Robot Learning

Group 80:Lukas Schneider(2565695), Si Jun Kwon(2453134), Michael Erni(2547105)

Sommersemester 2020
Sheet 2

Task 2.1

2.1a)

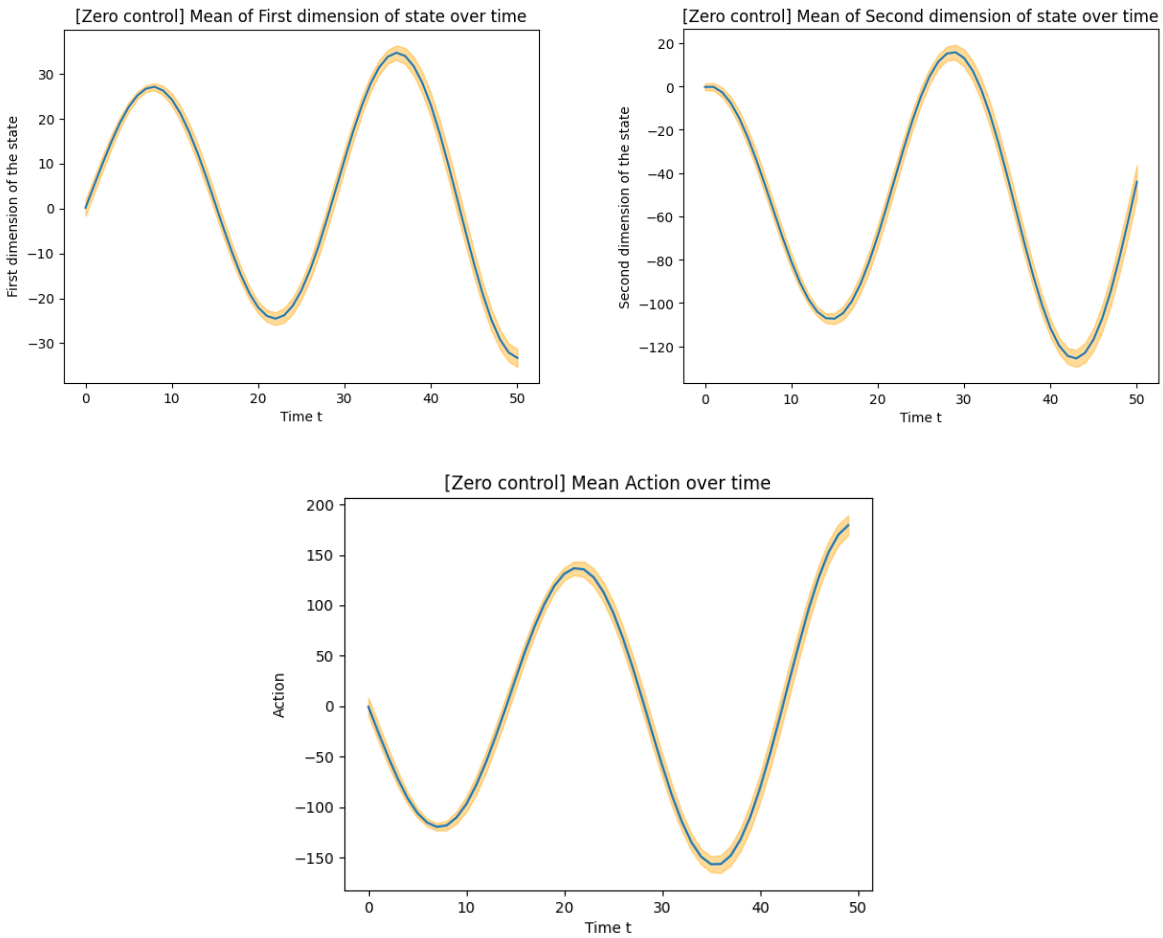

Figure 1: Mean position and 95% confidence interval of the joints with controller a), mean action of the controller a)

The system described in this subtask can be seen as a P controller with $s_t^{\text{des}} = 0$ for all $t = 0, 1, ..., T$. In fact if we take the formula from b and set $s_t^{\text{des}} = 0$ for all $t = 0, 1, ..., T$ we get the same formula to calculate the action in a). Thus the controller in this subtask does not consider the desired state r_t but instead desires for the point $(0\ 0)^T$. It is also

apparent from the plots that the gains are too great and the system is instable hinting at suboptimal gains K_T and offset k_t. First the system drifts off in the positive direction of the first dimension because of the drift component b_t of the modeled SLQR. To compensate this the system takes action into the negative direction(refer to Figure **??**). This propells the second dimension of state into the negative direction(at first the first dimension stays unaffected from the actions as the actions don't affect the first dimension directly). Due to the influence of the second joint on the first joint(refer to $A_{t,(1,2)} = 0.1$) the first joint slowly accelrates into negative direction. It then overshoots $s_t^{\text{des}} = 0$ because the second joint does not consider the velocity approaching s_t^{des}. Analogously the same happens for the compensation in the positive direction. This sequence of events repeats itself, the system becoming more instable with each repetition(the maximal distance to $(0\ 0)^T$ becoming bigger).
The average cumulative reward for this system was -2976946.972171863 with a std of 827060.1944006783.

Code

```
class LQRSim(abc.ABC):
    """
    Class that simulate the LQR system
    """
    def __init__(self) -> None:
        self.s_hist = np.array(
            [np.random.multivariate_normal(np.zeros((2,)), np.eye(2))]
        ).reshape(2, 1)
        self.a_hist = np.zeros((0, ))
        self.T = 50
        self.A_t = np.array([
            [1, .1],
            [0, 1]
        ])
        self.B_t = np.array([0, .1]).reshape(-1, 1)
        self.b_t = np.array([5., 0.])
        self.Sigma_T = np.array([
            [.01, 0],
            [0, .01]
        ])
        self.K_t = np.array([5, .3]).reshape(1, -1)
        self.k_t = .3
        self.H_t = 1
        self.R_t = np.array([[
            [.01, 0],
            [0, .1]
        ]] * self.T)
        self.R_t[14] = np.array([
            [100000, 0],
            [0, .1]
        ])
        self.R_t[40] = np.array([
            [100000, 0],
            [0, .1]
        ])
        self.r_t = np.append(
            np.array([[[10.], [0.]]] * 15),
            np.array([[[20.], [0.]]] * 36),
            axis=0
        ).T[0]
        self.ran = False

    @abc.abstractclassmethod
    def step_func(self, t):
        pass

    def run(self):
        """
        runs the simulation
        """
        if self.ran:
            return
```

```python
        for t in range(self.T):
            current_state = self.s_hist[:, t].reshape(-1, 1)
            action = self.step_func(t)
            noise = np.random.multivariate_normal(self.b_t, self.Sigma_T)\
                .reshape(-1, 1)
            next_state = self.A_t@current_state + self.B_t*action + noise
            self.s_hist = np.append(self.s_hist, next_state, axis=1)
            self.a_hist = np.append(self.a_hist, action)
        self.ran = True

    def calc_reward(self):
        if not self.ran:
            raise Exception(
                "The simulation must be run before rewards can calculated"
            )
        self.reward = np.zeros((self.T, ))
        err = (self.s_hist[:, -1] - self.r_t[:, -1]).reshape(-1, 1)
        self.reward[-1] = -err.T@self.R_t[-1]@err
        for t in range(self.T - 2, -1, -1):
            err = (self.s_hist[:, t] - self.r_t[:, t])
            self.reward[t] = -err.T@self.R_t[t]@err-self.a_hist[t].T *\
                self.H_t*self.a_hist[t]
        return self.reward
```

Listing 1: Base SLQR class that simulates the system

This abstract class summorizes all the common variables and functions used by all tasks commonly. These consists of the static system matrices like A_t, the calculation of the next state given a action taken and calculation of the reward. The subclasses which derive from this class would implement the stepping function that return the action. The stepping function for the controller a is shown in the code listing below.

```python
class LQR1(LQRSim):
    def step_func(self, t):
        current_state = self.s_hist[:, t].reshape(-1, 1)
        action = -self.K_t@current_state + self.k_t
        return action
```

Listing 2: SLQR with controller a

Finally a plotter is used plotting the mean position of the joints, mean action and the mean cumulative rewards. Refer to the listing below.

```python
class Plotter:
    i = 0

    def plot(self, sims, s1=True, s2=True, a=True, r=True, prefix=''):
        to_plot = {}
        if s1:
            to_plot['s1'] = self.i
            self.i += 1

        if s2:
            to_plot['s2'] = self.i
            self.i += 1

        if a:
            to_plot['a'] = self.i
            self.i += 1

        if r:
            to_plot['r'] = self.i
            self.i += 1

        time_states = np.linspace(0, 50, 51)
        time_r_a = np.linspace(0, 49, 50)
        [plt.figure(ii) for ii in range(self.i)]

        states = np.array([sim.s_hist for sim in sims])
```

```
        states_mean = np.mean(states, axis=0)
        states_var = np.var(states, axis=0)

        actions = np.array([sim.a_hist for sim in sims])
        action_mean = np.mean(actions, axis=0)
        action_var = np.var(actions, axis=0)

        cumulative_rewards = np.array(
            [np.cumsum(sim.calc_reward()) for sim in sims]
        )
        reward_mean = np.mean(cumulative_rewards, axis=0)
        reward_std = np.sqrt(np.var(cumulative_rewards, axis=0))
        print(f'{prefix} Mean cumulative reward {reward_mean[-1]}')
        print(f'{prefix} Variance of cumulative reward {reward_std[-1]}')

        if s1:
            plt.figure(to_plot['s1'])
            plt.title(f'[{prefix}] Mean of First dimension of state over time')
            plt.plot(time_states, states_mean[0])
            plt.fill_between(
                time_states,
                states_mean[0] - 2 * np.sqrt(states_var[0]),
                states_mean[0] + 2 * np.sqrt(states_var[0]),
                color='orange',
                alpha=.4
            )
            plt.xlabel('Time t')
            plt.ylabel('First dimension of the state')

        if s2:
            plt.figure(to_plot['s2'])
            plt.title(
                f'[{prefix}] Mean of Second dimension of state over time'
            )
            plt.plot(time_states, states_mean[1])
            plt.fill_between(
                time_states,
                states_mean[1] - 2 * np.sqrt(states_var[1]),
                states_mean[1] + 2 * np.sqrt(states_var[1]),
                color='orange',
                alpha=.4
            )
            plt.xlabel('Time t')
            plt.ylabel('Second dimension of the state')

        if a:
            plt.figure(to_plot['a'])
            plt.title(f'[{prefix}] Mean Action over time')
            plt.plot(time_r_a, action_mean)
            plt.fill_between(
                time_r_a,
                action_mean - 2 * np.sqrt(action_var),
                action_mean + 2 * np.sqrt(action_var),
                color='orange',
                alpha=.4
            )
            plt.xlabel('Time t')
            plt.ylabel('Action')

        if r:
            plt.figure(to_plot['r'])
            plt.title(f'[{prefix}] Mean Cumulative Reward over time')
            plt.plot(time_r_a, reward_mean)
            plt.xlabel('Time t')
            plt.ylabel('Reward')
```
Listing 3: Plotter for the simulations

2.1b)

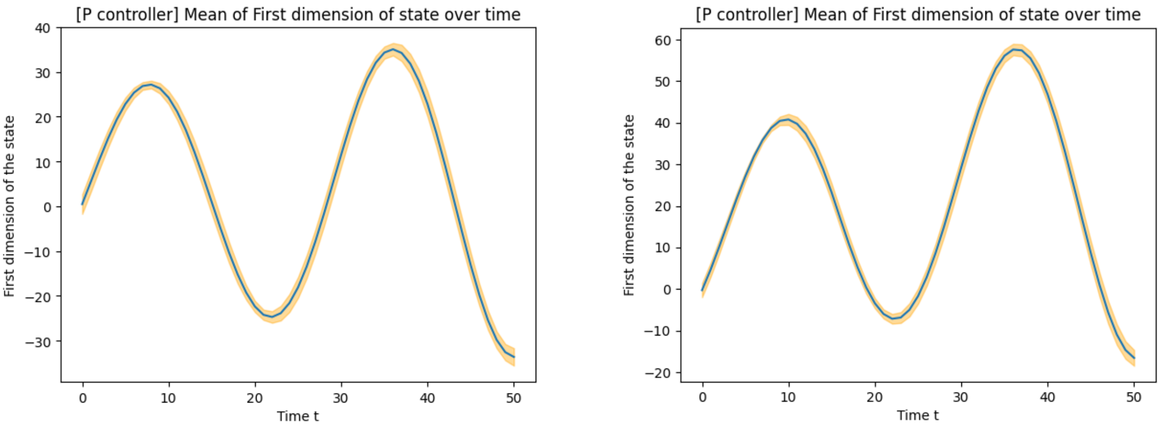

Figure 2: P controller with $s_t^{\text{des}} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ (left) amd with $s_t^{\text{des}} = r_t$ (right)

2.1c)

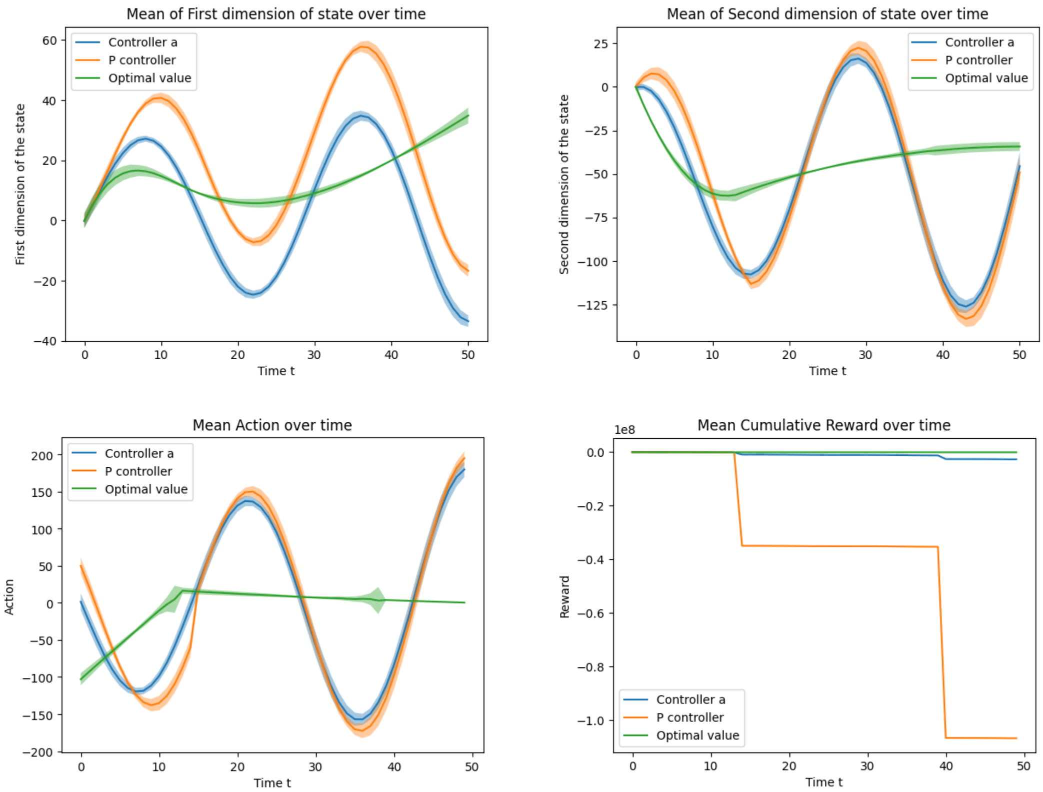

Figure 3: All three controllers

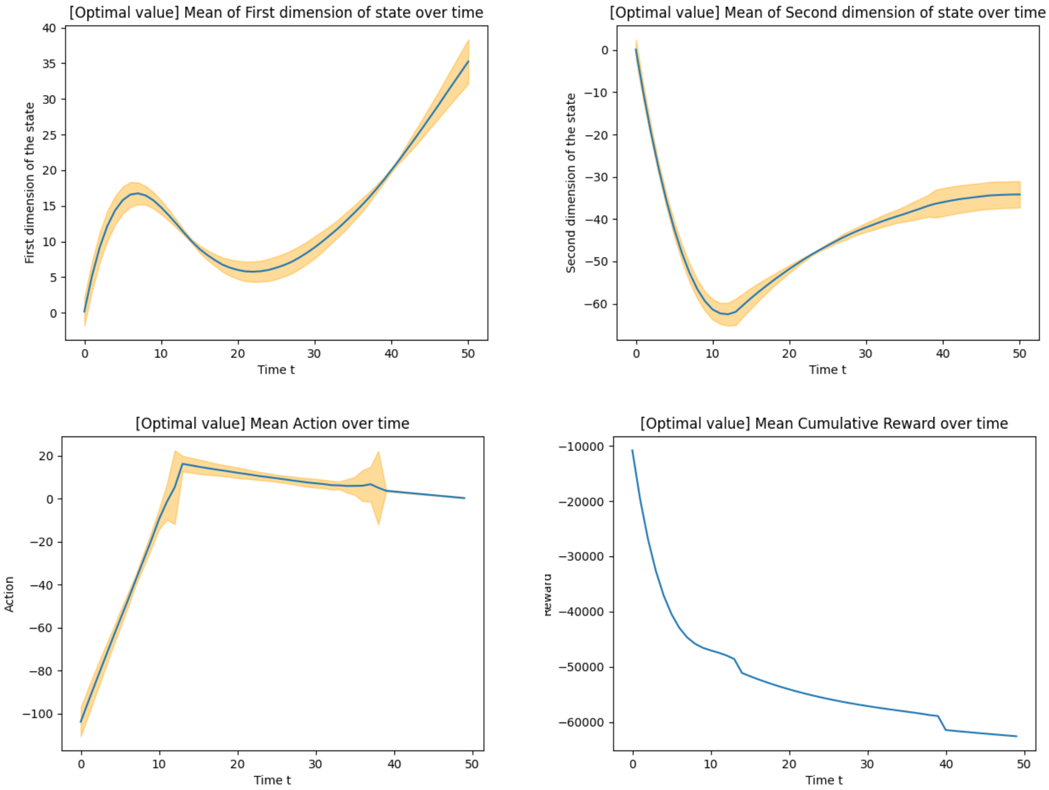

Figure 4: Optimal value controller

Controller type	Mean reward(the bigger the better)	Standard deviation of the rewards
Optimal Controller	-62584.876481951775	5182.129498148187
Controller a/P Controller($s_t^{\text{des}} = 0$)	-2976946.972171863	827060.1944006783
P Controller($s_t^{\text{des}} = r_t$)	-105344044.62913443	8505038.421848932

Figure 5: Table of mean rewards and standard deviation of the rewards in descending order of mean rewards

Looking at each controller one can make following observations:

- The Optimal controller outperfoms the other variants by an outstanding amount. This is due to the fact that the optimal controller does not just take the current error into account but also the long-term reward and maximizes the long-term reward. This is shown by the trajectory of the system being controlled by the optimal controller. The controller overshoots by a little margin first so it can precisely hit the desired position (for the first joint as the first joint gives the biggest reward) at the time point where the reward is maximal($t = 14, 40$). Furthermore the optimal controller delivers is more consistent with its rewards as the standard deviation of rewards is fairly small. Refer to Figure 3 and Figure 4.
- Controller a) does not consider the desired trajectory but still manages to hit closeto the desired point at the maximal reward time points($t = 14, 40$) by sheer coincidence(it desires for the point $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$). Refer to Figure 3.
- Ironically the P controller with $s_t^{\text{des}} = r_t$ performs worse than the Controller a). This is due to the overshooting problem of the P controller. This inprecision is amplified with the given desired trajectory. Refer to Figure 3.

Refer to the next page for the codes used for this subtask.

Code

```
class LQR_PCONTROLLER(LQRSim):
    def __init__(self, s_des=None) -> None:
        super().__init__()
        self.s_des = s_des if s_des is not None else self.r_t

    def step_func(self, t):
        current_state = self.s_hist[:, t].reshape(-1, 1)
        action = self.K_t@(self.s_des[:, t].reshape(-1, 1) - current_state) + \
            self.k_t
        return action
```

Listing 4: Class for the P controller

```
class LQR_OPTIMAL(LQRSim):
    def __init__(self) -> None:
        super().__init__()
        self.V_t = np.zeros_like(self.R_t)
        self.V_t[-1] = self.R_t[-1]
        self.v_t = np.zeros_like(self.r_t)
        self.v_t[:, -1] = self.R_t[-1]@self.r_t[:, -1]
        for t in range(self.T - 2, -1, -1):
            M_t = (self.B_t/(self.H_t + self.B_t.T@self.V_t[t+1]@self.B_t))\
                @ self.B_t.T @ self.V_t[t + 1]@self.A_t
            self.V_t[t] = self.R_t[t+1] + (self.A_t - M_t).T@self.V_t[t + 1]\
                @ self.A_t
            self.v_t[:, t] = (
                self.R_t[t+1]@self.r_t[:, t+1].reshape(-1, 1)
                + (self.A_t - M_t).T
                @ (self.v_t[:, t + 1].reshape(-1, 1) - self.V_t[t + 1]
                    @ self.b_t.reshape(-1, 1))
            ).reshape(-1)

    def step_func(self, t):
        current_state = self.s_hist[:, t].reshape(-1, 1)
        action = -(self.H_t+self.B_t.T@self.V_t[t]@self.B_t)**-1\
            * self.B_t.T\
           @ (self.V_t[t]
                @ (self.A_t@current_state + self.b_t.reshape(-1, 1))
                - self.v_t[:, t].reshape(-1, 1))
        return action
```

Listing 5: Class for the optimal controller