Robot Learning

TECHNISCHE UNIVERSITÄT DARMSTADT

Winter Semester 2016, Homework 3 (54 points + 8 bonus)
Prof. Dr. J. Peters, F. Veiga, S. Parisi

Due Date: Tuesday, 31 January 2017 (3 pm)

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

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

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

$$\boldsymbol{a}_t = -\boldsymbol{K}_t \boldsymbol{s}_t + \boldsymbol{k}_t \tag{2}$$

and the reward function r_t is

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

Note: r_t and $\boldsymbol{r}_t$ are different!

Note 2: 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

$$\boldsymbol{s}_0 \sim \mathcal{N}(0, 1) \qquad T = 50$$

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

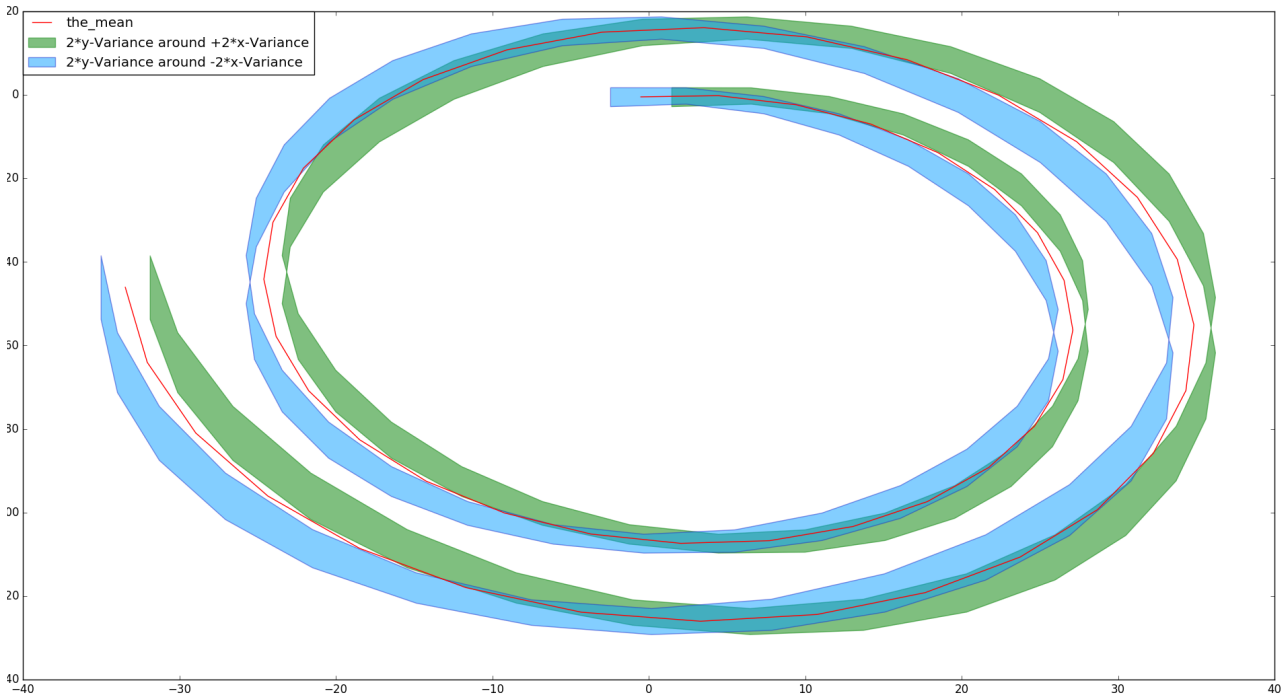
$$\boldsymbol{b}_t = \begin{bmatrix} 5 \\ 0 \end{bmatrix} \qquad \Sigma_t = 0.01$$

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

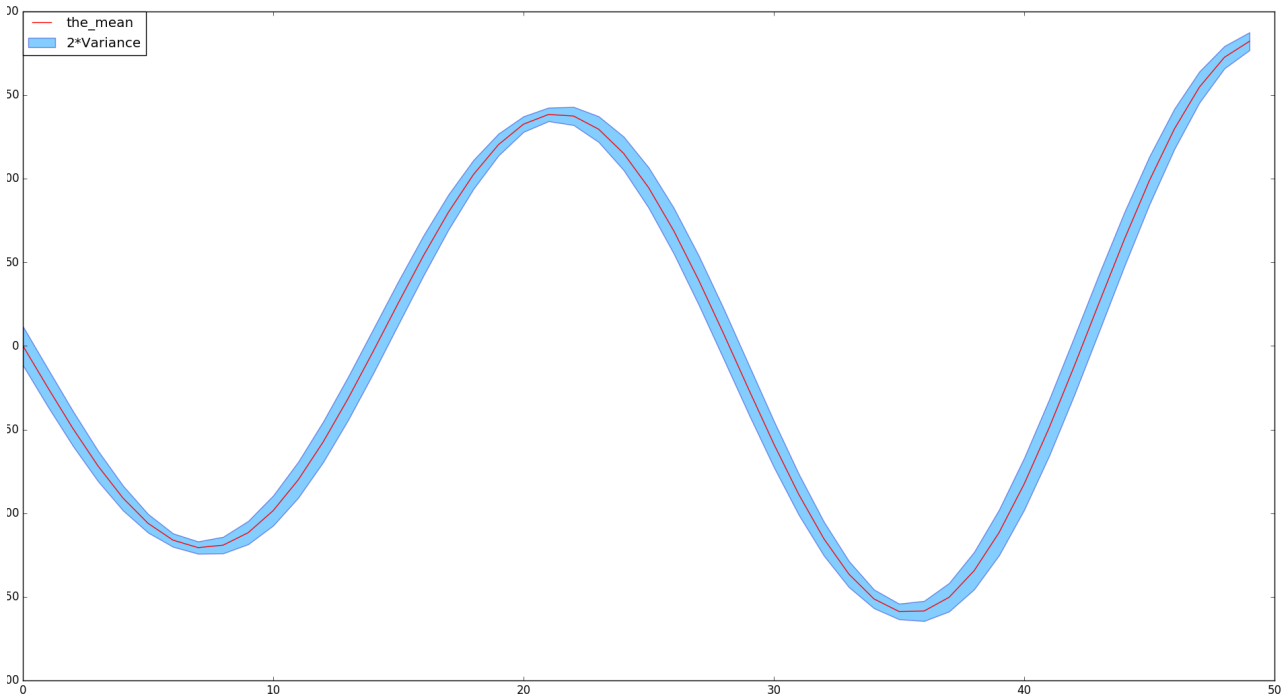
$$\boldsymbol{H}_t = 1 \qquad \boldsymbol{R}_t = \begin{cases} \begin{bmatrix} 100000 & 0 \\ 0 & 0.1 \end{bmatrix} & \text{if} \quad t = 14, 40 \\ \begin{bmatrix} 0.01 & 0 \\ 0 & 0.1 \end{bmatrix} & \text{otherwise} \end{cases} \qquad \boldsymbol{r}_t = \begin{cases} \begin{bmatrix} 10 \\ 0 \end{bmatrix}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

Execute the system 20 times. Plot the mean and 95% confidence 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.

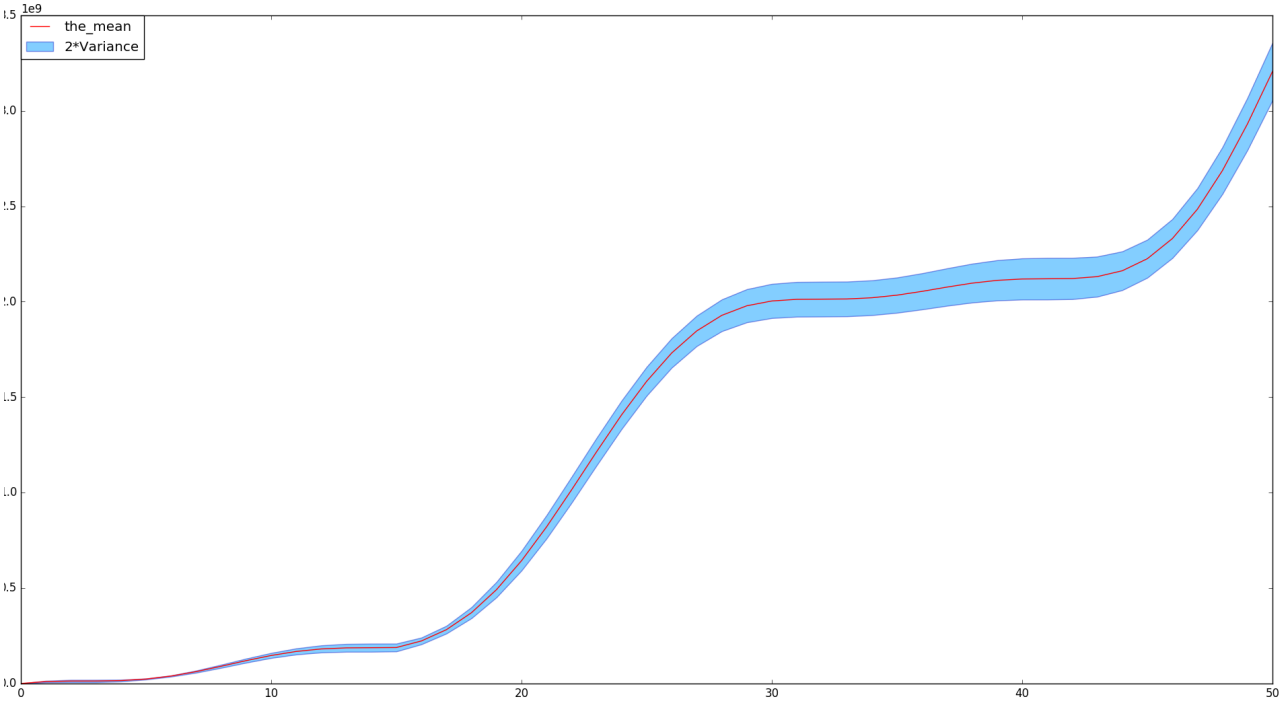
The system is following a spiral trajectory. The actual variance cant be shown with the plt plot function, because its a Tube around the red line with the height of the walls and just touching the walls in the middle.



The Actions are oscillating as aspected for such an spiral trajectory.



negative cumulative reward:



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


def lqr(startstate):
    # LQR parameter
    A_t = np.array([[1, 0.1], [0, 1]])
    B_t = np.array([[0], [0.1]])
    b_t = np.array([[5], [0]])
    Sig_t = 0.01
    K_t = np.array([5, 0.3])
    k_t = 0.3
    H_t = 1
    T = 50

    states = np.zeros((2, T + 1))
    states[:, 0] = startstate
    actions = np.zeros(T)
    rewards = np.zeros(T + 1)
    for i in range(1, T + 1):
        w_t = np.random.normal(b_t, Sig_t)
        actions[i - 1] = -1.0 * np.dot(K_t, states[:, i - 1]) + k_t
        rewards[i] = compute_rt(states[:, i - 1], actions[i - 1], H_t, i - 1, T)
        states[:, i] = np.reshape(np.reshape(np.dot(A_t, states[:, i - 1]), (2,
            1)) + B_t * actions[i - 1] + w_t, 2)

    return actions, states, rewards


def compute_rt(s_t, a_t, H_t, t, T):
    r_t = getr_t(t)
    R_t = getR_t(t)
    diff = np.reshape(s_t, (2, 1)) - r_t
    rslt = -1.0 * np.dot(np.dot(np.transpose(diff), R_t), diff)
    if (t == T):
        return rslt
    else:
        return rslt - np.dot(np.dot(np.transpose(a_t), H_t), a_t)


def getR_t(t):
    if t is 14 or 40:
        return np.array([[100000, 0], [0, 0.1]])
    else:
        return np.array([[0.01, 0], [0, 0.1]])


def getr_t(t):
    if t < 15:
        return np.array([[10], [0]])
    else:
        return np.array([[20], [0]])


Actions = np.zeros((20, 50))
States = np.zeros((20, 51, 2))
dev = np.zeros((51, 2))
Rewards = np.zeros((20, 51))
```

```
for i in range(20):
    s = np.random.normal([0, 0], 1)
    (a, st, r) = lqr(s)
    Actions[i] = a
    States[i] = st.transpose()
    Rewards[i] = r

mean = np.mean(States, axis=0)
for i in range(2):
    for j in range(51):
        for k in range(20):
            dev[j, i] = dev[j, i] + (States[k, j, i] - mean[j, i]) * (States[k,
                j, i] - mean[j, i])
        dev[j, i] = np.sqrt(dev[j, i] / 20)

the_mean = mean.transpose()
plt.plot(the_mean[0], the_mean[1], 'r', label='the_mean')
plt.fill_between(the_mean[0] + 2 * dev[:, 0], the_mean[1] - 2 * dev[:, 1],
    the_mean[1] + 2 * dev[:, 1], alpha=0.5, edgecolor='g', facecolor='g', label=
    '2*y-Variance around +2*x-Variance')
plt.fill_between(the_mean[0] - 2 * dev[:, 0], the_mean[1] - 2 * dev[:, 1],
    the_mean[1] + 2 * dev[:, 1], alpha=0.5, edgecolor='#1B2ACC', facecolor='#089
    FFF', label='2*y-Variance around -2*x-Variance')
plt.legend(bbox_to_anchor=(0, 1), loc=2, borderaxespad=0.)
plt.show()
a_mean = np.mean(Actions, axis=0)
a_dev = np.zeros(50)
for j in range(50):
    for k in range(20):
        a_dev[j] = a_dev[j] + (Actions[k, j] - a_mean[j]) * (Actions[k, j] -
            a_mean[j])
    a_dev[j] = np.sqrt(a_dev[j] / 20)
plt.plot(a_mean, 'r', label='the_mean')
plt.fill_between(range(50), a_mean + 2 * a_dev, a_mean - 2 * a_dev, alpha=0.5,
    edgecolor='#1B2ACC', facecolor='#089FFF', label='2*Variance')
plt.legend(bbox_to_anchor=(0, 1), loc=2, borderaxespad=0.)
plt.show()

Reward = np.zeros((20, 51))
r_dev = np.zeros(51)
Reward[:, 0] = - Rewards[:, 0]
for i in range(1, 51):
    Reward[:, i] = Reward[:, i-1] - Rewards[:, i]

r_mean = np.mean(Reward, axis=0)
for j in range(51):
    for k in range(20):
        r_dev[j] = r_dev[j] + (Reward[k, j] - r_mean[j]) * (Reward[k, j] -
            r_mean[j])
    r_dev[j] = np.sqrt(r_dev[j] / 20)

plt.plot(r_mean, 'r', label='the_mean')
plt.fill_between(range(51), r_mean + 2 * r_dev, r_mean - 2 * r_dev, alpha=0.5,
    edgecolor='#1B2ACC', facecolor='#089FFF', label='2*Variance')
plt.legend(bbox_to_anchor=(0, 1), loc=2, borderaxespad=0.)
plt.show()
```

b) LQR as a P controller [4 Points]

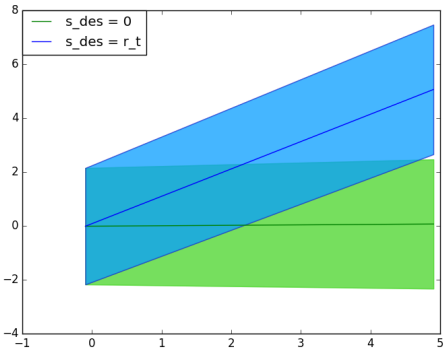
The LQR can also be seen as a simple P controller of the form

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

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 of the first state, for both $\boldsymbol{s}_t^{\text{des}} = \boldsymbol{r}_t$ and $\boldsymbol{s}_t^{\text{des}} = \boldsymbol{0}$.



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

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

which can be decomposed into

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

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

where

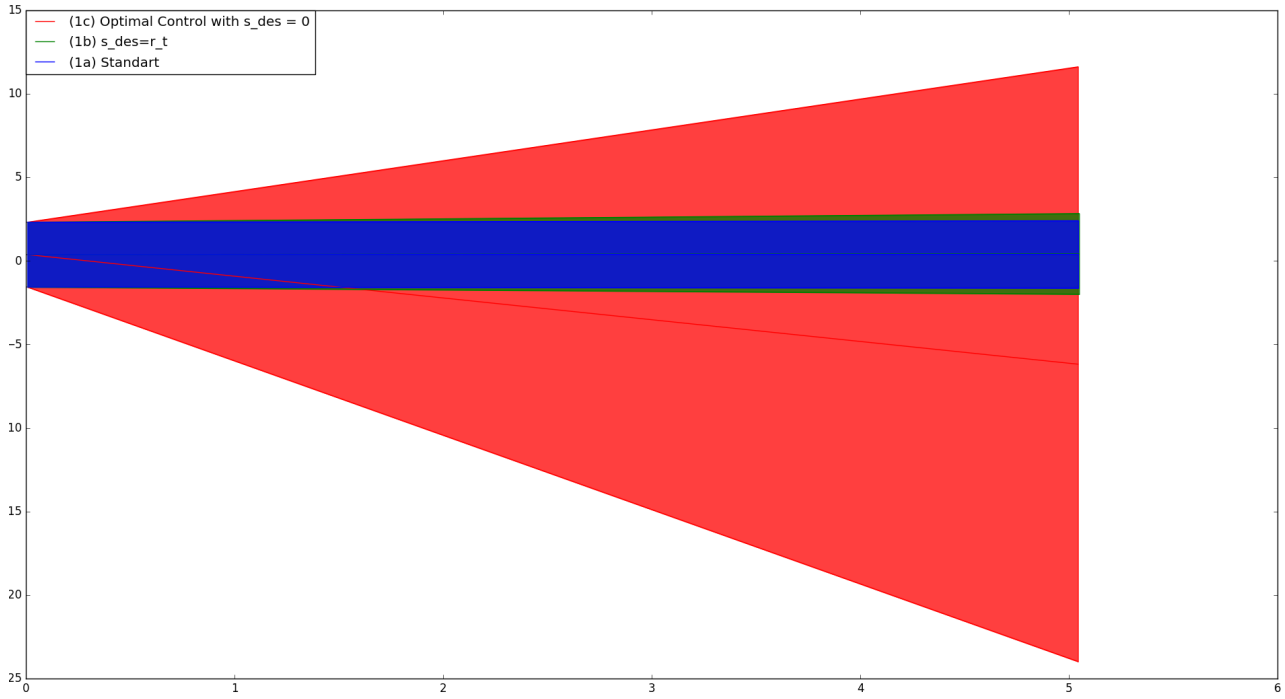
$$\boldsymbol{M}_t = \boldsymbol{B}_t \left(\boldsymbol{H}_t + \boldsymbol{B}_t^T \boldsymbol{V}_{t+1} \boldsymbol{B}_t\right)^{-1} \boldsymbol{B}_t^T \boldsymbol{V}_{t+1} \boldsymbol{A}_t \tag{9}$$

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

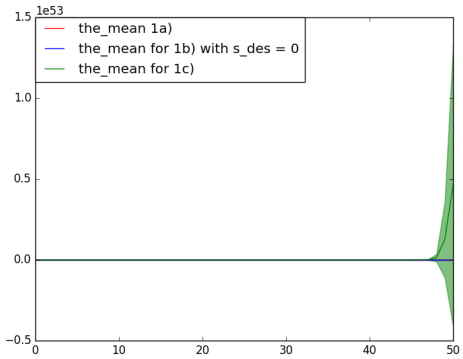
$$\boldsymbol{v}_t = \begin{cases} \boldsymbol{R}_t \boldsymbol{r}_t + \left(\boldsymbol{A}_t - \boldsymbol{M}_t\right)^T \left(\boldsymbol{v}_{t+1} - \boldsymbol{V}_{t+1} \boldsymbol{b}_t\right) & \text{when} \quad t = 1 \ldots T-1 \\ \boldsymbol{R}_t \boldsymbol{r}_t & \text{when} \quad t = T \end{cases}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

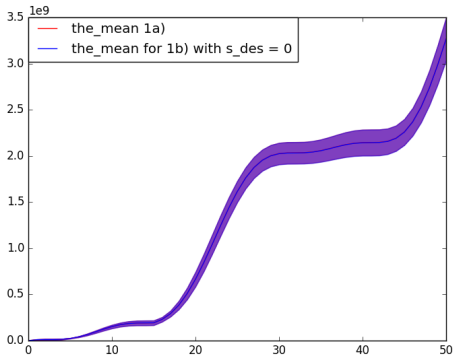
Run 20 experiments with $\boldsymbol{s}_t^{\text{des}} = \boldsymbol{0}$ computing the optimal gains $\boldsymbol{K}_t$ and $\boldsymbol{k}_t$. Plot the mean and 95% confidence of both states for all three different controllers used so far. Use one figure per state. Report the mean and std of the cumulative reward for each controller and comment the results. Attach a snippet of your code.

The contoller of 1c) has a much higher Variance than the others.



As you can see rewards for 1c) are way too big, therefore we plot 1a) and 1b) together again (voilet = red+ blue)
negative cumulative reward:





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


def lqr(startstate):
    # LQR parameter
    A_t = np.array([[1, 0.1], [0, 1]])
    B_t = np.array([[0], [0.1]])
    b_t = np.array([[5], [0]])
    Sig_t = 0.01
    K_t = np.array([5, 0.3])
    k_t = 0.3
    H_t = 1
    T = 50

    states = np.zeros((2, T + 1))
    states[:, 0] = startstate
    actions = np.zeros(T)
    rewards = np.zeros(T + 1)
    for i in range(1, T + 1):
        w_t = np.random.normal(b_t, Sig_t)
        actions[i - 1] = -1.0 * np.dot(K_t, states[:, i - 1]) + k_t
        rewards[i] = compute_rt(states[:, i - 1], actions[i - 1], H_t, i - 1, T)
        states[:, i] = np.reshape(np.reshape(np.dot(A_t, states[:, i - 1]), (2,
            1)) + B_t * actions[i - 1] + w_t, 2)

    return actions, states, rewards


def compute_rt(s_t, a_t, H_t, t, T):
    r_t = getr_t(t)
    R_t = getR_t(t)
    diff = np.reshape(s_t, (2, 1)) - r_t
    rslt = -1.0 * np.dot(np.dot(np.transpose(diff), R_t), diff)
    if (t == T):
        return rslt
    else:
        return rslt - np.dot(np.dot(np.transpose(a_t), H_t), a_t)


def getR_t(t):
    if t is 14 or 40:
        return np.array([[100000, 0], [0, 0.1]])
    else:
        return np.array([[0.01, 0], [0, 0.1]])


def getr_t(t):
    if t < 15:
        return np.array([[10], [0]])
    else:
        return np.array([[20], [0]])


Actions = np.zeros((20, 50))
States = np.zeros((20, 51, 2))
dev = np.zeros((51, 2))
Rewards = np.zeros((20, 51))
```

```
for i in range(20):
    s = np.random.normal([0, 0], 1)
    (a, st, r) = lqr(s)
    Actions[i] = a
    States[i] = st.transpose()
    Rewards[i] = r

mean = np.mean(States, axis=0)
for i in range(2):
    for j in range(51):
        for k in range(20):
            dev[j, i] = dev[j, i] + (States[k, j, i] - mean[j, i]) * (States[k,
                j, i] - mean[j, i])
        dev[j, i] = np.sqrt(dev[j, i] / 20)

the_mean = mean.transpose()
plt.plot(the_mean[0], the_mean[1], 'r', label='the_mean')
plt.fill_between(the_mean[0] + 2 * dev[:, 0], the_mean[1] - 2 * dev[:, 1],
    the_mean[1] + 2 * dev[:, 1], alpha=0.5, edgecolor='g', facecolor='g', label=
    '2*y-Variance around +2*x-Variance')
plt.fill_between(the_mean[0] - 2 * dev[:, 0], the_mean[1] - 2 * dev[:, 1],
    the_mean[1] + 2 * dev[:, 1], alpha=0.5, edgecolor='#1B2ACC', facecolor='#089
    FFF', label='2*y-Variance around -2*x-Variance')
plt.legend(bbox_to_anchor=(0, 1), loc=2, borderaxespad=0.)
plt.show()
a_mean = np.mean(Actions, axis=0)
a_dev = np.zeros(50)
for j in range(50):
    for k in range(20):
        a_dev[j] = a_dev[j] + (Actions[k, j] - a_mean[j]) * (Actions[k, j] -
            a_mean[j])
    a_dev[j] = np.sqrt(a_dev[j] / 20)
plt.plot(a_mean, 'r', label='the_mean')
plt.fill_between(range(50), a_mean + 2 * a_dev, a_mean - 2 * a_dev, alpha=0.5,
    edgecolor='#1B2ACC', facecolor='#089FFF', label='2*Variance')
plt.legend(bbox_to_anchor=(0, 1), loc=2, borderaxespad=0.)
plt.show()

Reward = np.zeros((20, 51))
r_dev = np.zeros(51)
Reward[:, 0] = - Rewards[:, 0]
for i in range(1, 51):
    Reward[:, i] = Reward[:, i-1] - Rewards[:, i]

r_mean = np.mean(Reward, axis=0)
for j in range(51):
    for k in range(20):
        r_dev[j] = r_dev[j] + (Reward[k, j] - r_mean[j]) * (Reward[k, j] -
            r_mean[j])
    r_dev[j] = np.sqrt(r_dev[j] / 20)

plt.plot(r_mean, 'r', label='the_mean')
plt.fill_between(range(51), r_mean + 2 * r_dev, r_mean - 2 * r_dev, alpha=0.5,
    edgecolor='#1B2ACC', facecolor='#089FFF', label='2*Variance')
plt.legend(bbox_to_anchor=(0, 1), loc=2, borderaxespad=0.)
plt.show()
```

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

Your graphic could be here. It just wasn't included.

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 `{row,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]

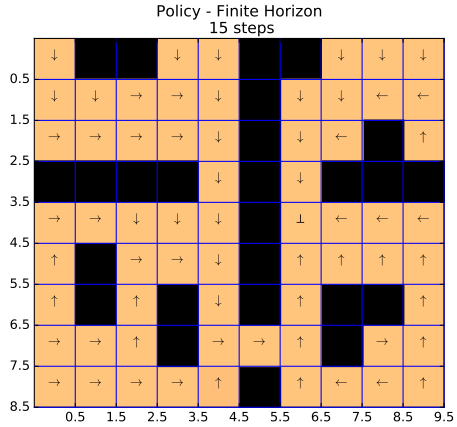
In the first exercise we consider the finite horizon problem, with horizon $T = 15$ steps. The goal of the robot is to maximize the expected return

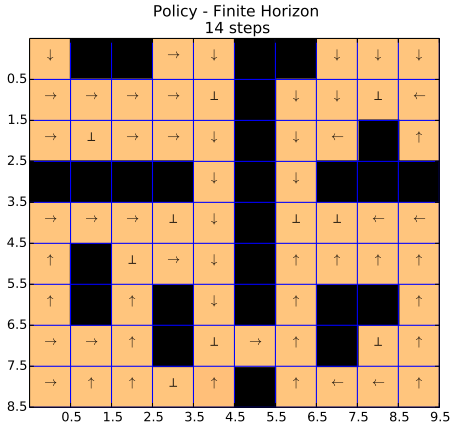
$$J_\pi = \mathbb{E}_\pi \left[\sum_{t=1}^{T-1} r_t(s_t, a_t) + r_T(s_T) \right], \tag{12}$$

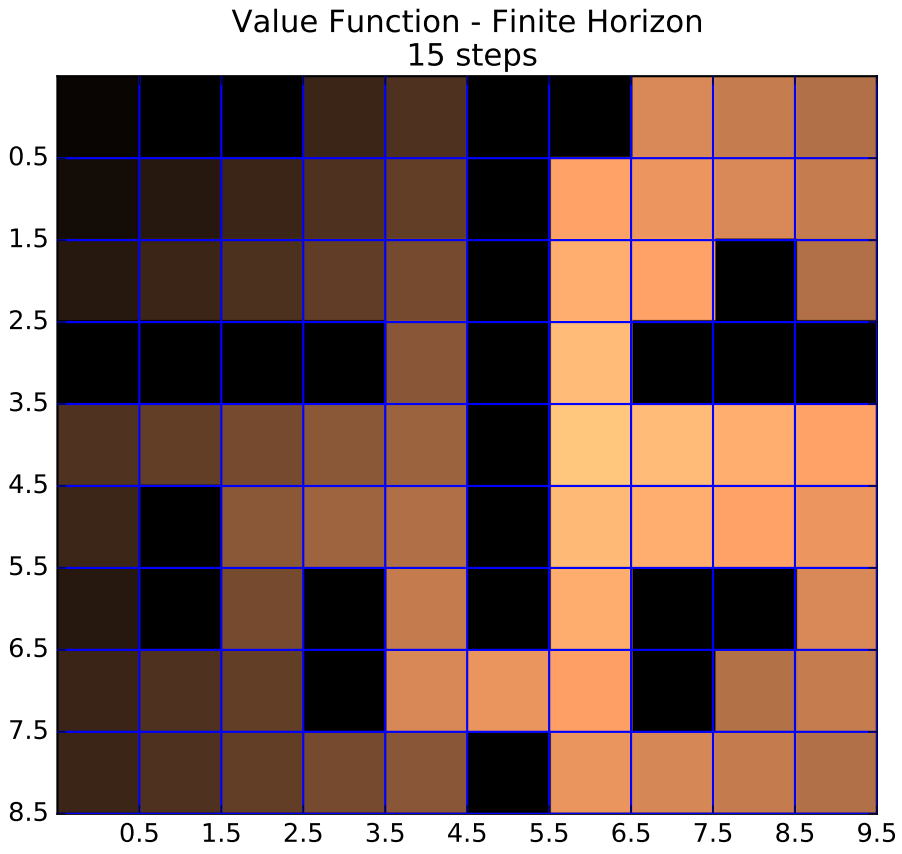
according to policy π, state s, action a, reward r, and horizon T. Since rewards in our case are independent of the action and the actions are deterministic, Equation (12) becomes

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

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? Which would be the time horizon that makes the robot acts differently in state $(9, 4)$?







Our policy seems appropriate given our explanation. The robot does collect dirt if it is far enough away from the toy (5,1) but otherwise will prioritize getting towards the toy. It seems so reckless that it even walks over the cat space to get to it when starting in (9,4). This however changes once the robot is limited to 14 steps forcing our robot to simply collect dirt beside the cat in (9,4).

```
V = ValIter(R, 1, 15, False)
```

...

```
def ValIter(R, discount, maxSteps, infHor, probModel=np.array([])):
        V = np.copy(R)
        if(np.array_equal(probModel,np.array([]))):
            if not infHor:
                for i in range(maxSteps):
                    V = np.copy(maxAction(V, R, discount, probModel))
            else:
```

```
                V_old = np.copy(V)
                V = np.copy(maxAction(V_old, R, discount, probModel))
                while(not np.array_equal(V,V_old) ):
                    V_old = np.copy(V)
                    V = np.copy(maxAction(V_old, R, discount, probModel))
        else:
            for i in range(maxSteps):
                V = np.copy(maxAction(V, R, discount, probModel))
        return V

...

def maxAction(V, R, discount, probModel=np.array([])):
    V_append = np.copy(R)
    if(np.array_equal(probModel, np.array([]))):
        for x in range(R.shape[0]):
            for y in range(R.shape[1]): #for each starting point
                current_max = V[x,y] #staying case
                Act = doAction(R,x,y)   #
                for i in range(4):              # translates the booleans to numbers
                    if Act[i]:                  #
                        Act[i] = 1              #
                    else:                               #
                        Act[i] = 0              #
                if V[x-Act[0],y] > current_max: current_max = V[x-Act[0],y];
                if V[x,y+Act[1]] > current_max: current_max = V[x,y+Act[1]];
                if V[x+Act[2],y] > current_max: current_max = V[x+Act[2],y];
                if V[x,y-Act[3]] > current_max: current_max = V[x,y-Act[3]];

                V_append[x,y] = R[x,y]+ discount * current_max;
    else:
        for x in range(R.shape[0]):
            for y in range(R.shape[1]): #for each starting point
                current_max = V[x,y] #staying case
                Act = doAction(R,x,y)   #
                for i in range(4):              # translates the booleans to numbers
                    if Act[i]:                  #
                        Act[i] = 1              #
                    else:                               #
                        Act[i] = 0              #
                if  probActSum(Act,0,x,y,V)>current_max:
                    current_max=probActSum(Act,0,x,y,V)
                if  probActSum(Act,1,x,y,V)>current_max:
                    current_max=probActSum(Act,1,x,y,V)
                if  probActSum(Act,2,x,y,V)>current_max:
                    current_max=probActSum(Act,2,x,y,V)
                if  probActSum(Act,3,x,y,V)>current_max:

...

                V_append[x,y] = R[x,y]+ discount * current_max;
    return V_append


def findPolicy(V, probModel=np.array([])):
    P = np.copy(V)
    for x in range(P.shape[0]):
        for y in range(P.shape[1]): #for each starting point
            P[x,y] = 4
            current_max = V[x,y]    #stay case
```

```
            Act = doAction(P,x,y)          #
            for i in range(4):             # translates the booleans to numbers
                if Act[i]:                         #
                    Act[i] = 1             #
                else:                              #
                    Act[i] = 0             #
            if V[x+Act[2],y] > current_max: P[x,y] = 0;current_max = V[x+Act[2],y]#Down
            if V[x,y+Act[1]] > current_max: P[x,y] = 1;current_max = V[x,y+Act[1]]#Right
            if V[x-Act[0],y] > current_max: P[x,y] = 2;current_max = V[x-Act[0],y]#Up
            if V[x,y-Act[3]] > current_max: P[x,y] = 3;current_max = V[x,y-Act[3]]#Left
    return P

def doAction(R, x, y):#returns wether action can be performed:
        return [((x-1) >= 0), (y+1) < R.shape[1], (x+1) < R.shape[0], (y-1) >= 0]
```

...

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.

$$J_\pi = \mathbb{E}_\pi\left[\sum_{t=1}^{\infty}\gamma^{t-1} r_t(s_t, a_t)\right], \tag{16}$$

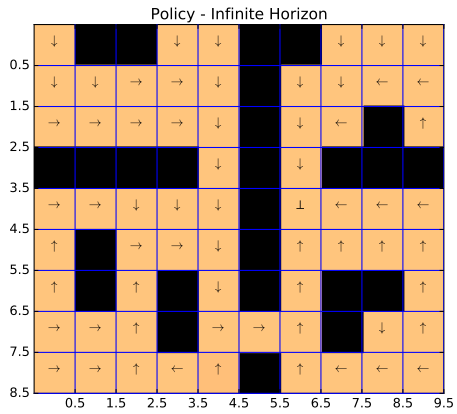
Since like above described rewards in our case are independent of the action and the actions are deterministic, Equation (16) becomes

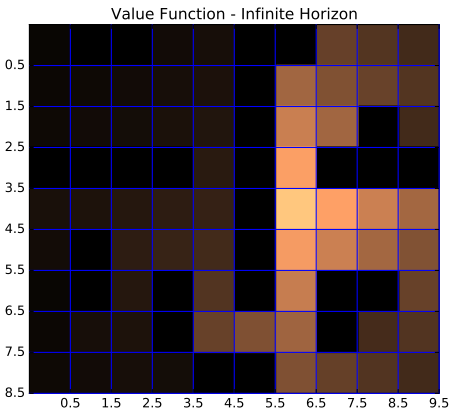
$$J_\pi = \mathbb{E}_\pi\left[\sum_{t=1}^{\infty}\gamma^{t-1} r_t(s_t)\right], \tag{17}$$

The discount factor is needed for convergence. Without the discount factor state values would be summed up until infinity. Therefore the discount factor always has to be smaller than 1, because than later summands of (17) go against 0, since the coefficient of the rewards are γ^t, and $\lim_{t\to\infty}(\gamma^t) = 0$ for $\gamma < 1$ .In addition to that it makes decisions, which are made in the future ,be considered less important than more imminent decisions. The smaller the discount factor is, the faster the algorithm converges.

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?





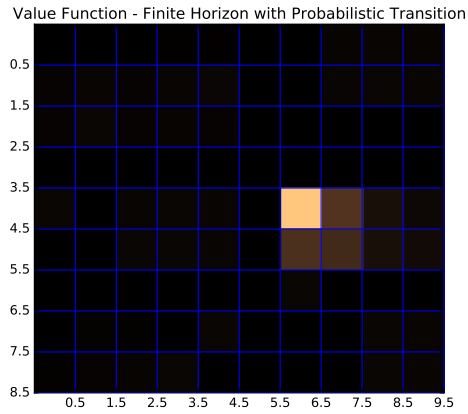
We can say, that it is the optimal policy for the grid world, since iterating the Bellman Equation (Value Iteration) converges to the optimal value function, from which we extracted the optimal policy. It is different, because it decides at (9,4) to rather take the detour instead of passing by the cat. Here we have the optimal policy, the value iteration from a) would need more steps to realize that the route via the cat has a smaller return.

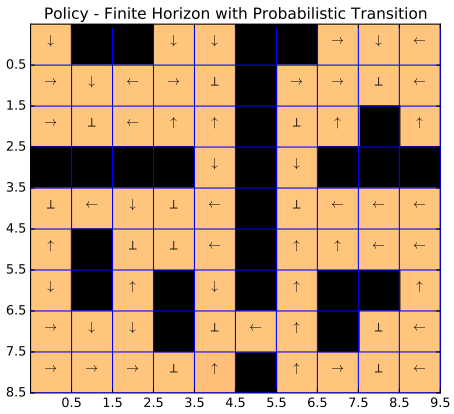
```
V = ValIter(R, 0.8, 15, True)
```

...

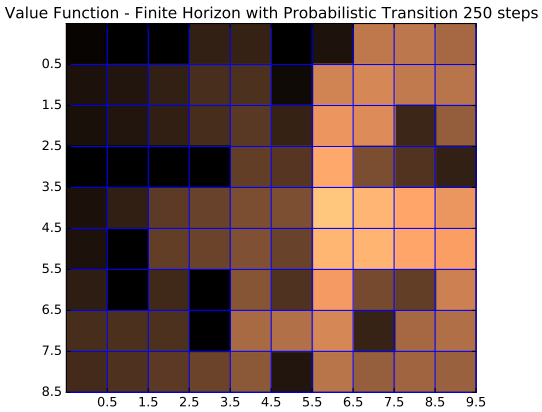
d) Finite Horizon Problem with Probabilistic Transition Function [10 Points]

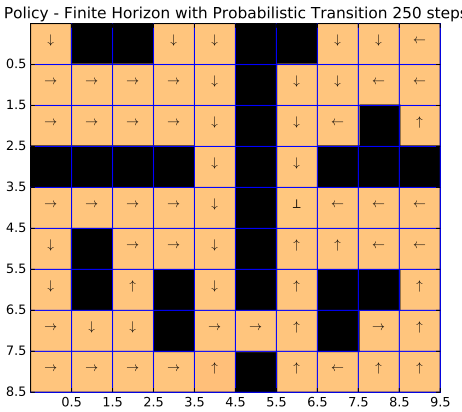
After a fight with the cat, the robot experiences control problems. For each of the actions up, left, down, right, the robot has now a probability 0.7 of correctly performing it and a probability of 0.1 of performing another action according to the following rule: if the action is left or right, the 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?





The fact that the robot can by accident do other actions than expected slows learning tremendous since the negative values of the walls influence other state values. Because of that the most common action the robot chooses is to stay at his position. At all the 'Dirt' postions, the robot rather remains at the Dirt than risking running into a wall.
However, if you run the algorithm for longer time, the results become similar to a). For e.g. T=250 we get:






```
V = ValIter(R,1,15,False,probModel)
```

...

```
def probActSum(Act, action,x,y ,V):
    if action == 0 :
        if Act[0]:
            return (0.7 * V[x-Act[0],y] + 0.1 * Act[1] * V[x,y+Act[1]]+
                    0.1 * Act[3] * V[x,y-Act[3]] + 0.1 * V[x,y])
        else:
            return 0

    if action == 1 :
```

```
        if Act[1]:
            return (0.7 * V[x,y+Act[1]] + 0.1 * Act[0] * V[x-Act[0],y]+
                    0.1 * Act[2] * V[x+Act[2],y] + 0.1 * V[x,y])
        else:
            return 0

    if action == 2 :
        if Act[2]:
            return (0.7 * V[x+Act[2],y] + 0.1 * Act[1] * V[x,y+Act[1]]+
                    0.1 * Act[3] * V[x,y-Act[3]] + 0.1 * V[x,y])
        else:
            return 0

    if action == 3 :
        if Act[3]:
            return (0.7 * V[x,y-Act[3]] + 0.1 * Act[0] * V[x-Act[0],y]+
                    0.1 * Act[2] * V[x+Act[2],y] + 0.1 * V[x,y])
        else:
            return 0
    return 0
```

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.

The Value Iteration algorithm is based on two fundemental assumptions . The first is, that we know completly our environment and therefore can compute the Value function and the associated policy. The second assumption is that its computation doesn't exceed the scale of complexity. That is to say, that we can compute it with the computer.
If these assumptions weren't given, we probably would use the Temporal Difference Learning algorithm, which doesn't compute the value funtion directly, but approximates it. Through the approximation of the value function we could get our policy. Its fundamental equation is given by the following update rule:

$$V_{t+1}(s_t) = V_t(s_t) + \alpha[r_t + \gamma V_t(s_{t+1}) - V_t(s_t)] \tag{19}$$