Robot Learning

Winter Semester 2020/2021, Homework 2

Prof. Dr. J. Peters, J. Watson, J. Carvalho, J. Urain and T. Dam

TECHNISCHE UNIVERSITÄT DARMSTADT

Total points: 43
Due date: Midnight, Tuesday, 01 December 2020
Übungsblatt 2

Aufgabe 2.1: Optimal Control (20 Punkte)

In this exercise, we consider a finite-horizon discrete time-varying Stochastic linear Quadratic Regulator with Gaussian noise and time-varying quadratic reward function. Such system is defined as

$$s_{t+1} = A_t s_t + B_t a_t + w_t$$

where s_t is the state, a_t is the control signal, $w_t \sim \mathcal{N}(b_t, \Sigma_t)$ is Gaussian additive noise with mean b_t and covariance Σ_t and $t = 0, 1, \ldots, T$ is the time horizon. The control signal a_t is computed as

$$a_t = -K_t s_t + k_t$$

and the reward function is

$$\text{reward}_t = \begin{cases} -(s_t - r_t)^\top R_t (s_t - r_t) - a_t^\top H_t a_t & \text{when} \quad t = 0, 1, \ldots, T-1 \\ -(s_t - r_t)^\top R_t (s_t - r_t) & \text{when } t = T \end{cases}$$

2.1a) Implementation (8 Punkte)

Implement the LQR with the following properties

$$s_0 \sim \mathcal{N}(0, I) \qquad T = 50$$

$$A_t = \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix} \qquad B_t = \begin{bmatrix} 0 \\ 0.1 \end{bmatrix}$$

$$b_t = \begin{bmatrix} 5 \\ 0 \end{bmatrix} \qquad \Sigma_t = \begin{bmatrix} 0.01 & 0 \\ 0 & 0.01 \end{bmatrix}$$

$$K_t = \begin{bmatrix} 5 & 0.3 \end{bmatrix} \qquad k_t = 0.3$$

$$H_t = 1 \qquad R_t = \begin{cases} \begin{bmatrix} 100000 & 0 \\ 0 & 0.1 \end{bmatrix} & \text{if} \quad t = 14 \text{ or } 40 \\ \begin{bmatrix} 0.01 & 0 \\ 0 & 0.1 \end{bmatrix} & \text{otherwise} \end{cases} \qquad r_t = \begin{cases} \begin{bmatrix} 10 \\ 0 \end{bmatrix} & \text{if} \quad t = 0, 1, \ldots, 14 \\ \begin{bmatrix} 20 \\ 0 \end{bmatrix} & \text{if} \quad t = 15, 16, \ldots, T \end{cases}$$

Execute the system 20 times. Plot the mean and 95% confidence (see ”68-95-99.7 ruleänd matplotlib.pyplot.fill_between function) over the different experiments of the state and s_t and of the control signal a_t over time. How does the system behave? Compute and write down the mean and the standard deviation of the cumulative reward over the experiments. Attach a snippet of your code.

Vorname	Name	Matrikel-Nr.
Yi	Cui	2758172
Yuting	Li	2547040
Liaotian	Zhihao	2897965

Lösungsvorschlag:

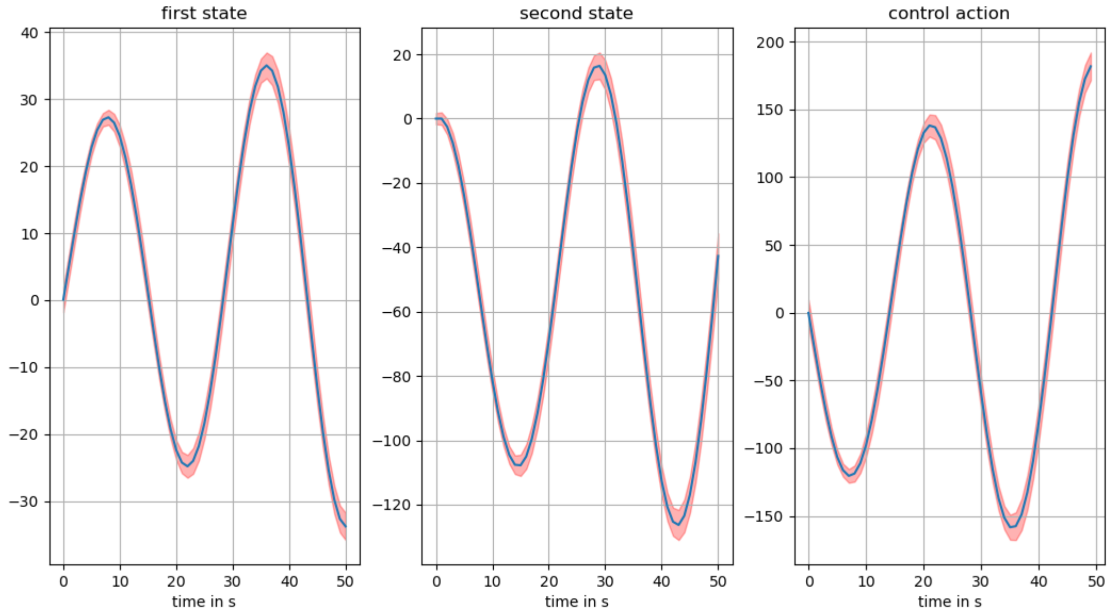

Abbildung 1: mean and 95% confidence of state and control

* mean of cumulative reward: -2645604.75126133
* standard deviation of cumulative reward: 579933.57403004

The system behaves instabile, that the amplitude of all signals are oscillated divergent, which means the pole points locate at the real plane of complex space (Not on the real axle).

```
import numpy as np
import matplotlib.pyplot as plt


class LQR():
    def __init__(self):
        self.A = np.array([[1, .1],
                           [0, 1]])
        self.B = np.array([0, .1]).reshape(-1, 1)
        self.b = np.array([5, 0]).reshape(-1, 1)
        self.Sigma = np.diag([0.01, 0.01])
        self.K = np.array([5, 0.3])
        self.T = 50
        self.k = 0.3
        self.H = 1
        self.R = [np.diag([0.01, 0.1]), np.diag([1e5, 0.1])]
        self.r = [np.array([10, 0]).reshape(-1, 1),
                  np.array([20, 0]).reshape(-1, 1)]

    def Gaussian(self, mean, cor, size=(2, 1)):
        """
```

```
        Contribute a multivariate Gausian distribution array
        :param mean:    [n, 1] array, Mean Array
        :param cor:     [n, n] array, Covariance Matrix
        :param size:    tuple
        :return G:      [n, 1] array, multivariate Gausian Matrix
        """
        # Gaussian_M = 1/( (2*np.pi) * np.linalg.norm(cor, 2)) * \
        #              np.exp(-1/2 * mean.T @ np.linalg.pinv(cor) @ mean)
        if mean.shape != size:
            size = mean.shape

        G = np.zeros(size)
        for i in range(size[0]):
            G[i] = np.random.normal(mean[i], np.sqrt(cor[i, i]))
        return G

    def iteration(self):
        """
        execute the iteration to compute the final states
        :return s_list:  [2, n] array. all states under LQR
        :return a_list:  [1, n] array. all action under LQR
        :return rewards:  [1, n] array. rewards of each step
        """
        # initialize
        s_list = np.zeros((2, self.T + 1))
        a_list = np.zeros((1, self.T + 1))
        rewards = np.zeros((1, self.T + 1))

        for i in range(0, self.T+1):
            # compute state
            if i == 0:
                s = self.Gaussian(np.zeros((2, 1)), np.eye(2))
            else:
                s = self.A @ s + self.B * a + w

            if i < self.T:
                a = -self.K @ s + self.k
            else:
                a = 0
            w = self.Gaussian(self.b, self.Sigma)

            # compute reward
            if i == 14 or i == 40:
                R = self.R[1]
            else:
                R = self.R[0]

            if i <= 14:
                r = self.r[0]
            else:
                r = self.r[1]

            reward = -(s - r).T @ R @ (s - r) - a * self.H * a

            # assign in array
            s_list[:, i] = s.reshape(-1)
            a_list[:, i] = a
            rewards[:, i] = reward

        return s_list, a_list, rewards
```

```python

    def visualisation(self, execution=20):
        """
        plot the mean and 95% confidence with 20 times execution
        :param execution:    int, execution times
        """
        # initial
        states_1 = np.zeros((execution, self.T + 1))
        states_2 = np.zeros((execution, self.T + 1))
        actions = np.zeros((execution, self.T + 1))
        rewards_cum = np.zeros((execution, 1))

        for i in range(execution):
            s_list, a_list, rewards = self.iteration()
            states_1[i, :] = s_list[0, :]
            states_2[i, :] = s_list[1, :]
            actions[i, :] = a_list.reshape(-1)
            rewards_cum[i, 0] = np.sum(rewards)

        print("cumulatgive reward: {}  +- {}".format(np.mean(rewards_cum), np.std(rewards_cum)))

        mean_s = np.zeros((2, self.T + 1))
        std_s = np.zeros((2, self.T + 1))
        mean_s[0, :] = np.mean(states_1, axis=0)
        mean_s[1, :] = np.mean(states_2, axis=0)
        std_s[0, :] = np.std(states_1, axis=0)
        std_s[1, :] = np.std(states_2, axis=0)
        mean_a = np.mean(actions, axis=0)
        std_a = np.std(actions, axis=0)
        time_series = np.linspace(0, self.T, self.T+1)

        plt.figure()
        ax1 = plt.subplot(1, 3, 1)
        ax1.plot(time_series, mean_s[0, :], color='tab:blue')
        ax1.fill_between(time_series, 2*std_s[0, :]+mean_s[0, :],
                         -2*std_s[0, :]+mean_s[0, :], color='red', alpha=0.3)
        plt.xlabel('time in s')
        plt.title('first state')

        ax1 = plt.subplot(1, 3, 2)
        ax1.plot(time_series, mean_s[1, :], color='tab:blue')
        ax1.fill_between(time_series, 2*std_s[1, :]+mean_s[1, :],
                         -2*std_s[1, :]+mean_s[1, :], color='red', alpha=0.3)
        plt.xlabel('time in s')
        plt.title('second state')

        ax1 = plt.subplot(1, 3, 3)
        ax1.plot(time_series, mean_a, color='tab:blue')
        ax1.fill_between(time_series, 2*std_a+mean_a, -2*std_a+mean_a, color='red', alpha=0.3)
        plt.xlabel('time in s')
        plt.title('control action')

        plt.show()


if __name__ == '__main__':
    self = LQR()
    self.visualisation()
```

2.1b) LQR as a P controller (4 Punkte)

The LQR can also be seen as a simple P controller of the form

$$a_t = K_t \left(s_t^{\text{des}} - s_t \right) + k_t$$

which corresponds to the controller used in the canonical LQR system with the introduction of the target s_t^{des} . Assume as target

$$s_t^{\text{des}} = r_t = \begin{cases} \begin{bmatrix} 10 \\ 0 \end{bmatrix} & \text{if } t = 0, 1, \ldots, 14 \end{cases}$$

Use the same LQR system as in the previous exercise and run 20 experiments. Plot in one figure the mean and 95% confidence (see ”68-95-99.7 ruleänd matplotlib.pyplot.fill_between function) of the first dimension of the state, for both $s_t^{\text{des}} = r_t$ and $s_t^{\text{des}} = 0$.

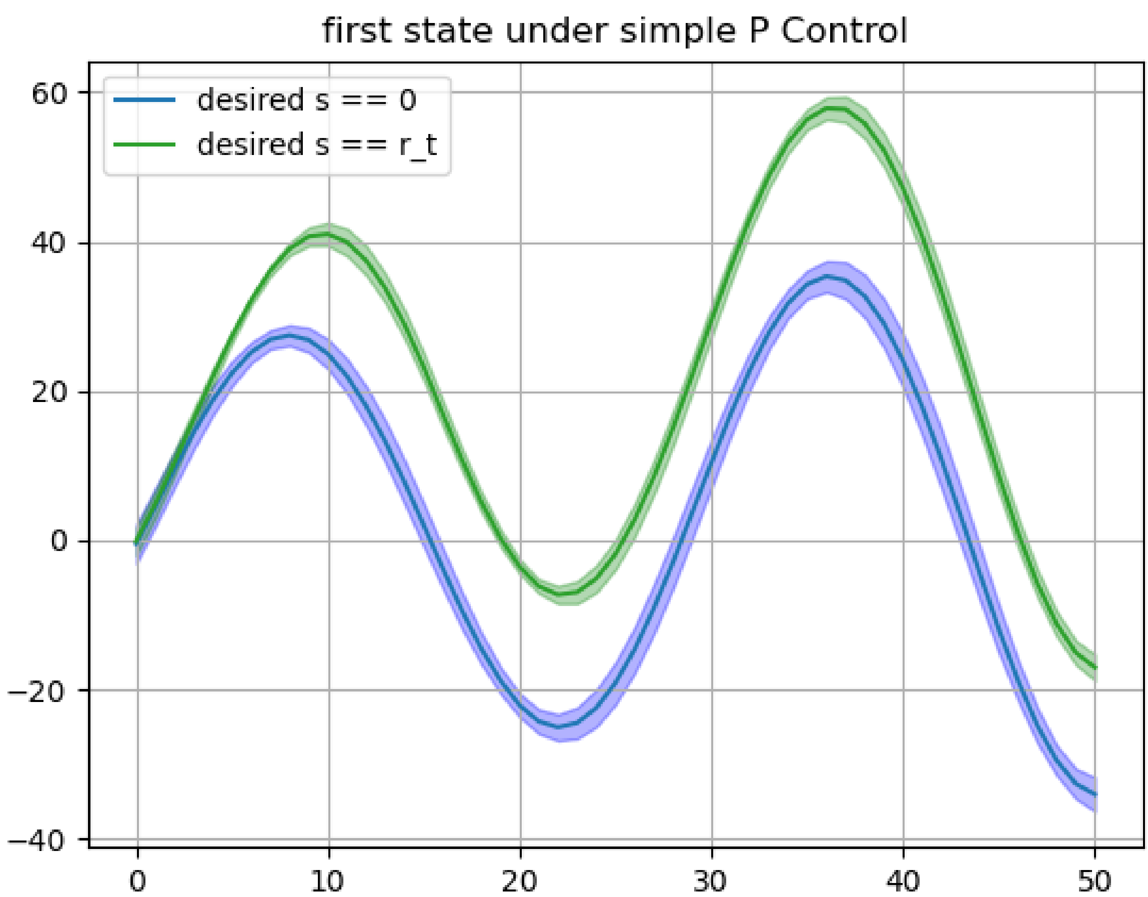

Abbildung 2: comparison the mean and 95% confidence of first state between different desired state

Vorname	Name	Matrikel-Nr.
Yi	Cui	2758172
Yuting	Li	2547040
Liaotian	Zhihao	2897965

```python
def P_controll(self, i, s):
    """
    modify control task as a simple P Controller
    :param i: int, iteration times
    :param s: int, actual state in i.th iteration
    :param k: int, constant action
    :return: a int, control action under P-Controller
    """
    if i <= 14:
        s_des = self.r[0]
    else:
        s_des = self.r[1]
    a = self.K @ (s_des - s) + self.k
    return a
```

```python
def iteration(self, case="default"):
    """
    execute the iteration to compute the final states
    :param case:    string, case of different task
                        "default" : task 2.1
                        "P_Control" : task 2.2
    :return s_list:  [2, n] array. all states under LQR
    :return a_list:  [1, n] array. all action under LQR
    :return rewards:  [1, n] array. rewards of each step
    """

    # initialize
    s_list = np.zeros((2, self.T + 1))
    a_list = np.zeros((1, self.T + 1))
    rewards = np.zeros((1, self.T + 1))

    for i in range(0, self.T+1):
        # compute state
        if i == 0:
            s = self.Gaussian(np.zeros((2, 1)), np.eye(2))
        else:
            s = self.A @ s + self.B * a + w

        if i < self.T:
            if case == "default":
                a = -self.K @ s + self.k
            elif case == "P_Control":
                a = self.P_controll(i, s)
        else:
            a = 0
        w = self.Gaussian(self.b, self.Sigma)

        # compute reward
        if i == 14 or i == 40:
            R = self.R[1]
        else:
            R = self.R[0]

        if i <= 14:
            r = self.r[0]
        else:
            r = self.r[1]

        reward = -(s - r).T @ R @ (s - r) - a * self.H * a

        # assign in array
```

```
            s_list[:, i] = s.reshape(-1)
            a_list[:, i] = a
            rewards[:, i] = reward

        return s_list, a_list, rewards
```

2.1c) Optimal LQR (8 Punkte)

To compute the optimal gains K_t and k_t, which maximize the cumulative reward, we can use an analytic optimal solution. This controller recursively computes the optimal action by

$$a_t^* = -\left(H_t + B_t^T V_{t+1} B_t\right)^{-1} B_t^T \left(V_{t+1}\left(A_t s_t + b_t\right) - v_{t+1}\right), \tag{6}$$

which can be decomposed into

$$K_t = -\left(H_t + B_t^T V_{t+1} B_t\right)^{-1} B_t^T V_{t+1} A_t, \tag{7}$$

$$k_t = -\left(H_t + B_t^T V_{t+1} B_t\right)^{-1} B_t^T \left(V_{t+1} b_t - v_{t+1}\right). \tag{8}$$

where

$$M_t = B_t \left(H_t + B_t^T V_{t+1} B_t\right)^{-1} B_t^T V_{t+1} A_t \tag{9}$$

$$V_t = \begin{cases} R_t + \left(A_t - M_t\right)^T V_{t+1} A_t & \text{when} \quad t = 1 \dots T-1 \\ R_t & \text{when} \quad t = T \end{cases} \tag{10}$$

$$v_t = \begin{cases} R_t r_t + \left(A_t - M_t\right)^T \left(v_{t+1} - V_{t+1} b_t\right) & \text{when} \quad t = 1 \dots T-1 \\ R_t r_t & \text{when} \quad t = T \end{cases} \tag{11}$$

Run 20 experiments with $s_t^{\text{des}} = 0$ computing the optimal gains K_t and k_t. Plot the mean and 95% confidence (see "68–95–99.7 rule" and matplotlib.pyplot.fill_between function) of both states for all three different controllers used so far. Use one figure per state. Report the mean and std of the cumulative reward for each controller and comment the results. Attach a snippet of your code.

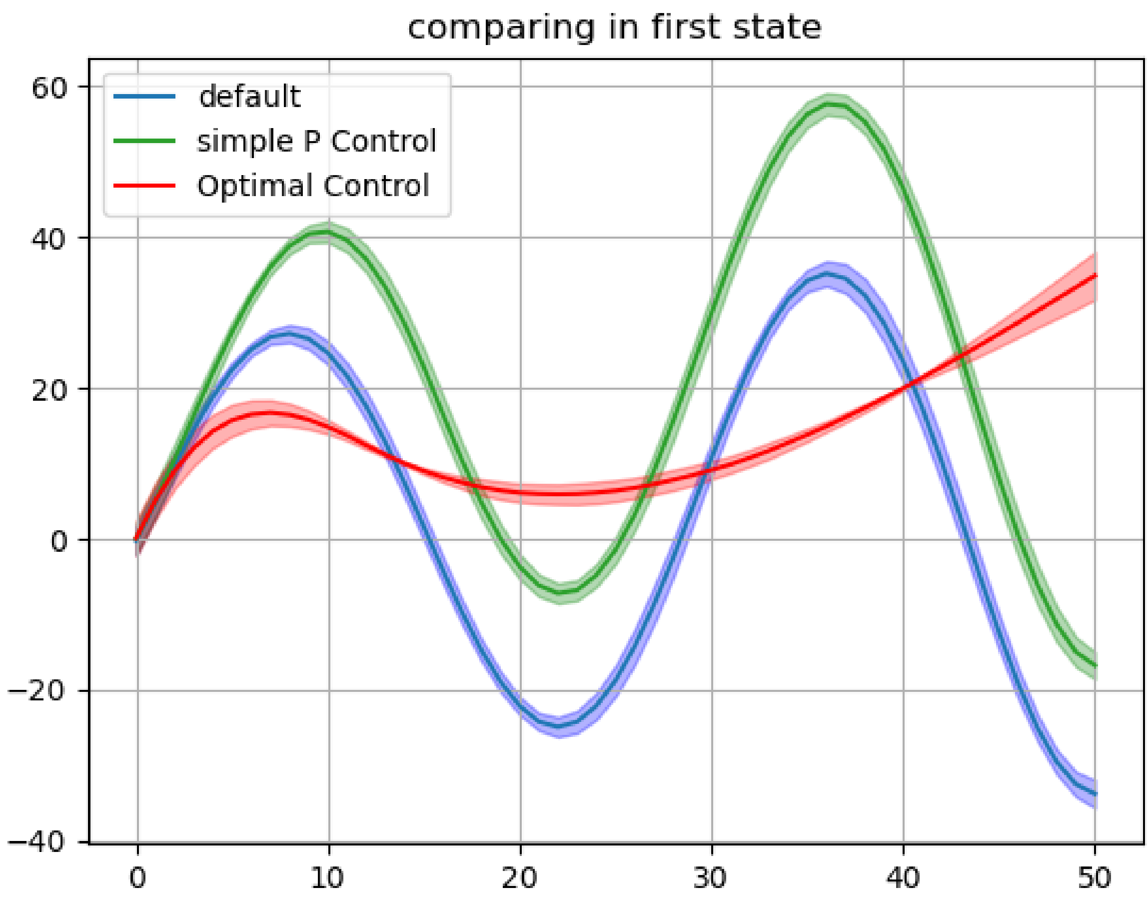

Abbildung 3: first state under different control strategies

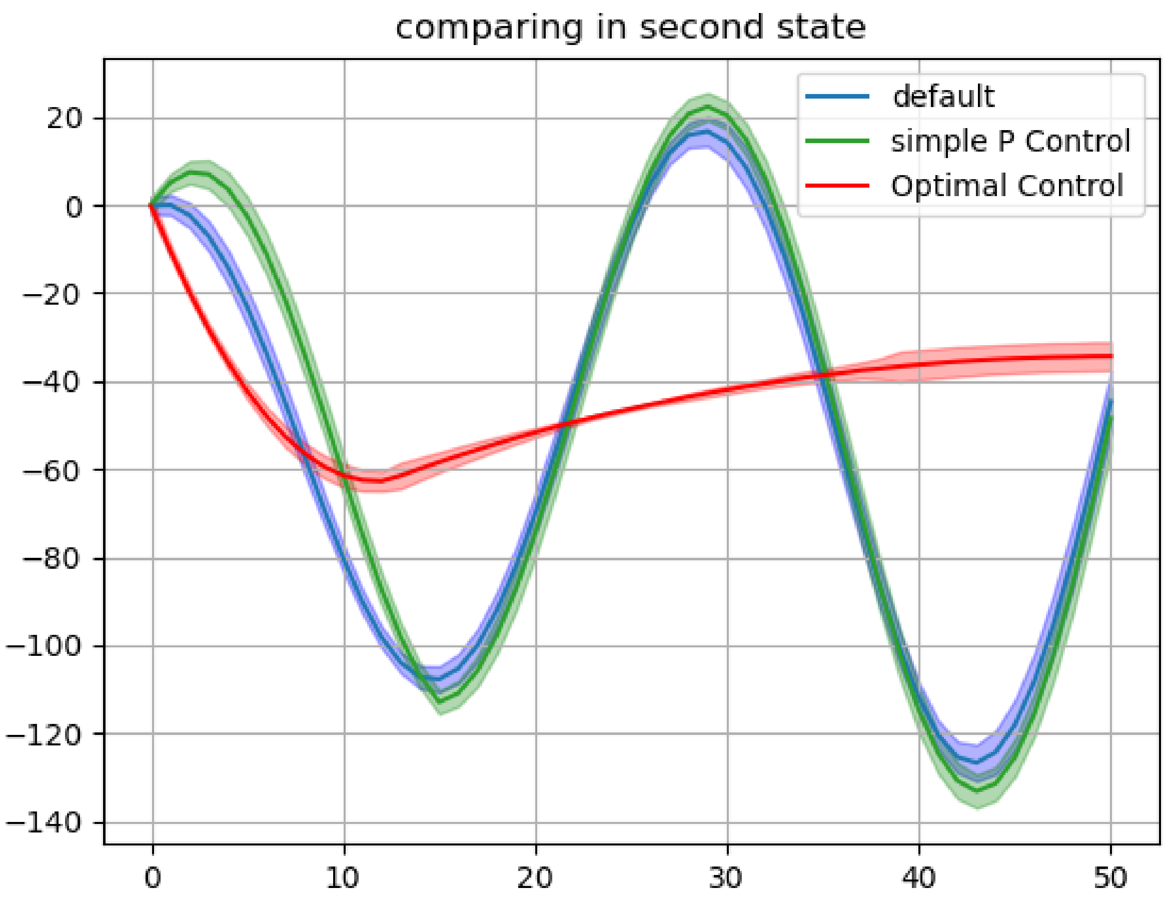

Abbildung 4: second state under different control strategies

	default	P Control	optimal Control
mean	-2645604.7512	-107234043.6590	-62085.9937
std	579933.5740	10610422.2334	4435.3507

From the response we could find, that the default strategy is a special case of simple P Control, which means the different only occurs in desired states. Both of them are unstable under respective desired states.
In contrast to this optimal control performs much better, especially in stability. There isn't any periodic oscillation in response, moreover the response of second state will be convergent.
Reward value proves that the optimal control has the last cost of three strategies, which correspond with the response of state.
In each step, optimal strategy always chooses the best control parameter (e.g. local optimal parameter) of actual state. In ideal case it will be fast convergent.
Furthermore, the first element of B matrix is zero, which means the control action has none effect on this state. Hence it performs seemingly divergent.

```
    def optimal(self):
        """
        implement the Optimal LQR with reverse
        :return K_list:  [n x 1] list, catch of all actual Control Matrix
        :return k_list:  [n x 1] list, catch of all actual Control Constant
        """
        K_list = []
        k_list = []

        for i in reversed(range(1, self.T+1)):
            if i == 14 or i == 40:
                R = self.R[1]
            else:
                R = self.R[0]

            if i <= 14:
                r = self.r[0]
            else:
                r = self.r[1]

            if i == self.T:
                v_t = R @ r
                V_t = R
            else:
                v_t = R @ r + (self.A - M_t).T @ (v_t - V_t @ self.b)
                V_t = R + (self.A - M_t).T @ V_t @ self.A

            M_t = 1/(self.H + self.B.T @ V_t @ self.B) * self.B @ self.B.T @ V_t @ self.A  # V_t from l

            K_t = - 1/(self.H + self.B.T @ V_t @ self.B) * self.B.T @ V_t @ self.A
            k_t = - 1/(self.H + self.B.T @ V_t @ self.B) * self.B.T @ (V_t @ self.b - v_t)

            # assignment
            k_list.insert(0, k_t)
            K_list.insert(0, K_t)

        return K_list, k_list
```

```
    def iteration(self, case="default"):
        """
        execute the iteration to compute the final states
        :param case:     string, case of different task
                                "default" : task 2.1
                                "P_Control" : task 2.2
                                "Optimal" : task 2.3
        :return s_list:  [2, n] array. all states under LQR
        :return a_list:  [1, n] array. all action under LQR
        :return rewards:  [1, n] array. rewards of each step
        """

        # initialize
        s_list = np.zeros((2, self.T + 1))
        a_list = np.zeros((1, self.T + 1))
        rewards = np.zeros((1, self.T + 1))

        if case == "Optimal":
            K_lst, k_lst = self.optimal()

        for i in range(0, self.T+1):
            # compute state
            if i == 0:
```

```
                s = self.gaussian(np.zeros((2, 1)), np.eye(2))
            else:
                s = self.A @ s + self.B * a + w

            if i < self.T:
                if case == "default":
                    a = -self.K @ s + self.k
                elif case == "P_Control":
                    a = self.p_controll(i, s)
                elif case == "Optimal":
                    a = K_lst[i] @ s + k_lst[i]
            else:
                a = 0
            w = self.gaussian(self.b, self.Sigma)

            # compute reward
            if i == 14 or i == 40:
                R = self.R[1]
            else:
                R = self.R[0]

            if i <= 14:
                r = self.r[0]
            else:
                r = self.r[1]

            reward = -(s - r).T @ R @ (s - r) - a * self.H * a

            # assign in array
            s_list[:, i] = s.reshape(-1)
            a_list[:, i] = a
            rewards[:, i] = reward

        return s_list, a_list, rewards
```