ARTHUR MORIGGI PIMENTA

Deep Reinforcement Learning for Simulated Autonomous Drone Control

São Paulo

2020

ARTHUR MORIGGI PIMENTA

Deep Reinforcement Learning for Simulated Autonomous Drone Control

Trabalho apresentado à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro Mecânico Mecatrônico.

Universidade de São Paulo – USP

Escola Politécnica

Engenharia Mecatrônica

Orientador: Eduardo Aoun Tannuri

São Paulo

2020

ARTHUR MORIGGI PIMENTA

Deep Reinforcement Learning for Simulated Autonomous Drone Control/ A. M. Pimenta. – São Paulo, 2020-

65 p. : il. (algumas color.) ; 30 cm.

Orientador: Eduardo Aoun Tannuri

Trabalho de Conclusão de Curso – Universidade de São Paulo – USP
Escola Politécnica
Engenharia Mecatrônica, 2020.

1. Deep Reinforcement Learning. 2. Control. 3. Drone. 4. PPO. I. Eduardo Aoun Tannuri. II. Universidade de São Paulo. III. Escola Politécnica . IV. Pimenta, Arthur Moriggi.

CDU 02:141:005.7

ARTHUR MORIGGI PIMENTA

Deep Reinforcement Learning for Simulated Autonomous Drone Control

Trabalho apresentado à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro Mecânico Mecatrônico.

Trabalho aprovado. São Paulo, 7 de dezembro de 2020:

Eduardo Aoun Tannuri
Orientador

Professor
Convidado 1

Professor
Convidado 2

São Paulo
2020

Agradecimentos

Primeiramente a Deus que me capacita, me fortalece e me inspira a ser melhor todos os dias.

À minha família e amigos que, com seu apoio, tornaram a elaboração deste trabalho mais branda. Um destaque especial à Kamylle Barreto, minha companheira, que me ofereceu toda estrutura e suporte emocional necessário à execução deste trabalho.

Ao professor Eduardo Aoun Tannuri pela orientação, apoio e confiança.

"Não vos amoldeis às estruturas deste mundo,
mas transformai-vos pela renovação da mente,
a fim de distinguir qual é a vontade de Deus:
o que é bom, o que Lhe é agradável, o que é perfeito.
(Bíblia Sagrada, Romanos 12, 2)

Resumo

Com o aumento recenete no número de aplicações envolvendo drones, algoritmos dedicados ao controle de quadricópteros ganharam relevância. Um dos algoritmos de controle mais comum é o Proporcional, Integrativo e Derivativo, o PID. No entanto, pesquisas recentes afirmam que algoritmos de Deep Reinforcement Learning, como o Proximal Policy Optimization (PPO), apresentam resultados satisfatórios na tarefa de controle, podendo ser ainda melhores que o PID para condições extremas. Nesse contexto, desenvolvemos e comparamos os modelos de PID e PPO com o objetivo de controlar a atitude de um drone. Os ganhos do PID foram calculados através do método Lugar das Raízes e o modelo foi desenvolvido e simulado no software SIMULINK. O modelo PPO foi desenvolvido em python com a biblioteca Stable Baselines e o ambiente de treinamento do agente foi implementado com base nos padrões de ambiente GYM. O conjunto de hiperparâmetros e a função de recompensa utilizados no treinamento do modelo PPO foram inspirados no conjunto utilizado para treinar o ambiente Pendulum disponível no GYM. Os resultados obtidos mostram que em situações que exigem pequenas variações angulares, ambos modelos obtiveram resultados bons e similares, enquanto que em condições extremas, como partir de cabeça para baixo até 30 graus de atitude, o PPO se mostrou consideravelmente melhor e mais robusto do que o PID.

Palavras-chaves: Proximal Policy Optimization, PID, Drone, controle de atitude.

Abstract

As the number of applications involving drones has increased recently, algorithms dedicated to control quadcopters have gained relevance. One of the most common controlling algorithm is the Proportional Integrative and Derivative, the PID. However, Recent research state that Deep Reinforcement Learning algorithms, such as Proximal Policy Optimization (PPO), yields satisfactory results in the controlling task, even better than PID for extreme conditions. In this context, we developed and compared the PID and PPO models in the task of attitude control. The PID gains were calculated with Pole Placement and the model was developed and simulated on the SIMULINK software. The PPO model was developed in python with the library Stable Baselines and the agent's training environment was implemented based on the GYM's standards. The set of hyperparameters and the reward function used on the training of the PPO model were inspired by the set used to train the Pendulum environment from GYM. The results obtained show that in soft conditions, such as small angular variations, both models perform similarly well whereas in extreme conditions, such as starting from upside down to 30 degrees of attitude, the PPO was considerably better and more robust than the PID.

Key-words: Proximal Policy Optmization, PID, Drone, attitude control.

List of Figures

List of Tables

List of abbreviations and acronyms

UAV	Unmanned Aerial Vehicles
PPO	Proximal Policy Optimization
DRL	Deep Reinforcement Learning
PID	Proportional, Integral and Derivative
RL	Reinforcement Learning
NN	Neural Networks
DDGP	Deep Deterministic Gradient Policy
TRPO	Trust Region Policy Optimization
$E - frame$	Earth frame
$B - frame$	Body-fixed frame
MDP	Markov Decision Process
AC	Actor-Critic

List of symbols

θ	Roll angle
ϕ	Pitch angle
ψ	Yaw angle
K_P	Proportional Gain
K_D	Derivative Gain
K_I	Integral Gain
Ω_1	Speed Propeller 1
Ω_2	Speed Propeller 2
Ω_3	Speed Propeller 3
Ω_4	Speed Propeller 4
Ω_{max}	Maximum Speed Propeller
Ω_h	Hover Speed
x_c	Coordinate x of the mass center of the Drone
y_c	Coordinate y of the mass center of the Drone
z_c	Coordinate z of the mass center of the Drone
τ	Throttle
m	Drone's mass
l	Distance between the drone's center and any of the propellers
J_x	Moments of inertia around the axis x
J_y	Moments of inertia around the axis y
J_z	Moments of inertia around the axis z
d	Drag Factor
g	Gravity acceleration

b	Throttle Factor
F	Force Vector
F_x	Force in x direction
F_y	Force in y direction
F_z	Force in z direction
F_1	Propeller's 1 Force
F_2	Propeller's 2 Force
F_3	Propeller's 3 Force
F_4	Propeller's 4 Force
M	Torque Vector
M_x	Torque in x direction
M_y	Torque in y direction
M_z	Torque in z direction
Ω	Angular Velocities Vector
J	Inertia Matrix
J_{xx}	Inertia around x axis
J_{yy}	Inertia around y axis
J_{zz}	Inertia around z axis
I_{3x3}	Identity Matrix
0_{3x3}	Null Matrix
$U(s)$	Controller Signal in Laplace Domain
$E(s)$	Error Signal in Laplace Domain
U_x	Controller Signal of the x moment
U_y	Controller Signal of the y moment
U_z	Controller Signal of the z moment
MP	Percentage of overshoot

t_s	Settling time
ζ	Damping Factor
w_n	Natural angular frequency
j	Imaginary number
z_{c1}	Position of the PID's Zero 1
z_{c2}	Position of the PID's Zero 2
p_c	Position of the PID's Pole
p_1	Position of the old Pole 1
p_2	Position of the old Pole 2
θ_1	Pole Angle 1
θ_2	Pole Angle 2
θ_c	PID Angle Pole
ϕ_{c1}	PID Angle Zero 1
ϕ_{c2}	PID Angle Zero 2
S	Set of possible process states
A	Set of actions that interfere in the process
T	Transferring function
R	Reward Function
γ	Discount Factor
r_k	Reward in the step k
π	Policy
V_π	Value Function
w	Neuron Network weight
$\hat{q}$	Critic Neuron Network
Θ	Set of the Actor weights
α	Learning Rate of the Actor

β	Learning Rate of the Critic
A	Advantage Function
$r_t(\Theta)$	Policies ratio
L_t^{PPO}	PPO Loss function
T_s	Time step

Contents

1 INTRODUCTION

The first Unmanned Aerial Vehicles (UAV) were developed during the world wars, in the twentieth century, where their main function was basically launching missiles for military matters. After this period, the UAVs started to be used to perform tasks within dangerous or limited access environments for humans (AUSTIN, 2010). Among the several UAVs designs, the quadcopter has earned relevance due to its simple configuration, applicability as well as low cost. All these advantages have turned the quadcopters very useful for agriculture, mining and monitoring in general. In order to make the UAV able to perform these activities by itself, control algorithms have been developed in the past few years (AL., 2018; HWANGBO et al., 2017; YU; PALEFSKY-SMITH; BEDI, 2016). In this regard, this work intends to evaluate the Proximal Policy Optimization (PPO) (AL, 2017) update rule to control a simulated attitude's drone via Deep Reinforcement Learning (DRL) as well as compare its performance to a conventional PID control which will be built based on Newton's laws.

1.1 Related Work

Different types of machine learning algorithms have been developed in order to control autonomous vehicles rather than using classic approaches such as Proportional, Integral and Derivative (PID) control systems. Below we discuss some of them and subsequently present our project.

1.1.1 Controlling Autonomous Vehicle with Reinforcement Learning

Recently the number of Machine Learning applications has increased considerably. In this scenario, autonomous vehicles have gained an important relevance [2,3,4]. However, the control system of an autonomous vehicle is based, in most cases, on the classic control theory, the PID. Although the PID performs satisfactorily in stable environments, in unpredictable environments it doesn't achieve good results. Current researches (AL., 2018) state that, for this type of environment, Machine Learning algorithms could be the best approach. Among these algorithms, Reinforcement Learning has been used as a reasonable solution [2,3,4]. Additionally, in order to tune the PID's parameters it is required previous knowledge of the drone's dynamic model which is not a trivial task, whereas most of the RL algorithms, including PPO, are not model based.

1.1.2 Reinforcement Learning for Quadcopter Control

The quadcopter is one of the most important autonomous vehicles currently, and its control system is a great challenge due to its 6 degrees of freedom and its dynamic complexity. Besides, quadcopters often perform flights in very harsh conditions which makes the system even more complicated. Therefore RL algorithms have been used for this matter. The first use of reinforcement learning in quadcopter control was presented by (WASLANDER et al., 2016) for altitude control. The authors in (WASLANDER et al., 2016) state that there are several nonlinear disturbances that affect the altitude control loop such as the complex airflow induced by the four interacting rotors, ground effect and battery discharge dynamics. Because of these disturbances the classical linear techniques fail to provide altitude stability. Reinforcement Learning control was used for accommodating the nonlinear disturbances in altitude control and the results showed great improvements over classical control techniques. Furthermore, RL is also capable of controlling even more complicated tasks. For instance, in (HWANGBO et al., 2017) the authors demonstrated with simulation as well as with a real drone that stabilization from an upside-down throw can be performed satisfactorily with RL. According to (AL., 2018), the control system of autonomous vehicles is typically composed of an "inner-loop" that provides stability and an "outer-loop" that is in charge of mission-levels tasks. Both (WASLANDER et al., 2016) and (HWANGBO et al., 2017) are examples of "outer-loop" applications and (AL., 2018) is an "inner-loop" example once RL was used for attitude control. For (AL., 2018) as well as for (HWANGBO et al., 2017) the approach used was combining neural networks (NN) with RL applied in continuous tasks. In (HWANGBO et al., 2017), the authors trained a deterministic policy represented as a NN that provides the best action given the state of the quadrotor. Whereas in (AL., 2018) the authors evaluated and trained three different policies: Deep Deterministic Gradient Policy (DDGP), Trust Region Policy Optimization (TRPO) and Proximal Policy Optimization (PPO) where the authors found that PPO, the most recent one and the chose method for this work, converges way faster than DDGP and TRPO.

2 AIMS

The primary aim of this work is to properly control a simulated drone's attitude, i.e, the roll, pitch and yaw angles, using a Deep Reinforcement Learning algorithm with the Proximal Policy Optimization as an update rule. Furthermore, this work also intends to compare the performance of the PPO model with the classic PID model.

3 METHODOLOGY

This work develops and compares two different approaches for drone's attitude control: A deep reinforcement learning model, the PPO, and the PID control. The increase in the machine learning applications in the past few years yielded several powerful open source tools such as GYM and Stable Baselines. The GYM package is a toolkit which provides a bunch of environments designed to train Reinforcement Learning agents. As most of RL models are designed to learn in a GYM environment, this library also provides the possibility to custom your own environment and then be able to use the many of the open source models available. Stable Baselines for instance is a set of implementations of Reinforcement Learning (RL) algorithms based on OpenAI Baselines. There is a large and active community using these libraries which makes the learning process easier. In this context, we decided to program the PPO model by using the Stable Baselines library and the simulated drone environment following the GYM standard architecture, both in Python.

For the PID control we first design the quadcopter's dynamic model based on Newton's laws, then we apply the Laplace transform on the dynamic equations in order to evaluate the system behaviour in the Laplace Domain. And then we estimate the proportional (KP), derivative (KD) and integrative (KI) constants through the Pole Placement approach.

To evaluate the performance of the PPO approach we'll train two Neurons Networks to estimate the best action to take given the current state and the target attitude. These NNs will be discussed further in the next chapters. The action is then performed on the environment which yields a reward that depends on how good it was that action. In these conditions, the NNs are updated according to the PPO policy. Then this procedure is repeated intensively until the model converges and the agent becomes optimized. Afterwards, the environment will be set to the initial conditions and be put to predict actions and update the states for 10 seconds of simulation. In these conditions it will provide a time response that will be used to compare with the PID result.

To evaluate the PID method we used Simulink. On Simulink we built a closed loop represented by a block diagram that is composed of two main subsystems: The PID and the Drone. On the PID subsystem there are the controller equations that are based on the error. On the Drone subsystem there are the equations that describe the drone's attitude based on the controller signal. The flow goes as follows: Given the drone's state and the target position, the PID computes the error and applies an action on the quadcopter that is the summation of the terms P, D and I. P represents the product between KP and the

error. Similarly, D is the product between KD and the time derivative of the error and I is the product between KI and the time integration of the error. To evaluate the model we compare the time response plot for the both models when step function is applied as input. Three metrics are used to analyse the both methods: the overshoot, the settling time and the steady-state error.

4 REQUIREMENTS

In order to achieve the goal of the work, i.e, the comparison of PID and PPO models for attitude control, we define the requirements presented below. For all of them we are considering zero initial conditions and the target position equals to 30 degrees for Roll, Pitch and Yaw.

R1 (No steady-state error): The response for the both models yields no steady-state error for a step function as input.

R2 (Maximum overshoot): The maximum overshoot for the response of both models is 15% for a step function as input.

R3 (Maximum settling time): The maximum settling time for the both models must be 2 seconds for Roll and Pitch response and 4 seconds for Yaw response.

R4 (Maximum engine speed): The output signal of both models must be compatible with the maximum engine speed.

R5 (Quadcopter's state): The state of the quadcopter is a vector composed by 6 parameters: 3 angles and their time derivatives: $\left\{\phi, \dot{\phi}, \theta, \dot{\theta}, \psi, \dot{\psi}\right\}$ which define the drone's orientation in the space. θ is the angle around the X axis, ϕ is the angle around the Y axis and ψ is the angle around the Z axis.

R6 (Quadcopter's action): The quadcopter action is the four propellers velocity: Ω_1, Ω_2, Ω_3 and Ω_4.

R7 (Reward): The reward function must punish actions that lead to states where the current error is larger than the previous.

R8 (Simulation Step): The time step is constant and equal to 0.01s

R9 (Action interval): The interval between two actions must be fixed and larger than the time the algorithm takes to run.

5 BASIC CONCEPTS

A conventional quadcopter has six degrees of freedom: the angles , and illustrated by figure 1 and the space coordinates of the center of mass xc, yc and zc. However, it is capable to reach a desirable set point just for four degrees of freedom once the movement along the x and y axes depends on the angles. Thus, we define the four principal drone's movements: Roll, Pitch, Yaw and Throttle.

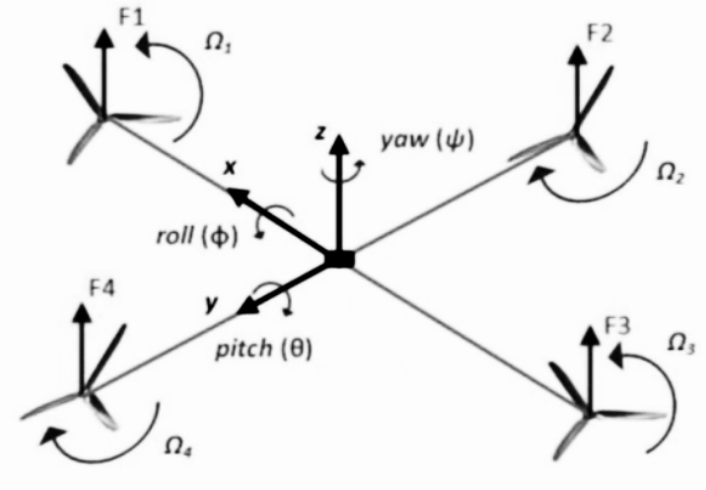


Figure 1 – Drone's Diagram (Sá, 2012)

The Throttle (τ) is in charge of the vertical movement, i.e, along the z axis. It is performed when all the four propellers have the same absolute angular velocity ($\Omega_1 = \Omega_2 = \Omega_3 = \Omega_4$). By increasing and decreasing equally the four engines speed the quadcopter goes up and downwards respectively as illustrated by figure 2, where the arrow's thickness is related to amount of speed.

Figure 2 – Throttle movement (Sá, 2012)

The Roll(θ) is the movement that allows the quadcopter to move along y axis by setting different speeds on propellers 2 and 4 ($\Omega_2 \neq \Omega_4$) whereas 1 and 3 have the same speed ($\Omega_1 = \Omega_3$). In the same way, the Pitch (ϕ) allows the movement along x axis and it is performed by setting different speeds on 1 and 3 ($\Omega_1 \neq \Omega_3$) and the same speed on 2 and 4 ($\Omega_2 = \Omega_4$) as shown by figure 3

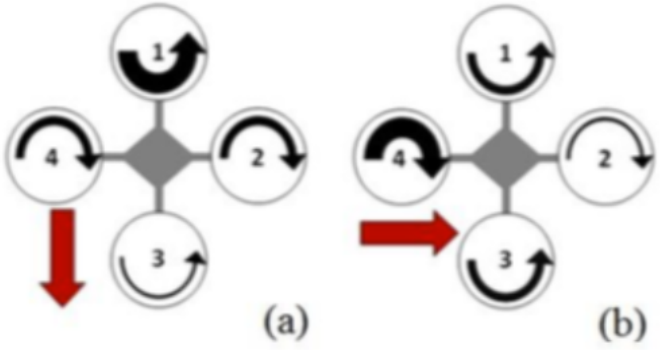


Figure 3 – Pitch (a) and Roll (b) movements (Sá, 2012)

The Yaw (ψ) is the movement and it is shown in the figure 4. It is accomplished by setting on Ω_1 and Ω_3 certain amount of speed ($\Omega_1 = \Omega_3 = \Omega_{13}$) and on 2 and 4 a different amount ($\Omega_2 = \Omega_4 = \Omega_{24} \neq \Omega_{13}$) the quadcopter rotates around the z axis.

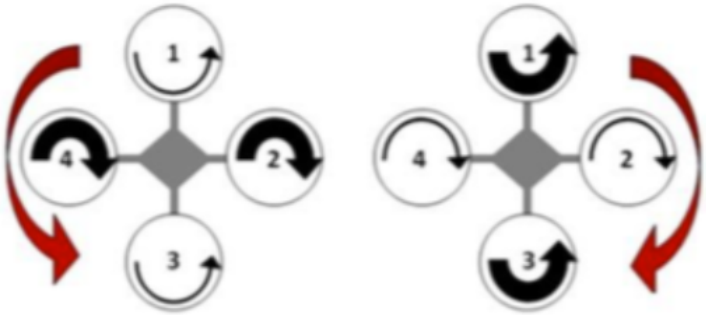


Figure 4 – Movement around z axis (yaw) (Sá, 2012)

6 QUADCOPTER'S PARAMETERS

In order to provide a reliable simulation, we'll build our dynamic model based on a real quadcopter that was constructed by (QUEIROZ, 2014).

Figure 5 – Real Drone (QUEIROZ, 2014)

For this work, the relevant parameters given by (QUEIROZ, 2014) are the mass (m); the distance between the drone's center and any of the propellers (l); the moments of inertia around the axes x (Jx), y (Jy) and z (Jz); The Throttle Factor (b); The Drag Factor (d) and the maximum engine's speed (Ω_{max}). These specifications are shown below:

Parameter	Value
l	0.28 m
J_{xx}	0.01817 Kgm^2
J_{yy}	0.01883 Kgm^2
J_{zz}	0.03572 Kgm^2
b	$1.0927 \times 10^{-5} Ns^2$
d	$3.7343 \times 10^{-7} Nms^2$
m	0.76137 Kg
Ω_{max}	700 Rad/s

Table 1 – Drone's Parameters

7 DYNAMIC MODEL

For the PID model and simulation matters it is quite important to obtain mathematical equations that can describe the system behaviour. Motivated by (BRESCIANI, 2008) we decided to design our dynamic model with the Newton-Euler method. The figure 1 presents a schematic diagram that shows the forces produced by the four engines, the angular velocities of the propellers and the angles we intend to control: Roll, Pitch and Yaw.

Before we apply the Newton-Euler method we need to define two frames: The Earth frame (E-frame) and the Body-fixed frame (B-frame) as shown in figure 6. We placed the B-frame in the center of the drone's mass in order to get a diagonal inertia matrix and thus simplify the calculations.

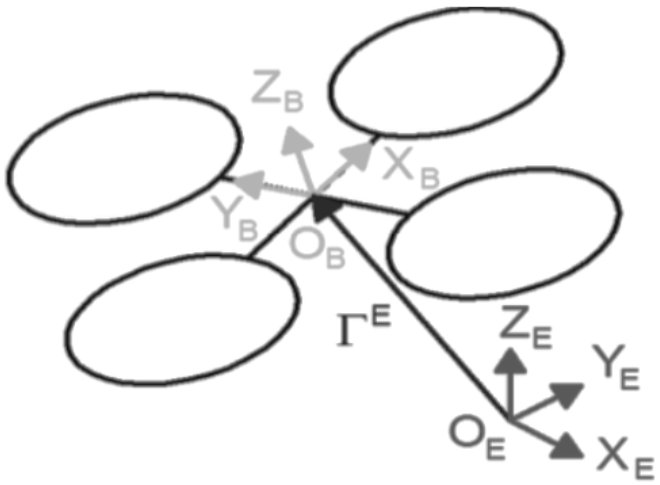


Figure 6 – System's frames (BRESCIANI, 2008)

The Newton-Euler equations state that the summation of all forces applied on a body is equal to the product between the body's mass and the acceleration of the body with respect to an inertial frame of reference. Similarly, for angular movement, the summation of all torques is equal to the product between the body's moment of inertia and the angular acceleration of the body with respect to an inertial frame of reference. In our case, the E-frame is the inertial frame of reference. Thus, by describing the linear and angular acceleration of the center of mass with respect to the E-frame and applying the Newton-Euler method, we end up with the following movement equations (BRESCIANI,

2008):

$$\begin{bmatrix} F \\ M \end{bmatrix} = \begin{bmatrix} \Omega \times (mV) \\ \Omega \times (J\Omega) \end{bmatrix} + \begin{bmatrix} m_{3x3} & 0_{3x3} \\ 0_{3x3} & J \end{bmatrix} \begin{bmatrix} \dot{V} \\ \dot{\Omega} \end{bmatrix} \tag{7.1}$$

Where F is the force vector, M is the torque vector, Ω is the angular velocities vector, J is the inertia matrix, I_{3x3} is the identity matrix and 0_{3x3} is a square matrix composed with just zero values:

$$\begin{gathered} J = \begin{bmatrix} J_{xx} & 0 & 0 \\ 0 & J_{yy} & 0 \\ 0 & 0 & J_{zz} \end{bmatrix}, I_{3x3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, 0_{3x3} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \\ F = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix}, M = \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix}, V = \begin{bmatrix} V_x \\ V_y \\ V_z \end{bmatrix}, \Omega = \begin{bmatrix} \dot{\theta} \\ \dot{\phi} \\ \dot{\psi} \end{bmatrix} \end{gathered} \tag{7.2}$$

However, once that we are interested in attitude control, i.e, controlling the roll (θ), pitch(ϕ) and yaw (ψ), we'll focus on the equation that describes the drone's angular movement, the second row of matrix equation in (7.1):

$$M = \Omega \times (J\Omega) + J\dot{\Omega} \tag{7.3}$$

The first term of this equation, $\Omega \times (J\Omega)$, is known as the gyroscopic effect and, for classical control design, it can be ignored (BOUABDALLAH, 2007). By expanding the equation (7.3) and removing the gyroscopic effect, we get these three main equations:

$$\begin{cases} M_x = J_{xx}\ddot{\phi} \\ M_y = J_{xx}\ddot{\theta} \\ M_x = J_{xx}\ddot{\psi} \end{cases} \tag{7.4}$$

Mx, My and Mz are torques produced by the engine forces F1, F2, F3 and F4 as illustrated by figure 5. Then, we can rewrite the torques as follows:

$$M_x = (F4 - F2)l \tag{7.5}$$

$$M_x = (F3 - F1)l \tag{7.6}$$

Where l is the distance between the quadcopter's center and any propeller's center. For Mz is a bit different. Because the engines 1 and 3 rotate in the same direction they

produce a torque around the z axis that is proportional to $(\Omega_1^2 + \Omega_3^2)$ (BRESCIANI, 2008). Similarly the engines 2 and 4 produce a torque around z proportional to $(\Omega_2^2 + \Omega_4^2)$, however it is the opposite of 1 and 3. Therefore, the Mz can be described as follows:

$$M_z = d(\Omega_2^2 + \Omega_4^2 - \Omega_1^2 - \Omega_3^2) \tag{7.7}$$

Where d is the drag factor. The forces F1, F2, F3 and F4 also are, according with (BRESCIANI, 2008), proportional to the angular velocities 1, 2, 3 and 4 respectively:

$$F1 = b\Omega_1^2 \tag{7.8}$$

$$F2 = b\Omega_2^2 \tag{7.9}$$

$$F3 = b\Omega_3^2 \tag{7.10}$$

$$F4 = b\Omega_4^2 \tag{7.11}$$

Where b is the Throttle factor. We can now combine (7.5), (7.6), (7.8 - 7.10) and rewrite Mx and My in terms of the angular velocities:

$$M_x = bl(\Omega_4^2 - \Omega_2^2) \tag{7.12}$$

$$M_y = bl(\Omega_3^2 - \Omega_1^2) \tag{7.13}$$

So far we have described the entities that produce three of the four principal movements detailed in the chapter 3. The Roll variations are produced by Mx as the Pitch variations by My and Yaw variations by Mz. The fourth and simplest remaining movement, the Throttle, is easily defined as the summation of the four engines's forces (7.8-7.11). Below we explicit it in terms of angular velocities:

$$\tau = b(\Omega_1^2 + \Omega_2^2 + \Omega_3^2 + \Omega_4^2) \tag{7.14}$$

We also introduce the Hover speed (Ω_h) as the propeller velocity required to make the resultant force applied on the vehicle along the z axis equals to 0. In other words, the velocity needed to make the drone hover. We can estimate the Hover speed by setting the Throttle force equals the gravity force applied on the vehicle as shown below:

$$Mg = \tau \tag{7.15}$$

Setting $\Omega_1 = \Omega_2 = \Omega_3 = \Omega_4 = \Omega_h$ and combining the equations (7.14) and (7.15) we define Ω_h as follows:

$$\Omega_h = \sqrt{\frac{mg}{4b}} \tag{7.16}$$

According to the parameters in Table 1 and assuming g = 9.8, Ω_h is approximately 413.17 rad/s.

8 PID

The PID control is a well known closed loop technique which consists in the summation of three control signals related to the system's error: Proportional, Derivative and Integral. These signals are manipulated by constants (Kp, Kd and Ki) and each of them produces different effects on the system behaviour. By picking specific constants the controller enforces the system to behave conveniently. It is shown below the PID signal on Laplace Domain:

$$U(s) = K_P E(s) + K_D s E(s) + \frac{K_I}{s} E(s) \tag{8.1}$$

As shown in (7.4), the angles Roll, Pitch and Yaw which we are interested in controlling, depend directly on the moments My, Mx and Mz respectively. Therefore, the PID signal will act on the moments in order to obtain the desired angles. There will be three PID signals Ux, Uy and Uz, one for each moment. Assuming we are dealing with ideal sensors, the Figure 7 illustrates the block diagram of the system, where the equations (8.1) and (7.4) are inside of the subsystems PID and Drone respectively.

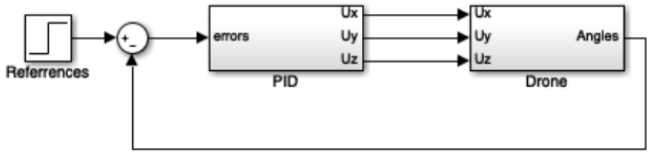


Figure 7 – Controller's Block Diagram

In order to define the controller constants we will use the Pole Placement approach. This method modifies the Root Locus of the resultant system in such a way that the desired pole, i.e, the pole which satisfies the dynamic requirements, be within the new Root Locus. To accomplish that, poles and zeros are added in order to make the desired pole satisfy the Angle Condition. Since the PID adds poles and zeros to the system according to the constants, the Pole Placement determines which K_P, K_D and K_I make the Root Locus cross the desired pole. Rewriting (8.1) as a transfer function expliciting the zeros and poles, the PID action can be understood as an addition of a pole and two zeros to the system:

$$\frac{U(s)}{E(s)} = \frac{s^2 K_D + s K_P + K_I}{s} \tag{8.2}$$

The desired pole has the maximum overshoot equals to 15% and the maximum settling time equals to 1 second as determined by the requirements R2 and R3. The percentage of overshoot (MP) and the settling time (t_s) can be written in terms of the damping factor ζ and the natural angular frequency (W_n) (OGATA, 2010):

$$\begin{aligned} Mp &= \exp \frac{-\pi\zeta}{\sqrt{1-\zeta^2}} \\ t_s &= \frac{4}{W_n\zeta} \end{aligned} \tag{8.3}$$

By defining an overshoot equals to 5%, i.e, MP= 0.05, and settling time equals to 1 we get, through the equation (8.3), that $\zeta = 0.7$ and W_n= 5.79 [rad/s]. This pole can be represented as a complex number with the form $-\zeta W_n \pm W_n\sqrt{1-\zeta^2}j$ where $j = \sqrt{-1}$. Below the desired pole is shown on the complex plan.

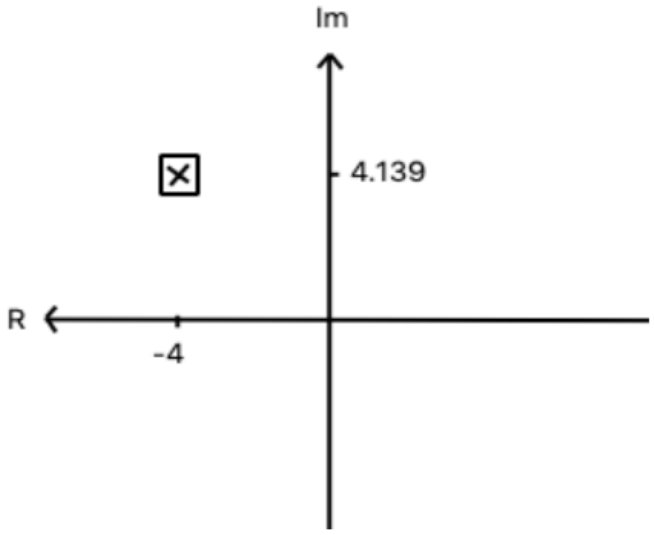


Figure 8 – Desired pole

Having Defined the desired pole, the next step is to verify whether this pole belongs to the current Root Locus of the system. Applying the Laplace Transform on equations in (7.4) we obtain the following transfer functions:

$$\begin{aligned} \frac{\phi(s)}{M_x(s)} &= \frac{1}{J_{xx}s^2} \\ \frac{\theta(s)}{M_y(s)} &= \frac{1}{J_{yy}s^2} \\ \frac{\psi(s)}{M_z(s)} &= \frac{1}{J_{zz}s^2} \end{aligned} \tag{8.4}$$

The Root Locus of all of these transfer functions are equal and it is illustrated as follows:

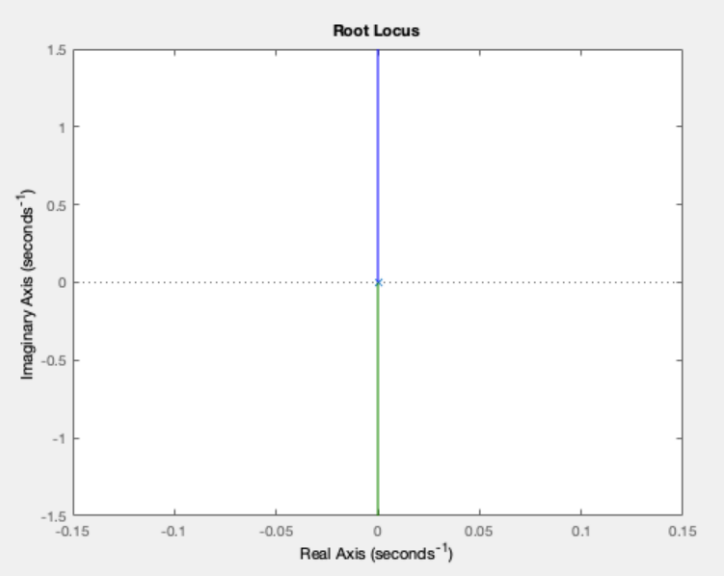


Figure 9 – System's Root Locus

As shown above, the current Root Locus does not cross the desired pole. As discussed later, the PID controller will add one pole on the origin and two zeros whose position depends on the constants. We decided, for simplicity, to place both zeros at the same location. The Figure 10 shows on the complex plan the poles and zeros of the system, after adding the PID action, where zc1, zc2 and pcare the position of the PID zeros and poles whereas p1and p2 are the position of the old system poles.

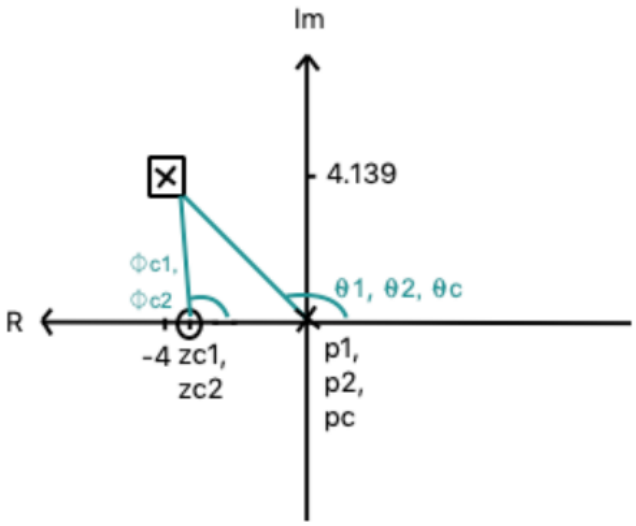


Figure 10 – Poles and zeros of the new system

The angles θ_1, θ_2 and θ_c can be easily defined geometrically:

$$\begin{aligned} \theta_1 = \theta_2 = \theta_c &= 180^\circ - \arccos(0.7) \\ \theta_1 = \theta_2 = \theta_c &= 134,42^\circ \end{aligned} \tag{8.5}$$

For the desired pole to be part of the new Root Locus, the Angle condition must be satisfied, i.e:

$$\phi_{c1} + \phi_{c2} - \theta_1 - \theta_2 - \theta_c = -180^\circ \tag{8.6}$$

As $\phi_{c1} = \phi_{c2}$, combining (8.6) and (8.5):

$$\phi_{c1} = \phi_{c2} = 111,63^\circ \tag{8.7}$$

With ϕ_{c1} and ϕ_{c2} we define the PID zeros position geometrically:

$$\begin{aligned} zc1 = zc2 &= -4 + \frac{4.139}{\tan(68.37)} \\ zc1 = zc2 &= -2.63 \end{aligned} \tag{8.8}$$

Therefore, the transfer function of the controller is given by:

$$\begin{aligned} \frac{U(s)}{E(s)} &= \frac{(s+2.36)^2}{s} \\ \frac{U(s)}{E(s)} &= \frac{s^2 + 4.72s + 5.57}{s} \end{aligned} \tag{8.9}$$

Comparing the equation (8.9) with the equation (8.2) we define the constants as $K_P = 4.72, K_D = 1$ and $K_I = 5.57$. In these conditions, the new Root Locus passes reasonably close to the desired pole as shown by Figure 11.

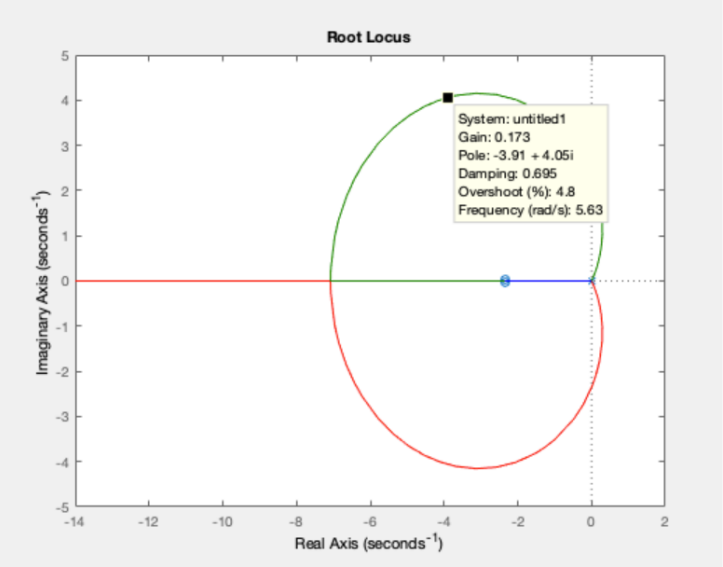
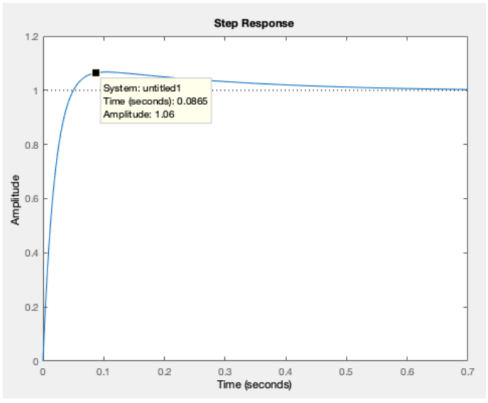
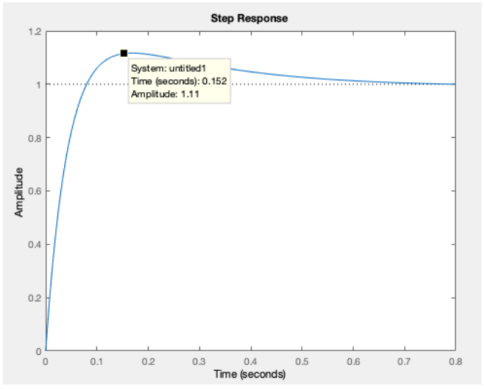


Figure 11 – New Root Locus

The Figure 12 shows the responses of the controlled Roll, Pitch and Yaw in closed loop given a unit step as input. Once Jxx and Jyy are approximately equal, the Roll and Pitch response are the same, where the overshoot is 6%, settling time is 0.7s and there is no steady-state error. In the Yaw response there is an overshoot of 11% , settling time of 0.8s and no steady-state error. Therefore, the attitude response satisfies the requirements R1, R2 and R3.



(a) Roll and Pitch response



(b) Yaw response

Figure 12 – Attitude response

However, it is also necessary to verify whether R4 is satisfied, i.e, we need to introduce the engine's speed limitations to the model. By setting Mx = Ux, My = Uy and Mz = Uz and combining the equations (7.7), (7.12), (7.13), (7.14) and (7.16) we can write

the engine's speed in terms of PID signals as shown below:

$$\begin{aligned}
\Omega_1^2 &= \Omega_h^2 - \frac{U_y}{2bl} - \frac{U_z}{4d} \\
\Omega_2^2 &= \Omega_h^2 - \frac{U_x}{2bl} + \frac{U_z}{4d} \\
\Omega_3^2 &= \Omega_h^2 + \frac{U_y}{2bl} - \frac{U_z}{4d} \\
\Omega_4^2 &= \Omega_h^2 + \frac{U_y}{2bl} + \frac{U_z}{4d}
\end{aligned} \tag{8.10}$$

We then add the equations (8.4) in the Drone subsystem as well as apply a saturator at four speeds to ensure that them are between 0 and Ω_max. Then, based on the speeds we obtain the moments through the equations (7.7), (7.12) and (7.13). And finally, with the equations (7.4) we estimate the angles Roll, Pitch and Yaw. Therefore, the Drone subsystem becomes the following:

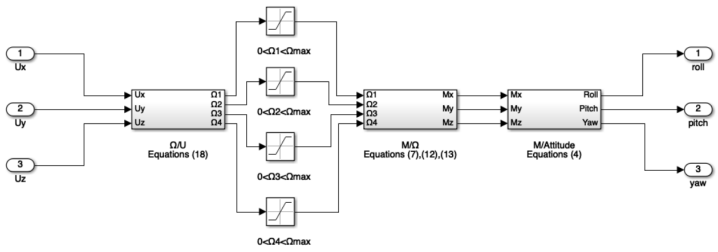


Figure 13 – Drone Subsystem

The speed constraint decreases significantly the performance of the controller once that before it has been added to the model, the engines were assuming speeds way higher than the maximum and even negative speeds which would represent rotations in the opposite direction. It is shown by figure 14:

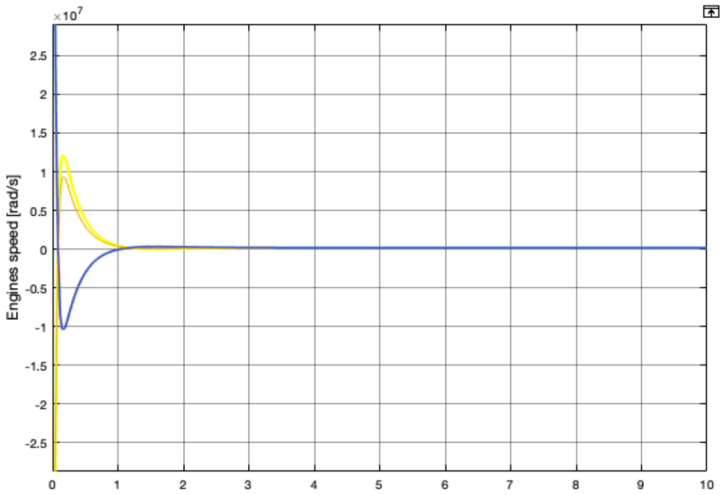


Figure 14 – Engines Speed with no limitation

In these conditions, the roll response shown below, for a step of 30 degrees as input, is oscillatory and the error increases with the time.

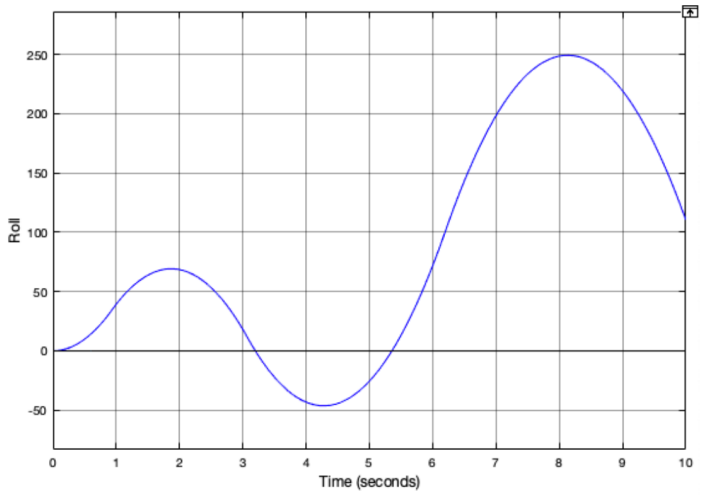


Figure 15 – Roll response with speed constraint

It is possible to overcome this issue by regulating the PID constants. The K_P constant is mainly responsible for increasing the speed response. An important characteristic of the K_D term is to deal with transitions. And the K_I term is well used to suppress the steady-state error. In this light, we'll first act on the oscillation issue by increasing the K_D from 1 to 2, keeping K_P and setting K_I to 0. This configuration yields the following roll response for a step of 30 degrees as input.

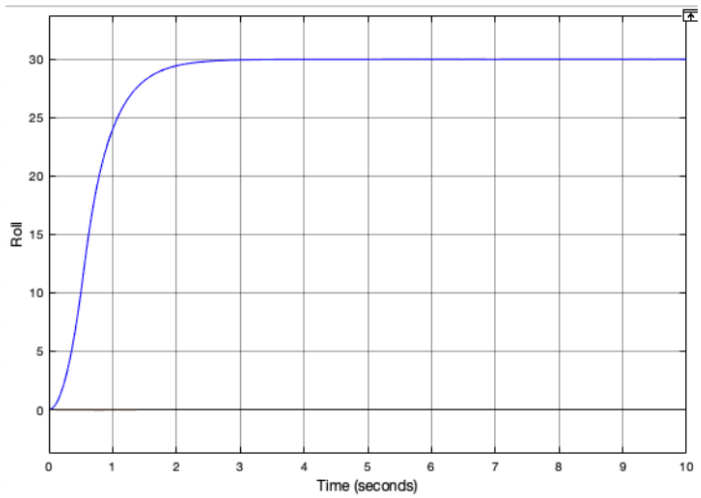


Figure 16 – Roll response with $K_P = 4.72$, $K_D = 2$ and $K_I = 0$

Although the oscillations have gone, the response got slower taking approximately 3 seconds to settle. As discussed before, the K_P term can make the response faster. Therefore, by increasing K_P from 4.72 to 6 and keeping K_D and K_I, we obtained the following response:

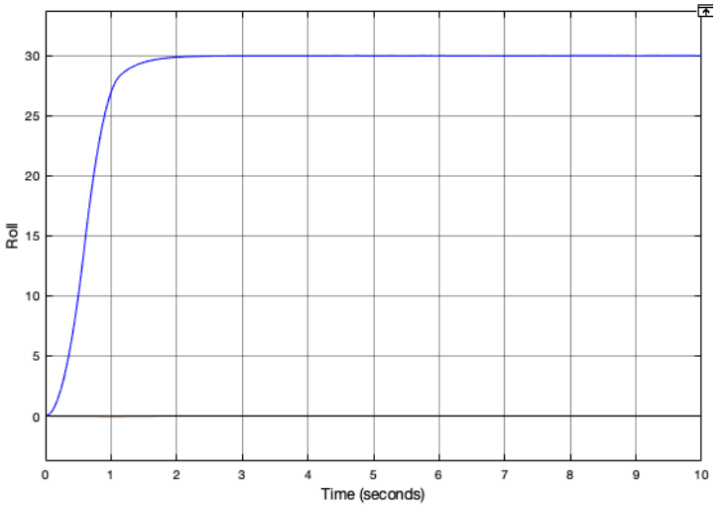


Figure 17 – Roll response with $K_P = 6$, $K_D = 2$ and $K_I = 0$

As shown by figure 17, the response became faster settling in 1.8 seconds which satisfies the requirement R3. Additionally, there is no overshoot and no steady-state error which means that the requirements R1 and R2 also are satisfied and there is no need to include the Integral term to the controller. These constants K_P, K_D and K_I defined to control the Roll angle also can be used to control the Pitch angle once the dynamic behaviour of both angles is almost identical. However, it cannot be applied to control the Yaw angle. Once the rotational inertia around the z axis is almost 3 times higher than around x and y axes, it is required a higher damping, i.e, a higher K_D, to handle the transitions of the Yaw angle. Thus, by increasing K_D from 2 to 5.5 and keeping K_P equals to 6 and K_I equals to 0, we get a response with 4 seconds of settling time, no steady-state error and no overshoot conforming determined on the chapter 4. It is illustrated below in the figure 18.

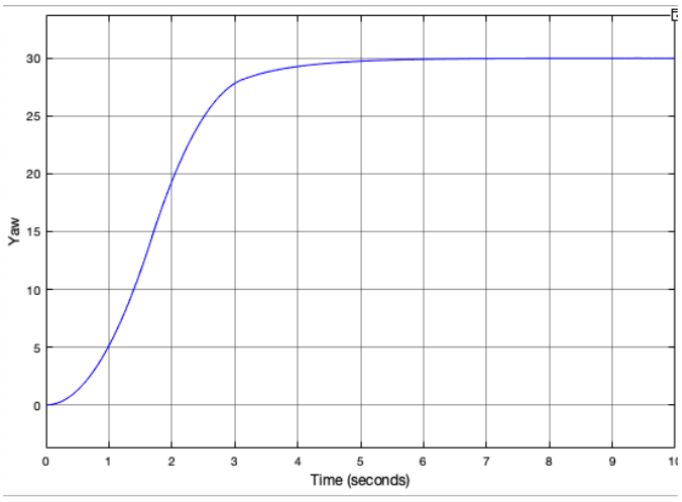


Figure 18 – Yaw response with $K_P = 6$, $K_D = 5.5$ and $K_I = 0$

9 MACHINE LEARNING BACKGROUND

In this chapter we'll be discussing the theory behind the PPO model. It'll be given a background in Markov Decision Process, Neural Networks and Actor-Critic method.

9.1 Markov Decision Process

The Markov Decision Process (MDP) is used to model processes probabilistically. It is called markovian once the probability distribution of the future state of the process depends only on the current state and the action selected. In a MDP model, the process changes in steps as actions are taken. The set of steps required to go from the initial state to the target state is called by episode. Each action is rated, or rewarded, based on how good is the state reached due to the action performed. According to (PELLEGRINI,), MDP can be defined as a tuple <S,A,T,R> where:

- S is a set of possible process states;
- A is a set of actions that interfere in the process;
- T :S×A×S→[0,1] is a function that provides the probability of transferring to s' S, given the current state s S and the selected action a A. It is symbolically represented by T (s' |s, a);
- R : S × A→R is a function that provides the reward for taking the action a A when the process is in a state s S.

In this model there also must be a decision maker, or agent, who decides which actions should be taken. The agent picks an action based on a specific rule named policy (π). The policy is basically a function that maps states to actions. The action selected is then performed in the environment which yields a new state. The figure 19 shows an overview of the idea behind the MDP.

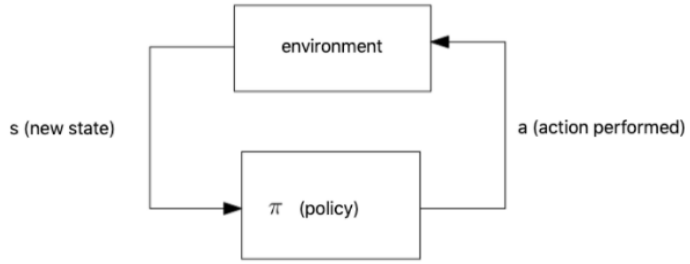


Figure 19 – MDP structure (PELLEGRINI,)

Different policies can be compared based on how much reward it provides. A very common technique is to compare the expectation of the discounted cumulative reward, or Return Value (R), each policy produced in a given episode. R can be described as follows:

$$R = E\left[\sum_{k=1}^{z} \gamma^{k-1} r_k\right] \tag{9.1}$$

Where z is the episode size, k is the step index, r_k is the reward obtained in step k and γ]0,1[is a discount factor. In order to evaluate as well as update a policy we also use a Value Function,V(s). In a nutshell, it estimates how good is to assume the state s by computing the expectation of the Return Value given the initial state s0 = s. It can be written as:

$$V_\pi(s) = E[R|s_0 = s] \tag{9.2}$$

In this work we´ll train a Neural Network in order to obtain a Value Function estimator. In the next section we are introducing the main concepts behind Neural Networks.

9.2 Neural Networks

A Neural Network is a model inspired by the human neural system which is basically a set of nodes, or neurons, interconnected to each other (BISHOP, 2006). Once a neuron receives a signal, it gets processed and then passed to another neuron or to outside the network. The connections between neurons are called edges. Through them, information or signals can be passed. Each edge commonly has a weight (w) which indicates the strength or relevance of the signal being passed. These weights are updated during the training in such a way that the final NN model produces the desired output given a particular input. A common NN structure, shown by figure 20, is composed of layers, often of three types: Input, Hidden and Output layer.

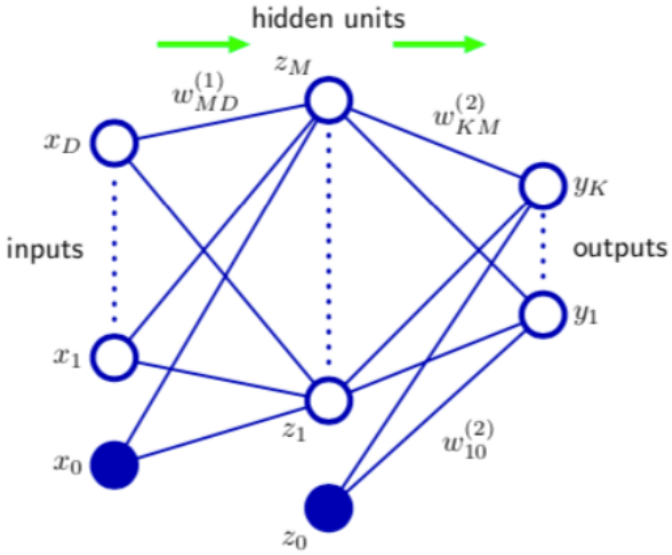


Figure 20 – Neural Network structure (BISHOP, 2006)

The nodes within the hidden layer as well as in the output layer receive multiple signals coming from the previous layer. In each of these nodes will be made a linear combination of the signals received according to their respective weights. A nonlinear function, also known as activation function, may be applied in the result of the linear combination, such as the logistic sigmoid function (BISHOP, 2006). The signal being processed is illustrated below in figure 21.

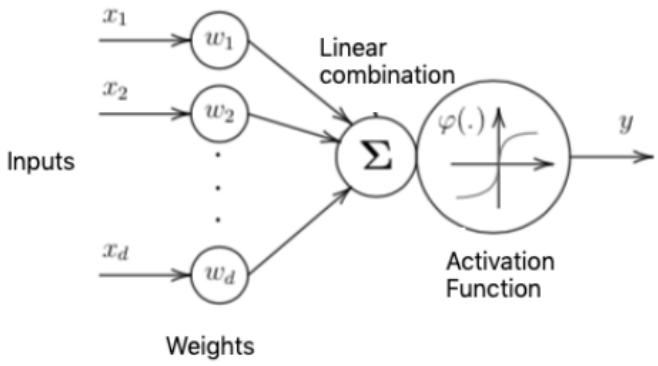


Figure 21 – Neuron processing signal (FIGUEIREDO, 2018)

The sigmoid function is widely used for neuron activation since it is differentiable and provides a number in a range of 0 to 1, which allows us to work probabilistically.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{9.3}$$

In this work, we use the hyperbolic tangent as activation function since it is the Stable Baselines standard.

Once the NN output depends on its weights, the learning process is focused on updating them in order to obtain the expected output. It starts with a set of random weights that, for a particular input, provides a response. The model then compares the response obtained to the desired response and computes the error. Thus, new weights are calculated based on the error and a learning rule. As in this work we are interested in making the drone go from a random attitude to a target attitude, our neural network will be trained to minimize the difference between them, i.e, the error. Or, in other words, we'll be focused on maximizing the accumulated reward in an episode, but it'll be discussed further in the next sections.

9.3 Actor-Critic Method

The Actor-Critic (AC) method is a procedure that intends to provide an optimum policy in a MDP model. It is composed of two main components: The Actor and Critic. As mentioned in section 9.1, the policy is in charge of, given a state, decides the best action to take. In other words, we can say that the policy assumes an Actor role in the process. Therefore, in this method the policy is represented by the Actor. The Critic, as the name suggests, will criticize the action selected by the Actor, i.e, it'll estimate how good is the state reached due to the action taken. Thus, the Critic is represented by the Value function of the MDP. Both Actor and Critic will be estimated by a NN each. Whereas algorithms such as Monte Carlo require the full episode to be performed before the policy gets updated, the AC updates its NN at each step, which reduces the variance considerably. We represent the Actor (π) and Critic (q) NNs as follows:

$$\begin{aligned} &\pi(s, \Theta) \\ &q(s, w) \end{aligned} \tag{9.4}$$

Where s is the state, Θ is the set of the Actor weights and w is the set of the Critic weights. Despite both NNs estimate responses independently, updates based on the estimate provided by the Critic q as shown below:

$$\Delta_\Theta = \alpha \nabla_\Theta (log\pi(s))(r_k + \gamma \hat{q}(s') - \hat{q}(s))) \tag{9.5}$$

Where α is the learning rate of the Actor NN, r_k is the reward at the step k, γ is the discount factor, s' is the next state provided by the current action. We also can

describe the update rule in terms of the Advantage function, A(s) shown as follows:

$$A(s) = r_k + \gamma \hat{q}(s') - \hat{q}(s) \tag{9.6}$$

Similarly, the Value function weights update according to the following expression:

$$\Delta_w = \beta \nabla_w \hat{q}(s) A(s) \tag{9.7}$$

Where β is the learning rate of the Critic NN. It is very useful defining the Advantage function once it can provide meaningful information regarding the training process. For example, when A(s) assumes positive values it means that the current action provides a reward above the average, thus the weights will be updated in such a way that this action will be more likely to be selected for that particular state in next steps. On other hand, if A(s)is negative, the gradient will be pushed in the opposite direction making this action be avoided in the future. The PPO model, presented in the next section is derived from the AC method

10 PPO MODEL

Policy gradient methods as AC are becoming more and more popular due to their vast applicability. However, getting good results through them is challenging because of their sensibility to hyperparameters such as the step size. For a small step size, most of the policy gradient algorithms will converge extremely slowly, whereas for a large step size the signal will be affected by noise. Besides, they often are inefficient once they take enormous quantities of time steps to learn very simple tasks. These issues happen mainly because the design of most of those algorithms do not prevent them from having large policy updates. The PPO showed up to overcome this problem.

The Proximal Policy Optimization model consists in a policy gradient method that follows a Actor-Critic architecture. In this project we were motivated by Stable Baselines library which defines the neural networks for both critic and actor with 2 hidden layers with 64 neurons and tanh as activation function. This structure is illustrated below.

Figure 22 – Actor and Critic

The PPO model has as main goal avoiding too large policy updates. In order to do that, (AL, 2017) defines the probability ratio $r_t(\Theta)$ that is the ratio between the current and the old policy:

$$r_t(\Theta) = \frac{\pi_\Theta}{\pi_{\Theta old}} \tag{10.1}$$

The algorithm will basically be focused on keeping $r_t(\Theta)$ within a convenient range. In light of this, (AL, 2017) introduces the main objective function that PPO optimizes, $L^{CLIP}(\Theta)$, presented below:

$$L^{CLIP}(\Theta) = \hat{E}_t[min(r_t(\Theta)\hat{A}_t, clip(r_t(\Theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{10.2}$$

The objective function is the expectation of the minimum of two terms: $r_t(\Theta)\hat{A}_t$ and $clip(r_t(\Theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$. The first one is the default objective for normal policy gradients which pushes the policy towards actions that yield a high positive advantage. The second term is a truncated version of the first one by applying clipping operation

between $1-\epsilon$ and $1+\epsilon$, where is recommended by (AL, 2017) to be 0.2. It ensures that the probability ratio will remain between 0.8 and 1.2. The final Loss function that is used to train the Agent presented by (AL, 2017) is shown below:

$$L_t^{PPO}(\Theta) = \hat{E}_t[L_t^{CLIP}(\Theta) - c_1 L_t^{VF}(\Theta) + c_2 S[\pi_\Theta] s_t] \tag{10.3}$$

Where $c_1 L_t^{VF}(\Theta)$ is in charge of updating the Critic network and $c_2 S[\pi_\Theta] s_t$ is called the Entropy term. It is in charge of making sure that our agent does enough exploration during training. It happens because the entropy of a stochastic variable which is driven by an underlying probability distribution is the average amount of bits that is needed to represent its outcome. It is a measure of how unpredictable an outcome of this variable really is and so maximizing its entropy will force it to have a wide spread over all the possible options resulting in the most unpredictable outcome. We also have a couple of hyperparameters c1 and c2 that do the contributions of these different parts in the loss function. So in contrast to discrete action policies that output the action choice probabilities, the PPO policy outputs a Gaussian distribution for each available action type and when running the agent in training mode the policy will then sample from these distributions to get a continuous output value for each action.

10.1 Environment

As the PPO model follows the AC architecture, it is necessary to define the environment where our Agent learns from. The environment must provide the state of the drone as well as update it based on the action taken. Besides, the environment also must be able to set an initial state randomly for training matters. In order to design it, we've created a GYM Environment class. In the constructor function of this class is set the drone's parameters as defined in the chapter 6 and the target attitude fixed in 30 degrees for all angles. Additionally, we also defined the action and the observation space of the environment according to the GYM standard.

The Action Space is a vector of the four engines velocities which can assume continuous values. In order to do so, the Actor Neuron Network provides four Gaussian distributions where each speed is sampled from, as mentioned in the previous chapter.

The Observation Space also provides continuous values for the attitude angles and constraints them from -2π to 2π.

Besides, the Environment class has also two main functions: The step and the reset function. In the step function we describe how the drone's state changes given an action. Therefore this function has the actions as input and returns the next state. As defined in

the Requirements section, the state and action of the drone are the following vectors:

$$\begin{aligned} s &= \left\{\phi, \dot{\phi}, \theta, \dot{\theta}, \psi, \dot{\psi}\right\} \\ a &= \{\Omega_1, \Omega_2, \Omega_3, \Omega_4\} \end{aligned} \tag{10.4}$$

The step function estimates the next state based on the Euler integration presented below:

$$s_{t+1} = s_t + T_s \dot{s}_t \tag{10.5}$$

Where T_s is the time step and it is defined as 0.01 as determined by R8. In order to compute the time derivative of the state, firstly the step function computes the four engine´s forces with the equations (7.8-7.11) through the action vector. Then it computes the moments based on the equations (7.5-7.7) and finally determines the second order time derivative of the angles Roll, Pitch and Yaw by using (7.4). With them, we express the time derivative of the state as follows:

$$\dot{s}_t = \left\{\dot{\phi}, \frac{M_x}{J_{xx}}, \dot{\theta}, \frac{M_y}{J_{yy}}, \dot{\psi}, \frac{M_z}{J_{zz}}\right\} \tag{10.6}$$

Besides, the step function also provides the reward the current step yielded. We've designed our reward function to penalize the difference between the angles reached and the target angles as well as large angular velocities and large torques in order to make our agent achieve the task as smooth as possible. Therefore, inspired by the reward function used in the Pendulum GYM environment we ended up with the following:

$$R = -(|angularError|^2 + 0.1|angularVel|^2 + 0.001|torque|^2) \tag{10.7}$$

Where

$$\begin{aligned} angularError &= [\theta, \phi, \psi] - [\theta_{target}, \phi_{target}, \psi_{target}] \\ angularVel &= [\dot{\theta}, \dot{\phi}, \dot{\psi}] \\ torque &= [M_x, M_y, M_z] \end{aligned} \tag{10.8}$$

The reset function, on the other hand, simply sets the state vector randomly. It is fundamental because at the end of each episode the environment has to start from a random position which forces the agent to learn how to go to the target attitude starting from a large range of initial states, including the most extreme conditions such as starting from upside down.

10.2 Training

The implementation of the PPO model as well as the training algorithm used in this work were provided by the Stable Baselines library due to its simplicity and compatibility with GYM environments. The set of hyperparameters used to train our model is shown on the table below. This was inspired by the set used to train the Pendulum GYM environment by rl-baselines-zoo since it has a dynamic behaviour quite similar to the drone's attitude. We made the agent run 976 episodes which means that the policy have been updated 976 times. The agent takes 2048 time steps per episode and randomly resets itself at the end of each episode as explained in the previous section. Therefore, there were a total of 2 million of time steps. It took approximately 54 minutes to get the training completed with the i5 Intel processor 2,5 GHz. Furthermore, the discount factor, γ, and the learning rate, α, used to train are 0.99 and 3e-4 respectively. These parameters are shown below:

Hyperparameter	Value
Number of time steps	2.000.000
Number of steps per episode	2048
γ	0.99
α, β	3e-4

Table 2 – PPO hyperparameters

With these hyperparameters our agent was able to reach the mean convergence of rewards around the episode 244, i.e, after 500.000 time steps as illustrated by the figure 23.

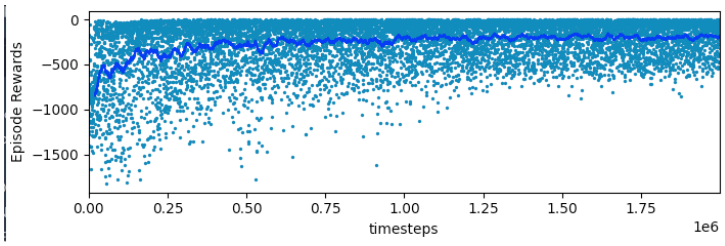
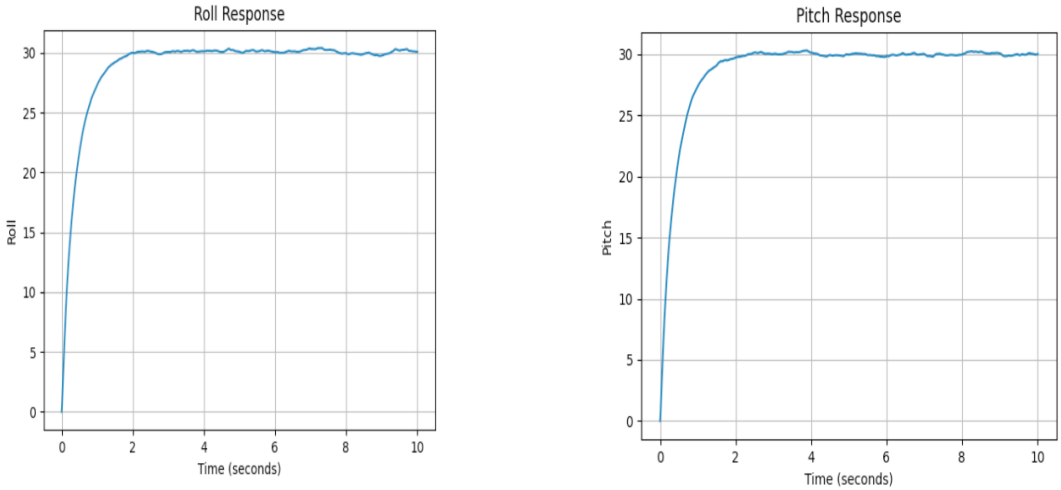


Figure 23 – Reward convergence

10.3 Results and comparison with PID

In order to test as well as compare the PPO performance with the PID model, we set the initial state of the environment equals to 0 for all state parameters, i.e, Roll, Pitch, Yaw and their time derivatives and let the drone's simulation run for ten seconds. We trained the Neural Nets to reach the target angle of 30 degrees for all attitude's angles, thus allowing us to obtain a time response similar to the step response we got for the PID model. The Roll and Pitch time response are shown as follows.



(a) Roll response (b) Pitch response

Figure 24 – Roll and Pitch time response

As expected, the Roll and Pitch response are quite similar due to their almost identical dynamic behaviour. For both of them we got 1.5 seconds to settle, with no overshoot and no steady-state error, satisfying R1, R2 and R3. Compared to the PID model, we got a response with settling time slightly better. However, the PPO model provides a response with random oscillations whereas PID does not. On the other hand, the Yaw response of the PPO model, shown in figure 24, got a settling time around 1.2 second which was significantly better than the PID model that got a settling time of 4 seconds. Additionally, the requirements R1, R2 and R3 are also satisfied although there are oscillations higher than those seemed in the Roll and Pitch responses.

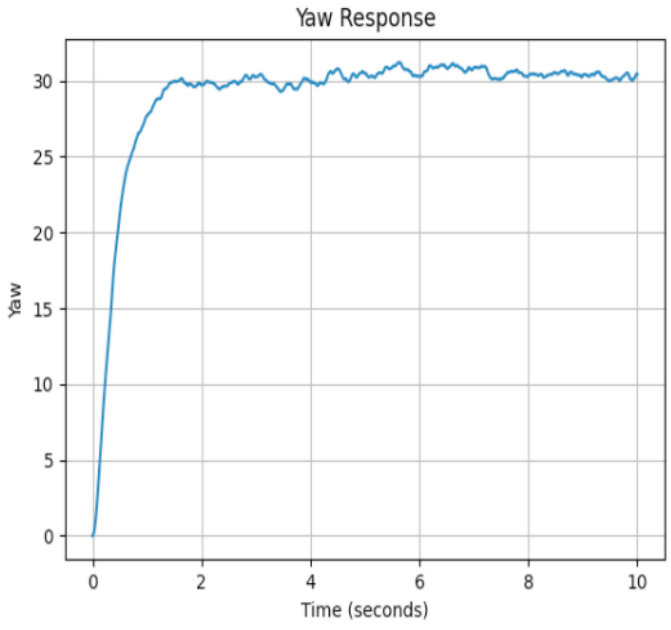


Figure 25 – Yaw time response

However, so far we have tested both models just in soft conditions, i.e, starting from the initial state equals to 0 and reaching the attitude of 30 degrees that could be considered as a small angular variation. As discussed by (AL., 2018), in soft conditions the PID model performs satisfactory well and we indeed have already shown it in the chapter 8. But for harsh conditions the PPO model tends to perform better than the PID. In order to verify it, we set the initial Roll angle to 180 degrees and kept the Pitch and Yaw to 0 which would represent the upside down drone. In this condition and with the best set of controller gains found in the chapter 8, i.e, $K_p = 6, K_D = 2$ and $K_I = 0$, we got the following time response for Roll angle using the PID model: (AL., 2018)

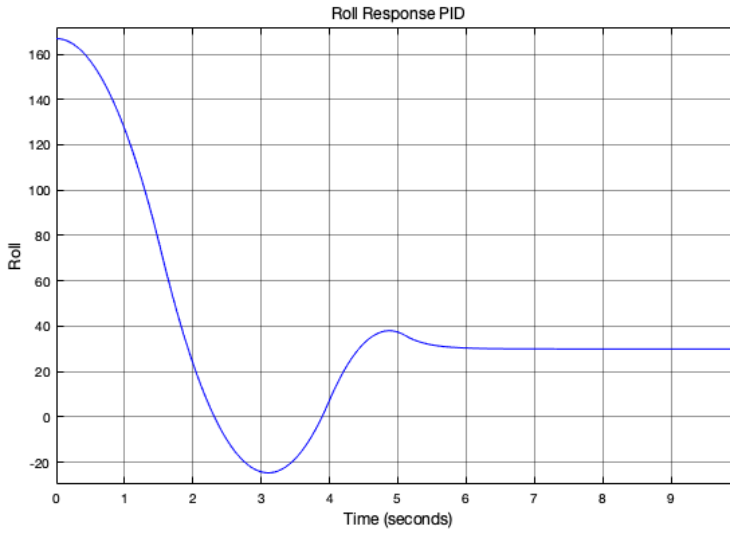


Figure 26 – Roll time response with PID in harsh condition

As we can see, the response gets a relevant overshoot of 33,33% and takes more than 5 seconds to settle thus it does not satisfy the requirements described in the chapter 4. On the other hand, the Roll response with the PPO model with the same set of hyperparameters used so far got no overshoot and still settles in almost 2 seconds. It is illustrated as follows:

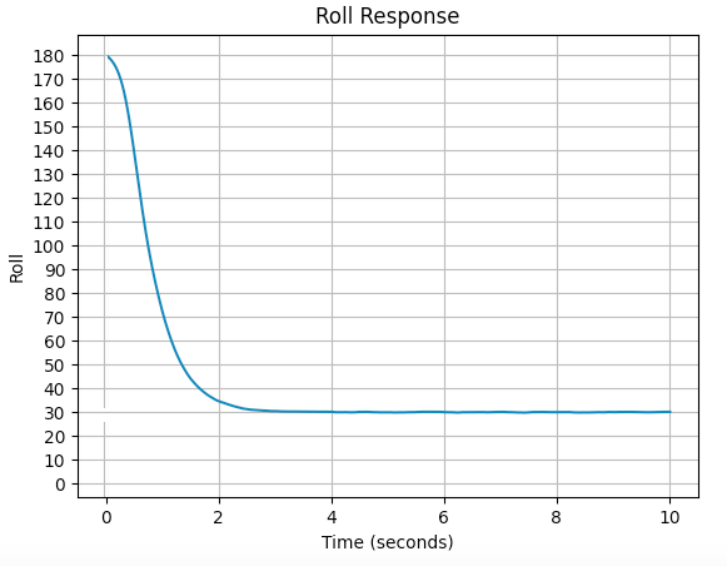


Figure 27 – Roll time response with PPO in harsh condition

This good performance of the PPO in extreme conditions might be due to the fact that the agent starts from a random attitude each episode during training which allowed the drone to learn how to accomplish the task regardless the initial condition.

The tables below summarize the results achieved by both methods in Soft Conditions and in Harsh Conditions for comparison purposes:

	PID			PPO		
	Roll	Pitch	Yaw	Roll	Pitch	Yaw
Settling Time [s]	1.8	1.8	4	1.5	1.5	1.2
Steady-State Error	0	0	0	0	0	0
Overshoot	0	0	0	0	0	0

Table 3 – Summary of Results in Soft Conditions

	PID (Roll)	PPO (Roll)
Settling Time [s]	5.5	1.8
Steady-State Error	0	0
Overshoot	33,33%	0

Table 4 – Summary of Results in Harsh Conditions

One of the biggest advantages of the PPO over PID is that it is not model based which would make the control task easier. However, most of the reinforcement learning applications train their models in a simulated environment and test it in real projects afterwards due to the harsh conditions present during training. In order to build the

simulated environment it is necessary to know the dynamic of the model. Therefore, this specific advantage actually is not often applicable.

Whereas the PID has the controllers gains that require a non trivial analysis, the PPO has the hyperparameters that are very complex to understand and to set. However, the PPO constants seem to be more robust than the PID´s once that with the same set of hyperparameters we obtained a satisfactory response for Roll, Pitch and Yaw in soft and harsh conditions whereas with the same PID gains used to control Roll and Pitch do not stabilize satisfactorily the Yaw angle and perform poorly in extreme conditions.

In this work we developed a PPO model that was trained to reach the attitude target of 30 degrees for all angles, i.e, this model is only capable to lead the drone to this specific configuration. On the other hand, once the PID gains are tuned, the controller is able to stabilize the drone in several other attitudes without losing performance. This limitation of the PPO model can be overcome by setting the target attitude to be variable during training but it would demand a different set of hyperparameters.

11 CONCLUSION

Due to the notable increasing of applications involving drones currently, the need of controlling algorithms for this matter has earned relevance. One of the most common approaches used in this task is the PID. However, recent research point that Deep Reinforcement Learning models can perform better and be more robust than PID and also could be simpler to implement once most of them are not model based. In this context, we designed and compared two models in controlling the drone's attitude task: the PID and PPO. In the PID model we estimated the controller gains based on the Newton's Laws and the Pole Placement approach. For PPO we trained two Neuron Networks to be able to map states to actions and thus control the drone. The hyperparameters and the reward function used to train the PPO model were inspired by similar open source projects such as Pendulum by GYM.

As presented by section 10.3, when was set a soft task to the Drone as going from attitude 0 to 30 degrees both model performed well, satisfying the requirements previously defined, i.e, they yielded a response with no steady-state error and no overshoot for all attitude angles and an acceptable settling time. Additionally, for the same PPO's hyperparameters we got a reasonably response for Roll, Pitch and Yaw angles whereas in the PID model we had to adjust the gains used to control the Roll and Pitch to stabilize the Yaw angle. This shows that the PPO model is less sensitive to changes than the PID. However, for extreme conditions such as starting from upside down and go to the target of 30 degrees, the PID model was not capable to provide a response that satisfies the requirements with the gains used previously whereas the PPO model did. This advantage of the PPO over PID might be due to the fact that, during training, the agent is forced to start from a different attitude each episode even it is an extreme condition or not. As the Drone trained for 976 episodes is evident that it learned how to act in harsh configurations.

However, despite the PPO response satisfies all the requirements established, it is noisy. Besides, as the main limitation of the developed model is that it was trained to accomplish a very specific task that is to reach the target attitude of 30 degrees. The PID on the other hand is capable to stabilize the drone in a larger range of targets keeping the quality of the response.

In future work we intend to train the PPO model to reach a random target per episode in order to obtain an agent that is able to lead the drone to different attitudes during test. We also are interested in overcome the noise issue by optimizing the hyperparameters as well as build a physical environment to test and compare both models.

Bibliography

AL, J. S. et. Proximal policy optimization algorithms. *ArXiv*, 2017. Cited 3 times on the pages 25, 55, and 56.

AL., W. K. et. Reinforcement learning for uav attitude control. *Boston University*, 2018. Cited 3 times on the pages 25, 26, and 60.

AUSTIN, R. Unmanned aircraft systems: Uavs design, development and deployment. *John Wiley Sons Ltd: Chichester*, 2010. Cited on page 25.

BISHOP, M. C. *Pattern Recognition and Machine Learning*. [S.l.]: New York: Springer, 2006. Cited 3 times on the pages 13, 50, and 51.

BOUABDALLAH, S. Design and control of quadrotors with application to autonomous flying. *Faculté des Sciences et Techniques de L'ingénieur, École Polytechnique Fédérale de Lausanne*, 2007. Cited on page 38.

BRESCIANI, T. Modelling, identification and control of a quadrotor helicopter. *Department of Automatic Control, Lund University*, 2008. Cited 4 times on the pages 13, 37, 38, and 39.

FIGUEIREDO, J. Aplicação de algoritmos de aprendizagem por reforço para controle de navios em águas restritas. *Universidade de São Paulo*, 2018. Cited 2 times on the pages 13 and 51.

HWANGBO, J. et al. Control of a quadrotor with reinforcement learning. *IEEE ROBOTICS AND AUTOMATION LETTERS*, 2017. Cited 2 times on the pages 25 and 26.

OGATA, K. *Engenharia de Controle Moderno*. 5º. ed. [S.l.]: Prentice-Hall, 2010. Cited on page 42.

PELLEGRINI, J. *Processos de Decisão de Markov: um tutorial*. Xiv. [S.l.]: RITA. Cited 3 times on the pages 13, 49, and 50.

QUEIROZ, J. Construção de um veículo aéreo não tripulado na configuração quadrirrotor como plataforma de estudos. *Universidade Tecnológica Federal do Paraná*, 2014. Cited 2 times on the pages 13 and 35.

Sá, R. C. Construção, modelagem dinâmica e controle pid para estabilidade de um veículo aéreo não tripulado do tipo quadrirotor. *Centro de Tecnologia, Universidade Federal do Ceará*, 2012. Cited 3 times on the pages 13, 33, and 34.

WASLANDER, S. L. et al. ulti-agent quadrotor testbed control design: Integral sliding mode vs. reinforcement learning in intelligent robots and systems. In: *IEEE/RSJ International Conference on. IEEE*. [S.l.: s.n.], 2016. Cited on page 26.

YU, A.; PALEFSKY-SMITH, R.; BEDI, R. Deep reinforcement learning for simulated autonomous vehicle control. *Course Project Reports*, 2016. Cited on page 25.