Electronics and Computer Science

Faculty of Physical and Applied Sciences

University of Southampton

Daniel Lee

12/12/16

Linear Quadratic Regulation using Reinforcement Learning

Project supervisor: Bing Chu

Second examiner: Mohammed El-Hajjar

A project progress report submitted for the award of

MEng Electronic Engineering with Artificial Intelligence

Abstract

The purpose of this project is to demonstrate the application of concepts from *reinforcement* *learning* into discrete-time *linear quadratic regulation (LQR)* control systems. The progress that has been achieved so far includes the background reading of the core topics, numerical derivations of the *Bellman equations* for *state* and *action-state* pair *values*, simulations on *greedy* and *-greedy* algorithms and mathematical analysis of *policy evaluation*, *policy iteration* and *policy improvement* in dynamic systems. The plans for the remaining work consists of further analysis on the theory of *optimal control in LQR*, derivations of the quadratic *cost function* of discrete-time LQR using *Ricatti’s algebraic equation*, application of the *Q-Learning* method in LQR and simulation of the inverted pendulum balancing problem using *Markov decision process* and Bellman equations.

Contents

Abstract 2

Contents 3

1 Project Goals 4

2 Background and report of literature search 5

2.1. Reinforcement Learning 5

2.2. System Elements 5

2.3. Markov Decision Process 6

2.4. Bellman Equations 6

3 Report on Technical Progress 8

3.1 K-Arm Bandit Problem 8

3.2 Greedy Method 8

3.3 e-Greedy Method 9

3.4 Optimal Policy 10

3.5 Grid World Problem 10

4 Plan of remaining work 12

References 14

Appendices 15

Appendix A - MATLAB Code for Greedy function 15

Appendix B - MATLAB Code for e-Greedy function 17

Appendix C - Grid World Problem Solution 19

# Project Goals

In *optimal control* theory, the goal is to compute and build a *controller* that can minimise the cost, which is expressed as a quadratic function, of operating a dynamic system described by ordinary differential equations. However, in practice, often times a complete knowledge of the system’s dynamics is not known and therefore, finding the optimal controller for the system usually involves an exhaustive iterative process. The aim of this project is to show the procedure in which solutions to linear quadratic regulation can be obtained through the use of methods from the field of reinforcement learning.

# Background and report of literature search

With the purpose of getting familiar with the core topics of this project, a great portion of the available time was invested in reading the appropriate literature. For the topic of reinforcement learning, the main source of reference was the book titled ‘Reinforcement Learning: An Introduction’ by Sutton and Barto [1]. With respect to Linear Quadratic Regulation, the research was mainly based on the article ‘*Reinforcement Learning and Feedback Control*’ by Lewis, Vrabie and Vamvoudakis [2].

**2.1. Reinforcement Learning**

To begin with, it is important to define what *reinforcement learning* is. Reinforcement learning is ‘*learning by interacting with an environment’* [3]. An agent can be considered to be using reinforcement learning if it adjusts the way it behaves at different states of the environment by considering the consequences of its past experiences. This can be compared to the concept of ‘*trial and error’*, which is the simplest natural learning behavior presented in animals and humans alike. By assessing the consequences of the actions taken in the past, the agent will seek to select the action that provided it with the best outcome. By following an iterative process for this procedure, the agent’s knowledge on the consequences for the different actions will become increasingly more ‘*clear*’. This will consequently shape the agent’s behavior in favor of selecting the most rewarding actions.

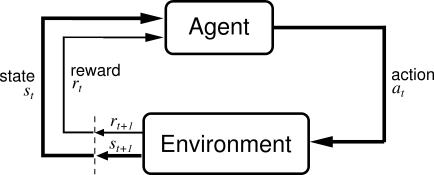


Figure 1: Agent-environment interaction (*Sourced from:* [1])

**2.2. System Elements**

For a more formal description, the reinforcement learning system can be partitioned into three main elements: a *policy*, a *reward* and a *value* [4]. A *policy* is a rule that maps the actions to the states of the environment. In simpler words, it determines the way in which the agent should behave in different situations that occur during its interaction with the environment. When an agent selects an action from a state, it collects a numerical value called *reward*. A reward quantitatively expresses how positive the performance of the action taken at a state was. While a reward represents an immediate acquisition, the *value* of a state provides the information of the total expected reward, or *return*, that is expected to be collected by following the state. Therefore, it can be considered as a quantity that expresses how much the agent can benefit in the long-term. A graphical representation of the agent-environment interaction, illustrating the elements described above, is shown in Figure 1.

**2.3. Markov Decision Process**

A dynamic system for which ‘*the environment’s response at depends only on the state and action representations at time*  [1], is said to have the *Markov property*. In other words, a dynamic system has the Markov property if at any each state-action pair, , in the environment, equation (1) is equivalent to equation (2).

A reinforcement learning task that has the Markov property and which its environment has a finite number of states and actions is called a *Markov Decision Process*. In these systems, the agent’s interaction with the environment is governed by a set of equations called the *transition probabilities*, equations (3) and (4).

Equation (3) describes the probability that the agent will transition to a certain state from choosing action at state and equation (4) provides the estimated reward from transitioning to state by choosing action from state . These set of equations describe the dynamics of the system and consequently determine the agent’s behavior throughout its interaction with the environment.

**2.4. Bellman Equations**

The *value* of a state or state-action pair represents how positively an action or state-action pair performed in the past. This information is essential for the agent to take the most appropriate decision. Values for each state, or *state-value*s , can be computed using equation (5), where is the *discounted return* for and is the policy in which the agent is acting on.

As illustrated in the equation, the value of a function is an estimate of the total discounted return from the current time, , to . If the discount factor is set closer to 1, it means that the agent becomes more *farsighted*. The same expression can be obtained for the state-action pair, or *action-values* . This is shown in equation (6).

By further expanding equation (5), the derivation for the stochastic Bellman equation for value functions can be obtained. This procedure is shown in equation (7).

The same can be performed on the action-value to derive the Bellman equation for the action-value, equation (8).

These two Bellman equations are the base equations necessary to approach a reinforcement learning problem through *on-line* *dynamic programming*.

# Report on Technical Progress

In parallel to the background research, practical simulations were performed on the relevant topics with the aim to complement the theory.

**3.1. K-Arm Bandit Problem**

A good example that illustrates the basic concepts of reinforcement learning in practice is the k-arm bandit problem. In this scenario, the aim of the agent is to maximize its return by pulling the desired arm, among the k number of arms, from the bandit, at each finite discrete step, or turn. To solve this problem, two simple methods can be implemented. These are the *greedy method* and the *ε-greedy* method.

**3.2. Greedy Method**

The greedy method represents a policy where the agent always chooses the action that has the highest expected reward. In the case of the k-arm bandit problem, the expected reward of an action, or value , can be expressed with equation (9), where and represents the number of times that the actions was selected before the current time .

The equation shows that every time the agent selects the action, or arm, , its value is updated by averaging all the past collected rewards. This means that if an action was selected an infinite amount of times, , its value would eventually converge to its actual value . Therefore, at every turn, the agent will pull the arm that returned the most profit in past experiences. This can be represented by (10).

However, as in many real cases, the bandit problem is a stochastic system where the immediate reward has a probability distribution, or noise, that might negatively affect the agent’s shortsighted decisions. An agent could be ignoring the optimal action only based on few past experiences and never come back to it again. To further demonstrate this, a simulation of this example was performed by creating a MATLAB function called greedy, Appendix A. The greedy function’s inputs are, the number of bandits, the number of actions, the number of turns, and the Gaussian distribution parameters to set the actual action values and the Gaussian distribution parameters to set the noise for every immediate reward. For this simulation, the initial actual values for all were selected from a set of numbers from a Gaussian distribution . In addition to this, Gaussian noise was added to all immediate rewards. For a more accurate evaluation, the results for the 15-arm bandit problem were averaged over 1000 bandits. Figure 2 illustrates how often the agent converged to the optimal action over a 1000 steps.

By analyzing the graph, it can be seen that greedy algorithm led the agent to converge to the optimal action in around %35 of the cases. To improve the performance of the agent in the long term, it is possible to use a slight variation of this method called the ε-Greedy method.



Figure 2: Total Average Return Figure 3: Optimal Action

**3.3. ε-Greedy Method**

From the previous subsection, the theory and implementation of the greedy algorithm were shown. In this section, a variation of the greedy algorithm called ε-greedy will be explored. One of the issues from the ε-greedy was that because it always forced the agent to choose the best action-value in every state, if a particular action-value was low from the start, the agent never chose it for the rest of the simulation, therefore, never being able to discover its actual value. To avoid this problem, a variant of this algorithm that allows the agent to explore from time to time can be used. In every state, the agent will randomly choose an action with equal probability in all of them. This means that because if an action is chosen infinite times it converges to its actual value, this will allow the agent to have a more precise information of its options. This algorithm is called ε-greedy. To simulate this method, a function was develop in MATLAB and can be found in Appendix B.

The first simulation was created by using an ε of 0.10. The 15-bandit problem was run for 1000 steps and averaged over 1000 bandits. The result was then compared to the total average return for the greedy method in Figure 2. It can be seen that although for the first 400 steps the greedy algorithm was able to collect a higher average return, the ε-greedy method for shows a better performance in the long term. In theory, for ε-greedy algorithms, as the number of steps goes to infinity, the average reward would converge to the optimal reward.

A plot showing the percentage in which the agent converged to choose the optimal value can be seen in Figure 3. The graph shows that as it reaches the last time step, the optimal action percentage reaches around 60 percent, which is almost as double as the performance for the greedy method. This is because we allow the agent to explore for 10 percent of the time. This means that although 10 percent of the time the action might not receive the maximum immediate reward, it will aid the agent to get a more precise estimate of the actual state-action values.

For the last ε-greedy simulation, an epsilon value of 0.05 was used. From Figure 3, it can be seen that although the optimal action percentage does not increase as steeply as the previous ε-greedy simulation, it still outperforms the greedy method. If the number of steps were to be increased, this version of the e-greedy algorithm would outperform the other ones and eventually converge to an accuracy of optimal action to 99.9 percent. Taking these simulations into account, depending on the requirements of the reinforcement learning problem, it is important to set a suitable balance between *exploration* and *exploitation*. If the immediate performance is important, it would be more effective to use a lower exploration factor, or in this case epsilon, and if the priority lies in the long term performance, a higher exploration factor would be more convenient.

**3.4. Optimal Policy**

The *optimal policy* for a finite MDP can be obtained through two methods called *policy evaluation* and *policy iteration,* which are based on the Bellman equations for state and action-state values derived in equations (7) and (8). Policy evaluation is used to compute the deterministic state-values for all for an arbitrary policy , using the values obtained in the last episode, . This can be represented using equation (9).

By iterating this process for many times, , a state-value for all with an increasing accuracy can be computed. With the resulting information, an optimal policy for the dynamic system can be found. This process is called policy iteration.

**3.5. Grid World Problem**

To show these methods in practice the ‘*grid world*’ problem from *Sutton and Barto’s* book [1] was reproduced solved by hand in Appendix C. The grid world problem can be classified as an *episodic* finite MDP, therefore there are a finite number of states and actions and does not require a discounting factor . For the grid shown in Figure 4, the 14 states are represented by each of the squares and the four possible actions that can be taken at each state are . The objective is to find the optimal policy for each state, that leads the agent to one of the terminal states, shaded in grey, in the minimum number of steps. For every step taken the immediate reward is -1 and if the agent takes an action that makes it go off the grid, it will return back to the last state. With the described system dynamics, policy iteration was applied for 3 times, starting with initial state-values set to 0. After three iterations, the optimal policy was retrieved using the equation for the optimal action-value in equation (10).

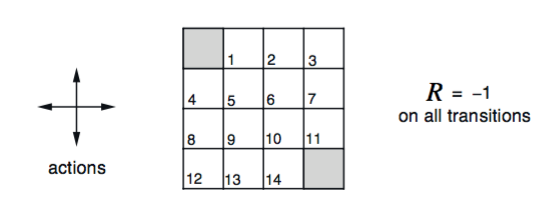
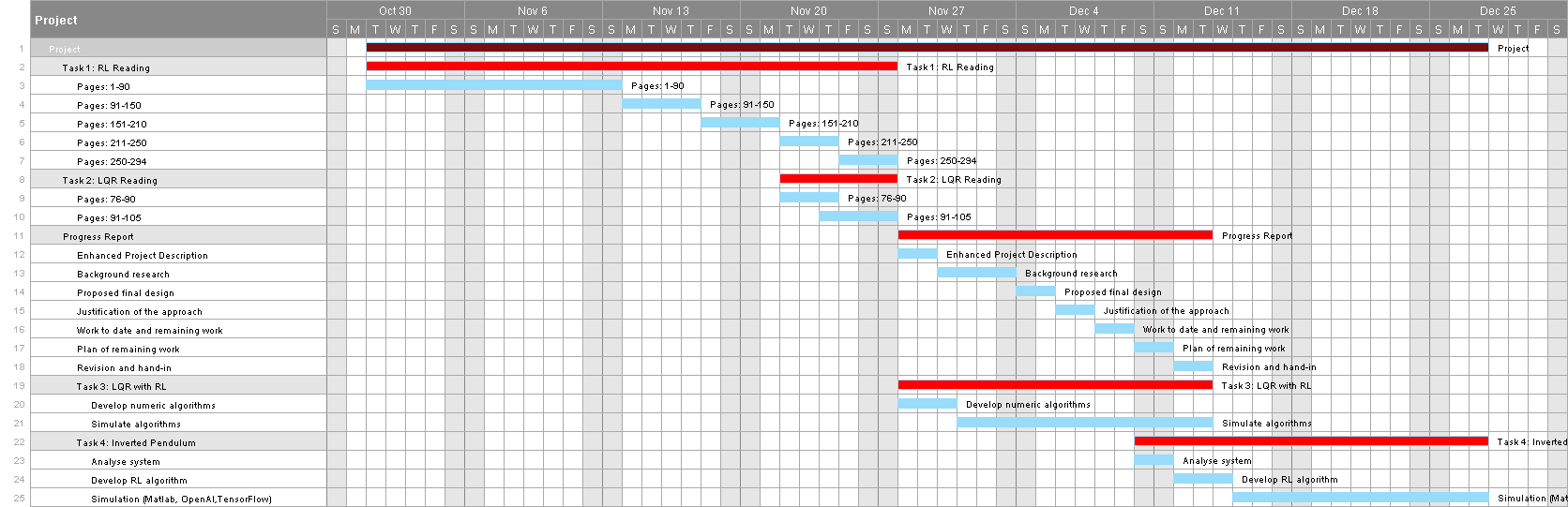
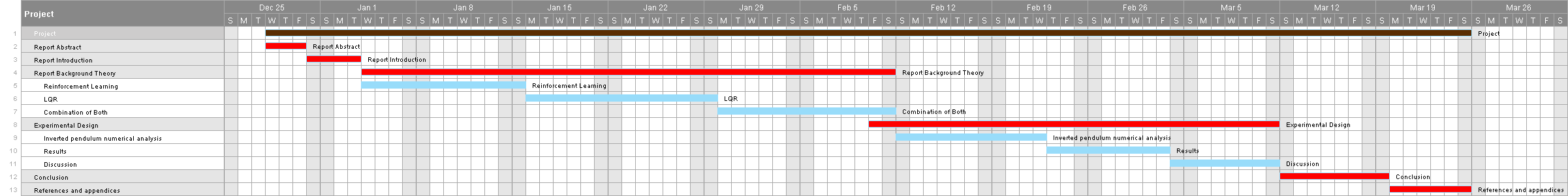


Figure 4: 4x4 Grid world Problem (*Sourced from:* [1])

# Plan on Remaining Work

Due to the fact, that most of the research and simulations done up to date were mainly based on the area of reinforcement learning, the remaining work will be focused on the methods for applying reinforcement learning in LQR problems. Sub-tasks including a deeper research on the theory of *optimal control*, derivations of the cost function of the *infinite-horizon discrete-time LQR* system, *Q-Learning application* in LQR systems and solution to the *episodic* inverted pole-balancing problem will be made. For a broader view of the project plan, a *Gantt Chart* has been included below.

Figure 5: Gant Chart for Semester 1

Figure 6: Gantt Chart for Semester 2

References

[1] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. London, England: The MIT Press, 2012.

[2] D. V. Frank L. Lewis and K. G. Vamvoudakis, “Reinforcement learning and feedback control,” IEEE Control Systems Magazine, pp. 76–105, November 2012.

[3] F. Woergoetter and B. Porr, “Reinforcement learning,” vol. 3, no. 3, p. 1448, 2008.

[4] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, pp. 7–10. The MIT Press, 2012.

Appendices

**Appendix A – MATLAB Code for greedy function**

function [total\_average\_return, optimal\_action, optimal\_return] = greedy(bandit\_n, actions\_n, timesteps, actual\_distr, noise\_distr)

% This function generates the total average return for n-bandit problems

% with the assigned number of bandits, number of timesteps, number of actions,

% actual values and noise.

% Algorithm Implementation Steps

% Step 1: Set actual action-values Q\* from nrand(0,1)

% Step 2: Initialize action-values adding Gaussian Noise

% Step 3: Start Greedy decision process

% Step 4: Choose action with the maximum estimated action-value

% Step 5: Update accumulated reward

% Step 6: Update estimated action-value

% Step 7: Store total average return for each bandit

% Step 8: Check if it converged to the optimal value

% Step 9: Plot average total return against optimal average return

% CONSTANTS

% bandit\_n = 20;

% actions\_n = 5;

% timesteps = 1000;

% actual\_distr = [0,1];

% noise\_distr = [0,1];

% External variables

tot\_avg\_rew = zeros(1,bandit\_n);

tot\_opt\_rew = zeros(1,bandit\_n);

optimal\_choice = 0;

for bandit=1:bandit\_n

% Clean action-values every loop

actual\_q = zeros(1,actions\_n);

estimate\_q = zeros(1,actions\_n);

% 1. Generate actual values q\*(A) and store them in q[] OK

for i=1:actions\_n

actual\_q(i) = normrnd(actual\_distr(1),actual\_distr(2));

end

% 2. Initiliaze action-values

for i=1:actions\_n

gauss\_noise = normrnd(noise\_distr(1),noise\_distr(2));

estimate\_q(i) = actual\_q(i) + gauss\_noise;

end

% 3. Greedy decision process

% Clean values every loop

total\_reward = 0;

acc\_reward = zeros(1, actions\_n);

opt\_reward = 0;

action\_counter = zeros(1,actions\_n);

for i=1:timesteps

% 4. Choose option with max(estimate\_q)

[~, index] = max(estimate\_q);

[maxval,~] = max(actual\_q);

gauss\_noise = normrnd(noise\_distr(1),noise\_distr(2));

gauss\_noise2 = normrnd(noise\_distr(1),noise\_distr(2));

% Rewards update

imm\_reward = actual\_q(index) + gauss\_noise;

total\_reward = total\_reward + imm\_reward;

opt\_reward = opt\_reward + maxval + gauss\_noise2;

% 5. Update accumulated reward

acc\_reward(index) = acc\_reward(index) + imm\_reward;

% 6. Update estimated value

action\_counter(index) = action\_counter(index) + 1;

estimate\_q(index) = acc\_reward(index) / action\_counter(index);

end

% 7. Store total return for each bandit

tot\_avg\_rew(bandit) = total\_reward / timesteps;

tot\_opt\_rew(bandit) = opt\_reward / timesteps;

% 8. Check it it converged to the optimal value

[~, optimal\_index] = max(action\_counter);

[~, actual\_index] = max(actual\_q);

if optimal\_index == actual\_index

optimal\_choice = optimal\_choice + 1;

end

end

% 9. Plot total average return against optimal average return

total\_average\_return = sum(tot\_avg\_rew) / bandit\_n;

optimal\_action = optimal\_choice / bandit\_n;

optimal\_return = sum(tot\_opt\_rew) / bandit\_n;

**Appendix B – MATLAB Code for e-greedy function**

function [total\_average\_return, optimal\_action, optimal\_return] = egreedy(bandit\_n, actions\_n, timesteps, actual\_distr, noise\_distr, e)

% EGREEDY Returns the total average return, optimal return and optimal action percentage for the

% n-bandit problem with the given number of bandits, actions, timesteps, action-values and noise distributions

% and epsilon.

% EGREEDY(bandits, actions, timesteps, action\_value\_distribution[mean,

% variance], noise\_distribution[mean,variance], epsilon)

% Algorithm Implementation Steps

% Step 1: Set actual action-values Q\* from nrand(0,1)

% Step 2: Sort action-values in ascending or

% Step 3: Initialize action-values adding Gaussian Noise

% Step 4: Start e-Greedy decision process

% Step 5: Check if x is lower than epsilon

% Step 6: Choose action with the maximum estimated action-value

% Step 7: Update accumulated reward

% Step 8: Update estimated action-value

% Step 9: Store total average return for each bandit

% Step 10: Plot average total return against optimal average return

% DEBUG VARIABLES

% bandit\_n = 20;

% actions\_n = 5;

% timesteps = 1000;

% actual\_distr = [0,1];

% noise\_distr = [0,1];

% External variables

tot\_avg\_rew = zeros(1,bandit\_n);

tot\_opt\_rew = zeros(1,bandit\_n);

optimal\_choice = 0;

for bandit=1:bandit\_n

% Clean action-values every loop

actual\_q = zeros(1,actions\_n);

estimate\_q = zeros(1,actions\_n);

% 1. Generate actual values q\*(A) and store them in q[] OK

for i=1:actions\_n

actual\_q(i) = normrnd(actual\_distr(1),actual\_distr(2));

end

% 2. Initiliaze action-values

for i=1:actions\_n

gauss\_noise = normrnd(noise\_distr(1),noise\_distr(2));

estimate\_q(i) = actual\_q(i) + gauss\_noise;

end

% 3. Greedy decision process

% Clean values every loop

total\_reward = 0;

acc\_reward = zeros(1, actions\_n);

opt\_reward = 0;

action\_counter = zeros(1,actions\_n);

for i=1:timesteps

% 4. Choose option with max(estimate\_q)

[~, index] = max(estimate\_q);

% 5. If we fall under epsilon, choose random action

x = rand;

if x < e

index = randi(actions\_n);

end

[maxval,~] = max(actual\_q);

gauss\_noise = normrnd(noise\_distr(1),noise\_distr(2));

gauss\_noise2 = normrnd(noise\_distr(1),noise\_distr(2));

% Rewards update

imm\_reward = actual\_q(index) + gauss\_noise;

total\_reward = total\_reward + imm\_reward;

opt\_reward = opt\_reward + maxval + gauss\_noise2;

% 6. Update accumulated reward

acc\_reward(index) = acc\_reward(index) + imm\_reward;

% 7. Update estimated value

action\_counter(index) = action\_counter(index) + 1;

estimate\_q(index) = acc\_reward(index) / action\_counter(index);

end

% 8. Store total return for each bandit

tot\_avg\_rew(bandit) = total\_reward / timesteps;

tot\_opt\_rew(bandit) = opt\_reward / timesteps;

% 9. Check it it converged to the optimal value

[~, optimal\_index] = max(action\_counter);

[~, actual\_index] = max(actual\_q);

if optimal\_index == actual\_index

optimal\_choice = optimal\_choice + 1;

end

end

% 10. Plot total average return against optimal average return

total\_average\_return = sum(tot\_avg\_rew) / bandit\_n;

optimal\_action = optimal\_choice / bandit\_n;

optimal\_return = sum(tot\_opt\_rew) / bandit\_n;

**Appendix C – Grid World Problem Solution**