System theory homework 3

- Implementation--Briefly describe your implementation

```
def dyna_q(args, q_value, model, maze):

    state = maze.START_STATE
    steps=0
    #loop forever
    while(1):
        #choose action using epislon-greedy method
        action = choose_action(state, q_value, maze, args.epislon)
        #derived next state and reward by taking action we choose
        next_state=maze.step(state,action)[0]
        reward=maze.step(state,action)[1]
        #update q_value[state[0],state[1],action]
        q_value[state[0],state[1],action]+= args.alpha*(reward+ args.gamma*max(q_value[next_state[0],next_state[1]])-q_value[state[0],state[1],action])
        #derived model(Model(S,A)) from reward and next state we choose
        model.store(state, action, next_state, reward)
        #repeat in n times
        for i in range(args.planning_steps):
            sample_state,sample_action,sample_next_state,sample_reward=model.sample()
            #update q_value [sample_state[0],sample_state[1],sample_action]
            q_value[sample_state[0],sample_state[1],sample_action] += args.alpha*(sample_reward+
args.gamma*max(q_value[sample_next_state[0],sample_next_state[1]])-q_value[sample_state[0],sample_state[1],sample_action])
        steps+=1
        #update the state
        state=next_state
        #if it arrive goal state then break
        if(state in maze.GOAL_STATES):
            break
    return steps
```

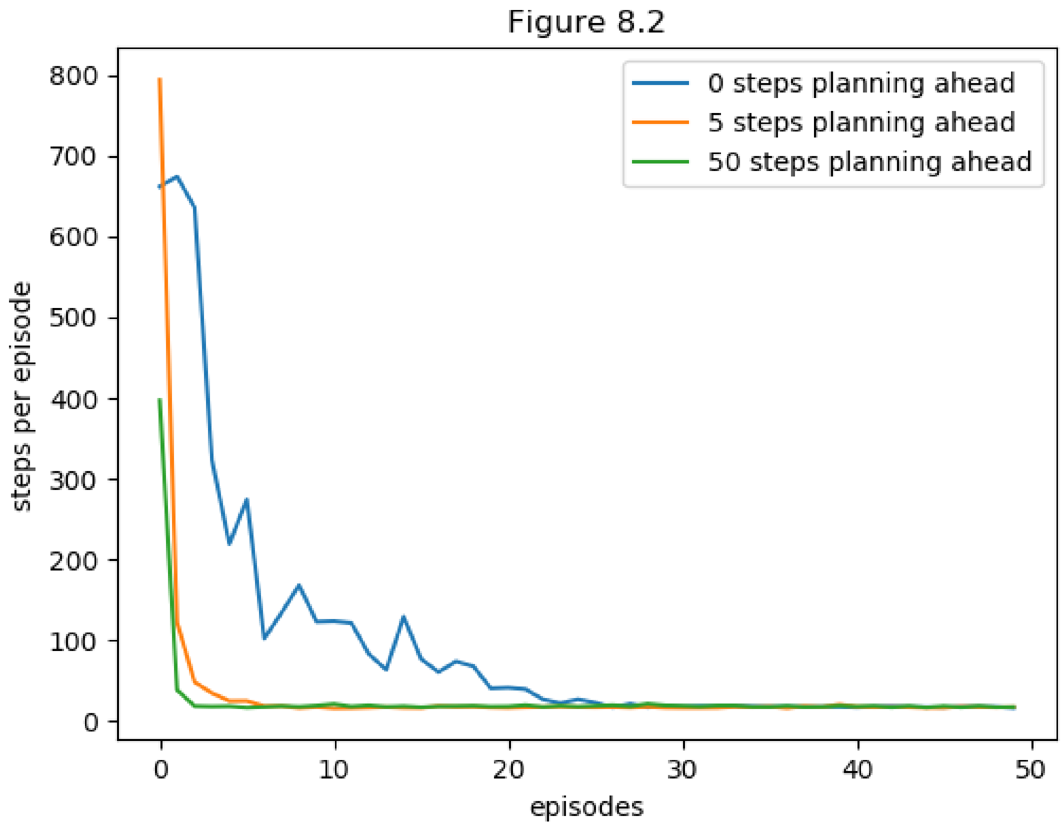
```python
class InternalModel(object):
    """
    Description:
        We'll create a tabular model for our simulated experience. Please complete the following code.
    """
    def __init__(self):
        self.model = dict()
        self.rand = np.random

    def store(self, state, action, next_state, reward):
        #store the previous experience into the model
        self.model[state[0],state[1], action]= [reward,next_state[0],next_state[1]]

    def sample(self):
        #choose state and action randomly from internal model
        s0, s1, action= random.choice(list(self.model.keys()))
        #derived reward and next state from internal model (model[s0, s1, action])
        [reward,next_state_0,next_state_1]=self.model[s0, s1, action]
        state=[s0,s1]
        next_state = [next_state_0,next_state_1]

        return state, action, next_state, reward
```

- Experiments and Analysis---Plot result. (As example above)



- Experiments and Analysis---Explain how learned model improves the performance

From the graphic above we can observe that, when 0 steps planning ahead, it performances bad. And its performance better when 5 steps planning ahead and its performance is best when 50 steps planning ahead, both of them can converge in just a few episodes, so this can prove that the learned model improves the performance. We can also discover that the convergent rate of 50 steps planing ahead is faster than 5 steps planning ahead. This is because of that we can learn more from experience .From the above, we can conclude that learned model improves the performance.