Problem Set 1

Dynamic Programming

Patrick Altmeyer

10 June, 2021

1 Policy evaluation

The first thing to do is to define the Markov Decision Process (MDP). It is useful to implement the MDP as its own class and the different functions and algorithms as its corresponding methods. The mdp.R script contains the code that implements all of this. To inspect the code, either open the script directly or instead inspect the companion code report. The code chunk below defines the parameters and then eventually uses them to define the MDP.

```
N <- 100
# State space:
state_space <- seq(0,N-1,by=1)
# Action space:
action_space <- c(0.51, 0.6)
# Reward function:
reward_fun <- function(state, action, state_space) {
  reward <-  (-1) * ((state/max(state_space+1))^2 + ifelse(action==0.51,0,0.01))
}
# Transition function:
transition_fun <- function(new_state,state,action, p) {

  # Standard rule:
  state_diff <- data.table(new_state = new_state, state = state, q=action)
  state_diff[,diff:=new_state-state]
  state_diff[,prob:=0] # initialize all as zero
  state_diff[diff==1,prob:=p*(1-q)]
  state_diff[diff==0,prob:=p*q + (1-p)*(1-q)]
  state_diff[diff==-1,prob:=(1-p)*q]

  # Boundary cases:
  state_diff[state==min(state) & new_state==min(state),prob:=1 - p * (1-q)]
  state_diff[state==max(state) & new_state==max(state),prob:=1 - (1-p) * q]

  return(state_diff$prob)
}
# Discount factor:
discount_factor <- .9
# Additional arguments:
p <- 0.5
# Define the MDP:
mdp <- define_mdp(
```

```
  state_space = state_space,
  action_space = action_space,
  reward_fun = reward_fun,
  transition_fun = transition_fun,
  discount_factor = discount_factor,
  p=p
)
```

The following to chunks first define our two policies (lazy and aggressive), then they apply the policies and finally evaluate them using the closed-form solution for the limit of the power iteration. Figure 1 plots the difference between the value of the lazy and the aggressive policy. Evidently, the lazy policy outperforms the aggressive policy for states around 50-60, but then loses significantly for later states around 70-90 up until the terminal state. This is intuitive: for long queue lengths it should pay off to take high action, but due to the associated cost there a risk of overshooting when taking the high action to early. This is exactly what happens in the range 50-60. It also only pays off to take high action until the terminal state, where the length cannot increase any more anyway, so incurring the cost will never lead to higher reward.

```
# Policies:
lazy <- function(state, action_space) {
  action <- rep(action_space[1], length(state))
  return(action)
}

aggressive <- function(state, action_space) {
  action <- ifelse(state<50, action_space[1], action_space[2])
  return(action)
}
```

```
# Lazy:
policy_lazy <- lazy(state = mdp$state_space, action_space = mdp$action_space)
V_pi_lazy <- evaluate_policy(mdp, policy = policy_lazy)
# Aggressive:
policy_aggr <- aggressive(state = mdp$state_space, action_space = mdp$action_space)
V_pi_aggr <- evaluate_policy(mdp, policy = policy_aggr)
```

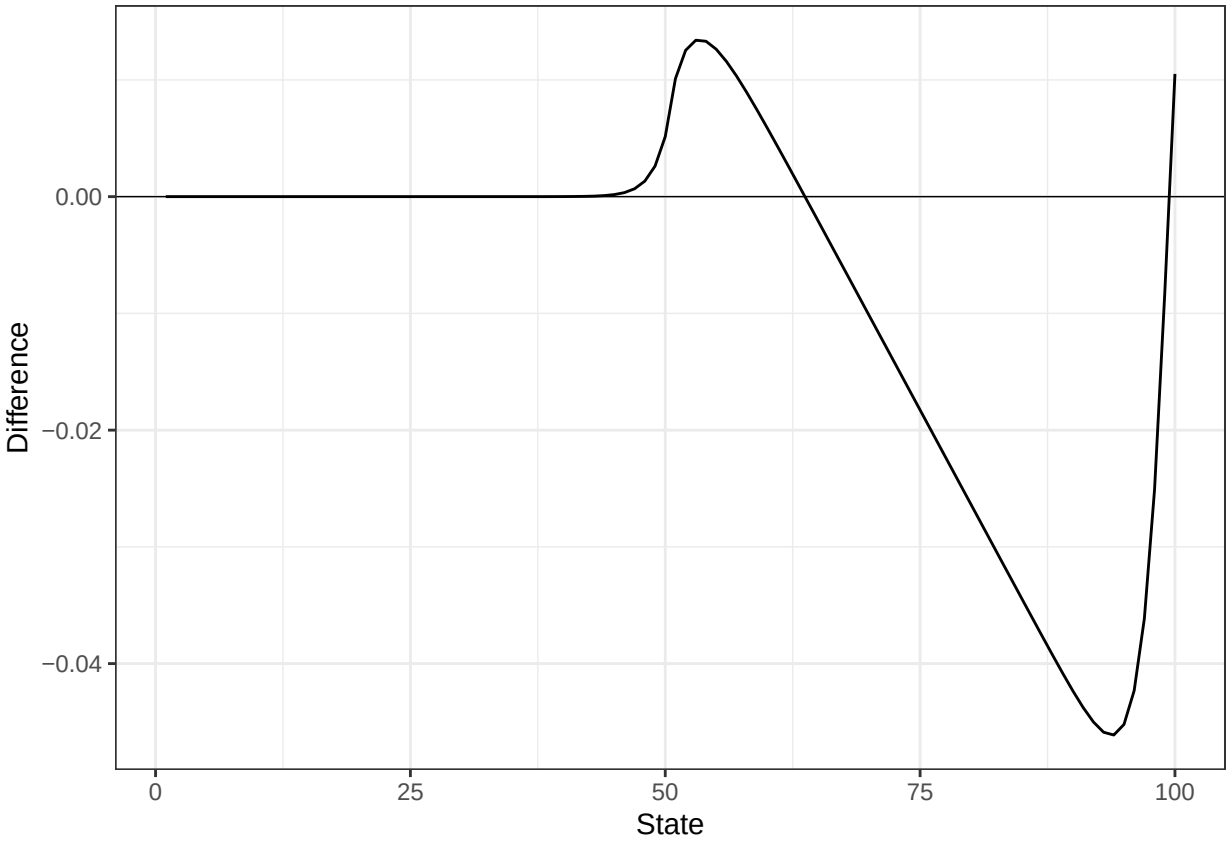

Figure 1: Comparison of lazy and aggressive policy.

2 Value Iteration and Policy Iteration

Now we will estimate the optimal policy both through Value Iteration and Policy Iteration. The key steps of **Value Iteration** involve the following

```
# 1.) Value function:
V_new <- policy_improvement(mdp, V, output_type = "value")

# 2.) Observe and update:
delta <- min(delta, max(abs(V_new-V)))
V <- V_new
iter <- iter + 1
finished <- (delta < accuracy) | (iter == max_iter)
```

where `accuracy` defines how accurate we want the estimation to be as in Sutton and Barto (2018). Note that Value Iteration (VI) does not involve a full evaluation sweep at each iteration. Conversely, evaluation forms one of the key steps of **Policy Iteration** which are as follows

```
# 1.) Policy evaluation:
V <- power_iteration(mdp, policy, V, accuracy = accuracy)

# 2.) Policy improvement:
policy_proposed <- policy_improvement(mdp, V)

# 3.) Check if stable:
policy_stable <- policy == policy_proposed
iter <- iter + 1
finished <- all(policy_stable) | iter == max_iter
```

where as in Sutton and Barto (2018) convergence is reached when the policy is unchanged (stable) from one iteration to the next. As mentioned above the full code can be inspected in the corresponding report.

> NOTE: I follow Sutton and Barto (2018) as closely as possible. An alternative approach would be to simply run iterate N times and not use accuracy and stability as convergence criteria. In that case one would use only one power iteration to evaluate the policy in PI and iterate exactly N times through the different sweeps in both PI and VI. This approach will also eventually converge to the optimal policy and value function. It may be closer aligned to the one in the slides. The code I put together here can be easily adapted to instead use that approach.

Figures 2 and 3 show the results of running Value and Policy Iteration, respectively, for different numbers of iterations. Figure 2 illustrates nicely how VI gradually learns the optimal policy. It appears that for a desired accuracy of 10^{-4} VI finds a close-to-optimal policy and corresponding value function after 50 iterations. An inspection of the results showed that convergence with respect to the accuracy criterion was reached after 90 iterations (see Table 1).

The optimal policy and corresponding value function returned by VI corresponds exactly to the one returned by Policy Iteration (PI). But note that with PI convergence is reached even when the maximum number if iterations is set to 10, so all panels in Figure 3 look similar. In general, Policy Iteration needs only for 3 to 4 iterations to converge (Table 1). The variation of the numbers largely varies because the initial policy is random and depending on how good a bad the initial guess is, it may take one iteration less or more to converge.

While PI clearly takes less iterations than VI to reach convergence, this comes at a cost: evaluating the policy at each step through power iteration with the desired level of accuracy is computationally costly. Consequently the computational run times of the two different methods are not as much in favour of PI as one might have expected (Figure 4).[1]

[1] I found that simply using the closed-form solution at led to a massive reduction in run time for PI.

Table 1: Number of iterations until convergence or timeout is reached.

Value Iteration	Policy Iteration
10	3
20	3
50	4
90	3

With respect to the estimated optimal policy Figure 5 plots the actual choices for each state and Figure 6 demonstrates the associated improvement in value compared to the lazy and aggressive policies investigated above. The results are very intuitive:

1. The lazy policy can be improved mainly around states 60-90, when it pays off to choose high action in light of the long queue.
2. The aggressive policy can be improved by avoiding to overshoot (states 50-60) and avoiding the wasteful effort in the terminal state.

These two points correspond to what the estimated optimal policy prescribes and therefore yield to improvement in value.

Finally another note on initial values: above the value function was initialised as -10 across as states. This is not necessary as Figure 7 illustrates. Here the initial values were drawn from a Gaussian distribution with mean -10.

References

Sutton, Richard S, and Andrew G Barto. 2018. *Reinforcement Learning: An Introduction.* MIT press.

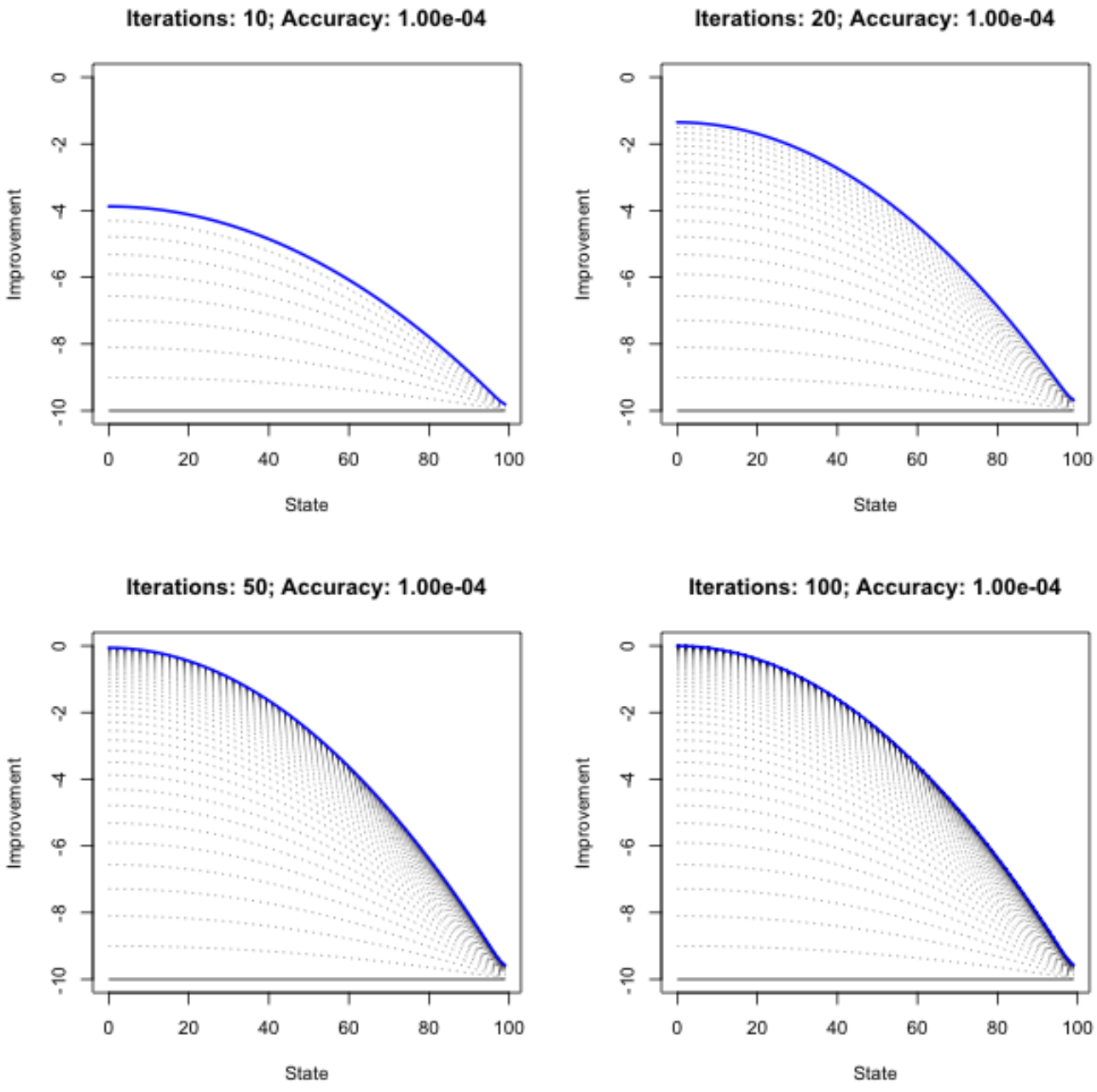

Figure 2: Value Iteration with 10, 20, 50 and 100 iterations. The blue line represents the value function corresponding to the final estimate of the optimal value function.

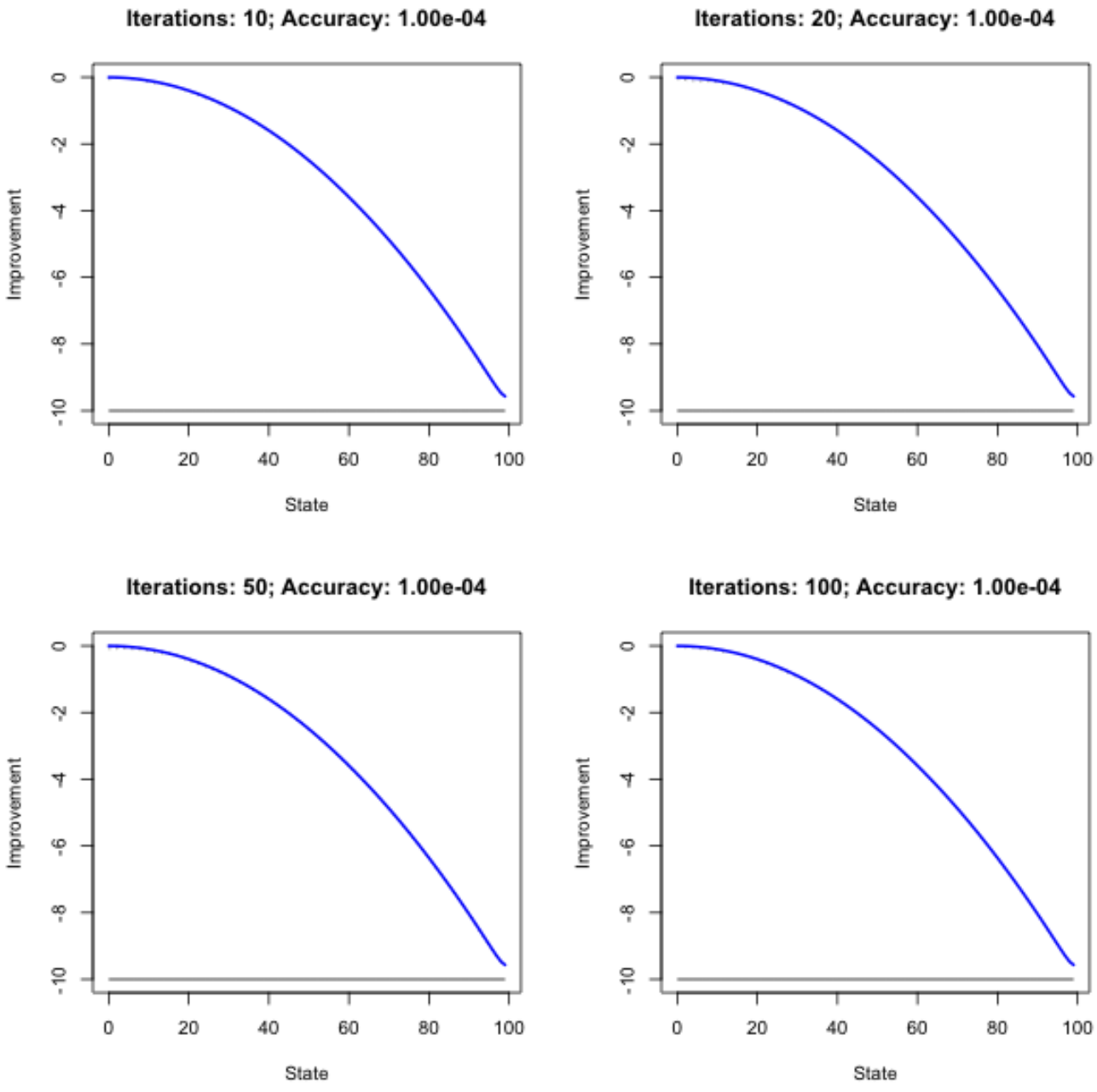

Figure 3: Policy Iteration with 10, 20, 50 and 100 iterations. The blue line represents the value function corresponding to the final estimate of the optimal value function.

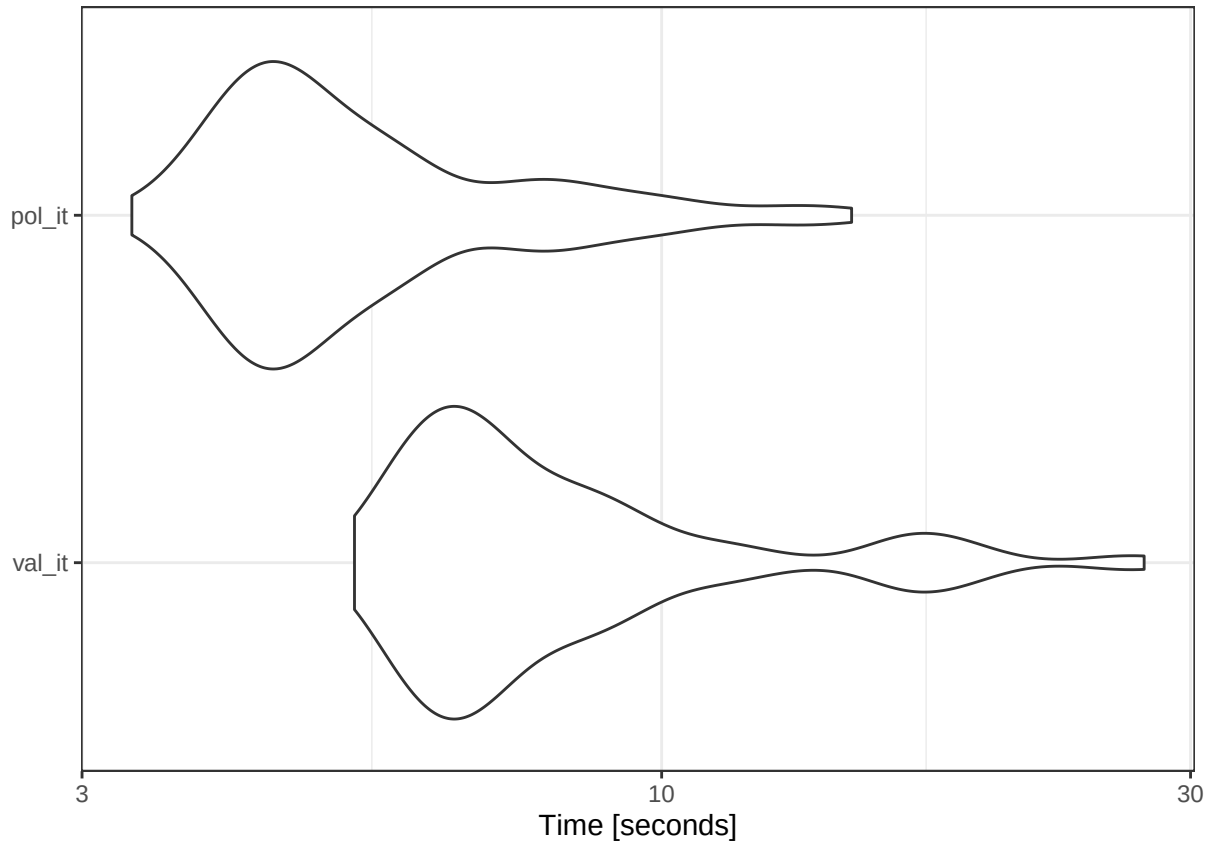

Figure 4: Microbenchmark of computational times for Value Iteration and Policy Iteration.

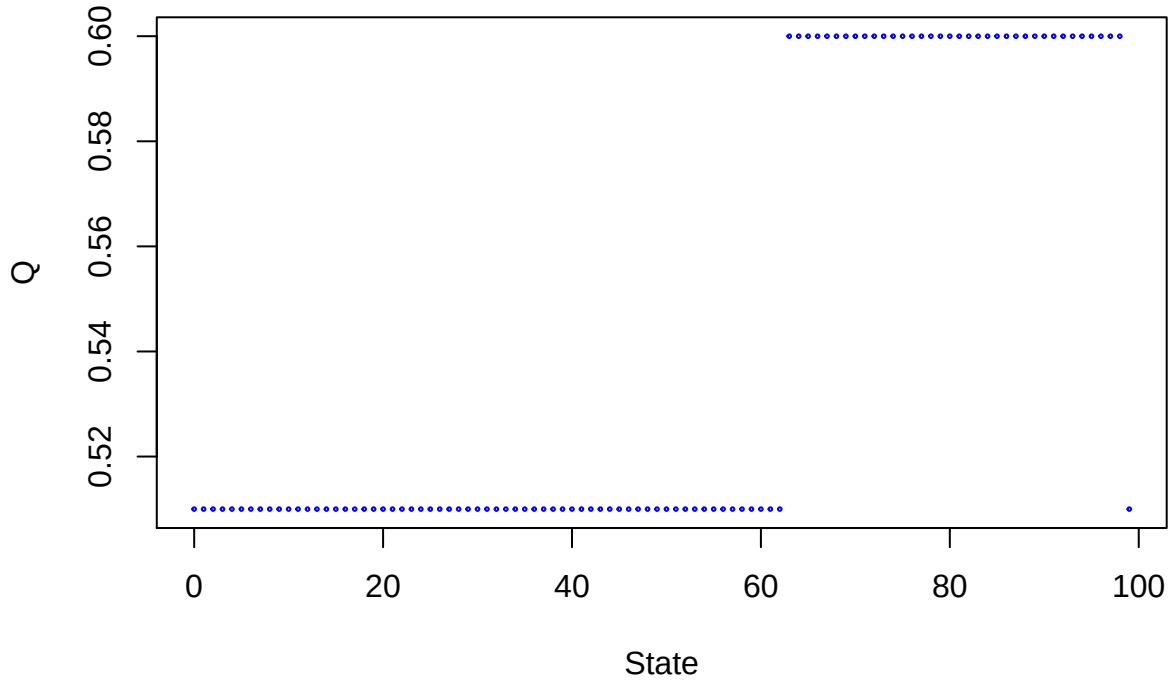

Figure 5: The estimated optimal policy.

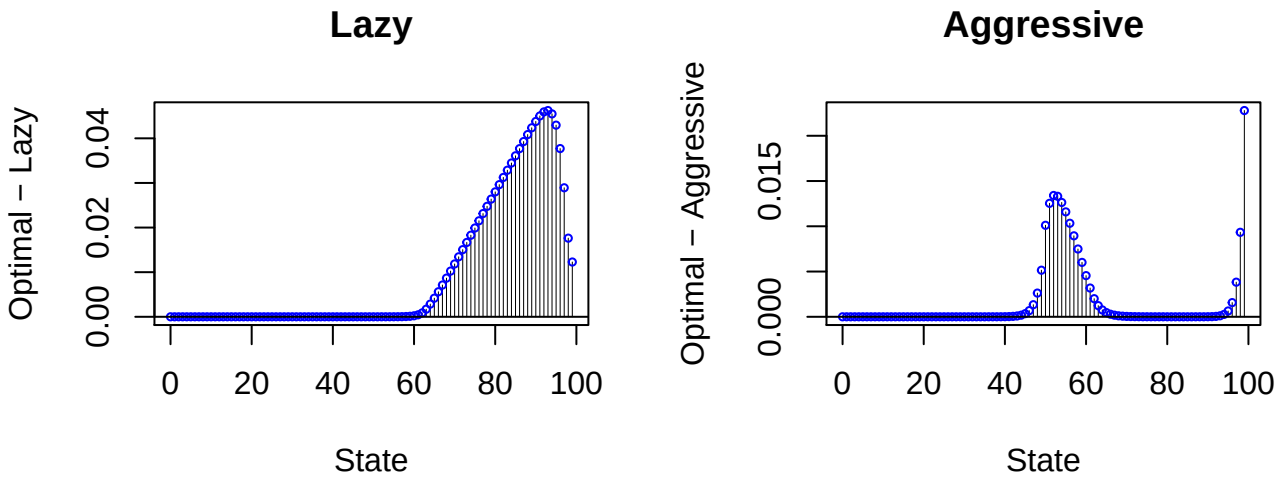

Figure 6: Improvement of estimated optimal policy compared to lazy and aggressive policy.

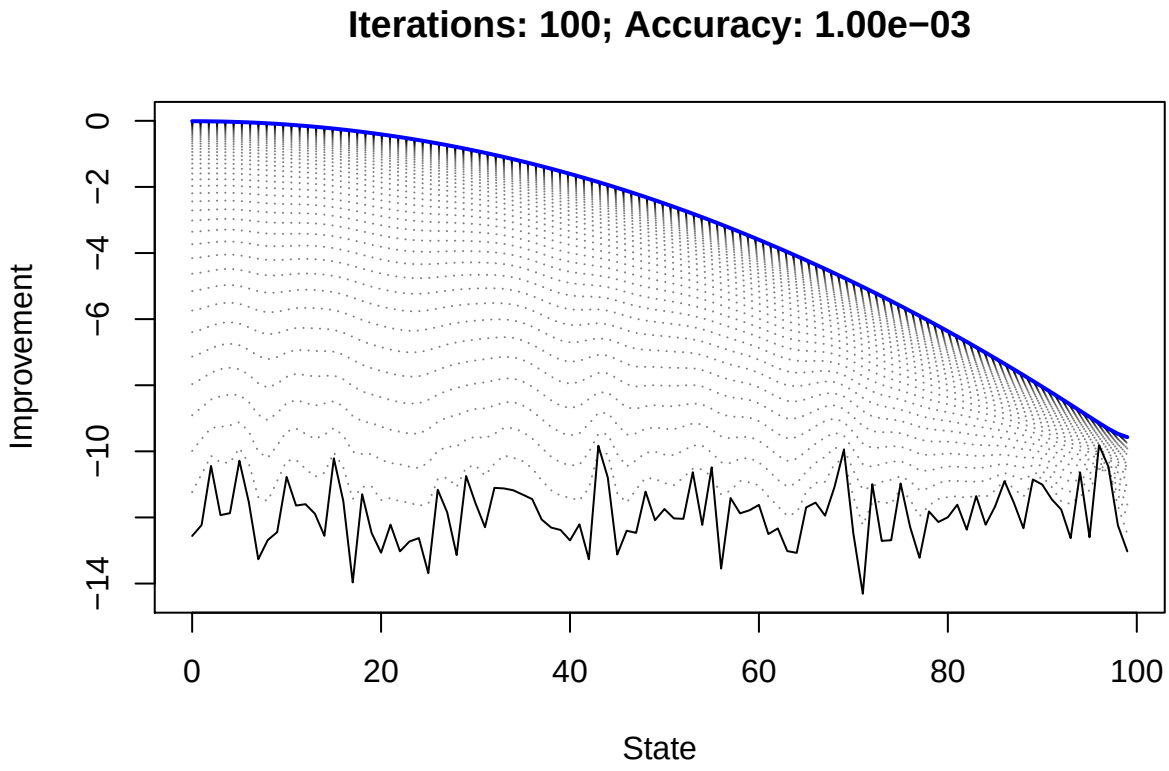

Figure 7: Initializing with random values.