HW #4: Value/Policy Iteration, Discretization

Name: Gyanig Kumar

Deliverable: PDF write-up and zip file of project directory. Your PDF should be generated by replacing the placeholder images in this LaTeX document with the appropriate solution images for each question. Your PDF and code is to be submitted into the course Canvas. The scripts will automatically generate the appropriate images, so you only need to recompile the LaTeX document to populate it with content. If you do not have/do not want to install LaTeX on your machine, you may use a website like Overleaf instead.

Graduate Students: You are expected to complete the entire assignment.
Undergraduate Students: You need only complete questions that do not have **(GRAD)** next to them. (You do not need to implement max-ent Value Iteration or look-ahead policies.)

You will need to install matplotlib and the Gymnasium Environment for Python:

```
pip install gymnasium
pip install matplotlib
```

or

```
conda install gymnasium
conda install matplotlib
```

1 Gridworld Env

1.1 Value Iteration [20pts]

First, you will implement the value iteration algorithm for the tabular case. You will need to fill the code in `code/tabular_solution.py` below the lines `if self.policy_type == 'deterministic_vi'`. Run the script for the two gridworld domains and report the heatmap of the converged values.

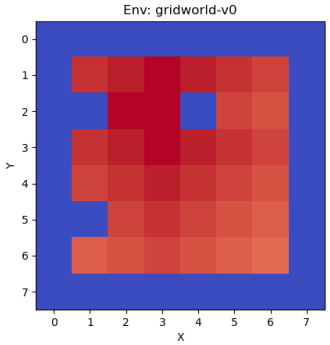


Figure 1: Value function heatmap for Gridworld (v0) using Deterministic Value Iteration

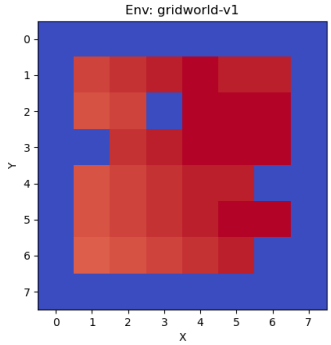


Figure 2: Value function heatmap for Gridworld (v1) using Deterministic Value Iteration

1.2 Policy Iteration [20 points]

Next, you will implement the policy iteration algorithm for the tabular case. You will need to fill the code in `code/tabular_solution.py` below the lines `if self.policy_type == 'deterministic_pi'`. Run the script for the first gridworld domain and turn in a graph with policy value (accumulated reward) on the vertical axis and iteration number on the horizontal axis.

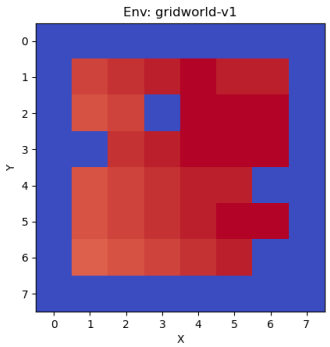


Figure 3: Value function heatmap for Gridworld (v1) using Stochastic Policy Iteration

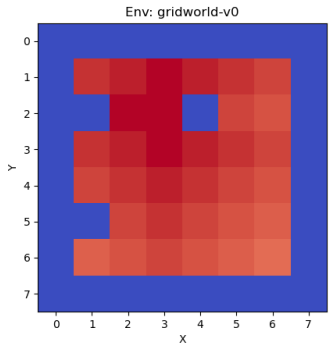


Figure 4: Value function heatmap for Gridworld (v0) using Stochastic Policy Iteration

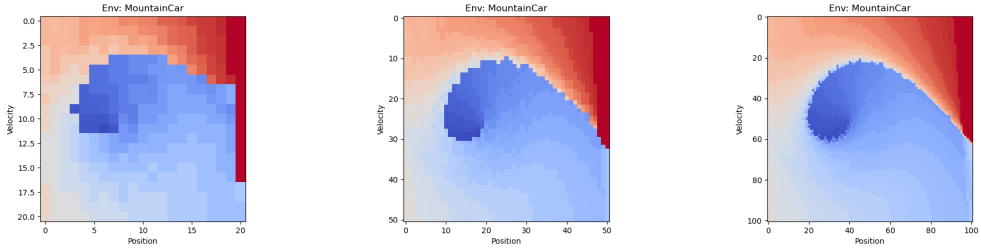
2 Mountain Car

2.1 Near neighbors interpolation [20pt]

Value Iteration can only work when the state and action spaces are discrete and finite. If these assumptions do not hold, we can approximate the problem domain by coming up with a discretization such that the previous algorithm is still valid. The *MountainCar* domain, as described in class, has a continuous state space that will prevent us from using value or policy iteration to solve it directly. One of the solutions that we came up with for this was *nearest-neighbor interpolation*, where we discretize the actual state space S into finitely many states $\Xi = \xi_1, \xi_2, ..., \xi_n$, and act as if we are in the ξ nearest to our actual state s.

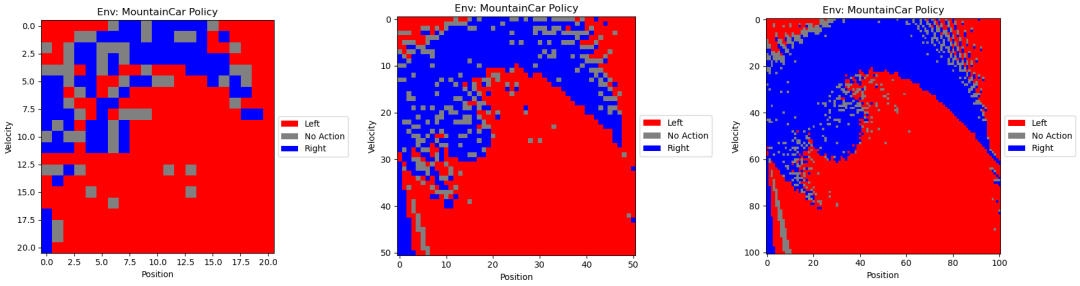
Implement Value Iteration with nearest-neighbor interpolation on the *MountainCar* domain. You will need to add code in *code/continuous_solution.py* below the lines *if self._mode == 'nn'*. Run the script and report the state value heatmap for `MountainCar`, discretizing each dimension of the state space (position and velocity) into 21, 51, and 101 bins.

Deterministic :



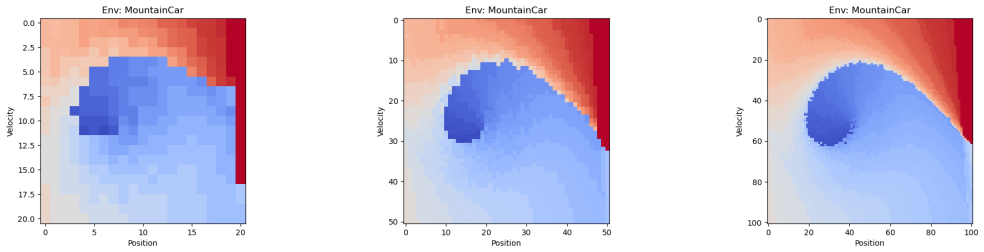
(a) 21 bins (b) 51 bins (c) 101 bins

Figure 5: Deterministic VI STATE : State value heatmaps for MountainCar with nearest-neighbor interpolation



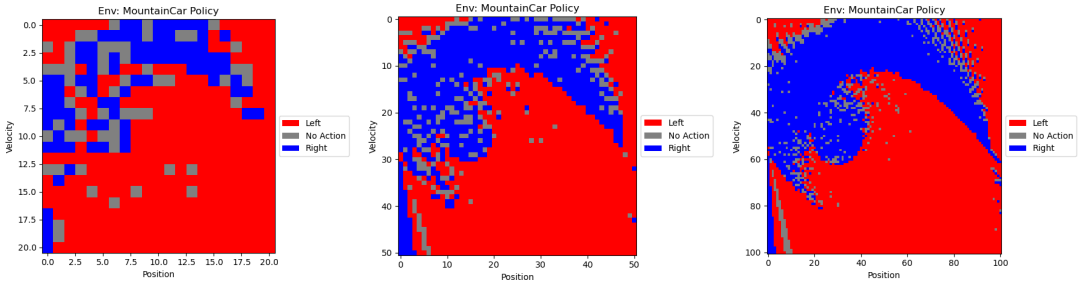
(a) 21 bins (b) 51 bins (c) 101 bins

Figure 6: Deterministic VI POLICY : State value heatmaps for MountainCar with nearest-neighbor interpolation



(a) 21 bins (b) 51 bins (c) 101 bins

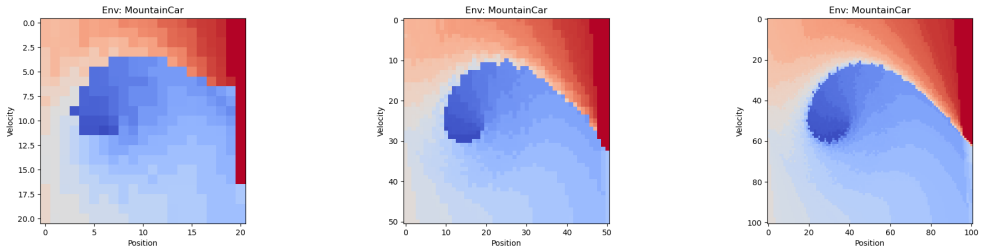
Figure 7: Deterministic PI STATE : State value heatmaps for MountainCar with nearest-neighbor interpolation



(a) 21 bins (b) 51 bins (c) 101 bins

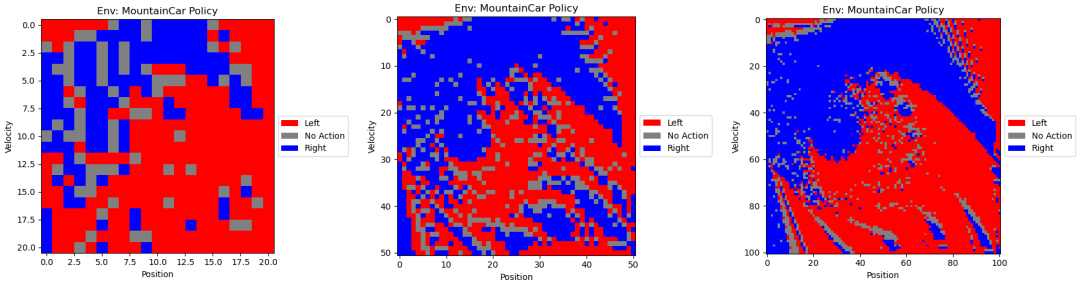
Figure 8: Deterministic PI POLICY : State value heatmaps for MountainCar with nearest-neighbor interpolation

Stochastic :



(a) 21 bins (b) 51 bins (c) 101 bins

Figure 9: Stochastic PI STATE : State value heatmaps for MountainCar with nearest-neighbor interpolation



(a) 21 bins (b) 51 bins (c) 101 bins

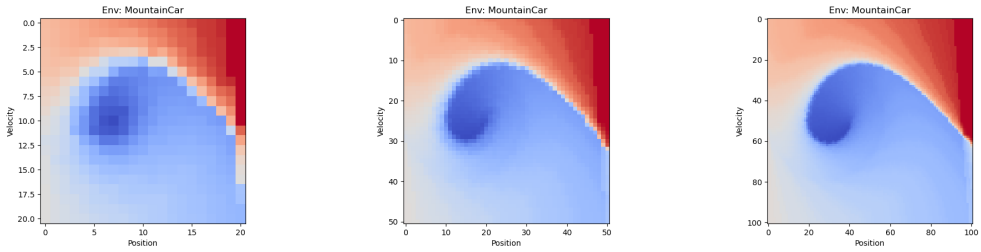
Figure 10: Stochastic PI POLICY : State value heatmaps for MountainCar with nearest-neighbor interpolation

2.2 Linear interpolation

Nearest-neighbor interpolation is able to approach the optimal solution if you use a fine-grained approximation, but doesn't scale well as the dimensionality of your problem increases. A more powerful discretization scheme that we discussed in class is *n-linear interpolation*, an *n*-dimensional analogue of linear interpolation. Add your code within `code/continuous_solution.py` below the line `if self._mode == 'linear'`. Just as before, report the state value heatmap for `MountainCar` with discretization resolutions of 21, 51, and 101 points per dimension.

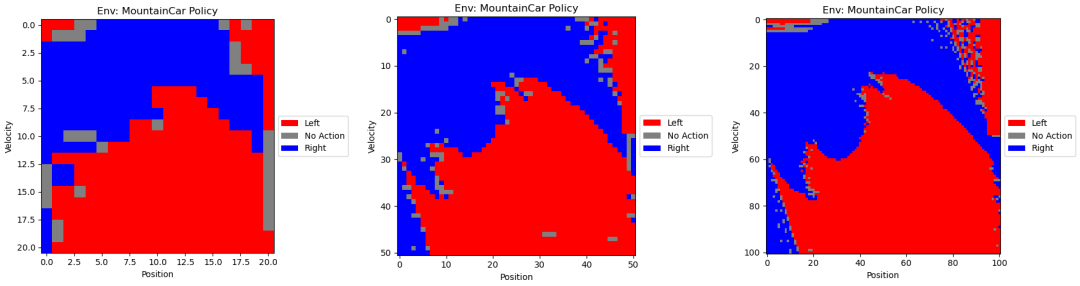
Deterministic :

Stochastic :



(a) 21 bins (b) 51 bins (c) 101 bins

Figure 11: Deterministic VI : State value heatmaps for MountainCar with nearest-neighbor interpolation



(a) 21 bins (b) 51 bins (c) 101 bins

Figure 12: Deterministic VI : State value heatmaps for MountainCar with nearest-neighbor interpolation

2.3 Stochastic Policy Iteration - Grad Only

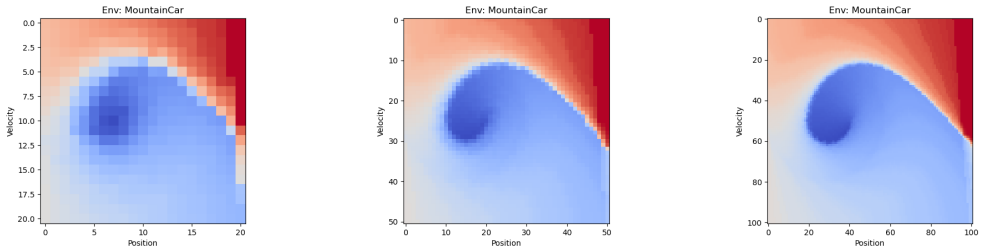
Grad students, implement stochastic policy iteration for the Mountain Car problem. This should be a small update to your existing policy iteration implementation.

a) You should not be considered if your stochastic policy iteration does not work as well as your deterministic implementation, this is expected. Why might this be? Explain your hypothesis.

Answer: Stochastic policy iteration might underperform compared to deterministic policy iteration because introducing randomness in action selection can lead to exploration of suboptimal paths, potentially slowing convergence to the optimal policy. In deterministic PI, actions are chosen greedily based on the current value function, ensuring consistent improvement, while stochastic policies might oscillate or get trapped in local optima due to probabilistic action selection.

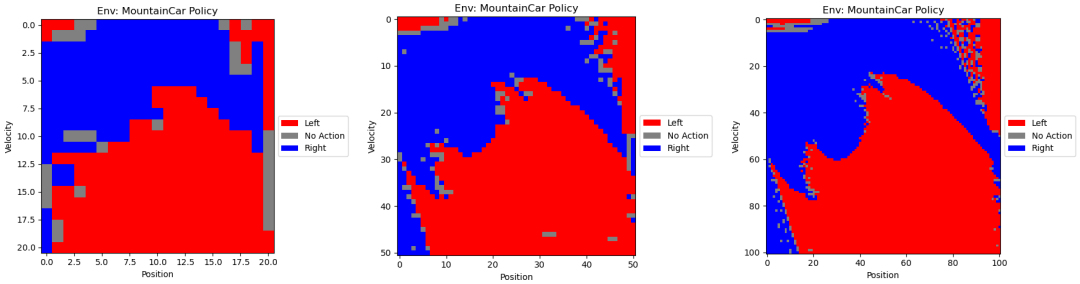
b) The idea of a stationary policy is more difficult to determine in the stochastic case. We have provided an implementation of KL-divergence for you in the codebase, you may use this to determine if your policy is approximately stationary. What is KL-divergence, and why might it be a good way of determining if your policy is stable? If you chose to use something other than KL-divergence, briefly explain.

Answer: KL-divergence (Kullback-Leibler divergence) measures the difference between two probability distributions. In this context, it can compare the policy distribution between successive



(a) 21 bins (b) 51 bins (c) 101 bins

Figure 13: Deterministic PI : State value heatmaps for MountainCar with nearest-neighbor interpolation

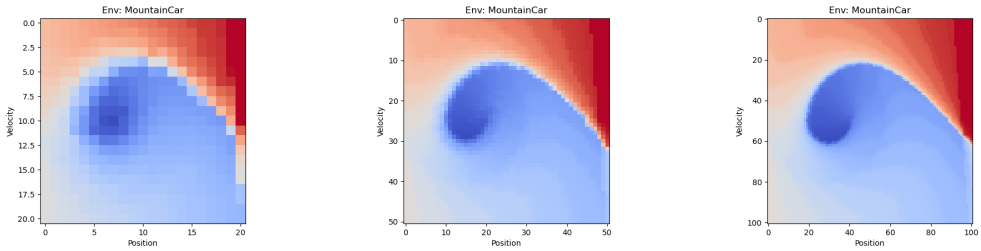


(a) 21 bins (b) 51 bins (c) 101 bins

Figure 14: Deterministic PI : State value heatmaps for MountainCar with nearest-neighbor interpolation

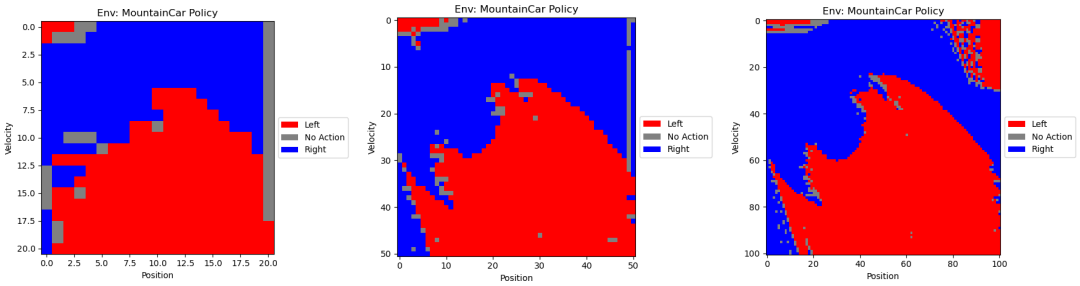
iterations. A low KL-divergence indicates that the policy is changing minimally, suggesting it's approximately stationary. It's effective because it quantifies distributional similarity, which is crucial for stochastic policies where actions are probabilistic rather than fixed. For Nearest Neighbors :

For Linear Interpolation :



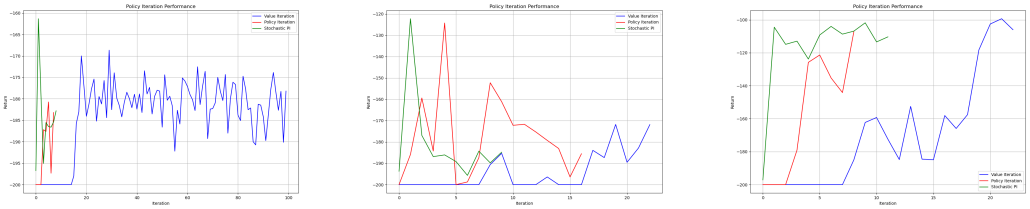
(a) 21 bins (b) 51 bins (c) 101 bins

Figure 15: Stochastic PI : State value heatmaps for MountainCar with nearest-neighbor interpolation



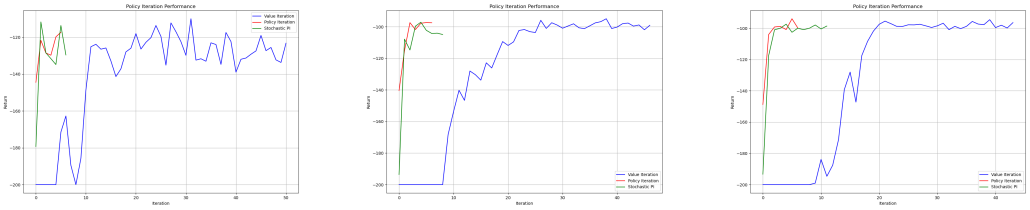
(a) 21 bins (b) 51 bins (c) 101 bins

Figure 16: Stochastic PI : State value heatmaps for MountainCar with nearest-neighbor interpolation



(a) 21 bins (b) 51 bins (c) 101 bins

Figure 17: Stochastic PI : State value heatmaps for MountainCar with nearest-neighbor interpolation



(a) 21 bins (b) 51 bins (c) 101 bins

Figure 18: Stochastic PI : State value heatmaps for MountainCar with nearest-neighbor interpolation