

MECH6910T Mid-Project Presentation

Aircraft Pitch Control via Traditional and Intelligent Methods

Alfiyandy HARIANSYAH

mahariansyah@connect.ust.hk

26-Nov-2024



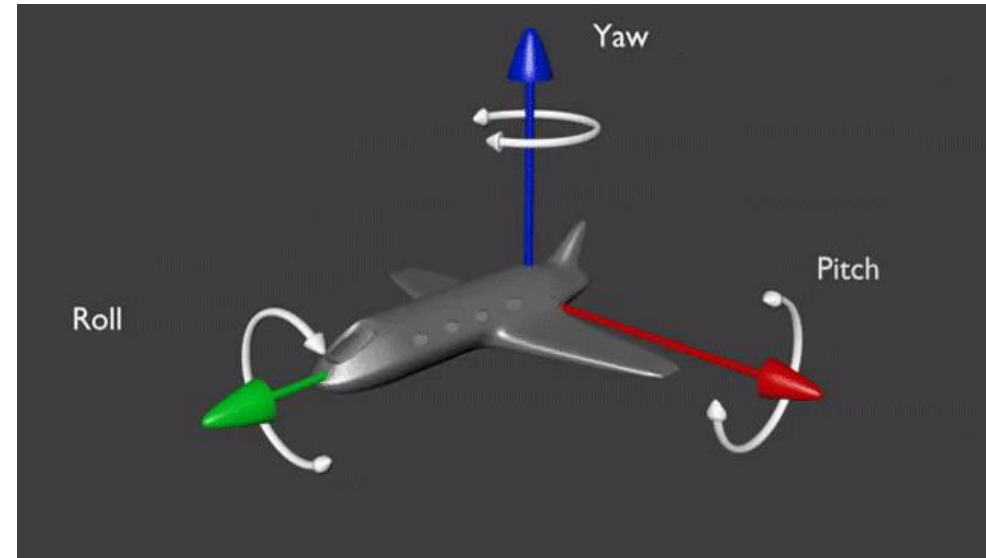
THE HONG KONG UNIVERSITY OF
SCIENCE AND TECHNOLOGY

DEPARTMENT OF MECHANICAL
AND AEROSPACE ENGINEERING

Introduction

Introduction: aircraft's motion

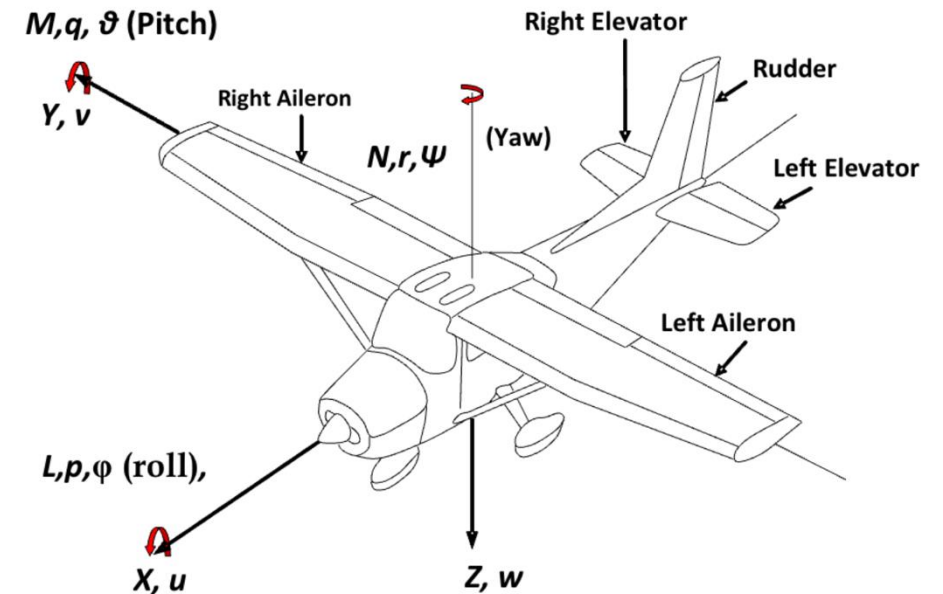
- In aviation, precise control over an aircraft's motion is important
 - ensuring stability, responsiveness, and safety during flight operations
- The aircraft's motion is complicated, coupled, and non-linear.
- It is described via a 6-DOF motion:
 - 3 translational motions (x, y, z)
 - horizontal, transverse, vertical
 - 3 rotational motions (ϕ, θ, ψ)
 - roll, pitch, yaw



(<https://pale.blue/2019/09/05/when-simulation-starts-with-motion/>)

Introduction: aircraft's control

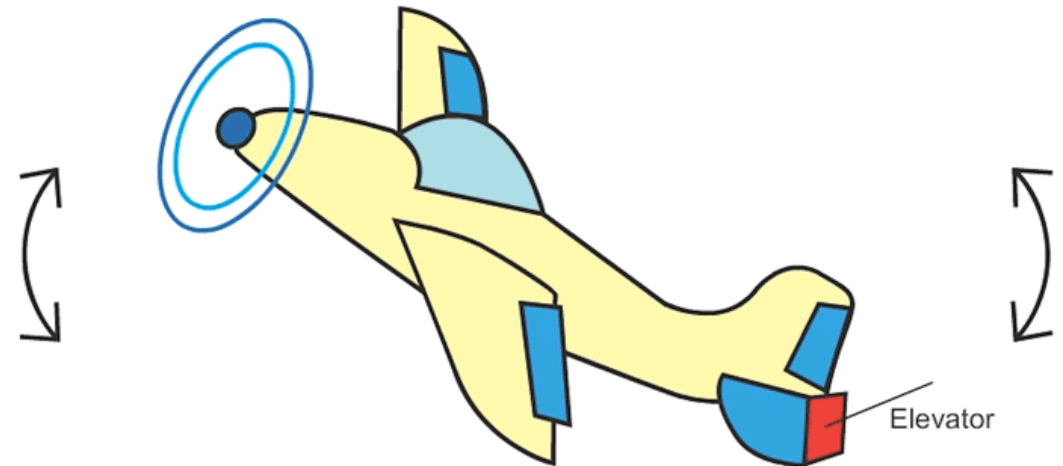
- The control over the aircraft's motions is achieved by:
 - deflecting control surfaces: elevator, aileron, rudder
 - regulating throttle
- Two uncoupled motions and controls:
 - longitudinal motion
 - states: pitch angle and velocity
 - controls: elevator and throttle
 - lateral motion
 - states: roll and yaw angles
 - controls: aileron and rudder
- We focus only on longitudinal motion



Typical aircraft motion with control surfaces
(Aliyu et al., 2015)



Introduction: aircraft pitch control

- When an aircraft needs to climb or descend, it must pitch up/down.
 - it is thus **important** to develop a controller for achieving the desired pitch angle
- In this project, aircraft pitch control is achieved by:
 - deflecting the elevator up/down
 - keeping the thrust constant
- Project specification
 - Aircraft: Cessna-172
 - Motion: longitudinal-only
 - Control methods:
 - traditional: static PID control
 - intelligent: reinforcement learning



***Aircraft pitch control
by deflecting the elevator***
(google.com)

Literature review

- Aircraft pitch control is not a new problem.
- Others have developed traditional controllers:
 - [Deepa et al., 2016](#): Proportional-Integral-Derivative
 - [Vishal et al., 2014](#): Linear-Quadratic-Regulator
 - [Arrosida et al., 2017](#): Observer-State-Feedback static gains
- Recent works attempted intelligent methods:
 - [Wahid et al., 2012](#): Fuzzy PID
 - [Liang et al., 2024](#): Artificial Neural Network
 - [Richter et al., 2024](#): Reinforcement Learning adaptive gains
- This project tries a new method: **PID + Reinforcement Learning**

Motivation: PID + RL

- PID + RL → PID with adaptive gain tuning by reinforcement learning
- The idea of adaptive tuning is not new, but few have attempted reinforcement learning algorithms (e.g., DDPG) to tune the PID gains specifically for aircraft pitch control.
- Adaptive tuning means the PID gains are changing over time.
- Why adaptive tuning?
 - Traditional PID with static gains might only be good at specific flight conditions
 - Robust across various flight conditions even in the event of a disturbance, e.g, gusts

Methodologies

Non-linear coupled 6-DOF aircraft motion

$$\mathbf{x} = (x, y, z, \phi, \theta, \psi, p, q, r, u, v, w)^T$$



12 state variables

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}) = \mathbf{f}(x, y, z, \phi, \theta, \psi, p, q, r, u, v, w)$$

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \\ \dot{u} \\ \dot{v} \\ \dot{w} \\ \dot{p} \\ \dot{q} \\ \dot{r} \end{pmatrix} = \begin{pmatrix} u C_{\theta} C_{\psi} + v (S_{\phi} S_{\theta} C_{\psi} - C_{\phi} S_{\psi}) + w (C_{\phi} S_{\theta} C_{\psi} + S_{\phi} S_{\psi}) \\ u C_{\theta} S_{\psi} + v (S_{\phi} S_{\theta} S_{\psi} + C_{\phi} C_{\psi}) + w (C_{\phi} S_{\theta} S_{\psi} - S_{\phi} C_{\psi}) \\ -u S_{\theta} + v S_{\phi} C_{\theta} + w C_{\phi} C_{\theta} \\ p + q S_{\phi} T_{\theta} + r C_{\phi} T_{\theta} \\ q C_{\phi} - r S_{\phi} \\ (q S_{\phi} + r C_{\phi}) / C_{\theta} \\ r v - q w + F_{\text{ext},x} / m \\ p w - r u + F_{\text{ext},y} / m \\ q u - p v + F_{\text{ext},z} / m \\ \frac{I_{xz} M_{\text{ext},z} + I_{zz} M_{\text{ext},x} + (I_{xx} I_{xz} - I_{xz} I_{yy} + I_{zx} I_{zz}) p q - (I_{xz}^2 + I_{zz}^2 - I_{yy} I_{zz}) q r}{I_{xx} I_{zz} - I_{xz} I_{zx}} \\ \frac{M_{\text{ext},y} - (I_{xx} - I_{zz}) p r - I_{zx} p^2 + I_{xz} r^2}{I_{yy}} \\ \frac{I_{xx} M_{\text{ext},z} + I_{zx} M_{\text{ext},x} + (I_{xx}^2 + I_{zz}^2 - I_{xx} I_{yy}) p q + (I_{zx} I_{yy} - I_{xz} I_{xx} - I_{zx} I_{zz}) q r}{I_{xx} I_{zz} - I_{xz} I_{zx}} \end{pmatrix}$$

} body-to-earth velocities
 $\vec{v}_E = \mathcal{L}_{B \rightarrow E} \vec{v}_B$

} body-to-earth angular rates

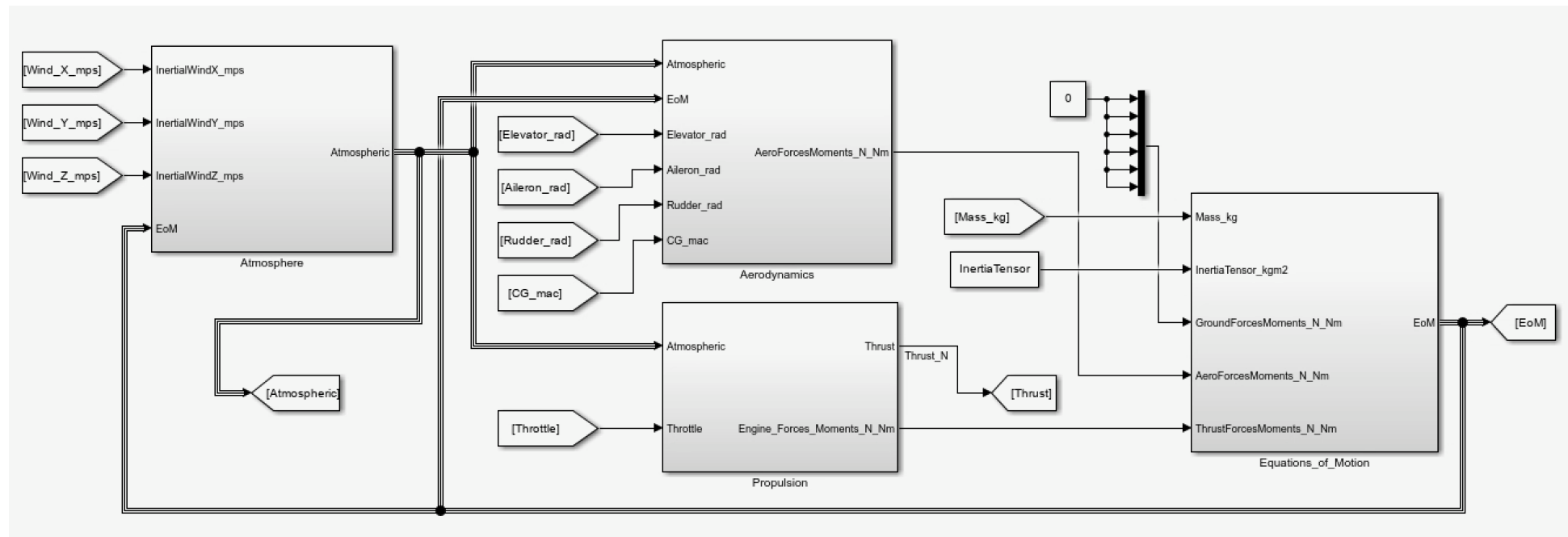
} translational force equations
 $\vec{F}_{\text{ext}} = m (\vec{v}_B + \vec{\Omega}_B \times \vec{v}_B)$

} rotational moment equations
 $\vec{M}_{\text{ext}} = I_B \dot{\vec{\Omega}}_B + \vec{\Omega}_B \times (I_B \vec{\Omega}_B)$

Simulink implementation

Four modules developed from scratch

- Atmosphere : models the air using international standard atmosphere (ISA)
- Aerodynamics : models the aerodynamics using published derivative values
- Propulsion : models the generic propulsion
- Equations_of_Motion : models the non-linear coupled 6-DOF aircraft motion



Trim condition

- Assumption → level unaccelerated flight at 5000 ft.

I used the Matlab trim function
to solve these equations

- The trim condition solves: $\dot{\mathbf{x}}_0 = \mathbf{f}(\mathbf{x}_0) = 0$



12 state variables

$$\mathbf{x} = (x, y, z, \phi, \theta, \psi, p, q, r, u, v, w)^T$$



trimmed state values

$$\mathbf{x}_0 = (0, 0, -1524, 0, 0, 0, 62.3866, 0, 0, 0, 0, 0)^T$$

4 control variables

$$\mathbf{u} = (\delta_e, \delta_a, \delta_r, \delta_t)^T$$



trimmed control values

$$\mathbf{u}_0 = (-0.0032115, 0, 0, 0.6792)^T$$

Linearized equations: complete motions

- Assumes a small deviation from the previously trimmed states:

$$\mathbf{x} = \mathbf{x}_0 + \Delta \mathbf{x}$$

- Linearization is performed in Matlab to yield the following equations:

$$\Delta \dot{\mathbf{x}} = A \Delta \mathbf{x} + B \Delta \mathbf{u}$$

$$\Delta \mathbf{y} = C \Delta \mathbf{x} + D \Delta \mathbf{u}$$

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 62.39 & 0 & 1.0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -62.39 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 \\ 0 & 0 & -0.0001 & 0 & -9.807 & 0 & -0.0477 & 0 & 0.2238 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9.807 & 0 & 0 & 0 & -0.1582 & 0 & -0.103 & 0 & -61.8 \\ 0 & 0 & -0.0022 & 0 & 0 & 0 & -0.3152 & 0 & -2.64 & 0 & 60.9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.3765 & 0 & -11.57 & 0 & 2.272 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.0005 & 0 & -0.2494 & 0 & -3.971 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.137 & 0 & -0.3595 & 0 & -1.159 \end{pmatrix} \quad B = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1.91 & 0 & 0 & 1.462 \\ 0 & 0 & 5.953 & 0 \\ -13.69 & 0 & 0 & 0.0255 \\ 0 & -50.19 & 3.178 & 0 \\ -33.99 & 0 & 0 & -0.0146 \\ 0 & -7.202 & -8.754 & 0 \end{pmatrix} \quad C = \begin{pmatrix} 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.0 \end{pmatrix} \quad D = \mathbf{0}$$

Linearized equations: longitudinal and lateral

- The 12 linear equations can be decomposed into two sets of equations:
 - 6 longitudinal equations of motion

$$\Delta \dot{\mathbf{x}} = A_{\text{long}} \Delta \mathbf{x} + B_{\text{long}} \Delta \mathbf{u}$$

$$\Delta y = C_{\text{long}} \Delta \mathbf{x} + D_{\text{long}} \Delta \mathbf{u}$$

- 6 lateral equations of motion

$$\Delta \dot{\mathbf{x}} = A_{\text{lat}} \Delta \mathbf{x} + B_{\text{lat}} \Delta \mathbf{u}$$

$$\Delta y = C_{\text{lat}} \Delta \mathbf{x} + D_{\text{lat}} \Delta \mathbf{u}$$

- We focus only on longitudinal equations since they govern the aircraft pitch attitude

$$A_{\text{long}} = \begin{pmatrix} 0 & 0 & 0 & 1.0 & 0 & 0 \\ 0 & 0 & -62.39 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.0 \\ 0 & -0.0001 & -9.807 & -0.0477 & 0.2388 & 0 \\ 0 & -0.0022 & 0 & -0.3152 & -2.64 & 60.9 \\ 0 & 0 & 0 & 0.0005 & -0.2494 & -3.971 \end{pmatrix}$$

$$B_{\text{long}} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1.91 & 1.462 \\ -13.69 & 0.0255 \\ -33.99 & -0.0146 \end{pmatrix}$$

$$C_{\text{long}} = (0 \ 0 \ 1.0 \ 0 \ 0 \ 0)$$

$$D_{\text{long}} = (0 \ 0)$$

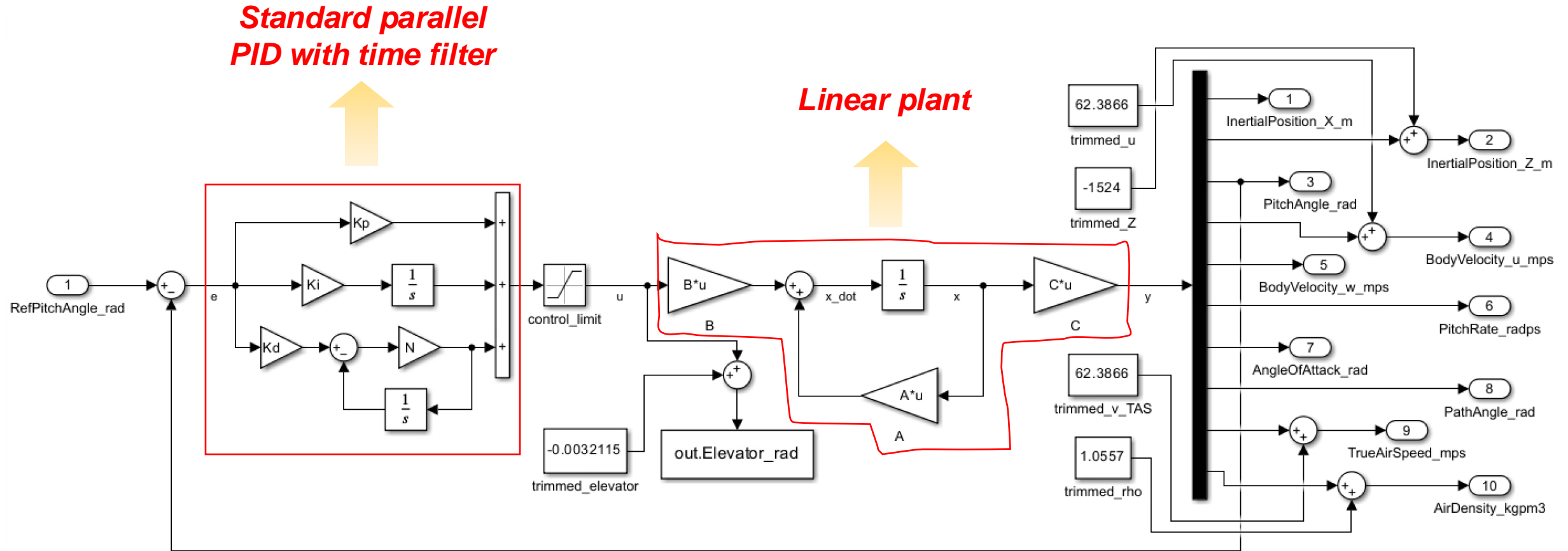
$$A_{\text{lat}} = \begin{pmatrix} 0 & 0 & 62.39 & 1.0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.0 \\ 0 & 9.807 & 0 & -0.1582 & -0.103 & -61.8 \\ 0 & 0 & 0 & -0.3765 & -11.57 & 2.272 \\ 0 & 0 & 0 & 0.137 & -0.3595 & -1.159 \end{pmatrix}$$

$$B_{\text{lat}} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & -5.953 \\ -50.19 & 3.178 \\ -7.202 & -8.754 \end{pmatrix}$$

$$C_{\text{lat}} = (0 \ 0 \ 1.0 \ 0 \ 0 \ 0)$$

$$D_{\text{lat}} = (0 \ 0)$$

Traditional PID controller with static gain



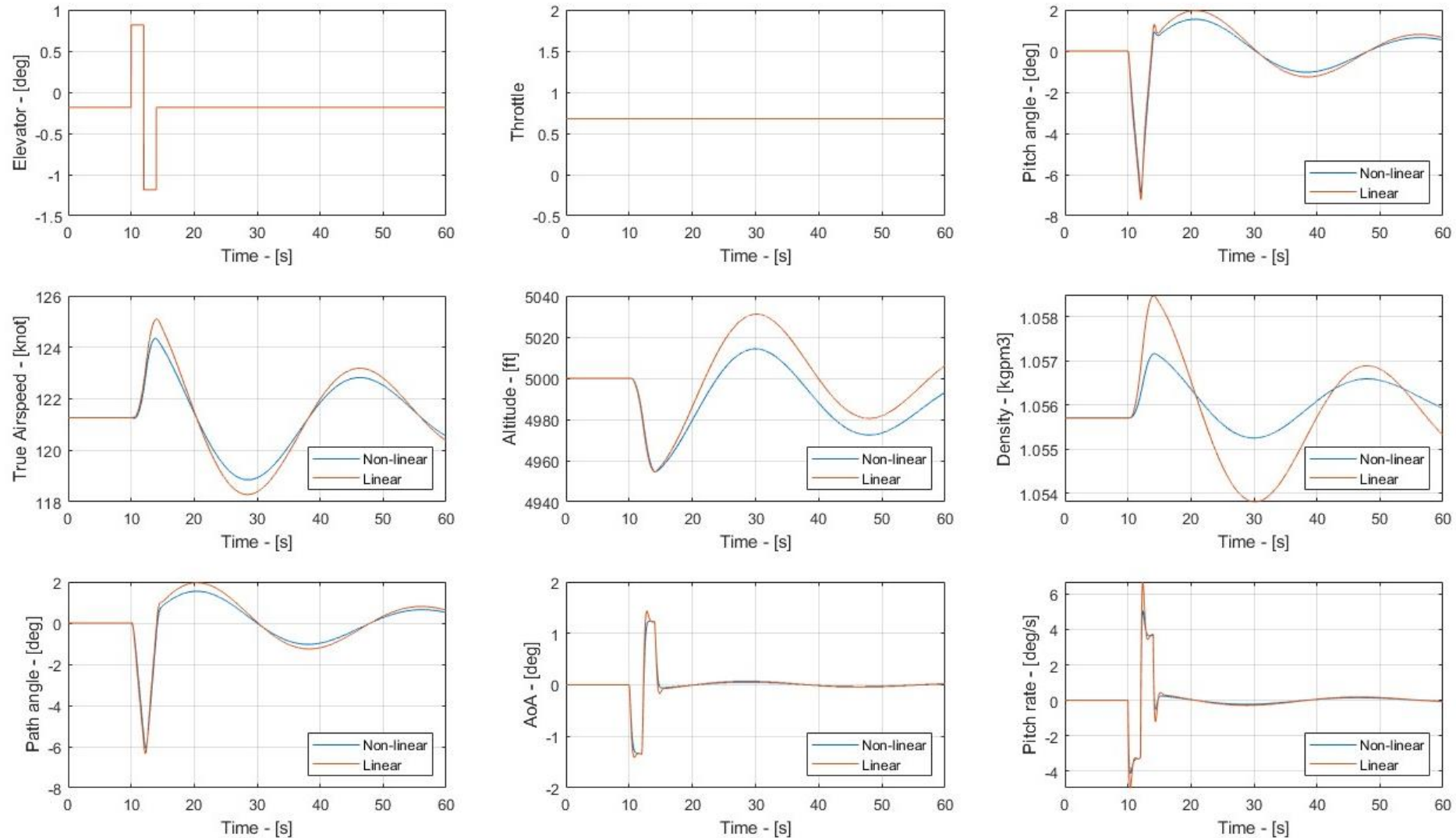
$$C_{\text{PID}}(s) = K_P + \frac{K_I}{s} + \frac{K_D s}{s/N + 1}$$

$$\begin{aligned} \Delta \dot{\mathbf{x}} &= A_{\text{long}} \Delta \mathbf{x} + B_{\text{long}} \Delta \mathbf{u} \\ \Delta y &= C_{\text{long}} \Delta \mathbf{x} + D_{\text{long}} \Delta \mathbf{u} \end{aligned}$$

Results and Discussion

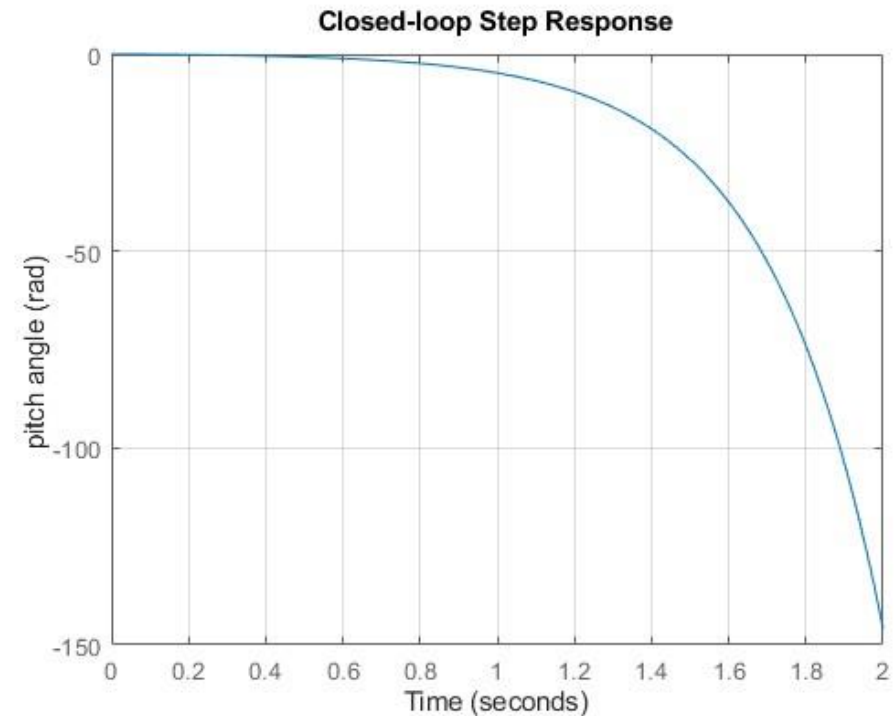
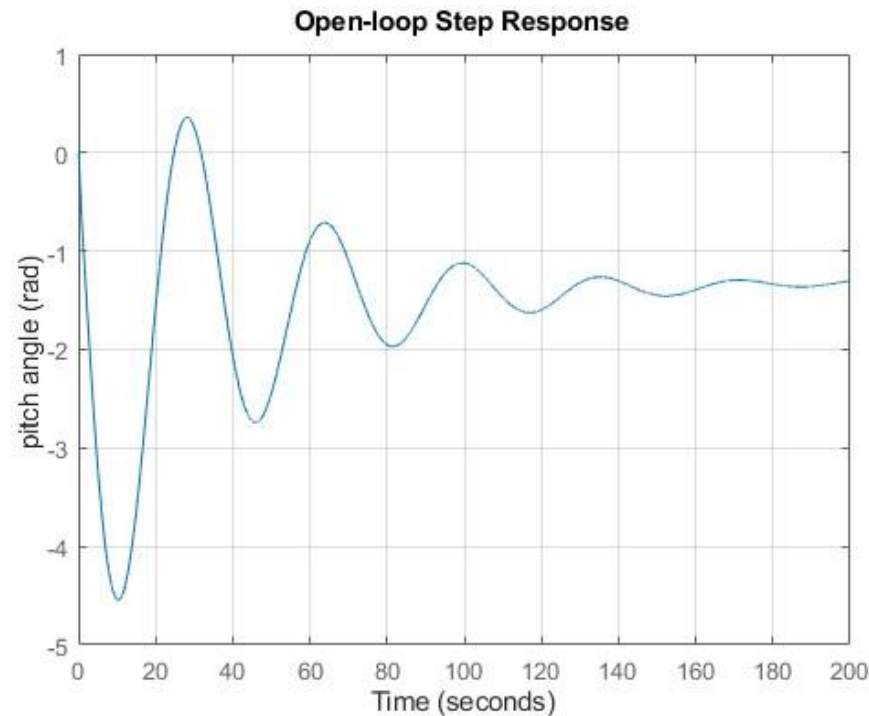
Verification of linear model: elevator doublet

Longitudinal dynamics under elevator doublet



Open-loop and closed-loop responses

The open-loop and closed-loop (with unity feedback) systems are checked against a step response of 0.2 rad (≈ 11.46 deg).



*The open loop is marginally stable and has a large steady-state error.
The closed loop with unity feedback makes the system unstable.*

Time-domain performance

Controller design specification

Under a 0.2-radian step input, it should have:

- Rise time < 2 seconds
- Settling time < 10 seconds
- Overshoot < 10%
- Steady-state error < 2%

PID_1: fastest rise/settling time, but largest overshoot

PID_5: slowest rise/settling time, but smallest overshoot

PID_4: the compromise which satisfies all the specifications

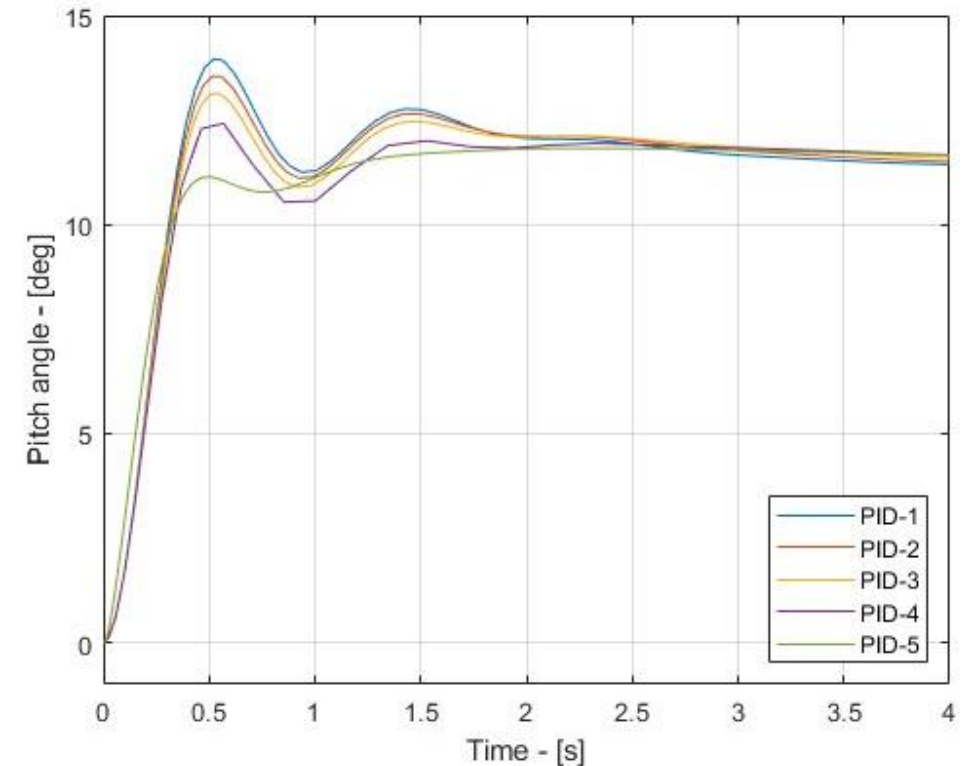


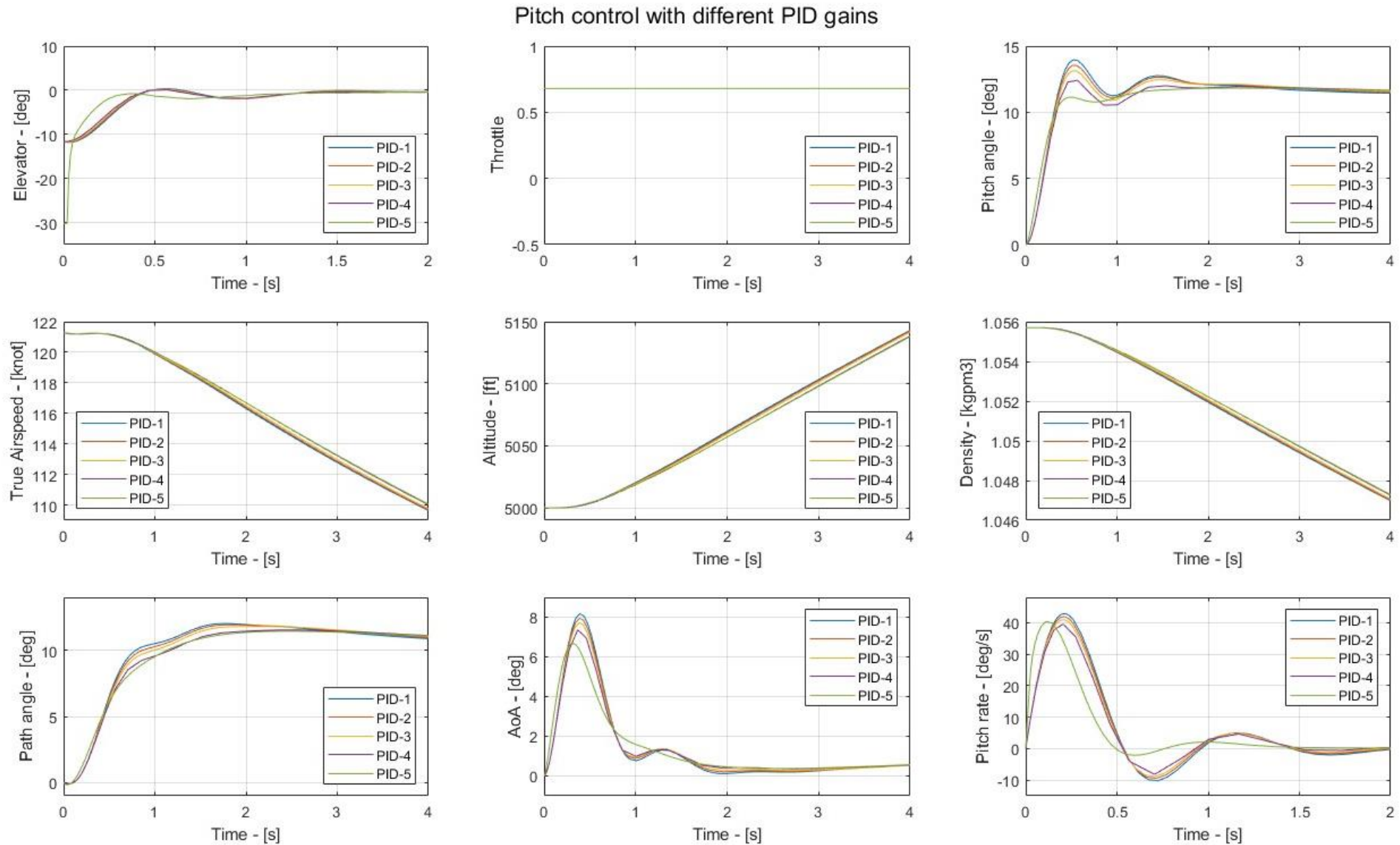
TABLE I: Time-domain performance of PID with different gains

PID Name	K_P	K_I	K_D	Rise Time (s)	Settling Time (s)	Overshoot (%)
PID_1	-1	-1	0	0.2378	2.9133	21.7818
PID_2	-1	-0.8	0	0.2449	3.2556	18.3613
PID_3	-1	-0.6	0	0.2512	3.5790	14.7780
PID_4	-1	-0.3	0	0.2639	3.8517	9.0304
PID_5	-1	-0.3	-0.1	0.3476	4.0409	3.9849

Might need to compare the requested control input!

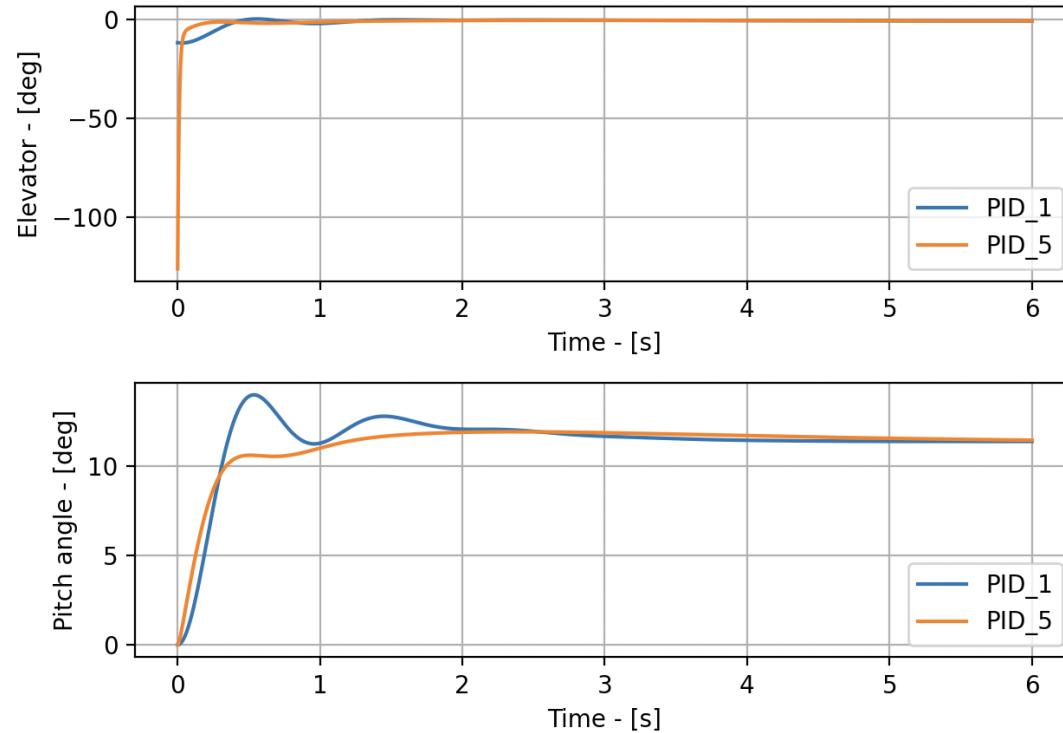
PID_5 initially seems to request a very high control input due to the presence of a derivative term.

System responses under different PID gains

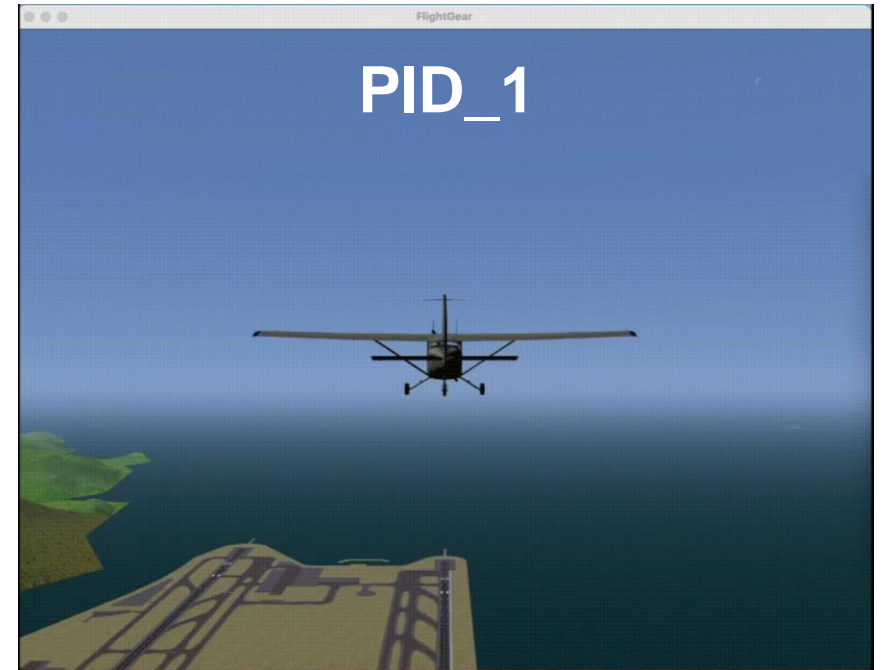


Flightgear visualization

System response: PID_1 vs PID_5



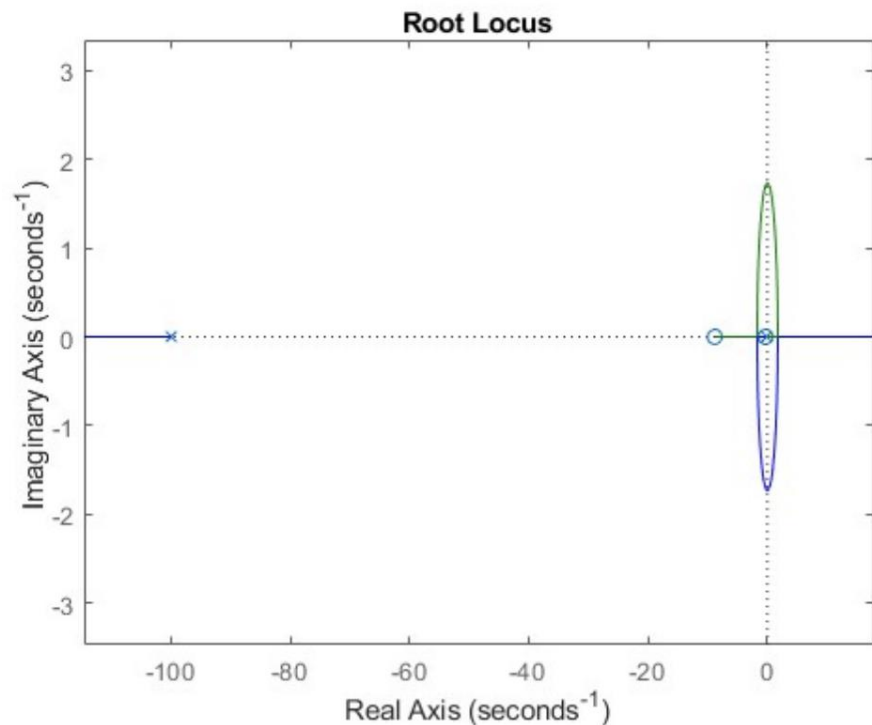
PID Name	K_P	K_I	K_D	Rise Time (s)	Settling Time (s)	Overshoot (%)
PID_1	-1	-1	0	0.2378	2.9133	21.7818
PID_5	-1	-0.3	-0.1	0.3476	4.0409	3.9849



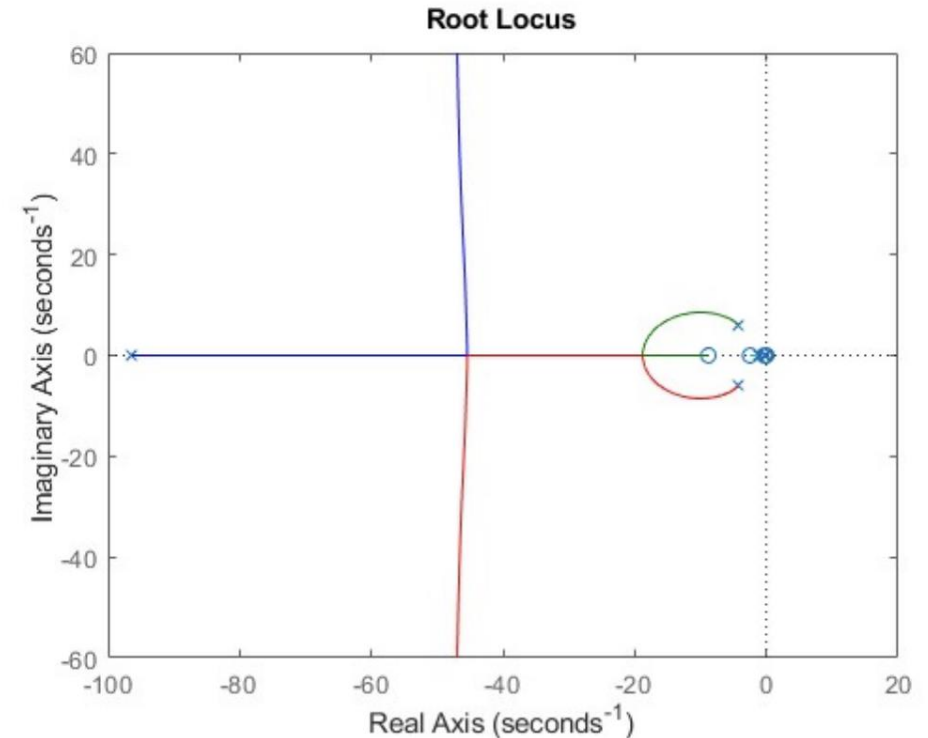
Root locus analysis

Under a 0.2-radian step input, it should have:

- The introduction of a PID controller is equivalent to placing two real poles and two real zeros
- All the poles of the closed-loop system have negative real-parts \rightarrow stable system



(a)



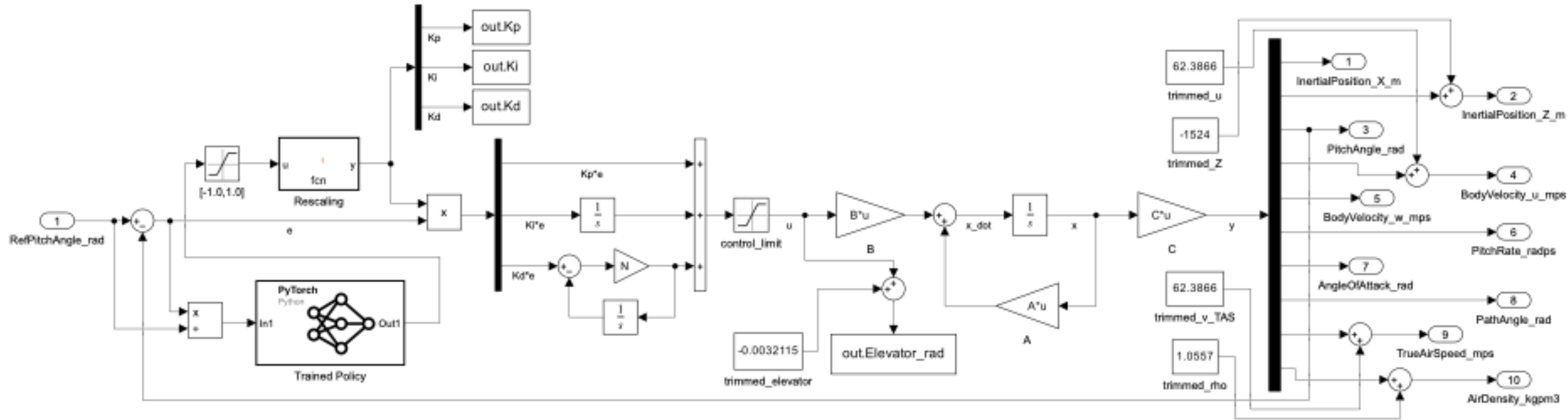
(b)

Fig. 7: Root locus plots: (a) PID controller-only. (b) Closed-loop system with PID controller.

Work in Progress

Adaptively-tuned PID via Reinforcement Learning

The university's Matlab license does not have the RL toolbox license: implementation must be done in Python, and it takes longer time than expected. After training in python, the trained policy will be utilized in Simulink as follows.



Implementing DDPG algorithm in Python

Work is still ongoing; results will be included in the project's final report.

Actor Network

```
class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, action_space_high, action_space_low, seed, fc1_units=400, fc2_units=300):
        """Initialize parameters and build model.
        Params
        =====
        state_size (int): Dimension of each state
        action_size (int): Dimension of each action
        seed (int): Random seed
        fc1_units (int): Number of nodes in first hidden layer
        fc2_units (int): Number of nodes in second hidden layer
        """
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
        self.reset_parameters()
        # action rescaling
        self.register_buffer(
            "action_scale", torch.tensor((action_space_high - action_space_low) / 2.0, dtype=torch.float32)
        )
        self.register_buffer(
            "action_bias", torch.tensor((action_space_high + action_space_low) / 2.0, dtype=torch.float32)
        )

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = torch.tanh(self.fc3(x))
        return x * self.action_scale + self.action_bias
```

Critic Network

```
class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fcs1_units=400, fc2_units=300):
        """Initialize parameters and build model.
        Params
        =====
        state_size (int): Dimension of each state
        action_size (int): Dimension of each action
        seed (int): Random seed
        fcs1_units (int): Number of nodes in the first hidden layer
        fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(state_size, fcs1_units)
        self.fc2 = nn.Linear(fcs1_units + action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        """Build a critic (value) network that maps (state, action) pairs -> Q-values."""
        xs = F.relu(self.fcs1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```


Thank You.