Problem Set 1

Dynamic Programming

Patrick Altmeyer

09 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 very long queue lengths it pays off to take high action (until it does not any more in the terminal state, where the length cannot increase any more anyway).

```
# 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)
```

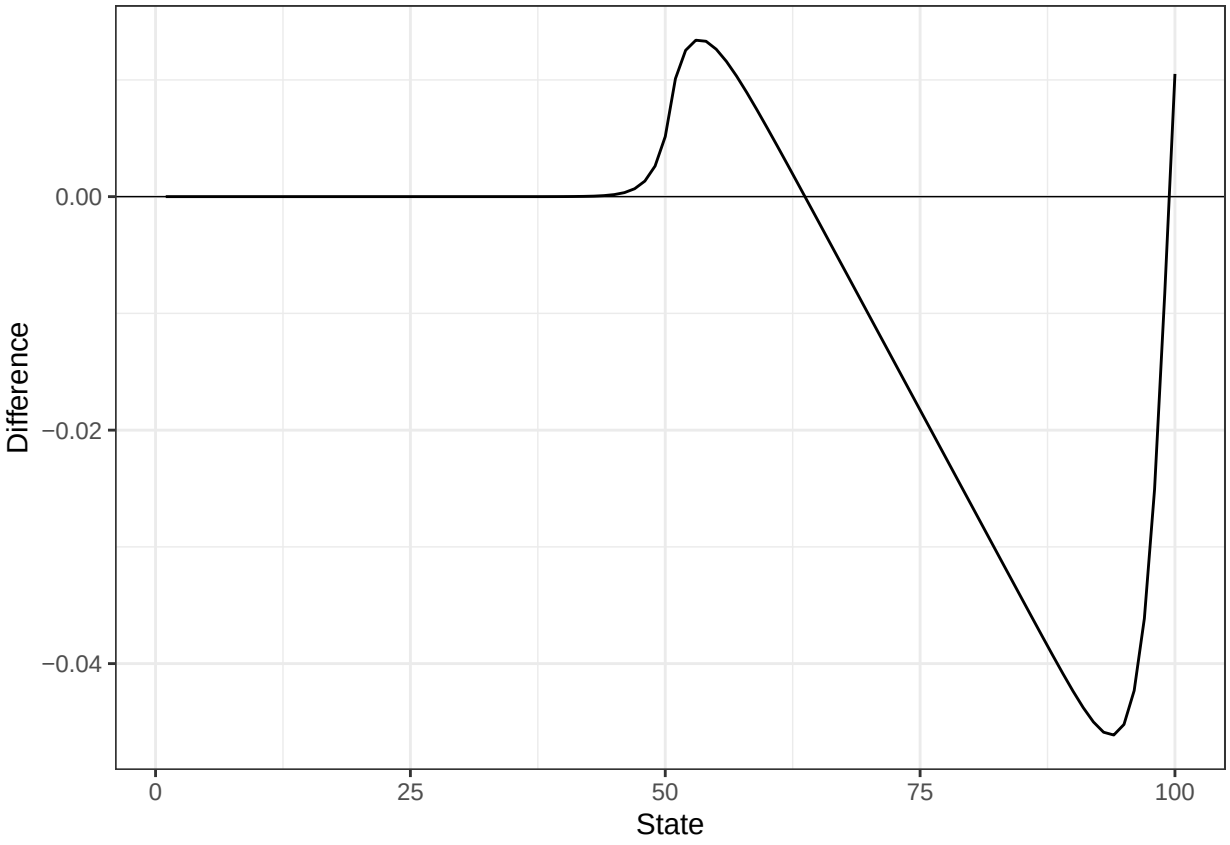

Figure 1: Comparison of lazy and aggressive policy.

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 (**sutton2018reinforcement?**). The key steps of **Policy Iteration** are the following

```
# 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 (**sutton2018reinforcement?**) convergence is reached when the policy from one iteration to the next is stable. As mentioned above the full code can be inspected in the corresponding report.

Figures 2 and 3 show the results of running Value and Policy Iteration, respectively, for different numbers of iterations. Figure 2 illustrates nicely how Value Iteration gradually learns the optimal policy.

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

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

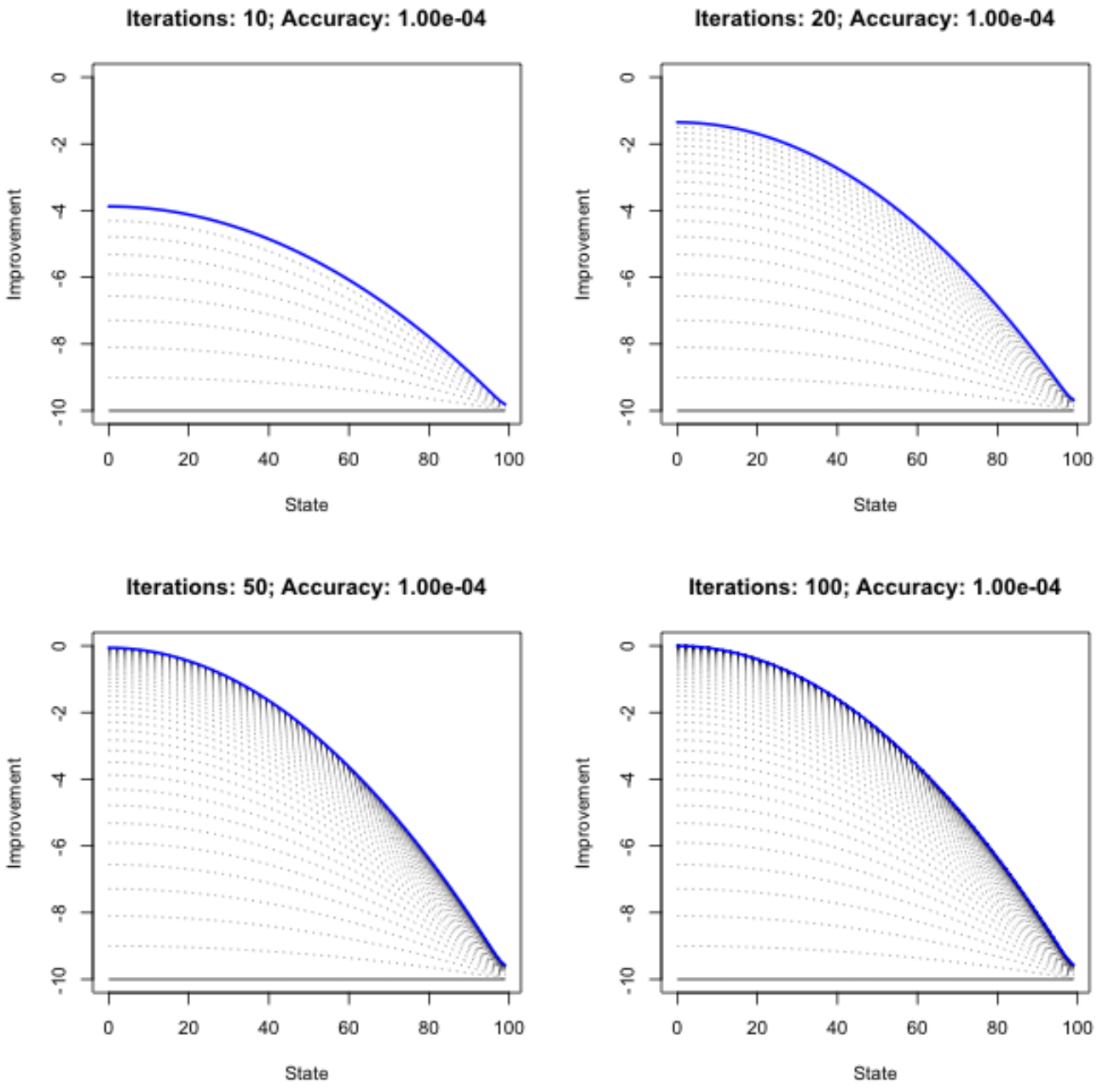

Figure 2: Value Iteration with 10, 20, 50 and 100 iterations. The blue line represents the value function corresponding to the final estimate of the optimal value function.

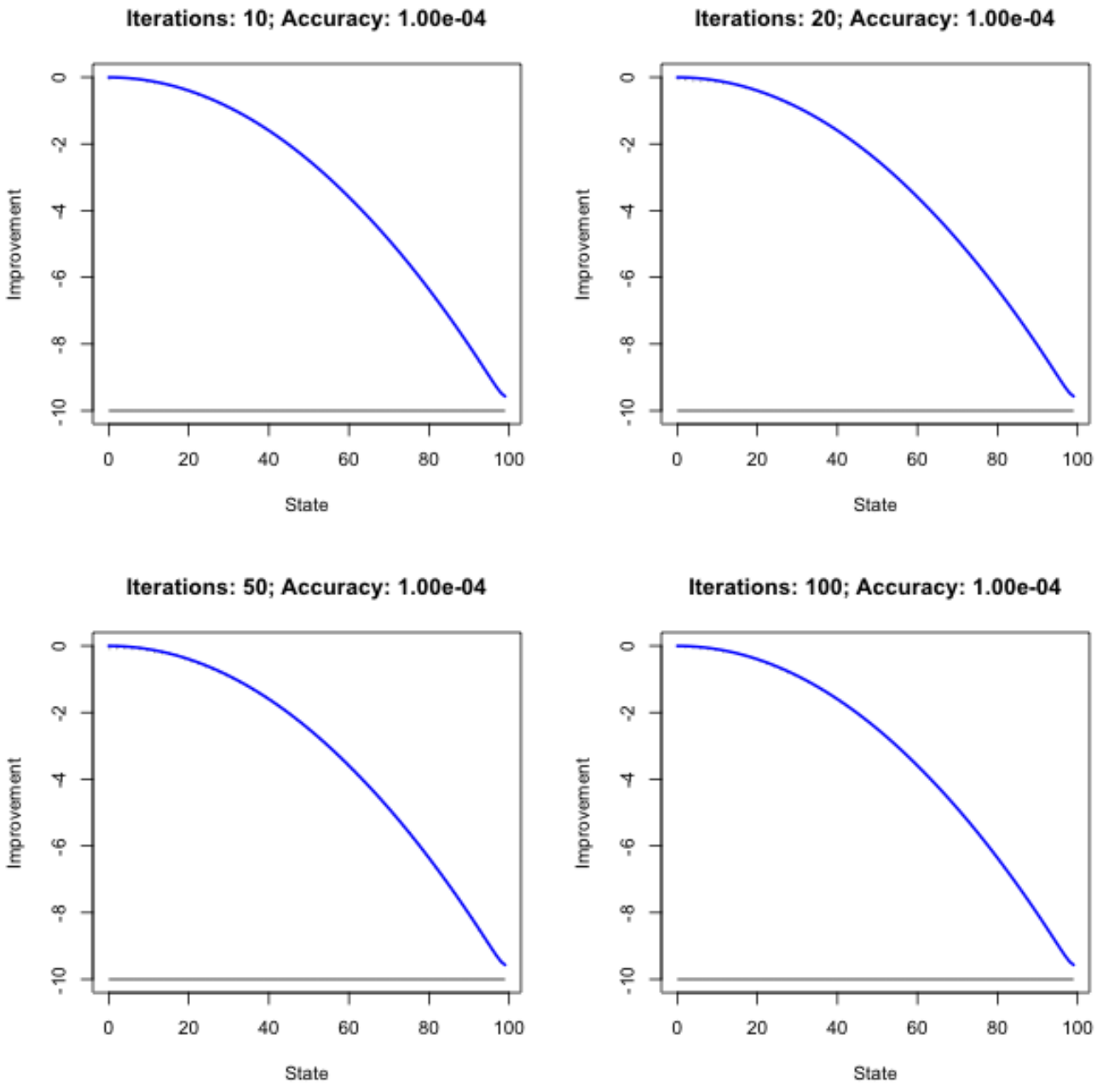

Figure 3: Policy Iteration with 10, 20, 50 and 100 iterations. The blue line represents the value function corresponding to the final estimate of the optimal value function.

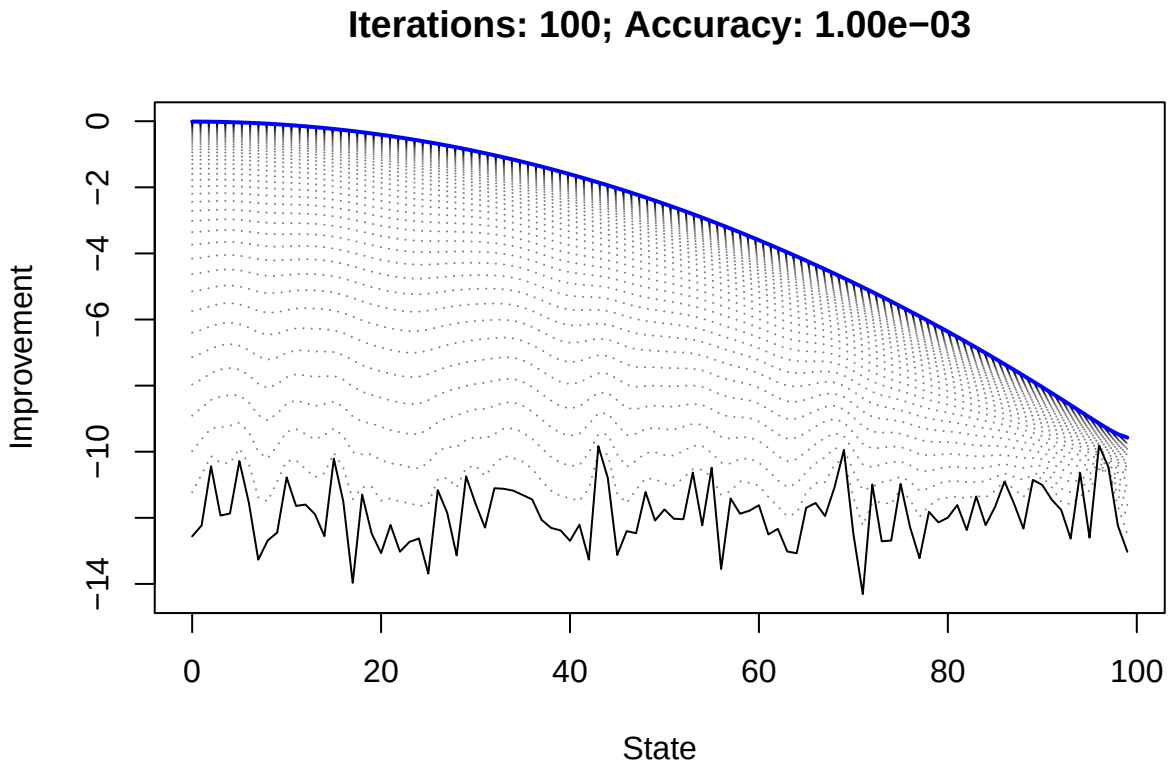

Figure 4: Initializing with random values.

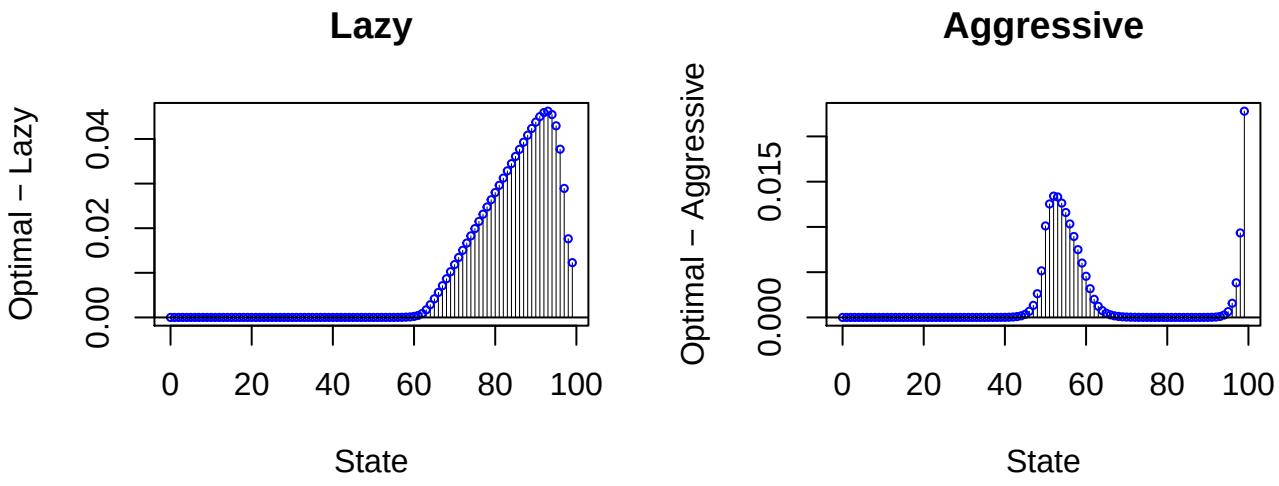

Figure 5: Improvement of estimated optimal policy compared to lazy and aggressive policy.

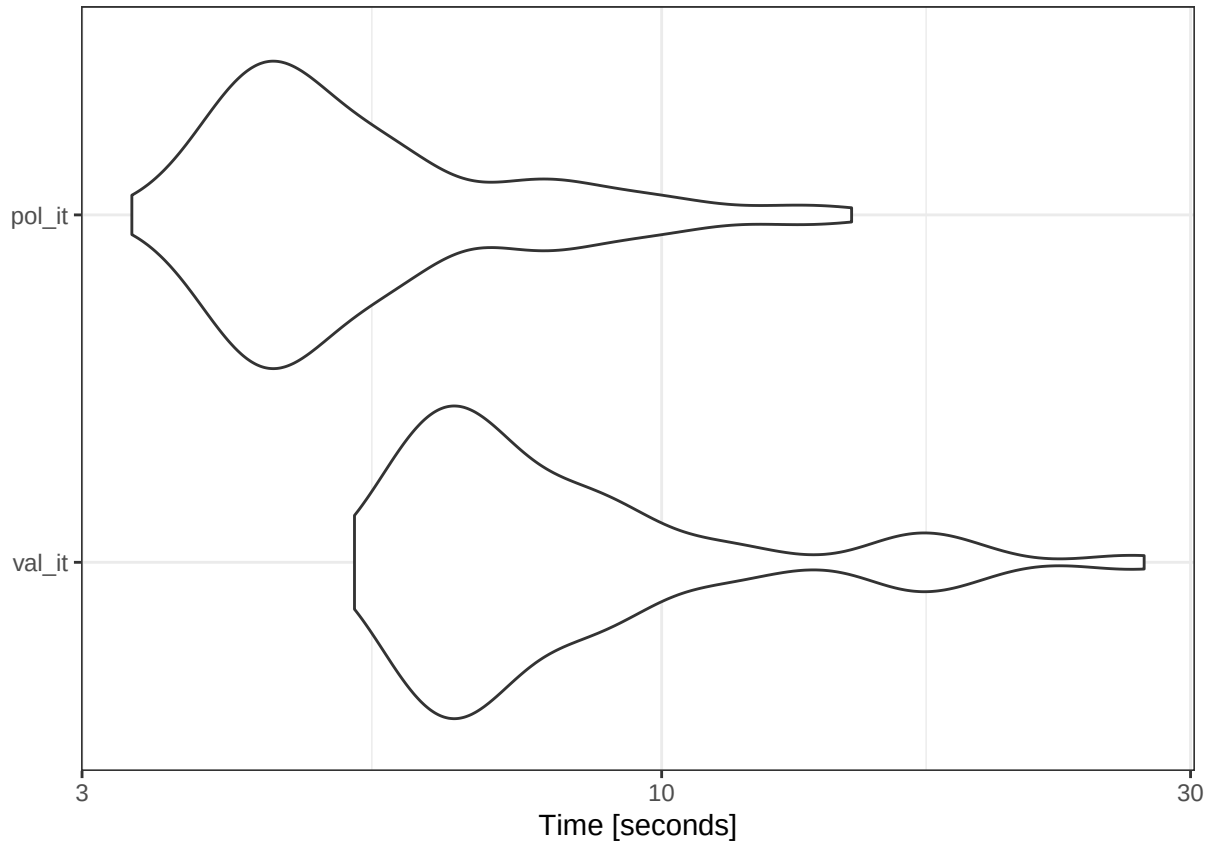

Figure 6: Microbenchmark of computational times for Value Iteration and Policy Iteration.