and velocity (vx, vy, vz).
- Implements **natural cubic spline interpolation** using a custom tridiagonal solver.
- Allows evaluation of interpolated values at any desired time via the interpolate() method.
- Ensures modular and reusable implementation through clean OOP structure.

Key Methods

PathPlanner(x, y, z, vx, vy, vz, time)
Constructor that initializes the spline coefficients for each dimension.

- Automatically reshapes inputs to column vectors.
- Uses dospline() method to calculate spline coefficients for all six vectors (3 position, 3 velocity).

---

dospline(values) *(private)*

Computes natural cubic spline coefficients for a given vector using finite difference method.

- Constructs the tridiagonal system of equations.
- Solves second derivative values using custom Thomas Algorithm (solvee).
- Outputs [a b c d] coefficients for spline segments.

---

interpolate(t_query)

Evaluates the interpolated values at specific query times.

- Calls findd() to locate the correct spline segment.
- Uses evaluate_spline() to compute interpolated values for:
  - x, y, z positions
  - vx, vy, vz velocities

Returns all six vectors as outputs.

---

evaluate_spline(coeffs, index, t_query) *(private)*

Evaluates a single spline function at given times.

- Uses standard cubic spline equation: f(t) = a·(dt)^3 + b·(dt)^2 + c·dt + d

---

findd(t) *(private)*

Finds the spline interval index for a given time query.

- Ensures safe bounds within the spline segments.
- Returns the index of the last time point less than or equal to t.

---

solvee(a, b, c, d) *(private)*

Solves a tridiagonal matrix system using the **Thomas algorithm**.

- Used to calculate second derivatives in spline construction.

- Custom implementation avoids built-in solvers for full transparency and learning.

Code

```
classdef PathPlanner
    properties
        coeff_x;
        coeff_y;
        coeff_z;
        coeff_vx;
        coeff_vy;
        coeff_vz;
        t;
    end

    methods
        function obj= PathPlanner(x,y,z, vx,vy,vz, time)
            x=x(:);
            y=y(:);
            z=z(:);
            vx=vx(:);
            vy=vy(:);
            vz=vz(:);
            time= time(:);



            dx= diff (x);
            dy= diff (y);
```

```
    dz= diff(z);
    obj.t= time;

    obj.coeff_x=obj.dospline(x);
    obj.coeff_y=obj.dospline(y);
    obj.coeff_z=obj.dospline(z);
    obj.coeff_vx=obj.dospline(vx);
    obj.coeff_vy=obj.dospline(vy);
    obj.coeff_vz=obj.dospline(vz);

  end

  function coeffs= dospline(obj, values)

    n= length(values);
    h= diff(obj.t);
    if any(h<=0)
      error('Time must me motionally increasing!');
    end

    daigonal_matrix= 2*(h(1:end-1) + h(2:end));
    upper=h(2:end-1); %eikahane bhul korte pari previously it was 1:end-2
    lower=h(1:end-2);
    rhs=6*((values(3:end)-values(2:end-1))./h(2:end)-(values(2:end-1)-values(1:end-
2))./h(1:end-1));
```

```
        derivatives= obj.solvee(lower, daigonal_matrix, upper, rhs);
        derivatives= [0; derivatives; 0];

        coeffs=zeros(n-1,4);
            for i=1:n-1
            a=derivatives(i+1)/(6*h(i));
            b=derivatives(i)/(2);
            c=(values(i+1)-values(i))/h(i)-h(i)*(2*derivatives(i)+derivatives(i+1))/6;
            d=values(i);
            coeffs(i,:) = [a, b, c, d];
            end
    end

    function [x_interp, y_interp, z_interp, vx_interp, vy_interp, vz_interp]= interpolate(obj,
t_query)

        funn= arrayfun(@(t) obj.findd(t), t_query);

        x_interp=obj.evaluate_spline(obj.coeff_x, funn, t_query);
        y_interp=obj.evaluate_spline(obj.coeff_y, funn, t_query);
        z_interp=obj.evaluate_spline(obj.coeff_z, funn, t_query);
        vx_interp=obj.evaluate_spline(obj.coeff_vx, funn, t_query);
        vy_interp=obj.evaluate_spline(obj.coeff_vy, funn, t_query);
        vz_interp=obj.evaluate_spline(obj.coeff_vz, funn, t_query);
    end
```

```
    function val= evaluate_spline(obj,coeffs,funn,t_query)

      dt= t_query-obj.t(funn);
      a=coeffs(funn,1);
      b=coeffs(funn,2);
      c=coeffs(funn,3);
      d=coeffs(funn,4);

      val= a.*dt.^3+ b.*dt.^2 +c.*dt +d;
    end

    function funn= findd(obj, t)

      funn= find(obj.t <= t, 1, 'last');
      funn= max(1, min(numel(obj.t)-1, funn));

    end
end

methods(Access=private)

function x= solvee (obj, a,b,c,d)

assert(length(b) == length(d), "Main diagonal and RHS size mismatch");
assert(length(a) == length(c), "Lower/upper diagonal size mismatch");
```

```
    n=length(d);
    for i= 2:n

      w= a(i-1)/b(i-1);
      b(i)= b(i)-w*c(i-1);
      d(i)= d(i)-w*d(i-1);
    end

    x=zeros(n,1);
    x(n)=d(n)/b(n);

    for i=n-1:-1:1
       x(i)=(d(i)-c(i)*x(i+1))/b(i);
    end
    end
    end
end
```

4.3 Interceptor Class

Purpose

Implements Proportional Navigation (PN) and Euler's integration method to simulate the interceptor UAV's real-time trajectory as it attempts to intercept a moving target UAV.

Design Highlights

- Assumes constant speed for the interceptor UAV.
- Calculates the line-of-sight (LOS) angle rate between target and interceptor.
- Uses Proportional Navigation Law to determine the interceptor's turning rate.
- Updates position and velocity using Euler integration at each timestep.

Key Method

navigate(x_t, y_t, x_i, y_i, t, i)
Calculates the updated position and velocity of the interceptor at time t(i) using proportional navigation.

- **Inputs:**
  - x_t, y_t: Target UAV position at time t(i)
  - x_i, y_i: Current position of the interceptor
  - t: Time vector
  - i: Current time index

Algorithm:

- Compute line-of-sight angle λ using atan2(y_t - y_i, x_t - x_i).
- Approximate rate of change of λ (λ_dot) using finite difference.
- Compute heading angle change using heading_rate = N * λ_dot (where N is the navigation constant, e.g., 3).
- Update the heading angle and interceptor’s position using:
  - x_i = x_i + speed * cos(heading) * dt
  - y_i = y_i + speed * sin(heading) * dt

Output:
Returns updated [x_i, y_i, vx_i, vy_i] for the next time step.

Code:

```
classdef Interceptor
    properties
        position;
        velocity;
        nav;
        dt;
        min_speed=20;
    end

    methods

        function obj= Interceptor(starting_pos, starting_vel, nav, dt)
            obj.position=starting_pos(:);
            obj.velocity=starting_vel(:);
            obj.nav=nav;
            obj.dt=dt;

            if norm(obj.velocity) < obj.min_speed
                if norm(obj.velocity) == 0
                    obj.velocity = obj.min_speed * [1; 0; 0];
            else
                obj.velocity = obj.min_speed * (obj.velocity / norm(obj.velocity));
            end
        end
        end
```

```
    function obj= update(obj, target_pos, target_vel)

        relative_pos= target_pos(:) -obj.position;
        relative_vel= target_vel(:) -obj.velocity;

        horizontal_distance=norm(relative_pos(1:2))+ 1e-6;
        los_azimuth= atan2(relative_pos(2), relative_pos(1));
        los_elevation=atan2(relative_pos(3),horizontal_distance);

        los_unit = relative_pos / norm(relative_pos);
        los_rate_vector = cross(los_unit, relative_vel) / norm(relative_pos);
        los_rate = norm(los_rate_vector);

        turn_rate=obj.nav*los_rate;
        new_heading= los_azimuth+turn_rate*obj.dt;

        speed=norm(obj.velocity);
        desired_speed = max(obj.min_speed, min(40, norm(relative_vel) + 5));

        obj.velocity= desired_speed*[cos(new_heading)*cos(los_elevation);
sin(new_heading)*cos(los_elevation); sin(los_elevation)];

        obj.position= obj.position+ obj.velocity*obj.dt;
    end
```

```
    function [pos, vel] = getState(obj)


        pos = obj.position;
        vel = obj.velocity;
    end


  end
end
```

4.4 Simulation Class

Purpose

Coordinates the simulation process by integrating the PathPlanner (for target UAV path), Interceptor (for guidance logic), and UAVdata (for reading input data). It handles data input, initialization, interpolation, trajectory calculation, and visualization**.**

Design Highlights

- Initializes all core components: UAVdata, PathPlanner, and Interceptor.
- Performs high-resolution interpolation of target UAV path and velocity.
- Applies proportional navigation at every timestep to simulate the interceptor's trajectory.
- Plots both UAV paths and visually highlights interception dynamics.

Key Method

run()

Executes the full simulation sequence from data loading to final visualization.

- **Workflow:**
    1. **Read Data:**

- Uses UAVdata to load target's position (x, y, z) and velocity (vx, vy, vz) vectors, and the time vector t.

2. **Trajectory Interpolation:**
    - Passes the data to PathPlanner to generate cubic spline coefficients.
    - Interpolates the path and velocity of the target UAV for a smooth high-resolution simulation.
3. **Interceptor Initialization:**
    - Starts from a predefined position (e.g., ground level).
    - Initializes the Interceptor object with a given speed.
4. **Simulation Loop:**

    For each timestep:
    - Gets the interpolated target position.
    - Updates interceptor position and velocity using navigate() from the Interceptor class.
    - Stores all positions for plotting.
5. **Visualization:**
    - Uses MATLAB plotting functions to show:
        - Target UAV path (smooth spline trajectory).
        - Interceptor path.
        - Interception dynamics in 2D or 3D as needed.

Output:
Final plots visualizing both UAV paths and their interaction.

Code:

```
classdef Simulation
    properties
```

```
    UAV;
    interceptor;
    planner;
    dt;
    t;
    fig;
    ax;
end

methods
    function obj = Simulation(UAV, planner, interceptor, dt, ax)
        obj.UAV = UAV;
        obj.planner = planner;
        obj.interceptor = interceptor;
        obj.dt = dt;
        obj.t = 0;
        obj.ax = ax;
    end

    function run(obj)

        time_end = max(obj.planner.t);

        while obj.t < time_end

            [uav_x, uav_y, uav_z, uav_vx, uav_vy, uav_vz] = obj.planner.interpolate(obj.t);
```

```
            obj.interceptor = obj.interceptor.update([uav_x, uav_y, uav_z], [uav_vx, uav_vy,
uav_vz]);



            obj.updatePlot();



            obj.t = obj.t + obj.dt;



            pause(obj.dt);
        end
    end

    function updatePlot(obj)

        cla(obj.ax);
        plot3(obj.ax, obj.UAV.cartesian(:,1), obj.UAV.cartesian(:,2), obj.UAV.cartesian(:,3), 'b-',
'LineWidth', 2);  % Plot UAV's path
        hold(obj.ax, 'on');



        scatter3(obj.ax, obj.interceptor.position(1), obj.interceptor.position(2),
obj.interceptor.position(3), 100, 'r', 'filled');
```

```
            xlabel(obj.ax, 'X (m)');
            ylabel(obj.ax, 'Y (m)');
            zlabel(obj.ax, 'Altitude (m)');
            grid(obj.ax, 'on');
            drawnow;
        end
    end
end
```

5.Simulation Workflow

- **Data Preparation:** The UAVdata class reads the target's trajectory data (latitude, longitude, altitude, and time) from a file, converting it to Cartesian coordinates for easier manipulation.
- **Path Planning:** The PathPlanner class interpolates the target's position and velocity over time using cubic splines to generate a smooth trajectory.
- **Interceptor Simulation:** The Interceptor class uses proportional navigation to adjust its heading and speed to intercept the target UAV. The interceptor's position is updated iteratively based on the relative position and velocity to the target.
- **Real-Time Visualization:** The Simulation class continuously updates the positions of the target and interceptor UAVs, plotting their trajectories in 3D space. The simulation runs until the target's trajectory is completed.

6. Key Features and Innovations

- **Proportional Navigation**: The interceptor uses proportional navigation, adjusting its heading and speed based on the relative motion of the target, ensuring realistic interception behavior.

- **Smooth Path Interpolation**: The use of cubic spline interpolation provides a smooth and continuous representation of the target UAV's trajectory, which is essential for accurate simulation and interception planning.
- **Real-Time Visualization**: The simulation includes real-time plotting of the UAV's trajectories, allowing for visual analysis of the interceptor's performance.

7. **Limitations**

1. **Simplified Aerodynamic Model**
    - Wind, drag, and other environmental factors are not modeled. This simplifies the physics but may reduce accuracy in real-world applications.
2. **Constant Velocity Assumption**
    - Both UAVs maintain constant speeds. This limits the ability to simulate dynamic scenarios like evasive maneuvers or acceleration during pursuit.
3. **Ideal Sensor and Communication Assumptions**
    - The simulation assumes perfect knowledge of the target's position and velocity, without accounting for sensor noise or communication delays.
4. **No Obstacle or Terrain Consideration**
    - The environment is considered obstacle-free. In practical UAV applications, terrain awareness and collision avoidance are essential.

8. **Future Improvements**

1. **Aerodynamic Forces & Environmental Effects**
    - Introduce wind models, air resistance, and turbulence to more accurately simulate real flight conditions.
2. **Adaptive Velocity and Acceleration Profiles**
    - Allow both UAVs to vary speed and acceleration based on engagement conditions, improving realism and control fidelity.
3. **Sensor Simulation and Delay Modeling**

- Implement sensor noise, latency, and estimation algorithms (e.g., Kalman Filters) for more robust real-time simulation.

4. **Obstacle and Terrain Integration**
    - Integrate terrain data and 3D obstacle maps into the path planning and navigation logic.
5. **Multi-Agent Coordination**
    - Extend the framework to simulate swarm UAV behavior, cooperative interception, or formation flight.
6. **Real-Time 3D Visualization Interface**
    - Develop an interactive 3D UI with playback controls, perspective switching, and telemetry display for demonstration and testing.

9.Conclusion

This project delivers a fully functional 3D interception simulation framework in MATLAB, structured with a clean object-oriented architecture. It demonstrates a working pipeline from raw trajectory data, through cubic spline interpolation, to real-time interception using proportional navigation and Euler's method. Despite idealistic assumptions, the modular class design enables rapid adaptation for more advanced control strategies and real-world integration. It lays a strong groundwork for future UAV research and intelligent autonomous systems.