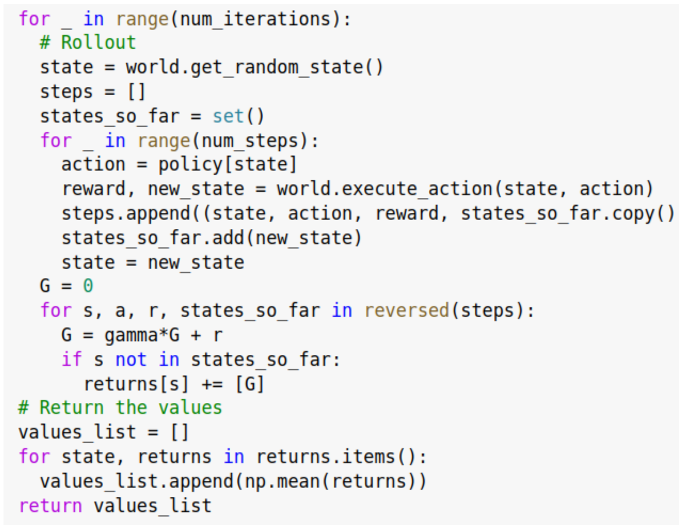
Project 4 - MDPs and Reinforcement Learning

Hoang-Dung Bui,
George Mason University
Fairfax, USA
hbui20@gmu.edu

I. Evaluating Policies

A. First-Visit MC Policy Evaluation

The code for the first visit MC Policy Evaluation is shown in Fig. 1. After getting the data in the set *states_so_far*, we calculate the returns in reversed direction. If the state is not in the *states_so_far*, we will add the returns of that state into the *returns* list. After that, we calculate the average values of state function and return it.



```
for _ in range(num_iterations):
  # Rollout
  state = world.get_random_state()
  steps = []
  states_so_far = set()
  for _ in range(num_steps):
    action = policy[state]
    reward, new_state = world.execute_action(state, action)
    steps.append((state, action, reward, states_so_far.copy()
    states_so_far.add(new_state)
    state = new_state
  G = 0
  for s, a, r, states_so_far in reversed(steps):
    G = gamma*G + r
    if s not in states_so_far:
      returns[s] += [G]
# Return the values
values_list = []
for state, returns in returns.items():
  values_list.append(np.mean(returns))
return values_list
```

Fig. 1: The code of evaluate_policy_first_visit_mc function

For two policy (1 and 2), with the chances varying from 0.0 to 1.0, the average values of the value function are shown in Table I. For policy 1 (random action selection), obviously, it is not changed much as the chances varying because its decision relies on random already. For the policy 2, the action selection is chosen by chance of 0.2 providing the best average values of all states. If the chances of action selection are over than 0.2, the averages value of all states reduces significantly.

Chances	0.0	0.2	0.5	0.8	1.0
Policy 1-avar. values	-49.12	-48.89	-48.48	-47.72	-46.81
Policy 2-avar. values	15.23	20.75	-12.22	-36.61	-46.81

TABLE I: The chances vs Policies average values

The generated plot of the computed value for each policy is shown in Fig. 2 (for a random move chance of 0.5). For policy 1 with random action selection, the average states values are similar and low. However, for policy 2, some states get the values of 100. It seems correct, because those states are in middle of the environment, which are close to the goal.

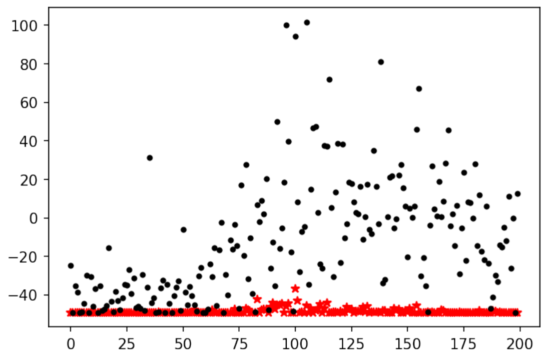
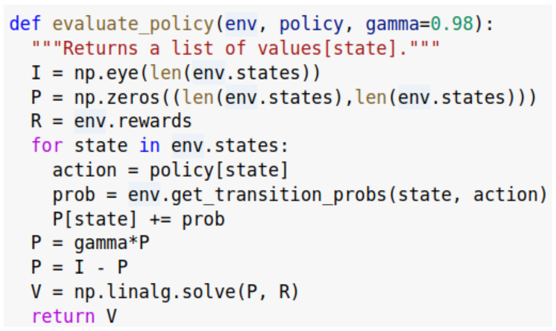


Fig. 2: The average values of all states of policy 1 (red) and policy 2 (black)

Question: How do the costs of each policy compare as the random move chance becomes 1.0? Why?
Answer: As the chance increases from 0.5 to 1.0, the average values will reduce gradually. Because the chance is 1.0, it means the actions are selected random completely, so the updated policy is not improved the action's selection. That's why the average value function goes down significantly.

B. Linear Algebra Solution

The code of *evaluate_policy* function is shown in Fig. 3. We created two matrices I and P with the same dimensions of $env.states \times env.states$. The initial reward vector is assigned to $env.rewards$. The *for* loop runs through all states in $env.states$, and add the transition probabilities of each state into matrix P. The value function of the state set is calculated by the function *np.linalg.solve(P,R)*.



```
def evaluate_policy(env, policy, gamma=0.98):
  """Returns a list of values[state]."""
  I = np.eye(len(env.states))
  P = np.zeros((len(env.states),len(env.states)))
  R = env.rewards
  for state in env.states:
    action = policy[state]
    prob = env.get_transition_probs(state, action)
    P[state] += prob
  P = gamma*P
  P = I - P
  V = np.linalg.solve(P, R)
  return V
```

Fig. 3: evaluate_policy function

The average values of all states from policy 1 and 2 with the chances varying are shown in Table II. The results are similar to the *First-Visit MC Policy Evaluation* approach.

	Chances				
Chances	0.0	0.2	0.5	0.8	1.0
Policy 1-avar. values	-49.94	-49.67	-49.23	-48.46	-47.56
Policy 2-avar. values	14.35	22.62	-13.93	-37.61	-47.56

TABLE II: The chances vs Policies average values with linear algebra

The states' values of linear algebra approach is shown in Fig. 4. Comparing to *First-Visit MC Policy Evaluation*, the average states' values are neat and concentrated in some area. The max value in this approach is smaller than the *First-Visit MC Policy Evaluation*. However, it is significantly faster.

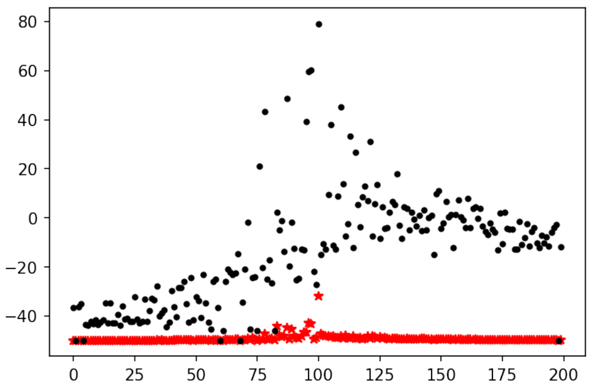


Fig. 4: The states' values of linear algebra solution

Question: How do the costs of each policy compare as the random move chance becomes 1.0? Why?

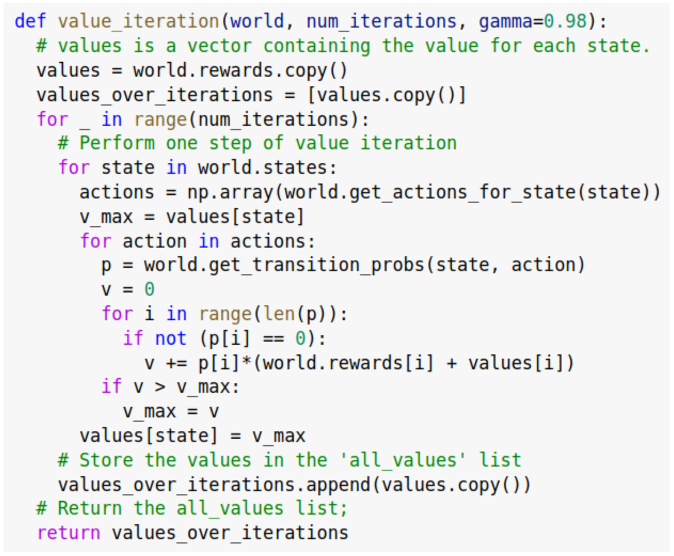
Answer: It is similar to the *First-Visit MC Policy Evaluation*, as the chances reach 1.0, the average value function reduces gradually. The reason is explained in section I-A.

II. Value Iterations

The implementation of *value_iteration* and *compute_policy_from_values* functions are shown in Fig. 5 and 6. In *value_iteration* function, for each *iteration*, we run through all the states and determine the possible actions set for each state. Then we determine the best action which returns the maximum rewards by calculating the expectation value of the state with the action. This function outputs the state's values list.

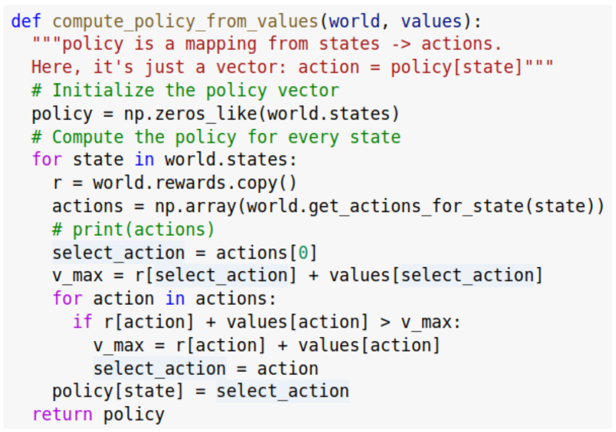
The state's value's list will be used to update the policy by the function *compute_policy_from_values*.

As the running the code, the results for three policies are shown in Table III. The Value Iteration approach provides the average states' values which is superior to the *First-Visit MC Policy Evaluation* approach. However, as increasing the chances close to 1.0, the results of both approaches will be similar.



```
def value_iteration(world, num_iterations, gamma=0.98):
  # values is a vector containing the value for each state.
  values = world.rewards.copy()
  values_over_iterations = [values.copy()]
  for _ in range(num_iterations):
    # Perform one step of value iteration
    for state in world.states:
      actions = np.array(world.get_actions_for_state(state))
      v_max = values[state]
      for action in actions:
        p = world.get_transition_probs(state, action)
        v = 0
        for i in range(len(p)):
          if not (p[i] == 0):
            v += p[i]*(world.rewards[i] + values[i])
        if v > v_max:
          v_max = v
      values[state] = v_max
    # Store the values in the 'all_values' list
    values_over_iterations.append(values.copy())
  # Return the all_values list;
  return values_over_iterations
```

Fig. 5: The value_iteration function code



```
def compute_policy_from_values(world, values):
  """policy is a mapping from states -> actions.
  Here, it's just a vector: action = policy[state]"""
  # Initialize the policy vector
  policy = np.zeros_like(world.states)
  # Compute the policy for every state
  for state in world.states:
    r = world.rewards.copy()
    actions = np.array(world.get_actions_for_state(state))
    # print(actions)
    select_action = actions[0]
    v_max = r[select_action] + values[select_action]
    for action in actions:
      if r[action] + values[action] > v_max:
        v_max = r[action] + values[action]
        select_action = action
    policy[state] = select_action
  return policy
```

Fig. 6: The compute_policy_from_values function code

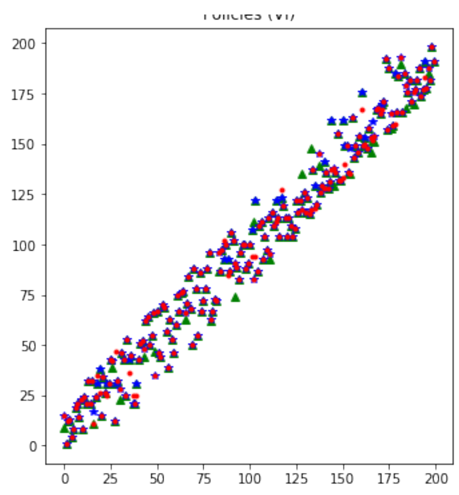


Fig. 7: The results of policy 1, policy 2, and Value Iteration Policy

The results the policy, value function and values vs iteration are then plotted in Fig. 7, 8 and 9. In Fig. 7, the blue, red, and green stars are not overlapped, it means the three policies

	Chances				
	0.0	0.2	0.5	0.8	1.0
Policy 1-avar. values	-49.94	-49.67	-49.23	-48.46	-47.56
Policy 2-avar. values	14.35	22.62	-13.93	-37.61	-47.56
Policy value-iter.	422.53	329.35	153.61	-10.39	-47.56

TABLE III: The chances vs Policies average values with linear algebra

provide the different results, and one of them is better than the others. The one with *zero* randomness in action selection gives the best average values for the states, and as we increases the randomness, the average values will reduce significantly.

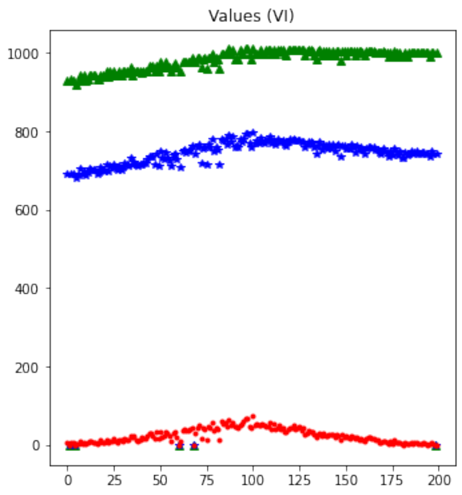


Fig. 8: The aver. values of the states from of policy 1, policy 2, and Value Iteration Policy

Fig. 9 shows the relationship between the number of iteration and the average value of states. This algorithm requires a lot of iteration to provide a good result.

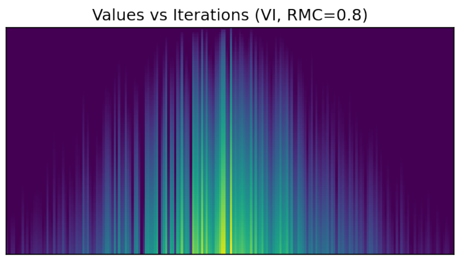
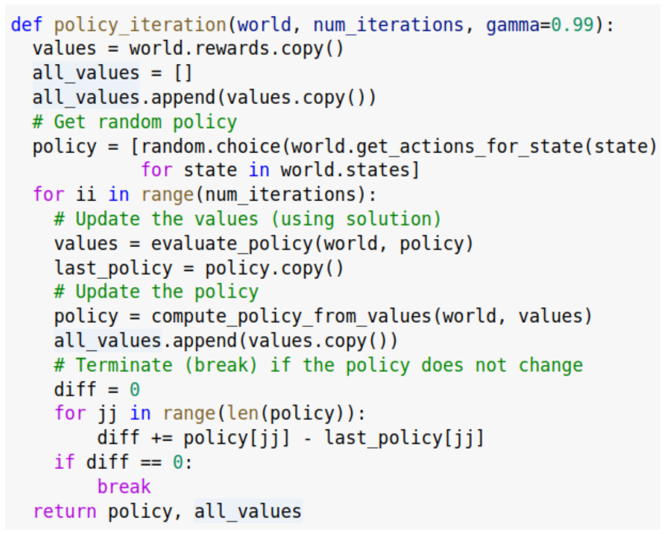


Fig. 9: Value Iteration vs number of Iteration

Question: There are a few states for which the value is very low for all three values of random move chance. Why is this the case?
Answer: In my opinion, the reason is the transition probabilities to those states from other states are too low. It means the states are rarely visited, and it makes the values of those states being very low.

III. Policy Iteration

The implementation of *policy_iteration* is shown in Fig. 10. In the *for* loop, we use *evaluate_policy* (linear algebra solution) to calculate the value function for all states. From the values, we will update the policy. The new policy will be compared with the last policy, and if there is not different, the *for* loop will be terminated.



```
def policy_iteration(world, num_iterations, gamma=0.99):
  values = world.rewards.copy()
  all_values = []
  all_values.append(values.copy())
  # Get random policy
  policy = [random.choice(world.get_actions_for_state(state)
          for state in world.states]
  for ii in range(num_iterations):
    # Update the values (using solution)
    values = evaluate_policy(world, policy)
    last_policy = policy.copy()
    # Update the policy
    policy = compute_policy_from_values(world, values)
    all_values.append(values.copy())
    # Terminate (break) if the policy does not change
    diff = 0
    for jj in range(len(policy)):
        diff += policy[jj] - last_policy[jj]
    if diff == 0:
        break
  return policy, all_values
```

Fig. 10: The code for Policy Iteration

The results of *Value Iteration* and *Policy Iteration* are then shown following.

- Value Iteration Time: 10.68
- Policy Iteration Time: 0.05
- Value Iteration Avg. Value: 154.97
- Policy Iteration Avg. Value: 155.75

The policy of the Policy Iteration algorithm is shown in Fig. 11. The figure is similar the transition probability figure of the environment. There is slightly difference between the policy of Value Iteration and the one of Policy Iteration.

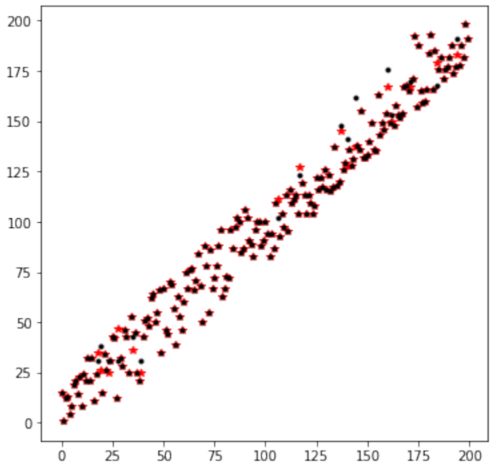


Fig. 11: The Policy of Policy Iteration algorithm

The Fig. 12 and 13 show the difference of average values of all states to the number of iteration. In Value Iteration (VI) algorithm, it needs 100 iterations to get the similar results with Policy Iteration (PI) one with only 5 iterations.

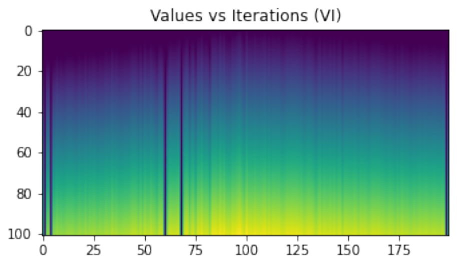


Fig. 12: The Average Values of VI algorithm

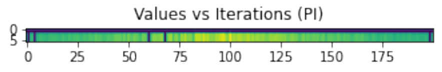


Fig. 13: The Average Values of PI algorithm

Question: Which approach converges more quickly? Why?
Answer: The average values of both approaches are similar, however, the computation times are significant difference. The *Value Iteration* needs 10.68 seconds, meanwhile the *Policy Iteration* is much faster, requiring only 0.05 second to get the same results. The reason is in *Value Iteration*, several *for* loops are used to calculate the value function all over the environment's states. On the other side, *Policy Iteration* process only one time by converting the problem in to linear algebra system (matrix and vector form), which can make use of linear solver of numpy.

IV. Q LEARNING

The code of the Q-update is shown in Fig. 14. In this code, we get the new sets of possible actions *new_actions* of the *new_state*, determine the best value function of the *new_state* among all the possible action in *new_actions*. The best value will be used to update the action-value function $Q(s, a)$.

```
# Update Q
new_actions = env.get_actions_for_state(new_state)
max_Q_s_prime_a = np.max(Q_s_a[new_state, new_actions])
Q_s_a[state, action] = (1-learning_rate)*Q_s_a[state, action] \
                + learning_rate*(r + gamma*max_Q_s_prime_a)
```

Fig. 14: The code of Q-learning algorithm

The average of the value function from Q-learning algorithm are listed following. As the learning rate increases from 0.0 to 0.02, the value function increase its values gradually. As the learning rate go over 0.02, the value function reduces significantly. Because as the learning rate is larger, amount of update for the action-value function will be huge, and make it jumping in the value space. As a consequence, it cannot gradually go to the optimal value. Comparing to the Policy Iteration, the value function of Q-learning need more iteration go get the same result.

- Policy Iteration Avg. Value: 279.69
- Q Learning Avg. Value (0.001): 180.65
- Q Learning Avg. Value (0.005): 248.74
- Q Learning Avg. Value (0.02): 260.45
- Q Learning Avg. Value (0.1): 228.84
- Q Learning Avg. Value (1.0): -43.14

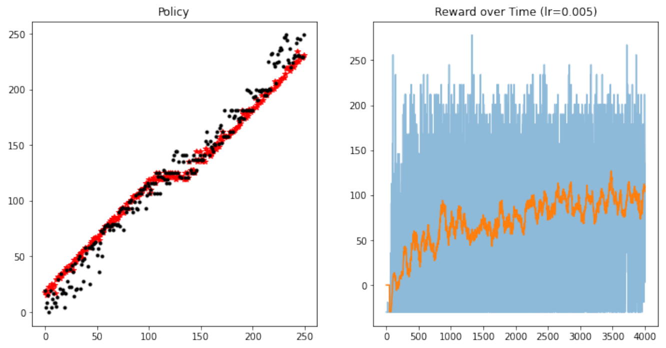


Fig. 15: The Policy and Reward as learning rate = 0.001

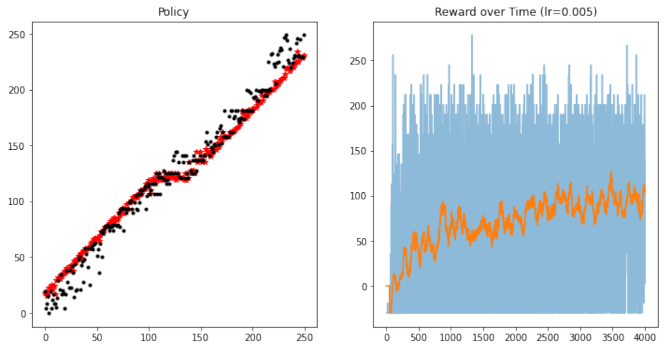


Fig. 16: The Policy and Reward as learning rate = 0.005

From Fig. 16 to Fig. 19 show the policy and the reward over time. The policies are similar with all the learning rate, however, the rewards are so different. The reward reaches maximum as *learning_rate* = 0.02, and reduce the *learning_rate* is too large. At *learning_rate* = 1.0, the rewards values are similar to random action selection.

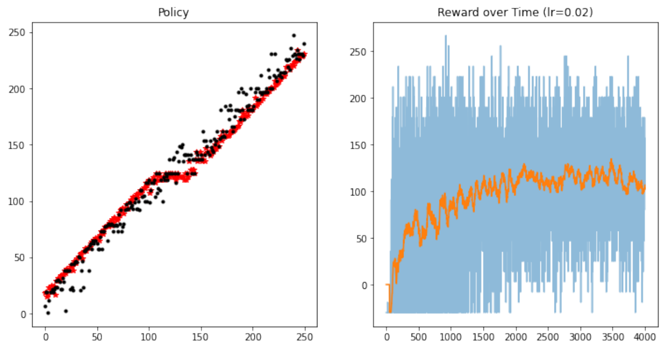


Fig. 17: The Policy and Reward as learning rate = 0.02

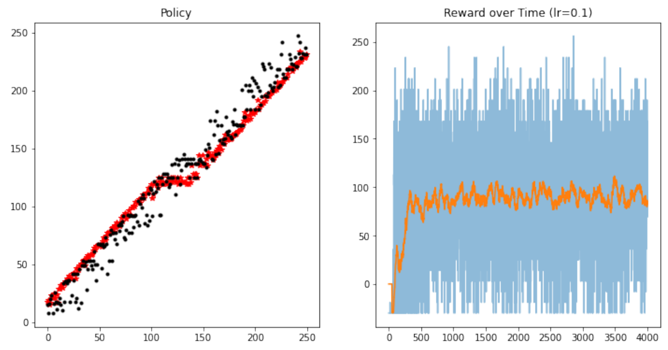


Fig. 18: The Policy and Reward as learning rate = 0.1

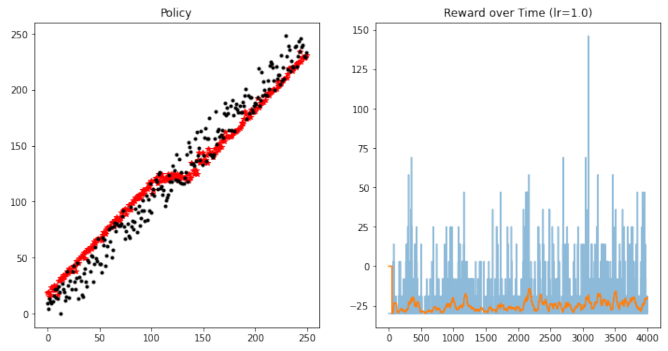


Fig. 19: The Policy and Reward as learning rate = 1.0

Question: The rate of convergence for Q learning is significantly slower than that of Value Iteration. What information does Value Iteration have access to (and indeed makes use of) that makes it converge faster?

Answer: The Value Iteration can access to the transition probability of the environment, that makes its convergence much faster than Q-learning. Q-learning does not know the transition probability function.

Question: When the learning rate is very low, the performance is not particularly good. From looking at the plots of the total reward over time, what is likely the cause? How would you fix this issue (without changing the learning rate)?

Answer: As learning rate is too low, the performance is not good. It is due to the value function and policy will be updated too slow and changed very little. It will require a lot iteration to make the policy convergence (This is the way to fix the small learning rate problem)

Question: When the learning is too high, the performance is also not very good. Why does this happen?

Answer: As the learning rate is too high, the performance is also bad. It is because the learning rate is high, the a mount of update for value function is huge and the function will be jumping in the value space and cannot converge to the optimal value.