lqr_inifinite_horizon_solution

```
function [L, P] = lqr_infinite_horizon_solution(Q, R)

%% find the infinite horizon L and P through running LQR back-ups
%%   until norm(L_new - L_current, 2) <= 1e-4
dt = 0.1;
mc = 10; mp = 2.; l = 1.; g= 9.81;

% TODO write A,B matrices
I = eye(4);
a1 = (mp*g)/mc;
a2 = ((mc+mp)*g)/(l*mc);
df_s = [0 0 1 0; 0 0 0 1; 0 a1 0 0; 0 a2 0 0];
df_u = [0; 0; 1/mc; 1/(l*mc)];

A = I + dt*df_s;
B = dt*df_u;

global P_k1, P_k1 = Q;
global L_k1, L_k1 = 0;
L=Riccati_recursion(P_k1,Q,R);
P=P_k1;
% TODO implement Riccati recursion
function[L] = Riccati_recursion(P,Q,R)
        L_k = -inv(R+transpose(B)*P*B)*(transpose(B)*P*A);
        if norm(L_k-L_k1, 2)<=1e-4
        L= L_k;
        else
        P_old = P;
        P_k1 = Q+transpose(L_k)*R*L_k + transpose(A + B*L_k)*P_old*(A+B*L_k);
        L_k1 = L_k;
        L=Riccati_recursion(P_k1,Q,R);
        end
end
end
```

Problem 4: Value Iteration of MCP

```
import numpy as np;
from numpy import linalg as LA
```

```
import seaborn as sns; sns.set()
from random import choices
import matplotlib.pyplot as plt

#givens
n=20
sigma = 10
discount_factor = 0.95
eye = np.array([15,15])
goal = (19,9)
start = (9,19)

#initialize values, storm influence (storm_inf), state space, and reward arrays.
values=np.zeros([n,n])
storm_inf=np.zeros([n,n])
#calculate p(x) for all locations on the grid, store in storm_inf
for i in range(0,n):
    for j in range(0,n):
        storm_inf[i][j] = np.exp(-((LA.norm(np.array([i,j])-eye))**2)/(2*sigma**2))
reward = np.zeros([n,n])
reward[goal[0], goal[1]] = 1
states = []
for i in range(n):
    for j in range(n):
        states.append((i,j))

def next_pos(state,action,direction,n=n):
    """
    Args:
    n is size of board
    state is position on grid (x1,x2)
    action is the called for movement
    direction is the actual direction of movement
    0:up    (x1,x2)->(x1-1, x2  )
    1:down  (x1,x2)->(x1+1, x2  )
    2:left  (x1,x2)->(x1,   x2-1)
    3:right (x1,x2)->(x1,   x2+1)

    given a direction and an action (up, down, left, right)
    return the new state in that direction with the probability of moving in that direction
    up: (x1,x2) -> (x1-1, x2)
    down: (x1,x2) -> (x1+1, x2)
    left: (x1,x2) -> (x1, x2-1)
```

```
    right: (x1,x2) -> (x1, x2+1)

    If the action takes you beyond the boundary of the board, do nothing"""

    x1=state[0]
    x2=state[1]
    x1_n = 0
    x2_n = 0
    prob = 0
    if [x1,x2] == goal:
        x1_n,x2_n=x1,x2
        prob = 0.25
    else:
        if direction == 0:
            x1_n = max(x1 - 1,0)
            x2_n = x2
        if direction == 1:
            x1_n = min(x1 + 1, n-1)
            x2_n = x2
        if direction == 2:
            x1_n = x1
            x2_n = max(x2 - 1,0)
        if direction == 3:
            x1_n = x1
            x2_n = min(x2 + 1,n-1)
        prob = get_prob((x1,x2), action, direction)
    return (x1_n, x2_n), prob

def get_prob(state,a,d,p=storm_inf):
    """
    Args:
    state: position on grid (x1,x2)
    a: direction [0,1,2,3] called for by the action [up, down, left, right] (used to check probability
function)
    d: direction [0,1,2,3] of movement
    p: probability p(x) outlined in problem formulation

    returns:
    prob: probability of ending up in the direction d given a specified action a
    """
    x1=state[0]
    x2=state[1]
    prob=0
```

```
    if d==a:
        prob = p[x1][x2]/4 + (1-p[x1][x2])
    else:
        prob = p[x1][x2]/4
    return prob


def next_state_value(V, a, s, gamma=discount_factor):
    """
    Args:
    V: value function at current step
    a: action
    s: current state

    returns:
    Value at next state
    """
    dir_value=0
    for d in range(4):
        next_state, prob = next_pos(s,a,d)
        new_reward = reward[next_state[0]][next_state[1]]
        new_value = V[next_state[0]][next_state[1]]
        dir_value+=prob*(new_reward+gamma*new_value)
    #dir_value = min(dir_value, 1)
    return dir_value

def update_value_function(v,states=states):
    v_new = np.zeros([v.shape[0],v.shape[1]])
    p_new = np.zeros([v.shape[0],v.shape[1]])
    for state in states:
        if state == (19,9):
            v_new[19][9]=1
            continue
        action_value = np.zeros(4)
        for a in range(4):
            action_value[a]=next_state_value(v,a,state)
        best_value = np.max(action_value)
        best_action=np.argmax(action_value)
        v_new[state[0]][state[1]] = best_value
        p_new[state[0]][state[1]] = best_action
    return v_new, p_new

def value_iter(v, epsilon,gamma):
```

```
    v_current=v
    delta = 1
    while delta>epsilon:
        v_new, p_new=update_value_function(v_current)
        delta=LA.norm(v_new-v_current)
        v_current = v_new
    return v_new, p_new

value,policy=value_iter(values,.01,discount_factor)

heatmap = sns.heatmap(value, cmap="YlGnBu")
fig=heatmap.get_figure()
fig.savefig('heatmap.png')

def compute_trajectory(p, start = start, goal = goal):
    traj = []
    state = start
    while state != goal:
        density = [0,0,0,0]
        new_state = [0,0,0,0]
        action = p[state[0]][state[1]]
        for d in range(4):
            new_state[d], density[d] = next_pos(state, action, d)
        next_state = choices(new_state, density)
        traj.append([state,action,next_state[0]])
        state = next_state[0]
    return traj

x1=np.zeros(len(traj))
x2=np.zeros(len(traj))
for t in range(len(traj)):
    x1[t]=traj[t][0][0]
    x2[t]=traj[t][0][1]


plt.plot(x1,x2)
plt.title('Drone Trajectory')
plt.xticks(np.arange(0,20, step=1.0))
plt.ylim(19,0)
plt.savefig("Drone Trajectory1.png",bbox_inches="tight",dpi=600)
plt.show()
```

Ilqr_solution:

```
function[x_bar,u_bar,l,L] = ilqr_solution(f,linearize_dyn, Q, R, Qf, goal_state, x0, u_bar,
num_steps, dt)

% init l,L
n = size(Q,1);
m = size(R,1);

l = zeros(m,num_steps);
L = zeros(m,n,num_steps);

% init x_bar, u_bar_prev
x_bar = zeros(n,num_steps+1);
x_bar(:,1) = x0;
u_bar_prev = 100*ones(m,num_steps); %arbitrary value that will not result in termination

% termination threshold for iLQR
epsilon = 0.001;

% initial forward pass
for t=1:num_steps
	x_bar(:,t+1) = f(x_bar(:,t),u_bar(:,t),dt);
end
x_bar_prev = x_bar;

while norm(u_bar - u_bar_prev) > epsilon
	% we use a termination condition based on updates to the nominal
	% actions being small, but many termination conditions are possible.

	% ----- backward pass

	% We quadratize the terminal cost C_T around the current nominal trajectory
	% C_T(dx,du) = 1/2 dx' * QT * dx + qf' * dx + const

	% the quadratic term QT=Qf, but you will need to compute qf

	% the constant terms in the cost function are only used to compute the
	% value of the function, we can ignore them if we only care about
	% getting our control
```

```
        % TODO: compute linear terms in cost function
        qf = Qf*(x_bar(:,end)-goal_state);

        % initialize value terms at terminal cost
        P = Qf;
        p = qf;

        for t=num_steps:-1:1
        % linearize dynamics
        [A,B,c] = linearize_dyn(x_bar(:,t),u_bar(:,t),dt);

        % TODO: again, only need to compute linear terms in cost function
        q = Q*(x_bar(:,t) - goal_state);
        r = R * u_bar(:,t);

        [lt,Lt,P,p] = backward_riccati_recursion(P,p,A,B,Q,q,R,r);
        l(:,t) = lt;
        L(:,:,t) = Lt;
        end

        % ----- forward pass
        u_bar_prev = u_bar; % used to check termination condition

        for t=1:num_steps
        u_bar(:,t) = u_bar(:,t)+l(:,t)+L(:,:,t)*(x_bar(:,t)-x_bar_prev(:,t));
        x_bar(:,t+1) = f(x_bar(:,t),u_bar(:,t),dt);
        end

        x_bar_prev = x_bar; % used to compute dx

end
end

function [l,L,P,p] = backward_riccati_recursion(P,p,A,B,Q,q,R,r)
% TODO: write backward riccati recursion step,
% return controller terms l,L and value terms p,P
% refer to lecture 4 slides
L=-inv(R+transpose(B)*P*B)*(transpose(B)*P*A);
l=-inv(R+transpose(B)*P*B)*(r+transpose(p)*B);
P=(Q+transpose(A)*P*A)-transpose(L)*(R+transpose(B)*P*B)*L;
p=(q+transpose(A)*p)-transpose(L)*(R+transpose(B)*P*B)*l;
end
```