CarMDP Environment, which is provided for your convenience. You should not change code of this environment. This Jupyter notebook is prepared by Kui Wu

In [1]:

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

from gym import Env

class CarMDP(Env):
    """
    Car MDP with simple stochastic dynamics.
    The states are tuples with two elements:
        - a position index (i, j)
        - an integer from (0, 1, 2, 3) representing absolute orientation (se
    For example, the state
        s = (0, 1, 2)
    represents the car in the cell with indices (0, 1) and oriented to face
    """
    def __init__(self, width, height, obstacles, goal_transition, initial_st
                 collision_reward=-5., goal_reward=10., stagnation_penalty=-
        self.width = width
        self.height = height
        self.grid_map = np.ones((width, height))
        for cell in obstacles:
            self.grid_map[cell[0], cell[1]] = 0.
        self.obstacles = obstacles
        self.orientations = {0: 'North', 1: 'East', 2: 'South', 3: 'West'}
        self.A = {0: 'Forward', 1: 'Left', 2: 'Right', 3: 'Brake'}
        self.goal_transition = goal_transition  # Tuple containing start and

        self.p_corr = p_corr
        self.p_err = (1. - p_corr)/2.

        self.base_reward = base_reward
        self.collision_reward = collision_reward
        self.goal_reward = goal_reward
        self.stagnation_penalty = stagnation_penalty
        self.state_history = []
        self.action_history = []
        self.reward_history = []

        assert initial_state[0] >= 0 and initial_state[1] >= 0 and initial_s
               initial_state[2] in self.orientations, "ERROR: initial state
        self.state_history = [initial_state]
        self.action_history = []
        self.reward_history = []
        self.init_state=initial_state


    def reset(self):
        self.state_history = [self.init_state]
        self.action_history = []
        self.reward_history = []

    def is_collision(self, state):
        is_out_of_bounds = state[0] < 0 or state[0] >= self.width or state[1
            state[1] >= self.height
        return is_out_of_bounds or (state[0], state[1]) in self.obstacles
```

```
    def transition_dynamics(self, state, action):
        assert not self.is_collision(state), "ERROR: can't take an action f
        delta = 1
        orientation = state[2]

        if self.orientations[orientation] == 'North':
            left = (state[0] - delta, state[1] - delta)
            forward = (state[0], state[1] - delta)
            right = (state[0] + delta, state[1] - delta)
        elif self.orientations[orientation] == 'West':
            left = (state[0] - delta, state[1] + delta)
            forward = (state[0] - delta, state[1])
            right = (state[0] - delta, state[1] - delta)
        elif self.orientations[orientation] == 'South':
            left = (state[0] + delta, state[1] + delta)
            forward = (state[0], state[1] + delta)
            right = (state[0] - delta, state[1] + delta)
        elif self.orientations[orientation] == 'East':
            left = (state[0] + delta, state[1] - delta)
            forward = (state[0] + delta, state[1])
            right = (state[0] + delta, state[1] + delta)

        # p gives categorical distribution over (state, left, forward, righ
        if self.A[action] == 'Forward':
            p = np.array([0., self.p_err, self.p_corr, self.p_err])
        elif self.A[action] == 'Right':
            p = np.array([0., 0., 2.*self.p_err, self.p_corr])
        elif self.A[action] == 'Left':
            p = np.array([0., self.p_corr, 2. * self.p_err, 0.])
        elif self.A[action] == 'Brake':
            p = np.array([self.p_corr, 0., 2. * self.p_err, 0.])

        candidate_next_state_positions = (state, left, forward, right)
        next_state_position = candidate_next_state_positions[categorical_sa

        # Handle orientation dynamics (deterministic)
        new_orientation = orientation
        if self.A[action] == 'Right':
            new_orientation = (orientation + 1) % 4
        elif self.A[action] == 'Left':
            new_orientation = (orientation - 1) % 4

        return next_state_position[0], next_state_position[1], new_orientat


    def step(self, action):
        assert action in self.A, f"ERROR: action {action} not permitted"
        terminal = False
        current_state = self.state_history[-1] # -1 means the current eleme
        next_state = self.transition_dynamics(current_state, action)
        if self.is_collision(next_state):
            reward = self.collision_reward
            terminal = True
        elif (current_state[0], current_state[1]) == self.goal_transition[0
                (next_state[0], next_state[1]) == self.goal_transition[1]:
            reward = self.goal_reward
            terminal = True  # TODO: allow multiple laps like this?
        elif current_state == next_state:
            reward = self.stagnation_penalty
```

```
            terminal = False
        else:
            reward = self.base_reward
            terminal = False

        self.state_history.append(next_state)
        self.reward_history.append(reward)
        self.action_history.append(action)

        return next_state, reward, terminal, []

    def render(self, title):
        self._plot_history(title)

    def _plot_history(self, title):
        """
        Plot the MDP's trajectory on the grid map.
        :param title:
        :return:
        """
        fig = plt.figure()
        plt.imshow(self.grid_map.T, cmap='gray')
        plt.grid()
        x = np.zeros(len(self.state_history))
        y = np.zeros(x.shape)
        for idx in range(len(x)):
            x[idx] = self.state_history[idx][0]
            y[idx] = self.state_history[idx][1]
            if self.state_history[idx][2] == 0:
                plt.arrow(x[idx], y[idx], 0., -0.25, width=0.1)
            elif self.state_history[idx][2] == 1:
                plt.arrow(x[idx], y[idx], 0.25, 0., width=0.1)
            elif self.state_history[idx][2] == 2:
                plt.arrow(x[idx], y[idx], 0., 0.25, width=0.1)
            else:
                plt.arrow(x[idx], y[idx], -0.25, 0., width=0.1)

        plt.plot(x, y, 'b-')  # Plot trajectory
        plt.xlim([-0.5, self.width + 0.5])
        plt.ylim([self.height + 0.5, -0.5])
        plt.title(title)
        plt.xlabel('x')
        plt.ylabel('y')
        plt.show()
        return fig


def categorical_sample_index(p: np.ndarray) -> int:
    """
    Sample a categorical distribution.

    :param p: a categorical distribution's probability mass function (i.e.,
              returning idx for an integer 0 <= idx < len(p)). I.e., np.sum
    :return: index of a sample weighted by the categorical distribution desc
    """
    P = np.cumsum(p)
    sample = np.random.rand()
    return np.argmax(P > sample)
```

Below is the skeleton code of your agent. Your solution should be filled here.

In [2]:

```
class ReinforcementLearningAgent:
    """
    Your implementation of a reinforcement learning agent.
    Feel free to add additional methods and attributes.
    """
    def __init__(self):
        ### STUDENT CODE GOES HERE
        # Set any parameters
        # You can add arguments to __init__, so log as they have default va
        self.gamma = 1
        self.epsilon = 0.001
        self.alpha = 0.5
        self.Q = np.zeros((9*6, 4))
        self.shape = (9,6)
        self.state_history = []
        self.action_history = []

    def reset(self, init_state) -> int:
        """
        Called at the start of each episode.

        :param init_state:
        :return: first action to take.
        """
        ### STUDENT CODE GOES HERE
        action = 0
        self.state_history = []
        self.action_history = []
        self.state_history.append(init_state)
        state = (init_state[0], init_state[1])
        state = np.ravel_multi_index(tuple(state), self.shape)
        if np.random.random() < self.epsilon:
            action = np.random.randint(4)
        else:
            action = np.argmax(self.Q[state,:])
        self.action_history.append(action)
        return action

    def next_action(self, reward: float, state: int, terminal: bool) -> int
        """
        Called during each time step of a reinforcement learning episode

        :param reward: reward resulting from the last time step's action
        :param state: state resulting from the last time step's action
        :param terminal: bool indicating whether state is a terminal state
        :return: next action to take
        """
        if terminal:
            return 0
        ### STUDENT CODE GOES HERE
        # Produce the next action to take in an episode as a function of the
        # You may find it useful to track past actions, states, and rewards
        # Additionally, algorithms that learn during an episode (e.g., temp
        self.state_history.append(state)
        prev_state = self.state_history[-1]
        #print("prev: ",prev_state)
        prev_state = (prev_state[0], prev_state[1])
        #print("next: ",prev_state,"\n")
```

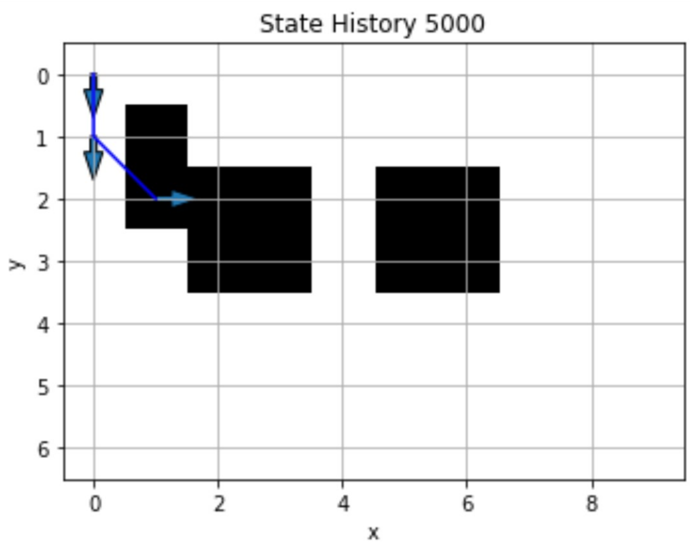
```
        #prev_orientation = prev_state[2]
        prev_action = self.action_history[-1]
        prev_state = np.ravel_multi_index(tuple(prev_state), self.shape)
        next_state = (state[0], state[1])
        #next_orientation = state[2]
        next_state = np.ravel_multi_index(tuple(next_state), self.shape)

        eps_action = 0
        if np.random.random() < self.epsilon:
            eps_action = np.random.randint(4)
        else:
            eps_action = np.argmax(self.Q[next_state,:])

        self.Q[prev_state, prev_action] += self.alpha*(reward + self.gamma*s

        return eps_action  # Random policy (CHANGE THIS)

    def finish_episode(self):
        """
        Called at the end of each episode.
        :return: nothing
        """
        ### STUDENT CODE GOES HERE
        # Algorithms that learn from an entire episode (e.g., Monte Carlo) r
        pass
```

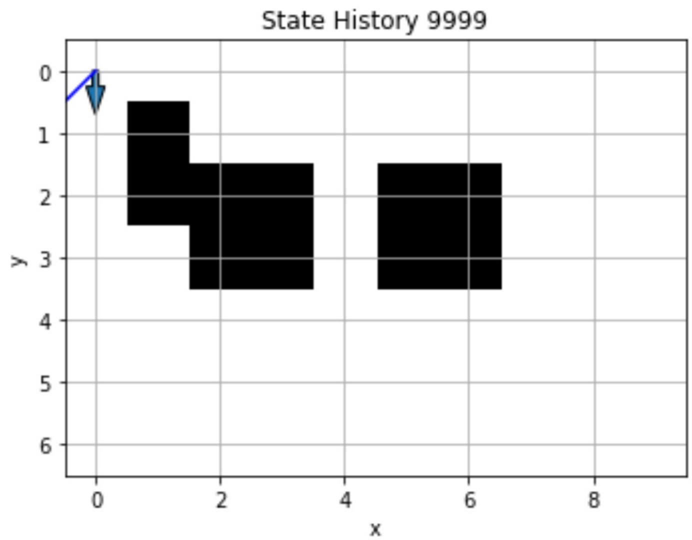
Below is the sample test code. In the final print out you need to print out the correct policy name (It is random so far). Note that since this is a model-free solution, we will use a different test environment (i.e., the locations and the sizes of barriers are different) to test your code.

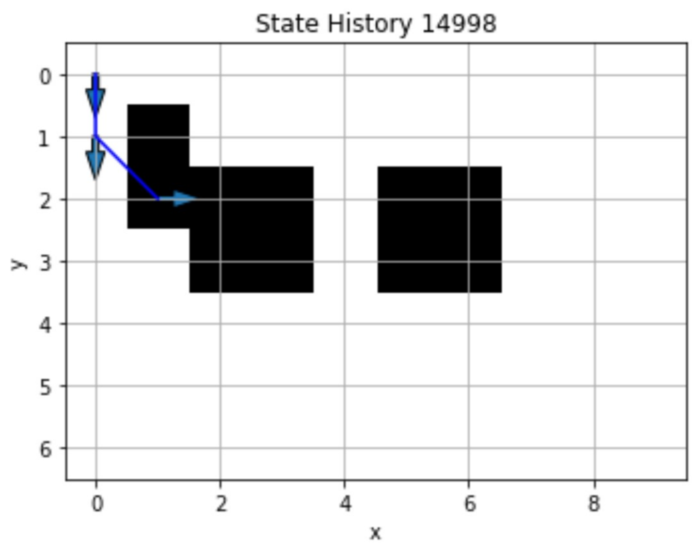
In [3]:

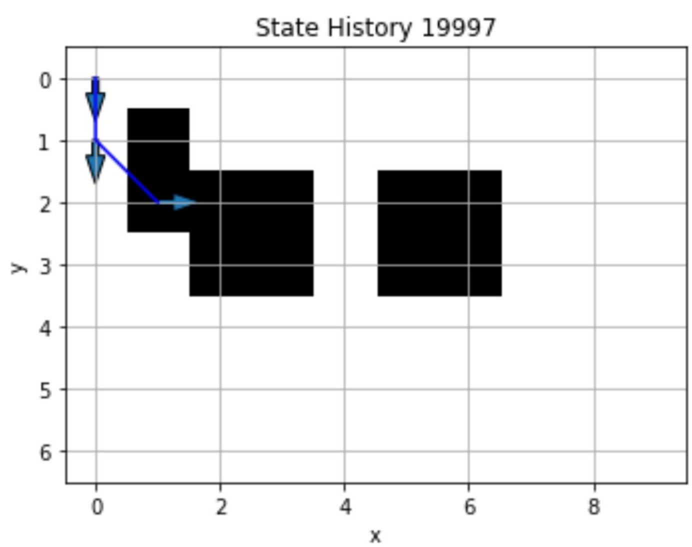
```
def test_rl_algorithm(rl_agent, car_mdp, initial_state, n_episodes=10000, n_
    """
    Code that will be used to test your implementation of ReinforcementLearn
    As you can see, you are responsible for implementing three methods in Re
        - reset (called at the start of every episode)
        - next_action (called at every time step of an episode)
        - finish_episode (called at the end of each episode)

    :param rl_agent: an instance of your ReinforcementLearningAgent class
    :param car_mdp: an instance of CarMDP
    :param init_state: the initial state
    :param n_episodes: number of episodes to use for this test
    :param n_plot: display a plot every n_plot episodes
    :return:
    """
    returns = []
    for episode in range(n_episodes):
        G = 0.  # Keep track of the returns for this episode (discount fact
        # Re-initialize the MDP and the RL agent
        car_mdp.reset();
        action = rl_agent.reset(initial_state)
        terminal = False
        while not terminal:  # Loop until a terminal state is reached
            next_state, reward, terminal, [] = car_mdp.step(action)
            G += reward
            #print(terminal)
            action = rl_agent.next_action(reward, next_state, terminal)
        rl_agent.finish_episode()
        returns += [G]

        # Plot the trajectory every n_plot episodes
        if episode % n_plot == 0 and episode > 0:
            car_mdp.render('State History ' + str(episode + 1))

    return returns


if __name__ == '__main__':

    # Size of the CarMDP map (any cell outside of this rectangle is a termi
    width = 9
    height = 6
    initial_state = (0, 0, 2)  # Top left corner (0, 0), facing "Down" (2)
    obstacles = [(2, 2), (2, 3), (3, 2), (3, 3),  # Cells filled with obsta
                 (5, 2), (5, 3), (6, 2), (6, 3),
                 (1, 1), (1, 2)]
    goal_transition = ((1, 0), (0, 0))  # Transitioning from cell (1, 0) to
    p_corr = 0.95  # Probability of actions working as intended

  # Create environment
    car_mdp = CarMDP(width, height, obstacles, goal_transition, initial_stat

 # Create RL agent. # You must complete this class in your solution, it is 

    rl_agent = ReinforcementLearningAgent()


    student_returns = test_rl_algorithm(rl_agent, car_mdp, initial_state, n_
```

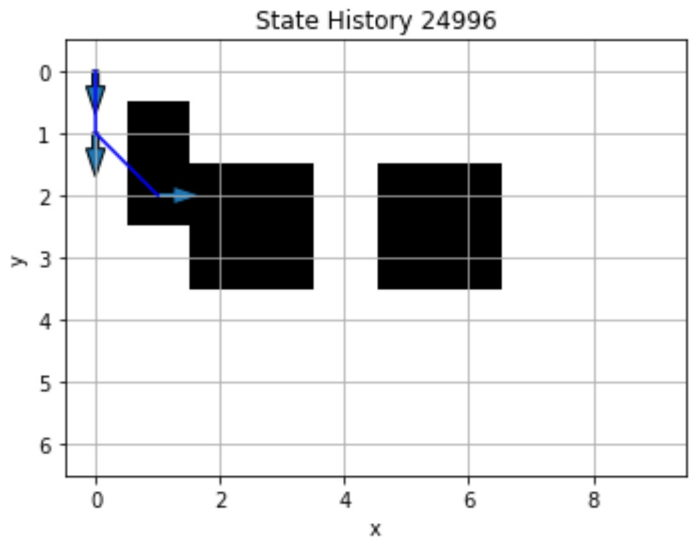
```
# Example plot. You need to change it according to the assignment requi
n_runs = 50
n_episodes = 10000
returns = np.zeros((n_runs, n_episodes))
for run in range(n_runs):
    rl_agentnew = ReinforcementLearningAgent()
    returns[run, :] = test_rl_algorithm(rl_agentnew, car_mdp, initial_s

# Plot one curve like this for each parameter setting - the template co
# returns a random action, so this example curve will just be noise. Wh
# should increase as the number of episodes increases. Feel free to cha
rolling_average_width = 100
# Compute the mean (over n_runs) for each episode
mean_return = np.mean(returns, axis=0)
# Compute the rolling average (over episodes) to smooth out the curve
rolling_average_mean_return = np.convolve(mean_return, np.ones(rolling_
plt.figure()
plt.plot(rolling_average_mean_return, 'b-')  # Plot the smoothed averag
plt.grid()
plt.title('Learning Curve')
plt.xlabel('Episode')
plt.ylabel('Average Return')
plt.legend(['alpha = 0.5, epsilon = 0.01'])
plt.show()
```

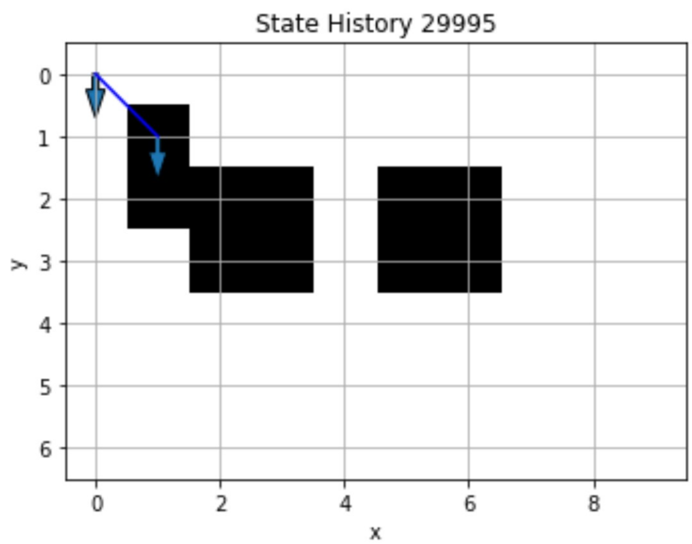


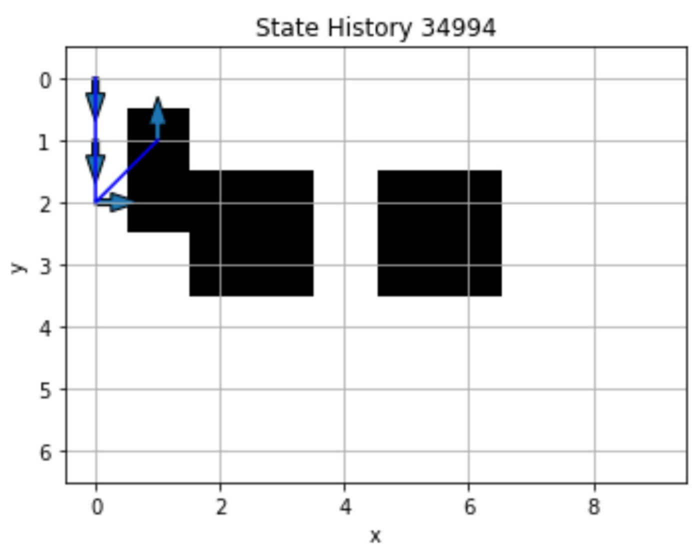

State History 9999
x
y

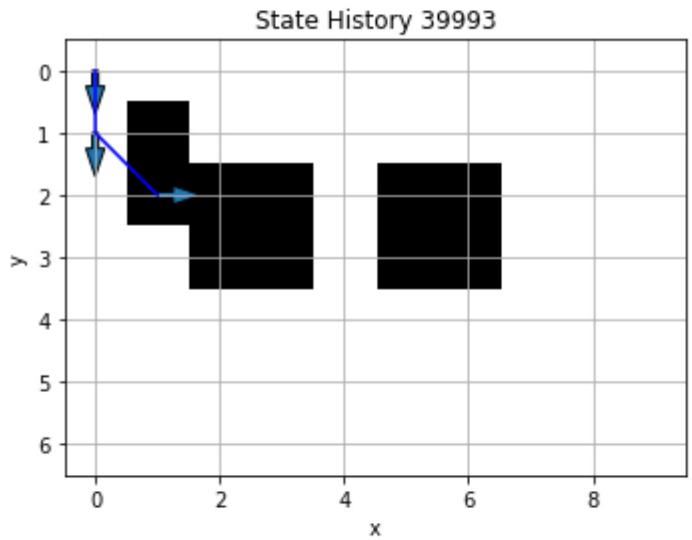

State History 14998
x
y

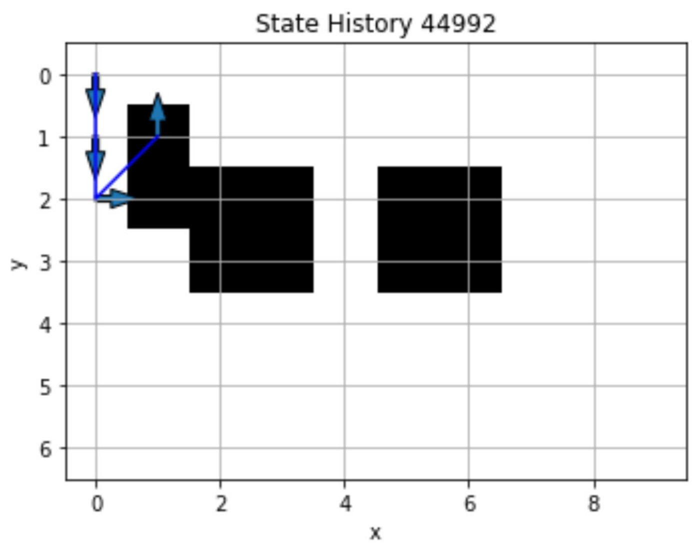

State History 19997
x
y

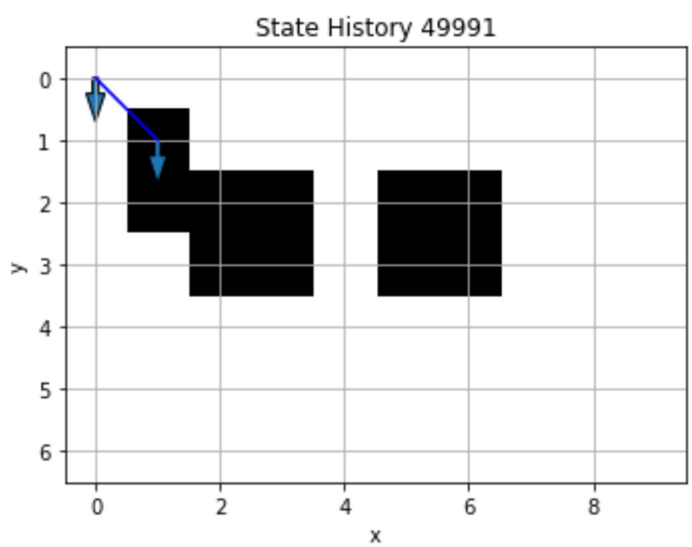

State History 24996
0
1
2
3
4
5
6
y
0
2
4
6
8
x

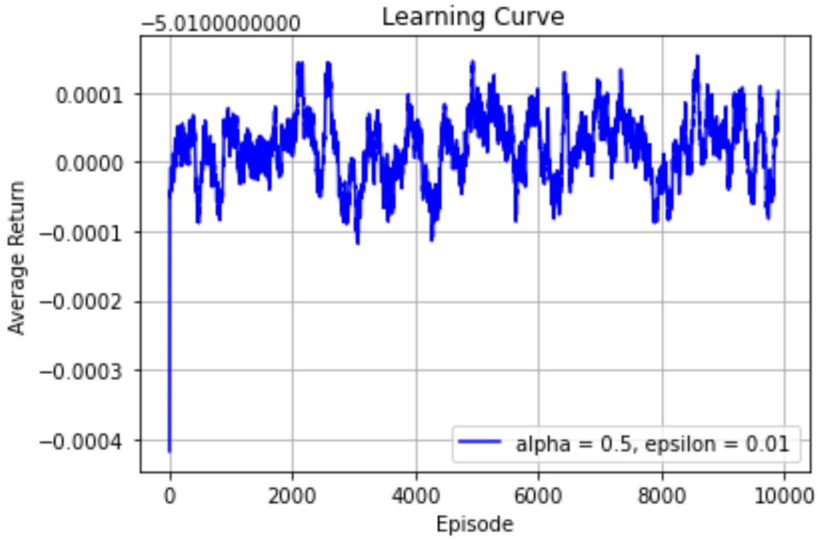

State History 29995
0
1
2
3
4
5
6
y
0
2
4
6
8
x


State History 34994
0
1
2
3
4
5
6
y
0
2
4
6
8
x


State History 39993
y
x


State History 44992
y
x


State History 49991
y
x


Learning Curve
−5.0100000000
0.0001
0.0000
−0.0001
−0.0002
−0.0003
−0.0004
Average Return
0
2000
4000
6000
8000
10000
Episode
alpha = 0.5, epsilon = 0.01

```
In [ ]:
```