Control for Robotics: From Optimal Control to Reinforcement Learning

Lingjie Zhang

2024 SS

Assignment 1: Optimal Control and Dynamic Programming

Contents

Problem 1.1 Finite Horizon Dynamic Programming

(a)

The policy found using the dynamic programming algorithm is expected to behave in a way that minimizes the total cost over the given time horizon.

Different values of q and r will change the robot's behavior as follows:

- Increasing q will increase penalty of position errors in the cost function, which leading to a policy that prioritizes minimizing position errors.
- Increasion r will increase penalty of control inputs in the cost function, which leading to a policy that prioritizes smoother control inputs to save energy.

(b)

· **Dynamics**

$$x_{k+1} = x_k + u_k + w_k \tag{1.1}$$

$$\tag{1.2}$$

· **Cost**

$$x_2^2 + \sum_{k=0}^{1}(qx_k^2 + ru_k^2) \tag{1.3}$$

Let $q = 5/2$ and $r = 1$ and assume that $w_k = 0$ for all k:

· **Initialization**

$$J_2(x_2) = x_2^2 \tag{1.4}$$

· **Recursion**

$\triangleright$ *Step* $k = 1$

$$J_1(x_1) = \min_{u_1} \mathbb{E}_{w_1}[\frac{5}{2}x_1^2 + u_1^2 + J_2(x_2)] \tag{1.5}$$

$$= \min_{u_1} \mathbb{E}_{w_1}[\frac{5}{2}x_1^2 + u_1^2 + (x_1 + u_1)^2] \text{ (sub dynamics)} \tag{1.6}$$

$$= \min_{u_1} \mathbb{E}_{w_1}[\frac{7}{2}x_1^2 + 2u_1x_1 + 2u_1^2] \tag{1.7}$$

Solve for optimal u_1 by differentiating the cost and setting it to zero:

$$u_1 = -\frac{x_1}{2} \tag{1.8}$$

Substitute u_1 back to $J_1(x_1)$:

$$J_1(x_1) = 3x_1^2 \tag{1.9}$$

▷ *Step* $k = 0$

$$J_0(x_0) = \min_{u_0} \mathbb{E}_{w_0}[\frac{5}{2}x_0^2 + u_0^2 + J_1(x_1)] \tag{1.10}$$

$$= \min_{u_0}[\frac{11}{2}x_0^2 + 4u_0^2 + 6x_0u_0] \tag{1.11}$$

Solve for optimal u_0 by differentiating the cost and setting it to zero:

$$u_0 = -\frac{3x_0}{4} = -\frac{3 \cdot (-1)}{4} = \frac{3}{4} \tag{1.12}$$

Substitute u_0 back to $J_0(x_0)$:

$$J_0(x_0) = \frac{13}{4}x_0^2 = \frac{13}{4} \tag{1.13}$$

For $x_0 = -1$:

$$x_1 = x_0 + u_0 = -1 + \frac{3}{4} = -\frac{1}{4} \tag{1.14}$$

The answer:

- Optimal policy: $\pi^* = \{-\frac{3x_0}{4}, -\frac{x_1}{2}\} = \{\frac{3}{4}, \frac{1}{8}\}$
- $J(x_0) := \frac{13}{4}x_0^2 = \frac{13}{4}$

(c)

$$\text{Var}[w_k] \;= \mathbb{E}[w_k^2] - \mathbb{E}[w_k]^2 \tag{1.15}$$

$$\Rightarrow \quad \mathbb{E}[w_k^2] \quad = 1 + 1 = 2 \tag{1.16}$$

We keep $q = 5/2$ and $r = 1$,but now assume there is some variation:

· Initialization

$$J_2(x_2) = x_2^2 \tag{1.17}$$

· **Recursion**

▷ *Step* $k = 1$

$$J_1(x_1) = \min_{u_1} \mathbb{E}_{w_1}[\frac{5}{2}x_1^2 + u_1^2 + J_2(x_2)] \tag{1.18}$$

$$= \min_{u_1}(\frac{5}{2}x_1^2 + u_1^2 + (x_1 + u_1 + w_1)^2) \text{ (sub dynamics)} \tag{1.19}$$

$$= \min_{u_1}(\frac{7}{2}x_1^2 + 2u_1^2 + w_1^2 + 2x_1u_1 + 2x_1w_1 + 2u_1w_1) \tag{1.20}$$

$$= \min_{u_1}(\frac{7}{2}x_1^2 + 2u_1^2 + 2 + 2x_1u_1 + 2x_1 + 2u_1) \tag{1.21}$$

Solve for optimal u_1 by differentiating the cost and setting it to zero:

$$u_1 = -\frac{x_1 + 1}{2} \tag{1.22}$$

Substitute u_1 back to $J_1(x_1)$:

$$J_1(x_1) = 3x_1^2 + x_1 + \frac{3}{2} \tag{1.23}$$

▷ *Step* $k = 0$

$$J_0(x_0) = \min_{u_0} \mathbb{E}_{w_0}[\frac{5}{2}x_0^2 + u_0^2 + J_1(x_1)] \tag{1.24}$$

$$= \min_{u_0} \mathbb{E}_{w_0}[\frac{5}{2}x_0^2 + u_0^2 + (3(x_0 + u_0 + w_0)^2 + x_0 + u_0 + w_0 + \frac{3}{2})] \tag{1.25}$$

$$= \min_{u_0}(\frac{11}{2}x_0^2 + 4u_0^2 + 6x_0u_0 + 7x_0 + 7u_0 + \frac{17}{2}) \tag{1.26}$$

Solve for optimal u_0 by differentiating the cost and setting it to zero:

$$u_0 = -\frac{6x_0 + 7}{8} = -\frac{1}{8} \tag{1.27}$$

Substitute u_0 back to $J_0(x_0)$:

$$J_0(x_0) = \frac{13}{4}x_0^2 + \frac{7}{4}x_0 + \frac{87}{16} \tag{1.28}$$

For $x_0 = -1$:

$$x_1 = x_0 + u_0 + w_0 = -1 - \frac{1}{8} + 1 = -\frac{1}{8} \tag{1.29}$$

The answer:

- Optimal policy: $\pi^* = \{-\frac{6x_0+7}{8}, -\frac{x_1+1}{2}\} = \{-\frac{1}{8}, -\frac{7}{16}\}$
- $J(x_0) := \frac{13}{4}x_0^2 + \frac{7}{4}x_0 + \frac{87}{16} = \frac{111}{16}$

(d)

Disturbance is not considered in (b), there are no constant terms in the optimal policy, and no linear terms and constant terms in $J(x_0)$; Disturbance is considered in (c), there are constant terms in the optimal policy, and there are linear terms and constant terms in $J(x_0)$;

Problem 1.2 Dynamic Programming for a Robot Vacuum Cleaner

```
% cfr_a1_2: Main script for Problem 1.2 Dynamic Programming for a
%               Robot Vacuum Cleaner.
%
% --
% Control for Robotics
% Summer 2023
% Assignment 1
%
% --
% Technical University of Munich
% Learning Systems and Robotics Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistants:
% SiQi Zhou: siqi.zhou@tum.de
% Lukas Brunke: lukas.brunke@tum.de
%
% --
% Revision history
% [22.01.17, LB]    first version
% [22.01.23, LB]    added 2 (c) to the code, removed N

clear all
close all
clc

%% calculate optimal control using dynamic programming

% initialize the grid world
grid = GridWorld();

% allocate arrays for optimal control inputs and cost-to-go
U = zeros(grid.num_rows, grid.num_columns);
J = zeros(grid.num_rows, grid.num_columns);

% set the cost for the obstacle
J(grid.obstacle_pos(1), grid.obstacle_pos(2)) = inf;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TODO: YOUR CODE HERE - Exercise 2 (a)
epsilon = 1e-6; % Convergence basis
max_iter = 1000; % Maximum number of iterations
[J, U] = compute_optimal_policy(grid, epsilon, max_iter, J, U);
% Function at the end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```
%% Simulate robot vacuum cleaner
x_0 = [4; 3];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TODO: YOUR CODE HERE - Exercise 2 (b)

optimal_actions = plot_optimal_trajectory(grid, x_0, U);
% Function at the end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

grid.plot_moves(x_0, optimal_actions)

% %% Simulate robot vacuum cleaner
x_0 = [4; 3];
%
% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TODO: YOUR CODE HERE - Exercise 2 (c)
grid = GridWorld();
grid.cost_dirt = 5;
grid.stage_cost(1, 1) = grid.cost_dirt;
grid.stage_cost(1, 2) = grid.cost_dirt;
grid.stage_cost(2, 1) = grid.cost_dirt;
grid.stage_cost(3, 1) = grid.cost_dirt;
grid.stage_cost(3, 2) = grid.cost_dirt;
% allocate arrays for optimal control inputs and cost-to-go
U1 = zeros(grid.num_rows, grid.num_columns);
J1 = zeros(grid.num_rows, grid.num_columns);

% set the cost for the obstacle
J1(grid.obstacle_pos(1), grid.obstacle_pos(2)) = inf;
epsilon = 1e-6; % Convergence basis
max_iter = 1000; % Maximum number of iterations
[J1, U1] = compute_optimal_policy(grid, epsilon, max_iter, J1, U1);
optimal_actions = plot_optimal_trajectory(grid, x_0, U1);

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
grid.plot_moves(x_0, optimal_actions)

function [J, U] = compute_optimal_policy(grid, epsilon, max_iter, J, U)
    iter = 0;
    while true
        iter = iter + 1;
        old_J = J;
        for i = 1:grid.num_rows
            for j = 1:grid.num_columns
                state = [i; j];
                actions = grid.available_actions(state);
                min_cost = inf;
                best_action = -1;
                for a = actions
                    next_state = grid.next_state(state, a);
                    cost = grid.stage_cost(next_state(1), next_state(2));
                    if cost < inf
                        new_cost = cost + old_J(next_state(1), next_state(2));
```

```
                        if new_cost < min_cost
                            min_cost = new_cost;
                            best_action = a;
                        end
                    end
                end
                J(i, j) = min_cost;
                U(i, j) = best_action;
            end
        end
        if max(max(abs(J - old_J))) < epsilon || iter >= max_iter
            break;
        end
    end
end

function optimal_actions = plot_optimal_trajectory(grid, x_0, U)
    optimal_actions = [];
    x = x_0;
    trajectory = [x'];

    while true
        action = U(x(1), x(2));
        optimal_actions = [optimal_actions, action];
        x_next = grid.next_state(x, action);
        trajectory = [trajectory; x_next'];
        x = x_next;
        if all(x == grid.charger_pos)
            break;
        end
    end

    figure;
    hold on;

    for i = 1:grid.num_rows
        for j = 1:grid.num_columns
            if grid.stage_cost(i, j) == grid.cost_obstacle
                fill([j-1 j j j-1], [i-1 i-1 i i], 'k'); % obstacle
            elseif grid.stage_cost(i, j) == grid.cost_dirt
                fill([j-1 j j j-1], [i-1 i-1 i i], 'y'); % dirt
            elseif grid.stage_cost(i, j) == grid.cost_charger
                fill([j-1 j j j-1], [i-1 i-1 i i], 'g'); % charger
            elseif grid.stage_cost(i, j) == grid.cost_carpet
                fill([j-1 j j j-1], [i-1 i-1 i i], 'm'); % carpet
            else
                fill([j-1 j j j-1], [i-1 i-1 i i], 'w'); % blank
            end
            plot([j-1 j-1], [i-1 i], 'k');
            plot([j-1 j], [i i], 'k');
        end
    end

    plot(trajectory(:, 2)-0.5, trajectory(:, 1)-0.5, 'r', 'LineWidth', 2);
    legend('Dirt', 'Optimal trajectory', 'Location','northeastoutside');
    set(gca, 'YDir', 'reverse');
```

```
    yticks(1:grid.num_rows);
    yticklabels(arrayfun(@num2str, flip(1:grid.num_rows), 'UniformOutput', false));
    set(gca, 'YTick', [], 'XTick', []);
    xlabel('Column');
    ylabel('Row');
    title('Robot Vacuum Cleaner Trajectory');

    for i = 1:grid.num_rows
        for j = 1:grid.num_columns
            if grid.stage_cost(i, j) == grid.cost_dirt
                text(j-0.5, i-0.5, 'Dirt', 'HorizontalAlignment', 'center', ...
                'VerticalAlignment', 'middle');
            elseif grid.stage_cost(i, j) == grid.cost_charger
                text(j-0.5, i-0.5, 'Charger', 'HorizontalAlignment', 'center', ...
                'VerticalAlignment', 'middle');
            elseif grid.stage_cost(i, j) == grid.cost_carpet
                text(j-0.5, i-0.5, 'Carpet', 'HorizontalAlignment', 'center', ...
                'VerticalAlignment', 'middle');
            elseif grid.stage_cost(i, j) == grid.cost_obstacle
                text(j-0.5, i-0.5, 'Obstacle', 'HorizontalAlignment', 'center', ...
                'VerticalAlignment', 'middle', 'Color', 'w');
            end
        end
    end
    hold off;
end
```

(a)

Run the code, we can get cost-to-go and optimal policies:

```
    J =
     13     12      6      0      0
     14    Inf     12      6      0
     15     16     17     12      6
     16     17     23     18     24
    U =
     2      2      2      2      0
     1     -1      1      1      1
     1      4      4      1      1
     1      1      1      1      4
```

(b)

Run the code, and we can get the map:

```
    map =
     6      7      8      9     10
     5      0      0      0      0
     4      3      2      0      0
     0      0      1      0      0
```

The robot will first move towards the dirt, then pass through all dirts, and then go straight to the charger.

(c)

Run the code, and we can get optimal policy and the map:

```
U1 =
  2     2     2     2     0
  1    -1     1     1     1
  1     2     1     1     1
  1     1     1     1     4
map =
  0     0     4     5     6
  0     0     3     0     0
  0     0     2     0     0
  0     0     1     0     0
```

The robot moves straight up, then to the right, and finally to the Charger

(d)

1. Set a reward for leaving the charging station to encourage the robot to move;
2. Set lower costs for dirts to make sure that clean these dirts a priority for the robot;
3. Set lower costs for the paths between dirts to make sure that the robot will find the efficient way to move.
4. After all dirts are cleaned, make the cost of returning to the charging station the lowest among all other cells to make sure the robot will return.
5. Obstacle and carpet have a high cost.

Problem 1.3 Approximate Dynamic Programming

```
% cfr_a1_3: Main script for Problem 1.3 Approximate Dynamic Programming.
%
% adapted from: Borrelli, Francesco: "ME 231 Experiential Advanced Control
% Design I"
%
% --
% Control for Robotics
% Summer 2023
% Assignment 1
%
% --
% Technical University of Munich
% Learning Systems and Robotics Lab
%
% Course Instructor:
% Angela Schoellig
% schoellig@utias.utoronto.ca
%
% Teaching Assistants:
% SiQi Zhou: siqi.zhou@tum.de
% Lukas Brunke: lukas.brunke@tum.de
%
% --
% Revision history
% [22.01.17, LB]    first version
% [22.01.23, LB]    added 2 (c) to the code, removed N
%
% --
% Revision history
% [22.01.17, LB]    first version
% [22.01.24, LB]    updated horizon and initial state

clear all
close all
clc

%% set up system

% inverted pendulum parameters
l = 1.0; % length
g = 9.81; % gravitational constant
m = 1.0; % mass

% create inverted pendulum system
sys = InvertedPendulum(l, g, m);

% controller parameters
Q = diag([1, 0.1]);
R = 1;
N = 25;

% linearization point
x_up = [pi; 0];

```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TODO: YOUR CODE HERE - Exercise 3 (a)
%% Linearize the nonlinear continuous-time control system

% Calculate the Jacobian matrix around the upright position x_up = [pi; 0] with no control input
% Define the system dynamics function
f = @(x) [x(2); -g/l * sin(x(1)) + (1/m*l^2).* u];
% A_c and B_c
A_c = [0, 1; -g/l * cos(x_up(1)), 0];
B_c = [0; 1];

save('a1_3.mat', 'A_c', 'B_c');
%% Discretize the continuous-time control system

% Sampling time
Delta_t = 0.1;
% Use Matlab's c2d function to discretize the system
sys_d = c2d(ss(A_c, B_c, eye(2), 0), Delta_t);
% Extract discrete-time system matrices
A_d = sys_d.A;
B_d = sys_d.B;

save('a1_3.mat', 'A_d', 'B_d');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% cost functions

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TODO: YOUR CODE HERE - Exercise 3 (b)

stage_cost = @(x, u) x' * Q * x + u' * R * u;
initial_cost_to_go = @(x) x' * Q * x;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% calculate optimal control using dynamic programming and gridding

% grid state-space
num_points_x1 = 10;
num_points_x2 = 5;
X1 = linspace(-pi/4, pi/4, num_points_x1);
X2 = linspace(-pi/2, pi/2, num_points_x2);

% allocate arrays for optimal control inputs and cost-to-go
U = zeros(num_points_x1, num_points_x2);
J = zeros(num_points_x1, num_points_x2);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% TODO: YOUR CODE HERE - Exercise 3 (c)

% Initialize J with initial cost-to-go
for i = 1:num_points_x1
    for j = 1:num_points_x2
        x = [X1(i); X2(j)];
        J(i,j) = initial_cost_to_go(x);
```

```
    end
end

% Dynamic programming to compute optimal policy
for k = N-1:-1:0
    J_new = J;
    for i = 1:num_points_x1
        for j = 1:num_points_x2
            x = [X1(i); X2(j)];
            min_cost = inf;
            optimal_u = 0;
            for u = -1:0.1:1
                x_next = A_d * x + B_d * u;
                % Interpolate cost-to-go for x_next
                cost_to_go = interp2(X1, X2, J', x_next(1), x_next(2), 'spline');
                cost = stage_cost(x, u) + cost_to_go;
                if cost < min_cost
                    min_cost = cost;
                    optimal_u = u;
                end
            end
            J_new(i, j) = min_cost;
            U(i, j) = optimal_u;
        end
    end
    J = J_new;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% plot optimal control and cost-to-go
figure
subplot(1, 2, 1)
surf(X1, X2, U')
xlabel('x_1')
ylabel('x_2')
zlabel('u')
subplot(1, 2, 2)
surf(X1, X2, J')
xlabel('x_1')
ylabel('x_2')
zlabel('J')

%% apply control law and simulate inverted pendulum
% create the controlled inverted pendulum system
control_sys = InvertedPendulum(l, g, m, X1, X2, U, x_up);

% initial condition
x0 = x_up + [-pi/6; 0];

% duration of simulation
t = [0, 10];

% simulate control system
[t, x] = ode45(@control_sys.controlled_dynamics, t, x0);

```

```
% determine control inputs from trajectory
u = zeros(size(t));
for i = 1 : length(t)
    u(i) = control_sys.mu(x(i, :)' - x_up);
end

%% plot state and input trajectories
figure
subplot(2, 1, 1)
hold on
plot(t, x(:, 1))
plot(t, x(:, 2))
xlabel('t')
ylabel('x_1 and x_2')
hold off
legend('\theta','d\theta/dt')
grid on
subplot(2, 1, 2)
plot(t, u)
xlabel('t')
ylabel('u')
grid on
```

(a)

```
% TODO: YOUR CODE HERE - Exercise 3 (a)
%% Linearize the nonlinear continuous-time control system

% Calculate the Jacobian matrix around the upright position x_up = [pi; 0] with no control input
% Define the system dynamics function
f = @(x) [x(2); -g/l * sin(x(1)) + (1/m*l^2).* u];
% A_c and B_c
A_c = [0, 1; -g/l * cos(x_up(1)), 0];
B_c = [0; 1];

save('a1_3.mat', 'A_c', 'B_c');
%% Discretize the continuous-time control system

% Sampling time
Delta_t = 0.1;
% Use Matlab's c2d function to discretize the system
sys_d = c2d(ss(A_c, B_c, eye(2), 0), Delta_t);
% Extract discrete-time system matrices
A_d = sys_d.A;
B_d = sys_d.B;

save('a1_3.mat', 'A_d', 'B_d');
```

Run the code, get:

```
    A_c =
            0    1.0000
       9.8100         0

```

```
B_c =
    0
    1

A_d =
    1.0495    0.1016
    0.9971    1.0495

B_d =
    0.0050
    0.1016
```

(b)

```
% TODO: YOUR CODE HERE - Exercise 3 (b)

stage_cost = @(x, u) x' * Q * x + u' * R * u;
initial_cost_to_go = @(x) x' * Q * x;
```

(c)

Yes, the controller stabilize the system at the upright position.

(d)

- **Effect of Q:**
  Q controls the weighting of the state variables, indicating their impact on the cost function. If certain states have a greater impact on the system's performance, larger values can be assigned to the corresponding elements in Q to emphasize their importance.
- **Effect of r:**
  r controls the weighting of the control input, indicating its impact on the cost function. Higher values of r imply a greater emphasis on minimizing the magnitude of the control input, whereas smaller values of r indicate a willingness to allow larger control inputs.

(e)

In a high-dimensional system, the computational complexity will be much higher than that of two-dimensional systems, which is likely to bring about curse of dimensionality in optimization. Therefore, some measures need to be considered.

- Provide more memory and computational requirements.
- Use more efficient interpolation methods.

- Increase the number of iterations and iteration time.
- Consider using parallel computing.

Problem 1.4 Infinite Horizon Dynamic Programming

(a)

$$y_k = x_k + v_k \tag{4.1}$$
$$u_k = -\frac{1}{2}y_k \tag{4.2}$$
$$\Rightarrow u_k = -\frac{1}{2}(x_k + v_k) \tag{4.3}$$
$$x_{k+1} = x_k + u_k + w_k \tag{4.4}$$
$$x_{k+1} = x_k - \frac{1}{2}(x_k + v_k) + w_k \tag{4.5}$$
$$\tag{4.6}$$

So the closed-loop dynamics:

$$x_{k+1} = \frac{1}{2}x_k - \frac{1}{2}v_k + w_k \tag{4.7}$$

(b)

Considering that the expected values of v_k and w_k are zero:

$$\mathbb{E}[v_k] = 0 \tag{4.8}$$
$$\mathbb{E}[w_k] = 0 \tag{4.9}$$

Taking the expectation of the closed-loop dynamics equation:

$$\mathbb{E}[x_{k+1}] = \mathbb{E}[\frac{1}{2}x_k - \frac{1}{2}v_k + w_k] \tag{4.10}$$

Since expectation is a linear operator:

$$\mathbb{E}[x_{k+1}] = \frac{1}{2}\mathbb{E}[x_k] - \frac{1}{2}\mathbb{E}[v_k] + \mathbb{E}[w_k] \tag{4.11}$$
$$\mathbb{E}[x_{k+1}] = \frac{1}{2}\mathbb{E}[x_k] - \frac{1}{2} \cdot 0 + 0 \tag{4.12}$$
$$\mathbb{E}[x_{k+1}] = \frac{1}{2}\mathbb{E}[x_k] \tag{4.13}$$

So the eigenvalue of this system is $r = \frac{1}{2}$, and $|r| = \frac{1}{2} < 1$, so this closed-loop system is stable.

(c)

$$\mathrm{Var}(v_k) = \mathbb{E}[v_k^2] - \mathbb{E}[v_k]^2 = 1 \tag{4.14}$$

$$\mathrm{Var}(w_k) = \mathbb{E}[w_k^2] - \mathbb{E}[w_k]^2 = 1 \tag{4.15}$$

Variance of x_k

$$\mathrm{Var}(x_{k+1}) = \mathrm{Var}(\frac{1}{2}x_k - \frac{1}{2}v_k + w_k) \tag{4.16}$$

$$\mathrm{Var}(x_{k+1}) = (\frac{1}{2})^2\mathrm{Var}(x_k) + (\frac{1}{2})^2\mathrm{Var}(v_k) + \mathrm{Var}(w_k) \tag{4.17}$$

$$\mathrm{Var}(x_{k+1}) = \frac{1}{4}\mathrm{Var}(x_k) + \frac{1}{4} + 1 = \frac{1}{4}\mathrm{Var}(x_k) + \frac{5}{4} \tag{4.18}$$

At steady state: $\mathrm{Var}(x_{k+1}) = \mathrm{Var}(x_k)$, so:

$$\mathrm{Var}(x_{k+1}) = \frac{1}{4}\mathrm{Var}(x_k) + \frac{5}{4} \tag{4.19}$$

$$\Rightarrow \mathrm{Var}(x_k) = \frac{5}{3} \tag{4.20}$$

Variance of u_k

$$u_k = -\frac{1}{2}(x_k + v_k) \tag{4.21}$$

$$\mathrm{Var}(u_k) = \mathrm{Var}(-\frac{1}{2}(x_k + v_k)) \tag{4.22}$$

$$\mathrm{Var}(u_k) = (-\frac{1}{2})^2 \cdot \mathrm{Var}(x_k + v_k) \tag{4.23}$$

$$\mathrm{Var}(u_k) = \frac{1}{4} \cdot (\frac{5}{3} + 1) = \frac{2}{3} \tag{4.24}$$

Because the goal is to move the robot's position to $x = 0$. So $\mathbb{E}[x_k] = 0$ and $\mathbb{E}[u_k] = 0$

So:

$$\mathbb{E}[x_k^2] = \mathrm{Var}[x_k] + \mathbb{E}[x_k]^2 = \frac{5}{3} \tag{4.25}$$

$$\mathbb{E}[u_k^2] = \mathrm{Var}[u_k] + \mathbb{E}[u_k]^2 = \frac{2}{3} \tag{4.26}$$

So the infinite horizon cost J:

$$J = \mathbb{E}_{w_k,v_k}[\frac{x_k^2}{2} + u_k^2] \tag{4.27}$$

$$J = \frac{1}{2} \cdot \mathbb{E}[x_k^2] + \mathbb{E}[u_k^2] \tag{4.28}$$

$$J = \frac{1}{2} \cdot \frac{5}{3} + \frac{2}{3} \tag{4.29}$$

$$J = \frac{3}{2} \tag{4.30}$$

(d)

The *Algebraic Riccatti equation*

$$P = A^T PA - (A^T PB)(R + B^T PB)^{-1}(B^T PA) + Q \tag{4.31}$$

Because there is no noise for this problem, so

$$x_{k+1} = x_k + u_k \tag{4.32}$$
$$y_k = x_k \tag{4.33}$$

So $A = 1, B = 1, Q = \frac{1}{2}, R = 1$

$$P = 1 \cdot P \cdot 1 - (1 \cdot P \cdot 1)(1 + 1 \cdot P \cdot 1)^{-1}(1 \cdot P \cdot 1) + \frac{1}{2} \tag{4.34}$$
$$\Rightarrow P = 1 \text{ or } P = -\frac{1}{2} \tag{4.35}$$

The optimal feedback gain α is given by:

$$\alpha = -(B^T PB + R)^{-1}(B^T PA) \tag{4.36}$$
$$\alpha = -(P + 1)^{-1}(P) \tag{4.37}$$
$$\alpha = -\frac{1}{2} \text{ or } \alpha = \frac{1}{4} \tag{4.38}$$

Because $\alpha < 0$, so the optimal feedback gain $\alpha = -\frac{1}{2}$.