University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialization: MSc. Computational Engineering	Spring semester, 2022 Open
Author: Muhammad Usama	.... Muhammad Usama
Programme coordinator: Karina Sanni Supervisors: UiS - Prof. Dan Sui	
Title of master's thesis: APPLICATION OF REINFORCEMENT LEARNING IN MANAGED PRESSURE DRILLING	
Credits: 30	
Keywords: Reinforcement Learning, Managed Pressure Drilling, Proportional Integral Differential Controller, Drilling.	Number of pages: 80 + supplemental material/other: 25 Stavanger, 15th July 2022

Muhammad Usama

APPLICATION OF REINFORCEMENT LEARNING IN MANAGED PRESSURE DRILLING

Master Thesis Project for the degree of
MSc in Computational Engineering

Stavanger, July 2022

University of Stavanger
Faculty of Science and Technology
Department of Energy Resources

Universitetet
i Stavanger

Automation in any industry has a control system as its base, and control systems are composed of a controllers.In recent years an area of machine learning known as reinforcement learning (RL) has been focused on solving control problems for engineers and scientists. RL methods are actively applied to design control mechanisms for various industrial applications and in this study the focus will be on designing such algorithms and modelling the given control problem into a structure where these RL algorithms can be applied.

In oil and gas industry there has been a push to expand operations into areas where usual drilling methods are not successful mainly because of narrow operational windows and technologies such as Managed Pressure Drilling (MPD) are found to be very successful in solving this issue. MPD is a control technique which is aimed at controlling the bottom hole pressure between narrow operational windows.

The standard technique used for automating MPD is proportional-integral-derivative (PID) controller, but many other non-linear control systems have also been employed to do the same task. This study seeks to add value to the drilling process by developing an Reinforcement Learning (RL) based agent to tune the PID controller. After tuning the PID controller, the system dynamics will be optimized and kept under boundary conditions of the drilling environment. The goal is to provide a reference bottom hole pressure set point and tuning parameters to the PID controller so that the optimum pressure can be reached safely at a certain depth.

During the study, the most important features were depth of the drilling bit, the fracture pressure and pore pressure at that depth. The RL agent first proposes a suitable reference bottom hole pressure based on the fracture and pore pressure and then tune the PID controller to achieve the desired pressure set point. The task of training this RL agent is handled in a specialized simulator environment which can calculate the bottom hole pressure at every simulation step and give feedback to the agent about the status.

The agent uses a policy gradient method called Proximal Policy Optimization (PPO) and then later on Multi-armed bandit algorithms. PPO is implemented using Mathwork's Reinforcement Learning Toolbox, and after some tuning of hyper parameters the agent is able to narrow down to a optimal policy for various depth scenarios, whereas the latter is developed in python. At the end of this study, the agent is able to replace the decision maker and automatically suggest reference bottom hole pressure and tune the PID accordingly.

Acknowledgments

First, I would like to thank God, for providing me the strength needed to complete my studies satisfactorily.

Secondly, I would like to thank my supervisor Prof. Dan Sui for her guidance and smart ideas. I would also like to thank Jie Cao for all the productive discussion and ideas which have contributed to the successful completion of this study.

I would like to thank every one at University of Stavanger who have supported me and helped me through out my studies and to Jan Einar Gravdal for giving me the support and guidance to work in this exciting collaborative study.

Finally, my deepest gratitude to my family and friends for their unconditional support, without you all, this would not have been possible.

Muhammad Usama

Stavanger, July 15th, 2022.

List of Abbreviations

A	Set of Actions
A2C	Advantage Actor Critic
A3C	Asynchronous Advantage Actor Critic
AI	Artificial Intelligence
API	Application Programming Interface
BHP	Bottom hole pressure
DDPG	Deep Deterministic Policy Gradient
DRL	Deep Reinforcement Learning
DQN	Deep Q Network
G	Expected Return
HPHT	High pressure High Temperature
IADC	International Association of Drilling Contractors
MAB	Multi-armed bandit
MDP	Markov Decision Process
MPD	Managed pressure drilling
NPT	Non productive time
P	Proportional
PI	Proportional Integral
PID	Proportional Integral Differential
PPO	Proximal Policy Optimization
R	Reward
RL	Reinforcement Learning
ROP	Rate of Penetration
RSS	Rotary Steering System
S	State vector
SSE	Steady state error
TD3	Twin-delayed deep deterministic policy gradient
WOB	Weight on Bit

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background, Motivation and Challenge

According to the prevailing global policies, oil demand is projected to peak near mid-2030s and then gradually decline as the energy industry shifts towards renewable energy [18]. This projected peak in the oil demand and the depletion of current reservoirs has derived the oil and gas industry to look into more exploration and production projects in areas which are regarded as deep-water, High-Pressure High-Temperature (HPHT) zones, especially in the arctic regions. In HTPT zones, the decision-makers face several technical challenges while drilling, a major challenge is narrow operational window, where the difference between pore and fracture pressure is very low [19]. Narrow operational windows are regarded as a serious drilling problem, since a minor change in the down hole pressure (or bottom hole pressure) can potentially lead to a situation like gas kick, fluid loss or blowout which can increase the Non-Productive Time (NPT).[20]

Managed pressure drilling (MPD) is used in those unconventional drilling prospects where the difference between the pore pressure and fracture pressure is so low that even the fluctuations in bottom hole pressure (BHP) caused by mud pump operation threatens the well-bore integrity [21]. MPD is an adaptive technique employed in the world of drilling with the purpose of monitoring and controlling the annular pressure profile in the wellbore. The main objective is to manage the annular hydraulic pressure according to the down-hole pressure environment limits [20].

In this study, the aim is to enhance the control process employed in the process of MPD by using advanced artificial intelligence techniques and most importantly model this drilling problem in a way where AI algorithms can be readily deployed. The motivation behind this work is to replace the manual process of ascertaining the down-hole pressure environment limits and develop a robust and smooth pressure control mechanism for the drilling process.

1.2 Objectives and Scope

The aim or purpose of this study is to explore a hybrid control system approach involving both PID controller and Reinforcement Learning (RL) agent, which can potentially add value to the process of MPD. To estimate or quantify the potential value generated, the proposed study aims to compare the performance of the proposed hybrid control scheme, with the existing controllers implemented by researchers in previous research works. In order to develop such a hybrid control system following objectives are proposed.

- Understand all the parameters involved in managed pressure drilling control.
- Understand the concepts of reinforcement learning and model the control problem so that Reinforcement Learning algorithms can be applied to it.
- Developing a stable simulation environment based on OpenLab Drilling Simulator for managed pressure drilling.
- Defining clear inputs and outputs for the simulation environment, to tune the controller and assess its performance.
- Designing a reward function, aimed at training the reinforcement learning agent to keep the pressure in the drilling window.
- Training and testing various reinforcement learning agents, and choosing the best technique in terms of accuracy and computational time .

The first objective is of utter relevance, understand how to managed pressure drilling works and which parameters are most important for the controller. This aids in designing a robust simulation environment where we can easily define inputs and outputs. Once a stable simulation environment is developed, it can be used by various reinforcement learning agents for learning purposes and the most optimum technique can be chosen.

1.3 Methodology

In this thesis a novel technique for tuning the PID controller with the use of reinforcement learning (RL) is explored, the main goal is to automate the decision-making process for the selection of reference bottom hole pressure and tuning parameters for PID controller.

The RL agent is used to provide a set-point BHP and tuning parameters for the PID controller, this set of values is called an action. The training environment for reinforcement learning is based on OpenLab drilling simulator and configurations such as mud density, type of rig, bit diameter and total depth are set inside the simulator. During each training episode, flow rate is varied in a specific range randomly and response of bottom hole pressure is measured to provide feedback for the selection of tuning parameters for the PID controller. Reward function is designed to keep the bottom hole pressure closer to pore pressure to maximize the rate of penetration (ROP) and minimize the oscillations and overshoot.

In this study, the control problem is modelled as a Markov Decision Process as well as a multi-armed bandit problem. In Markov Decision Process, the problem is handled as a sequential decision making process, where as in multi-armed bandit problem the model is reduced to one state markov decision process.

In the development phase of the simulation, MATLAB is used to develop the environment, design the reward function, and implement some RL agents. In case of multi-armed bandit problem, python is used for the implementation of epsilon-greedy algorithm.

Chapter 2

Literature Review

2.1 Managed Pressure Drilling

To completely understand this study, it is very important to understand the process of Managed Pressure Drilling (MPD) and draw a brief comparison with other drilling techniques. MPD is a technique employed in drilling, aimed to manage the annulus pressure throughout the wellbore. Bottom hole pressure (BHP) can be controlled in a more robust and safer manner by using MPD and it allows the engineers to perform drilling operations in a critical zone where the difference between the pore and fracture pressure is quite small i.e., small operation pressure window. The conventional drilling methods do not allow drillers and engineers to perform drilling operations in such conditions as these methods use mud weight to control the well pressure.

The Underbalanced Operations and Managed Pressure Drilling Committee of the International Association of Drilling Contractors (IADC) have defined Managed Pressure Drilling as (Malloy et al., 2009) [1]:

"Managed Pressure Drilling is an adaptive drilling process used to precisely control the annular pressure profile throughout the wellbore. The objectives are to ascertain the down hole pressure environment limits and to manage the annular hydraulic pressure profile accordingly. The intention of MPD is to avoid continuous influx of formation fluids to the surface. Any influx incidental to the operation will be safely contained using an appropriate process."

Furthermore some salient features of MPD operations are discussed below to further elaborate its usage, utility and importance in drilling world.

• "MPD employs a collection of tools and techniques which may reduce the risks and costs linked with well operations that have limitations with respect to the down hole environment, by controlling the annular hydraulic profile" (Malloy et al., 2009) [1].

• "MPD deals with the control of back pressure, fluid density, fluid rheology, annular fluid level, circulating friction and hole geometry" (Malloy et al., 2009) [1].

• "MPD allows quick corrective actions against pressure variations. The ability to dynamically control annular pressures facilitates drilling of what might otherwise be economically unattainable prospects" (Malloy et al., 2009) [1] .

2.1.1 Conventional Drilling vs Under-balanced Drilling vs Managed Pressure Drilling

In this section, the comparison of managed pressure drilling, with conventional drilling and under balanced drilling will be discussed. The operational range of these techniques, under comparison can be seen in the Figure 2.1 below.

It can be observed in Figure 2.1, that the under-balanced drilling operates below the pore pressure and in conventional drilling bottom hole pressure is always higher than the pore pressure. The MPD region lies above but closer to the pore pressure to maximize rate of penetration while drilling in the safe pressure window. Further details of these techniques will be discussed below.

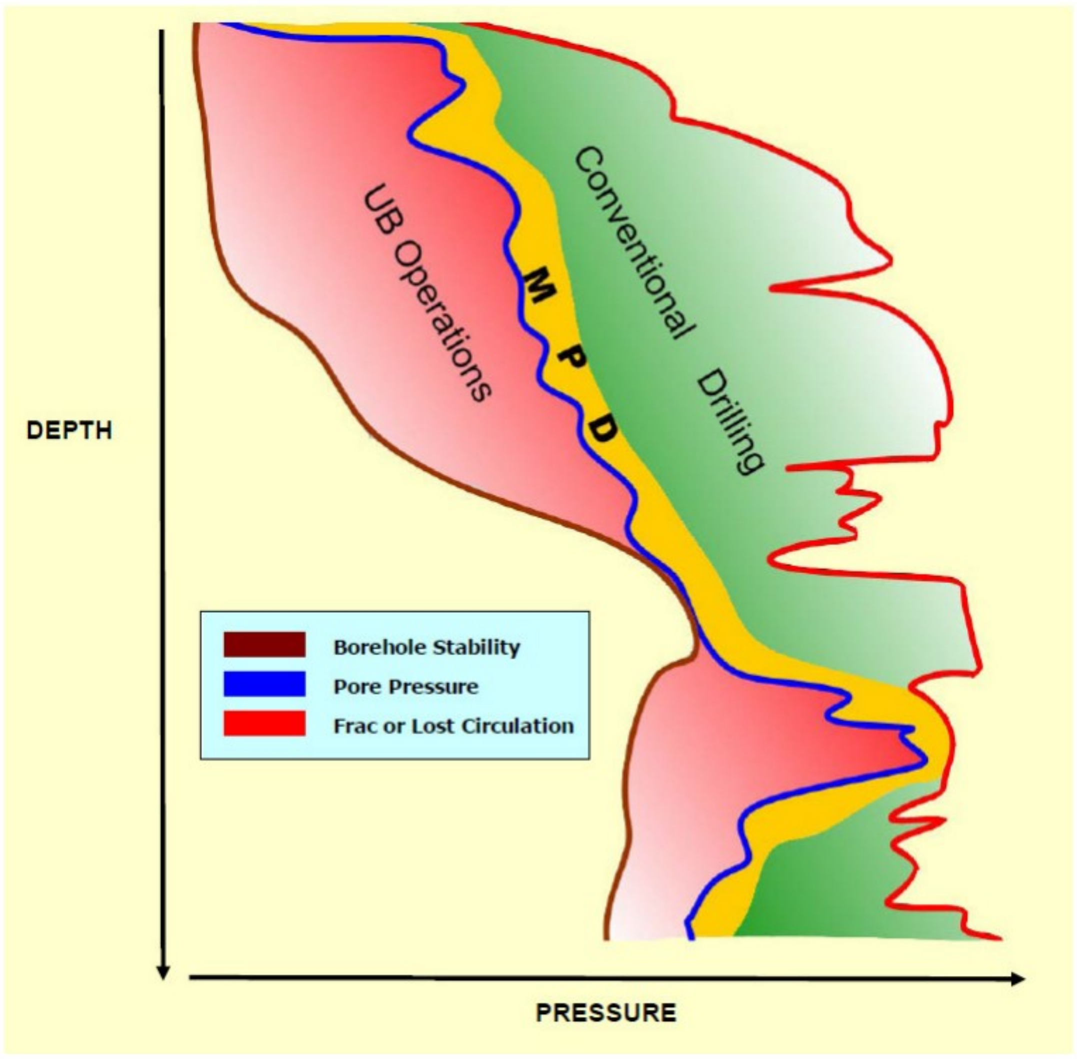


Figure 2.1: Example of Drilling Windows for various drilling techniques, taken from Malloy et al., 2009. [1]

Conventional Drilling

In conventional drilling, an over balanced pressure is maintained in the well throughout the drilling process. This means that the pressure applied in the well-bore is above the pore pressure at every point of the exposed formation. Mud density and mud flow rates are the parameters used to control the annular pressure in conventional drilling (Malloy et al., 2009) [1].

In static condition, bottom hole pressure is a function of hydro-static columns pressure whereas in dynamic condition another term annular friction pressure is also introduced. The static condition for conventional drilling is discussed in equation 2.1 below, where P_{Hyd} is the hydro-static pressure and P_{BH} is the bottom hole pressure. (Malloy et al., 2009) [1]

$$P_{Hyd} \geq P_{BH} \tag{2.1}$$

In dynamic condition, when the mud pump circulation is present the relation for bottom hole pressure is given by equation 2.2, where P_{AF} is the annular friction pressure.

$$P_{BH} = P_{Hyd} + P_{AF} \tag{2.2}$$

Figure 2.2 discusses the static and dynamic pressure conditions from equations 2.1 and 2.2, in more detail and it can be observed that as the true vertical depth (TVD) increases the effect of P_{AF} also increases and the static and dynamic pressure profiles diverge further.

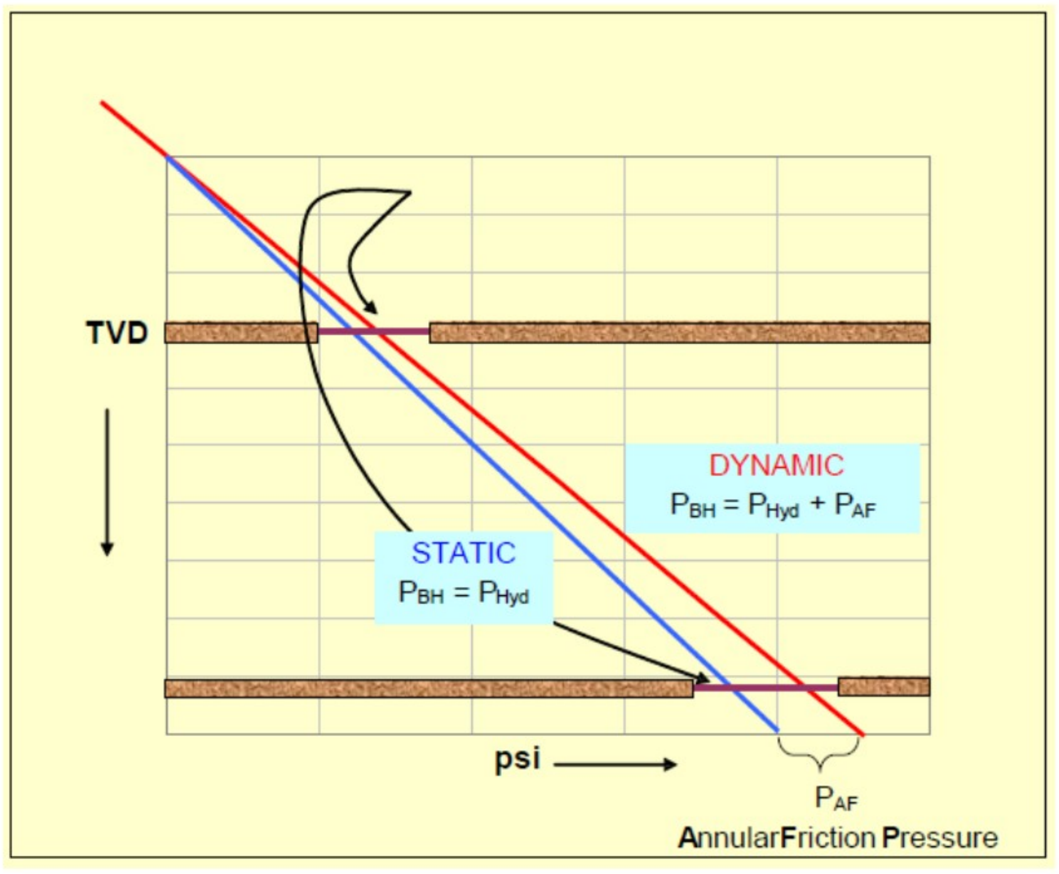


Figure 2.2: Static and Dynamic Pressure, taken from Malloy et al., 2009. [1]

In conventional drilling operations, there are higher chances of potentially dangerous situations, such as kick, stuck pipe, lost circulation etc., which can harm human life and environment.

Under-balanced Drilling

Under balanced drilling is a term used for such a drilling activity where the pressure applied by the drilling fluid in the well-bore is deliberately less than the pore pressure at any point of the exposed formations. The aim is to bring the formation fluids to the surface, where the P_{Hyd} is less than the P_{BH} as shown in equation 2.3. (Malloy et al., 2009) [1]

$$P_{Hyd} < P_{BH} \tag{2.3}$$

This technique is often considered to enhance reservoir productivity, prevent or mitigate potential damage to formation, minimize the risk of lost-circulation and maximize rate of penetration (ROP). However, drilling while maintaining a bottom hole pressure lower than the pore pressure will usually increase the risk of borehole instability because of yielding or failure of the rock adjacent to the borehole. (Mclellan & Hawkes, 2001) [22]

Managed Pressure Drilling

Managed pressure drilling (MPD) is used in those unconventional drilling prospects where the difference between the pore pressure and fracture pressure is so low that even the fluctuations in bottom hole pressure (BHP) caused by mud pump operation threatens the well-bore integrity (Hilts, 2013). [23]

The main aim of MPD is to solve a series of drilling problems, enhance drill-ability, reduce costs by decreasing NPT. Compared with conventional drilling methods MPD has a special set of characteristics which are summarized below (Guo et al., 2011). [2]

- The core purpose of MPD is to control the bottom hole pressure within a desirable range. Figure 2.3 shows a comparison between the performance of MPD and conventional drilling method under various drilling situations such as pumping up the flow rate, tripping, connection, pumping down the flow rate. It can be observed that the dynamic response of MPD is very stable and less noisy then conventional drilling. The smoother response in case of MPD is because of precise well-head pressure control as well as the drilling fluid density which is kept below the formation pore pressure equivalent drilling fluid density. (Guo et al., 2011). [2]

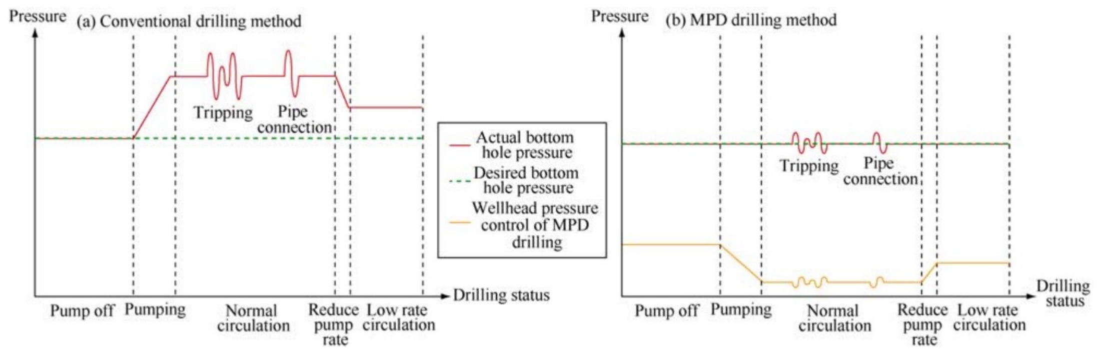


Figure 2.3: Bottom hole pressure comparison between conventional drilling and MPD drilling, taken from Guo et al., 2011. [2]

- The MPD technology can be divided based on application into several types, some of them are Constant bottom hole pressure MPD, Dual gradient MPD, Pressurized mud-cap, Down-hole pumping MPD etc. Constant bottom hole pressure MPD is relatively easier to implement and has most use cases. Similarly MPD can be characterized according to the implementation method as Reactive and Proactive MPD. The former is designed before drilling and operated as designed procedure, where as latter is operated with installed equipment when required (Guo et al., 2011). [2]

- MPD has more control variables (discussed in Table 2.1) as compared to the conventional drilling methods which enable MPD to control the BHP in narrow operational windows much more effectively. For example in case of conventional drilling annulus friction cannot be applied when pump stop and drilling density cannot be adjusted at every desired moment, in these scenarios MPD can control pressure much more effectively by using back pressure control or dual drilling fluid density (Guo et al., 2011). [2]

- MPD uses specific equipment which distinguishes it from conventional drilling such as rorating control device, surface pressure control system, continuous circulation system, multi-phase separator and other specialized equipment that have been used in various applications of MPD. (Guo et al., 2011). [2]

Table 2.1: Brief comparison between MPD and conventional method, taken from (Guo et al., 2011) [2]

Drilling Methods	Control Variables	Control methods
Conventional Drilling	Flow rate of drilling fluid	Adjust annulus friction
	Density of drilling fluid	Adjust density of drilling fluid
MPD Drilling	Flow rate of drilling fluid	Adjust annulus friction
	Density of drilling fluid	Adjust density of drilling fluid
	Wellhead back pressure	Sealed wellhead or choke valve
	Downhole pressure at certain depth	Special down hole tool

It can be summarized at this point that MPD performs a special adjustment on one of its control-parameters to optimize the bottom-hole pressure. One of these methods is the tweaking of well-head back pressure, this adjustment can be understood by the following equations 2.4 and 2.5. (Guo et al., 2011) [2]

$$p_w = p_h + p_f \tag{2.4}$$

In conventional drilling the BHP is given by equation 2.4. Where, the BHP is denoted by p_w, p_h represents static hydraulic pressure and p_f is annular friction loss pressure. Equation 2.5 explains the calculation of BHP p_w when MPD is employed in the drilling procedure.

$$p_w = p_h + p_f + p_b \tag{2.5}$$

The new term p_b, represents the surface back pressure. (p_b) is the ideal control variable, as adjustment of p_b will result in an instant change in p_w (Guo et al., 2011). In automated MPD, generally the surface pressure term (p_b) is automatically adjusted by controlling a choke valve opening by a controller (Gravdal et al., 2014) [24].

2.1.2 Automation techniques in Managed Pressure Drilling

In the last section, managed pressure drilling and its utility in drilling processes was discussed in detail, the benefits of using MPD include reduced formation damage, longer reach drilling, rapid change in BHP during influx situations and potentially early kick detection [25]. The focus of this section will be on highlighting various automation techniques used in the implementation of MPD.

Application of Proportional Integral Differential Controller in MPD

In this subsection, a specific control architecture that has become almost universally used in industrial control will be discussed. It is based on a particular fixed structure controller family, the so-called, Proportional Integral Differential (PID) controller family. They have proven to be robust in the control of many important applications. [26]

A standard PID controller is also known as the "three-term" controller, and its transfer function (in Laplace domain) is generally written in the "parallel form" given by equation 2.6 or the "ideal form" given by equation 2.7. [17]

$$G(s) = K_P + K_I \frac{1}{s} + K_D s \tag{2.6}$$

$$G(s) = K_P(1 + \frac{1}{T_I s} + T_D s) \tag{2.7}$$

In the equations 2.6 and 2.7, K_P is the proportional gain, K_I the integral gain,K_D the derivative gain, T_I the integral time constant and, $T_D$the derivative time constant. The "three-term" functionalities are highlighted by the following.

- The proportional term (K_P), is responsible for providing an overall control action proportional to the error signal through the all-pass gain factor.

- The integral term (K_I), is responsible for reducing steady-state errors through low-frequency compensation by an integrator.

- The derivative term (K_D), is used for improving transient response through high-frequency compensation by a differentiator.

The design process of controller requires tuning of K_P, K_I, K_D according to the given performance criteria which is dependent on the application where the PID controller is employed [26] .The usual constraints given for a system design are in terms of percentage overshoot, rise time, settling time and steady state error (SSE).

Rise Time is the time required by the system to reach 90% from 10% of the steady-state, or final value. Percent overshoot is the relative percentage amount by which the control variable (process variable) overshoots the final value. Settling time is the time taken by the control variable to settle to within a certain percentage (commonly 5%) of the final value. Steady-State Error is the final difference between the control variable and set point. These constraints are graphically visualized in Figure 2.4. [26]

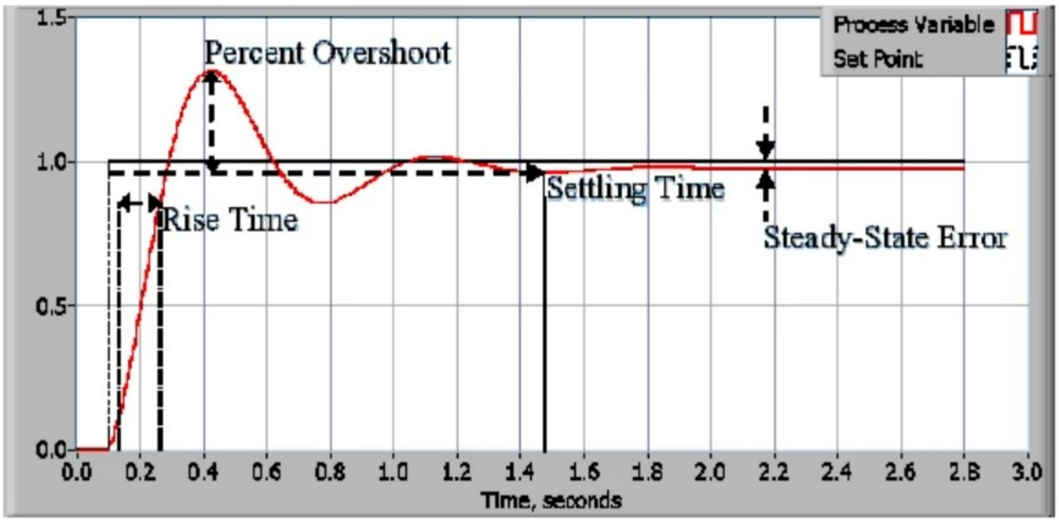


Figure 2.4: Rise Time, Settling Time, Peak Time and Overshoot and Steady state error. [3]

The individual effects of the three terms K_P, K_I, K_D on the closed-loop performance are summarized in Table 2.2. This table serves as a first guide for stable open-loop plants only. For optimum performance K_P, K_I (or T_I) $and K_I$ (or T_D) are mutually dependent in tuning. [17]

Closed Loop Response	Rise Time	Overshoot	Settling Time	SSE	Stability
Increasing K_P	Decrease	Increase	Small Increase	Decrease	Degrade
Increasing K_I	Small Decrease	Increase	Increase	Large Decrease	Degrade
Increasing K_D	Small Decrease	Decrease	Decrease	Minor Change	Improve

Table 2.2: Effects of independent P, I, AND D tuning [17]

After this brief introduction to PID controllers, its application to the process of MPD can be discussed. In manual MPD operation, a human operator adjusts the choke opening manually. For example, a look-up table is used to get the back-pressure for different flow rates, and the operator adjusts the choke opening until this desired pressure set-point is reached. The choke valve must be opened to decrease the back-pressure if it is too high and closed if the pressure is too low. This is a challenging task, as the pressure might change abruptly, especially during a drilling situation called a connection (See Figure 2.3), when the pump rate is ramped down and up. [4]

In the case of manual choke operator is required to coordinate the choke movement with the drillers operation of the main pump. In automatic MPD, this choke operation is done by the control system without any manual interaction. Godhavn (2009), discusses an automated control system for MPD, and the results obtained from it after its implementation at Kvitebjørn, which is a HPHT field located in North Sea. [4]. In Figure 2.5, a schematic of the MPD setup at Kvitebjørn is discussed along with the rig pump, auxiliary pump, choke, control system, and hydraulic model, later on the well is replaced by a mathematical model for ease of testing during the study.

The automatic control system reads the surface back-pressure and adjusts the choke valve opening accordingly. The performance in a manually operated drilling system highly depends on the operators as it can be affected by their interpretation, attention and skill-set and in drilling scenarios involving narrow operational windows there is little room for mistakes. Another advantage of automatic control is the response time or as discussed earlier constraints such as Rise

Time and Settling Time which can be adjusted by tuning K_P, K_I and K_D of the PID controller. [4]

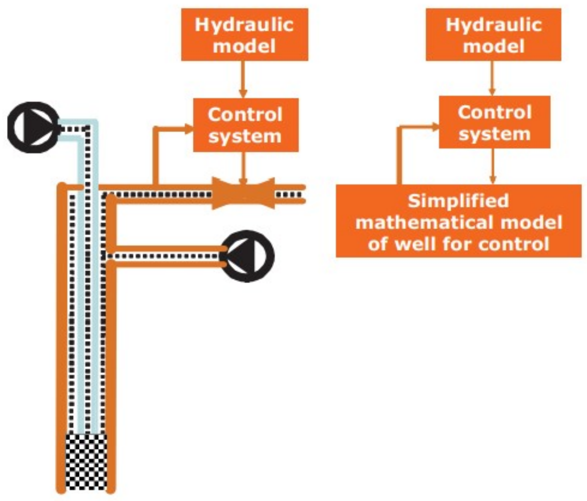


Figure 2.5: — Left: Simple schematic drawing of MPD setup at Kvitebjørn field. Right: Well is replaced with simple model for controller development and testing. [4]

In Figure 2.6, the results obtained from employing PID controller for MPD implementation at Kvitebjørn are discussed. The results obtained after tuning the feed-back loop based proportional, integral, and differential (PID) controller are satisfactory. It can be observed in Figure 2.6 that the measured choke pressure follows the set-point generated by the hydraulic model closely.

Application of Deep Reinforcement Learning in MPD Control

Recent research in MPD automation explores the usage of Deep Reinforcement Learning (DRL) in control systems. The idea behind using DRL agents as control systems is based on eliminating the common weaknesses of PID controllers such as their linearity and in-ability to adapt to the dynamic physical system. [27]

ArnØ et al. (2020), [5] discusses the application of Deep Reinforcement Learning (DRL) for bottom hole pressure control in MPD. As shown in Figure 2.7, the automation method used by ArnØ et al. (2020) and developed by Kaasa et al. (2012) consists of two main parts: a hydraulic model and a pressure control system. The hydraulic model estimates the desired BHP

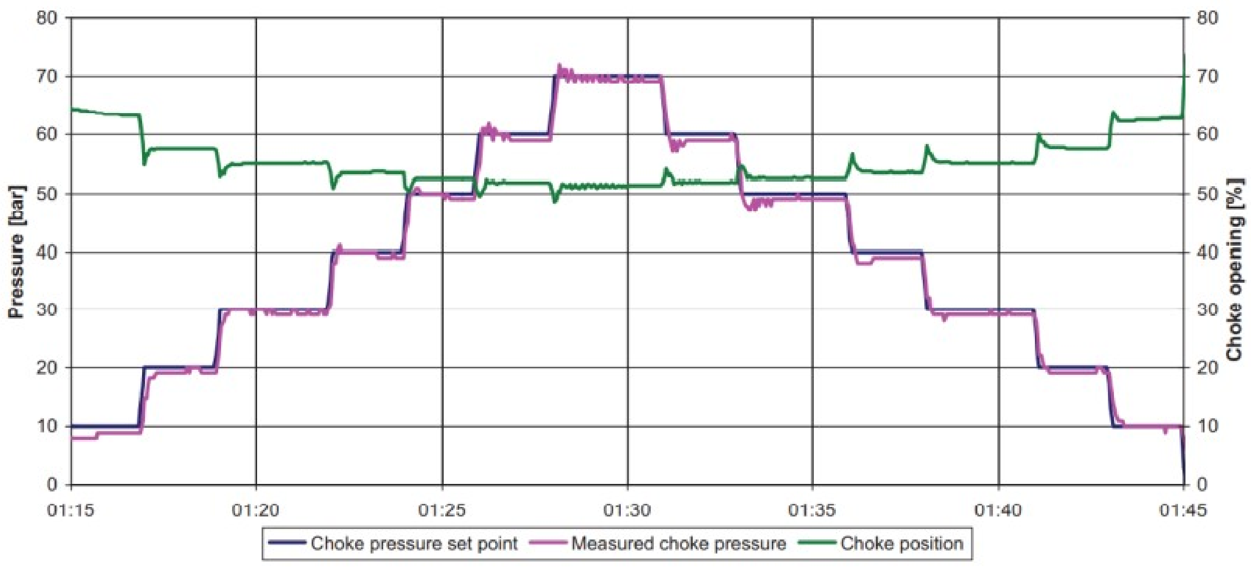


Figure 2.6: Closed-loop test of PID controller with step responses to changes in the choke pressure set-point (From Godhavn, 2009) [4]

(p_{dh}) and yields a set point called p_{cref} for the pressure control system. The pressure control system controls the choke valve opening to adjust surface back pressure and choke pressure (p_c). The pressure control approach discussed by ArnØ et al. (2020) involves a Q-learning based RL agent which is trained on a diverse dataset corresponding to varying operational depths. Deep Q learning is used to compute the optimum choke valve opening to keep the BHP balanced. [6]

The process of finding the optimum choke opening which can be refered to as an optimum action depends on design of training environment, states and most importantly reward function. The reward function used by ArnØ et al. (2020) is designed such that the pressure control block closely follows the given set point and avoids excessive actuation of the choke valve while doing its job. The plot for cumulative reward per episode during training is shown in Figure 2.8. This plot shows how good the actions taken by the pressure control block (recall Figure 2.7) are in every training simulation run. The behavior is as expected, the performance improves as the training process progresses, the variance (or oscillations) of the reward is reduced significantly towards the end which means the model can generalize well.

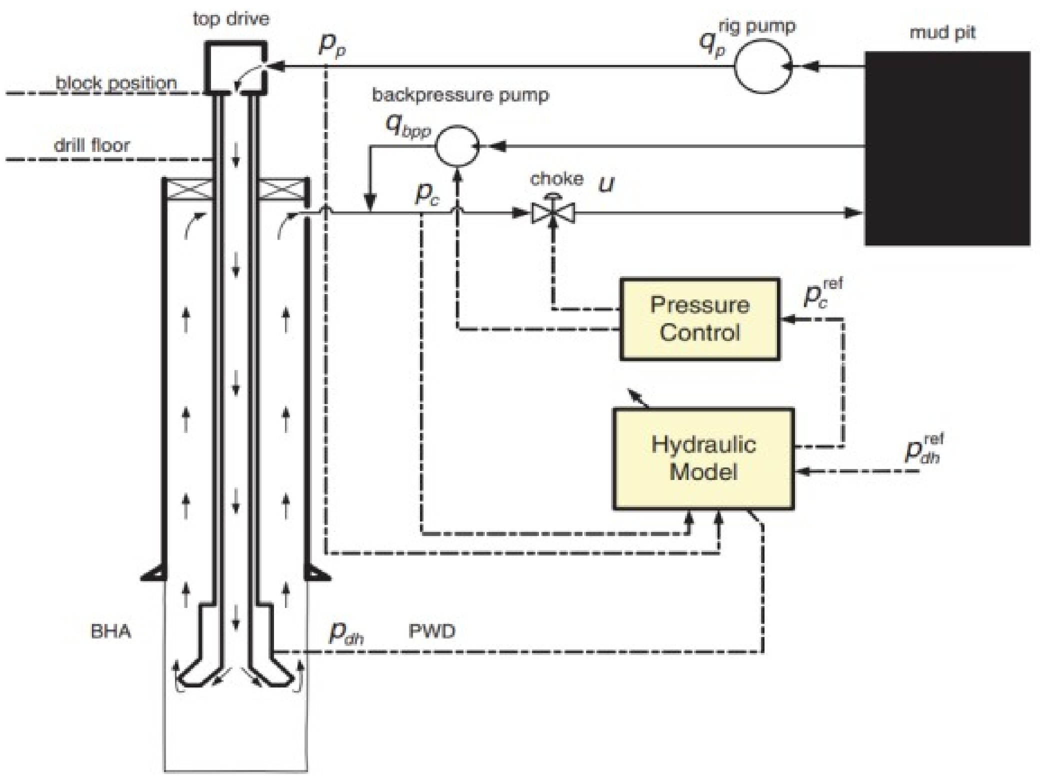


Figure 2.7: MPD schematics used by ArnØ et al. (2020) [5] (Taken from Kaasa et al., 2012) [6]

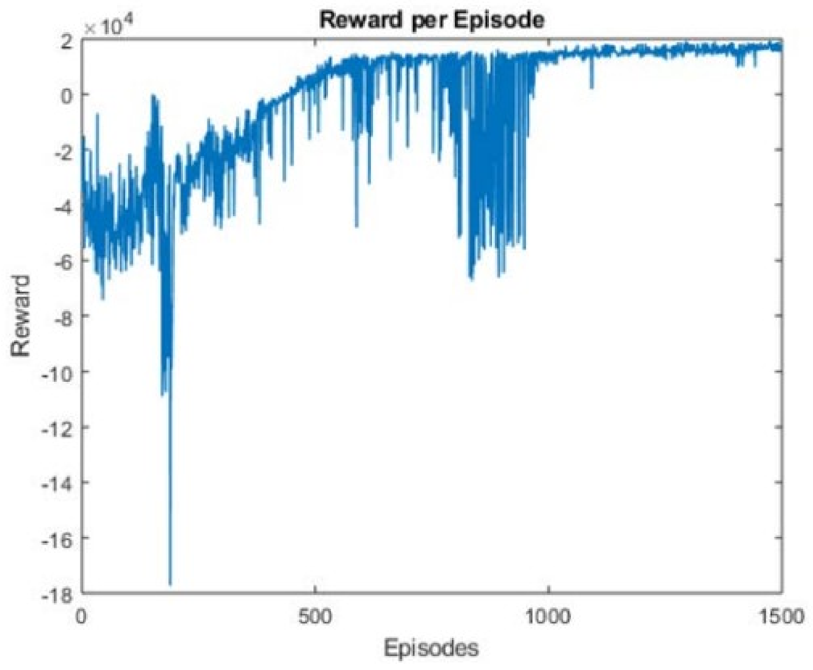


Figure 2.8: Reward per episode for training, taken from ArnØ et al. (2020) [5]

2.2 Reinforcement Learning

Reinforcement Learning (RL) is an interesting area in the domain of artificial intelligence (AI) which is very dynamic in terms of theory and also its application. Reinforcement Learning algorithms investigate the behavior of parameters in environments and learn to optimize their behavior [7].

2.2.1 Brief Introduction

RL algorithms is generally classified classified into two types, Model-Free RL and Model-Based RL as shown in Figure 2.9. Model-based RL uses planning and models of the environment to complete the learning tasks, as opposed to simpler model-free methods that are trial-and-error learners and the value is associated with actions, this can be regarded as opposite of planning or modelling. [8]

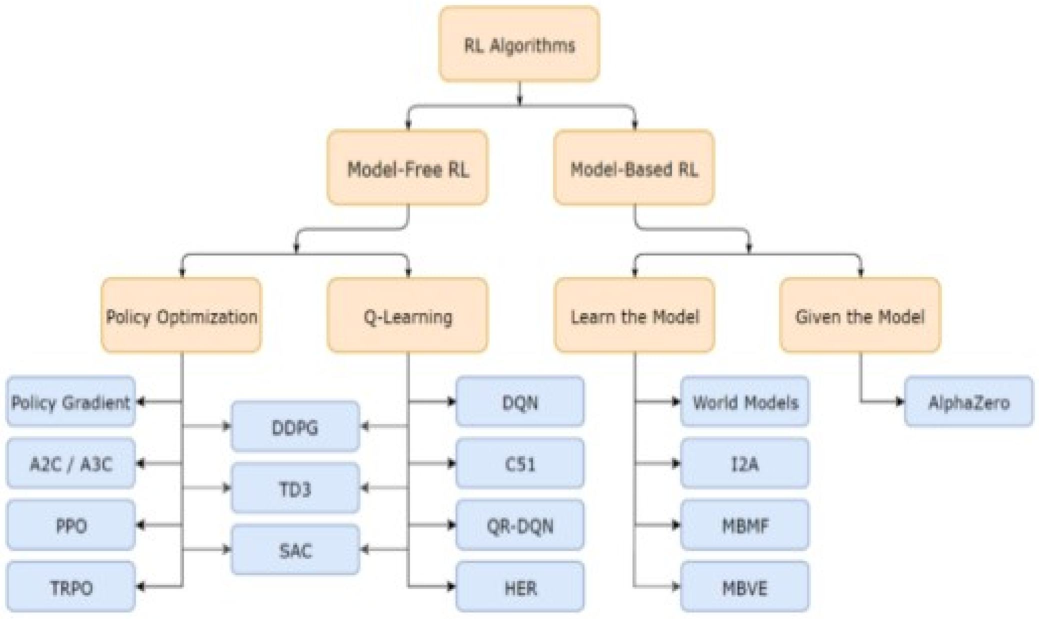


Figure 2.9: Reinforcement Learning algorithm types. [7]

The model -free algorithms are the area which is focused on in this study, and these algorithms can be further categorized into Value-based (Q-Learning) RL, and Policy-based RL. The examples of policy-based algorithms include Proximal Policy Optimization (PPO), Advantage Actor Critic (A2C) method and others. PPO is a method which is used later on in this study [7].

Basic Nomenclature

There are a little terminologies like Agent, Action (A), State (S), Environment and Reward (R). These terms are repeatedly used in reinforcement learning and they are defined precisely in Table 2.3 [7]. Based upon these definitions we can define some more terms such as Expected Return(G), Discounted Return and Policy(π) and value functions.

Agent or Actor	The learner and decision-maker.
Environment	This is the model/program where actions take effect and agent observe it to make decisions.
Action	Set of actions (a) which can be performed in an environment.
State	The set of possible states (s) of an agent in the environment.
Reward	Each action performed by the agent provides a positive or a negative reward.

Table 2.3: Terms used in Reinforcement Learning [7]

Expected Return (G) is the cumulative sum of rewards which the agent tries to maximize as shown in the equation 2.8 below.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.8}$$

where T is a final time step of the environment episode. *An episode* is considered to end when an agent took a series of actions which led to different states but after one of the actions the agent arrived at a state called a terminal state and there can no more states, hence that sequence of states and actions marks completion of one episode in the environment. Usually in training procedure the environment is reset (goes to the initial state) and the agent starts again onto the next episode. In case of *Discounted Return*, discount rate ($\gamma \in$ [0,1]) is used to discount the future rewards and determine the present value of future rewards so that more immediate rewards are given more importance. Hence, expression of Discounted Return becomes as shown in equation 2.9 below.

$$G_t = \sum_{k=1}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \tag{2.9}$$

Policy (π): the function that is responsible for mapping any given state in an environment, to probabilities of each possible action from that state. *Value function*: This is function of state, it gives a measure about how appropriate it is for the agent to be in the current state. It is denoted by “V(π)”. There are also functions of state-action pairs that estimate how good it is for an agent to perform a given action in a given state (*Action-value function*) which is denoted by “Qπ”. Both of these functions are given in terms of Expected Return “Eπ” as shown in equation 2.10 and 2.11. [28] [29]

$$V_\pi(s) = E_\pi[G_t|S_t = s] \tag{2.10}$$

$$Q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a] \tag{2.11}$$

2.2.2 Modelling a Reinforcement Learning problem

In this section various problem frameworks within Reinforcement Learning will be discussed which are useful within the scope of this study. *Markov Decision Processes* (MDP) will be the starting point and then a special case of MDP will be discussed which is called *Bandit Problem*.

Markov Decision Process

Markov Decision Processes (MDPs) is a framework for solving a learning problem, where an agent interacts with an environment to achieve a goal. The learner is called the *agent* in this framework. The object with which the agent interacts, is called the *environment*, it is basically everything outside the agent in the defined problem. During the learning process the agent and environment interact continuously, the agent selects various actions and the environment responds to these actions and presenting new situations (or states) to the agent. As a result of these actions environment gives out rewards, which are special numerical values that the agent seeks to maximize over time through its choice of actions. Figure 2.10 describes this interaction visually where subscripts t and $t + 1$ denote current time step and next time step with-in one episode. [8]

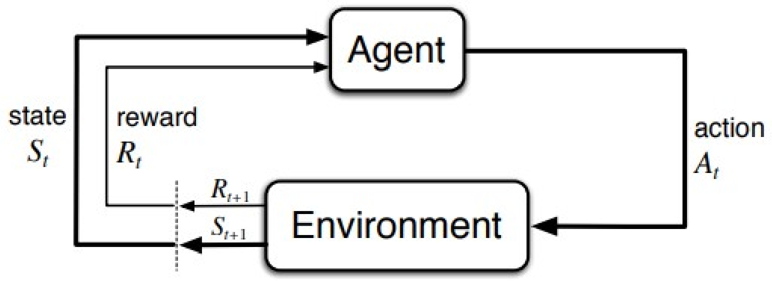


Figure 2.10: The agent–environment interaction in a Markov decision process. [8]

In Figure 2.10, the agent and environment interact with each-other sequentially in discrete time steps, $t = 0, 1, 2, 3, ...$ At each time step *t*, the agent gets a representation of the current state of the environment, $s_t \in$ S, and based on this knowledge selects an action, $a_t \in$ A(s). One time step later, as a result of the previously made action, the agent now gets a positive or a negative numerical reward, $r_{t+1} \in$ R (set of possible rewards), which is always a realnumber, and finds itself in a new state, s_{t+1}. [8] The MDP and agent together thereby give rise to a sequence or trajectory that looks like equation 2.12 [8].

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, ... \tag{2.12}$$

In a finite MDP, the states, actions, and rewards (denoted as S, A and R in the above paragraph), exists in form of finite sets of elements. This basically means that the action space/ set has a limited number of values and considering our environment to be stable on all those actions it will yield a finite number of states as a result of these finite number of actions assuming one of them is a terminal state. In this case, the discrete probability distributions of random variables R_t and S_t are only dependent on the preceding state and action. That is, for particular values of these random variables (S, A and R). $s^{`} \in$ S and r $\in$ R, there is a probability of those values occurring at time t, given particular values of the preceding state and action. [8]

$$p(s^{`}, r|s, a) = Pr(s_t = s^{`}, r_t = r|s_{t-1} = s, a_{t-1} = a) \tag{2.13}$$

In equation 2.13, for all the states "s", $s^{`} \in$ "S", all rewards "r" $\in$ "R" and all actions "a" $\in$ "A(s)", there is a function "p" defining the dynamics of the MDP. The ‘|’ in the middle of it comes from the notation for conditional probability but here it just reminds us that p specifies a probability distribution for each choice of s and a, that is, given by equation 2.14. [8]

$$\sum_{s^{`} \in s} \sum_{a^{`} \in A(s)} p(s^{`}, r|s, a) = 1 \tag{2.14}$$

For all s ∈ S, a ∈ A(s).

"In a Markov decision process, the probabilities given by *p* completely characterize the environment's dynamics. That is, the probability of each possible value for s_t and r_t depends only on the immediately preceding state and action, S_{t-1} and A_{t-1}, and, given them, not at all on earlier states and actions. This is best viewed a restriction not on the decision process, but on the state. The state must include information about all aspects of the past agent–environment interaction that make a difference for the future. If it does, then the state is said to have the *Markov property*". [8]

"The MDP framework is abstract and flexible and can be applied to many different problems in many different ways. For example, the time steps need not refer to fixed intervals of real time; they can refer to arbitrary successive stages of decision making and acting. The actions can be low-level controls, such as the voltages applied to the motors of a robot arm, or high-level decisions, such as whether or not to have dinner or to go to university. Similarly, the states can take a wide variety of forms. They can be completely determined by low-level sensations, such as direct sensor data, or they can be more high-level and abstract, such as symbolic representations of objects in a studio. Some of what makes up a state could be based on memory of past sensations or even be entirely mental or subjective." [8]

"For example, an agent could be in the state of not being sure where an object is, or of having just been surprised in some clearly defined sense. Similarly, some actions might be totally mental or computational. For example, some actions might control what an agent chooses to think about, or where it focuses its attention. In general, actions can be any decisions we want to learn how to make, and the states can be anything we can know that might be useful in making them." [8]

Multi Armed Bandit Problems or k-armed Bandit Problems

"Multi-armed bandit (MAB) problems are a class of sequential resource allocation problems concerned with allocating one or more resources among several alternative (competing) projects. Such problems are paradigms of a fundamental conflict between making decisions (allocating resources) that yield high current rewards, versus making decisions that sacrifice current gains with the prospect of better future rewards". [30]

In order to explain bandit problems, lets consider the following learning problem. The agent is faced repeatedly with a choice among k different options or actions. After each choice a specific numerical reward is assigned to the agent depending upon the action selected. The objective is to maximize the total reward over some time period. For example over 1000 action selections or time steps. [8]

"The name bandit problem is an analogy to a slot machine, or "one-armed bandit" except that it has *k* number of levers instead of one. Each action selection is like a play of one of the slot machine's levers, and the rewards are the payoffs for hitting the jackpot. *Through repeated action selections you are to maximize your winnings by concentrating your actions on the best levers*. Another analogy is that of a doctor choosing between experimental treatments for a series of seriously ill patients. Each action is the selection of a treatment, and each reward is the survival or well-being of the patient." [8]

"In Multi-armed bandit problems each of the k actions has an expected or mean reward given that that action is selected; it can be called the value of that specific action. The notation for the action selected on *time step t* as A_t, and the corresponding reward as R_t. The value then of an *arbitrary action a*, denoted $q_*(a)$, is the expected reward given that a is selected." [8]

$$q_*(a) = \mathbf{E}[R_t|A_t = a] \tag{2.15}$$

Obviously, it is assumed that the value of each action is not known otherwise the problem would become deterministic, but the estimated values of actions are known and these are denoted as $Q_t(a)$, which is the estimated value of action *a* at time *t*, the final goal is to make $Q_t(a)$ as close to $q_*(a)$, as possible. [8]

If the estimated values of corresponding actions are maintained, then the action with highest estimated value at a given time step *t* is referred to as the greedy action or actions in some cases. If the agent keeps on selecting the greedy action then it is referred to as *exploitation* of the current knowledge and if the agent tries other action rather the one with the highest estimated value or reward then this phenomenon is referred to as *exploration* and this enables the agent to improve its knowledge of the environment or knowledge about the rewards associated with pulling different arms. There is always a exploration and exploitation trade off and it is normally understood by testing on the problem under study. [8]

2.2.3 Reinforcement Learning Techniques

In this section various reinforcement learning techniques which are considered in this study will be discussed but their actual implementation to MPD will be discussed in later chapters.

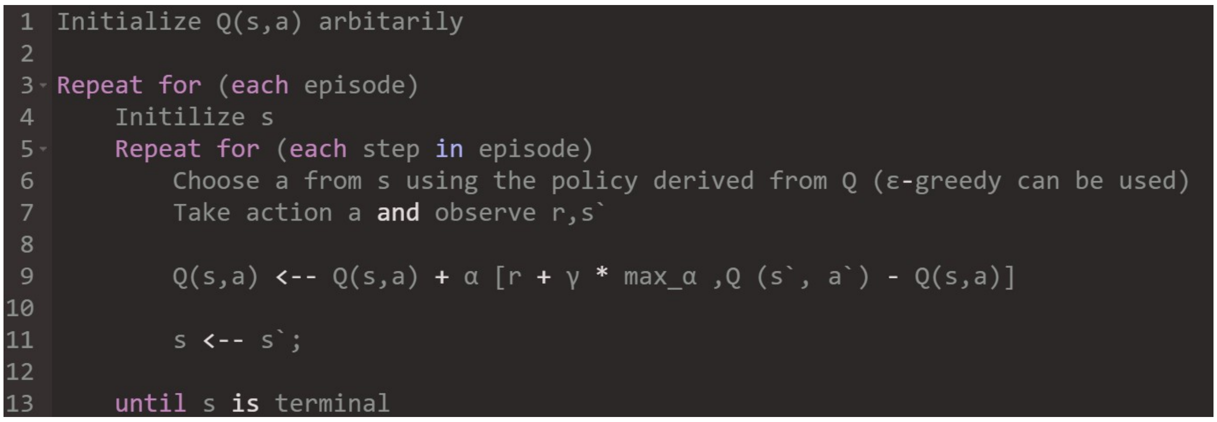
Q Learning and Deep Q-Learning Networks (DQN)

In order to solve sequential decision problems the agent need to learn estimates for the optimal value of each action, defined as the expected sum of future rewards when taking that action and following the optimal policy thereafter. Under a given policy (π), the true value of an action *a* in a state *s* is given by equation 2.16. [31]

$$Q_\pi(s,a) = \mathbf{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ...|S_0 = s, A_0 = a, \pi] \tag{2.16}$$

where $\gamma \in$ [0, 1] is a discount factor that trades off the importance of immediate and later rewards. The optimal value is then Q_* (s, a) = max_π Q_π(s, a). An optimal policy is easily derived from the optimal values by selecting the highest valued action in each state [31]. Estimates for the optimal action values can be learned using Q-learning which is an Off-Policy algorithm for Temporal Difference learning algorithm. The basic psuedocode for Q learning is shown in Figure 2.11. The symbol α is the learning rate in the psuedocode which controls how often the function Q(s,a) is updated. The update done in simple Q-learning is given by equation 2.17. [9]

$$Q(s,a) + \alpha[r + \gamma max_a, Q(s`, a`) - Q(s,a)] \rightarrow Q(s,a,\theta) \tag{2.17}$$



```
Initialize Q(s,a) arbitarily

Repeat for (each episode)
    Initilize s
    Repeat for (each step in episode)
        Choose a from s using the policy derived from Q (ε-greedy can be used)
        Take action a and observe r,s`

        Q(s,a) <-- Q(s,a) + α [r + γ * max_α ,Q (s`, a`) - Q(s,a)]

        s <-- s`;

    until s is terminal
```

Figure 2.11: Psuedocode for Q Learning . [9]

It can be observed that if the action and state space is very large this algorithm will become computationally expensive as all state-action pairs will be stored in a Q-Table. The Function Approximation approach is designed to solve the problem of large state spaces, also known as "dimensional disasters". By using functions instead of Q-tables to represent Q(s,a), this function which can be linear or non-linear $Q(s, a; \theta) \approx Q(s, a)$ [10]. If action-value function $Q(s, a)$ is estimated by a multi-layered perceptron (MLP) and this is main idea behind deep Q Learning [32]. The θ is the weight, so by combining the supervised learning algorithms, the algorithm can calculate the weight θ . Deep Q-Network(DQN), which is a combination of Q-Learning and neural network, turns Q-Learning's Q-table into Q Network, stochastic gradient descent (SGD) is used to iteratively solve the problem and obtain θ. Figure 2.12 discusses the pseudocode for DQN which explains the execution of the technique. Figure 2.13 visually illustrates the difference between simple Q learning and DQN.[10]

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
**for** episode $1, M$ **do** Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    **for** $t = 1, T$ **do**
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in the emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store experience $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of experiences $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the weights $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    **end for**
**end for**
```

Figure 2.12: The psuedo code for DQN. [10]

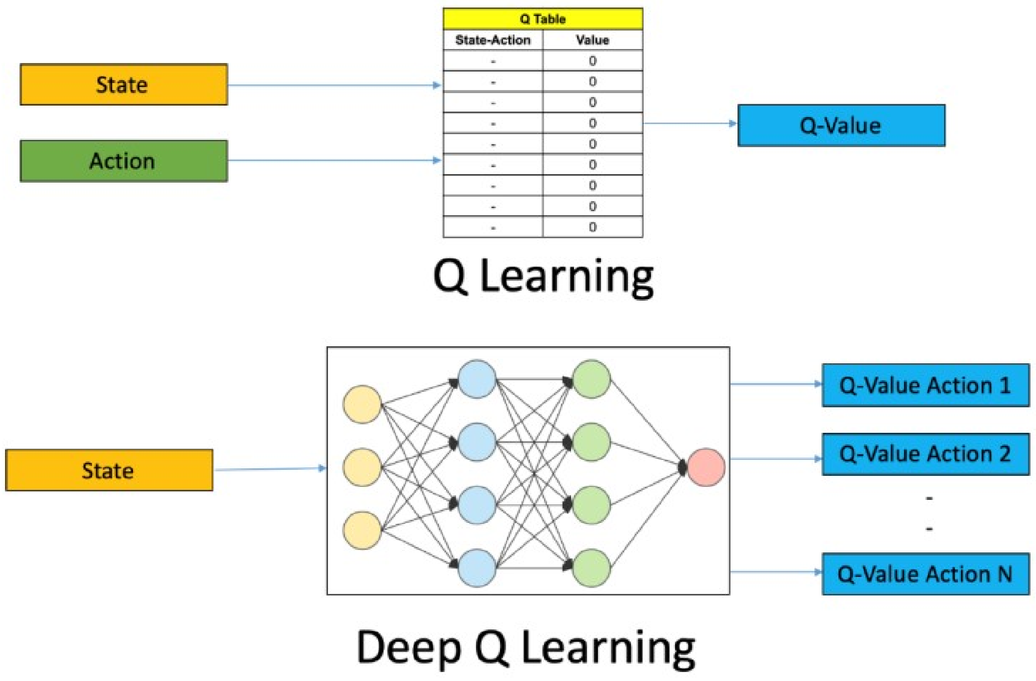


Figure 2.13: Comparison of Q-Learning and DQN. [10]

Actor Critic Methods

Actor-Critic methods are successors of policy gradient methods which unlike Q-Learning are on-policy. The input of the Policy-based algorithm is the same as Value-based, but the output is the probability of each action being selected in the action space. These methods converge much faster than DQN and are more useful in applications where computational time is an important factor. [10]

In Actor-Critic methods the actors are policy networks and they need reward and punishment information to regulate the probability of taking various actions in different states. Traditionally in Policy Gradient algorithms, the reward and punishment information is calculated by iterating through a complete episode. This inevitably leads to very slow learning rate and takes a long time. At the same time Critic Network is a value-based learning method, it can perform a single step update and calculate the reward and penalty values for each step. Then combine the two, the Actor chooses the action, and the Critic tells the Actor whether the action it chooses is appropriate or not. In this process, the Actor iterates to get a reasonable probability of choosing each action in each state, while the Critic iterates to refine the reward and penalty values for choosing each action in each state. Figure 2.14 shows the structure of this above mentioned approach where actor tries to find the best action where as critic tries to refine the choice made by the actor. [10]

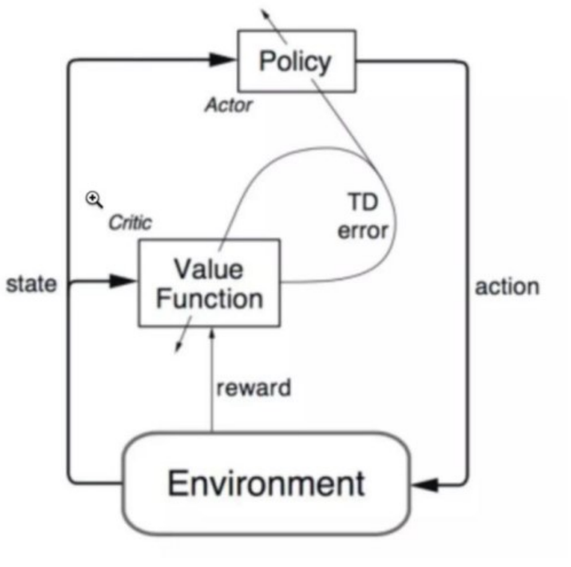


Figure 2.14: The simple structure of Actor-Critic. [10]

An interesting actor-critic method called Proximal Policy Optimization (PPO) is known for its fast convergence properties [33]. PPO, on finds a new balance between the ease of implementation, sampling complexity, and debugging effort required by trying to compute a new policy at each iteration step, which minimizes the loss function while still ensuring relatively small deviations from the policy of the previous iteration. PPO proposes a new target function to achieve small batch updates in multiple training steps, solving the problem of difficulty to determine the step length in the Policy Gradient algorithm. [10] PPO is also one of the main algorithms implemented in this study. The psuedocode for PPO is shown in figure 2.15 below.

```
for iteration=1, 2, ... do
    for actor=1, 2, ..., $N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

Figure 2.15: The simple structure of Actor-Critic. [10]

Epsilon Greedy Methods

The epsilon greedy method is method where the person in charge of the experiment tries to adjust the trade-off between *exploration* and *exploitation*. In other words as it is explained in Figure 2.16, π_ϵ randomly samples an action from Action space (A) with a probability $\epsilon_t \in$ [0,1] and otherwise selects the greedy action according to Q_t. As a result, ϵ_t can be interpreted as the relative importance placed on exploration. [11]

$$\pi^{\varepsilon}(a|s) = \begin{cases} 1 - \varepsilon_t + \frac{\varepsilon_t}{|\mathcal{A}|} & \text{if } a = \arg\max_{a' \in \mathcal{A}} Q_t(s, a') \\ \frac{\varepsilon_t}{|\mathcal{A}|} & \text{otherwise} \end{cases}$$

Figure 2.16: Mechanism of epsilon-greedy method. [11]

The optimal value of the parameter ϵ_t is typically problem-dependent, and found through experimentation. Often, ϵ_t is annealed over time in order to favor exploration at the beginning, and exploitation closer to convergence[8]. However, such approaches are not adaptive since they do not take into account the learning process of the agent [11].

2.3 Applications of Reinforcement Learning

The applications of reinforcement learning techniques range from playing games to self driving cars, playing at casino, solving various control problems or in algorithmic trading. In this section some sample examples will be discussed which will aid in the modelling of the MPD problem in later sections.

2.3.1 Control Problems

Reinforcement learning has been used as a replacement for traditional control system techniques and one of the examples discussed earlier is the application of DQN as a controller for choke valve opening in MPD [5]. It was observed that if the action space and state space is set carefully then under a stable testing environment, DQN was able to perform in an excellent manner (see Figure 2.8).

Some times the traditional control techniques such as PID are combined with the RL algorithms to attain better results. (Hynes et al., 2020) discusses an approach of using residual policy reinforcement learning combined with PID controller for suspension control system of cars. OpenAI gym simulator was used to make a simulation of the suspension and but in this approach both Agent and the PID controller were able to take independent actions to ensure optimum control. [27]

2.3.2 Other applications

In other applications reinforcement learning agents have been tested on various Atari games [34], the main purpose of this research is to test the performance of various RL agents on a similar environment. Some other applications of reinforcement learning includes trading problems, where RL agent is used to make an optimum trading decision at any point in time [35]. In the next section the modelling of our MPD problem will be discussed and the approach to apply reinforcement learning will be discussed.

Chapter 3

Reinforcement Learning for MPD

3.1 Modelling MPD as a RL Problem

The aim of this study is to use reinforcement learning to tune the PID controller gains (K_p, K_I, K_D) as mentioned in Table 2.2, and suggest a suitable pressure set-point, unlike the approach discussed by ArnØ et al. (2020) where the PID controller was replaced by a DQN agent [5]. This way of using reinforcement learning requires a closer look at the definition of terms like episodes, actions, states, steps and rewards which will be discussed in the subsections below.

3.1.1 System Block Diagram

Figure 3.1 shows the basic idea behind the implementation of reinforcement learning in the MPD process. This figure can be related to figure 2.10 where the original structure of Markov Decision Process (MDP) is discussed. It can be observed that the agent is responsible for giving the action to the environment and it receives states and rewards in a step-wise manner.

Furthermore inside the training environment, a PID controller along with a function that specifies the choke valve characteristics and a function which is responsible for well-bore hydraulics is required. The PID controller is obviously implemented in a feed-back loop manner to receive the step-wise value of bottom hole pressure and perform the corrective action if required by controlling the choke opening. The effectiveness and robustness of this corrective action is dependent on the values of the $K_p, K_I and K_D$ which are provided by the agent as well as the bottom hole pressure set-point which is a suggestion from the agent.

Now the step-wise reward is computed by the reward function and this function should be designed to take into account the goodness of the bottom hole pressure set-point as well as decide whether the incoming dynamic response from the hydraulic model is good or not. The former simply means whether the set-point is below the fracture pressure and at the same time above & closer to the pore pressure, whereas the latter means that reward function should account for the percentage overshoot, rise time and over-all stability of the controller response (reduced steady state error and reduced oscillations).

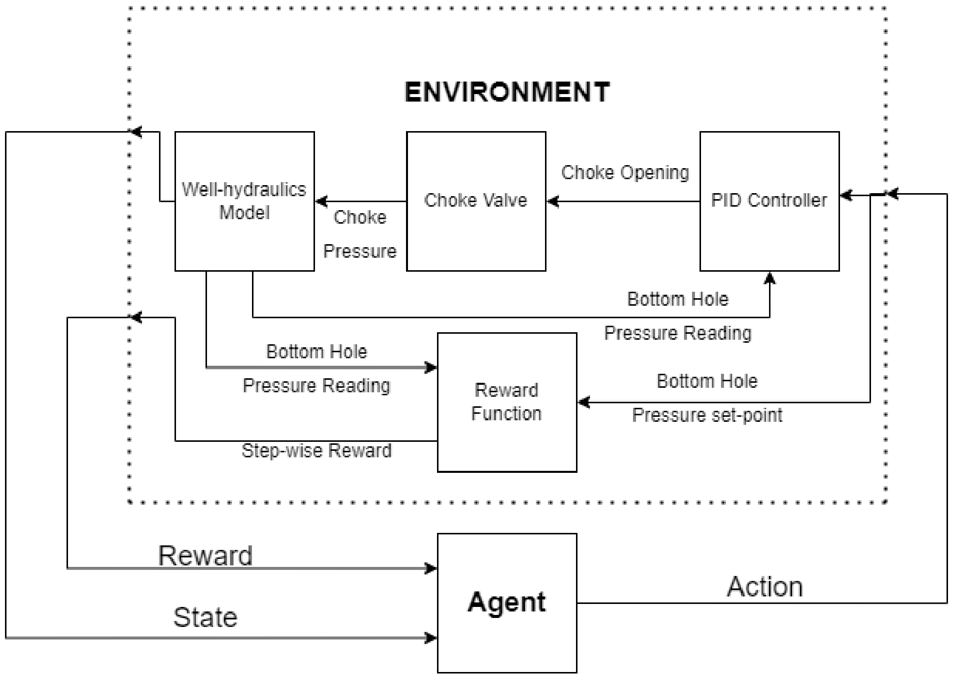


Figure 3.1: Basic block diagram of the MPD problem modelled as Markov Decision Process

It can be observed that the choice of actions will clearly affect the reward function if it is designed according to the above mentioned criteria. More details regarding the reward function design, PID controller used in the current study and well-bore hydraulics will be discussed in the next chapters but in the next section the state and action space will be designed and also the structure of a single training episode and meaning of one step inside an episode will be discussed.

3.1.2 States, Actions and Episode

Lets discuss the states for the MPD problem discussed in the section above. The most basic job for MPD is to keep the bottom hole pressure between pore pressure and fracture pressure, this would automatically mean that the bottom hole pressure set-point sent to the controller must also be kept inside this range. Hence the most important parameters needed to make this decision is the pore pressure and fracture pressure at every depth where the agent is supposed to be trained on. From Figure 2.1 in Chapter 2, it can be observed that pore pressure (P_P) and fracture pressure (P_F) change with respect to true vertical depth (D_{Bit}). These parameters are shown in vector form in equation 3.1. Another parameter which is needed to be kept track of is the step-wise bottom hole pressure incoming from environment at every step in an episode, it is added as the last element in the vector.

$$S = [D_{Bit}\ P_F\ P_P\ P_{BH}] \tag{3.1}$$

Similarly the action space should ideally consist of 4 parameters as shown in equation 3.2 but in case PID controller is replaced with PI or P controller then 3.3 or 3.4 can be the action space as well.

$$A = [K_P\ K_I\ K_D\ P_{Ref}] \tag{3.2}$$

$$A = [K_P\ K_I\ P_{Ref}] \tag{3.3}$$

$$A = [K_P\ P_{Ref}] \tag{3.4}$$

Even in the simplest case where action space is reduced to two actions, it can be observed that agent has to give at least two signals as actions to the environment. Now the definition of episode is the only factor which remains ambiguous in our modelling approach.

An episode is defined as a simulation which should run for a custom number of time steps, not analogous to actual time units as one step can take any defined time, but the key is to keep the K_P K_I K_D P_{Ref} constant during each episode. *The reason behind keeping them constant is that it will enable the reward function to give a reward based on a single set of tuning parameters during one episode* and this will also aid in optimizing the rise time. Considering Figure 3.2, this whole simulation can be regarded as one episode, now while recording this response it is required that we maintain the tuning parameters and set-point to be constant so that the reward function can assign a reward based on stability and overall behaviour of the controller response.

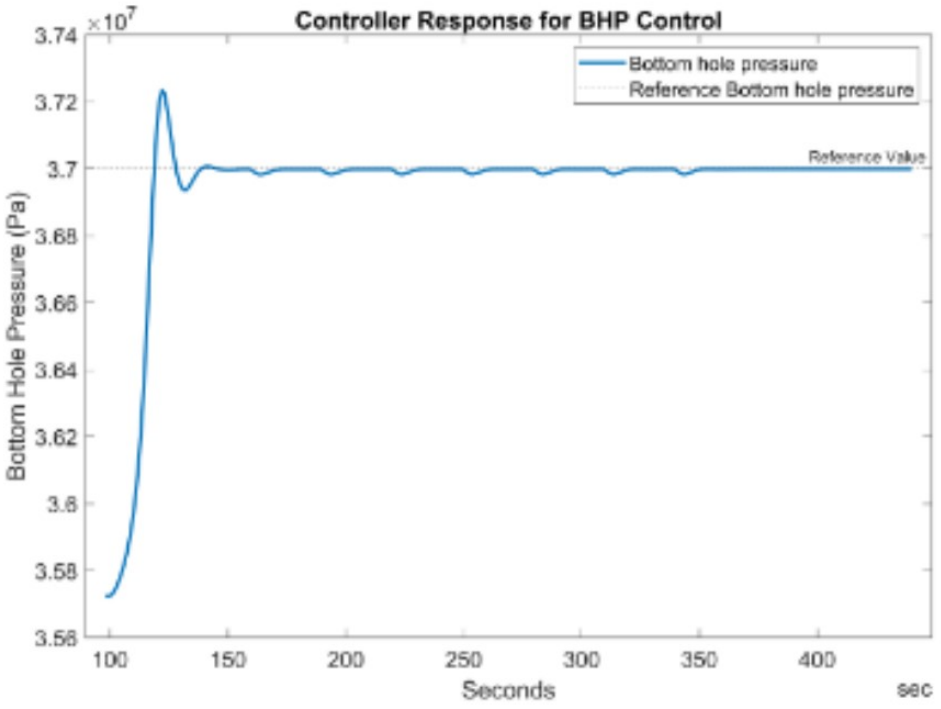


Figure 3.2: Sample Dynamic response of PID controller employed in MPD (Generated by python)

To summarize this our MDP trajectory sequence should look like the following equation 3.5 instead of the one discussed in 2.12.

$$s_0, a_0, r_1, s_1, a_0, r_2, s_2, a_0, r_3, ... \tag{3.5}$$

Just to recall in equation 2.12, the sequence was like this $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3...$ and now a_1 and a_2 are same as a_0.

Further more lets say if a human was to solve this problem then during an episode the human only looks at the initial state to make a decision about the actions. The reason being that decision of BHP set-point depends on formation Geo-pressures i.e. pore pressure (P_P) and fracture pressure (P_F) as well as the depth. The tuning parameters basically decide whether the controller can safely reach the set-point.

This would mean if we are able to train the agent separately at for every depth scenario i.e. dividing the whole depth range into several intervals then the problem is reduced to a single-state MDP problem which can be modelled as a multi-armed bandit (MAB) problem with every arm being a action vector of size 2,3 or 4 depending upon choice of controller (either P, PI or PID recall equations 3.2 to 3.4.). This idea is illustrated in Figure 3.3.

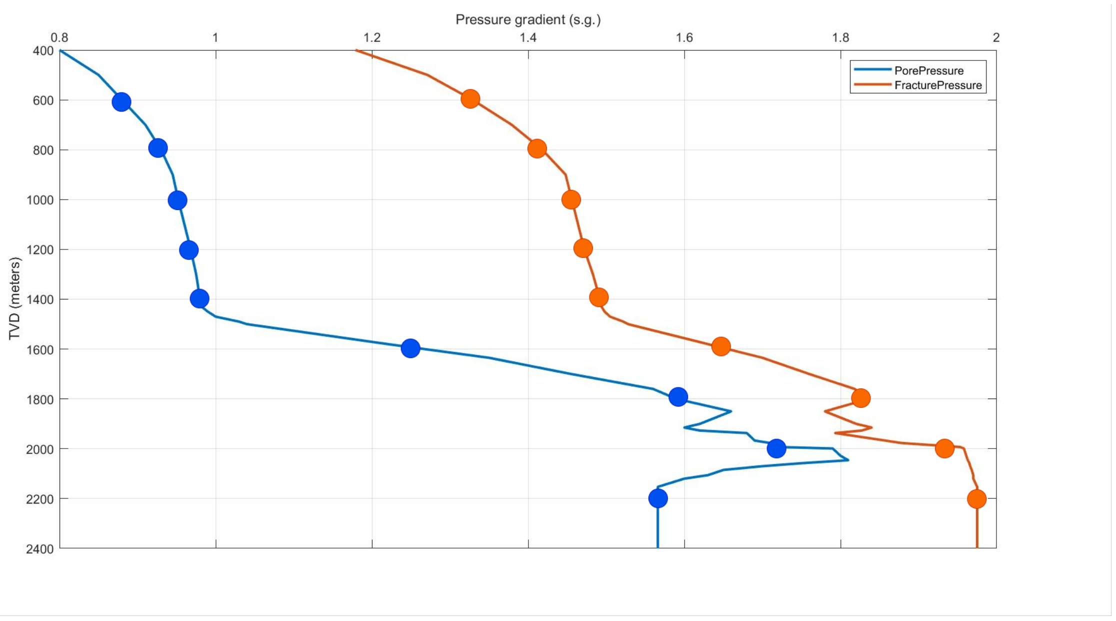


Figure 3.3: The idea of dividing depth into intervals to model a MAB problem (Generated by python)

If the agent is trained at every depth separately it might cause an increase in the training time and it might be less applicable in a practical setting, this constraint will be discussed in the next sections later on.

3.2 Resources for implementation of the Reinforcement Learning agents and environment

3.2.1 MATLAB Reinforcement Learning Toolbox

MATLAB Reinforcement Learning toolbox is used for the implementation of the RL agents. This toolbox provides an application, functions and a Simulink block for training various policies using different reinforcement learning algorithms.

Some of the algorithms provided by this toolbox include DQN, PPO, Soft Actor Critic (SAC) and Deep Deterministic Policy Gradient (DDPG). These policies are used by researchers and students to develop controllers and implement decision making policies for complex applications such as autonomous system, resource allocation and robotics. [12]

The basic workflow followed while using reinforcement learning tool box developed at Mathworks is discussed in Figure 3.4. The formulation of problem has already been discussed in the earlier sub-sections and in the next chapter the development of environment and reward function will be discussed. [12]

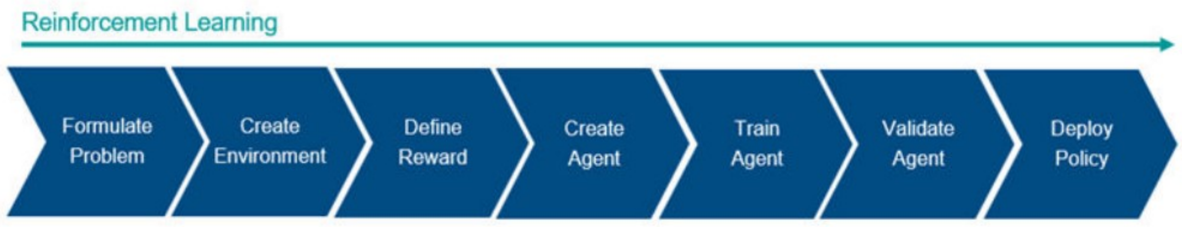


Figure 3.4: Reinforcement Learning workflow using MATLAB RL Toolbox [12]

The toolbox provides different options for agent selection based on the nature of action and observation (state) space i.e. they can be discrete or continuous.

3.2.2 Python - Multi armed bandit algorithms

A part of this study also deals with the modelling the MPD control problem as multi-armed bandit (MAB) problem, the methods used in MAB problems are not available in MATLAB RL toolbox and some minor changes in environment approach are also required hence the development of this part of the proposed solution is done using python but the reward function is kept the same.

3.2.3 Environment design in MATLAB

As discussed in Figure 3.1. the environment needs to have an implementation of PID controller (Appendix A.8), information about of Geo-pressures as the formation changes with depth, it needs to have well-hydraulics model, it should allow changes to choke valve characteristics and most importantly it should be stable in all scenarios. The last attribute means that the implementation of the environment should incorporate excellent error handling and a lot of focus was put into this area which will be discussed in the upcoming section.

Chapter 4

Simulation Environment and Reward Function design

4.1 Simulation Environment Design

In order to develop a robust simulation environment for reinforcement learning, the OpenLab drilling simulator was found to be applicable. The simulator is set up with back-pressure MPD simulation capabilities using high-fidelity models. OpenLab drilling simulator is developed and maintained by the Drilling Well & Modelling group at NORCE Energy, this work is done in collaboration with University of Stavanger (UiS).

4.1.1 Backend Models

The OpenLab simulator uses some back-end models to simulate MPD process in a well of selected settings. The theoretical details of these models are discussed in this sub-section.

Flow Model

The well flow model is based on a framework derived from multi-phase well flow modeling [36] [37]. The model is based on a one dimensional approach of two-phase flow in pipelines formulated by nonlinear partial differential equations, where mass, momentum and energy balance for each phase has been formulated [38]. The dynamics of well flow is determined by these balance equations involving terms for interactions of mass, momentum and energy. The mass conservation equation is given as:

$$\frac{\partial}{\partial t}(\alpha_k A_r \rho_k v_k) + \frac{\partial}{\partial s}(\alpha_k A_r \rho_k v_k^2) = 0, \quad k = \ell, g, \tag{4.1}$$

where the subscripts ℓ, g represent the liquid and gas phase respectively, t is the time variable, α is the volume fraction, A_r is the cross-section area, ρ is the density, s represents the length of the well and v is the velocity of the fluids.

The fundamental two-phase model consists of separate momentum conservation equations for each phase with complicated terms related to phase interaction. To omit modeling of the complex phase interaction term [39] [40] the drift flux model that simplifies the model (equation 4.1) is used in the OpenLab simulator, given as below:

$$\begin{aligned}&\frac{\partial}{\partial t}A_r(\alpha_\ell \rho_\ell v_\ell) + \frac{\partial}{\partial s}A_r(\alpha_\ell \rho_\ell v_\ell^2 + \alpha_g \rho_g v_g^2) + A_r \frac{\partial}{\partial s}p \\ &= -A_r(K - \rho_{mix} g \sin\theta),\end{aligned} \tag{4.2}$$

where p is the pressure, K is a friction pressure-loss term, θ is the well inclination, g is the gravitational acceleration and $\rho_{mix} = \alpha_\ell \rho_\ell + \alpha_g \rho_g$ is the mixture density.

In equation 4.2 there are many unknown model parameters and correlations. The missing information in the mixture momentum equation has to be filled by empirical closure relations that provide estimations of phase velocities and pressure losses. These relations consider complicated time-dependent equations [41][42], and include several estimated parameters. Inaccuracies in these parameters such as complex fluid properties, well-bore geometry, fluid impurities, flow regime and unknown pipe properties will impact on the model accuracy leading to the uncertainties.

Pressure Control

Pressure control is one of the main critical elements of drilling operations. The conventional drilling method implies break in circulation and axial movement of drill-string affecting the well-bore pressure. e.g. before a connection where new drill pipe elements is added or removed from the drill string, the mud circulation need to be stopped causing a rapid change in bottom hole pressure due to the changes in frictional pressure loss. This pressure drop may cause well instability if formation fluids enter into to wellbore in the open hole section because of higher pore pressure compared to well pressure. Therefore, to prevent unintended influx and to ensure safe and stable drilling operations, the bottom hole pressure should be kept within some safe bounds between pore and fracture pressure. Exceeding the fracture pressure, will cause lost circulation where the drilling fluid enters into the formation, and potentially causing instability. Hence, if the pressure in the well is lower than the pore pressure, it will not serve as a barrier against unintended influx into the wellbore (a kick). Potential drilling problems such as formation fracturing, formation ballooning, lost circulation, kick and formation collapse, differential sticking, stuck pipe, and slugging of cuttings return may be encountered as a consequence of improper pressure control.

By utilizing MPD systems, the bottom hole pressure variations related to connection operations, tripping in/out, and changes in flow rate can be significantly reduced by managing the choke valve opening, equivalently, adjusting the surface choke pressure. Although this method has several advantages over conventional drilling, there are limitations and challenges. MPD control manifolds and embedded control algorithms may suffer from limited proof of reliability and viability in the actual operational environment, and this have created a drive for more research on artificial intelligent and automated MPD systems.

In this study, the MPD controller is based on the well-known Proportional-Integral-Derivative (PID) controller. At the beginning, BHP is kept constant and shall be close to a given set-point of the BHP in the controller design. The set-point is allowed to be changed during the operation. The choke valve opening is the manipulated variable and is automatically adjusted by the control system. The PID controller may be based on balancing flow rate in and out of the well [43]. The proposed discrete time PID controller algorithm is expressed as follows:

$$z_c(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{4.3}$$

where

$$e(t) = P_{BH}(t) - P_{BH,sp}(t). \tag{4.4}$$

In the PID controller (equation 4.3), z_c is the choke opening, K_p is the proportional gain, K_i is the integral gain and K_d is the derivative gain. K_p is adjustable to affect the response and reaction of choke pressure. K_i is used to eliminate the difference between the BHP measurements and setpoints. K_d is used to increase the rise time. Normally the range of pressure variations could be accepted within ±2.5 bar.

During drilling, the BHP can be measured with a downhole pressure sensor, but the sensor value is usually transmitted with mud pulse telemetry, suffering from slow and low band-width. Several uncertain factors, for instance, movement of drill pipes, variation of drilling operational parameters, and reservoir influx, may influence this measurement, leading to measurement uncertainties. Another consequence by mud pulse telemetry is that the BHP measurement is not available in real time when the mud flow rate is low and during pipe connections. If the BHP measurements are not available for a while, BHP calculations from real-time flow models or pre-calculated tables are being used. On the other hand, the high uncertainties on measurements increase the difficulty level of finding the suitable setpoints $P_{BH,sp}$ with respect to the time or depth for the well pressure control. The performance of the MPD control will thus depend on the accuracy of models and the choice of the setpoint of the BHP, which require high advanced transient models and the agent for setpoint recommendations.

4.1.2 Environment Block Diagram

The OpenLab simulator is set up with back-pressure MPD simulation capabilities using high-fidelity models as described in the previous sub-section. The simulator is available as a web service and accessible from a web client using the documented OpenLab API [44]. The simulator can be accessed through a web browser, and templates are provided for Matlab and Python clients. The simulator is developed as part of a larger research infrastructure project, and with particular emphasis on drilling automation [45]. The architecture facilitates development and testing of smart agents such as the described RL agent.

A simulation in OpenLab requires a configuration, which is a description of well architecture, trajectory, drill string, drilling fluid, geological properties, and rig equipment. It also requires an initialization of each simulation describing the initial state of the system, and selection of simulation models. Default configurations can be used directly or edited. To ensure that users can not create configurations that are nonphysical or otherwise out of bounds, a built-in validation check ensures that new configurations satisfies a set of validation rules. To ensure stability and reliability, simulations can only be run on approved configurations.

In this work a Matlab client has been used for running the simulations and getting the results from OpenLab. Figure 4.1, discusses the basic architecture of the environment developed using OpenLab simulator API in MATLAB.

In Figure 4.1, the blocks are colored according to their nature and occurrence during a single episode. The block diagram is made to deliver the concept of the environment during a selected number of training or testing episodes. This block diagram does not represents how the environment is implemented in code to match the requirements of MATLAB RL toolbox, but it gives the general idea about the use of OpenLab simulator as a training environment.

Color code for Figure 4.1	
Color	Function
Green	While conditions or If-else conditions
Yellow	This process occurs once in an episode.
Pink	This action occurs once every step in an episode.

Table 4.1: Table to for color code in Figure 4.1.

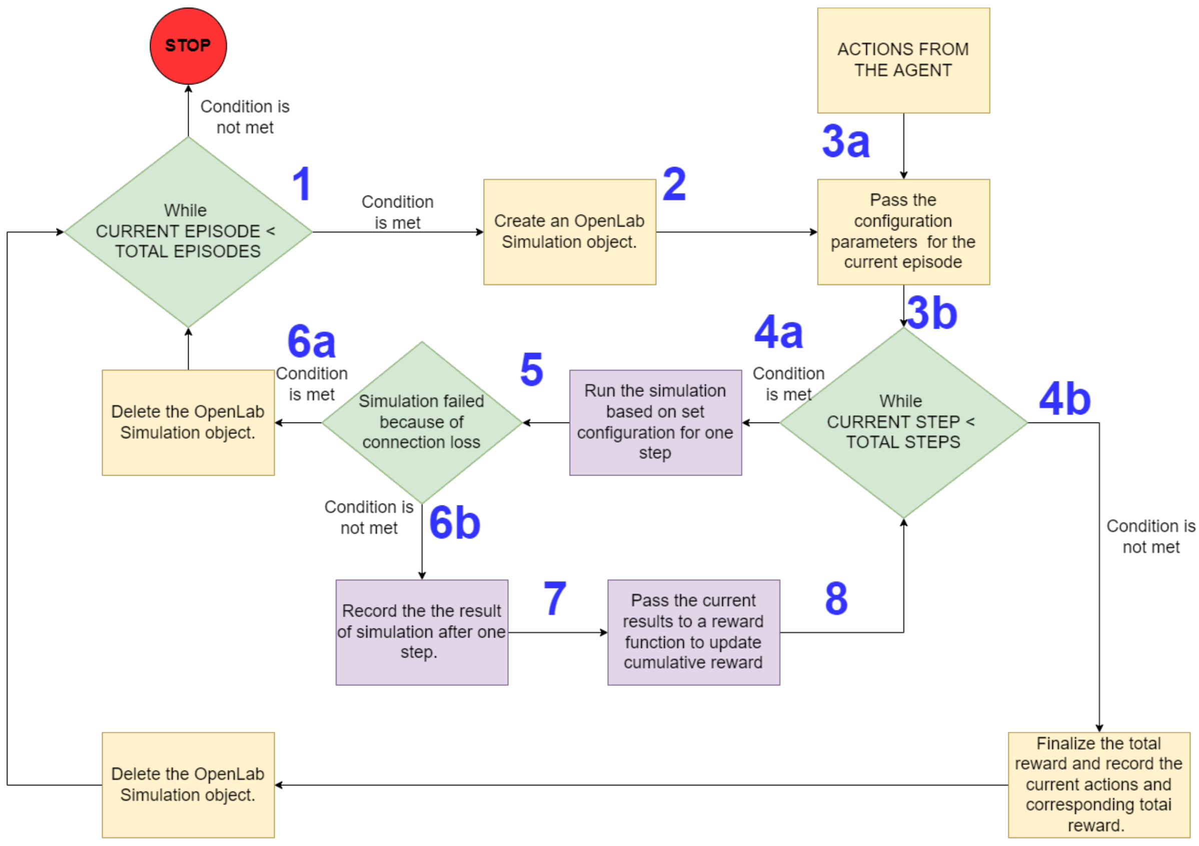

STOP
Condition is not met
While CURRENT EPISODE < TOTAL EPISODES
1
Condition is met
Create an OpenLab Simulation object.
2
ACTIONS FROM THE AGENT
3a
Pass the configuration parameters for the current episode
3b
While CURRENT STEP < TOTAL STEPS
4a
Condition is met
4b
Condition is not met
Run the simulation based on set configuration for one step
5
Simulation failed because of connection loss
6a
Condition is met
Delete the OpenLab Simulation object.
6b
Condition is not met
Record the the result of simulation after one step.
7
Pass the current results to a reward function to update cumulative reward
8
Finalize the total reward and record the current actions and corresponding total reward.
Delete the OpenLab Simulation object.

Figure 4.1: Basic Block Diagram for Environment

The concept shared in Figure 4.1 will be discussed in steps as highlighted in the figure itself (in blue). Some small details are not mentioned in the block diagram and will be covered in the points enumerated below.

1. The process starts from point one, if the current episode number is lesser than the total number of episodes set for the session then the algorithm proceeds ahead and the condition is considered to be met otherwise the simulation is stopped.

2. In step 2 an OpenLab simulation object was created using the OpenLab API for MATLAB client, this helps us in communicating with the OpenLab Drilling Simulator server. User ID and other credentials are required to get this access and only a specific number of simulations are allowed to run in parallel.

3. After creation of OpenLab simulator object, some configurations are needed to be done and some configurations depend upon the actions incoming from the agent.

   (a) The actions from agent are stored in local variables which mainly include controller gains and pressure set-point for the controller.

   (b) The configurations done from MATLAB client mainly include Configuration name, simulation name, initial bit depth, initial tool string location, K_P and K_I, Initial choke opening, Reference BHP and flow rate. The configuration name needs to match the name in the web server and one configuration can have multiple simulation and each simulation corresponds to one episode which have multiple steps.

4. Here the algorithm decides if the total number of steps are still greater than the current step number. Total number of steps also define the length of the dynamic response from our controller a suitable number is chosen in implementation to find a sacrifice between robust response and computational time as a longer response will take more time per episode to train as number of steps would be larger.

   (a) If the condition is met and the current step number is less than the total number of steps then the MATLAB client passes a request OpenLab simulator server to run one simulation step. What this means is that the back-end models are used for a specific time step to compute the next values of bottom hole pressure, annular pressure, choke opening (from controller) and flow rates.

   (b) In case the algorithm has already reached the total number of steps in an episode then the total reward accumulated during the steps is stored and the actions from the controller are also stored. After ward the current simulation object is deleted (as the user has limited number of simulations) and algorithm returns back to the step 1.

5. In case during the any error occurs most of the time error faced in implementation was connection error from OpenLab server then a condition is used to snap out of it and continue the simulation. This is achieved using a try and except structure in MATLAB.

6. In this step it is decided whether the algorithm ran into an error or exception during the step.

   (a) If any error or exception occurs then the algorithm deletes the current simulation object as it is no longer usable and moves on to the next episode by going to the

Step 1.

(b) If there is no error then the result of simulation is recorded, some of these results re needed to take the next step and some are required for computations done in the reward function.

7. The cumulative reward is updated after the step-wise reward it added to it.
8. The algorithm returns to the while-condition where the algorithm checks the current step number against total steps allowed in an episode.

Configuration

As explained briefly in the above sub-section that various configurations can be changed while creating the basic configuration on the OpenLab web portal, and then this configuration is given a specific name which is then used during communication required to run the simulation and each simulation can also have some specific settings (also referred to as configurations) but these were highlighted in the Step 4-a of the Figure 4.1.

The configurations available at the OpenLab web portal include hole-section, well-path, drilling fluid, drill string, geology and rig settings. The most important one for our case is geology and drilling fluid. The parameters which can be changed from the configurations in OpenLab simulator web interface are discussed in Table 4.2. The row highlighted in green are the main headings and then sub-headings are highlighted as purple and parameters are indicated in grey colored rows.

Geo-Pressures in Geology

An important setting in the OpenLab drilling simulator is the Geology setting and especially the Geo-pressures where the user can manually design the pressure profiles and hence define the narrow pressure windows where the MPD is supposed to be most useful. Figure 4.2, shows the geo-pressure setting made in OpenLab simulator‘s web portal and it can be imported in the form

HOLE SECTION					
Riser					
Riser Depth (m)	Riser ID (m)	Riser OD (m)			
Casings					
Casing Type	Hanger depth (m)	Shoe depth (m)	OD (in)	ID (in)	
Openhole					
Length	Openhole Diameter	From Length	To Length		
WELL PATH					
MD (m)	Inc. (°)	Azimuth (°)	TVD (m)	DLS (°/30m)	
DRILLING FLUID					
Fluid density	Fluid type	Mass fractions	Volume fractions	Rheology	
DRILLSTRING					
Drillpipe					
Type	Length (m)	OD (in)	ID (in)	Lin. weight (kg/m)	Cum. length (m)
Bottomhole assembly					
Type	Length (m)	OD (in)	ID (in)	Lin. weight (kg/m)	Cum. length (m)
Bit					
Type	Length (m)	OD (in)	TFA (cm²)	Mass (kg)	Cum. length (m)
GEOLOGY					
Geo-Pressures					
TVD (m)	Pore pressure (s.g.)	Fracture pressure (s.g.)			
Geo-Thermal					
From TVD (m)	Solid Temp. gradient (°C/100m)	Water Temp. gradient (°C/100m)	Air Temperature		
Formation					
MD (m)	UCS (MPa)	Friction angle (°)			
RIG					
Main pump max flowrate acceleration	MPD pump max flowrate acceleration	MPD Choke characteristics	BOP Choke characteristics		
Travelling block weight	Top drive max rot. acceleration.	Draw-works max top string acceleration	Main Tank Volume	Reserve Tank Volume	

Table 4.2: Table for configuration parameters in OpenLab Drilling simulator

of excel sheet and then stored locally and used by the MATLAB script to obtain the pore and fracture pressure at the corresponding true vertical depth (TVD). The algorithm discussed in Figure 4.1, uses the same method to obtain the pore and fracture pressures. The absolute value of pressure is derived in terms of Pascals from the s.g. which is the unit of pressure gradient, the depth the point of calculation is also required in this calculation (Equation 4.5).

$$Pressure_{Pa} = (PressureGradient_{s.g.}*8.345)*(0.051948*BitDepth*3.28)*6894.76 \quad (4.5)$$

Where,

1 SG = 8.345 ppg, 1 ppg = 0.051948 psi/ft, 1 meter = 3.28 ft, and 1 psi = 6894.76 Pa.

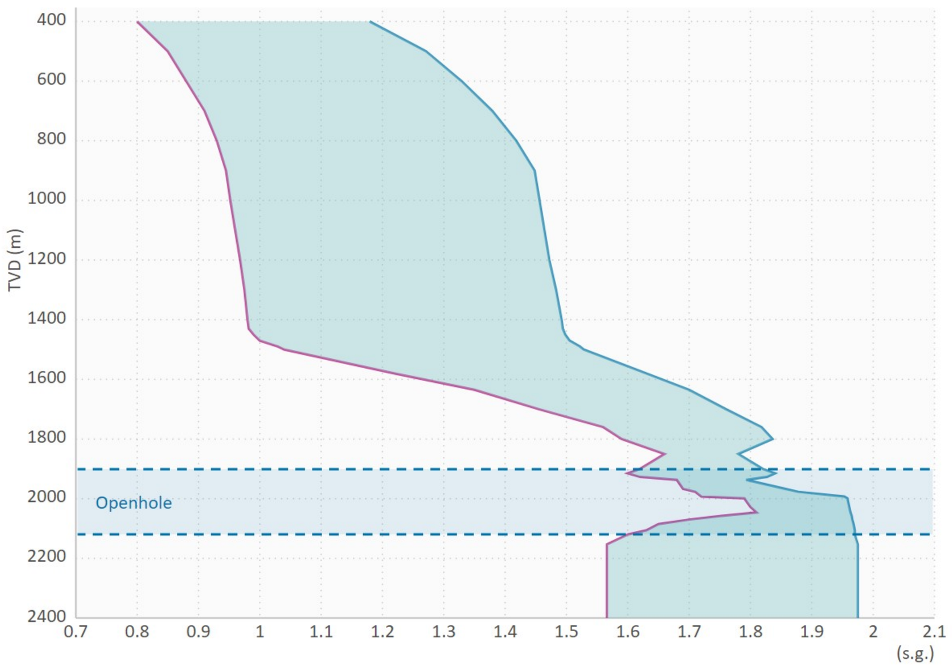


Figure 4.2: Geo-Pressures setting under Geology in OpenLab simulator web interface

Summarizing Inputs and Outputs of the Environment

Hence to summarize the inputs of the environment, it can be concluded from the details discussed in the above sections and sub-sections that environment needs *Configuration Data* (Fracture and Pore pressure), *Action from the agent* (Tuning parameters and pressure set-point) and *Stable API communication* (to record dynamic response of PID controller). The outputs at the end of the simulation are the cumulative reward and it should be noted the concept of terminal state is not discussed here because episode is ended by a maximum number of steps.

4.2 Reward Function

The reward function is used to evaluate the agent performance and therefore crucial for agent training. One approach is to evaluate the suggested bottom hole pressure (BHP) set point against the pressure window i.e. pore pressure and fracture pressure. Another approach is to, consider the oscillation of BHP in the control, it is quite obvious that both of the ideas are needed to be implement to get the best results.

4.2.1 Reward function design for Reference point decision

In this section, the penalty/reward is determined against the ideal pressure within the given pressure window. Here, a reward function is proposed by evaluating BHP against the pressure window, i.e. whether the BHP is slightly above the pore pressure and within the pressure window.

$$r(P_{BH}) = \begin{cases} p_1, & P_{BH} < P_p - \delta \\ r_1 - \frac{P_{BH}-P_p}{P_f-P_p}(r_1 - r_2), & P_p - \delta <= P_{BH} < P_f \\ p_2, & P_{BH} > P_f \end{cases} \tag{4.6}$$

where p_1 and p_2 are penalties (negative rewards) when bottom hole pressure is below pore pressure (P_p) or above fracture pressure (P_f), respectively; r_1 and r_2 are the maximum rewards approaching pore pressure and minimum reward approaching fracture pressure, respectively; δ is the tolerance of bottom hole pressure, meaning that the slight lower BHP than the pore pressure is allowed. A sample case of the reward function is shown in Fig. 4.3, where $p_1 = p_2$ for simplicity. An example of the reward function in given pore and fracture pressure is shown in Figure 4.3.

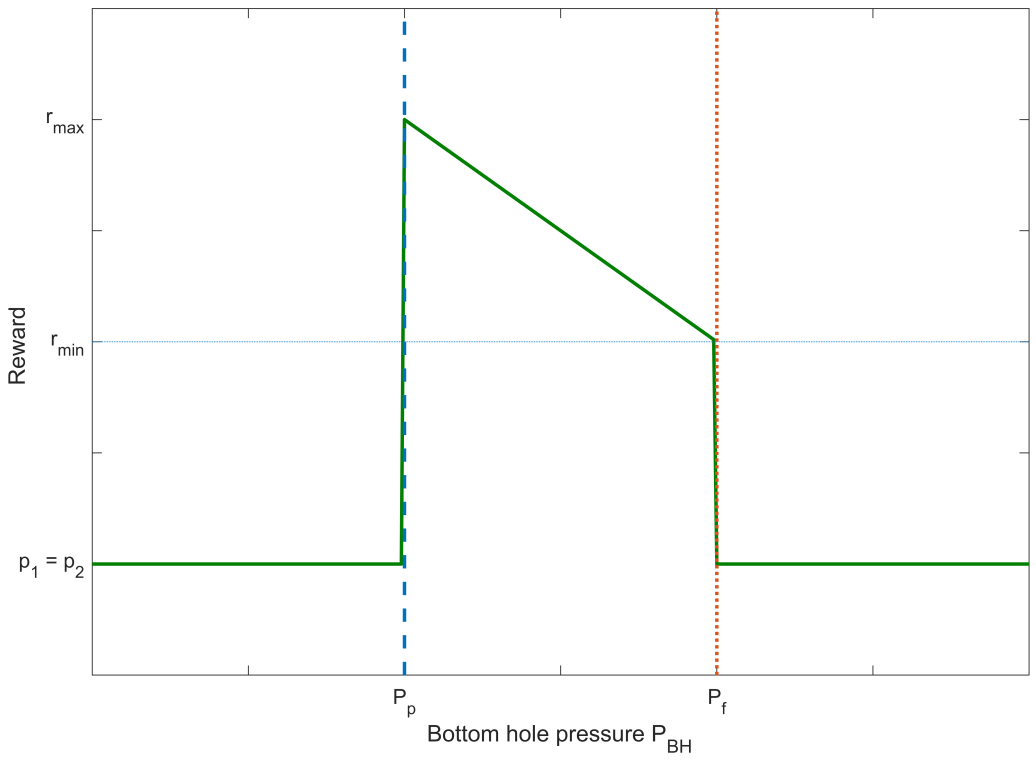


Figure 4.3: The behaviour of reward function designed for suggested set-point evaluation

4.2.2 Reward Function design for oscillations

The reward function for oscillation is based on the dynamic response of the PID controller. A dynamic response from a PID controller id shown in Figure 3.2 and in Figure 2.4 some terms such as rise time, settling time, overshoot and steady state error are discussed.

Figure 4.4 discusses the different controller behaviours under different tuning conditions. Here the goal is observe that some of the responses (in orange) are completely unstable as shown in the first plot and some of the responses are matching the set-point almost perfectly with minor mismatches. The proposed reward function tries to take into account overshoot, steady state error, settling time and rise time.

Figure 4.5 discusses the algorithm for the reward function designed to minimize the oscillations in the controller response. If the absolute difference between the set-point and bottom hole pressure is greater than 1 percent, then a step-wise negative penalty is accumulated otherwise no penalty is given. Since this procedure takes place every step this will automatically give a high negative reward to an unstable response or a response with a steady state error, and if the response reaches the set-point quicker and in a stable manner then the reward would be less negative in total.

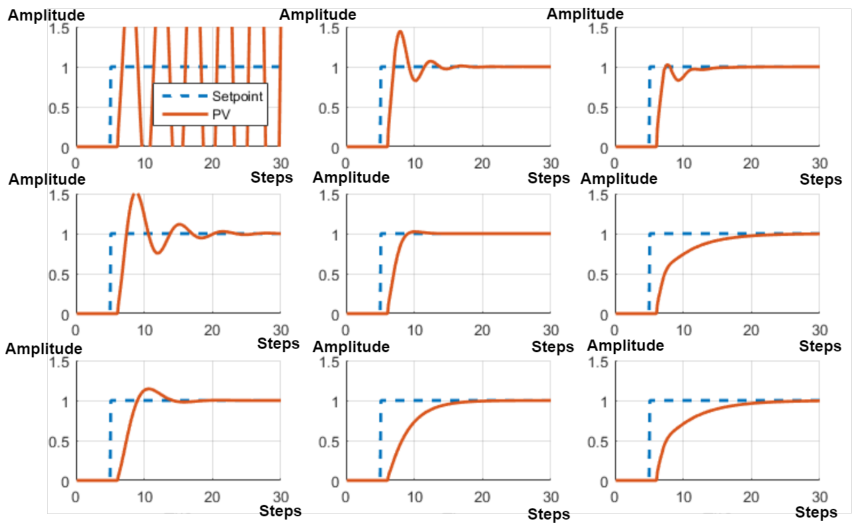


Figure 4.4: Controller behaviour under different choices of tuning parameters [13]

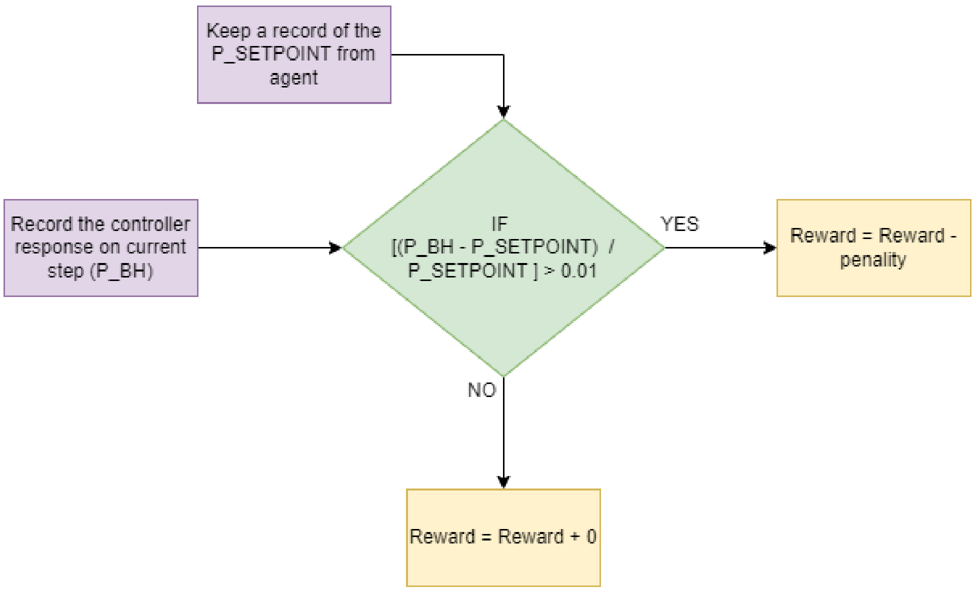

Keep a record of the P_SETPOINT from agent
Record the controller response on current step (P_BH)
IF [(P_BH - P_SETPOINT) / P_SETPOINT] > 0.01
YES
Reward = Reward - penality
NO
Reward = Reward + 0

Figure 4.5: Reward function for oscillation

These reward functions discussed above will be discussed in the implementation in the next chapter and further elaborated in the results sections as well.

Chapter 5

Reinforcement Learning Implementation

5.1 Implementation of Environment and Agents

The environment and agent development is mainly handled in MATLAB but some work is also done in python for multi-armed bandit problem which will be discussed towards the end. Lets start the discussion with the analysis of problem in terms of MDP. The implementation work will be divided into various Case Studies to make the steps taken during development easier to understand.

5.1.1 Case 1: Markov Decision Process Problem

In this case the implementation is based on the modelling done for the case of MDP problem. Figure 5.1 shows the basic layout of the implementation but various steps regarding the implementation will be discussed in the form of a block diagram later. In Figure 5.1, the communication between environment and an Actor-Critic RL agent is seen. The choice of Actor-Critic over DQN is mainly because of their fast convergence properties which is very useful when computational time is an important factor (discussed in Section 2.2.3) [10]. As discussed earlier the actor provides the action which are the PID tuning parameters and pressure set-point, then the PID controller adjusts the choke opening which in-turn affects the P_{BH}. The reward and states are fed back to the agent every step.

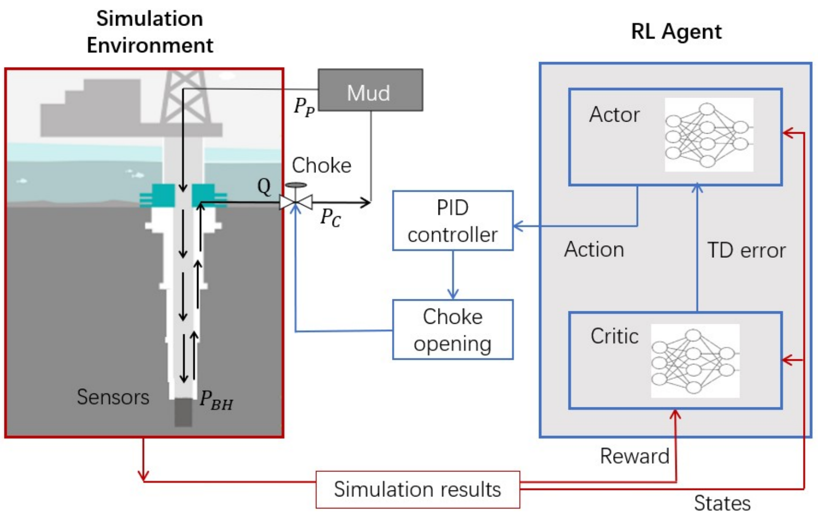


Figure 5.1: System diagram with Actor-Critic Agent

The simulation environment as discussed in last chapter is implemented in MATLAB in the form of *STEP FUNCTIONS and RESET FUNCTIONS*. These functions are responsible for making a step inside an episode and then resetting the states at the end of an episode to start-over for the next one. Figure 5.2 shows the structure of the environment in terms of step function and reset function. It can be observed that the step function is called every step (color-coded blue) where reset episode is just called once in the start of the next episode (color-coded yellow). If the total number of episodes have been reached then the simulation is stopped.

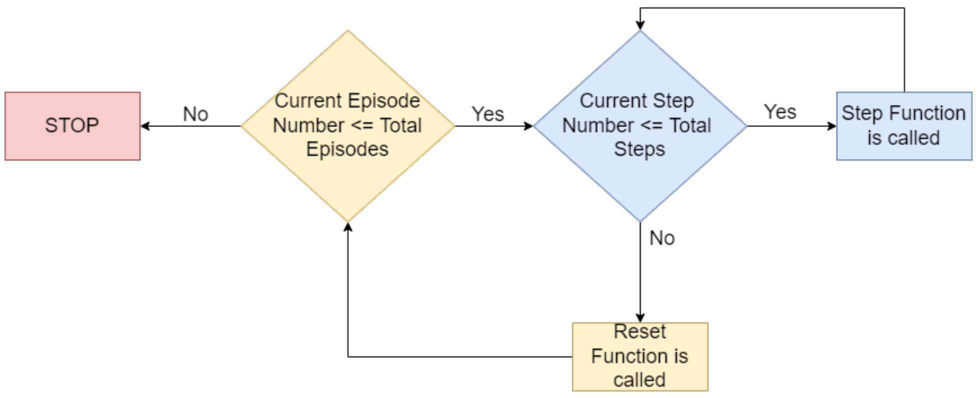

STOP
No
Current Episode Number <= Total Episodes
Yes
Current Step Number <= Total Steps
Yes
Step Function is called
No
Reset Function is called

Figure 5.2: Structuring environment with step and reset functions

First Implementation

In the step function various parameters are configured which will be discussed in this section. In this case study of MDPs various versions of the step function were created, the one discussed below has different state space and different reward function design as compared to what was discussed in Section 3.1.2 and 4.2 respectively. In Figure 5.3 the structure of step function is shown and the code is attached in Appendix A.2. and titled *STEP FUNCTION VERSION 1*.

In the light of Figure 5.3, some important points for this step function are described below:

- The step function takes in Actions which in this case are K_P and $BHP_{set-point}$. K_I is derived from K_P instead of taking as an independent input and this reduces our Action space to equation 3.4, where as the controller is now a PI controller. Here $K_I = K_P/13$, this was chosen by trying various values and this had the most stable response.
- The configuration includes initial choke opening (which is set to 0.25 initially, 0 means closed and 1 means fully open). The main pump flow rate is randomly selected between 2900 to 2800 l/min and then it is ramped down slowly to a value 700 l/min below the starting value in steps. The simulation takes in total 500 time steps to ramp reach the end value and this is considered an end of episode, the procedure of ramping it down is shown in Figure 5.5, the unit used there is m_3/s.
- The step limit automatically define a constraint on the total length of the dynamic response yielded from the controller and it is about 8 min and 20 secs.
- The geo-pressure profile is setup in the OpenLab simulator configuration and it is imported by the environment by placing the downloaded geo-pressure data from the simulator in the local directory.
- The state space in this initial implementation is based on the following variables, Relative Overshoot, Transient Time , Settling Time, Depth, Choke Opening, Flow rate out, Pressure set-point, Simulation BHP, Pore Pressure and Fracture Pressure.
- The reward for the controller dynamic response is based on Settling time, overshoot and transient time and these values are obtained from the system dynamics toolbox of MAT-

LAB. If the response is not stable towards the end then a very large negative reward is incurred.

- Is Done is a flag used to indicate the completion of episode. The block which occur once per episode are coded yellow and they ones which occur every step are coded as orange in the block diagram.

- This training takes place at the depth of 2046 meters where is window is the most narrow and the depth is not changed through out training.

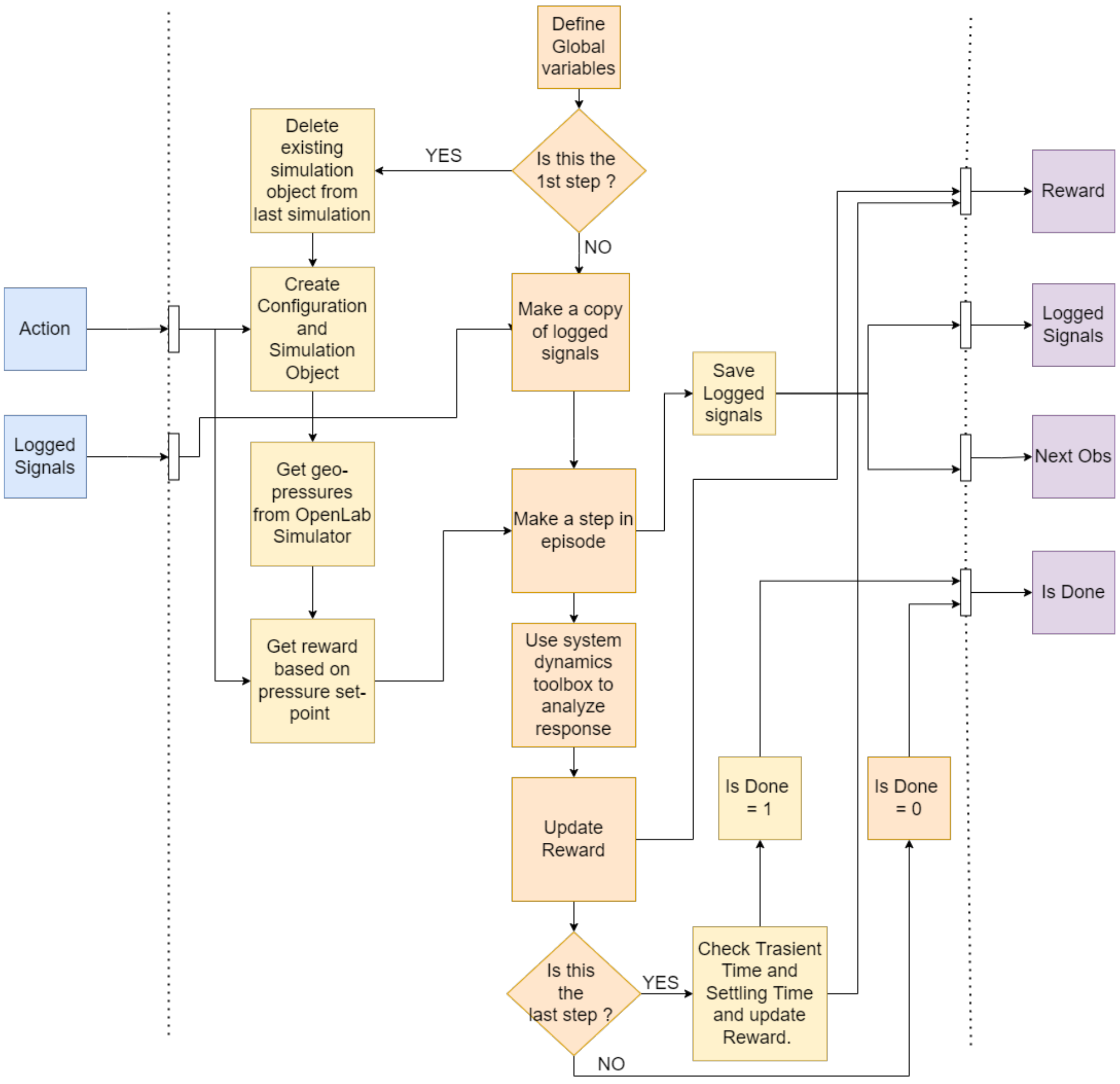

Define Global variables
Is this the 1st step ?
YES
NO
Delete existing simulation object from last simulation
Create Configuration and Simulation Object
Get geo-pressures from OpenLab Simulator
Get reward based on pressure set-point
Action
Logged Signals
Make a copy of logged signals
Make a step in episode
Use system dynamics toolbox to analyze response
Update Reward
Is this the last step ?
YES
NO
Check Trasient Time and Settling Time and update Reward.
Save Logged signals
Is Done = 1
Is Done = 0
Reward
Logged Signals
Next Obs
Is Done

Figure 5.3: Block Diagram for the first version of Step Function (Appendix A.2)

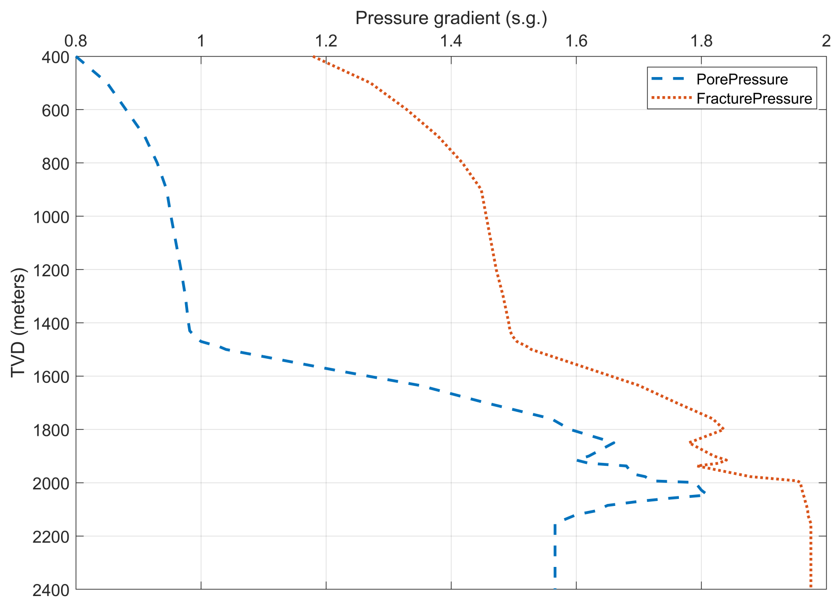


Figure 5.4: The formation Geo-Pressures along True Vertical Depth

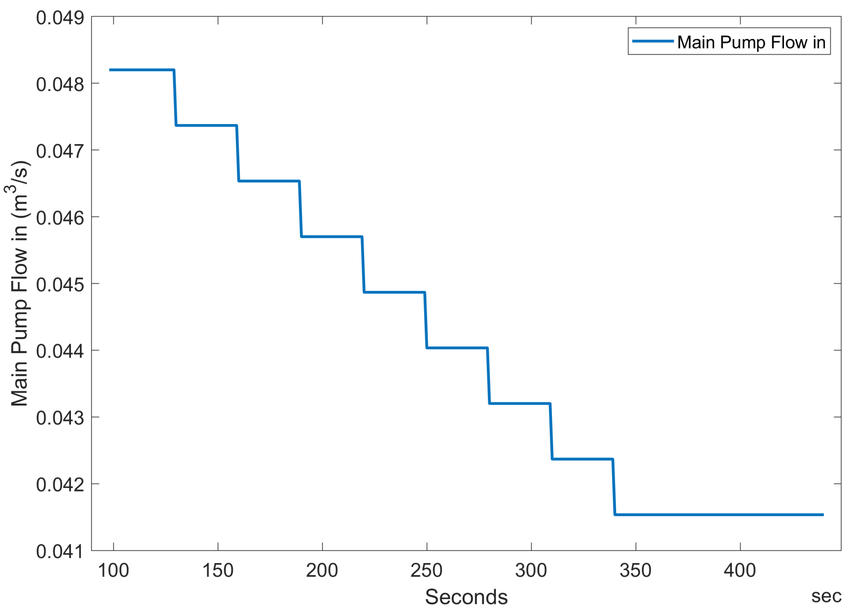


Figure 5.5: Main Pump flow rate ramped down, starts after 100 secs

MODIFICATIONS TO FIRST IMPLEMENTATION

The main problem with the above mentioned technique is the large state space and some variables are not really critical to the decision the agent needs to make where as some are not computed as accurately as hoped. The use of system dynamics toolbox to calculate overshoot, transient time and settling time at every time step, as the new value of BHP comes in from the simulator is not very stable and the values become Nan even when the response is good. To solve this problem the reward function approach discussed in the Section 4.2.2 was adopted and implemented in the next version of Step function.

Larger state space leads to more computational time [46] and since the decision was being made at one depth (2046m), as the depth was not varied (only the main pump flow rate was varied), many parameters don't change or their change does not matter. Given these reasons the state space is reduced to what was discussed in section 3.1.2 where state space included Depth of the bit, pore pressure, fracture pressure and bottom bole pressure. Although it is known that geo-pressure at a constant depth are going to stay constant but their values define the reward for the Pressure set-point and the bottom hole pressure is required to be recorded to make a dynamic response of the controller.

Another observation made on this way of implementing the environment is that in-between steps there is a delay due to the computations done by the agent and it makes the simulation in the web simulator wait which sometimes lead to unstable behaviour and the simulation crashes which leads to additional waste of time although this behaviour is handled using try and except structure in the code.

Apart from these changes we can now reduce the environment to one state MDP because all the states are constant in the problem except bottom hole pressure. For bottom hole pressure, it can be recorded in an inner loop where the whole simulation just runs automatically and and the last value can be shared with the agent at the end of episode. This way of implementing the solution was found to be very stable and effective and it is discussed in the next subsection.

ONE STATE MDP

In the above section we discussed the modifications we intend to make to the first implementation. Figure 5.6, incorporates these changes and the codes including these changes are included in the appendix A.4 to A.7. In contrast to Figure 5.3 the states are now only shared with the agent at the end of whole simulation which makes the problem one state MDP.

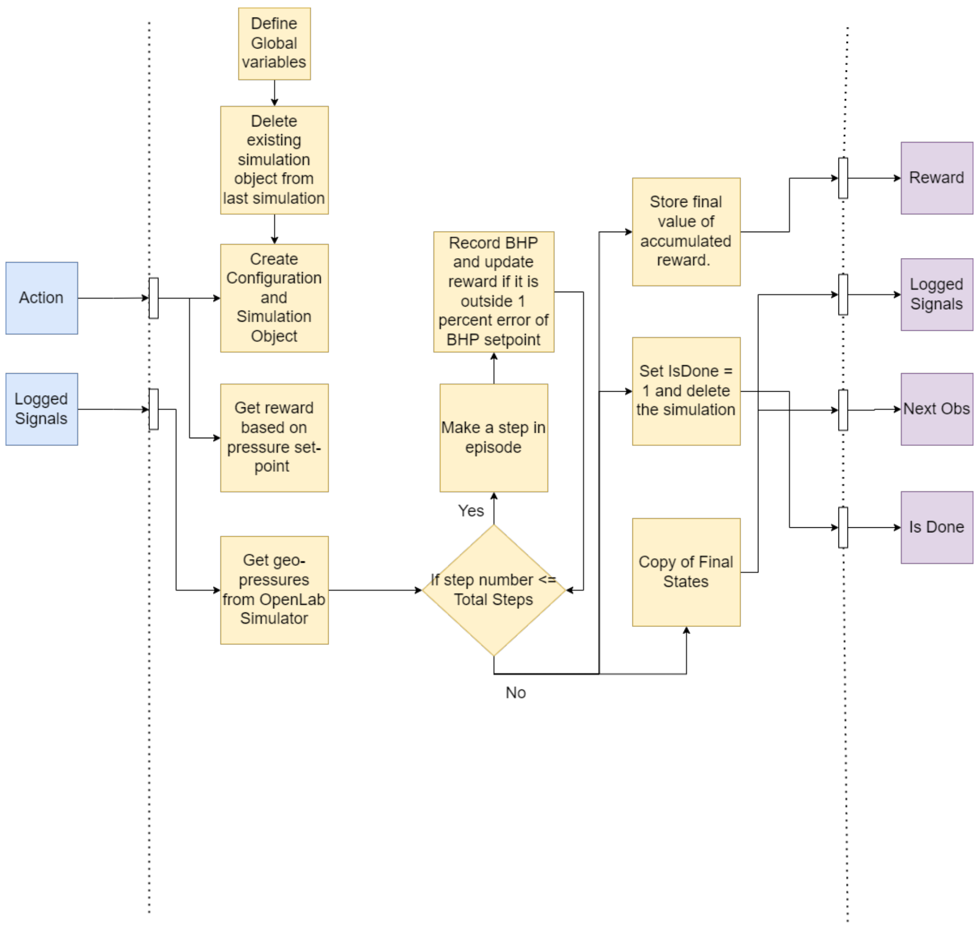

Define Global variables
Delete existing simulation object from last simulation
Create Configuration and Simulation Object
Get reward based on pressure set-point
Get geo-pressures from OpenLab Simulator
Action
Logged Signals
Record BHP and update reward if it is outside 1 percent error of BHP setpoint
Make a step in episode
Yes
If step number <= Total Steps
No
Store final value of accumulated reward.
Set IsDone = 1 and delete the simulation
Copy of Final States
Reward
Logged Signals
Next Obs
Is Done

Figure 5.6: Block Diagram for final version of Step function (For one episode - Appendix A.5)

The reason behind making this problem a one state MDP is that the rewards are only directly dependent on the actions, when making the decision at one depth. The configurations in both implementations are kept the same and the results will be shared in the next chapter. The only difference in this implementation can be seen in Appendix A.4 that the initial choke opening is changed to 1 as the valve is always fully opened when the simulation starts. Apart from this the simulation turns out to much more stable and faced almost no crashes as compared to the previous implementation.

PPO Agent in MATLAB

The RL agent used for these implementations is called Proximal policy optimization (PPO) agent. "PPO is a model-free, online, on-policy, policy gradient RL method. This algorithm is a type of policy gradient training that alternates between sampling data through environmental interaction and optimizing a clipped surrogate objective function using stochastic gradient descent".[14] The agent is implemented using MATLAB Reinforcement Learning toolbox. [15] [16].

The PPO agent in this implementation is trained on an environment with discrete observation and action spaces. During training the agent estimates the probability of taking each action in the action space and randomly selects actions from the discrete action space, furthermore it interacts with the training environment for multiple time steps using current policy before updating actor and critic properties after mini batches. The policy $\pi(S)$ and value function V(S) are estimated using two function approximators [14],

- "Actor $\pi(A|S;\theta)$ — The actor, with parameters θ, outputs the conditional probability of taking each action A when in state S." [14]

- "Critic $V(S;\phi)$ — The critic, with parameters ϕ, takes observation S and returns the corresponding expectation of the discounted long-term reward." [14]

  The detailed training algorithm is shown in the Figure 5.7 below, the psuedocode is discussed on the official Mathworks website for PPO implementation using RL Toolbox as well [14].

Constraints on Actions

The values for actions are constrained under realistic limits. The pressure set-point suggestion mentioned as P_{ref} in equation 3.4 is limited between pore and fracture pressure K_P is bounded by the values which are applicable if either P_{ref} is equal to pore pressure or fracture pressure as it controls whether the controller will be able to reach the pressure set-point at the end of the simulation. The range for K_P is set according to the narrowest window and the results will

```
1. Initialize Actor $\pi(A|S;\theta)$ and Critic $V(S;\phi)$, with random parameters $\theta$ and $\phi$ respectively.
2. Follow the current policy and generate an experience sequence.
        $S_{ts}, A_{ts}, R_{ts+1}, S_{ts+1}, \ldots, S_{ts+N-1}, A_{ts+N-1}, R_{ts+N}, S_{ts+N}$
   Here,
   $S_t$ is a state observation. $A_t$ is an action taken from that state. $S_{t+1}$ is the next state. $R_{t+1}$ is the reward
   received for moving from $S_t$ to $S_{t+1}$. $ts$ is the starting time step of the current set of $N$ experiences.
   $S_N$ is the terminal state.
3. For each episode step $t = ts+1, ts+2, \ldots, ts+N$, the return and advantage function are calculated using
   generalized advantage estimator. We compute advantage function $D_t$, which is the discounted sum of
   temporal difference errors.
        $D_t = \sum_{k=t}^{ts+N-1} (\gamma\lambda)^{k-t} \delta_k$
        $\delta_k = R_t + b\gamma\ V(S_t;\phi)$
        $G_t = D_t + V(S_t;\phi)$
   Here,
   $D_t$ is the advantage function. b is 0 if $S_{ts+N}$ is a terminal state and 1 otherwise. $\lambda$ is a smoothing
   factor. $\gamma$ is the discount factor. $ts$ is the starting time step of the current set of $N$ experiences. t =
   episode step, ts+1 , … , ts + N. N = Total number of states. $G_t$ is the return.
4. Learn from mini batches of experiences over $K$ epochs.
    a. Sample a random mini-batch data set of size $M$ from the current set of experiences. Each
       element of the mini-batch data set contains a current experience and the corresponding return
       and advantage function values.
    b. Update the critic parameters ($\phi$) and actor parameters ($\theta$) by minimizing loss functions designed
       for critic and actor respectively. The loss functions are described below.

        $L_{critic}(\phi) = (1/M) \sum_{i=1}^{M} (G_i - V(S_i;\phi))^2$
        $L_{actor}(\theta) = \left(\frac{1}{M}\right) \sum_{i=1}^{M} (-\min(\mathrm{ri}(\theta).\mathrm{Di}, \mathrm{ci}(\theta).\mathrm{Di}) + w\mathcal{H}\mathrm{i}(\theta, \mathrm{Si}))$
        $r_i(\theta) = \frac{\pi(A_i|S_i;\theta)}{\pi(A_i|S_i;\theta_{old})}$
        $c_i(\theta) = \max(\min(r_i(\theta), 1+\varepsilon), 1-\varepsilon)$
   Here,
   $D_i$ is the advantage function of i-th element of mini batch. $G_i$ is the return value of i-th element of mini
   batch.
   $\pi(A_i|S_i;\theta)$ is the probability of taking action $A_i$ when in state $S_i$, given the updated policy parameters $\theta$.
   $\pi(A_i|S_i;\theta_{old})$ is the probability of taking action $A_i$ when in state $S_i$, given the previous policy
   parameters $\theta_{old}$ from before the current learning epoch. $\varepsilon$ is the clip factor. $\mathcal{H}_i(\theta)$ is the entropy loss. $w$ is
   the entropy loss weight factor.
5. Repeat steps 2 through 4 until the training episode reaches a terminal state.
```

Figure 5.7: Psuedocode of PPO Implementation used in this study [14][15] [16]

be discussed in the next section. Figure 5.8, illustrates the set-point range for the current implementation, but the depth is obviously 2046m where the operational window is the narrower. The implementation of these constraints are handled in the code snippets discussed in appendix A.4 and A.7.

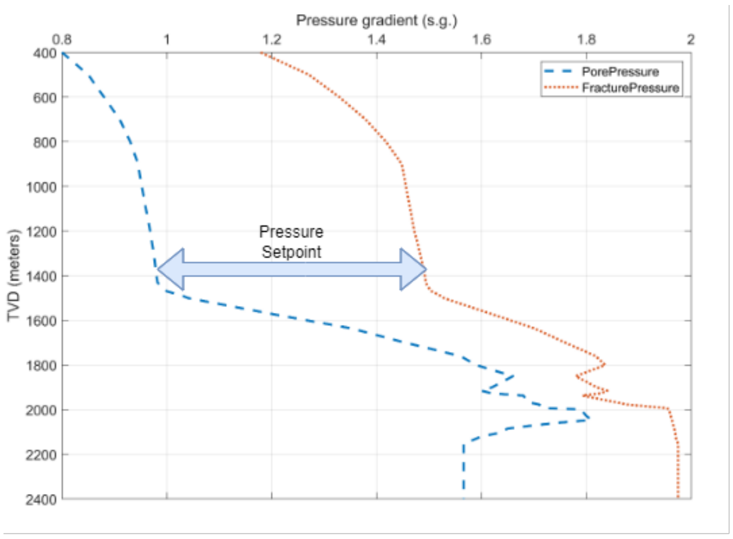


Figure 5.8: Range of pressure set-point suggestion

5.1.2 Limitations of MDP framework for this study

In this section the journey of implementing the environment was discussed and the reasons about the shift from one implementation to another were also discussed. The major problem to overcome in this study is the computational time which will discussed in the next section. Apart from this a major limitation of these implementations have been that they were implemented on a single depth scenario which is not very practical, and even at a single depth scenario the agent took more than 10 hours to train it self (discussed in the next chapter as well).

5.2 Implementation of Multi-Armed Bandit Problem

In the last section MDP framework was discussed and it was eventually reduced to one-state MDP which is analogous to Multi-Armed Bandit problem where the value function is only dependent on action (see equation 2.15). The multi-armed bandit problem is used to device a plan where controller can be tuned at certain intervals of depth. There are two phases of this implementation, in one phase the data is collected from the OpenLab simulator to create an environment suitable for implementation of multi-armed bandit algorithm and in the next phase epsilon-greedy approach is used to find a suitable trade-off between exploration and exploitation and then after this hyper-parameter tuning the epsilon-greedy agent is employed to find the best arm (or action). The notion of best arm means the action which carries the most reward. The reward function definition are same as the ones defined in Section 4.2. Figure 5.9 shows the block diagram of the first phase where data is collected from running the simulation on OpenLab simulator.

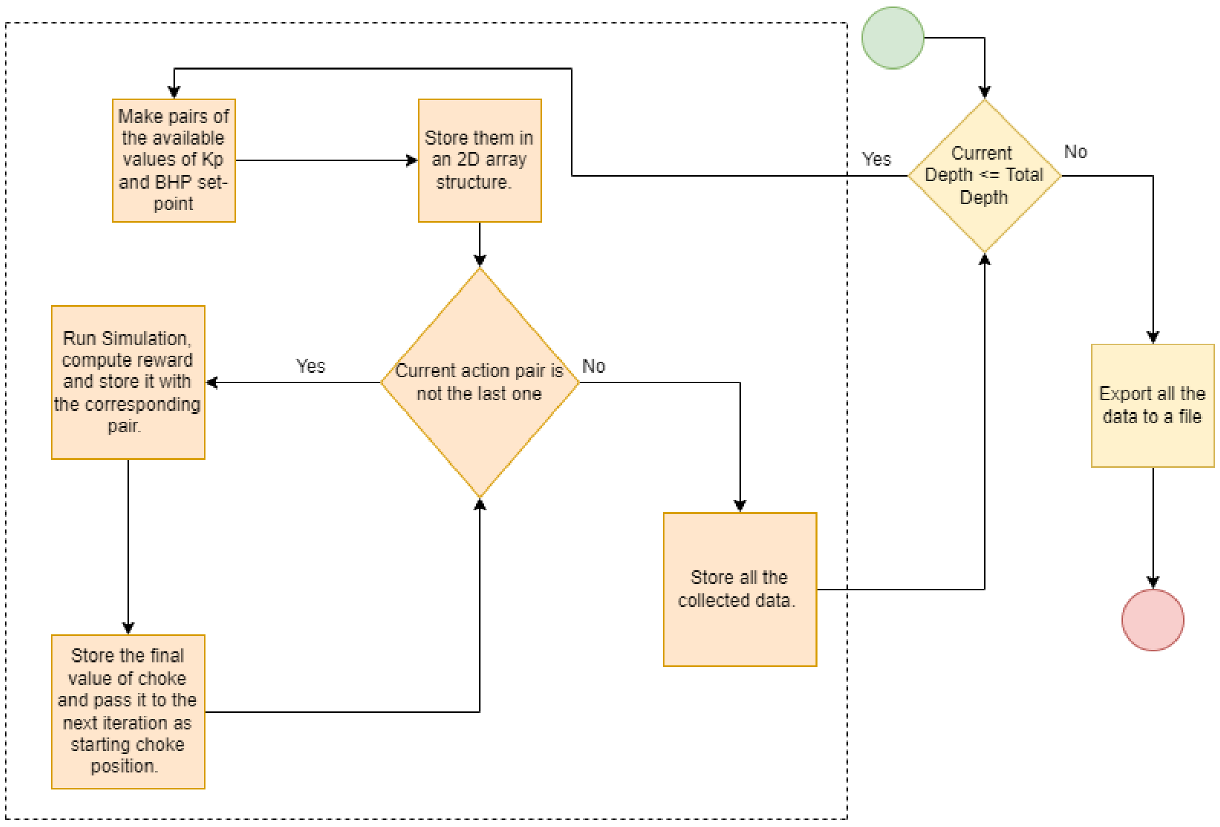
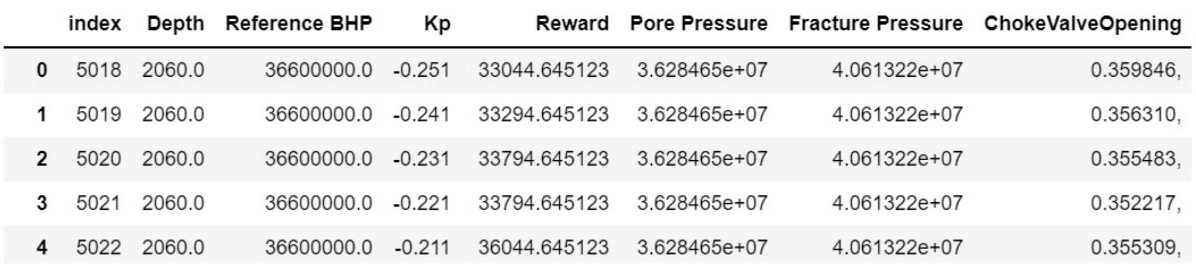

Make pairs of the available values of Kp and BHP set-point
Store them in an 2D array structure.
Yes
Current Depth <= Total Depth
No
Run Simulation, compute reward and store it with the corresponding pair.
Yes
Current action pair is not the last one
No
Export all the data to a file
Store all the collected data.
Store the final value of choke and pass it to the next iteration as starting choke position.

Figure 5.9: Block diagram for data collection Appendix A.10

The configuration in Figure 5.9 is kept same as the last PPO implementation the only difference being that the choke position at the end of each complete simulation is fed to the next simulation as the starting choke position. The resulting data which comes out at the end of looks like the snapshot in Figure 5.10. This figure is for a specific depth and a similar table is generated for all the selected depths where the decision for for set point and tuning parameters is needed to be made.

The next step after data acquisition is to start developing the epsilon greedy approach. Figure



	index	Depth	Reference BHP	Kp	Reward	Pore Pressure	Fracture Pressure	ChokeValveOpening
0	5018	2060.0	36600000.0	-0.251	33044.645123	3.628465e+07	4.061322e+07	0.359846,
1	5019	2060.0	36600000.0	-0.241	33294.645123	3.628465e+07	4.061322e+07	0.356310,
2	5020	2060.0	36600000.0	-0.231	33794.645123	3.628465e+07	4.061322e+07	0.355483,
3	5021	2060.0	36600000.0	-0.221	33794.645123	3.628465e+07	4.061322e+07	0.352217,
4	5022	2060.0	36600000.0	-0.211	36044.645123	3.628465e+07	4.061322e+07	0.355309,

Figure 5.10: Structure of the colected data

5.11 shows the approach taken in for implementing the epsilon greedy method. The idea is to select action greedily in-general i.e. the actions which have the best estimated reward but at some time steps another action may be selected instead randomly out of all possible actions, the random action is chosen with a probability of ϵ (epsilon). Hence epsilon controls the *exploration* and saves the algorithm from going into a completely *exploitation* approach (also discussed in Section 2.2.3).

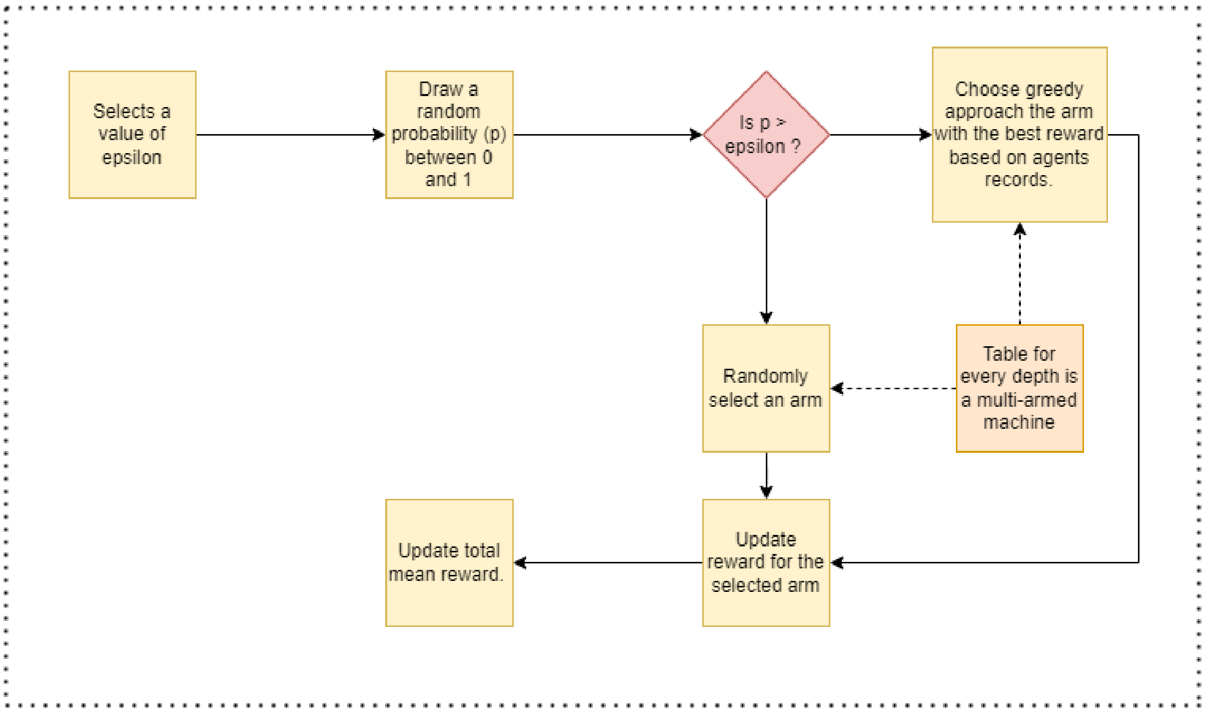

Selects a value of epsilon
Draw a random probability (p) between 0 and 1
Is p > epsilon ?
Choose greedy approach the arm with the best reward based on agents records.
Randomly select an arm
Table for every depth is a multi-armed machine
Update total mean reward.
Update reward for the selected arm

Figure 5.11: Simplified epsilon greedy approach

The overall goal is to come-up with a value of epsilon which gives maximum reward over a number of simulation steps, and then that value of epsilon is tested on various depths and if it suits them well then it the similar technique is applied on the whole depth section. The idea of a discrete Geo-pressure profile was discussed in Figure 3.3, and a similar idea is used here (in Figure 5.12) where the depth interval is kept to be *10m*, The geo-pressure profile is made a little more narrower here as compared to one for the PPO implementation just to make the problem a little tougher for agent. Although the starting depth is sometimes varied to save up computational time during testing. The mechanism discussed in Figure 5.12 is applied at every depth point in Figure 5.13. To summarise the idea of implementation it can be said that, *every depth corresponds to a casino machine and every pair of Kp and pressure set-point corresponds to an arm*. This makes our problem simpler to understand and reward distributions of every arm are handled by the reward functions we defined in the section 4.2.

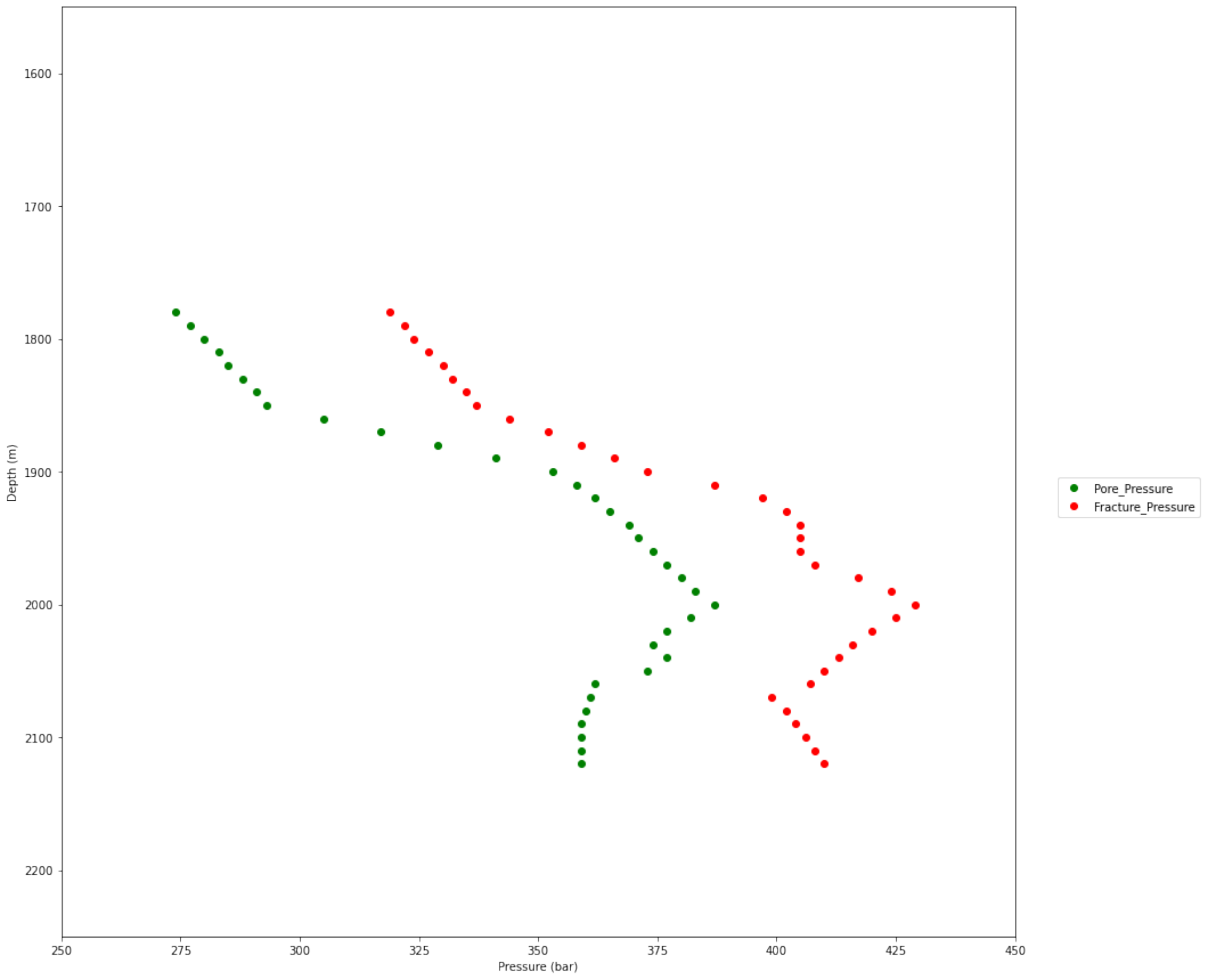


Figure 5.12: Discrete depth intervals with a step of 10m on a selected depth section.

Various values for ϵ are tested but the chosen one is 0.1, which shows that this approach automatically better than the completely greedy approach which means after testing we add value to our initial estimates of the rewards. Figure 5.13 shows the result of various ϵ used in testing at a random depths.

After deciding upon the appropriate value of epsilon the goal is to find the best arm, after

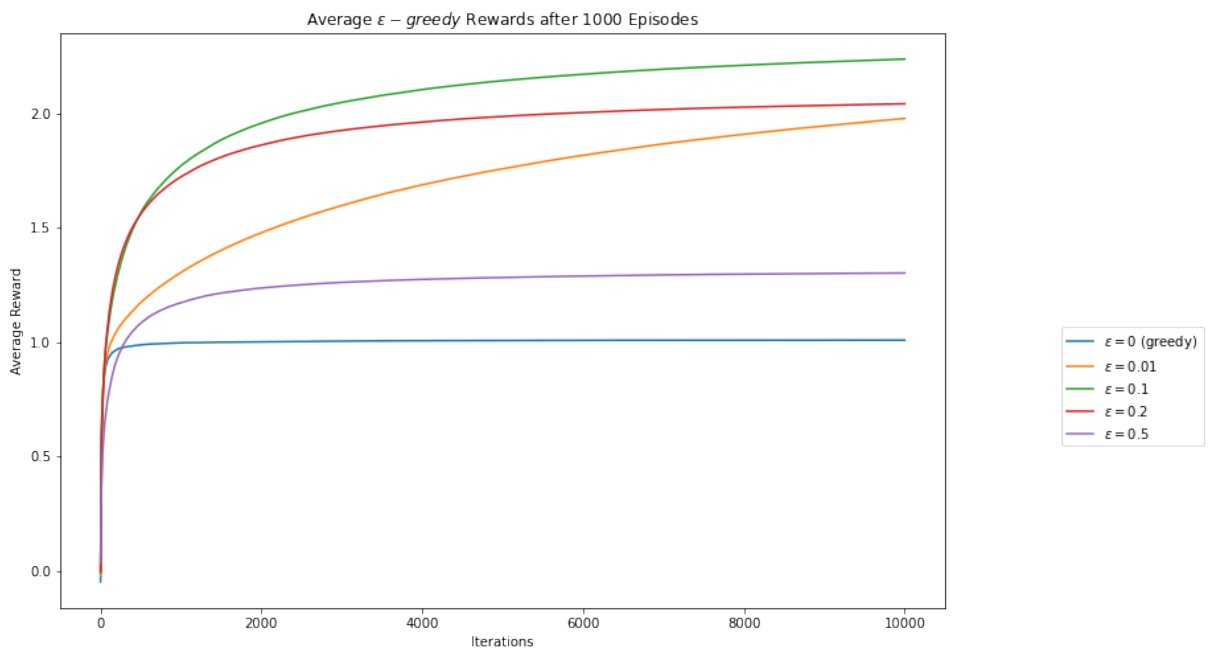


Figure 5.13: Testing various values of epsilon and choosing the one with the maximum long-term rewards.

running a test for a good number of episodes. The method is simple as the arm with the highest reward will be the one which is pulled the most as value of epsilon is low. Figure 5.14 shows the results of the arms selections for various values of epsilon and for $\epsilon = 0.1$, the optimal arm in this plot is around arm indexed at 46, which can be easily identified in code.

Now if a recap is given, then the best arm represented the best action pair, which was the value of K_P, $Pressure_{setpoint}$ and the definition of best is based on the reward functions which are designed to keep the pressure set-point between the pore and fracture pressure as well as to limit the oscillation, decrease rise time and settling time. This means that the best action for a specific depth has been found and this method can be applied to all the depths, the results of this application and former implementations will be discussed in the next chapter. The code for this implementation is attached in Appendix A.11.

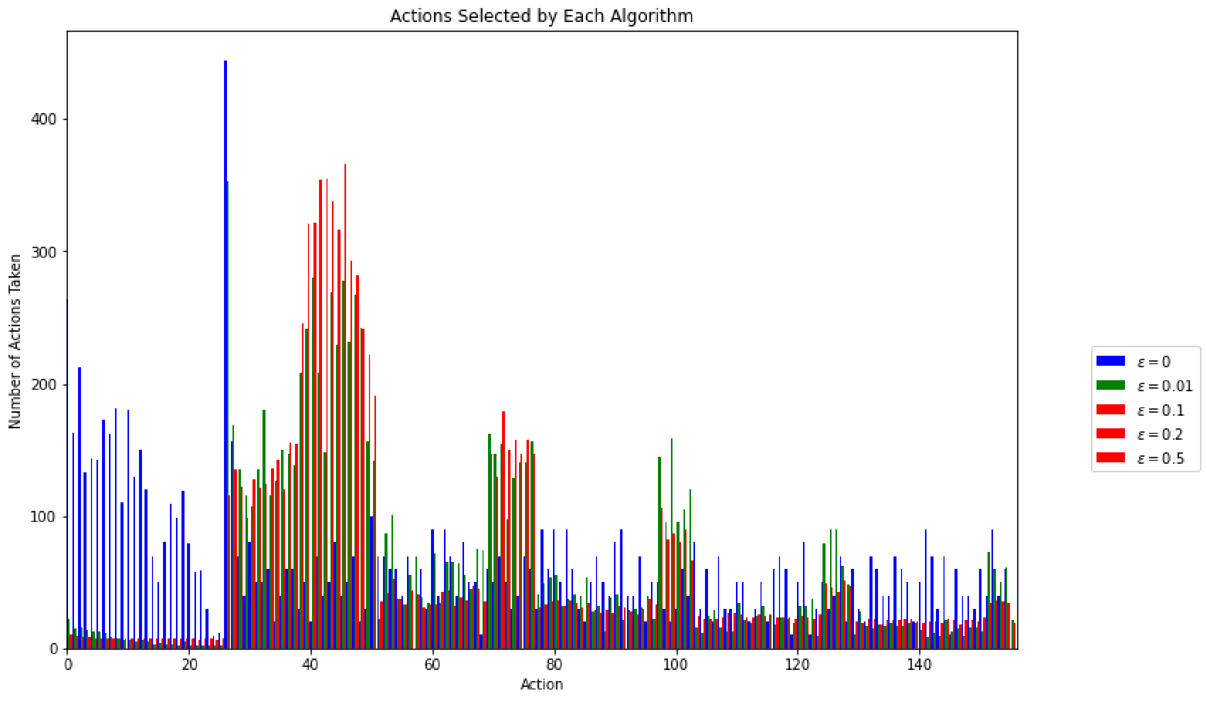


Figure 5.14: Arms selected by each epsilon selection.

Chapter 6

Results and discussions

6.1 Results from PPO Implementation

In the implementation of PPO, the reference bottom hole pressure value suggested by the agent is *370 bar* at the depth of *2046 meter* (See Figure 5.4). The agent also suggests an initial value for the proportional coefficient (Kp) of the PI controller. The *value for Kp suggested by the RL agent is -0.055 for the given depth*, and *as mentioned in Section 5.1.1 a good value Ki is assumed to be Kp / 13 based on manual tuning*.

For the reward function from section 4.2.1, we set p1 = p2 = -10^5, r1 = 10^5, and r2 = 0. The training results after 1000 episodes are shown in Fig. 6.1, in which it can be observed that the variance in the episodic reward (light blue) decreases as the plot progresses towards the right. The dark blue plot is the result of episodic reward passing through an averaging window filter with window size of 5. Most importantly, the reward stabilizes on a positive value, which means that a good estimate for reference pressure PREF was made, and the controller was tuned to a good KP , to reach that reference BHP.

The simulated bottom hole pressure response is shown in Fig. 6.2 where it can be seen that BHP is rapidly stabilized around the set-point pressure PREF (i.e. 370 bar in this scenario), and that the overshoot is less than 5 bar, which is considered to be acceptable when considering the short time of this overshoot. The controller action is turned on after 100 initial time steps and

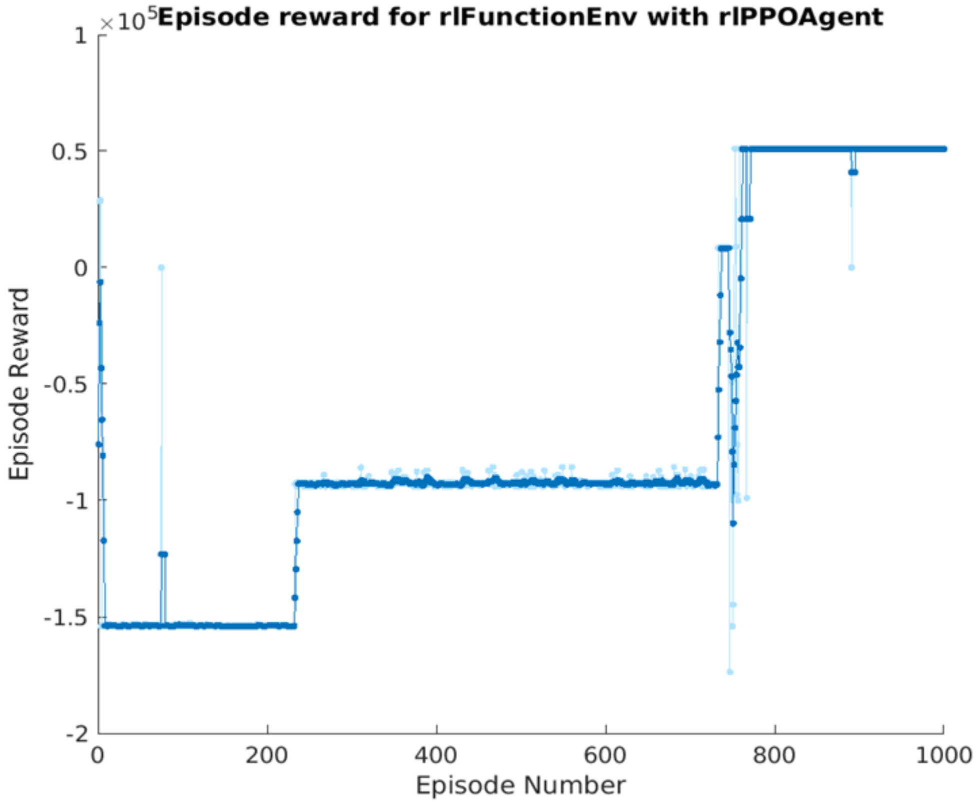


Figure 6.1: Plot of training episodes, reward per episode is visualzied

has a good rise time and settling time.

The choke valve opening is visualized in Fig. 6.3, and after 100 time steps when the controller starts acting to achieve the reference bottom hole pressure, it can be seen the response is swift and it settles around 0.2 percent.

During the training processes, various learning rates for actor and critic were tested, and the optimal results were achieved using a learning rate of 0.01 for both actor and critic. The batch size used to produce the results was 128 and the clip factor was 0.2 with number of epochs equal to 3. The details of the selected hyper parameters are shown in Figure 6.4.

At the end of the simulation the agent is also tested several times to observe its stability and this is visualized in Figure 6.5 where the agent always settles on the same policy.

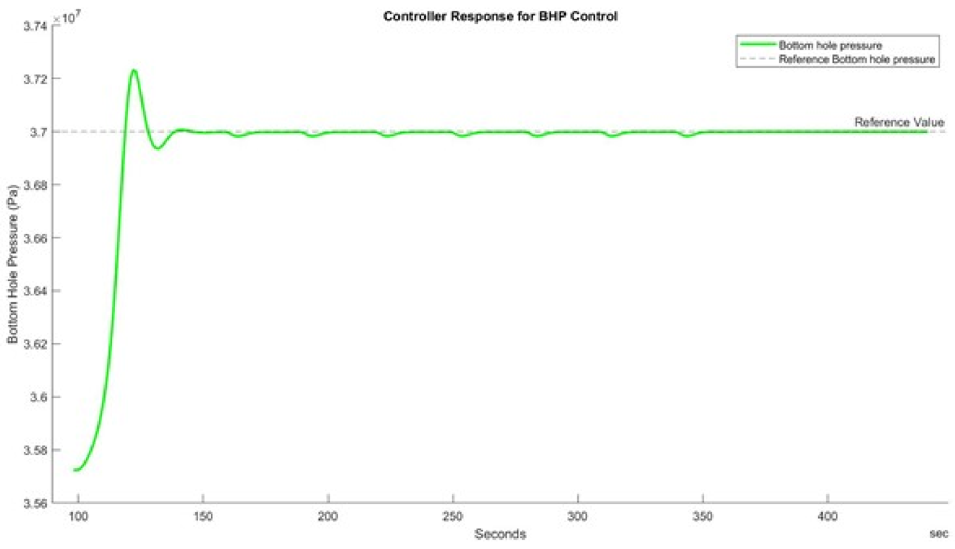


Figure 6.2: Bottom Hole Pressure response recorded in OpenLab simulator

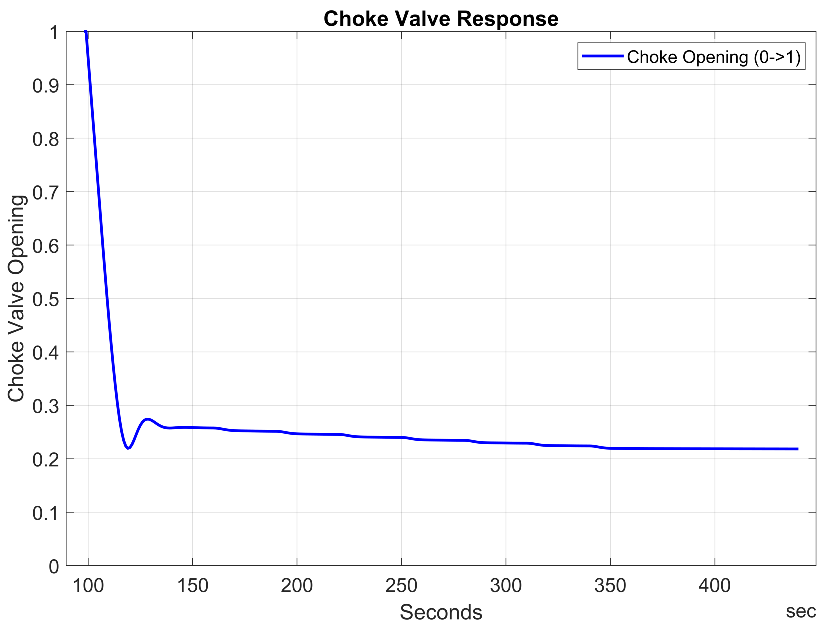
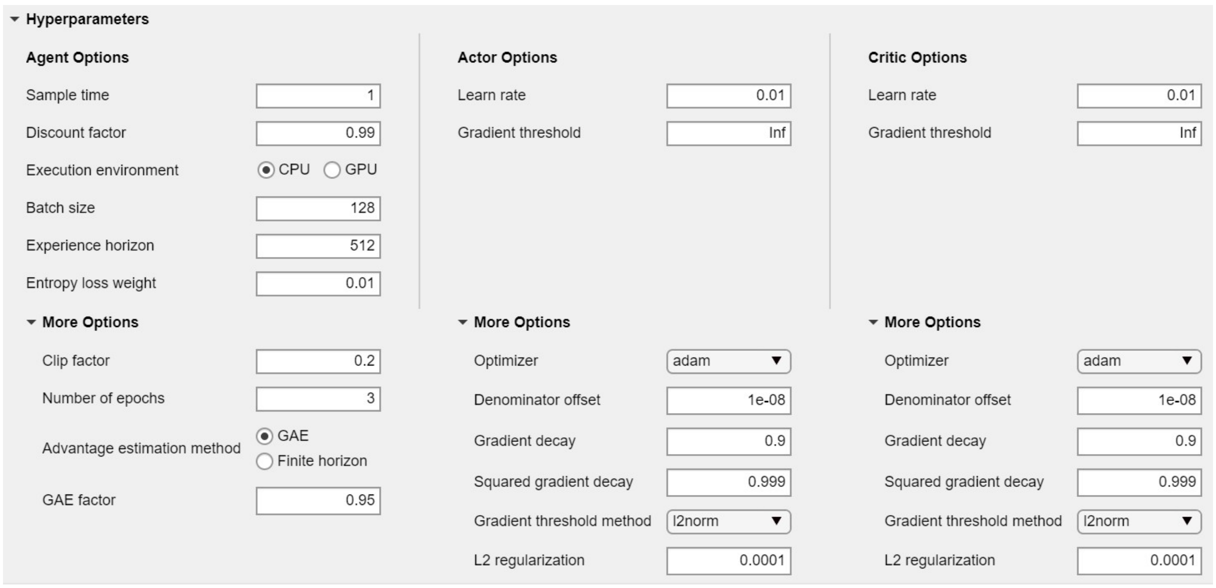


Figure 6.3: Response for choke opening



Hyperparameters

Agent Options		Actor Options		Critic Options	
Sample time	1	Learn rate	0.01	Learn rate	0.01
Discount factor	0.99	Gradient threshold	Inf	Gradient threshold	Inf
Execution environment	[✓ CPU] GPU				
Batch size	128				
Experience horizon	512				
Entropy loss weight	0.01				
More Options		**More Options**		**More Options**	
Clip factor	0.2	Optimizer	adam	Optimizer	adam
Number of epochs	3	Denominator offset	1e-08	Denominator offset	1e-08
Advantage estimation method	[✓ GAE] Finite horizon	Gradient decay	0.9	Gradient decay	0.9
		Squared gradient decay	0.999	Squared gradient decay	0.999
GAE factor	0.95	Gradient threshold method	l2norm	Gradient threshold method	l2norm
		L2 regularization	0.0001	L2 regularization	0.0001

Figure 6.4: Hyper-parameter selection during the training process (Snapshot from Mathworks RL Toolbox)

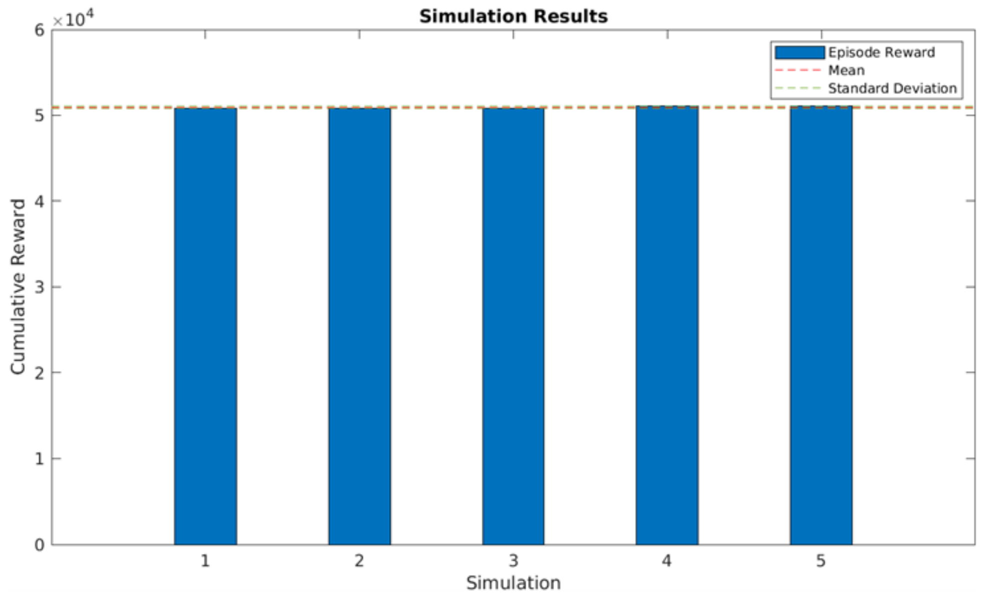


Figure 6.5: Testing the agent on the same simulation environment to observe stability

6.2 Results from $\epsilon - greedy$ method

The epsilon greedy algorithm is applied to a wide section of depth from 1780m to 2120m. This section covers the most challenging areas in the geo-pressure profile. It can be observed that given our design of reward function the technique performs perfectly, and stays very close to the pore pressure (a cushion of 4 bar) is selected between the absolute pore pressure and the limit for the set-point suggestion and this is always respected by the agent. An interesting phenomenon happens at the depths of 1880m and 1890m where it can be seen that the set-point suggestion has gone closer to the fracture pressure but it still stays with-in the pressure window. This behaviour will be discussed in this section at a later stage.

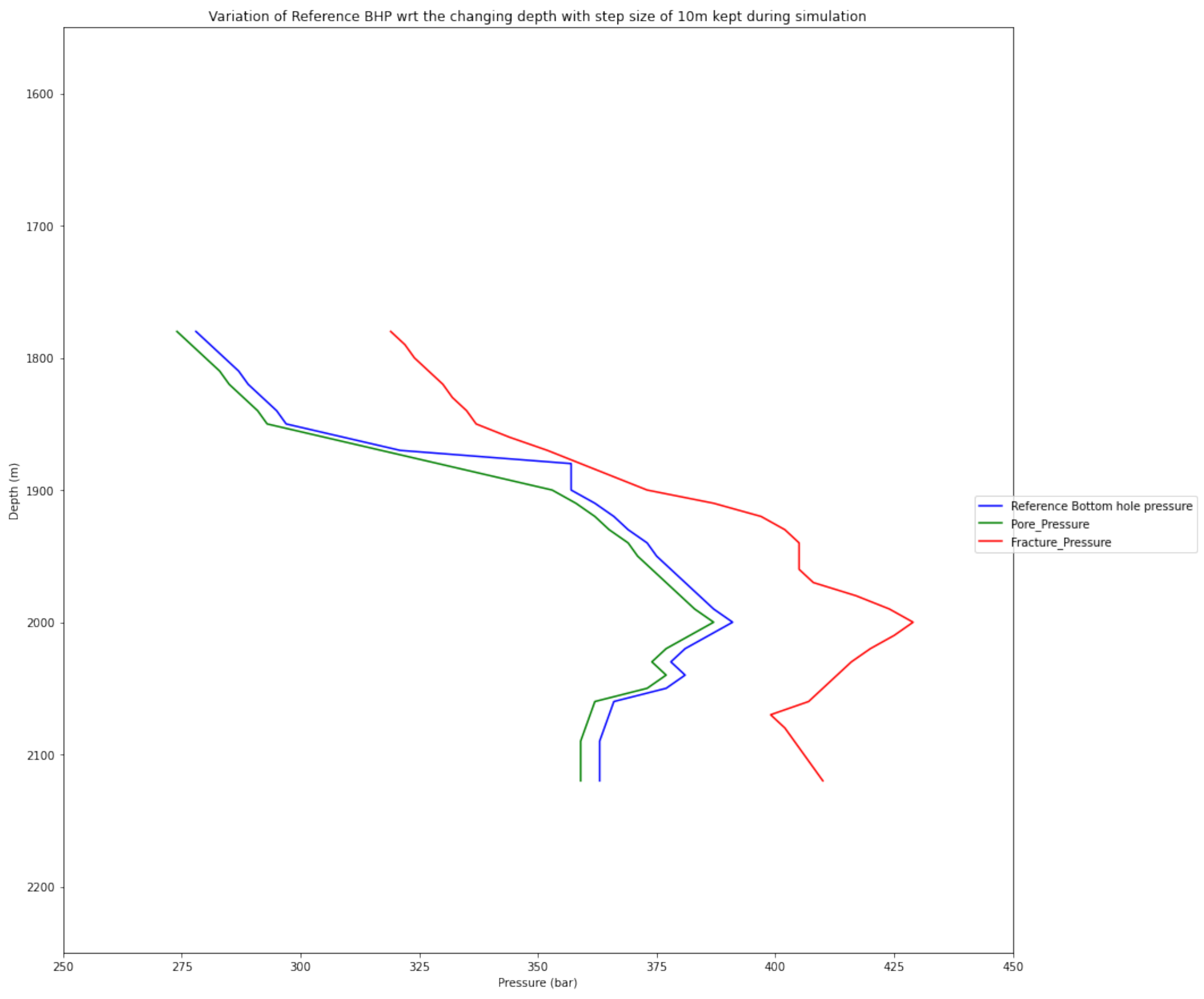


Figure 6.6: Variations in the set-point suggestion using $\epsilon - greedy$ method, Pore pressure (red), Fracture pressure (green) and set-point suggested (blue)

It is interesting to visualize the Kp and Choke opening in parallel as shown in Figure 6.7. It can be seen that Kp and Choke opening follow each others behaviour most of the time and the less negative Kp becomes the less the choke valve opening is opened. The reason for having having a negative Kp is because of the models defined in the backend of OpenLab simulator.

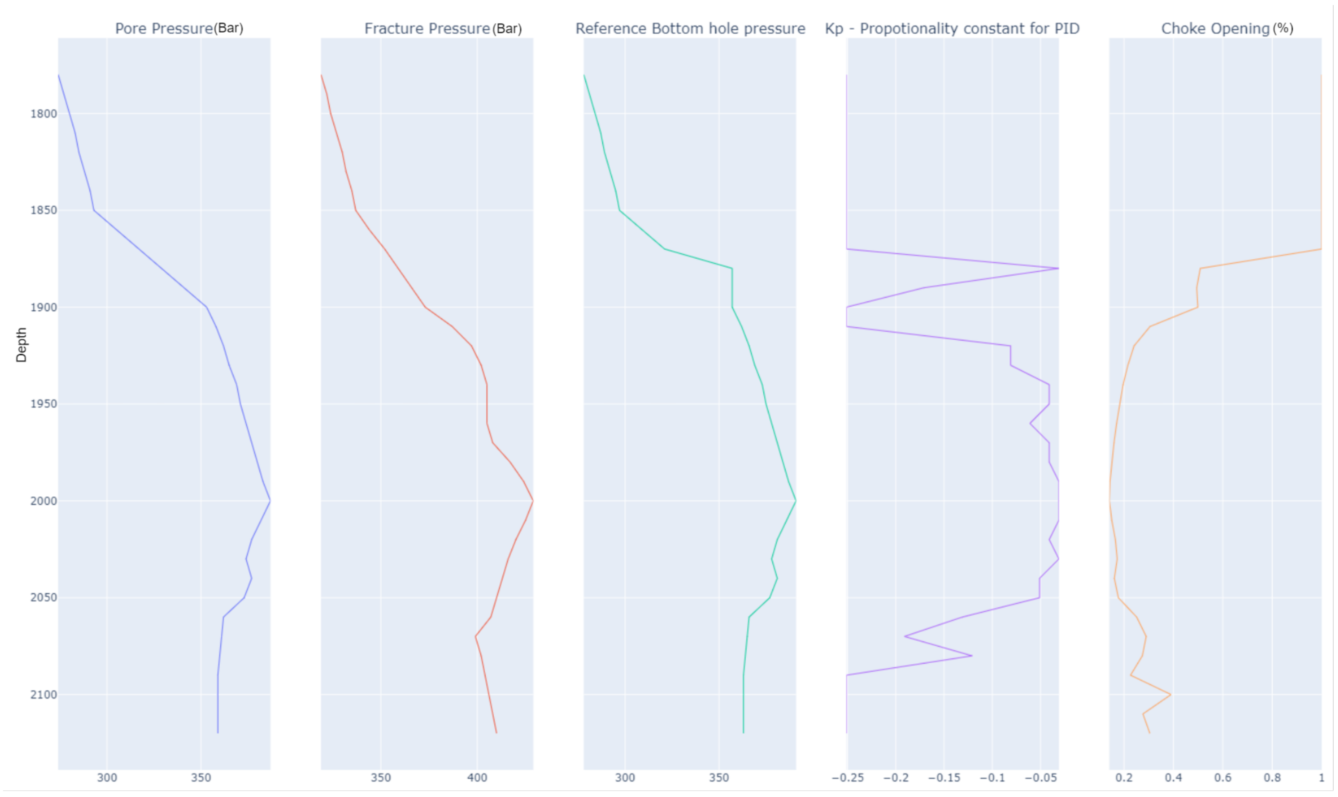


Figure 6.7: Pore Pressure, Fracture Pressure, Reference BHP, Kp, Choke Opening from left to right

Lets visualize the bottom hole pressure and choke opening at one of the points where the window is really narrow lets say 1920m. The pore pressure at this window is 362 bar and fracture pressure is 397 bar. The set-point suggested at this point is 366 Bar and the suggested value of Kp is -0.081. Figure 6.8 and 6.9 show the bottom hole pressure and choke opening respectively. It can be seen the pressure window is respected by the set-point suggestion and the maximum overshoot is also about 5 bar which is normally considered as acceptable even at the narrowest point. This also depends on how many points do we have in the discrete action space if there are more actions more densely placed closer together it would result in an even better policy.

For PPO and for multi-armed bandit problem Kp was ranged from -0.251 to -0.001 with a step of 0.01 where as set-point for bottom hole pressure starts from Pore pressure (+ 4 bar at least) to fracture pressure and the step size is of 6 bar.At the 1920 the pressure window is 35 bar.

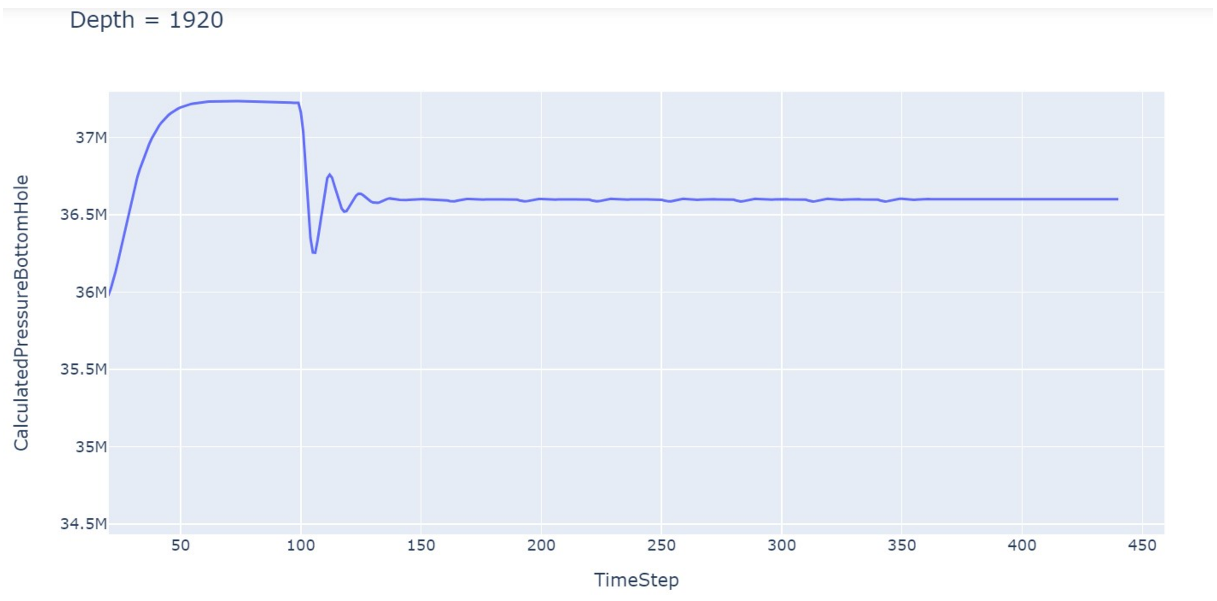


Figure 6.8: Bottom hole pressure at 1920 meters

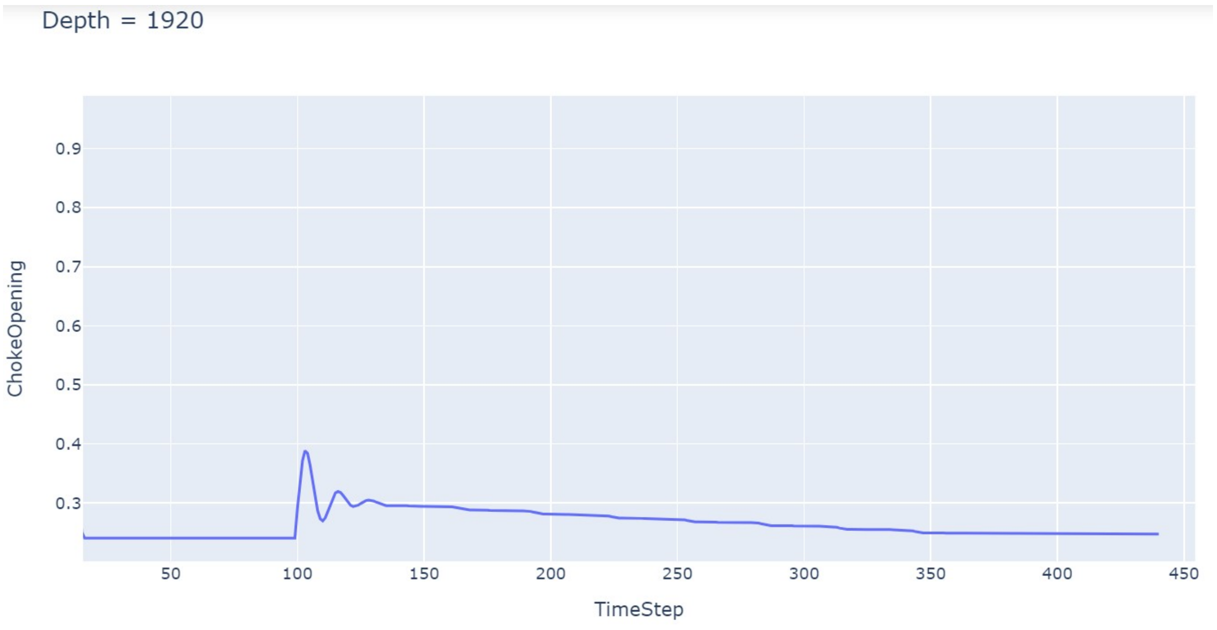


Figure 6.9: Choke Opening at 1920 meters

Now lets look at the interesting region where the set-point went closer to the fracture pressure. In order to troubleshoot this problem lets run a simulation where no negative penalty from oscillations is incurred meaning only the first part of reward function discussed in section 4.2.1 is used. Figure 6.10 shows such an implementation.

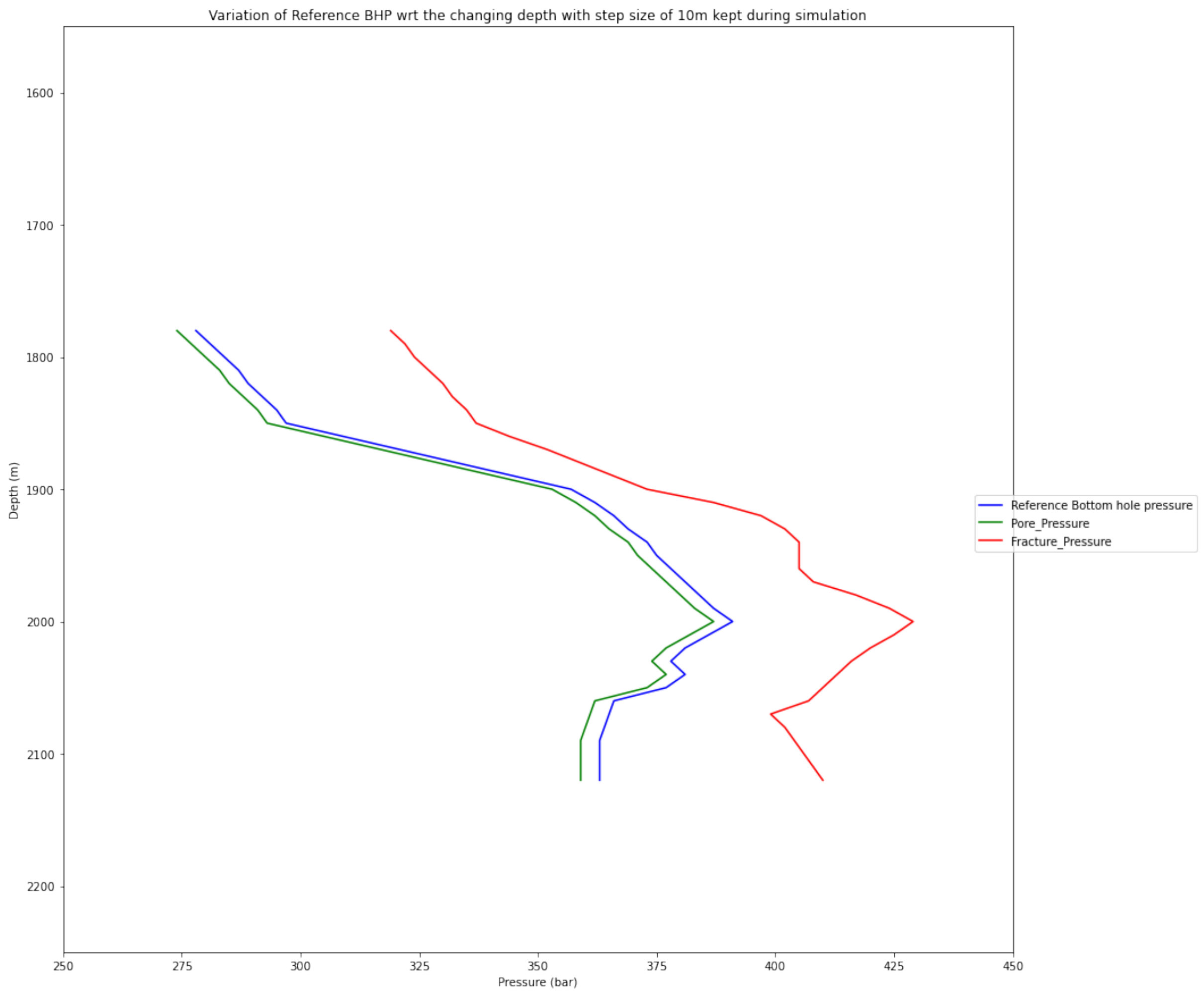


Figure 6.10: Variations in the set-point suggestion using $\epsilon - greedy$ method, Pore pressure (red), Fracture pressure (green) and set-point suggested (blue) with REWARD FOR OSCILLATIONS SET TO ZERO

In the figure 6.10 above, it can be observed that the point (1880m and 1890m) where the set-point suggestion went closer to the fracture pressure (in Figure 6.6) is absent. This explains the reason behind the peculiar behaviour, the agent tries to find a sacrifice between oscillations and a very good set-point suggestion, at these depths the set-point closer to the pore pressure could not be reached due to the selected range of Kp and flow rate setting as it falls well below the minimum pressure that can be achieved even with the valve fully open. Hence the agent found another set-point which was closer to the fracture pressure but the response was very stable and

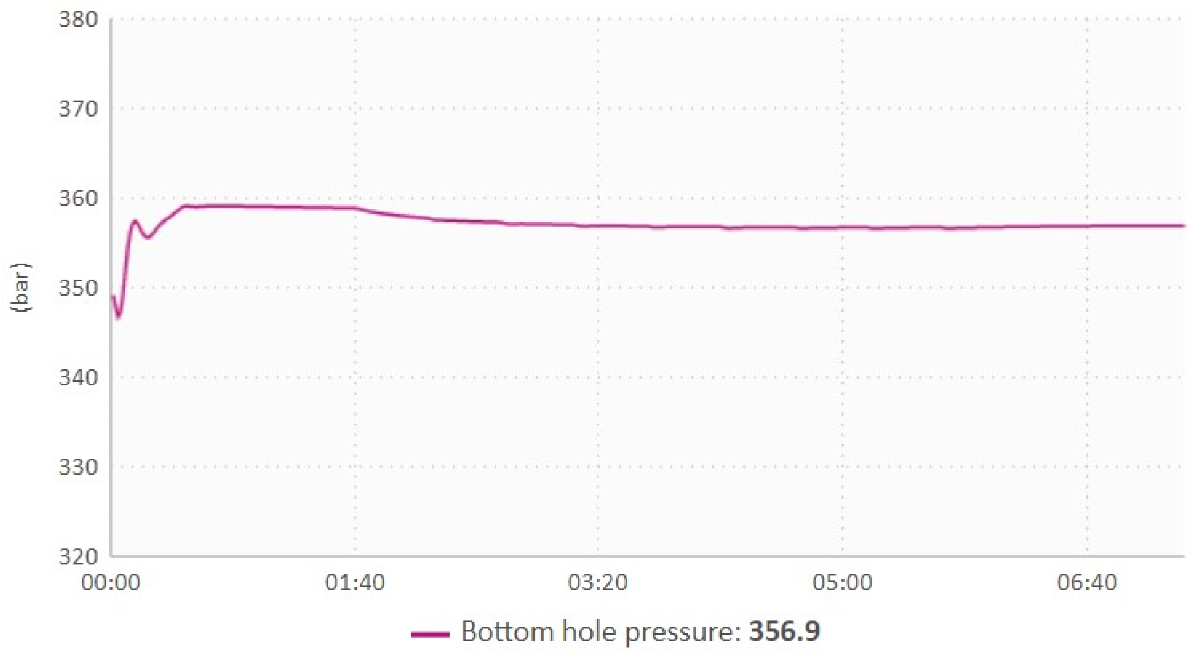


Figure 6.11: Bottom hole pressure response when oscillations reward (penalty) is considered, set-point is 357 bar here

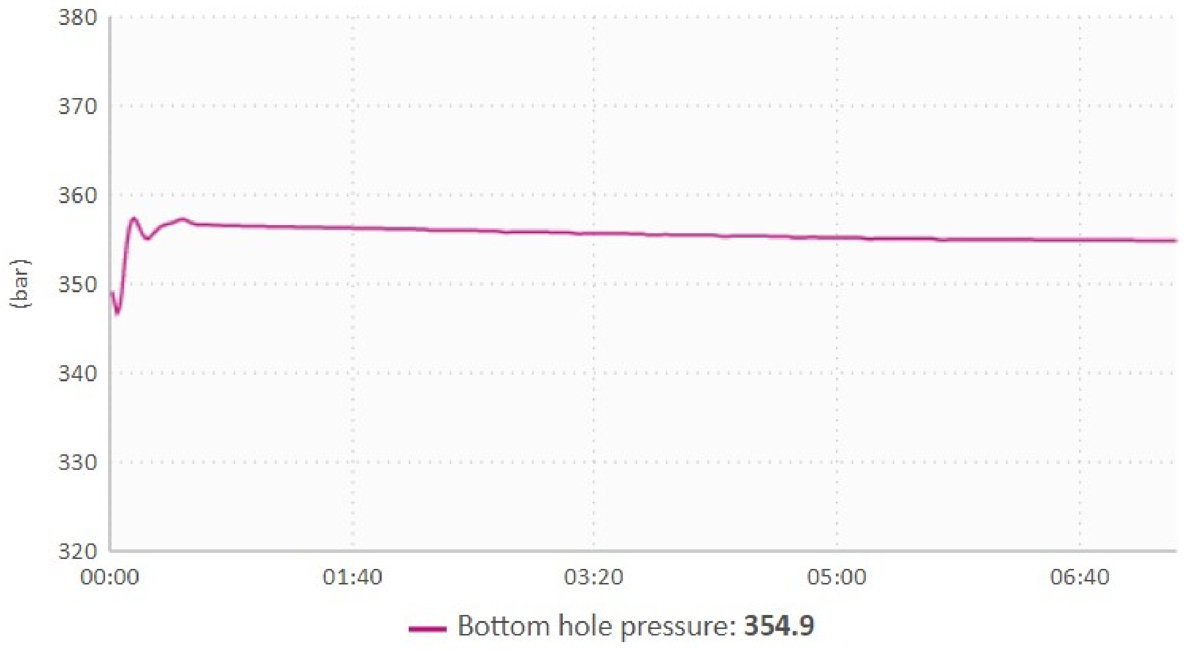


Figure 6.12: Bottom hole pressure response when oscillations reward is not considered, set-point here is 333 bar

good. Figure 6.11 and 6.12 show BHP response for both methods (with and without oscillations reward) at the depth of 1880m where the pore pressure is 329 bar and fracture pressure is 359 bar.

At The final value of BHP in Figure 6.11 (oscillations reward considered) is 356.9 bar where as the set-point in this case was 357 bar, where as the final value in Figure 6.12, is 354.9 bar whereas the set-point was 333 bar. This means that a negative penalty was incurred at every time step if oscillations reward was taken into account and because of this reason the agent chose a pressure closer to fracture to avoid this situation. This peculiar situation depends on the the flow rate settings, geo-pressures set by developer during simulations and also on the action space i.e. either it is very dense or very sparse.

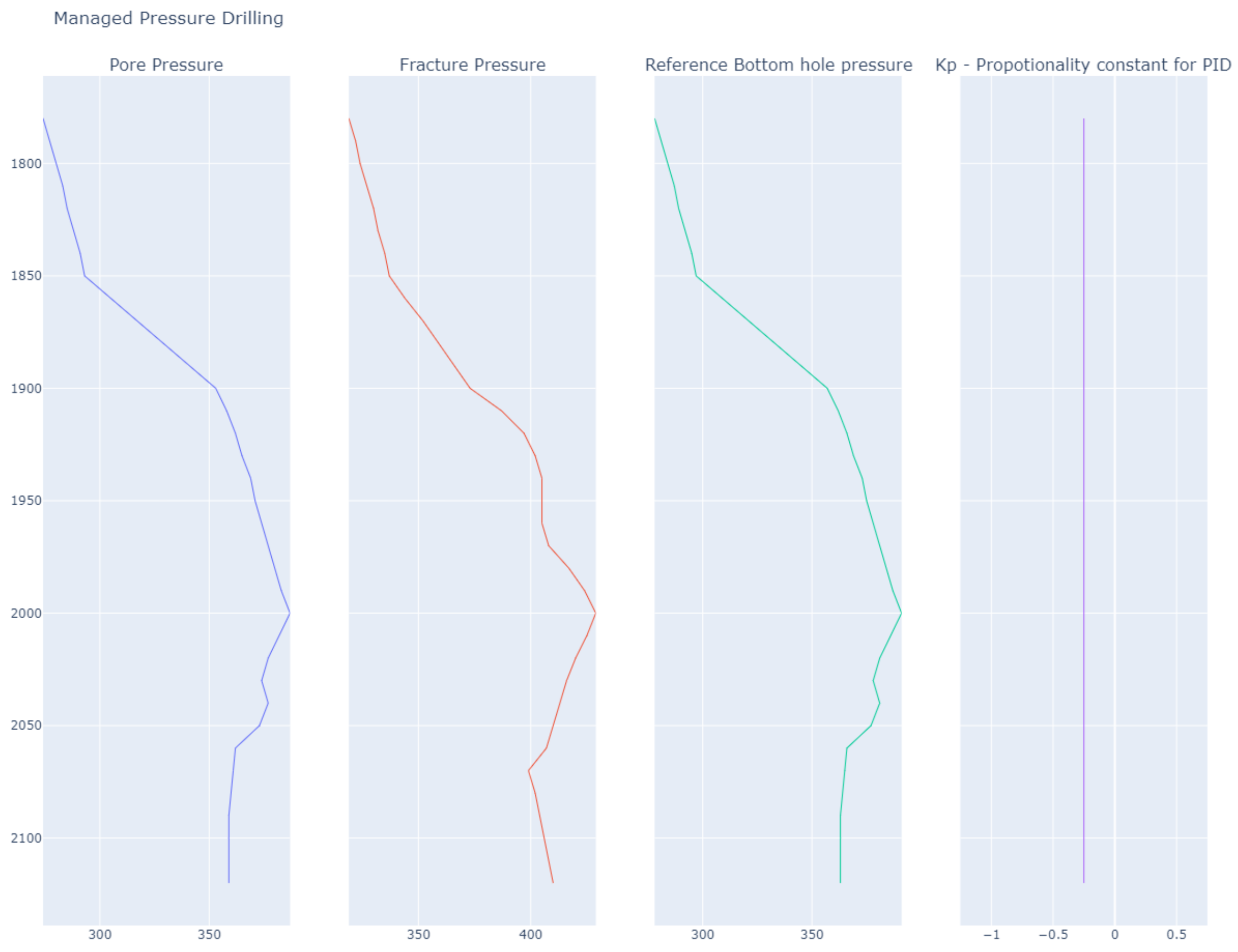


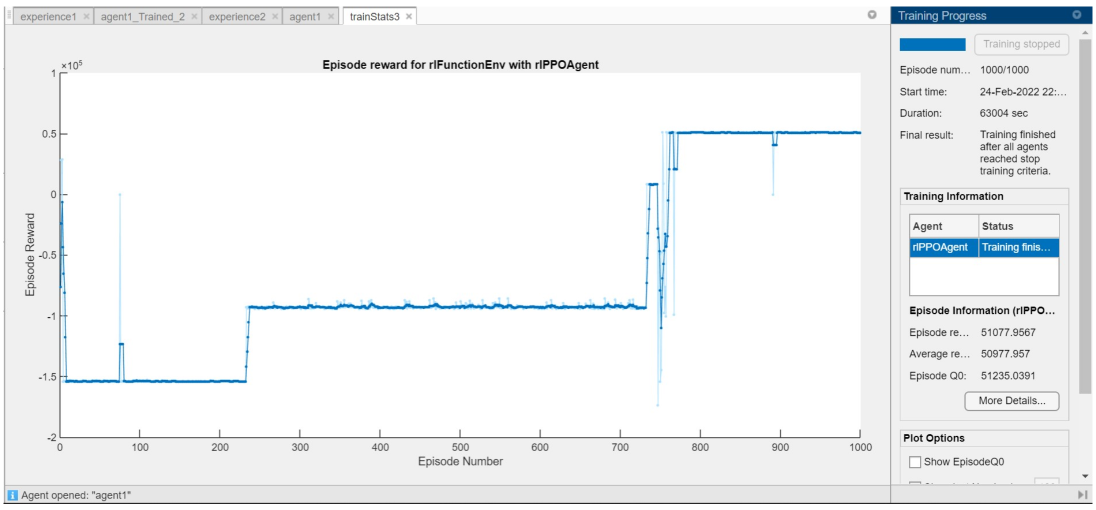
Figure 6.13: Pore Pressure, Fracture Pressure, Reference BHP, Kp from left to right - Case without oscillations reward

In the case where the part of reward function for oscillations is not considered the Kp value is never tuned, which means only half of the job is done and this can be observed in Figure 6.13 shown above.

6.3 Brief Comparison

The main advantage of multi-armed bandit approach has been the efficiency with respect to computational time. Figure **??** shows that for the implementation discussed in chapter 5 for the case of PPO at one depth scenario (2046m), it took 63004 secs (17.5 hours). The main problem is the simulation time taken by each episode it is a little over 1 min (in OpenLab simulator it takes 7 mins but fast forward mode is used to get results faster). Even though OpenLab simulator allows the trained agent to be exported in many forms and deployment in the field is much easier but the sheer time consumption makes it difficult to be useful.

On the other hand it takes under 2 hours to run the multi-armed bandit problem on a system with very humble specifications. Due to these reasons the approach of multi-armed bandit is clear choice in cases where time is critical. Not only this because of its efficiency the method was applied to a whole depth section instead of just one point.



Agent opened: "agent1"

Episode num… 1000/1000
Start time: 24-Feb-2022 22:…
Duration: 63004 sec
Final result: Training finished after all agents reached stop training criteria.

Training Information

Agent	Status
rlPPOAgent	Training finis…

Episode Information (rlPPO…
Episode re… 51077.9567
Average re… 50977.957
Episode Q0: 51235.0391

Figure 6.14: Snapshot of training for one depth scenario

6.4 Objectives completed

The objectives and scope for this study which were set in section 1.2 have been kept in check through out the execution of this study. The major part of this study has been devoted to the development of a stable and robust simulation environment because the whole study depends upon it. To ensure the stability the code includes try and except structures which handle all the possible errors which can be faced during the training process, this process of critical importance because the simulation environment is not run locally but on OpenLab servers which are accessed remotely using their web API. The inputs and outputs of the system have been defined clearly to ensure that the decision making process for agent is smooth and easy to troubleshoot for the developer. The study however is tested on a virtual simulator but the goals set are kept closely related to actual drilling scenarios.

Chapter 7

Future work and Conclusion

7.1 Conclusion

During the design and development phases of this study, many alternatives were explored and many other algorithms were also analysed. Some interesting conclusions are mentioned in the bullet points below.

- The most important parameters the application phase have been pore pressure and fracture pressure and how they change with respect to depth.
- The simulation environment was tested with various state spaces and various versions were made to test the stability. OpenLab simulator API becomes non-responsive if a very long sequence runs, by a long sequence it means if lots of episodes are run consecutively.
- Having good knowledge of geo-pressure profile or some estimate is critically important for this study to be applicable in field.
- Various agents such as DQN, TD3, DDPG were also tested along with PPO but PPO was found to be the one which converges on a optimum policy in the most time efficient manner.
- The design of pressure profile and flow rate setting indicate the lowest pressures that can be reached during a simulation. Sometimes this can lead to peculiar behaviour which

can be easily understood by looking at the action values and then running independent simulations.

- The size of action space dictates the smoothness in the tuning process, but it also increases the computational complexity as the agent has to sample more actions before it starts exploitation.
- The integration of OpenLab simulator with the RL toolbox in MATLAB has been one of the most important milestones and this can lead to many more interesting applications.

7.2 Future works

Some of the future works where this technique can be applied are mentioned below and some possibilieties of additional work on the same study are also discussed.

- One of the applications of where the very similar methodology can be applied is the ROP optimization on OpenLab simulator. A PI controller is used by the OpenLab for ROP optimization and a similar work can be done for the tuning of its parameters and for set-point of weight on bit (Weight on bit). This application is possible because of the ground work laid out in this study where the OpenLab simulator was used as a simulation environment for the reinforcement learning.
- In this study if the OpenLab simulator is replaced with the a local flow model as in on the same machine, this can help in reducing the computational time and RL toolbox from MATLAB can be directly applied to the choke opening to control the pressure but this will be a different way of tackling the problem using a similar tool as used here. The goal here was to tune the PID controller instead of replacing it completely.
- Another interesting implementation could be the application of this independent agent in the RSS simulator developed by the Drillbotics team at UiS as an automated pressure set-point and controller tuning will help yeild much better and stable results.

References

[1] Kenneth P Malloy, Rick Stone, George Harold Medley, Don M Hannegan, Oliver D Coker, Don Reitsma, Helio Mauricio Santos, Joseph Irvin Kinder, Johan Eck-Olsen, John Walton McCaskill, et al. Managed-pressure drilling: What it is and what it is not. In *IADC/SPE Managed Pressure Drilling and Underbalanced Operations Conference & Exhibition*. OnePetro, 2009.

[2] Wang Guo, Fan Honghai, and Liu Gang. Design and calculation of a mpd model with constant bottom hole pressure. *Petroleum Exploration and Development*, 38(1):103–108, 2011.

[3] National Instruments. Pid theory explained, 2022. Available at `https://www.ni.com/en-no/innovations/white-papers/06/pid-theory-explained.html`.

[4] John-Morten Godhavn. Control requirements for high-end automatic mpd operations. In *SPE/IADC Drilling Conference and Exhibition*. OnePetro, 2009.

[5] Mikkel ArnØ, John-Morten Godhavn, and Ole Morten Aamo. Deep reinforcement learning applied to managed pressure drilling. In *SPE Norway Subsurface Conference*. OnePetro, 2020.

[6] Glenn-Ole Kaasa, Øyvind Nistad Stamnes, Lars Imsland, and Ole Morten Aamo. Simplified hydraulics model used for intelligent estimation of downhole pressure for a managed-pressure-drilling control system. *SPE Drilling & Completion*, 27(01):127–138, 2012.

[7] NR Ravishankar and MV Vijayakumar. Reinforcement learning algorithms: survey and classification. *Indian J. Sci. Technol*, 10(1):1–8, 2017.

[8] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[9] Canyu Sun. Fundamental q-learning algorithm in finding optimal policy. In *2017 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, pages 243–246. IEEE, 2017.

[10] Yi Zhou. Robotics motion control by using deep reinforcement learning. 2021.

[11] Michael Gimelfarb, Scott Sanner, and Chi-Guhn Lee. {\epsilon}-bmc: A bayesian ensemble approach to epsilon-greedy exploration in model-free reinforcement learning. *arXiv preprint arXiv:2007.00869*, 2020.

[12] Mathworks. Reinforcement learning toolbox user's guide 2022a, 2022. Available at `https://www.mathworks.com/help/pdf_doc/reinforcement-learning/rl_ug.pdf`.

[13] DOTX CONTROL SOLUTIONS. The pid controller, 2022. Available at `https://www.pid-tuner.com/pid-control/`.

[14] Mathworks. Proximal policy optimization agents, 2022. Available at `https://www.mathworks.com/help/reinforcement-learning/ug/ppo-agents.html`.

[15] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[16] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[17] Kiam Heong Ang, Gregory Chong, and Yun Li. Pid control system analysis, design, and technology. *IEEE transactions on control systems technology*, 13(4):559–576, 2005.

[18] Antonio Turiel. The energy crisis in the world today: analysis of the world energy outlook 2021. 2022.

[19] Truong H Nguyen, Wisup Bae, and Nhan T Hoang. Effect of high pressure high temperature condition on well design development in offshore vietnam. In *Offshore Technology Conference Asia*. OnePetro, 2016.

[20] Maurizio Antonio Arnone and Paco Vieira. Drilling wells with narrow operating windows applying the mpd constant bottom hole pressure technology—how much the temperature and pressure affects the operation's design. In *SPE/IADC Drilling Conference and Exhibition*. OnePetro, 2009.

[21] Fred NG. Kick handling with losses in an hpht environment. *World oil*, 230(3), 2009.

[22] P McLellan and C Hawkes. Borehole stability analysis for underbalanced drilling. *Journal of Canadian Petroleum Technology*, 40(05), 2001.

[23] Brandon Hilts. Managed pressure drilling. In *SPE annual technical conference and exhibition*. OnePetro, 2013.

[24] Jan Einar Gravdal, Hardy Siahaan, and Knut S Bjørkevoll. Back-pressure mpd in extended-reach wells-limiting factors for the ability to achieve accurate pressure control. In *SPE Bergen One Day Seminar*. OnePetro, 2014.

[25] Kenneth P Malloy. Managed pressure drilling-: What is it anyway?: Managed pressure drilling. *World oil*, 228(3), 2007.

[26] Graham Clifford Goodwin, Stefan F Graebe, Mario E Salgado, et al. *Control system design*, volume 240. Prentice Hall Upper Saddle River, 2001.

[27] Andrew Hynes, Elena P Sapozhnikova, and Ivana Dusparic. Optimising pid control with residual policy reinforcement learning. In *AICS*, pages 277–288, 2020.

[28] Abhijit Gosavi. Reinforcement learning: A tutorial survey and recent advances. *INFORMS Journal on Computing*, 21(2):178–192, 2009.

[29] Asis Kumar Chattopadhyay and Tanuka Chattopadhyay. Monte carlo simulation. In *Statistical Methods for Astronomical Data Analysis*, pages 241–275. Springer, 2014.

[30] Aditya Mahajan and Demosthenis Teneketzis. Multi-armed bandit problems. In *Foundations and applications of sensor management*, pages 121–151. Springer, 2008.

[31] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[33] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Leonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, et al. What matters for on-policy deep actor-critic methods? a large-scale study. In *International conference on learning representations*, 2020.

[34] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[35] Thibaut Théate and Damien Ernst. An application of deep reinforcement learning to algorithmic trading. *Expert Systems with Applications*, 173:114632, 2021.

[36] J Frøyen and O Savareid. Model equations and solution techniques for multiphase flow in pipe networks. *IRIS, Stavanger, Norway*, 2000.

[37] J Frøyen, O Sævareid, and EH Vefring. Discretization, implementation and testing of a semi-implicit method. *Technical ReportRF-2000/157, International Research Institute of Stavanger (IRIS), Stavanger, Norway*, 2000.

[38] Mamoru Ishii. Thermo-fluid dynamic theory of two-phase flow. *NASA Sti/recon Technical Report A*, 75:29657, 1975.

[39] Dalila Gomes, Knut Steinar Bjørkevoll, Johnny Frøyen, Kjell Kåre Fjelde, Dan Sui, John Emeka Udegbunam, and Fatemeh Moeinikia. Probabilistic flow modelling approach for kick tolerance calculations. In *International Conference on Offshore Mechanics and Arctic Engineering*, volume 57762, page V008T11A062. American Society of Mechanical Engineers, 2017.

[40] MJ Jellison, R Urbanowski, H Sporker, and ME Reeves. Intelligent drill pipe improves drilling efficiency, enhances well safety and provides added value. In *IADC world drilling conference, Dubrovnik, Croatia*, pages 1–2, 2004.

[41] ACVM Lage. Two-phase flow models and experiments for low-head and underbalanced drilling. *PhD dissertation*, 2000.

[42] TD Reed and AA Pilehvari. A new model for laminar, transitional, and turbulent flow of drilling muds. In *SPE Production Operations Symposium*. OnePetro, 1993.

[43] Jing Zhou, Jan Einar Gravdal, Per Strand, and Svein Hovland. Automated kick control procedure for an influx in managed pressure drilling operations. 2016.

[44] NORCE Norwegian Research Centre AS. Openlab api, 2022. Available at https://https://live.openlab.app/swagger/index.html.

[45] Jan Einar Gravdal, Dan Sui, Attila Nagy, Nejm Saadallah, and Robert Ewald. A hybrid test environment for verification of drilling automation systems. In *SPE/IADC International Drilling Conference and Exhibition*. OnePetro, 2021.

[46] Yasutaka Kishima and Kentarou Kurashige. Reduction of state space in reinforcement learning by sensor selection. *Artificial Life and Robotics*, 18(1):7–14, 2013.

[47] Steve Roberts. Bandit algorithms, 2020. Available at https://towardsdatascience.com/bandit-algorithms-34fd7890cb18.

Appendices

Appendix A

Development Code

A.1 Installed Packages

Python package	Version	MATLAB 2021a Add-On	Version
anaconda-client	1.9.0	Signal Processing Toolbox	8.7
anaconda-navigator	2.1.4	Control System Toolbox	10.11
anaconda-project	0.10.1	Reinforcement Learning Toolbox	2.1
click	8.0.3	Deep Learning Toolbox	14.3
conda	4.12.0	Deep Learning HDL Toolbox	1.2
conda-build	3.21.6	Statistics and Machine Learning Toolbox	12.2
conda-content-trust	0.0.0	–	–
conda-pack	0.0.0	–	–
conda-package-handling	1.7.3	–	–
conda-repo-cli	1.0.4	–	–
conda-token	0.3.0	–	–
conda-verify	3.4.2	–	–
jupyter	1.0.0	–	–
jupyter-client	6.1.12	–	–
jupyter-console	6.4.0	–	–
jupyter-core	4.8.1	–	–
jupyter-server	1.4.1	–	–
jupyterlab	3.2.1	–	–
jupyterlab-pygments	0.1.2	–	–
jupyterlab-server	2.8.2	–	–
jupyterlab-widgets	1.0.0	–	–
kaleido	0.2.1	–	–
matplotlib	3.4.3	–	–
matplotlib-inline	0.1.2	–	–
math	3.4.2	–	–
numpy	1.20.3	–	–
openlab	2.5.3	–	–
pandas	1.3.4	–	–
plotly	5.5.0	–	–
python-dateutil	2.8.2	–	–
python-lsp-black	1.0.0	–	–
python-lsp-jsonrpc	1.0.0	–	–

python-lsp-server	1.2.4	–	–
python-slugify	5.0.2	–	–
spyder	5.1.5	–	–
spyder-kernels	2.1.3	–	–

A.2 Code for Step Function - Version 1

This is the code for the step function in the initial implementation of MDP and it uses some inbuilt functions to call the OpenLab API and these functions are provided in form of scripts by the OpenLab.

```
function [NextObs,Reward,IsDone,LoggedSignal] = mpdStepFunction(
    Action,LoggedSignal)


% This function applies the given action to the environment and
    evaluates
% the system dynamics for one simulation step.

%% Define the environment constants.
global Sim MaxTimeSteps timeStep RampIndex RampStartTime
    RampTimeSteps RampValues FlowRateIn PI ReferenceBHPPressure
    InitialChokeOpening Kp Ki Ts BHP_arr InitialBitDepth PorePressure
    FracturePressure tableOfPressures;

% NOTE: You need Matlab R2016b version to run the OpenLab simulator
disp("MPD STEP FUNCTION CALLED");

%% Clipping for Continuous Space
%if (Action > 1)
%     Action = 1;
%     Reward = -10000;
%elseif(Action < -1)
%     Action = -1;
%     Reward = -10000;
%else
%     Reward = 0;
%end
%% Putting a check on  if this is a new simulation

disp("Check for episode end");

if (LoggedSignal.state(3) == -1)  %Meaning reset function was called
    and ISDone was set to 1 in the last episode
    try
        if (Sim.IsOK)
            Sim.Stop;
            Sim.Delete;
        end

    catch exception
        disp("No simulation to stop");
    end

    disp("Time to update value of KP, incomming action value is");

    disp(Action);
    Action = squeeze(Action);
    Action = double (Action);

    Kp = Action(1);
    Ki = Kp/10;
    disp("Kp value updated");

    %Initial Settings
    disp("OpenLab initial settings .. ");
    IdentityServerURL = 'https://live.openlab.app/';
    [username, api_key, license_guid] = GetLoginData();
    ConfigurationName = 'MPD TEST';
    SimulationName = 'Flow sweep with bhp control (Matlab)';
    InitialBitDepth = 2046; % [m]
    InitialTopOfStringPosition = 20;
    disp("Openlab initial setting done");

    % Create simulation object
    disp("Creating simulation object");
    Sim = OpenLabClient(IdentityServerURL, username,api_key,
   license_guid,ConfigurationName,SimulationName,InitialBitDepth,
   InitialTopOfStringPosition);
    disp("Simulation object created");

    % Ramp settings
    disp("Ramp setting creation");
    RampIndex = 1;

    % Randomize the Ramp settings we get a high Ramp value between
   2800 to
    % 2900 and then lower is always 700 less than higher.

    higherRampValue = 2800 + (2900-2800) .* rand(1,1);
    lowerRampValue = higherRampValue - 700;
    RampValuesDown = (higherRampValue:-50:lowerRampValue)/60000; % [
   m3/sec]
    %RampValuesUp = (lowerRampValue:50:higherRampValue)/60000;  % [m3
   /sec]
    RampValues = [RampValuesDown]; % [m3/sec]
    RampStepDuration = 30; % [sec]
    RampStartTime = 100; % [sec]
    RampTimeSteps = RampStartTime:RampStepDuration:length(RampValues)
    *RampStepDuration-1+RampStartTime;

    % Time steps to simulate
    MaxTimeSteps = (RampTimeSteps(end)+100);
    % Controller settings
    Ts = 1;
    ReferenceBHPPressure = Action(2) * 1E5; % [Pa] (Action(2) is in
    Bar)
    InitialChokeOpening = 0.25; % [closed: 0, open: 1]
    PI = PIcontroller(Kp, Ki, Ts, ReferenceBHPPressure,
    InitialChokeOpening); % Create PI controller object, pressure in [
    Pa]
    timeStep = 1;
    BHP_arr= []; %Empty Simulation BHP array

    %Pore and Fracture Pressure extraction and calculation from the
    %gradient
    tableOfPressures = readtable('geopressures.csv','NumHeaderLines'
    ,1);
    Depths = table2array(tableOfPressures(:,1));
    %Precaution for extrapolation error.
    if (InitialBitDepth > Depths(end))
            error('Unable to determine pressure window at this depth.
    %g',...
        InitialBitDepth);
    end

    PorePressure = interp1(table2array(tableOfPressures(:,1)),
    table2array(tableOfPressures(:,2)),InitialBitDepth);
    FracturePressure = interp1(table2array(tableOfPressures(:,1)),
    table2array(tableOfPressures(:,3)),InitialBitDepth);
    %https://www.sciencedirect.com/topics/engineering/formation-pore-
    pressure
    PorePressure = (8.345*PorePressure)*(0.051948*InitialBitDepth
    *3.28) *6894.76 ; % s.g to ppg, then psi then to Pascals
    FracturePressure = (8.345*FracturePressure)*(0.051948*
    InitialBitDepth*3.28) *6894.76 ; % Same to Pascals

    %Recording the estimated setpoints for record keeping
    fid1 = fopen("record.txt",'a+');
    fprintf(fid1,'%f  %f, \n ',ReferenceBHPPressure,Kp);
    fclose(fid1);


    %Initializing Reward based on estimated set point
    Reward = Reward_PBH(table2array(tableOfPressures(:,2)),
    table2array(tableOfPressures(:,3)), table2array(tableOfPressures
    (:,1)), Action(2), InitialBitDepth);

else
```

```
    Reward = 0;
end

%% OUT OF BOUNDS ACTION CHECK
%display((FracturePressure/1E5))

if ~ismember(Action(2),((ceil(PorePressure/1E5):6:FracturePressure/1E5)))
    error('Action is out of bounds. %g',...
        Action(2));
end

if ~ismember(Action(1),((-0.255:0.01:-0.005)))
    error('Action is out of bounds. %g',...
        Action(1));
end

disp("Check for episode status completed");

%% TAKING STEPS IN AN EPISODE
if Sim.IsOK
    try
        if (timeStep <= MaxTimeSteps)
        %for timeStep = 1 : MaxTimeSteps
            disp(Kp);
            tStartStep = tic;

            if timeStep >= RampStartTime % Flow sweep and PI control of the choke
                if (RampIndex < length(RampTimeSteps) && RampIndex < length(RampValues))
                    if (timeStep >= RampTimeSteps(RampIndex) && timeStep < RampTimeSteps(RampIndex + 1))
                        FlowRateIn = RampValues(RampIndex);
                        if (timeStep == RampTimeSteps(RampIndex + 1) - 1)
                            RampIndex = RampIndex + 1;
                        end
                    end
                elseif (RampIndex == length(RampTimeSteps) && RampIndex == length(RampValues))
                    FlowRateIn = RampValues(RampIndex);
                end


                % Reset PI controller before usage, set reference value and initial output (= initial
                % choke opening)
                PI.Reset(ReferenceBHPPressure,Sim.ChokeOpeningStatus);


                ChokeOpening   = PI.GetOutput(Sim.BottomHolePressure);
 % Get choke opening from PI controller

            else % Constant flow rate and choke opening
                FlowRateIn = RampValues(1);
                ChokeOpening = InitialChokeOpening;
            end

            % Set setpoints to the simulator
            Sim.FlowRateIn = FlowRateIn;  %  [m3/sec]
            Sim.ChokeOpening = ChokeOpening;  %   [0-1]
            Sim.ChokePumpFlowRateIn = 0/60000;  %  [m3/sec]
            Sim.TopOfStringVelocity = 0;  %  [m/sec]
            Sim.SurfaceRPM = 0/60;  %  [revolutions per sec.]
            Sim.ROP = 0/3600;  %  [m/sec]

            % Step simulator
            Sim.Step();


            disp("Step Taken")
            if ~Sim.IsOK % Simulator fails
                    error('Error: Simulator Failed');
            end

            %Appending Operation
            BHP_arr= [BHP_arr, Sim.BottomHolePressure];

            %Reading System Dynamics
            SysDynamics = stepinfo(BHP_arr,(1 : timeStep),
ReferenceBHPPressure,'SettlingTimeThreshold',0.005);
            LoggedSignal.state = [SysDynamics.("Overshoot")/
ReferenceBHPPressure;SysDynamics.("TransientTime");SysDynamics.("
SettlingTime");InitialBitDepth;Sim.ChokeOpening;Sim.FlowRateOut;
Action(2);Sim.BottomHolePressure;PorePressure;FracturePressure];

            %Checking pressure window

            %disp(Sim.BottomHolePressure);
            %disp([InitialBitDepth]);
            %disp([Sim.ChokeOpening]);
            %disp([Sim.ChokePumpFlowRateIn]);
            %disp([Sim.ROP]);
            %disp(LoggedSignal.state);

            %Reward updates at the end of every simulation step
            if (abs((Sim.BottomHolePressure - ReferenceBHPPressure)/
ReferenceBHPPressure) > 0.01)
                Reward = Reward - 250;
                display(Reward);
                display("Overshoot penality in")
            else
                Reward = Reward + 0;
            end

            %if (isnan(LoggedSignal.state(2)))
            %     Reward = Reward - 25;
            %end

            %If settling point has reached we should step ot of the loop
            %and break it and update reward .

            %if (~isnan(LoggedSignal.state(3)))
            %     Reward = Reward + 100 ;
            %     break;
            %end
            timeStep = timeStep + 1;
            disp("Episode Continued")
            IsDone = 0;
        else
            % Stop simulation process and complete the episode
            Sim.Stop;
            IsDone = 1;
            %Reward update at the end of simulation

            if (isnan(LoggedSignal.state(3))) % End of simulation/episode check
                Reward = Reward - 100000;
            elseif(isnan(LoggedSignal.state(2)))
                Reward = Reward - 100000;
            else
                Reward = 0;
            end

            disp("Episode Ended");

        end

        %assign the next observation
        NextObs = LoggedSignal.state;

        %Calculation for Decay Ratio of the BHP
        %Oscilations = BHP_arr - ReferenceBHPPressure;
        %plot(1:numel(Oscilations),Oscilations);


        %Fixing Nans errors for Times
        if (isnan(LoggedSignal.state(3)))
            NextObs(3) = -5;
        end

        if (isnan(LoggedSignal.state(2)))
            NextObs(2) = -5;
        end


        %disp(BHP_arr)

    catch exception
        % Stop simulation
        display("Warning reached")
        Sim.Stop;
        warning(exception);

    end
end

end
```

A.3 Code for Reset Function - Version 1

This is the code for the reset function used in the intial implementation of the MDP.

```
function [InitialObservation, LoggedSignal] = mpdResetFunction()
% Reset function to place environment in initial setup
% initial state.

display("ResetFunctionCalled");
global PorePressure FracturePressure InitialBitDepth

% Create simulation object
LoggedSignal.state = [-1;-1;-1;InitialBitDepth;-1;-1;-1;-1;
    PorePressure;FracturePressure];

InitialObservation = LoggedSignal.state;
%display(InitialObservation)

end
```

A.4 Code for setting up Environment - Version 1

This is the code for setting up the environment, action and state space for the initial implementation of MDP.

```
clear variables;
clc;
close all;
clear global;

global PorePressure FracturePressure InitialBitDepth

InitialBitDepth = 2046;
tableOfPressures = readtable('geopressures.csv','NumHeaderLines',1);
Depths = table2array(tableOfPressures(:,1));
PorePressure = interp1(table2array(tableOfPressures(:,1)),table2array
    (tableOfPressures(:,2)),InitialBitDepth);
FracturePressure = interp1(table2array(tableOfPressures(:,1)),
    table2array(tableOfPressures(:,3)),InitialBitDepth);

%https://www.sciencedirect.com/topics/engineering/formation-pore-
    pressure
PorePressure = (8.345*PorePressure)*(0.051948*InitialBitDepth*3.28)
    *6894.76 ; % s.g to ppg, then psi then to Pascals
FracturePressure = (8.345*FracturePressure)*(0.051948*InitialBitDepth
    *3.28) *6894.76 ; % Same to Pascals

% Add to the matlab path the folder where this file is located and
    all the
% subfolders
disp("Display Environment Variables");
addpath(genpath(pwd));

ObservationInfo = rlNumericSpec([10 1]);
ObservationInfo.Name = 'Controller tunning parameters and system
    parameters states';
ObservationInfo.Description = 'Overshoot ratio, TransientTime ,
    SettlingTime, InitialBitDepth, ChokeOpening, Flow Rate Out, BHP
    Reference, Current Simulation BHP, Pore Pressure, Fracture
    Pressure';

% Continuous Action Space Configuration - Comment out ismemeber check
     in
% the step function

%ActionInfo = rlNumericSpec([1 1],UpperLimit = 1,LowerLimit = -1); %
    Dont assign limits outisde this definition - weird error.
%ActionInfo.Name = 'Controller Action';
%env = rlFunctionEnv(ObservationInfo,ActionInfo,'mpdStepFunction','
    mpdResetFunction');

%Discrete Action Space Configuration - Must be reflected in the Step
%function as well.
kpRange = -0.255:0.01:-0.005;
setptsRange = ceil(PorePressure/1E5):6:FracturePressure/1E5;
disp("Pore and Fracture Pressures at Depth");
disp(InitialBitDepth);
disp("are as follows");
disp(ceil(PorePressure/1E5));
disp(FracturePressure/1E5);

C = {kpRange,setptsRange};
D = C;
[D{:}] = ndgrid(C{:});
Z = cell2mat(cellfun(@(m)m(:),D,'uni',0));

Act = transpose(num2cell(Z,2));

 ActionInfo = rlFiniteSetSpec(Act);


 ActionInfo.Name = 'Controller Action';
env = rlFunctionEnv(ObservationInfo,ActionInfo,'mpdStepFunction','
    mpdResetFunction');
```

A.5 ENVIRONMENT STEP FUNCTION ONE STATE MDP

This code is for the step function of the one step MDP case which is also referred to as the modified version of first implementation and it is based on A.2. with some changes.

```
    function [NextObs,Reward,IsDone,LoggedSignal] = mpdStepFunction(
    Action,LoggedSignal)

global InitialBitDepth PorePressure FracturePressure tableOfPressures
    ;

disp("MPD STEP FUNCTION CALLED");

addpath(genpath(pwd));

IdentityServerURL = 'https://live.openlab.app/';

[username, api_key, license_guid] = GetLoginData();

ConfigurationName = 'MPD TEST';
SimulationName = 'Flow sweep with bhp control (Matlab) lab';

InitialTopOfStringPosition = 20;


% Create simulation object
while true
    try
```

```
        Sim = OpenLabClient(IdentityServerURL, username,api_key,license_guid,ConfigurationName,SimulationName,InitialBitDepth,InitialTopOfStringPosition);
    catch exception
        Sim.Stop;
        Sim.Delete;
    end

    if(Sim.IsOK)
        break;
    end
end

% Ramp settings
RampIndex = 1;

higherRampValue = 2800 + (2900-2800) .* rand(1,1);
lowerRampValue = higherRampValue - 400;

RampValuesDown = (higherRampValue:-50:lowerRampValue)/60000; % [m3/sec]
%RampValuesUp = (lowerRampValue:50:higherRampValue)/60000;  % [m3/sec]
RampValues = [RampValuesDown]; % [m3/sec]
RampStepDuration = 30; % [sec]
RampStartTime = 100; % [sec]
RampTimeSteps = RampStartTime:RampStepDuration:length(RampValues)*RampStepDuration-1+RampStartTime;

% Time steps to simulate
MaxTimeSteps = (RampTimeSteps(end)+100);

% Controller settings
Kp = Action(1); Ki = Kp/10; Ts = 1;
ReferenceBHPPressure = Action(2) * 1E5; % [Pa]
InitialChokeOpening = 1; % [closed: 0, open: 1]
PI = PIcontroller(Kp, Ki, Ts, ReferenceBHPPressure, InitialChokeOpening); % Create PI controller object, pressure in [Pa]

%Recording the estimated setpoints for record keeping
fid1 = fopen("record.txt",'a+');
fprintf(fid1,'%f %f, \n ',ReferenceBHPPressure,Kp);
fclose(fid1);

%disp("About to call Reward Function")
Reward = Reward_PBH(table2array(tableOfPressures(:,2)), table2array(tableOfPressures(:,3)), table2array(tableOfPressures(:,1)), Action(2), InitialBitDepth);

```

```
if ~ismember(Action(2),((ceil(PorePressure/1E5):6:FracturePressure/1E5)))
    error('Action is out of bounds. %g',...
        Action(2));
end

if ~ismember(Action(1),((-0.255:0.01:-0.005)))
    error('Action is out of bounds. %g',...
        Action(1));
end

if Sim.IsOK
    CompletionFlag = true;

    while(CompletionFlag == true)
        try
            BHP_arr= [];
            for timeStep = 1 : MaxTimeSteps
                tStartStep = tic;

                if timeStep >= RampStartTime % Flow sweep and PI control of the choke
                    if (RampIndex < length(RampTimeSteps) && RampIndex < length(RampValues))
                        if (timeStep >= RampTimeSteps(RampIndex) && timeStep < RampTimeSteps(RampIndex + 1))
                            FlowRateIn = RampValues(RampIndex);
                            if (timeStep == RampTimeSteps(RampIndex + 1) - 1)
                                RampIndex = RampIndex + 1;
                            end
                        end
                    elseif (RampIndex == length(RampTimeSteps) && RampIndex == length(RampValues))
                        FlowRateIn = RampValues(RampIndex);
                    end


                    % Reset PI controller before usage, set reference value and initial output (= initial
                    % choke opening)
                    PI.Reset(ReferenceBHPPressure,Sim.ChokeOpeningStatus);


                    ChokeOpening  = PI.GetOutput(Sim.BottomHolePressure); % Get choke opening from PI controller

                else % Constant flow rate and choke opening
                    FlowRateIn = RampValues(1);
                    ChokeOpening = InitialChokeOpening;
                end

                % Set setpoints to the simulator
                Sim.FlowRateIn = FlowRateIn;  %  [m3/sec]
                Sim.ChokeOpening = ChokeOpening;  %   [0-1]
                Sim.ChokePumpFlowRateIn = 0/60000;  %  [m3/sec]
                Sim.TopOfStringVelocity = 0;  %  [m/sec]
                Sim.SurfaceRPM = 0/60;  %  [revolutions per sec.]
                Sim.ROP = 0/3600;  %  [m/sec]

                % Step simulator
                Sim.Step();

                if ~Sim.IsOK % Simulator fails
                    CompletionFlag = true;
                    break;
                else
                    CompletionFlag = false;
                end

                %display(['Total step duration: ' num2str(toc(
tStartStep))]);
                BHP_arr= [BHP_arr, Sim.BottomHolePressure];
                SysDynamics = stepinfo(BHP_arr,(1 : timeStep),
ReferenceBHPPressure,'SettlingTimeThreshold',0.0005);
                LoggedSignal.state = [Sim.BottomHolePressure;Sim.
BitDepth;PorePressure;FracturePressure];

                %Reward updates at the end of every simulation step
                if (abs((Sim.BottomHolePressure -
ReferenceBHPPressure)/ReferenceBHPPressure) > 0.01)
                    Reward = Reward - 250;
                    disp(Reward);
                    disp("Overshoot penality in");
                else
                    Reward = Reward + 0;
                end

            end

            NextObs = LoggedSignal.state;
            % Stop simulation process

            IsDone = 1;
            Sim.Stop;
            Sim.Delete;


            if (isnan(SysDynamics.("SettlingTime"))) % End of
simulation/episode check
                Reward = Reward - 100000;
```

```
              elseif(isnan(SysDynamics.("TransientTime")))
                  Reward = Reward - 100000;
              else
                  Reward = Reward - 0;
              end

          catch exception
              % Stop simulation
              Sim.Stop;
              Sim.Delete;
              rethrow(exception);
              CompletionFlag = true;
          end
      end
end
end



```

A.6 ENVIRONMENT RESET FUNCTION ONE STATE MDP

This code is for the reset function for the modified environment for MDP case.

```
    function [InitialObservation, LoggedSignal] = mpdResetFunction()
% Reset function to place custom cart-pole environment into a random
% initial state.

global PorePressure FracturePressure InitialBitDepth Depths
    tableOfPressures

InitialBitDepth = 2046;

tableOfPressures = readtable('geopressure.csv','NumHeaderLines',1);
Depths = table2array(tableOfPressures(:,1));
PorePressure = interp1(table2array(tableOfPressures(:,1)),table2array
    (tableOfPressures(:,2)),2046);
FracturePressure = interp1(table2array(tableOfPressures(:,1)),
    table2array(tableOfPressures(:,3)),2046);
%https://www.sciencedirect.com/topics/engineering/formation-pore-
    pressure
PorePressure = (8.345*PorePressure)*(0.051948*InitialBitDepth*3.28)
    *6894.76 ; % s.g to ppg, then psi then to Pascals
FracturePressure = (8.345*FracturePressure)*(0.051948*InitialBitDepth
    *3.28) *6894.76 ; % Same to Pascals

%disp("ResetFunctionCalled");

```

```
InitialBitDepth = 2046;

% Create simulation object
LoggedSignal.state = [0;InitialBitDepth;PorePressure;FracturePressure
    ];

InitialObservation = LoggedSignal.state;
%display(InitialObservation)

end


```

A.7 ENVIRONMENT SETUP CODE ONE STATE MDP

This code is for the modified environment, this piece of code is used to setup the environment and define action and state spaces.

```

    clear variables;
clc;
close all;
clear global;

global PorePressure FracturePressure InitialBitDepth tableOfPressures


InitialBitDepth = 2046;

tableOfPressures = readtable('geopressure.csv','NumHeaderLines',1);
Depths = table2array(tableOfPressures(:,1));
PorePressure = interp1(table2array(tableOfPressures(:,1)),table2array
    (tableOfPressures(:,2)),2046);
FracturePressure = interp1(table2array(tableOfPressures(:,1)),
    table2array(tableOfPressures(:,3)),2046);

%https://www.sciencedirect.com/topics/engineering/formation-pore-
    pressure
PorePressure = (8.345*PorePressure)*(0.051948*InitialBitDepth*3.28)
    *6894.76 ; % s.g to ppg, then psi then to Pascals
FracturePressure = (8.345*FracturePressure)*(0.051948*InitialBitDepth
    *3.28) *6894.76 ; % Same to Pascals



% Add to the matlab path the folder where this file is located and
    all the
```

```
% subfolders
disp("Display Environment Variables");
addpath(genpath(pwd));

ObservationInfo = rlNumericSpec([4 1]);
ObservationInfo.Name = 'Controller tunning parameters and system
    parameters states';
ObservationInfo.Description = 'Current Simulation BHP, BitDepth, Pore
    Pressure, Fracture Pressure';

% Continuous Action Space Configuration - Comment out ismemeber check
    in
% the step function

%ActionInfo = rlNumericSpec([1 1],UpperLimit = 1,LowerLimit = -1); %
    Dont assign limits outisde this definition - weird error.
%ActionInfo.Name = 'Controller Action';
%env = rlFunctionEnv(ObservationInfo,ActionInfo,'mpdStepFunction','
    mpdResetFunction');

%Discrete Action Space Configuration - Must be reflected in the Step
%function as well.

kpRange = -0.255:0.01:-0.005;
setptsRange = ceil(PorePressure/1E5):6:FracturePressure/1E5;

C = {kpRange,setptsRange};
D = C;
[D{:}] = ndgrid(C{:});
Z = cell2mat(cellfun(@(m)m(:),D,'uni',0));

Act = transpose(num2cell(Z,2));

ActionInfo = rlFiniteSetSpec(Act);


ActionInfo.Name = 'Controller Action';
env = rlFunctionEnv(ObservationInfo,ActionInfo,'OneshotmpdStep','
    mpdResetFunction');


```

A.8 Code for PI controller

This code is for the PI controller used in the implementation. It is provided by the OpenLab.

```
classdef PIcontroller < handle
% ---------------------------------------------
% Description: Simple PI controller class
%
% Create PIcontroller object:
%   Constructor: PI = PIcontroller(Kp, Ki, Ts, Reference,
    InitialChokeOpening);
%     Kp = Proportional gain
%     Ki = Integral gain
%     Ts = Time constant
%     Reference = Reference value / setpoint
%     InitialChokeOpening = Initial choke opening
%
% Methods:
%   Reset( Reference, InitialOutput) - Reset controller
%     Reference = Reference value / setpoint
%     InitialOutput = Output value to initialize to
%
%   Output = GetOutput( Measured ) - Get output value
%     Measured - Measure value
%     Output - Output value
%
% Created: 2017-05-06 by Sonja Moi
% History:
% ---------------------------------------------------------------

    properties (Access = public)
        Kp;
        Ki;
        Ts;
        Reference;
    end

    properties (Access = private)
        MeasuredLast;
        OutputLast;
        Error;
        ErrorLast;

        ResetStatus = true;
        Measured;
        Output;
    end

    methods
        function this = PIcontroller(Kp, Ki, Ts, Reference,
    InitialOutput)
            this.Kp = Kp;
            this.Ki = Ki;
            this.Ts = Ts;
            this.Reference = Reference/1E5;
            this.OutputLast = InitialOutput;
        end

        function this = Reset(this, Reference, InitialOutput)
            this.ResetStatus = true;
            this.Reference = Reference/1E5;
            this.OutputLast = InitialOutput;
        end

        function Output = GetOutput(this, Measured)
            this.Measured = Measured/1E5;
            if this.ResetStatus
                this.Error=(this.Reference-this.Measured);
                this.ErrorLast=this.Error;

                this.ResetStatus = false;
            else
                this.Error=(this.Reference-this.Measured);
                this.ErrorLast=(this.Reference-this.MeasuredLast);
            end

            this.Output = this.OutputLast + (this.Kp+this.Ki*this.Ts
    /2)*this.Error - (this.Kp-this.Ki*this.Ts/2)*this.ErrorLast;

            if this.Output>1
                this.Output=1;
            end
            if this.Output<0
                this.Output=0;
            end

            this.MeasuredLast = this.Measured;
            this.OutputLast = this.Output;
            Output = this.Output;
        end
    end
end
```

A.9 Code for Reward function for BHP Setpoint

This code is for the basic reward function for the pressure set-point suggestion.

```
function [Reward] = Reward_PBH(Pp, Pf, Depth, Pbh, depth)
% Reward function for bottom hole pressure Pbh @ depth
% Pressure window is defined by Pp and Pf in the Depth range

% Determine the Pp and Pf at given depth
Pp_depth = interp1(Depth, Pp, depth);
Pf_depth  = interp1(Depth, Pf, depth);

```

```matlab
Pp_depth = (8.345*Pp_depth)*(0.051948*depth*3.28) *6894.76/100000 ; % s.g to ppg, then psi then to Pascals
Pf_depth = (8.345*Pf_depth)*(0.051948*depth*3.28) * 6894.76/100000 ; % Same to Pascals

% Calculate the reward
% use a decrease line for the rewards as rw_max at Pp and rw_min at Pf
rw_max = 100000;  %max reward @ Pbh = Pp
rw_min = 0; % min reward @ Pbh --> Pf
rw_neg = -1000000; % punishment for out of window

if Pbh < Pp_depth
    reward = rw_neg;
elseif Pbh >= Pf_depth
    reward = rw_neg;
else
    reward = rw_max - (Pbh - Pp_depth)./(Pf_depth - Pp_depth).*(rw_max - rw_min);
end

Reward = reward;
end
```

A.10 Code for collecting data for Multi-armed bandit problem

This code is implemented in MATLAB and is used to calculate the corresponding reward of every possible action in the action space, the action space becomes bigger and smaller based on the difference between the pore and fracture pressure. The overshoot reward (penalty) in the Code line 131 is set to zero here but normally it is set to -250 if oscillations are desired to be minimized. The reward function from A.9 is used here.

```matlab
1  tableOfPressures = readtable('MPD TEST v6-geopressure.csv','NumHeaderLines',1);
2  InitialBitDepth = 1700;
3  Depths = table2array(tableOfPressures(:,1));
4  InitialDepthRange = 1750:10:2200;
5  
6  InitialChokeOpening = 1;
7  
8  for idr = 1:length(InitialDepthRange)
9  
10     InitialBitDepth = InitialDepthRange(idr);
```

```
 PP = interp1(table2array(tableOfPressures(:,1)),table2array(tableOfPressures(:,2)),InitialBitDepth) %1915 comes from geopressures table
 FP = interp1(table2array(tableOfPressures(:,1)),table2array(tableOfPressures(:,3)),InitialBitDepth); %2046 comes from geopressures table
 %https://www.sciencedirect.com/topics/engineering/formation-pore-pressure -
 %Always recheck the depth value multiplied here
 PorePressureLimit = (8.345*PP)*(0.051948*InitialBitDepth*3.28)*6894.76;  % s.g to ppg, then psi then to Pascals
 FracturePressureLimit = (8.345*FP)*(0.051948*InitialBitDepth*3.28) *6894.76 ; % Same to Pascals

 kpRange = -0.251:0.01:-0.001;
 setptsRange = (ceil(PorePressureLimit/1E5)+3):6:FracturePressureLimit/1E5;
 C = {kpRange,setptsRange};
 D = C;
 [D{:}] = ndgrid(C{:});
 Z = cell2mat(cellfun(@(m)m(:),D,'uni',0));
 Act = transpose(num2cell(Z,2));

 for idx = 1:length(Act)
     % Add to the matlab path the folder where this file is located and all the
     % subfolders
     actions =  cell2mat(Act(idx));

     addpath(genpath(pwd));

     IdentityServerURL = 'https://live.openlab.app/';

     [username, api_key, license_guid] = GetLoginData();

     ConfigurationName = 'MPD TEST v6';
     SimulationName = 'Flow sweep with bhp control (Matlab)';
     InitialBitDepth = InitialDepthRange(idr); % [m]
     InitialTopOfStringPosition = 20;

     % Create simulation object
     try
         Sim = OpenLabClient(IdentityServerURL, username,api_key,license_guid,ConfigurationName,SimulationName,InitialBitDepth,InitialTopOfStringPosition);
     catch err
         idx = idx-1;
     end
     % Ramp settings
     RampIndex = 1;

```

```
        higherRampValue = 2800 + (2900−2800) .* rand(1,1);
        lowerRampValue = higherRampValue − 400;

        RampValuesDown = (higherRampValue:−50:lowerRampValue)/60000; % [m3/sec]
        %RampValuesUp = (lowerRampValue:50:higherRampValue)/60000;  % [m3/sec]
        RampValues = [RampValuesDown]; % [m3/sec]
        RampStepDuration = 30; % [sec]
        RampStartTime = 100; % [sec]
        RampTimeSteps = RampStartTime:RampStepDuration:length(RampValues)*RampStepDuration−1+RampStartTime;

        % Time steps to simulate
        MaxTimeSteps = (RampTimeSteps(end)+100);

        % Controller settings
        Kp = actions(1); Ki = Kp/13; Ts = 1;
        ReferenceBHPPressure = actions(2) * 1E5; % [Pa]
        InitialChokeOpening = 1; % [closed: 0, open: 1]
        PI = PIcontroller(Kp, Ki, Ts, ReferenceBHPPressure, InitialChokeOpening); % Create PI controller object, pressure in [Pa]

        Reward = Reward_PBH(table2array(tableOfPressures(:,2)), table2array(tableOfPressures(:,3)), table2array(tableOfPressures(:,1)), actions(2), InitialBitDepth);

        lol = Sim.BitDepth;
        if Sim.IsOK
            try
                BHP_arr= [];
                for timeStep = 1 : MaxTimeSteps
                    tStartStep = tic;

                    if timeStep >= RampStartTime % Flow sweep and PI control of the choke
                        if (RampIndex < length(RampTimeSteps) && RampIndex < length(RampValues))
                            if (timeStep >= RampTimeSteps(RampIndex) && timeStep < RampTimeSteps(RampIndex + 1))
                                FlowRateIn = RampValues(RampIndex);
                                if (timeStep == RampTimeSteps(RampIndex + 1) − 1)
                                    RampIndex = RampIndex + 1;
                                end
                            end
                        elseif (RampIndex == length(RampTimeSteps) && RampIndex == length(RampValues))
                            FlowRateIn = RampValues(RampIndex);
                        end


                    % Reset PI controller before usage, set
reference value and initial output (= initial
                    % choke opening)
                    PI.Reset(ReferenceBHPPressure,Sim.
ChokeOpeningStatus);


                    ChokeOpening  = PI.GetOutput(Sim.
BottomHolePressure); % Get choke opening from PI controller

                else % Constant flow rate and choke opening
                    FlowRateIn = RampValues(1);
                    ChokeOpening = InitialChokeOpening;
                end

                % Set setpoints to the simulator
                Sim.FlowRateIn = FlowRateIn;  %  [m3/sec]
                Sim.ChokeOpening = ChokeOpening;  %   [0-1]
                Sim.ChokePumpFlowRateIn = 0/60000;  %  [m3/sec]
                Sim.TopOfStringVelocity = 0;  %  [m/sec]
                Sim.SurfaceRPM = 0/60;  %  [revolutions per sec.]
                Sim.ROP = 0/3600;  %  [m/sec]

                % Step simulator
                try
                    Sim.Step();
                catch err
                    timeStep = timeStep-1;
                end

                if ~Sim.IsOK % Simulator fails
                    break;
                end

                %display(Sim.SPP);
                %display(Sim.ChokePressure);
                display(InitialBitDepth);
                if (abs((Sim.BottomHolePressure -
ReferenceBHPPressure)/ReferenceBHPPressure) > 0.01)
                    Reward = Reward - 0;
                    disp(Reward);
                    disp("Overshoot penality in");
                else
                    Reward = Reward + 0;
                end

            end
            display(Sim.ChokeOpening);
            InitialChokeOpening = Sim.ChokeOpening;
```

```
                    fid1 = fopen("Data_record.txt",'a+');
                    fprintf(fid1,'%f %f %f %f %f %f %f, \n ',
    InitialBitDepth,ReferenceBHPPressure,Kp, Reward,PorePressureLimit,
    FracturePressureLimit, InitialChokeOpening);
                    fclose(fid1);

                    % Stop simulation process
                    Sim.Stop;
                    Sim.Delete;
               catch err
                    % Stop simulation
                    Sim.Stop;
                    Sim.Delete;
               end


          end

     end

end
```

A.11 Code for implementing Epsilon-greedy on the data collected in Appendix A.10

This code is responsible for implementing the epsilon-greedy algorithm on the multi-armed bandit problem at hand, the data used in this code is derived from appendix A.10. The implementation is inspired from the work of Steve Roberts [47].

```
# -*- coding: utf-8 -*-
"""
@author: muham
"""

# LIBRARIES

import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
import matplotlib.widgets as widgets
from plotly.graph_objs.scatter import Line
import plotly.graph_objs as go
from plotly.subplots import make_subplots

```

```

#FUNCTIONS AND CLASSES USED

def plots(n_rows,n_cols,titles ,y_axis,x_axis,plot_together = False):

    """
    Function for plotting verything together together

    """

    if (plot_together == False):
        # initial subplot with two traces
        fig = make_subplots(rows=n_rows, cols=n_cols,shared_yaxes=
    True,
                                    subplot_titles=titles)

        for i in range(n_cols):

            fig.add_trace(
                go.Scatter(x=x_axis[i], y=y_axis,name=titles[i],
    line = Line({'width': 1})),
                row=1, col=i+1
            )
    else:
        None

    fig.update_yaxes(autorange="reversed")
    fig.update_layout(height=1000, width=1800, title_text="Managed
    Pressure Drilling")
    fig.show()
    #fig.write_image("fig1.png")
    #Open an HTML File

    with open('Withreward.html', 'a') as f:
        f.write(fig.to_html(full_html=False, include_plotlyjs='cdn'))

class eps_bandit:
    '''
    epsilon-greedy k-bandit problem

    Inputs
    ======================================================
    k: number of arms - actions in our case (int)
    eps: probability of random action 0 < eps < 1 (float)
    iters: number of steps (int)
    mu: set the average rewards for each of the k-arms.
        Set to "random" for the rewards to be selected from
        a normal distribution with mean = 0.
        Set to "sequence" for the means to be ordered from
        0 to k-1.
        Pass a list or array of length = k for user-defined
        values.
    '''

    def __init__(self, k, eps, iters, mu='random'):

        # Number of arms
        self.k = k
        # Search probability
        self.eps = eps
        # Number of iterations
        self.iters = iters
        # Step count
        self.n = 0
        # Step count for each arm
        self.k_n = np.zeros(k)
        # Total mean reward
        self.mean_reward = 0
        self.reward = np.zeros(iters)
        # Mean reward for each arm
        self.k_reward = np.zeros(k)

        if type(mu) == list or type(mu).__module__ == np.__name__:
            # User-defined averages
            self.mu = np.array(mu)
        elif mu == 'random':
            # Draw means from probability distribution
            self.mu = np.random.normal(0, 1, k)
        else:
            # Increase the mean for each arm by one
            self.mu = mu

    def pull(self):

        """
        The action pair selecting function or here we call it pulling
    the arm

        """

        # Generate random number
        p = np.random.rand()
        if self.eps == 0 and self.n == 0:
            a = np.random.choice(self.k)
        elif p < self.eps:
            # Randomly select an action
            a = np.random.choice(self.k)
        else:
            # Take greedy action
            a = np.argmax(self.k_reward)

        reward = np.random.normal(self.mu[a], 1)
```

```python

        # Update counts
        self.n += 1
        self.k_n[a] += 1

        # Update total
        self.mean_reward = self.mean_reward + (
            reward - self.mean_reward) / self.n

        # Update results for a_k
        self.k_reward[a] = self.k_reward[a] + (
            reward - self.k_reward[a]) / self.k_n[a]

    def run(self):

        """
        Using iterations in run functions to get a stable measure of reward.
        Not really used in MPD as Reward function does this job
        """
        for i in range(self.iters):
            self.pull()
            self.reward[i] = self.mean_reward

    def reset(self):

        """
        Resets results while keeping settings
        """
        self.n = 0
        self.k_n = np.zeros(k)
        self.mean_reward = 0
        self.reward = np.zeros(iters)
        self.k_reward = np.zeros(k)

def epsilon_method(data):

    """
    Only epsilon = 0.1 is kept and rest are omitted in final testing,
    Code lines can be uncommented to test other values of epsilon
    This should be done carefully.

    This function runs the epsilon-greedy method for the selected data

    For the case of MPD this means a seclected depth
    """
    socket_order = data.index.to_list()
    socket_means = data["Reward"].to_list()
    #socket_means = [word[:8] for word in socket_means]
    socket_means = np.array(socket_means).astype(float)
    lo, hi = min(socket_means), max(socket_means);
    return_array = []


    for i in socket_means:
        return_array.append((i - lo) / (hi-lo))
    socket_means = return_array

    k = len(socket_means)
    iters = 10000

    #eps_0_rewards = np.zeros(iters)
    #eps_01_rewards = np.zeros(iters)
    eps_1_rewards = np.zeros(iters)
    #eps_2_rewards = np.zeros(iters)
    #eps_5_rewards = np.zeros(iters)

    episodes = 1000
    # Run experiments
    for i in range(episodes):
        # Initialize bandits
        #eps_0 = eps_bandit(k, 0, iters ,mu = socket_means)
        #eps_01 = eps_bandit(k, 0.01, iters, eps_0.mu.copy())
        eps_1 = eps_bandit(k, 0.1, iters, mu = socket_means)
        #eps_2 = eps_bandit(k, 0.2, iters, eps_0.mu.copy())
        #eps_5 = eps_bandit(k, 0.5, iters, eps_0.mu.copy())

        # Run experiments
        #eps_0.run()
        #eps_01.run()
        eps_1.run()
        #eps_2.run()
        #eps_5.run()

        # Update long-term averages
        #eps_0_rewards = eps_0_rewards + (
        #    eps_0.reward - eps_0_rewards) / (i + 1)
        #eps_01_rewards = eps_01_rewards + (
        #    eps_01.reward - eps_01_rewards) / (i + 1)
        eps_1_rewards = eps_1_rewards + (
            eps_1.reward - eps_1_rewards) / (i + 1)
        #eps_2_rewards = eps_2_rewards + (
        #    eps_2.reward - eps_2_rewards) / (i + 1)
        #eps_5_rewards = eps_5_rewards + (
        #    eps_5.reward - eps_5_rewards) / (i + 1)


    plt.figure(figsize=(12,8))
    #plt.plot(eps_0_rewards, label="$\epsilon=0$ (greedy)")
    #plt.plot(eps_01_rewards, label="$\epsilon=0.01$")
    plt.plot(eps_1_rewards, label="$\epsilon=0.1$")
    #plt.plot(eps_2_rewards, label="$\epsilon=0.2$")
    #plt.plot(eps_5_rewards, label="$\epsilon=0.5$")
    plt.legend(bbox_to_anchor=(1.3, 0.5))

    plt.xlabel("Iterations")
    plt.ylabel("Average Reward")
    plt.title("Average $\epsilon-greedy$ Rewards after " + str(
episodes)
        + " Episodes")
    plt.show()


    iters = 10000
    #eps_0_rewards = np.zeros(iters)
    #eps_01_rewards = np.zeros(iters)
    eps_1_rewards = np.zeros(iters)
    #eps_2_rewards = np.zeros(iters)
    #eps_5_rewards = np.zeros(iters)

    #eps_0_selection = np.zeros(k)
    #eps_01_selection = np.zeros(k)
    eps_1_selection = np.zeros(k)
    #eps_2_selection = np.zeros(k)
    #eps_5_selection = np.zeros(k)

    episodes = 1000
    # Run experiments
    for i in range(episodes):
        # Initialize bandits
        #eps_0 = eps_bandit(k, 0, iters, mu = socket_means)
        #eps_01 = eps_bandit(k, 0.01, iters, eps_0.mu.copy())
        eps_1 = eps_bandit(k, 0.1, iters, mu = socket_means)
        #eps_2 = eps_bandit(k, 0.2, iters, eps_0.mu.copy())
        #eps_5 = eps_bandit(k, 0.5, iters, eps_0.mu.copy())

        # Run experiments
        #eps_0.run()
        #eps_01.run()
        eps_1.run()
        #eps_2.run()
        #eps_5.run()

        # Update long-term averages
        #eps_0_rewards = eps_0_rewards + (
        #    eps_0.reward - eps_0_rewards) / (i + 1)
        #eps_01_rewards = eps_01_rewards + (
        #    eps_01.reward - eps_01_rewards) / (i + 1)
        eps_1_rewards = eps_1_rewards + (
            eps_1.reward - eps_1_rewards) / (i + 1)
        #eps_2_rewards = eps_2_rewards + (
        #    eps_2.reward - eps_2_rewards) / (i + 1)
        #eps_5_rewards = eps_5_rewards + (
        #    eps_5.reward - eps_5_rewards) / (i + 1)

        # Average actions per episode
        #eps_0_selection = eps_0_selection + (
        #    eps_0.k_n - eps_0_selection) / (i + 1)
        #eps_01_selection = eps_01_selection + (
        #    eps_01.k_n - eps_01_selection) / (i + 1)
        eps_1_selection = eps_1_selection + (
            eps_1.k_n - eps_1_selection) / (i + 1)
        #eps_2_selection = eps_2_selection + (
        #    eps_2.k_n - eps_2_selection) / (i + 1)
        #eps_5_selection = eps_5_selection + (
        #    eps_5.k_n - eps_5_selection) / (i + 1)

    plt.figure(figsize=(12,8))
    #plt.plot(eps_0_rewards, label="$\epsilon=0$ (greedy)")
    #plt.plot(eps_01_rewards, label="$\epsilon=0.01$")
    plt.plot(eps_1_rewards, label="$\epsilon=0.1$")
    #plt.plot(eps_2_rewards, label="$\epsilon=0.2$")
    #plt.plot(eps_5_rewards, label="$\epsilon=0.5$")

    for i in range(k):
        plt.hlines(eps_1.mu[i], xmin=0,
                   xmax=iters, alpha=0.5,
                   linestyle="--")

    plt.legend(bbox_to_anchor=(1.3, 0.5))
    plt.xlabel("Iterations")
    plt.ylabel("Average Reward")
    plt.title("Average $\epsilon-greedy$ Rewards after " +
         str(episodes) + " Episodes")
    plt.show()

    bins = np.linspace(0, k-1, k)

    plt.figure(figsize=(12,8))
    #plt.bar(bins, eps_0_selection,
    #        width = 0.33, color='b',
    #        label="$\epsilon=0$")
    #plt.bar(bins+0.33, eps_01_selection,
    #        width=0.33, color='g',
    #        label="$\epsilon=0.01$")
    plt.bar(bins+0.66, eps_1_selection,
            width=0.33, color='r',
            label="$\epsilon=0.1$")

    #plt.bar(bins+0.66, eps_1_selection,
    #        width=0.33, color='r',
    #        label="$\epsilon=0.2$")

    #plt.bar(bins+0.66, eps_1_selection,
    #        width=0.33, color='r',
    #        label="$\epsilon=0.5$")

    plt.legend(bbox_to_anchor=(1.2, 0.5))
    plt.xlim([0,k])
    plt.title("Actions Selected by Each Algorithm")
    plt.xlabel("Action")
    plt.ylabel("Number of Actions Taken")
    plt.show()

    #opt_per = np.array([eps_0_selection, eps_01_selection,
    #                    eps_1_selection, eps_2_selection,
   eps_5_selection]) / iters * 100

    opt_per = np.array([eps_1_selection]) / iters * 100

    #df = pd.DataFrame(opt_per, index=['$\epsilon=0$',
    #    '$\epsilon=0.01$', '$\epsilon=0.1$' , '$\epsilon=0.2$' , '$\
   epsilon=0.5$' ],
    #                  columns=["a = " + str(x) for x in range(0, k)])

    df = pd.DataFrame(opt_per, index=['$\epsilon=0.1$'],
                      columns=["a = " + str(x) for x in range(0, k)])

    print("Percentage of actions selected:")
    df.max(axis = 1)
    arm_index = df.idxmax(axis = 1).to_list()

    return arm_index


#Cleaning the data - Preprocessing data
columns = ["Depth", "Reference BHP", "Kp", "Reward", "Pore Pressure",
     "Fracture Pressure","ChokeValveOpening"]
df = pd.read_table('Data_continuouschoke.txt',names=columns,sep=" ",
   lineterminator='\n')
df.reset_index(drop=False, inplace=True)
df = df.drop('level_0', 1)
df.columns = ["Depth", "Reference BHP", "Kp", "Reward", "Pore
   Pressure", "Fracture Pressure","ChokeValveOpening","extra"]
df = df.drop('extra', 1)
data = df.dropna()
data["Depth"] = pd.to_numeric(data["Depth"], downcast="float")
Depths= np.arange(1780,2130,10)
Data_frames = []

#Dividing data into Depths and every depth gets a dataframe
for d in Depths:
    df = data.loc[data['Depth'] == d]
    df.reset_index(inplace = True)
    Data_frames.append(df)

#Printing the selected arm at every depth in each iteration
arms = []
Data_frames[9]
for frame in Data_frames:
    arms.append(epsilon_method(frame))
    print(arms[-1])

#Extracting the index from the string
def indexstringextraction(item):
    string = item[0]
    from itertools import groupby
    arm_number = [''.join(v) for k,v in groupby(string,str.isdigit)
    ][-1]

    return int(arm_number)

map_object = map(indexstringextraction, arms)

new_list_arms = list(map_object)

class SnaptoCursor(object):
    """
    Adding cursor to the plot
    """
    def __init__(self, ax, x, y):
        self.ax = ax
        self.ly = ax.axvline(color='k', alpha=0.2)  # the vert line
        self.marker, = ax.plot([0],[0], marker="o", color="crimson",
    zorder=3)
        self.x = x
        self.y = y
        self.txt = ax.text(0.7, 0.9, '')

    def mouse_move(self, event):
        if not event.inaxes: return
        x, y = event.xdata, event.ydata
        indx = np.searchsorted(self.x, [x])[0]
        x = self.x[indx]
        y = self.y[indx]
        self.ly.set_xdata(x)
        self.marker.set_data([x],[y])
        self.txt.set_text('x=%1.2f, y=%1.2f' % (x, y))
        self.txt.set_position((x,y))
        self.ax.figure.canvas.draw_idle()

#Initializing empty lists to store required parameters
Plotlist = []
Ref_BHP = []
Fracture_Pressure = []
```

```
Pore_Pressure = []
Kp = []
Depth = []
Choke = []

for i in range(len(new_list_arms)):
     Plotlist.append(Data_frames[i].iloc[new_list_arms[i]])
    Kp.append(Plotlist[-1].loc["Kp"])
    Ref_BHP.append(Plotlist[-1].loc["Reference BHP"])
    Pore_Pressure.append(Plotlist[-1].loc["Pore Pressure"])
    Fracture_Pressure.append(Plotlist[-1].loc["Fracture Pressure"])
    Depth.append(Plotlist[-1].loc["Depth"])
    Choke.append(Plotlist[-1].loc["ChokeValveOpening"])

#Preparation of data from
conversion = 100000 #Conversion to bar

#Fracture_Pressure[:] = [x[:-1] for x in Fracture_Pressure]
Fracture_Pressure = list(map(float, Fracture_Pressure))
Fracture_Pressure = np.array(Fracture_Pressure)/conversion
Fracture_Pressure = np.ceil(Fracture_Pressure)

#Pore_Pressure[:] = [x[:9] for x in Pore_Pressure]
Pore_Pressure = list(map(float, Pore_Pressure))
Pore_Pressure = np.array(Pore_Pressure)/conversion
Pore_Pressure = np.floor(Pore_Pressure)

Ref_BHP = list(map(float, Ref_BHP))
Ref_BHP[:] = [float(x) /conversion for x in Ref_BHP]
Ref_BHP = np.array(Ref_BHP)
Ref_BHP = np.ceil(Ref_BHP)

Kp = list(map(float, Kp))

Depth = list(map(float, Depth))


#Pore #Fracture #Reference BHP #Kp #Choke
plot_list=[]

plot_list.append(Pore_Pressure)
plot_list.append(Fracture_Pressure)
plot_list.append(Ref_BHP)
plot_list.append(Kp)
plot_list.append(Choke)

plots(1,5,["Pore Pressure","Fracture Pressure","Reference Bottom hole
    pressure","Kp - Propotionality constant for PID","Choke Opening"
   ],Depth,plot_list)

#indexstringextraction(arms[39])
```