Control of a Nonlinear Liquid Level System

RATNESH RAVIKIRAN YADAV (962052529), PARASTOU FAHIM (920268631)

In this project, we applied two distinct reinforcement learning (RL) approaches—value-approximation-based RL and policy-approximation-based RL—to control a nonlinear two-tank liquid-level system subjected to external disturbances. An adaptive reward function was designed to enhance controller robustness by dynamically adjusting the control objectives based on real-time system performance. The reward structure aimed to balance multiple components, such as reducing steady-state error, mitigating overshoots, minimizing control effort, and avoiding abrupt fluctuations in the liquid level.
The simulation results revealed contrasting strengths and limitations of the two RL approaches. The value-approximation-based RL method significantly reduced control effort, optimizing energy consumption during transient states but showed limited improvement in steady-state accuracy. Conversely, the policy-based RL approach achieved better steady-state tracking of the desired liquid levels but required higher control effort to maintain precision. These findings highlight the need for a balanced strategy that combines the strengths of both methods.

1. INTRODUCTION

The control of nonlinear systems, such as two-tank liquid-level systems, poses significant challenges due to their inherent complexities, nonlinear dynamics, and sensitivity to external disturbances. These systems are prone to instabilities and performance degradation under fluctuating environmental conditions, making it difficult for traditional control methods, such as proportional-integral-derivative (PID) controllers or model-based control strategies, to maintain optimal performance. While classical approaches rely heavily on precise system modeling, they often fail when confronted with external disturbances and model uncertainties [1].

Reinforcement learning (RL) has emerged as a promising approach to address these challenges by leveraging data-driven control strategies that do not require explicit system models. Several studies have demonstrated the potential of RL-based controllers for nonlinear systems, including liquid-level systems, by autonomously learning control policies through iterative interaction with the environment. However, many of these studies focus primarily on achieving performance objectives under nominal conditions while neglecting robustness in the presence of external disturbances and uncertainties [1].

Building upon this foundation, our project aims to evaluate the robustness of RL controllers for nonlinear two-tank liquid-level systems under external disturbances. Specifically, we investigate whether the integration of an adaptive reward function designed to adjust the learning process can enhance the resilience and adaptability of the controller. By introducing disturbances into the system and dynamically modifying the reward structure, we aim to achieve a trade-off between steady-state accuracy, control effort optimization, and disturbance mitigation.

2. METHODOLOGY

In this project, we utilize reinforcement learning (RL) to design a controller for a nonlinear two-tank liquid level system subjected to external disturbances. The methodology is based on two primary approaches outlined in the referenced paper: value approximation for policy computation and policy approximation using neural networks. Below, we describe these approaches in detail.

A. Reinforcement Learning Framework

The reinforcement learning framework used in this project involves discretizing the state and action spaces of the two-tank system. The system dynamics are governed by:

$$\frac{dh_1}{dt} = \frac{q_1 - r_1\sqrt{h_1} - r_3\sqrt{h_1 - h_2}}{A_1}, \quad \frac{dh_2}{dt} = \frac{q_2 - r_2\sqrt{h_2} + r_3\sqrt{h_1 - h_2}}{A_2}, \tag{S1}$$

where h_1 and h_2 are the liquid levels in the two tanks, q_1 and q_2 are the inflow rates, and r_1, r_2, r_3, A_1, and A_2 are system parameters.

The RL framework defines:

- s: State vector $[h_1, h_2]^T$ representing the liquid levels.
- a: Action vector $[q_1, q_2]^T$ representing the inflow rates.
- $P(s'|s,a)$: Transition probabilities between states given an action.
- $R(s,a)$: Reward function for taking action a in state s.
- $\pi(s)$: Policy mapping states to actions.
- $V^\pi(s)$: Value function under policy π, representing the expected cumulative reward starting from state s.

B. Value approximation for Policy Computation

Value approximation is a dynamic programming technique used to compute the optimal policy π^* and value function $V^*(s)$.

B.1. Mathematical Formulation

The Bellman optimality equation for the value function is given by:

$$V^*(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \right], \tag{S2}$$

where $\gamma \in [0,1)$ is the discount factor. The optimal policy π^* is derived as:

$$\pi^*(s) = \arg\max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \right]. \tag{S3}$$

The discretization of state and action spaces allows for iterative updates of the value function:

$$V_{k+1}(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V_k(s') \right]. \tag{S4}$$

C. Policy Approximation Using Neural Networks

Given the complexity of representing the policy over continuous state spaces, the second approach uses a neural network to approximate the policy function $\pi(s)$.

C.1. Mathematical Formulation

The policy function is parameterized as:

$$\hat{\pi}(s;\theta) \approx \pi(s), \tag{S5}$$

where θ are the parameters of the neural network. The network is trained to minimize the error between the predicted actions and the optimal actions obtained from value approximation. The training objective is:

$$\min_{\theta} \sum_{s} |\hat{\pi}(s;\theta) - \pi^*(s)|^2. \tag{S6}$$

D. Reward Function Design

The reward function is a critical component of the RL framework, designed to guide the learning process by assigning a numerical value to each state-action pair. It directly influences the behavior of the RL agent and its ability to achieve the desired system performance.

In the original paper [1], the reward function $R(h)$ is defined solely based on the deviation of the current state h (liquid level) from the target state h_{desired}. It is expressed as:

$$R(h) = -C\left|h - h_{\text{desired}}\right|, \tag{S7}$$

where:

- h is the current liquid level,
- h_{desired} is the target liquid level,
- $C > 0$ is a positive scaling constant.

This reward function focuses only on minimizing the deviation of the liquid level from its target, without accounting for control actions, energy consumption, or system dynamics such as oscillations and abrupt changes.

Several studies have explored reinforcement learning (RL) strategies to improve control performance and stability. For instance, Khan et al. [2] highlight the integration of optimal adaptive control techniques to enhance system performance, while Lin and Zheng [3] emphasize constrained adaptive strategies for controlling nonlinear systems and reducing control effort. Furthermore, Wiering et al. [4] introduce a novel RL framework that effectively optimizes decision-making processes using value functions, providing valuable insights applicable to adaptive reward designs. Building on these works, our approach incorporates dynamic penalties to balance system stability, overshoot mitigation, and smoother control actions.

The adaptive reward function is mathematically defined as:

$$R(s, a, s_{\text{prev}}, a_{\text{prev}}) = -\left[\alpha\left|h_1 - h_{\text{desired}}\right| + \lambda_1 \,\text{Overshoot}(h_1) + \lambda_2\left|h_1 - h_{\text{prev}}\right| + \lambda_3\left|a\right|\right], \tag{S8}$$

where:

- h_1 is the current liquid level in Tank 1,
- h_{desired} is the target liquid level for Tank 1,
- h_{prev} is the previous state of Tank 1,
- a is the current control action (inflow rate to Tank 1),
- a_{prev} is the previous control action.

The reward function components are described as follows:

1. **Error Penalty:** The absolute deviation of h_1 from the target liquid level h_{desired}:

$$\text{Error} = \left|h_1 - h_{\text{desired}}\right|. \tag{S7}$$

2. **Overshoot Penalty:** A quadratic penalty applied when the liquid level h_1 exceeds the target level h_{desired}. It is defined as:

$$\text{Overshoot}(h_1) = \begin{cases} (h_1 - h_{\text{desired}})^2 & \text{if } h_1 > h_{\text{desired}}, \\ 0 & \text{otherwise.} \end{cases} \tag{S8}$$

3. **State Damping Penalty:** A penalty on the rate of change of the state to avoid abrupt fluctuations:

$$\text{State Damping} = |h_1 - h_{\text{prev}}| . \tag{S9}$$

4. **Control Action Penalty:** A penalty on the magnitude of the control action to encourage smoother control:

$$\text{Action Penalty} = |a| . \tag{S10}$$

The final reward function integrates these components with their respective weights $\alpha, \lambda_1, \lambda_2, \lambda_3 > 0$, which determine the relative importance of each term:

$$R(s, a, s_{\text{prev}}, a_{\text{prev}}) = -\left[\alpha\, |h_1 - h_{\text{desired}}| + \lambda_1 (h_1 - h_{\text{desired}})^2 + \lambda_2\, |h_1 - h_{\text{prev}}| + \lambda_3\, |a|\right] . \tag{S8}$$

Here:

- α controls the penalty for deviation from the target,
- λ_1 amplifies overshoot penalties,
- λ_2 reduces state fluctuations,
- λ_3 penalizes control effort for smoother actions.

Algorithm S1. Adaptive Reward Function Algorithm

```
Data: state, prev_state (optional), action, prev_action (optional)
Result: Reward value $r$
Define: global state_desired
Step 1: Compute error penalty
error $\leftarrow \alpha \times$ |state(1) − state_desired|
Step 2: Compute overshoot penalty
if state(1) > state_desired then
    overshoot_penalty $\leftarrow \lambda_1 \times$ (state(1) − state_desired)$^2$
else
    overshoot_penalty $\leftarrow 0$
end
Step 3: Compute state damping penalty
if prev_state is provided then
    state_damping_penalty $\leftarrow \lambda_2 \times$ |state(1) − prev_state(1)|
else
    state_damping_penalty $\leftarrow 0$
end
Step 4: Compute action penalty
action_penalty $\leftarrow \lambda_3 \times$ |action|
Step 5: Compute total reward
$r \leftarrow -$[error + overshoot_penalty + state_damping_penalty + action_penalty]
return $r$
```

This adaptive reward function aims to provide a more balanced control strategy by simultaneously reducing errors, mitigating overshoots, minimizing state fluctuations, and ensuring smoother control actions.

E. Disturbance Handling

To evaluate the robustness of the approaches, external disturbances are introduced in the system dynamics. The trained controller is tested under various disturbance scenarios, and its performance in maintaining the target liquid levels is assessed.

E.1. System Dynamics with Disturbances

The modified dynamics include random noise, sinusoidal disturbances, and step changes:

$$\frac{dh_1}{dt} = \frac{q_1 - r_1\sqrt{h_1} - r_3\sqrt{h_1 - h_2} + d_1}{A_1}, \quad \frac{dh_2}{dt} = \frac{q_2 - r_2\sqrt{h_2} + r_3\sqrt{h_1 - h_2} + d_2}{A_2}, \tag{S11}$$

where d_1 and d_2 represent the disturbances.

In this work, three types of disturbances are introduced to evaluate the robustness of the RL-based controllers:

- **Random Noise Disturbance**: Gaussian noise is applied to simulate stochastic variations in the system dynamics.
- **Sinusoidal Disturbance**: Periodic sinusoidal disturbances with a specific frequency and amplitude are added to represent oscillatory changes.
- **Step Disturbance**: A step change at a predefined time introduces sudden variations to the system.

These disturbances are combined to test the controller's ability to maintain target liquid levels under varying and challenging conditions.

3. EXPERIMENT

A. Evaluating RL Approaches: With and Without External Disturbances

The paper [1] introduces two reinforcement learning (RL) approaches—**Value approximation** and **Artificial Neural Network (ANN)-Based Policy Approximation**—to control a nonlinear two-tank liquid level system. Both approaches are evaluated for their ability to achieve the desired liquid level h_{desired}. However, the study does not consider the system's performance under external disturbances, which are critical in real-world scenarios.

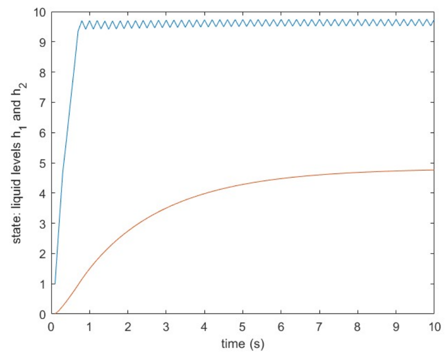
In this work, we extend the evaluation by introducing external disturbances to the system dynamics and analyzing the robustness of the RL-based controllers. The following observations are made:

- **Figure S1a** illustrates the performance of the RL-based value approximation controller without any external disturbances. The liquid levels h_1 and h_2 smoothly reach their desired values with minimal oscillations.
- **Figure S1b** shows the behavior of the same controller in the presence of external disturbances. The disturbance introduces oscillations and slower convergence, particularly for h_2, highlighting the limitations of the original approach in maintaining system stability.

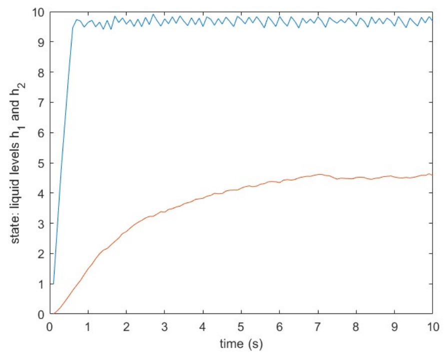
The results demonstrate that while the RL controllers perform effectively in disturbance-free conditions, their robustness is reduced when external disturbances are introduced. This highlights the need for **adaptive reward mechanisms** or **more robust RL approaches** that explicitly account for disturbances and uncertainties in the system dynamics.

Similar to the Value approximation results shown in Figures S1b and S1a, the Policy Approximation approach also experiences the impact of external disturbances, leading to oscillations. Figures S2a and S2b demonstrate the performance of the Policy Approximation (ANN-Based) approach without and with disturbances, respectively.

However, the oscillations observed in Policy Approximation are less pronounced compared to those in Value approximation. This improvement can be attributed to the neural network's ability to generalize policies across states, resulting in smoother control actions. Despite this advantage, disturbances still affect the system performance, particularly for the liquid level h_2, which stabilizes more slowly under both approaches.

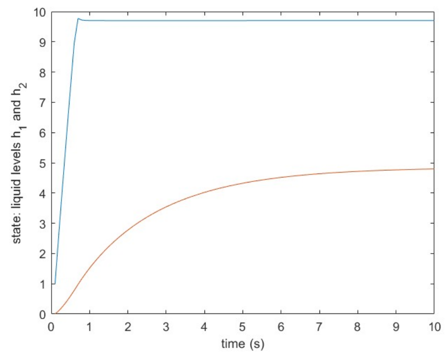


(a) Liquid Level h_1 and h_2 without an external disturbance

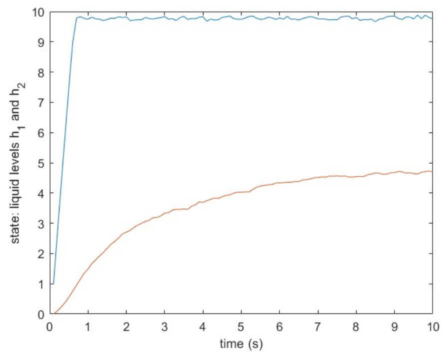


(b) Liquid Level h_1 and h_2 in the presence of disturbance

Fig. S1. Performance Comparison of RL-Based Controller: (a) Value approximation Without Disturbance, (b) Value approximation With Disturbance.



(a) Liquid Level h_1 and h_2 without an external disturbance



(b) Liquid Level h_1 and h_2 in the presence of disturbance

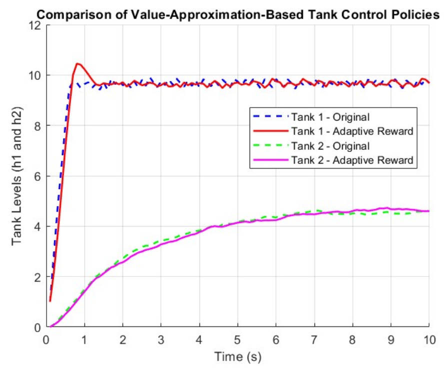
Fig. S2. Performance Comparison of RL-Based Controller: (a) Policy Approximation Without Disturbance, (b) Policy Approximation With Disturbance.

B. RL-Based Control with Adaptive Reward Function: System Behavior Under Disturbance

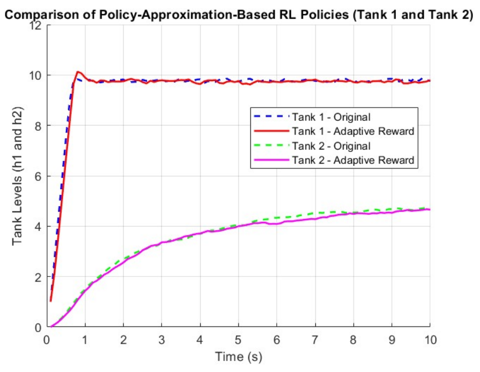
Figure S3 compares the performance of RL-based controllers using *Value approximation* (Figure S3a) and *Policy Approximation* (Figure S3b) under disturbances with the original and adaptive reward functions. As shown in Table S1 and the corresponding figures, the *Value approximation* approach with the adaptive reward function achieves a significant reduction in control effort (30.9% improvement), while the steady-state error remains unchanged at 0.33. While slight improvements in oscillations are observed, disturbances still impact system performance, particularly for h_1, indicating limited robustness.

For the *Policy Approximation* approach, the adaptive reward function achieves better results by leveraging the neural network's generalization capabilities. It reduces the steady-state error from 0.24 to 0.21 (14.78% improvement) and achieves a slight improvement in control effort (2.28%). The control response is smoother with fewer oscillations, demonstrating greater resilience to disturbances compared to Value approximation.

Overall, while disturbances affect both approaches, Policy Approximation outperforms Value approximation by achieving smoother control, reduced steady-state error, and improved robustness due to the adaptive reward function.



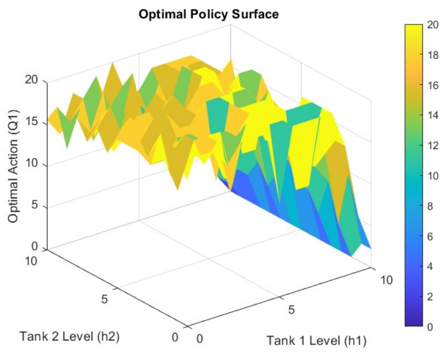
(a) Liquid Level h_1 and h_2



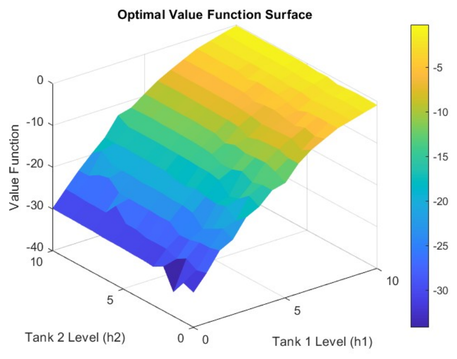
(b) Liquid Level h_1 and h_2

Fig. S3. Performance Comparison of RL-Based Controller in the presence of the Disturbance: (a) Value approximation (b) Policy Approximation

The 3D plots compare the best policy and value functions for *Value approximation* (Fig. S4) and *Policy Approximation* (Fig. S5). The Value approximation approach shows sharp variations due to discretization, while Policy Approximation smooths the functions using a neural network, resulting in fewer oscillations.

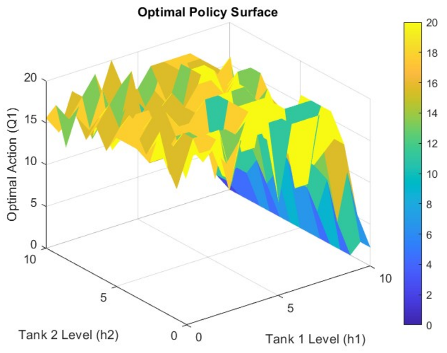


(a) Best Policy Function for Value approximation

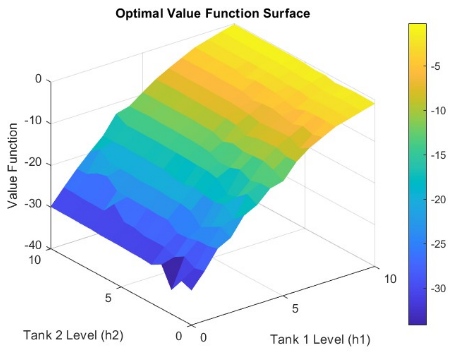


(b) Functions learned by the RL controller with discretization levels

Fig. S4. 3D Plots Value approximation



(a) Best Policy Function for Policy Approximation



(b) Functions learned by the RL controller with discretization levels

Fig. S5. 3D Plots Policy Approximation

Table S1. Comparison of RL Approaches with Original and Adaptive Reward Functions

Metric	Value approximation		ANN-Based Policy Approximation	
	Original	Adaptive	Original	Adaptive
Control Effort	27.76	19.18	12.99	12.70
Steady-State Error	0.33	0.33	0.24	0.21
Improvement After Adaptive Reward				
Steady-State Error Improvement (%)	-0.93%		14.78%	
Control Effort Improvement (%)	30.90%		2.28%	

4. CONCLUSION

In this project, we explored two reinforcement learning (RL) approaches—value-approximation-based RL and policy-approximation-based RL—to control a nonlinear two-tank liquid-level system under external disturbances. An adaptive reward function was designed to dynamically guide the learning process by balancing multiple objectives, including steady-state accuracy, control effort optimization, overshoot mitigation, and stability. Simulation results demonstrated that while the value-approximation-based RL method effectively minimized control effort, it struggled to achieve precise steady-state tracking. Conversely, the policy-based RL approach achieved better steady-state accuracy but required higher control effort to maintain precision.

These findings underscore the importance of developing a balanced RL strategy that combines the strengths of both methods to achieve robust performance. Although the adaptive reward function showed potential, its disturbance mitigation capabilities were limited. Future work will focus on integrating advanced RL frameworks that explicitly account for model uncertainties and external disturbances, as well as refining the reward structure to further enhance control stability, energy efficiency, and resilience. This study will provide a foundation for working on RL-based control systems, particularly in nonlinear and uncertain environments.

REFERENCES

1. M. M. Noel and B. J. Pandian, "Control of a nonlinear liquid level system using a new artificial neural network based reinforcement learning approach," Appl. Soft Comput. **23**, 444–451 (2014).
2. S. G. Khan, G. Herrmann, F. L. Lewis, *et al.*, "Reinforcement learning and optimal adaptive control: An overview and implementation examples," Annu. Rev. Control. **36**, 42–59 (2012).
3. W.-S. Lin and C.-H. Zheng, "Constrained adaptive optimal control using a reinforcement learning agent," Automatica **48**, 2614–2619 (2012).
4. M. A. Wiering, H. van Hasselt, A.-D. Pietersma, and L. Schomaker, "Reinforcement learning algorithms for solving classification problems," in *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL),* (2011), pp. 91–96.

APPENDIX: MATLAB SCRIPTS

A. tank.m - system dynamics with disturbances

```
function hdot = tank(t,h)

global Q2;

global action;

Q1=action;

a1=1;
a2=1;
a3=1;
A1=1;
A2=1;

% Disturbances
% 1. Random noise disturbance
d1_random = 0.5 + 1.5 * randn; % Random disturbance for Tank 1
d2_random = -0.5 -1.0 * randn; % Random disturbance for Tank 2

% 2. Sinusoidal disturbance
frequency = 0.2; % Frequency of the sinusoidal disturbance
amplitude = 0.5; % Amplitude of the sinusoidal disturbance
d1_sine = amplitude * sin(2 * pi * frequency * t); % Sinusoidal
    disturbance for Tank 1
d2_sine = amplitude * sin(2 * pi * frequency * t + pi/4); % Phase-shifted
    sine disturbance for Tank 2

% 3. Step disturbance
step_magnitude = 2.0; % Magnitude of step disturbance
step_time = 5; % Time at which the disturbance occurs
d1_step = step_magnitude * (t > step_time); % Step disturbance for Tank 1
d2_step = -step_magnitude * (t > step_time); % Step disturbance for Tank 2

% Combine disturbances
d1 = d1_random + d1_sine + d1_step;
d2 = d2_random + d2_sine + d2_step;

% % No disturbances:
% d1 = 0;
% d2 = 0;

if h(1) >= h(2)
    hdot= [(Q1  -a3*sqrt(h(1)-h(2))+d1)/A1; (Q2 - a2*sqrt(h(2)) + a3*sqrt(
        h(1)-h(2))+d2)/A2];
else
    hdot= [(Q1  +a3*sqrt(h(2)-h(1))+d1)/A1; (Q2 - a2*sqrt(h(2)) - a3*sqrt(
        h(2)-h(1))+d2)/A2];
end

end
```

B. Reward.m - reward function for RL

```
function r = Reward(state)

global state_desired;

r = -abs(state(1)-state_desired);

end
```

C. Reward_adaptive.m - adaptive reward function for RL

```
function r = Reward_adaptive(state, prev_state, action, prev_action)
% Adaptive reward function for smoother control and better stability

global state_desired;

% Error penalty
error = abs(state(1) - state_desired);

% Overshoot penalty
if state(1) > state_desired
    overshoot_penalty = 10 * (state(1) - state_desired)^2; % Stronger
        penalty for overshoot
else
    overshoot_penalty = 0;
end

% State damping penalty
if nargin > 1
    state_damping_penalty = 0.3 * abs(state(1) - prev_state(1)); %
        Moderate penalty for state rate changes
else
    state_damping_penalty = 0;
end

% Action penalty (increased for smoother control)
action_penalty = 0.02 * abs(action); % Slightly higher weight

% Compute total reward
r = -error - overshoot_penalty - state_damping_penalty - action_penalty;

end
```

D. closest.m - computes the closest discretized state

```
% % computes the closest discretized state to snext
function [r, s] = closest(snext)
% Computes the closest discretized state to snext

global h1 h2 delta;

% Handle edge cases for Tank 1
if snext(1) > h1(end) + delta
    r = length(h1); % Cap at the maximum index
elseif snext(1) < h1(1) - delta
    r = 1; % Minimum index
else
    % Find the closest index
    r = find(abs(h1 - snext(1)) <= delta, 1);
end

% Handle edge cases for Tank 2
if snext(2) > h2(end) + delta
    s = length(h2); % Cap at the maximum index
elseif snext(2) < h2(1) - delta
    s = 1; % Minimum index
else
    % Find the closest index
    s = find(abs(h2 - snext(2)) <= delta, 1);
end

% Default to the closest point if no valid match is found
if isempty(r), r = length(h1); end
if isempty(s), s = length(h2); end

end
```

E. val_approx_RL.m - value approximation RL

```
function results = val_approx_RL()
rng(50); % Set the seed once for reproducibility
% REINFORCEMENT LEARNING CONTROL OF TWO TANK LIQUID LEVEL SYSTEM

clear all;
clc;

% Define final desired goal state
global state_desired;
state_desired= 10;

% Flow to Tank 2 is not controlled and hence set to zero
global Q2;
Q2=0;

% Discretize state space
global h1;
global h2;
h1=linspace(0,10,15);
h2=h1;

global delta;
delta= (h1(2)-h1(1))/2;

% Discretize action space
global action;
Q1=linspace(0,20,10);

N1 = length(h1);
N2 = length(h2);

% Initialize policy and value.
pibest = zeros(N1,N2);

gamma =0.99;

% Set the initial guess for V(s) to be zero for each state s.
V = zeros(N1,N2);
policy = zeros(N1,N2);

% Compute the optimal value function using the Value Iteration algorithm.
for runs=1:1000

    for m=1:N1
        for n=1:N2
            for p =1:length(Q1)

                % Take all possible actions.
                action = Q1(p);

                snext = [h1(m); h2(n)]+ 0.1*tank(0,[h1(m); h2(n)]);

                % Compute the closest discretized state.
                [r,s] = closest(snext);

                nextV(p)=V(r,s);
            end

            [Vbest,bestind] = max(nextV);

            % Improve value function estimate using Bellman's equation.
            V(m,n)= Reward([h1(m); h2(n)] ) +  gamma*Vbest ;
        end
    end

end

% Compute the optimal policy from the optimal value function.
for m=1:N1
    for n=1:N2
```

```
        % Take all possible actions.
        for p =1:length(Q1)

            action = Q1(p);

            snext = [h1(m); h2(n)]+ 0.1*tank(0,[h1(m); h2(n)]);

            % Compute the closest discretized state.
            [r,s] = closest(snext);

            nextV(p)=V(r,s);
        end

        [Vbest,bestind] = max(nextV);

        pibest(m,n) = Q1(bestind);
    end
end

N = 100;
state=[1 0]; %Initial state
states = zeros(N,2);
states(1,:)= state ;
Ts = 0.1; % Define time between control actions.

% Simulate the system with the optimal control policy.
for n=2:N

    [r,s] = closest(state);

    % Use linear regression to interpolate between control actions for
    % discretized states
    if r > 1 && s > 1 && r < N1 && s < N2
        X = [h1(r) h2(s);h1(r-1) h2(s);h1(r+1) h2(s);h1(r) h2(s-1);h1(r)
            h2(s+1)];
        Y = [pibest(r,s) pibest(r-1,s) pibest(r+1,s) pibest(r,s-1) pibest(
            r,s+1)]';
        lin_model = fitlm(X,Y);
        action = predict(lin_model,state);
    else
        action = pibest(r,s);
    end

    %Simulate the system for one time step.
    [t,y]=ode45(@tank,[0 Ts],state);
    state = real(y(end,:));
    states(n,:) = state;

end

% 3D Plot of Optimal Policy
[H1, H2] = meshgrid(h1, h2);
figure;
surf(H1, H2, pibest', 'EdgeColor', 'none'); % Transpose pibest for correct
     orientation
colorbar;
xlabel('Tank 1 Level (h1)');
ylabel('Tank 2 Level (h2)');
zlabel('Optimal Action (Q1)');
title('Optimal Policy Surface');

% 3D Plot of Value Function
V_transposed = V'; % Transpose V for correct orientation
figure;
surf(H1, H2, V_transposed, 'EdgeColor', 'none');
colorbar;
xlabel('Tank 1 Level (h1)');
ylabel('Tank 2 Level (h2)');
zlabel('Value Function');
title('Optimal Value Function Surface');
```

```
% Time vector and results
time = (1:length(states)) * Ts;
results.states = states;
results.time = time;

end
```

F. val_approx_RL_adaptive.m - value approximation RL with adaptive rewards

```
function results = val_approx_RL_adaptive()
rng(50); % Set the seed once for reproducibility
% REINFORCEMENT LEARNING CONTROL OF TWO TANK LIQUID LEVEL SYSTEM

clear all;
clc;

% Define final desired goal state
global state_desired;
state_desired= 10;

% Flow to Tank 2 is not controlled and hence set to zero
global Q2;
Q2=0;

% Discretize state space
global h1;
global h2;
h1=linspace(0,10,15);
h2=h1;

global delta;
delta= (h1(2)-h1(1))/2;

% Discretize action space
global action;
Q1=linspace(0,20,10);

N1 = length(h1);
N2 = length(h2);

% Initialize policy and value.
pibest = zeros(N1,N2);
gamma =0.99;

% Set the initial guess for V(s) to be zero for each state s.
V = zeros(N1,N2);

% Value Iteration with Enhanced Adaptive Reward
for runs = 1:1000
    for m = 1:N1
        for n = 1:N2
            for p = 1:length(Q1)
                % Take action
                action = Q1(p);

                % Simulate next state
                snext = [h1(m); h2(n)] + 0.1 * tank(0, [h1(m); h2(n)]);

                % Compute the closest discretized state
                [r, s] = closest(snext);

                % Adaptive reward calculation
                prev_state = [h1(m); h2(n)];
                prev_action = pibest(m, n); % Use previous best action for
                     rate penalty
                reward = Reward_adaptive([h1(m); h2(n)], prev_state,
                    action, prev_action);

                % Store value of next state
                nextV(p) = reward + gamma * V(r, s);
            end

            % Update value function and policy
            [Vbest, bestind] = max(nextV);
            V(m, n) = Vbest;
            pibest(m, n) = Q1(bestind);
        end
    end
end

% Simulation with Optimal Policy
N = 100;
state = [1 0];              % Initial state
states = zeros(N, 2);       % State storage
states(1, :) = state;
prev_action = 0;            % Initial action
momentum = 0.3;             % Momentum factor for action smoothing
Ts = 0.1;                   % Time step
hysteresis_band = 0.05;     % Threshold to prevent jitter

for n = 2:N
    [r, s] = closest(state);

    % Use linear regression to interpolate between control actions for
        discretized states
    if r > 1 && s > 1 && r < N1 && s < N2
        X = [h1(r) h2(s); h1(r-1) h2(s); h1(r+1) h2(s); h1(r) h2(s-1); h1(
            r) h2(s+1)];
        Y = [pibest(r, s) pibest(r-1, s) pibest(r+1, s) pibest(r, s-1)
            pibest(r, s+1)]';
        lin_model = fitlm(X, Y);
        raw_action = predict(lin_model, state);
    else
        raw_action = pibest(r, s);
    end

    % Apply momentum to smooth control actions
    action = (1 - momentum) * raw_action + momentum * prev_action;

    % Low-pass filter for action (smoother filtering)
    alpha = 0.9; % Stronger filtering
    action = alpha * action + (1 - alpha) * prev_action;

    % Gradual action reduction near desired state
    if state(1) >= state_desired
        action = action * 0.9; % Reduce inflow incrementally
    end

    % Implement hysteresis to prevent excessive control jitter
    if state(1) >= state_desired + hysteresis_band
        action = action * 0.5; % Further reduce inflow
    end

    % Clamp action to valid range
    action = max(0, min(20, action));

    % Simulate the system for one time step
    [t, y] = ode45(@tank, [0 Ts], state);
    prev_state = state;       % Store previous state
    state = real(y(end, :));  % Update state
    states(n, :) = state;

    % Reward evaluation (optional for debugging)
    Reward_adaptive(state, prev_state, action, prev_action);

    % Update previous action
    prev_action = action;
end



% 3D Plot of Optimal Policy
```

```
[H1, H2] = meshgrid(h1, h2);
figure;
surf(H1, H2, pibest', 'EdgeColor', 'none'); % Transpose pibest for correct
     orientation
colorbar;
xlabel('Tank 1 Level (h1)');
ylabel('Tank 2 Level (h2)');
zlabel('Optimal Action (Q1)');
title('Optimal Policy Surface');

% 3D Plot of Value Function
V_transposed = V'; % Transpose V for correct orientation
figure;
surf(H1, H2, V_transposed, 'EdgeColor', 'none');
colorbar;
xlabel('Tank 1 Level (h1)');
ylabel('Tank 2 Level (h2)');
zlabel('Value Function');
title('Optimal Value Function Surface');

% Time vector and results
time = (1:length(states)) * Ts;
results.states = states;
results.time = time;

end
```

G. compare_val_approx_RL.m - comparison of value-approximation RL

```
% Comparison of RL-Based Tank Control Policies
clear all; clc; close all;

% Run the first script (Original RL)
results_original = val_approx_RL();

% Run the second script (Adaptive RL)
results_adaptive = val_approx_RL_adaptive();

% Extract time and states
time = results_original.time; % Both scripts have the same time vector
states_original = results_original.states;
states_adaptive = results_adaptive.states;

% Plot Comparison of Tank 1 and Tank 2 Levels
figure;
hold on;
plot(time, states_original(:, 1), 'b--', 'LineWidth', 1.5); % Tank 1 -
    Original
plot(time, states_adaptive(:, 1), 'r-', 'LineWidth', 1.5);  % Tank 1 -
    Adaptive
plot(time, states_original(:, 2), 'g--', 'LineWidth', 1.5); % Tank 2 -
    Original
plot(time, states_adaptive(:, 2), 'm-', 'LineWidth', 1.5);  % Tank 2 -
    Adaptive
xlabel('Time (s)');
ylabel('Tank Levels (h1 and h2)');
title('Comparison of Value-Approximation-Based Tank Control Policies');
legend({'Tank 1 - Original', 'Tank 1 - Adaptive Reward', 'Tank 2 -
    Original', 'Tank 2 - Adaptive Reward'}, 'Location', 'Best');
grid on;
hold off;

% Performance Metrics
% Settling Time for Tank 1
settling_threshold = 0.05; % 5% threshold for settling
settled_original = find(abs(states_original(:, 1) - 10) <
    settling_threshold, 1);
settled_adaptive = find(abs(states_adaptive(:, 1) - 10) <
    settling_threshold, 1);
```

```
if isempty(settled_original)
    settling_time_original = NaN; % Did not settle
else
    settling_time_original = time(settled_original);
end

if isempty(settled_adaptive)
    settling_time_adaptive = NaN; % Did not settle
else
    settling_time_adaptive = time(settled_adaptive);
end

% Control Effort
control_effort_original = sum(abs(diff(states_original(:, 1))));
control_effort_adaptive = sum(abs(diff(states_adaptive(:, 1))));

% Steady-State Error
final_h1_original = states_original(end, 1);
final_h1_adaptive = states_adaptive(end, 1);

error_original = abs(final_h1_original - 10); % Steady-state error for
    Original
error_adaptive = abs(final_h1_adaptive - 10); % Steady-state error for
    Adaptive

% Improvement Metrics
% Steady-state error improvement
if error_original ~= 0
    steady_state_error_improvement = ((error_original - error_adaptive) /
        error_original) * 100;
else
    steady_state_error_improvement = NaN;
end

% Settling time improvement
if ~isnan(settling_time_original) && ~isnan(settling_time_adaptive)
    settling_time_improvement = ((settling_time_original -
        settling_time_adaptive) / settling_time_original) * 100;
else
    settling_time_improvement = NaN;
end

% Control effort improvement
if control_effort_original ~= 0
    control_effort_improvement = ((control_effort_original -
        control_effort_adaptive) / control_effort_original) * 100;
else
    control_effort_improvement = NaN;
end

% Display Results
fprintf('Performance Metrics:\n');
fprintf('Control Effort (Original): %.2f\n', control_effort_original);
fprintf('Control Effort (Adaptive): %.2f\n', control_effort_adaptive);
fprintf('Steady-State Error (Original): %.2f\n', error_original);
fprintf('Steady-State Error (Adaptive): %.2f\n', error_adaptive);

fprintf('\nImprovement Metrics:\n');
fprintf('Steady-State Error Improvement: %.2f%%\n',
    steady_state_error_improvement);
fprintf('Control Effort Improvement: %.2f%%\n', control_effort_improvement
    );
```

H. policy_approx_RL.m - policy approximation RL

```
function results = policy_approx_RL()
rng(50); % Set the seed once for reproducibility

% REINFORCEMENT LEARNING CONTROL OF TWO TANK LIQUID LEVEL SYSTEM
% Control of a nonlinear liquid level system using a new artificial neural
     network based reinforcement learning approach,

clear all;
clc;

% Define final desired goal state
global state_desired;
state_desired= 10;

% Flow to Tank 2 is not controlled and hence set to zero
global Q2;
Q2=0;

% Discretize state space
global h1;
global h2;
h1=linspace(0,10,15);
h2=h1;

global delta;
delta= (h1(2)-h1(1))/2;

% Discretize action space
global action;
Q1=linspace(0,20,10);

N1 = length(h1);
N2 = length(h2);

% Initialize policy and value.
pibest = zeros(N1,N2);
gamma =0.99;

% Set the initial guess for V(s) to be zero for each state s.
V = zeros(N1,N2);
policy = zeros(N1,N2);

% Compute the optimal value function using the Value Iteration algorithm.
for runs=1:1000

    for m=1:N1
        for n=1:N2
            for p =1:length(Q1)

                % Take all possible actions.
                action = Q1(p);

                snext = [h1(m); h2(n)]+ 0.1*tank(0,[h1(m); h2(n)]);

                % Compute the closest discretized state.
                [r,s] = closest(snext);

                nextV(p)=V(r,s);
            end

            [Vbest,bestind] = max(nextV);

            % Improve value function estimate using Bellman's equation.
            V(m,n)= Reward([h1(m); h2(n)] ) +  gamma*Vbest ;
        end
    end

end

% Compute the optimal policy from the optimal value function.
```

```
for m=1:N1
    for n=1:N2

        % Take all possible actions.
        for p =1:length(Q1)

            action = Q1(p);

            snext = [h1(m); h2(n)]+ 0.1*tank(0,[h1(m); h2(n)]);

            % Compute the closest discretized state.
            [r,s] = closest(snext);

            nextV(p)=V(r,s);
        end

        [Vbest,bestind] = max(nextV);

        pibest(m,n) = Q1(bestind);
    end
end

%train a feedforward neural net to approximate pbest
p=1;
targetQ = zeros(1,length(h1)*length(h2));
input_states = zeros(2,length(h1)*length(h2));

for m=1:length(h1)

    for n=1:length(h2)

        input_states(:,p)= [h1(m) ; h2(n)];
        targetQ(p) = pibest(m,n);
        p=p+1;
    end
end

net = feedforwardnet(1);
net=init(net);
[net,tr] = train(net,input_states,targetQ);

N = 100;
state=[1 0]; %Initial state
states = zeros(N,2);
states(1,:)= state ;
Ts = 0.1; % Define time between control actions.

% Simulate the system with the optimal control policy.
for n=2:N

    % Use the optimal action learnt by the ANN
    action = net(state');

    %Simulate the system for one time step.
    [t,y]=ode45(@tank,[0 Ts],state);
    state = real(y(end,:));
    states(n,:) = state;

end

% Plot 3D surfaces for Optimal Policy and Value Function
[H1, H2] = meshgrid(h1, h2);

% Plot Optimal Policy
figure;
surf(H1, H2, pibest', 'EdgeColor', 'none');
colorbar;
xlabel('Tank 1 Level (h1)');
ylabel('Tank 2 Level (h2)');
zlabel('Optimal Action (Q1)');
title('Optimal Policy Surface');
```

```
% Plot Value Function
V_transposed = V'; % Transpose V to match dimensions
figure;
surf(H1, H2, V_transposed, 'EdgeColor', 'none');
colorbar;
xlabel('Tank 1 Level (h1)');
ylabel('Tank 2 Level (h2)');
zlabel('Value Function');
title('Optimal Value Function Surface');

% Plot time history of states with optimal policy.
% Time vector and results
time = (1:length(states)) * Ts;
results.states = states;
results.time = time;

end
```

I. policy_approx_RL_adaptive.m - policy approximation RL with adaptive rewards

```
function results = policy_approx_RL_adaptive()
rng(50); % Set the seed once for reproducibility

% REINFORCEMENT LEARNING CONTROL OF TWO TANK LIQUID LEVEL SYSTEM
% Control of a nonlinear liquid level system using a new artificial neural
     network based reinforcement learning approach,

clear all;
clc;

% Define final desired goal state
global state_desired;
state_desired= 10;

% Flow to Tank 2 is not controlled and hence set to zero
global Q2;
Q2=0;

% Discretize state space
global h1;
global h2;
h1=linspace(0,10,15);
h2=h1;

global delta;
delta= (h1(2)-h1(1))/2;

% Discretize action space
global action;
Q1=linspace(0,20,10);

N1 = length(h1);
N2 = length(h2);

% Initialize policy and value.
pibest = zeros(N1,N2);
gamma =0.99;

% Set the initial guess for V(s) to be zero for each state s.
V = zeros(N1,N2);
policy = zeros(N1,N2);

% Value Iteration with Enhanced Adaptive Reward
for runs = 1:1000
    for m = 1:N1
        for n = 1:N2
            for p = 1:length(Q1)
                % Take action
                action = Q1(p);
                % Simulate next state
                snext = [h1(m); h2(n)] + 0.1 * tank(0, [h1(m); h2(n)]);

                % Compute closest discretized state
                [r, s] = closest(snext);

                % Reward calculation
                prev_state = [h1(m); h2(n)];
                prev_action = pibest(m, n); % Use previous best action for
                     penalty
                reward = Reward_adaptive([h1(m); h2(n)], prev_state,
                    action, prev_action);

                % Store value of next state
                nextV(p) = reward + gamma * V(r, s);
            end

            % Update value function and policy
            [Vbest, bestind] = max(nextV);
            V(m, n) = Vbest;
            pibest(m, n) = Q1(bestind);
        end
    end
end

%train a feedforward neural net to approximate pbest

p=1;
targetQ = zeros(1,length(h1)*length(h2));
input_states = zeros(2,length(h1)*length(h2));

for m=1:length(h1)

    for n=1:length(h2)

        input_states(:,p)= [h1(m) ; h2(n)];
        targetQ(p) = pibest(m,n);
        p=p+1;
    end
end

% Train Feedforward Neural Network with Smoother Policy Approximation
net = feedforwardnet(1);
net.trainParam.lr = 0.005; % Smaller learning rate for smoother learning
net.trainParam.epochs = 200; % Increase epochs for more training
    iterations
net = init(net);
[net, tr] = train(net, input_states, targetQ);

N = 100;
state=[1 0]; %Initial state
states = zeros(N,2);
states(1,:)= state ;
prev_action = 0;          % Initial action
momentum = 0.3;           % Momentum factor for action smoothing
Ts = 0.1; % Define time between control actions.

% Simulate the system with the optimal control policy.
for n = 2:N
    % Use ANN to determine current action
    raw_action = net(state');

    % Apply momentum to smooth control actions
    action = (1 - momentum) * raw_action + momentum * prev_action;

    % Apply stronger low-pass filter
    alpha = 0.98; % Further increase smoothing
    action = alpha * action + (1 - alpha) * prev_action;

    % Gradual action reduction near desired state
    if state(1) >= state_desired
        action = action * 0.97; % Even more gradual reduction
    end

    % Implement hysteresis to avoid excessive adjustments
    hysteresis_band = 0.1; % Larger hysteresis band to prevent jitter
    if state(1) >= state_desired + hysteresis_band
        action = action * 0.8; % Further reduce inflow for significant
            overshoot
    end

    % Clamp action to valid range
    action = max(0, min(20, action));

    % Simulate system for one step
    [t, y] = ode45(@tank, [0 Ts], state);
    prev_state = state;       % Store previous state
    state = real(y(end, :));  % Update state
    states(n, :) = state;

    % Reward evaluation (optional for debugging)
    Reward_adaptive(state, prev_state, action, prev_action);

    % Update previous action
    prev_action = action;
end

% Plot 3D surfaces for Optimal Policy and Value Function
[H1, H2] = meshgrid(h1, h2);

% Plot Optimal Policy
figure;
surf(H1, H2, pibest', 'EdgeColor', 'none');
colorbar;
xlabel('Tank 1 Level (h1)');
ylabel('Tank 2 Level (h2)');
zlabel('Optimal Action (Q1)');
title('Optimal Policy Surface');

% Plot Value Function
V_transposed = V'; % Transpose V to match dimensions
figure;
surf(H1, H2, V_transposed, 'EdgeColor', 'none');
colorbar;
xlabel('Tank 1 Level (h1)');
ylabel('Tank 2 Level (h2)');
zlabel('Value Function');
title('Optimal Value Function Surface');

% Time vector and results
time = (1:length(states)) * Ts;
results.states = states;
results.time = time;

end
```

J. compare_policy_approx_RL.m - comparison of policy-approximation RL

```
% Comparison of ANN-Based RL Policies
clear all; clc; close all;

% Run the first script (Original ANN)
results_original = policy_approx_RL();

% Run the second script (Adaptive ANN)
results_adaptive = policy_approx_RL_adaptive();

% Extract time and states
time = results_original.time; % Both scripts have the same time vector
states_original = results_original.states;
states_adaptive = results_adaptive.states;

% Single Plot for Comparison of Tank 1 and Tank 2 Levels
figure;
hold on;

% Plot Tank 1 levels
plot(time, states_original(:, 1), 'b--', 'LineWidth', 1.5); % Tank 1 -
    Original
plot(time, states_adaptive(:, 1), 'r-', 'LineWidth', 1.5);  % Tank 1 -
    Adaptive

% Plot Tank 2 levels
plot(time, states_original(:, 2), 'g--', 'LineWidth', 1.5); % Tank 2 -
    Original
plot(time, states_adaptive(:, 2), 'm-', 'LineWidth', 1.5);  % Tank 2 -
    Adaptive

% Add labels, legend, and title
xlabel('Time (s)');
ylabel('Tank Levels (h1 and h2)');
title('Comparison of Policy-Approximation-Based RL Policies (Tank 1 and
    Tank 2)');
legend({'Tank 1 - Original', 'Tank 1 - Adaptive Reward', 'Tank 2 -
    Original', 'Tank 2 - Adaptive Reward'}, 'Location', 'Best');
grid on;
hold off;


% Calculate Settling Time (Tank 1)
settling_threshold = 0.05; % 5% threshold for settling around target (10)
settled_original = find(abs(states_original(:, 1) - 10) <
    settling_threshold, 1);
settled_adaptive = find(abs(states_adaptive(:, 1) - 10) <
    settling_threshold, 1);

if isempty(settled_original)
    settling_time_original = NaN; % Did not settle
else
    settling_time_original = time(settled_original);
end

if isempty(settled_adaptive)
    settling_time_adaptive = NaN; % Did not settle
else
    settling_time_adaptive = time(settled_adaptive);
end

% Calculate Control Effort
control_effort_original = sum(abs(diff(states_original(:, 1))));
control_effort_adaptive = sum(abs(diff(states_adaptive(:, 1))));

% Calculate Steady-State Error
final_h1_original = states_original(end, 1);
final_h1_adaptive = states_adaptive(end, 1);

error_original = abs(final_h1_original - 10); % Steady-state error for
    Original
```

```
error_adaptive = abs(final_h1_adaptive - 10); % Steady-state error for
    Adaptive

% Performance Improvements
% Steady-state error improvement
if error_original ~= 0 % Avoid division by zero
    steady_state_error_improvement = ((error_original - error_adaptive) /
        error_original) * 100;
else
    steady_state_error_improvement = NaN; % Undefined improvement
end

% Settling time improvement
if ~isnan(settling_time_original) && ~isnan(settling_time_adaptive)
    settling_time_improvement = ((settling_time_original -
        settling_time_adaptive) / settling_time_original) * 100;
else
    settling_time_improvement = NaN; % Undefined improvement
end

% Control effort improvement
if control_effort_original ~= 0 % Avoid division by zero
    control_effort_improvement = ((control_effort_original -
        control_effort_adaptive) / control_effort_original) * 100;
else
    control_effort_improvement = NaN; % Undefined improvement
end

% Display Results
fprintf('Performance Metrics:\n');
% fprintf('Settling Time (Original): %.2f s\n', settling_time_original);
% fprintf('Settling Time (Adaptive): %.2f s\n', settling_time_adaptive);
fprintf('Control Effort (Original): %.2f\n', control_effort_original);
fprintf('Control Effort (Adaptive): %.2f\n', control_effort_adaptive);
fprintf('Steady-State Error (Original): %.2f\n', error_original);
fprintf('Steady-State Error (Adaptive): %.2f\n', error_adaptive);

fprintf('\nImprovement Metrics:\n');
fprintf('Steady-State Error Improvement: %.2f%%\n',
    steady_state_error_improvement);
% fprintf('Settling Time Improvement: %.2f%%\n', settling_time_improvement
    );
fprintf('Control Effort Improvement: %.2f%%\n', control_effort_improvement
    );
```