# POLITECNICO DI TORINO

**MASTER's Degree in Aerospace Engineer**


Politecnico
di Torino

**MASTER's Degree Thesis**

# Study and design of an autotuning system for a helicopter flight simulator

**Supervisors**

**Prof. Giorgio GUGLIERI**

**Prof. Pierangelo MASARATI**

**Ing. Francesco BORGATELLI**

**Candidate**

**Gabriele LUZZANI**

**October 2021**

**Abstract**

Flight simulators for pilot training are constantly evolving, and their role in the training process is becoming increasingly important. The synthetic environment is used during the initial phases of training, in familiarisation with the aircraft, and then maintains its importance in the subsequent phases, culminating in training for mission-critical and safety-critical procedures. The simulation devices, although having different levels of complexity, realism and fidelity compared to the real aircraft, are linked by a common set of standards and requirements, coming from the designated National and Supranational Authorities, which must be met to allow the certification of simulators. This suitability is also demonstrated through objective tests that link the behaviour of the simulated aircraft within the synthetic environment with the real aircraft performances, by comparing flight manoeuvres accomplished in the simulator with the same manoeuvres recorded in flight. This tests, named QTGs (Qualification Test Guide), are executed by a software model and normally take a lot of time due to the difficulty in tuning the gain of the control system embedded in the software.

Hence, the main goal of this project is to develop an automatic control system, to be used during the validation tests of a helicopter flight simulator, which is able to faithfully replicate the behaviour of the pilot and which is able to auto-tune, through an automatic process, in order to reduce the time required for the definition and execution of the tests, lowering the man-time currently required by the process and consequently reducing costs.

This goal was achieved starting from a state-of-the-art analysis to understand the today environment in auto-tuning control system and to find the more suitable solution for this project. Then, the next step consisted in studying the helicopter flight model, in particular the MathPilot section that is the core part analysed during this work, to figure out the context in which the model would be implemented. Consequently, the modelling section began using a dedicated calculation environment. Hence, it followed a part of analysis in which the correct performances of the model were verified. Finally, the project ended with conclusions and some considerations on the whole project.

This work was developed in a model-based environment that runs together with the helicopter simulation model in order to perfectly integrate itself in its loop.

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

**EASA**
European Union Aviation Safety Agency

**FSTD**
Flight Simulation Training Device

**NTSB**
National Transportation Safety Board

**QTG**
Qualification Test Guide

**CAS**
Calibrated Air Speed

**IAS**
Indicated Air Speed

**PID**
Proportiona Integral Derivative

**DoF**
Degree of Freedom

**SISO**
Single Input Single Output

**MIMO**
Multiple Input Multiple Output

# Chapter 1

# Introduction

The use of flight simulators is constantly growing in the last few years, indeed simulation plays a key role in all levels of aviation. It is adopted to limit and solve a wide range of different and heterogeneous issues including training, operations, initial and continuing airworthiness, the environment and innovation. According to the *European Union Aviation Safety Agency - EASA*, training is considered both an opportunity and a risk because the amount of fatal accidents occurred during training flights corresponds to the 15-20% of the total crashes, especially during helicopter training. Thanks to the exponential growth of technologies and software application in the field of aviation, simulation is the answer to reduce those risks and to improve safety in rotorcraft applications. In fact, it is possible to identify the following benefits provided by the Flight Simulation Training Devices (FSTDs): [1]

- **Safety**: this is a core value throughout the aerospace world. The training environment in simulators is designed and controlled to avoid actual safety risks for both the trainees and instructors. This context allows a trainee to make mistakes and errors in a safe condition, learning from them through a review of his performances after the simulated flight, and repeat normal and emergency procedures, which could not be suitable or without risks when performed in a helicopter.

- **Emergency and Procedure Training**: thanks to the FSTD's flexibility a trainee has also the opportunity to simulate and deal with complex and dangerous scenarios, practicing specific emergency procedures. In fact, the FSTD with a sufficient fidelity level can be used to train emergency procedures, different manoeuvres and flight regimes like Degraded Visual Environment, Vortex Ring, Autorotation, etc. Moreover, the *National Transportation Safety Board - NTSB*, investigating several helicopter accidents, demonstrated that if pilots had acquired more knowledge and skills through the use of certified simulators, they would certainly have been better able to handle in-flight emergencies and avoid mistakes during manoeuvres.

- **Experience of Realism**: using flight simulators allows the trainee to experience a realistic and effective training, including the startle management, the practice of

diagnostic processes and troubleshooting procedures in a safe context where it is possible to learn from own mistakes;

- **Availability and Serviceability**: another benefit provided by FSTD is certainly its twenty-four hour availability throughout the week. Moreover, it is possible to avoid problems such as delays due to traffic congestion, in flight conditions (Day/Night, etc.) or icing conditions leading to a serviceability typically of 96-98%;

- **Economic Benefits**: it is a faster and more cost-effective solution, especially for complex helicopter types in a multi-crew context. Nevertheless, the helicopter is available to be used for revenue generating flights, while pilots are training on the FSTD;

- **Environmental Benefits**: this benefit include lower emissions, a lower carbon footprint, zero noise pollution and minimal impact on the local area particularly during night time.

However, flight simulators have different levels of complexity, realism and fidelity with respect to the real aircraft. They must satisfy a set of standards and requirements imposed by the National and Supranational Authorities, which must be met to allow the certification of simulators as suitable for use in training. This capability is also demonstrated through objective tests that link the behaviour of the simulated aircraft with the real aircraft performances, by comparing flight manoeuvres accomplished in the simulator with the same recorded in flight. This tests, named QTGs (Qualification Test Guide), are executed by a software model and normally take a lot of time due to the difficulty in tuning the gain of the control system embedded in the software. The activity of tuning this controller is carried out by experienced technicians and is usually time-consuming, in proportion to the number of tests. In this context, this thesis project was carried out at the *TXT e-Solutions*, that is a developer of simulator software, working especially with helicopters.

**Thesis goal** The main goal of this project is to develop an automatic control system, to be used during the validation tests of a helicopter flight simulator, which is able to faithfully replicate the behaviour of the pilot and which is able to auto-tune. This should be an automatic process, in order to reduce the time required for the definition and execution of tests, lowering the man-time currently required by the process and consequently reducing costs. The requirements of this project are:

1. the structure of automatic control systems in use should be studied;

2. it is required to design control systems that can increase the efficiency of the testing process, based on available literature. They can be developments of current models or ex-novo redesigns of them;

3. the solutions studied must be implemented within the TXT simulation environment, under real-time simulation constraints;

4. possible auto-regulation processes of the designed control systems should be defined;

5. it is required to evaluate the performance of the developed controller and identify improvement areas.

# Chapter 2

# Problem analysis

The first step taken in the thesis work development at *TXT e-Solutions* was to understand how their flight simulator code works. As mentioned in the previous chapter, because the project objective is to optimise and speed up the process of tuning the controllers' gains during tests, learning the functioning of the code was absolutely essential to understand how to structure the analysis of the optimisation algorithms state-of-the-art, which will be shown in the following section.

## 2.1 MathPilot analysis

The TXT simulation software is extremely complex and divided in several parts working in different periods of the simulation. In particular, it was observed that the least optimised code part is the so-called *MathPilot* section. This part is linked to the stabilization process during the simulation before the starting of QTG tests. In fact, before such tests can be carried out to verify the reliability of the simulator, the helicopter must be in a stable condition. To do this, different loops are used, called *MathPilot*, which aim is to simulate the action on commands that the pilot would perform to stabilise the helicopter. Since the QTG tests are numerous and different (e.g. static, dynamic, etc.) then the code must be also able to replicate various actions that the pilot will have to perform to stabilise the aircraft; for this reason, there are twenty-six different MathPilot blocks within the TXT code. These are implemented in the C++ language and are defined using special classes. The inputs received by these blocks are the reference values that they must follow, which can be for example an horizontal speed (*MathPilot HcCAs*), a pitch angle value (*MathPilot Pitch*), an angular roll rate (*MathPilot Roll Rate*), etc., and also the limit excursion values that the controlled variable can assume. Furthermore, what became evident from the code analysis was that, despite the numerous and varied MathPilots that have been implemented, in the end, their functioning can be summarized in the following two architectures:

- a **single loop** architecture based on a single PID controller directly controlling the value given by reference;

- a **double-loop** architecture that receives as input a certain reference value on a different variable than the one to be controlled at the end of the loop. This variable is controlled with a first PID controller that returns a value as output that corresponds to the input of a second PID controller. This second controller, on the other hand, outputs the value controlled on the main variable in question.

What differentiates one *MathPilot* from another are the variables that can be controlled (speed, angles, angular rates, torque, etc.) and the output commands that are actuated to stabilize the aircraft (longitudinal cyclic, lateral cyclic, collective, pedal). Therefore, the core part in all of these architectures is the PID controller that will be deeper explained in the following section.

### 2.1.1 MathPilot HcCAS

To better understand how these blocks actually works, the *MathPilot HcCas* was chosen as the reference one. In particular, its aim is to follow a horizontal speed given as reference in terms of Calibrated Air Speed (CAS). This block receives a reference velocity profile as input and provide the angle of the longitudinal cyclic as output through a double PID architecture. As shown in Fig. 2.1, this loop is defined as a C++ class. The structure of this block is therefore described below:

1. As can be seen from Fig. 2.1, the *MathPilot HcCAS* architecture begins with an initialisation of the main parameters used:

   - *CAS_reference*: the Calibrated Air Speed value that must be followed. It is the input of the first PID loop read from a file or given by the code itself;
   - *pitch_reference*: the pitch value set as reference obtained as an output of the outer loop and given as input to the inner loop;
   - *PITCH_MIN*: the minimum value of pitch angle to achieve the desired CAS;
   - *PITCH_MAX*: the maximum value of the pitch angle to achieve the desired CAS;
   - *value*: it is the output of this MathPilot;
   - *pitch_reference_actual*: this is the actual value of the helicopter pitch angle, it is necessary for the successful implementation of the loop.

2. After the parameters initialization, the time series of the reference CAS is then assessed, as this choice depends on the type of test to be carried out. As shown in Fig. 2.1 it is possible to load a value from a file, keep the current value of the aircraft, set a square wave reference, etc. Only the initial part of this calculation is shown in Fig. 2.1, due to company constraints.

```
double MathPilot::MathPilot_HcCas(const double dt)
{
    double value = 0.0;
    double CAS_reference = 0.0;
    double pitch_reference = 0.0;
    double pitch_reference_actual = 0.0;
    double PITCH_MIN = -25; //Minimum Pitch to achieve desired CAS
    double PITCH_MAX = 25;  //Maximum Pitch to achieve desired CAS

    if (1 == mathPilotHCData.rolePlayInProgress)
    {
            PITCH_MIN = pidPar.speedPitchLimited_outerLoopMinLimitation;//mathPilotSetting.outerLoopMinLimitation; //Minimum Pitch to achieve desired CAS
            PITCH_MAX =  pidPar.speedPitchLimited_outerLoopMaxLimitation;//mathPilotSetting.outerLoopMaxLimitation;  //Maximum Pitch to achieve desired CAS
    }

    switch(mathType.mathPlt_HcCasType)
    {
        case attMathPltCtrlType_refFile:
            if(casRefFromFlightTest != NULL && casRefFromFlightTest->samples != NULL && casRefFromFlightTest->samples->size() > 0)
            {
                CAS_reference = casRefFromFlightTest->samples->front();
                casRefFromFlightTest->samples->erase(casRefFromFlightTest->samples->begin());
            }
            else
            {
                status.mathPlt_hcCas = false;
            }
            break;
        case attMathPltCtrlType_squareWave:
            CAS_reference = squareWave(currentVal.cas + refVal.mathPlt_casRefValue, 10, dt);
            break;
        case attMathPltCtrlType_refValue:
            CAS_reference = refVal.mathPlt_casRefValue;
            break;
```

Parameter initialization

Definition of the CAS reference value

**Figure 2.1:** MathPilot HcCas extract - parameter initialization

3. Then the value of the reference CAS is compared with the value of the horizontal speed that the helicopter has at the moment of the simulation launch within the function *PIDEnhanced*, that will be better explained in the following chapter. This is the outer loop PID controller, which aim is to feed the second PID, not controlling directly the CAS, but the pitch angle. Its initialization is shown in Fig. 2.2. Therefore, the output of this function is the pitch reference.

```
pitch_reference = PIDenhanced(-mathPilotHCData.mathPilIn_cas,
                              -CAS_reference,
                              dt,
                              mathPilotGain.HC_CAS_Kp,
                              mathPilotGain.HC_CAS_Ti,
                              mathPilotGain.HC_CAS_Td,
                              mathPilotGain.HC_CAS_alpha,
                              gamma,
                              pidPar.pitch_u_n_1,
                              pidPar.pitch_ep_n_1,
                              pidPar.pitch_edf_n_2,
                              pidPar.pitch_edf_n_1,
                              PITCH_MIN,
                              PITCH_MAX,
                              1,
                              4,
                              0.025);
```

**Figure 2.2:** MathPilot HcCAS - PID outer loop

4. To obtain the command value in terms of longitudinal cyclic, a second PID controller is adopted, which receives as input the value of the reference pitch obtained from the previous block.

```
value = PIDenhanced(mathPilotHCData.mathPilIn_pitch,
                    pitch_reference,
                    dt,
                    mathPilotGain.HC_CAS_innerLoop_Kp,
                    mathPilotGain.HC_CAS_innerLoop_Ti,
                    mathPilotGain.HC_CAS_innerLoop_Td,
                    mathPilotGain.HC_CAS_innerLoop_alpha,
                    gamma,
                    pidPar.lon_u_n_1,
                    pidPar.lon_ep_n_1,
                    pidPar.lon_edf_n_2,
                    pidPar.lon_edf_n_1,
                    DCR_LONGITUDINAL_MIN,
                    DCR_LONGITUDINAL_MAX,
                    0,
                    0,
                    0.025);
```

**Figure 2.3:** MathPilot HcCAS - PID inner loop

So, as can be seen from this analysis, the key function of *MathPilots* is the PID controller, which is simple in its operation, but complicated in defining the optimum gain parameters for it to work properly.

## 2.2 PID Controller

The PID controller is a control algorithm with a predefined structure consisting mainly of three terms: proportional, integrative and derivative. Its analytical structure can be represented by:

$$u(t) = K\left(e(t) + \frac{1}{T_i}\int e(\tau)d\tau + T_d\frac{de(t)}{dt}\right) \tag{2.1}$$

which in Laplace's domain corresponds to:

$$\bar{U} = K\left(1 + \frac{1}{sT_i} + sT_d\right)\bar{E} \tag{2.2}$$

where $u(t)$ is the command value (the output of the PID controller), $e(t)$ correspond to the error, $T_i$ the integral time constant, $T_d$ the derivative time constant and $K$ the proportional gain. The equation 2.2 can be also written in the following expression, where the integral $K_i$, derivative $K_d$ and proportional $K_p$ gains are highlighted:

$$\bar{U} = \left(K_p + \frac{K_i}{s} + K_d s\right)\bar{E} \tag{2.3}$$

The PID controller goal is to provide an output command $u(t)$ that is function of the error $e(t)$ between a given reference value $\theta_{ref}$ and the actual measure of the same signal $\theta$. This makes it possible to cancel out this error and thus allow the system to evolve exactly as one would wish it to develop. To achieve this goal, PID controller exploits the three different contribution already mentioned providing the following benefits:

- The **proportional contribution** links input and output through a proportionality coefficient $K_p$ called *proportional gain*. This allows to provide a corrective action to the error, the faster the greater the gain, lowering, however, the stability of the

system. In fact, by increasing $K_p$ the system response becomes faster but more unstable, characterised by higher overshoots and oscillatory phenomena in the initial instants. In addition, if a purely proportional controller were used, it would be found that it is characterised by a static error at steady state in the case of step input and that, due to the instability phenomena that arise beyond a certain limit of $K_p$, it is not possible to control the system with the single action of this contribution. For this reason, the integrative contribution was introduced.

- The **integrative contribution** indeed allows to cancel the steady error between reference and measured value. It can be seen that the integrative contribution increases when $\theta_{ref} - \theta > 0$, while it decreases when the error is negative and it becomes equal to zero when the positive and negative contributions are equal. This is very important to take into account when modelling a PID controller as the above phenomenon can cause problems with the operation of the controller related to the wind-up phenomenon. Increasing the integrative contribution degrade the stability of the system. Given the negative effect on the system of the increase in the proportional and integrative contribution, a derivative effect is also introduced in this controller.

- The function of the **derivative contribution** is to increase the rapidity of the controller's response to possible error variances. In fact the derivative effect allows to anticipate the corrective action of the PID and therefore providing a more immediate and fast response. In fact, by definition, the derivative increases proportionally to the speed of variation of its argument (in this case of $e(t)$); so, in the case of a step input, the derivative action has the task of chasing the error and correcting it quickly. In practice, the derivative term damps the oscillations around the reference value. Finally, providine a filter to the derivative is essential in order to avoid possible amplification of disturbances.

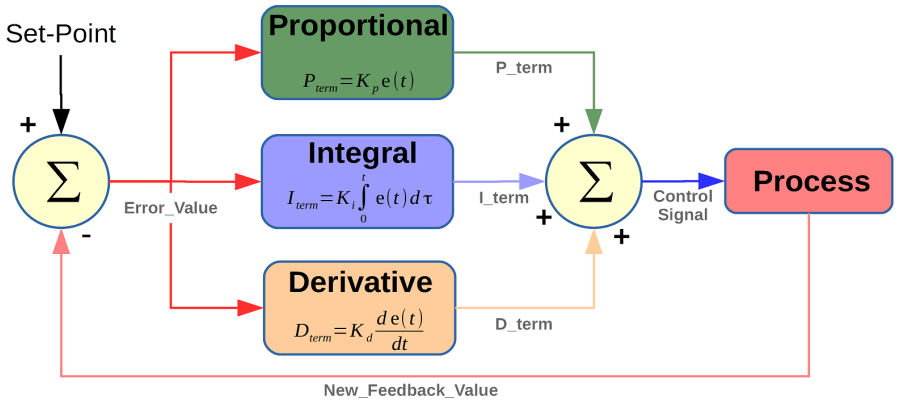


**Figure 2.4:** PID Controller scheme

**Comments** The PID controller scheme represented in Fig. 2.4 is the most simple structure that can be implemented in a control system. However, already in this example,

it is possible to observe that in order to obtain a system response that is optimal and conforms to requirements, it is necessary to find the correct combination of controller gain values $K_p$, $K_d$ and $K_i$. This process is exactly the problem that was highlighted in the *Introduction section 1*, which is time-consuming and therefore leads cost increases and inefficiency in the validation process of the simulator code. This is even more difficult and costly when using a more complex PID controller like the one implemented in the TXT simulator, where more parameters must be tuned.

### 2.2.1 TXT PID controller

The *PIDEnhanced* function, introduced during the description of the *MathPilot HcCAS*, follows the PID logic described in the previous section. Furthermore, in order to guarantee a better response it is written in a more complex structure where more parameters are set. Because of the key role that this element played in the development of the thesis project, the PID algorithm currently implemented by TXT is now described. In particular, it is written in C++ language and it follows the algorithm shown below.

Firstly, error $err$, proportional error $err_p$ and derivative error $err_d$ are defined as follows:

$$err_p(n) = \beta \cdot ref(n) - y(n) \tag{2.4}$$

$$err(n) = ref(n) - y(n) \tag{2.5}$$

$$err_d(n) = \gamma \cdot ref(n) - y(n) \tag{2.6}$$

where $ref$ is reference value, $y$ is measured value and $\beta$, $\gamma$ are two parameters to be tuned. It is also relevant to observe that this description is always referred to a discrete-time logic in which $n$ is the considered time instant.

Then, the filtered derivative error $err_{d_f}(n)$ is defined, including another parameter called $\alpha$:

$$T_f = \alpha \cdot T_d \tag{2.7}$$

$$err_{d_f}(n) = \frac{err_{d_f}(n-1)}{\frac{T_s}{T_f} + 1} + \frac{err_d(n)\frac{T_s}{T_f}}{\frac{T_s}{T_f} + 1} \tag{2.8}$$

where $T_s$ is the sample time of the simulation, $T_d$ is the derivative time constant and $T_f$ is the filtered time constant defined in equation 2.7.

The next step corresponds to the definition of incremental command $\Delta u(n)$:

$$\Delta u(n) = K_p \left[ (err_p(n) - err_p(n-1)) + \frac{T_s}{T_i} err(n) + \frac{T_d}{T_s} \left( err_{d_f}(n) - 2err_{d_f}(n-1) + err_{d_f}(n-2) \right) \right] \tag{2.9}$$

where $T_i$ is the integral time constant and $K_p$ is the proportional gain. It is possible to observe that values referred to previous time instants are present in the equation 2.9. Their modelling in a Simulink environment will be shown in the following chapters.

Before computing the absolute output $u(n)$, the incremental output $\Delta u(n)$ is checked through the **anti wind-up filter** to avoid disturbances amplifications. The filter is implemented as follows:

$$\begin{cases} \Delta u(n) = u_{max} - u(n-1) & if(\Delta u(n) > u_{max} - u(n-1)) \\ \Delta u(n) = u_{min} - u(n-1) & if(\Delta u(n) < u_{min} - u(n-1)) \end{cases} \tag{2.10}$$

where $u_{max}$ and $u_{min}$ are the two physical limits that the command can be assume depending on which kind of helicopter is considered.

Finally, the absolute output command $u(n)$ is computed through the following equation:

$$u(n) = u(n-1) + \Delta u(n) \tag{2.11}$$

**Comments** It is clear that there are more parameters in this PID algorithm than in the standard literature example. In fact, this allows to a more detailed control on the dynamic of the controller, but this means more time spent in tuning to find the best combination for these parameters. Therefore, the main goal of this project is now clear. This is finding an auto-tuning method to implement in the TXT code that is able to calculate the best combination of the PID parameters already mentioned:

$$\alpha, \beta, \gamma, K_p, T_i, T_d$$

Hence, a state-of-the-art analysis on the auto-tuning methods for controllers is now addressed.

## 2.3 State-of-the-art analysis

Proportional Integral and Derivative controllers have been used in industrial control applications for a long time and lots of industries nowadays still use this kind of controller, despite it is a simple technology used for a long time. It is possible to find several different types of techniques applied for PID tuning, of which one of the first was the Ziegler Nichols technique. These tuning methods can be broadly classified as *classical* and *computational or optimization* methods. [2]

### 2.3.1 Classical Techniques

Classical techniques are based on the concept to make particular assumptions about the plant and the desired output, trying to obtain analytically, or graphically some peculiarities of the process that is then used to decide the controller settings. The classical methods are simple to implement and computationally very fast, and work well as a first iteration

for the tuning process. Because of the assumptions made, the controller settings usually do not return the desired results directly and further tuning is required. The most famous methods are *Ziegler-Nichols Method* and *Cohen-Coon Method*.

**Ziegler-Nichols Method** The Ziegler-Nichols Method, proposed by John Ziegler and Nathaniel Nichols in 1942 is nowadays the most famous tuning method adopted in industrial applications. It is still a simple, fast and effective PID tuning method. This technique includes two different methods to obtain PID gains: [3]

1. The first one is shown in Fig. 2.5 and assigns a step disturbance to the system, and some specific variables are then measured on the step response: the delay $\tau$, the amplitude $T$ and the process static gain $\mu = \frac{\Delta y}{\Delta u}$.

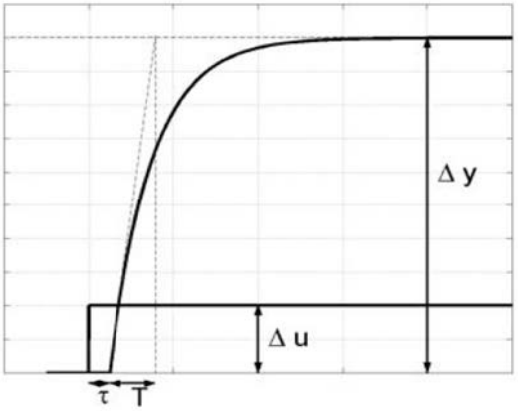


**Figure 2.5:** Ziegler-Nichols method

Then it is possible to obtain the PID gains through the expressions shown in Tb. 2.1.

| | $K_p$ | $T_i$ | $T_d$ |
|---|---|---|---|
| **P** | $\frac{T}{\mu\tau}$ | | |
| **PI** | $\frac{0.9T}{\mu\tau}$ | $3\tau$ | |
| **PID** | $\frac{1.2T}{\mu\tau}$ | $2\tau$ | $5\tau$ |

**Table 2.1:** Ziegler-Nichols PID gains expression

2. The system response to specific frequencies is the main feature of the second Zieglers-Nichols method. In fact, the controller gain parameters can be obtained on the most critical frequency points for stability, that can be found by increasing the proportional gain of the controller, until the system becomes marginally stable. The gain is defined as $K_u$ and $T_p$ is the time period. The PID parameter expressions are given in [3].

These techniques are simple and fast but their response is not optimized because the controller settings derived are rather aggressive and thus result in excessive overshoot and oscillatory response. So, they are usually adopted as a first iteration in the PID gain setting.

**Cohen-Coon Method** This method is similar to the Ziegler-Nichols technique. It is based on a step disturbance assignment to the system and a consequent evaluation of the variables $\tau$, $T$ and $\mu = \frac{\Delta y}{\Delta u}$. In this case the PID gain expressions are shown in Tab. 2.2.

| | $K_p$ | $T_i$ | $T_d$ |
|---|---|---|---|
| **P** | $\frac{3T+\tau}{3\mu\tau}$ | | |
| **PI** | $\frac{10.8T+\tau}{12\mu\tau}$ | $\tau\frac{30T+30\tau}{9T+20\tau}$ | |
| **PID** | $\frac{16T+3\tau}{12\mu\tau}$ | $2\tau\frac{32T+6\tau}{13T+8\tau}$ | $\frac{4T\tau}{11T+2\tau}$ |

**Table 2.2:** Cohen-Coon gains expression

Despite this method assesses a better model, the results of Cohen-Coon are not much better compared to the Ziegler Nichols method.

### 2.3.2 Computational or Optimization Techniques

These methods are usually adopted for data analysis or cost functions optimization, but they are also becoming increasingly important for PID tuning, using techniques like neural networks, differential evolution and genetic algorithm. In particular, computational models are used for auto tuning or self tuning of PID controllers. This means that computational or optimization techniques set the PID parameters and describe the system dynamic by using some computational models. Then they compare the outputs coming from the real system and the model, to highlight if there are any process variations. In this case, the desired response is achieved thanks to a reset of PID parameters, in fact the controller is able to compensate these possible variations by automatically adapting its parameters. There are two possible solutions to achieve these goal linked to the two types of process dynamic variations: unpredictable and predictable. The most commonly used approach to deal with predictable variations is the *gain scheduling*. In this method, different controller parameters are found and scheduled for different operating conditions thanks to an auto-tuning process. There are several algorithms found in literature, that are adopted to implement computational and optimization techniques.

**Genetic Algorithm** The *Genetic Algorithm* is a specific algorithm that investigates the search space in a similar way to nature's evolution. These methods are commonly adopted to provide high-quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover and selection. They leverage on probabilistic rules to search the most suitable solution for a given problem, using a cost function to analyze its fitness. The genetic algorithms follow the flowchart presented in

Fig. 2.6. [4]

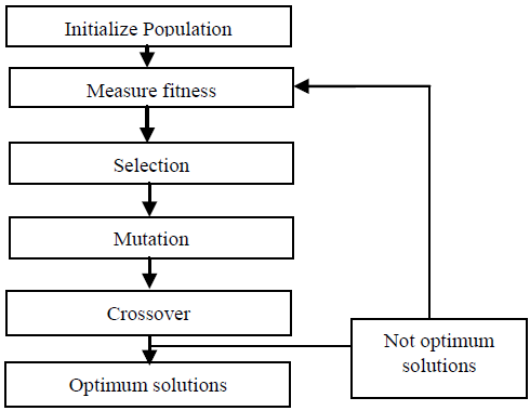


**Figure 2.6:** Genetic Algorithm flowchart

Nowadays it is possible to find lots of different application of GA in PID auto-tuning processes. However, it is a complex algorithm to be implemented in TXT code, due to its difficult integration and its high computational cost for a such structured program as TXT code.

**Artificial Neural Networks** The *Artificail Neural Networks* (ANN), also named *Neural Networks* (NNs), are computing systems inspired by the biological neural networks that constitute animal brains. This biological neural network is represented by interconnected group of processes information and artificial neurons, using a connectionist approach to computation. They are adaptive systems that change their structure based on external or internal information that flows through the network during the learning phase. A possible application in control theory is shown in Fig. 2.7. [5]

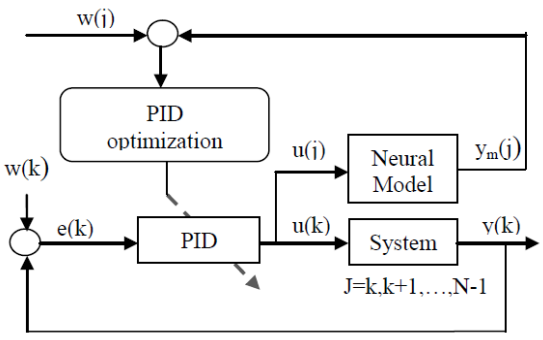


**Figure 2.7:** ANN controller flowchart

These methods have a great potential, but these are still scarcely used in control systems today because of some inherent drawbacks such as the number of layers and the numbers of neurons per layer that are often hard to be determined.

**Fuzzy Logic** *Fuzzy Logic Control* (FLC) is one of the possible interfaces between artificial intelligence and control engineering. This fuzzy logic can be applied on PID controller allowing to vary its parameters according to the change of the signals error. The core part of this auto-tuning method is the *fuzzy rule* that must be adopted and depends on the kind of plant to be controlled and practical experience. In particular, it is difficult to apply this technique to an integral rule, leading to a not optimal solution in PID applications. The scheme of a fuzzy-tuned PID controller is shown in Fig. 2.8. [6]

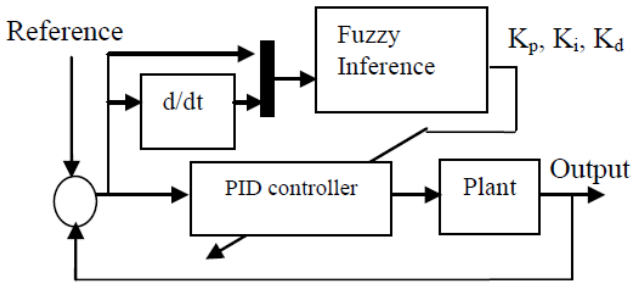


**Figure 2.8:** Fuzzy PID scheme

## 2.4 Comments

It can be seen that both the *classical techniques* and the *computational and optimisation methods* analysed so far have important limitations for the main objective of this thesis project. In fact, the writing of a code that follows these control logic is in one case not optimal, as for the classical techniques, while in the other it would be too complex and difficult to integrate into the company model. For this reason, it was decided to analyse whether a software was available on the market today which, if appropriately used, could be adapted to the problem of TXT.
What was found after a deep analysis is that the *Matlab-Simulink* programming software presents itself as a very advantageous development environment. In fact, its flexibility, the possibility of interaction between *Matlab* and *Simulink*, together with the availability of using special control toolboxes included in the software, influenced the decision of trying whether it was possible to find a solution to the problem highlighted by TXT by exploiting this environment.
In particular, it was decided to understand how to use and adapt, to the TXT problem, the *Control System Toolbox* linked to *Matlab* and the *Control Design Toolbox* integrated in *Simulink*. Through an appropriate modelling of the system in question, these applications allow to exploit some optimisation algorithms based on *Hinf norm*, that have already been implemented in these toolboxes, to obtain the best tuning of PID controllers.

Once the problem to be addressed was understood, the state of the art was analysed and the reference software to be used was chosen, it was then possible to start modelling and searching for possible solutions to the thesis problem.

# Chapter 3

# Helicopter model used

Before proceeding with the modelling of the system on *Matlab* and *Simulink*, it is necessary to focus on the chosen helicopter and reference system. In fact, it is important to note that the definition of the reference system according to the type of helicopter is absolutely fundamental, since a sign change in the reference system or a different kind of aircraft can influence the convergence of its model.

## 3.1 Non linear model

It was found that the flight models of the helicopters used by TXT all refer to a single source, namely the one presented in the reference [7]. The **equations of translational and rotational motion** of the helicopter are written considering the helicopter as a rigid body, referred to an axes system fixed at the centre of the aircraft (assumed to be fixed). The three reference axes $x, y, z$ are shown in Fig. 3.1.

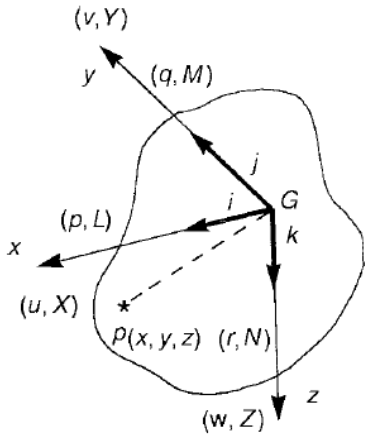


**Figure 3.1:** Helicopter fuselage reference axes system

Where $u, v, w$ measured in $m/s$ are the velocity components, $p, q, r$ measured in $rad/s$ are the angular rates that are time-varying under the action of applied forces $X, Y, Z$ measured in $N$ and moments $L, M, N$ measured in $Nm$.

Starting from this reference system, the **orientation** of the body axes relative to the Earth is defined through the Euler angles following a standard rotation sequence used in flight dynamics 321. This allows to completely define the attitude of the helicopter thanks to the yaw $\psi$, pitch $\theta$ and roll $\phi$, measured in $rad$, as can be seen in Fig. 3.2.

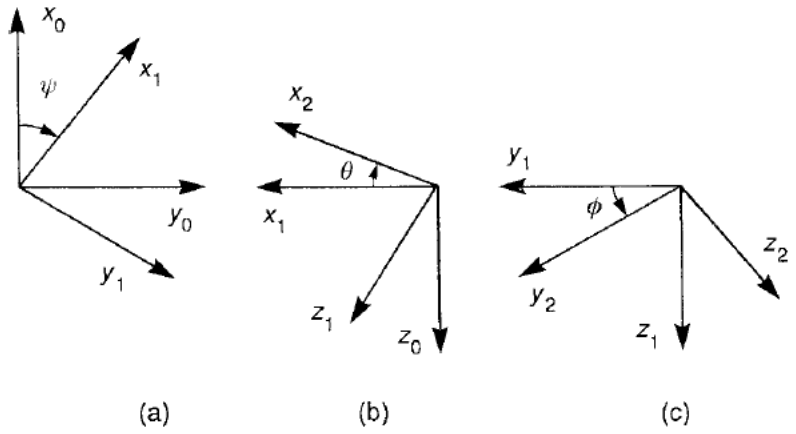


**Figure 3.2:** Attitude angles definition: (a) yaw, (b) pitch, (c) roll

In **rotor dynamics analysis**, there are mainly three reference axes systems used in literature: the hub system, the no-feathering system and the tip-path plane system. In Fig. 3.3 are shown these reference plane where the hub plane is set horizontal for convenience.

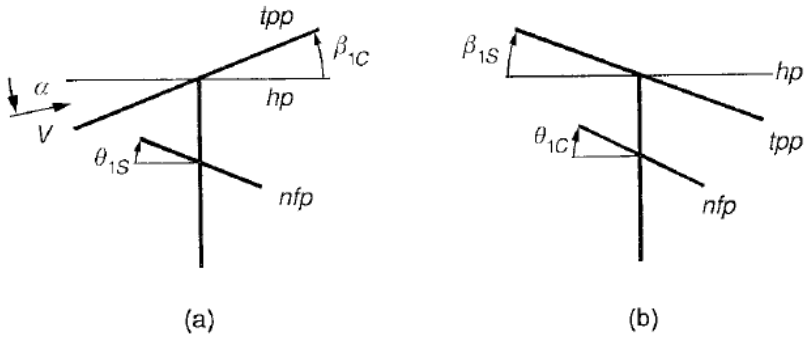


**Figure 3.3:** Rotor dynamics reference plane: (a) longitudinal plane, (b) lateral plane

In Fig. 3.3, the angles $\theta_{1S}$, $\theta_{1C}$, $\beta_{1S}$ and $\beta_{1C}$ are drawn. The first two angles correspond to the longitudinal and lateral cyclic pitch (subscript w denotes hub/wind axes) measured in *rad*, while $\beta_{1S}$ and $\beta_{1C}$ are the rotor blade longitudinal and lateral flapping angles (subscript w denotes hub/wind axes), written in multi-blade coordinates and measured in *rad*.

A better representation of the considered aircraft dynamic model is shown in Fig. 3.4. In particular, this figure depicts a trim condition in hover. .

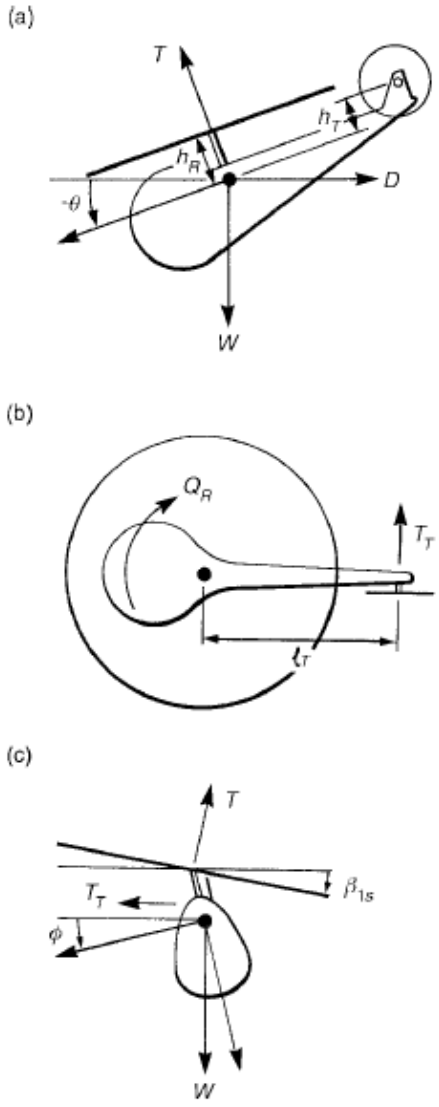


**Figure 3.4:** Representation of trim in hover: (a) longitudinal (view from port); (b) yaw (view from above); (c) roll (view from front)

## 3.2 Linear model

The aircraft model described by [7] and consequently the one adopted by TXT are therefore based on the reference systems seen previously with regard to rigid body dynamics and rotor dynamics. These are described by non-linear equations which are essentially based on a 6 DoF model of which the main variables are contained in the *states vector* $\boldsymbol{x}$:

$$\boldsymbol{x} = [u, v, w, \phi, \theta, \psi, p, q, r] \tag{3.1}$$

whose meaning has already been introduced in the previous section.

As far as the *command vector* $\boldsymbol{u}$ is concerned, it is composed by four components: main rotor collective $\theta_0$, longitudinal cyclic $\theta_{1S}$, lateral cyclic $\theta_{1C}$ and tail rotor collective $\theta_{0T}$.

$$\boldsymbol{u} = [\theta_0, \theta_{1S}, \theta_{1C}, \theta_{0T}] \tag{3.2}$$

However, it is possible to linearise this non-linear model by considering the helicopter around a specific equilibrium point. This allows to obtain a simplified description of the flight model of the aircraft, but accurate when considering the dynamics of the helicopter around its equilibrium point. Following, in fact, the linearization process presented in [7], it is possible to obtain the linearized equations of motion for the full 6 DoFs, describing perturbed motion about a general trim condition, that can then be written as:

$$\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{u} \tag{3.3}$$

where $\boldsymbol{A}$ and $\boldsymbol{B}$ are respectively the *state matrix* and the *command matrix*. The coefficients of these matrices contain the linearized aerodynamic forces and moment, perturbational inertial, and gravitational and kinematic effects linearized about the trim condition. In fact, these matrices vary not only depending on the aircraft, but also on the specific equilibrium condition in which the helicopter is at the moment of linearization. Moreover, the elements of $\boldsymbol{x}$ and $\boldsymbol{u}$ in 3.3 correspond to the perturbation $\Delta$ of the states and commands from the initial trim condition.

## 3.3 Choice of the model

Therefore, it is clear that to describe the behaviour of the aircraft, two models are available: a linear model, that describe the aircraft dynamics within an equilibrium point, and a non-linear model, that fully describes the behaviour of the helicopter. Since the aim of this thesis is to design a system that can automatically obtain the gains to be included in the PID controllers that are used in TXT's *MathPilot*, it is evident that the linear model is more flexible and easy to use to pursue this aim in a *Matlab-Simulink* environment. Thus, a linear model allows analyses to be carried out quickly and rapidly, providing greater flexibility than a non-linear model. However, it has the main problem of the calculation of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, which could make the implementation of the linear model impossible in specific conditions.

Furthermore, thanks to a private algorithm, TXT is able to calculate such matrices from each non-linear helicopter flight model. *This allowed to start the development of the thesis project using a linearized approach to the problem.*

### 3.3.1 Description of the linearized flight model

To begin the modelling that will be presented in the next chapter, it was decided to start by using the linearized matrices that are present in literature at reference [7]. They are related to the *Westland Lynx Mk7*, a utility/battlefield helicopter in the 4.5-ton category of the British Army Air Corps shown in Fig. 3.5.

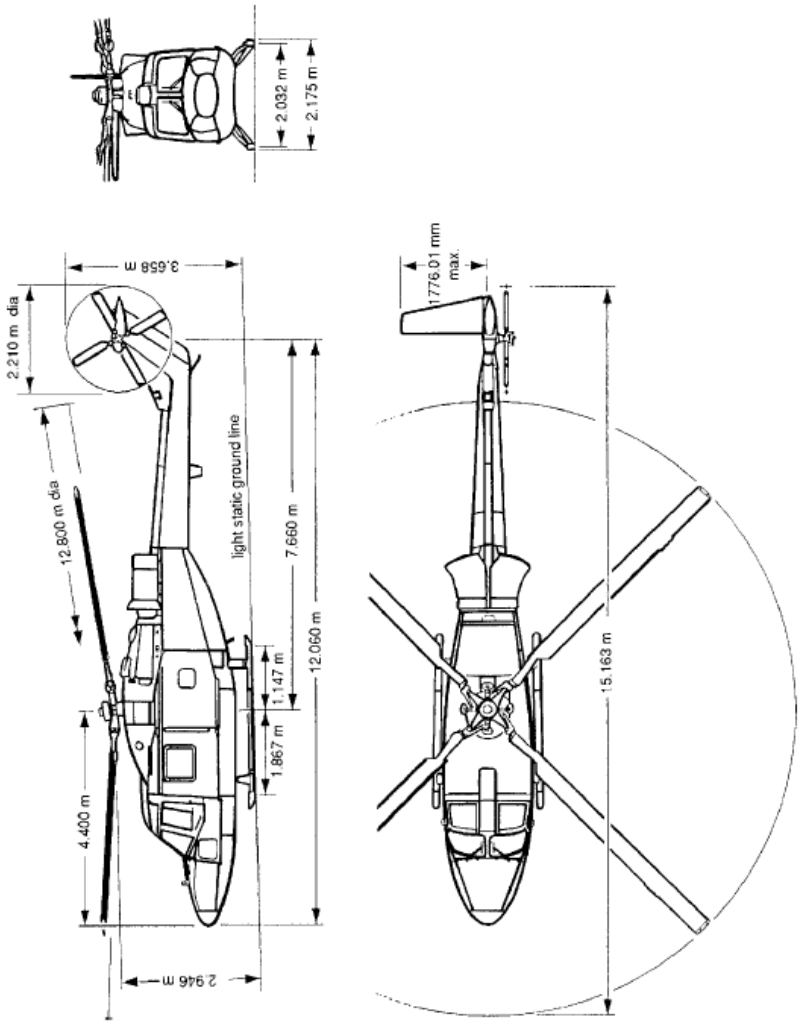


**Figure 3.5:** Westland Lynx Mk7

The $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices of this aircraft were obtained for several flight conditions corresponding to different flight regimes ranging from initial 0 kts to final 120 kts, with a sampling step of 20 kts. A states vector $\boldsymbol{x}$ composed by 8 components has been considered, as the heading is neglected, structured as follows:

$$\boldsymbol{x} = [u, w, q, \theta, v, p, \phi, r] \tag{3.4}$$

while the commands vector $\boldsymbol{u}$ is the same defined in equation 3.2.

It was decided to adopt as reference matrices those related to the flight condition of $V = 40\ kts$. The states matrix $\boldsymbol{A}$ and the command matrix $\boldsymbol{B}$ are shown below:

$$\boldsymbol{A} = \begin{bmatrix} -0.0146 & 0.0347 & -0.5681 & -9.7934 & -0.0083 & -0.1321 & 0.0000 & 0.0000 \\ -0.1186 & -0.6156 & 20.6855 & -0.5779 & -0.0180 & -0.2022 & 0.3519 & 0.0000 \\ 0.0319 & 0.0212 & -2.1033 & 0.0000 & 0.0277 & 0.4210 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.9994 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0359 \\ 0.0070 & 0.0184 & -0.1303 & 0.0205 & -0.0915 & 0.5342 & 9.7869 & -20.3077 \\ -0.0255 & 0.3040 & -2.1361 & 0.0000 & -0.1949 & -10.7839 & 0.0000 & -0.1441 \\ 0.0000 & 0.0000 & -0.0021 & 0.0000 & 0.0000 & 1.0000 & 0.0000 & 0.0590 \\ -0.0325 & 0.0314 & -0.2522 & 0.0000 & 0.0316 & -1.8857 & 0.0000 & -0.68597 \end{bmatrix}$$

$$\boldsymbol{B} = \begin{bmatrix} 4.8686 & -8.5123 & 2.0305 & 0.0000 \\ -95.5241 & -12.7586 & 0.0003 & 0.0000 \\ 7.2883 & 27.0667 & -5.7827 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 1.1239 & -1.8435 & -9.3132 & 3.3289 \\ 27.3295 & -30.1532 & -153.4552 & -0.6662 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 15.9423 & -5.8252 & -27.2699 & -8.9726 \end{bmatrix}$$

Thanks to the definition of the helicopter reference linearized model, it is now possible to describe the process followed to obtain a system that can auto-tune the PID controllers of TXT's MathPilots.

# Chapter 4

# Modelling phase

The first step in modelling process was to reproduce in *Matlab-Simulink* environment the *MathPilot HcCAS* already described in chapter 2.1.1.

## 4.1 MathPilot HcCAS

As the key function of the *MathPilot HcCAS* was the PID controller highlighted in chapter 2.2.1, it was decided to start trying to implement this block following the diagram shown in Fig. 4.1.

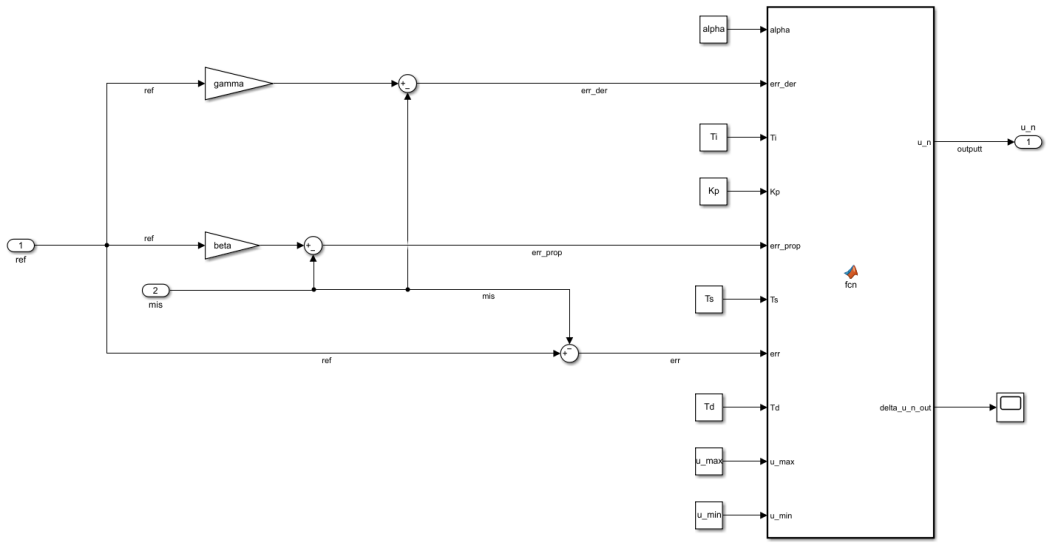


**Figure 4.1:** PIDEnhanced Simulink implementation - first version

In this first implementation of the *PIDEnhanced* block, it can be seen that the core functionality of this model is contained in the Simulink's *Matlab function* block that is fully

reported in appendix A.1. Since the *MathPilot* code contained values referring to different instants of time (in particular, there were values related to $n, n-1, n-2$) it was essential to take into account this feature. As shown in code in appendix A.1, this goal was achieved exploiting the Matlab function named *persistent*. In fact, it is possible to declare a variable as *persistent* and this remains local to the specific function in which it is declared, yet its values are retained in memory between calls to the function. Moreover, it can be noticed that in Fig. 4.1 the parameters of the PID controller ($T_i, T_d, K_p, \alpha, \beta, \gamma, u_{max}, u_{min}$) are initialized through the Simulink *constant* and *gain* blocks and than given to the *Matlab function*.

### 4.1.1 First model

It was then possible to move forward by inserting this PID controller model into the MathPilot loop. This has been done by creating a special Simulink subsystem which can be parameterised appropriately by interacting with an interface mask. As shown in Fig. 4.2 a reference input was given to the loop through a step command in terms of horizontal velocity $V_x$ measured in $m/s$. This is the input of the outer loop PID controller which, depending on the error between the reference and the measured value, outputs a command that corresponds to the input of the inner loop PID controller. This command is a pitch reference that is compared with the pitch state coming from the linear model of the helicopter. The units of measurement are made consistent by setting everything in *rad*. The output of the inner loop PID controller is the effective command related to the longitudinal cyclic $\theta_{1S}$. The linearised model of the aircraft is described using a Simulink block called *State-Space*, in which the state and command matrices ($\boldsymbol{A}$, $\boldsymbol{B}$) are inserted. Opening this block, it can be seen that two matrices are also required: matrix $\boldsymbol{C}$ (output matrix) and matrix $\boldsymbol{D}$ (equal to the null matrix for physical systems).

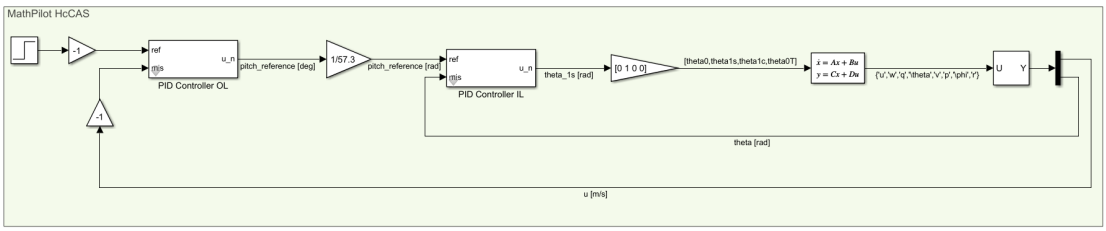


**Figure 4.2:** Simulink modelling of HcCAS loop

Matrix $\boldsymbol{C}$ allows the user to select which states are to be provided as system outputs. In this specific case, due to the intention of observing all the states of the aircraft, but also the necessity of feedback only the horizontal velocity $u$ and the pitch angle $\theta$, the output of the *State-Space* block is a vector of eight components, then selected thanks to the Simulink *Selector* block.

### 4.1.2 Second model

Before running the simulation, it was obviously necessary to carry out a tuning process of the PID controller parameters which was still done manually using a trial-and-error method. In fact, in this initial phase of the project, the intention was to validate the MathPilot model and the PID controller, and only at a later stage to introduce the *Control System Toolbox* for the auto-tuning process. After several attempts, it was observed that, due to the fact that this MathPilot allowed only one command to be controlled, while the aircraft dynamics was influenced by the eight state variables introduced in the previous chapter, the tuning of the controller was complex. It was therefore decided to introduce two additional controllers that allowed the lateral cyclic $\theta_{1C}$ and the collective $\theta_0$ commands to be controlled simultaneously, as shown in Fig. 4.3.

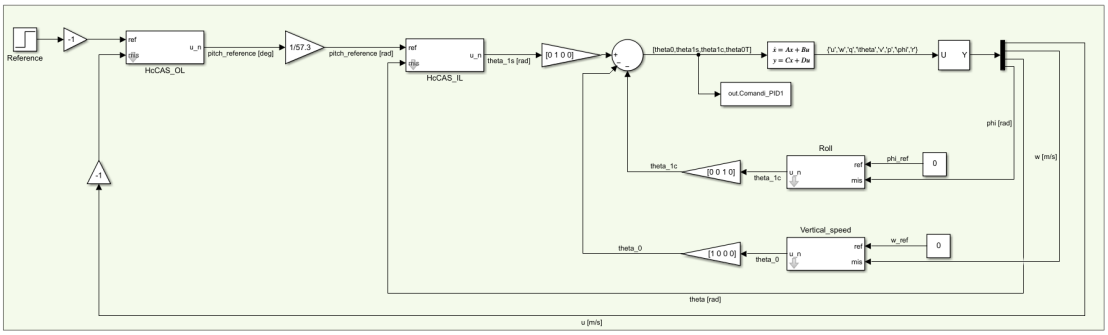


**Figure 4.3:** Simulink modelling of HcCAS loop - second version

The following four controllers were then considered:

- **PID HcCAS Outer Loop**: it receives the reference and the measured values in terms of horizontal velocity and output the pitch reference for the following PID controller;

- **PID HcCAS Inner Loop**: it receives the reference and the measured values in terms of pitch angle and output the longitudinal cyclic command $\theta_{1S}$;

- **PID Roll** it receives the reference and the measured values in terms of roll angle (the reference roll is set equal to zero $\phi_{ref} = 0\ rad$) and output the lateral cyclic command $\theta_{1C}$;

- **PID Vertical Speed** it receives the reference and the measured values in terms of vertical speed (the reference vertical speed is set equal to zero $w = 0\ m/s$) and output the collective command $\theta_0$.

The controller's gains were therefore as follows:

| | $K_p$ | $T_i$ | $T_d$ | $\alpha$ | $\beta$ | $\gamma$ | $u_{min}$ | $u_{max}$ |
|---|---|---|---|---|---|---|---|---|
| **PID HcCAS Outer Loop** | 4 | 1 | 1 | 1 | 0.5 | 2 | 25° | -25° |
| **PID HcCAS InnerLoop** | 2 | 0.5 | 0.08 | 1 | 0.9 | 0.1 | 1 rad | -1 rad |
| **PID Roll** | 0.05 | 10 | 20 | 1 | 1 | 0 | 1 rad | -1 rad |
| **PID VS** | 1.2 | 0.8 | 1 | 1 | 1 | 1.2 | 1 rad | -1 rad |

**Table 4.1:** MathPilot HcCAS - PID controllers gain

Running the simulation for a time period of $t_{sim} = 30\ s$, considering a sample time of $dt = 1/120 = 0.008\ s$ and giving a step reference signal of $\Delta u = 1\ m/s$ the following results were obtained in Fig. 4.4:

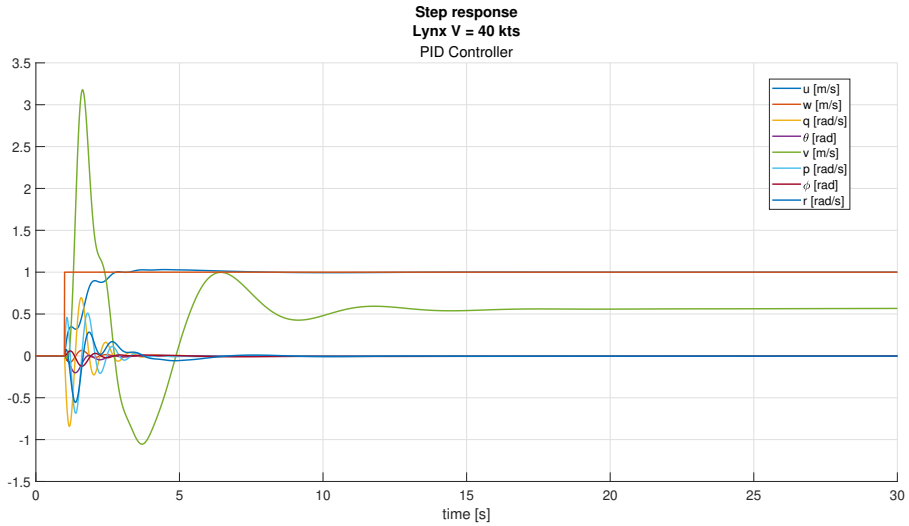


**Figure 4.4:** Step response - MathPilot HcCAS

It can be observed that all the states that are controlled through the three PIDs, after a short transient period of about $t = 10\ s$, tend to follow the references. This means that the controllers are working in a correct way. However, it can be seen that the lateral speed $v$ tends to oscillate significantly in the transient and stabilise at a non-zero value. This is due to the lack of active control of the PID on this axis, this is acted by the tail rotor collective $\theta_{0T}$.

Moreover, the following results, shown in Fig. 4.5, were obtained from the commands:

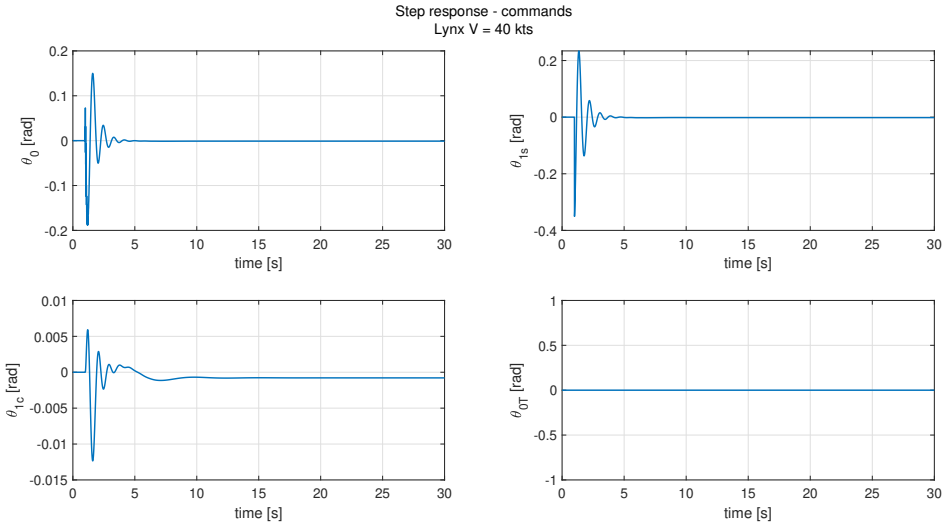


**Figure 4.5:** Step response commands - MathPilot HcCAS

As expected, indeed, it can also be seen from this plot that only three commands are operated, while the tail rotor remains at zero as no control has been placed on this axis.

**Comments** It is now clear what the significance of the MathPilots used by TXT is. It can be seen that what is being operated are the pilot commands. These models, indeed, represent the action that the pilot would virtually perform on the controls to stabilise the aircraft when a horizontal speed step of $\Delta u = 1\ m/s$ is required. In addition, it can be seen that the Simulink model implemented, and in particular the PID controller taken from the TXT code, allow the aircraft to be stabilised and thus validate their performance.

### 4.1.3 Third model

The next step wanted to fully stabilise the aircraft on all axes by inserting an additional PID controller. In addition to the four controllers shown in Fig. 4.3, the following one was inserted in the loop:

- **PID Lateral Speed** it receives the reference and the measured values in terms of lateral speed (the reference lateral speed is set equal to zero $v = 0\ m/s$) and output the tail rotor collective command $\theta_{0T}$.

It allowed to asses the feedback loop shown in Fig. 4.6.

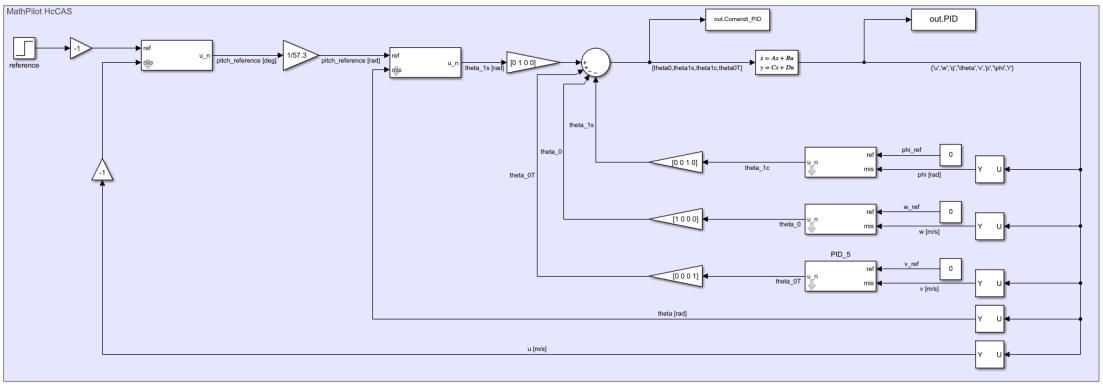


**Figure 4.6:** Simulink modelling of HcCAS loop - third version

The gains in Tab. 4.2 were adopted in this model, obtained through a trial and error process. Compared to the previous case, only the lateral speed PID controller has been added.

| | $K_p$ | $T_i$ | $T_d$ | $\alpha$ | $\beta$ | $\gamma$ | $u_{min}$ | $u_{max}$ |
|---|---|---|---|---|---|---|---|---|
| **PID HcCAS Outer Loop** | 4 | 1 | 1 | 1 | 0.5 | 2 | 25° | -25° |
| **PID HcCAS InnerLoop** | 2 | 0.5 | 0.08 | 1 | 0.9 | 0.1 | 1 | -1 |
| **PID Roll** | 0.05 | 10 | 20 | 1 | 1 | 0 | 20° | -20° |
| **PID VS** | 1.2 | 0.8 | 1 | 1 | 1 | 1.2 | 20° | -20° |
| **PID LS** | 1.2 | 0.8 | 1 | 1 | 1 | 1.2 | 20° | -20° |

**Table 4.2:** MathPilot HcCAS fully controlled - PID controllers gain

It was then possible to run the simulation with a time period of $t = 30\ s$ and a sample time of $dt = 1/120\ s$. The results displayed in Fig. 4.7 and 4.8 were obtained.

**Comments** It is very important to note that when analysing a linearised system, what is plotted are the perturbations of states and commands values with respect to a very precise equilibrium condition. In this specific case, as pointed out in the previous chapter, the Lynx helicopter is considered in a flight condition with a horizontal velocity of $u = 40\ kts$, and then this velocity is increased in the simulation of $\Delta u = 1m/s = 1.9kts$ . This triggers the aircraft's free dynamic modes, which must be balanced by the pilot's action on the controls. TXT's MathPilot loop (and in particular the PID controllers contained within it) replicates this action. In Fig. 4.7 and 4.8, it can be observed that all states, after a transient period, reach the reference value. In particular, the horizontal velocity $u$ reaches the step input, while all the other states, including the lateral velocity $v$, tend to stabilise and become zero.

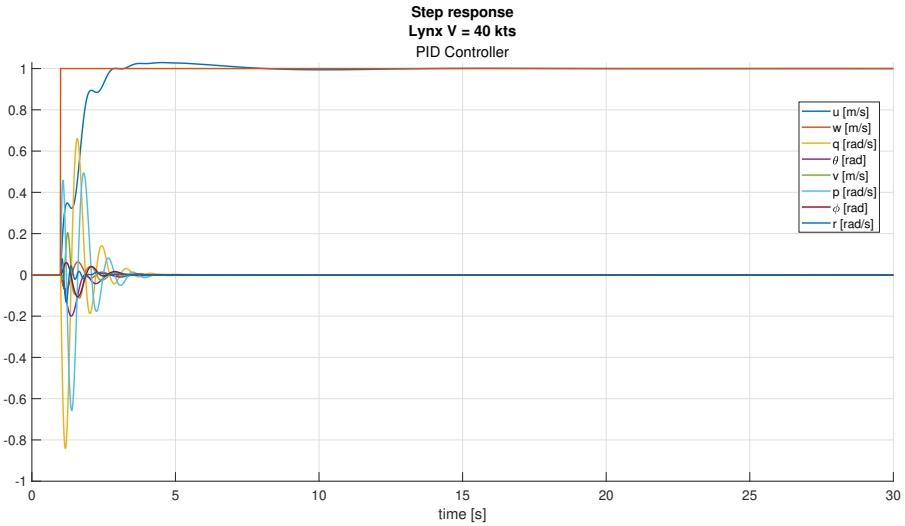


**Figure 4.7:** Step response - MathPilot HcCAS third version

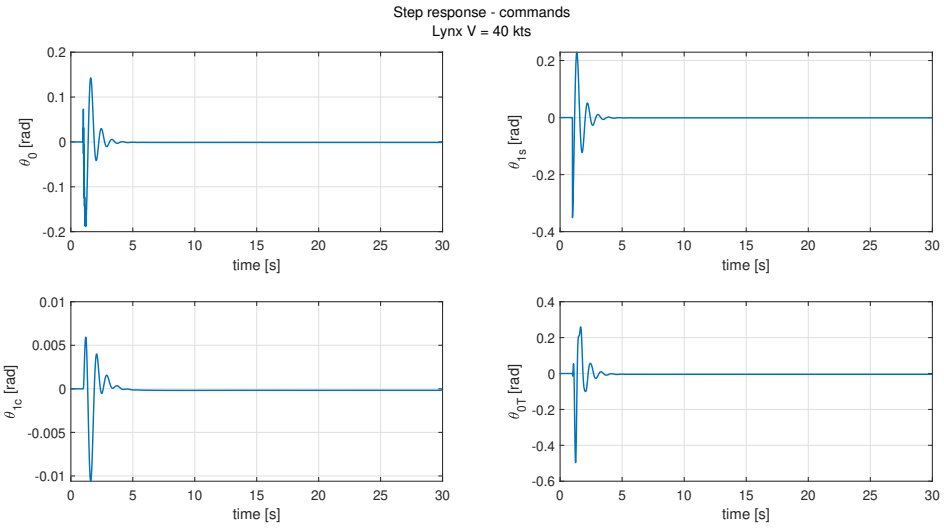


**Figure 4.8:** Step response commands - MathPilot HcCAS third version

Once the MathPilots and PID controllers had been validated, it was possible to analyse the functioning of Matlab's *Control System Toolbx* and Simulink's *Control Design App* to understand how to insert them into the model just obtained.

## 4.2 PID autotuning design

The state-of-the-art analysis of current technologies used in the PID auto-tuning process showed that the Matlab-Simulink environment offers possible solutions for this project. In particular, two specific toolboxes were used in the developing of the autotuning system:

- the *Control System Toolbox*, described in [8] provides algorithms and apps for systematically analyzing, designing, and tuning linear control systems. This toolbox automatically tunes both SISO and MIMO compensators, including PID controllers. It allows to tune gain-scheduled controllers and specify multiple tuning objectives, such as reference tracking, disturbance rejection, and stability margins, validating the model design by verifying rise time, overshoot, settling time, gain and phase margins, and other requirements;

- *Simulink Control Design* allow to design and analyze control systems modeled in Simulink. It is possible to automatically tune arbitrary SISO and MIMO control architectures, including PID controllers. The key functionality of this toolbox is that PID autotuning can be deployed to embedded software for automatically computing PID gains in real time. In fact, the user can find operating points and compute exact linearizations of Simulink models at various operating conditions. [9]

In particular, the interaction between Matlab and Simulink environments is made possible by these two applications. The key functions used in the development of TXT's PID controller autotuning model are the following ones:

1. *slTuner* creates an interface, named *st* for tuning the control system blocks of a specific Simulink model. The interface adds the linear analysis points marked in the model as analysis points of *st*. In particular, *slTuner* provides an interface between a Simulink model and the tuning commands *systune* in Matlab code and allow the user to specify the control architecture, designate and parameterize blocks to be tuned, tune the control system, validate design by computing (linearized) open-loop and closed-loop responses and write tuned values back to the model. Because tuning commands such as *systune* operate on linear models, the *slTuner* interface automatically computes and stores a linearization of the Simulink model. [10]

2. *systune* tunes fixed-structure control systems subject to both soft and hard design goals. This function can tune multiple fixed-order, fixed-structure control elements distributed over one or more feedback loops. This optimisation function receives as input the linearised model *st* obtained by *slTuner* and, depending on certain tuning goals chosen by the user, calculates the gains of the selected blocks that best meet the optimiser's requirements. [11]

However, when analysing the working process of these functions, it can be seen that not all Simulink blocks can be parameterised and therefore provided as input to the optimiser. In fact, only the following blocks are compatible with the *systune* function: [12]

- *tunable Gain* block;

- *tunable PID Controller* block;
- *tunable Transfer Function* block;
- *tunable State-Space Model* block.

Therefore, it is clear that the PID controller model shown in Fig. 4.1 could not be used, as it was incompatible with the functions just described. In fact, it was necessary to reformulate this model so that it could be used by the *systune* function.

### 4.2.1 MathPilot HcCAS - autotuning

Before proceeding with this reformulation of the controller, it was decided to understand how to use these functions appropriately in a more simplified context. In fact, it was planned to test the functioning of the autotuning process by replacing the PID subsystems within the MathPilot HcCAS loop with the default Simulink *tunable PID Controller* blocks as shown in Fig. 4.9. These controllers have a simpler structure than the algorithm implemented by TXT in *PIDEnhanced*. In fact, they are characterised by the following expression:

$$u = K_p + K_i \frac{1}{s} + K_d \frac{s}{T_f \cdot s + 1} \tag{4.1}$$

It is defined in frequency domain and it contains four parameters: the proportional gain $P$, the integrative gain $I$, the derivative gain $D$ and the filter parameter $N$.

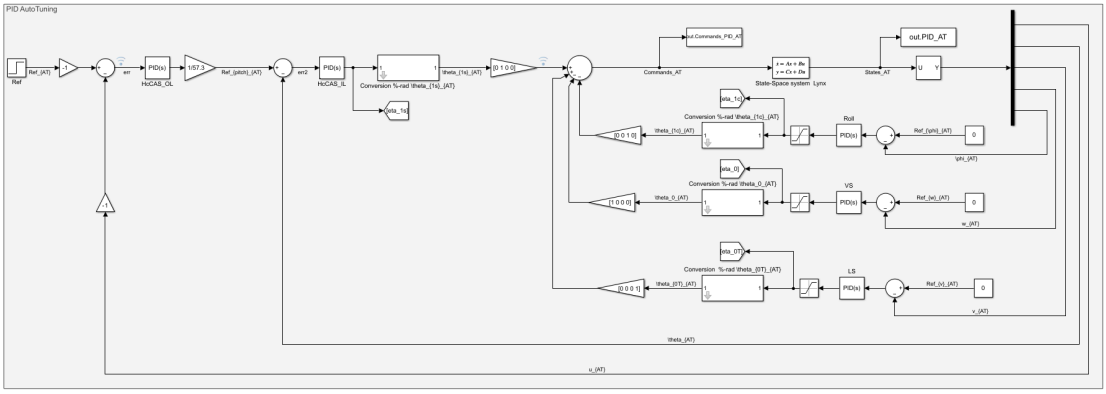


**Figure 4.9:** MathPilot HcCAS - PID controller autotuning method

It can be seen that in this model, subsystems have been placed downstream of the PID controllers. In fact, in this case, the output of the PID controller is not directly the tilt angle of the swash plate but the percentage value of the command. In fact, unlike the previous case in which the main aim was to validate the operation of the MathPilot implemented in Simulink, in this case the goal is to proceed by refining the model. In particular, the percentage value of the command is multiplied by the difference between the maximum value that this command can assume and its neutral value. These values

are reported in appendix A.2.Therefore, by using the code in appendix A.3 and running the program, it was possible to verify the correct functioning of the autotuning algorithm described above. In particular, the following goals were imposed on the optimiser:

- *TuningGoal.StepTracking* to specify a target step response from specified inputs to specified outputs of the control system. The following reference model was set to constraint the linearized system to match its dynamics: [13]

$$ReferenceModel = \frac{1}{s+1} \tag{4.2}$$

  Moreover, the constraint is satisfied when the relative difference between the tuned and target responses falls within a tolerance of 0.2.

- *TuningGoal.Margins* creates a tuning goal that specifies the minimum gain and phase margins at the specified location in the control system. They were set as in appendix A.3, based on literature values. [14]

- *TuningGoal.Poles* to constrain the closed-loop dynamics of a control system within a region defined by the user. This goal was set based on literature values presented in appendix A.3.

Finally, it can also be seen that upper (Maximum) and lower (Minimum) limits have been imposed on the parameters of the Simulink PID blocks. This forces the optimiser to search for gains optimal values within a space that is actually compatible with TXT MathPilots. These limit values have been deduced by assessing the values that TXT currently includes in its controllers. It was then possible to run the simulation with a time period of $t = 30\ s$ and a sample time of $dt = 1/120\ s$. The results obtained are shown in Fig. 4.10 and 4.11.

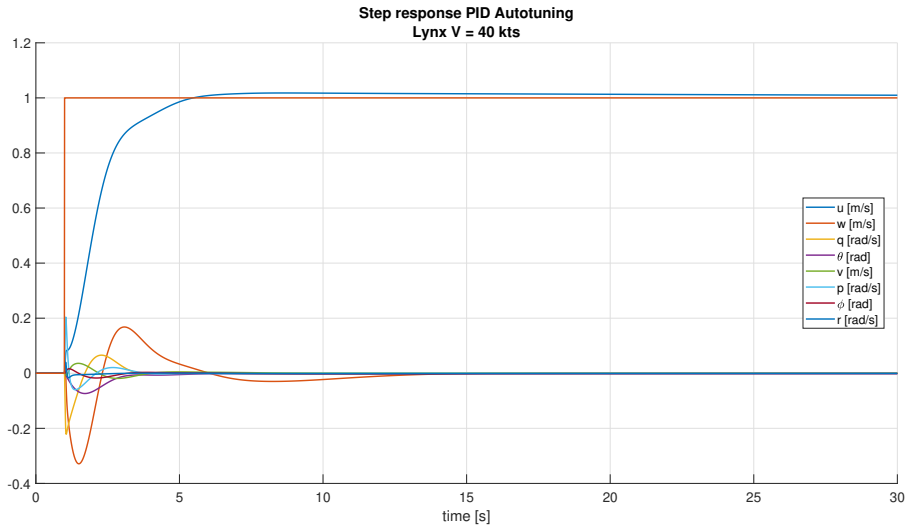


**Figure 4.10:** Step response - MathPilot HcCAS autotuning

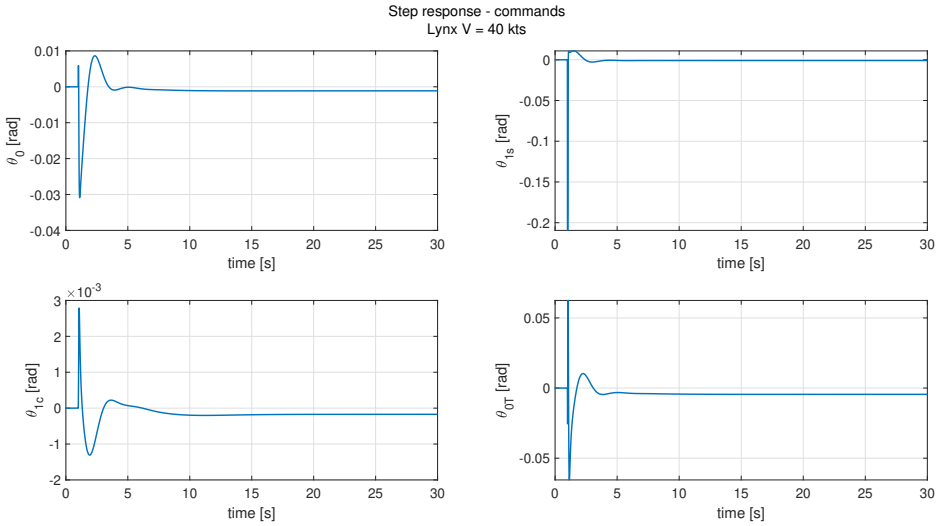


**Figure 4.11:** Commands step response - MathPilot HcCAS autotuning

The gains calculated by the optimiser are therefore as follows:

| | $K_p$ | $K_i$ | $K_d$ | $T_f$ |
|---|---|---|---|---|
| **PID HcCAS Outer Loop** | 4.72 | 0.142 | 6e-11 | 0.108 |
| **PID HcCAS InnerLoop** | 5 | 9.75 | 2 | 0.0143 |
| **PID Roll** | 0.61 | 0.0208 | 0.106 | 0.0178 |
| **PID VS** | 0.0857 | 0.0382 | 0.0944 | 0.0872 |
| **PID LS** | 1.28 | 0.775 | 0.996 | 0.0157 |

**Table 4.3:** PID controllers gains - Autotuning

**Comments** As shown in the previous plots, in this case the optimiser allowed all states to be adequately controlled. In fact, the solutions tend towards the reference values provided as input to the controllers. Fig. 4.12 shows the three responses obtained with the three models presented so far for comparison. It can be seen that the linear speed response in the case of autotuning is slower than in the previous cases, but with a very low percentage error of less than 1%. Obviously, the response of the first two manually tuned models is coincident as far as the horizontal speed $u$ is concerned, the only difference indeed is related to the lateral speed $v$ control. A variation of $v$ does not particularly affect the horizontal velocity $u$ as shown in the Lynx's state matrices.

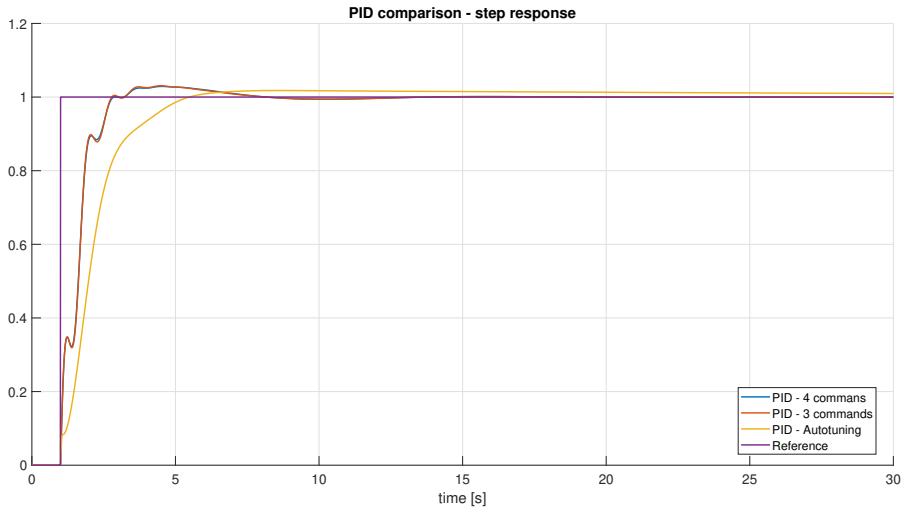


**Figure 4.12:** Step respons - models comparison

Once the functioning of the optimiser had been validated, the PID controller using the TXT algorithm was remodelled so that it could be exploited by the *systune* function.

### 4.2.2 Remodelling of PIDEnhanced for autotuning process

Since the systune function only accepts specific types of Simulink blocks, it is necessary to understand how to appropriately model the PIDEnhanced algorithm presented above. The blocks that provide the most flexibility in modelling correspond to the *tunable Gain*. Analysing the algorithm presented in section 2.2.1, it can be seen that the proportional, integrative and derivative contributions are easily linked to the gain blocks through the following equations:

$$Gain_1 = K_p \tag{4.3}$$

$$Gain_2 = \frac{T_s}{T_i} \tag{4.4}$$

$$Gain_2 = \frac{T_d}{Ts} \tag{4.5}$$

However, a problem occurs with regard to the contribution of the filter. In fact, within the definition of the filtered derivative error $err_{d_f}$ it is included the filter time constant $T_f = \alpha \cdot T_d$, that is also included in the equation 2.9 to obtain $\Delta u(n)$. This does not allow to close the system and leads to the necessity of finding another solution. Hence, it was therefore decided to elaborate on the expression of $err_{d_f}$ as follows:

$$err_{d_f}(n) = \frac{err_{d_f}(n-1)}{\frac{T_s}{T_f}+1} + \frac{err_d(n)\frac{T_s}{T_f}}{\frac{T_s}{T_f}+1} + err_d(n) - err_d(n) \tag{4.6}$$

$$err_{d_f}(n) = err_d(n) + \frac{1}{\frac{T_s}{T_f} + 1} \left[ err_{d_f}(n-1) + err_d \right] \tag{4.7}$$

in order to obtain an expression in which it is possible to isolate the contribution of $T_f$, increasing the system by one degree of freedom. In fact, the following gain was considered for the Simulink model:

$$Gain_4 = \frac{1}{\frac{T_s}{T_f} + 1} \tag{4.8}$$

Therefore, it was possible to create the Simulink model in Fig. 4.13 in which the parameters to be tuned $K_p, T_i, T_d, T_f$ are contained in the four gains just highlighted.

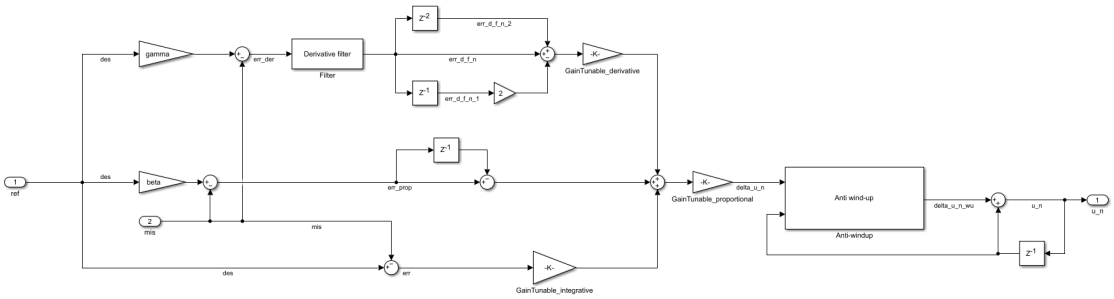


**Figure 4.13:** Remodelled PID controller

The derivative filter is presented in Fig. 4.14.

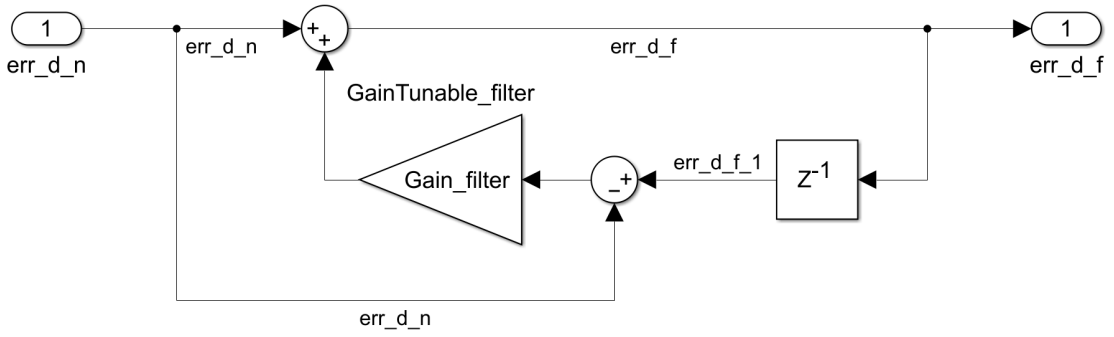


**Figure 4.14:** Remodelled PID controller - filter detail

The wind-up filter has been modelled according to Simulink logic using the *Switch* block as shown in Fig. 4.15.

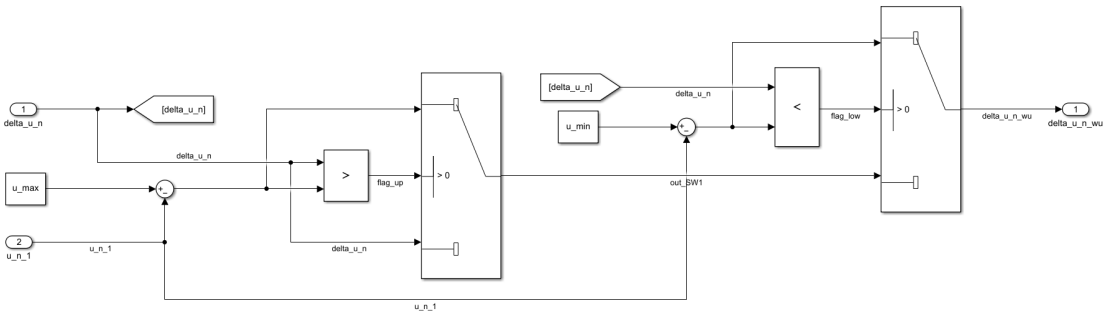


**Figure 4.15:** Remodelled PID controller - anti wind-upfilter detail

**Comments** It can be seen that the structure used is very different from the one presented at the beginning of this chapter, even though it implements the same algorithm. In fact, in this case it was not possible to use the Simulink's *Matlab function*. The discrete-time logic and the consequent need to store the simulation values at different time instants $(n, n-1, n-2)$ led to the use of Simulink's *delay* blocks. In fact, this allows to delay input signals by a specified number of samples time. Thus it can be seen the presence of the four gains presented above which, if properly parameterised, can be provided as input to the *systune* optimisation function.

### 4.2.3 PIDEnhanced autotuning system validation

The next step involved verifying if this model allowed the optimiser to properly tune the gains from which the parameters to be inserted into the TXT code can be derived. In fact, the model shown in Fig. 4.16 was built.

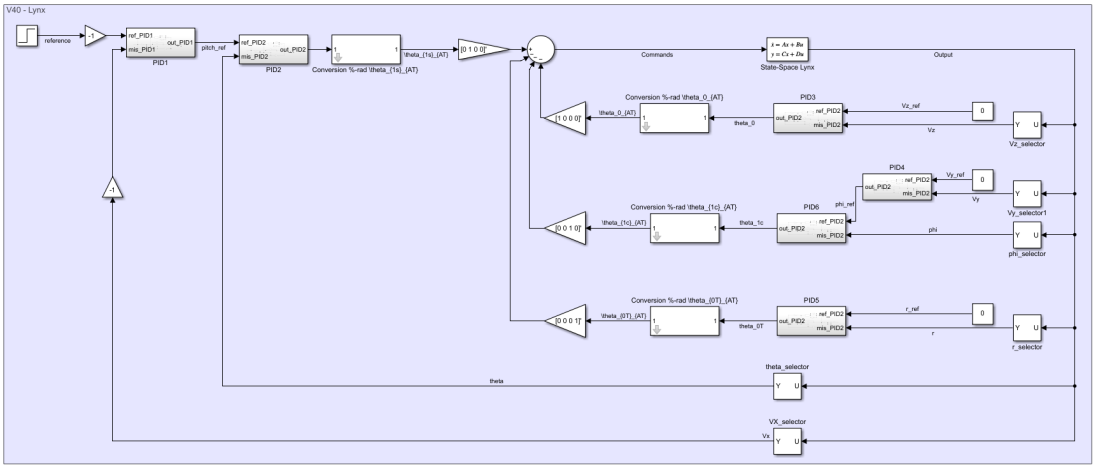


**Figure 4.16:** Autotuning model - Lynx

This model consists of six PID controllers placed on the four commands:

- a double PID to control the horizontal velocity and the pitch angle (named in Fig. 4.16 PID1 and PID2) through the longitudinal cyclic $\theta_{1S}$;
- a single PID to control the vertical speed (named PID3) through the collective $\theta_0$;
- a double loop PID to control the lateral velocity and the roll angle (PID4 and PID5) thorough the lateral cyclic $\theta_{1C}$;
- a single PID to control the roll rate (PID6) through the tail rotor collective $\theta_{0T}$.

In appendix A.4 is reported the code written to interact with this model. It is based on the already mentioned functions. What is important to note is the choice of how to define the maximum and minimum limits that have been imposed on the gain parameters on which the optimiser acts. In fact, by analysing the TXT codes, it was found that the gains that are usually assigned to MathPilots loop never exceed the limits shown in the Tab. 4.4.

| | **max** | **min** |
|---|---|---|
| $T_i$ | 0.001 | 20 |
| $T_d$ | 0.5 | 0 |
| $K_p$ | 1 | 0 |
| $\alpha$ | 1 | 0 |

**Table 4.4:** TXT gain edges

These values were introduced in the equations 4.3, 4.4, 4.5 and 4.8 to obtain the maximum and minimum blocks parameters to assign to the *systune* function. Moreover, the tuning goals for all the six PID controllers were set as *Transient*. This Matlab's *TuningGoal* constraints the transient response from specified inputs to specified outputs following a reference system described by the equation: [15]

$$RefSystem = \frac{1}{s^2 + s + 1} \tag{4.9}$$

Hence it was possible to run this model through the code shown in appendix A.4. A reference value of $\Delta u = 1\ m/s$ was imposed on the horizontal velocity and a zero reference value to the other states. A sampling time of $dt = 1/120\ s$ and a simulation time of $t = 30\ s$ were adopted. It was possible to obtain the results shown in Fig. 4.17 and 4.18.

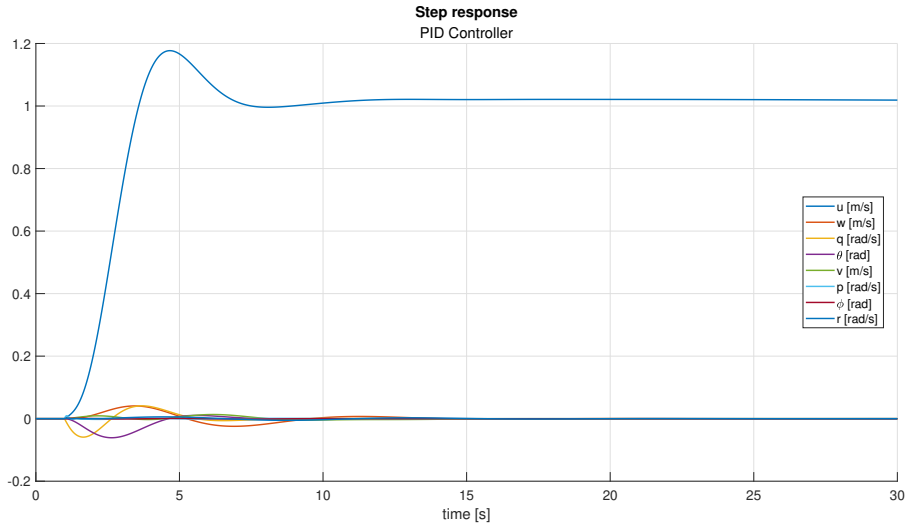


**Figure 4.17:** States step response - Lynx autotuning model

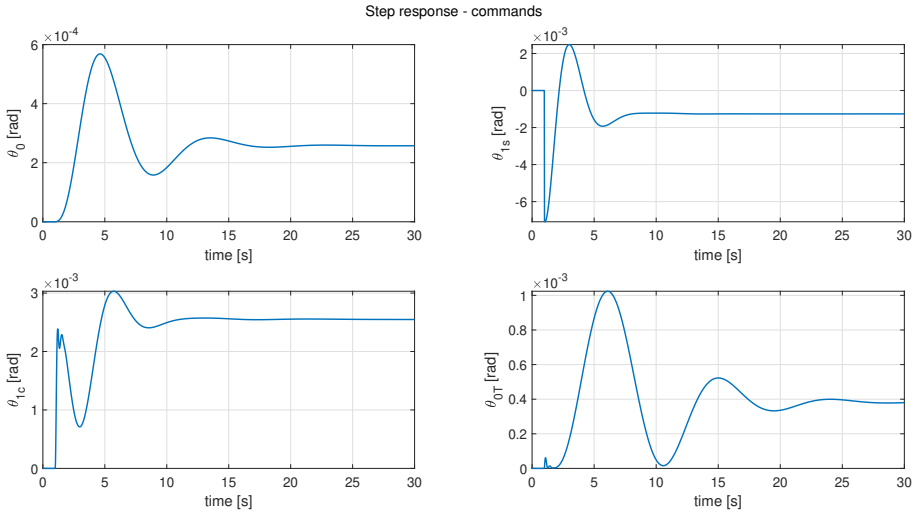


**Figure 4.18:** Commands step response - Lynx autotuning model

In particular, the gains obtained from the autotuning process are shown in Tab. 4.5.

| | $K_p$ | $T_i$ | $T_d$ | $T_f$ | $\alpha$ |
|---|---|---|---|---|---|
| **PID1** | 0.0716 | 3.5731 | 8e-4 | 9e-4 | 1.1098 |
| **PID2** | 0.4713 | 20 | 0.1409 | 0.3727 | 2.6453 |
| **PID3** | 0.0150 | 0.3028 | 0.0825 | 0.0032 | 0.0384 |
| **PID4** | 0.0479 | 2.7166 | 8e-4 | 0.076 | 91.49 |
| **PID5** | 0.117 | 0.2604 | 8e-2 | 0.5 | 6 |
| **PID6** | 5 | 0.2136 | 0.499 | 0.1744 | 0.3489 |

**Table 4.5:** Lynx autotuning model - gain obtained

**Comments** In this case it can be observed that the auto-tuning process allowed to obtain a response that satisfies the imposed requirements. The horizontal velocity tends to its reference value, while all the other states tend to cancel the initial perturbation. Moreover, the response of the horizontal speed is very fast; this reaches a steady-state error of less than 1% after 10 seconds. On the other hand, the range of the controls is extremely small, allowing adequate control of the aircraft without ever reaching limit values. The tuning process based on the Lynx linearised matrices is therefore considered validated and it was possible to proceed with the application of this model to the specific case of TXT.

### 4.2.4 TXT helicopter analysis

Having validated the functioning of the PID controllers model with the auto-tuning process, it was then possible to focus on the company's real cases. In particular, a *medium-weight twin engine helicopter* was chosen as reference. This analysis was essential to validate the quality of the work carried out. In fact, the main purpose was to calculate through the linearised model in Simulink the gains of the PID controllers of TXT to then subsequently insert these numbers in the non-linear model of the reference helicopters and compare the responses obtained. If the responses were comparable and the aircraft stabilised in both cases then this implied that the auto-tuning process was correct.

#### TXT helicopter matrices

Due to company constraints, it is not possible to show the precise matrix values for the aircraft taken as reference in TXT, but plots of the percentage differences from the Lynx matrix values are shown. As the linearised matrices also depend on the equilibrium condition of the aircraft at the time of linearisation, it is essential to report this condition in Tab. 4.6.

| | Value | Unit of measure |
|---|---|---|
| **Airspeed** | 70.2 | [kts] |
| **Ground Speed** | 82 | [kts] |
| **Pitch Angle** | 2.6 | [deg] |
| **Bank angle** | -1.6 | [deg] |
| **Heading** | 343.2 | [deg] |
| **Pitch Rate** | 0.2 | [deg/s] |
| **Roll Rate** | -0.5 | [deg/s] |
| **Yaw Rate** | 0.3 | [deg/s] |
| **Longitudinal Cyclic Pos.** | 61.1 | [%] |
| **Lateral Cyclic Pos** | 73.9 | [%] |
| **Pedals Pos.** | 59 | [%] |
| **Collective Pos.** | 38 | [%] |

**Table 4.6:** Trim conditions - TXT helicopter

The percentage differences between the linearised matrix of the Lynx at 40 kts and that of the TXT helicopter are therefore shown in Fig. 4.19 and 4.20.

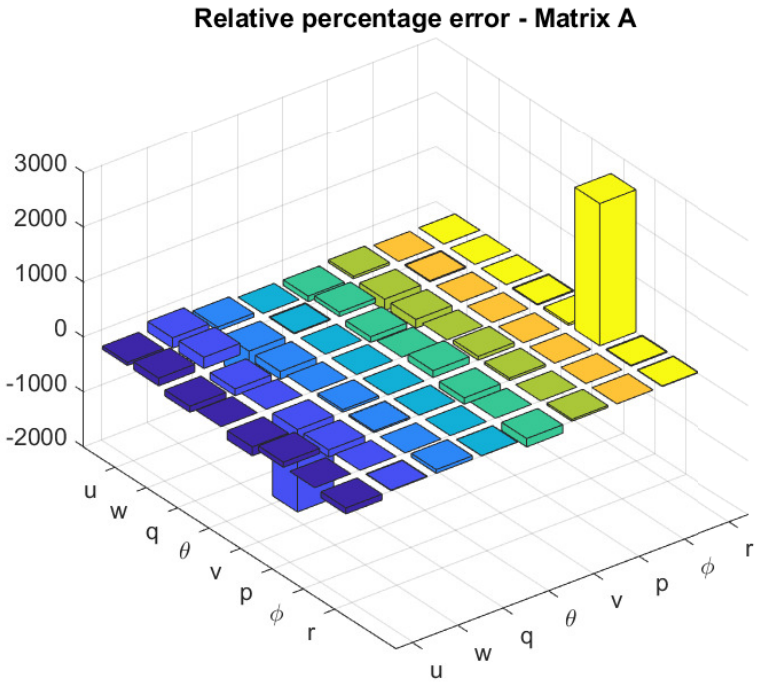


**Figure 4.19:** $\boldsymbol{A}$ relative percentage difference - TXT helicopter and Lynx

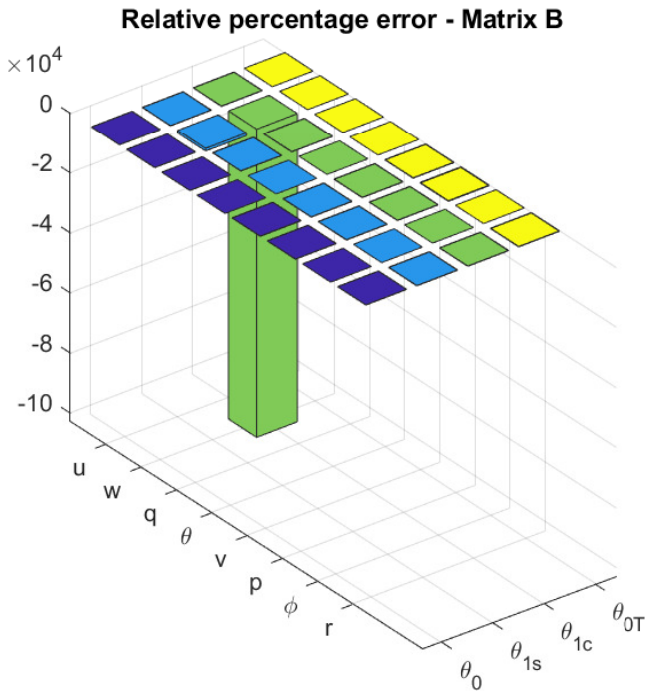


**Figure 4.20:** $\boldsymbol{B}$ relative percentage difference - TXT helicopter and Lynx

There are some values that are very different between the two helicopter models due to the fact that different types of aircraft imply dynamic characteristics that can varying.

### TXT helicopter model

Having the linearised matrices of the aircraft, it was possible to build the control loop of the aircraft using the model shown in Fig. 4.21. In this case, only the feedback loops that control the aircraft's attitude (pitch $\theta$, roll $\phi$ and yaw $\psi$) and vertical speed $w$ have been implemented. The code shown in A.5 was written to run the autotuning process of this model. In particular, a sample time of $dt = 1/60\ s$ was considered, while the tuning goals for all the four PID controllers were set as *Transient* with a reference system exploited by the equation:

$$RefSystem = \frac{1}{s^2 + s + 1} \tag{4.10}$$

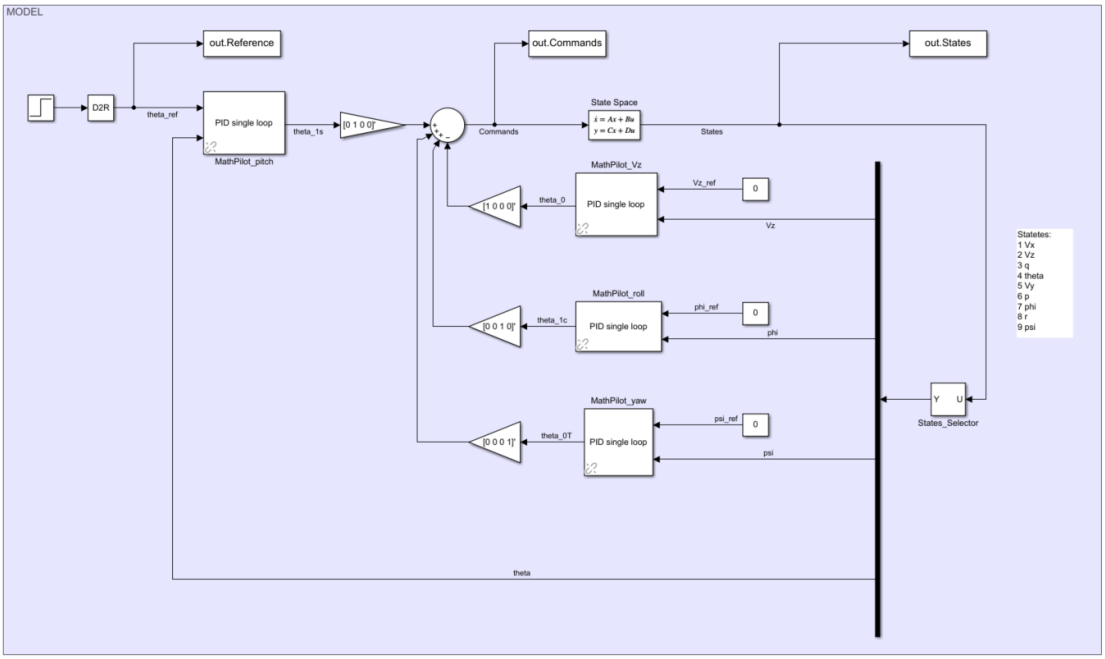


**Figure 4.21:** TXT helicopter model

Considering a perturbation step input on the pitch angle of $\Delta\theta = 1\,\text{deg}$, it was possible to obtain the results in Fig. 4.22, 4.23 and 4.24. The gains are shown in Tab. 4.7:

| | $K_p$ | $T_i$ | $T_d$ | $T_f$ | $\alpha$ |
|---|---|---|---|---|---|
| **MathPilot Pitch** | 0.8143 | 2.3455 | 0.5 | 0.002 | 0.004 |
| **MathPilot VS** | 0.01 | 0.3491 | 0.0032 | 0.1457 | 46.05 |
| **MathPilot Roll** | 0.587 | 1.8141 | 0.0017 | 0.0019 | 1.11 |
| **MathPilot Yaw** | 0.33 | 1.4 | 0.5 | 0.0169 | 0.0338 |

**Table 4.7:** TXT helicopter - gains autotuned

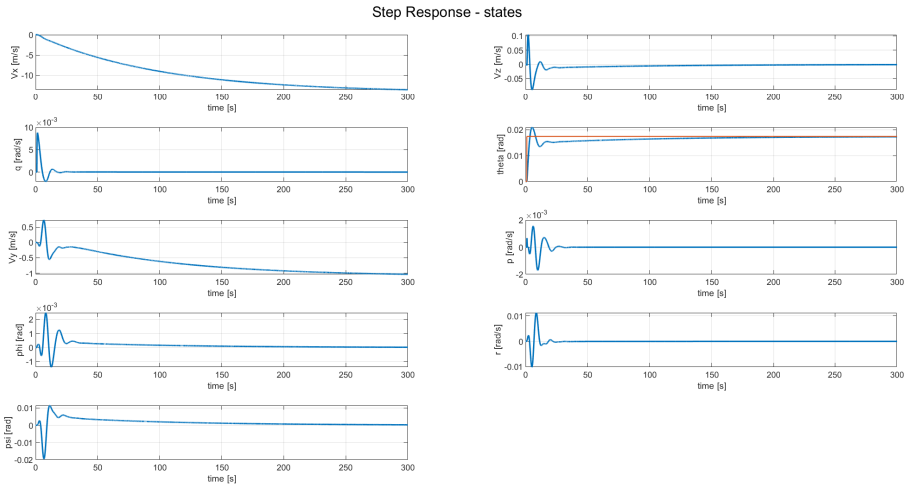


**Figure 4.22:** States step response - TXT helicopter model

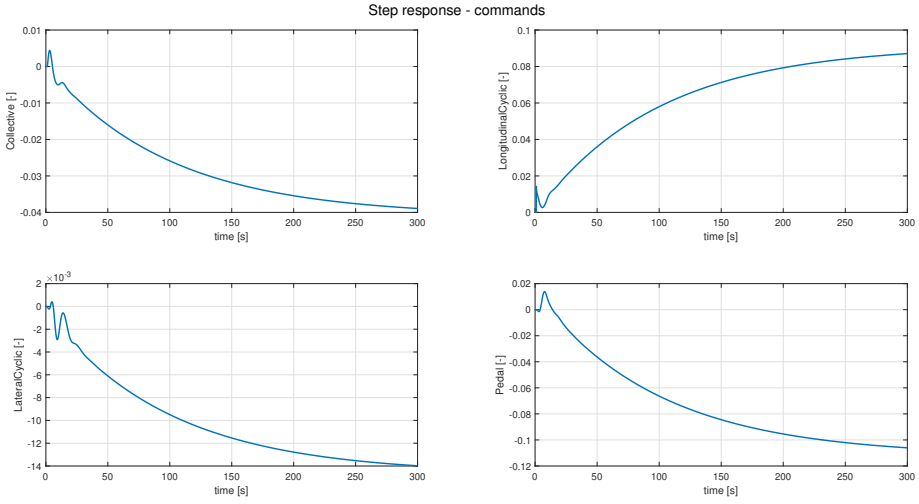


**Figure 4.23:** Commands step response - TXT helicopter model

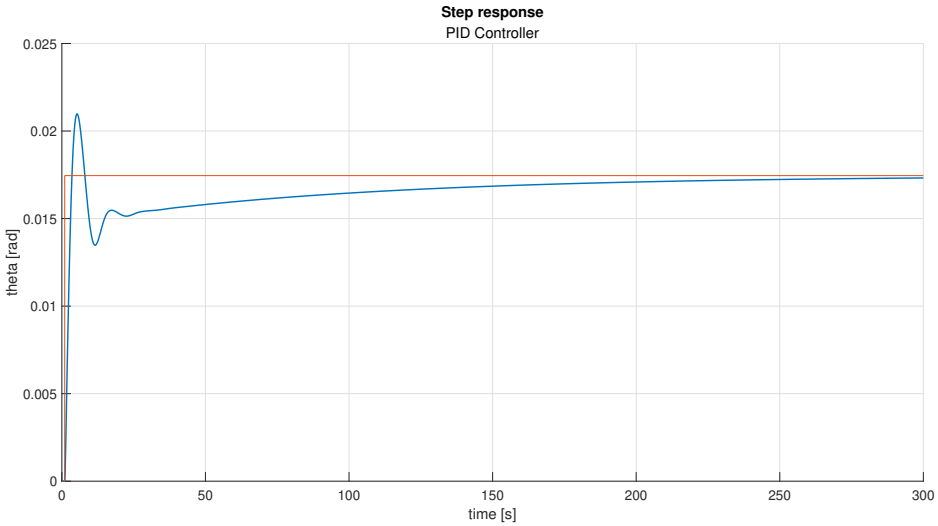


**Figure 4.24:** $u$ step response - TXT helicopter model

In this case, linearised matrices relating to a real case of TXT were used. It can be seen that the system is controlled and stabilised. The response is relatively slow, as shown in Fig. 4.24, but the difference between the reference and measured values are less than 10% after 20 seconds of the simulation. Moreover, the commands all tend towards an equilibrium condition and their change from the initial value is always very small. This characteristic made it possible to validate the correct functioning of the PID controllers and their auto-tuning process. The final step was certainly to put the gains shown in Tab. 4.7 on the TXT nonlinear model and check if these gains also allowed the nonlinear model to stabilise the aircraft around this equilibrium point.

**Comments** One of the main goals of this thesis work was to reduce the man-time normally spent on PID controllers tuning. In fact, nowadays, the minimum time required to test the PID gains corresponds to the duration of a QTG test, thus 3/5 minutes. This is a typical situation when the helicopter model is known and used. However, changing the aircraft model these tuning periods can reach time intervals of hours and days. Thanks to the linearized model implementation on *Matlab-Simulink*, this time is considerably reduced running the autotuning simulations just described. In fact, the time measured for running the optimizator algorithm obtaining the results shown in the previous figures corresponds to $T_{sim} = 50.238\ s$. This time differences highlight the importance of considering autotuning processes in control system modelling. Furthermore, it is important to notice that the man-time is not neglected, but it is decreased thanks to the use of optimization algorithms. In fact, the two drawbacks of this model are related to the necessity of finding the best tuning goals to fed the optimizer and the availability of the linearized model of the helicopter in order to obtain the matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ and $\boldsymbol{D}$. Further considerations on the benefits and drawbacks of the thesis work are addressed in the last chapter.

# Chapter 5

# Verification process

To validate the gains obtained with the linear model, it was decided to use a special QTG test of the helicopter simulator in question. In fact, the main aim was not to reproduce a QTG test, but to simulate with the non-linear flight model the same test carried out in the previous chapter with the linear model. However, the MathPilot loops were included within the operating logic of the QTG tests, thus making it essential to start a simulator test in order to verify the functioning of the gains obtained with the autotuning process. The following procedure was then followed:

- a specific QTG test was chosen. In particular, a test where the initial conditions were the same of the matrices introduced in the previous chapter;
- the matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ and $\boldsymbol{D}$ were obtained in trim condition (these were the same matrices of the previous chapter);
- the gains obtained in Tab. 4.7 were inserted in the PID controllers of TXT code;
- a pitch perturbation of $\Delta\theta = 1\,\text{deg}$ was introduced into the code, while the reference value of the other states has been left unchanged;
- the simulation was launched.

TXT's simulation software allowed to obtain an output file to assess the evolution of the aircraft's states during the flight simulation. Since this software was born from the need to certify the simulator through special tests in which the virtual response of the aircraft is compared with the actual flight data of the same, these flight data were usually reported on the output plot of these simulation. Therefore, all subsequent graphs will show a blue line (relating to the flight data) which should be neglected, and a green line representing the actual response of the non-linear flight model to the test performed. This was because TXT's code configuration requirements made it necessary to rely on the operating logic of a QTG test, even though a certification test was not carried out. Therefore, the following results were obtained.

### Simulation results - States

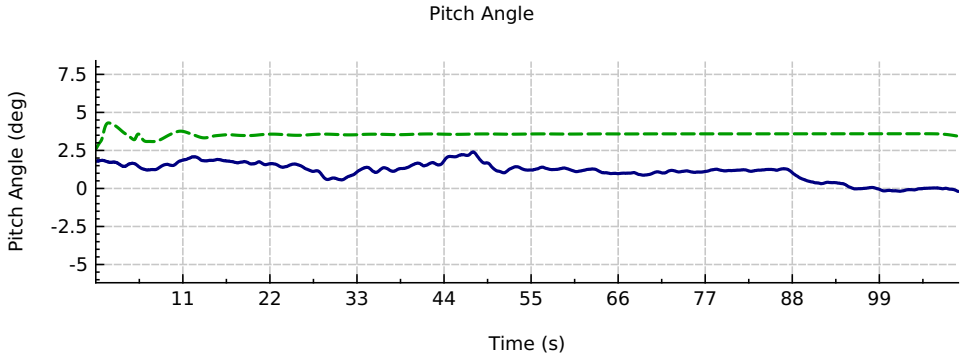


**Figure 5.1:** Pitch Angle

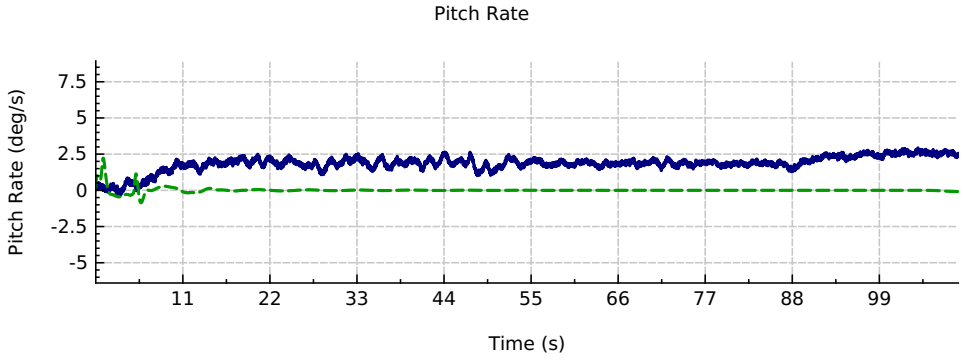


**Figure 5.2:** Pitch Angle Rate

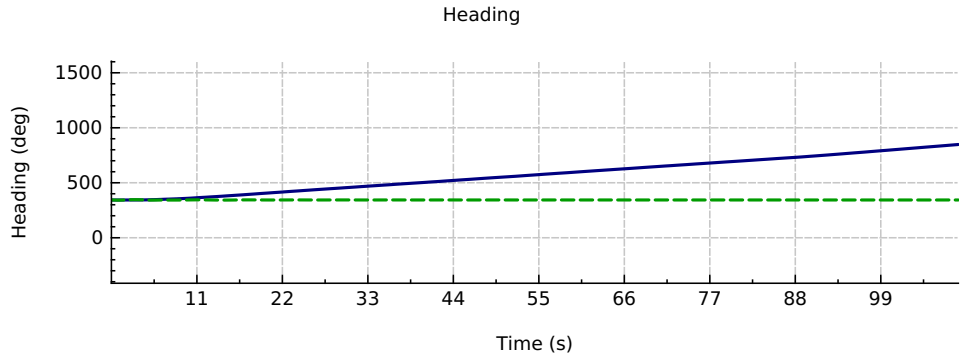


**Figure 5.3:** Yaw Angle

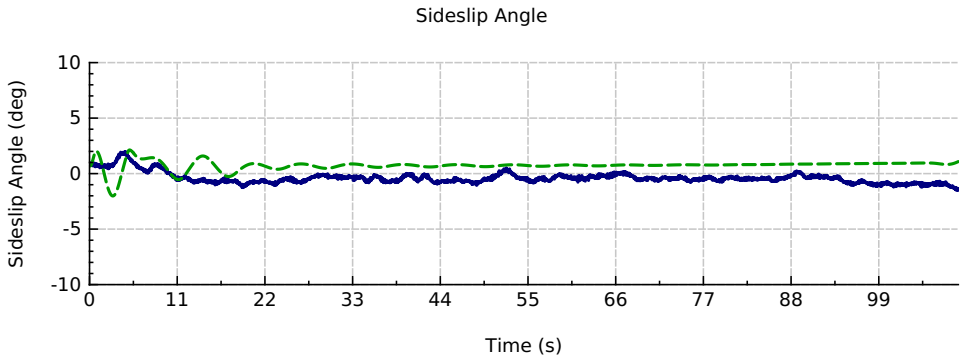


**Figure 5.4:** Angle of Sideslip

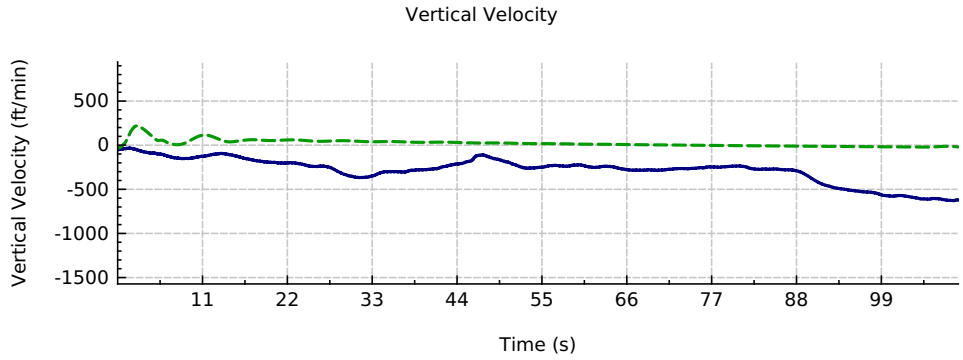


**Figure 5.5:** Vertical Speed

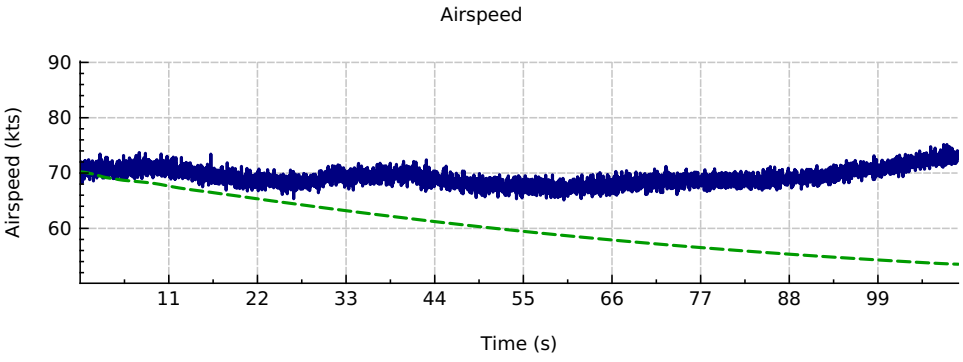


**Figure 5.6:** Indicated Airspeed

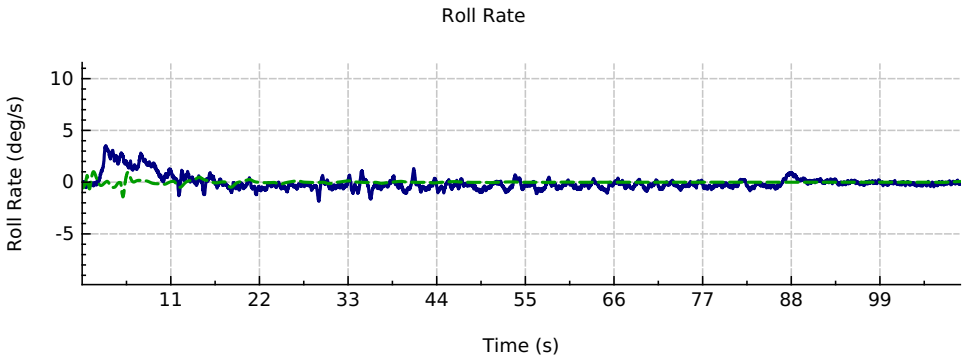


**Figure 5.7:** Roll Angle Rate

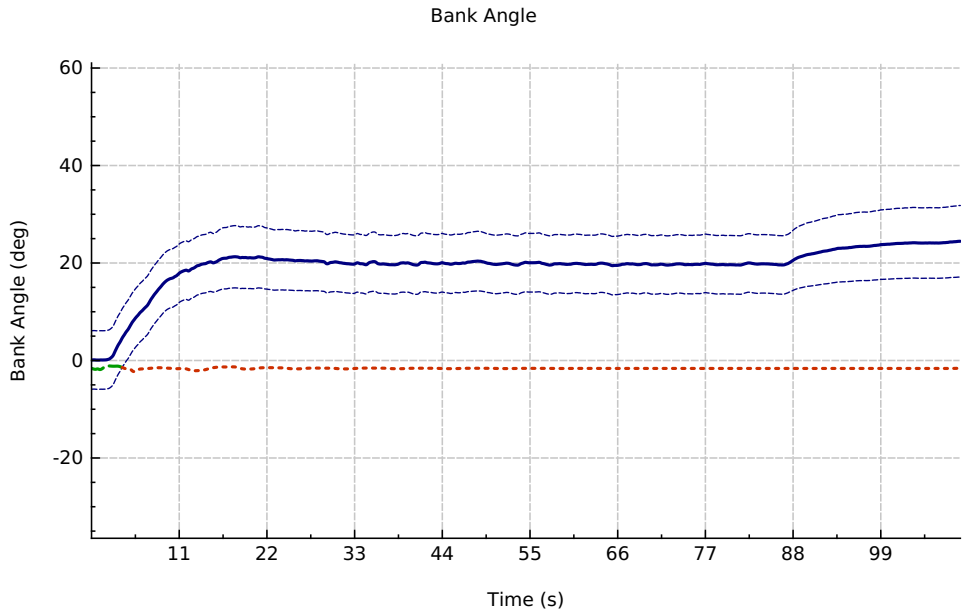


**Figure 5.8:** Roll Angle

These results should be compared with those shown in Fig. 4.22. It can be seen that in general all states tend towards a boundary value and no divergence is found. In all the plots shown in this process, the states and commands values are given in absolute terms. In fact, in the previous linearised treatment, the perturbation values $\Delta$ with respect to their initial condition were always considered. Therefore, in this case it is assumed stable a condition in which states tends to return to theirs initial value or to the input reference. In particular, it can be observed that the pitch angle in Fig. 5.1 tends to the value $\theta = 3.6$ deg that is equal the sum between the pitch initial values $\theta = 2.6$ deg (observed in Tab. 4.6) and the $\Delta\theta = 1$ deg given by the test. The other states, such

as the roll angle $\phi$, the roll angle rate $p$ and the vertical speed $w$ tend to their initial value after a transient period. Fig. 5.6 shows a decreasing trend for IAS. This is consitent with Fig. 5.1, where commanded pitch angle is higer than real aircraft, thus forcing the simulated flight model to slow down. Moreover, these trends are the same of the linearized output obtained in Fig. 4.22. Obviously, there are differences in terms of transients as the linearised model does not accurately capture all the dynamic characteristics of the non linear system. However, the trends obtained from linear and non-linear analysis are the same as well as the orders of magnitude of the transients. It is then possible to observe the commands output

### Simulation results - Commands

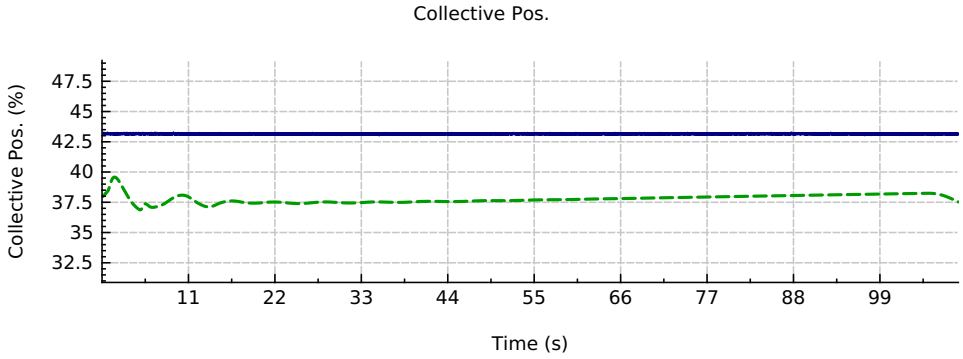


**Figure 5.9:** Control Position Collective

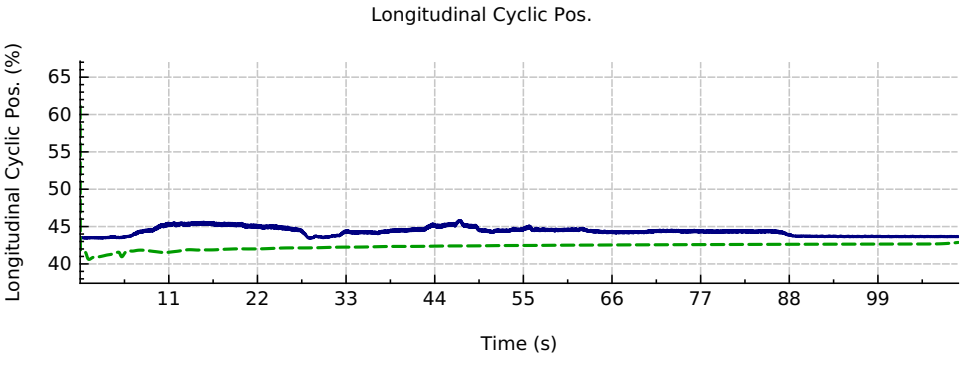


**Figure 5.10:** Control Position Pitch

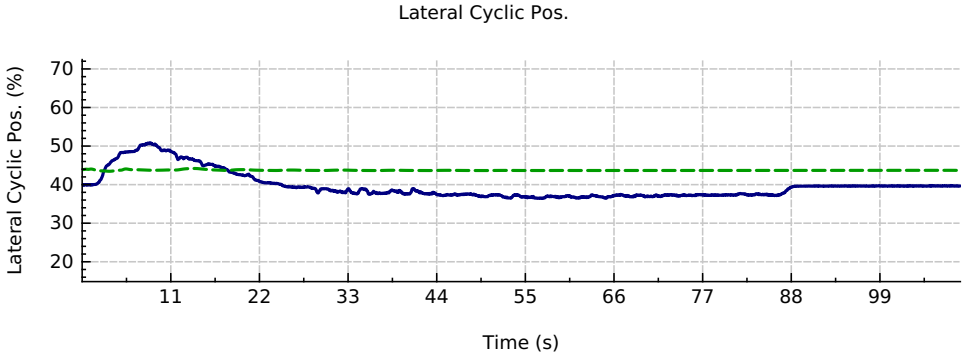


**Figure 5.11:** Control Position Roll

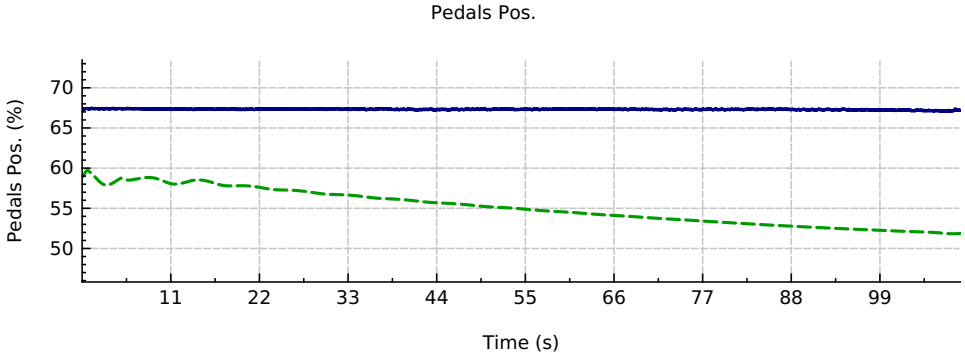


**Figure 5.12:** Control Position Roll

Also considering the commands output, it is observed that the trends obtained from the non-linear model are consistent with those found in the previous chapter. In fact, the perturbation of the lateral cyclic is approximately zero both in Fig. 5.11 and Fig. 4.23. Other commands (pedal, longitudinal cyclic and collective) show a trend that is similar on what was obtained with the linear model. It should be noted that this analysis was carried out knowing that in this case the absolute values of the commands were reported, whereas in the linearised case what was plotted were their perturbations. It could be seen that the adoption of gains obtained with the linearised model through the explained autotuning process led to the PID controllers tuning in such a way that the helicopter was adequately stabilized. Moreover, both the trend and the values obtained in the two cases are comparable.

# Chapter 6

# Conclusions

The main goal that was provided by the TXT company at the beginning of this thesis was to develop an automatic control system, to be used during the validation tests of a helicopter flight simulator, which is able to faithfully replicate the behaviour of the pilot and which is able to auto-tune. The principal aim of the company was related to reduce time required for the definition and execution of certification tests, lowering the man-time currently required by the process and consequently reducing costs. The expected results could in fact be summarised as follows:

1. the structure of automatic control systems in use should be studied;
2. it is required to design control systems that can increase the efficiency of the testing process, based on available literature. They can be developments of current models or ex-novo redesigns of them;
3. the solutions studied must be implemented within the TXT simulation environment, under real-time simulation constraints;
4. possible auto-regulation processes of the designed control systems should be defined;
5. it is required to evaluate the performance of the developed controller and identify improvement areas.

Analyzing the path followed during the thesis development it can be seen that the starting phase of the project was dedicated to a deep understanding on the problem proposed. In fact, a considerable amount of time was spent studying the code of TXT and then the current state of the art in PID control techniques, as shown in chapter 2 according to the first goal just mentioned. Since the requirement imposed by the company was to develop a model that could be implemented in TXT's operating logic and that could increase the efficiency of the simulator's certification processes, it was considered more efficient to rely on algorithms already on the market and to understand how to adapt them to the specific case of this thesis. Designing an auto-tuning algorithm from scratch would have required a much longer development period and carried a greater risk of unsuitability for TXT code due to its complexity. In fact, knowing that the company adopts *Matlab*

- *Simulink* to develop its simulators, the decision to study the auto-tuning algorithms already implemented in this software was a consequence. This choice allowed to respect the second and third requirements just mentioned.
In fact, thanks to the Matlab and Simulink toolboxes, it was possible to obtain a MathPilot model that can auto-tune and autonomously calculate the PID controller gains according to the type of aircraft and references that are imposed from outside, as shown in chapter 4. However, the biggest drawback of this solution is surely the fact that it only works with a linearised helicopter model. In fact, this solution implies the need for linearised aircraft matrices in a specific flight condition. In the TXT situation, thanks to the availability of a private algorithm for obtaining matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, this constraint has been overcome and the solution proposed with *Matlab* could be considered viable.
The main challenge of this project was definitely to validate at the end of the thesis the gains obtained with Matlab on the non-linear model currently implemented in the company's simulator. As shown in chapter 5, by inserting this gains obtained with the linearised model within a QTG test of the TXT simulator, it was possible to replicate the same test run on *Matlab*. This allowed to compare the two responses and to conclude that both linear and non-linear models stabilize the helicopter. In addition, the trend followed by states and commands is essentially the same in the two simulations. The fact that same trends were obtained was considered by TXT as a satisfactory evidence to demonstrate the validity and operation of the MathPilots' linearised model. This satisfies the last requirements previously mentioned.

This thesis work allowed the company to have a model tailored on their needs that can be used before the certification tests in order to obtain the optimized gains to insert into the PID controllers. This is provided by an autotuning process developed in *Matlab* that can reduce the man-time required. Moreover, it is not only the amount of hours spent in the tuning process that is reduced, but also the experience of people carrying out this work: these algorithms are easier to understand.

To sum up, thanks to this work *TXT e-Solutions* has a tool to reduce its costs in two ways:

- decreasing the man-time hours spent in tuning;
- allowing a less experienced person to deal with tuning.

**Future developments** The main area for future developments is related to the optimisation of the best goals to set to the *systune* function. Several different types of *TuningGoals* were shown during the project, which were tested with a trial and error method. However, the link between these targets and aircraft flight conditions needs to be investigated to further reduce the time required for certification tests. Moreover, it is necessary to test the *Matlab-Simulink* algorithm with more different kinds of helicopter in order to improve the model and try to make this model as flexible and versatile as possible. Finally, an improvement in the code and modelling of the system would also be profitable.

# Reference


[1] M. Masson. *Use and Benefits of Simulators*. EASA Network, 2021 (cit. on p. 1).

[2] H. Bansal and S. Ponpathirkoottam. «PID Controller Tuning Techniques: A Review». In: *Journal of Control Engineering and Technology (JCET)* 2 (Oct. 2012), pp. 168–176 (cit. on p. 10).

[3] J. G. Ziegler and N. B Nichols. «Optimum Settings for Automatic Controller». In: *Transaction of ASME* 64 (1942), pp. 759–768 (cit. on p. 11).

[4] Attyar S. M. and Abdullah I. A. «PID Controller Design and Simulation for Aircraft Roll Control Based on Evolutionary Technique Using MATLAB». In: *International Journal of Engineering and Innovative Technology (IJEIT)* 8 (2018) (cit. on p. 13).

[5] Dolezel P. and Mares J. «Self-tuning PID Control using Genetic Algorithm and Artificial Neural Networks». In: *ASR 2009 Instruments and Control,* (2009), pp. 33–39 (cit. on p. 13).

[6] Zulfatman and M. F. Rahmat. «Application of self-tuning Fuzzy PID controller on industrial hydraulic actuator using system identification approach». In: *International Journal on Smart Sensing and Intelligent Systems* 2 (2009), pp. 246–261 (cit. on p. 14).

[7] Gareth D. Padfield. *HELICOPTER FLIGHT DYNAMICS*. Blackwell Publishing, 2007 (cit. on pp. 16, 19, 20).

[8] MathWorks. *Control System Toolbox*. https://it.mathworks.com/help/control/, 2021 (cit. on p. 29).

[9] MathWorks. *Simulink Control Design*. https://it.mathworks.com/help/slcontrol/, 2021 (cit. on p. 29).

[10] MathWorks. *slTuner*. https://it.mathworks.com/help/slcontrol/ug/sltuner.html, 2021 (cit. on p. 29).

[11] MathWorks. *systune*. https://it.mathworks.com/help/slcontrol/ug/sltuner.systune.html, 2021 (cit. on p. 29).

[12] MathWorks. *Building Tunable Models*. https://it.mathworks.com/help/control/ug/building-tunable-models.html, 2021 (cit. on p. 29).

[13] MathWorks. *TuningGoal.StepTracking class*. https://it.mathworks.com/help/control/ref/tuninggoal.steptracking-class.html, 2021 (cit. on p. 31).

[14] MathWorks. *TuningGoal.Margins class*. https://it.mathworks.com/help/control/ref/tuninggoal.margins-class.html, 2021 (cit. on p. 31).

[15] MathWorks. *TuningGoal.Transient class*. https://it.mathworks.com/help/control/ref/|tuninggoal.transient-class.html, 2021 (cit. on p. 36).

# Appendix A

# Matlab Codes

## A.1 PIDEnhanced first version - Matlab function

```
function [u_n, delta_u_n_out] = fcn(alpha, err_der, Ti, Kp, err_prop,
Ts, err, Td, u_max, u_min)

persistent  CI

% CI = {'err_prop_1','err_der_f_1','err_der_f_2','u_n_1'};

if isempty(CI)
    CI = [0, 0, 0, 0];
end

if(Td > 0.0)

        Tf = alpha * Td;

        err_der_f = CI(2) / (Ts / Tf + 1) + err_der * (Ts / Tf) / (Ts
/ Tf + 1);
else
        err_der_f = err_der;
end

if(Ti > 0.0)
    delta_u_n = Kp * ((err_prop - CI(1)) + ((Ts / Ti) * err)  + ((Td
/ Ts) * (err_der_f - 2 * CI(2) + CI(3))));

else
    delta_u_n = Kp * ((err_prop - CI(1)) + ((Td / Ts) * (err_der_f -
2 * CI(2) + CI(3))));
end

if(delta_u_n > (u_max - CI(4)))
    delta_u_n = u_max - CI(4);
```

```
29
30      elseif(delta_u_n < (u_min - CI(4)))
31          delta_u_n = u_min - CI(4);
32      end
33
34      u_n = CI(4)+ delta_u_n;
35
36       CI(1)  = err_prop;
37       CI(3) = CI(2);
38       CI(2) = err_der_f;
39       CI(4) = u_n;
40
41       Res = CI;
42
43       delta_u_n_out = delta_u_n;
44
45      return
```

## A.2 Commands Limits

```
% Longitudina cyclic
Pitch_min = deg2rad(-11);
Pitch_max = deg2rad(13);
Pitch_neutral = (Pitch_max+Pitch_min)/2;
% Lateral cyclic
Roll_min = deg2rad(-6.5);
Roll_max = deg2rad(6.5);
Roll_neutral = (Roll_max+Roll_min)/2;
% Collective tail rotor
Yaw_min = deg2rad(-8);
Yaw_max = deg2rad(24);
Yaw_neutral = (Yaw_max+Yaw_min)/2;
% Collective
Coll_min = deg2rad(0.45);
Coll_max = deg2rad(14.45);
```

## A.3 Autotuning code - HcCAs with PID Controller Simulink's blocks

```
    %% PID Controller - autotuning
mdl = 'HcCAS_controller';
open_system('HcCAS_controller.slx');

st0 = slTuner(mdl,{'G1','G2','G3','G4','G5'});
```

```
addPoint(st0,{'Ref_{AT}','Ref_{\phi}_{AT}','Ref_{w}_{AT}','Ref_{v}_{AT}',
    'Ref_{pitch}_{AT}'});
addPoint(st0,{'u_{AT}','\phi_{AT}','w_{AT}','v_{AT}','\theta_{AT}'});
addPoint(st0,{'Comandi_AT','Stati_AT'});


% Setting auto-tuning requirements and objectives
G1 = getBlockParam(st0,'G1');
G2 = getBlockParam(st0,'G2');
G3 = getBlockParam(st0,'G3');
G4 = getBlockParam(st0,'G4');
G5 = getBlockParam(st0,'G5');

G1.Kp.Minimum = 0;
G1.Kp.Maximum = 5;
G2.Kp.Minimum = 0;
G2.Kp.Maximum = 5;
G3.Kp.Minimum = 0;
G3.Kp.Maximum = 5;
G4.Kp.Minimum = 0;
G4.Kp.Maximum = 5;
G5.Kp.Minimum = 0;
G5.Kp.Maximum = 5;

G1.Ki.Minimum = 0;
G1.Ki.Maximum = 10;
G2.Ki.Minimum = 0;
G2.Ki.Maximum = 10;
G3.Ki.Minimum = 0;
G3.Ki.Maximum = 10;
G4.Ki.Minimum = 0;
G4.Ki.Maximum = 10;
G5.Ki.Minimum = 0;
G5.Ki.Maximum = 10;

G1.Kd.Minimum = 0;
G1.Kd.Maximum = 2;
G2.Kd.Minimum = 0;
G2.Kd.Maximum = 2;
G3.Kd.Minimum = 0;
G3.Kd.Maximum = 2;
G4.Kd.Minimum = 0;
G4.Kd.Maximum = 2;
G5.Kd.Minimum = 0;
G5.Kd.Maximum = 5;

G1.Tf.Minimum = dt;
% G1.Tf.Maximum = 5;
G2.Tf.Minimum = dt;
% G2.Tf.Maximum = 5;
G3.Tf.Minimum = dt;
% G3.Tf.Maximum = 5;
G4.Tf.Minimum = dt;
```

```
% G4.Tf.Maximum = 5;
G5.Tf.Minimum = dt;
% G5.Tf.Maximum = 5;

setBlockParam(st0,'G1',G1);
setBlockParam(st0,'G2',G2);
setBlockParam(st0,'G3',G3);
setBlockParam(st0,'G4',G4);
setBlockParam(st0,'G5',G5);

% Less than 20% mismatch with reference model 1/(s+1)
TrackReq = TuningGoal.StepTracking({'Ref_{AT}','Ref_{\phi}_{AT}','Ref_{w}
    _{AT}','Ref_{v}_{AT}'},{'u_{AT}','\phi_{AT}','w_{AT}','v_{AT}'},1);
TrackReq.RelGap = 0.2;

%% Gain and phase margins at plant inputs and outputs
MarginReq1 = TuningGoal.Margins('Comandi_AT',5,40);
MarginReq2 = TuningGoal.Margins('Stati_AT',5,40);

%% Limit on fast dynamics
MaxFrequency = 25;
PoleReq = TuningGoal.Poles(0,0,MaxFrequency);

AllReqs = [TrackReq,MarginReq1,MarginReq2,PoleReq];

% Processing - Autotuning e PID gain rewriting
opt = systuneOptions('RandomStart',30);
rng(0);
[st,fSoft,~,info] = systune(st0,AllReqs,opt);
showTunable(st)
```

## A.4 PIDEnhanced autotuning code

```
%% Simulation Data e input of linearised model matrices
dt = 1/120;
run('Input\Lynx_V40.m')
run('Valori_max_min_comandi.m')

%% Simulink MathPilot model with PID Controllers
% PID1
Ti1 = 2;
Kp1 = 0.3 ;
Td1 = 0.15;
Ts = dt;
alpha1 = 0.2;
beta1 = 1;
gamma1 = 0;
Tf1 = Td1*alpha1;

```

```
%PID2
Ti2 = 1.5;
Kp2 = 0.3;
Td2 = 0.2;
alpha2 = 0.2;
beta2 = 1;
gamma2 = 0;
Tf2 = Td2*alpha2;

% Maximum and minimum gain values
Gain1max = 1/(Ts/(1*0.5) + 1);
      % 1/(Ts/Tf+1)
Gain1min = 0;
Gain2max = 0.5/Ts;
      % Td/Ts
Gain2min = 0;
Gain3max = 5;
      % Kp
Gain3min = 0.01;
Gain4max = Ts/0.01;
      % Ts/Ti
Gain4min = Ts/20;

[Gain1,Gain2,Gain3,Gain4] = calcolo_gain(Kp1,Ti1,Td1,alpha1,Ts);
[Gain5,Gain6,Gain7,Gain8] = calcolo_gain(Kp2,Ti2,Td2,alpha2,Ts);
[Gain9,Gain10,Gain11,Gain12] = calcolo_gain(Kp1,Ti1,Td1,alpha1,Ts);
[Gain13,Gain14,Gain15,Gain16] = calcolo_gain(Kp2,Ti2,Td2,alpha2,Ts);
[Gain17,Gain18,Gain19,Gain20] = calcolo_gain(Kp2,Ti2,Td2,alpha2,Ts);
[Gain21,Gain22,Gain23,Gain24] = calcolo_gain(Kp2,Ti2,Td2,alpha2,Ts);

mdl = 'Lynx_prove_qtg_doppioloop';
open_system('Lynx_prove_qtg_doppioloop');

options = slTunerOptions('RateConversionMethod','prewarp','PreWarpFreq'
    ,10);
st0 = slTuner(mdl,{'Lynx_prove_qtg_doppioloop/PID1/Filtro1/GainTunable1',
    'Lynx_prove_qtg_doppioloop/PID1/GainTunable2','
    Lynx_prove_qtg_doppioloop/PID1/GainTunable3','
    Lynx_prove_qtg_doppioloop/PID1/GainTunable4',...
     'Lynx_prove_qtg_doppioloop/PID2/Filtro2/GainTunable5','
    Lynx_prove_qtg_doppioloop/PID2/GainTunable6','
    Lynx_prove_qtg_doppioloop/PID2/GainTunable7','
    Lynx_prove_qtg_doppioloop/PID2/GainTunable8',...
     'Lynx_prove_qtg_doppioloop/PID3/Filtro3/GainTunable9','
    Lynx_prove_qtg_doppioloop/PID3/GainTunable10','
    Lynx_prove_qtg_doppioloop/PID3/GainTunable11','
    Lynx_prove_qtg_doppioloop/PID3/GainTunable12',...
     'Lynx_prove_qtg_doppioloop/PID4/Filtro4/GainTunable13','
    Lynx_prove_qtg_doppioloop/PID4/GainTunable14','
    Lynx_prove_qtg_doppioloop/PID4/GainTunable15','
    Lynx_prove_qtg_doppioloop/PID4/GainTunable16',...
```

```
51      'Lynx_prove_qtg_doppioloop/PID5/Filtro5/GainTunable17','
        Lynx_prove_qtg_doppioloop/PID5/GainTunable18','
        Lynx_prove_qtg_doppioloop/PID5/GainTunable19','
        Lynx_prove_qtg_doppioloop/PID5/GainTunable20',...
52      'Lynx_prove_qtg_doppioloop/PID6/Filtro6/GainTunable21','
        Lynx_prove_qtg_doppioloop/PID6/GainTunable22','
        Lynx_prove_qtg_doppioloop/PID6/GainTunable23','
        Lynx_prove_qtg_doppioloop/PID6/GainTunable24'},options);
53  addPoint(st0,{'reference','pitch_ref','Vz_ref','Vy_ref','r_ref','phi_ref'
        });
54  addPoint(st0,{'Vx','theta','Vz','Vy','r','phi'});
55  addPoint(st0,{'Comandi','Output'});
56
57  Gain1_t = getBlockParam(st0,'GainTunable1');
58  Gain2_t = getBlockParam(st0,'GainTunable2');
59  Gain3_t = getBlockParam(st0,'GainTunable3');
60  Gain4_t = getBlockParam(st0,'GainTunable4');
61  Gain5_t = getBlockParam(st0,'GainTunable5');
62  Gain6_t = getBlockParam(st0,'GainTunable6');
63  Gain7_t = getBlockParam(st0,'GainTunable7');
64  Gain8_t = getBlockParam(st0,'GainTunable8');
65  Gain9_t = getBlockParam(st0,'GainTunable9');
66  Gain10_t = getBlockParam(st0,'GainTunable10');
67  Gain11_t = getBlockParam(st0,'GainTunable11');
68  Gain12_t = getBlockParam(st0,'GainTunable12');
69  Gain13_t = getBlockParam(st0,'GainTunable13');
70  Gain14_t = getBlockParam(st0,'GainTunable14');
71  Gain15_t = getBlockParam(st0,'GainTunable15');
72  Gain16_t = getBlockParam(st0,'GainTunable16');
73  Gain17_t = getBlockParam(st0,'GainTunable17');
74  Gain18_t = getBlockParam(st0,'GainTunable18');
75  Gain19_t = getBlockParam(st0,'GainTunable19');
76  Gain20_t = getBlockParam(st0,'GainTunable20');
77  Gain21_t = getBlockParam(st0,'GainTunable21');
78  Gain22_t = getBlockParam(st0,'GainTunable22');
79  Gain23_t = getBlockParam(st0,'GainTunable23');
80  Gain24_t = getBlockParam(st0,'GainTunable24');
81
82  Gain1_t.Gain.Maximum = Gain1max;
83  Gain1_t.Gain.Minimum = Gain1min;
84  Gain5_t.Gain.Maximum = Gain1max;
85  Gain5_t.Gain.Minimum = Gain1min;
86  Gain9_t.Gain.Maximum = Gain1max;
87  Gain9_t.Gain.Minimum = Gain1min;
88  Gain13_t.Gain.Maximum = Gain1max;
89  Gain13_t.Gain.Minimum = Gain1min;
90  Gain17_t.Gain.Maximum = Gain1max;
91  Gain17_t.Gain.Minimum = Gain1min;
92  Gain21_t.Gain.Maximum = Gain1max;
93  Gain21_t.Gain.Minimum = Gain1min;
94
95  Gain2_t.Gain.Maximum = Gain2max;
96  Gain2_t.Gain.Minimum = Gain2min;
```

```
97  Gain6_t.Gain.Maximum = Gain2max;
98  Gain6_t.Gain.Minimum = Gain2min;
99  Gain10_t.Gain.Maximum = Gain2max;
100 Gain10_t.Gain.Minimum =  Gain2min;
101 Gain14_t.Gain.Maximum = Gain2max;
102 Gain14_t.Gain.Minimum =  Gain2min;
103 Gain18_t.Gain.Maximum = Gain2max;
104 Gain18_t.Gain.Minimum =  Gain2min;
105 Gain22_t.Gain.Maximum = Gain2max;
106 Gain22_t.Gain.Minimum =  Gain2min;
107
108 Gain3_t.Gain.Maximum = Gain3max;
109 Gain3_t.Gain.Minimum = Gain3min;
110 Gain7_t.Gain.Maximum = Gain3max;
111 Gain7_t.Gain.Minimum = Gain3min;
112 Gain11_t.Gain.Maximum = Gain3max;
113 Gain11_t.Gain.Minimum = Gain3min;
114 Gain15_t.Gain.Maximum = Gain3max;
115 Gain15_t.Gain.Minimum = Gain3min;
116 Gain19_t.Gain.Maximum = Gain3max;
117 Gain19_t.Gain.Minimum = Gain3min;
118 Gain23_t.Gain.Maximum = Gain3max;
119 Gain23_t.Gain.Minimum = Gain3min;
120
121 Gain4_t.Gain.Maximum = Gain4max;
122 Gain4_t.Gain.Minimum = Gain4min;
123 Gain8_t.Gain.Maximum = Gain4max;
124 Gain8_t.Gain.Minimum = Gain4min;
125 Gain12_t.Gain.Maximum = Gain4max;
126 Gain12_t.Gain.Minimum = Gain4min;
127 Gain16_t.Gain.Maximum = Gain4max;
128 Gain16_t.Gain.Minimum = Gain4min;
129 Gain20_t.Gain.Maximum = Gain4max;
130 Gain20_t.Gain.Minimum = Gain4min;
131 Gain24_t.Gain.Maximum = Gain4max;
132 Gain24_t.Gain.Minimum = Gain4min;
133
134 setBlockParam(st0,'GainTunable1',Gain1_t);
135 setBlockParam(st0,'GainTunable2',Gain2_t);
136 setBlockParam(st0,'GainTunable3',Gain3_t);
137 setBlockParam(st0,'GainTunable4',Gain4_t);
138 setBlockParam(st0,'GainTunable5',Gain5_t);
139 setBlockParam(st0,'GainTunable6',Gain6_t);
140 setBlockParam(st0,'GainTunable7',Gain7_t);
141 setBlockParam(st0,'GainTunable8',Gain8_t);
142 setBlockParam(st0,'GainTunable9',Gain9_t);
143 setBlockParam(st0,'GainTunable10',Gain10_t);
144 setBlockParam(st0,'GainTunable11',Gain11_t);
145 setBlockParam(st0,'GainTunable12',Gain12_t);
146 setBlockParam(st0,'GainTunable13',Gain13_t);
147 setBlockParam(st0,'GainTunable14',Gain14_t);
148 setBlockParam(st0,'GainTunable15',Gain15_t);
149 setBlockParam(st0,'GainTunable16',Gain16_t);
```

```
150 setBlockParam(st0,'GainTunable17',Gain17_t);
151 setBlockParam(st0,'GainTunable18',Gain18_t);
152 setBlockParam(st0,'GainTunable19',Gain19_t);
153 setBlockParam(st0,'GainTunable20',Gain20_t);
154 setBlockParam(st0,'GainTunable21',Gain21_t);
155 setBlockParam(st0,'GainTunable22',Gain22_t);
156 setBlockParam(st0,'GainTunable23',Gain23_t);
157 setBlockParam(st0,'GainTunable24',Gain24_t);
158
159 controlSystemTuner(st0);
160
161 opt = systuneOptions('RandomStart',30);
162 rng(0);
163
164 % Less than 20% mismatch with reference model 1/(s+1)
165 refsys = tf(1,[1 1 1]);
166 TrackReq = TuningGoal.Transient({'reference'},{'Vx'},refsys);
167 TrackReq.RelGap = 0.2;
168
169 TrackReq1 = TuningGoal.Transient({'Vz_ref'},{'Vz'},refsys);
170 TrackReq1.RelGap = 0.2;
171
172 % TrackReq2 = TuningGoal.Transient({'phi_ref'},{'phi'},refsys);
173 % TrackReq2.RelGap = 0.2;
174
175 TrackReq2 = TuningGoal.Transient({'r_ref'},{'r'},refsys);
176 TrackReq2.RelGap = 0.2;
177
178 TrackReq3 = TuningGoal.Transient({'Vy_ref'},{'Vy'},refsys);
179 TrackReq3.RelGap = 0.2;
180
181
182 % Gain and phase margins at plant inputs and outputs
183 MarginReq1 = TuningGoal.Margins('Comandi',5,40);
184 MarginReq2 = TuningGoal.Margins('Output',5,40);
185 %
186 % Limit on fast dynamics
187 MaxFrequency = 25;
188 PoleReq = TuningGoal.Poles(0,0,MaxFrequency);
189
190 AllReqs = [MarginReq1,MarginReq2,PoleReq];
191
192 [st,fSoft,~,info] = systune(st0,AllReqs,[TrackReq,TrackReq1,TrackReq2,
        TrackReq3],opt);
193 showTunable(st)
194
195 writeBlockValue(st);
196
197 Gain1_tnd = getBlockValue(st,'GainTunable1');
198 Gain2_tnd = getBlockValue(st,'GainTunable2');
199 Gain3_tnd = getBlockValue(st,'GainTunable3');
200 Gain4_tnd = getBlockValue(st,'GainTunable4');
201 Gain5_tnd = getBlockValue(st,'GainTunable5');
```

```
Gain6_tnd = getBlockValue(st,'GainTunable6');
Gain7_tnd = getBlockValue(st,'GainTunable7');
Gain8_tnd = getBlockValue(st,'GainTunable8');
Gain9_tnd = getBlockValue(st,'GainTunable9');
Gain10_tnd = getBlockValue(st,'GainTunable10');
Gain11_tnd = getBlockValue(st,'GainTunable11');
Gain12_tnd = getBlockValue(st,'GainTunable12');
Gain13_tnd = getBlockValue(st,'GainTunable13');
Gain14_tnd = getBlockValue(st,'GainTunable14');
Gain15_tnd = getBlockValue(st,'GainTunable15');
Gain16_tnd = getBlockValue(st,'GainTunable16');
Gain17_tnd = getBlockValue(st,'GainTunable17');
Gain18_tnd = getBlockValue(st,'GainTunable18');
Gain19_tnd = getBlockValue(st,'GainTunable19');
Gain20_tnd = getBlockValue(st,'GainTunable20');
Gain21_tnd = getBlockValue(st,'GainTunable21');
Gain22_tnd = getBlockValue(st,'GainTunable22');
Gain23_tnd = getBlockValue(st,'GainTunable23');
Gain24_tnd = getBlockValue(st,'GainTunable24');

Tuning_value.PID1 = gain_tuned(Gain1_tnd.D,Gain2_tnd.D,Gain3_tnd.D,
    Gain4_tnd.D,Ts);
Tuning_value.PID2 = gain_tuned(Gain5_tnd.D,Gain6_tnd.D,Gain7_tnd.D,
    Gain8_tnd.D,Ts);
Tuning_value.PID3 = gain_tuned(Gain9_tnd.D,Gain10_tnd.D,Gain11_tnd.D,
    Gain12_tnd.D,Ts);
Tuning_value.PID4 = gain_tuned(Gain13_tnd.D,Gain14_tnd.D,Gain15_tnd.D,
    Gain16_tnd.D,Ts);
Tuning_value.PID5 = gain_tuned(Gain17_tnd.D,Gain18_tnd.D,Gain19_tnd.D,
    Gain20_tnd.D,Ts);
Tuning_value.PID6 = gain_tuned(Gain21_tnd.D,Gain22_tnd.D,Gain23_tnd.D,
    Gain24_tnd.D,Ts);

%% Plot
Result_PID = sim('Lynx_prove_qtg_doppioloop.slx',100);
```

## A.5 TXT Helicopter model

```
%% Simulation data and input

Ts = 1/60;
PID = txtImport('Input.txt');
stateSpace = txtImportScript('H145_GW_2990_IAS_73_PA_8605.tab');

% run('Valori_max_min_comandi.m'); % RICORDO -> da mettere nel file di
    input modello

%% Truncated system stability analysis
ss_PID = truncated(stateSpace,[1:4,7:11],[1,4,7,10],[1:4,7:11]);
```

```
11
12 Gol_PID = tf(ss_PID);
13 Gol_zpk_PID = zpk(Gol_PID);
14 lambda_PID = eig(Gol_zpk_PID);
15
16 if isstable(Gol_zpk_PID)
17     disp('Truncated system is stable')
18 else
19    disp('Truncated system is unstable')
20 end
21
22 figure
23 pzmap(Gol_zpk_PID(4,1))
24 rlocus(Gol_zpk_PID(4,1))
25 grid on
26 saveas(gcf,'Results/Root_Locus_truncated_openloop.png')
27
28 %% Model
29 % Reorganization of the H145 matrices in accordance with
30 % the Lynx matrices
31 stateSpace_ord = matrix_analysis(ss_PID
       ,[3,4,1,2,9,5,7,6,8],[4,1,2,3],[3,4,1,2,9,5,7,6,8]);
32
33 mdl = 'Param_model';
34 open_system(mdl);
35
36 count = 1;
37
38 u_max_OL = 20/57.3;
39 u_min_OL = -20/57.3;
40 u_max = 1;
41 u_min = -1;
42 u_max_IL = 1;
43 u_min_IL = -1;
44
45
46 Gain_filter_OL = 0;
47 Gain_derivative_OL = 0;
48 Gain_proportional_OL = 0;
49 Gain_integrative_OL = 0;
50 Gain_filter_IL = 0;
51 Gain_derivative_IL = 0;
52 Gain_proportional_IL = 0;
53 Gain_integrative_IL = 0;
54 Gain_filter = 0;
55 Gain_derivative = 0;
56 Gain_proportional = 0;
57 Gain_integrative = 0;
58
59 gamma = 0;
60 beta = 1;
61
62 % Mi definisco i valori massimi e minimi che possono assumere i diversi
```

```
% gain

Gain1max = 1;
      % Kp
Gain1min = 0.01;
Gain2max = Ts/0.01;
      % Ts/Ti
Gain2min = Ts/20;
Gain3max = 0.5/Ts;
      % Td/Ts
Gain3min = 0.1;
Gain4max = 1/(Ts/(1*0.5) + 1);
      % 1/(Ts/Tf+1)
Gain4min = 0.1;

for i = 1 : length(PID)

      if PID(i).Type == 2

          TuningBlocks(count:count+7) = [strcat(mdl,'/',PID(i).Name,'/
    PID_OL/Filter_OL/GainTunable_filter_OL')...
              strcat(mdl,'/',PID(i).Name,'/PID_OL/
    GainTunable_derivative_OL'),...
              strcat(mdl,'/',PID(i).Name,'/PID_OL/','
    GainTunable_proportional_OL'),...
              strcat(mdl,'/',PID(i).Name,'/PID_OL/','
    GainTunable_integrative_OL'),...
              strcat(mdl,'/',PID(i).Name,'/PID_IL/Filter_IL/
    GainTunable_filter_IL'),...
              strcat(mdl,'/',PID(i).Name,'/PID_IL/
    GainTunable_derivative_IL'),...
              strcat(mdl,'/',PID(i).Name,'/PID_IL/','
    GainTunable_proportional_IL'),...
              strcat(mdl,'/',PID(i).Name,'/PID_IL/','
    GainTunable_integrative_IL')];

          count = count + 8;

          n_gain(i) = 8;

      elseif PID(i).Type == 1

         TuningBlocks(count:count+3) = [strcat(mdl,'/',PID(i).Name,'/
    Filter/GainTunable_filter')...
              strcat(mdl,'/',PID(i).Name,'/GainTunable_derivative'),...
              strcat(mdl,'/',PID(i).Name,'/GainTunable_proportional'),...
              strcat(mdl,'/',PID(i).Name,'/GainTunable_integrative')];

          count = count + 4;

          n_gain(i) = 4;

      end
```

```
103
104 end
105
106 tot_TuningBlocks = sum(n_gain);
107
108 options = slTunerOptions('RateConversionMethod','prewarp','PreWarpFreq'
    ,10);
109 st0 = slTuner(mdl,TuningBlocks,options);
110 addPoint(st0,{'theta_ref','Vz_ref','phi_ref','psi_ref'});
111 addPoint(st0,{'theta','Vz','phi','psi'});
112 addPoint(st0,{'Commands','States'});
113
114 count2 = 1;
115
116 for i = 1:length(PID)
117
118     for j = 1 : n_gain(i)
119
120         if PID(i).Type == 2
121
122             [Gain_f_OL, Gain_d_OL, Gain_p_OL, Gain_i_OL] =
    gain_computation(PID(i).Kp_OL, PID(i).Ti_OL, PID(i).Td_OL, PID(i).
    alpha_OL, Ts);
123             [Gain_f_IL, Gain_d_IL, Gain_p_IL, Gain_i_IL] =
    gain_computation(PID(i).Kp_IL, PID(i).Ti_IL, PID(i).Td_IL, PID(i).
    alpha_IL, Ts);
124
125             Gain_t = getBlockParam(st0,TuningBlocks{count2});
126
127             if ~isempty(strfind(TuningBlocks{count2},'proportional_OL'))
128
129                 Gain_t.Gain.Value = Gain_p_OL;
130                 Gain_t.Gain.Maximum = Gain1max;
131                 Gain_t.Gain.Minimum = Gain1min;
132
133             elseif ~isempty(strfind(TuningBlocks{count2},'integrative_OL
    '))
134
135                 Gain_t.Gain.Value = Gain_i_OL;
136                 Gain_t.Gain.Maximum = Gain2max;
137                 Gain_t.Gain.Minimum = Gain2min;
138
139             elseif ~isempty(strfind(TuningBlocks{count2},'derivative_OL'
    ))
140
141                 Gain_t.Gain.Value = Gain_d_OL;
142                 Gain_t.Gain.Maximum = Gain3max;
143                 Gain_t.Gain.Minimum = Gain3min;
144
145             elseif ~isempty(strfind(TuningBlocks{count2},'filter_OL'))
146
147                 Gain_t.Gain.Value = Gain_f_OL;
148                 Gain_t.Gain.Maximum = Gain4max;
```

```
149                 Gain_t.Gain.Minimum = Gain4min;
150
151             elseif ~isempty(strfind(TuningBlocks{count2},'
    proportional_IL'))
152
153                 Gain_t.Gain.Value = Gain_p_IL;
154                 Gain_t.Gain.Maximum = Gain1max;
155                 Gain_t.Gain.Minimum = Gain1min;
156
157             elseif ~isempty(strfind(TuningBlocks{count2},'integrative_IL
    '))
158
159                 Gain_t.Gain.Value = Gain_i_IL;
160                 Gain_t.Gain.Maximum = Gain2max;
161                 Gain_t.Gain.Minimum = Gain2min;
162
163             elseif ~isempty(strfind(TuningBlocks{count2},'derivative_IL'
    ))
164
165                 Gain_t.Gain.Value = Gain_d_IL;
166                 Gain_t.Gain.Maximum = Gain3max;
167                 Gain_t.Gain.Minimum = Gain3min;
168
169             elseif ~isempty(strfind(TuningBlocks{count2},'filter_IL'))
170
171                 Gain_t.Gain.Value = Gain_f_IL;
172                 Gain_t.Gain.Maximum = Gain4max;
173                 Gain_t.Gain.Minimum = Gain4min;
174
175             end
176
177         elseif PID(i).Type == 1
178
179             [Gain_f,Gain_d,Gain_p,Gain_i] = gain_computation(PID(i).Kp,
    PID(i).Ti, PID(i).Td, PID(i).alpha, Ts);
180
181             Gain_t = getBlockParam(st0,TuningBlocks{count2});
182
183             if ~isempty(strfind(TuningBlocks{count2},'proportional'))
184
185                 Gain_t.Gain.Value = Gain_p;
186                 Gain_t.Gain.Maximum = Gain1max;
187                 Gain_t.Gain.Minimum = Gain1min;
188
189             elseif ~isempty(strfind(TuningBlocks{count2},'integrative'))
190
191                 Gain_t.Gain.Value = Gain_i;
192                 Gain_t.Gain.Maximum = Gain2max;
193                 Gain_t.Gain.Minimum = Gain2min;
194
195             elseif ~isempty(strfind(TuningBlocks{count2},'derivative'))
196
197                 Gain_t.Gain.Value = Gain_d;
```

```
198                     Gain_t.Gain.Maximum = Gain3max;
199                     Gain_t.Gain.Minimum = Gain3min;
200
201                 elseif ~isempty(strfind(TuningBlocks{count2},'filter'))
202
203                     Gain_t.Gain.Value = Gain_f;
204                     Gain_t.Gain.Maximum = Gain4max;
205                     Gain_t.Gain.Minimum = Gain4min;
206
207                 end
208
209             else
210
211                 disp('Error in MathPilot definition, define right parameters
      in the settings file')
212
213             end
214
215             setBlockParam(st0,TuningBlocks{count2},Gain_t);
216
217             count2 = count2 + 1;
218
219         end
220
221 end
222
223 % controlSystemTuner(st0)
224
225 opt = systuneOptions('RandomStart',15);
226 rng(0);
227
228 % Less than 20% mismatch with reference model 1/(s+1)
229 refsys = tf(1,[1 1 1]);
230 TrackReq = TuningGoal.Transient({'theta_ref'},{'theta'},refsys);
231 TrackReq.RelGap = 0.2;
232
233 TrackReq1 = TuningGoal.Transient({'Vz_ref'},{'Vz'},refsys);
234 TrackReq1.RelGap = 0.2;
235
236 TrackReq2 = TuningGoal.Transient({'phi_ref'},{'phi'},refsys);
237 TrackReq2.RelGap = 0.2;
238
239 TrackReq3 = TuningGoal.Transient({'psi_ref'},{'psi'},refsys);
240 TrackReq3.RelGap = 0.2;
241
242
243 % Gain and phase margins at plant inputs and outputs
244 MarginReq1 = TuningGoal.Margins('Commands',5,40);
245 MarginReq2 = TuningGoal.Margins('States',5,40);
246 %
247 % Limit on fast dynamics
248 MaxFrequency = 25;
249 PoleReq = TuningGoal.Poles(0,0,MaxFrequency);
```

```
250
251 AllReqs = [MarginReq1,MarginReq2,PoleReq];
252
253 [st,fSoft,~,info] = systune(st0,[],[TrackReq,TrackReq1,TrackReq2,
        TrackReq3],opt);
254 showTunable(st)
255
256 writeBlockValue(st);
257
258
259 %% Tuning values definition
260
261 count = 1;
262
263 for i = 1:4:tot_TuningBlocks
264
265     Gain1_tnd = getBlockValue(st,TuningBlocks{i});
266     Gain2_tnd = getBlockValue(st,TuningBlocks{i+1});
267     Gain3_tnd = getBlockValue(st,TuningBlocks{i+2});
268     Gain4_tnd = getBlockValue(st,TuningBlocks{i+3});
269
270     Tuning_value(count) = gain_tuned(Gain1_tnd.D,Gain2_tnd.D,Gain3_tnd.D,
        Gain4_tnd.D,Ts);
271
272     if Tuning_value(count).alpha_tuned < 1e-3
273
274         Tuning_value(count).alpha_tuned = 1e-3;
275
276     %elseif Tuning_value(count).alpha_tuned > 10
277
278         %Tuning_value(count).alpha_tuned = 10;
279
280     end
281
282     count = count + 1;
283
284 end
285
286 %% Results plot
287
288 Result_PID = sim('Param_model.slx',300);
289
290 [r_PID,c_PID_at] = size(Result_PID.States.signals.values);
291
292 figure
293 hold on
294 for i = 1:c_PID_at
295     subplot(5,2,i)
296     plot(Result_PID.tout,Result_PID.States.signals.values(:,i),'Linewidth
        ',1.5)
297     xlabel('time [s]')
298     ylabel([stateSpace_ord.StateName{i},' [',stateSpace_ord.StateUnit{i},
        ']'])
```

```
299     grid on
300 end
301 subplot(5,2,4)
302 hold on
303 plot(Result_PID.Reference,'Linewidth',1)
304 hold off
305 grid on
306 sgtitle('Step Response - states')
307 saveas(gcf,'Results/States.png')
308
309 figure
310 hold on
311 plot(Result_PID.tout,Result_PID.States.signals.values(:,4),'Linewidth'
    ,1.5)
312 plot(Result_PID.Reference,'Linewidth',1)
313 hold off
314 title('Step response','PID Controller')
315 ylabel('theta [rad]')
316 xlabel('time [s]')
317 grid on
318 set(gca,'FontSize',14)
319 saveas(gcf,'Results/Step_response.png')
320
321 figure
322 for i = 1:4
323     subplot(2,2,i)
324     plot(Result_PID.tout,Result_PID.Commands.signals.values(:,i),'
    Linewidth',1.5)
325     xlabel('time [s]')
326     ylabel([stateSpace_ord.InputName{i},' [',stateSpace_ord.InputUnit{i},
    ']',])
327     grid on
328 end
329 grid on
330 sgtitle('Step response - commands')
331 saveas(gcf,'Results/Commands.png')
```

# Acknowledgements

There are too many people to thank that helped me during all these five university years.

The biggest "thank you" goes to my family, who have always supported me in every decision and helped me through the most difficult times. It is not possible to quantify your support that is present in each time instant of my life and encourages me to never give up in any situation.

Moreover, I want to express my gratitude to all my supervisors: to Prof. Guglieri who always suggested the right thing to do and to Francesco and Tito from TXT who always found a moment to spend with me throughout my thesis project, even when they had a thousand other things to do .

Of course, I would like to thank all my friends - it would take too long to list them all. In particular, I want to say "thank you" to Toni and Teo with whom I spent all our study sessions and without whom I would certainly not have achieved what I have so far.