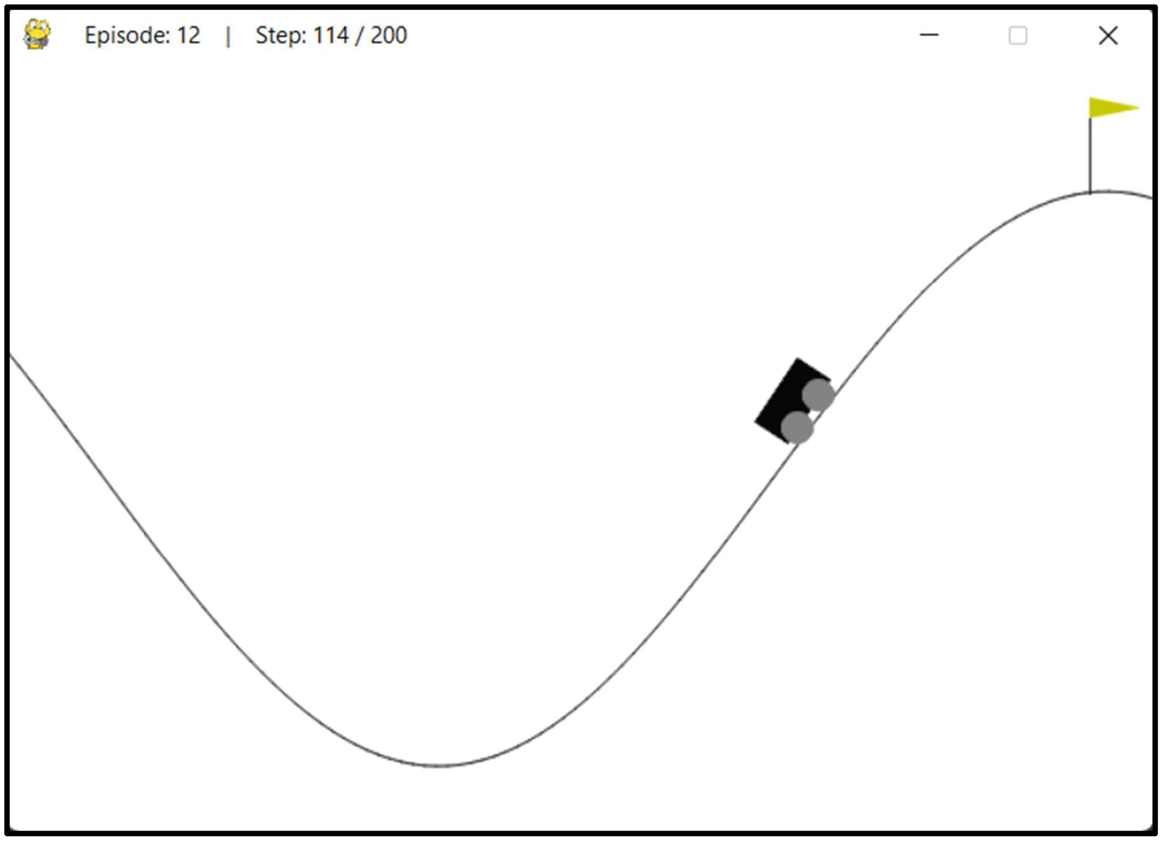
AI Report

CENG 449 – Advanced Topics Artificial Intelligence

SARSA(λ) - Mountain Car

Written by: Noah Zikmund

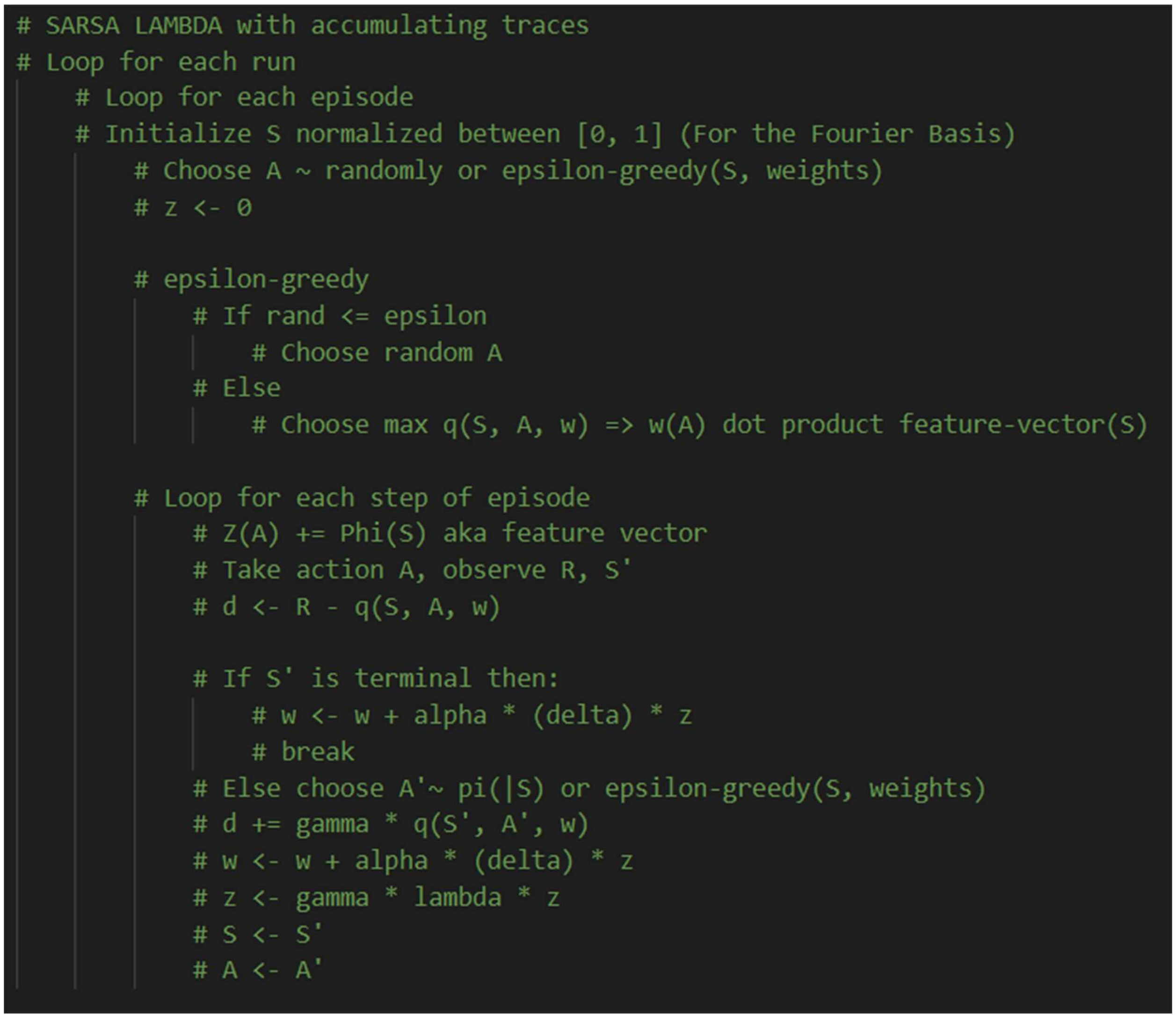
Date: 12/2/2022



SOUTH DAKOTA
SCHOOL OF MINES
& TECHNOLOGY

Overview

The task for this program was to implement Sarsa(λ) for the Mountain Car problem as described in Sutton and Barto. To accomplish this, OpenAI gym was used to aid in establishing the environment for the mountain car. Then, instead of discretizing the state space, Fourier basis functions were used for linear value function approximation. The exact Sarsa(λ) algorithm that was used was Sarsa(λ) with accumulating trace, and it can be found below in Figure 1.



```
# SARSA LAMBDA with accumulating traces
# Loop for each run
    # Loop for each episode
    # Initialize S normalized between [0, 1] (For the Fourier Basis)
        # Choose A ~ randomly or epsilon-greedy(S, weights)
        # z <- 0

        # epsilon-greedy
            # If rand <= epsilon
                # Choose random A
            # Else
                # Choose max q(S, A, w) => w(A) dot product feature-vector(S)

        # Loop for each step of episode
            # Z(A) += Phi(S) aka feature vector
            # Take action A, observe R, S'
            # d <- R - q(S, A, w)

            # If S' is terminal then:
                # w <- w + alpha * (delta) * z
                # break
            # Else choose A'~ pi(|S) or epsilon-greedy(S, weights)
            # d += gamma * q(S', A', w)
            # w <- w + alpha * (delta) * z
            # z <- gamma * lambda * z
            # S <- S'
            # A <- A'
```

Figure 1: Sarsa(λ) with Accumulating Trace

Program details

The mountain car solution outlined in this report consists of two code files: MountainCar.py and agent.py. MountainCar.py contains a mountain car class that inherits from OpenAI gym and makes it easier to access different elements of the state space. Please note that I am not the author of MountainCar.py and that it was written by Ryan Finn. Agent.py contains all the work for the program, Sarsa(λ) with Fourier value function approximation, and is entirely my work. In order to compile, please note that the OpenAI gym, numpy, and matplotlib dependencies will need to be installed. To compile the program after downloading both files and installing dependencies, follow this usage:

```
$ python agent.py
```

Along with the compile-time requirements, there are settings within the code itself that will alter how the program works. At the top of agent.py, there are 9 global constants that can affect the code:

• ALPHA	-	Learning rate
• GAMMA	-	Discount factor
• LAMBDA	-	Trace parameter
• EPSILON	-	Totally greedy
• MAX_STEPS	-	Maximum episode length
• EPISODES	-	Number of episodes in each run
• ANIMATE	-	Boolean value that determines if animations are on or off
• RUNS	-	Number of runs
• ORDER	-	Order or degree at which the program is run (3, 5, 7)

Part 1 – Show learning curves for order 3, 5, and 7 using Fourier basis functions

The following figures show the learning curves for orders 3, 5, and 7 for Fourier basis function approximation with fixed settings of $\alpha = 0.001$, $\gamma = 1$, and $\lambda = 0.9$.

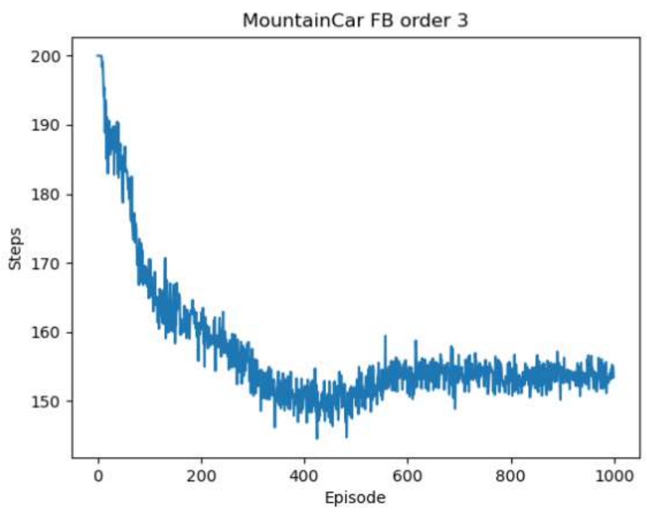


Figure 2: Learning Curve (Order 3)

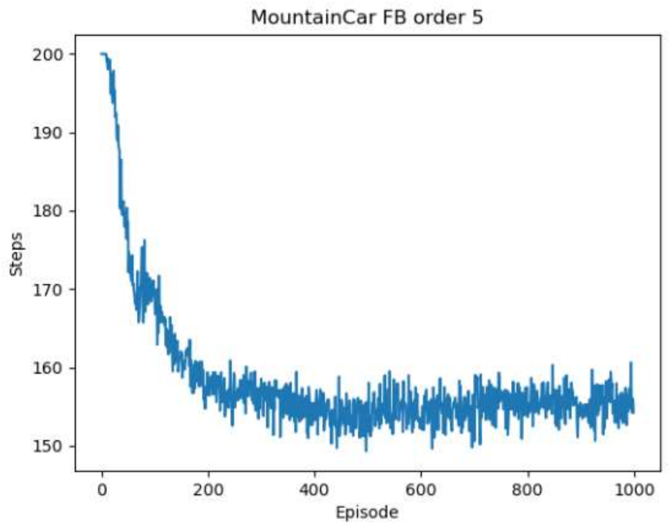


Figure 3: Learning Curve (Order 5)

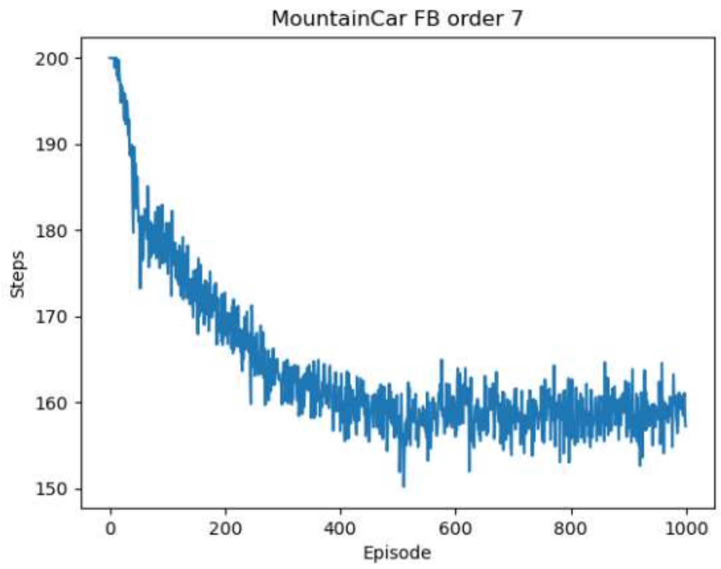


Figure 4: Learning Curve (Order 7)

Figures 2-4 show the learning curves summed and averaged over 100 runs, and as it is apparent, the number of episodes increase as the number of steps to reach the terminal state decrease. For these series of runs, a maximum step size of 200 was set to increase the speed of the program without sacrificing accurate results. Because Sutton and Barto's examples converge at 150 steps per episode, it was assumed that the difference between 200 and 150 would be great enough to yield accurate, graphical results.

As mentioned above, a Fourier basis was used for linear value function approximation. For the purpose of this program, a simplified version of multivariate Fourier series was used:

$$\phi_i(\mathbf{x}) = \cos\left(\pi \mathbf{c}^i \cdot \mathbf{x}\right)$$

Figure 5: Fourier Equation

Phi is the result (basis function), C is the coefficients vector, and X is the state space vector. To define Phi, C, and X even further, Phi is a basis function, but when multiple basis functions are calculated, the result is a feature vector. C is a 2D vector, that has a size dependent upon the order at which the program is ran. This size is the same as the size of the feature vector and is determined by the following:

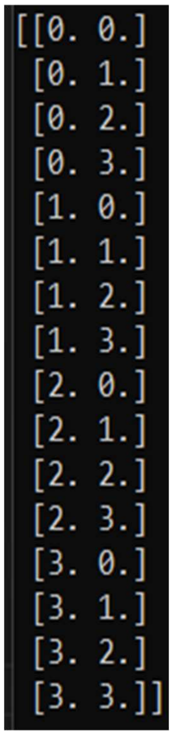
Vector Size = (Order + 1) ^ Dimensions

For the mountain car problem, there are two dimensions: position and velocity (aka state space vector). Therefore, order 3 has 16 entries, order 5 has 36 entries, and order 7 has 64 entries (See Figure 6 & 7 for examples of C at orders 3 & 5).

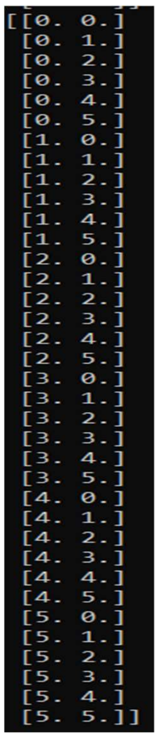
The mountain car's position is a value between -1.2 and 0.6 and the velocity is a value between -0.07 and 0.07 (Note that these values are defined by OpenAI gym and are the same as the values described by Sutton and Barto). It is also important to note that before a state is passed into the Fourier basis function, the state should be normalized between a value of 0 and 1 to accommodate the cosine function. If the state is not normalized first, the solution may never converge. If the solution does converge, the value function will not resemble the expected value function given by Sutton and Barto.

Part 2 - Surface plot of the value function

The surface plots in the following figures are the negative of the value function learned after 1,000 episodes. The expected results are shown in Figure 8 (from Sutton and Barto). As it is apparent, the surface plot is a 3D plot with x, y, and z being position, velocity, and Q-value accordingly. The Q-value in this case is the value of the value function at that state in the state space.



```
[[0. 0.]
 [0. 1.]
 [0. 2.]
 [0. 3.]
 [1. 0.]
 [1. 1.]
 [1. 2.]
 [1. 3.]
 [2. 0.]
 [2. 1.]
 [2. 2.]
 [2. 3.]
 [3. 0.]
 [3. 1.]
 [3. 2.]
 [3. 3.]]
```



```
[[0. 0.]
 [0. 1.]
 [0. 2.]
 [0. 3.]
 [0. 4.]
 [0. 5.]
 [1. 0.]
 [1. 1.]
 [1. 2.]
 [1. 3.]
 [1. 4.]
 [1. 5.]
 [2. 0.]
 [2. 1.]
 [2. 2.]
 [2. 3.]
 [2. 4.]
 [2. 5.]
 [3. 0.]
 [3. 1.]
 [3. 2.]
 [3. 3.]
 [3. 4.]
 [3. 5.]
 [4. 0.]
 [4. 1.]
 [4. 2.]
 [4. 3.]
 [4. 4.]
 [4. 5.]
 [5. 0.]
 [5. 1.]
 [5. 2.]
 [5. 3.]
 [5. 4.]
 [5. 5.]]
```

Figures 6 & 7: Coefficient Vectors (C) for Orders 3 & 5

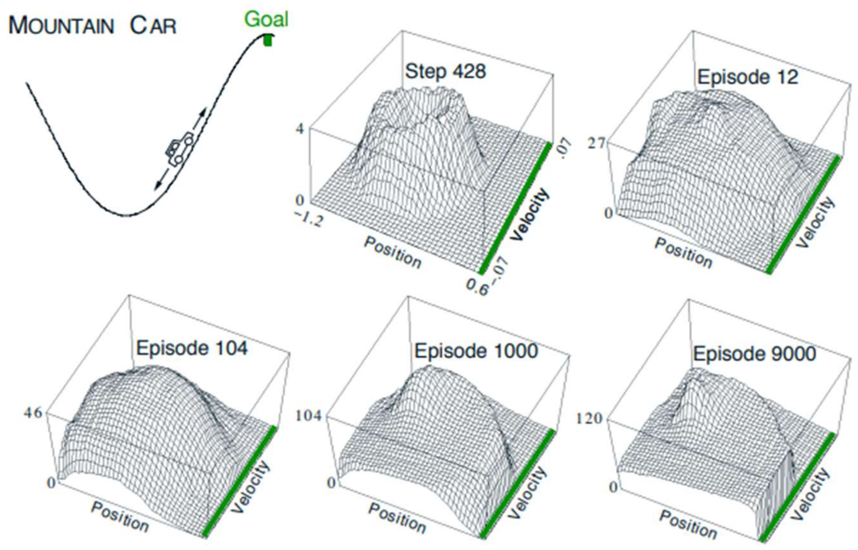


Figure 8: Sutton and Barto Surface Plots

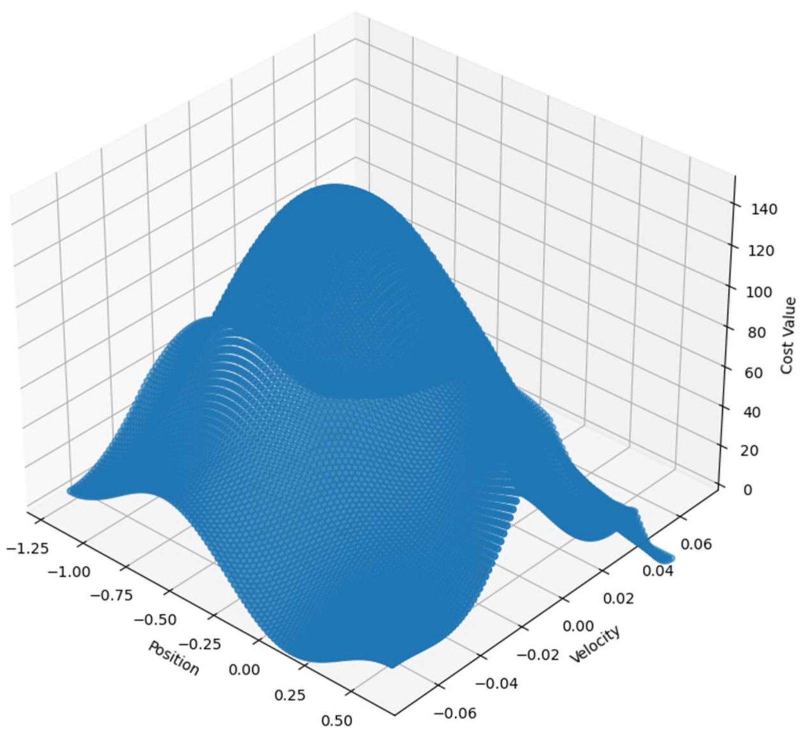


Figure 9: Surface Plot Over 1000 Episodes (Order 3)

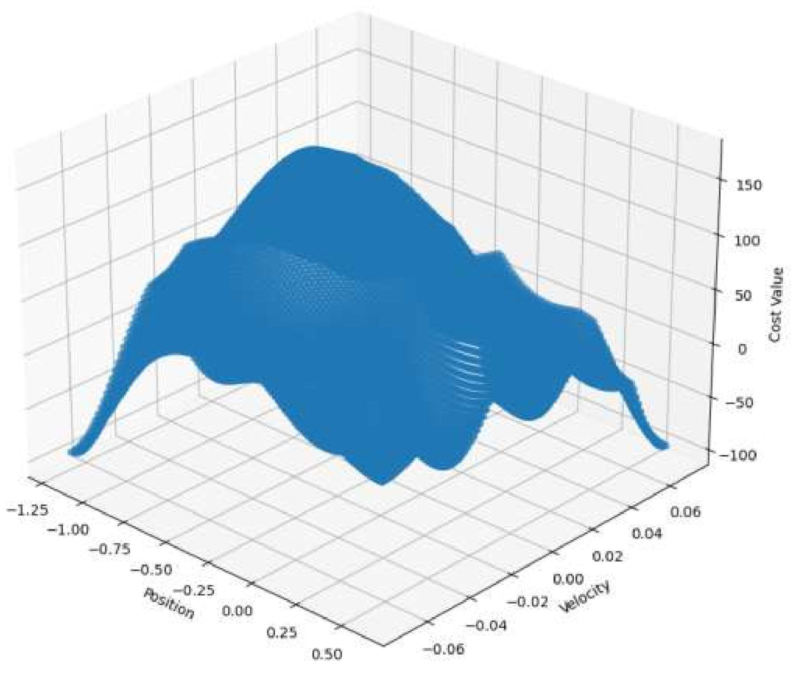


Figure 10: Surface Plot Over 1000 Episodes (Order 5)

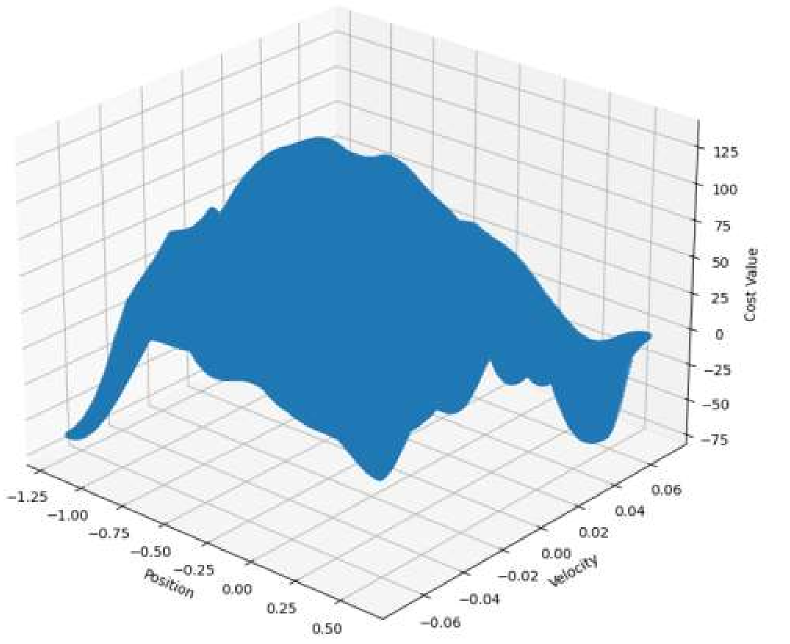


Figure 11: Surface Plot Over 1000 Episodes (Order 7)

With OpenAI gym, each episode starts from about -0.5, or in the middle between the two hills and the two continuous state space variables are converted to binary features using multivariate Fourier series from Figure 5. The Sarsa(λ) algorithm in Figure 1 (using accumulating traces) solved this task, learning a near optimal policy within 300 episodes. Figures 9-11 show the negative of the value function (cost to go function) learned on one run, using the parameters $\lambda = 0.9$, $\varepsilon = 0$, and $\alpha = 0.001$. As shown by the surface plots from the figures above, all states visited frequently are valued worse than unexplored states. This is because the actual rewards have been worse than what was expected. This continually drives the mountain car away from wherever it has been, to explore new states until a solution is found.

Part 3 - Consider if γ was less than 1 and the solution was many steps long? Additionally, what would happen if we had a zero-step cost and a positive goal reward, for the case where $\gamma = 1$, and the case where $\gamma < 1$?

Gamma (γ), in reinforcement learning, is the discount factor. Gamma influences an agent by giving importance to future rewards. Depending on the goal of agent and its environment, it can be handy to approximate the noise in future rewards. Gamma's value varies from 0 to 1, and if Gamma is closer to zero, the agent will tend to consider only immediate rewards.

Therefore, if Gamma was less than 1 and the solution was many steps long, the mountain car would still reach its terminal state. Because there is still a cost (negative reward) for each step that is not the terminal state, the agent will still trend to explore states with a less negative reward until the terminal state is found.

In the case that we had a zero-step cost and a positive goal reward for $\gamma = 1$, the agent would give less importance to future rewards ($\gamma = 1$). With the only valued reward being the terminal state, this scenario would take a very long time to converge (if it converges) because the discount value (γ) is so high. In the case that there is a zero-step cost and a positive goal reward for $\gamma < 1$, the agent will give more importance to future rewards. The lower the discount factor, the greater the chance that the solution will converge instead of just swiveling back and forth.

Results

From this program, I learned a great deal about Sarsa(λ), back tracing, value function approximation using Fourier basis, and Matplotlib. Prior to this program, I was still struggling with understanding how a lot of these RL algorithms worked. However, with the successful implementation of Sarsa(λ), I can say that I now understand, not only Sarsa(λ), but all the other methods that we have learned in class. Additionally, I grasp the concept of value function approximation and why it is useful. In hindsight, using Fourier basis over the continuous state space to approximate values seems much easier than discretizing the entire state space and playing around with bins until the solution finally converges. Finally, using Matplotlib for the first time was a good learning experience and I am sure it will come in handy in the future. However, I still couldn't completely get Matplotlib right. In Figures 10 and 11, the cost to go function had a couple corners that dip below zero. The cause for this is unknown, but I am certain that I am making a mistake in how I am plotting the graph with Matplotlib. In conclusion, multiple concepts from CENG 449: Advanced Topics Artificial Intelligence were put into practice for one program: Sarsa(λ) for the learning algorithm, traces for improved learning, basis functions for value function approximation, and verification of the program using graphical data.