CS780: Deep Reinforcement Learning

Assignment #1

Name: Shreyansh Pachauri
Roll No.: 200954

Solution to Problem 1: Multi-armed Bandits

1. To test for the correct implementation of the we need to generate the test cases as follows:

   Test Cases and Expected Outcomes Test Case 1: $\alpha = 0.4$, $\beta = 0.6$ For this test case, set the action to be constant for a large number of simulations ($\sim 1000 - 5000$).

   Action 0: When only Action 0 is taken (left arm pulled every time), the average reward should tend to $\alpha = 0.4$ for $N \to \infty$.

   $$E[R_0] = \alpha \tag{1}$$

   Action 1: When only Action 1 is taken (right arm pulled every time), the average reward should tend to $\beta = 0.6$ for $N \to \infty$.

   $$E[R_1] = \beta \tag{2}$$

   Similarly when checked for different values of (α, β) such as $(1, 1)$, $(0, 0)$, $(1, 0)$, $(0, 1)$, $(0.5, 0.5)$ we see similar results. Also since there are only 2 actions, 2 Terminal States and only 2 possibilities when an action is taken the actions are expected to be distributed as a Bernoulli Distribution.

```
print("Average Reward over ", n," actions = ", sum/n)
```

```
Alpha = 0.4
Beta = 0.8
Constant Action 0 or 1 := 1
Number of Episodes to run = 1264
Average Reward over  1264  actions =  0.7776898734177216
```

Figure 1: Test Results on TwoArmedBernoulliBandit()

2. To Test the correct implementation of a TenArmedGassianBandit we took different values of sigma such as 1, 10, 100 etc and calculated the average reward

3. **To Test various agents we generate a TwoArmedBernoulli Bandit with $\alpha = 0.6$ and $\beta = 0.4$:**

```
environment = make('TwoArmedBernoulliBandit-v0', alpha=0.6, beta=0.4)
```

Figure 2: Test Environment

a. For the PureExploitation() agent we see that the agent only pulls the arm which gives it a good reward during the first episode and there is no scope for exploration hence it ignores the other arm. Hence the Q_est which is returned by the function contains only one non-zero value which corresponds to the maximum value of reward. See Figure 3:

```
print(PureExploitation(environment, 10000)[0])
```

```
[[0.         0.        ]
 [1.         0.        ]
 [0.5        0.        ]
 ...
 [0.59771954 0.        ]
 [0.59765977 0.        ]
 [0.5977     0.        ]]
```

Figure 3: Pure Exploitation Agent

b. For the PureExploitation agent we run the same agent and this time the final Q_est estimate gives values for all the actions. However even when $Q_estconvergestheagentdoesnotactgreedilyasinFigure4-$

```
print(PureExploration(environment, 1000)[0])
```

```
[[0.         0.        ]
 [0.         0.        ]
 [1.         0.        ]
 ...
 [0.57606491 0.39207921]
 [0.57489879 0.39207921]
 [0.57489879 0.39328063]]
```

Figure 4: Pure Exploration Agent

c. For the Epsilon Greedy Agent we check the implementation for 3-4 different values of ϵ and see that the Q_est converges to the final values as expected. We also check for extreme values of ϵ i.e. 0 and 1 and find that at $\epsilon = 0$, there is no exploration (purely exploitation) and at $\epsilon = 1$ there is no exploitation (pure exploratory). See Figure 5

```
print(EpsilonGreedy(environment, 100000, epsilon=0.1)[0])
```

```
[[0.         0.        ]
 [0.         0.        ]
 [0.5        0.        ]
 ...
 [0.5985848  0.37621794]
 [0.59858903 0.37621794]
 [0.59859325 0.37621794]]
```

Figure 5: Epsilon Greedy Agent

d. For the decaying-ϵ-Greedy policy look at Figure 6:

e. For the Softmax Function the result is as follows in Figure 7:

f. For the UCB Agent the result is in Figure 8:

```
print(EpsilonGreedy(environment, 100000, epsilon=0.1, decayType = "linear")[0])

[[0.         0.        ]
 [0.         0.        ]
 [1.         0.        ]
 ...
 [0.60171306 0.40167015]
 [0.6017069  0.40167015]
 [0.60171098 0.40167015]]
```

Figure 6: Decaying Epsilon Greedy Agent

```
print(Softmax(environment, 50000, 100)[0])

[[0.         0.        ]
 [0.         0.        ]
 [0.         0.        ]
 ...
 [0.60165608 0.39988057]
 [0.60165608 0.39986466]
 [0.60167209 0.39986466]]
```

Figure 7: Softmax Agent

```
print(UCB(environment, maxEpisodes=100000, c=0.6)[0])

[[0.         0.        ]
 [0.         0.        ]
 [0.         0.        ]
 ...
 [0.59784686 0.31372549]
 [0.59785088 0.31372549]
 [0.5978449  0.31372549]]
```

Figure 8: UCB Agent

4. Plots of Rewards vs Episodes for 50 different 2-armed Bernoulli Bandit Environments in Figure 9 we observe that the rewards in general are in the order UCB $\geq$ Softmax$\sim$Decaying Epsilon Greedy$\sim$Epsilon Greedy $\geq$ Pure Exploitation $\geq$ Pure Exploration

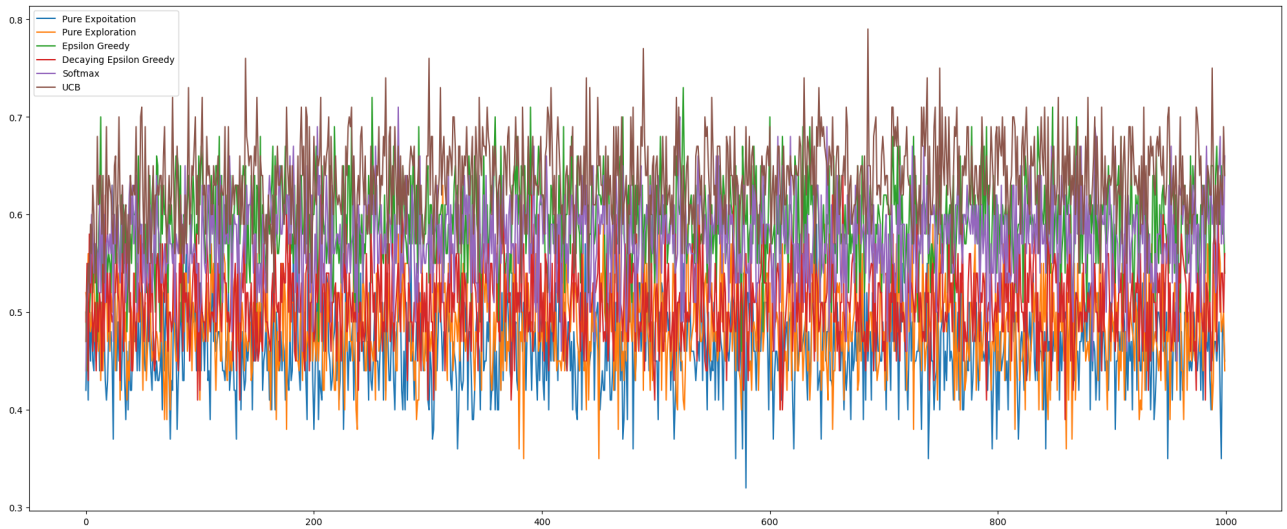

Figure 9: Reward vs Episodes for different agents in a Two Armed Bernoulli Bandit Environment

5. Plots of Rewards vs Episodes for 50 different 10-armed Gaussian Bandit Environments in Figure 10. The

general observations are that the instantaneous rewards are finally in the order UCB > Epsilon Greedy > Decaying Epsilon Greedy > Softmax > Pure Exploitation > Pure Exploration .

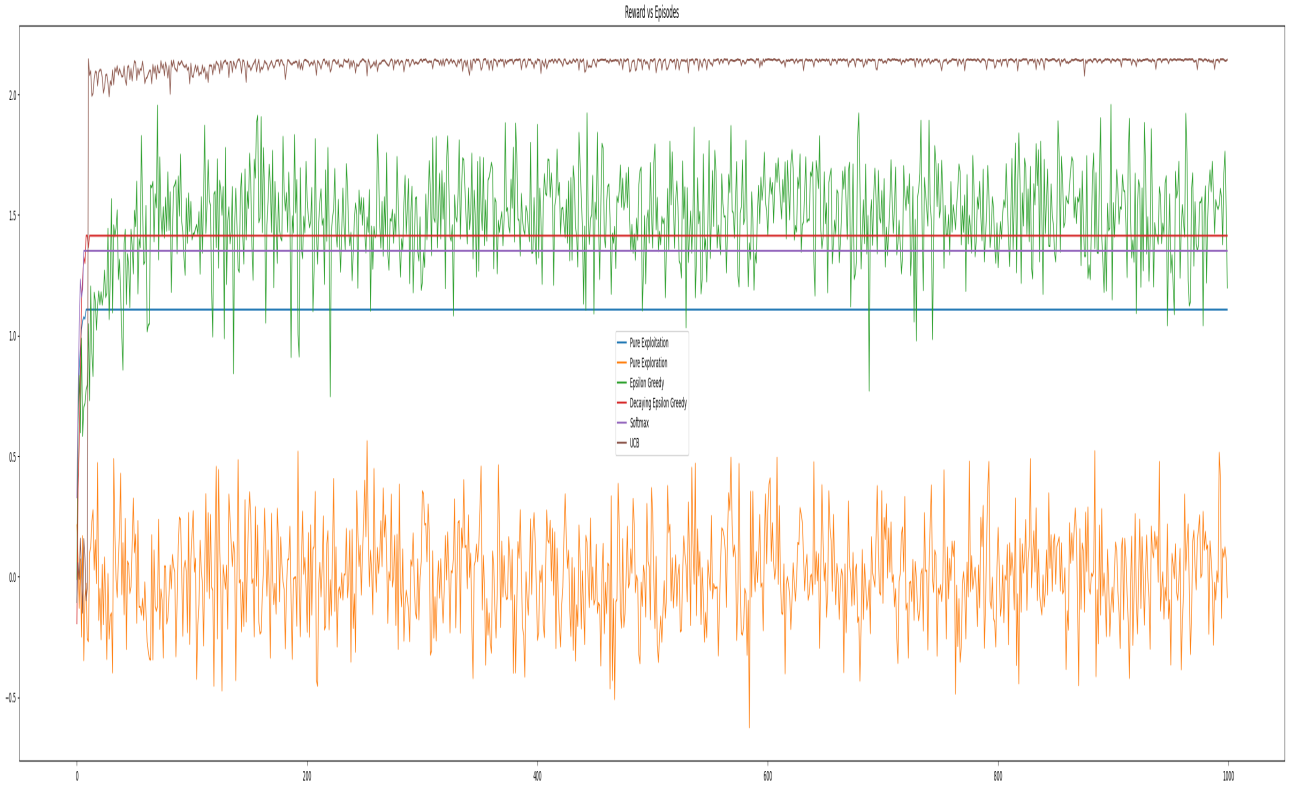

Figure 10: Rewards vs Episodes in Ten-Armed Gaussian Bandit

6. Plot of Regret at a particular episode vs the Number of Episodes (e) in a Two Armed Bernoulli Bandit is given in Figure 11. The Regret for Pure Exploration > Epsilon Greedy > UCB ∼ Softmax ∼ UCB ∼ Decaying Epsilon Greedy ∼0

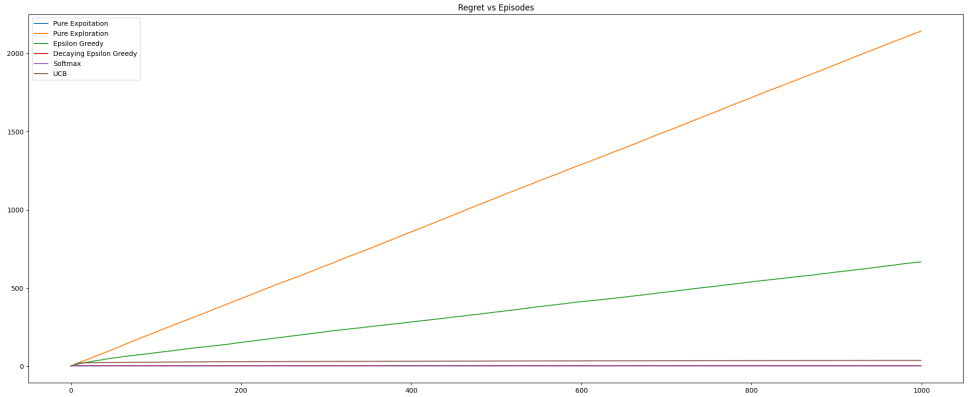

Figure 11: Regret vs Episodes for TwoArmedBernoulli Bandit

7. Plot of Regret vs Episodes for a Ten Armed Gaussian Bandit is given in Figure 12. The observations are similar as in 1.6. .

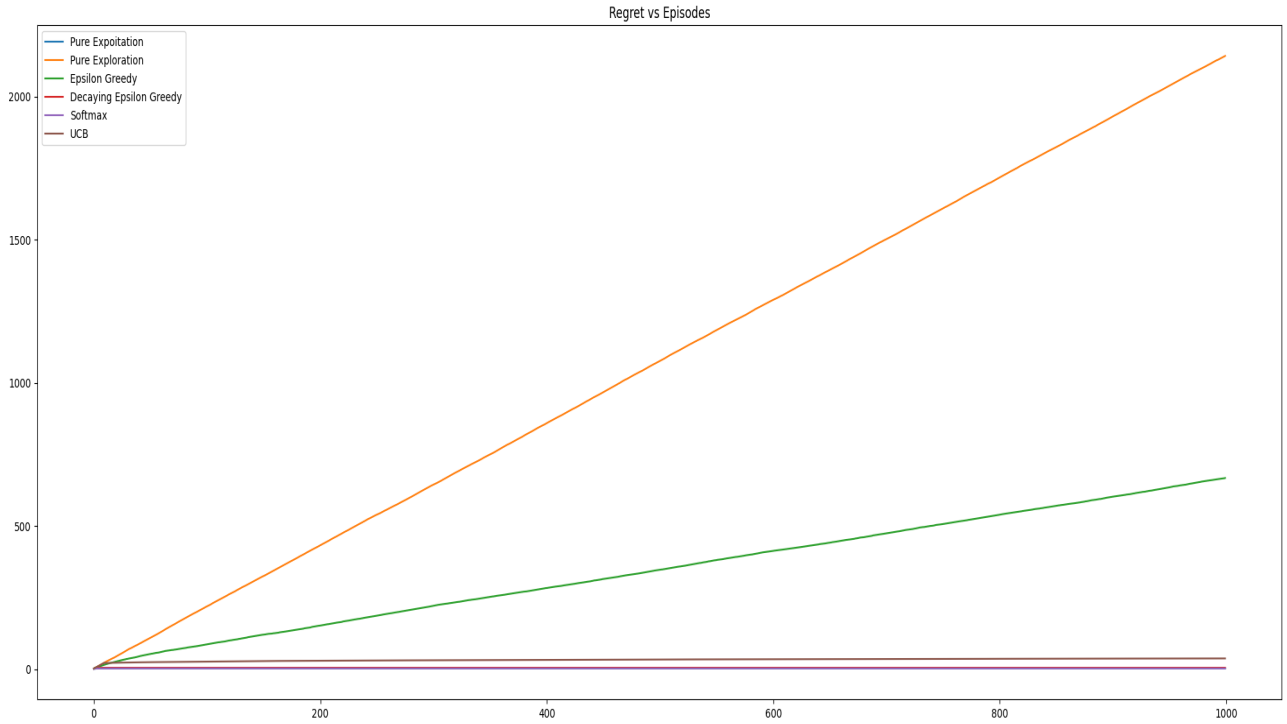

Figure 12: Regret vs Episodes for Ten Armed Gaussian Bandit

8. Plot of Average Reward vs Episodes for a Two Armed Bernoulli Bandit is given in Figure 13. We see that the Final Average Reward for UCB > Epsilon Greedy > Decaying Epsilon Greedy > Softmax > Pure Exploitation >> Pure Exploration

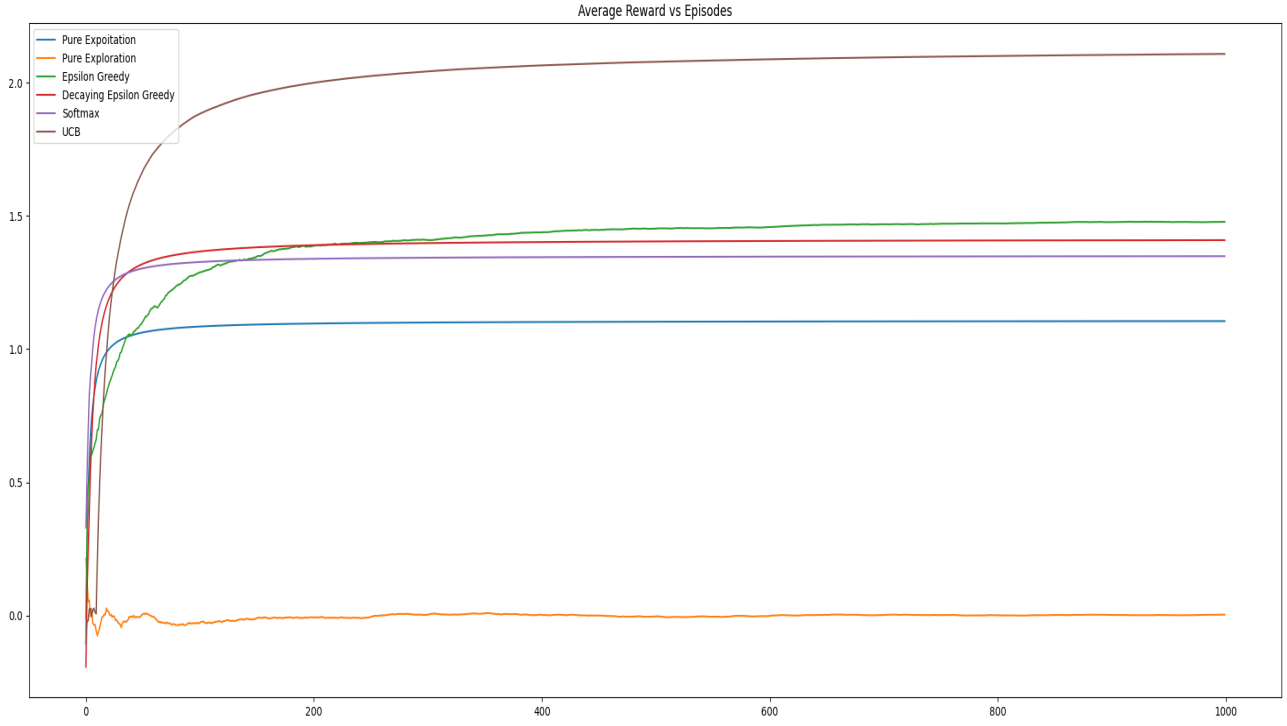

Figure 13: Average Reward vs Episodes for a 2 armed Bernoulli Bandit

9. Solution of 1.10: Plot of Average Reward vs Episodes for a Ten Armed Gaussian Bandit is given in Figure 15. We see that the Final Average Reward for UCB > Epsilon Greedy > Decaying Epsilon Greedy > Softmax > Pure Exploitation >> Pure Exploration .

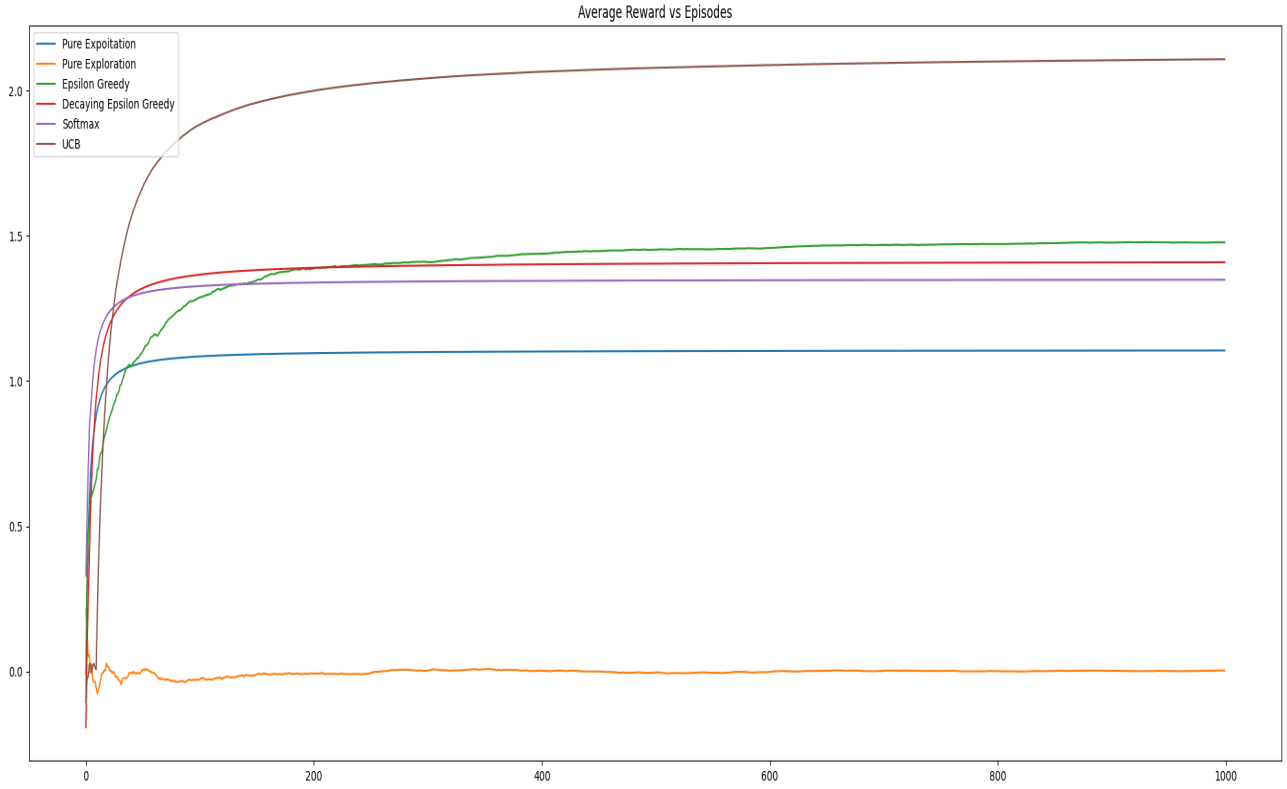

Figure 14: Average Reward vs Episodes for a 10 armed Gaussian Bandit

Solution to Problem 2: MC Estimates and TD Learning

1. The testing for generateTrajectory() Function with the policy that goes only right

Testing the generateTrajectory() Function

```
environment = make('RWE-v0')
policy = {
    0: random.choice([1]),
    1: random.choice([1]),
    2: random.choice([1]),
    3: random.choice([1]),
    4: random.choice([1]),
    5: random.choice([1]),
    6: random.choice([1])}

trajectory = generateTrajectory(environment, policy, 10)
print(trajectory)
```

```
[(1, (2, 1, 0.0, 1)), (2, (1, 1, 0.0, 2)), (3, (2, 1, 0.0, 1)), (4, (1, 1, 0.0, 2)), (5, (2, 1, 0.0, 1)), (6, (1, 1, 0.0, 0))]
```

Figure 15: generateTrajectory() testing

2. The plots for decayingAlpha Functions are as follows for 1000 steps. Initial value of alpha is 1

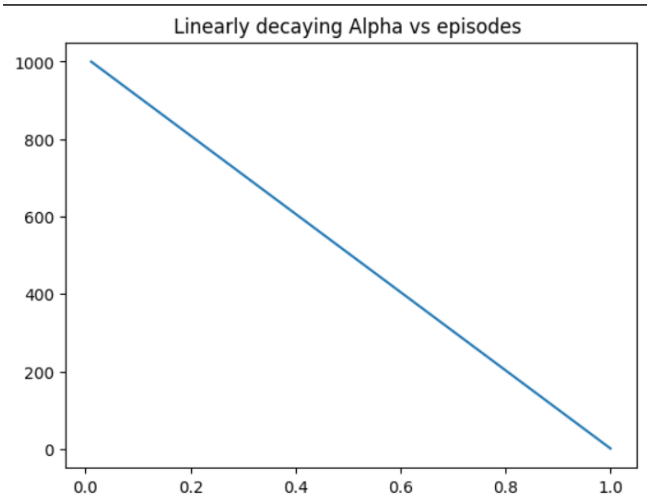

Figure 16: Linear Decay

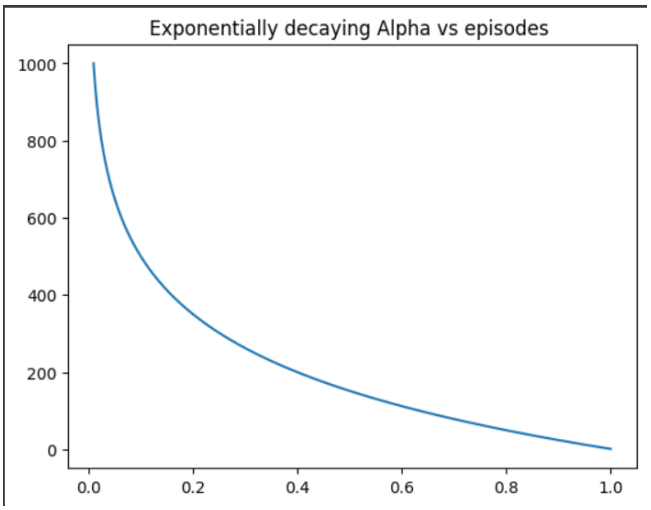

Figure 17: Exponential Decay

3. Solution for 2.3

```
n=50000
policyL = {0:0, 1:0, 2:0, 3:0, 4:0, 5:0, 6:0}
v, v_r = MonteCarloPrediction(environment, policy = policyL, gamma=0.99, alpha = 0.5, maxSteps = 1000, noEpisodes = n, firstVisit = True)
```

Figure 18: Parameters for Monte Carlo Estimate Testing

```
[[0.         0.45219104 0.44766913 ... 0.46603267 0.47074007 0.        ]
 [0.         0.22698181 0.224712   ... 0.23392976 0.47074007 0.        ]
 [0.         0.11438068 0.224712   ... 0.23392976 0.47074007 0.        ]
 ...
 [0.         0.14204003 0.32594099 ... 0.63148171 0.81545078 0.        ]
 [0.         0.14061963 0.32268158 ... 0.63148171 0.81545078 0.        ]
 [0.         0.14061963 0.32268158 ... 0.63148171 0.81729627 0.        ]]
```

Figure 19: Estimated State values for different episodes

4. Solution for 2.4

```
n=50000
v, v_r = TemporalDifferencePrediction(environment, policy=policyL, gamma=0.99, alpha = 0.5, noEpisodes = n)
```

Figure 20: Parameters for TD-Estimates Testing

```
[[0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 ...
 [0.         0.15664246 0.33905904 ... 0.65312628 0.80930093 0.        ]
 [0.         0.15664246 0.33905904 ... 0.65312628 0.81120792 0.        ]
 [0.         0.15664246 0.33905904 ... 0.65462598 0.81309584 0.        ]]
```

Figure 21: TD-Estimate for different state values

5. Solution for 2.5:

   Plot for the MC-FVMC estimate of each non-terminal state of RWE as it progress through different episodes. along with the True Estimate. The plot is for 500 episodes.The step size parameter () starts from 0.5 and decreases exponentially to 0.01 till 500 episodes. The policy being tested is "Always Right" in Figure 22 while "Always Left" in Figure 23

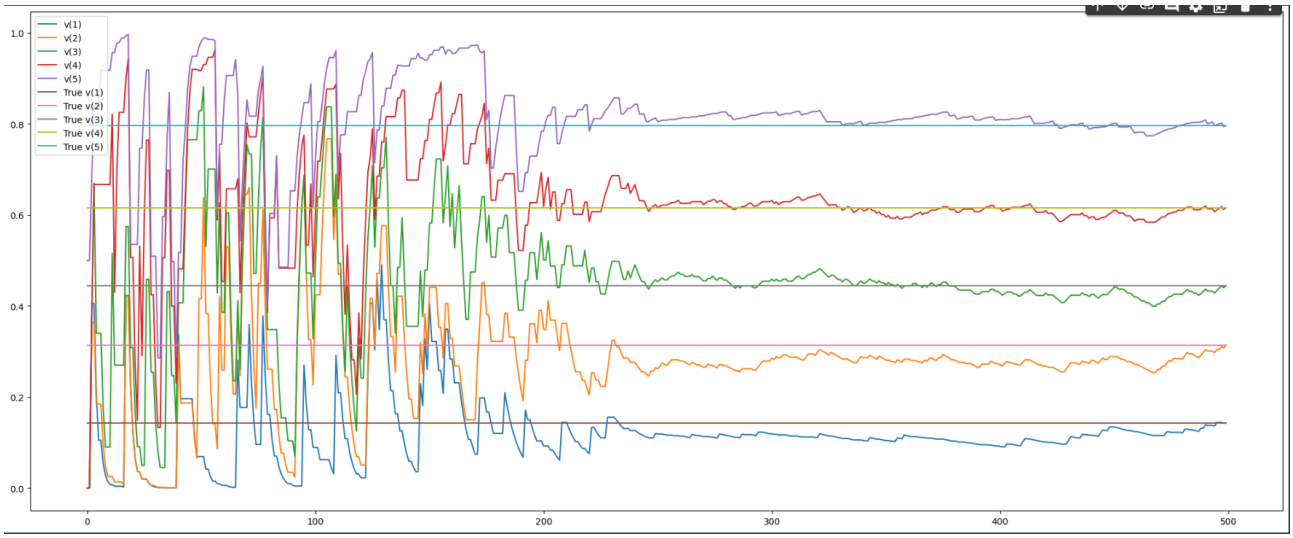

Figure 22: MC-FVMC Estimate for Always Right policy

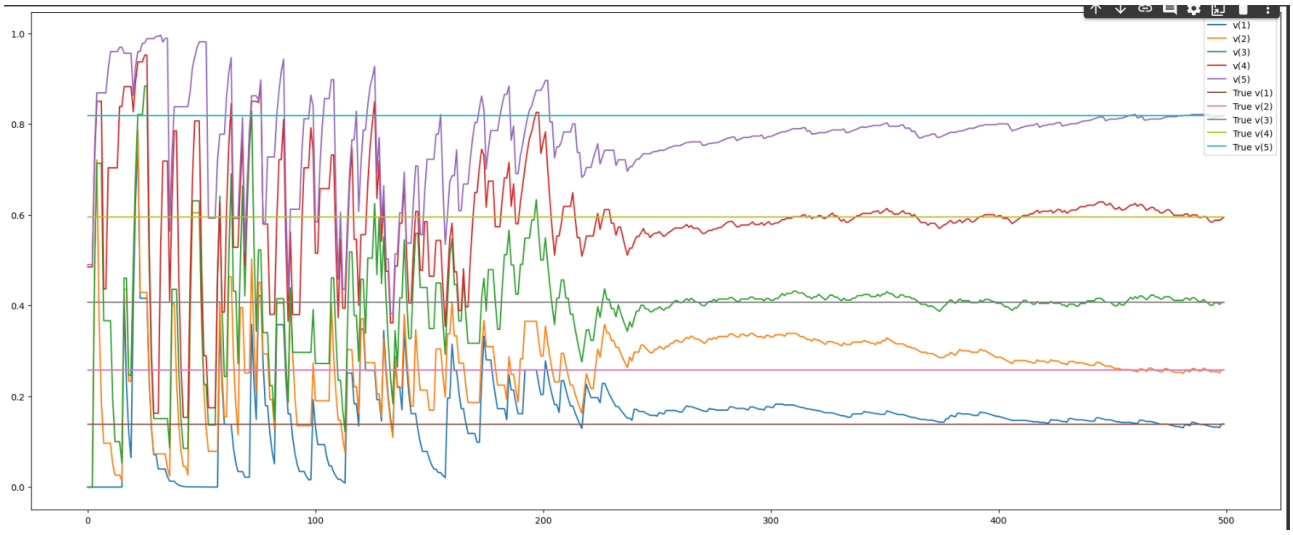

Figure 23: MC-FVMC Estimate for Always Left policy

6. Solution for 2.6:

   Plot for the MC-EVMC estimate of each non-terminal state of RWE as it progress through different episodes. along with the True Estimate. The plot is for 500 episodes.The step size parameter () starts from 0.5 and decreases exponentially to 0.01 till 500 episodes. The policy being tested is "Always Right" in Figure 24 while "Always Left" in Figure 25

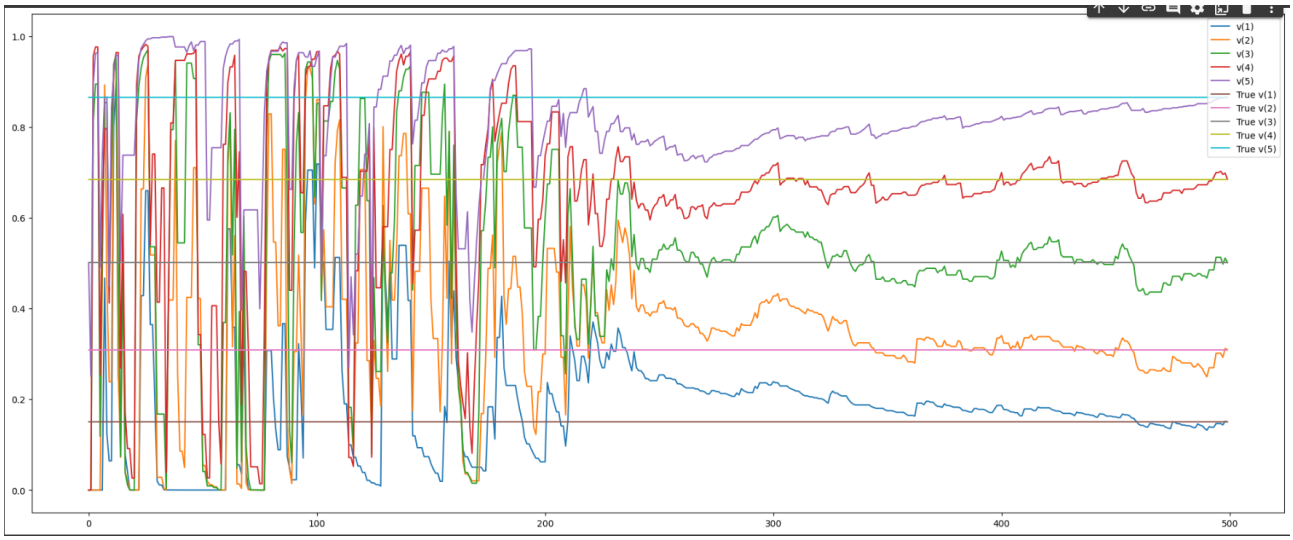

Figure 24: MC-EVMC For Always Right policy

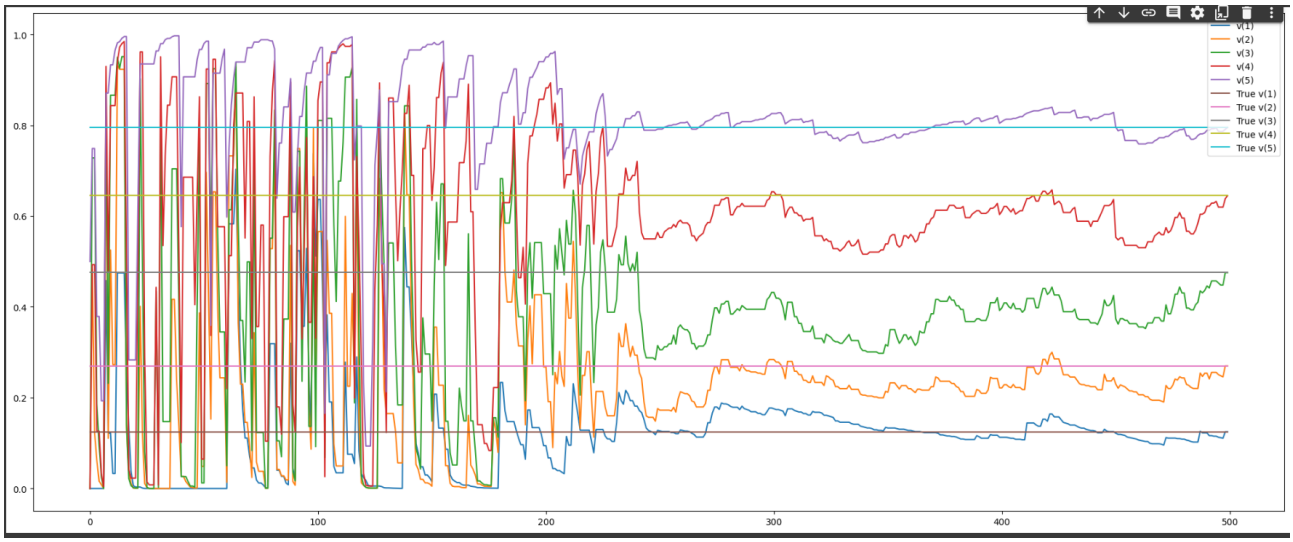

Figure 25: MC-EVMC for Always Left Policy

7. Solution 2.7

   Plot for the TD-estimate of each non-terminal state of RWE as it progress through different episodes. along with the True Estimate. The plot is for 500 episodes. The step size parameter () starts from 0.5 and decreases exponentially to 0.01 till 500 episodes. The policy being tested is "Always Right" in Figure 26 while "Always Left" in Figure 27

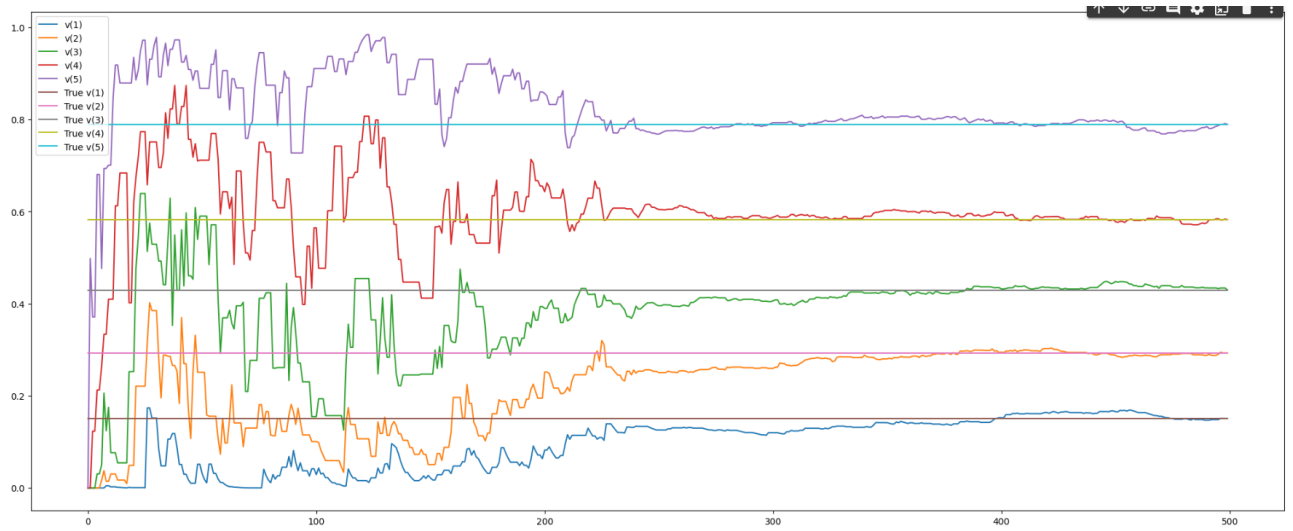

Figure 26: TD-Estimate for always Right policy

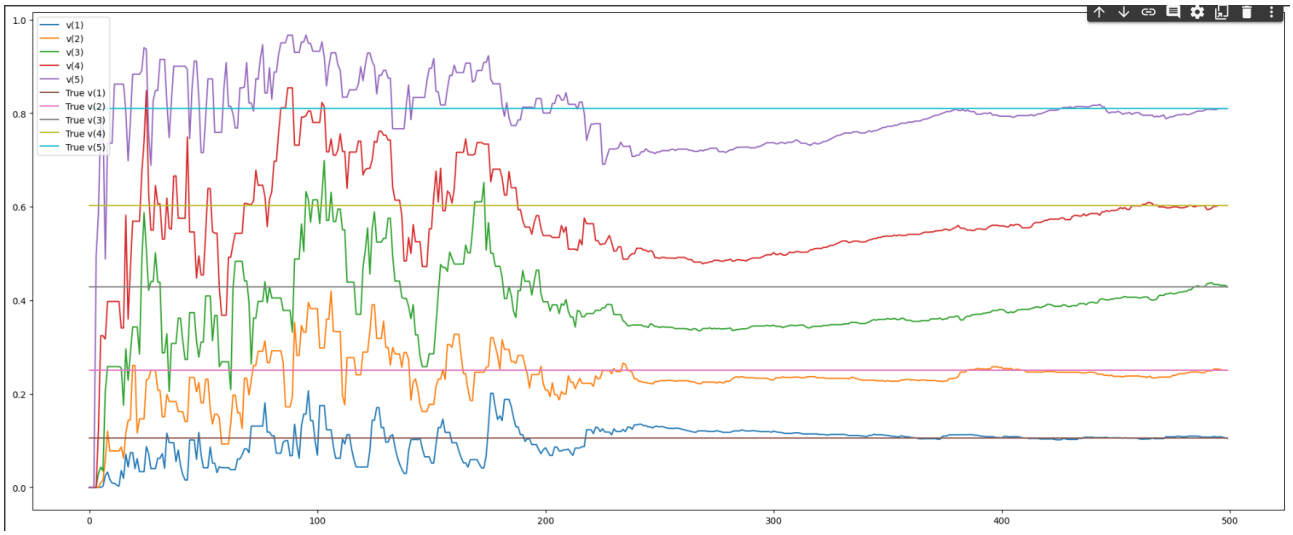

Figure 27: TD-Estimate for always Left policy

8. Solution for 2.8: Figure 28, 29, 30

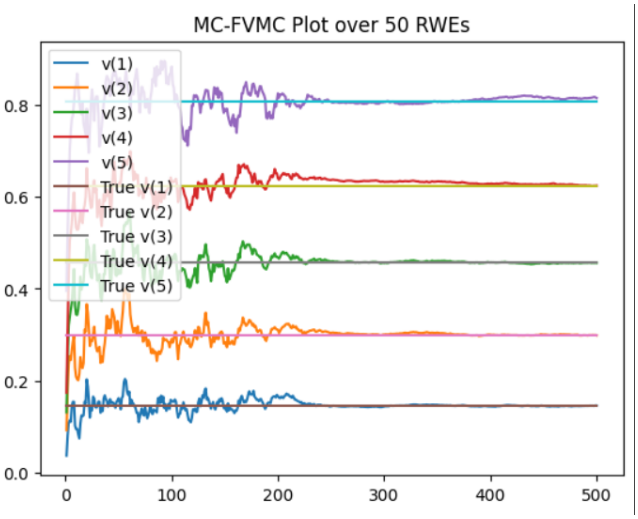

Figure 28: MC-FVMC

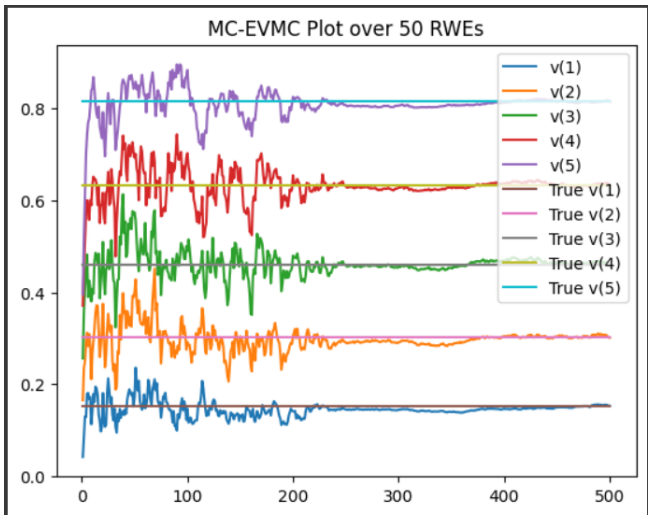

Figure 29: MC-EVMC

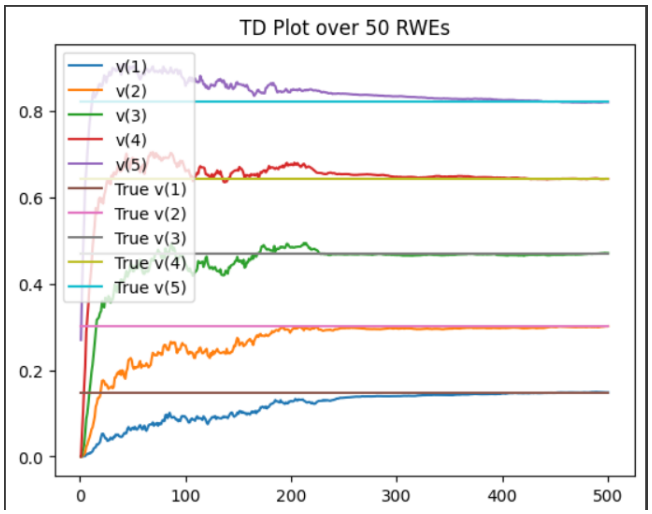

Figure 30: TD

9. Solution for 2.9: Figure 31, Figure 32

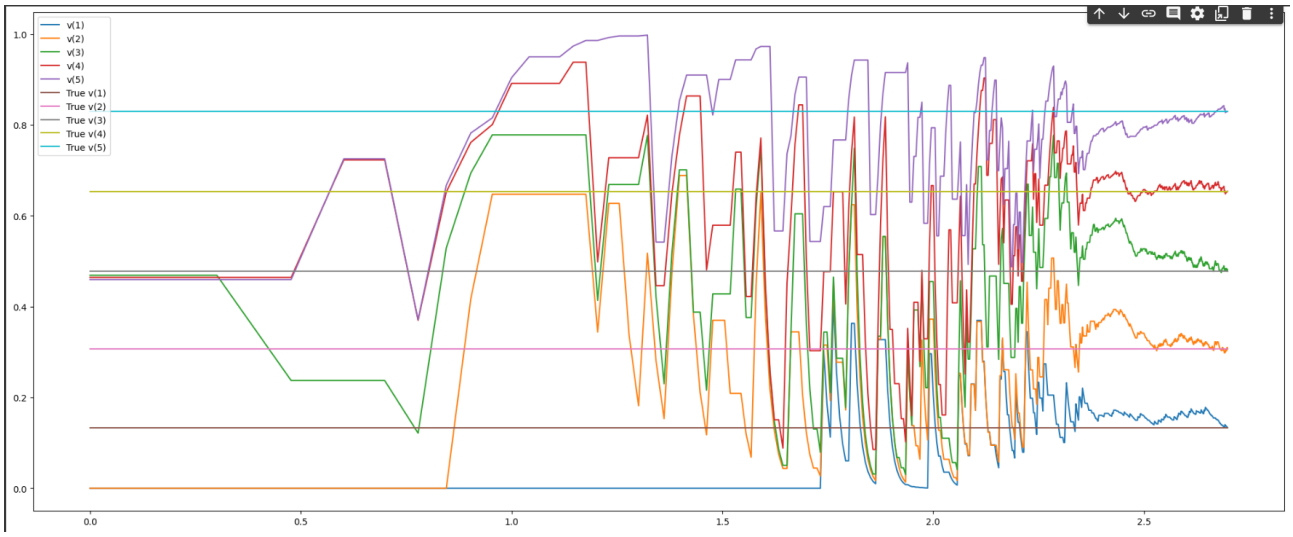

Figure 31: MC-FVMC Log Plot for "Always Right" policy

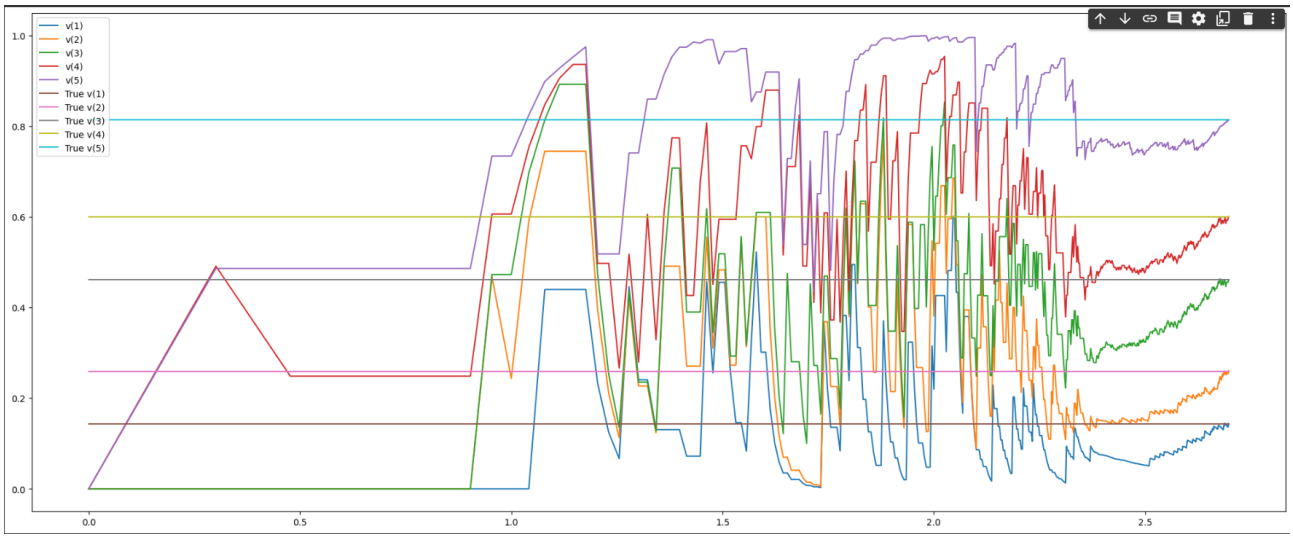

Figure 32: MC-FVMC Log Plot for "Always Left" policy

10. Solution for 2.10: Figure 30, Figure 31

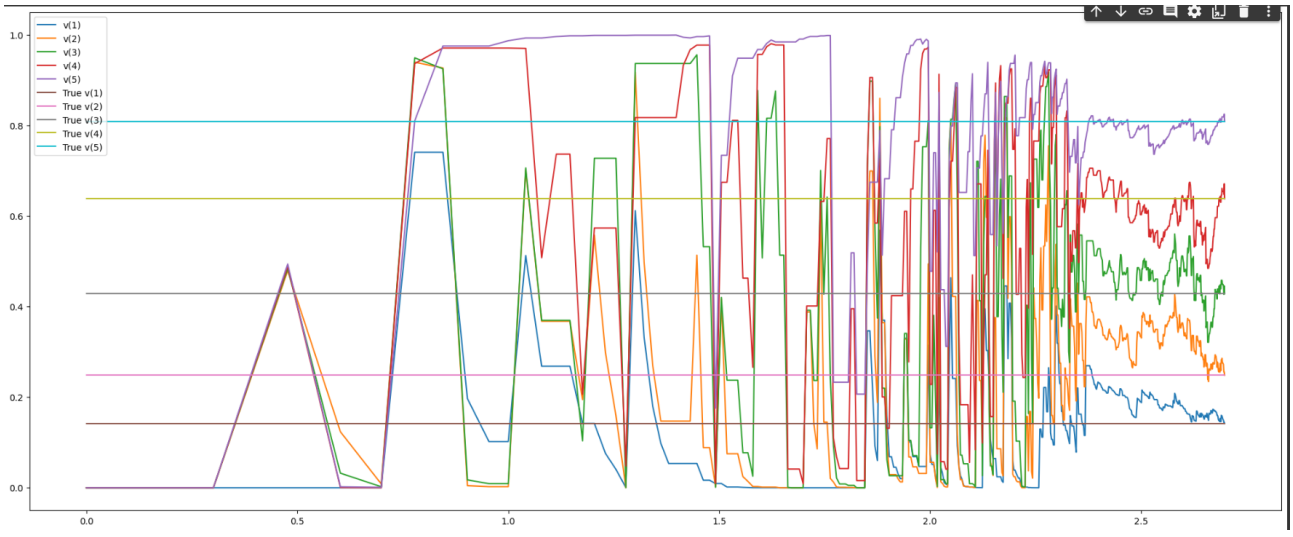

Figure 33: MC-EVMC Log Plot for "Always Right" Policy

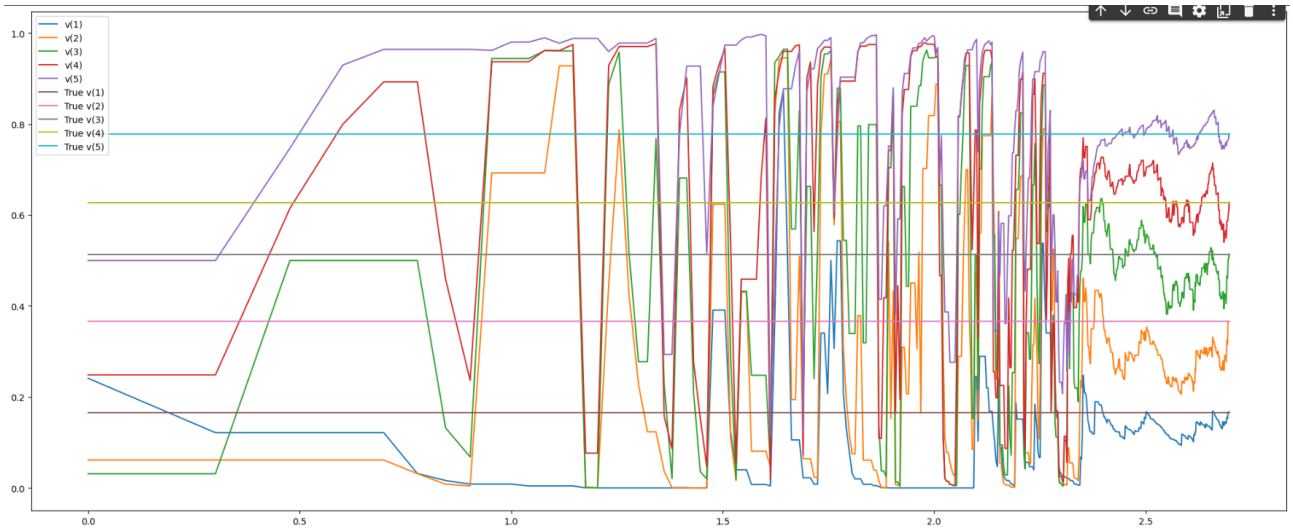

Figure 34: MC-EVMC Log Plot for "Always Left" Policy

11. Solution for 2.11: Figure 32, Figure 33

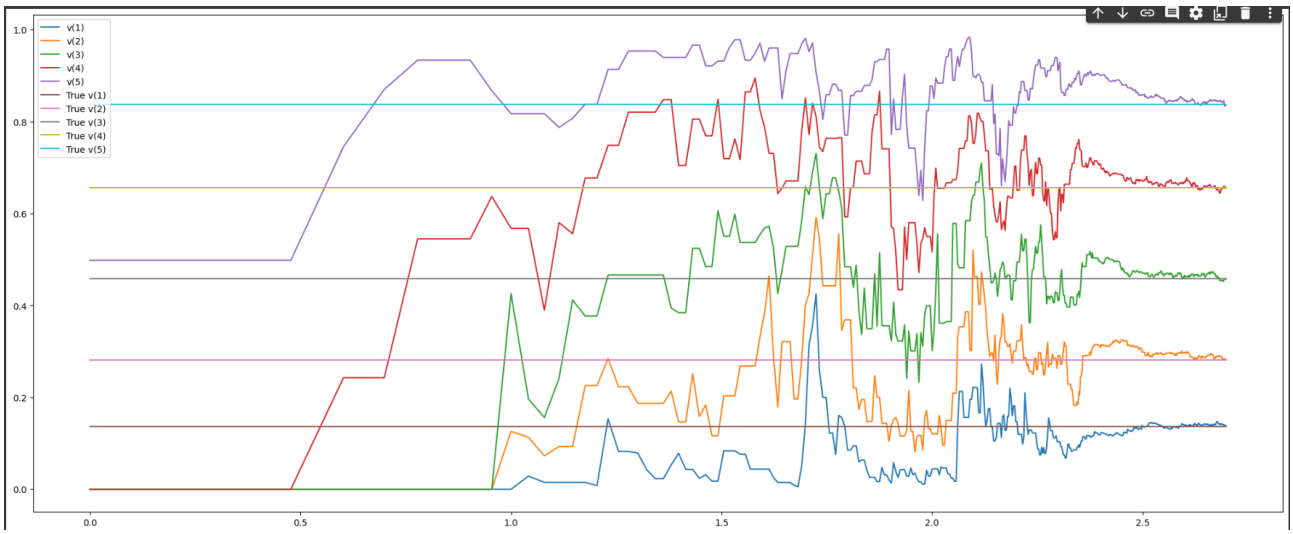

Figure 35: TD-Estimate Log Plot for "Always Right" Policy

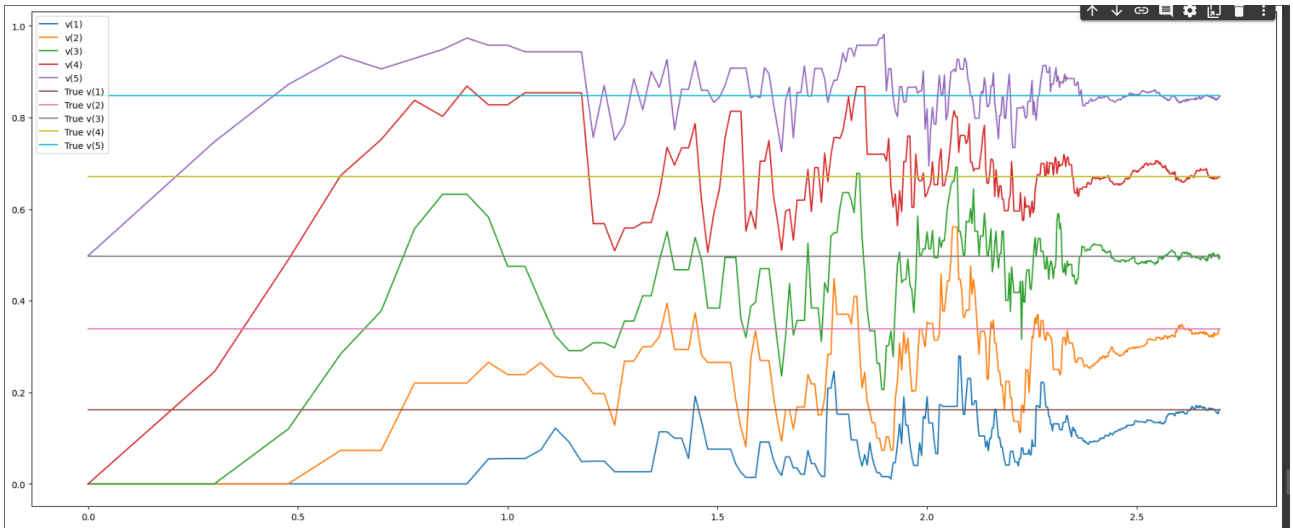

Figure 36: TD-Estimate Log Plot for "Always Left" Policy

12. Solution for 2.12 Analysing the Plots

    We observe that in both the policies 0:"Always Left" and 1:"Always Right" the final value of V tends in such a way that $v(1) < v(2) < v(3) < v(4) < v(5)$ which is expected since there is only a unique Optimal Value Function. Also we observe that the variance in V values for

    MC-EVMC >> MC-FVMC > TD-estimate (For $\sim$ 50000 iterations)

    As far as the Computational Efficiency is concerned we see that TD-estimate takes lesser time to run while the MC estimates take longer times because TD-Estimate updates after every step