Robot Learning

Winter Semester 2017/2018, Homework 3

Prof. Dr. J. Peters, D. Tanneberg, M. Ewerton

TECHNISCHE UNIVERSITÄT DARMSTADT

Total points: 54 + 8 bonus
Due date: Wednesday, 17 January 2018 (before the lecture)

Name, Surname, ID Number

Problem 3.1 Optimal Control [20 Points]

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

$$\boldsymbol{s}_{t+1} = \boldsymbol{A}_t \boldsymbol{s}_t + \boldsymbol{B}_t a_t + \boldsymbol{w}_t, \tag{1}$$

where $\boldsymbol{s}_t$ is the state, a_t is the control signal, $\boldsymbol{w}_t \sim \mathcal{N}(\boldsymbol{b}_t, \boldsymbol{\Sigma}_t)$ is Gaussian additive noise with mean $\boldsymbol{b}_t$ and covariance $\boldsymbol{\Sigma}_t$ and $t = 0, 1, \ldots, T$ is the time horizon. The control signal a_t is computed as

$$a_t = -\boldsymbol{K}_t \boldsymbol{s}_t + k_t \tag{2}$$

and the reward function is

$$reward_t = \begin{cases} -(\boldsymbol{s}_t - \boldsymbol{r}_t)^\mathsf{T} \boldsymbol{R}_t (\boldsymbol{s}_t - \boldsymbol{r}_t) - a_t^\mathsf{T} \boldsymbol{H}_t a_t & \text{when} \quad t = 0, 1, \ldots, T-1 \\ -(\boldsymbol{s}_t - \boldsymbol{r}_t)^\mathsf{T} \boldsymbol{R}_t (\boldsymbol{s}_t - \boldsymbol{r}_t) & \text{when} \quad t = T \end{cases} \tag{3}$$

Note: the notation used in Marc Toussaint's notes "*(Stochastic) Optimal Control*" is different from the one used in the lecture's slides.

a) **Implementation [8 Points]**

Implement the LQR with the following properties

$$\boldsymbol{s}_0 \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}) \qquad T = 50$$

$$\boldsymbol{A}_t = \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix} \qquad \boldsymbol{B}_t = \begin{bmatrix} 0 \\ 0.1 \end{bmatrix}$$

$$\boldsymbol{b}_t = \begin{bmatrix} 5 \\ 0 \end{bmatrix} \qquad \boldsymbol{\Sigma}_t = \begin{bmatrix} 0.01 & 0 \\ 0 & 0.01 \end{bmatrix}$$

$$\boldsymbol{K}_t = \begin{bmatrix} 5 & 0.3 \end{bmatrix} \qquad k_t = 0.3$$

$$\boldsymbol{H}_t = 1 \qquad \boldsymbol{R}_t = \begin{cases} \begin{bmatrix} 100000 & 0 \\ 0 & 0.1 \end{bmatrix} & \text{if} \quad t = 14 \text{ or } 40 \\ \begin{bmatrix} 0.01 & 0 \\ 0 & 0.1 \end{bmatrix} & \text{otherwise} \end{cases} \qquad \boldsymbol{r}_t = \begin{cases} \begin{bmatrix} 10 \\ 0 \end{bmatrix} & \text{if} \quad t = 0, 1, \ldots, 14 \\ \begin{bmatrix} 20 \\ 0 \end{bmatrix} & \text{if} \quad t = 15, 16, \ldots, T \end{cases}$$

Execute the system 20 times. Plot the mean and 95% confidence (see "68–95–99.7 rule" and matplotlib.pyplot.fill_between function) over the different experiments of the state $\boldsymbol{s}_t$ and of the control signal $\boldsymbol{a}_t$ 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.

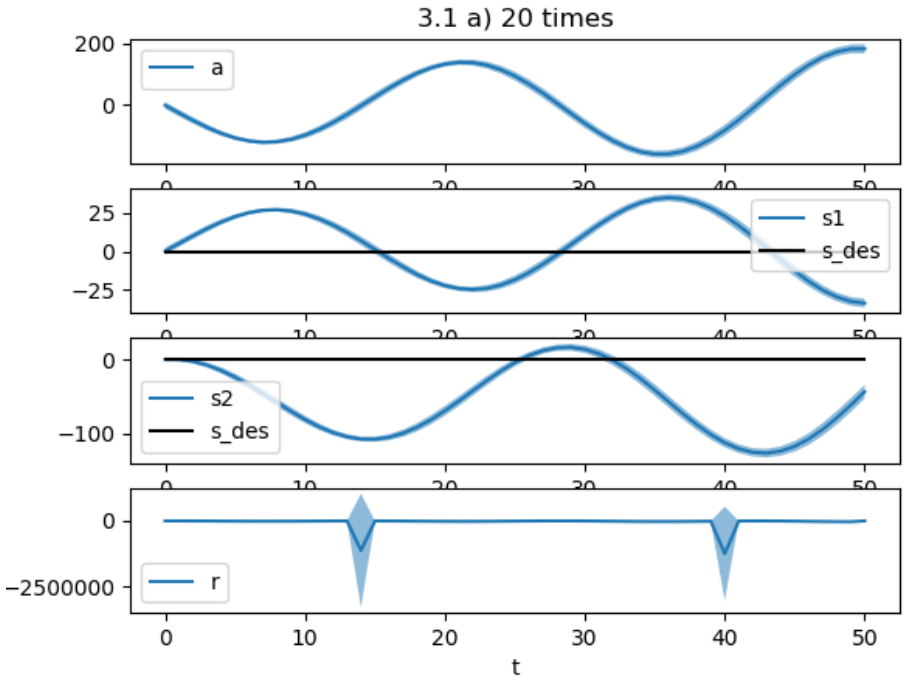


Figure 2: States and action of an instable controller. Mean and 95 % confidence level.

The system is instable cause the eigenvalues of the discrete system (derived from the A-BK matrix) are: $\lambda_i = 0.985 \pm i0.2231i$. *The eigenvalues are outside the unit circle. For this reason the system is instable and oscillates (see state s1, s2).*

The mean of the cummulative reward over the experiments: $\sum rw = -2717991.1$

The standard deviation of the cummulative reward is: 681486

```python
import numpy as np

def next_state(s, a, t):
    # calculate the next state
    s_next = np.dot(A, s) + np.dot(B, a) + np.diag(np.random.normal(b, np.sqrt(cov))).reshape((2, 1))
    return s_next.squeeze()

def control_signal(s, s_des, t, K_t, k_t):
    # calculate the action
    a = np.dot(K_t, s_des - s) + k_t
    return a

def reward(s, a, t):
    rew = 0
    R = get_R(t)
    r = get_r(t)

    if t < T:
        rew = -np.dot(np.dot((s - r).T, R), (s - r)) - np.dot(np.dot(a.T, H), a)
    elif t == T:
        rew = -np.dot(np.dot((s - r).T, R), (s - r))
    else:
        rew = 'error'
    return rew

def run_system(s_des, K_t=0, k_t=k):
    if K_t == 0:
        K_t = K

```

```
    s = np.zeros((2, T + 1))
    a = np.zeros(T + 1)
    r = np.zeros(T + 1)
    s[:, 0] = np.diag(np.random.normal(0, np.eye(2)))
    a[0] = control_signal(s[:, 0].reshape(2, 1), s_des[:, 0].reshape(2, 1), 0, K_t, k_t)
    for t in range(T):
        r[t] = reward(s[:, t].reshape(2, 1), a[t], t)
        s[:, t + 1] = next_state(s[:, t].reshape(2, 1), a[t], t)
        a[t + 1] = control_signal(s[:, t + 1].reshape(2, 1), s_des[:, t + 1].reshape(2, 1), t + 1,
K_t, k_t)

    r[T] = reward(s[:, T].reshape(2, 1), a[T], T)
    return s, a, r
```

Listing 2: Code too calc the states and actions

b) **LQR as a P controller [4 Points]**

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

$$a_t = \boldsymbol{K}_t \left(\boldsymbol{s}_t^{\text{des}} - \boldsymbol{s}_t \right) + k_t \,, \tag{4}$$

which corresponds to the controller used in the canonical LQR system with the introduction of the target $\boldsymbol{s}_t^{\text{des}}$. Assume as target

$$\boldsymbol{s}_t^{\text{des}} = \boldsymbol{r}_t = \begin{cases} \begin{bmatrix} 10 \\ 0 \end{bmatrix} & \text{if} \quad t = 0, 1, \ldots, 14 \\ \begin{bmatrix} 20 \\ 0 \end{bmatrix} & \text{if} \quad t = 15, 16, \ldots, T \end{cases} \tag{5}$$

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 rule” and matplotlib.pyplot.fill_between function) of the first dimension of the state, for both $\boldsymbol{s}_t^{\text{des}} = \boldsymbol{r}_t$ and $\boldsymbol{s}_t^{\text{des}} = \boldsymbol{0}$.

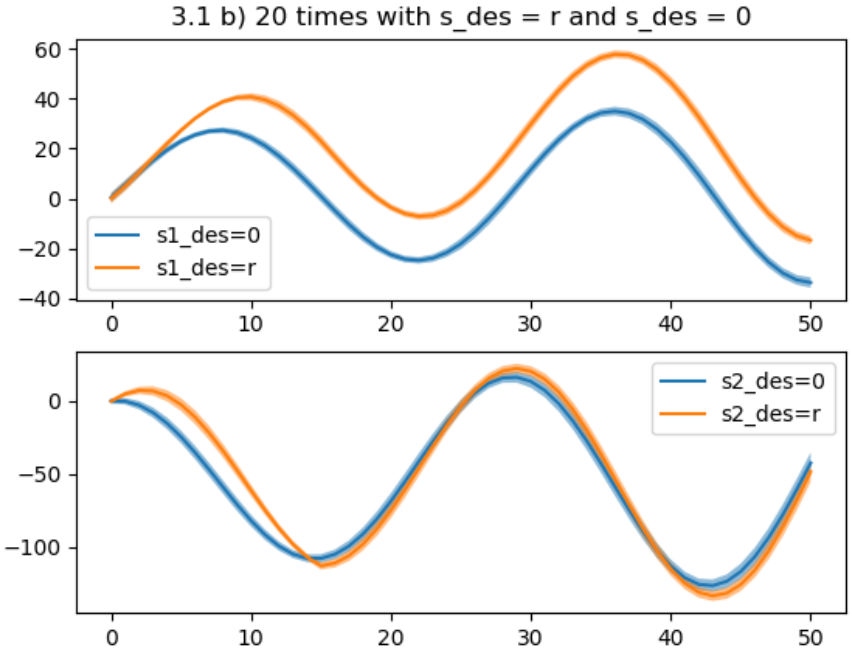


Figure 4: States of an instable controller. Mean and 95 % confidence level. Once with s_des=0 and once with s_des=r.

Since t=40 the state s1_des=r oscillates around 20. Whereas the state s1_des = 0 oscillates a little bit lower.

c) **Optimal LQR [8 Points]**

To compute the optimal gains $\boldsymbol{K}_t$ and $\boldsymbol{k}_t$, which maximize the cumulative reward, we can use an analytic optimal solution. This controller recursively computes the optimal action by

$$a_t^* = -\left(\boldsymbol{H}_t + \boldsymbol{B}_t^T \boldsymbol{V}_{t+1} \boldsymbol{B}_t\right)^{-1} \boldsymbol{B}_t^T \left(\boldsymbol{V}_{t+1} \left(\boldsymbol{A}_t \boldsymbol{s}_t + \boldsymbol{b}_t\right) - v_{t+1}\right), \tag{6}$$

which can be decomposed into

$$\boldsymbol{K}_t = -\left(\boldsymbol{H}_t + \boldsymbol{B}_t^T \boldsymbol{V}_{t+1} \boldsymbol{B}_t\right)^{-1} \boldsymbol{B}_t^T \boldsymbol{V}_{t+1} \boldsymbol{A}_t, \tag{7}$$

$$\boldsymbol{k}_t = -\left(\boldsymbol{H}_t + \boldsymbol{B}_t^T \boldsymbol{V}_{t+1} \boldsymbol{B}_t\right)^{-1} \boldsymbol{B}_t^T \left(\boldsymbol{V}_{t+1} \boldsymbol{b}_t - v_{t+1}\right). \tag{8}$$

where

$$\boldsymbol{M}_t = \boldsymbol{B}_t \left(\boldsymbol{H}_t + \boldsymbol{B}_t^T \boldsymbol{V}_{t+1} \boldsymbol{B}_t\right)^{-1} \boldsymbol{B}_t^T \boldsymbol{V}_{t+1} \boldsymbol{A}_t \tag{9}$$

$$\boldsymbol{V}_t = \begin{cases} \boldsymbol{R}_t + \left(\boldsymbol{A}_t - \boldsymbol{M}_t\right)^T \boldsymbol{V}_{t+1} \boldsymbol{A}_t & \text{when} \quad t = 1 \ldots T-1 \\ \boldsymbol{R}_t & \text{when} \quad t = T \end{cases} \tag{10}$$

$$v_t = \begin{cases} \boldsymbol{R}_t \boldsymbol{r}_t + \left(\boldsymbol{A}_t - \boldsymbol{M}_t\right)^T \left(v_{t+1} - \boldsymbol{V}_{t+1} \boldsymbol{b}_t\right) & \text{when} \quad t = 1 \ldots T-1 \\ \boldsymbol{R}_t \boldsymbol{r}_t & \text{when} \quad t = T \end{cases} \tag{11}$$

Run 20 experiments with $\boldsymbol{s}_t^{\text{des}} = \boldsymbol{0}$ computing the optimal gains $\boldsymbol{K}_t$ and $\boldsymbol{k}_t$. 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.

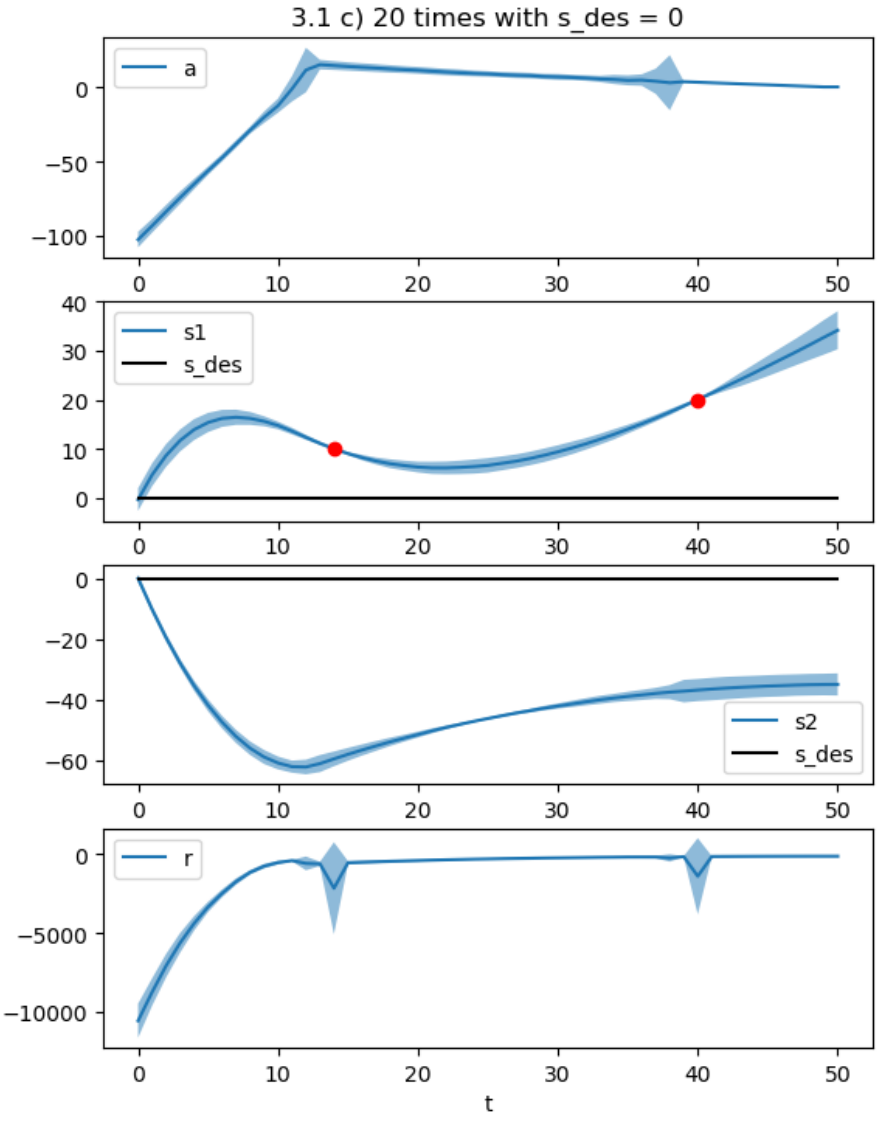


Figure 8: Action, states and reward of the optimal controller.

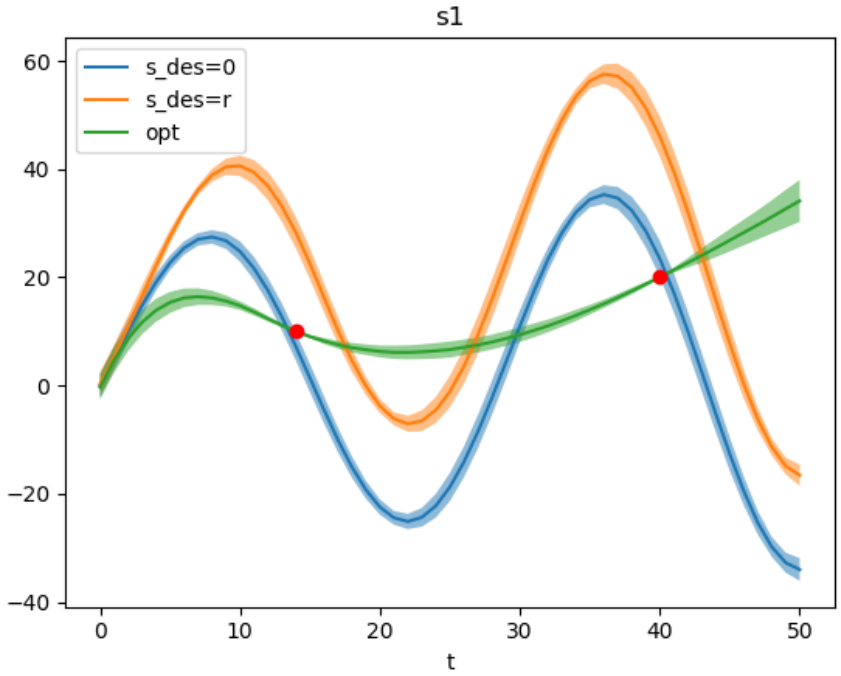


Figure 9: State 1 of all three controllers compared.

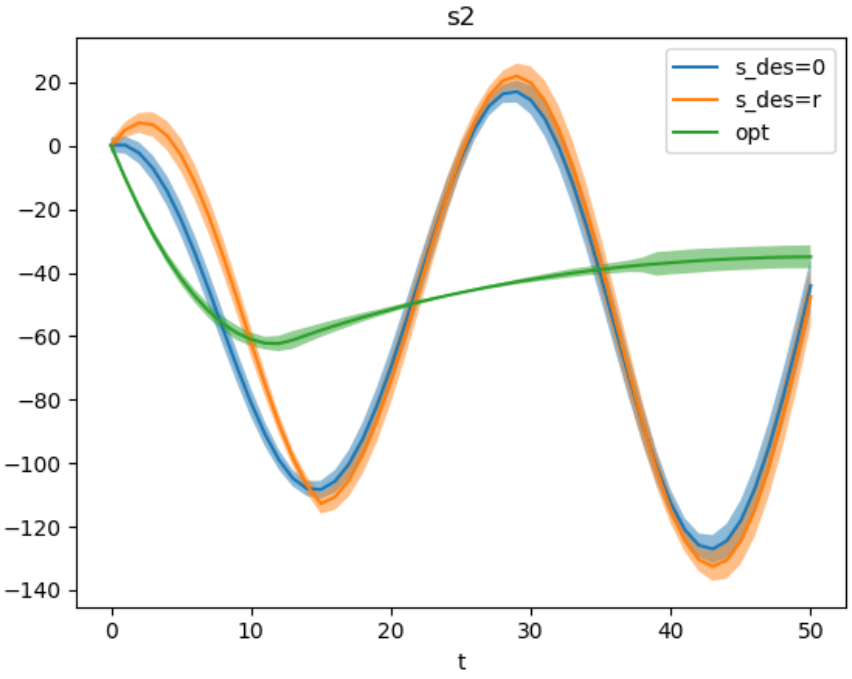


Figure 10: State 2 of all three controllers compared.

cumulative reward	*mean*	*std*
s_des = 0	*-2739815.9*	*424072.8*
s_des = r	*-105631266.4*	*11544950.4*
opt	*-61296.8*	*3966.4*

Table 2: Rewards of the diffeerent controller

The only important part of the reward is the first state at t=14 and t=40. There R equals 100000 and because of this these timesteps have a much higher weight in the cumulative reward. All other timesteps have a R smaller 1 and a really small weight. State 2 has a small R at every timestep, so it is not important for the cumulative reward.

The desired state value is described by r. It equals 10 at t=14 and 20 at t=40. So the trajectory should go through this points (marked with red dots) to minimize the reward.

The optimal controller meats this requirement, like you can see in the plot. The first controller is unstable and oscillates, but its trajectory also is near to the desired points. The second trajectory is also unstable and it is far away from the desired points. Because of this, the second trajectory has a really high negative reward. The first controller has a much smaller negative reward, but it also is very high. The optimal controller has a really small negative reward, compared too the two other controllers.

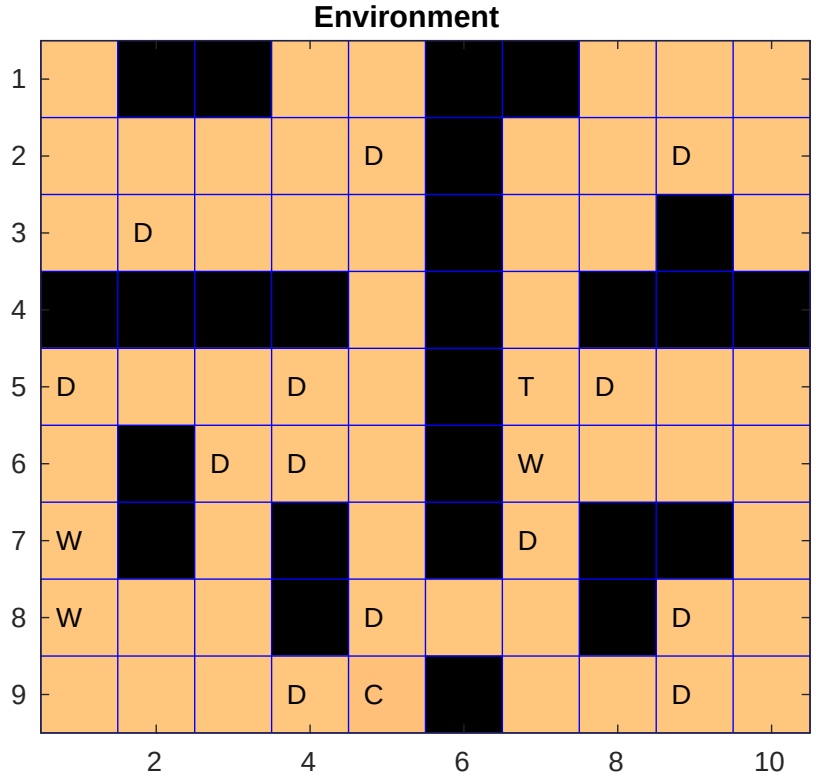
```
class calcOptK:
    def __init__(self):
        self.K_t = nan_matrix((2, T + 1))
        self.k_t = nan_matrix(T + 1)
        self.M_t = nan_matrix((2, 2, T + 1))
        self.V_t = nan_matrix((2, 2, T + 1))
        self.v_t = nan_matrix((2, T + 1))

    def getOptReg(self):
        for i in range(T):
            self.k_t[i] = self.get_k(i)
            self.K_t[:, i] = self.get_K(i)
        return self.K_t, self.k_t

    def get_M(self, t):
        if np.isnan(self.M_t[:, :, t]).all():
            V_next = self.get_V(t + 1)
            temp1 = inv(H + np.dot(np.dot(B.T, V_next), B))
            temp2 = np.dot(np.dot(B.T, V_next), A)
            M = np.dot(np.dot(B, temp1), temp2)
            self.M_t[:, :, t] = M
            return M
        else:
            return self.M_t[:, :, t]

    def get_V(self, t):
        if np.isnan(self.V_t[:, :, t]).all():
            R = self.get_R(t)
            if t < T:
                V_next = self.get_V(t + 1)
                M = self.get_M(t)
                V = R + np.dot(np.dot((A - M).T, V_next), A)
            elif t == T:
                V = R
            else:
                V = 'error'
            self.V_t[:, :, t] = V
            return V
        else:
            return self.V_t[:, :, t]

    def get_v(self, t):
        if np.isnan(self.v_t[:, t]).all():
            R = self.get_R(t)
            r = self.get_r(t)

            if t < T:
                M = self.get_M(t)
                v_next = self.get_v(t + 1)
                V_next = self.get_V(t + 1)
                v = np.dot(R, r) + np.dot((A - M).T, (v_next - np.dot(V_next, b)))
            elif t == T:
                v = np.dot(R, r)
            else:
                v = 'error'
```

```
            self.v_t[:, t] = v.squeeze()
            return v
        else:
            return self.v_t[:, t].reshape(2, 1)

    def get_K(self, t):
        V_next = self.get_V(t + 1)
        temp1 = inv(H + np.dot(np.dot(B.T, V_next), B))
        temp2 = np.dot(np.dot(B.T, V_next), A)
        K = np.dot(temp1, temp2)
        return K

    def get_k(self, t):
        V_next = self.get_V(t + 1)
        v_next = self.get_v(t + 1)
        temp1 = inv(H + np.dot(np.dot(B.T, V_next), B))
        temp2 = np.dot(B.T, (np.dot(V_next, b) - v_next))
        k = -np.dot(temp1, temp2)
        return k.squeeze()

    def get_R(self, t):
        if t == 14 or t == 40:
            R = R1
        else:
            R = R2
        return R

    def get_r(self, t):
        if t <= 14:
            r = r1
        else:
            r = r2
        return r
```

Listing 4: code too calc the optimal controller

Problem 3.2 Reinforcement Learning [34 Points + 8 Bonus]

You recently acquired a robot for cleaning you apartment but you are not happy with its performance and you decide to reprogram it using the latest AI algorithms. As a consequence the robot became self-aware and, whenever you are away, it prefers to play with toys rather than cleaning the apartment. Only the cat has noticed the strange behavior and attacks the robot. The robot is about to start its day and its current perception of the environment is as following



The black squares denote extremely dangerous states that the robot must avoid to protect its valuable sensors. The reward of such states is set to $r_{\text{danger}} = -10^5$ (NB: the robot can still go through these states!). Moreover, despite being waterproof, the robot developed a phobia of water (W), imitating the cat. The reward of states with water is $r_{\text{water}} = -100$. The robot is also afraid of the cat (C) and tries to avoid it at any cost. The reward when encountering the cat is $r_{\text{cat}} = -3000$. The state containing the toy (T) has a reward of $r_{\text{toy}} = 1000$, as the robot enjoys playing with them. Some of the initial specification still remain, therefore the robot receives $r_{\text{dirt}} = 35$ in states with dirt (D).

State rewards can be collected at every time the robot is at that state. The robot can perform the following actions: *down, right, up, left* and *stay*.

In our system we represent the actions with the an ID (0, 1, 2, 3, 4), while the grid is indexed as $\{\texttt{row}, \texttt{column}\}$. The robot can't leave the grid as it is surrounded with walls. A skeleton of the gridworld code and some plotting functions are available at the webpage. For all the following questions, always attach a snippet of your code.

a) **Finite Horizon Problem [14 Points]**

In the first exercise we consider the finite horizon problem, with horizon $T = 15$ steps. The goal of the robot is to maximize the expected return

$$J_\pi = \mathbb{E}_\pi \left[\sum_{t=1}^{T-1} r_t(s_t, a_t) + r_T(s_T) \right], \tag{12}$$

according to policy π, state s, action a, reward r, and horizon T. Since rewards in our case are independent of the action and the actions are deterministic, Equation (12) becomes

$$J_\pi = \sum_{t=1}^{T} r_t(s_t). \tag{13}$$

Using the Value Iteration algorithm, determine the optimal action for each state when the robot has 15 steps left. Attach the plot of the policy to your answer and a mesh plot for the value function. Describe and comment the policy: is the robot avoiding the cat and the water? Is it collecting dirt and playing with the toy? With what time horizon would the robot act differently in state $(9, 4)$?

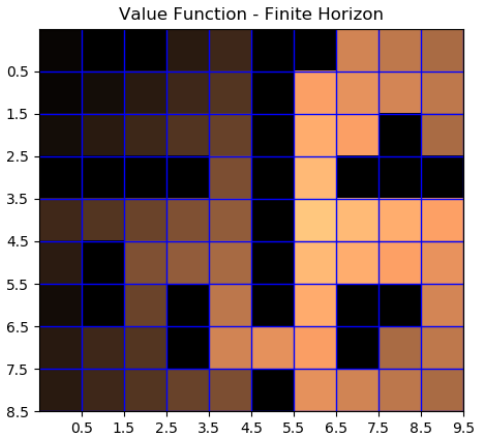


Figure 14: T=15, finite horizon Value Function

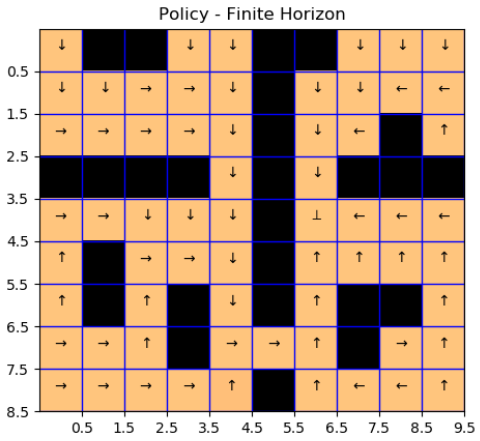


Figure 15: T=15, finite horizon Policy

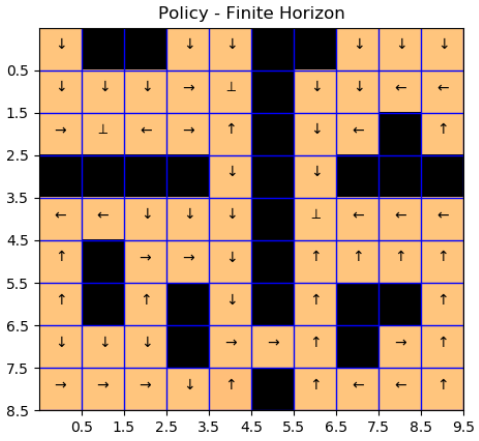


Figure 16: T=10, finite horizon Policy

Policy of the robot:
The robot is trying to avoid the water and the cat(for T=15) if possible (see for example state(8,1) the robot is not moving to state 7,1 as this state is punished cause of its water). The robot is moving from all start states to the max. reward position (the toy at 5,7) avoiding all dangerous states. Thereby the robots is not so afraid of small punishments by the cat or water and moves also over these fields in order to reach the toy! Note: At the toy's position the robot earns a reward as long as it stays at this position. For this reason the robot does not move around the cat but moves over the cat's position directly to the toy in order to play with the toy as long as possible. If it is worth it the robot also collects dirt but this is only a minor aim of the robot.

For T=10 the robot is not moving to the toy's position, as it can not play long enough with the toy and earn such a high reward that compensates moving over the cat's position. (see Figure 16).

The Algorithmic Description of Value Iteration slide can be found in the lecture. The equation on this slide is used to code the current problem.

```
def prob_model(row, col, action, use_prob_model=False):
    if not use_prob_model:
        if action == 0:
            # down
            row_next = row + 1
            col_next = col
            if row_next > 8:
                row_next = row
        elif action == 1:
            # right
            row_next = row
            col_next = col + 1
            if col_next > 9:
                col_next = col
        elif action == 2:
            # up
            row_next = row - 1
            col_next = col
            if row_next < 0:
                row_next = row
        elif action == 3:
            # left
            row_next = row
            col_next = col - 1
            if col_next < 0:
                col_next = col
        elif action == 4:
            # prep -> stay.
            row_next = row
            col_next = col
        return row_next, col_next


def ValIter(R, discount=None, maxSteps=None, infHor=False, probModel=False):
    #R = grid_world matrix! state metric for reward
    if infHor==False:
        T = maxSteps
        # down, right, up, left, prep
        actions = np.array([0, 1, 2, 3, 4])
        Q = np.zeros((R.shape[0], R.shape[1], actions.shape[0], T))
        V = np.zeros((R.shape[0], R.shape[1], T))
        # r = np.zeros((R.shape[0], R.shape[1], actions.shape[0], T))
        # r[:,:,T-1] = R
        V[:, :, T - 1] = R
        # iterate over all ts
        # iterate backwards in time!
        for t in range(T - 2, -1, -1):
            # iterate over all state rows
            for row in range(R.shape[0]):
                # iterate over all columns
                for col in range(R.shape[1]):
                    # iterate over all actions
```

```
                    for action in range(actions.shape[0]):
54                      if not probModel:
                            row_next, col_next = prob_model(row, col, action)
56                          Q[row, col, action, t] = R[row, col] + V[row_next, col_next, t + 1]
                        ...
58                  V[row, col, t] = max(Q[row, col, :, t])

60      return V, Q

62  def findPolicy(V, Q, probModel=None):
        pi = np.zeros((Q.shape[0], Q.shape[1], Q.shape[3]))
64      for t in range(Q.shape[3]):
            for row in range(Q.shape[0]):
66              for col in range(Q.shape[1]):
                    pi[row, col, t] = np.argmax(Q[row, col, :,t])
68      return pi
```

Listing 6: code of the value iteration algorithm

b) **Infinite Horizon Problem - Part 1 [4 Points]**

We now consider the infinite horizon problem, where $T = \infty$. Rewrite Equation (12) for the infinite horizon case adding a discount factor γ. Explain briefly why the discount factor is needed.

Theory:
In the infinite time horizon case (when you live forever), the time index is not part of the state. The optimal policy is time-independent: $\pi_t^*(a|s) = \pi^*(a|s)$. *The reward and the transition model can no longer be time-dependent. There is a single stationary value function for all times. There are two different approaches to learning:*

- *Value Iteration: Same as before just choose a large T for which the value function converges (this is what we use)*
- *Policy Iteration: Learn a value function for the current policy. Update this policy and learn a new value function. Repeat.*

Optimal policy for infinite horizon: $\pi^* = argmax_\pi J_\pi, \quad J_\pi = \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)$
In this case here:

$$J_\pi = \sum_{t=1}^{\infty} \gamma^t r_t(s_t). \tag{15}$$

The discount factor $0 <= \gamma < 1$ *trades-off long term vs. immediate reward. Without a discount factor many or all policies have infinite expected reward. For this reason a discount factor is required. Thereby future rewards* $r_t(s,a)$ *are discounted by* γ *per time step.*

c) **Infinite Horizon Problem - Part 2 [6 Points]**

Calculate the optimal actions with the infinite horizon formulation. Use a discount factor of $\gamma = 0.8$ and attach the new policy and value function plots. What can we say about the new policy? Is it different from the finite horizon scenario? Why?

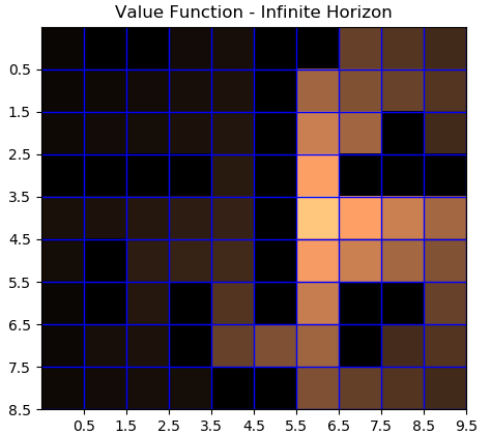


Figure 19: T=100, gamma=0.8 infinite horizon Value Function

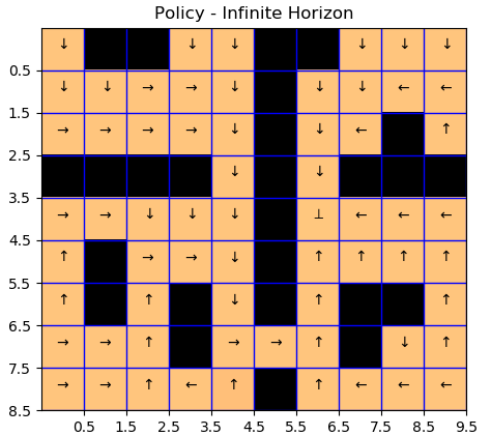


Figure 20: T=100, gamma=0.8, infinite horizon Policy

If the robot lives forever it avoids the cat to reach the toy! This is the only huge difference compared to the finite horizon scenario. The reason for this policy is the discount factor. It weights future rewards smaller than immediate rewards. The smaller the discount factor (e.g 0.1) the higher immediate rewards are weighted. The higher the discount factor the more attractive is the toy to the robot.

Another small difference is that if the robot lives forever it collects a little bit more Dirt on its way to the toy position. That is why the policy of state 8,9 changes to a down arrow to collect the Dirt on 9,9 on its way to the toy position.

```
1
def ValIter(R, discount=None, maxSteps=None, infHor=False, probModel=False):
3    #R = grid_world matrix! state metric for reward
     ...
5    else:
         # infinite horizon:
7        # down, right, up, left, prep
         actions = np.array([0, 1, 2, 3, 4])
9        max_k = maxSteps
         Q = np.zeros((R.shape[0], R.shape[1], actions.shape[0], max_k))
11       V = np.zeros((R.shape[0], R.shape[1], max_k))
         k = 0
13       # for schleife vorwÃ€rts:
         for k in range(max_k-1):
15           # iterate over all state rows
             for row in range(R.shape[0]):
17               # iterate over all columns
                 for col in range(R.shape[1]):
19                   # iterate over all actions
                     for action in range(actions.shape[0]):
21                       row_next, col_next = prob_model(row, col, action)
                         Q[row, col, action, k+1] = R[row, col] + discount*V[row_next, col_next, k]
23                   V[row, col, k+1] = max(Q[row, col, :, k+1])

25           #c heck for convergence
             if (np.abs(V[:,:,k]-V[:,:,k+1])<0.01).all():
27               V = V[:,:,0:k+1]
                 Q = Q[:,:,:,0:k+1]
29               break
```

```
        V = np.flip(V, axis=2)
        Q = np.flip(Q, axis=3)

    return V, Q
```

Listing 8: code of the value iteration algorithm with converge

d) **Finite Horizon Problem with Probabilistic Transition Function [10 Points]**

After a fight with the cat, the robot experiences control problems. For each of the actions *up, left, down, right*, the robot has now a probability 0.7 of correctly performing it and a probability of 0.1 of performing another action according to the following rule: if the action is *left* or *right*, the robot could perform *up* or *down*. If the action is *up* or *down*, the robot could perform *left* or *right*. Additionally, the action can fail causing the robot to remain on the same state with probability 0.1. Using the finite horizon formulation, calculate the optimal policy and the value function. Use a time horizon of $T = 15$ steps as before. Attach your plots and comment them: what is the most common action and why does the learned policy select it?

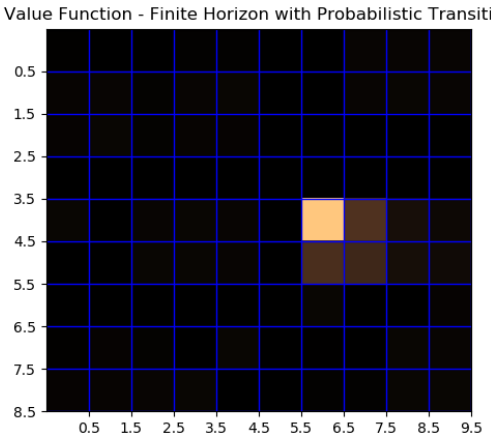


Figure 23: T=15, finite horizon Value Function

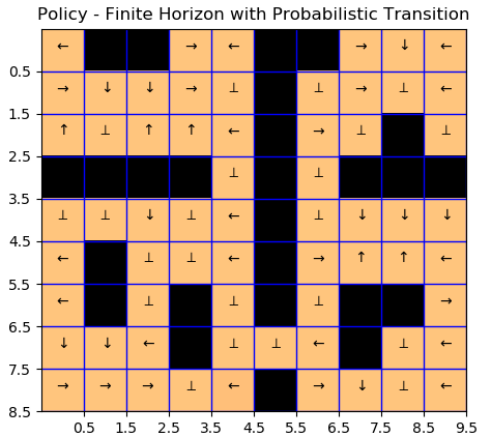


Figure 24: T=15, finite horizon Policy

One of the more common action's is to stay at a reward position not risking a wrong movement to a punished position(like Cat or dangerous field). One example is the state at 9,4. At this position the robot does not risk to move wrong in the right direction (into the cat). The learned policy rather let's the robot wait at a certain position than risking to move wrong into a punished state. One reason for that is the quite high probability of moving wrong (0.1+0.1).

```
def prob_model(row, col, action, use_prob_model=False):
    if not use_prob_model:
        ...
    elif use_prob_model:
        row_next = np.zeros(4)
        col_next = np.zeros(4)
        prob = np.zeros(4)
        if action == 0:
            # down
            row_next[0], col_next[0] = prob_model(row, col, 0)
            prob[0] = np.array([0.7])
            row_next[1], col_next[1] = prob_model(row, col, 1) # right
            prob[1] = 0.1
            row_next[2], col_next[2] = prob_model(row, col, 3) # left
            prob[2] = 0.1
            row_next[3], col_next[3] = prob_model(row, col, 4) # stay
            prob[3] = 0.1
        elif action == 1:
            # right
            row_next[0], col_next[0] = prob_model(row, col, 1)
            prob[0] = np.array([0.7])
            row_next[1], col_next[1] = prob_model(row, col, 0)  # down
            prob[1] = 0.1
            row_next[2], col_next[2] = prob_model(row, col, 2)  # up
            prob[2] = 0.1
            row_next[3], col_next[3] = prob_model(row, col, 4)  # stay
            prob[3] = 0.1
        elif action == 2:
```

```
            # up
            row_next[0], col_next[0] = prob_model(row, col, 2)
            prob[0] = np.array([0.7])
            row_next[1], col_next[1] = prob_model(row, col, 1)  # right
            prob[1] = 0.1
            row_next[2], col_next[2] = prob_model(row, col, 3)  # left
            prob[2] = 0.1
            row_next[3], col_next[3] = prob_model(row, col, 4)  # stay
            prob[3] = 0.1
        elif action == 3:
            # left
            row_next[0], col_next[0] = prob_model(row, col, 3)
            prob[0] = np.array([0.7])
            row_next[1], col_next[1] = prob_model(row, col, 0)  # down
            prob[1] = 0.1
            row_next[2], col_next[2] = prob_model(row, col, 2)  # up
            prob[2] = 0.1
            row_next[3], col_next[3] = prob_model(row, col, 4)  # stay
            prob[3] = 0.1
        elif action == 4:
            # prep
            row_next = np.array([row])
            col_next = np.array([col])
            prob = np.array([1])
        return row_next, col_next, prob

def ValIter(R, discount=None, maxSteps=None, infHor=False, probModel=False):
    #R = grid_world matrix! state metric for reward
    if infHor==False:
        T = maxSteps
        # down, right, up, left, prep
        actions = np.array([0, 1, 2, 3, 4])
        Q = np.zeros((R.shape[0], R.shape[1], actions.shape[0], T))
        V = np.zeros((R.shape[0], R.shape[1], T))
        # r = np.zeros((R.shape[0], R.shape[1], actions.shape[0], T))
        # r[:,:,T-1] = R
        V[:, :, T - 1] = R
        # iterate over all ts
        # iterate backwards in time! for schleife rÃŒckwÃ€rts
        for t in range(T - 2, -1, -1):
            # iterate over all state rows
            for row in range(R.shape[0]):
                # iterate over all columns
                for col in range(R.shape[1]):
                    # iterate over all actions
                    for action in range(actions.shape[0]):
                        if not probModel:
                            ...
                        elif probModel:
                            #Task 2.d) : each next state or action has additionally a probability
    value
                            row_next, col_next, prob = prob_model(row, col, action, use_prob_model=
    True)
                            Q[row, col, action, t] = R[row, col]
                            for i in range(row_next.shape[0]):
                                Q[row, col, action, t] = Q[row, col, action, t]+prob[i] * V[int(
    row_next[i]), int(col_next[i]), t + 1]
                    V[row, col, t] = max(Q[row, col, :, t])
    return V, Q
```

Listing 10: code of the value iteration algorithm with probabilistic model

e) **Reinforcement Learning - Other Approaches [8 Bonus Points]**

What are the two assumptions that let us use the Value Iteration algorithm? What if they would have been not satisfied? Which other algorithm would you have used? Explain it with your own words and write down its fundamental equation.

Some Theory:
Reinforcement Learning:
Contains a bunch of methods of machine learning. An agent(robot) learns a policy (Strategie) in order to increase its reward(Belohnung). At some particular states the agent get's a reward. With the knowledge of these rewards the agent optimizes a cost-function in order to learn the its optimal actions to increase its reward.

TD-Learning:
Temporal Difference Learning is a method of reinforcement learning. Compared to reinforcement learning the agent adapts after each action its policy according to an estimated reward. In reinforcement learning the agent adapts its policy after it collected some rewards in order to maximize its rewards.

Value Iteration Assumptions:
Assume that the agent(robot) accurately knows the transition function(How the robot can move at each state) and the reward(R matrix) for all states in the environment. This knowledge however can be learned with a method called Q-Learning.

Q-Learning:
Q-Learning is a specific method of TD-Learning in reinforcement learning. Q-learning is a form of model-free learning, meaning that an agent does not need to have any model of the environment; it only needs to know what states exist and what actions are possible in each state. The way this works is as follows: we assign each state an estimated value, called a Q-value. When we visit a state and receive a reward, we use this to update our estimate of the value of that state. (Since our rewards might be stochastic, we may need to visit a state many times.)

Update Equations for learning the Q-Functions:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha\left(r_t + \gamma Q_t\left(s_{t+1}, a_?\right) - Q_t(s_t, a_t)\right) \tag{19}$$

With:

$$Q-Learning: \quad a_? = argmax_a Q_t(s_{t+1}, a) \tag{20}$$

Sidenote at SARSA:

$$SARSA: \quad a_? = a_{t+1} \tag{21}$$

See also: https://de.wikipedia.org/wiki/Temporal_Difference_Learning