Electronics and Computer Science
Faculty of Physical and Applied Sciences
University of Southampton

Daniel Lee

<Date>

Linear Quadratic Regulation using Reinforcement Learning

Project supervisor: Dr Bing Chu
Second examiner: Dr Mohammed El-Hajjar

A project report submitted for the award of
MEng Electronic Engineering with Artificial Intelligence

Abstract

The aim of this project is to demonstrate that the *optimal control* for a *Linear Quadratic Regulation* (LQR) problem can be obtained by the use of *Reinforcement Learning* methods, without prior knowledge of the system's dynamics. For a better understanding of the core topics of reinforcement learning, simulations and demonstrations of core reinforcement learning topics, including *Bellman equations*, *greedy* and ε*-greedy* algorithms, *policy evaluation*, *policy iteration*, *policy improvement*, *temporal difference learning*, *exploration* and *exploitation, Markov Decision Process* and *Q-Learning*, were performed. Related to control systems, research on *optimal control theory*, *state-space modelling*, *cost function* and the *Algebraic Riccati Equation* was also made, as they were essential concepts for this project. Finally, the inverted pendulum problem, which consists of balancing an inverted pendulum attached to a cart that can only move sideways, was solved through the application of the *adaptive policy iteration* algorithm using Q-Learning.

Contents

Acknowledgements

I would like to thank Dr Bing Chu and Dr Modammed El-Hajjar for their guidance and support on this project. I would also like to thank my colleagues and closest friends Alois Klink and Reuben Ng for helping me maintain my motivation and interest on the topic of Artificial Intelligence.

1 Introduction

If an accurate description of the dynamics of a system is available, the solution for the optimal control, within the constraints of LQR, can be trivially obtained by solving the Algebraic Riccati equation. However, in many cases, an exact representation of a dynamic system's model cannot be easily found. This is where reinforcement learning can fill in the gaps. A very powerful feature of reinforcement learning is that it can *learn* through experience. This means that by analysing several aspects of the behaviour of a system throughout many finite iterations, or *episodes*, a close estimate of the system dynamics can be computed, therefore allowing us to find the optimal policy.

1.1 Background Reading

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].

1.1.1 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 behaviour 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 behaviour in favour of selecting the most rewarding actions.

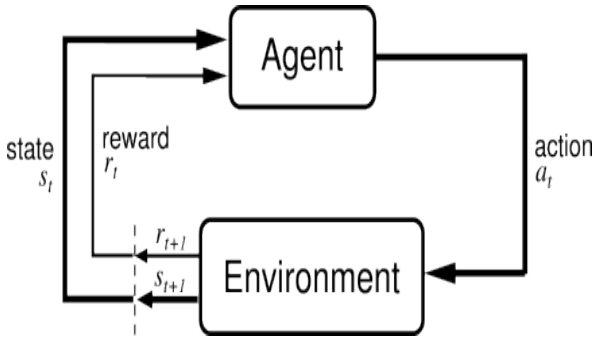


Figure 1: Agent-environment interaction (Sourced from: [1])

1.1.2 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.

1.1.3 Markov Decision Process

A dynamic system for which '*the environment's response at* $t+1$ *depends only on the state and action representations at time* t [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, S_t, A_t, in the environment, equation (1) is equivalent to equation (2).

$$\mathbf{P}\,\mathbf{r}\{R_{t+1} = r, S_{t+1} = s' \,|\, S_0, A_0 R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\}\ (1)\ [1]$$

$$\mathbf{P}\,\mathbf{r}\{R_{t+1} = r, S_{t+1} = s' \,|\, S_t, A_t\}\ (2)[1]$$

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).

$$\mathrm{p}(s'|\mathrm{s},\mathrm{a}) = \mathbf{P}\,\mathbf{r}\{S_{t+1} = s' \,|\, S_t, A_t\}\ (3)[1]$$

$$\mathrm{r}(\mathrm{s},\mathrm{a},\mathrm{s}) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'](4)[1]$$

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

1.1.4 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 $v_\pi(s)$, can be computed using equation (5), where G_t is the *discounted return* for $0 < \gamma \leq 1$ and π is the policy in which the agent is acting on.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] (5)[1]$$

As illustrated in the equation, the value of a function is an estimate of the total discounted return from the current time, t, to $t = \infty$. If the discount factor γ 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 $q_\pi(s, a)$. This is shown in equation (6).

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\,, A_t = a\right] (6)[1]$$

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).

$$\begin{aligned} v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} p(s'|s, a)[r(s, a, s'] \\ &+ \gamma\, \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_{t+1} = s'\right] \\ &= \sum_a \pi(a|s) \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma\, v_\pi(s')] (7)[1] \end{aligned}$$

The same can be performed on the action-value $q_\pi(s, a)$ to derive the Bellman equation for the action-value, equation (8).

$$\begin{aligned} q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_t = s, A_t = a\right] \\ &= \sum_{s'} p(s'|s, a)[r(s, a, s'] \\ &+ \gamma\, \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2}|S_{t+1} = s', A_{t+1} = a\right] \\ &= \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma\, v_\pi(s')] (8)[1] \end{aligned}$$

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

1.1.5 Linear Quadratic Regulation

Linear quadratic regulation defines a problem in which the system dynamics are defined by linear differential equations and the cost is represented by a quadratic function [5]. In this context, the aim is to minimize the cost generated by the system during its lifetime.

For a discrete-time LQR, the governing equations are defined by equations (9), (10), (11) and (12).

$$x_{k+1} = Ax_k + Bu_k (9)[2]$$

$$u_k = -Kx_k (10)[2]$$

$$r = (x_i^T Q x_i + u_i^T R u_i) (11)[2]$$

$$J_k = \sum_{i=k}^{\infty} r = \frac{1}{2} \sum_{i=k}^{\infty} (x_i^T Q x_i + u_i^T R u_i)\ (12)[2]$$

Equation (9) defines the state transition, (10) is the control policy, (11) the immediate reward and (12) represents the cost function, in this case, the infinite-horizon cost, since it is the total sum of the costs from the starting point to infinity. The matrices Q and R defined in (11) are arbitrary matrices that regulate the weight, or importance, of the cost that the states x and inputs u contribute to the system. These equations contain all the required information to describe a reinforcement learning problem, thus allowing us to build the bellman equations from them, as shown in equations (13) and (14).

$$V(x_k) = \frac{1}{2}(x_k^T Q x_k + u_k^T R u_k) + V(x_{k+1})$$

$$V(x_k) = \frac{1}{2}(x_k^T Q x_k + u_k^T R u_k) + \frac{1}{2} \sum_{i=k+1}^{\infty} (x_i^T Q x_i + u_i^T R u_i)\ (13)[2]$$

$$Q(x_k, u_k) = \frac{1}{2}(x_k^T Q x_k + u_k^T R u_k) + V(x_{\square+1}) (14)[2]$$

Equations (13) and (14), are the value and the action-value functions, respectively, and these are equivalent to the standard bellman previously defined in equations (7) and (8). For quadratic value functions, equation (13) can be rewritten as (15).

$$V_k(x_k) = \frac{1}{2} x_k^T P x_k (15)[2]$$

As in any reinforcement learning problem, the objective is to find the optimal policy. In the LQR case, this is the policy that will minimize the cost function J. For the infinite horizon discrete-time LQR, this can be achieved by minimizing the *Hamiltonian function*, which is the function that represents the total energy of the system [6], and can be expressed as the *temporal difference error* when transitioning from a state to another. By doing so, as in (16), the optimal control policy (17) can be obtained. Moreover, by substituting (17) into (15) and

rearranging, the algebraic Riccati equation can be computed and solved to find the matrix P, required in the optimal control (17).

$$\frac{\partial H(x_k, u_k)}{\partial u_k} = 0 (16)[2]$$

$$u_k = -Kx_k = -(B^T PB + R)^{-1} B^T PAx_k (17)[2]$$

$$A^T PA - P + Q - A^T PB(B^T PB + R)^{-1} B^T PA = 0 (18)[2]$$

The resulting optimal control solution will minimize the cost of the system along its entire trajectory.

1.2 Previous Work

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

1.2.1 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.

1.2.2 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 $Q_t(a)$, can be expressed with equation (19), where $a \in A$ and k represents the number of times that the actions was selected before the current time t.

$$Q_t(a) = \frac{R_1 + R_2 + \cdots + R_{K_a}}{K_a} (19)$$

The equation shows that every time the agent selects the action, or arm, a, its value is updated by averaging all the past collected rewards. This means that if an action was selected an infinite amount of times, $k \to \infty$, its value would eventually converge to its actual value $Q_*(a)$. Therefore, at every turn, the agent will pull the arm that returned the most profit in past experiences. This can be represented by (20).

$$Q_t(A_t^*) = max_a Q_t(a) (20)$$

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 B.1. 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 $Q_*(a)$ for all $a \in A$ were selected from a set of numbers from a Gaussian distribution $N(0,1)$. In addition to this, Gaussian noise $N(0,3)$ was added to all immediate rewards $r \in R$. 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 a^* 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.

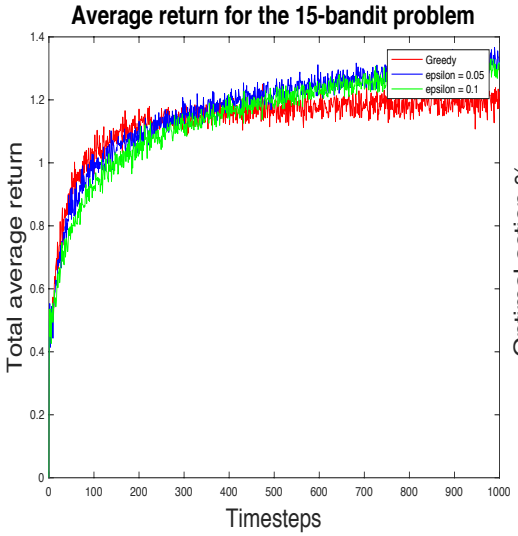


Figure 2: Total Average Return

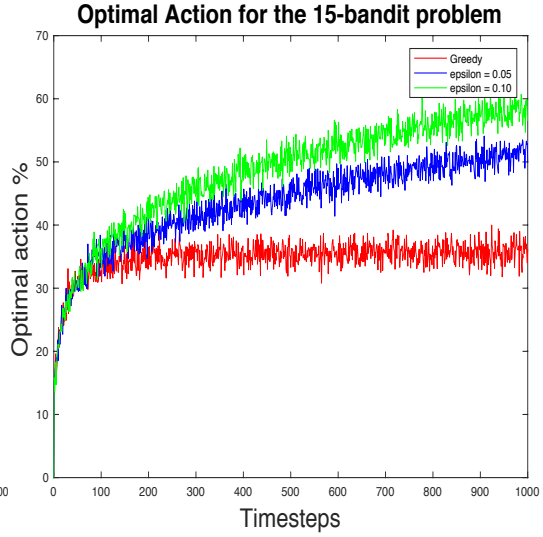


Figure 3: Optimal Action

1.2.3 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.2.

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 $\varepsilon = 0.10$ 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.

1.2.4 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 $V_{k+1}(s)$ for all $s \in S$ for an arbitrary policy π, using the values obtained in the last episode, $V_k(s)$. This can be represented using equation (21).

$$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s](21)[1]$$

By iterating this process for many times, $k \to \infty$, a state-value $V(s)$ for all $s \in S$ 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.

1.2.5 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 γ. 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 $A = \{Up, Down, Left, Right\}$. 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 (22).

$$\pi'(s) = argmax_a q_\pi(s,a)$$
$$= argmax_a \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v_\pi(s')](22)[1]$$

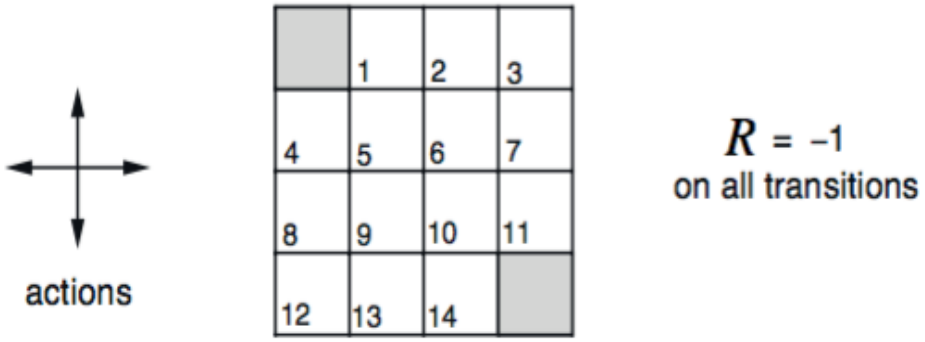


Figure 4: 4x4 Grid world Problem (*Sourced from:* [1])

1.3 Adaptive controller using Q Learning

In contrast to all the problems analysed so far, such as the grid world problem analysed above, where all the environment's characteristics where known in priori, there are systems in which this information is unavailable or simply unobtainable by mere observation. Aspects of the system such as state transitions, immediate rewards and number of states, may be unknown. In some other cases, not only the absence of information is a problem, but also, the scale of the system states. This is because if the number of states in a system were extremely large, e.g. $>10^{99}$, generating the value function of all of these states would take an excessive amount of time and computational power.

The adaptive controller algorithm can overcome these issues because it does not require having knowledge of the model of a system to converge to the optimal policy. Furthermore, it is unaffected by the number of states in the system as it uses *value function approximation*, which is a powerful tool to estimate the value of a state [7], with the aid of a neural network.

1.3.1 Q Function

The term Q Learning is derived from the action-value function, or *Q function*, defined in equation (6) and stores the information of the expected reward, or cost, in the case of LQR, of taking a particular action in a state. For a given policy, the Q Learning algorithm consists of applying policy evaluation and policy improvement in an online procedure, and updating the Q function of the states in the state space every discrete step [8].

In LQR, the Q function can be defined as in the following equation (23).

$$Q(x,u) = r(x,u) + \gamma V(x,u)$$

$$= x'Qx + u'Ru + \gamma(Ax + Bu)'K(Ax + Bu)$$

Rearranging into matrix form,

$$= [x,u]' \begin{bmatrix} Q + \gamma A'KA & \gamma A'KB \\ \gamma B'KA & F + \gamma B'KB \end{bmatrix} [x,u] (23)[9]$$

$$S = \begin{bmatrix} Q + \gamma A'KA & \gamma A'KB \\ \gamma B'KA & F + \square B'KB \end{bmatrix} = \begin{bmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{bmatrix} (24)[9]$$

In parametric form, the Q function defined in (23) can be written as (24), where $z = [x^T\ u^T]$.

$$Q(x,u) = Q(z) = z^T S z\ (25)[10]$$

Finally, by grouping the quadratic terms of the z vector into $\phi(z)$, the expression for the Q function becomes (26), where the vector W^T contains the variables in the S matrix defined in (24) and the $\phi(z)$ vector contains the state and input variable values $[x_1^k, \dots, x_1 u_n, x_2^k, \dots, x_2 u_n, \dots, u_n^k]'$.

$$Q(x,u) = Q(z) = W^T \phi(z) (26)[10]$$

The main purpose of transforming the definition of the Q function in this form is that the unknown matrix S in (24), which is dependant on the system's dynamics, A and B, can be estimated by the adaptive controller algorithm without any prior knowledge of system's model.

1.3.2 Value function approximation

Considering that the system dynamics are not known, which implies that the S matrix (24) can't be directly computed, a method is required to estimate these parameters. This can be achieved by finding solutions for the variables in the W vector, which are equivalent to the variables in the unknown S matrix, but put into vector form, by applying *Recursive Least Squares* (RLS) [10] on equation (27), until convergence of its components.

$$W^T\big(\phi(z_k) - \phi(z_{k+1})\big) = \frac{1}{2}(x_k^T Q x_k + u_k^T R u_k) (27)[10]$$

RLS can be applied to estimate W by the use of the following set of equations, where $\phi = \phi(z_k) - \phi(z_{k+1})$ and $P_0 = \beta I$ ($\beta = arbitrary\ large\ positive\ constant$).

$$P_k(0) = P_0 (28)[10]$$

$$P_k(i) = P_k(i-1) - \frac{P_k(i-1)\phi_t \phi_t' P_k(i-1)}{1 + \phi_t' P_k(i-1)\phi_t} (29)[10]$$

$$e_k(i) = r_t - \phi_t' W_k(i-1) (28) [10]$$

$$W_k(i) = W_k(i-1) + \frac{P_k(i-1)\phi_t e_k(i)}{1 + \phi_t' P_k(i-1)\phi_t} (30) [10]$$

The W vector can be seen as a simple single layer *neural network* [11], as it takes several inputs from ϕ, multiplies them by a weight and adds the resulting products to output a single output, and more importantly, it gets trained every RLS iteration.

Following this procedure will allow us to obtain the components of the W vector that can be unpacked to obtain the S matrix, which is needed to approximate the value function across the states in the system's trajectory. Finally, the control policy can be updated by applying the derivative $\frac{dQ}{du} = 0$, which leads to $u = -S_{22}^{-1} S_{21} x$.

1.3.3 Algorithm implementation and procedure

A step-by-step implementation can be achieved by following the steps below, Figure 5.
This algorithm makes use of a method named *one-step look ahead temporal difference* learning (TD-1) [12], since it requires the data from the current state and the following state, to perform RLS on the W vector.

INITIALIZE
At $j = 0$:
Set initial policy: K^0.

STEP j
Apply current control policy: $u_k = -K^j x_{k+1}$.
Measure: $x_k, u_k, x_{k+1}, u_{k+1}$.
Compute quadratic basis: $\phi(z_k)\phi(z_{k+1})$.
Update W_j by applying RLS to equation (25).
Repeat until convergence and W_{j+1} is found.

UPDATE CONTROL POLICY
Unpack the vector W_{j+1} into matrix S (22).
Update control policy: $K = -S_{22}^{-1} S_{21}$.

SET $j = j + 1$. **GO TO STEP** j.

Figure 5: Adaptive controller algorithm using Q Learning (*Adapted from:* [2])

This iterative procedure finishes when the control policy K is no longer being updated. This algorithm is equivalent to solving the algebraic Riccati equation, without prior knowledge of the system dynamics, in an online manner, and will eventually converge to the optimal policy K.

2 Design Strategies and Testing

In this section, the adaptive controller algorithm described in section 1.3 is applied to two different dynamic systems through software simulations. An analysis of the results and an overview of the performance using different parameters are also discussed and illustrated.

2.1 Power Plant

An electric power system is a dynamic system, which requires regulation to provide a stable output power to its load. In this scenario, the optimal input that regulates the speed of the turbine, which is the generator, will be obtained to provide a stable output to the power plant.

2.1.1 System Modelling

Due to its complexity, the exact parameters of its system dynamics are usually unobtainable. This means that the matrices A and B are inaccurate. The model of the system can be described as in equations (31) and (32).

$$A = \begin{bmatrix} -\frac{1}{T_P} & \frac{K_P}{T_P} & 0 & 0 \\ 0 & -\frac{1}{\square_T} & \frac{1}{T_T} & 0 \\ -\frac{1}{RT_G} & 0 & -\frac{1}{T_G} & -\frac{1}{T_G} \\ K_E & 0 & 0 & 0 \end{bmatrix} (31)[2]$$

$$B = \begin{pmatrix} 0 \\ 0 \\ \frac{1}{T_G} \\ 0 \end{pmatrix} (32)[2]$$

$T_G = governor\ time\ constant$
$T_T = turbine\ time\ constant$
$T_P = plant\ time\ constant$
$K_P = plant\ model\ gain$
$K_E = integral\ control\ gain$

The states of the system are $x(t) = [\Delta f(t) \quad \Delta P_g(t) \quad \Delta X_g(t) \quad \Delta E(t)]^T$ and they give an intuition of how stable the system is at a particular time. Due to the fact that the exact values of the system dynamics are not known, values for the parameters used in (31) and (32) were chosen randomly within a reasonable range of possible values. The resulting continuous-time matrices A and B are shown in (33) and (34).

$$A = \begin{bmatrix} -0.0665 & 8 & 0 & 0 \\ 0 & -3.663 & 3.663 & 0 \\ -6.86 & 0 & -13.736 & -13.736 \\ 0.6 & 0 & 0 & 0 \end{bmatrix} (33)[2]$$

$$B = \begin{pmatrix} 0 \\ 0 \\ 13.7355 \\ 0 \end{pmatrix} (34)[2]$$

In order to apply the sequential algorithm from Figure 5, the system dynamics were discretized using the zero-order hold method using a sampling frequency of 100Hz.
The resulting discrete-time state-space matrices A and B are (35) and (36).

$$A = \begin{bmatrix} 0.999 & 0.078 & 0.001 & -6.424e^{-5} \\ -0.001 & 0.964 & 0.0336 & -0.002 \\ -0.065 & -0.002 & 0.872 & -0.128 \\ 0.006 & 2.370e^{-4} & 2.806e^{-6} & 1 \end{bmatrix} (35)[2]$$

$$B = \begin{pmatrix} 6.424e^{-5} \\ 0.002 \\ 0.128 \\ 9.719e^{-8} \end{pmatrix} (36)[2]$$

2.1.2 Adaptive Controller Application

To solve the system described above, values for the weight matrices Q and R, and the discount factor γ, were arbitrarily chosen to be (37), (38) and (39).

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.3 & 0 \\ 0 & 0 & 0 & 0.3 \end{bmatrix} (37)$$

$$R = [0.1] (38)$$

$$\gamma = 1 (39)$$

Taking into account all the system's aspects described above, the adaptive controller algorithm can be applied to obtain the optimal policy, K. From the first step described in Figure 5, the initial policy was set to be $K = [0.5\ 0.5\ 0.5\ 0.5]$. The system was then run for an episode of $600\ \text{discrete steps} = \frac{episode\ duration\ (6s)}{sampling\ period\ (0.01s)}$ in which the variables of the W vector were updated for every step using RLS. In the next step, the parameters stored in the W vector were unpacked to the S matrix and used to update the control policy K. This whole procedure was iterated for 300 episodes until the final policy was obtained and implemented to stabilize the dynamic system.

Figure 6a shows the resulting plot of the states response from the final obtained policy. Next to it, in figure 6b the states from using the optimal policy were drawn. From observation, it can be seen that the policy obtained from the adaptive controller algorithm drags the states to 0, and thus to stability. Moreover, it can be seen that the states' trajectories are very similar to those obtained from applying the optimal policy. The same happens with the actions u, which are directly proportional to the policy from the equation $u = -Kx$.

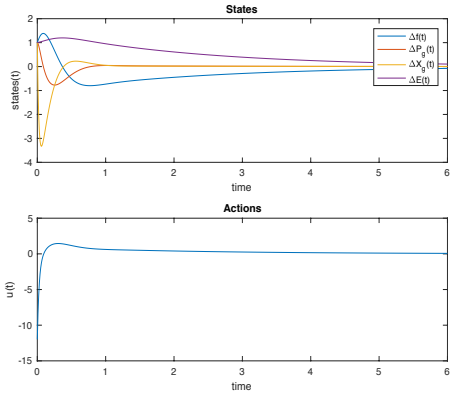


Figure 6a: States using obtained policy

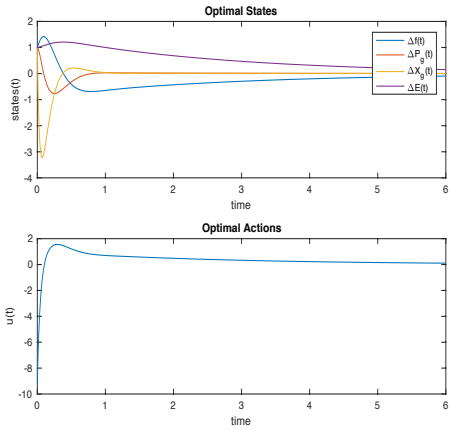


Figure 6b: States using optimal policy

For a better visualization of the performance algorithm, the mean squared error (MSE) between the optimal policy, which was obtained by solving the ARE and the final policy after 300 iterations, was computed every episode. This is illustrated in Figure 7.

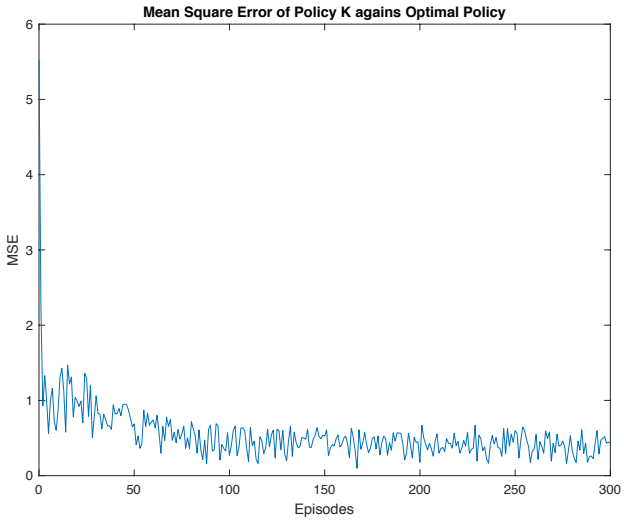


Figure 7: MSE between obtained policy and optimal policy, power plant

The Figure above shows that after a few iterations the initial MSE rapidly drops from around 5.5 to 1 and continues to steadily drop for the following 100 episodes. It then converges at around and MSE of 0.5 at the end of the episodes. This offset is caused due to the exploration factor used in the actions, in which in

randomly selected discrete-steps, the optimal action is not chosen to allow the agent to look for possible better actions.

The system was also simulated using different Q, R and initial policy K, shown below (40).

$$Q = \begin{bmatrix} 0.1 & 0 & 0 & 0 \\ 0 & 0.7 & 0 & 0 \\ 0 & 0 & 0.8 & 0 \\ 0 & 0 & 0 & 0.1 \end{bmatrix}$$

$$R = [0.3]$$

$$K_{init} = [0.1\ 0.7\ 0.2\ 0.4] \quad (40)$$

The resulting states trajectories and actions are shown in Figure 7a and 7b.

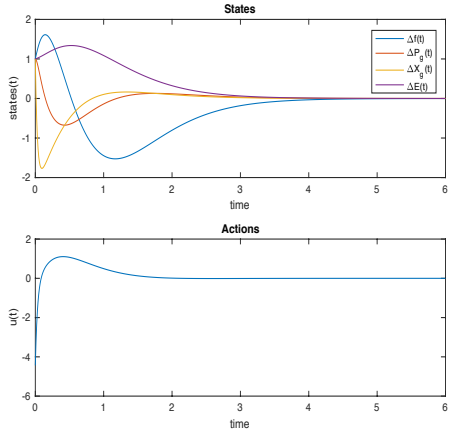


Figure 7a: States using obtained policy

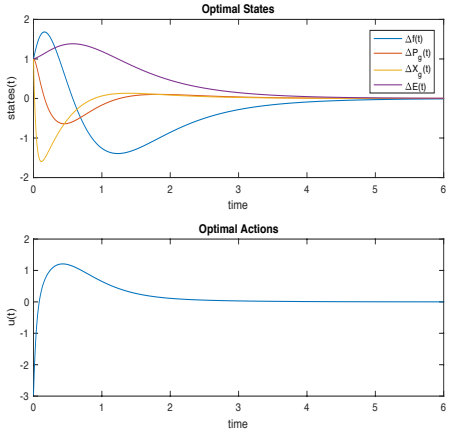


Figure 7b: States. optimal policy

Again, the states obtained from the adaptive algorithm show convergence and behave very similar to the optimal solution.

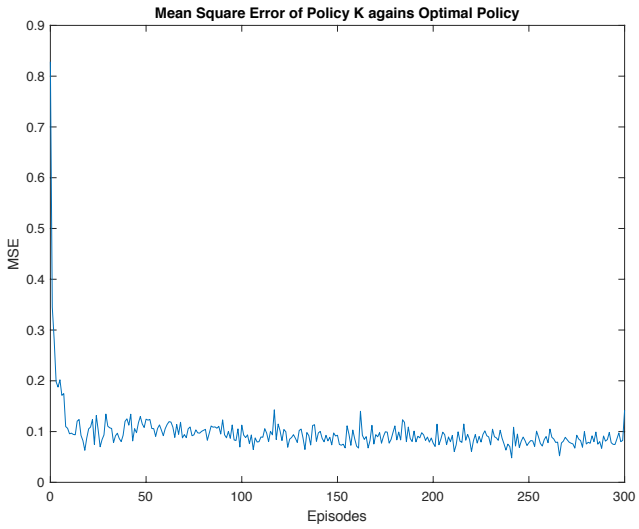
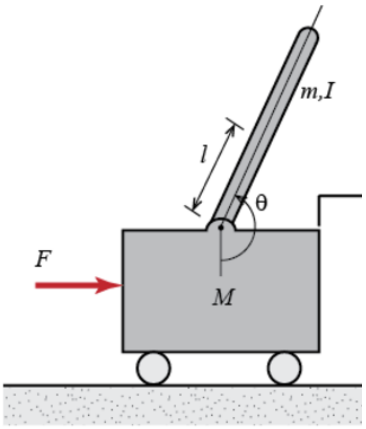


Figure 7: MSE between obtained policy and optimal policy, power plant

With this new configuration, we can also observe that the policy MSE against episodes converges very quickly to around 0.1, which is much closer than the previous simulation.

2.2 Inverted Pendulum

The inverted pendulum problem consists of stabilizing a pole that is fixed to a cart that can only move sideways. A visual representation of the setup can be seen in Figure 8.



(M)	mass of the cart	0.5 kg
(m)	mass of the pendulum	0.2 kg
(b)	coefficient of friction for cart	0.1 N/m/sec
(l)	length to pendulum center of mass	0.3 m
(I)	mass moment of inertia of the pendulum	0.006 kg.m^2
(F)	force applied to the cart	
(x)	cart position coordinate	

Figure 8: Inverted pendulum model [13]

2.2.1 System Modelling

To obtain the system dynamics in state-space form, a force analysis was performed by extracting the equations of motion for the cart and the pendulum from the free-body diagram in Figure 9.

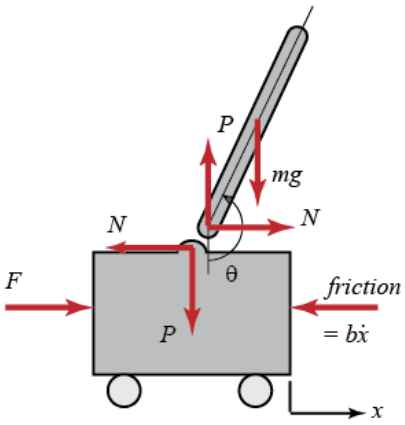


Figure 9: Free-body diagram [13]

In the horizontal directions, the equation of motion of the cart can be written as (41). The vertical forces of the cart are not relevant for this problem, so they were not analysed.

$$M\ddot{x} + b\dot{x} + N = F (41)[13]$$

Regarding the pendulum, the reactionary force N, in the horizontal direction, can be expressed as (42).

$$N = m\ddot{x} + ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2\sin\theta\ (42)[13]$$

By substituting (39) in (38), one of the core equations can be obtained, (43).

$$(M + m)\ddot{x} + b\dot{x} + ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2\sin\theta = F (43)[13]$$

For the vertical axis, the equation of motion of the pendulum is shown in (44).

$$P\sin\theta + N\cos\theta - mg\sin\theta = ml\ddot{\theta} + m\ddot{x}\cos\theta\ (44)[13]$$

To eliminate the terms P and N, the equation for the moments acting on the centroid of the pendulum can be used (45).

$$-Pl\sin\theta - Nl\cos\theta = I\ddot{\theta} (45)[13]$$

These two equations, (44) and (45), can be used to obtain the second core equation of the system (46).

$$(I + ml^2)\ddot{\theta} + mgl\sin\theta = -ml\ddot{x}\cos\theta\ (46)[13]$$

The next step is to linearize the system to perform the required analysis. The point of reference for the linearization will be in the point of equilibrium, at $\theta = \pi$, as our objective will be to keep the pendulum as close at that point as possible, $\theta = \pi + \phi$, with only a small deviation of ϕ. By substituting the terms with the

linearized parameters and by replacing the force F with our input u, we obtain the final equations of motion that will describe our system, (47) and (48).

$$(I + ml^2)\ddot{\phi} - mgl\phi = ml\ddot{x}(47)[13]$$

$$(M + m)\ddot{x} + b\dot{x} - ml\ddot{\phi} = u(48)[13]$$

With the equations obtained from the force analysis, the Laplace transform was used to obtain the transfer function of the system. The resulting equations in the frequency domain after taking the transform can be seen in (49) and (50).

$$(I + ml^2)\Phi(s)s^2 - mgl\Phi(s) = mlX(s)s^2(49)[13]$$

$$(M + m)X(s)s^2 + bX(s)s - ml\Phi(s)s^2 = U(s)(50)[13]$$

In order to obtain an expression for the transform function, we need to obtain an equation with the only the input $U(s)$ and output $\Phi(s)$ as the unknowns. Therefore, we remove the term $X(s)$ by replacing in term of $\Phi(s)$, as in (51), which was derived from (49).

$$X(s) = \left[\frac{I + ml^2}{ml} - \frac{g}{s^2}\right]\Phi(s)(51)[13]$$

Substituting (51) back to (50) we obtain (52).

$$(M + m)[\tfrac{I+ml^2}{ml} - \tfrac{g}{s^2}]\Phi(s)s^2 + b[\tfrac{I+ml^2}{ml} - \tfrac{g}{s^2}]\Phi(s)s - ml\Phi(s)s^2 = U(s)(52)[13]$$

Finally, by rearranging (52) the main transfer functions can be obtained. The transfer function for the angle of the pendulum is shown (53) and the transfer function for the cart position in (51).

$$T_{pend}(s) = \frac{\Phi(s)}{U(s)} = \frac{\frac{ml}{q}s}{s^3 + \frac{b(I + ml^2)}{q}s^2 - \frac{(M + m)mgl}{q}s - \frac{bmgl}{q}}\left[\frac{rad}{N}\right](53)[13]$$

$$T_{cart}(s) = \frac{X(s)}{U(s)} = \frac{\frac{(I + ml^2)s^2 - gml}{q}}{s^4 + \frac{b(I + ml^2)}{q}s^3 - \frac{(M + m)mgl}{q}s^2 - \frac{bmgl}{q}s}\left[\frac{m}{N}\right](54)[13]$$

$$where\ q = [(M + m)(I + ml^2) - (ml)^2]$$

The obtained transfer functions can be represented in the matrix of the state-space representation, (55) and (56).

$$\begin{bmatrix}\dot{x}\\ \ddot{x}\\ \dot{\phi}\\ \ddot{\phi}\end{bmatrix} = \begin{bmatrix}0 & 1 & 0 & 0\\ 0 & \frac{-(I+ml^2)b}{I(M+m)+Mml^2} & \frac{m^2gl^2}{I(M+m)+Mml^2} & 0\\ 0 & 0 & 0 & 1\\ 0 & \frac{mlb}{I(M+m)+Mml^2} & \frac{mgl(M+m)}{I(M+m)+Mml^2} & 0\end{bmatrix}\begin{bmatrix}x\\ \dot{x}\\ \phi\\ \dot{\phi}\end{bmatrix} + \begin{bmatrix}0\\ \frac{I+ml^2}{I(M+m)+Mml^2}\\ 0\\ \frac{ml}{I(M+m)+Mml^2}\end{bmatrix} u(55)[13]$$

$$y = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ \dot{x}\\ \phi\\ \dot{\phi}\end{bmatrix} + \begin{bmatrix}0\\ 0\end{bmatrix} u(56)[13]$$

With respect to the constants M, m, b, l, I, F and X, used in the system's model, the values used are shown in Figure 8. The system was then discretized using the zero-order hold method.

2.2.2 Adaptive controller application

Using the matrix form representation derived above and values of Q, R and discounting factor γ from equation (57) the adaptive controller algorithm was tested in the system.

$$Q = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0.01 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 0.01\end{bmatrix}$$

$$R = [0.1]$$

$$\gamma = 1 \ (57)$$

The simulation was run for 100 episodes in discrete intervals of 0.01s over 6s, initialized with a policy $K = [3, 3, 3, -1]$. The resulting states' response from using the policy obtained at the end of the sequence was plotted in Figure 10a. By observation, it can be seen that the states converge very quickly and much faster than the power plant, at around 0.5s after the initialization of the simulation. Moreover, compared to the states using the optimal policy, the states trajectories look almost identical, as well as the actions.

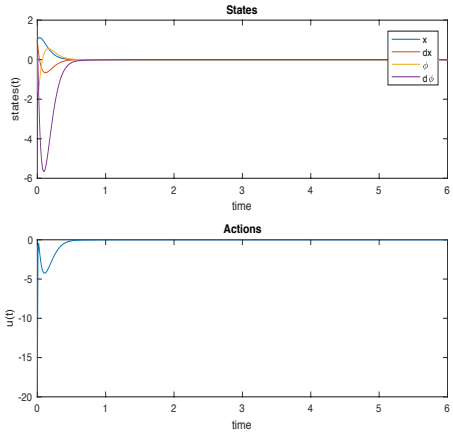


Figure 10a: States using obtained policy

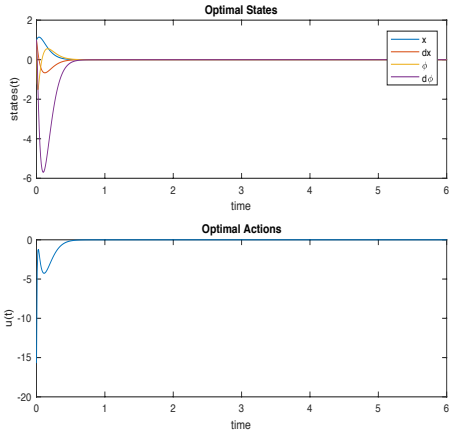


Figure 10b: States, optimal policy

With respect to the MSE between the policy obtained every episodes and the optimal policy, the fluctuations between each episode are very abrupt, as it can be seen in Figure 11.

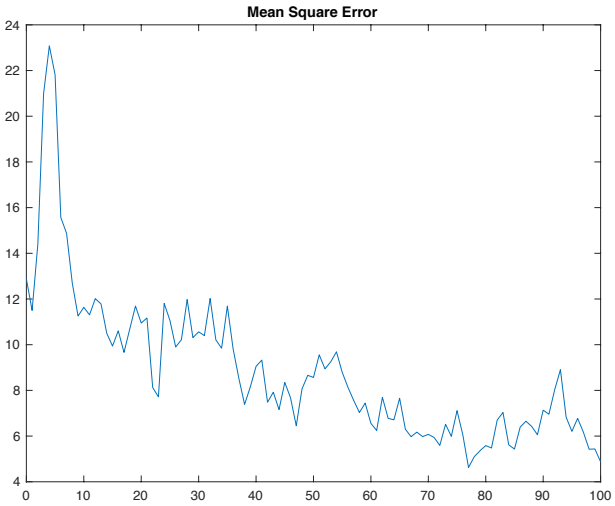


Figure 11: MSE between obtained policy and optimal policy, inverted pendulum

After a 100 episodes, the MSE decays to around 5, which proved to be enough to provide fast stability to the dynamic system.

Another simulation of the system was run with different values of the matrices Q, R and initial policy K_{init}, as in (58).

$$Q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.01 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.01 \end{bmatrix}$$

$$R = [0.01]$$

$$K_{init} = [2\ 1\ 2\ -0.5] (58)$$

The results can be seen below in Figures 10 and 11. Again, the graphs demonstrate that the adaptive algorithm finds a final policy that drags the states to stability.

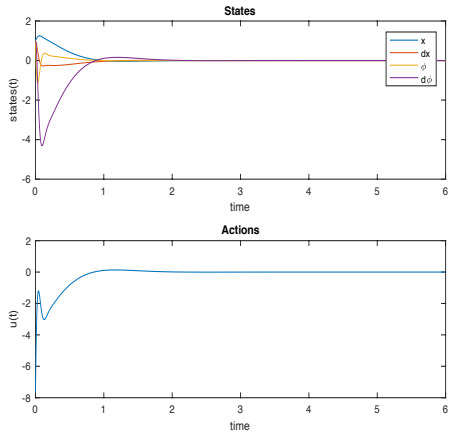


Figure 11a: States using obtained policy

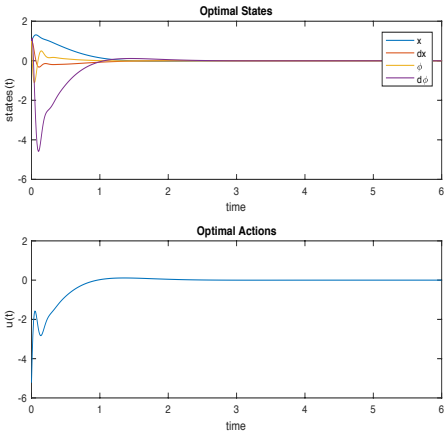


Figure 11b: States using optimal policy

Moreover, although the drop is not as sharp as the previous simulation, the MSE trend through the episodes show that the policy slowly converges to the optimal policy.

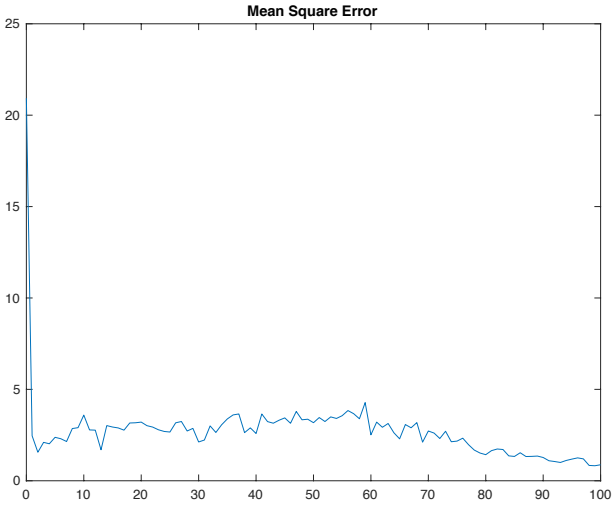


Figure 12: MSE between obtained policy and optimal policy, inverted pendulum

Overall, these simulations show that the initial parameters have a huge impact in the performance and response of the adaptive algorithm.

2.3 Alternative Approach Using Pure Reinforcement Learning (Q-Learning)

An alternative method of solving these problems is by exclusively using reinforcement learning methods, specifically Q-Learning, as suggested by the online article written by *Matthew Chan* [14]. This way to tackle the problem takes

longer to stabilize the inverted pendulum but it eventually achieves the goal. The platform used to test this method was *Open-AI* with the python programming language.

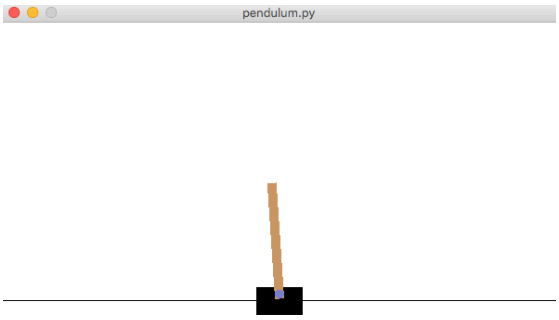


Figure 13: Open-AI platform animation

2.3.1 Hidden Markov Model

The first step for solving this problem is to define its states, which are the position of the cart in the horizontal axis, its velocity, the angle of the pendulum and its angular velocity. This system is considered to be a Markov model because the next state is only dependent on the current state. This means that the whole information of the past is contained in the current state. With respect to the actions space, the agent has only two options, to move the cart to the right or to the left. A simple visualization of these characteristics was summarised in Figure 13.

States	Actions
x	$\rightarrow$
$\dot{x}$	$\leftarrow$
θ	
$\dot{\theta}$	

Figure 14: Inverted Pendulum states and actions

2.3.2 States discretization

In order to make the simulation process manageable, the system's states were discretized into equally sized sections. This can be seen in Figure 14.

States	Units	Lower Bound	Upper Bound	Number of Sections	Section Length
x	m	-4.8	+4.8	3	3.2
$\dot{x}$	m/s	-0.5	+0.5	3	0.33
θ	rad	-0.42	+0.42	6	0.14
$\dot{\theta}$	rad/s	-0.87	+0.87	3	0.58

Figure 15: Discrete Sections

Although it might be counter intuitive, increasing the number of sections per state, did not necessarily improved its performance. This is because as the number of

sections increases, the more learning is required to find an accurate Q-Function for each of these sections.

2.3.3 Q-Learning

The objective of this simulation was to keep the pendulum within the maximum bounds for more than 195 steps and for longer than 100 consecutive episodes. In order to apply Q-Learning, a table containing the Q-Values for each of the states of the system, with a learning rate of $\gamma = 0.01$, shown previously in equation (8), was constructed and updated online after every step. After several simulations, this algorithm was able to solve the problem repeatedly in around 140 episodes. The results along each episode (x-axis) followed the shape and trend that can be seen in the following Figure 16.

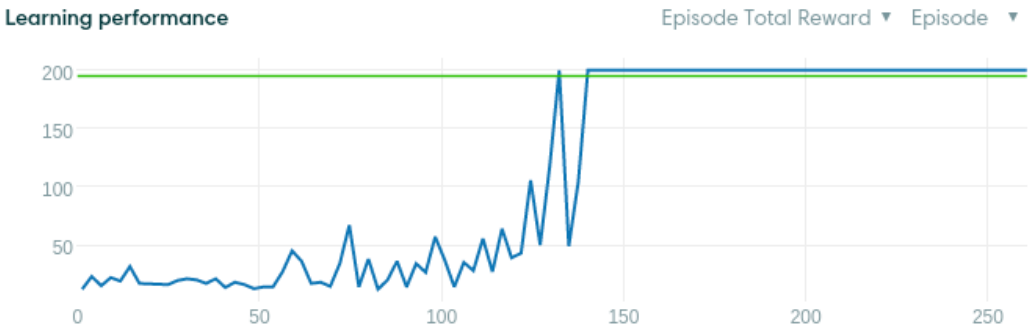


Figure 16: Trend from using Q-Learning with the current setup (Sourced from [14])

The y-axis of the graph shows how many consecutive steps the pendulum remained within the bounds. As it can be seen at around 140 episodes, the agent is able to continuously succeed.

3 Progress, Results and Further work

Taking into account all the work done so far, the core objectives of the initial project plan was completed. With respect to the results, the simulations shown in the preceding section show that reinforcement learning can indeed be used to solve LQR problems and find the optimal policy for different systems with enough training. Moreover, it shows that by tuning different parameters, such as, Q, R and K_{init}, slightly different performances can be obtained. However, an issue that was encountered during the simulations was that some times the computed policy values would overshoot and take far too many episodes (> 500) to show signs of convergence. This is an issue that could be addressed in the future to find a mathematical explanation.

4 Conclusion

All in all, this project was a good demonstration that the application of reinforcement learning is not only limited to a small set of problems. In this project, a simple algorithm using Q-Learning was used to solve a very complex optimal control theory problem without exact knowledge of the exact system dynamics.

Lastly, this project was a very valuable learning experience in one of the fields of personal greatest interest, which is artificial intelligence and has given me even more interest for

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.

[5] Wikipedia.com, (2017, March 30), *Linear-quadratic regulator* (1st ed.) [Online]. Available: https://en.wikipedia.org/wiki/Linear%E2%80%93quadratic_regulator.

[6] Harvard University, *Hamiltonian Equations of Motion (Chapter 8)* [Online], Available: http://users.physics.harvard.edu/~morii/phys151/lectures/Lecture18.pdf.

[7] David Silver, *Lecture 6: Value Function Approximation* [Online], Available: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/FA.pdf.

[8] David Poole and Alan Mackworth, (2010), *Artificial Intelligence: Foundations of computational agents* [Online], Available: http://artint.info/html/ArtInt_265.html.

[9] S. J. Bradtke, "Reinforcement Learning Applied to Linear Quadratic Regulation", pp. 297–300.

[10] S. J. Bradtke, B. E. Ydstie, A. G. Barto, "Adaptive linear quadratic control using policy iteration", pp. 3475-3479, June 1994.

[11] C. Stergiou, D. Siganos, *Neural Networks* [Online], Available: https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html.

[12] Stanford University, *Chapter 9: Temporal-Difference Learning* [Online], Available: https://web.stanford.edu/group/pdplab/pdphandbook/handbookch10.html.

[13] Michigan University, *Inverted pendulum: System Modelling* [Online], Available: http://ctms.engin.umich.edu/CTMS/index.php?example=InvertedPendulum&section=SystemModeling.

[14] Mattew Chan, *Cart-Pole Balancing with Q-Learning* [Online], Available: https://medium.com/@tuzzer/cart-pole-balancing-with-q-learning-b54c6068d947

Appendix A. Hardware Costs

There weren't any costs for this project as everything was done through free available software.

Appendix B. 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 C. MATLAB Code for e-greedy Function

```
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 D. Archive Table of Contents

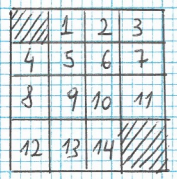
1. Power Plant (power.m)
2. Inverted Pendulum (pendulum.m)
3. Pendulum Pure Q-Learning (pendulum.py) [14]

Appendix E. Grid World Problem Solution

Dynamic Programming

Topics → Policy Evaluation, Policy Iteration, Policy Improvement

Gridworld Example



Possible actions ↔ ↕.

Reward for each step = −1. $r(s,a,s')$

Reward for reaching ▨ = 0.

1) Policy Evaluation: With arbitrary policy, π, we calculate the estimated state-value function for each state using Bellman's equation.

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(s') \mid S_t = s]$$

In addition to this, we have to perform a back-up from the previous step, so the equation becomes.

$$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(s') \mid S_t = s]$$

For this example, we assume that $v_0(s) = 0$ for $s \in S$ and $\pi(a|s) = 0.25$ for $a \in A$ and $\gamma = 1$.

Starting from State 1:

$$v_1(1) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)\,[r(s,a,s') + \gamma v_k(s')]$$

UP = LEFT = $0.25 \times 1 \times [0 + 0] = 0$

TOP = DOWN = RIGHT = $0.25 \times 1 \times [-1 + 0] = -0.25$

$\Sigma = -1$

From examining the grid we can clearly see that for every state excluding the shaded boxes the $v_1(s)$ is -1.0.

Therefore, at $t=1$:

0		-1	-1
-1	-1	-1	-1
-1	-1	-1	
-1	-1		0

These $v_\pi(s)$ will differ from the rest.

For the states , they will all have the same v_π in $t=2$.

2) Policy iteration

We can update our v_π by 'running' episodes using our initial policy, π.

At $t=2$, for $S=1,4,11,14$

$$v_2(1,4,11,14) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma v_1(s')\right]$$

DOWN = TOP = RIGHT $= 0.25 \times 1 \times [-1 - 1] = -0.5$

LEFT $= 0.25 \times 1 \times [-1 + 0] = -0.25$

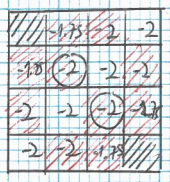
$$\boxed{\Sigma = -1.75}$$

For the rest of the states

$$v_2(s) = \updownarrow = 0.25 \times 1 \times [-1 - 1] = -0.5$$

$$\boxed{\Sigma = -2}$$

Therefore, the new grid is:



At first inspection, we can see that all of these value functions will be affected by different v_3 surrounding them.

$t=2$

For the states with $v_3 = -1.75$.

At $t=3$,

$$v_3(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v_2(s')]$$

LEFT = UP = $0.25 \times 1 \times [-1 - 2] = -0.75$

RIGHT = $0.25 \times 1 \times [-1 - 1.75] = -0.69$

DOWN = $0.25 \times [-1 - 0] = -0.25$

$\Sigma = -2.44$

For the states containing -2 and shaded in red and circled

$v_3(s)$ = LEFT = UP = $0.25 \times 1 \times [-1 - 2] = -0.75$

DOWN = RIGHT = $0.25 \times 1 \times [-1 - 1.75] = -0.69$

$\Sigma = -2.88$

For the states 2, 6, 7, 11,

$v_3(s)$ = LEFT = RIGHT = DOWN = $0.25 \times 1 \times [-1 - 2] = -0.75$

UP = $0.25 \times 1 \times [-1 - 1.75] = -0.69$

$\Sigma = -2.94$

Lastly for the states in the centre of the grid.

$v_3(s) =$ ↕, $0.25 \times 1 \times (-1-2) = -0.75$

$\Sigma = -3$

Optimal Policy for each state.

Therefore:

	-2.94	-2.94	-3
-2.94	-2.88	-3	-2.94
-2.94	-3	-2.88	-2.94
-3	-2.94	-2.94	

$k=3$

π_* →

	←	←	↰
↑	↰	↰	↓
↑	↳	↳	↓
↳	→	→	

Result

At $k=4$; for states 1, 4, 11, 14

$v_4(s) =$ LEFT = ~~UP~~ $0.25 \times (-1-0) = -0.25$

TOP = $0.25 \times (-1-2.44) = -0.86$

RIGHT = $0.25 \times (-1-2.94) = -0.99$

DOWN = $0.25 \times (-1-2.88) = -0.97$

$\Sigma = -3.07$

For states 5 and 10

$v_4(s) =$ UP = LEFT = $0.25 \times (-1-2.44) = -0.86$ $\Sigma = -3.72$

RIGHT = DOWN = $0.25 \times (-1-3) = -1$

For states 2 and 8, ~~7~~, 13

UP = $0.25 \times (-1-2.94) = -0.99$, RIGHT DOWN = $0.25 \times (-1-3) = -1$

LEFT = $0.25 \times (-1-2.44) = -0.86$ $\Sigma = -3.85$

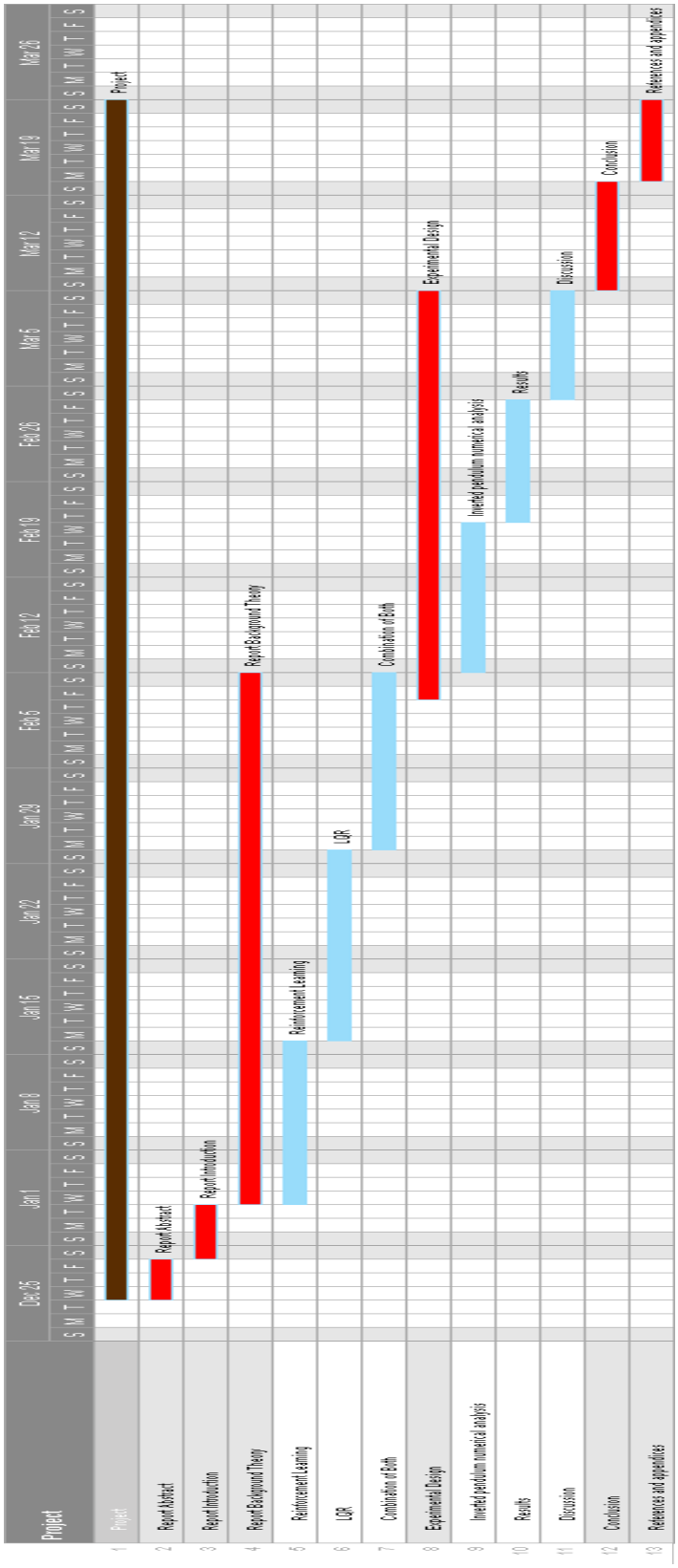
For 0, $[0.25 \times (-4)] \times 2$

$[0.25 \times (-3.94)] \times 2$ $\Sigma = -4.21$

For ▨, $[0.25 \times -3.88] \times 2$ $\Sigma = -3.91$

$[0.25 \times -2.94] \times 2$

Appendix F. Gantt Chart


Project
Dec 25
Jan 1
Jan 8
Jan 15
Jan 22
Jan 29
Feb 5
Feb 12
Feb 19
Feb 26
Mar 5
Mar 12
Mar 19
Mar 26
1 Project
2 Report Abstract
3 Report Introduction
4 Report Background Theory
5 Reinforcement Learning
6 LQR
7 Combination of Both
8 Experimental Design
9 Inverted pendulum numerical analysis
10 Results
11 Discussion
12 Conclusion
13 References and appendixes