Learning Robots Exercise 2

TECHNISCHE UNIVERSITÄT DARMSTADT

Dominik Marino - 2468378, Victor Jimenez - 2491031, Daniel Piendl - 2586991
December 7, 2020

Contents

1 Optimal Control

1.1 Implementation

- Example of $s_0 \sim \mathcal{N}(0; I)$ and $w_t \sim \mathcal{N}(\mathbf{b}_t; \Sigma_t)$

$$s_0 = \underbrace{\begin{bmatrix} 1.76405 & 0 \\ 0 & 2.24089 \end{bmatrix}}_{\text{we are only using the diagonals}} = \begin{bmatrix} 1.76405 \\ 2.24089 \end{bmatrix}$$

$$w_t = \underbrace{\begin{bmatrix} 5.18676 & 0 \\ 0 & -0.01514 \end{bmatrix}}_{\text{we are only using the diagonals}} = \begin{bmatrix} 5.18676 \\ -0.01514 \end{bmatrix}$$

- Example calculation of control signal a_t with t = 0:

$$\begin{aligned} a_t &= -\mathbf{K}_t \mathbf{s}_t + k_t = \\ a_0 &= -\mathbf{K}_t \mathbf{s}_0 + k_t = \\ &= -\begin{bmatrix} 5 & 0.3 \end{bmatrix} \begin{bmatrix} 1.76405 \\ 2.24089 \end{bmatrix} + 0.3 = -9.1925 \end{aligned}$$

- Example calculation of state s_{t+1} with t = 0:

$$\begin{aligned} \mathbf{s}_{t+1} &= \mathbf{A}_t \mathbf{s}_t + \mathbf{B}_t a_t + \mathbf{w}_t = \\ \mathbf{s}_1 &= \mathbf{A}_t \mathbf{s}_0 + \mathbf{B}_t a_0 + \mathbf{w}_t = \\ &= \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1.76405 \\ 2.24089 \end{bmatrix} + \begin{bmatrix} 0 \\ 0.1 \end{bmatrix} * -9.1925 + \begin{bmatrix} 5.18676 \\ -0.01514 \end{bmatrix} = \begin{bmatrix} 7.17490 \\ 1.30650 \end{bmatrix} \end{aligned}$$

- Example calculation of reward with t = 0:

$$\begin{aligned} reward_t &= -(\mathbf{s}_t - \mathbf{r}_t)^T \mathbf{R}_t (\mathbf{s}_t - \mathbf{r}_t) - a_t^T \mathbf{H}_t a_t = \\ reward_0 &= -(\mathbf{s}_0 - \mathbf{r}_0)^T \mathbf{R}_t (\mathbf{s}_0 - \mathbf{r}_0) - a_0^T \mathbf{H}_t a_0 = \\ &= -(\begin{bmatrix} 1.76405 \\ 2.24089 \end{bmatrix} - \begin{bmatrix} 10 \\ 0 \end{bmatrix})^T \begin{bmatrix} 0.01 & 0 \\ 0 & 0.1 \end{bmatrix} (\begin{bmatrix} 1.76405 \\ 2.24089 \end{bmatrix} - \begin{bmatrix} 10 \\ 0 \end{bmatrix}) - (-9.1925 * 1 * -9.1925) = \\ &= -(\begin{bmatrix} -8.23595 \\ 2.24089 \end{bmatrix})^T \begin{bmatrix} 0.01 & 0 \\ 0 & 0.1 \end{bmatrix} (\begin{bmatrix} -8.23595 \\ 2.24089 \end{bmatrix}) - 84.50205 \\ &= -\begin{bmatrix} -8.23595 & 2.24089 \end{bmatrix} \begin{bmatrix} 0.01 & 0 \\ 0 & 0.1 \end{bmatrix} \begin{bmatrix} -8.23595 \\ 2.24089 \end{bmatrix} - 84.50205 = \\ &= \begin{bmatrix} 0.08236 & -0.22489 \end{bmatrix} \begin{bmatrix} -8.23595 \\ 2.24089 \end{bmatrix} - 84.50205 \\ &= -1.182266 - 84.50205 = -85.68431 \end{aligned}$$

  - We have to calculate every reward to T = 50 and sum it up

- 68-95-99.7 rule

$$\begin{aligned} Pr(\mu - 1\sigma \leq \mu + 1\sigma) &\approx 0.6827 \\ Pr(\mu - 2\sigma \leq \mu + 2\sigma) &\approx 0.9545 \\ Pr(\mu - 3\sigma \leq \mu + 3\sigma) &\approx 0.9973 \end{aligned}$$

The system's dynamic using the given controller is unstable. The eigenvalues of the closed-loop system, calculated with $\det(\lambda \mathbf{I} - \mathbf{A} + \mathbf{BK}) = 0$, lie around $\lambda_{1,2} = 0.985 \pm 0.2231\, i$. Figure 1 and 2 show the unstable oscillating behavior of the system and 3 the control input / action.

Task 2.1a Reward Mean: -2757057.09, Reward Std: 540289.02

```python
experiments_1 = np.zeros((n, T, 2))
actions_1 = np.zeros((n, T, 1))
for j in range(n):
    s0 = np.random.default_rng().standard_normal((2, 1))
    s = np.zeros((T, 2, 1))
    s[0] = s0
    for i in range(T - 1):
        wt = np.random.default_rng().normal(bt.flatten(), np.diag(Sigma)).reshape((2, 1))
        at = -Kt @ s[i] + kt
        ds = At @ s[i] + Bt @ at + wt
        s[i + 1] = ds

        if i < 15:
            rt = rt1
        else:
            rt = rt2

        if i == 14 or i == 40:
            Rt = Rt1

        else:
            Rt = Rt2
        actions_1[j, i] = at
        rw = reward(s[i], rt, Rt, at, Ht)
        rewards[j, i] = rw.flatten()

    rewards[j, -1] = reward(s[-1], rt2, Rt2)
    experiments_1[j] = s.reshape((-1, 2))

mean_r = np.mean(np.sum(rewards, axis=1))
std_r = np.std(np.sum(rewards, axis=1))
print("CONTROLLER 1\nmean = %d \nstandard deviation = %d\n====================" % (mean_r, std_r))

mean1 = experiments_1.mean(axis=0)
std1 = experiments_1.std(axis=0)

mean_a = actions_1.mean(axis=0)
std_a = actions_1.std(axis=0)
ci = 1.96*std1  # 95% Konfidenzintervall
```

Listing 1: Code Snippet for Task 2.1a

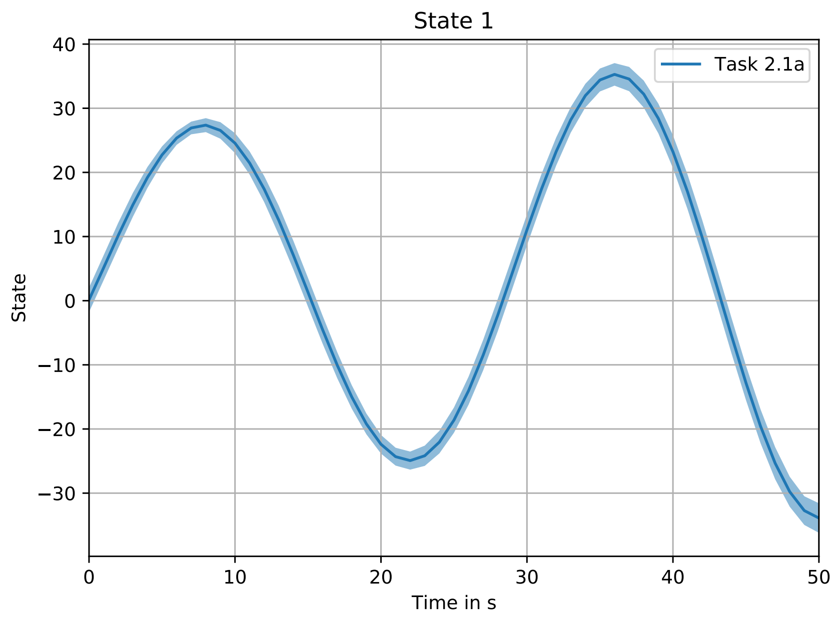

Figure 1: Trend for state 1 in task 2.1a and the confidence interval of 95%

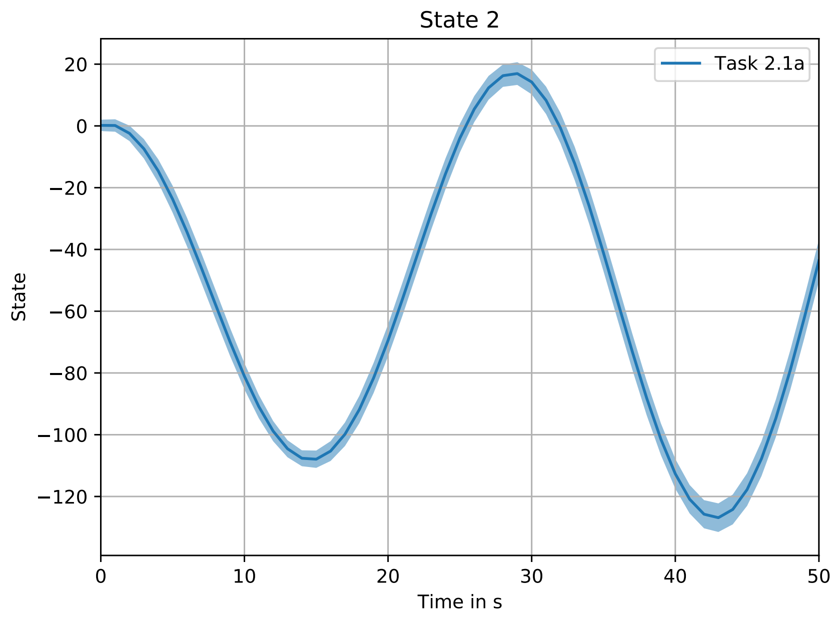

Figure 2: Trend for state 2 in task 2.1a and the confidence interval of 95%

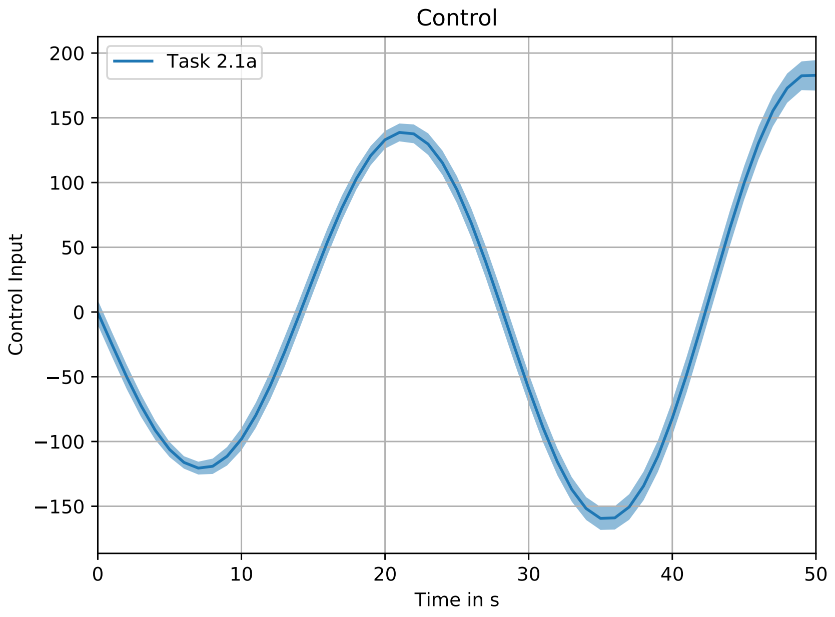

Figure 3: Trend for the action in task 2.1a and the confidence interval of 95%

Figure 4 and 5 show the trends for the states in task 2.1a and task 2.1b with and without a supplied desired state, respectively.

1.2 LQR as a P controller

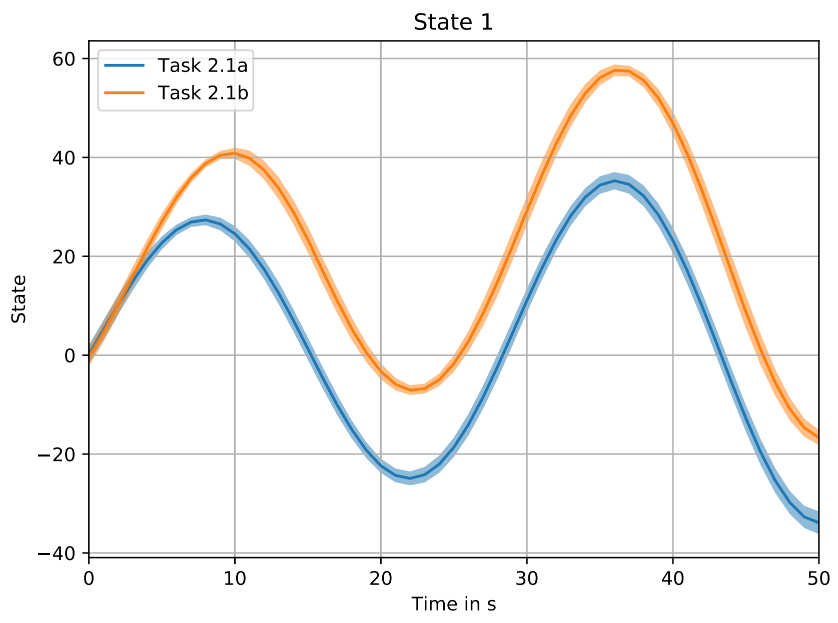

Figure 4: Plot for the state 1 without desired state (task 2.1a) and with a desired state (task 2.1b), both with their respective confidence interval of 95%

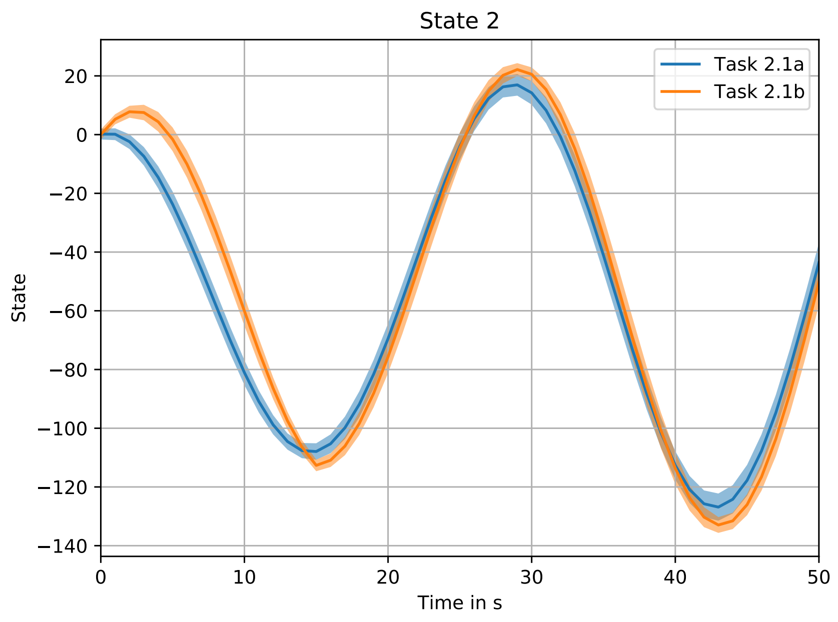

Figure 5: Plot for the state 2 without desired state (task 2.1a) and with a desired state (task 2.1b), both with their respective confidence interval of 95%

1.3 Optimal LQR

The overall behavior is unstable. The desired values for the first state are only reached in the time steps T=14 and T=40 (see figure 6 to figure 8). The matrix R_t has large entries for the first state at these time steps, because of that the controller achieves the desired values for the first state.

Because these two time steps receive by far the largest reward, the controller tries to reach the desired state 1 as exact as possible. This can be seen in the larger variance for the control input shortly before the two time steps and by the smaller variance for state 1 at these time steps. Since only the first state is given a large reward for the two time steps, the desired value for state 2 is not reached at the two time steps.

Apart from the two time steps at T=14 and T=40 the control input is more penalized than a derivation of the actual state from the desired state. This yields more stable control inputs than for the controllers in 2.1a and 2.1b for the simulated time.

The first two controllers have a lower reward for several reasons (refer to table 1). While the third controller only "cares" about the first state in order to maximize the reward, the closed-loop state controllers in the first two tasks try to achieve the desired states for state 1 and state 2 (for the first task the desired state is (0, 0)). Furthermore, because the control parameters are not well chosen, the closed-loop system is unstable and thus a steady-state cannot be achieved. Because the difference between the actual state and the desired states for the time steps 14 and 40 happens to be a lot larger for the controller in task 2.1b than for the controller in task 2.1a, its reward is a lot lower.

Table 1: Mean reward and standard deviation for all tasks

Task	Reward Mean	Reward Std
2.1a	-2757057.09	540289.02
2.1b	-109694418.88	10972447.37
2.1c	-61105.73	5045.69

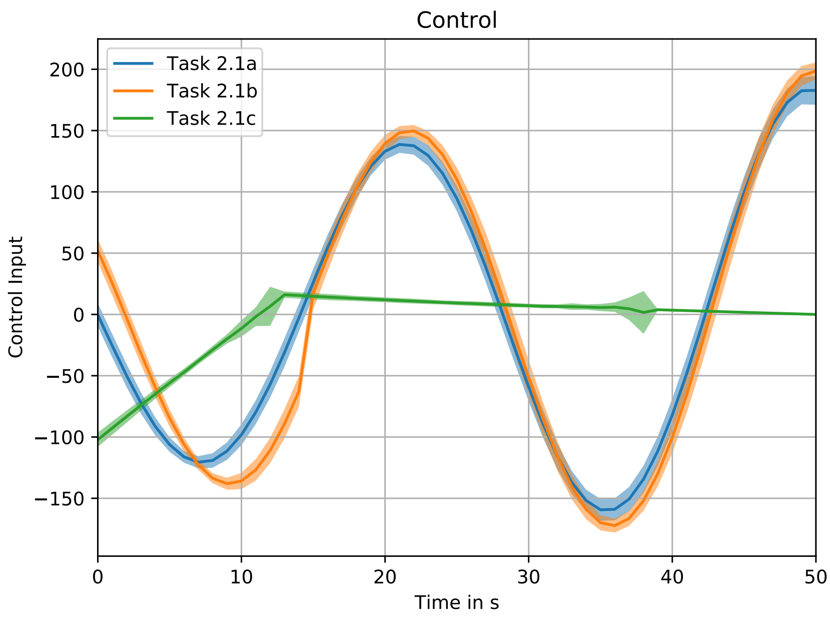

Figure 6: Plot for the actions in task 2.1a to 2.1c and their 95 % confidence interval

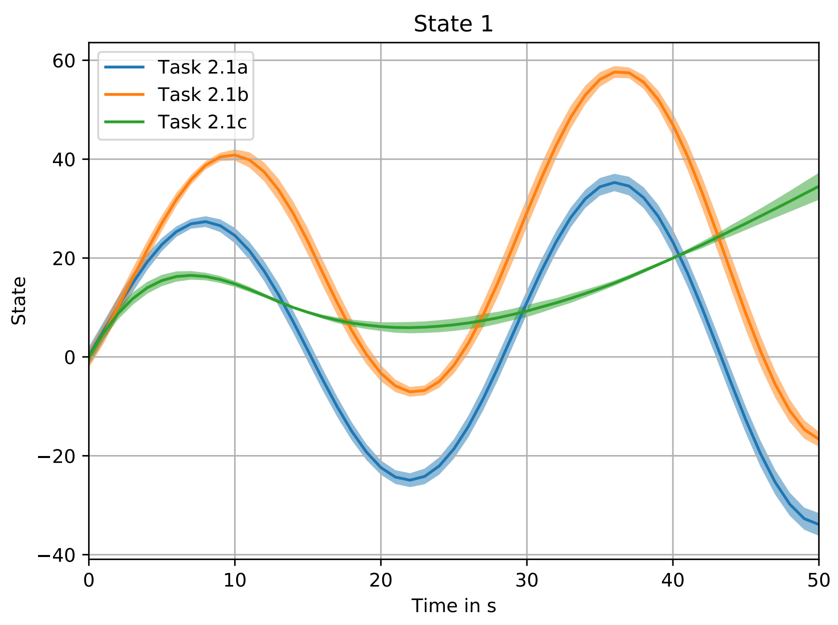

Figure 7: Plot for all states 1 in task 2.1a to 2.1c and their 95 % confidence interval

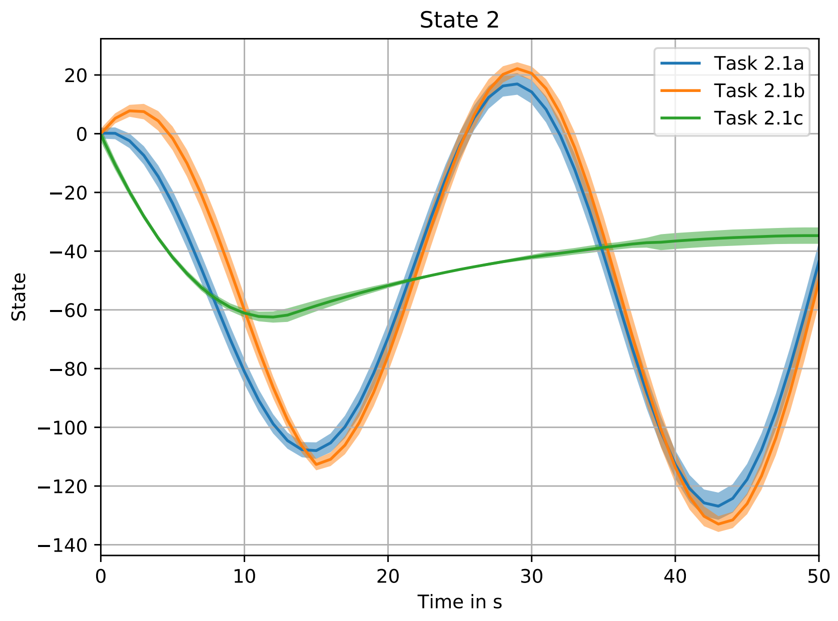

Figure 8: Plot for all states 2 in task 2.1a to 2.1c and their 95 % confidence interval

```python
# computation of the optimal action for all time steps
# begin at the last time step t=T
Vt = np.zeros((T, Rt1.shape[0], Rt1.shape[1]))
Vt[-1] = Rt2
vt = np.zeros((T, rt1.shape[0], 1))
vt[-1] = Rt2@rt2
# calculating for Value function
for i in range(T-2, 0, -1):
    Mt = Bt@np.linalg.inv(Ht + Bt.T@Vt[i+1]@Bt)@Bt.T@Vt[i+1]@At
    if i < 15:
        rt = rt1
    else:
        rt = rt2

    if i == 14 or i == 40:
        Rt = Rt1
    else:
        Rt = Rt2
    Vt[i] = Rt + (At - Mt).T@Vt[i+1]@At
    vt[i] = Rt@rt + (At - Mt).T@(vt[i+1] - Vt[i+1]@bt)

# calculating states
states = list()
actions = np.zeros((n, T, 1))

for j in range(n):
    s0 = np.random.default_rng().standard_normal((2, 1))
    s = np.zeros((T, 2, 1))
    s[0] = s0
    for i in range(T - 1):
        wt = np.random.default_rng().normal(bt.flatten(), np.diag(Sigma)).reshape((2, 1))
        at = -np.linalg.inv(Ht + Bt.T@Vt[i+1]@Bt)@Bt.T@(Vt[i+1]@(At@s[i] + bt) - vt[i+1])
        ds = At @ s[i] + Bt*at + wt.reshape((2, 1))
        s[i + 1] = ds
        actions[j, i] = at
        if i < 15:
            rt = rt1
        else:
            rt = rt2
        if i == 14 or i == 40:
            Rt = Rt1
        else:
            Rt = Rt2
        rw = reward(s[i], rt, Rt, at, Ht)
        rewards[j, i] = rw.flatten()
    rewards[j, -1] = reward(s[-1], rt2, Rt2)
    states.append(s)

mean_r = np.mean(np.sum(rewards, axis=1))
std_r = np.std(np.sum(rewards, axis=1))
print("CONTROLLER 3\nmean = %d \nstandard deviation = %d\n==============" % (mean_r, std_r))

states = np.asarray(states)
s_mean = states.mean(axis=0).reshape(T, 2)
s_std = states.std(axis=0).reshape(T, 2)
a_mean = actions.mean(axis=0).flatten()
a_std = actions.std(axis=0).flatten()

ci_s = 1.96 * s_std  # 95% Konfidenzintervall
ci_a = 1.96 * a_std  # 95% Konfidenzintervall
```

Listing 2: Code Snippet for Task 2.1c