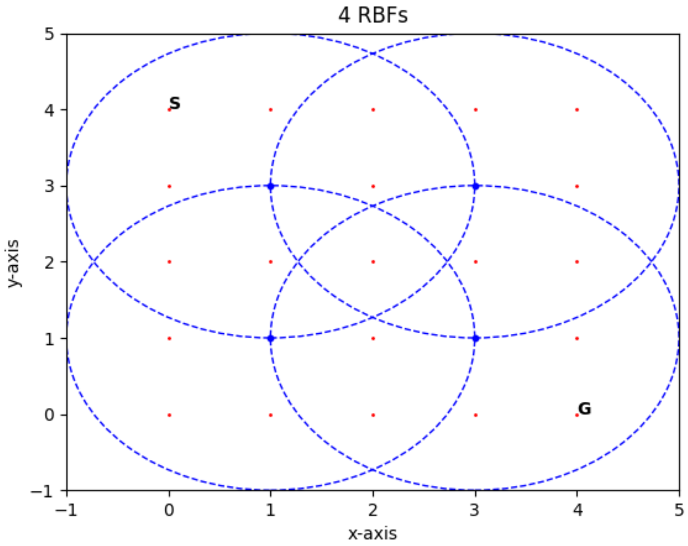
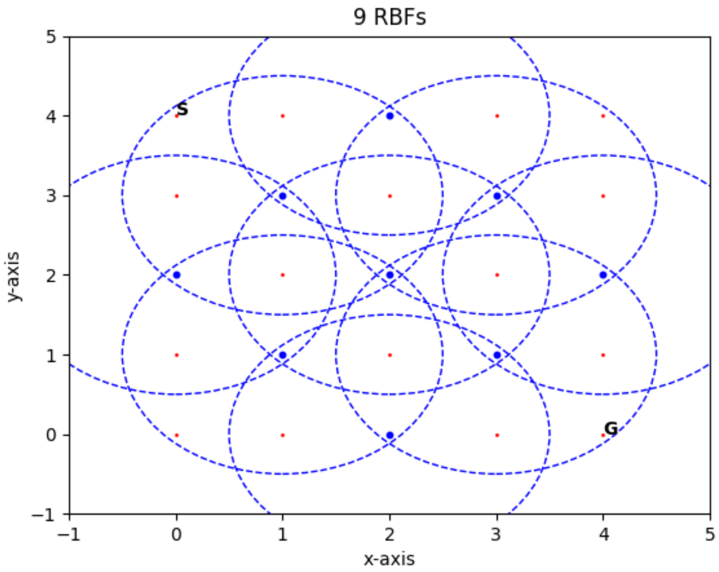
Project 2: Instruction

Dr. Jim La
University of Nevada Reno

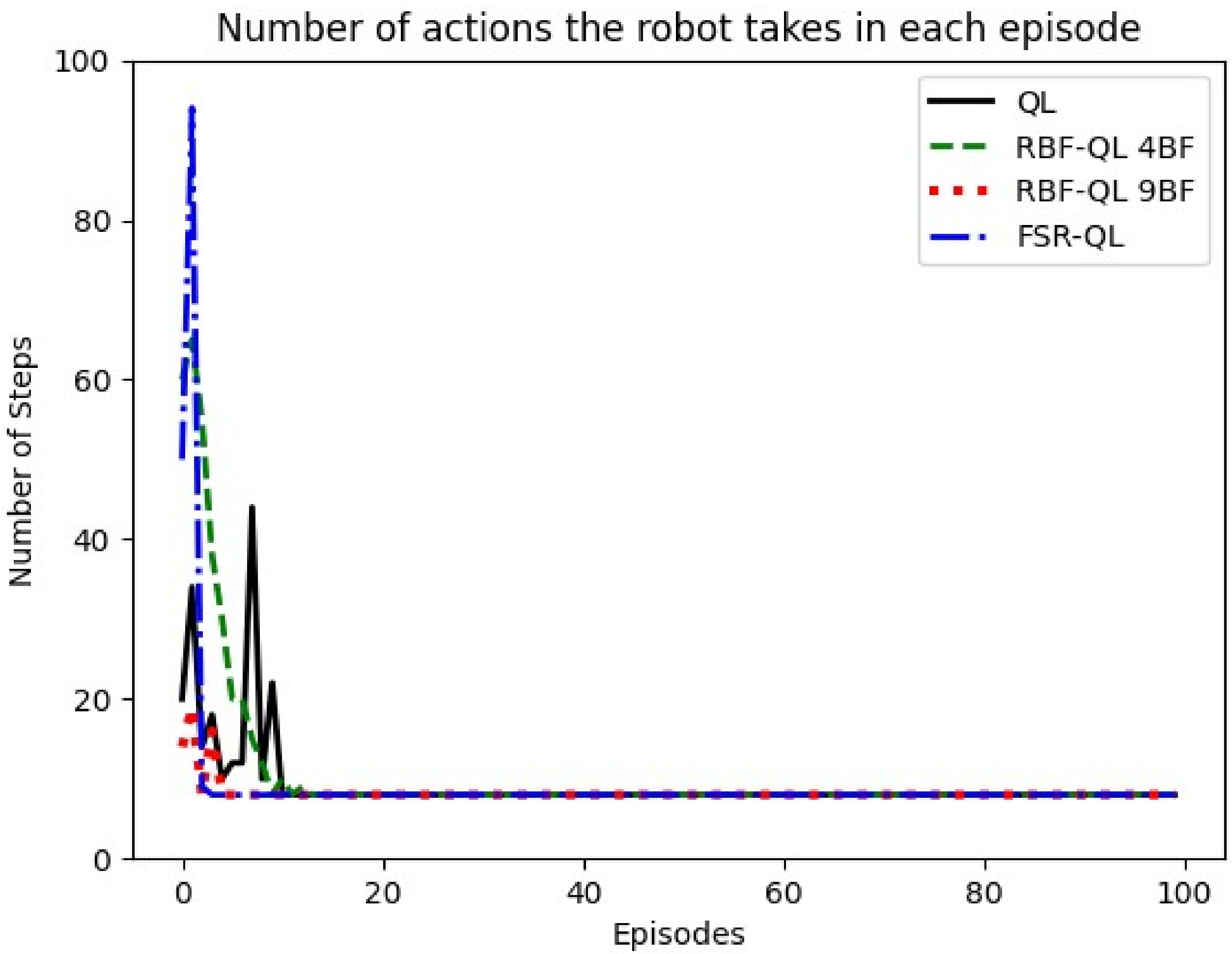
Project 2

- The problem is the same as Project 1. Using Radial Basis Functions (RBF) to approximate the state space. You can use 4 RBF to approximate the state space. Something is similar to these figures.

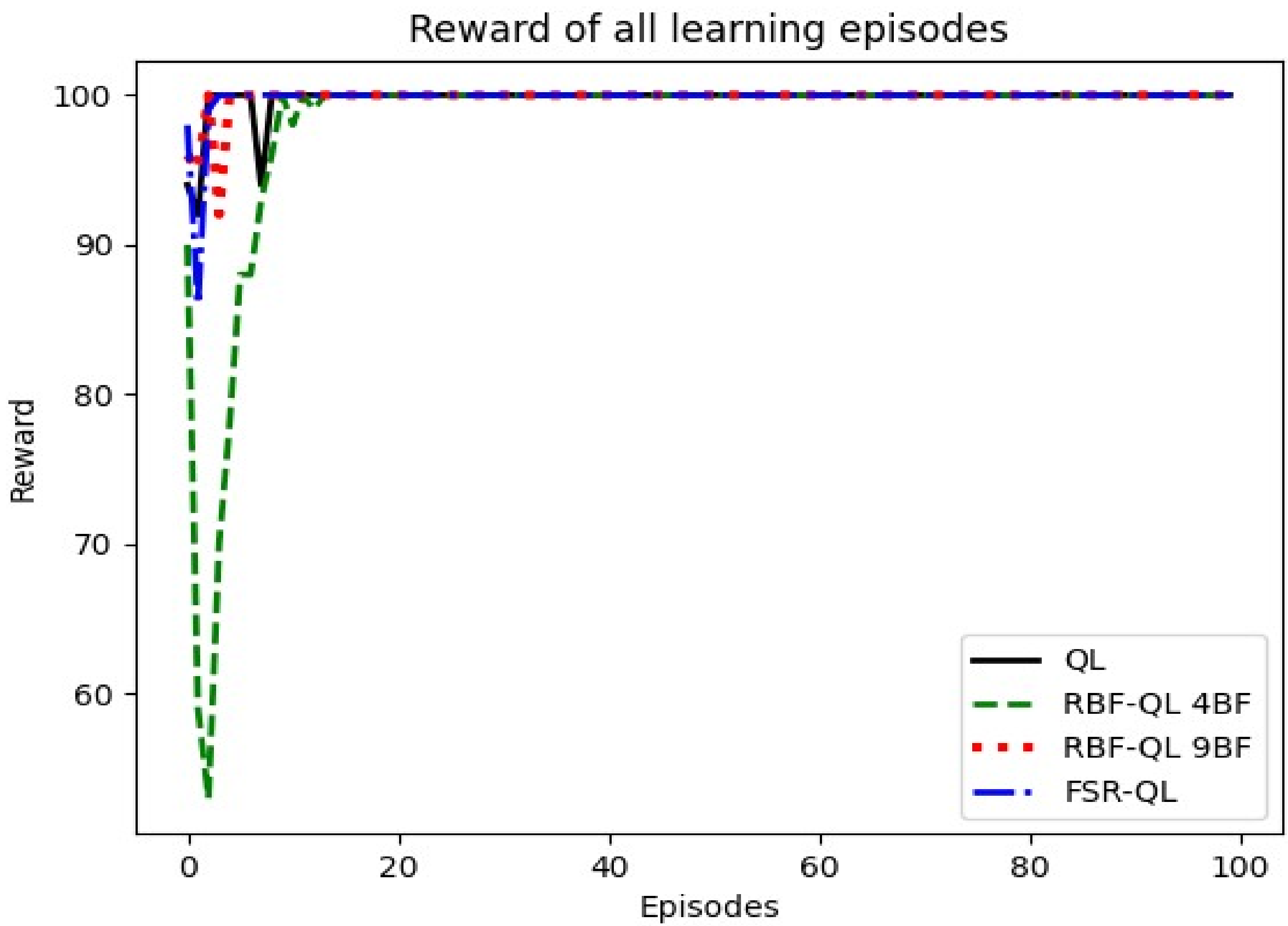




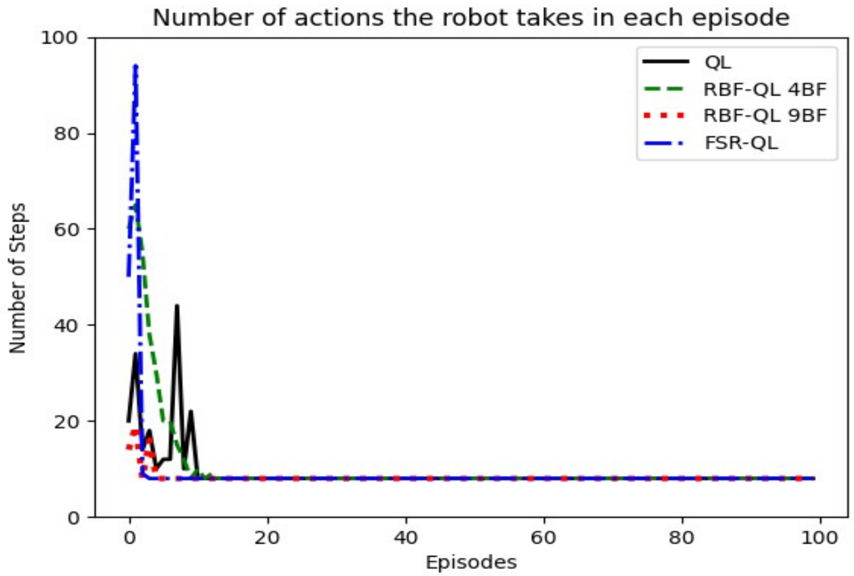
1. (UG: 50 points, G: 40 points) Plot the number of actions the robot takes in each episode for normal Q learning (QL), RBF-QL with 4BF, and RBF-QL with 9BF

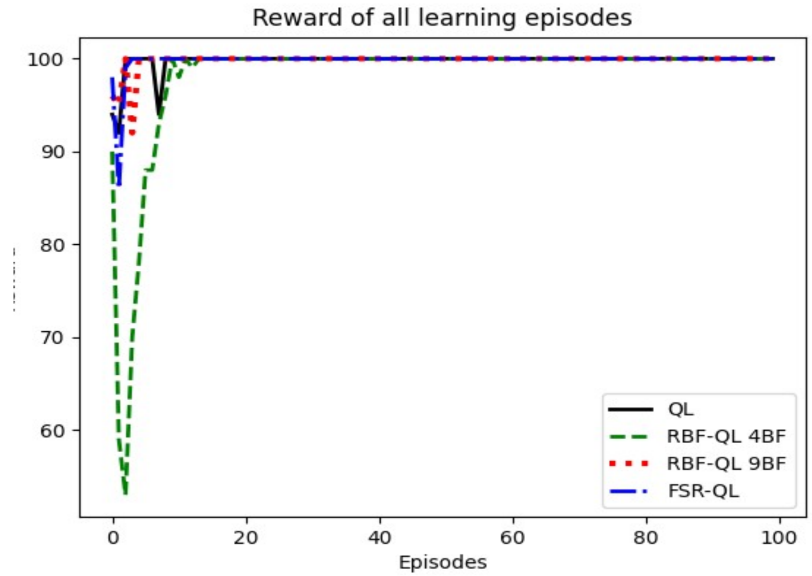


2. (UG: 50 points, G: 40 points) Plot the reward of all learning episodes for each method: Q learning (QL), RBF-QL with 4BF, and RBF-QL with 9BF.



3. (Grad: 20 points) For Grad students (CPE671 only): Implement the FSR and plot its results to compare to the RBF-QL with 9BF. Undergrad students are welcome to implement this FSR with +10 points bonus.





Code: Python

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

Code: Normal Q learning

```
# functions for normal QL
def choose_action_QL_normal(s,Q_table,epsilon,p_array,iteration):
    # select the action with highest estimated action value
    # if several actions with same Q value: randomize
    # get maximum value
    max_action_value = max(Q_table[s].values())
    # get all keys with maximum value
    max_action_keys = [key for key,value in Q_table[s].items() if value == max_action_value]
    # decaying epsilon. Higher epsilon at start of training for more exploration. In later episodes uses exploitation.
    if episode == 0 or episode == 1:
        epsilon = 0.5
    elif episode < 10:
        epsilon = 0.01
    else:
        epsilon = 0.0
    p = p_array[iteration]
    if p >= epsilon:
        # if more than one maximum value action found
        if len(max_action_keys) > 1:
            # randomize
            action = random.choice(max_action_keys)
        else:
            action = max_action_keys[0]
    else:
        # randomize
        action = random.choice(list(Q_table[s]))
    return action
```

Action selection: Normal Q Learning

```
def take_action_QL_normal(action, s):
    # set new state s'
    if action == 'up':
        sprime = (s[0]-1, s[1])
    if action == 'down':
        sprime = (s[0]+1, s[1])
    if action == 'left':
        sprime = (s[0], s[1]-1)
    if action == 'right':
        sprime = (s[0], s[1]+1)
    # set reward r
    # • Action that makes the robot tend to go out of the grid will get a reward of -1 (when the robot is in the border cells)
    # • Action that makes the robot reach the goal will get a reward of 100
    # • All other actions will get a reward of 0
    if sprime == (4,4):
        r = 100
    elif sprime[0] == -1 or sprime[1] == -1 or sprime[0] == 5 or sprime[1] == 5:
        r = -1
    else:
        r = 0
    # if action would be out of the border, robot stays in current cell
    if r == -1:
        sprime = s
    return r, sprime
```

Q Learning Function

```
def QL_normal(gamma, alpha, epsilon, p_array, s, Q_table):
    reward_sum_episode_QL_normal = 0
    action_sum_episode_QL_normal = 0
    iteration = 0
    iteration_terminate = 0
    """ Repeat (for each step of episode): """
    # while loop until goal is reached
    while s != (4,4) and iteration_terminate < 10000:
        """ Choose a from s using policy derived from Q (e.g. epsilon-greedy) """
        action = Your code is here …
        """ Take action a, observe r,s' """
        r, sprime = Your code is here …
        """ Q[s,action] += alpha * (reward + (gamma * predicted_value) - Q[s,action]) """
        predicted_value = Your code is here …
        Q_table[s][action] += Your code is here …

        action_sum_episode_QL_normal += 1
        reward_sum_episode_QL_normal += r
        if iteration < 199:
            iteration += 1
        else:
            iteration = 0
        iteration_terminate += 1
        """ s = s' """
        s = sprime
    """ Until s is terminal """
    return action_sum_episode_QL_normal, reward_sum_episode_QL_normal
```

Action Selection Function for RBF-QL with 4RBF

```
def choose_action_4RBF(s,theta,epsilon,p_array,c,mu,iteration):
    # select the action with highest estimated action value
    # if several actions with same value: randomize
    # calculate phi for all actions
    phi_all_actions = []
    # for all 4 actions
    for i in range(4):
        phi = np.zeros(16)
        for l in range(4):
            phi[i*4+l] = Your code is here … (using Gaussian function  described in the lecture)
        phi_all_actions.append(phi)
    # calculate phi_transp*theta for each action
    phi_t_mult_theta_list = []
    for phi in phi_all_actions:
        phi_t_mult_theta_list.append(phi@theta)

    max_action_keys = [jj for jj, j in enumerate( phi_t_mult_theta_list ) if j == max( phi_t_mult_theta_list )]
    # decaying epsilon. Higher epsilon at start of training for more exploration. In later episodes uses exploitation.
    if episode == 0 or episode == 1:
        epsilon = 0.5
    elif episode < 10:
        epsilon = 0.01
    else:
        epsilon = 0.0
    p = p_array[iteration]
    if p >= epsilon:
        # if more than one maximum value action found
        if len(max_action_keys) > 1:
            # randomize
            action = random.choice(max_action_keys)
        else:
            action = max_action_keys[0]
    else:
        # randomize
        action = random.randint(0, 3)
    return action
```

Take action

```
def take_action_4RBF(action, s):

    # set new state s'
    if action == 0:
        sprime = (s[0]-1, s[1])
    if action == 1:
        sprime = (s[0]+1, s[1])
    if action == 2:
        sprime = (s[0], s[1]-1)
    if action == 3:
        sprime = (s[0], s[1]+1)

    # set reward r
    # • Action that makes the robot tend to go out of the grid will get a reward of -1 (when the robot is in the border cells)
    # • Action that makes the robot reach the goal will get a reward of 100
    # • All other actions will get a reward of 0
    if sprime == (4,4):
        r = 100
    elif sprime[0] == -1 or sprime[1] == -1 or sprime[0] == 5 or sprime[1] == 5:
        r = -1
    else:
        r = 0

    # if action would be out of the border, robot stays in current cell
    if r == -1:
        sprime = s

    return r, sprime
```

Q Learning 4RBF

```
def QL_4RBF(gamma, alpha, epsilon, p_array, s, c, mu, theta):
    reward_sum_episode_4RBF = 0
    action_sum_episode_4RBF = 0
    iteration = 0
    iteration_terminate = 0
    """ Repeat (for each step of episode): """
    # while loop until goal is reached
    while s != (4,4) and iteration_terminate < 10000:
        """ Choose a from A using greedy policy with probability p """
        action = choose_action_4RBF(s,theta,epsilon,p_array,c,mu,iteration)
        """ Take action a, observe r,s' """
        r, sprime = take_action_4RBF(action, s)
        """ Estimate phi_s """
        phi_s = np.zeros(16)
        for i in range(4):
            if action == i:
                for l in range(4):
                    phi_s[i*4+l] = math.exp( - (np.linalg.norm( s - c[l,:])**2 ) /(2*(mu[l]**2)) )
        """ Update """
        # calculate predicted value
        # calculate phi_sprime for all actions
        phi_sprime_all_actions = []
        # for all 4 actions
        for i in range(4):
            phi_sprime = np.zeros(16)
            for l in range(4):
                phi_sprime[i*4+l] = Your code is here … using Gaussian function
            phi_sprime_all_actions.append(phi_sprime)
        # calculate phi_sprime_transp*theta for each action
        phi_sprime_t_mult_theta_list = []
        for phi_sprime in phi_sprime_all_actions:
            phi_sprime_t_mult_theta_list.append(phi_sprime@theta)
        predicted_value = max(phi_sprime_t_mult_theta_list)
        # theta update
        theta +=  Your code is here …
        action_sum_episode_4RBF += 1
        reward_sum_episode_4RBF += r
        if iteration < 199:
            iteration += 1
        else:
            iteration = 0
        iteration_terminate += 1
        """ s = s' """
        s = sprime
    """ Until s is terminal """
    return action_sum_episode_4RBF, reward_sum_episode_4RBF
```

Action Selection FSR

```
def choose_action_FSR(s,theta,epsilon,p_array,iteration):
    # select the action with highest estimated action value
    # if several actions with same Q value: randomize
    # calculate phi for all actions
    phi_all_actions = []
    # for all 4 actions
    for i in range(4):
        phi = np.zeros(40)
        for l in range(10):
            # dimension x
                Your code is here …
            # dimension y
                Your code is here …
        phi_all_actions.append(phi)
    # calculate phi_transp*theta for each action
    phi_t_mult_theta_list = []
    for phi in phi_all_actions:
        phi_t_mult_theta_list.append(phi@theta)
    # calculate argmax phi_transp*theta
    max_action_keys = [jj for jj, j in enumerate( phi_t_mult_theta_list ) if j == max( phi_t_mult_theta_list )]
    # decaying epsilon. Higher epsilon at start of training for more exploration. In later episodes uses exploitation.
    if episode == 0 or episode == 1:
        epsilon = 0.5
    elif episode < 10:
        epsilon = 0.01
    else:
        epsilon = 0.0
    p = p_array[iteration]
    if p >= epsilon:
        # if more than one maximum value action found
        if len(max_action_keys) > 1:
            # randomize
            action = random.choice(max_action_keys)
        else:
            action = max_action_keys[0]
    else:
        # randomize
        action = random.randint(0, 3)

    return action
```

Take action FSR

```
def take_action_FSR(action, s):

    # set new state s'
    if action == 0:
        sprime = (s[0]-1, s[1])
    if action == 1:
        sprime = (s[0]+1, s[1])
    if action == 2:
        sprime = (s[0], s[1]-1)
    if action == 3:
        sprime = (s[0], s[1]+1)

    # set reward r
    # • Action that makes the robot tend to go out of the grid will get a reward of -1 (when the robot is in the border cells)
    # • Action that makes the robot reach the goal will get a reward of 100
    # • All other actions will get a reward of 0
    if sprime == (4,4):
        r = 100
    elif sprime[0] == -1 or sprime[1] == -1 or sprime[0] == 5 or sprime[1] == 5:
        r = -1
    else:
        r = 0

    # if action would be out of the border, robot stays in current cell
    if r == -1:
        sprime = s

    return r, sprime
```

Q Learning FSR

```
def QL_FSR(gamma, alpha, epsilon, p_array, s, theta):
    reward_sum_episode_FSR = 0
    action_sum_episode_FSR = 0
    iteration = 0
    iteration_terminate = 0
    """ Repeat (for each step of episode): """
    # while loop until goal is reached
    while s != (4,4) and iteration_terminate < 10000:
        """ Choose a from A using greedy policy with probability p """
        action = choose_action_FSR(s,theta,epsilon,p_array,iteration)
        """ Take action a, observe r,s' """
        r, sprime = take_action_FSR(action, s)
        """ Estimate phi_s """
        phi_s = np.zeros(40)
        for i in range(4):
            # length 10
            if action == i:
                for l in range(10):
                    # dimension x
                     Your code is here …
                    # dimension y
                     Your code is here …
        """ Update """
        # calculate predicted value # calculate phi_sprime for all actions
        phi_sprime_all_actions = []
        # for all 4 actions
        for i in range(4):
            phi_sprime = np.zeros(40)
            for l in range(10):
                # dimension x
                 Your code is here …
                # dimension y
                 Your code is here …
            phi_sprime_all_actions.append(phi_sprime)
        # calculate phi_sprime_transp*theta for each action
        phi_sprime_t_mult_theta_list = []
        for phi_sprime in phi_sprime_all_actions:
            phi_sprime_t_mult_theta_list.append(phi_sprime@theta)
        predicted_value = max(phi_sprime_t_mult_theta_list)
        # theta update
        theta += alpha * (r + (gamma * predicted_value) - phi_s@theta) * phi_s
        action_sum_episode_FSR += 1
        reward_sum_episode_FSR += r
        if iteration < 199:
            iteration += 1
        else:
            iteration = 0
        iteration_terminate += 1
        """ s = s' """
        s = sprime
    """ Until s is terminal """
    return action_sum_episode_FSR, reward_sum_episode_FSR
if __name__=="__main__":
```