1. Title Page

Title: Chaser Cart Docking Control and Path Planning

Submitted By: HEMANT PANDEY

Date: 22/01/25

2. Table of Contents

3. System Modeling

3.1 State-Space Representation

The system models the Chaser Cart's translational and rotational dynamics using a state-space approach. The states, inputs, and outputs are defined as follows:

State Vector:

$x(t) = [x, y, \theta, \dot{x}, \dot{y}, \dot{\theta}]^T$

Where:
x, y: Position coordinates.
θ: Orientation of the Chaser in radians.
$\dot{x}$, $\dot{y}$: Linear velocities.
$\dot{\theta}$: Angular velocity.

Input Vector:

$u(t) = [Fx, Fy, \tau]^T$

Where:
Fx, Fy: Forces in x and y directions.
τ: Torque for rotational control.

Output Vector:

The system outputs the state vector for full observation of all states we get position in x, y and theta as well as their 1st derivative which are essentially the velocities.

y(t) = x(t)

The state-space equations are:

$\dot{x}(t) = Ax(t) + Bu(t)$
$y(t) = Cx(t) + Du(t)$

With the following matrices:

```
A = [[0, 0, 0, 1, 0, 0],
     [0, 0, 0, 0, 1, 0],
     [0, 0, 0, 0, 0, 1],
     [0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0]]

B = [[0, 0, 0],
     [0, 0, 0],
     [0, 0, 0],
     [1/m, 0, 0],
     [0, 1/m, 0],
     [0, 0, 1/I]]
```

3.2 Assumptions and Parameters

Mass of Chaser (m): 1 kg.
Moment of Inertia (I): 0.1 kg·m^2.
Thruster Force Limits: ±0.5 N.
Gas Consumption Rate: 0.05 g/s per thruster.
Initial Conditions: Randomized using Monte Carlo simulation.

Example ranges:
Position: x, y ∈ [-20, 5] m.
Orientation: θ ∈ [-180°, 180°].

3.3 MATLAB code

As we have described the basic behavior for the system model, now we are ready to use the state-space command of MATLAB.

Filename: state_space_01.m

```
% Parameters
m = 1; % Mass of Chaser
I = 0.1; % Moment of inertia

% State-space matrices
A = [0 0 0 1 0 0;
0 0 0 0 1 0;
0 0 0 0 0 1;
0 0 0 0 0 0;
0 0 0 0 0 0;
0 0 0 0 0 0];
B = [0 0 0;
0 0 0;
0 0 0;
1/m 0 0;
0 1/m 0;
0 0 1/I];
C = eye(6); % Identity matrix to output all states
D = zeros(6, 3); % No direct feedthrough
```

```
% Create state-space system
sys = ss(A, B, C, D);
% Display system
disp(sys);
```

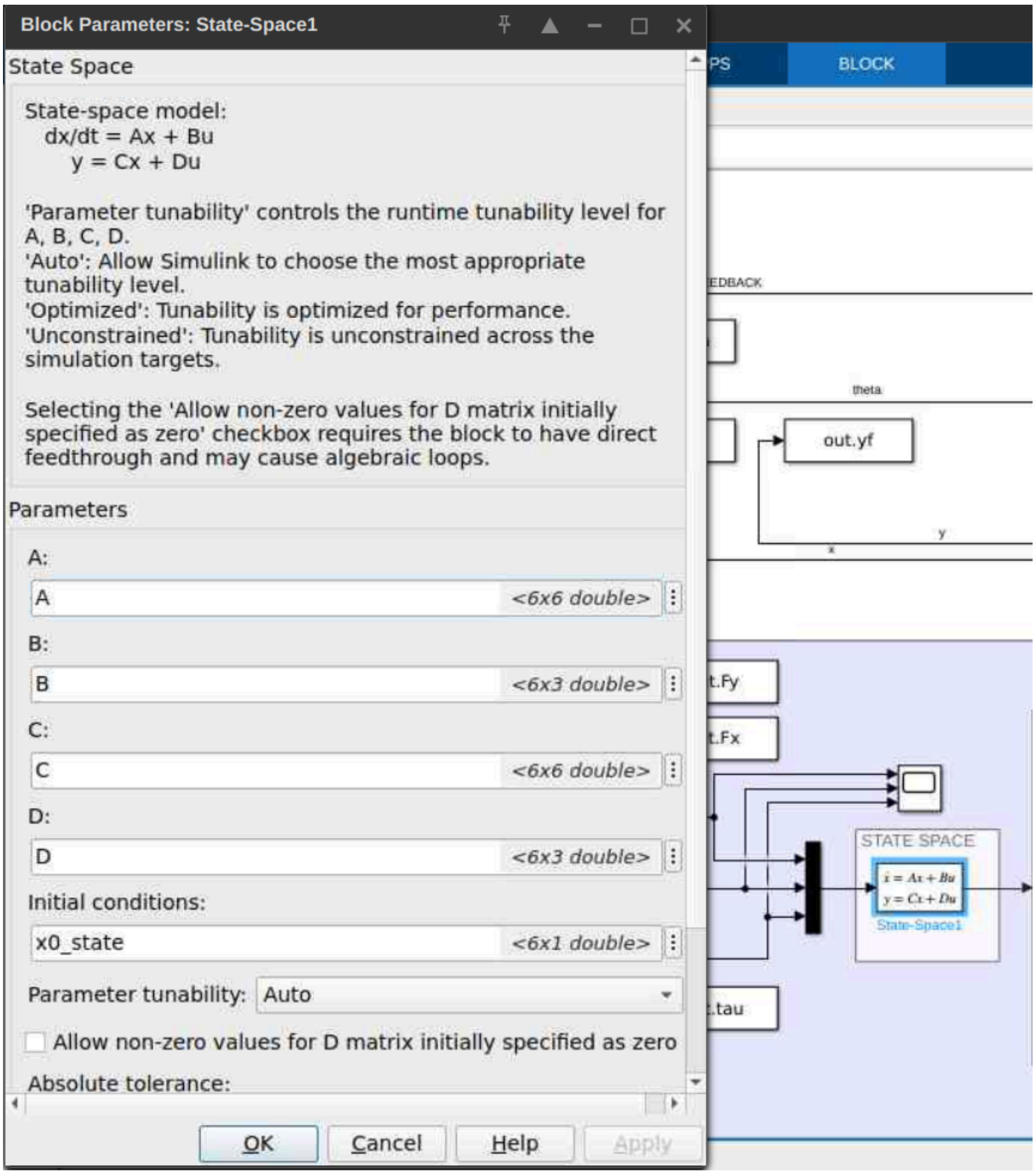
Now this code will create our state-space model and store all the relevant information. I will be using this model in SIMULINK with a state-space block. This will complete system modeling. Now according to the requirement the initial conditions are needed to be random, hence I will be using following code:

Filename: random_initial_conditions.m

```
% Define the range for random values
x_range = [-20, 20]; % Range for x position in meters
y_range = [-20, 20]; % Range for y position in meters
theta_range = [-180, 180]; % Range for orientation in
degrees
% Generate random initial conditions using Monte
Carlo
x0 = (x_range(2) - x_range(1)) * rand() + x_range(1);
% Random x position
y0 = (y_range(2) - y_range(1)) * rand() + y_range(1);
% Random y position
theta0 = (theta_range(2) - theta_range(1)) * rand() +
theta_range(1); % Random orientation

% Convert theta to radians for further use
theta0_rad = deg2rad(theta0);
x0_state = [x0; y0; theta0_rad; 0; 0; 0];
```

The x0_state variable will be put in initial conditions for the state-space block in simulink

The following image is showing state-space block and it’s parameters:



Block Parameters: State-Space1

State Space

State-space model:
dx/dt = Ax + Bu
y = Cx + Du

'Parameter tunability' controls the runtime tunability level for A, B, C, D.
'Auto': Allow Simulink to choose the most appropriate tunability level.
'Optimized': Tunability is optimized for performance.
'Unconstrained': Tunability is unconstrained across the simulation targets.

Selecting the 'Allow non-zero values for D matrix initially specified as zero' checkbox requires the block to have direct feedthrough and may cause algebraic loops.

Parameters

A: A <6x6 double>

B: B <6x3 double>

C: C <6x6 double>

D: D <6x3 double>

Initial conditions: x0_state <6x1 double>

Parameter tunability: Auto

☐ Allow non-zero values for D matrix initially specified as zero

Absolute tolerance:

This concludes our system modelling.

4. Control Algorithm

System design will allow us to replicate the system dynamics, since we have chosen state space modeling the input states are Fx, Fy and Tau which correspond to thruster force in x and y and torque that needs to be generated by combination of 2 thrusters.

Now the input I want to give is initial random conditions and final position for docking moving forward I have assumed fixed final condition that is (x, y, θ) to be (0, 0, 0), However this can be changed to be any variable location as per need or can be generated randomly with random_initial_conditions.m file.

For achieving this we will need a controller and I am going for the simplest and most versatile controller PID controller.

4.1 PID Controller Design

A PID (Proportional-Integral-Derivative) controller is implemented for each degree of freedom: translational motion in the x and y directions, and rotational motion for orientation.

Control Law:

$$Fx(t) = Kpx * ex(t) + Kix * \int_0^t ex(\tau)d\tau + Kdx * d/dt\ ex(t)$$
$$Fy(t) = Kpy * ey(t) + Kiy * \int_0^t ey(\tau)d\tau + Kdy * d/dt\ ey(t)$$
$$\tau(t) = K\theta * e\theta(t) + Ki\theta * \int_0^t e\theta(\tau)d\tau + Kd\theta * d/dt\ e\theta(t)$$

Where:

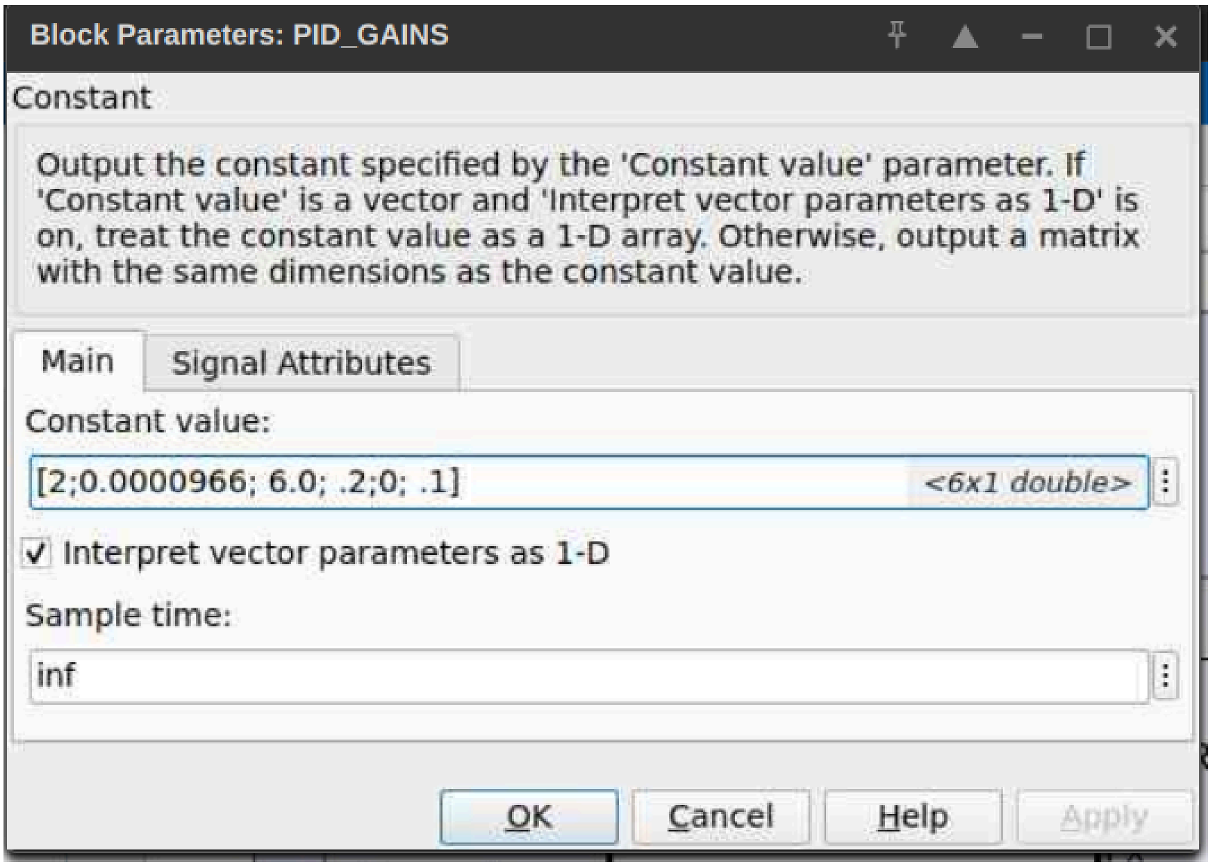
ex(t) = xtarget - xcurrent
ey(t) = ytarget - ycurrent
eθ(t) = θtarget − θcurrent

Kp, Ki, Kd: Proportional, Integral, and Derivative gains, respectively.

4.2 Tuning Gains

The following methods were explored:

Manual Tuning:
Initial gains were set to small values and incrementally adjusted.


Block Parameters: PID_GAINS

Constant

Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.

Constant value: [2;0.0000966; 6.0; .2;0; .1] <6x1 double>

[✓] Interpret vector parameters as 1-D

Sample time: inf

Ziegler-Nichols Method:
Provided a systematic starting point for tuning. However, I have not applied for it. Instead I used insight from this method to manually tune the gains.

I have written simple implementation using fmincon function which can be seen in the MATLAB files tuning.m and auto_tune_control.m

This implementation is written with various factors affecting the minimum and maximum values of available PID gains for use. This can help us in visualizing constraint based optimization for optimum values of gains covering various factors like cost, material strength, weight available and other similar factors.

4.3 Control Implementation

MATLAB Function:

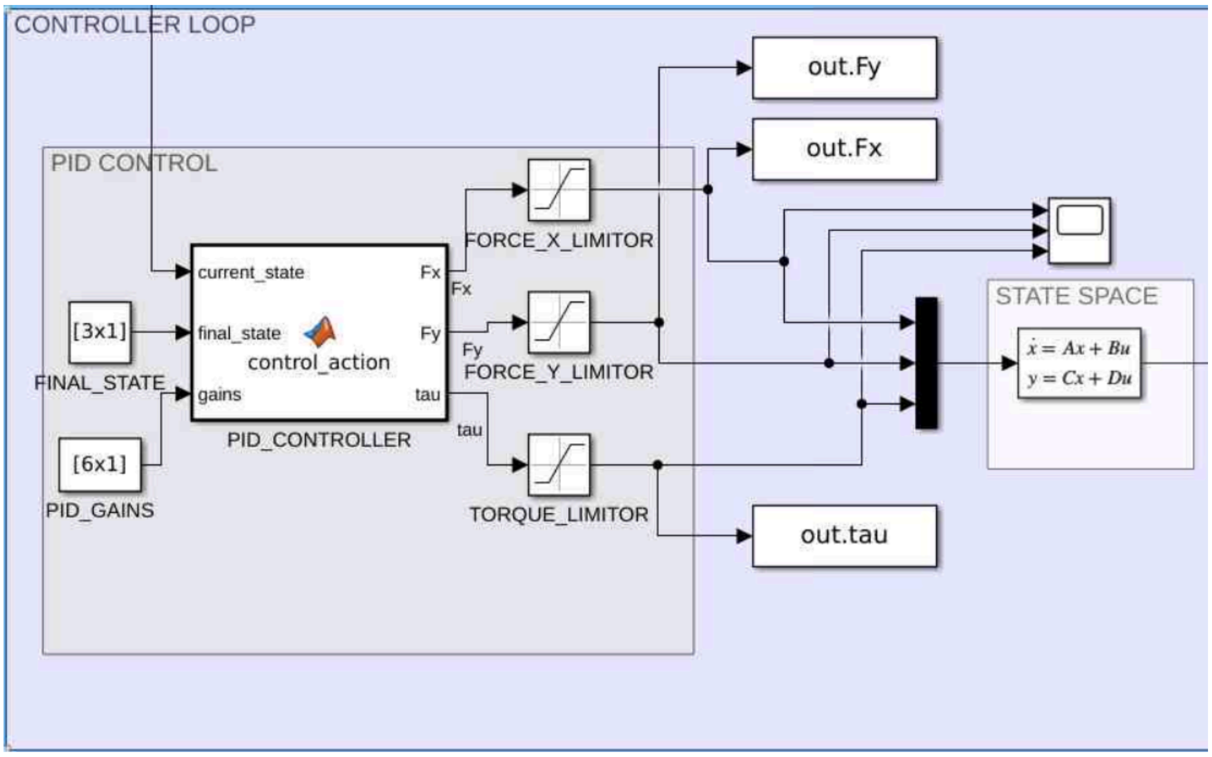
Used a MATLAB function block in simulink for implementation of PID controller design in section 4.1
Here is the code for control_action function which is used in PID control in simulink:

```
function [Fx, Fy, tau] =
control_action(current_state, final_state, gains)
    % Extract current and final positions and
orientation
```

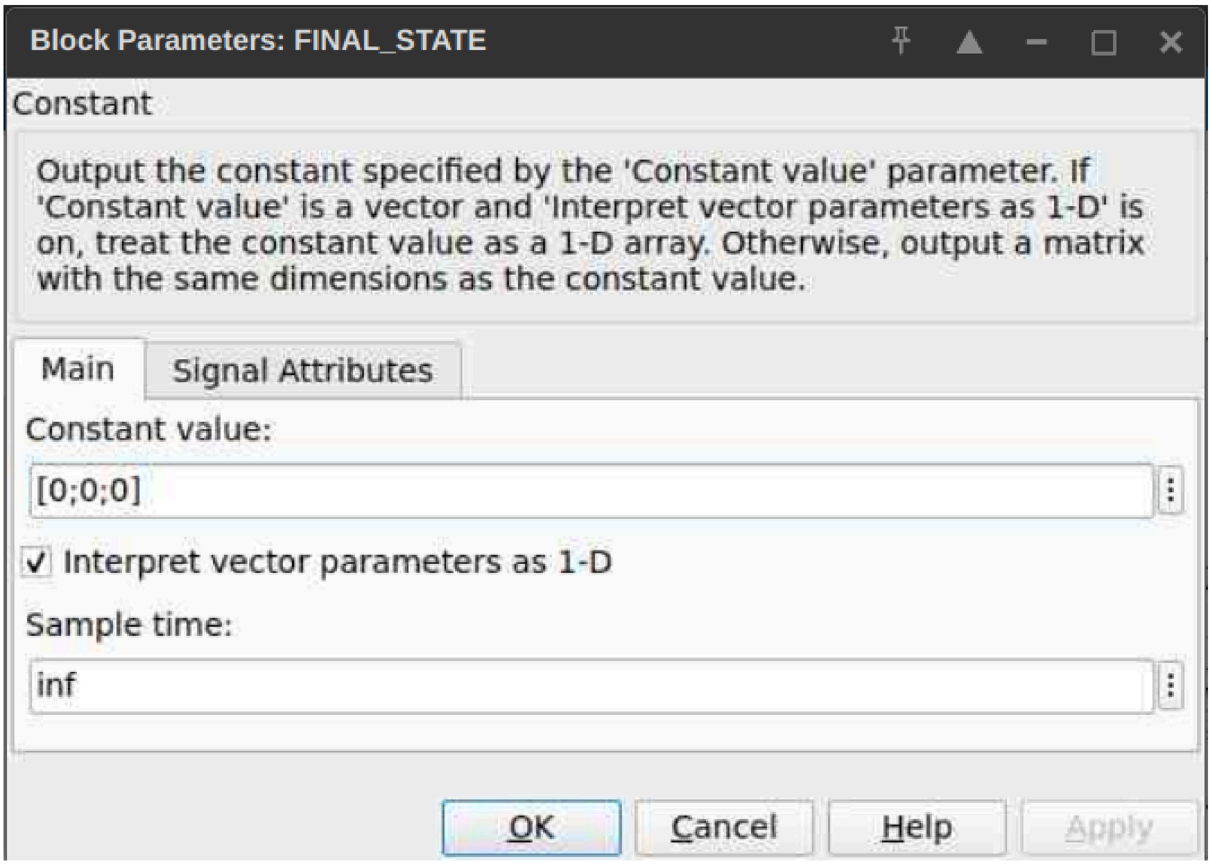
```
    persistent integral_x integral_y integral_theta
prev_ex prev_ey prev_etheta

    if isempty(integral_x)
        % Initialize integral and previous error
terms
        integral_x = 0;
        integral_y = 0;
        integral_theta = 0;
        prev_ex = 0;
        prev_ey = 0;
        prev_etheta = 0;
    end

    % Current state values
    x_current = current_state(1);
    y_current = current_state(2);
    theta_current = current_state(3);
    x_dot_current = current_state(4);
    y_dot_current = current_state(5);
    theta_dot_current = current_state(6);

    % Final state (target position and orientation)
    x_final = final_state(1);
    y_final = final_state(2);
    theta_final = final_state(3);

    % Control gains
    Kp = gains(1); Ki = gains(2); Kd = gains(3);
    K_theta = gains(4); Ki_theta = gains(5); K_omega
= gains(6);
```

```
    % Compute errors
    ex = x_final - x_current;
    ey = y_final - y_current;
    etheta = theta_final - theta_current;
    dt=0.01;

    % Integral of errors (accumulating errors)
    integral_x = integral_x + ex * dt;
    integral_y = integral_y + ey * dt;
    integral_theta = integral_theta + etheta * dt;

    % Derivative of errors (rate of change of error)
    de_x = (ex - prev_ex) / dt;
    de_y = (ey - prev_ey) / dt;
    de_theta = (etheta - prev_etheta) / dt;



    % PID control inputs for forces and torque
    Fx = Kp * ex + Ki * integral_x + Kd * de_x;
    Fy = Kp * ey + Ki * integral_y + Kd * de_y;
    tau = K_theta * etheta + Ki_theta *
integral_theta + K_omega * de_theta;

    % Update previous errors for next time step
    prev_ex = ex;
    prev_ey = ey;
    prev_etheta = etheta;
end
```

Following is simulink snippet for the controller:



The following snippet shows the FINAL_STATE:


Block Parameters: FINAL_STATE

Constant

Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.

Main | Signal Attributes

Constant value:

[0;0;0]

[✓] Interpret vector parameters as 1-D

Sample time:

inf

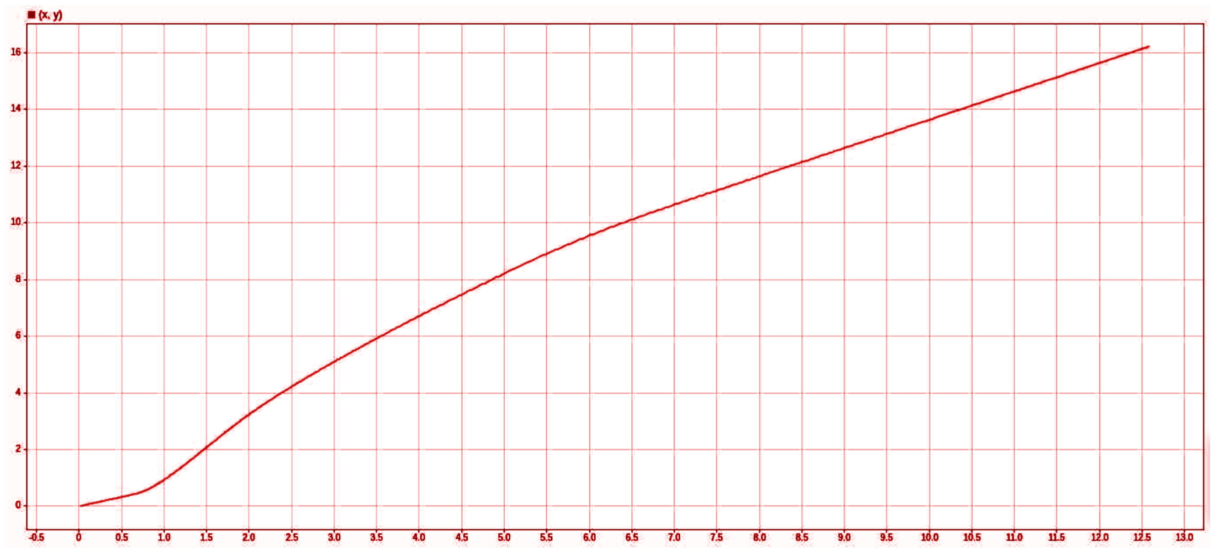
5. Path Planning

5.1 Trajectory Optimization

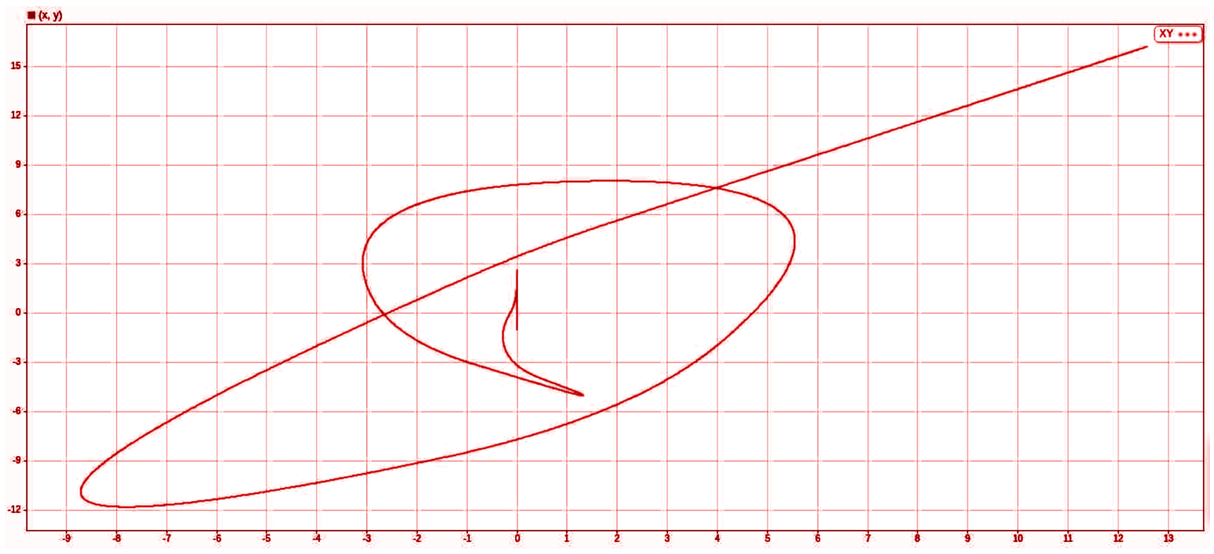
An optimal path is required to move the Chaser from a random initial position and orientation to the target position (0,0,0). The trajectory is planned in both position and velocity space.

Here I have tried to do the optimization in position and velocity profiles, for position optimization I have tried to adjust my gains for best and fastest trajectory.

Since while tuning the gains I came across more than one feasible option for the set of gains. However not all them took the fastest path for example with the gains in section 4.2 with some random initial condition generated as per section 3.3:



Now if I adjust the gains a bit for example increasing Kp to 10 will give following graph:



Currently for path optimization I have tried to find the best possible gains to get the optimum path for minimum deviation in path, now there are various other options that can give us different paths based on requirement we can adjust trajectory.
For example, if we want a roundabout way instead of directly going, then we can try different options.

This is simple optimization and did not involve any technique, however I propose using Artificial Potential Field(APF) method which works nicely for getting optimum trajectory with obstacle avoidance I have not implemented it here due to time and resource limitations, but I have worked on it and details can be found here:

https://github.com/HemantP02/apf_pathp_02

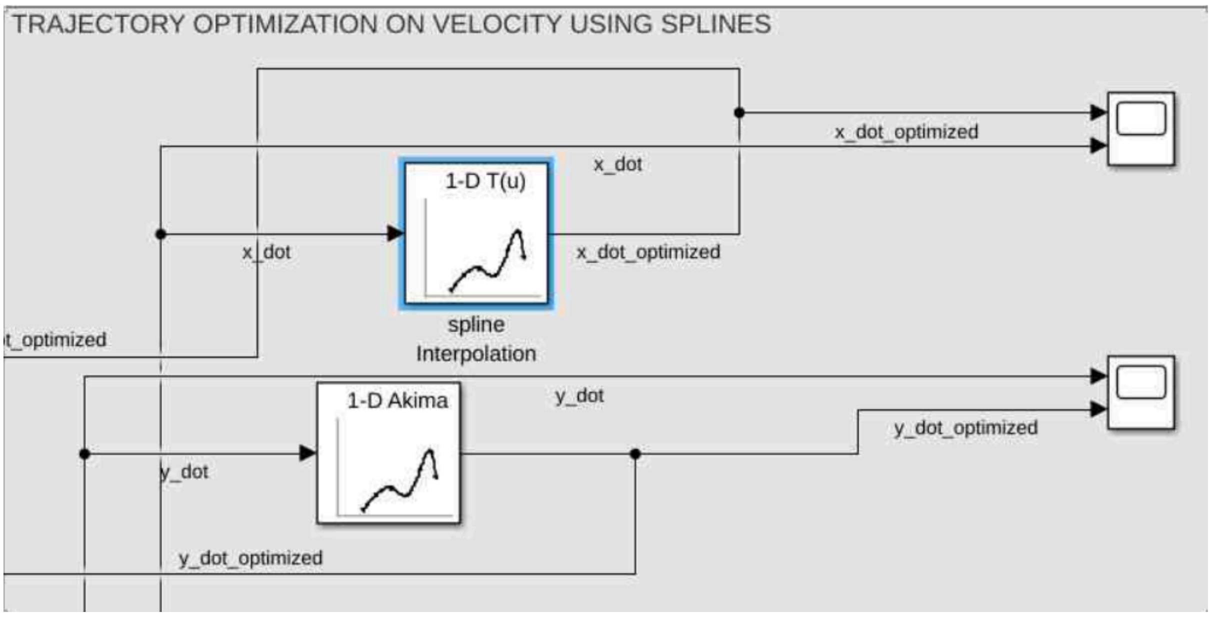
5.2 Velocity optimization Algorithm

A polynomial trajectory planning approach is chosen. At this stage after getting optimum trajectory based on our requirement I adjusted the gains and recorded the velocity profile,
However I got sharp corners in the velocity profile that will lead to difficulties in tracking hence,
I am using spline interpolation to get the smoother velocity profile and finally feeding the improved velocities in the feedback loop for current state in control_function.

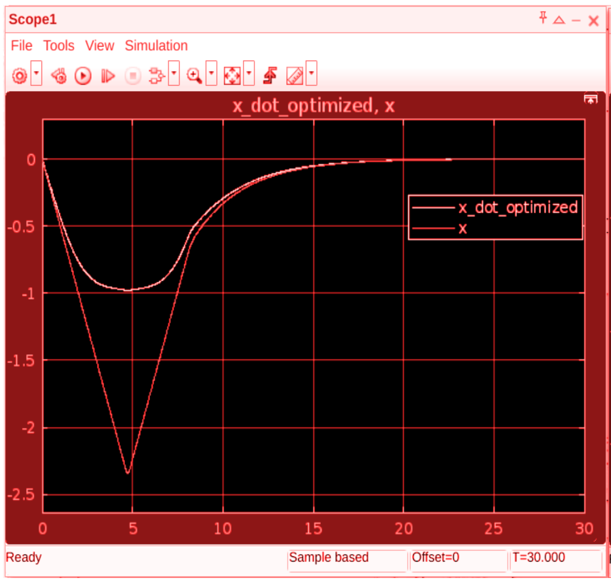
The trajectory is expressed as a cubic polynomial for position components x(t) and y(t):

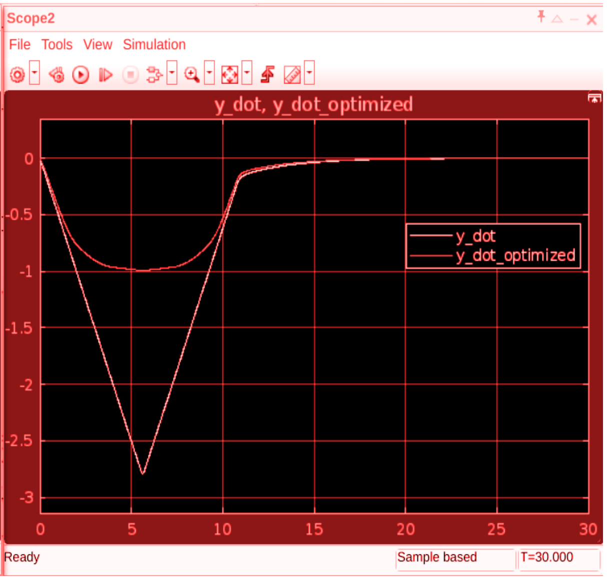
$x(t) = a_0 + a_1t + a_2t^2 + a_3t^3$
$y(t) = b_0 + b_1t + b_2t^2 + b_3t^3$

Here is the simulink snippet for velocity optimization:

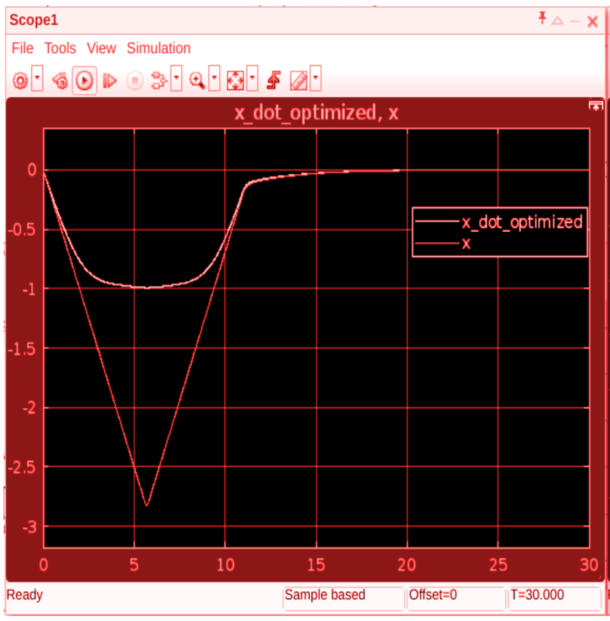


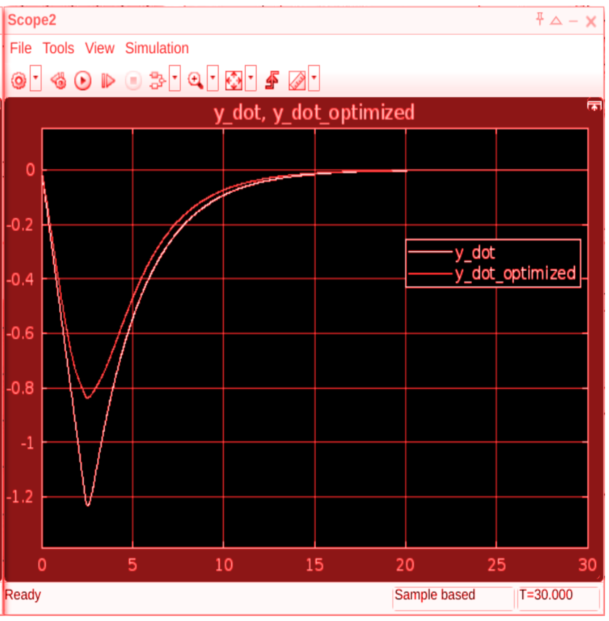
Now here is basic comparison between improved and default velocity profiles:





Here is one more example with a different random condition:





This concludes our trajectory and velocity optimization.

6. Visualization

The docking process is animated to visualize the Chaser Cart's motion.

Visualization Objectives:

Display the 2D path of the Chaser.
Animate the orientation changes.

6.1 Docking Animation

The following components are animated:

Chaser Position: The (x, y) coordinates are plotted over time.

Orientation Indicator: A quiver plot shows the current orientation θ(t).

I have created a MATLAB script for this process
Filename:animation.m

Here is one example:

docking_animation_01

Here is another example with different initial condition:

docking_animation_02

6.2 Implementation Details

Here is animation.m file:

```
% data (simulation results)
time = out.theta;  % Time
x_state = out.xf;  % x positions
y_state = out.yf; % y positions
theta_state = out.theta;  % orientation (radians)
% Target position
x_target = 0;
y_target = 0;
theta_target = 0;
% Plot setup
figure;
axis([-20 20 -20 20]);  % Set axis limits
hold on;
grid on;
% Plot the target
plot(x_target, y_target, 'ro', 'MarkerSize', 10, 'LineWidth', 2);  % Target
in red
% Initialize plot objects for the Chaser
chaser_body = plot(0, 0, 'gX-', 'MarkerSize', 8, 'LineWidth', 1);  % Chaser
path
chaser_orientation = quiver(0, 0, 0, 0, 0, 'Color', 'r', 'MaxHeadSize', 4);
% Orientation arrow
% Animation loop
for k = 1:length(time)
   % Extract current state
   x = x_state(k);
   y = y_state(k);
   theta = theta_state(k);

   % Update Chaser position
   set(chaser_body, 'XData', x_state(1:k), 'YData', y_state(1:k));  %
Trajectory

   % Update Chaser orientation
   % Compute orientation vector (length of arrow for visualization)
   arrow_length = 4;  % Arbitrary scale for visualization
   dx = arrow_length * cos(theta);
   dy = arrow_length * sin(theta);
   set(chaser_orientation, 'XData', x, 'YData', y, 'UData', dx, 'VData', dy);
   % Pause to simulate real-time animation
   pause(0.02);
end
% Add labels
title('Docking Process Animation');
xlabel('X Position (m)');
ylabel('Y Position (m)');
legend('Target Position', 'Chaser Trajectory');
```

Now to reach at this stage following process needs to be followed:

1. Run the state_space_01.m file.
2. Run the random_initial_conditions.m file.
3. Above steps will set up all the initial conditions and variable data for simulink model to run.
4. Now open the t_11.slx file and run the simulation with fixed step in model parameters and value of 0.01 for step size.
5. Run the simulation for 20 to 30 seconds more than 10 seconds is preferred.
6. Now everything is ready and finally animation.m can be run for the visualization.

This concludes the implementation of tasks from my end next we will go into technical insights and additional task sections.

7. Additional Tasks

7.1 Actuator and Sensor Hardware Selection

7.1.1 Theoretical Description

Actuators:

- **Thrusters:** For translation motion, small cold gas thrusters or electric propulsion systems(e.g., ion thrusters) are suitable. Cold gas thrusters are simple and reliable, while ion thrusters offer higher efficiency but are more complex.
- **Reaction Wheels or Control Moment Gyros(CMGs):** For rotational control, reaction wheels are commonly used as they are simple and precise. CMGs offer higher torque but are more complex and expensive.
- **Magnetic Torquers:** These can be used for coarse attitude control in environments where magnetic fields are present(e.g., near earth).

Sensors:

- **Inertial Measurement Unit(IMU):** Combines accelerometers and gyroscopes to measure linear acceleration and angular velocity. Essential for state estimation.
- **Star Trackers:** Provide high-precision attitude determination by tracking stars. Useful for rotational state estimation.
- **LIDAR or Vision-Based Sensors:** For relative position and orientation estimation between the Chaser and Target. LIDAR offers high precision,

while vision-based systems are lighter and consume less power.

- **GPS :** For absolute position tracking in Earth-based simulations.

Selection Criteria:

- **Precision:** High precision is required for docking operations.
- **Reliability:** Components must be reliable, especially in space-like environments.
- **Power Consumption:** Low power consumption is critical for longer-duration environments.
- **Mass and Size:** Compact and lightweight components are preferred for minimizing the overall mass of the Chaser.

7.1.2 Mathematical Description

Actuators:

- **Thrusters:** The thrusters produce a force $\boldsymbol{F_i}$ in the direction of their orientation. The total force **F** and torque τ acting on the chaser can be expressed as:

$$\mathbf{F} = \sum_{i=1}^{n} F_i \hat{\mathbf{d}}_i$$

$$\tau = \sum_{i=1}^{n} (\mathbf{r}_i \times F_i \hat{\mathbf{d}}_i)$$

where:

- $\hat{\mathbf{d}}_i$ is the unit vector in the direction of the i-th thruster,
- $\mathbf{r}_i$ is the position vector of the i-th thruster relative to the Chaser's center of mass,
- n is the number of thrusters.

- **Reaction Wheels:** The angular momentum **H** is given by:

$$\mathbf{H} = I_w \omega_w$$

where:

- I_w is the moment of inertia of the wheel,
- ω_w is the angular velocity of the wheel.

The torque τ generated by the reaction wheel is:

$$\tau = \frac{d\mathbf{H}}{dt} = I_w \dot{\omega}_w$$

Sensors:

- **IMU:** The IMU measures linear acceleration **a** and angular velocity ω. The measured $\mathbf{a}_m$ is:

$$\mathbf{a}_m = \mathbf{a} + \mathbf{g} + \eta_a$$

where:

- $\mathbf{g}$ is the gravitational acceleration,
- η_a is the accelerometer noise.

The measured angular velocity ω_m is:

$$\omega_m = \omega + \eta_\omega$$

where η_ω is the gyroscope noise.

- **LIDAR/VISION Sensors:** These sensors measure the relative position $\mathbf{r}_{rel}$ and orientation $\mathbf{q}_{rel}$ between the chaser and target. The measurement model should have generally the following format:

$$\mathbf{r}_{rel,m} = \mathbf{r}_{rel} + \eta_r$$

$$\mathbf{q}_{rel,m} = \mathbf{q}_{rel} \otimes \eta_q$$

where:

- η_r is the position measurement noise,
- η_q is the orientation measurement noise (quaternion noise),
- $\otimes$ denotes quaternion multiplication.

7.2 Thruster Configuration

7.2.1 Theoretical Description

Configuration:

- **Four Thrusters in a Cross Configuration:** Placing at 90-degrees intervals around the Chaser allows for independent control of translational motion in the x and y directions, as well as rotational control(torque).
- **Redundancy:** Adding an additional thruster pair can provide redundancy, ensuring that the system can still operate if one thruster fails.

Justification

- **Control Authority:** The cross configuration provides full control authority over both translational and rotational degrees of freedom.
- **Simplicity:** This configuration is relatively simple to implement and control.

- **Redundancy:** Redundant thrusters increase system reliability, which is crucial for docking operations.

7.2.2 Mathematical Description

1. **Fundamental Thruster Configurations:** The thruster configuration must enable control over 3 DOF(translational x, y and rotational θ_z). Key configurations include:

a) Cross Configuration (4 Thrusters)

- **Placement:** Thrusters at 90° intervals (e.g., $+\hat{x}, +\hat{y}, -\hat{x}, -\hat{y}$).
- **Force-Torque Equations:**

$$\begin{bmatrix} F_x \\ F_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ r & -r & -r & r \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix}$$

  - $r =$ distance from CoM to thrusters.
  - **Advantage:** Full controllability with minimal thrusters.
  - **Disadvantage:** No redundancy if one thruster fails.

b) Hexagonal Configuration (6 Thrusters)

- **Placement:** Thrusters at 60° intervals.
- **Force-Torque Equations:**

$$\begin{bmatrix} F_x \\ F_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} \cos(0^\circ) & \cos(60^\circ) & \cos(120^\circ) & \cos(180^\circ) & \cos(240^\circ) & \cos(300^\circ) \\ \sin(0^\circ) & \sin(60^\circ) & \sin(120^\circ) & \sin(180^\circ) & \sin(240^\circ) & \sin(300^\circ) \\ r & r & r & r & r & r \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5 \\ F_6 \end{bmatrix}$$

  - **Advantage:** Redundancy (up to 2 thrusters can fail).
  - **Disadvantage:** Higher mass and power consumption.

c) Asymmetric Configurations

- **Example:** 3 thrusters arranged non-uniformly (e.g., two on $+\hat{x}$, one on $-\hat{y}$).

$$\begin{bmatrix} F_x \\ F_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & -1 \\ r_1 & -r_2 & r_3 \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \end{bmatrix}$$

  - **Advantage:** Flexibility in design for specific missions.
  - **Disadvantage:** Potential controllability gaps (e.g., inability to generate pure $+\hat{y}$ force).

2. Redundancy and Fault tolerance

To ensure robustness, the thruster system must tolerate failures. This requires:

- **Over-Actuation:** More thrusters than DOFs(e.g., 6 thrusters for 3 DOFs).
- **Control Reallocation:** Solving for remaining thrusters when one fails.

Example: If thruster F_1 fails in a cross configuration, the system becomes:

$$\begin{bmatrix} F_x \\ F_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & -1 \\ r & -r & r \end{bmatrix} \begin{bmatrix} F_2 \\ F_3 \\ F_4 \end{bmatrix}$$

- **Feasibility Check:** The rank of the modified thruster matrix must still be 3 for full controllability.

3. Control Allocation Strategies

Thrusters are often **over-actuated**(more actuators than DOFs). Control allocation maps desired forces/torques to individual thruster commands.

a) Pseudo-Inverse Method

$$\mathbf{F} = \mathbf{T}^{\dagger} \begin{bmatrix} F_x \\ F_y \\ \tau_z \end{bmatrix}$$

where $\mathbf{T}^{\dagger} = \mathbf{T}^T(\mathbf{T}\mathbf{T}^T)^{-1}$ is the Moore-Penrose pseudo-inverse.

- **Advantage:** Computationally simple.
- **Disadvantage:** Does not account for thruster saturation.

b) Constrained Optimization

Formulate as a quadratic program (QP):

$$\min_{\mathbf{F}} \left\| \mathbf{T}\mathbf{F} - \begin{bmatrix} F_x \\ F_y \\ \tau_z \end{bmatrix} \right\|^2 + \lambda \|\mathbf{F}\|^2$$

subject to:

$$F_{min} \leq F_i \leq F_{max}$$

- $\lambda = \text{fuel efficiency weighting factor.}$
- **Advantage:** Explicitly handles saturation and optimizes fuel use.

4. Thruster Placement Optimization

Optimal thruster placement maximizes torque while minimizing fuel use. A cost function for placement might include:

$$J = \alpha \cdot \text{Torque Authority} + \beta \cdot \text{Fuel Efficiency} + \gamma \cdot \text{Mass}$$

- **Torque Authority:** $\sum_{i=1}^{n} \|\mathbf{r}_i \times \hat{\mathbf{d}}_i\|$.
- **Fuel Efficiency:** $\sum_{i=1}^{n} \|F_i\|$.

5. Edge Case: Thruster Failure Modes

Failure Scenario	Controllability	Mitigation Strategy
Single thruster fails	Degraded but usable	Reallocate control to remaining thrusters.
Two opposing thrusters fail	Loss of translational control in one axis	Use rotational thrusters to “spin and translate”.
All thrusters on one side fail	Uncontrollable	Enter safe mode; deploy backup thrusters.

Mathematical Test for controllability:

System is controllable if rank(C) = 3,C = [B, AB, A^2B]
Where A and B are state-space matrices.

6. Advanced Configurations

- **Differential Thrust:** Use thruster asymmetrically for combined translation/rotation.
  $\tau_z = (F_{right} - F_{left}).r$
- **Pulsed Operation:** Use PWM to approximate variable thrust with on/off thrusters.
  $F_{eff} = F_{max} \cdot \frac{t_{on}}{t_{on} + t_{off}}$
- **Hybrid Systems:** Combine thrusters with reaction wheels for precision.

7. Practical Considerations

- **Thruster Delay:** Modeled as a first-order lag:
  - $\frac{dF_i}{dt} = \frac{1}{\tau}(F_{desired} - F_i)$
- **Minimum Impulse Bit:** Smallest achievable thrust increment(critical for docking).

7.3 Optimal Strategy for State Estimation

7.3.1 Theoretical Description

Sensor Fusion:

- **Kalman Filter:** A Kalman filter can be used to fuse data from IMU, star trackers, and LIDAR/vision sensors. The filter estimates the Chaser's state (position, velocity, orientation, and angular velocity) by combining noisy sensor data.
- **Extended Kalman Filter (EKF):** If the system dynamics are nonlinear, and EKF can be used to linearize the system around the current state estimate.
-

State Estimation Process:

1. **Prediction Step:** Use the system model to predict the next state based on the current state and control inputs.
2. **Update Step:** Update the state estimate using sensor measurements. The Kalman filter optimally combines the predicted state and the sensor data to produce a refined state estimate.

Challenges:

- **Sensor Noise:** Sensor noise can degrade the accuracy of the state estimate. The Kalman filter mitigates this by weighting the sensor data based on it's reliability.
- **Latency:** Sensor data may have latency, which can be accounted for in the filter design.
- **Calibration:** Sensors must be carefully calibrated to ensure accurate measurements.

7.3.2 Mathematical Description

Kalman Filter:

The Kalman filter consists of two steps:

1. **Prediction Step:**

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{A}\hat{\mathbf{x}}_{k-1|k-1} + \mathbf{B}\mathbf{u}_k$$
$$\mathbf{P}_{k|k-1} = \mathbf{A}\mathbf{P}_{k-1|k-1}\mathbf{A}^T + \mathbf{Q}$$

where:

- $\hat{\mathbf{x}}_{k|k-1}$ is the predicted state,
- $\mathbf{P}_{k|k-1}$ is the predicted covariance,
- $\mathbf{A}$ is the state transition matrix,
- $\mathbf{B}$ is the control input matrix,
- $\mathbf{Q}$ is the process noise covariance.

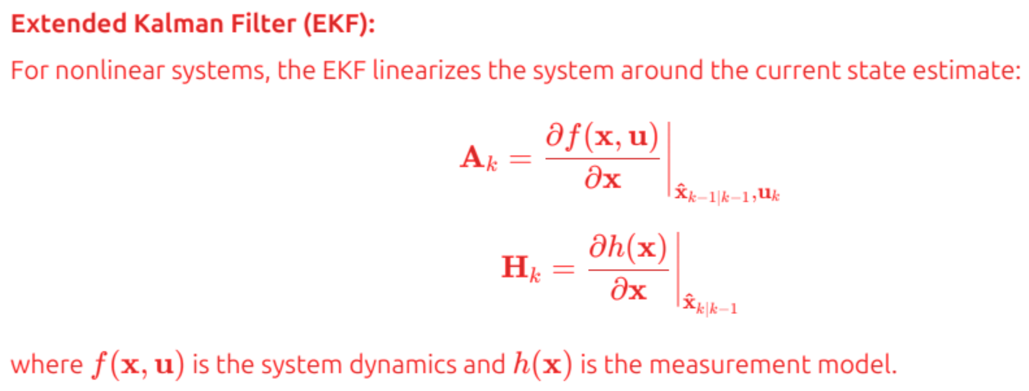
2. **Update Step:**

$$\mathbf{K}_k = \mathbf{P}_{k|k-1}\mathbf{H}^T(\mathbf{H}\mathbf{P}_{k|k-1}\mathbf{H}^T + \mathbf{R})^{-1}$$
$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k(\mathbf{z}_k - \mathbf{H}\hat{\mathbf{x}}_{k|k-1})$$
$$\mathbf{P}_{k|k} = (\mathbf{I} - \mathbf{K}_k\mathbf{H})\mathbf{P}_{k|k-1}$$

where:

- $\mathbf{K}_k$ is the Kalman gain,
- $\mathbf{H}$ is the measurement matrix,
- $\mathbf{R}$ is the measurement noise covariance,
- $\mathbf{z}_k$ is the measurement.

Extended Kalman Filter(EKF):



Extended Kalman Filter (EKF):

For nonlinear systems, the EKF linearizes the system around the current state estimate:

$$\mathbf{A}_k = \frac{\partial f(\mathbf{x}, \mathbf{u})}{\partial \mathbf{x}}\bigg|_{\hat{\mathbf{x}}_{k-1|k-1}, \mathbf{u}_k}$$

$$\mathbf{H}_k = \frac{\partial h(\mathbf{x})}{\partial \mathbf{x}}\bigg|_{\hat{\mathbf{x}}_{k|k-1}}$$

where $f(\mathbf{x}, \mathbf{u})$ is the system dynamics and $h(\mathbf{x})$ is the measurement model.

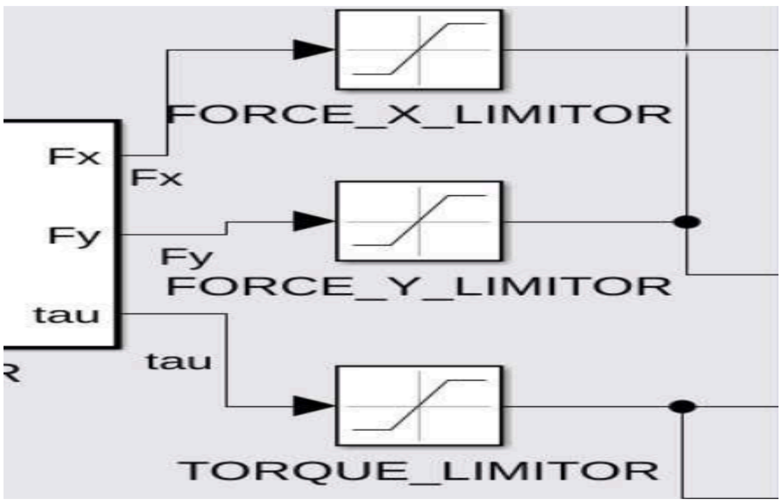
7.4 Practical Implementation Challenges

Actuator Limitations:

- **Actuation Delays:** The control input **u**(t) is delayed by τ:
  - $\mathbf{u}(t) = u_{desired}(t - \tau)$

  This can be compensated for using Smith predictor or predictive control techniques.
- **Maximum Thrust:** The thrusters have a saturation limit F_{max}:
  - $|F_i| \leq F_{max}$

  The control algorithm must ensure that the commanded thrust does not exceed this limit. For this I have applied saturation blocks in simulink as shown:



Sensor Noise and Calibration:

- **Noise Reduction:** Sensor noise η is typically modeled as Gaussian white noise with zero mean and covariance **R:**
  - $\eta \sim N(0, \mathbf{R})$

  The Kalman filter optimally reduces the effect of this noise.
- **Calibration:** Sensor calibration involves estimating the bias **b** and scale factor **S:**
  - $\mathbf{Z}_{calibrated} = \mathbf{S}(\mathbf{z}_{raw} - \mathbf{b})$

Practical Implementation: For practical implementation we will need a kinematic and dynamic model of the system whose input will be fed from the output of the state-space block and whose output will be fed to the controller for the calculation of feedback. This will bring all above mentioned challenges and hence their mitigation as required.

This finally brings me to the last part of the report and that is force profile and gas consumption. For this I have written a MATLAB code.

Basically I have taken output from the controller using the to and from workspace blocks in simulink for the F_x, F_y and τ.
Then they are mapped with respect to time, since gas consumption was one of the issues I tried to link it with activation of these variables.

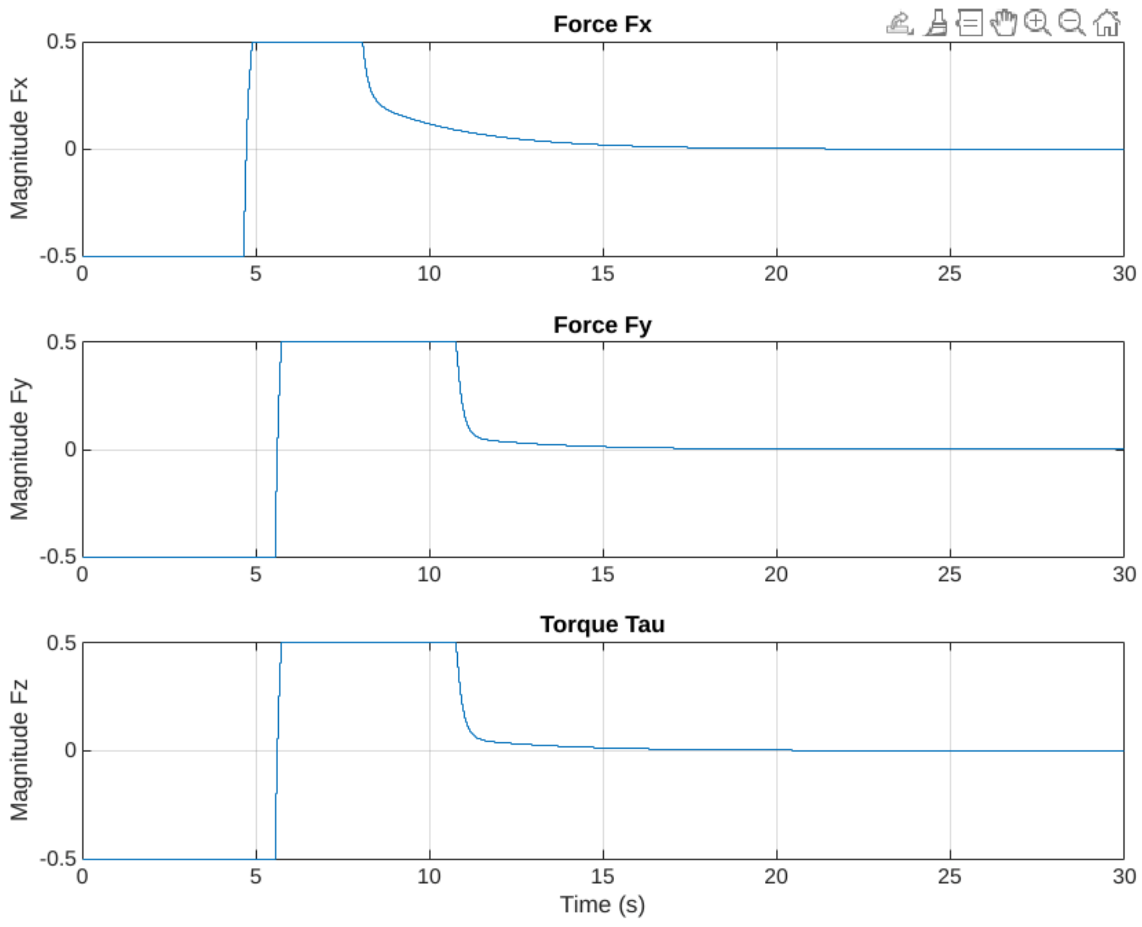
Taking the Minimum Impulse bit as activation of the thruster I have tried to calculate the gas consumption as we will see in the following MATLAB code.

Filename: thruster.m

```
% Thruster Profile and Gas Consumption Script
% Load Simulink output data (Fx, Fy, tau) from the workspace
% Ensure 'Fx', 'Fy', 'tau', and 'time' are variables in the MATLAB workspace
using to and from workspace blocks in simulink
% Configuration
gas_rate_per_thruster = 0.05;  % Gas consumption rate in grams per second
thrust_force = 0.5;            % Thruster force in Newtons
min_threshold = 0.00;           % Minimum force to activate a thruster
time_step = 0.01;              % Time step in seconds (adjust if different)
% Compute number of samples
num_steps = length(out.Fx);
% Initialize thruster profile and total gas consumed
thruster_profile = zeros(num_steps, 3);  % Columns for Fx, Fy, and Tau
thrusters
total_gas_consumed = 0;

% Compute thruster activations and gas consumption
for k = 1:num_steps
   if abs(out.Fx(k)) > min_threshold
       thruster_profile(k, 1) = out.Fx(k);  % Fx thruster activated
       total_gas_consumed = total_gas_consumed + gas_rate_per_thruster *
time_step;
   end
   if abs(out.Fy(k)) > min_threshold
       thruster_profile(k, 2) = out.Fy(k);  % Fy thruster activated
       total_gas_consumed = total_gas_consumed + gas_rate_per_thruster *
time_step;
   end
   if abs(out.tau(k)) > min_threshold
       thruster_profile(k, 3) = out.Fy(k);  % Tau thruster activated
       total_gas_consumed = total_gas_consumed + gas_rate_per_thruster *
time_step;
   end
end
% Display thruster profile and total gas consumed
disp('Thruster Profile (1 = Active, 0 = Inactive):');
disp('Columns: [Fx_thruster, Fy_thruster, Tau_thruster]');
disp(thruster_profile);
disp(['Total Gas Consumed: ', num2str(total_gas_consumed), ' grams']);
% Visualization of thruster firings
time = (0:num_steps-1) * time_step;
figure;
subplot(3, 1, 1);
stairs(time, thruster_profile(:, 1));
title('Thruster Firing for Fx');
```

```
ylabel('Activation (0 or 1)');
grid on;
subplot(3, 1, 2);
stairs(time, thruster_profile(:, 2));
title('Thruster Firing for Fy');
ylabel('Activation (0 or 1)');
grid on;
subplot(3, 1, 3);
stairs(time, thruster_profile(:, 3));
title('Thruster Firing for Tau');
ylabel('Activation (0 or 1)');
xlabel('Time (s)');
grid on;
```

Following is one of the example:



==

==

END OF THE REPORT

THANK YOU REACHING THE END