Chapter 5 Monte Carlo Methods

5.1 Monte Carlo Prediction

```
using StatsBase ✓
```

monte_carlo_pred (generic function with 2 methods)

```
function monte_carlo_pred(π::Dict{T, Vector{Float64}}, states::Vector{T}, actions,
simulator::Function, γ, nmax = 1000) where T
    avec = collect(actions)
    sample_π(s) = sample(avec, weights(π[s]))

    #initialize
    V = Dict(s => 0.0 for s in states)
    counts = Dict(s => 0 for s in states)
    for i in 1:nmax
        s0 = rand(states)
        a0 = sample_π(s0)
        (traj, rewards) = simulator(s0, a0, sample_π)

        #there's no check here so this is equivalent to every-visit estimation
        function updateV!(t = length(traj); g = 0.0)
            #terminate at the end of a trajectory
            t == 0 && return nothing
            #accumulate future discounted returns
            g = γ*g + rewards[t]
            (s,a) = traj[t]
            #increment count by 1
            counts[s] += 1
            V[s] += (g - V[s])/counts[s] #update running average of V
            updateV!(t-1, g = g)
        end

        #update value function for each trajectory
        updateV!()
    end
    return V
end
```

Example 5.1: Blackjack

```
const cards = ▸(2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, :A)
```

```julia
const cards = (2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, :A)
```

```
const blackjackactions = ▸(:hit, :stick)
```

```julia
const blackjackactions = (:hit, :stick)
```

```
deal (generic function with 1 method)
```

```julia
#deal a card from an infinite deck and return either the value of that card or an ace
deal() = rand(cards)
```

```
const blackjackstates =
▸[(12, 1, true), (12, 1, false), (12, 2, true), (12, 2, false), (12, 3, true), (12, 3, false
```

```julia
const blackjackstates = [(s, c, ua) for s in 12:21 for c in 1:10 for ua in (true,
false)]
```

```
addsum (generic function with 1 method)
```

```julia
#takes a previous sum, usable ace indicator, and a card to be added to the sum.
Returns the updated sum and whether an ace is still usable
function addsum(s::Int64, ua::Bool, c::Symbol)
    if !ua
        s >= 11 ? (s+1, false) : (s+11, true)
    else
        (s+1, true)
    end
end
```

```
addsum (generic function with 2 methods)
```

```julia
function addsum(s::Int64, ua::Bool, c::Int64)
    if !ua
        (s + c, false)
    else
        if (s + c) > 21
            (s + c - 10, false)
        else
            (s + c, true)
        end
    end
end
```

playersim (generic function with 2 methods)

```
function playersim(state, a, π::Function, traj = [(state, a)])
    (s, c, ua) = state
    a == :stick && return (s, traj)
    (s, ua) = addsum(s, ua, deal())
    (s >= 21) && return (s, traj)
    newstate = (s, c, ua)
    a = π(newstate)
    push!(traj, (newstate, a))
    playersim(newstate, a, π, traj)
end
```

dealer_sim (generic function with 1 method)

```
function dealer_sim(s::Int64, ua::Bool)
    (s >= 17) && return s
    (s, ua) = addsum(s, ua, deal())
    dealer_sim(s, ua)
end
```

blackjackepisode (generic function with 1 method)

```
#starting with an initial state, action, and policy, generate a trajectory for blackjack returning that and the reward
function blackjackepisode(s0, a0, π::Function)
    #score a game in which the player didn't go bust
    function scoregame(playersum, dealersum)
        #if the dealer goes bust, the player wins
        dealersum > 21 && return 1.0

        #if the player is closer to 21 the player wins
        playersum > dealersum && return 1.0

        #if the dealer sum is closer to 21 the player loses
        playersum < dealersum && return -1.0

        #otherwise the outcome is a draw
        return 0.0
    end

    (s, c, ua) = s0
    splayer, traj = playersim(s0, a0, π)
    rewardbase = zeros(length(traj) - 1)
    finalr = if splayer > 21
        #if the player goes bust, the game is lost regardless of the dealers actions
        -1.0
    else
        #generate hidden dealer card and final state
        hc = deal()
        (ds, dua) = if c == 1
            addsum(11, true, hc)
        else
            addsum(c, false, hc)
        end

        playernatural = (splayer == 21) && (length(traj) == 1)
        dealernatural = ds == 21

        if playernatural
            Float64(!dealernatural)
        else
            sdealer = dealer_sim(ds, dua)
            scoregame(splayer, sdealer)
        end
    end
    return (traj, [rewardbase; finalr])
end
```

```
const π_blackjack1 =
▶Dict((20, 8, false) ⇒ [0.0, 1.0], (16, 10, false) ⇒ [1.0, 0.0], (16, 2, false) ⇒ [1.0, (
```

```julia
#policy defined in Example 5.1
const π_blackjack1 = Dict((s, c, ua) => (s >= 20) ? [0.0, 1.0] : [1.0, 0.0] for (s, c, ua) in blackjackstates)
```

```
eval_blackjack_policy (generic function with 1 method)
```

```julia
#calculate value function for blackjack policy π and save results in plot-ready grid form
function eval_blackjack_policy(π, episodes; γ=1.)
    v_π = monte_carlo_pred(π, blackjackstates, blackjackactions, blackjackepisode, γ, episodes)
    vgridua = zeros(10, 10)
    vgridnua = zeros(10, 10)
    for state in blackjackstates
        (s, c, ua) = state
        if ua
            vgridua[s-11, c] = v_π[state]
        else
            vgridnua[s-11, c] = v_π[state]
        end
    end
    return vgridua, vgridnua
end
```

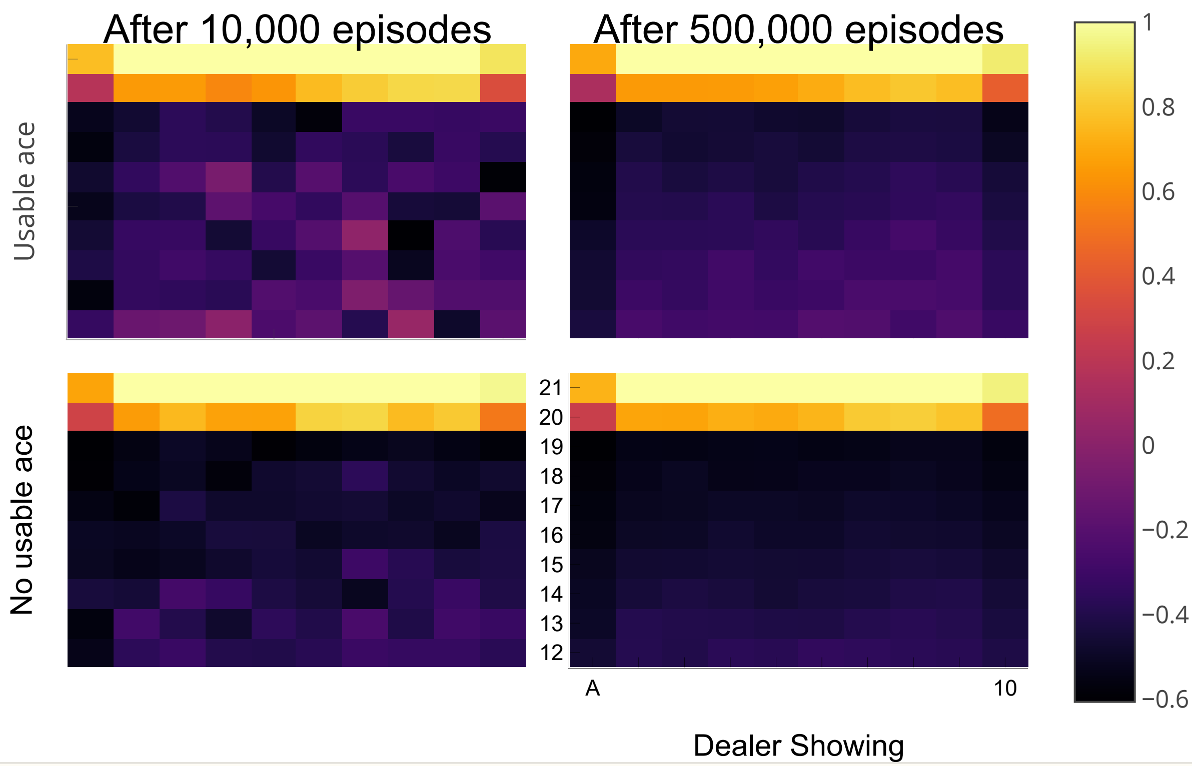
```
▶PlotlyBackend()
```

```julia
begin
    using Plots ✓
    plotly()
end
```

```
For saving to png with the Plotly backend PlotlyBase has to be installed.
```

```
plot_fig5_1 (generic function with 1 method)
```

```julia
function plot_fig5_1()
    (uagrid10k, nuagrid10k) = eval_blackjack_policy(π_blackjack1, 10_000)
    (uagrid500k, nuagrid500k) = eval_blackjack_policy(π_blackjack1, 500_000)

    p1 = heatmap(uagrid10k, title = "After 10,000 episodes", ylabel = "Usable ace", yticks = false, xticks = false)
    p2 = heatmap(nuagrid10k, ylabel = "No usable ace", yaxis = false, xaxis = false, legend = false)
    p3 = heatmap(uagrid500k, title = "After 500,000 episodes", yaxis = false, xaxis = false, legend = false)
    p4 = heatmap(nuagrid500k, yticks = (1:10, 12:21), xticks = (1:10, ["A", "", "", "", "", "", "", "", "", "10"]), legend = false, xlabel = "Dealer Showing")

    plot(p1, p3, p2, p4, layout = (2, 2))
end
```




```
plot_fig5_1()
```

> *Exercise 5.1* Consider the diagrams on the right in Figure 5.1. Why does the estimated value function jumpm for the last two rows in the rear? Why does it drop off for the whole last row on the left? Why are the frontmots values higher in the upper diagrams than in the lower?

The last two rows in the rear are for a player sum equal to 20 or 21. Per player policy, any sum less than this will result in a hit. Sticking on these sums is a good strategy and will likely result in a win, but the policy at 19 and lower is suboptimal.

The far left row represents cases where the dealer is showing an Ace. Since an Ace is a flexible card, the dealer policy will have more options that result in a win including the possibility of having another face card already. It is always a bad outcome for the player if the dealer is known to have an Ace.

The frontmost values represent cases where the player sum is 12. If there is a usable Ace this means that means that the player has two Aces which results in a sum of 12 when the first Ace is counted as 1 and the second is *usable* and counted as 11. If there is no usable Ace than a sum of 12 would have to result from some other combination of cards such as 10/2, 9/3, etc... Since the first case has two Aces, it means that potentially both could count as 1 if needed to avoid a bust. In the case without a usable Ace, the sum is the same, but there are more opportunities to bust if we draw a card worth 10, so having a sum of 12 with a usable Ace is strictly better.

> *Exercise 5.2* 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?

As an episode proceeds in blackjack the states will not repeat since every time a card is dealt the player sum changes or the usable Ace flag changes. Thus the check ensuring that only the first visit to a state is counted in the return average will have no effect on the MC evaluation.

5.2 Monte Carlo Estimation of Action Values

> *Exercise 5.3* What is the backup diagram for Monte Carlo estimation of q_π

Similar to the v_π diagram except the root is the s,a pair under consideration followed by the new state and the action taken along the trajectory. The rewards are still accumulated to the end, just the start of the trajectory is a solid filled in circle that would contain the value for that s,a pair.

5.3 Monte Carlo Control

```
monte_carlo_ES (generic function with 2 methods)
```

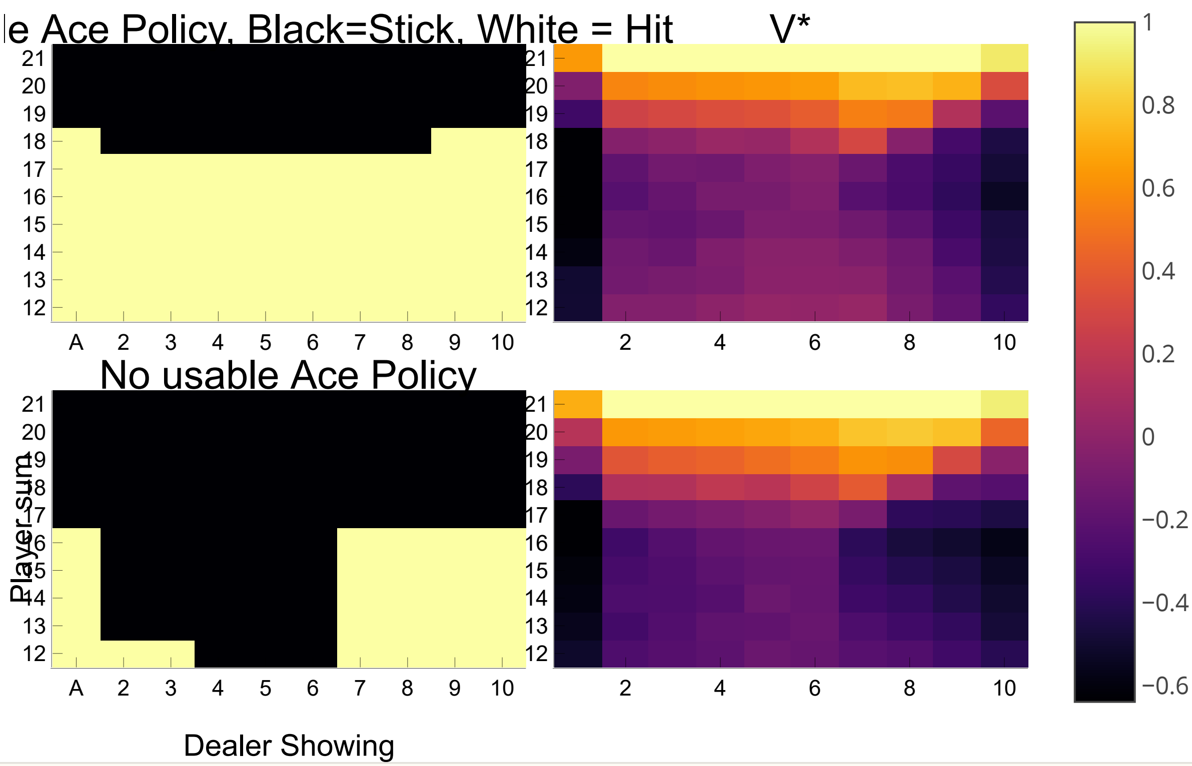
```
function monte_carlo_ES(states, actions, simulator, γ, nmax = 1000)
    #initialize
    π = Dict(s => rand(actions) for s in states)
    Q = Dict((s, a) => 0.0 for s in states for a in actions)
    counts = Dict((s, a) => 0 for s in states for a in actions)
    for i in 1:nmax
        s0 = rand(states)
        a0 = rand(actions)
        (traj, rewards) = simulator(s0, a0, s -> π[s])

        #there's no check here so this is equivalent to every-visit estimation
        t = length(traj)
        g = 0.0
        while t != 0
            g = γ*g + rewards[t]
            (s,a) = traj[t]
            counts[(s,a)] += 1
            Q[(s,a)] += (g - Q[(s,a)])/counts[(s,a)]
            π[s] = argmax(a -> Q[(s,a)], actions)
            t -= 1
        end
    end
    return π, Q
end
```

Example 5.3: Solving Blackjack

```
▸ (Dict((20, 8, false) ⇒ :stick, (16, 10, false) ⇒ :hit, (16, 2, false) ⇒ :stick, (19, 3,
```

```
(πstar_blackjack, Qstar_blackjack) = monte_carlo_ES(blackjackstates,
blackjackactions, blackjackepisode, 1.0, 10_000_000)
```

```
plot_blackjack_policy (generic function with 1 method)
```




```
#recreation of figure 5.2
plot_blackjack_policy(πstar_blackjack)
```

Exercise 5.4 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 dfficient 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.

Returns(s,a) will not maintain a list but instead be a list of single values for each state-action pair. Additionally, another list Counts(s,a) should be initialized at 0 for each pair. When new G values are obtained for state-action pairs, the Count(s,a) value should be incremented by 1. Then Returns(s,a) can be updated with the following formula:
$\text{Returns}(s,a) = [\text{Returns}(s,a) \times (\text{Count}(s,a) - 1) + G(s,a)]/\text{Count}(s,a)$

Alternatively, this can be written as:

$$\text{Returns}(s,a) = \text{Returns}(s,a) + \frac{G(s,a) - \text{Returns}(s,a)}{\text{Count}(s,a)}$$

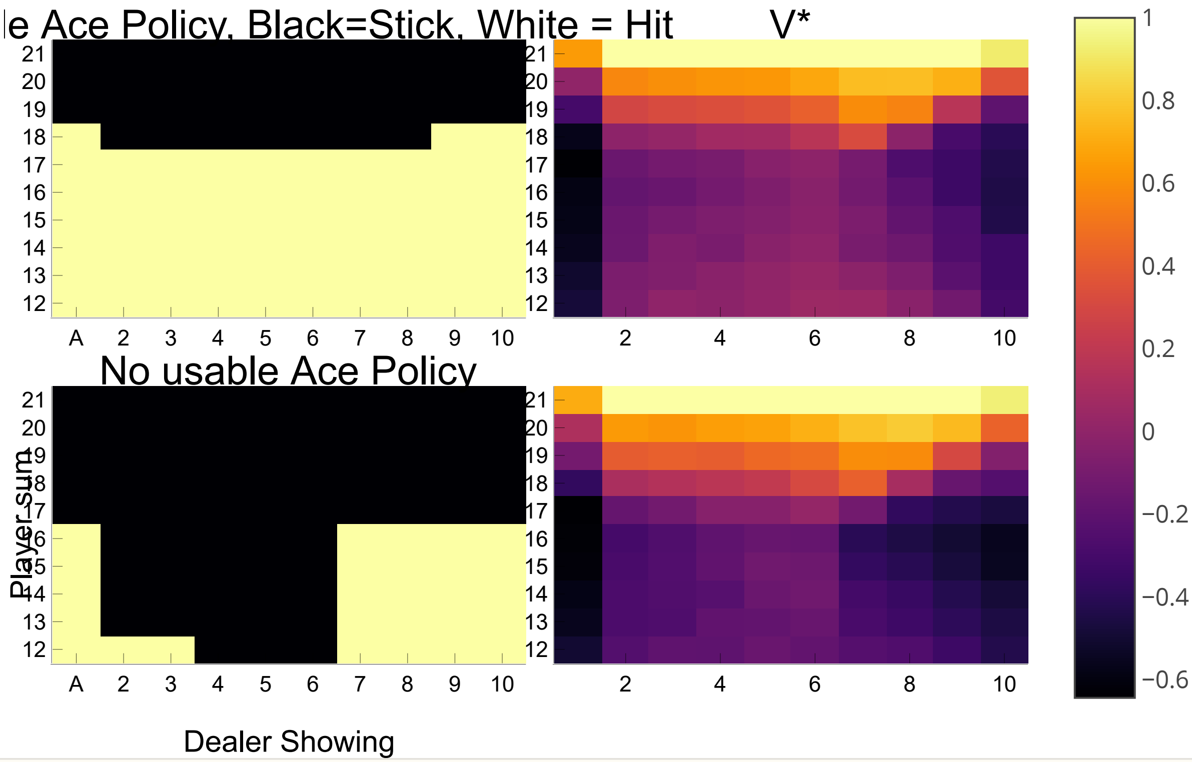
5.4 Monte Carlo Control without Exploring Starts

monte_carlo_ϵsoft (generic function with 2 methods)

```
function monte_carlo_ϵsoft(states, actions, simulator, γ, ϵ, nmax = 1000; gets0 = () -
> rand(states))
    #initialize
    nact = length(actions)
    avec = collect(actions)
    adict = Dict(a => i for (i, a) in enumerate(actions))
    π = Dict(s => ones(nact)./nact for s in states)
    Q = Dict((s, a) => 0.0 for s in states for a in actions)
    counts = Dict((s, a) => 0 for s in states for a in actions)
    sampleπ(s) = sample(avec, weights(π[s]))
    for i in 1:nmax
        s0 = gets0()
        a0 = sampleπ(s0)
        (traj, rewards) = simulator(s0, a0, sampleπ)

        #there's no check here so this is equivalent to every-visit estimation
        t = length(traj)
        g = 0.0
        while t != 0
            g = γ*g + rewards[t]
            (s,a) = traj[t]
            counts[(s,a)] += 1
            Q[(s,a)] += (g - Q[(s,a)])/counts[(s,a)]
            astar = argmax(a -> Q[(s,a)], actions)
            istar = adict[astar]
            π[s] .= ϵ/nact
            π[s][istar] += 1 - ϵ
            t -= 1
        end
    end
    π_det = Dict(s => actions[argmax(π[s])] for s in states)
    return π_det, Q
end
```

▶ (Dict((20, 8, false) ⇒ :stick, (16, 10, false) ⇒ :hit, (16, 2, false) ⇒ :stick, (19, 3,

```
(πstar_blackjack2, Qstar_blackjack2) = monte_carlo_ϵsoft(blackjackstates,
blackjackactions, blackjackepisode, 1.0, 0.05, 10_000_000)
```




```
#recreation of figure 5.2 using ϵ-soft method
plot_blackjack_policy(πstar_blackjack2)
```

5.5 Off-policy Prediction via Importance Sampling

Given a starting state S_t, the probability of the subsequent state-action trajectory, $A_t, S_{t+1}, A_{t+1}, \ldots, S_T$, occuring under any policy π is:

$$Pr_\pi\{traj\} = \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)$$

where p here is the state-transition probability function defined by (3.4). Thus, the relative probability of the trajectory under the target and behavior policies (the importance-sampling ratio) is

$$\rho_{t:T-1} \doteq \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

To estimate $v_\pi(s)$, we simply scale the returns by the ratios and average the results:

$$V(s) \doteq \frac{\sum_{t\in\mathscr{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathscr{T}(s)|}$$

When importance sampling is done as a simple average in this way it is called *ordinary importance sampling*.

An important alternative is *weighted importance sampling*, which uses a *weighted* average, defined as

$$V(s) \doteq \frac{\sum_{t\in\mathscr{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t\in\mathscr{T}(s)} \rho_{t:T(t)-1}}$$

, or zero is the denominator is zero.

Consider an implementation or ordinary importance sampling that updates $V(s)$ incrementally every time a G value is observed for that state. The equations should be similar to the incremental update rule previously derived for $V(s)$ without importance sampling.

Consider a sequence of returns $G_1, G_2, \dots, G_{n-1}$, all starting in the same state and each with a corresponding weight $W_i = \rho_{t_i:T(t_i)-1}$. We wish to form the estimate

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

and keep it up-to-date as we obtain a single additional return G_n. Observe that we can increment n by 1 to get an espression for V in terms of itself.

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

Using the original formula for V_n, we can make the following substitution:

$$\sum_{k=1}^{n-1} W_k G_k = (n-1) V_n$$

which results in

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

So, to calculate the value function, we can simply apply the following update rule after obtaining new values for W and G:

$$C \leftarrow C + 1$$

$$V \leftarrow V + \frac{WG - V}{C}$$

which looks very similar to the ordinary average update rule but with the weight multiplied by G. C just keeps a running total of the times the state was observed. Note that C needs to be updated even in the case where W is 0 which is not the case for weighted importance sampling. A similar inremental update rule is derived later for the weighted case as well as an algorithm for updating the action-value estimate using this method. Below are code examples for calculating the value estimates for both weighted and normal importance sampling using the incremental implementation.

```
#types allow dispatch for each sampling method based on different incremental update rules
abstract type ImportanceMethod end
```

```julia
struct Weighted <: ImportanceMethod end
```

```julia
struct Ordinary <: ImportanceMethod end
```

monte_carlo_pred (generic function with 4 methods)

```julia
function monte_carlo_pred(π_target, π_behavior, states, actions, simulator, γ, nmax = 1000; gets0 = () -> rand(states), historystate = states[1], samplemethod::ImportanceMethod = Ordinary())
    #initialize values and counts at 0
    V = Dict(s => 0.0 for s in states)
    Vhistory = zeros(nmax)
    counts = Dict(s => 0.0 for s in states)

    #maps actions to the index for the probability lookup
    adict = Dict(a => i for (i, a) in enumerate(actions))

    avec = collect(actions) #in case actions aren't a vector
    sample_b(s) = sample(avec, weights(π_behavior[s])) #samples probabilities defined in policy to generate actions

    #updates the denominator used in the value update.  For ordinary sampling, this is just the count of visits to that state.  For weighted sampling, this is the sum of all importance-sampling ratios at that state
    updatecounts!(::Ordinary, s, w) = counts[s] += 1.0
    updatecounts!(::Weighted, s, w) = counts[s] += w


    #updates the value estimates at a given state using the future discounted return and the importance-sampling ratio
    updatevalue!(::Ordinary, s, g, w) = V[s] += (w*g - V[s])/counts[s]
    updatevalue!(::Weighted, s, g, w) = V[s] += (g - V[s])*w/counts[s]

    for i in 1:nmax
        s0 = gets0()
        a0 = sample_b(s0)
        (traj, rewards) = simulator(s0, a0, sample_b)

        #there's no check here so this is equivalent to every-visit estimation
        function updateV!(t = length(traj); g = 0.0, w = 1.0)
            #terminate at the end of a trajectory
            t == 0 && return nothing
            (s,a) = traj[t]

            #since this is the value estimate, every action must update the importance-sampling weight before the update to that state is calculated.  In contrast, for the action-value estimate the weight is only the actions made after the current step are relevant to the weight
            w *= π_target[s][adict[a]] / π_behavior[s][adict[a]]

            updatecounts!(samplemethod, s, w)

            #terminate when w = 0 if the weighted sample method is being used.  under ordinary sampling, the updates to the count and value will still occur because the denominator will still increment
            (w == 0 && isa(samplemethod, Weighted)) && return nothing

            #update discounted future return from the current step
            g = γ*g + rewards[t]
```

```
            updatevalue!(samplemethod, s, g, w)

            #continue back through trajectory one step
            updateV!(t-1, g = g, w = w)
        end
        updateV!()
        Vhistory[i] = V[historystate] #save the value after iteration i for the specified state
    end
    return V, Vhistory
end
```

> *Exercise 5.5* 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?

For the first-visit estimator, we only consider the single future reward from the starting state which would be 10. There is nothing to average since we just have the single value of 10 for the episode.

For the every-visit estimator, we need to average together all 10 visits to the non-terminal state. For the first visit, the future reward is 10. For the second visit it is 9, third 8, and so forth. The final visit has a reward of 1, so the value estimate is the average of 10, 9, ..., 1 which is $\frac{(1+10)\times 5}{10} = \frac{55}{10} = 5.5$

Example 5.4: Off-policy Estimation of a Blackjack State Value

```
(
    1: [((13, 2, true), :hit), ((18, 2, true), :hit), ((14, 2, false), :hit)]
    2: [0.0, 0.0, -1.0]
)
```

```
blackjackepisode((13, 2, true), :hit, s -> sample(collect(blackjackactions), weights(π_blackjack1[s])))
```

```
([((13, 2, true), :hit), ((13, 2, false), :hit)], [0.0, -1.0])
```

```
blackjackepisode((13, 2, true), :hit, s -> rand(blackjackactions))
```

estimate_blackjack_state (generic function with 1 method)

```
function estimate_blackjack_state(n, π)
    avec = collect(blackjackactions)
    rewards = zeros(n)
    sampleπ(s) = sample(avec, weights(π[s]))
    s0 = (13, 2, true)
    a0 = sampleπ(s0)
    for i in 1:n
        ep = blackjackepisode(s0, a0, sampleπ)
        rewards[i] = ep[2][end]
    end

    return mean(rewards), var(rewards)
end
```

▶ (-0.26832, 0.887388)

```
#target policy state value estimate and variance, why is the mean squared error after 1 episode for weighted importance sampling less than the variance of the state values?  Also this value estimate does not match what it says in the book of -0.27726 so there might be something subtlely wrong with my simulator
estimate_blackjack_state(10_000_000, π_blackjack1)
```

const π_rand_blackjack =

▶ Dict((20, 8, false) ⇒ [0.5, 0.5], (16, 10, false) ⇒ [0.5, 0.5], (16, 2, false) ⇒ [0.5, (

```
const π_rand_blackjack = Dict(s => [0.5, 0.5] for s in blackjackstates)
```

▶ (-0.292786, 0.914276)

```
#behavior policy state value estimate and variance
estimate_blackjack_state(10_000_000, π_rand_blackjack)
```

v_offpol =

▶ (Dict((20, 8, false) ⇒ 0.0, (16, 10, false) ⇒ 0.0, (16, 2, false) ⇒ -0.663, (19, 3, true

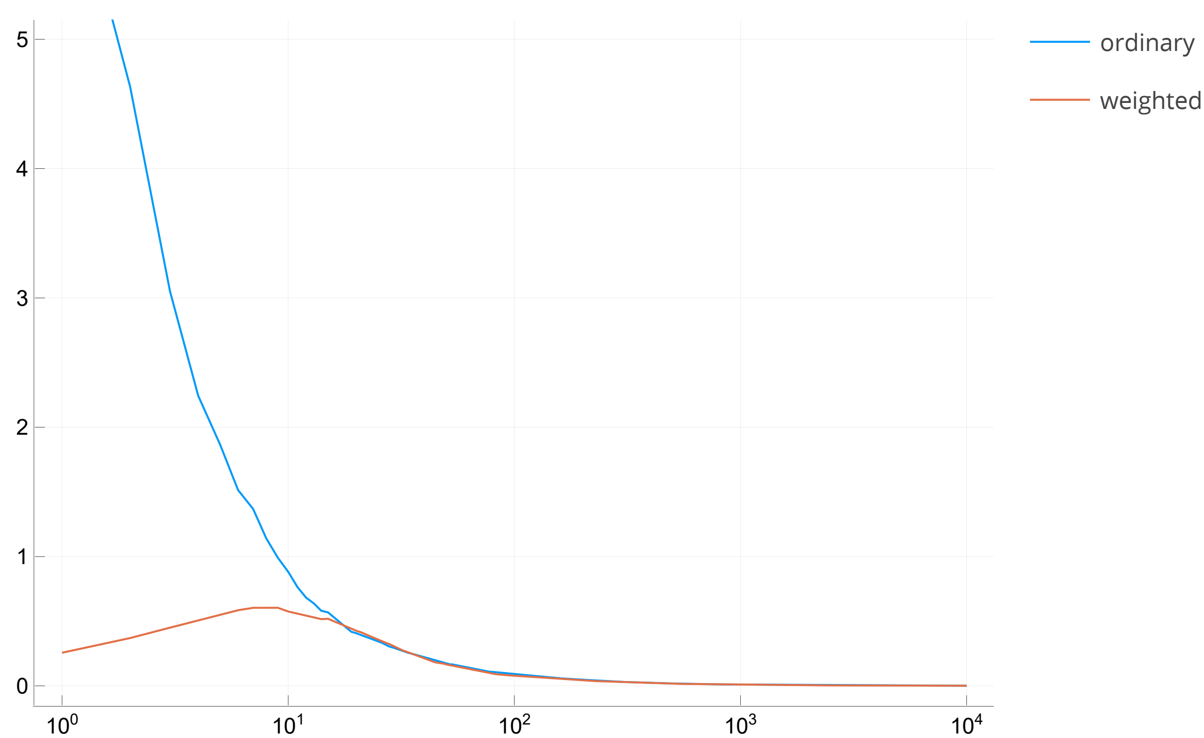
```
v_offpol = monte_carlo_pred(π_blackjack1, Dict(s => [0.5, 0.5] for s in blackjackstates), blackjackstates, blackjackactions, blackjackepisode, 1.0, 1_000_000, gets0 = () -> (13, 2, true))
```

-0.2713360000000126

```
v_offpol[1][(13, 2, true)]
```

figure5_3 (generic function with 2 methods)

```
function figure5_3(n = 100)
    s0 = (13, 2, true)
    gets0() = s0
    π_rand = Dict(s => [0.5, 0.5] for s in blackjackstates)
    vhist_ordinary = [(monte_carlo_pred(π_blackjack1, π_rand, blackjackstates,
    blackjackactions, blackjackepisode, 1.0, 10_000, gets0 = gets0, historystate = s0)
    [2] .+ 0.27726) .^2 for _ in 1:n]
    vhist_weighted = [(monte_carlo_pred(π_blackjack1, π_rand, blackjackstates,
    blackjackactions, blackjackepisode, 1.0, 10_000, gets0 = gets0, historystate =
    s0, samplemethod = Weighted())[2] .+ 0.27726) .^2 for _ in 1:n]
    plot(reduce((a, b) -> a .+ b, vhist_ordinary) ./ n, xaxis = :log, lab =
    "ordinary")
    plot!(reduce((a, b) -> a .+ b, vhist_weighted) ./ n, xaxis = :log, lab =
    "weighted", yaxis = [0, 5])
end
```




```
figure5_3(1000)
```

Example 5.5: Infinite Variance

const one_state_actions = ▶(:left, :right)

```
const one_state_actions = (:left, :right)
```

one_state_simulator (generic function with 1 method)

```
function one_state_simulator(s0, a0, π::Function)
    traj = [(s0,a0)]
    rewards = Vector{Float64}()
    function runsim(s, a)
        if a == :right
            push!(rewards, 0.0)
            return traj, rewards
        else
            t = rand()
            if t <= 0.1
                push!(rewards, 1.0)
                return traj, rewards
            else
                push!(rewards, 0.0)
                anew = π(s)
                push!(traj, (s, anew))
                runsim(s, anew)
            end
        end
    end
    runsim(s0, a0)
end
```

const onestate_π_target = ▸Dict(0 ⇒ [1.0, 0.0])

```
const onestate_π_target = Dict(0 => [1.0, 0.0])
```

const onestate_π_b = ▸Dict(0 ⇒ [0.5, 0.5])

```
const onestate_π_b = Dict(0 => [0.5, 0.5])
```

▸(Dict(0 ⇒ 0.424441), [3.0, 2.0, 1.0, 0.75, 0.666667, 0.461538, 0.428571, 12.381, 11.8182,

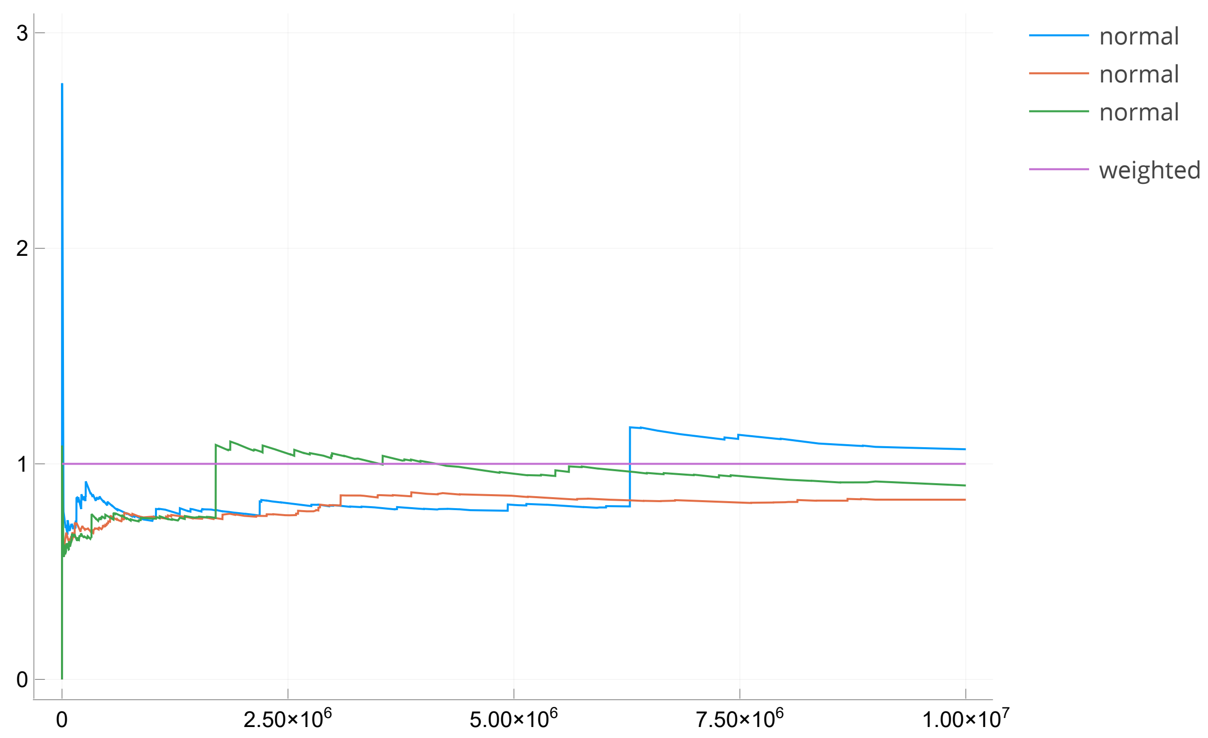
```
monte_carlo_pred(onestate_π_target, onestate_π_b, [0], one_state_actions,
one_state_simulator, 1.0, 1000, gets0 = () -> 0, historystate = 0, samplemethod =
Ordinary())
```

▸(Dict(0 ⇒ 1.0), [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, ⋯ more ,1.0])

```
monte_carlo_pred(onestate_π_target, onestate_π_b, [0], one_state_actions,
one_state_simulator, 1.0, 1000, gets0 = () -> 0, historystate = 0, samplemethod =
Weighted())
```

```
figure_5_4 (generic function with 1 method)
```

```
function figure_5_4(expmax, nsims)
    nmax = 10^expmax
    function makeplotinds(expmax)
        plotinds = mapreduce(i -> i:min(i, 1000):i*9, vcat, 10 .^(0:expmax-1))
        vcat(plotinds, 10^(expmax))
    end

    plotinds = makeplotinds(expmax)

    vhistnormal = [monte_carlo_pred(onestate_π_target, onestate_π_b, [0],
    one_state_actions, one_state_simulator, 1.0, nmax, gets0 = () -> 0, historystate
    = 0)[2][plotinds] for _ in 1:nsims]

    vhistweighted = monte_carlo_pred(onestate_π_target, onestate_π_b, [0],
    one_state_actions, one_state_simulator, 1.0, nmax, gets0 = () -> 0, historystate
    = 0, samplemethod = Weighted())[2][plotinds]


    plot(plotinds, vhistnormal, lab = "normal")
    plot!(plotinds, vhistweighted, lab = "weighted", yaxis = [0, 3])
end
```




```
figure_5_4(7, 3)
```

> *Exercise* 5.6 What is the equation analogous to (5.6) for *action* values $Q(s, a)$ instead of state values $V(s)$, again given returns generated using b?

Equation (5.6):

$$V(s) = \frac{\sum_{t \in \mathscr{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathscr{T}(s)} \rho_{t:T(t)-1}}$$

For $Q(s, a)$, there is no need to calculate the sampling ratio for the first action selected. This also assumes that the trajectory used for G and ρ has the first action being the one specified by $Q(s, a)$.

$$Q(s, a) = \frac{\sum_{t \in \mathscr{T}(s)} \rho_{t+1:T(t)-1} G_t}{\sum_{t \in \mathscr{T}(s)} \rho_{t+1:T(t)-1}}$$

> *Exercise* 5.7 In learning curves such as those shown in Figure 5.3 error generally decreases with training, as indeed happened for the ordinary importance-sampling method. But for the weighted importance-sampling method error first increased and then decreased. Why do you think this happened?

If the initial trajectories sampled are similar to ones we'd expect from the target policy, then the error will start low. Since the weighted method has bias which only converged to 0 with large episodes, we might expect to see the error rise as we sample trajectories which are less probable with the target policy. This bias will only dissappear as we add samples. In Figure 5.3, if the generator policy produces trajectories which are greater than 50% probable by the target policy, then early on our biased estimate will be low. But as we add more episodes the average will include more unlikely trajectories and push up the bias until the large numbers of samples has it convering back to 0 again.

Exercise 5.8 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?

Terms for each episode length are as follows:

Length 1 episode

$$\frac{1}{2} \cdot 0.1 \cdot 2^2$$

Length 2 episode, the term representing X is now an average of every visit along the trajectory. The probability of the trajectory is unchanged from before.

$$\frac{1}{2} \cdot 0.9 \cdot \frac{1}{2} \cdot 0.1 \left(\frac{2^2 + 2}{2} \right)^2$$

Length 3 episode

$$\frac{1}{2} \cdot 0.9 \cdot \frac{1}{2} \cdot 0.9 \cdot \frac{1}{2} \cdot 0.1 \left(\frac{2^3 + 2^2 + 2}{3} \right)^2$$

Length N episode

$$= 0.1 \left(\frac{1}{2} \right)^N 0.9^{N-1} \left(\frac{\sum_{i=1}^{N} 2^i}{N} \right)^2$$

So the expected value is the sum of these terms for every possible episode length

$$= 0.1 \sum_{k=1}^{\infty} \left(\frac{1}{2} \right)^k 0.9^{k-1} \left(\frac{\sum_{i=1}^{k} 2^i}{k} \right)^2$$

$$= 0.05 \sum_{k=1}^{\infty} .9^{k-1} \frac{1}{k^2} 2^{1-k} \left(\sum_{i=1}^{k} 2^i \right)^2$$

$$> 0.05 \sum_{k=1}^{\infty} .9^{k-1} \frac{1}{k^2} 2^{1-k} 2^{2k}$$

$$= 0.05 \sum_{k=1}^{\infty} .9^{k-1} \frac{1}{k^2} 2^{k+1}$$

$$= 0.2 \sum_{k=1}^{\infty} 1.8^{k-1} \frac{1}{k^2}$$

The expected value in question is greater than this expression, but as k approaches infinity, each term diverges so the expected value still diverges with every-visit MC.

5.6 Incremental Implementation

> *Exercise 5.9* 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

Returns(s) will not maintain a list but instead be a list of single values for each state. Additionally, another list Counts(s) should be initialized at 0 for each state. When new G values are obtained for state, the Count(s) value should be incremented by 1. Then Returns(s) can be updated with the following formula: $\text{Returns}(s) = [\text{Returns}(s) \times (\text{Count}(s) - 1) + G]/\text{Count}(s)$

Exercise 5.10 Derive the weighted-average update rule (5.8) from (5.7). Follow the pattern of the derivation of the unweighted rule (2.3).

Equation (5.7)

$$V_n = \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}$$

or

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

now we can expand the expresion for V_{n+1} to get an incremental rule

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

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

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

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

For a fully incremental rule we also have to replace the sum over W_k which can simply be a running total.

$$C_n = \sum_{k=1}^{n} W_k$$

the following update rule will produce an equivalent C_n assuming we take $C_0 = 0$

$$C_n = C_{n-1} + W_n$$

Now we can rewrite our last expression for V_{n+1}

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

```
monte_carlo_Q_pred (generic function with 2 methods)
```

```julia
function monte_carlo_Q_pred(π_target, π_behavior, states, actions, simulator, γ, nmax = 1000; gets0 = () -> rand(states))
    #initialize
    Q = Dict((s, a) => 0.0 for s in states for a in actions)
    counts = Dict((s, a) => 0.0 for s in states for a in actions)
    adict = Dict(a => i for (i, a) in enumerate(actions))
    avec = collect(actions)
    sample_b(s) = sample(avec, weights(π_behavior[s]))
    for i in 1:nmax
        s0 = gets0()
        a0 = sample_b(s0)
        (traj, rewards) = simulator(s0, a0, sample_b)

        #there's no check here so this is equivalent to every-visit estimation
        function updateQ!(t = length(traj); g = 0.0, w = 1.0)
            #terminate at the end of a trajectory or when w = 0
            ((t == 0) || (w == 0)) && return nothing
            #accumulate future discounted returns
            g = γ*g + rewards[t]
            (s,a) = traj[t]
            counts[(s, a)] += w
            Q[(s, a)] += (g - Q[(s, a)])*w/counts[(s, a)] #update running average of V
            w *= π_target[s][adict[a]] / π_behavior[s][adict[a]]
            updateQ!(t-1, g = g, w = w)
        end
        #update value function for each trajectory
        updateQ!()
    end
    return Q
end
```

```
q_offpol =
▸Dict(((14, 7, false), :hit) ⇒ 0.0, ((16, 7, false), :stick) ⇒ 0.0, ((19, 8, true), :stick
```

```julia
q_offpol = monte_carlo_Q_pred(π_blackjack1, Dict(s => [0.5, 0.5] for s in blackjackstates), blackjackstates, blackjackactions, blackjackepisode, 1.0, 10_000_000, gets0 = () -> (13, 2, true))
```

```
-0.2680117292661269
```

```julia
q_offpol[((13, 2, true), :hit)] #should converge to -0.27726 same as the value function for the policy that hits on this state
```

```
-0.2924541861803814
```

```julia
q_offpol[((13, 2, true), :stick)] #should be a lower value estimate because sticking is a worse action than hitting
```

```
▾Dict{Tuple{Int64, Symbol}, Float64}(
    ▸(0, :left) ⇒ 1.0
    ▸(0, :right) ⇒ 0.0
)
```

```julia
monte_carlo_Q_pred(onestate_π_target, onestate_π_b, [0], one_state_actions,
one_state_simulator, 1.0, 10000, gets0 = () -> 0)
```

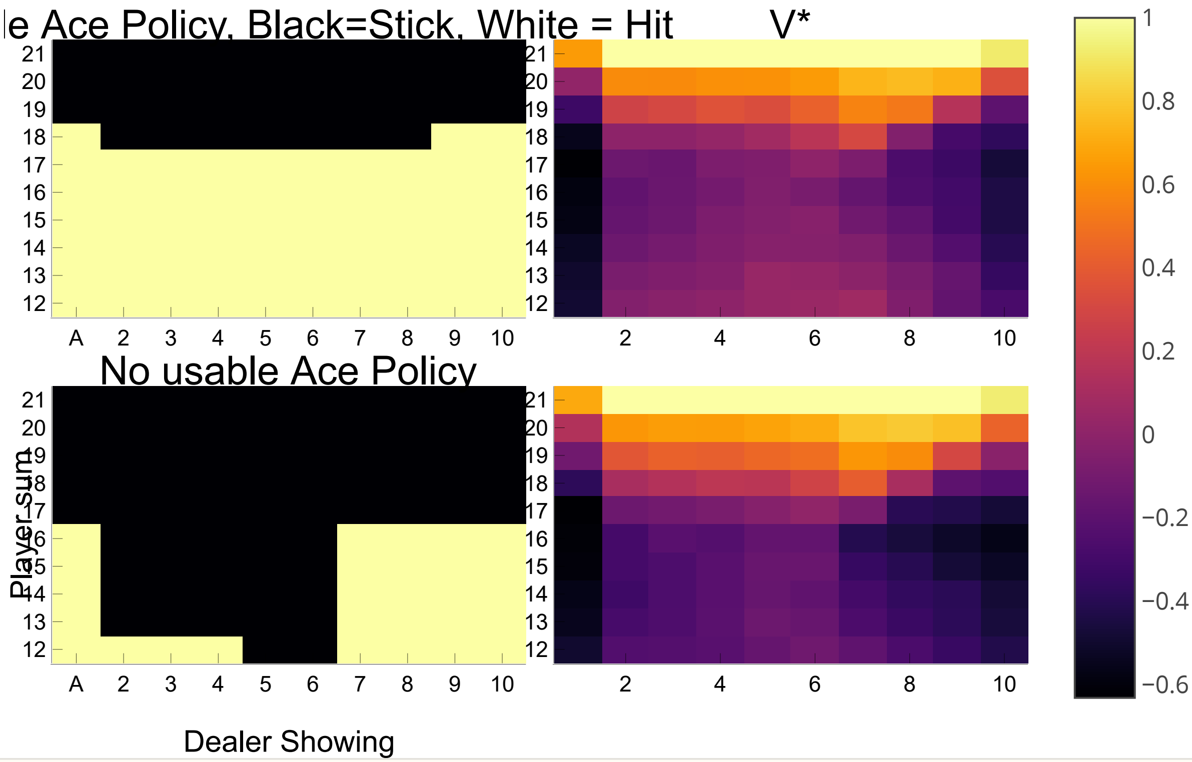
5.7 Off-policy Monte Carlo Control

```
off_policy_MC_control (generic function with 2 methods)
```

```julia
function off_policy_MC_control(states, actions, simulator, γ, nmax = 1000; gets0 = ()
-> rand(states))
    #initialize
    nact = length(actions)
    avec = collect(actions)
    π_b = Dict(s => ones(nact)./nact for s in states)
    Q = Dict((s, a) => 0.0 for s in states for a in actions)
    counts = Dict((s, a) => 0.0 for s in states for a in actions)
    adict = Dict(a => i for (i, a) in enumerate(actions))
    sample_b(s) = sample(avec, weights(π_b[s]))
    π_star = Dict(s => rand(actions) for s in states)
    for i in 1:nmax
        s0 = gets0()
        a0 = sample_b(s0)
        (traj, rewards) = simulator(s0, a0, sample_b)

        #there's no check here so this is equivalent to every-visit estimation
        function updatedicts!(t = length(traj); g = 0.0, w = 1.0)
            t == 0 && return nothing
            g = γ*g + rewards[t]
            (s,a) = traj[t]
            counts[(s,a)] += w
            Q[(s,a)] += (g - Q[(s,a)])*w/counts[(s,a)]
            astar = argmax(a -> Q[(s,a)], actions)
            π_star[s] = astar
            a != astar && return nothing
            w /= π_b[s][adict[a]]
            updatedicts!(t-1, g=g, w=w)
        end
        updatedicts!()
    end
    return π_star, Q
end
```

```
▸(Dict((20, 8, false) ⇒ :stick, (16, 10, false) ⇒ :hit, (16, 2, false) ⇒ :stick, (19, 3,
```

```julia
(πstar_blackjack3, Qstar_blackjack3) = off_policy_MC_control(blackjackstates,
blackjackactions, blackjackepisode, 1.0, 10_000_000)
```




```
#recreation of figure 5.2 using off-policy method
plot_blackjack_policy(πstar_blackjack3)
```

Exercise 5.11 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 nevertheless correct?

The target policy $\pi(s)$ is always deterministic, only selecting a single action according to $\pi(s) = \mathrm{argmax}_a Q(s, a)$. Therefore the numerator in importance-sampling ratio will either be 1 when the trajectory action matches the one given by $\pi(s)$ or it will be 0. The inner loop will exit if such as action is selected as it will result in zero values of W for the rest of the trajectory and thus no further updates to $Q(s, a)$ or $\pi(s)$. The only value of $\pi(s)$ that would be encountered in the equation is therefore 1 which is why the numerator is a constant.

Exercise 5.12: Racetrack (programming) 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 (3x3) 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 boundry, 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 probality 0.1 at each time step the velocity increments are both zero, independently of the intended increments. Apply a Monte Carlo control method to this task to compute the optimal policy from each starting state. Exhibit several trajectories following the optimal policy (but turn the noise off for these trajectories).

See code below to create racetrack environment

```
const racetrack_velocities =
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (2, 0), (2,
```

```
const racetrack_velocities = [(vx, vy) for vx in 0:4 for vy in 0:4]
```

```
const racetrack_actions =
[(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 0), (0, 1), (1, -1), (1, 0), (1, 1)]
```

```
const racetrack_actions = [(dx, dy) for dx in -1:1 for dy in -1:1]
```

```
project_path (generic function with 1 method)
```

```julia
#given a position, velocity, and action takes a forward step in time and returns the new position, new velocity, and a set of points that represent the space covered in between
function project_path(p, v, a)
    (vx, vy) = v
    (dx, dy) = a

    vxnew = clamp(vx + dx, 0, 4)
    vynew = clamp(vy + dy, 0, 4)

    #ensure that the updated velocities are not 0
    if vxnew + vynew == 0
        if iseven(p[1] + p[2])
            vxnew += 1
        else
            vynew += 1
        end
    end

    #position the car ends up at
    pnew = (p[1] + vxnew, p[2] + vynew)

    #how to check if the path intersects the finish line or the boundary?  Form a square from vxnew and vynew and see if the off-track area or finish line is contained in that square
    pathsquares = Set((x, y) for x in p[1]:pnew[1] for y in p[2]:pnew[2])

    (pnew, (vxnew, vynew), pathsquares)
end
```

```
const track1 =
(
    start = Set([(0, 0), (4, 0), (5, 0), (2, 0), (3, 0), (1, 0)])
    finish = Set([(13, 27), (13, 28), (13, 30), (13, 29), (13, 26), (13, 31)])
    body = Set([(1, 28), (-2, 10), (-1, 4), (2, 26), (5, 28), (-1, 22), (0, 17), (6, 29),
)
```

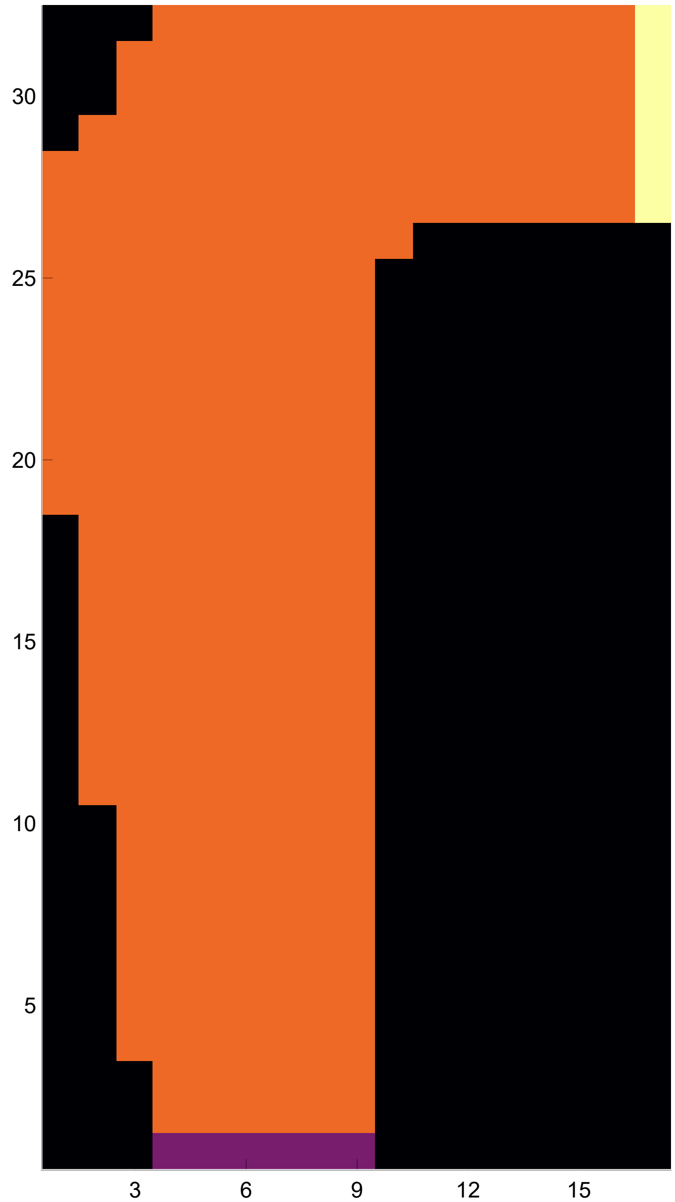
```julia
#track is defined as a set of points for each of the start, body, and finish
const track1 = (  start = Set((x, 0) for x in 0:5),
            finish = Set((13, y) for y in 26:31),
            body = union(   Set((x, y) for x in 0:5 for y in 1:2),
                            Set((x, y) for x in -1:5 for y in 3:9),
                            Set((x, y) for x in -2:5 for y in 10:17),
                            Set((x, y) for x in -3:5 for y in 18:24),
                            Set((x, 25) for x in -3:6),
                            Set((x, y) for x in -3:12 for y in 26:27),
                            Set((x, 28) for x in -2:12),
                            Set((x, y) for x in -1:12 for y in 29:30),
                            Set((x, 31) for x in 0:12))
        )
```

```
get_track_square (generic function with 1 method)
```

```julia
#convert a track into a grid for plotting purposes
function get_track_square(track)
    trackpoints = union(track...)
    xmin, xmax = extrema(p -> p[1], trackpoints)
    ymin, ymax = extrema(p -> p[2], trackpoints)

    w = xmax - xmin + 1
    l = ymax - ymin + 1

    trackgrid = Matrix{Int64}(undef, w, l)
    for x in 1:w for y in 1:l
            p = (x - 1 + xmin, y - 1 + ymin)
            val = if in(p, track.start)
                0
            elseif in(p, track.finish)
                2
            elseif in(p, track.body)
                1
            else
                -1
            end
            trackgrid[x, y] = val
    end end

    return trackgrid
end
```

```
const track1grid =
17×32 Matrix{Int64}:
 -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  …   1   1   1   1  1  1  -1  -1  -1  -1
 -1  -1  -1  -1  -1  -1  -1  -1  -1  -1   1       1   1   1   1  1  1   1  -1  -1  -1
 -1  -1  -1   1   1   1   1   1   1   1   1       1   1   1   1  1  1   1   1   1  -1
  0   1   1   1   1   1   1   1   1   1   1       1   1   1   1  1  1   1   1   1   1
  0   1   1   1   1   1   1   1   1   1   1       1   1   1   1  1  1   1   1   1   1
  0   1   1   1   1   1   1   1   1   1   1   …   1   1   1   1  1  1   1   1   1   1
  0   1   1   1   1   1   1   1   1   1   1       1   1   1   1  1  1   1   1   1   1
  ⋮                   ⋮                   ⋮   ⋱               ⋮                   ⋮
 -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1      -1  -1  -1  -1  1  1   1   1   1   1
 -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1      -1  -1  -1  -1  1  1   1   1   1   1
 -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1      -1  -1  -1  -1  1  1   1   1   1   1
 -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1      -1  -1  -1  -1  1  1   1   1   1   1
 -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  …  -1  -1  -1  -1  1  1   1   1   1   1
 -1  -1  -1  -1  -1  -1  -1  -1  -1  -1  -1      -1  -1  -1  -1  2  2   2   2   2   2
```

```julia
const track1grid = get_track_square(track1)
```




```
#visualization of first track in book with the starting line and finish line in
purple and yellow respectively.
heatmap(track1grid', legend = false, size = 20 .* (size(track1grid)))
```

race_track_episode (generic function with 1 method)

```julia
#starting in state s0 and with policy π, complete a single episode on given track returning the trajectory and rewards
function race_track_episode(s0, a0, π, track; maxsteps = Inf, failchance = 0.1)
	# @assert in(s0.position, track.start)
	# @assert s0.velocity == (0, 0)

	#take a forward step from current state returning new state and whether or not the episode is over
	function step(s, a)
		pnew, vnew, psquare = project_path(s.position, s.velocity, a)
		fsquares = intersect(psquare, track.finish)
		outsquares = setdiff(psquare, track.body, track.start)
		if !isempty(fsquares) #car finished race
			((position = first(fsquares), velocity = (0, 0)), true)
		elseif !isempty(outsquares) #car path went outside of track
			((position = rand(track1.start), velocity = (0, 0)), false)
		else
			((position = pnew, velocity = vnew), false)
		end
	end

	traj = [(s0, a0)]
	rewards = Vector{Float64}()

	function get_traj(s, a, nstep = 1)
		(snew, isdone) = step(s, a)
		push!(rewards, -1.0)
		while !isdone && (nstep < maxsteps)
			anew = π(snew)
			push!(traj, (snew, anew))
			(snew, isdone) = step(snew, rand() > failchance ? anew : (0, 0))
			push!(rewards, -1.0)
			nstep += 1
		end
	end

	isdone = get_traj(s0, a0)

	return traj, rewards
end
```

π_racetrack_rand (generic function with 1 method)

```julia
π_racetrack_rand(s) = rand(racetrack_actions)
```

```
race_episode =
(
    1: Tuple{NamedTuple{(:position, :velocity), Tuple{Tuple{Int64, Int64}, Tuple{Int64,
          1: ((position = (3, 0), velocity = (0, 0)), (1, -1))
          2: ((position = (4, 0), velocity = (1, 0)), (-1, -1))
          3: ((position = (5, 0), velocity = (1, 0)), (1, -1))
          4: ((position = (4, 0), velocity = (0, 0)), (0, 0))
          5: ((position = (5, 0), velocity = (1, 0)), (0, -1))
          6: ((position = (3, 0), velocity = (0, 0)), (-1, -1))
          7: ((position = (3, 1), velocity = (0, 1)), (1, 0))
          8: ((position = (4, 2), velocity = (1, 1)), (-1, 0))
          9: ((position = (4, 3), velocity = (0, 1)), (0, -1))
          ⋮ more
          2587: ((position = (11, 29), velocity = (2, 0)), (0, 1))
       ]
    2: [-1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, ⋯ more ,-1.0]
)
```

```julia
race_episode = race_track_episode((position = rand(track1.start), velocity = (0, 0)),
rand(racetrack_actions), π_racetrack_rand, track1)
```

```julia
using BenchmarkTools
```

```
runrace (generic function with 2 methods)
```

```julia
#run a single race episode from a valid starting position with a given policy and
track
function runrace(π, track = track1)
    s0 = (position = rand(track.start), velocity = (0, 0))
    a0 = π(s0)
    race_track_episode(s0, a0, π, track, maxsteps = 100000, failchance = 0.0)
end
```

```
sampleracepolicy (generic function with 2 methods)
```

```julia
#run n episodes of a race and measure the statistics of the time required to finish
function sampleracepolicy(π, n = 1000)
    trajs = [runrace(π)[1] for _ in 1:n]
    ls = length.(trajs)
    extrema(ls), mean(ls), var(ls)
end
```

```
((12, 27618), 2793.05, 7.6959e6)
```

```julia
#using a random policy, the mean time to finish on track 1 is ~2800 steps.  The best
possible time when we get "lucky" with random decisions is ~12 steps with worst times
~15-30k steps
sampleracepolicy(s -> rand(racetrack_actions), 10_000)
```

```
const track1states =
▸[(position = (1, 28), velocity = (0, 0)), (position = (1, 28), velocity = (0, 1)), (positi
```

```julia
const track1states = [(position = p, velocity = v) for p in union(track1.start, track1.body) for v in racetrack_velocities]
```

```
▾(
    1: ▸Dict((position = (11, 26), velocity = (1, 1)) => (-1, -1), (position = (1, 27), v
    2: ▾Dict{Tuple{NamedTuple{(:position, :velocity), Tuple{Tuple{Int64, Int64}, Tuple{I
          ▸((position = (8, 28), velocity = (4, 0)), (0, 0)) => 0.0
          ▸((position = (5, 24), velocity = (0, 1)), (-1, -1)) => 0.0
          ▸((position = (0, 15), velocity = (4, 0)), (-1, 0)) => 0.0
          ▸((position = (0, 7), velocity = (4, 2)), (1, 0)) => 0.0
          ▸((position = (4, 9), velocity = (0, 0)), (0, -1)) => 0.0
          ▸((position = (6, 26), velocity = (4, 3)), (-1, 1)) => 0.0
          ▸((position = (0, 24), velocity = (1, 1)), (1, 0)) => 0.0
          ▸((position = (2, 0), velocity = (4, 3)), (0, 1)) => 0.0
          ▸((position = (7, 29), velocity = (0, 2)), (0, 0)) => 0.0
          ▸((position = (4, 1), velocity = (0, 2)), (1, 1)) => 0.0
          ⋮ more
       )
)
```

```julia
(πstar_racetrack1, Qstar_racetrack1) = off_policy_MC_control(track1states, racetrack_actions, (s, a, π) -> race_track_episode(s, a, π, track1), 1.0, 10_000)
```

```
▸(Tuple{NamedTuple{(:position, :velocity), Tuple{Tuple{Int64, Int64}, Tuple{Int64, Int64}}
```

```julia
#off policy control doesn't produce a policy that can finish the race.  A cutoff of 100k steps is used to ensure the system doesn't run forever
runrace(s -> πstar_racetrack1[s])
```

```
▸(Dict{NamedTuple{(:position, :velocity), Tuple{Tuple{Int64, Int64}, Tuple{Int64, Int64}}}
```

```julia
(πstar_racetrack2, Qstar_racetrack2) = monte_carlo_ES(track1states, racetrack_actions, (s, a, π) -> race_track_episode(s, a, π, track1), 1.0, 100_000)
```

```
▾(
    1: ▸[((position = (1, 0), velocity = (0, 0)), (0, 0)), ((position = (1, 1), velocity =
    2: ▸[-1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, ⋯ more ,-1.0]
)
```

```julia
#exploring starts on policy training also doesn't produce a policy that can finish the race
runrace(s -> πstar_racetrack2[s])
```

```
(
  1: Dict((position = (11, 26), velocity = (1, 1)) ⇒ (-1, -1), (position = (1, 27), v
  2: Dict(((position = (8, 28), velocity = (4, 0)), (0, 0)) ⇒ 0.0, ((position = (5, 2
)
```

```
(πstar_racetrack3, Qstar_racetrack3) = monte_carlo_εsoft(track1states,
racetrack_actions, (s, a, π) -> race_track_episode(s, a, π, track1), 1.0, 0.25,
10_000_000, gets0 = () -> (position = rand(track1.start), velocity = (0, 0)))
```

```
(
  1: Tuple{NamedTuple{(:position, :velocity), Tuple{Tuple{Int64, Int64}, Tuple{Int64,
       1: ((position = (2, 0), velocity = (0, 0)), (-1, 1))
       2: ((position = (2, 1), velocity = (0, 1)), (-1, 1))
       3: ((position = (2, 3), velocity = (0, 2)), (-1, 0))
       4: ((position = (2, 5), velocity = (0, 2)), (-1, 1))
       5: ((position = (2, 8), velocity = (0, 3)), (-1, 1))
       6: ((position = (2, 12), velocity = (0, 4)), (0, -1))
       7: ((position = (2, 15), velocity = (0, 3)), (0, 0))
       8: ((position = (2, 18), velocity = (0, 3)), (-1, -1))
       9: ((position = (2, 20), velocity = (0, 2)), (-1, 0))
       ⋮ more
       14: ((position = (11, 27), velocity = (3, 0)), (-1, -1))
     ]
  2: [-1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, ⋯ more ,-1.0]
)
```

```
runrace(s -> πstar_racetrack3[s])
```

```
((14, 39), 14.8015, 2.31273)
```

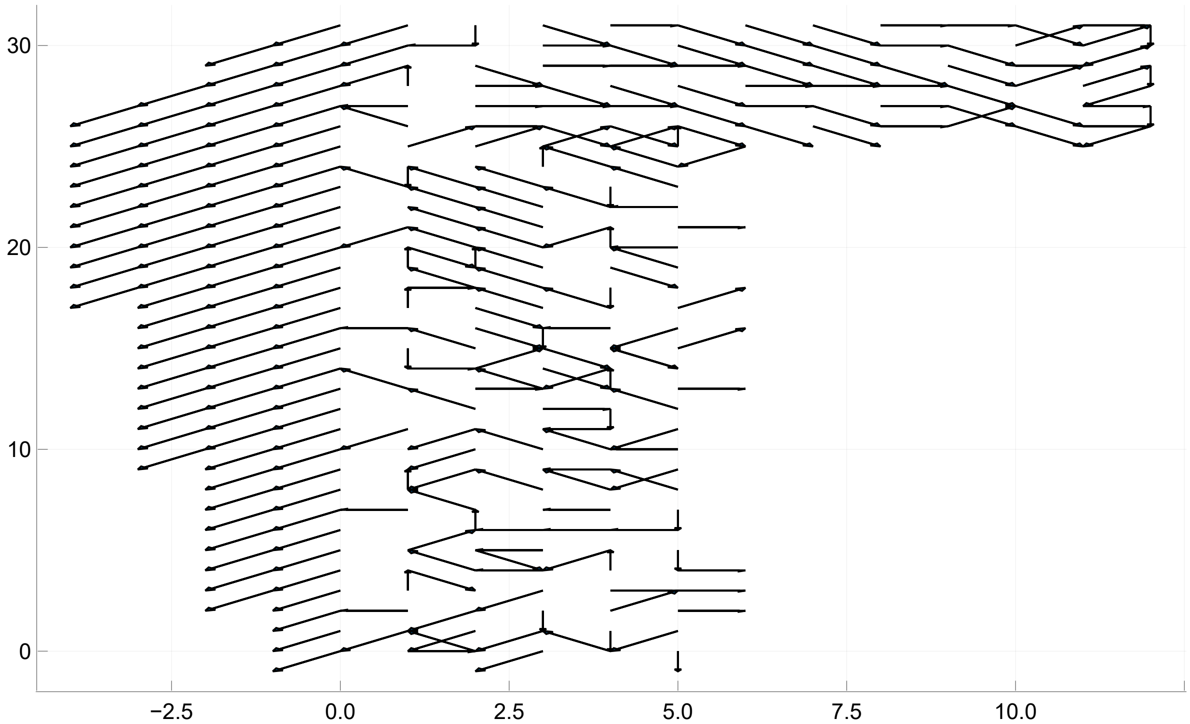
```
sampleracepolicy(s -> πstar_racetrack3[s], 10_000)
```

plotpolicy (generic function with 1 method)

```
function plotpolicy(π)
    x = [a[1] for a in union(track1...)]
    y = [a[2] for a in union(track1...)]
    dv = Dict(a => (0.0, 0.0) for a in union(track1...))
    v = Dict(a => (0.0, 0.0) for a in union(track1...))
    cv = Dict(a => 0 for a in union(track1...))
    for a in keys(π)
        (dx, dy) = π[a]
        (vx, vy) = a.velocity
        p = a.position
        cv[p] += 1
        dv[p] = ((dv[p][1] * (cv[p] - 1) + dx) / cv[p],  (dv[p][2] * (cv[p] - 1) +
        dy) / cv[p])
        v[p] = ((v[p][1] * (cv[p] - 1) + vx) / cv[p],  (v[p][2] * (cv[p] - 1) + vy) /
        cv[p])
    end
    dx = [dv[a][1] for a in zip(x, y)]
    dy = [dv[a][2] for a in zip(x, y)]
    vx = [v[a][1] for a in zip(x,y)]
    vy = [v[a][2] for a in zip(x,y)]
    quiver(x, y, quiver = (dx, dy))
end
```

plotpolicy2 (generic function with 1 method)

```
function plotpolicy2(π)
    positions = union(track1.start, track1.body)
    x = [a[1] for a in positions]
    y = [a[2] for a in positions]
    dv = Dict(a => (0.0, 0.0) for a in positions)
    cv = Dict(a => 0 for a in positions)
    for p in positions
        s = (position = p, velocity = (1, 0))
        (dx, dy) = π[s]
        dv[p] = (dx, dy)
    end
    dx = [dv[a][1] for a in zip(x, y)]
    dy = [dv[a][2] for a in zip(x, y)]
    quiver(x, y, quiver = (dx, dy))
end
```



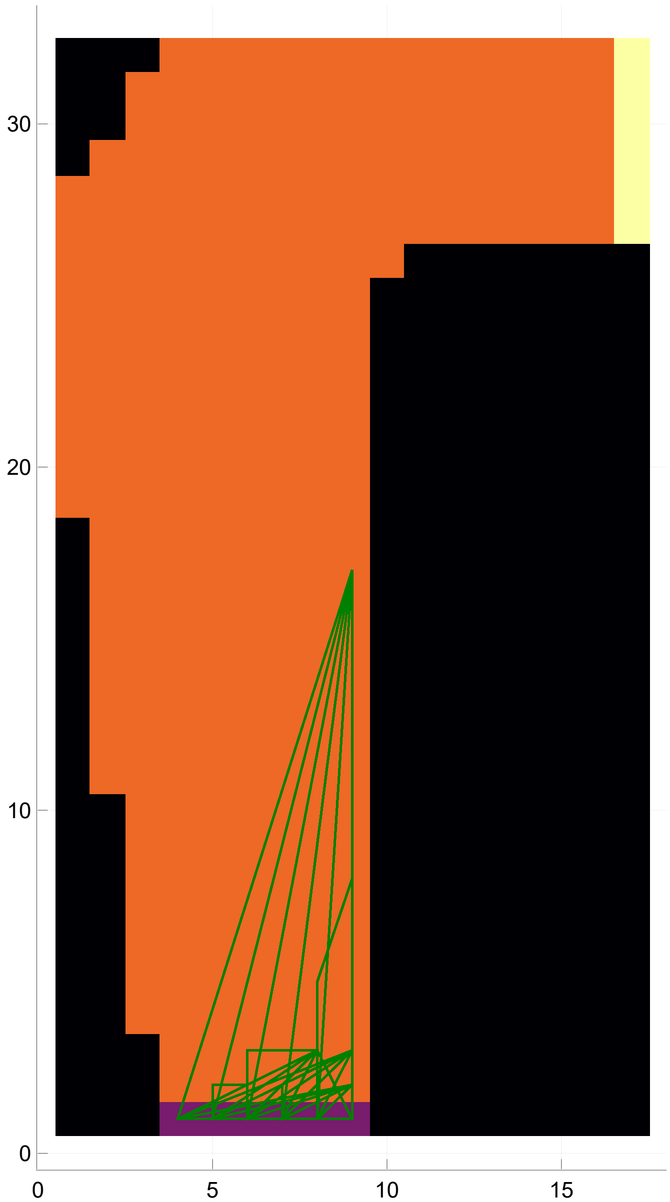

```
plotpolicy2(πstar_racetrack3)
```

visualize_policy_traj (generic function with 1 method)

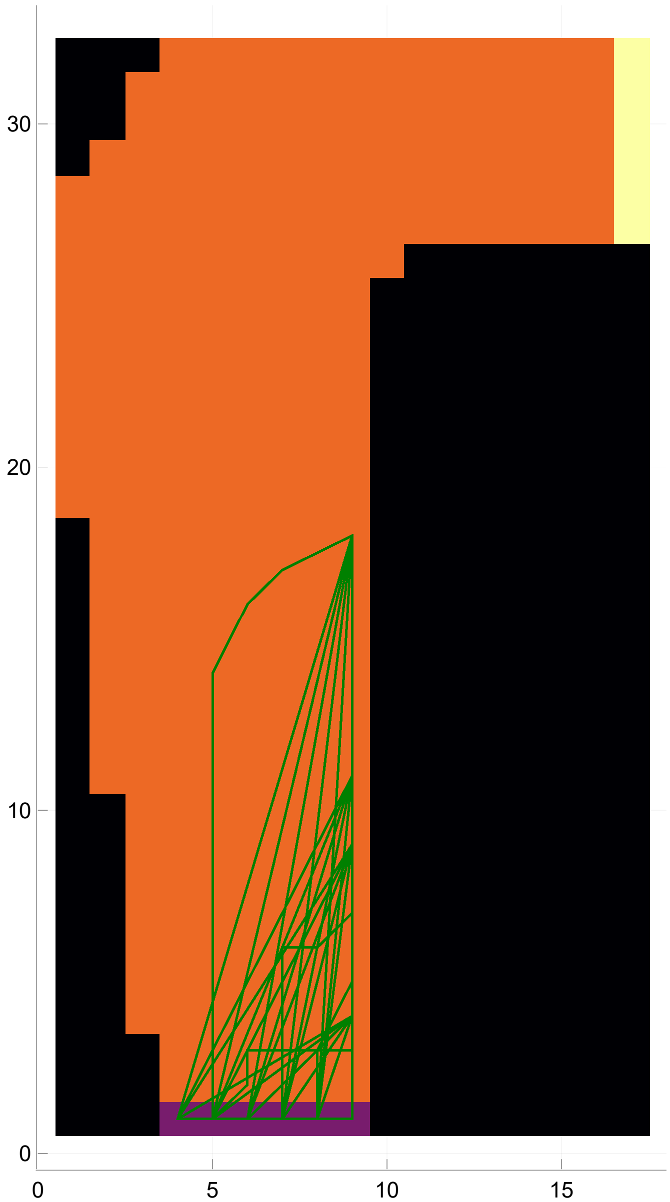
```
function visualize_policy_traj(π)
    fig = heatmap(track1grid', legend = false, size = 20 .* (size(track1grid)))
    for i in 0:4 #cycle through starting positions
        s0 = (position = (i, 0), velocity = (0, 0))
        a0 = π[s0]
        race_episode_star = race_track_episode(s0, a0, s -> π[s], track1, maxsteps =
        10000, failchance = 0.0)
        plot!([t[1].position .+ (4, 1) for t in race_episode_star[1]], color = :green)
    end
    plot(fig)
end
```

```
visualize_policy_traj2 (generic function with 1 method)
```

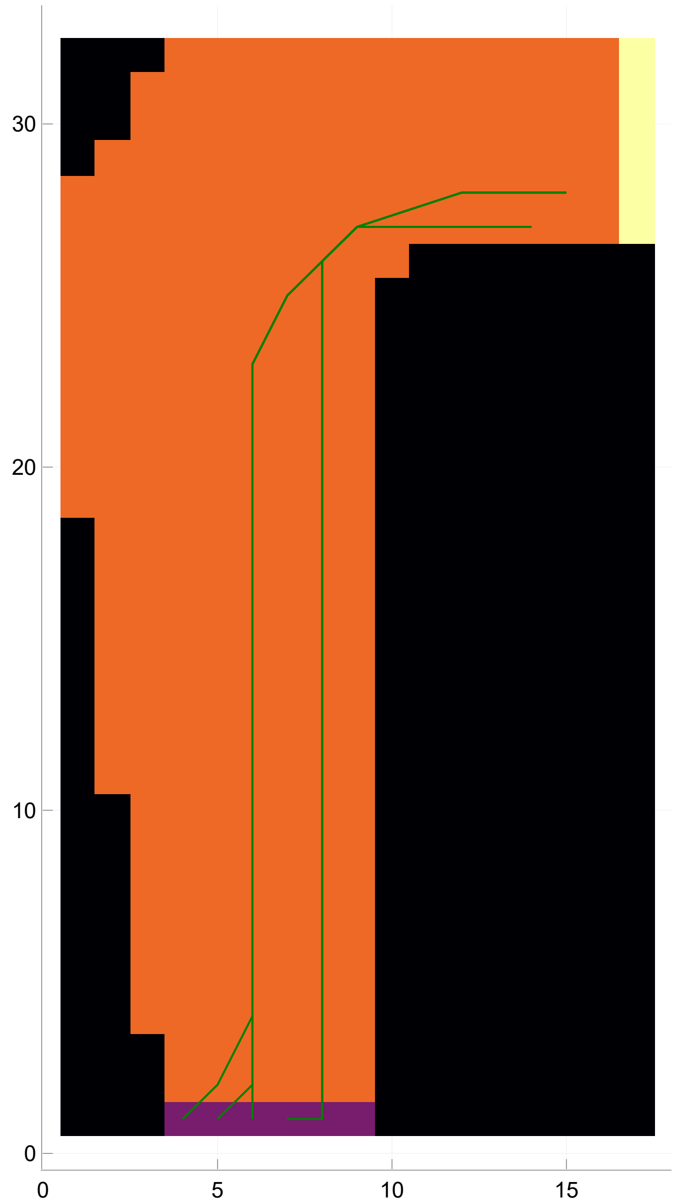
```julia
function visualize_policy_traj2(π)
    fig = heatmap(track1grid', legend = false, size = 20 .* (size(track1grid)))
    s0 = (position = (2, 0), velocity = (0, 0))
    a0 = π[s0]
    race_episode_star = race_track_episode(s0, a0, s -> π[s], track1, maxsteps =
    10000, failchance = 0.0)
    x = [t[1].position[1] + 4 for t in race_episode_star[1]]
    y = [t[1].position[2] + 1 for t in race_episode_star[1]]
    vx = [t[1].velocity[1] for t in race_episode_star[1]]
    vy = [t[1].velocity[2] for t in race_episode_star[1]]
    dx = [t[2][1] for t in race_episode_star[1]]
    dy = [t[2][2] for t in race_episode_star[1]]
    quiver!(x, y, quiver = (vx, vy))
    quiver!(x, y, quiver = (dx, dy), linecolor = :green)
    plot(fig)
end
```



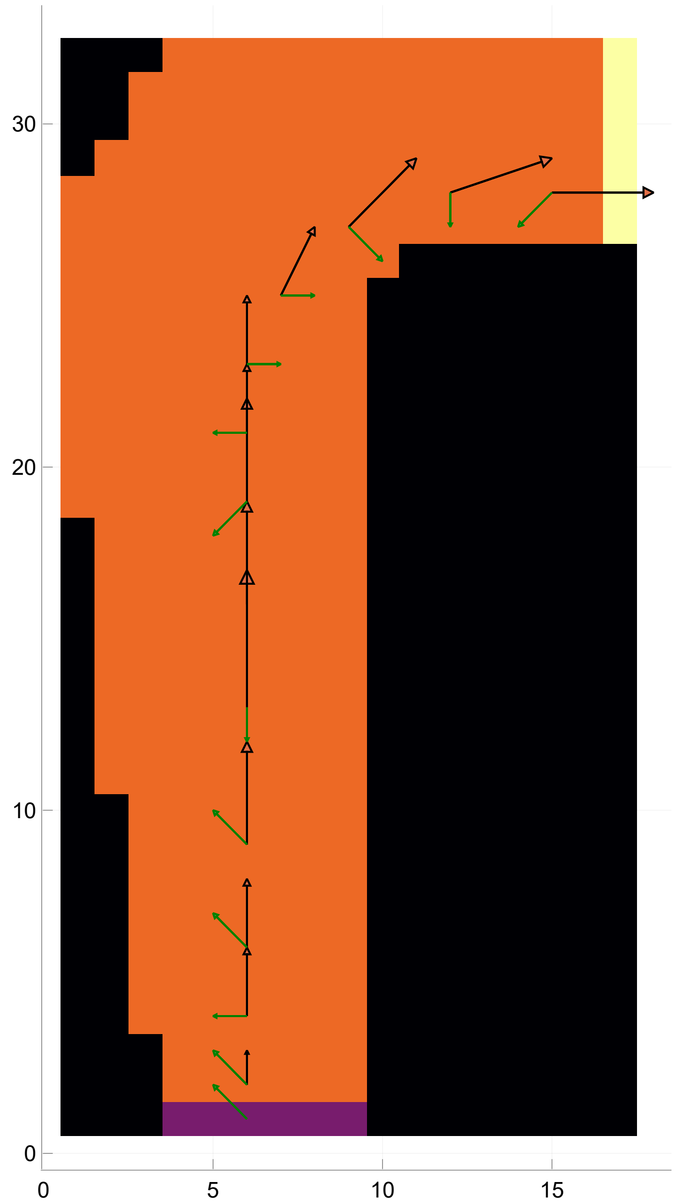

```
visualize_policy_traj(πstar_racetrack1)
```




```
visualize_policy_traj(πstar_racetrack2)
```




```
visualize_policy_traj(πstar_racetrack3)
```




```
#trajectory of a successful race policy, black arrows indicate velocity, green arrows indicate action.  Note that negative velocities are forbidden so any arrow pointing left on a vertical trajectory will have no impact.
visualize_policy_traj2(πstar_racetrack3)
```

5.8 Discounting-aware Importance Sampling

5.9 Per-decision Importance Sampling

Exercise 5.13 Show the steps to derive (5.14) from (5.12)

Starting at (5.12)

$$\rho_{t:T-1} R_{t+1} = \frac{\pi(A_t|S_t)}{b(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})} \frac{\pi(A_{t+2}|S_{t+2})}{b(A_{t+2}|S_{t+2})} \cdots \frac{\pi(A_{T-1}|S_{T-1})}{b(A_{T-1}|S_{T-1})} R_{t+1}$$

For (5.14) we need to turn this into an expected value

$$\mathbb{E}[\rho_{t:T-1} R_{t+1}]$$

Now we know that the reward at time step t+1 is only dependent on the action and state at time t. Moreover, the later parts of the trajectory are also independent of each other. So we can separate some of these terms into a product of expected values rather than an expected value of products:

$$\begin{aligned}
\mathbb{E}[\rho_{t:T-1} R_{t+1}] &= \mathbb{E}\left[\frac{\pi(A_t|S_t)}{b(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})} \frac{\pi(A_{t+2}|S_{t+2})}{b(A_{t+2}|S_{t+2})} \cdots \frac{\pi(A_{T-1}|S_{T-1})}{b(A_{T-1}|S_{T-1})} R_{t+1}\right] \\
&= \mathbb{E}\left[\frac{\pi(A_t|S_t)}{b(A_t|S_t)} R_{t+1}\right] \prod_{k=t+1}^{T-1} \mathbb{E}\left[\frac{\pi(A_k|S_k)}{b(A_k|S_k)}\right]
\end{aligned}$$

We know from (5.13) that $\mathbb{E}\left[\frac{\pi(A_k|S_k)}{b(A_k|S_k)}\right] = 1$ so the above expression simplifies to: $\mathbb{E}\left[\frac{\pi(A_t|S_t)}{b(A_t|S_t)} R_{t+1}\right]$. Using the original shorthand with ρ:

$$\mathbb{E}[\rho_{t:T-1} R_{t+1}] = \mathbb{E}\left[\frac{\pi(A_t|S_t)}{b(A_t|S_t)} R_{t+1}\right] = \mathbb{E}[\rho_{t:t} R_{t+1}]$$

Exercise 5.14 Modify the algorithm for off-policy Monte Carlo control (page 111) to use the idea of the truncated weighted-average estimator (5.10). Note that you will first need to convert this equation to action values.

Equation (5.10)

$$V(s) = \frac{\sum_{t\in\mathscr{T}(s)}\Big((1-\gamma)\sum_{h=t+1}^{T(t)-1}\gamma^{h-t-1}\rho_{t:h-1}\bar{G}_{t:h} + \gamma^{T(t)-t-1}\rho_{t:T(t)-1}\bar{G}_{t:T(t)}\Big)}{\sum_{t\in\mathscr{T}(s)}\Big((1-\gamma)\sum_{h=t+1}^{T(t)-1}\gamma^{h-t-1}\rho_{t:h-1} + \gamma^{T(t)-t-1}\rho_{t:T(t)-1}\Big)}$$

Converting this to action-value estimates:

$$Q(s,a) = \frac{\sum_{t\in\mathscr{T}(s,a)}\Big(R_{t+1} + (1-\gamma)\sum_{h=t+2}^{T(t)-1}\gamma^{h-t-1}\rho_{t+1:h-1}\bar{G}_{t+1:h} + \gamma^{T(t)-t-1}\rho_{t+1:T(t)-1}\bar{G}_{t+1:T}}{\sum_{t\in\mathscr{T}(s,a)}\Big(1 + (1-\gamma)\sum_{h=t+2}^{T(t)-1}\gamma^{h-t-1}\rho_{t+1:h-1} + \gamma^{T(t)-t-1}\rho_{t+1:T(t)-1}\Big)}$$

For the algorithm on page 111, need to add a variable in the loop to keep track of Ḡ both from the start of the episode forwards. The inner loop should also start from the beginning of each episode and go forwards rather than starting at the end going backwards. The term added to the numerator and denominator will be ready including Ḡ and ρ once the end of the episode is reached. A γ accumulator can be initiazed at 1 and kept track of in the inner loop by repeatedly multiplying by γ each iteration.