Chapter 4

Dynamic Programming

4.1 Policy Evaluation (Prediction)

```
using BenchmarkTools ✓ , Plots ✓
```

▸ PlotlyBackend()

```
plotly()
```

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

```
@enum GridworldAction up down left right
```

get_sa_keys (generic function with 1 method)

```
#p is the state transition function for an mdp which maps the 4 arguments to a probability.  This function uses p to generate two dictionaries.  The first maps each state to a set of possible actions in that state.  The second maps each state/action pair to a set of possible transition/reward pairs
function get_sa_keys(p::Dict{Tuple{A, B, A, C}, T}) where {T <: Real, A, B, C}
    #map from states to a list of possible actions
    state_actions = Dict{A, Set{C}}()

    #map from state action pairs to a list of possible newstate/reward pairs
    sa_s′rewards = Dict{Tuple{A, C}, Set{Tuple{A, B}}}()
    for k in keys(p)
        (s′, r, s, a) = k
        haskey(state_actions, s) ? push!(state_actions[s], a) : state_actions[s] = Set([a])
        haskey(sa_s′rewards, (s,a)) ? push!(sa_s′rewards[(s,a)], (s′, r)) : sa_s′rewards[(s,a)] = Set([(s′,r)])
    end
    return state_actions, sa_s′rewards
end
```

```
bellman_value! (generic function with 1 method)
```

```julia
function bellman_value!(V::Dict, p::Dict, sa_keys::Tuple, π::Dict, γ::Real)
    delt = 0.0
    for s in intersect(keys(sa_keys[1]), keys(π))
        v = V[s]
        actions = intersect(sa_keys[1][s], keys(π[s]))
        # if !isempty(actions)
            V[s] =  sum(π[s][a] *
                        sum(p[(s′,r,s,a)] * (r + γ*V[s′])
                            for (s′,r) in sa_keys[2][(s,a)])
                    for a in actions)
        # end
        delt = max(delt, abs(v - V[s]))
    end
    return delt
end
```

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

```julia
function iterative_policy_eval_v(π::Dict, θ::Real, mdp::NamedTuple, γ::Real, V::Dict,
delt::Real, nmax::Real)
    (p, sa_keys) = mdp
    if nmax <= 0 || delt <= θ
        return V
    else
        delt = bellman_value!(V, p, sa_keys, π, γ)
        iterative_policy_eval_v(π, θ, mdp, γ, V, delt, nmax - 1)
    end
end
```

```
iterative_policy_eval_v (generic function with 3 methods)
```

```julia
function iterative_policy_eval_v(π::Dict, θ::Real, mdp::NamedTuple, γ::Real, Vinit =
0.0; nmax = Inf)
    (p, sa_keys) = mdp
    V = Dict(s => Vinit for s in keys(sa_keys[1]))
    delt = bellman_value!(V, p, sa_keys, π, γ)
    iterative_policy_eval_v(π, θ, mdp, γ, V, delt, nmax - 1)
end
```

```
iterative_policy_eval_v (generic function with 4 methods)
```

```julia
function iterative_policy_eval_v(π::Dict, θ::Real, mdp::NamedTuple, γ::Real,
Vinit::Dict; nmax=Inf)
    (p, sa_keys) = mdp
    V = deepcopy(Vinit)
    delt = bellman_value!(V, p, sa_keys, π, γ)
    iterative_policy_eval_v(π, θ, mdp, γ, V, delt, nmax - 1)
end
```

Example 4.1

```
gridworld4x4_mdp (generic function with 1 method)
```

```julia
function gridworld4x4_mdp()
    S = collect(1:14)
    s_term = 0
    A = [up, down, left, right]
    #define p by iterating over all possible states and transitions
    p = Dict{Tuple{Int64, Int64, Int64, GridworldAction}, Float64}()

    #there is 0 reward and a probability of 1 staying in the terminal state for all
        actions taken from the terminal state
    for a in A
        push!(p, (0, 0, 0, a) => 1.0)
    end

    #add cases where end up in the terminal state
    push!(p, (s_term, -1, 14, right) => 1.0)
    push!(p, (s_term, -1, 11, down) => 1.0)
    push!(p, (s_term, -1, 1, left) => 1.0)
    push!(p, (s_term, -1, 4, up) => 1.0)

    for s in S
        for a in A
            for s′ in S
                check = if a == right
                    if (s == 3) || (s == 7) || (s == 11)
                        s′ == s
                    else
                        s′ == s+1
                    end
                elseif a == left
                    if (s == 4) || (s == 8) || (s == 12)
                        s′ == s
                    else
                        s′ == s-1
                    end
                elseif a == up
                    if (s == 1) || (s == 2) || (s == 3)
                        s′ == s
                    else
                        s′ == s - 4
                    end
                elseif a == down
                    if (s == 12) || (s == 13) || (s == 14)
                        s′ == s
                    else
                        s′ == s + 4
                    end
                end
                check && push!(p, (s′,-1,s,a) => 1.0)
            end
        end
    end
    sa_keys = get_sa_keys(p)
    return (p = p, sa_keys = sa_keys)
```

```
end
```

form_random_policy (generic function with 1 method)

```
#forms a random policy for a generic finite state mdp.  The policy is a dictionary that maps each state to a dictionary of action/probability pairs.
function form_random_policy(sa_keys)
    Dict([begin
        s = k[1]
        actions = k[2]
        l = length(actions)
        p = inv(l)
        s => Dict(a => p for a in actions)
    end
    for k in sa_keys[1]])
end
```

makefig4_1 (generic function with 2 methods)

```
function makefig4_1(nmax=Inf)
    gridworldmdp = gridworld4x4_mdp()
    π_rand = form_random_policy(gridworldmdp[2])
    V = iterative_policy_eval_v(π_rand, eps(0.0), gridworldmdp, 1.0, nmax = nmax)
    [(s, V[s]) for s in 0:14]
end
```

```
Tuple{Int64, Float64}[
     1: (0, 0.0)
     2: (1, -14.0)
     3: (2, -20.0)
     4: (3, -22.0)
     5: (4, -14.0)
     6: (5, -18.0)
     7: (6, -20.0)
     8: (7, -20.0)
     9: (8, -20.0)
    10: (9, -20.0)
    11: (10, -18.0)
    12: (11, -14.0)
    13: (12, -22.0)
    14: (13, -20.0)
    15: (14, -14.0)
]
```

```
makefig4_1(Inf)
```

Exercise 4.1 In Example 4.1, if π is the equiprobable random policy, what is $q_\pi(11, \text{down})$? What is $q_\pi(7, \text{down})$?

$$q_\pi(11, \text{down}) = -1$$

because this will transition into the terminal state and terminate the episode receiving the single reward of -1.

$$q_\pi(7, \text{down}) = -15$$

because we are gauranteed to end up in state 11 and receive a reward of -1 from the first action. Once we are in state 11, we can add $v_{\pi_{random}}(11) = -14$ to this value since the rewards are not discounted.

Exercise 4.2 In Example 4.1, supposed a new state 15 is added to the gridworld just below state 13, and its actions, left, up, right, and down, take the agent to states 12, 13, 14, and 15 respectively. Assume that the transitions *from* the original states are unchanged. What, then is $v_\pi(15)$ for the equiprobable random policy? Now supposed the dynamics of state 13 are also changed, such that action down from state 13 takes the agent to the new state 15. What is $v_\pi(15)$ for the equiprobable random policy in this case?

In the first case, we can never re-enter state 15 from any other state, so we can use the average of the value function in the states it transitions into.

$$v_\pi(15) = 0.25 \times (v_\pi(12) + v_\pi(13) + v_\pi(14) + v_\pi(15))$$

$$v_\pi(15) = 0.25 \times (-22 + -20 + -14 + v_\pi(15))$$

Solving for the value at 15 yields:

$$v_\pi(15) = \frac{0.25 \times -56}{0.75} = -18.666\ldots$$

In the second case, the value function at 13 and 15 become coupled because transitions back and forth are allowed. We can write down new Bellman equations for the equiprobably policy π of these states:

$$v_\pi(13) = -1 + \frac{1}{4}(v_\pi(9) + v_\pi(14) + v_\pi(12) + v_\pi(15))$$

$$v_\pi(15) = -1 + \frac{1}{4}(v_\pi(13) + v_\pi(14) + v_\pi(12) + v_\pi(15))$$

In the second equation we can simplify to get an equation for state 15 in terms of just 3 others.

$$v_\pi(15) \times \frac{3}{4} = -1 + \frac{1}{4}(v_\pi(13) + v_\pi(14) + v_\pi(12))$$

$$v_\pi(15) = \frac{1}{3}(-4 + v_\pi(13) + v_\pi(14) + v_\pi(12))$$

Let's try to approximate the new value at state 15 by substituting in the known values of the unmodified states.

$$v_\pi(15) \approx \frac{1}{3}(-4 - 20 - 14 - 22) = \frac{1}{3}(-60) = -20$$

Now let's get an implied updated value at state 13 by substituting in the approximate value at 15.

$$v_{\pi_{new}}(13) = -1 + \frac{1}{4}(-20 - 14 - 22 - 20) = -1 - \frac{76}{4} = -1 - 19 = -20 = v_{\pi_{old}}(13)$$

So we assumed that the value at state 13 was unchanged to get the approximation for state 15. Then using the self consistency equation for state 13 we confirmed that the original value is consistent with the approximate solution. This step of approximating the value at state 15 with a previous value function is analogous to what we would do in policy evaluation. However, when checking the value of state 13 we see that it remains unchanged after using state 15. If we were to carry this out for the other states that depend on 15, we would find that no futher changes are needed since 13 is the only state with a transition to 15 and states 12, 13, and 9 all now have new trasitions to 15 which would have been transitions to 13 previously. But the value estimate at 15 is identical to the original value at 13. This is the stopping condition for policy evaluation. Indeed if we carry out the full policy evaluation calculation below using the same method used to generate Figure 4.1, we see a value of -20 which is equal to the original value of state 13.

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

```julia
#Exercise 4.2 part 2
function gridworld_modified_mdp()
    S = collect(1:15)
    s_term = 0
    A = [up, down, left, right]

    #no discounting in this episodic task
    γ = 1.0

    #define p by iterating over all possible states and transitions
    p = Dict{Tuple{Int64, Int64, Int64, GridworldAction}, Float64}()

    #there is 0 reward and a probability of 1 staying in the terminal state for all
        actions taken from the terminal state
    for a in A
        push!(p, (0, 0, 0, a) => 1.0)
    end

    #add cases where end up in the terminal state
    push!(p, (s_term, -1, 14, right) => 1.0)
    push!(p, (s_term, -1, 11, down) => 1.0)
    push!(p, (s_term, -1, 1, left) => 1.0)
    push!(p, (s_term, -1, 4, up) => 1.0)

    for s in S
        for a in A
            for s′ in S
                check = if a == right
                    if (s == 3) || (s == 7) || (s == 11)
                        s′ == s
                    elseif s == 15
                        s′ == 14
                    else
                        (s != 14) && (s′ == s+1)
                    end
                elseif a == left
                    if (s == 4) || (s == 8) || (s == 12)
                        s′ == s
                    elseif (s == 15)
                        s′ == 12
                    else
                        s′ == s-1
                    end
                elseif a == up
                    if (s == 1) || (s == 2) || (s == 3)
                        s′ == s
                    elseif (s == 15)
                        s′ == 13
                    else
                        s′ == s - 4
                    end
                elseif a == down
                    if (s == 12) || (s == 14) || (s == 15)
```

```julia
                        s′ == s
                    elseif (s == 13)
                        s′ == 15
                    else
                        (s != 11) && (s′ == s + 4)
                    end
                end
                check && push!(p, (s′,-1,s,a) => 1.0)
            end
        end
    end
    sa_keys = get_sa_keys(p)
    return (p = p, sa_keys = sa_keys)
end
```

```
exercise4_2 (generic function with 2 methods)
```

```julia
function exercise4_2(nmax=Inf)
    gridworldmdp = gridworld_modified_mdp()
    π_rand = form_random_policy(gridworldmdp[2])
    V = iterative_policy_eval_v(π_rand, eps(0.0), gridworldmdp, 1.0, nmax = nmax)
    [(s, V[s]) for s in 0:15]
end
```

```
Tuple{Int64, Float64}[
    1: (0, 0.0)
    2: (1, -14.0)
    3: (2, -20.0)
    4: (3, -22.0)
    5: (4, -14.0)
    6: (5, -18.0)
    7: (6, -20.0)
    8: (7, -20.0)
    9: (8, -20.0)
    10: (9, -20.0)
    11: (10, -18.0)
    12: (11, -14.0)
    13: (12, -22.0)
    14: (13, -20.0)
    15: (14, -14.0)
    16: (15, -20.0)
]
```

```julia
#calculates value function for gridworld example in part 2 of exercise 4.2 with an added state 15
exercise4_2()
```

> *Exercise 4.3* What are the equations analogous to (4.3), (4.4), and (4.5), but for *action*-value functions instead of state-value functions?

Equation (4.3)

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]$$

action-value equivalent

$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$$

Equation (4.4)

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

action-value equivalent

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \sum_{a'} \pi(s',a') q_\pi(s',a')]$$

Equation (4.5)

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_k(s')]$$

action-value equivalent

$$q_{k+1}(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \sum_{a'} \pi(a'|s') q_k(s',a')]$$

4.3 Policy Iteration

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

```julia
function policy_improvement_v(π::Dict, mdp::NamedTuple, γ::Real, V::Dict)
    (p, sa_keys) = mdp
    π_new = Dict(begin
        actions = sa_keys[1][s]
        newdist = Dict(a =>
                sum(p[(s′,r,s,a)] * (r + γ*V[s′]) for (s′,r) in sa_keys[2][(s,a)])
                for a in actions)
        new_action = argmax(newdist)
        s => Dict(new_action => 1.0)
    end
    for s in keys(sa_keys[1]))

    policy_stable = mapreduce((a,b) -> a && b, keys(sa_keys[1])) do s
        argmax(π[s]) == argmax(π_new[s])
    end

    return (policy_stable, π_new)
end
```

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

```julia
function policy_iteration_v(mdp::NamedTuple, π::Dict, γ::Real, Vold::Dict, iters, θ,
evaln, policy_stable, resultlist)
    policy_stable && return (true, resultlist)
    V = iterative_policy_eval_v(π, θ, mdp, γ, Vold, nmax = evaln)
    (V == resultlist[end][1]) && return (true, resultlist)
    newresultlist = vcat(resultlist, (V, π))
    (iters <= 0) && return (false, newresultlist)
    (new_policy_stable, π_new) = policy_improvement_v(π, mdp, γ, V)
    policy_iteration_v(mdp, π_new, γ, V, iters-1, θ, evaln, new_policy_stable,
    newresultlist)
end
```

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

```julia
function begin_policy_iteration_v(mdp::NamedTuple, π::Dict, γ::Real; iters=Inf,
θ=eps(0.0), evaln = Inf, V = iterative_policy_eval_v(π, θ, mdp, γ, nmax = evaln))
    resultlist = [(V, π)]
    (policy_stable, π_new) = policy_improvement_v(π, mdp, γ, V)
    policy_iteration_v(mdp, π_new, γ, V, iters-1, θ, evaln, policy_stable, resultlist)
end
```

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

```julia
function gridworld_policy_iteration(nmax=10; θ=eps(0.0), γ=1.0)
    gridworldmdp = gridworld4x4_mdp()
    π_rand = form_random_policy(gridworldmdp[2])
    (policy_stable, resultlist) = begin_policy_iteration_v(gridworldmdp, π_rand, γ,
    iters = nmax)
    (Vstar, πstar) = resultlist[end]
    (policy_stable, Vstar, [(s, first(keys(πstar[s]))) for s in 0:14])
end
```

```
(
    1: true
    2: Dict{Int64, Float64}(
           5 ⇒ -2.0
           7 ⇒ -2.0
           12 ⇒ -3.0
           8 ⇒ -2.0
           1 ⇒ -1.0
           0 ⇒ 0.0
           4 ⇒ -1.0
           6 ⇒ -3.0
           13 ⇒ -2.0
           11 ⇒ -1.0
           2 ⇒ -2.0
           10 ⇒ -2.0
           9 ⇒ -3.0
           14 ⇒ -1.0
           3 ⇒ -3.0
       )
    3: Tuple{Int64, Main.workspace#3.GridworldAction}[
           1:  (0, up::GridworldAction = 0)
           2:  (1, left::GridworldAction = 2)
           3:  (2, left::GridworldAction = 2)
           4:  (3, down::GridworldAction = 1)
           5:  (4, up::GridworldAction = 0)
           6:  (5, up::GridworldAction = 0)
           7:  (6, left::GridworldAction = 2)
           8:  (7, down::GridworldAction = 1)
           9:  (8, up::GridworldAction = 0)
           10:  (9, up::GridworldAction = 0)
           11:  (10, right::GridworldAction = 3)
           12:  (11, down::GridworldAction = 1)
           13:  (12, up::GridworldAction = 0)
           14:  (13, right::GridworldAction = 3)
           15:  (14, right::GridworldAction = 3)
       ]
)
```

```
#seems to match optimal policy from figure 4.1
gridworld_policy_iteration()
```

> *Exercise 4.4* The policy iteration algorithm on page 80 has a subtle bug in that it may never terminate if the policy continually switches between two or more policies that are equally good. This is okay for pedagogy, but not for actual use. Modify the pseudocode so that convergence is guaranteed.

Initialize V_{best} at the start randomly and replace it with the first value function calculated. After each policy improvement, replace V_{best} with the new value function, however add a check after step 2. that if the value function is the same as V_{best} then stop. This would ensure that no matter how many equivalent policies are optimal, they would all share the same value function and thus trigger the termination condition.

Exercise 4.5 How would policy iteration be defined for action values? Give a complete algorithm for computer q_*, analogous to that on page 80 for computing v_*. Please pay special attention to this exercise, because the ideas involved will be used throughout the rest of the book.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$ using action-values

1. Initialization

$$Q(s,a) \in \mathbb{R} \text{ and } \pi(s) \in \mathcal{A}(s) \text{ arbitrarily for all } s \in \mathcal{S}; Q(terminal, a) \doteq 0 \; \forall \; a \in \mathcal{A}$$

2. Policy Evaluation

   Loop:

$$\Delta \leftarrow 0$$

Loop for each $s \in \mathcal{S}$:

Loop for each $a \in \mathcal{A}(s)$:

$$q \leftarrow Q(s,a)$$

$$Q(s,a) \leftarrow \sum_{s',r} p(s',r|s,a)[r + \gamma \sum_{a'} \pi(s',a')Q(s',a')]$$

$$\Delta \leftarrow \max(\Delta, |q - Q(s,a)|)$$

until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement

policy-stable $\leftarrow$ *true*

For each $s \in \mathcal{S}$:

$$old-action \leftarrow \pi(s)$$

$$\pi(s) \leftarrow \text{argmax}_a Q(s,a)$$

If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow false$

If *policy-stable*, then stop and return $Q \approx q_*$ and $\pi \approx \pi_*$; else go to 2

> *Exercise 4.6* Suppose you are restricted to considering only policies that are *ϵ-soft*, meaning that the probability of selecting each action in each state, s, is at least $\epsilon/|\mathcal{A}(s)|$. Describe qualitatively the changes that would be required in each of the steps 3,2,and 1, in that order, of the policy iteration algorithm for v_* on page 80.

For step 3: To get the old-action take the argmax over possible actions of the policy distribution for state s. Rewrite π as π(a|s). Instead of having a probability of 1.0 for the argmax of the expression, we must adjust the value to be $1.0 - \epsilon$. Similarly the *old-action* and *new-action* should be the argmax of the policy distribution at state s rather than the single value.

For step 2:

The expression for updating the value function should have a sum over possible actions weighted by the policy distribution for each action. The inner sum can remain the same except the policy argument for p should be replaced with the variable summing over actions.

For step 1:

The initialization of the policy function should be a uniform distribution over all possible actions for each state rather than a single action value.

> *Exercise 4.7 (programming)* Write a program for policy iteration and re-solve Jack's car rental problem with the following changes. One of Jack's employees at the first location rides a bus home each night and lives near the second location. She is happy to shuttle one car to the second location for free. Each additional car still costs 2, as do all cars moved in the other direction. In addition, Jack has limited parking space at each location. If more than 10 cars are kept overnight at a location (after any moving of cars), then an additional cost of 4 must be incurred to use a second parking lot (independent of how many cars are kept there). These sorts of nonlinearities and arbitrary dynamics often occur in real problems and cannot easily be handled by optimization methods other than dynamic programming. To check your program, first replicate the results given for the original problem.

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

```
poisson(n, λ) = exp(-λ) * (λ^n) / factorial(n)
```

car_rental_mdp (generic function with 1 method)

```julia
function car_rental_mdp(;nmax=20, λs = (3,4,3,2), movecost = 2, rentcredit = 10, movemax=5)
    #enumerate all possible states from which to transition
    S = ((x, y) for x in 0:nmax for y in 0:nmax)

    #check that new states are valid
    function checkstate(S)
        @assert (S[1] >= 0) && (S[1] <= nmax)
        @assert (S[2] >= 0) && (S[2] <= nmax)
    end

    #before proceeding, it will be useful to have a lookup table of probabilities for all of the possible rental and return requests at each location.  Since we can never rent more cars than are available, 0 to nmax-1 is the only range that needs to be considered.  For returns, we can receive any arbitrary number but any that exceed nmax will be returned.  Thus we may have a situation where receiving as a return any number greater than or equal to a given value will result in the same state.  To calculate such a probability we need to sum up all of the probilities for return values less than that and subtract it from 1.  If we have 0 cars at a given location prior to returns, then the maximum return value we would need to calculate is up to nmax-1.  That way the probability leading to nmax cars would be 1 minus the sum of every other probility calculated from 0 to nmax-1.
    rentprobs = Dict((loc, rent) => poisson(rent, λs[loc]) for rent in 0:nmax-1 for loc in 1:2)
    retprobs = Dict((loc, ret) => poisson(ret, λs[loc+2]) for ret in 0:nmax-1 for loc in 1:2)

    #define p by iterating over all possible states and transitions
    ptf = Dict{Tuple{Tuple{Int64, Int64}, Int64, Tuple{Int64, Int64}, Int64}, Float64}()

    for s in S
        #for actions a negative number indicates moving cars from 2 to 1
        #a positive number indicates moving cars from 1 to 2
        for a in -min(movemax, s[2]):min(movemax, s[1])
            #after taking action a, we have our first intermediate state for the next morning which cannot exceed nmax at each location
            sint1 = (min(s[1]-a, nmax), min(s[2]+a, nmax))
            checkstate(sint1)

            #the next day we can only rent cars from each location that are available
            for (rent1, rent2) in ((x,y) for x in 0:sint1[1] for y in 0:sint1[2])
                #after specifying the number of cars rented we have our final reward value
                r = rentcredit*(rent1+rent2) - movecost*abs(a)

                #if we n cars from a given location, we could have received rental requests for that number or higher.  So the probability of such a rental is 1 minus the sum of the probability of receiving every request less than that number
                function calcrentprob(loc, nrent)
                    ncars = sint1[loc]
                    @assert nrent <= ncars
```

```julia
                if ncars == 0
                    1.0
                elseif nrent < ncars
                    rentprobs[(loc, nrent)]
                else
                    1.0 - sum(rentprobs[(loc, r)] for r in 0:nrent-1)
                end
            end

            #calculate the probability of renting these cars at these locations
            prent = calcrentprob(1, rent1)*calcrentprob(2, rent2)

            #new intermediate state after renting cars
            sint2 = (sint1[1]-rent1, sint1[2]-rent2)
            checkstate(sint2)

            #after receiving returns, we can only increase the number of cars at
            each loaction, so the possible final transition states we can end up
            with are as follows
            for s′ in ((x,y) for x in sint2[1]:nmax for y in sint2[2]:nmax)
                checkstate(s′)

                #change in cars from returns
                delt1 = s′[1] - sint2[1]
                delt2 = s′[2] - sint2[2]

                function pdelt(loc, delt)
                    if sint2[loc] == nmax
                        #in this case the location already had the maximum number
                        of cars so any return value is possible
                        1.0
                    elseif s′[loc] < nmax
                        #in this the requested returns match delta
                        retprobs[(loc,delt)]
                    else
```

```
jacks_car_mdp =
▸(p = Dict(((19, 13), 136, (0, 15), -2) ⇒ 3.04753e-19, ((12, 8), 200, (12, 12), -5) ⇒ 9.10
```

```julia
jacks_car_mdp = car_rental_mdp()
```

convertcarpolicy (generic function with 2 methods)

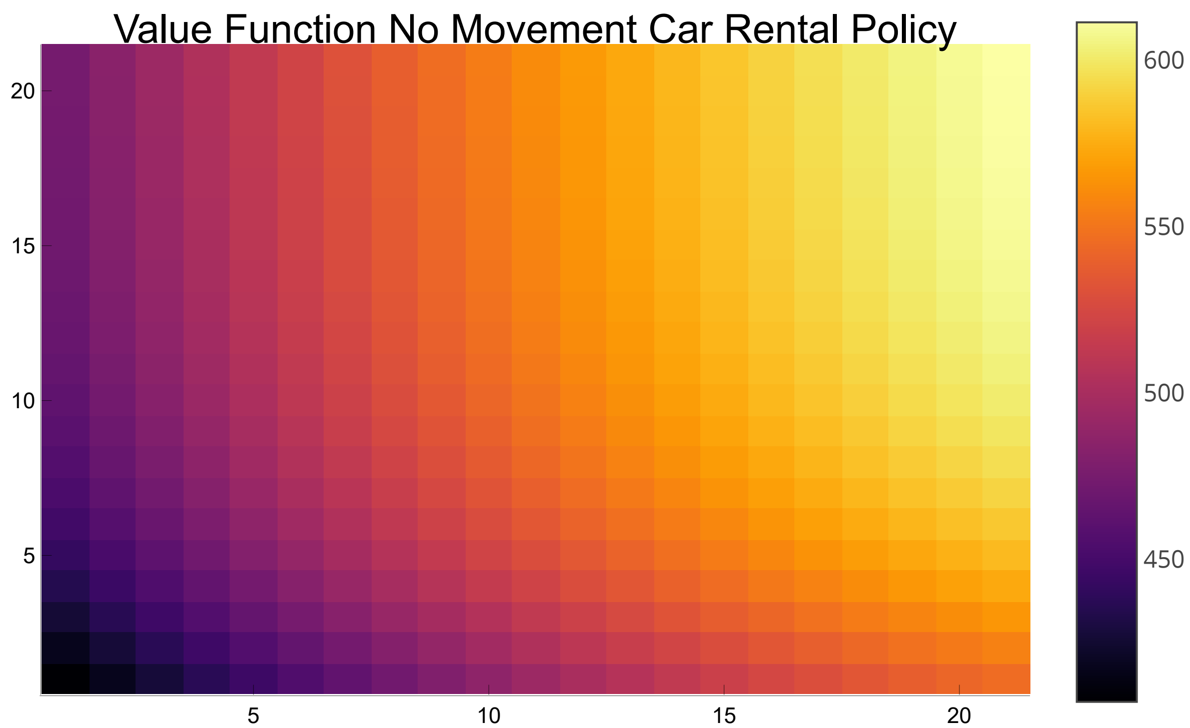
```julia
function convertcarpolicy(V, π, nmax=20)
    vmat = zeros(nmax+1, nmax+1)
    pmat = zeros(nmax+1, nmax+1)
    A = -nmax:nmax
    for i = 0:nmax
        for j = 0:nmax
            vmat[i+1,j+1] = V[(i,j)]
            a = argmax(π[(i,j)])
            pmat[i+1,j+1] = a
        end
    end
    return (value=vmat, policy=pmat)
end
```

car_rental_policy_eval (generic function with 2 methods)

```julia
#first test that the policy evaluation works on the mdp
function car_rental_policy_eval(mdp, nmax=Inf; θ = eps(0.0), γ=0.9)
    states = keys(mdp.sa_keys[1])
    π_0 = Dict(s => Dict(0 => 1.0) for s in states)
    V0 = iterative_policy_eval_v(π_0, θ, mdp, γ, nmax = nmax)
    nullpolicymats = convertcarpolicy(V0, π_0)
    (V0, π_0, nullpolicymats)
end
```

```
V0_car_rental_eval =
▸(Dict((18, 16) ⇒ 594.008, (16, 14) ⇒ 582.562, (11, 17) ⇒ 592.508, (17, 12) ⇒ 571.429,
```

```julia
V0_car_rental_eval = car_rental_policy_eval(jacks_car_mdp, Inf)
```




```
heatmap(V0_car_rental_eval[3][1], title="Value Function No Movement Car Rental Policy")
```

car_rental_policy_iteration (generic function with 2 methods)

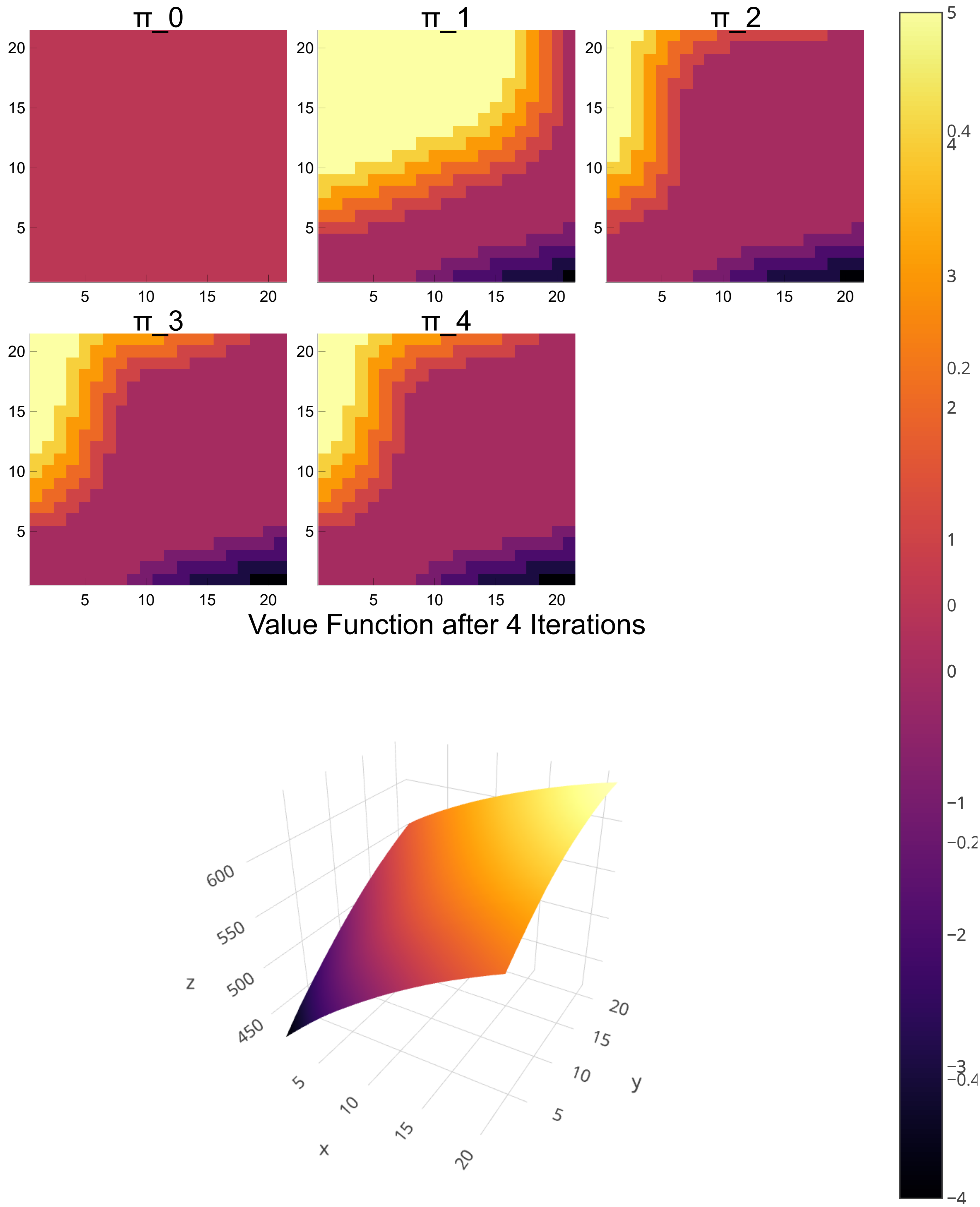
```
#now try policy iteration
function car_rental_policy_iteration(mdp, nmax=10; θ=eps(0.0), γ=0.9, null_policy_eval = car_rental_policy_eval(mdp))
    (V0, π_0, mats) = null_policy_eval
    (converged, resultlist) = begin_policy_iteration_v(mdp, π_0, γ, V = V0, iters = nmax, θ = θ)
    (converged, [(Vstar, πstar, convertcarpolicy(Vstar, πstar)) for (Vstar, πstar) in resultlist])
end
```

```
example4_2_results =
▸(true, [(Dict(⋯ more), Dict(⋯ more), (value = 21×21 Matrix{Float64}:
                                              407.179  417.153  427.018  436.65   445.9
                                              416.999  426.973  436.838  446.47   455.7
                                              426.233  436.207  446.072  455.704  464.9
                                              434.477  444.451  454.317  463.948  473.2
                                              441.539  451.513  461.378  471.009  480.2
                                              447.445  457.419  467.284  476.916  486.1
                                              452.339  462.313  472.179  481.81   491.0
                                                ⋮
                                              471.4    481.374  491.239  500.871  510.1
                                              472.073  482.047  491.912  501.544  510.8
                                              472.601  482.575  492.44   502.072  511.3
                                              473.003  482.977  492.843  502.474  511.7
                                              473.297  483.271  493.136  502.768  512.0
                                              473.498  483.472  493.337  502.969  512.2
```

```julia
example4_2_results = car_rental_policy_iteration(jacks_car_mdp, θ=0.01,
null_policy_eval=V0_car_rental_eval)
```

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

```julia
function plotcarpolicy(results)
    πheatmaps = [a[3][2] for a in results]
    finalvaluemap = results[end][3][1]
    plist = [heatmap(h, title="π_$(i-1)") for (i,h) in enumerate(πheatmaps)]
    pvalue = surface(finalvaluemap, title="Value Function after $(length(results)-1)
    Iterations", legend = false)
    πplots = plot(Tuple(plist)...)
    plot(πplots, pvalue, layout = (2,1), size=(700, 900))
end
```




```
plotcarpolicy(example4_2_results[2])
```

car_rental_modified_mdp (generic function with 1 method)

```julia
function car_rental_modified_mdp(;nmax=20, λs = (3,4,3,2), movecost = 2, rentcredit = 10, movemax=5)
    #enumerate all possible states from which to transition
    S = ((x, y) for x in 0:nmax for y in 0:nmax)

    #lookup tables for rental and return request probabilities
    rentprobs = Dict((loc, rent) => poisson(rent, λs[loc]) for rent in 0:nmax-1 for loc in 1:2)
    retprobs = Dict((loc, ret) => poisson(ret, λs[loc+2]) for ret in 0:nmax-1 for loc in 1:2)

    #define p by iterating over all possible states and transitions
    ptf = Dict{Tuple{Tuple{Int64, Int64}, Int64, Tuple{Int64, Int64}, Int64}, Float64}()

    for s in S
        #for actions a negative number indicates moving cars from 2 to 1
        #a positive number indicates moving cars from 1 to 2
        for a in -min(movemax, s[2]):min(movemax, s[1])
            #after taking action a, we have our first intermediate state for the next morning which cannot exceed nmax at each location
            sint1 = (min(s[1]-a, nmax), min(s[2]+a, nmax))

            move_expense = movecost * ((a > 0) ? (a-1) : -a)

            #the next day we can only rent cars from each location that are available
            for (rent1, rent2) in ((x,y) for x in 0:sint1[1] for y in 0:sint1[2])
                #if we n cars from a given location, we could have received rental requests for that number or higher.  So the probability of such a rental is 1 minus the sum of the probability of receiving every request less than that number
                function calcrentprob(loc, nrent)
                    ncars = sint1[loc]
                    if ncars == 0
                        1.0
                    elseif nrent < ncars
                        rentprobs[(loc, nrent)]
                    else
                        1.0 - sum(rentprobs[(loc, r)] for r in 0:nrent-1)
                    end
                end

                #calculate the probability of renting these cars at these locations
                prent = calcrentprob(1, rent1)*calcrentprob(2, rent2)

                #new intermediate state after renting cars
                sint2 = (sint1[1]-rent1, sint1[2]-rent2)

                #after receiving returns, we can only increase the number of cars at each loaction, so the possible final transition states we can end up with are as follows
                for s′ in ((x,y) for x in sint2[1]:nmax for y in sint2[2]:nmax)

```

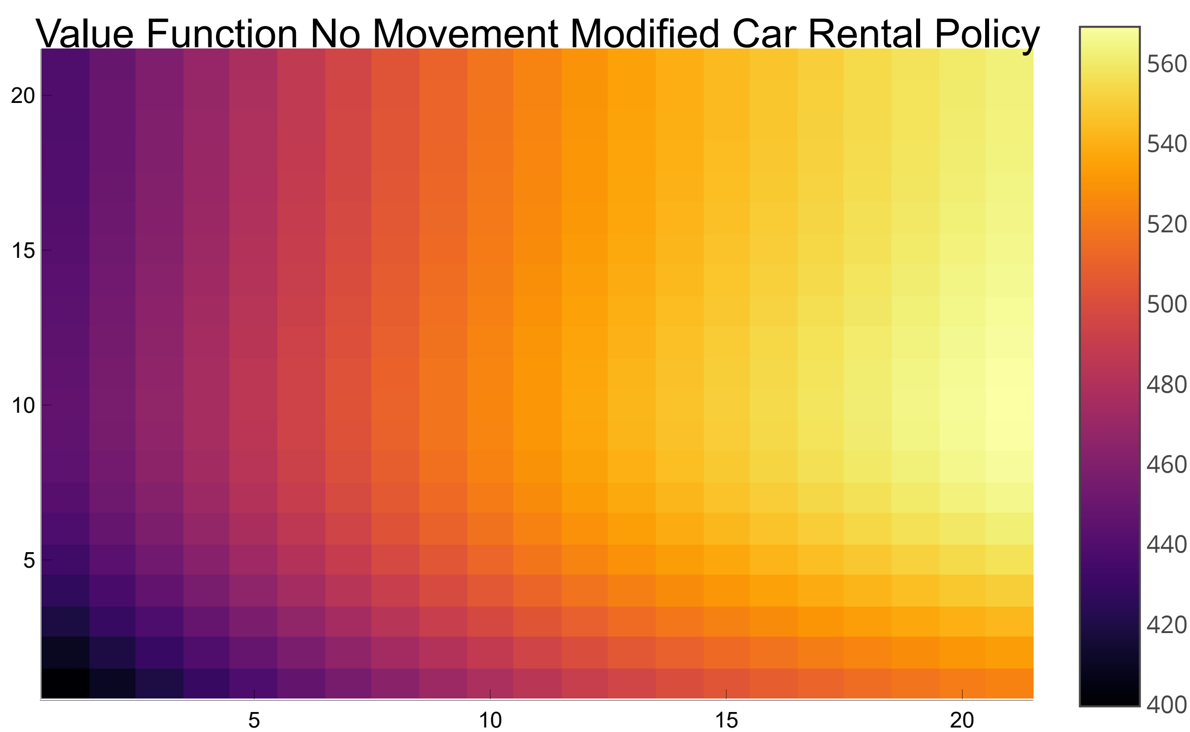
```
#change in cars from returns
delt1 = s′[1] - sint2[1]
delt2 = s′[2] - sint2[2]

function pdelt(loc, delt)
    if sint2[loc] == nmax
        #in this case the location already had the maximum number
        of cars so any return value is possible
        1.0
    elseif s′[loc] < nmax
        #in this the requested returns match delta
        retprobs[(loc,delt)]
    else
        1.0 - sum(retprobs[(loc, r)] for r in 0:delt-1)
    end
end

pret = pdelt(1, delt1)*pdelt(2, delt2)

#after specifing the number of cars rented, moved, and at each
location we can calculate our total reward
secondlotcost = 4 * ((s′[1] > 10) + (s′[2] > 10))
r = rentcredit*(rent1+rent2) - move_expense - secondlotcost
```

```
modified_jacks_car_mdp =
▸ (p = Dict(((1, 10), 96, (8, 4), 3) ⇒ 3.87208e-6, ((13, 18), 162, (11, 18), -5) ⇒ 4.77879e
```

```
modified_jacks_car_mdp = car_rental_modified_mdp()
```

```
V0_modified_car_rental_eval =
▸ (Dict((18, 16) ⇒ 550.474, (16, 14) ⇒ 543.964, (11, 17) ⇒ 559.458, (17, 12) ⇒ 534.878,
```

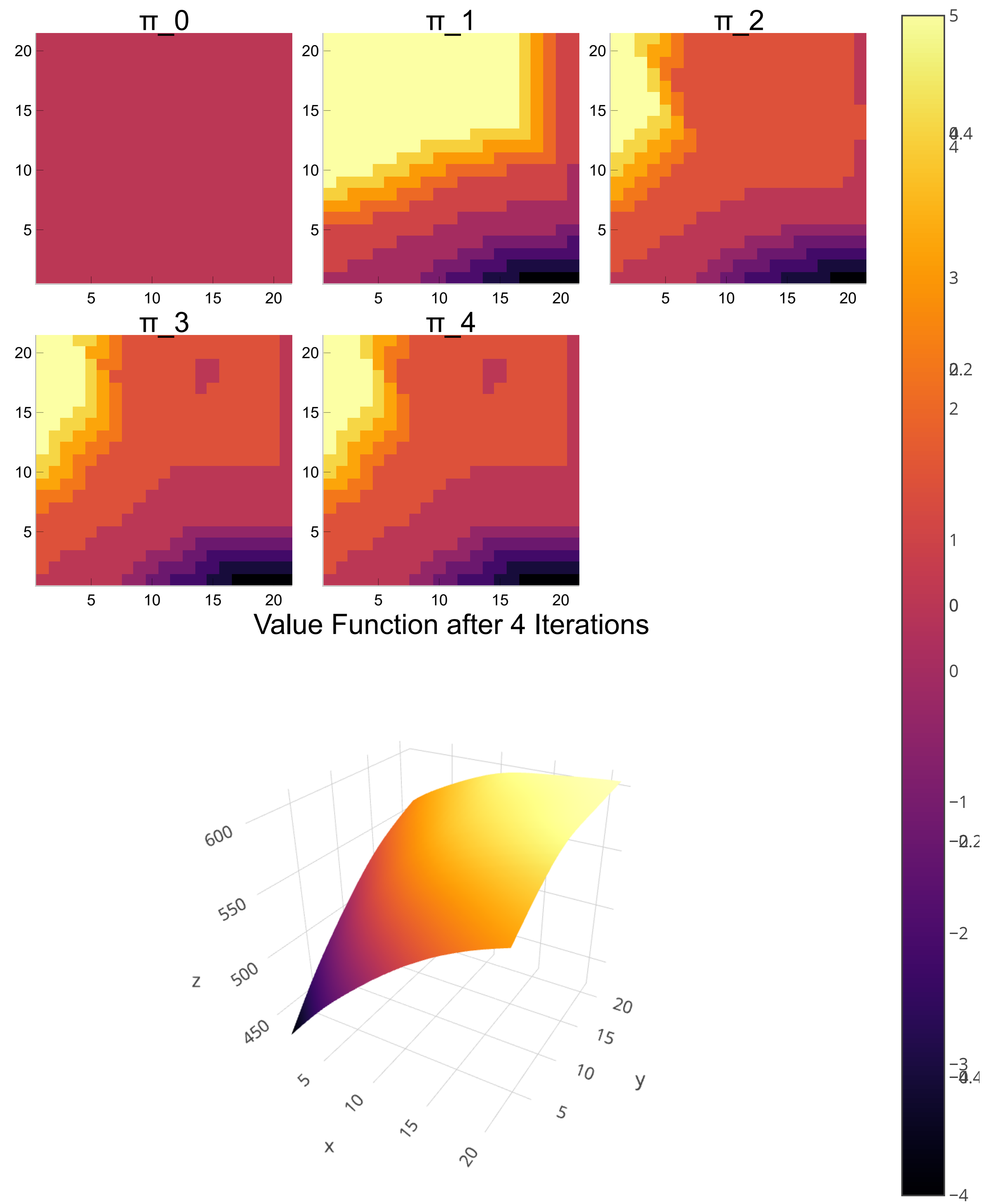
```
V0_modified_car_rental_eval = car_rental_policy_eval(modified_jacks_car_mdp)
```




```
heatmap(V0_modified_car_rental_eval[3][1], title="Value Function No Movement Modified Car Rental Policy")
```

```
exercise4_7_results =
▸ (true, [(Dict( ⋯ more), Dict( ⋯ more), (value = 21×21 Matrix{Float64}:
                                                    399.572  409.545  419.41   429.038  438.3
                                                    409.363  419.337  429.202  438.83   448.1
                                                    418.473  428.447  438.311  447.939  457.2
                                                    426.413  436.387  446.251  455.88   465.1
                                                    432.924  442.898  452.762  462.391  471.6
                                                    437.986  447.96   457.824  467.453  476.7
                                                    441.704  451.678  461.543  471.171  480.4
                                                      ⋮
                                                    440.863  450.837  460.701  470.33   479.6
                                                    440.201  450.175  460.04   469.668  478.9
                                                    439.683  449.657  459.522  469.15   478.4
                                                    439.29   449.264  459.128  468.756  478.0
                                                    439.003  448.977  458.841  468.47   477.7
                                                    438.806  448.78   458.644  468.272  477.5
```

```
exercise4_7_results = car_rental_policy_iteration(modified_jacks_car_mdp, θ=0.01,
null_policy_eval=V0_modified_car_rental_eval)
```




```
plotcarpolicy(exercise4_7_results[2])
```

4.4 Value Iteration

bellman_optimal_value! (generic function with 1 method)

```
function bellman_optimal_value!(V::Dict, p::Dict, sa_keys::Tuple, γ::Real)
    delt = 0.0
    for s in keys(sa_keys[1])
        v = V[s]
        actions = sa_keys[1][s]
        V[s] = maximum(sum(p[(s′,r,s,a)] * (r + γ*V[s′]) for (s′,r) in sa_keys[2]
        [(s,a)]) for a in actions)
        delt = max(delt, abs(v - V[s]))
    end
    return delt
end
```

value_iteration_v (generic function with 1 method)

```
function value_iteration_v(θ::Real, mdp::NamedTuple, γ::Real, V::Dict, delt::Real,
nmax::Real, valuelist)
    (p, sa_keys) = mdp
    if nmax <= 0 || delt <= θ
        (πstar, πraw) = calculatepolicy(mdp, γ, V)
        return (valuelist, πstar, πraw)
    else
        newV = deepcopy(V)
        delt = bellman_optimal_value!(newV, p, sa_keys, γ)
        value_iteration_v(θ, mdp, γ, newV, delt, nmax - 1, vcat(valuelist, newV))
    end
end
```

calculatepolicy (generic function with 1 method)

```
function calculatepolicy(mdp::NamedTuple, γ::Real, V::Dict)
    (p, sa_keys) = mdp
    πraw = Dict(begin
        actions = sa_keys[1][s]
        newdist = Dict(a =>
                sum(p[(s′,r,s,a)] * (r + γ*V[s′]) for (s′,r) in sa_keys[2][(s,a)])
                for a in actions)
        s => newdist
    end
    for s in keys(sa_keys[1]))
    πstar = Dict(s => Dict(argmax(πraw[s]) => 1.0) for s in keys(πraw))
    πstar, πraw
end
```

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

```julia
function begin_value_iteration_v(mdp::NamedTuple, γ::Real; θ = eps(0.0), nmax=Inf, Vinit = 0.0)
    (p, sa_keys) = mdp
    V = Dict(s => Vinit for s in keys(sa_keys[1]))
    newV = deepcopy(V)
    delt = bellman_optimal_value!(newV, p, sa_keys, γ)
    value_iteration_v(θ, mdp, γ, newV, delt, nmax-1, [V, newV])
end
```

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

```julia
function begin_value_iteration_v(mdp::NamedTuple, γ::Real, V; θ = eps(0.0), nmax=Inf)
    (p, sa_keys) = mdp
    newV = deepcopy(V)
    delt = bellman_optimal_value!(newV, p, sa_keys, γ)
    value_iteration_v(θ, mdp, γ, newV, delt, nmax-1, [V, newV])
end
```

```
▶ ([Dict(5 ⇒ 0.0, 7 ⇒ 0.0, 12 ⇒ 0.0, 8 ⇒ 0.0, 1 ⇒ 0.0, 0 ⇒ 0.0, ⋯ more), Dict(5 ⇒ -1
```

```julia
begin_value_iteration_v(gridworld4x4_mdp(), 1.0)
```

Example 4.3: Gambler's Problem

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

```julia
function make_gambler_mdp(p::Real)
    ptf = Dict{Tuple{Int64, Int64, Int64, Int64}, Float64}()
    stermwin = 100
    stermlose = 0
    for s in 1:99
        for a in 0:min(s,100-s)
            swin = s+a
            slose = s-a
            if swin == stermwin
                ptf[(swin, 1, s, a)] = p
            else
                ptf[(swin, 0, s, a)] = p
            end

            ptf[(slose, 0, s, a)] = 1.0-p
        end
    end
    sa_keys = get_sa_keys(ptf)
    V = Dict(s => 0.0 for s in keys(sa_keys[1]))
    V[stermwin] = 0.0
    V[stermlose] = 0.0
    return (p = ptf, sa_keys = sa_keys, Vinit = V)
end
```

```
gambler_mdp =
▸(p = Dict((35, 0, 22, 13) ⇒ 0.4, (27, 0, 53, 26) ⇒ 0.6, (96, 0, 59, 37) ⇒ 0.4, (65, 0, 8
```

```julia
gambler_mdp = make_gambler_mdp(0.4)
```

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

```julia
function multiargmax(π_s::Dict{A, B}) where {A, B}
#takes a distribution over actions and returns a set of actions that share the same maximum value.  If there is a unique maximum then only one element will be in the set
    a_max = argmax(π_s)
    p_max = π_s[a_max]
    a_set = Set([a_max])
    for a in keys(π_s)
        (π_s[a] ≈ p_max) && push!(a_set, a)
    end
    return a_set
end
```

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

```julia
function create_action_grid(action_sets, statelist)
#converts action_sets at each state into a square matrix where optimal actions are marked 1 and others are 0
    l = length(statelist)
    output = zeros(l+1, l)
    for i in 1:l
        for j in action_sets[i]
            output[j+1, i] = 1
        end
    end
    return output
end
```

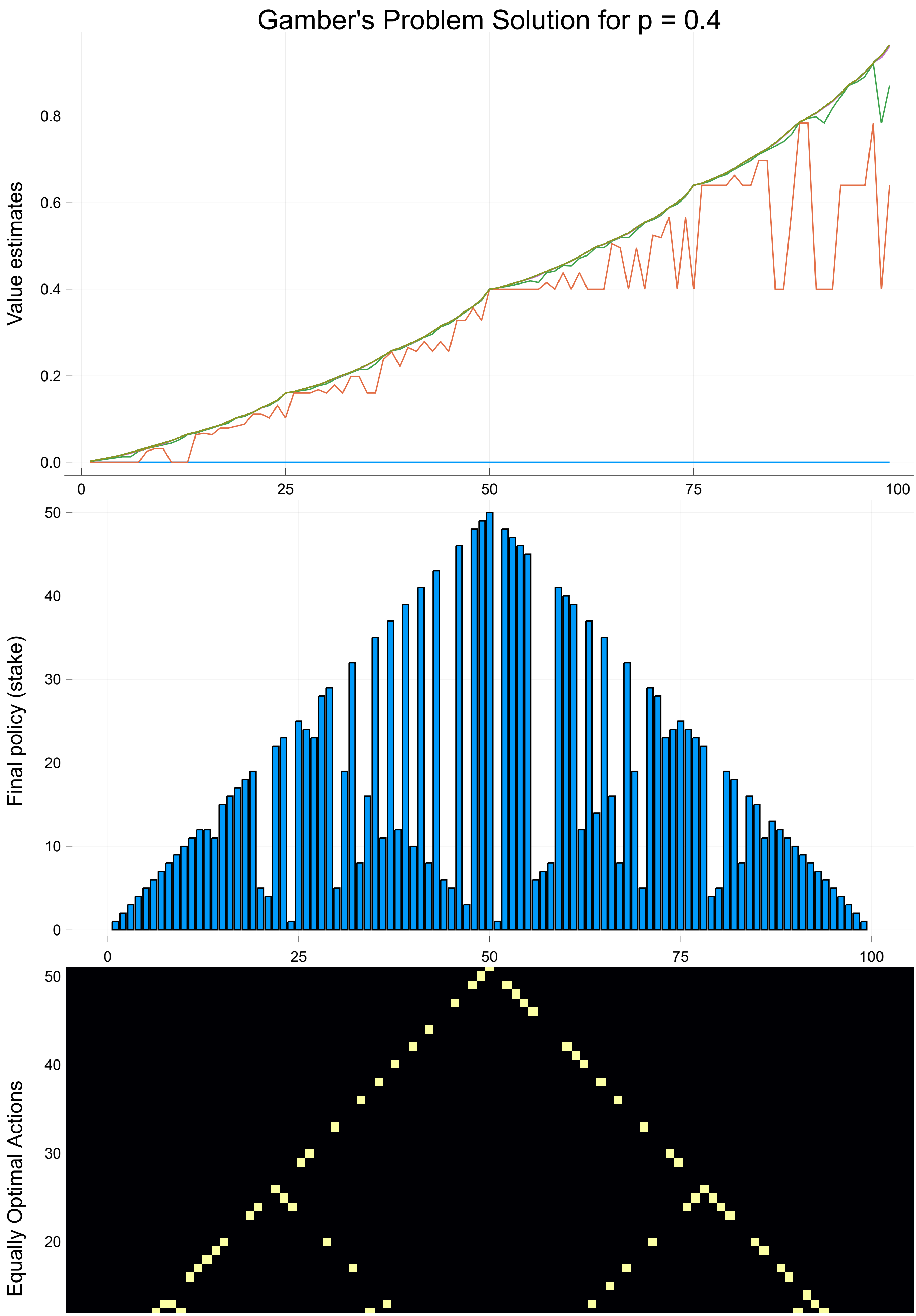
```
formindrange (generic function with 3 methods)
```

```julia
#for plotting purposes take a long list of lines and sample them coarsely
function formindrange(l, maxind::Int64 = 5, inds::AbstractVector = 1:5)
    if l < maxind
            return vcat(filter(a -> a < l, inds), l)
        else
            newmax = 5*maxind
            newrange = maxind*2:maxind:newmax
            formindrange(l, newmax, vcat(inds, newrange))
        end
    end
```

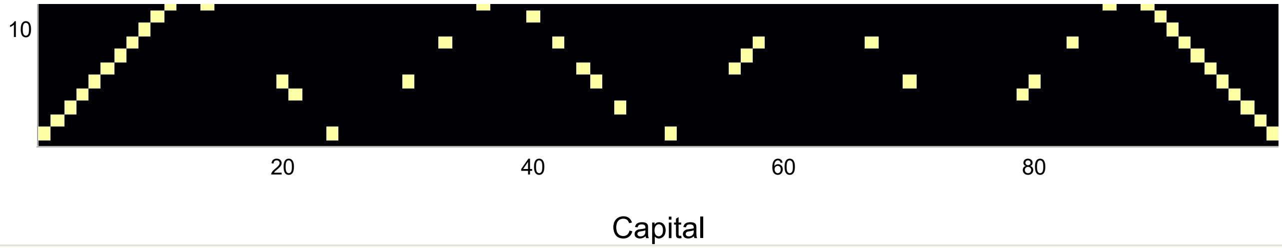
```
plot_gambler_results (generic function with 2 methods)
```

```julia
function plot_gambler_results(p, θ=eps(0.0))
    mdp = make_gambler_mdp(p)
    statelist = sort(collect(keys(mdp.sa_keys[1])))
    (valuelist, πstar, πraw) = begin_value_iteration_v(mdp, 1.0, mdp.Vinit, θ=θ)
    l = length(valuelist)
    indlist = formindrange(l)

    value_estimates = mapreduce(hcat, view(valuelist, indlist)) do v
        [v[s] for s in statelist]
    end

    p1 = plot(statelist, value_estimates, ylabel = "Value estimates", title =
    "Gamber's Problem Solution for p = $p", lab = reshape(["sweep $i" for i in
    indlist], 1, length(indlist)))
    optimal_actions = [argmax(πstar[s]) for s in statelist]
    optimal_action_sets = [multiargmax(πstar[s]) for s in statelist]
    p2 = bar(statelist, optimal_actions, ylabel = "Final policy (stake)")
    p3 = heatmap(create_action_grid(optimal_action_sets, statelist), xlabel =
    "Capital", ylabel = "Equally Optimal Actions", legend=false, yaxis = [1, 51],
    yticks = 0:10:100)
    plot(p1, p2, p3, layout=(3, 1), size = (670, 1100), legend = false)
end
```

Figure 4.3



Gamber's Problem Solution for p = 0.4
Value estimates
0.0
0.2
0.4
0.6
0.8
0
25
50
75
100
Final policy (stake)
0
10
20
30
40
50
0
25
50
75
100
Equally Optimal Actions
20
30
40
50



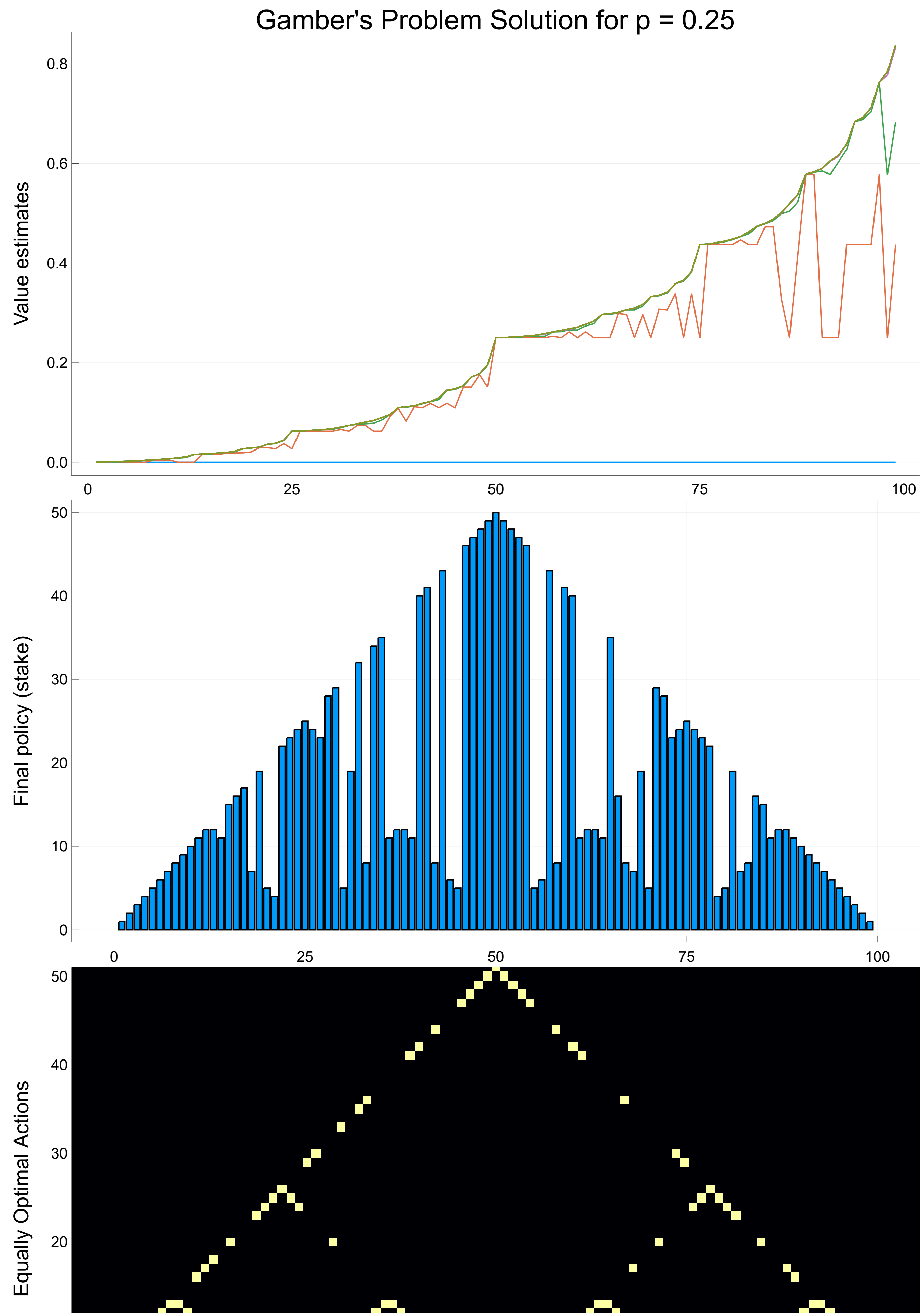

```
plot_gambler_results(0.4)
```

> *Exercise 4.8* Why does the optimal policy for the gambler's problem have such a curious form? In particular, for capital of 50 it bets it all on one flip, but for capital of 51 it does not. Why is this a good policy?

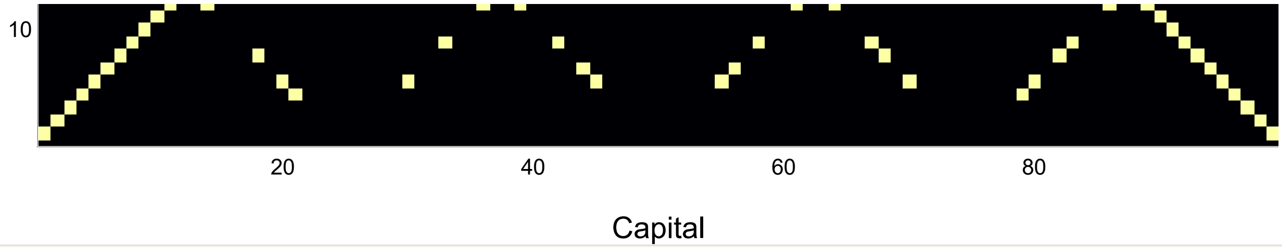
At capital of 50, it is possible to reach the terminal winning state with a 100% stake. In the value function estimate we see that this state is valued, as expected, at the probability of receiving a winning flip. Every capital state larger than 50 has a higher value estimate than this presumably because if we lose a flip we can always try again from the 50 state and otherwise we can more slowly advance up the capital states. Then again at 75, there is a potentially winning stake of 25. However, if we lose at the 75 state, we drop to 50 and have another chance to win. That is why the 75 state will always be valued higher than the 50 state. Since p_h is less than 50%, if we chose to play it safe and bet less than a winning amount at 50, it is actually most likely that we lose capital progressively and never again reach the 50 state. Therefore, it makes sense that the moment we reach the 50 state (one flip away from a win), we take the oppotunity to win immediately. The situation is completely different in a game where the probability of a winning flip is greater than half. In that case, it would never make sense to risk enough capital to lose in one turn, because we would expect in the long run to accumulate capital slowly.

> *Exercise 4.9 (programming)* Implement value iteration for the gamber's problem and solve it for $p_h = 0.25$ and $p_h = 0.55$. In programming, you may find it convenient to introduce two dummy states corresponding to termination with capital of 0 and 100, giving them values of 0 and 1 respectively. Show your results graphically as in Figure 4.3 Are you results stable as $\theta \to 0$?

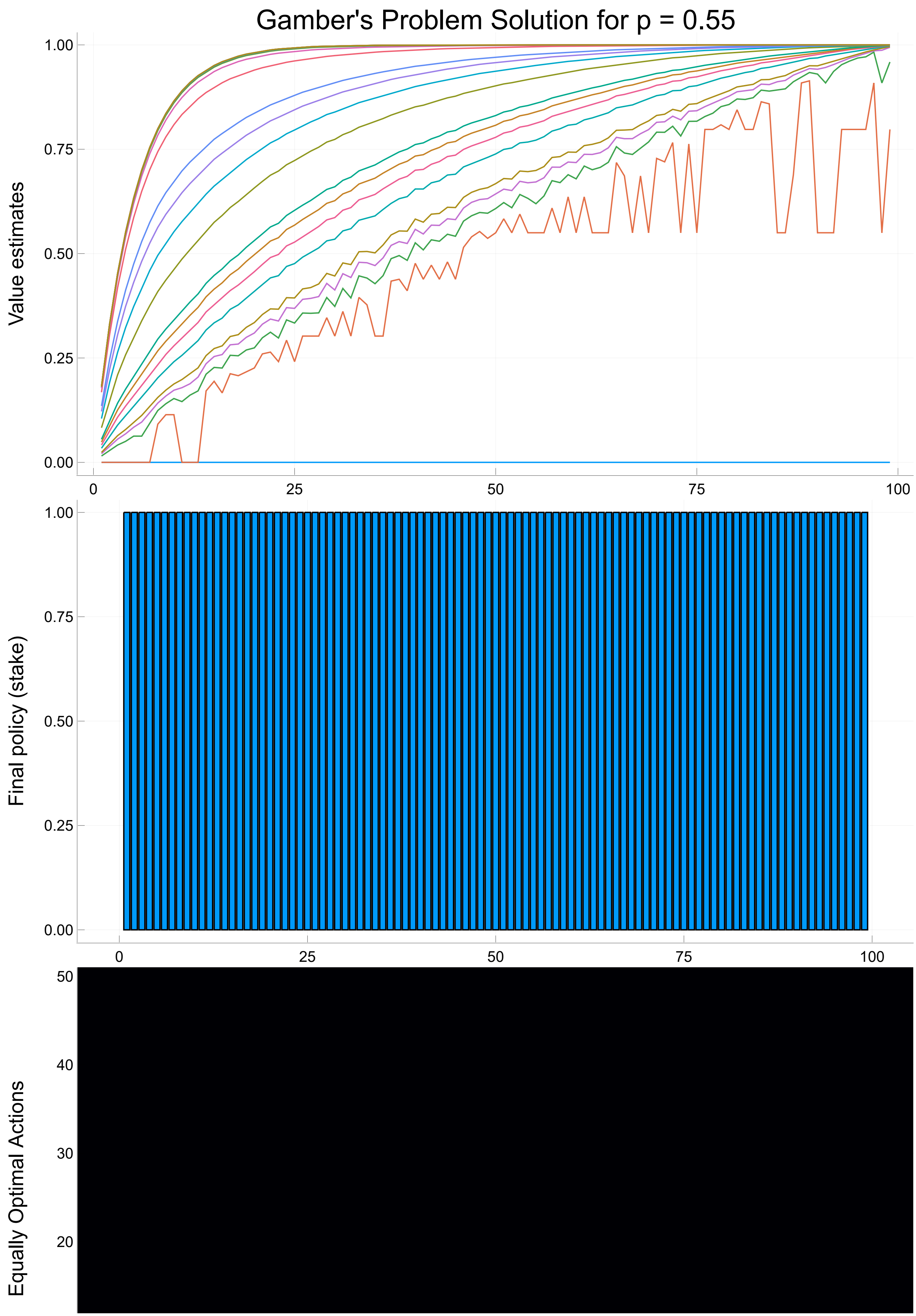
See code in the section for Example 4.3, below are plots for the desired p values. In both cases, as the tolerance is made arbitrarily low the value estimates converge to a stable curve. For $p_h > 0.5$ the curves are smoother as the policy and solution are more predictable.



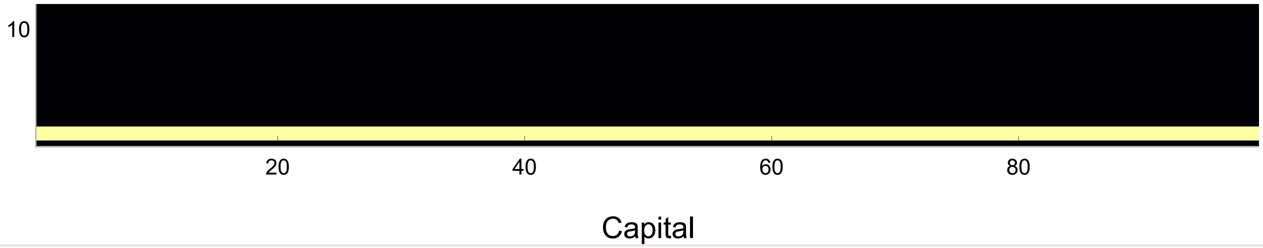
Gamber's Problem Solution for p = 0.25
Value estimates
0.0
0.2
0.4
0.6
0.8
0
25
50
75
100
Final policy (stake)
0
10
20
30
40
50
0
25
50
75
100
Equally Optimal Actions
20
30
40
50




```
plot_gambler_results(0.25)
```



Gamber's Problem Solution for p = 0.55
Value estimates
1.00
0.75
0.50
0.25
0.00
0
25
50
75
100
Final policy (stake)
1.00
0.75
0.50
0.25
0.00
0
25
50
75
100
Equally Optimal Actions
50
40
30
20




```
plot_gambler_results(0.55)
```

> *Exercise 4.10* What is the analog of the value iteration update (4.10) for action values, $q_{k+1}(s, a)$?

Copying equation 4.10 we have

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$$

To create the equivalent for action values, we need to use the Bellman Optimality Equation for q rather than v

$$q_{k+1}(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_k(s', a')]$$