CS 4180/5180: Reinforcement Learning and Sequential Decision Making (Fall 2021) Robert Platt
Northeastern University Due October 19, 2021

Exercise 4: Monte-Carlo Methods

1. **1 point.** (RL2e 5.9, 5.4) *Incremental implementation of Monte-Carlo methods.*
   Written:

   (a) Modify the algorithm for first-visit MC policy evaluation (Section 5.1) to use the incremental implementation for sample averages described in Section 2.4.

   First-visit MC prediction, for estimating $V \approx v_\pi$

   Input: a policy π to be evaluated
   Initialize:
   $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
   ~~$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$~~ $N(s) \leftarrow 0$

   Loop forever (for each episode):
   Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
   $G \leftarrow 0$
   Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
   $G \leftarrow \gamma G + R_{t+1}$
   Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
   ~~Append G to $Returns(S_t)$~~ $N(s) \leftarrow N(s) + 1$
   ~~$V(S_t) \leftarrow$ average$(Returns(S_t))$~~ $V(S_t) \leftarrow V(S_t) + \frac{1}{N(s)}(G - V(S_t))$

   (b) The pseudocode for Monte Carlo ES is inefficient because, for each state-action pair, it maintains a list of all returns and repeatedly calculates their mean. It would be more efficient to use techniques similar to those explained in Section 2.4 to maintain just the mean and a count (for each state–action pair) and update them incrementally. Describe how the pseudocode would be altered to achieve this.

   Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

   Initialize:
   $\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
   $Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
   ~~$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$~~ $N(s) \leftarrow 0$

   Loop forever (for each episode):
   Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
   Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
   $G \leftarrow 0$
   Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
   $G \leftarrow \gamma G + R_{t+1}$
   Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
   ~~Append G to $Returns(S_t, A_t)$~~ $N(s) \leftarrow N(s) + 1$
   ~~$Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$~~
   $\pi(S_t) \leftarrow \mathrm{argmax}_a\, Q(S_t, a)$ $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(s)}(G - Q(S_t, A_t))$

2. **1 point.** (RL2e 5.2, 5.5, 5.8) *First-visit vs. every-visit.*
Written:

(a) Suppose every-visit MC was used instead of first-visit MC on the blackjack task. Would you expect the results to be very different? Why or why not?

No. Blackjack does not contain two duplicate state in any episode, making first-visit and every-visit method essentially the same thing.

(b) Consider an MDP with a single nonterminal state and a single action that transitions back to the nonterminal state with probability p and transitions to the terminal state with probability $1-p$. Let the reward be $+1$ on all transitions, and let $\gamma = 1$. Suppose you observe one episode that lasts 10 steps, with a return of 10. What are the first-visit and every-visit estimators of the value of the nonterminal state?

$S_0, a_0, r_1, S_1, a_1, r_2, S_2, a_3, r_3, \ldots$

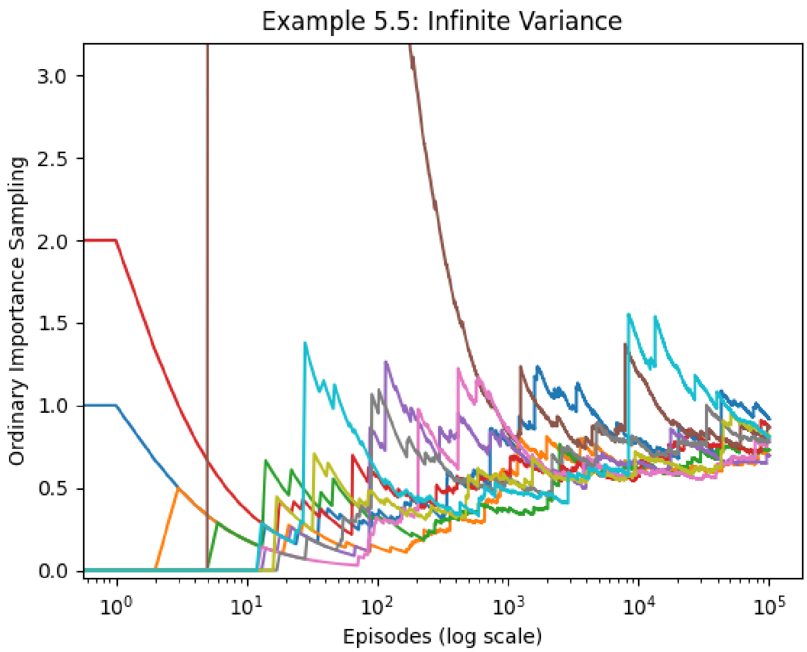
steps $= 10$, $\gamma = 1$

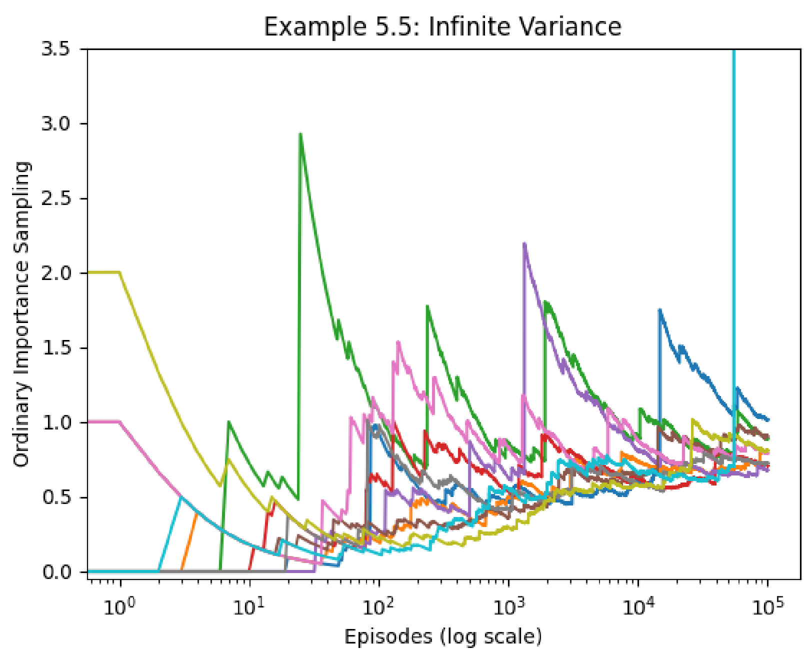
$$G_{10} = 0$$
$$G_9 = r_{10} + \gamma G_{10} = 1$$
$$G_8 = r_9 + \gamma G_9 = 2$$
$$\vdots$$

first-visit: $V(s) = 10$

all-visit $V(s) = \frac{1}{10}(1+2+3+4+5+6+7+8+9+10) = 5.5$

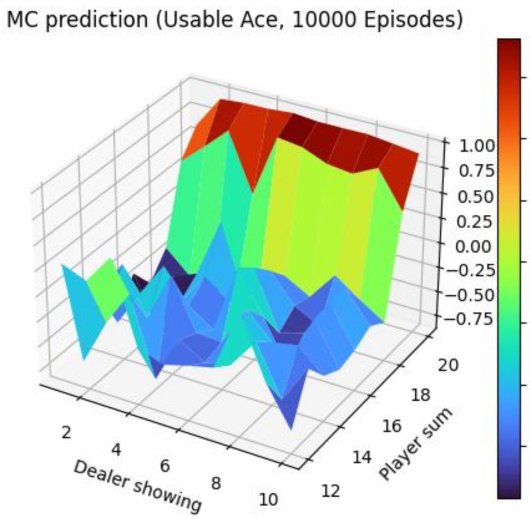
(c) **[Extra credit.]** Read and understand example 5.5 first. The results with Example 5.5 and shown in Figure 5.4 used a first-visit MC method. Suppose that instead an every-visit MC method was used on the same problem. Would the variance of the estimator still be infinite? Why or why not?
Code/plot: Implement Example 5.5 and reproduce Figure 5.4 to verify your answer.

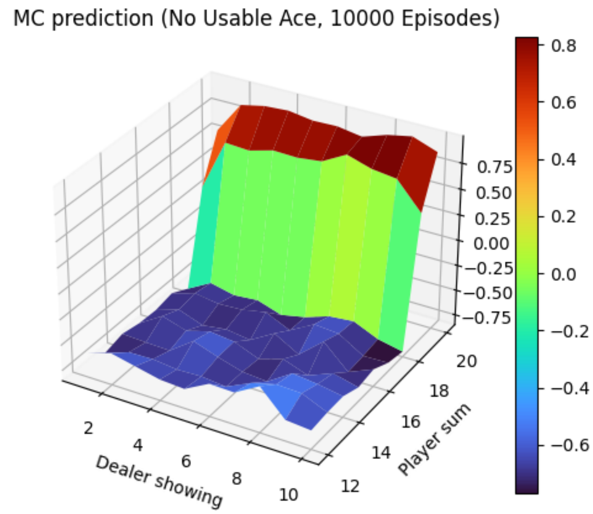


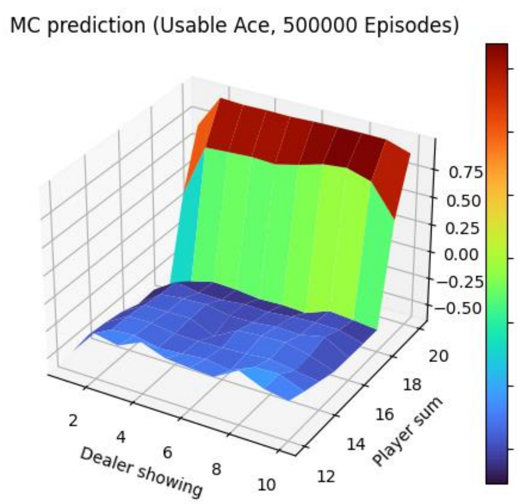


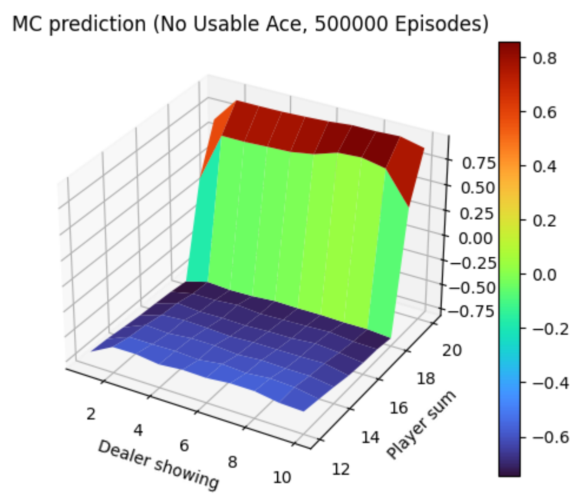
3. **2 points.** *Blackjack.*
Code/plot:

(a) Implement first-visit Monte-Carlo policy evaluation (prediction).
Apply it to the Blackjack environment for the "sticks only on 20 or 21" policy to reproduce Figure 5.1.

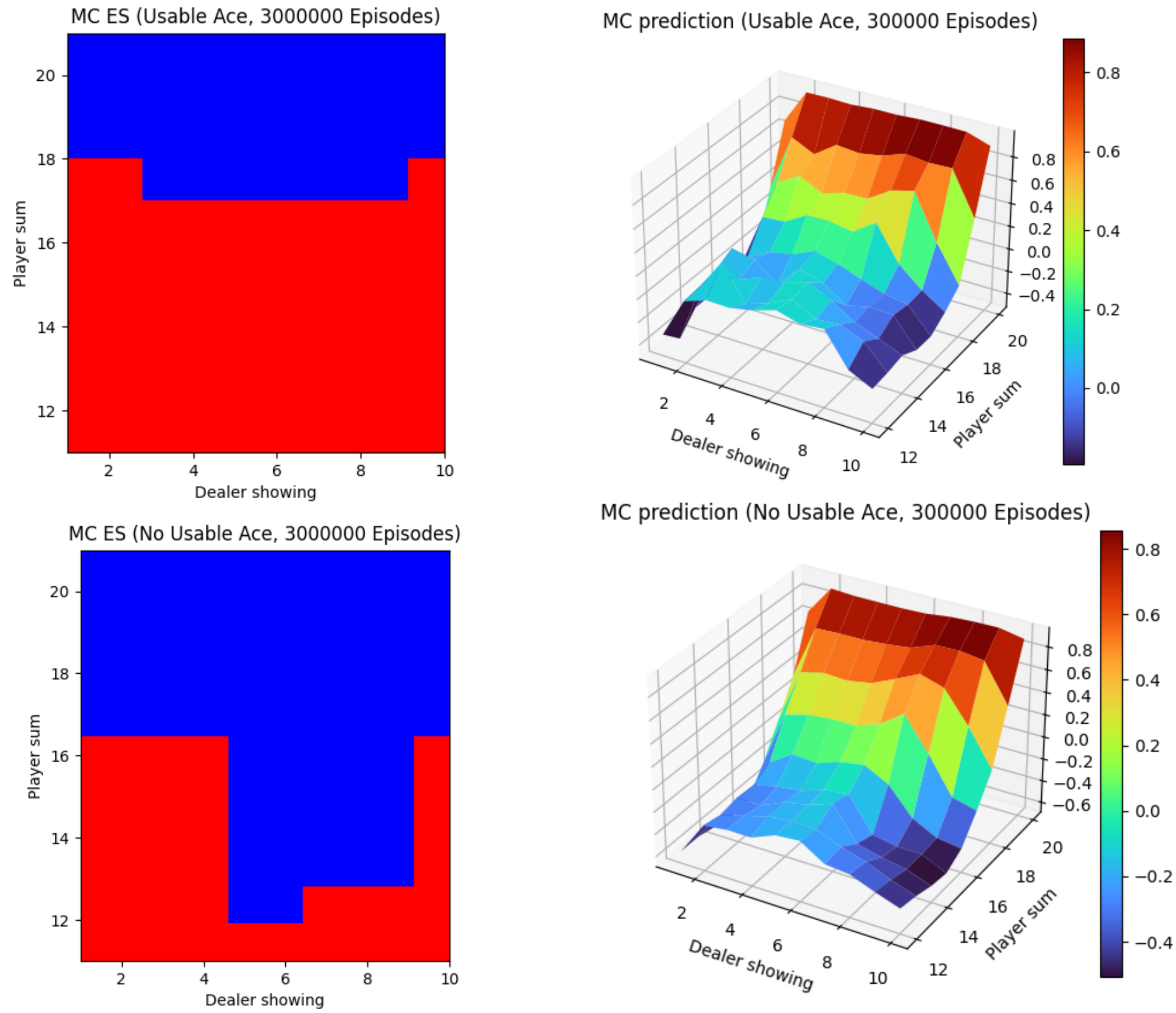








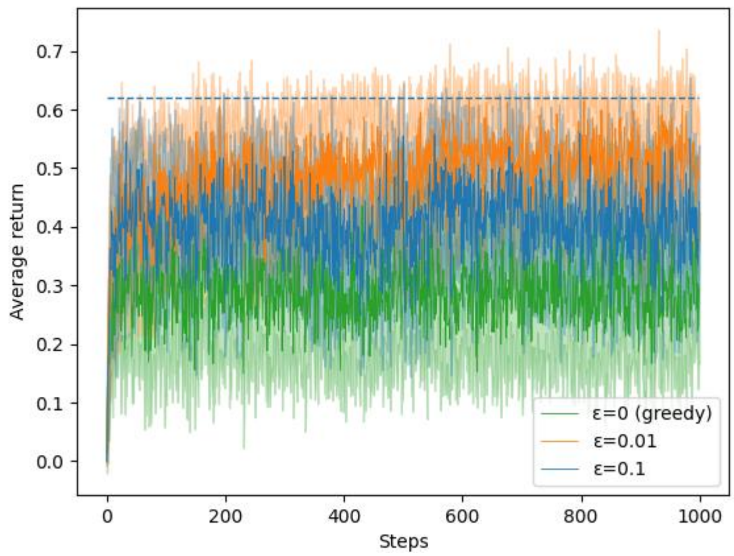
(b) Implement first-visit Monte-Carlo control with exploring starts (Monte-Carlo ES).
Apply it to the Blackjack environment to reproduce Figure 5.2.
Note that the reset mechanism already selects all states initially with probability > 0, but you must ensure that all actions are also selected with probability > 0.

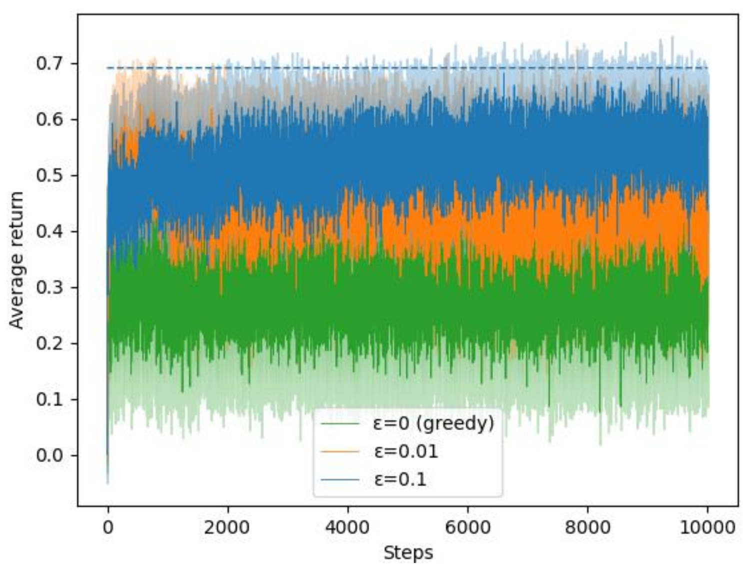


4. **2 points.** *Four Rooms, re-visited.*
We are now finally ready to re-visit the Four Rooms domain from Ex0, now with better learning algorithms. First, we must make the domain episodic to apply Monte-Carlo methods. Take your implementation from Ex0, but instead of teleporting to $(0, 0)$ after you reach the goal, make the goal a terminal state (i.e., end of episode). Also, to encourage reaching the goal faster, we will use a discount factor of $\gamma = 0.99$.
In addition, consider adding a timeout to your episodes, i.e., an episode terminates after some maximum number of steps. For example, a timeout of $T = 459$ may be reasonable, since this is the threshold for which $\gamma^T < 0.01$ (i.e., even if the agent reaches the goal after T, it will experience < 0.01 return).

   (a) <u>**Code:**</u> Implement on-policy first-visit Monte-Carlo control (for ε-soft policies).
   Apply it to the original goal state of $(10, 10)$, as well as some other randomly chosen goal states.
   (To choose a random goal state, select a random goal state before any trial, instead of using $(10, 10)$. Ensure that this goal state should remains fixed throughout all trials.)
   Verify that it learns to reach these (unknown) goals.

   (b) <u>**Code/plot:**</u> Let us focus on the $(10, 10)$ goal, which is initially unknown to the agent.
   To verify the agent is learning, plot learning curves similar to those in Ex1.





(c) **Written:** Explain how the results of the $\varepsilon = 0$ setting demonstrate the importance of doing exploring starts in Monte-Carlo ES.

At exploring start, the probability with running over all states have a result larger than 0, thus the greedy policy is improving from the original policy in stead of stuck on the first states with positive value forever.

5. **[CS 5180 only.] 1 point.** (RL2e 5.10, 5.11) *Off-policy methods.*
<u>**Written:**</u>

(a) Derive the weighted-average update rule (Equation 5.8) from (Equation 5.7). Follow the pattern of the derivation of the unweighted rule (Equation 2.3).

$$V_n \doteq \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}, \qquad n \geq 2, \tag{5.7}$$

$$V_{n+1} \doteq V_n + \frac{W_n}{C_n}\Big[G_n - V_n\Big], \qquad n \geq 1, \tag{5.8}$$

$$V_{n+1} \doteq \frac{\sum_{k=1}^{n} W_k G_k}{\sum_{k=1}^{n} W_k}$$

$$= \frac{W_n G_n + \sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k} \; \frac{\sum_{k=1}^{n-1} W_k}{\sum_{k=1}^{n} W_k}$$

$$= \left[\frac{W_n G_n}{C_{n-1}} + V_n\right] \frac{C_{n-1}}{C_n}$$

$$= \frac{W_n G_n}{C_n} + \frac{V_n C_{n-1}}{C_n}$$

$$= \frac{W_n G_n + V_n C_{n-1} + V_n C_n - V_n C_n}{C_n}$$

$$= V_n + \frac{W_n G_n + V_n (C_{n-1} - C_n)}{C_n}$$

$$= V_n + \frac{W_n G_n - V_n W_n}{C_n}$$

$$= V_n + \frac{W_n}{C_n}\left[G_n - V_n\right]$$

(b) In the boxed algorithm for off-policy MC control, you may have been expecting the W update to have involved the importance-sampling ratio $\frac{\pi(A_t|S_t)}{b(A_t|S_t)}$, but instead it involves $\frac{1}{b(A_t|S_t)}$. Why is this correct?

A_t is only allowed to change W if $A_t = \pi(S_t)$. And due to π is deterministic, we are safe to say $\pi(A_t|S_t) = 1$ during the update of W.

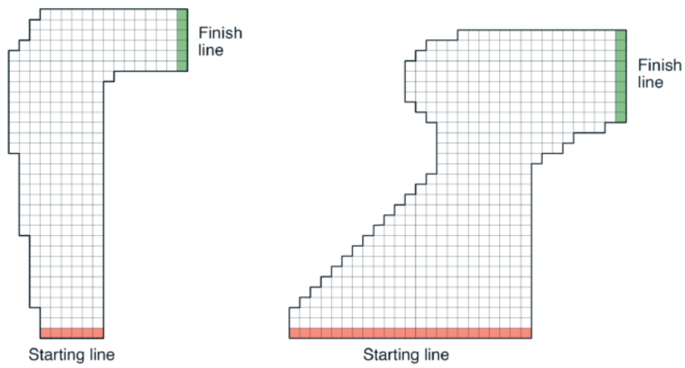


Figure 5.5: A couple of right turns for the racetrack task.

6. [**Extra credit.**] **2 points.** (RL2e 5.12) *Racetrack.*
Consider driving a race car around a turn like those shown in Figure 5.5. You want to go as fast as possible, but not so fast as to run off the track. In our simplified racetrack, the car is at one of a discrete set of grid positions, the cells in the diagram. The velocity is also discrete, a number of grid cells moved horizontally and vertically per time step. The actions are increments to the velocity components. Each may be changed by $+1$, -1, or 0 in each step, for a total of nine (3×3) actions. Both velocity components are restricted to be nonnegative and less than 5, and they cannot both be zero except at the starting line. Each episode begins in one of the randomly selected start states with both velocity components zero and ends when the car crosses the finish line. The rewards are -1 for each step until the car crosses the finish line. If the car hits the track boundary, it is moved back to a random position on the starting line, both velocity components are reduced to zero, and the episode continues. Before updating the car's location at each time step, check to see if the projected path of the car intersects the track boundary. If it intersects the finish line, the episode ends; if it intersects anywhere else, the car is considered to have hit the track boundary and is sent back to the starting line. To make the task more challenging, with probability 0.1 at each time step the velocity increments are both zero, independently of the intended increments.

(a) **Code:** Implement the racetrack domain (both tracks). Apply on-policy first-visit Monte-Carlo control (for ε-soft policies), with $\varepsilon = 0.1$ – ideally, this would be a simple application of the code from Q4(a).
Plot: For each racetrack, plot the learning curve (multiple trials with confidence bands), similar to Q4(b).

(b) **Code:** Implement off-policy Monte-Carlo control and apply it to the racetrack domain (both tracks). For the behavior policy, use an ε-greedy action selection method, based on the latest estimate of $Q(s, a)$ – i.e., this is similar to on-policy Monte-Carlo control, except that the target policy is kept as a greedy policy.
Plot: For each racetrack, plot the learning curve (multiple trials with confidence bands), similar to Q4(b). Show the performance of both the *behavior* and *target* policies; in the latter case, do this by collecting one rollout after each episode of training, which is collected solely for evaluation purposes. (The point of this is to inspect performance on the policy we actually care about, not the one used for data gathering.) Additionally, visualize several rollouts of the optimal policy; consider using: `matplotlib.pyplot.imshow`

(c) **Written:** Do you observe any significant differences between the on-policy and off-policy methods? Are there any interesting differences between the two racetracks?

Tip: You can find NumPy arrays containing the racetracks in `racetracks.py`.
Think about which racetrack you expect is easier, and develop your methods in that domain.