Q1)

The reason why there is no pseudocode of Monte Carlo is that Monte Carlo method is just an extreme variation of TD methods. If we write down its pseudocode, it will not be so much different from n-step SARSA except n = T (T: episode length or infinity). For Monte Carlo method, the algorithm would generate episodes using ε-greedy policy with weights then based on that generate the returns. Then fit weights to the newly generated returns. It will most likely perform poorly on the Mountain Car because it is computationally expensive in complex settings. In Mountain Car problem, Monte Carlo method will not start to learn unless it can complete the first full episode with a random policy, and it may never finish the first episode.

Q2)

a)

Episodic Semi Gradient Expected Sarsa for Estimating optimal q

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

$$\sum_a \pi(s'|a)\,\hat{q}(s', a, w)$$

b)

Episodic Semi Gradient Q-learning for Estimating optimal q

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

$\gamma \max_a q(S', a, w)$

To run different function approximators in Q3, uncomment the corresponding line in the main() in Algorithm file.

num_trials = 10

num_eps = 100

gamma = 0.95

epsilon = 0.05

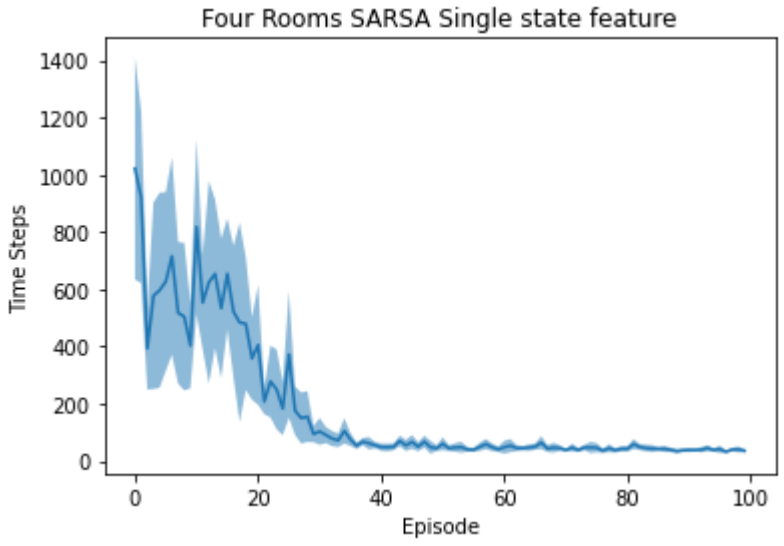
step_size = 0.05

Q3)

A)

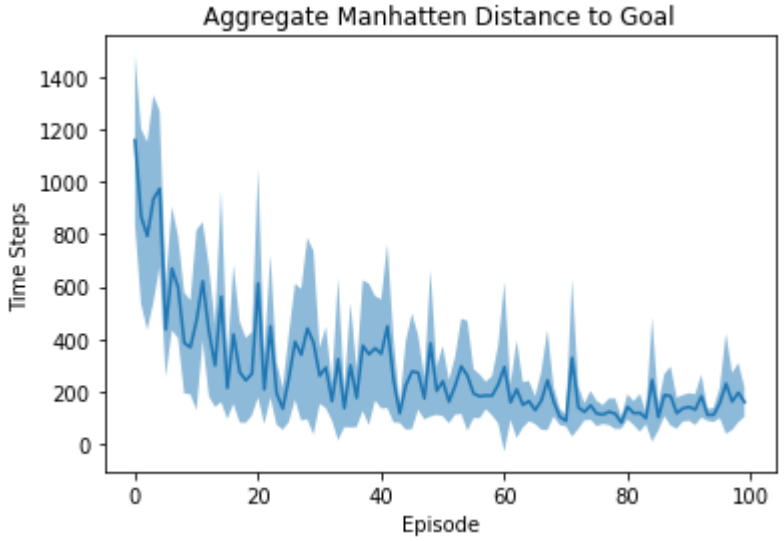
In order to compute features, I have created a separate function for each type of feature vector that simply takes the state and returns a vector. I use NumPy's broadcast feature to multiply the weight matrix and take the index of the corresponding action, giving a q_value for the state-action pair. To compute the gradient, I have created a 0's matrix and replaced the vector of the action taken with the state's feature vector.

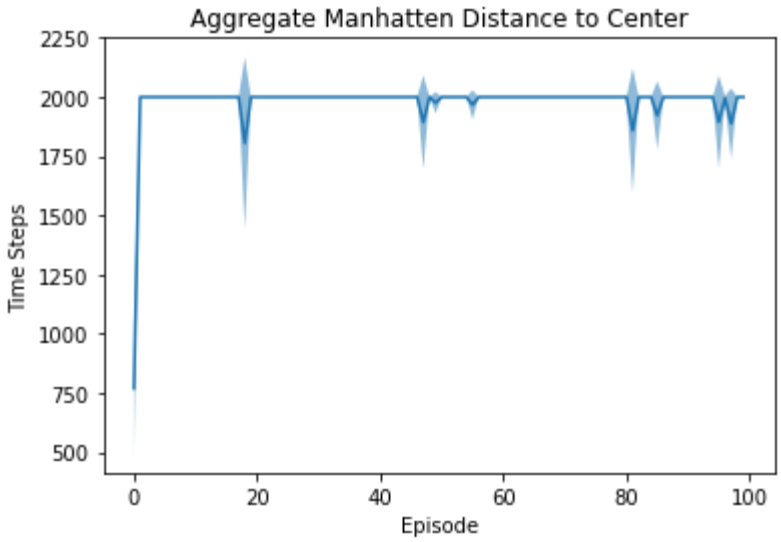
B)



C)

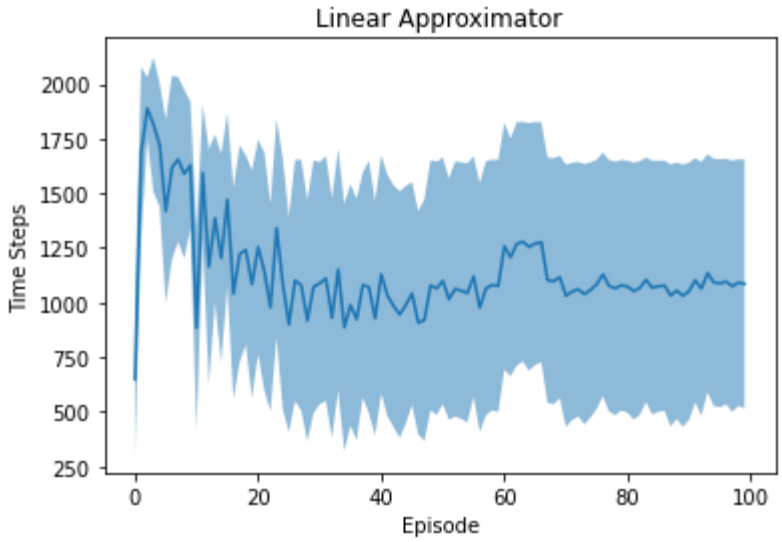
I tried aggregating by rows and Manhattan distance to goal and center. Manhattan distance to goal performed the best, but none performed as well as the tabular aggregator because the tabular aggregator seemed to find the optimal solution given the small episode length.



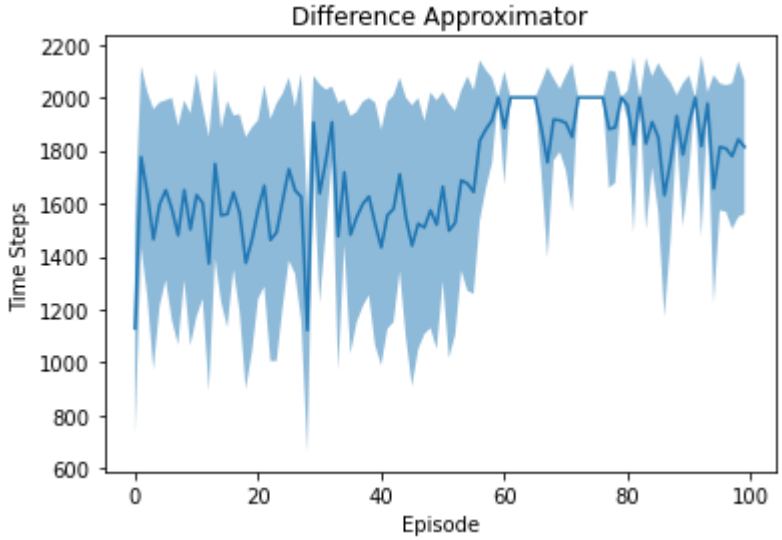


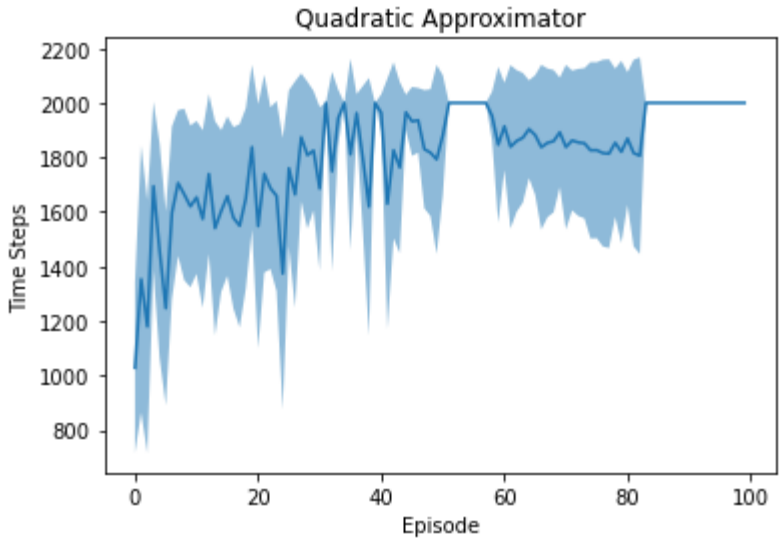
D)

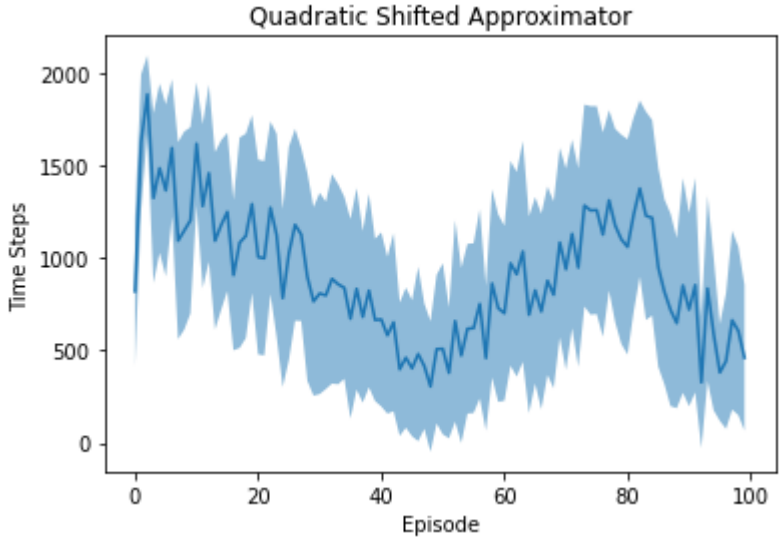
This constant is important because it limits the size of weights and determines the value of states with small feature values vs large feature values. Without it, there is no “default” value so that forces the feature weights to do undesired things, such as exploding in size/variance. I incorporated actions into the features by creating a separate weight vector for each action. There is much more variance than the state aggregation. It appears to perform better than most of the state aggregation methods, except the single state aggregation, because single state aggregation creates a q-table, and the state space of the Four Rooms problem is small enough so the entire table can be filled.


Linear Approximator
Time Steps
Episode
2000
1750
1500
1250
1000
750
500
250
0
20
40
60
80
100

E)


Difference Approximator
Time Steps
Episode
2200
2000
1800
1600
1400
1200
1000
800
600
0
20
40
60
80
100





The three features I used were “difference” which used x-y and y-x as features, along with 2 quadratic approximators, which used x^2, y^2, and x*y. The difference is that the original quadratic approximator used the [-.5,+.5] normalization, while the shifted approximator used [0,+1]. All three also had a bias term. The first two performed very poorly, which makes sense as they are somewhat similar to the “distance to center” state aggregator. In the original quadratic approximator, being nearly the start and goal give similar values since (-.5)^2 = (.5)^2. By shifting the normalized values to being only positive, we correct this. The shifted approximator appears to learn well, but then gets worse after about halfway. Perhaps this could be corrected by fine tuning the learning rate. The linear approximator relies on a linear relationship between x,y and value, which only works if the goal is at the largest possible x,y values (10,10).