Assignment_4_BuelentUendes_FINAL

June 17, 2021

1 Assignment 4: Approximate Dynamic Programming

1.1 by Buelent Uendes

[193]:
```
import numpy as np
import pandas as pd
from matplotlib.pyplot import figure
from time import process_time
import matplotlib.pyplot as plt
from IPython.display import Image
import random
```

1.2 Importing previous code from Assignment 3

[194]:
```
# Reward function

def reward_function(x, a, N):
    if a == 0:
        return -(x/N)**2
    else:
        return -(x/N)**2 - 0.01

#Transition function

def transition_function(N, p, q_high, q_low, x, effort):
    #First get the increment for the increase in que lenght
    I = np.random.binomial(1,p)

    if effort == 1: #If policy at state x tells me to use high effort!
        S = np.random.binomial(1, q_high)
    else:
        S = np.random.binomial(1, q_low)

    return min(N-1, max(x+I-S,0))

# Reward function
```

```python
def reward_policy_function(N, lazy = True):
    output = []
    if lazy == True:
        for i in range(0,N):
            output.append(reward_function(i, 0,N))
    else:
        for i in range(0,N):
            if i < 50:
                output.append(reward_function(i,0,N))
            else:
                output.append(reward_function(i,1,N))

    return np.asarray(output)

# Model the transition probability function

def transition_probability_function(N, p, q_high, q_low, lazy = True):
    #Rows represent the current state
    #Columns represent the next state

    #Initiliaze the matrix

    P = np.zeros((N,N))

    #Lazy policy
    if lazy == True:
        #Same effort for different que length if lazy policy
        q_1 = q_low
        q_2 = q_low
    else:
        #Different effort for different que length if aggressive policy
        q_1 = q_low
        q_2 = q_high

    for i in range(0,N): #Go through columns
        if i < 50:
            q = q_1
        else:
            q = q_2
        for j in range(0,N):#Go through rows
            if i == j == 0:
                P[i,j] = p * q + (1-p) * (1-q) + (1-p) * q
            elif i == j == 99:
                P[i,j] = p * q + p * (1-q) + (1-p) * (1-q)
            elif i == j:
                P[i,j] = p * q + (1-p) * (1-q)
            elif (j - i) == 1:
                P[i,j] = p * (1-q)
            elif (j - i) == -1:
                P[i,j] = (1-p) * q
    return P

def value_function_bellman(N, gamma,p,q_high, q_low, lazy = True):
    if lazy == True:
        dummy = True
    else:
        dummy = False

    A = np.identity(N) - gamma* transition_probability_function(N, p, q_high,␣
 ↪q_low, lazy = dummy)
    A_inv = np.linalg.inv(A)
    r = reward_policy_function(N, lazy = dummy)
    return A_inv @ r
```

2 Additional functions that are needed

[195]:
```
# Model the sample transitions

def sample_transitions(x_init,p, N, K, q_high, q_low, policy):
    output = []
    x = x_init
    for i in range(0,K):
        effort = policy[x]
        x_new = transition_function(N, p, q_high, q_low, x, effort)
        output.append(x_new)
        x = x_new
    return output
```

2.1 Problem description

The task is to evaluate simple policies and find near-optimal policies via temporal-difference learning methods and linear function approximation.

2.2 Linear functions

[196]:
```
def fine_feature_map(x,N):
    return np.asarray([1 if i == x else 0 for i in range(0, N)])
```

[197]:
```
def coarse_feature_map(x,N):
    return np.asarray([1 if x in np.arange(5*(i-1),(5*i-1)+1) else 0 for i in␣
 ↪range(1, int((N/5)+1))])
```

[198]:
```
def piecewise_linear_feature_map(x,N):
    a = np.asarray([1 if x in np.arange(5*(i-1),(5*i-1)+1) else 0 for i in␣
 ↪range(1, int((N/5)+1))])
    b = np.asarray([(1 * (x-5*(i-1))/5) if x in np.arange(5*(i-1),(5*i-1)+1)␣
 ↪else 0 for i in range(1,int((N/5)+1))])
    return np.hstack((a,b))
```

2.3 Policies

[199]:
```
# Lazy
N = 100 #States
#Action space is 0: low effort; 1: high effort

policy_lazy = [0 for i in range(0,N)]
policy_agg = [0 if i <50 else 1 for i in range(0,N)]
```

2.4 Optimal value function policy iteration: Assignment 3

[200]:
```
def optimal_value_policy_iteration(a, N, gamma,P_0, P_1, k, return_policy =␣
 ↪False):
    #Initialize optimal policy as lazy by default
    opt_policy = np.repeat(10,N).tolist()
    V_opt_policy = np.repeat(-10, N).tolist() #Initialize high negative values

    #Iterate through to find the optimal value
    t_start = process_time()
    for iteration in range(0, k):
        for j in range(0,N): #Go through each state
            V_x = V_opt_policy[j] #initialize a very high negative number for␣
 ↪policy chosen at state x
            for action in a: #Go through each action
                if action == 0:
                    P = P_0
                else:
                    P = P_1
                reward = reward_function(j, action,N)
                discounted_future_reward = gamma * P[j] @ V_opt_policy
                V_x_proposed = reward + discounted_future_reward
                if V_x_proposed >= V_x:
                    V_x = V_x_proposed
                    opt_policy[j] = action #Update the optimal action
            V_opt_policy[j] = V_x
    t_stop = process_time()
    t_end = t_stop - t_start
    print('The time for the whole iteration for {} iteration is {}'.
 ↪format(k,t_end))
```

```
    if return_policy == False:
        return V_opt_policy
    else:
        return opt_policy
```

[232]:
```
# Set the parameters for policy iteration

N = 100
gamma = 0.9
p = 0.5
q_high = 0.61
q_low = 0.51
P_0 = transition_probability_function(N,p,q_high,q_low, lazy = True)
P_1 = transition_probability_function(N,p,q_high,q_low, lazy = False)
a = [0,1]

iteration_number = 100

optimal_value_function_assignment_3 = optimal_value_policy_iteration(a, N,␣
 ↪gamma, P_0, P_1, iteration_number)
```

```
The time for the whole iteration for 100 iteration is 0.23441290499977185
```

3 Problem 1: Approximate policy evaluation

The first problem is to study the performance of the same simple policies as in the first problem set, but this time using TD(0) and LSTD with 10^4 , 10^5 , 10^6 , and 10^7 sample transitions, under the three feature maps. Sample the transitions from one single episode starting from initial state x 0 = N (a full queue). Plot the resulting value functions and compare them to the results you have obtained in the first problem set.

Important comment:

Important note: I only run the whole algorithms up to 10^6 given computational constraints. Yet, this decision should not alter the results obtained in this assignment in a qualitative manner!

3.1 Part 1: TD(0) method

The TD(0) algorithm is defined in the following way:

[202]:
```
Image(url= "TD(0)_LFA.png", width=500, height=500)
```

[202]:
```
<IPython.core.display.Image object>
```

[203]:
```
def init_theta(feature = "fine_feature_map"):
    if feature == "fine_feature_map":
        return np.asarray(np.random.uniform(0,0.5, 100))
    if feature == "coarse":
        return np.asarray(np.random.uniform(0,0.5, 20))
    if feature == "pwl":
        return np.asarray(np.random.uniform(0,0.5, 40))
```

[204]:
```
def td_zero_algorithm(gamma, a,b, transitions, N, policy, function, feature =␣
↪"fine_feature_map"):
    #Initialize the theta values
    theta_estimate = init_theta(feature)

    for i in range(0, len(transitions)-1):

        xt_0 = transitions[i]
        xt_1 = transitions[i+1]
        alpha = a/(i + b)
        effort = policy[xt_0] #action chosen at state xt_0!

        delta = reward_function(xt_0, effort, N) + gamma * theta_estimate @␣
↪function(xt_1,N) - theta_estimate @ function(xt_0,N)
        theta_estimate = theta_estimate + alpha * delta * function(xt_0,N)

    return theta_estimate
```

[205]:
```
def approximate_value_function(theta, N, function = "fine_feature_map"):
    if function == "fine_feature_map":
        return np.asarray([theta @  fine_feature_map(i, N) for i in range(0,␣
↪N)])
    if function == "coarse":
        return np.asarray([theta @  coarse_feature_map(i, N) for i in range(0,␣
↪N)])
    if function == "pwl":
        return np.asarray([theta @  piecewise_linear_feature_map(i, N) for i in␣
↪range(0, N)])
```

[210]:
```
#Set the parameters:

N_1 = 10**3
N_2 = 10**4
N_3 = 10**5
N_4 = 10**6

a = b = 10**5
sigma = 10**(-10)

q_low = 0.51
```

```
q_high = 0.6
p = 0.5
N = 100
x_init = 99
gamma = 0.9
```

3.2 Simulate the tranjectories

[212]:
```
agg_n1 = sample_transitions(x_init, p, N, N_1, q_high, q_low, policy_agg)
lazy_n1 = sample_transitions(x_init, p, N, N_1, q_high, q_low, policy_lazy)
agg_n2 = sample_transitions(x_init, p, N, N_2, q_high, q_low, policy_agg)
lazy_n2 = sample_transitions(x_init, p, N, N_2, q_high, q_low, policy_lazy)
agg_n3 = sample_transitions(x_init, p, N, N_3, q_high, q_low, policy_agg)
lazy_n3 = sample_transitions(x_init, p, N, N_3, q_high, q_low, policy_lazy)
agg_n4 = sample_transitions(x_init, p, N, N_4, q_high, q_low, policy_agg)
lazy_n4 = sample_transitions(x_init, p, N, N_4, q_high, q_low, policy_lazy)
```

3.3 Run the simulations

[213]:
```
# Lazy N1

theta_lazy_fine_n1 = td_zero_algorithm(gamma, a, b, lazy_n1, N, policy_lazy,␣
 ↪fine_feature_map, "fine_feature_map")
theta_lazy_coarse_n1 = td_zero_algorithm(gamma, a, b, lazy_n1, N, policy_lazy,␣
 ↪coarse_feature_map, "coarse")
theta_lazy_pwl_n1 = td_zero_algorithm(gamma, a, b, lazy_n1, N, policy_lazy,␣
 ↪piecewise_linear_feature_map, "pwl")

theta_lazy_values_n1 = {0: theta_lazy_fine_n1, 1: theta_lazy_coarse_n1, 2:␣
 ↪theta_lazy_pwl_n1}

# Agg N1
theta_agg_fine_n1 = td_zero_algorithm(gamma, a, b, agg_n1, N, policy_agg,␣
 ↪fine_feature_map, "fine_feature_map")
theta_agg_coarse_n1 = td_zero_algorithm(gamma, a, b, agg_n1, N, policy_agg,␣
 ↪coarse_feature_map, "coarse")
theta_agg_pwl_n1 = td_zero_algorithm(gamma, a, b, agg_n1, N, policy_agg,␣
 ↪piecewise_linear_feature_map, "pwl")

theta_agg_values_n1 = {0: theta_agg_fine_n1, 1: theta_agg_coarse_n1, 2:␣
 ↪theta_agg_pwl_n1}

# Lazy N2:

theta_lazy_fine_n2 = td_zero_algorithm(gamma, a, b, lazy_n2, N, policy_lazy,␣
 ↪fine_feature_map, "fine_feature_map")
```

```
theta_lazy_coarse_n2 = td_zero_algorithm(gamma, a, b, lazy_n2, N,␣
↪policy_lazy,coarse_feature_map, "coarse")
theta_lazy_pwl_n2 = td_zero_algorithm(gamma, a, b, lazy_n2, N, policy_lazy,␣
↪piecewise_linear_feature_map, "pwl")

theta_lazy_values_n2 = {0: theta_lazy_fine_n2, 1: theta_lazy_coarse_n2, 2:␣
↪theta_lazy_pwl_n2}

# Agg N2
theta_agg_fine_n2 = td_zero_algorithm(gamma, a, b, agg_n2, N, policy_agg,␣
↪fine_feature_map, "fine_feature_map")
theta_agg_coarse_n2 = td_zero_algorithm(gamma, a, b, agg_n2, N, policy_agg,␣
↪coarse_feature_map, "coarse")
theta_agg_pwl_n2 = td_zero_algorithm(gamma, a, b, agg_n2, N, policy_agg,␣
↪piecewise_linear_feature_map, "pwl")

theta_agg_values_n2 = {0: theta_agg_fine_n2, 1: theta_agg_coarse_n2, 2:␣
↪theta_agg_pwl_n2}

# Lazy N3:

theta_lazy_fine_n3 = td_zero_algorithm(gamma, a, b, lazy_n3, N, policy_lazy,␣
↪fine_feature_map, "fine_feature_map")
theta_lazy_coarse_n3 = td_zero_algorithm(gamma, a, b, lazy_n3, N,␣
↪policy_lazy,coarse_feature_map, "coarse")
theta_lazy_pwl_n3 = td_zero_algorithm(gamma, a, b, lazy_n3, N, policy_lazy,␣
↪piecewise_linear_feature_map, "pwl")

theta_lazy_values_n3 = {0: theta_lazy_fine_n3, 1: theta_lazy_coarse_n3, 2:␣
↪theta_lazy_pwl_n3}

# Agg N3
theta_agg_fine_n3 = td_zero_algorithm(gamma, a, b, agg_n3, N, policy_agg,␣
↪fine_feature_map, "fine_feature_map")
theta_agg_coarse_n3 = td_zero_algorithm(gamma, a, b, agg_n3, N, policy_agg,␣
↪coarse_feature_map, "coarse")
theta_agg_pwl_n3 = td_zero_algorithm(gamma, a, b, agg_n3, N, policy_agg,␣
↪piecewise_linear_feature_map, "pwl")

theta_agg_values_n3 = {0: theta_agg_fine_n3, 1: theta_agg_coarse_n3, 2:␣
↪theta_agg_pwl_n3}

# Lazy N4:

theta_lazy_fine_n4 = td_zero_algorithm(gamma, a, b, lazy_n4, N, policy_lazy,␣
↪fine_feature_map, "fine_feature_map")
```

```
theta_lazy_coarse_n4 = td_zero_algorithm(gamma, a, b, lazy_n4, N,␣
↪policy_lazy,coarse_feature_map, "coarse")
theta_lazy_pwl_n4 = td_zero_algorithm(gamma, a, b, lazy_n4, N, policy_lazy,␣
↪piecewise_linear_feature_map, "pwl")

theta_lazy_values_n4 = {0: theta_lazy_fine_n4, 1: theta_lazy_coarse_n4, 2:␣
↪theta_lazy_pwl_n4}

# Agg N4
theta_agg_fine_n4 = td_zero_algorithm(gamma, a, b, agg_n4, N, policy_agg,␣
↪fine_feature_map, "fine_feature_map")
theta_agg_coarse_n4 = td_zero_algorithm(gamma, a, b, agg_n4, N, policy_agg,␣
↪coarse_feature_map, "coarse")
theta_agg_pwl_n4 = td_zero_algorithm(gamma, a, b, agg_n4, N, policy_agg,␣
↪piecewise_linear_feature_map, "pwl")

theta_agg_values_n4 = {0: theta_agg_fine_n4, 1: theta_agg_coarse_n4, 2:␣
↪theta_agg_pwl_n4}
```

3.4 Calculate the approximate value functions

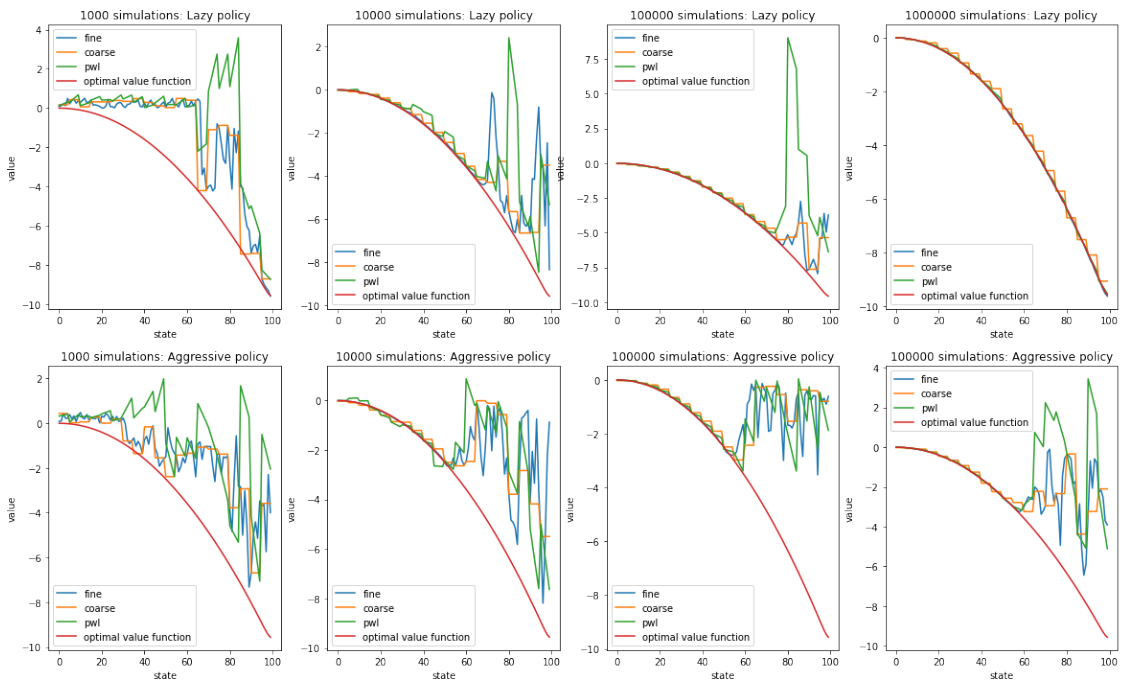
[215]:
```
#Initialize empty lists

value_lazy_n1 = []
value_lazy_n2 = []
value_lazy_n3 = []
value_lazy_n4 = []
value_agg_n1 = []
value_agg_n2 = []
value_agg_n3 = []
value_agg_n4 = []

function = {0: "fine_feature_map", 1: "coarse", 2: "pwl"}

for i in range(0,3):
    value_lazy_n1.append(approximate_value_function(theta_lazy_values_n1[i],␣
↪N,function[i]))
    value_agg_n1.append(approximate_value_function(theta_agg_values_n1[i],␣
↪N,function[i]))

    value_lazy_n2.append(approximate_value_function(theta_lazy_values_n2[i],␣
↪N,function[i]))
    value_agg_n2.append(approximate_value_function(theta_agg_values_n2[i],␣
↪N,function[i]))
```

```
    value_lazy_n3.append(approximate_value_function(theta_lazy_values_n3[i],␣
↪N,function[i]))
    value_agg_n3.append(approximate_value_function(theta_agg_values_n3[i],␣
↪N,function[i]))

    value_lazy_n4.append(approximate_value_function(theta_lazy_values_n4[i],␣
↪N,function[i]))
    value_agg_n4.append(approximate_value_function(theta_agg_values_n4[i],␣
↪N,function[i]))
```

3.5 Plot the results

[233]:
```
#Set the parameter and the style
x = np.arange(0,N)
plt.figure(figsize=(20, 12))

#First subplot
plt.subplot(2,4,1)
plt.plot(x, value_lazy_n1[0], label='fine')
plt.plot(x, value_lazy_n1[1], label='coarse')
plt.plot(x, value_lazy_n1[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
↪function')
plt.title('1000 simulations: Lazy policy')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,2)
plt.plot(x, value_lazy_n2[0], label='fine')
plt.plot(x, value_lazy_n2[1], label='coarse')
plt.plot(x, value_lazy_n2[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
↪function')
plt.title('10000 simulations: Lazy policy')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,3)
plt.plot(x, value_lazy_n3[0], label='fine')
plt.plot(x, value_lazy_n3[1], label='coarse')
plt.plot(x, value_lazy_n3[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
↪function')
```

```
plt.title('100000 simulations: Lazy policy')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,4)
plt.plot(x, value_lazy_n4[0], label='fine')
plt.plot(x, value_lazy_n4[1], label='coarse')
plt.plot(x, value_lazy_n4[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
 ↪function')
plt.title('1000000 simulations: Lazy policy')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,5)
plt.plot(x, value_agg_n1[0], label='fine')
plt.plot(x, value_agg_n1[1], label='coarse')
plt.plot(x, value_agg_n1[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
 ↪function')
plt.title('1000 simulations: Aggressive policy')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,6)
plt.plot(x, value_agg_n2[0], label='fine')
plt.plot(x, value_agg_n2[1], label='coarse')
plt.plot(x, value_agg_n2[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
 ↪function')
plt.title('10000 simulations: Aggressive policy')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,7)
plt.plot(x, value_agg_n3[0], label='fine')
plt.plot(x, value_agg_n3[1], label='coarse')
plt.plot(x, value_agg_n3[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
 ↪function')
plt.title('100000 simulations: Aggressive policy')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,8)
plt.plot(x, value_agg_n4[0], label='fine')
plt.plot(x, value_agg_n4[1], label='coarse')
plt.plot(x, value_agg_n4[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
 ↪function')
plt.title('100000 simulations: Aggressive policy')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.show()
```



Comment:

As we expected, as the number of simulations increases, the value functions as calculated by the TD(0) stabilize more and converge. Yet, as we can see, this happens only for the lazy policy. For the aggressive policy, even though less volatile with increasing number of simulations, the convergence does not emerge even after 1,000,000 simulations.

3.6 Part 2: LSTD(0) Method

Moreover, the LSTD(0) algorithm is described in the following pseudocode:

[13]:
```
Image(url= "LSTD(0).png", width=500, height=500)
```

[13]:
```
<IPython.core.display.Image object>
```

[227]:
```
Image(url= "LSTD(0)_part2.png", width=500, height=500)
```

[227]:
```
<IPython.core.display.Image object>
```

[223]:
```
def lstd_zero_algorithm(gamma, sigma, transitions, N, policy, function, feature␣
 ↪= "fine_feature_map"):
    if feature == "fine_feature_map":
        A = np.zeros((N,N))
        b = np.zeros((1, N))
    elif feature == "coarse":
        A = np.zeros((20,20))
        b = np.zeros((1,20))
    elif feature == "pwl":
        A = np.zeros((40,40))
        b = np.zeros((1,40))
    for i in range(0, len(transitions)-1):
        xt_0 = transitions[i]
        xt_1 = transitions[i+1]
        effort = policy[xt_0] #action chosen at state xt_0!
        A += function(xt_0,N) * np.atleast_2d(function(xt_0,N) - gamma *␣
 ↪function(xt_1, N)).T
        b += reward_function(xt_0, effort,N) * function(xt_0, N)

    A = A/(len(transitions)-1) + sigma * np.identity(A.shape[0])
    b = b/(len(transitions) -1)
    A_inv = np.linalg.inv(A)
    theta = A_inv @ b.T
    return theta.T[0]
```

3.7 Run the simulations

[ ]:
```
policy_lazy = [0 for i in range(0,N)]
policy_agg = [0 if i <50 else 1 for i in range(0,N)]
```

[225]:
```
# Lazy N1

theta_lazy_fine_n1_lstd = lstd_zero_algorithm(gamma, sigma, lazy_n1, N,␣
 ↪policy_lazy, fine_feature_map, "fine_feature_map")
theta_lazy_coarse_n1_lstd = lstd_zero_algorithm(gamma, sigma, lazy_n1, N,␣
 ↪policy_lazy, coarse_feature_map, "coarse")
theta_lazy_pwl_n1_lstd = lstd_zero_algorithm(gamma, sigma, lazy_n1, N,␣
 ↪policy_lazy, piecewise_linear_feature_map, "pwl")
```

```
theta_lazy_values_n1_lstd = {0: theta_lazy_fine_n1_lstd, 1:␣
 ↪theta_lazy_coarse_n1_lstd, 2: theta_lazy_pwl_n1_lstd}

# Agg N1

theta_agg_fine_n1_lstd = lstd_zero_algorithm(gamma, sigma, agg_n1, N,␣
 ↪policy_agg, fine_feature_map, "fine_feature_map")
theta_agg_coarse_n1_lstd = lstd_zero_algorithm(gamma, sigma, agg_n1, N,␣
 ↪policy_agg, coarse_feature_map, "coarse")
theta_agg_pwl_n1_lstd = lstd_zero_algorithm(gamma, sigma, agg_n1, N,␣
 ↪policy_agg, piecewise_linear_feature_map, "pwl")

theta_agg_values_n1_lstd = {0: theta_agg_fine_n1_lstd, 1:␣
 ↪theta_agg_coarse_n1_lstd, 2: theta_agg_pwl_n1_lstd}

# Lazy N2:

theta_lazy_fine_n2_lstd = lstd_zero_algorithm(gamma, sigma, lazy_n2, N,␣
 ↪policy_lazy, fine_feature_map, "fine_feature_map")
theta_lazy_coarse_n2_lstd = lstd_zero_algorithm(gamma, sigma, lazy_n2, N,␣
 ↪policy_lazy, coarse_feature_map, "coarse")
theta_lazy_pwl_n2_lstd = lstd_zero_algorithm(gamma, sigma, lazy_n2, N,␣
 ↪policy_lazy, piecewise_linear_feature_map, "pwl")

theta_lazy_values_n2_lstd = {0: theta_lazy_fine_n2_lstd, 1:␣
 ↪theta_lazy_coarse_n2_lstd, 2: theta_lazy_pwl_n2_lstd}


# Agg N2:
theta_agg_fine_n2_lstd = lstd_zero_algorithm(gamma, sigma, agg_n2, N,␣
 ↪policy_agg, fine_feature_map, "fine_feature_map")
theta_agg_coarse_n2_lstd = lstd_zero_algorithm(gamma, sigma, agg_n2, N,␣
 ↪policy_agg, coarse_feature_map, "coarse")
theta_agg_pwl_n2_lstd = lstd_zero_algorithm(gamma, sigma, agg_n2, N,␣
 ↪policy_agg, piecewise_linear_feature_map, "pwl")

theta_agg_values_n2_lstd = {0: theta_agg_fine_n2_lstd, 1:␣
 ↪theta_agg_coarse_n2_lstd, 2: theta_agg_pwl_n2_lstd}


# Lazy N3

theta_lazy_fine_n3_lstd = lstd_zero_algorithm(gamma, sigma, lazy_n3, N,␣
 ↪policy_lazy, fine_feature_map, "fine_feature_map")
```

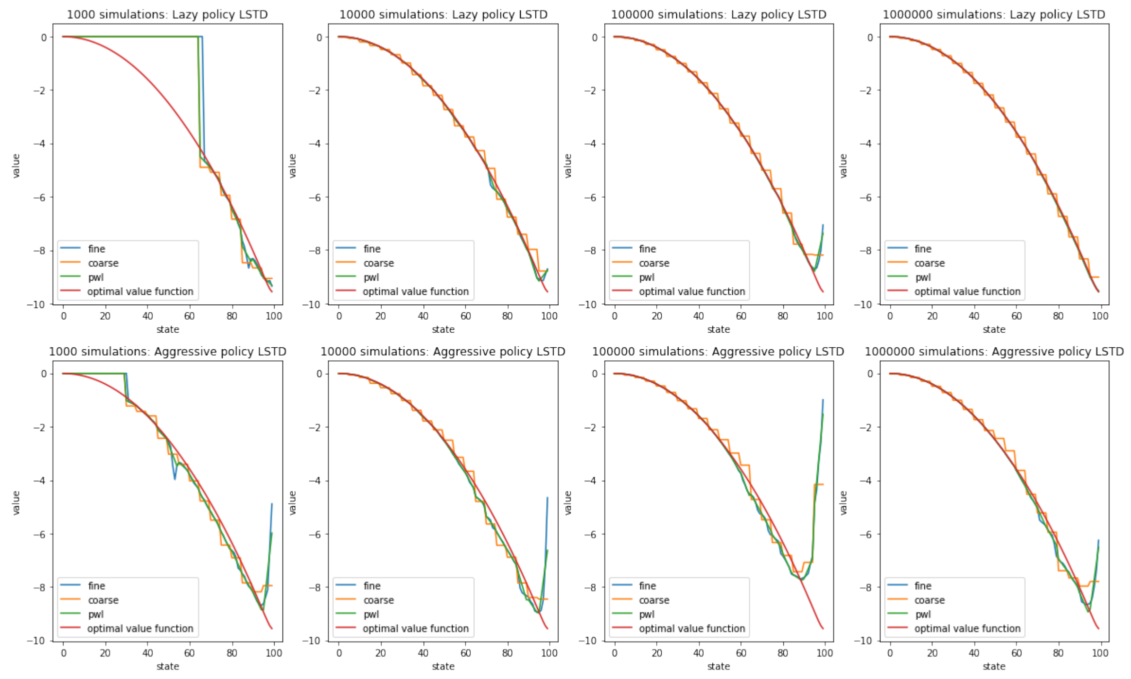
```
theta_lazy_coarse_n3_lstd = lstd_zero_algorithm(gamma, sigma, lazy_n3, N,␣
 ↪policy_lazy, coarse_feature_map, "coarse")
theta_lazy_pwl_n3_lstd = lstd_zero_algorithm(gamma, sigma, lazy_n3, N,␣
 ↪policy_lazy, piecewise_linear_feature_map, "pwl")

theta_lazy_values_n3_lstd = {0: theta_lazy_fine_n3_lstd, 1:␣
 ↪theta_lazy_coarse_n3_lstd, 2: theta_lazy_pwl_n3_lstd}


# Agg N3:
theta_agg_fine_n3_lstd = lstd_zero_algorithm(gamma, sigma, agg_n3, N,␣
 ↪policy_agg, fine_feature_map, "fine_feature_map")
theta_agg_coarse_n3_lstd = lstd_zero_algorithm(gamma, sigma, agg_n3, N,␣
 ↪policy_agg, coarse_feature_map, "coarse")
theta_agg_pwl_n3_lstd = lstd_zero_algorithm(gamma, sigma, agg_n3, N,␣
 ↪policy_agg, piecewise_linear_feature_map, "pwl")

theta_agg_values_n3_lstd = {0: theta_agg_fine_n3_lstd, 1:␣
 ↪theta_agg_coarse_n3_lstd, 2: theta_agg_pwl_n3_lstd}


# Lazy N4

theta_lazy_fine_n4_lstd = lstd_zero_algorithm(gamma, sigma, lazy_n4, N,␣
 ↪policy_lazy, fine_feature_map, "fine_feature_map")
theta_lazy_coarse_n4_lstd = lstd_zero_algorithm(gamma, sigma, lazy_n4, N,␣
 ↪policy_lazy, coarse_feature_map, "coarse")
theta_lazy_pwl_n4_lstd = lstd_zero_algorithm(gamma, sigma, lazy_n4, N,␣
 ↪policy_lazy, piecewise_linear_feature_map, "pwl")

theta_lazy_values_n4_lstd = {0: theta_lazy_fine_n4_lstd, 1:␣
 ↪theta_lazy_coarse_n4_lstd, 2: theta_lazy_pwl_n4_lstd}


# Agg N4:
theta_agg_fine_n4_lstd = lstd_zero_algorithm(gamma, sigma, agg_n4, N,␣
 ↪policy_agg, fine_feature_map, "fine_feature_map")
theta_agg_coarse_n4_lstd = lstd_zero_algorithm(gamma, sigma, agg_n4, N,␣
 ↪policy_agg, coarse_feature_map, "coarse")
theta_agg_pwl_n4_lstd = lstd_zero_algorithm(gamma, sigma, agg_n4, N,␣
 ↪policy_agg, piecewise_linear_feature_map, "pwl")

theta_agg_values_n4_lstd = {0: theta_agg_fine_n4_lstd, 1:␣
 ↪theta_agg_coarse_n4_lstd, 2: theta_agg_pwl_n4_lstd}
```

3.8 Calculating the value function values

[226]:
```
#Initialize empty lists

value_lazy_n1_lstd = []
value_lazy_n2_lstd = []
value_lazy_n3_lstd = []
value_lazy_n4_lstd = []
value_agg_n1_lstd = []
value_agg_n2_lstd = []
value_agg_n3_lstd = []
value_agg_n4_lstd = []

function = {0: "fine_feature_map", 1: "coarse", 2: "pwl"}

for i in range(0,3):
    value_lazy_n1_lstd.
 ↪append(approximate_value_function(theta_lazy_values_n1_lstd[i],␣
 ↪N,function[i]))
    value_agg_n1_lstd.
 ↪append(approximate_value_function(theta_agg_values_n1_lstd[i],␣
 ↪N,function[i]))

    value_lazy_n2_lstd.
 ↪append(approximate_value_function(theta_lazy_values_n2_lstd[i],␣
 ↪N,function[i]))
    value_agg_n2_lstd.
 ↪append(approximate_value_function(theta_agg_values_n2_lstd[i],␣
 ↪N,function[i]))

    value_lazy_n3_lstd.
 ↪append(approximate_value_function(theta_lazy_values_n3_lstd[i],␣
 ↪N,function[i]))
    value_agg_n3_lstd.
 ↪append(approximate_value_function(theta_agg_values_n3_lstd[i],␣
 ↪N,function[i]))

    value_lazy_n4_lstd.
 ↪append(approximate_value_function(theta_lazy_values_n4_lstd[i],␣
 ↪N,function[i]))
    value_agg_n4_lstd.
 ↪append(approximate_value_function(theta_agg_values_n4_lstd[i],␣
 ↪N,function[i]))
```

3.9 Plotting the results

[234]:
```python
#Set the parameter and the style
x = np.arange(0,N)
plt.figure(figsize=(20, 12))

#First subplot
plt.subplot(2,4,1)
plt.plot(x, value_lazy_n1_lstd[0], label='fine')
plt.plot(x, value_lazy_n1_lstd[1], label='coarse')
plt.plot(x, value_lazy_n1_lstd[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
 ↪function')
plt.title('1000 simulations: Lazy policy LSTD')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,2)
plt.plot(x, value_lazy_n2_lstd[0], label='fine')
plt.plot(x, value_lazy_n2_lstd[1], label='coarse')
plt.plot(x, value_lazy_n2_lstd[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
 ↪function')
plt.title('10000 simulations: Lazy policy LSTD')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,3)
plt.plot(x, value_lazy_n3_lstd[0], label='fine')
plt.plot(x, value_lazy_n3_lstd[1], label='coarse')
plt.plot(x, value_lazy_n3_lstd[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
 ↪function')
plt.title('100000 simulations: Lazy policy LSTD')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,4)
plt.plot(x, value_lazy_n4_lstd[0], label='fine')
plt.plot(x, value_lazy_n4_lstd[1], label='coarse')
plt.plot(x, value_lazy_n4_lstd[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
 ↪function')
plt.title('1000000 simulations: Lazy policy LSTD')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,5)
plt.plot(x, value_agg_n1_lstd[0], label='fine')
plt.plot(x, value_agg_n1_lstd[1], label='coarse')
plt.plot(x, value_agg_n1_lstd[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
 ↪function')
plt.title('1000 simulations: Aggressive policy LSTD')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,6)
plt.plot(x, value_agg_n2_lstd[0], label='fine')
plt.plot(x, value_agg_n2_lstd[1], label='coarse')
plt.plot(x, value_agg_n2_lstd[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
 ↪function')
plt.title('10000 simulations: Aggressive policy LSTD')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,7)
plt.plot(x, value_agg_n3_lstd[0], label='fine')
plt.plot(x, value_agg_n3_lstd[1], label='coarse')
plt.plot(x, value_agg_n3_lstd[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
 ↪function')
plt.title('100000 simulations: Aggressive policy LSTD')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()

plt.subplot(2,4,8)
plt.plot(x, value_agg_n4_lstd[0], label='fine')
plt.plot(x, value_agg_n4_lstd[1], label='coarse')
plt.plot(x, value_agg_n4_lstd[2], label='pwl')
plt.plot(x, optimal_value_function_assignment_3, label = 'optimal value␣
 ↪function')
plt.title('1000000 simulations: Aggressive policy LSTD')
plt.ylabel('value')
plt.xlabel('state')
plt.legend()
plt.show()
```



Comment:

As we can see, the LSTD method converges very well for both policies and gets a value a function that is very similar to the optimal value function! Also the LSTD algorith performs overall better than the TD method!

4 Problem 2: Approximate policy iteration

Using the LSTD implementation, implement an approximate policy iteration method where the following steps are repeated in each step k:

- Policy evaluation
- Policy improvement

In the following, I will use the lazy policy and use the approximate policy iteration method to improve the policy. As a feature map, I use pwl. As stated, lazy policy always chooses zero as the effort irrespective of the state!

4.1 Code for the functions used

```
[124]: Image(url= "Policy_improvement.png", width=500, height=500)
```

```
[124]: <IPython.core.display.Image object>
```

```
[235]: def Greedy_Q_function_max(gamma, p, q_high, q_low, N, theta, function):
           new_policy = []
           action_space = [0,1]
           for i in range(0, N):#skip the first and the last element as there is no␣
        ↪x-1 and x+1!
               Q_value = []
               for action in action_space:
                   if action == 0:
                       q = q_low
                   else:
                       q = q_high
                   V_x_prev = theta @ function(i-1, N)
                   V_x_now = theta @ function(i, N)
                   V_x_fut = theta @ function(i+1, N)
                   Q_value_x = reward_function(i, action, N) + gamma*(1-p)*(q *␣
        ↪V_x_prev + (1-q)*V_x_now) + gamma * p *(q * V_x_now + (1-q)*V_x_fut)
                   Q_value.append(Q_value_x)
               new_policy.append(np.argmax(Q_value))
           return new_policy
```

4.2 Run the simulations

```
[236]: k = [10, 100]
       a = b = 10**5
       sigma = 10**(-5)
       gamma = 0.9
       N = 100
       p = 0.5
       q_high = 0.61
       q_low  = 0.51
       x_init = 99 #starting point
       N_transitions = 10**5
       function = piecewise_linear_feature_map
       feature = "pwl"

       #initialize policies
       opt_policy_lazy = [0 for i in range(0, N)]
       policy_list_init = [opt_policy_lazy, opt_policy_lazy]

       #Store final theta results
       theta_list_final = []
       policy_list_final = []

       for i, iterations in enumerate(k):
           policy = policy_list_init[i]
           sample_transitions_list = sample_transitions(x_init,p, N, N_transitions,␣
        ↪q_high, q_low, policy)
```

```
    for j in range(0, iterations):
        theta = lstd_zero_algorithm(gamma, sigma, sample_transitions_list, N,␣
↪policy, function, feature = feature)
        policy = Greedy_Q_function_max(gamma, p, q_high, q_low, N, theta,␣
↪function)
    policy_list_final.append(policy)
    theta_list_final.append(theta)
```

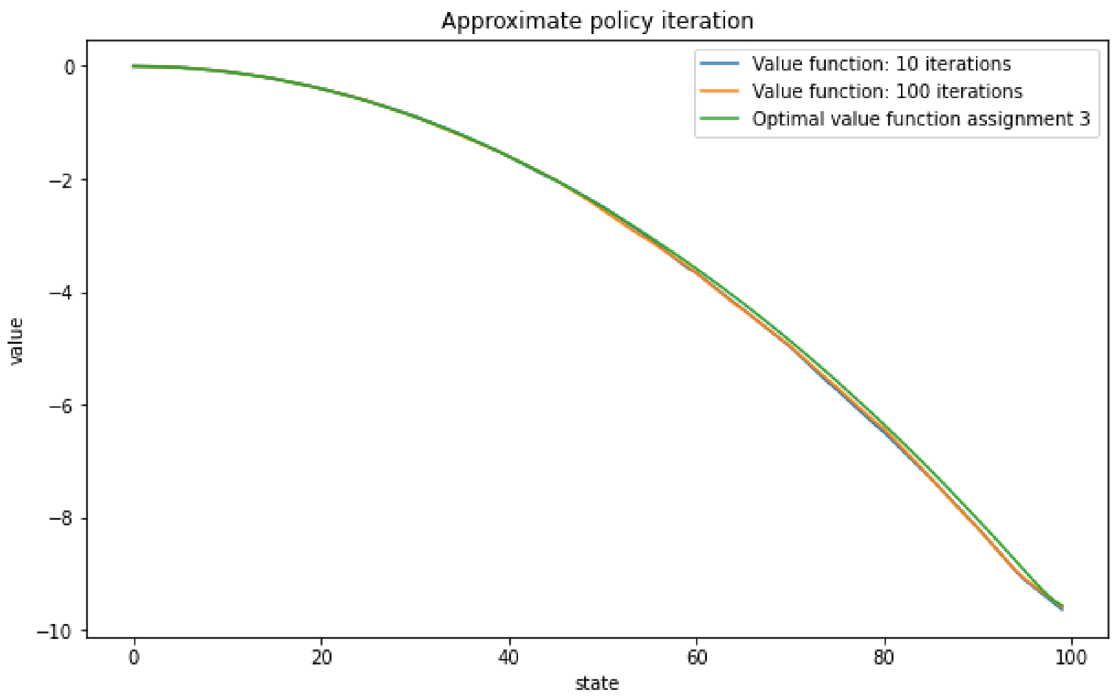
4.3 Plot the final value functions

In the following, I plot the resulting value functions:

4.3.1 Value function

[237]:
```
V_10 = approximate_value_function(theta_list_final[0], N, "pwl")
V_100 = approximate_value_function(theta_list_final[1], N, "pwl")

x = np.arange(0, N)
figure(figsize=(10, 6))
plt.plot(x, V_10, label = 'Value function: 10 iterations')
plt.plot(x, V_100, label = 'Value function: 100 iterations')
plt.plot(x, optimal_value_function_assignment_3, label = "Optimal value␣
↪function assignment 3")
plt.title('Approximate policy iteration')
plt.xlabel("state")
plt.ylabel("value")
plt.legend()
```

[237]:
```
<matplotlib.legend.Legend at 0x7f6d89982520>
```

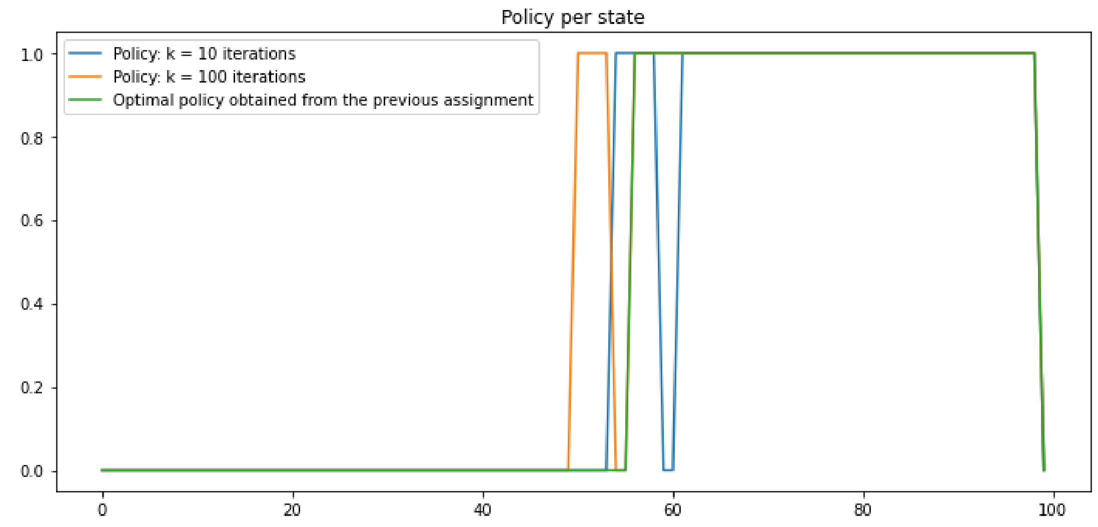


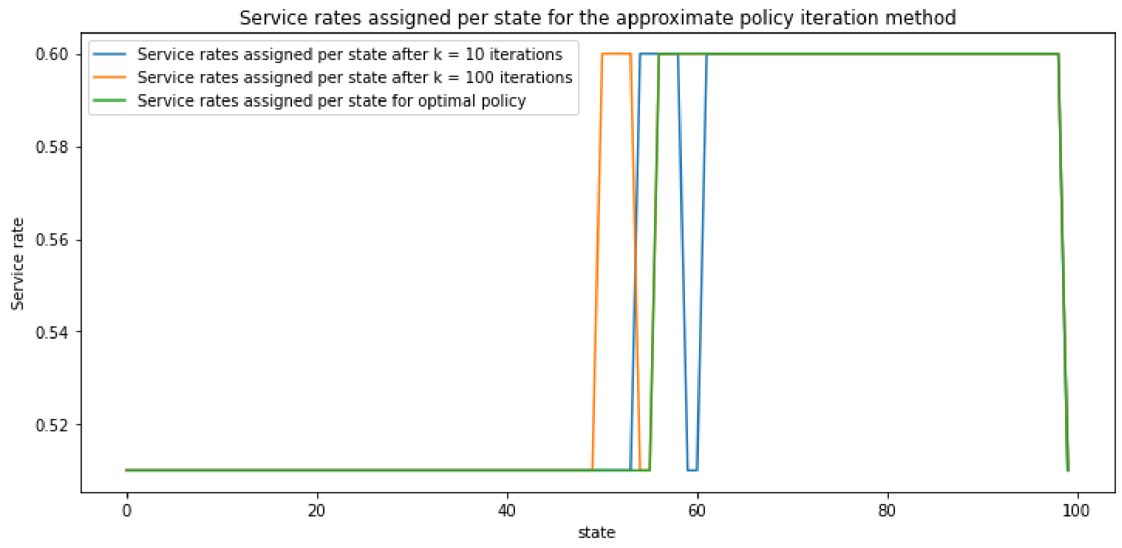
Comment: As we can see, the value functions are almost identical!

4.3.2 Policies

[238]:
```
opt_policy_10 = [0.6 if i == 1 else 0.51 for i in policy_list_final[0]]
opt_policy_100 = [0.6 if i == 1 else 0.51 for i in policy_list_final[1]]
#Optimal policy obtained from the previous assignment
optimal_policy_3 = [0 if (i <=55 or i == 99) else 1 for i in range(0,N)]
optimal_policy_3_service = [0.6 if i == 1 else 0.51 for i in optimal_policy_3]
```

[239]:
```
#Set the parameter and the style
x = np.arange(0,N)
plt.figure(figsize=(12, 12))

#First subplot
plt.subplot(2,1,1)
plt.plot(x, policy_list_final[0], label = 'Policy: k = 10 iterations')
plt.plot(x, policy_list_final[1], label = 'Policy: k = 100 iterations')
plt.plot(x, optimal_policy_3, label = 'Optimal policy obtained from the␣
 ↪previous assignment')
plt.title('Policy per state')
plt.legend()

plt.subplot(2,1,2)
plt.plot(x, opt_policy_10, label='Service rates assigned per state after k = 10␣
 ↪iterations')
plt.plot(x, opt_policy_100, label='Service rates assigned per state after k =␣
 ↪100 iterations')
plt.plot(x, optimal_policy_3_service, label='Service rates assigned per state␣
 ↪for optimal policy')
plt.title('Service rates assigned per state for the approximate policy␣
 ↪iteration method')
plt.ylabel('Service rate')
plt.xlabel('state')
plt.legend()

plt.show()
```





Comment:

As we can see, the algorithm improved the policy which was initialized as a lazy policy. Yet, even after 100 iteration it is still not optimal, which I expect to be better with more iterations. The policies obtained deviate from the optimal policy for the states between 50 and 60.