TECHNISCHE UNIVERSITÄT MÜNCHEN
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl fr Datenverarbeitung
Prof. Dr.-Ing. K. Diepold

Approximate Dynamic Programming & Reinforcement Learning

Programming Assignment 3

Laila Niazy
Matrikelnr. 03660840

January 30, 2019

Probabilistic Formulation of the Problem

- Description of my implementation of the transition probabilities:

  For implementing the transition probabilities, I used a nested dictionary (a dictionary inside a dictionary). The first dictionary has the current state s_k of the agent as the key and the corresponding value of the key is the possible allowed actions the agent can take at s_k. All actions that are not possible were filtered out in another function due to the successor states being a wall or outside the maze. Moreover, these allowed actions were used as the keys of another dictionary within the first, where the corresponding value is a tuple of the successor state and the probability of landing in that successor state. The output I get in the end is resembles this:

  $$T[s_k][a] = [(s_{k+1}^{(i)}, p^i), (s_{k+1}^{(i+1)}, p^{(i+1)})...], \tag{1}$$

  where i denoted the number of possible successor states $s_{k+1}^{(i)}$, since we have slippery maze, one action maps to three successor states.

- What are possible problems of a naive implementation of $p_{ij}(u)$ for all possible combinations of i, j and u?

  Two large problems that can arise due to naive implementation are memory overhead and CPU overhead. Memory overhead means that due to the many variables needed to get the final transition probabilities, one might run of memory. Furthermore, normally a program uses a lot of short methods, which makes it easier to understand and shortens the amount of code. However, when each method is called, a return address is needed and the parameters are copied etc. This represents CPU overhead when comparing to a program, which does everything in a single function.

Policy Description of my implementation of the policy:

For implementing the policy, I again used a dictionary. The current state s_k was the key, while the corresponding value was the policy $\mu(s_k)$. All states were initialized in the beginning of each algorithm with a random action, which is the current policy. This random action was taken from a list containing all allowed actions for this state. Further, depending on the optimality of the policies chosen, the dictionary with the current policy was updated using the list of allowed actions.

Solution with Dynamic Programming

- How did you represent the state space?

  For the state space, I used a nested list. The values inside the list are strings, which can take any of the following values: 1,0,G,S or T. These strings are read from the text-file of the maze. The indices of the nested list are the different states the agent can be in.

- How can you determine if the DP algorithms have converged?

  For determining how the algorithm converged, I subtracted the current value function from the old value function after each iteration and if the outcome was lower than $\frac{\epsilon \cdot (1-\gamma)}{\gamma}$ with $\epsilon = 0.001$, then the value function has converged. The fulfilled of this criteria stops the algorithm and outputs the optimal value function with the optimal policy. This approach is based on the monotonic error bounds of the value function and uses this proposition to obtain an optimal or near-optimal policy in a finite number of iterations. Further, this method was taken from the book [1], which contains the proof of the suitability of this approach.

- Visualize V and μ in a 2d plot.

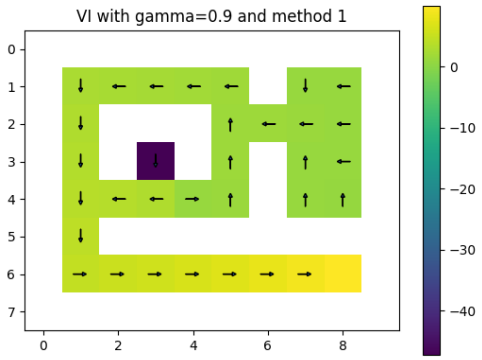
  In Fig.1, four different plots show the heatmap of the results of VI and PI using the the two reward methods with γ being 0.9. The colorbar on the right hand side of each map illustrates the computed values of the value function. If any of the colored squares in the heatmap is colored but does not contain an arrow that means the generated policy is ideal.

- Do the two methods generate the same policy?

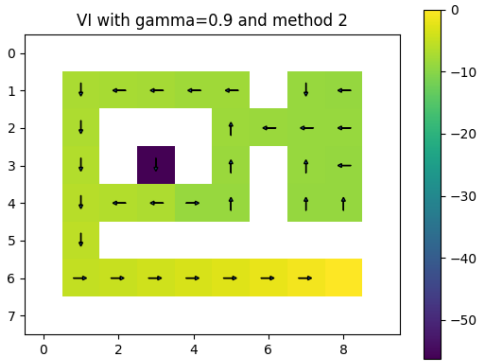
  As expected both iteration algorithms generate the same policy and value function, which can be seen Fig.1. This is due to both of them being based on the optimal Bellman equation but applying it in different manners. In VI, the optimal Bellman operator is applied directly to the value function in a recursive manner, therefore the value function converges to the optimal value. The optimal policy is the one, which is greedy in very state with respect to the obtained optimal value function. However, policy iteration applies the Bellman equation in two steps. In the first step, which is called policy evaluation, the Bellman operator is used for the current best policy in a recursive manner until the value function converges for this policy. Then, in the next step known as policy improvement, it improves the current policy by getting the action, which maximize the value function for every state and these steps are repeated, until the policy stops improving/converges.

- Has the changed γ an influence on the final policy?

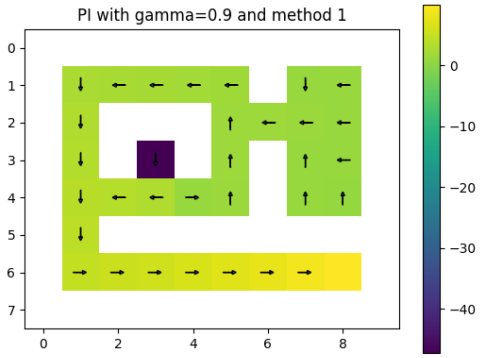
  Yes indeed it does, as can be seen in Fig.2. By adjusting γ, one can determine if the agent should prefer immediate rewards rather than rewards that are potentially received far away in the future or vice versa. The lower the γ, the more the agent will prefer immediate rewards and the higher the γ, the more the agent will prefer



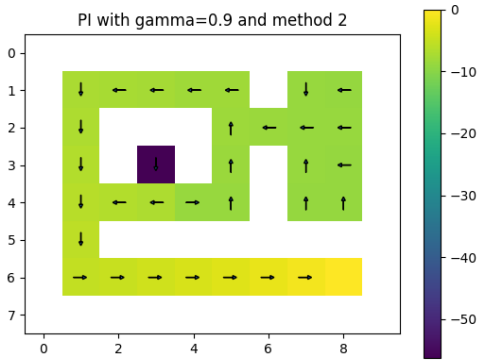
(a) VI with reward method 1



(b) VI with reward method 2



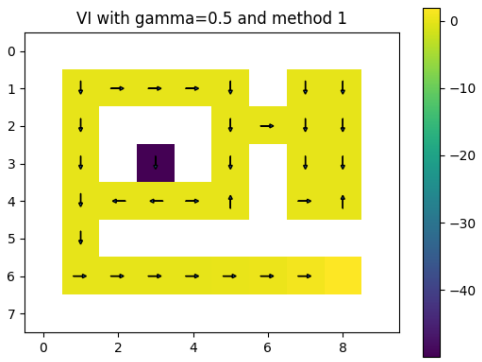
(c) PI with reward method 1



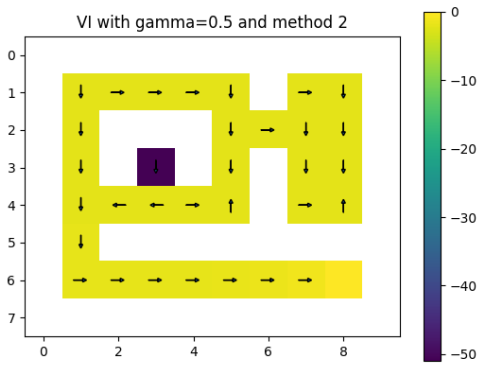
(d) PI with reward method 2

Figure 1: Visualization of V and μ for $\gamma = 0.9$

to wait to receive the rewards in the future. This influence can be seen clearing when looking at the policies generated under the trap in plot (a) and (e) of Fig.2.

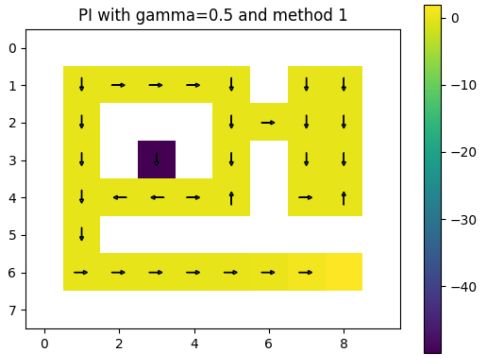


(a) VI with reward method 1 and $\gamma = 0.5$

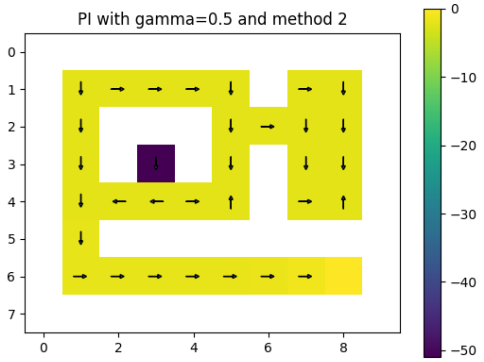


(b) VI with reward method 2 and $\gamma = 0.5$

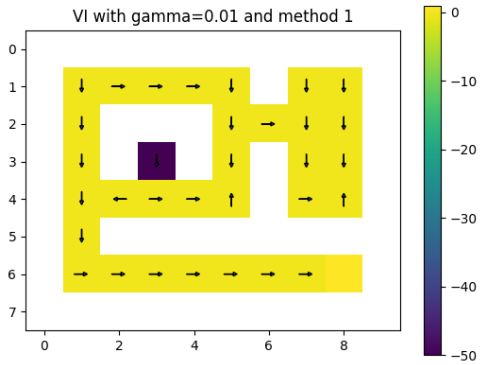
Figure 2: Visualization of V and μ for varying γ



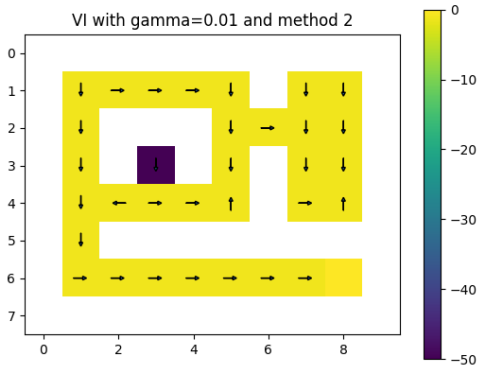
(c) PI with reward method 1 and $\gamma = 0.5$



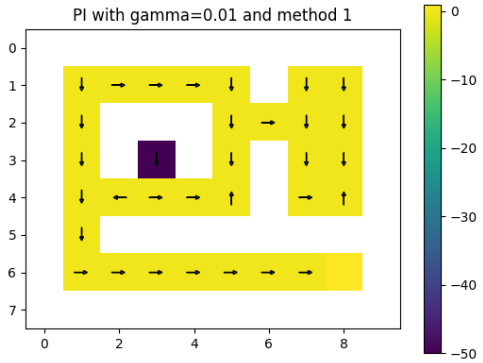
(d) PI with reward method 2 and $\gamma = 0.5$



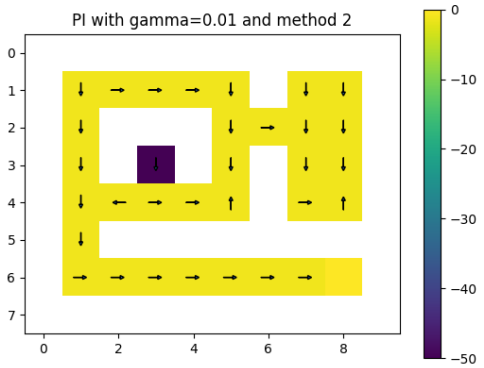
(e) VI with reward method 1 and $\gamma = 0.01$



(f) VI with reward method 2 and $\gamma = 0.01$



(g) PI with reward method 1 and $\gamma = 0.01$



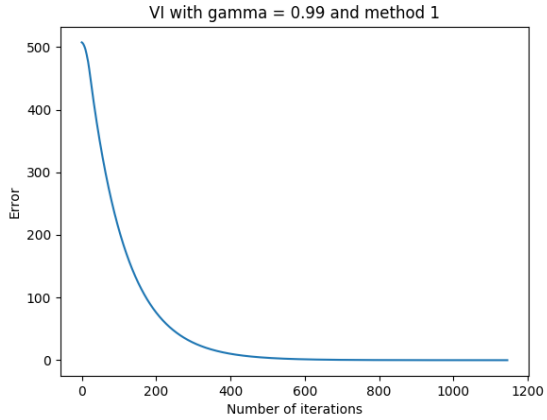
(h) PI with reward method 2 and $\gamma = 0.01$

Figure 2: Visualization of V and μ for varying γ

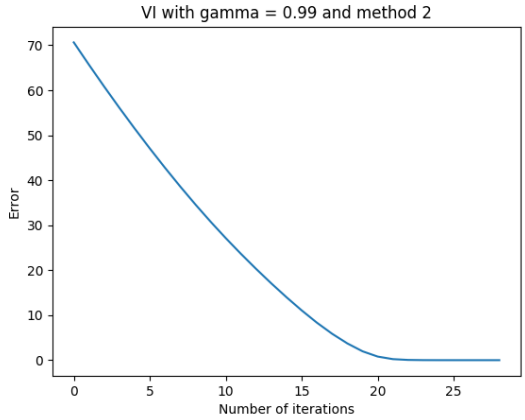
A Study of Algorithm Properties

- Plotting the error (i.e. the square distance) for both algorithms.

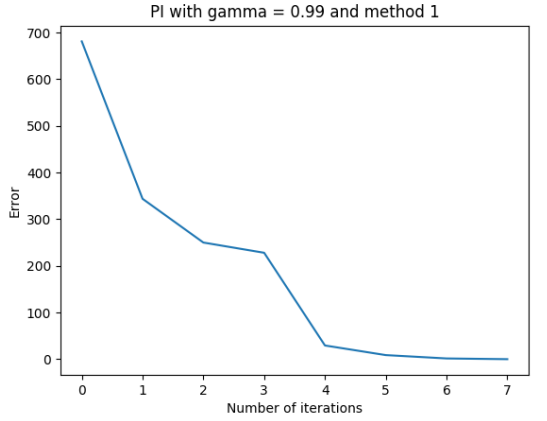
  In Fig.3, there are four plots for both algorithms using both reward methods with $\gamma = 0.99$. The plots depict the decrease of the error (y-axis) in each iteration (x-axis).



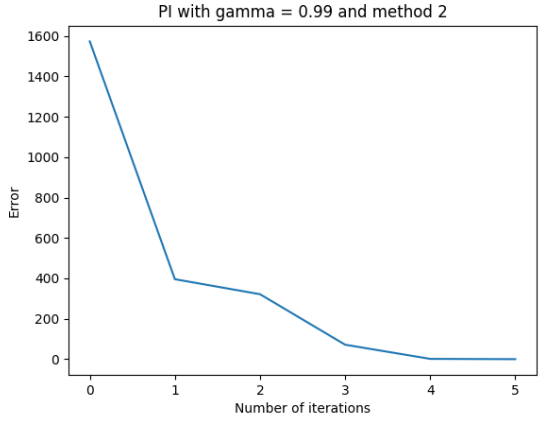
(a) VI with reward method 1



(b) VI with reward method 2



(c) PI with reward method 1



(d) PI with reward method 2

Figure 3: Error plots with $\gamma = 0.99$

- Create these error plots for three suitable values of γ. Which values did you use?

  I used the following values: $0.6, 0.7, 0.8$. I choose those values because one can clearly see the difference γ has on the number of iterations and the error in relation to the ground truth. The iteration number increases with higher γ values because the termination condition I used depends on γ and makes the threshold lower with a higher γ. This results in the algorithm taking longer to fulfill the termination condition, as can be seen by comparing plots (a),(c) and (e) in Fig.5 and Fig.4. At least in the case of the value iteration, for policy iteration, it can occur that the policy converges faster than the value function, resulting in a lower number of iterations although γ is lower. This can been seen in plots (d) and (f) of Fig.5. Moreover, the

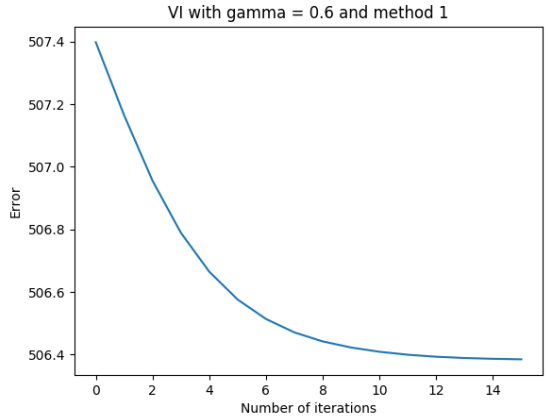
error each algorithm has on the final iteration decreases with increasing γ. This is due to the value function slowly changing to resemble the ground truth, which has a γ value of 0.99. The value function changes depending on when the agent prefers to receive his reward as explained in the final question of paragraph "Solution with Dynamic Programming".

- Is VI or PI better/faster? Why? Describe your findings.

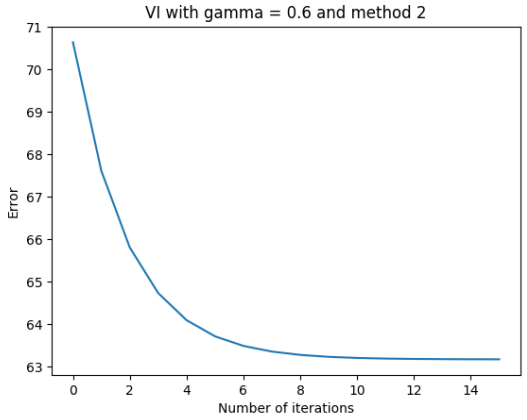
  Comparing VI and PI to each other, policy-iteration is computationally more efficient as it often takes fewer number of iterations to converge. However, each iteration is computationally more expensive, because each of these iterations requires a whole inner loop inside, which does policy evaluation (basically value iteration). As shown in plots (b) and (d) of Fig.3, policy iteration took about 5 iterations compared to the 28 iterations, which were needed by value iteration. In general, it is always expect for PI to converge in fewer iteration due to the policy converging more quickly than the value function.

References

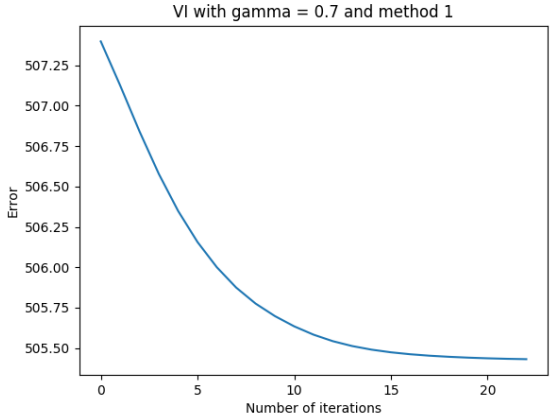
[1] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. II.* 2007.



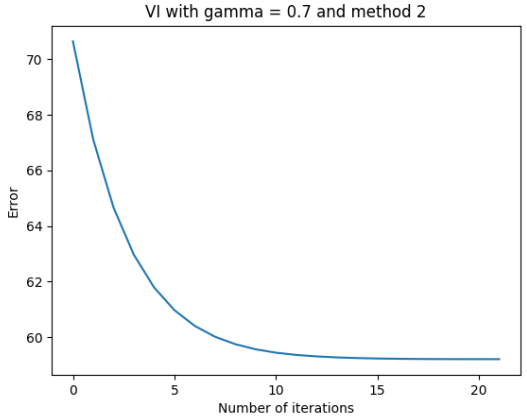
(a) VI with reward method 1 and $\gamma = 0.6$



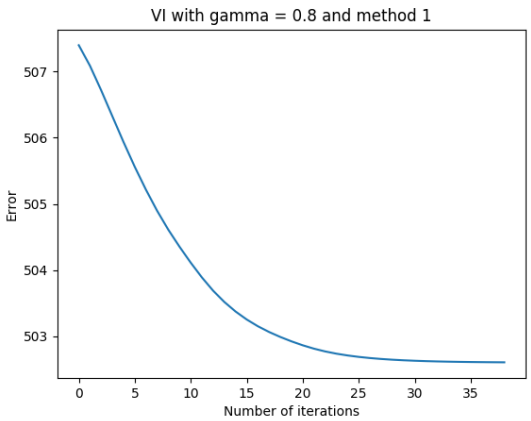
(b) VI with reward method 2 and $\gamma = 0.6$



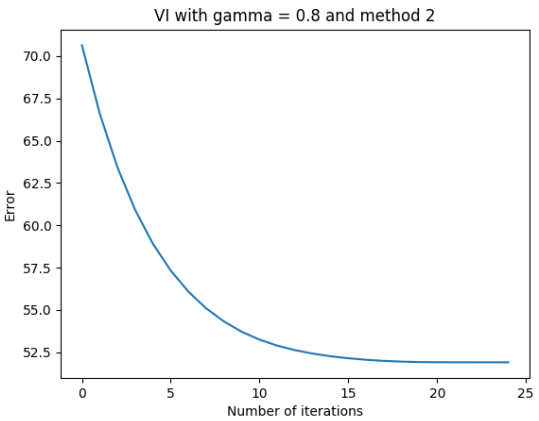
(c) VI with reward method 1 and $\gamma = 0.7$



(d) VI with reward method 2 and $\gamma = 0.7$

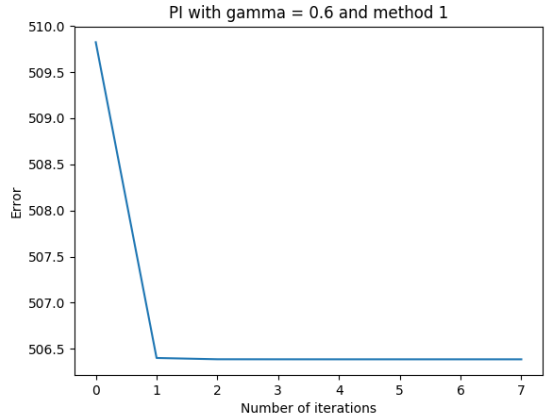


(e) VI with reward method 1 and $\gamma = 0.8$

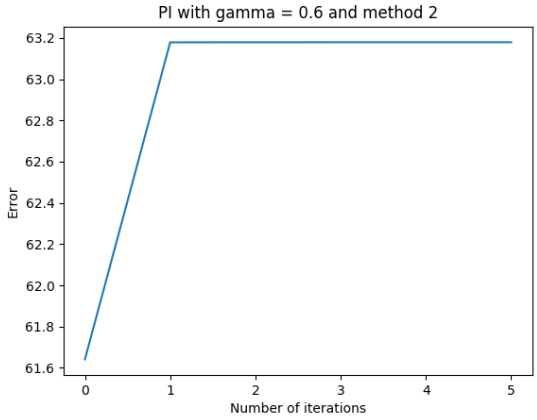


(f) VI with reward method 2 and $\gamma = 0.8$

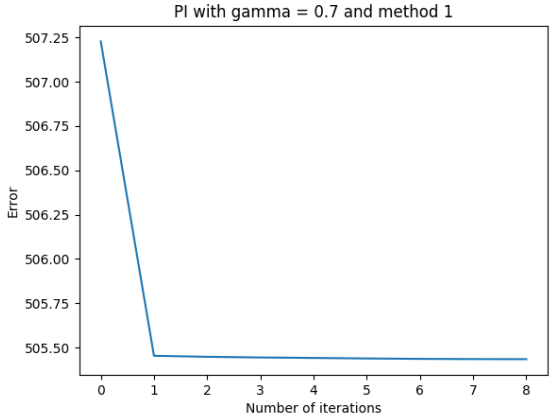
Figure 4: Error plots for VI with varying γ and reward method



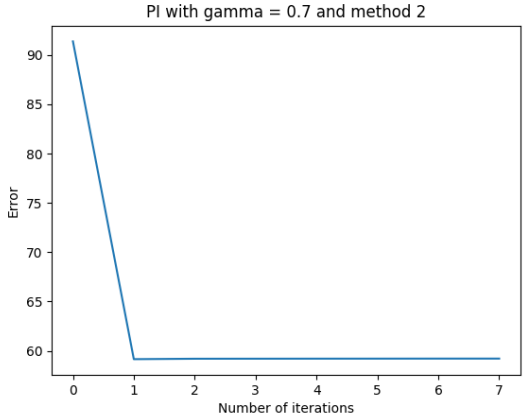
(a) PI with reward method 1 and $\gamma = 0.6$



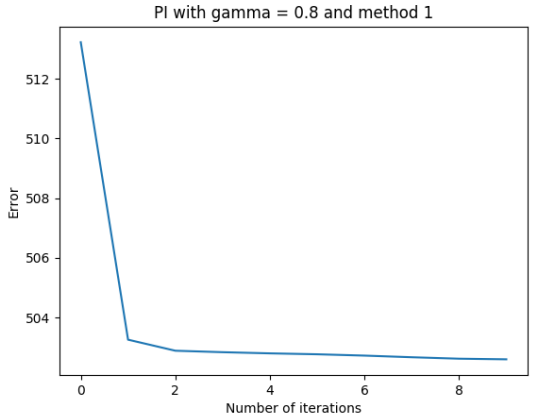
(b) PI with reward method 2 and $\gamma = 0.6$



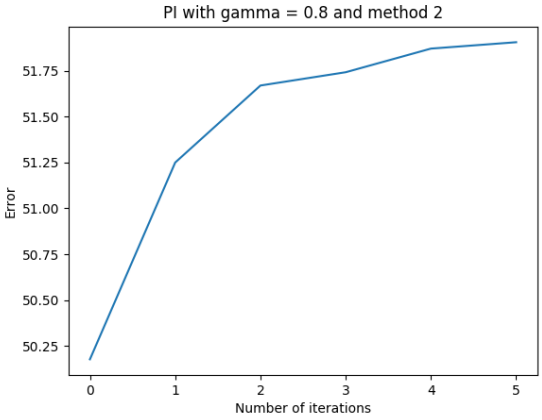
(c) PI with reward method 1 and $\gamma = 0.7$



(d) PI with reward method 2 and $\gamma = 0.7$



(e) PI with reward method 1 and $\gamma = 0.8$



(f) PI with reward method 2 and $\gamma = 0.8$

Figure 5: Error plots for VI with varying γ and reward method