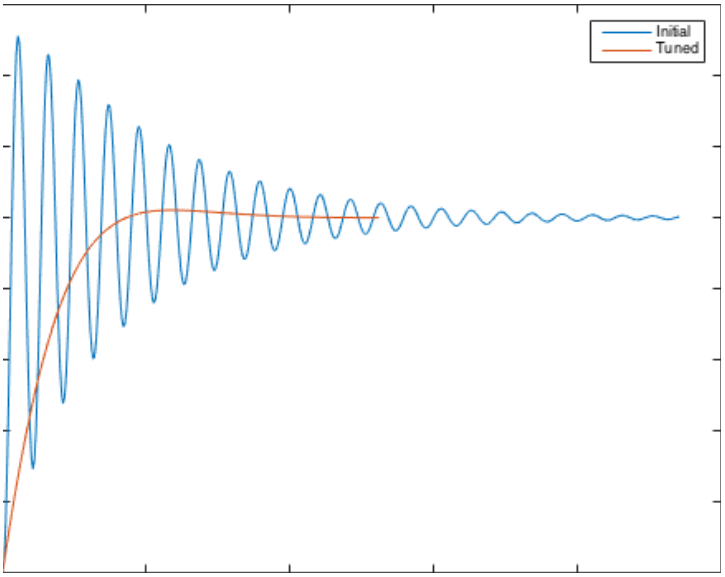
MANCHESTER 1824
The University of Manchester

AUTOTUNING OF AUTOPILOTS FOR SMALL UNMANNED AERIAL VEHICLES



Dimitrios Tsounis

MEng Aerospace Engineering

April 2016

University of Manchester

School of Mechanical, Aerospace and Civil Engineering

Abstract

This study focuses on the development of an autotuning solution for the gains of the pitch attitude PID controller of a fixed wing UAV. Tuning the PID gains of a UAV system is a difficult and lengthy process, which if done incorrectly could have serious consequences. Researchers have attempted to tackle this problem from many different approaches but no conclusive evidence have been presented. In this project, the pitch dynamics of a UAV are modelled as a second order transfer function. This is a simplification that is sufficient for the purposes of this study, as the primary aim is to create a proof-of-concept solution. The conclusions of the literature review suggested that analytical tuning methods, like Ziegler-Nichols, are not appropriate for use in UAV systems. Consequently, the field of heuristic techniques was further explored. The main output of this study is a metaheuristic algorithm that is able to tune the gains of a PI controller. The algorithm is a variation of a Genetic Algorithm with additional elements, specific to UAV systems, included. The tuning of the controller is based on a set of initial criteria defined by the user. The autotuning algorithm, which has no knowledge of the system's dynamics, was able to calculate gains that satisfied the user-defined criteria in more than 95% of all simulations. Furthermore, the gains obtained from the algorithm were within a 55% range of the ideal gains calculated from the Matlab built-in PID tuning function for the same system. Those results highlight the versatility of the autotuning algorithm, but further work is required before the algorithm can be ported to autopilots.

Nomenclature & Glossary

CAA: Civil Aviation Authority

CPU: Central Processing Unit

EoMs: Equations of Motion

GA: Genetic Algorithm

HIL: Hardware In the Loop

IMU: Inertial Measurement Unit

MIMO: Multi Input - Multi Output

MTOM: Maximum Take-off Mass

SISO: Single Input - Single Output

SMD: Spring, Mass and Damper

SPO: Short Period Oscillation

SUAV: Small Unmanned Aerial Vehicle

UAV: Unmanned Aerial Vehicle

VTOL: Vertical Take-Off and Landing

1 Introduction

An Unmanned Aerial Vehicle (UAV) is a flying vehicle that does not have a human pilot on board and is either controlled by a pilot on the ground or autonomously from an on-board computer. As defined by the CAA, a Small UAV (SUAV) is a vehicle with a maximum take-off mass (MTOM) of less than 20 kg. The popularity of SUAVs has soared in the past few years and they are now frequently used for hobby and commercial purposes. This comes as a result of a decrease in price and the enhanced safety standards of the vehicles.

Figure 1: Skywalker UAV

One of the enabling technologies for the safe operation of UAVs has been the development of reliable and affordable autopilots.

An autopilot is a computer board that implements an algorithm to control the position and orientation of a UAV and navigate it autonomously. The building blocks of modern autopilots are the processing unit (CPU), sensors (usually the IMU) and the output to the actuators. Most control algorithms use cascade PID controllers that send the appropriate commands to the UAV.

An important stage of a UAV development process is the selection of appropriate gains for the autopilot's PID controllers. If the gains of a UAV are not appropriate then the aircraft will exhibit poor handling qualities. Tuning the gains of a controller requires a manual trial-and-error approach, which

sometimes can yield suboptimal results. Using the current level of technology, the PID gains are only tuned once and remain the same for the duration of a flight.

An autopilot that is able to adjust the gains of the PID controller autonomously, based on the characteristics of each airframe and the operating conditions, would greatly enhance the safety, performance and stability of UAVs.

This project aims to explore the field of autotuning for the PID pitch attitude controller of fixed-wing UAVs. The objectives of the project are:

- Research the field of UAV autotuning and complete a thorough literature review, documenting the latest developments
- Analyse the longitudinal dynamics of a UAV
- Examine the dynamics of 2^{nd} order systems
- Create an autotuning algorithm for a 2^{nd} order system
- Test the performance and robustness of the autotuning algorithm

Autotuning a PID controller for an aircraft is a very challenging task and as seen in the literature review, many researchers have attempted to tackle this problem in the past. Because the motion of an aircraft is very complex, the nonlinearities of the system are usually not accounted for in computer models and other unknown parameters are estimated/extrapolated from known data. This results in autotuning methods that are obtained for simplified and idealised systems and do not perform well on real-life vehicles. In this project, the necessary assumptions will be made, but there will be an effort to keep the results as realistic as possible.

This report follows a logical structure with the required theory and literature review presented in chapters 2 and 3 accordingly. In chapter 4 the research method and preparatory work is presented. Chapter 5 includes a detailed description of the implemented autotuning algorithm. Chapter 6 presents the results from the software experiments that were conducted. Additionally, in chapter 6 the results are critically analysed and the strengths, limitations and applicability of the algorithm to UAVs are discussed. Chapter 8 summarises the main conclusions of the project. Finally, chapter 7 is a description of the future steps that should be taken based on the results obtained from the experiments and suggestions about similar projects.

2 Theory

2.1 Control Engineering

A thorough analysis of the autotuning challenge for UAVs requires a solid understanding of control engineering and the underlying physical and mathematical principles. Control engineering is a multidisciplinary field and has found applications in many engineering areas like, mechanical, electrical, chemical and aerospace. Control engineering deals with the behaviour and response of dynamical systems (Bolton, 1998). A dynamical system is a system whose state changes over time. As the name implies, control engineering aims to regulate ('control') the output of such systems to a desired value using a variety of techniques, like feedback. Feedback can be defined as the modification of the input to a system, based on its output (Bolton, 1998). This can be done intentionally, though the integration of a sensor to measure the process variable or it can be an inherent characteristic of the system (e.g. back-EMF of DC motors). PID controllers are used in feedback control systems to modify the input to the plant depending on the deviation of the plant's behaviour from the desired one. PID controllers are ubiquitous and can be found in most industrial applications, like the regulation of power plants (e.g. nuclear reactors) and the stability augmentation software of modern aircraft. Their popularity is due to the easiness of integration and their robustness.

2.1.1 History

James Watt was the first to introduce a negative feedback mechanism, the centrifugal governor (Figure 2), on a steam engine in 1788. The shaft of the

engine is connected to the fly-ball governor, which is then connected to the throttle control of the engine. Using the balance between angular momentum and gravity, the governor regulated the throttle lever of the engine, thus maintaining a constant speed. The centrifugal governor was the first example of *proportional control* usage (Åström, Murray, 2008).

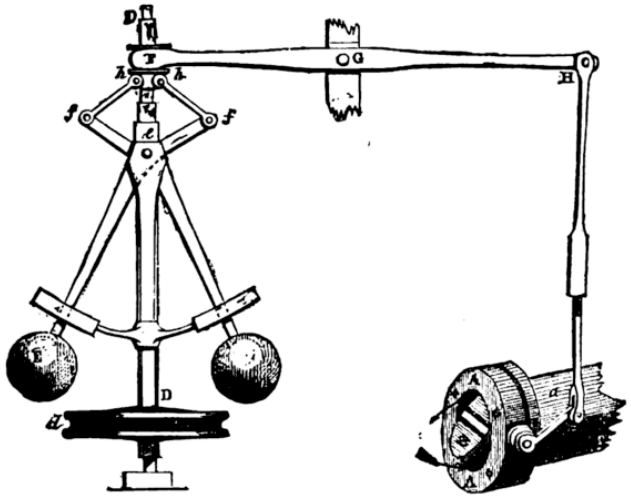


Figure 2: Centrifugal Governor

Over a century later, in 1911, Elmer Sperry developed the first “automatic ship steering mechanism”, termed *gyropilot* (Roberts, et.al.,2003). Sperry’s gyropilot was the result of his earlier invention, the gyrocompass. The gyropilot was the first industrial device that made use of the PD control concept. The ship’s rudder deflection was proportional to the heading error but it also included an “anticipator” in order to avoid overshoots (Roberts, et.al.,2003). A few years later, Nicholas Minorsky (1922) laid the theoretical foundation of PID controllers, by examining the controls applied by helmsmen. He theorised that experienced pilots accounted for the angular velocity of the ship (derivative term) and the magnitude of the heading error (integral term) (Roberts,et.al.,2003). Minorsky established the mathematical background that was necessary for the development of PID controllers.

At that time, control engineering was still a field at its infancy. In the years 1930-1950, the classical control theory was established. The development of the field was partly fuelled by World War II. Engineers were tasked with designing anti-aircraft guns that could reliably track aircraft (Bennett, S.). After the end of the war and with the development of computers, PID controllers were converted from mechanical devices to analogue and are now usually found in software programs.

2.2 PID Controllers

In a feedback system the error is defined as the difference between the desired and the actual output of the plant. The controller is the element in the closed-loop control system, which accepts as an input the error signal and produces an output that is fed to the plant. The *control law* is the formula that determines the value of the output of the controller (Bolton, 1998). The control law of a PID combines a proportional element, a derivative and an integral one. A block diagram of a PID controller integrated in a control system can be seen in Figure 3, where a unity feedback system is depicted.

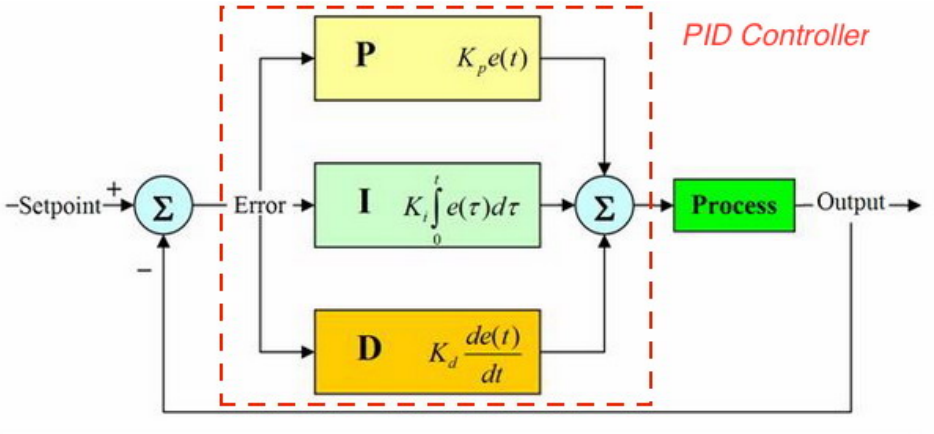


Figure 3: PID Controller

The proportional (P) element of the controller produces an output that is proportional to the input (the error, e). The term k_p is called the proportional gain. The proportional output of a PID controller is responsible for over 50% of the output of the plan. However, large proportional gains can cause a system to oscillate and potentially destabilise it (Bolton, 1998).

The integral (I) element of the controller integrates the error and multiplies that result with the corresponding gain, K_i. The integral term can be thought of as a time history of the error. If the steady-state error remains significant the integral part of the controller will also increase and try to reduce that error.

The D element calculates the rate of change of the error and that derivative is multiplied by the derivative gain, K_d. The derivative term "predicts" the value of the error in the future. Initially, the D term will be high because the error is large, hence its rate of change too. However, as the error becomes smaller, its rate will decrease, thus the D term will become smaller and prevent the system from overshooting (Kuo, 1995).

The output of the three different elements is then summed together and the control command is passed to the plant. It is evident that an appropriate value for the gains of the controller is required in order to achieve the desired behaviour.

2.2.1 Mathematical Form

The mathematical definition of a PID controller is given by Equation 1.

$$u(t) = K_p e(t) + K_i \int_0^t e(t)\,dt + K_d \frac{de(t)}{dt} \quad (1)$$

Equation 1 is also known as the *ideal form* of the PID controller formula (Bolton, 1998).

The transfer function of the controller is then:

$$\frac{u(s)}{e(s)} = G_c(s) = K_p + \frac{K_i}{s} + K_d s \quad (2)$$

At this point we can define the integral time constant $\tau_i = \frac{K_p}{K_i}$ and the derivative time constant $\tau_d = \frac{K_d}{K_p}$. The transfer function is,

$$G_c(s) = K_p \left(1 + \frac{K_i}{K_p s} + \frac{K_d s}{K_p}\right) \quad (3)$$

$$G_c(s) = K_P \left(1 + \frac{1}{\tau_i s} + \tau_d s\right) \quad (4)$$

Converting Equation 4 to the time domain and multiplying by $e(t)$:

$$u(t) = K_p(e(t) + \frac{1}{\tau_i} \int_0^t e(t)dt + \tau_d \frac{de(t)}{dt}) \quad (5)$$

Equation 5 is called the *standard form* of the PID controller formula. The standard form is used more often for practical applications of PID controllers whereas the ideal form is preferred for theoretical analysis (Kuo, 1995).

2.3 Autopilots

A great range of SUAV autopilots has been developed in the recent years. Some of those autopilots are proprietary and the source code is not publicly available. However, with the rise of the open source community both open source software and hardware autopilots were created. One of the open source autopilots that has enjoyed a significant adoption rate in the SUAV community is the Pixhawk Autopilot (Figure 4). Pixhawk is used by hobbyists, small-medium enterprises (SMEs) and research groups.

Figure 4: Pixhawk 3DR autopilot

This can be explained by the fact that Pixhawk is a very robust autopilot, highly customisable, can support different flight stacks and has a lower price when compared to other proprietary platforms. Pixhawk is equipped with a high-end processor and sensors, which are summarized in Table 1. The data collected through the sensors are fused together in order to obtain the current position and attitude of the UAV (state estimation). Using the current state and the input of the user (manual or waypoint navigation) the autopilot calculates the actions that needs to take (path planning).

A critical element of a UAV system is the software program of the autopilot that contains the control law of the vehicle. That software is frequently referred to as the flight stack. Two open source flight stacks are currently used for the Pixhawk autopilot. The first one is Ardupilot, which branches in

to three different fields: fixed wing aircraft with ArduPlane, multirotors with ArduCopter and land-based rovers with APMRover. The second one is the PX4 flight stack, which was originally developed at ETH Zurich (Meier et al. n.d.). This report will examine the PX4 flight stack, as it is the personal belief of the author that the PX4 software is the most customisable for research activities and its future use will increase. Hence, a solid understanding of the system's architecture is required.

Table 1: Pixhawk Specifications

Processor	168MHz 32bit STM32F427 Cortex M4 core
Gyroscope	16bit ST Micro L3GD20H
Accelerometer/Magnetometer	14bit ST Micro LSM303D
Accelerometer/Gyroscope	Invensense MPU 6000 3-axis
Barometer	MEAS MS5611

2.3.1 PX4 Flight Stack

The PX4 software consists of four different layers that are interchangeable, thus making the system truly modular. The different layers of the system are

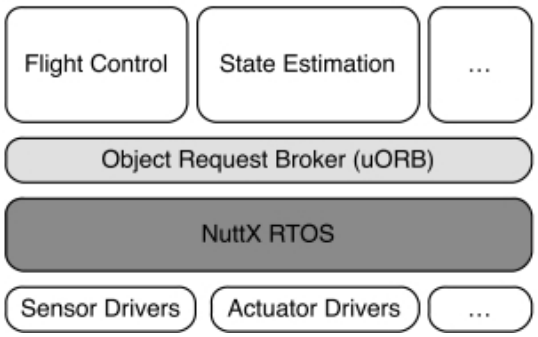


Figure 5: PX4 Flight Stack

depicted in Figure 5.

The bottom layer contains the sensor drivers, which are specific to the chosen hardware. In the second layer the NuttX Real Time Operating System (RTOS) can be found. NuttX is a highly scalable RTOS and strives to achieve a high degree of standards compliance (NuttX, 2015). The next layer contains the Object Request Broker (uORB), which is responsible for the message marshalling between the different components of the autopilot. The uORB is based on the previous work of Huang et al. (2010). Finally, on the top software level, the flight controllers and the state estimation software can be found. The flight controllers run as independent applications. This abstraction layer is of great benefit to engineers not familiar with low-level hardware design, as they can focus exclusively on the design of the controller. Furthermore, the implementation of an autotuning algorithm would take place at this layer.

2.4 PID Controllers & Autopilots

An important concept of control engineering is that of Single Input-Single Output (SISO) and Multi Input-Multi Output (MIMO) systems. As the name implies, a SISO system accepts only one input and has a unique output. On the other hand, a MIMO system has multiple inputs and outputs.

PID controllers are ideal for SISO systems. However, for MIMO systems PID controllers are not appropriate because PIDs can only have a single input/output.

An aircraft is considered to be a MIMO system because there are multiple inputs (e.g. different control surfaces) and multiple outputs (e.g. pitch, roll,

yaw). It is possible to decouple the equations of motion, but for an aircraft that is inherently difficult. For example, decoupling yaw and roll is almost impossible, as a roll input, will produce not just a roll output but also a yaw one. One way to convert a MIMO system to a SISO is to use quasi decoupling. In that case it is assumed that one input has only one output and all other results are ignored.

Under that assumption, it is possible to implement a PID controller for each of the decoupled systems. Furthermore, since the system is now in standard SISO form, the tuning of the PID controller can take place using one of the established PID tuning methods, like the Ziegler-Nichols method (Ziegler & Nichols, 1942). It should be noted that the Ziegler-Nichols method could produce uncontrolled oscillations in the system, which is an undesired behaviour. However, this is the case with other tuning algorithms, as it will be shown in the following chapters.

Another way to tune a PID controller is if the system's transfer function is known. The transfer function of a system can be obtained through the process of system identification. However, most system identification techniques require that the system is linear (Gondhalekar, 2009). A linear system is one that has only polynomials in each transfer function (e.g. no sine/cosines or other exponential expressions). This is the case for quadrotors and fixed wing aircraft flying at small angles, as the cosine and sine of the flight angles can be approximated to be equal to 1 and the flight angle respectively. However, for angles greater than 10-20 degrees the system becomes non-linear and the identification techniques are no longer applicable.

3 Literature Review

The aim of this chapter is to discuss and review the current approaches towards autotuning PID controllers for SUAVs. A thorough literature review has taken place in order to study the latest developments in the field. An iterative approach was followed for the identification and analysis of the most relevant papers. A search of the library catalogue for 'PID Autotuning' provided several results of papers discussing the concept of PID autotuning for industrial processes. Some of the principles used in industrial PIDs are applicable to UAV controllers and were retained for further analysis. However, the majority of relevant papers were discovered using search terms like 'UAV autotuning', 'aircraft autotuning', 'adaptive control', 'system identification' and 'UAV PID controllers'.

Fadil et al. (n.d.) examined a ducted fan Vertical Take Off and Landing (VTOL) UAV. The actuators were modelled as 2nd order transfer functions and three different tuning methods were analysed. Initially, the Ziegler-Nichols ultimate cycling method was used to tune the PID gains. The K_d, K_i gains were reduced to their minimum values and the K_p gain was gradually increased until a sustained periodic oscillation was obtained. The frequency of that oscillation, also known as the ultimate frequency, P_u, was calculated and then used to calculate the integral and derivative time constant τ_i, τ_d, according to Table 2. However, the obtained simulation results from the Ziegler-Nichols ultimate cycling method produced a diverging oscillation in the motion of the quadrotor, which is an undesired behaviour.

Table 2: Ziegler-Nichols Tuning Table (Fadil et al. n.d.)

	K_c	τ_i	τ_d
P	$\frac{K_u}{2}$		
PI	$\frac{K_u}{2.2}$	$\frac{p_u}{1.2}$	
PID	$\frac{K_u}{1.7}$	$\frac{p_u}{2}$	$\frac{p_u}{8}$

Fadil et al. then examined a heuristic method for calculating the optimum PID gains. The response obtained with the heuristic method was satisfactory, but a considerable amount of time is required to tune the controller, hence making this approach non-ideal.

Finally, Fadil et al. developed an Iterative Learning Control (ILC) autotuning algorithm. Three learning matrices, Φ, Γ, Ψ are utilised to update the values of K_p, K_i, K_d at each 'iteration'. The performance of the algorithm was tested in the Simulink environment and the obtained results were satisfactory. However, the paper presented no evidence of actual flight-testing and the algorithm was not tested on different UAV models.

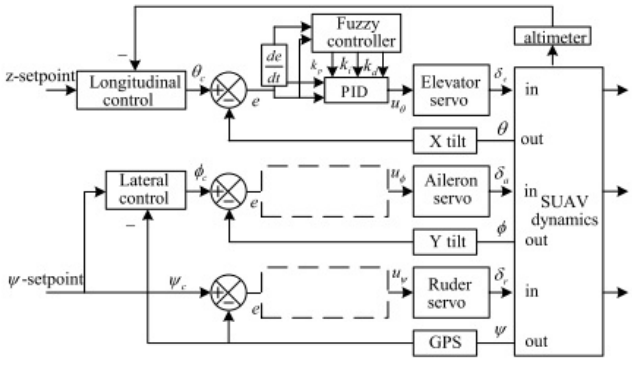


Figure 6: Fuzzy Logic PID Autotuner (Liu et al.)

Liu et al. (2007) suggested the use of an online (in-flight) fuzzy self-adaptive

PID attitude controller for SUAVs. A diagram of the controller can be seen in Figure 6.

Matko et al. (2013) developed a self-tuning Dynamic Matrix Controller (DMC) for a two-axis autopilot for small aeroplanes. Although the results they presented indicate a good response from the aircraft, the use of DMC makes the approach less applicable to current SUAVs, which utilise PID controllers.

Catena et al. (2013) described a novel autotuning procedure for the Volcan UAV using Aström and Hägglund's tuning method (Åström, Hägglund, 1995). The algorithm was developed in Simulink and was interfaced with the flying system using the CANAerospace protocol (Stock, 2006). Simulations were completed in a Hardware-In-the-Loop (HIL) environment, by connecting the Simulink algorithm to a model of the UAV in the X-Plane flight simulator. Catena et al. developed a Matlab GUI that initiated the autotuning procedure and reference signals were sent to the simulator and the response of the system was measured. The gains of the PID controller were modified until a satisfactory value for the tuning criteria has been achieved. Although this paper presented a novel technique, which is applicable to other systems as well, no tests on different platforms were presented. Additionally, no actual flight data were provided.

An important step towards autotuning a controller is the identification of the system (transfer function). Various researchers have worked on this problem.

Vanin (2013) identified the transfer functions of pitch, roll, yaw and throttle of a quadrotor using the Simulink System Identification Toolbox. After collecting the data and identifying the relevant parameters, the resulting model was

compared against a mathematical model of the quadrotor that was developed from first principle.

Safaee & Taghirad (2013) used a Trex-600 electric helicopter and by analysing the frequency response of the plant they were able to obtain the transfer functions of pitch, roll, yaw and altitude. Furthermore, using the information about the identified system, they developed a robust mixed sensitivity controller for each motion (transfer function). Finally, they designed an H2/H∞ controller and compared the performance of the two designs methods, but no conclusive results were presented.

One more approach that is being researched, as described by Rivas-echeverría et al. (n.d.) is the use of neural network for autotuning the PID gains. Pirabakaran & Becerra (2002) presented a neural network autotuner for PID controllers using the Model Reference Adaptive Control approach. However, their findings are related to industrial PID controllers and consequently not directly applicable to SUAV autopilots.

Ardupilot, one of the most popular open source autopilots in the market, has developed an autotuning solution for its flight stack. As the stack is split in two components, ArduCopter and ArduPlane, two different techniques have been implemented for the different types of UAVs. However, both systems are based on heuristic methods to tune the PID controllers.

The algorithm used for the autotuning of multirotors (ArduCopter) is presented in Table 3. When the multirotor is at a safe altitude above the ground and in an area clear of any obstructions the following actions are performed by the autopilot:

Table 3: ArduCopter Autotuning Procedure (Tridgell, 2015)

a) The autopilot invokes 90 deg/sec rate request
b) Records maximum "forward" roll rate and bounce back rate
c) When copter reaches 20 degrees or 1 second has passed, it commands level
d) Tries to keep max rotation rate between 80% ~ 100% of requested rate (90deg/sec) by adjusting rate P
e) Increases rate D until the bounce back becomes greater than 10% of requested rate (90deg/sec)
f) Decreases rate D until the bounce back becomes less than 10% of requested rate (90deg/sec)
g) Increases rate P until the max rotate rate becomes greater than the request rate (90deg/sec)
h) Invokes a 20deg angle request on roll or pitch
i) Increases stab P until the maximum angle becomes greater than 110% of the requested angle (20deg)
j) Decreases stab P by 25%

In the case of ArduPlane, a similar approach is used, but now the pilot is the one commanding the changes in attitude. When the pilot selects the Autotune mode, the demanded roll and pitch rate will be monitored and when they exceed 80% of the maximum rate, the Autotune system will use the response of the aircraft to learn the roll or pitch tuning values (ArduPilot, n.d.).

Paparazzi, another popular open source autopilot, has developed an adaptive controller for its flight stack (Ronfle-Nadaud 2014). However, the adaptive controller is not based on the online tuning of the gains.

Concluding the literature review, we can say that many researchers have examined the ‘autotuning problem’. Most results though, are based on HIL and SITL tests, which sometimes can provide suboptimal results. Finally, more flight tests should be conducted in order to validate autotuning algorithms.

4 Research Method

4.1 Introduction

The concept of autotuning, as described in the previous chapters, is a very abstract problem that is difficult to develop solutions for and then test the performance of those solutions.

In this project, it was decided that all experiments would be conducted in a software environment. The software of choice was Matlab and the built-in Simulink package. Matlab is a numerical computing programming language. Simulink is a graphical programming environment for modelling and simulating dynamical systems. Both Matlab and Simulink have been used for many years in the industry and their performance and reliability is well established.

In the initial stages of the project, the idea of implementing an autotuning algorithm directly in the source code of the PX4 flight stack was briefly considered. The main advantage of this approach would be the immediate application of the algorithm on an actual aircraft. This would allow for a variety of HIL and flight tests, which would provide the most meaningful information about the performance of the algorithm. However, this approach comes with significant drawbacks. A detailed knowledge of the PX4 source code and the Pixhawk hardware is required. A series of flight tests would be necessary in order to obtain accurate results about the algorithm. Conducting flight tests though, is a lengthy process that depends on outside factors (e.g. weather, airfield availability). Multiple airframes would also have to be tested to ensure the robustness of the algorithm. Consequently, developing the

autotuning solution on the PX4 flight stack would be a lengthy, cumbersome and expensive process.

4.2 System Modelling

Following the initial decision on the choice of software, the next phase was the selection of the modelling method of the aircraft dynamics. As stated in the introduction, this project focuses only on the longitudinal motion of a UAV. The lateral dynamics of a UAV were ignored in this analysis. However, this does not mean that they do not have an effect on the longitudinal motion of the aircraft. The dynamics of an aircraft are described by a set of non-linear, coupled equations of motion. Hence, ignoring the lateral dynamics is a simplification, which could be a potential source of error.

4.2.1 Equations of Motion

The full derivation of the Equations of Motion (EoMs) is available in (Cook 2007) and will not be developed here. The decoupled, linearised EoMs describing the longitudinal response of an aircraft to a small perturbation, while in level flight, in derivative form are:

$$m\dot{u} - X_u u - X_{\dot{w}}\dot{w} - X_w w - X_q q + mg\theta = X_\eta \eta \tag{6}$$

$$-Z_u u + (m - Z_{\dot{w}})\dot{w} - Z_w w - \left(Z_q + mU_e\right)q = Z_\eta \eta \tag{7}$$

$$-M_u u - M_{\dot{w}}\dot{w} - M_w w + I_y \dot{q} - M_q q = M_\eta \eta \tag{8}$$

A description of the derivatives in the EoMs is provided in Appendix C. An important assumption of Equations 6-8 is that the thrust does not vary with speed or incidence. Although this is technically possible, most SUAVs and

commercial aviation aircraft vary the thrust as a function of speed, altitude and incidence in order to maintain a trimmed state.

4.2.2 Solving the EoMs

The solution of the EoMs provides a time history of the motion of an aircraft following a disturbance. This is especially important as it allows the characterisation of the static and dynamic stability of the vehicle. The EoMs could be solved analytically, but this is an arduous and time-consuming process. Instead, by transforming the equations to the Laplace domain, the differential equations are converted to algebraic and can be solved with manipulation of the variable of interest. The ratio of the output to the input of a plant (e.g. aircraft) in the Laplace domain is defined as the transfer function of that plant (Bolton, 1998). The time response of an aircraft following a perturbation can be obtained by calculating the inverse Laplace transformation of the transfer function. Using the transfer function of a plant conceals some of the inner mechanics of a plant, but enables a systems approach to a problem. In this project the transfer functions of the EoMs were used to model the dynamics of the aircraft, as the main focus is on the autotuning methods of the plant.

The longitudinal transfer functions that are considered in this project are:

$$\frac{u(s)}{\eta(s)} = Change\ in\ axial\ speed\ due\ to\ elevator\ input \quad (9)$$

$$\frac{w(s)}{\eta(s)} = Change\ in\ vertical\ speed\ due\ to\ elevator\ input \quad (10)$$

$$\frac{q(s)}{\eta(s)} = Change\ in\ pitch\ rate\ due\ to\ elevator\ input \quad (11)$$

$$\frac{\theta(s)}{\eta(s)} = Change\ in\ pitch\ angle\ due\ to\ elevator\ input \quad (12)$$

Of particular interest to this project are the last two transfer functions, Equations 11-12. The mathematical form of the transfer functions is:

$$\frac{q(s)}{\eta(s)} = \frac{k_q s\left(s + \left(\frac{1}{T_{\theta 1}}\right)\right)\left(s + \left(\frac{1}{T_{\theta 2}}\right)\right)}{\left(s^2 + 2\zeta_p\omega_p s + \omega_p^2\right)(s^2 + 2\zeta_s\omega_s s + \omega_s^2)} \quad (13)$$

$$\frac{\theta(s)}{\eta(s)} = \frac{k_q \left(s + \left(\frac{1}{T_{\theta 1}}\right)\right)\left(s + \left(\frac{1}{T_{\theta 2}}\right)\right)}{\left(s^2 + 2\zeta_p\omega_p s + \omega_p^2\right)(s^2 + 2\zeta_s\omega_s s + \omega_s^2)} \quad (14)$$

The transfer functions have been expressed in terms of damping ratios and natural frequencies in order to simplify understanding.

It can be seen from the transfer functions that the longitudinal dynamics of an aircraft are described by a fourth order system. The denominator consists of two 2nd order polynomials. These two polynomials describe the two longitudinal dynamic modes of an aircraft, the *Phugoid* and the *Short Period Oscillation (SPO)*. The longitudinal stability of a UAV depends only on those two modes. Accurate estimations for the values of damping ratio and natural frequency, which depend on the design characteristics of the aircraft, can be obtained by examining the aerodynamic derivatives of the aircraft.

Previous studies (Aliyu et al. 2015)(Samuelsson 2012) have shown that the phugoid mode in UAV aircraft is heavily damped and is rarely observable.

The SPO is the principal observable mode in UAVs. Hence a treatment of the longitudinal dynamics of a UAV as a second order system is adequate for the purposes of this study. The short period pitching motion of an aircraft can be related to the motion of a mass connected to a spring and a damper, with the damping provided by the tail of the aircraft.

4.3 Second order linear system dynamics

The most common example of a second order system is that of a Spring, Mass and Damper (SMD) system. SMD systems are ubiquitous in everyday life and easy to visualise.

The equation describing a second order system can be written in terms of natural frequency (Bolton, 1998), where Θo is the output and Θi the input of the system:

$$\left(\frac{d^2\Theta_o}{dt^2}\right) + 2\zeta\omega_n\left(\frac{d\Theta_o}{dt}\right) + \omega_n^2\theta_o = b_o\omega_n^2\Theta_i \qquad (15)$$

Converting equation 15 to the Laplace domain:

$$G(s) = \frac{\Theta_o(s)}{\theta_i(s)} = \frac{b_o\omega_n^2}{s^2 + 2 * \zeta\omega_n s + \omega_n^2} \qquad (16)$$

Equation 16 is the standard form of the transfer function of a second order system. Specifying different values of damping ratio and natural frequency will describe different systems. According to (Cook 2007) the average range of ω_n values for SPO motion is between $1\ to\ 10 rad/s$. Rynaski (1985) estimated values of $\omega_n = 2 - 6\ rad/s$. The damping ratio varies between 0 (no damping) and 1 (critically damped). Extreme values, such as 0 and 1 are

not realistic, hence the range of damping ratios that will be examined in this study will be $0.2 < \zeta < 0.8$.

4.4 Experimental Setup

4.4.1 Introduction

It has been demonstrated that a 2nd order transfer function is adequate to model the pitch dynamics of a UAV. Consequently, this project focused on the development of an autotuning solution for a 2nd order system.

The first step was to study the behaviour and response of such systems when subjected to a unit step input. The open loop and closed loop response of various 2nd order systems was examined. In the next stage, a PID controller was added in order to assess the change in response due to different PID gains. It became apparent that the use of appropriate gains had a critical role on the performance, stability and robustness of the systems.

The next step of the project was the development of the autotuning algorithm, which is presented in chapter 5. After the development of the algorithm, two different experiments were designed to test its performance. Both of those experiments are described in chapter 6.

4.4.2 Open loop response

The following graphs show the open loop response of 2nd order systems when they are subjected to a unit step input. Each graph depicts a different system with different natural frequency and damping ratio. It can be assumed that the response obtained from the 2nd order systems in the following graphs

is similar to the angle of attack response of a UAV following a control (elevator) input.

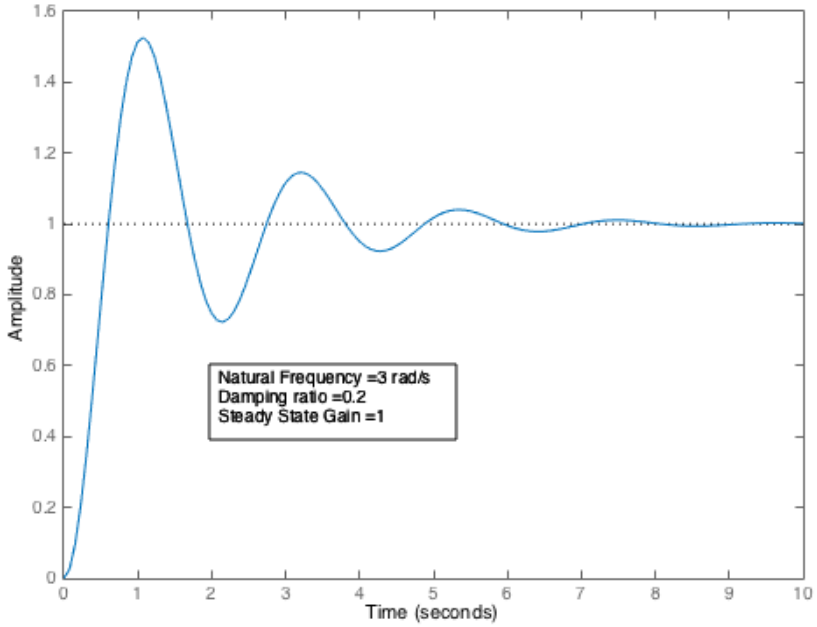


Figure 7: 2nd Order System Open Loop Unity Step Input Response

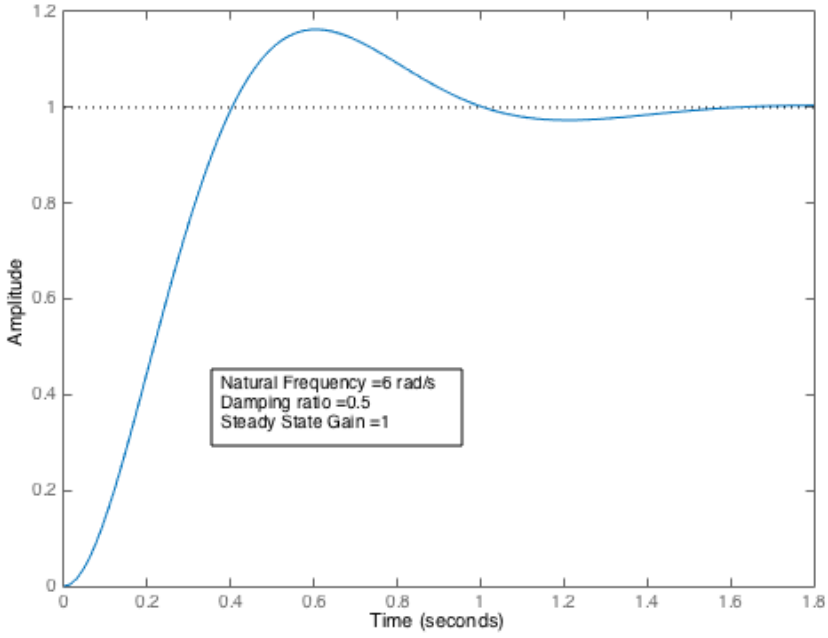


Figure 8: 2nd Order System Open Loop Unity Step Input Response

It is visible that a reduced value of damping ratio causes the system to oscillate. This is in accordance with what is normally expected from underdamped systems. Additionally, the oscillatory motion of the system persists for a longer period in Figure 7,which has a lower natural frequency. In Figure 8, the system reaches its steady state in less than 2 seconds, whereas in Figure 7 the system reach a steady state 8 seconds after the initial disturbance.

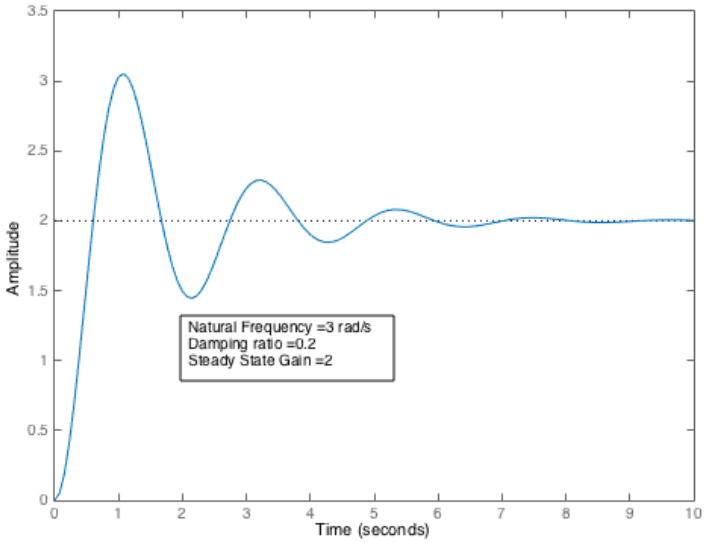


Figure 9: 2nd Order System Open Loop Unity Step Input Response

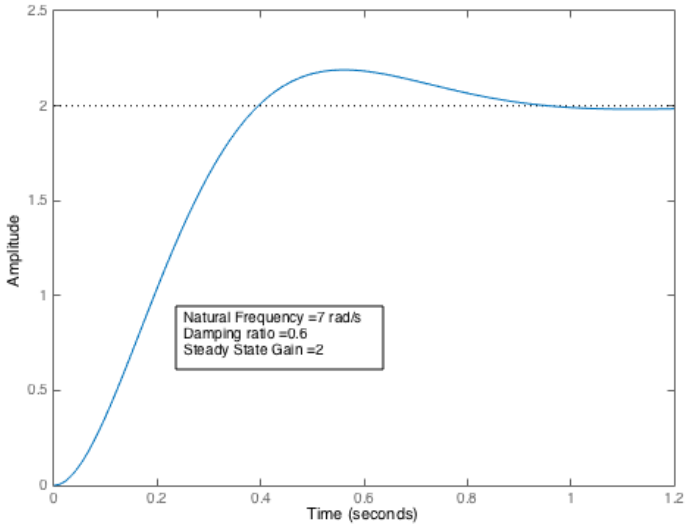


Figure 10: 2nd Order System Open Loop Unity Step Input Response

In Figure 9 and Figure 10 the gain (b_0) was increased to two times the initial value. As expected, the output scaled accordingly. The overshoot in Figure 9 has also increased, when compared to Figure 7. The settling and rise time in both cases remains similar though, which implies that the settling/rise time properties of a system are independent of the steady state gain.

4.4.3 Closed loop response with no control

In the next stage of the experiment a unity feedback path was added to the system. Feedback is used to modify the input to the plant based on its output in order to reach a desired state.

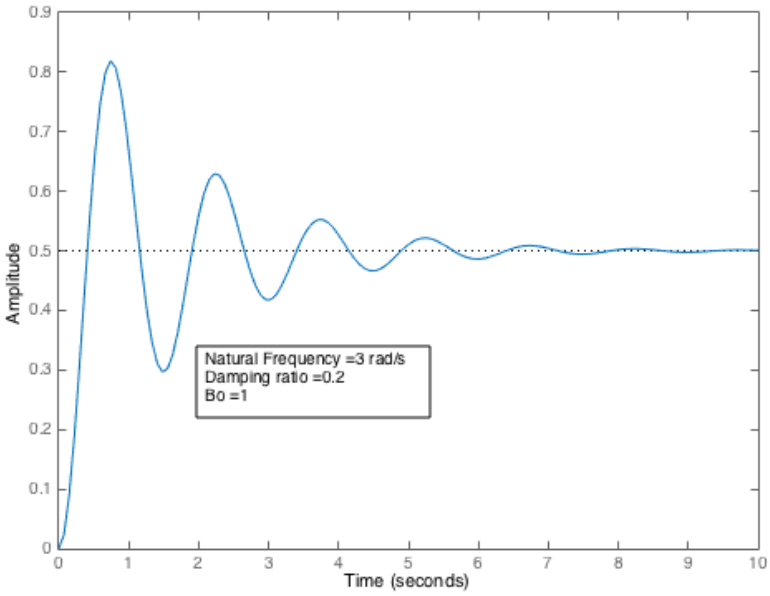


Figure 11: 2nd Order System Closed Loop Unity Step Input Response

It can be seen in Figure 11 and Figure 12 that the addition of a feedback path to the system has a destabilising effect. The system oscillates for longer periods and the steady state gain has also changed. The system actually never reaches its steady state, as the maximum amplitude achieved is that of 0.8 and the steady state gain is 0.5. Consequently, the system has a 50% steady state error.

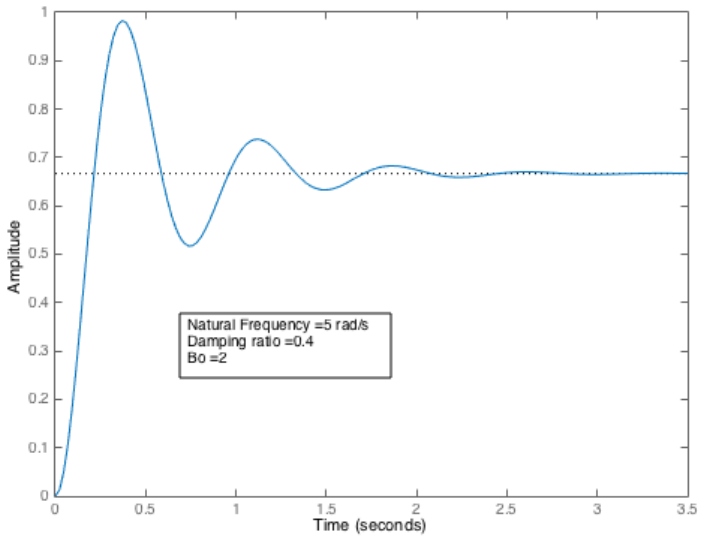


Figure 12: 2nd Order System Closed Loop Unity Step Input Response

In Figure 12, with a b_0 value of 2, the increased instability and undershoot are also observed. The steady state gain of the system has increased to the value of ≈ 0.7, but is less than the requested steady state value of 1. It is interesting to note at this point that although the value of b_0 was twofold increased, that did not translate to a twofold increase of the steady state gain.

4.4.4 Closed loop response with control

In this case, a PID controller has been added to the system. The parameters that were allowed to vary were: damping ratio, natural frequency and the gains of the PID controller. The purpose of this simulation was to analyse and quantify the relationship between the gains of a PID controller and the performance of a closed loop system. Samples responses are presented in the following graphs.

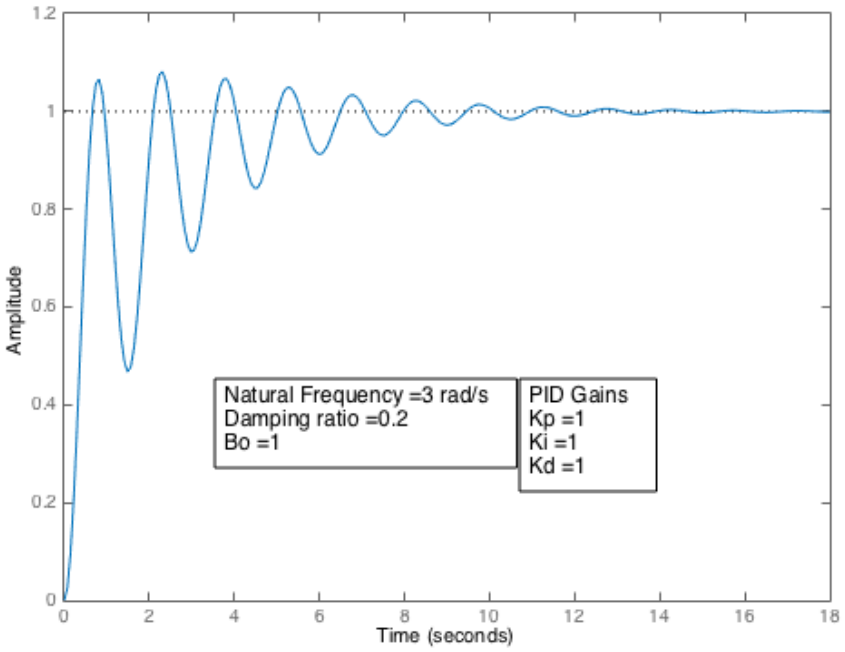


Figure 13: 2nd Order System Closed Loop Unity Step Response

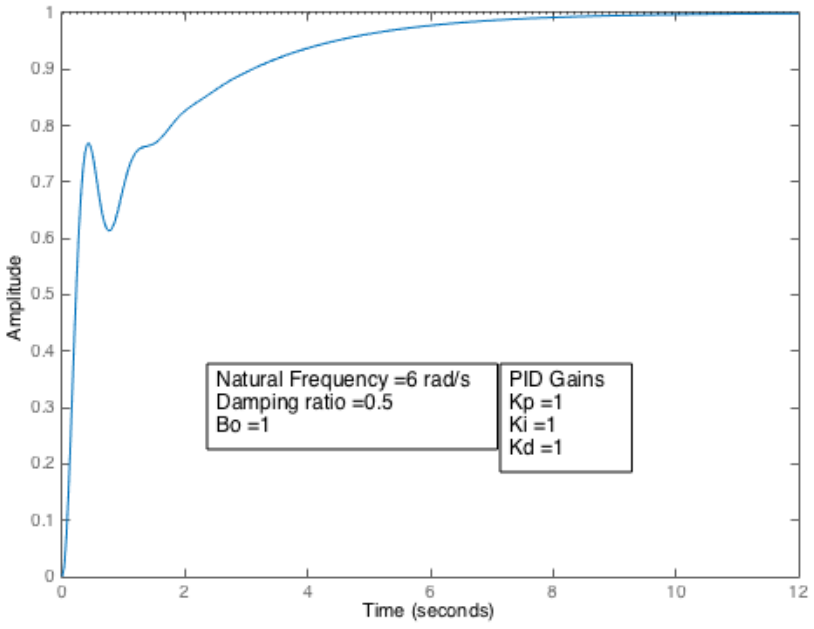


Figure 14: 2nd Order System Closed Loop Unity Step Response

In Figure 13 it is visible that an underdamped system, with a low value of damping ratio, will exhibit oscillatory behaviour when subjected to a step input. Contrasting that, Figure 14, indicates the response of a system with a higher value of damping ratio and natural frequency. The system does not

exhibit oscillatory behaviour, does not overshoot and reaches its steady state in less time.

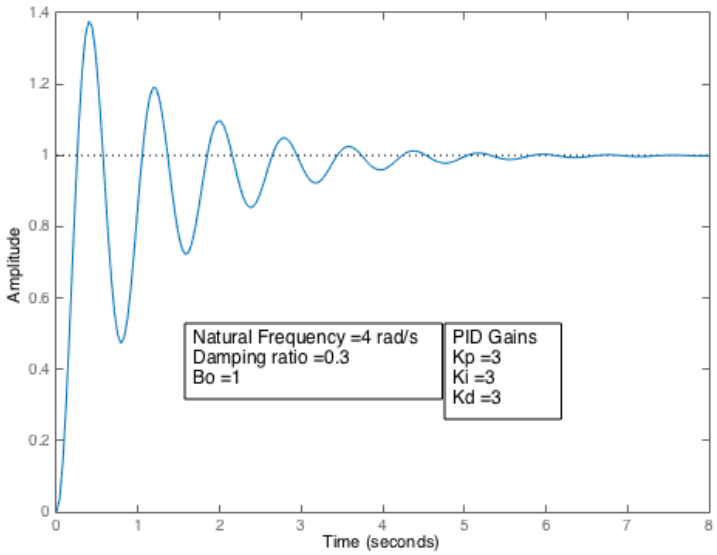


Figure 15: 2nd Order System Closed Loop Unity Step Response

In Figure 15, the characteristics of the system were kept the same as in Figure 13, but the gains of the PID controller have increased to a higher value. The system overshoots and the amplitude of the oscillations increases. It can be deduced then that for an overshooting & oscillating system, the value of the PID gains should be reduced.

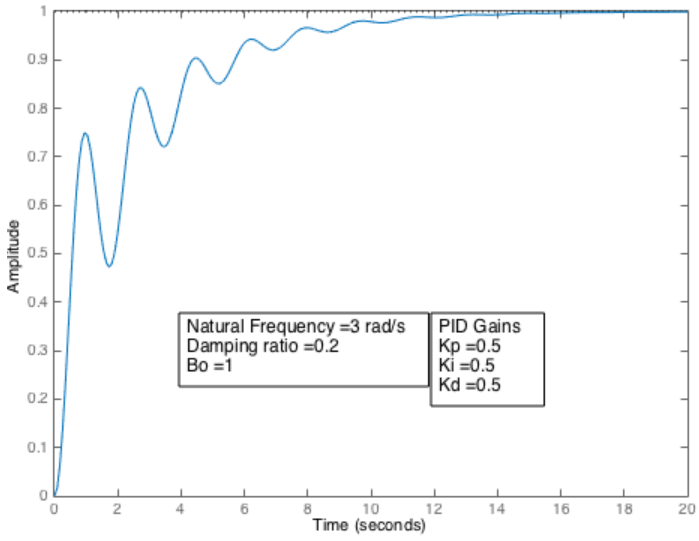


Figure 16: 2nd Order System Closed Loop Unity Step Response

The previous assumption is verified in Figure 16. The gains have been reduced to a value of 0.5 and the system is no more overshooting, but still exhibiting some oscillatory behaviour.

In Figure 17, the gains have been further reduced and the oscillatory behaviour has disappeared. The response obtained in Figure 17 is close to the ideal response expected from a 2nd order system. This knowledge will be used in the next chapter, where an autotuning algorithm will be implemented based on the insight obtained in this chapter.

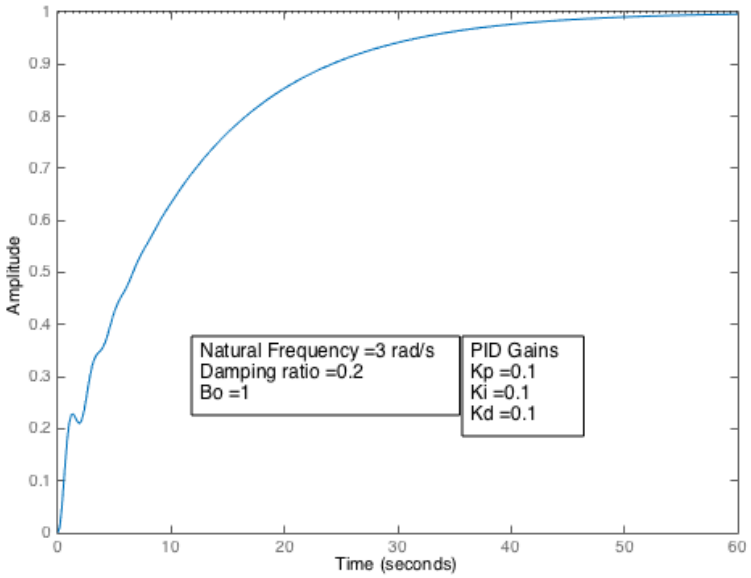


Figure 17: 2nd Order System Closed Loop Unity Step Response

5 Algorithm implementation

5.1 Introduction

The selection of an autotuning method for a second order system was one of the most difficult decisions of this project. Based on the literature review, classical linear tuning methods are not appropriate for UAV systems. The nonlinearities of such systems, but also other considerations like the control background of UAV operators, render techniques like Ziegler-Nichols impractical for UAVs. On the other hand, heuristic methods seem to be more suited for use in UAV systems. A few researchers (Fadil et al. 2013) have tried to implement heuristic algorithms for tuning PID controllers of UAVs but the results they obtained were not satisfactory. However, other researchers have been able to tune industrial PID controllers using heuristics methods (Kishnani et al. 2014).

In this project, a hybrid metaheuristic algorithm has been developed, which adjusts the gains a of PI controller until the response of the system is deemed satisfactory. The algorithm is a modified version of a Genetic Algorithm (GA), which is a family of algorithms for addressing optimisation problems. The implementation of the autotuning algorithm was based on the insights obtained from the analysis of 2^{nd} order systems. The algorithm focuses on PI controllers, for two reasons. Firstly, most UAVs can safely fly with only a PI controller (Szafranski & Czyba 2011). The derivative term is many times ignored as it causes the system to oscillate, due to noisy measurements. The second reason is that the introduction of the derivate term in the controller turns the PID tuning problem to a multi-dimensional

optimisation one, which was beyond the scope of this project. The decisions taken by the algorithm depend entirely on the initial excitation (unity step input in the experiment) and the response of the system. The algorithm has no knowledge of the system's dynamics (transfer function) or any other nonlinearities of the plant (e.g. actuators saturation). Hence, the autotuning algorithm can be implement in systems where the classical control tuning methods would not be applicable. The algorithm was coded in Matlab.

5.2 Algorithm Basics

An initial guess of proportional and integral gains is made and a set of gains (population) is created. Then, the response of the system to each pair of gains of the population is recorded. The signal recognition algorithm examines the responses of the system. If one of the responses is satisfactory a control statement terminates the execution of the loop and prints the current gains.

In the case that the response is not appropriate the system enters the tuning function, which updates the population, providing new inputs to the controller at the start of the loop. The calculation of the gains of the new population is based on the best performing elements of the previous population. This iterative process continues until a satisfactory set of gains has been found.

5.3 Signal Recognition

5.3.1 State Diagram

The first element that was required from the algorithm was to be able to process the response of a dynamic system and assess if it is the appropriate

one (i.e. the gains of the PID controller are tuned). A signal recognition algorithm has been developed for that purpose. The signal recognition algorithm is summarised in Figure 18.

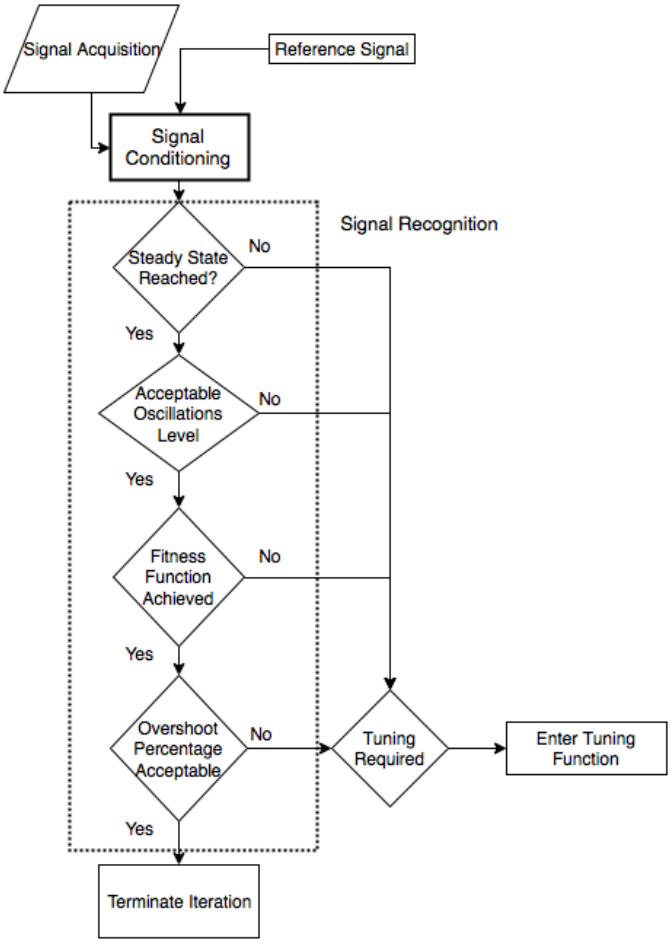


Figure 18: Signal Recognition Algorithm

The algorithm will enter the tuning function if any of the decision blocks outputs a False Boolean value. The tuning function has access to the outputs of the signal recognition process in order to tune the appropriate gains. If the output of the signal recognition process is True (i.e. the response of the system is acceptable) then the iteration is terminated and the current gains printed to the screen.

5.3.2 Assessment Criteria

The criteria used to assess the response of a system when it is subjected to a unit input are:

Table 4: Signal Analysis Criteria

1) System reaches steady state?
2) Is the system's response oscillatory?
3) If yes, what is the amplitude of the oscillation?
4) How fast is the system responding to the input?
5) Is the system overshooting or undershooting?

If all the criteria mentioned in Table 4 are satisfied, then the algorithm terminates and outputs the gains that produced the appropriate response. It is worth noting that of all the criteria listed, the speed of the response (4) is the most difficult to assess, as will be described later.

5.3.3 Steady State Test

Of all the criteria, the easiest one to assess is the final value of the system. Since the response of the system to the step input is known, it is only required to query the final value of the response and test if it is within $\mp 5\%$ of the steady state value. Matlab has a built-in function that returns the steady state of a system. However, for purposes of portability to other programming language, the code (Appendix A) estimates the average of the last 10 values and then compares that estimate with the step input. If a steady state error is

present in the system (i.e. steady state value different from reference input) then the program enters the autotuning routine.

5.3.4 Oscillatory behaviour Test

Testing the oscillatory behaviour of a system is a less straightforward task that requires a fair amount of signal manipulation. Using as an example Figure 15, it is visible that the oscillatory behaviour is described by the existence of peaks and troughs. The built-in Matlab function *findpeaks()* was used to identify the local peaks. Matlab has no built-in functions for identifying troughs. However, by changing the sign of the signal, the local troughs of the original signal became the local peaks of the new reversed signal and it was possible to identify them using the same function.

The next step was to count the number of peaks and troughs. If the number of peaks is higher than 3 and the number of troughs is higher than 2 that indicates an oscillatory behaviour. If an oscillatory behaviour is detected, then the magnitude of the oscillation is measured. The oscillation does not usually occur about a specific point. Hence, the magnitude was considered to be the half distance between the first peak and the first trough. Using the information about the number of oscillations and the magnitude of those oscillations a value between one (1) to five (5) is assigned to the variable *tuning_level*. A higher value of that variable indicates a persistent oscillatory behaviour, whereas a low value describes a more benign oscillation. This information is used in the autotuning routine.

If a system is undergoing a decreasing amplitude oscillation it is possible to estimate the logarithmic decrement using the value of each local peak and

the time difference between the peaks. The logarithmic decrement can then be used for the calculation of the damping ratio and natural frequency. Although the damping ratio and natural frequency are not used in this project for the tuning of the gains, a future project could approach the problem of autotuning from a system identification perspective, making use of the estimated damping and frequency values.

An alternative approach to the estimation of the oscillatory behaviour of the system is the calculation of the derivatives of the signal function at each data point. Using a central difference scheme (Equation 17), it is possible to numerically estimate the derivative of each point. Then, the percentage of negative-valued derivatives can be compared to the positive derivatives and using correlations from known signals, an approximation can be made about the oscillations. The main issue associated with this technique is high frequency noisy measurements that are present in feedback signals. A low-pass filter is necessary to block the noisy measurements. The derivative value of the signal is not used in the current version of the autotuning algorithm.

$$\frac{dy}{dx} \cong \frac{y(x+b) - y(x-a)}{b-a} \tag{17}$$

5.3.5 Fitness Function (Speed of response)

As mentioned earlier, the speed of the response is the most challenging criterion to assess. The rise and settling time is a unique property of each system. Imposing a global limit on the rise time would be unrealistic. Hence,

a completely different approach was used to assess the response time. A key concept of Genetic Algorithms, described in chapter 5.4.2, is that of a fitness function. A fitness function can be thought of as a figure of merit, which shows how close a system is to its ideal conditions. In this study the fitness function was chosen to be the sum of the vertical distance (y_i) of each data point of the response signal from the reference signal. In the case of the unit step input experiment (6.3), the fitness function was adapted to be the sum of the vertical distances of all data points, between $t = 0$ and $t = t_r$ (rise time) of each response. The purpose of that was to maintain the same benchmark for all tested systems. The fitness function is given by Equation 18.

$$f = \sum_{k=1}^{n} y_i \tag{18}$$

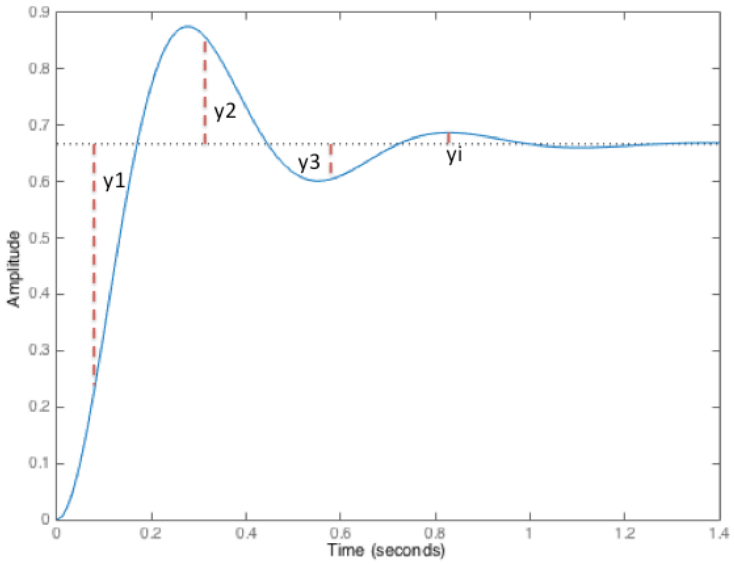


Figure 19: Fitness Function Graphical Representation

Figure 19 is a graphical representation of the fitness function. The value of the fitness function depends on the speed of the response and the oscillatory behaviour of the system. A system with fast response and small oscillations about the set point will have a fitness function with a small numerical value. Accordingly, a slow-moving system will have a fitness function with a higher value, since it will take more time to reach the steady state. Comparing the fitness functions of two responses is an easy task and requires no additional knowledge about the system, except from the response and the initial set point. The concept of the fitness function will become clearer when the tuning algorithm is described in the next chapter.

An additional way to the test the speed of the response is to measure the time ratio between the settling time and the rise time. If that ratio has a high value it implies that the system reaches its steady state value slowly and a new set of gains is required.

5.3.6 Overshoot

An overshooting system does not necessarily imply a system with inappropriate gains. In some systems, a certain level of overshooting is considered acceptable. Hence, in the Signal Recognition algorithm it was decided that the magnitude of the overshoot will be used as a test to establish if gain tuning is required. If the overshoot is more than 40% of the step input, then the tuning flag will be raised and the system will enter the tuning routine. The choice of the 40% value is based on the experimental data obtained in the analysis of 2^{nd} order systems (Chapter 4.4).

The case of severe undershooting is addressed by the previous criteria of steady state and fitness function testing.

5.4 Gain Tuning

5.4.1 Introduction

When the output of the signal recognition algorithm indicates that the assessment criteria have not been satisfied, then the system will enter the tuning segment of the algorithm. The aim of the tuning algorithm is to produce a set of gains that satisfy the required response characteristics. The gains are not updated while the system is responding to an input. They are updated based on the response of the system and then tested against a new unit step input. The tuning algorithm follows a metaheuristic approach and is a modified version of a Genetic Algorithm.

5.4.2 Genetic Algorithms

Genetic Algorithms (GA) stem from the principle that evolutionary theories can be used to tackle hard optimisation problems and can be thought as an application of the Darwinian theory in computer science. Originally developed by Holland (1975) they have been used in a variety of global minimum search problems.

In a GA, an initial guess of possible solutions is made. This set is called the population. The population is then tested against the problem in question to confirm if any of the population elements are solutions to the problem. If none of the results obtained from the initial population is a solution to the problem (which is the most common case), then a new population is created. The new

population is based on the performance of the different members of the initial population. The new, evolved, population (offspring) is the result of the mutation of the best-performing elements of the initial population. In order to assess the performance of each element of the initial population a fitness function is used. Elements with a ‘better’ fitness function have a higher chance of being selected for mutation and passed to the new population.

The new population is then plugged in to the problem and an updated solution is generated. The GA algorithm iterates and creates multiple populations, until an element of a population produces a satisfactory solution.

GAs fall in a category of algorithms termed metaheuristic. The main difference of a metaheuristic algorithm, when compared to a heuristic one, is that it can be applied to any optimisation problem. Heuristic algorithms are usually problem-specific. Hence, by using a metaheuristic algorithm it is possible to harness the knowledge and experience of many other researchers and experiments.

A more detailed introduction to GAs can be found in Mitchell (1996).

5.4.3 GAs for PI Autotuning

5.4.3.1 Algorithm Flowchart

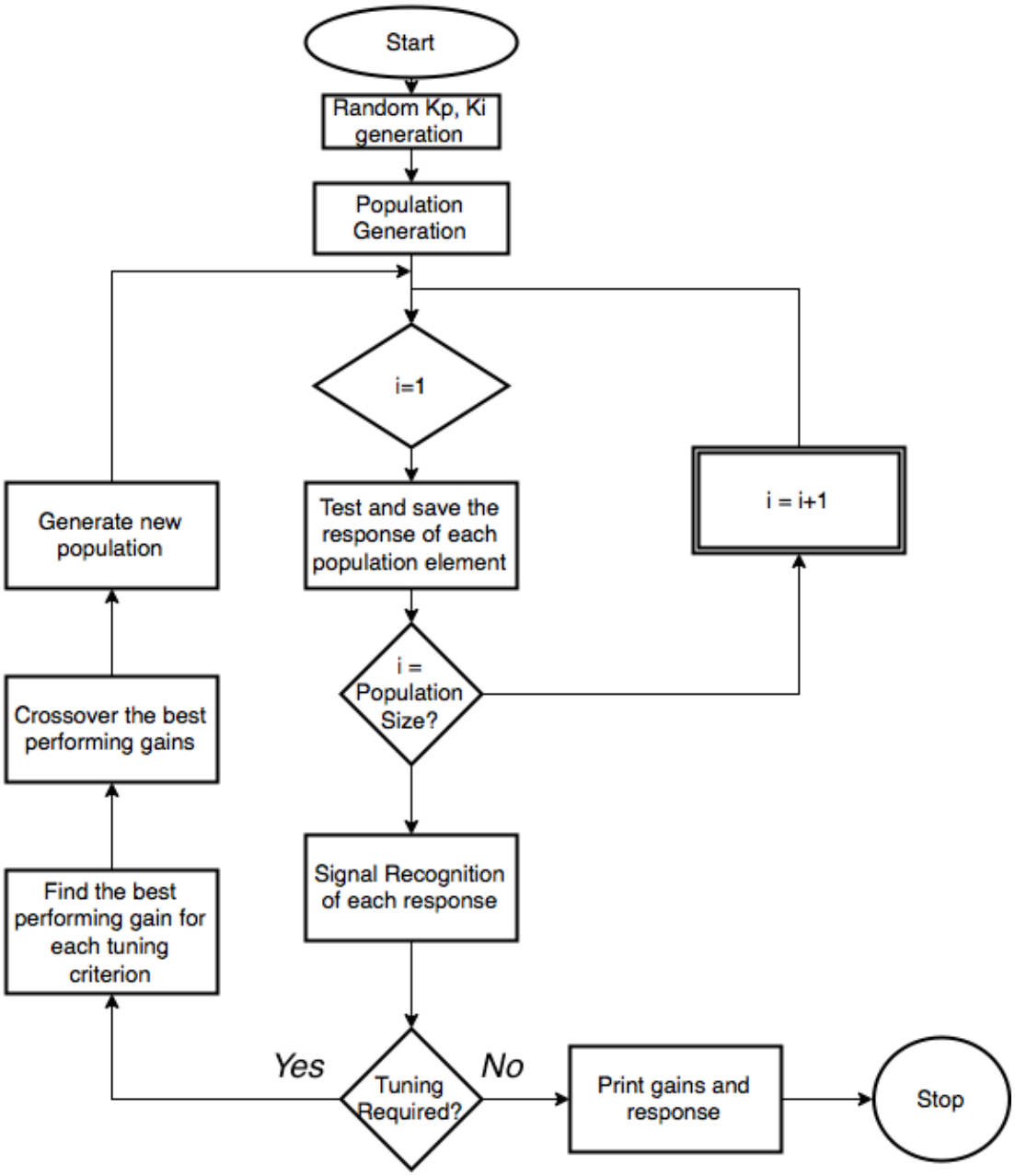


Figure 20: Autotuning Algorithm Flowchart

5.4.3.2 Population Initialisation

Having briefly examined GAs, it is possible to apply similar principles for the tuning of a PI controller.

The first step is the selection of an initial population. The solution space of the PI tuning problem is theoretically all positive proportional (k_p) and integral (k_i) gains. However, from chapter 4.4.4 it became apparent that gains higher than 4 will most likely have destabilising effects to the system. Hence, two random k_p, k_i variables are chosen, with a maximum value of 4. The maximum initial value of the gains can be updated depending on the system that is tested.

With the first pair of gains generated, the next step is to create the initial population. The main goal of the initial population is to check if a higher (or lower value of gains is required. Six additional sets of $k_p \; and \; k_i$ gains are required to check all possible permutations. That brings the total population to a set of 7 pairs of gain.

Using the first pair of k_p, k_i values, a PI controller is created and the response of the system to a step input is recorded. The initial response is used for comparison with the final tuned response.

5.4.3.3 Iteration Stage

In the next stage, the algorithm enters an infinite *while loop*. The while loop terminates only if an appropriate response (i.e. tuned gains) is obtained from the system or if after 50 iterations no solution has been found. The 50 iterations limit has been placed in order to prevent the computer from 'crashing'.

Then, an inner *for loop* iterates over the seven pairs of gains. The response of each pair of gains is recorded and assessed by the criteria described in

the Signal Recognition Chapter (5.3). The output of the signal recognition process is four arrays containing information about the response obtained from each population element. The first one is a Boolean array (*tuning_rqrd*), which indicates if a pair of gains requires tuning. The second is the array of fitness function values of each element. A *for loop* iterates over the elements of the fitness function (Chapter 5.3.5) array and checks if the sum is less than a specified number for each set of gains. The choice of the specified number depends on the type of tuning a user is after. A more aggressive tuning process requires a low value number, roughly 10-20. A more reasonable tuning is obtained with values between 20-40. The estimation of those numbers was based on the sample 2nd order responses in Chapters 4.4.2 - 4.4.4. If the fitness function of a pair of k_p, k_i gains has a higher value than the specified number, then additional tuning is required and the corresponding element of the *tuning_rqrd* becomes True.

The third one is the *tuning_level* array, which indicates the intensity of the oscillations in each response.

The final array contains the overshoot values for each element of the population.

It is worth noting here that the fitness function could be used as the sole tuning criterion. This is the case in the Reference Tracking experiment in section 6.2.

The next step is to sort the *tuning_rqrd* array. It contains only Boolean elements, which in Matlab can be expressed as 0 or 1. Sorting the array is a way to examine if a pair of k_p, k_i gains has satisfied all tuning criteria (i.e. the

value of *tuning_rqrd* equals to 0). If an element of the *tuning_rqrd* is zero, it means that an optimal solution has been found and the iteration terminates, printing the final gains to the screen. If no elements in the *tuning_rqrd* are zero though, then a new population of gains is produced. This is the actual gain-tuning phase of the algorithm.

5.4.3.4 Tuning

In the previous step of the algorithm, the arrays containing the performance of each pair of gains were sorted in an ascending order, with the best performing pair occupying the first element of the array. However, each array describes a different characteristic of the response. The fitness function indicates the speed of the response, whereas the *tuning_level* array indicates the amount of oscillations in the response. It is very common though, for systems to give conflicting requirements. For example, systems that might have a very low fitness function value might fail the tuning criteria because of heavy oscillations and vice versa.

A compromise is needed in order to select the best population of gains for the next iteration of the algorithm.

Following the principles of GAs, the algorithm records the set of gains that produced the optimal response for each criterion. That still retains 3 pairs of gains from each criterion (fitness function, oscillations level, overshoot). In order to select only one set of gains to be passed on to the next population the concepts of crossover and mutation were used.

The new set of k_p, k_i is the weighted sum of the gains that gave the best response for each criterion. In equation form,

$$K_p = COR_1 * K_p(fitness) + COR_2 * K_p(oscillations) + COR_3 * K_p(Overshoot \quad (19)$$

Where, $COR = Crossover\ Ratio$ and $COR_1 + COR_2 + COR_3 = 1$

Similarly,

$$K_i = COR_1 * K_i(fitness) + COR_2 * K_i(oscillations) + COR_3 * K_i(Overshoot \quad (20)$$

The crossover ratios are adjustable by the user and depending on the values of those variables, the new population will inherit different characteristics. If COR_2 has a high value, then the new population will produce gains that result in low oscillations, but this is going to be at the expense of speed and overshoot. A compromise has to be made between the different characteristics that the new population will inherit.

After the new set of gains is created the population is mutated, as in the case of the initial population, with the goal of creating a diverse population encompassing all possible combinations of gain values.

With the new population created, the program continues iterating until an acceptable set of gains has been found.

6 Results & Discussion

6.1 Introduction

After the design and implementation of the autotuning algorithm a series of experiments were set up to assess the performance of the method. The first experiment (section 6.2) focused on reference tracking and involved a sinusoidal excitation to the system. The main objective of the first experiment was to confirm the suitability of Genetic Algorithms for use in autotuning algorithms of 2^{nd} order systems. Thus the tuning of the PI gains was based only on the value of the fitness function.

The second experiment (section 6.3) included the full version of the autotuning algorithm described in section 5.4. The excitation to the system was provided by a unit step input.

It is worth noting here that the tuning criteria used in the two experiments were not the same. Hence, a direct comparison of the two sets of results would not be appropriate.

In both cases the autotuning algorithm performed adequately and was able to find an acceptable value for the gains of the controller. The gains generated by the algorithm did satisfy the initial tuning criteria. However, as it will be shown in the following chapters, the gains generated by the algorithm are not necessarily the optimal ones.

The initial gains used in the autotuning function were numbers generated from a random number generator. One of the main goals of the experiments was to assess the robustness of the algorithm to abnormal initial conditions.

6.2 Reference Tracking Experiment

6.2.1 Experimental Setup

The first experiment involved a 2nd order system that was subject to a sinusoidal wave input. The experiment was performed in Matlab and Simulink. As it can be seen on Figure 21, the unity feedback system consists of a 2nd order system and a PI controller, which accepts values for the proportional and integral gains from the block 'Gains'. The block Gains is connected to the Matlab file executing the autotuning code. The autotuning algorithm has access to the responses of the system through the *simout* variable.

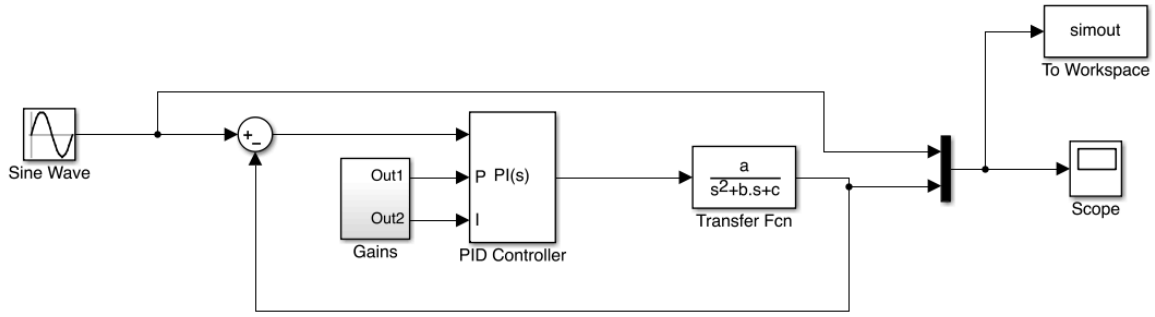


Figure 21: Block Diagram of Reference Tracking Experiment

6.2.2 Case 1

In Figure 22 the initial system response (red line) to the sinusoidal input (blue line) is shown. The response of the system is initially not sufficient for the system to reach its steady state value. However, the response of the system is sinusoidal.

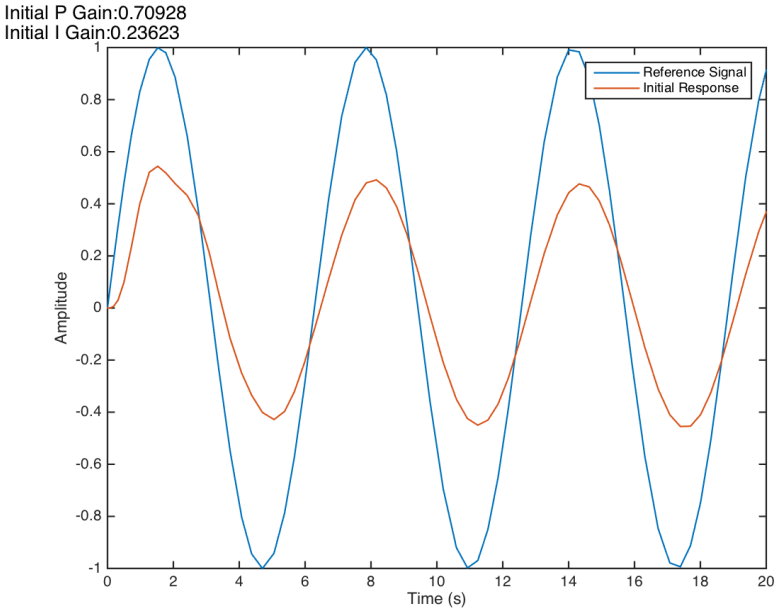


Figure 22: Initial system response

In Figure 23 the final, tuned, response of the system is presented. The algorithm has adjusted both the proportional and integral gains and the system tracks the reference input almost perfectly. A small amount of oscillatory behaviour is observed around the first peak of the signal, but the response curve becomes smoother in the rest of the signal. This is due to the large initial error that goes in to the PI controller.

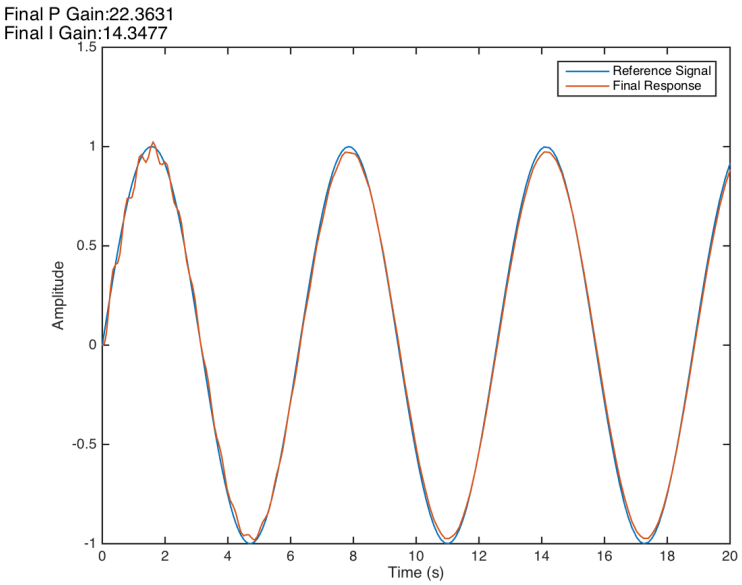


Figure 23: Final system response

Figure 24 shows all attempts of the autotuning algorithm to obtain the optimal response from the system. As defined in section 5.3.5, the fitness function is the sum of the vertical distances between each system response point and the corresponding reference point. The GA algorithm trials a variety of responses, though the use of different sets of gains, and the gains that yielded the minimum fitness function from each population are used for the production of the new population. It can be seen that with each attempt the algorithm approaches the reference signal. It is interesting to highlight that although some of the responses were diverging away from the reference signal, the algorithm was able identify and reject these behaviours.

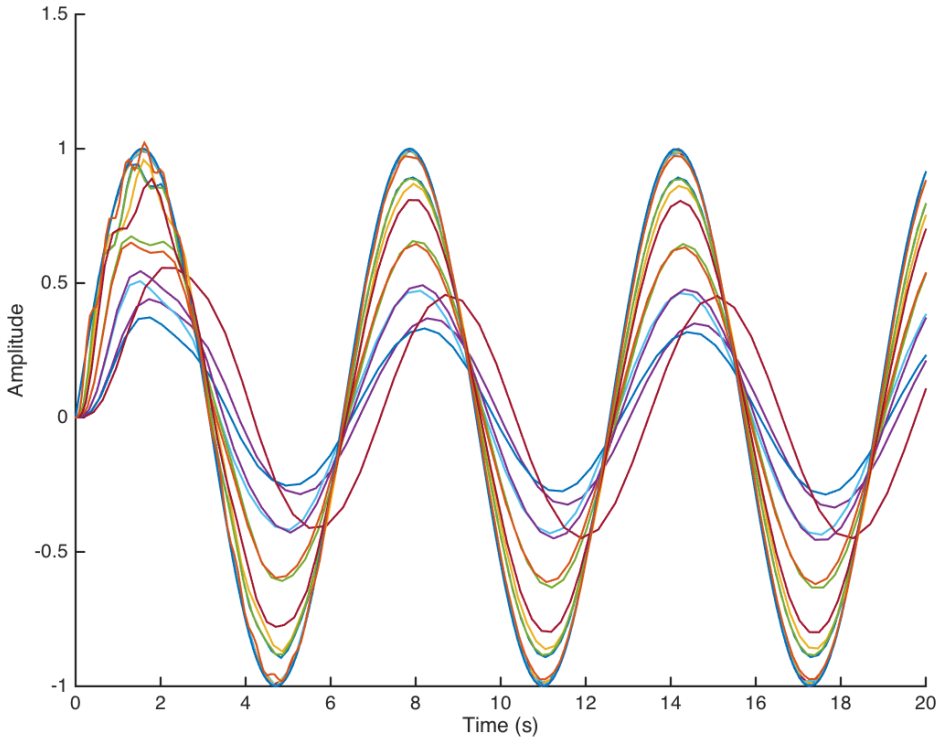


Figure 24: Intermediate system response

Figure 25 indicates the variation of the fitness function versus iterations number. As it can be seen on the x-axis, the algorithm was able to reach a converged result within only 3 iterations. This indicates a good

response and convergence to the appropriate value of gains, which is something desirable for a UAV system. An autotuning algorithm for a commercial UAV system is judged not only on its technical accuracy, but also on other practical considerations, like the time required to find an appropriate set of gains.

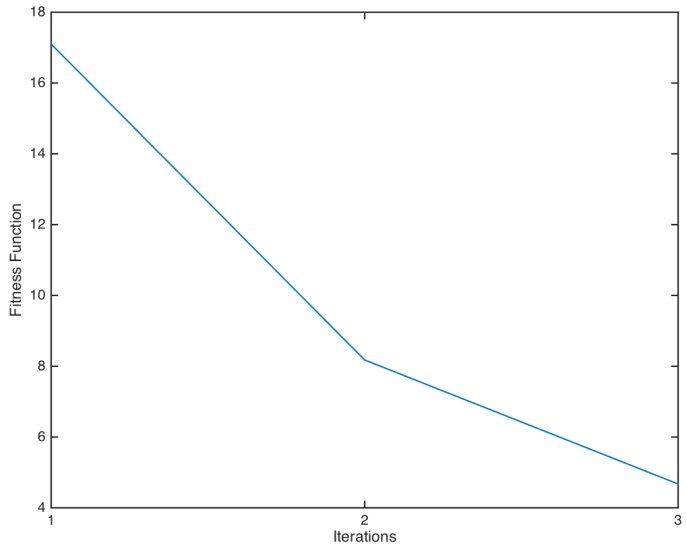


Figure 25: Fitness function variation

Similarly, Figure 26 shows the variation of the proportional and integral gains. In this case, the initial gains were too small. It is worth noticing the rate of change of the gains is a lot higher in the second iteration, when compared to the first one. This is related to the method that the new population is selected. The initial population is created from two random variables used for the k_p, k_i gains. Then, those random variables are multiplied with user-defined numbers in order to produce the population. Contrary to that, the secondary populations use as a starting point the best performing elements of the initial population and then create the remaining elements by multiplying the parent elements with random variables. This randomness, over which the user has no control, creates a space of solutions that encompasses

combinations of gains that a user would not be able to define. This trend is visible in the rest of the results as well.

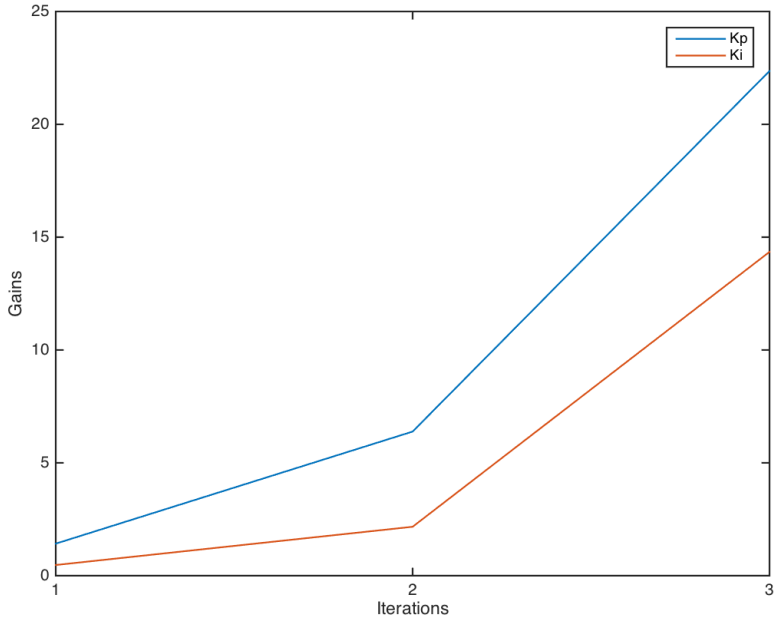


Figure 26: Gain variation

6.2.3 Case 2

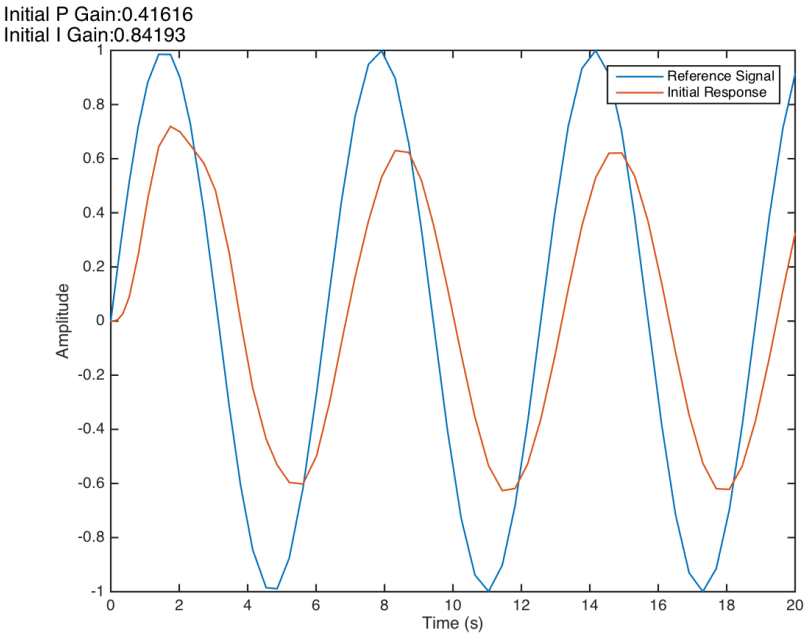


Figure 27: Initial system response

In the second case the same system (same transfer function) is subjected to a different set of initial gains. The response, as seen in Figure 27, is different from the response in Case 1. The most interesting thing to notice in this graph is that the system's response is out of phase with the reference signal. However, the autotuning algorithm was able to cope with this difficulty. In Figure 28, the tuned system response is shown to be tracking the reference signal adequately. The initial oscillation of the signal around the first peak, observed in Case 1, is present in this case as well. It is worth highlighting here that the gains obtained in Case 1 and Case 2 are not identical. This is expected, as the fitness function is not satisfied by a single value of gains, but by a range of gains. The proportional gain is in the same vicinity in both cases, whereas the integral gain is ~42% less in Case 2.

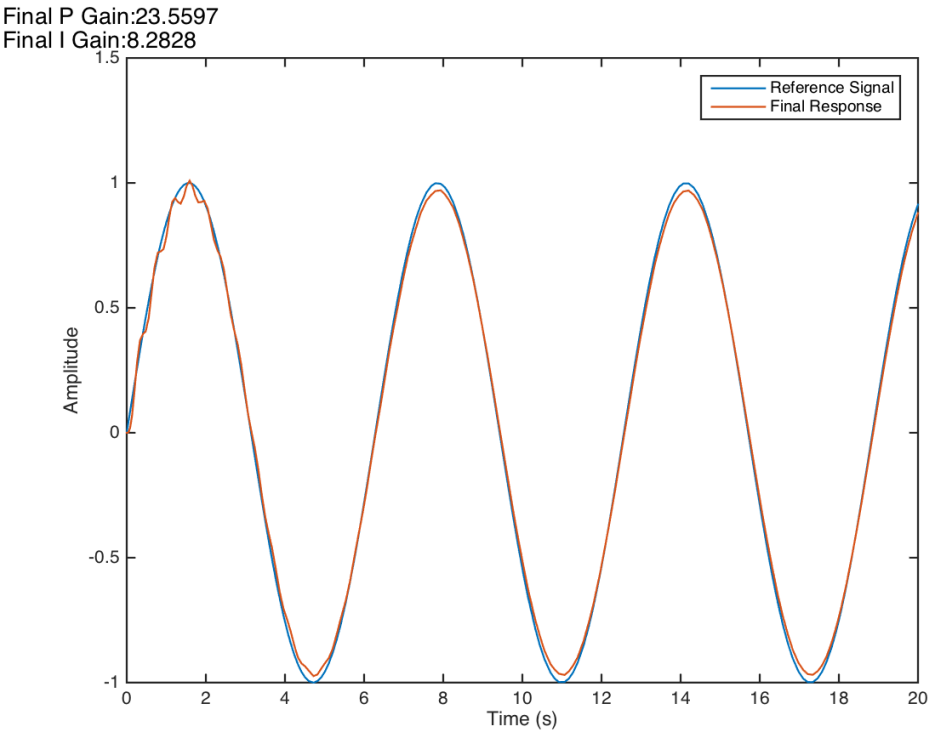


Figure 28: Final system response

Figure 29 indicates a worrying behaviour from the intermediate values tested by the algorithm. A particular set of gains has pushed the system to instability

and one of the responses overshoots to such a high value that makes all other responses unnoticeable on a graph of that scale. Obviously, this behaviour is highly undesirable in real UAVs, as it would probably lead to a crash. Hence, controls should be in implemented that stop the execution of the algorithm if one of the elements of the population produces an unstable set of gains.

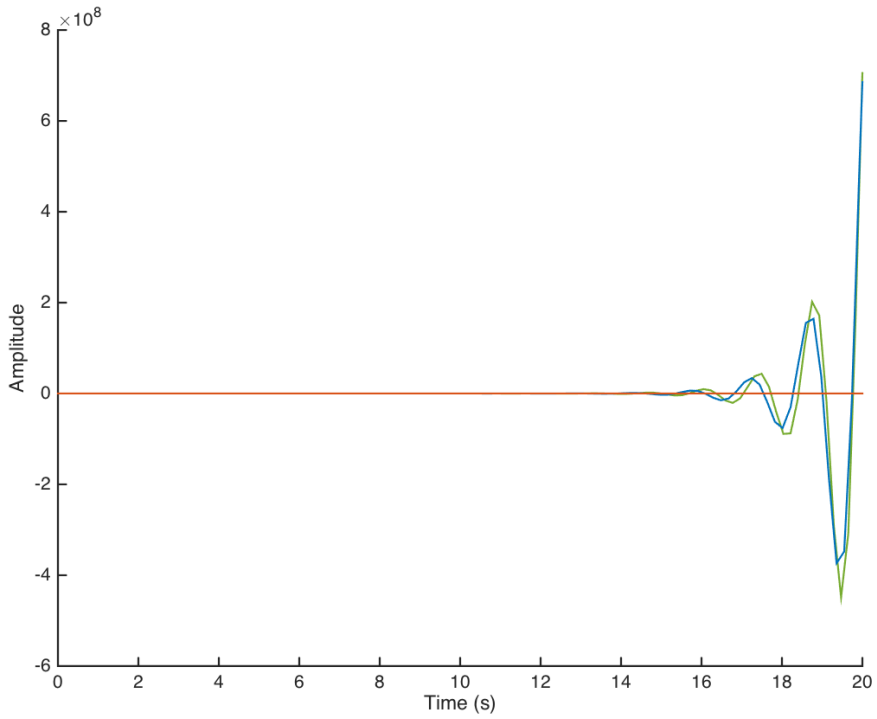


Figure 29: Intermediate system response

However, the fact that a converged solution was achieved is a testament to the robustness of the algorithm. Even though one of the responses produced an unbounded solution the algorithm was able to reject the unwanted value.

This is also clearly seen in Figure 30, where the value of the fitness function is initially increasing but then starts reducing until it satisfies the user-defined criterion. One would normally expect the value of the fitness function to be infinite for the value of the gains that gave the unbounded solution in Figure

29. However, the algorithm retains only the lowest fitness function from each population and completely ignores all other values.

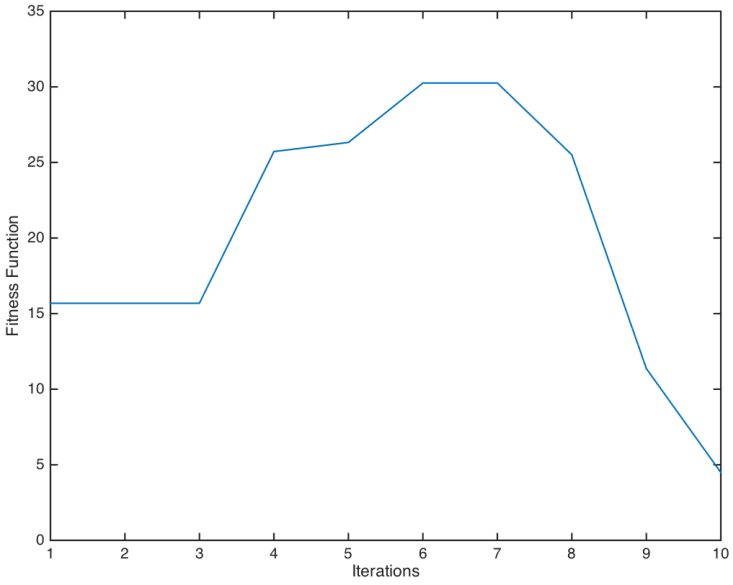


Figure 30: Fitness function variation

Finally, Figure 31 shows the variation of the gains. In this experiment, 10 iterations were required to reach the converged solution. This is higher than the number of iterations required in Case 1 and is related to the initial conditions. If this code is ported to a UAV, great care must be taken in order to ensure that the initial conditions (initial guesses) are appropriate for the system.

Between iterations 4 and 8 it is visible that the value of gains remained almost the same, with little variation. This is a behaviour observed in some earlier experiments and was a root of problems, as the gains would get stuck in a local minimum and not be able to find the global minimum. This algorithmic risk was mitigated with the introduction of a control statement before the selection of the new population. If the elements of the new population were the same as the elements of the previous population, then

all population elements were multiplied with a random ‘wildcard’ variable in order to enable the algorithm escape the local minima well.

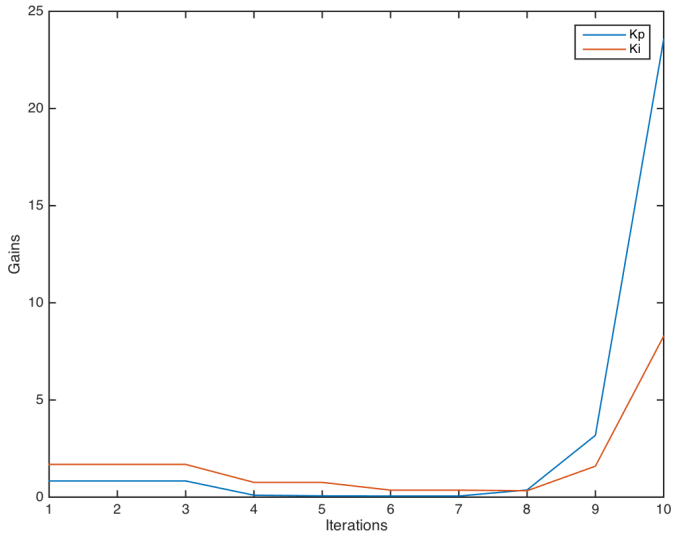


Figure 31: Gain variation

6.2.4 Conclusions of the experiment

Examining the results obtained from the first experiment the following can be concluded:

- Genetic Algorithms can be used to tune the gains of a controller.
- The initial conditions are critical to the accuracy and speed of the tuning process.
- The use of randomised population elements leads to faster tuning (less iterations).
- The population could end up getting stuck in a local minimum. In order to reach the global minimum appropriate methods should be introduced that update the values of the population.
- The heuristic tuning process can yield gains that destabilise the system. Hence, on a UAV controls should be in place to terminate the tuning process in that scenario.

- On a UAV, the fitness function would have to be modified and account for noisy measurements from the sensors

6.3 Unit Step Input Experiment

6.3.1 Introduction

In the second experiment the full version of the autotuning algorithm was tested. The response of various 2nd order systems was examined against an initial random set of gains and then those gains were adjusted until a satisfactory response was obtained. The dynamics and response of the system were simulated within the Matlab file containing the algorithm. In all of the following cases, a constant derivative gain ($k_d = 0.1$) was introduced, which was not updated by the autotuning algorithm. The purpose the derivative control was to assess the performance of a full PID controller. The gain was kept constant and at a low value, as in most UAV-related applications, the derivative term sometimes introduces unwanted oscillations and is frequently omitted.

6.3.2 Case 2.1

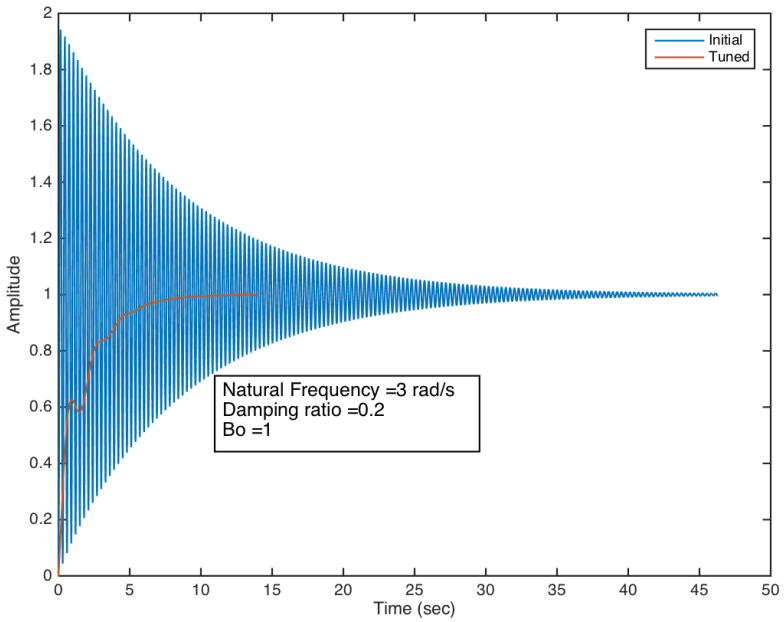


Figure 32: 2nd order system response

Figure 32 shows the initial (blue curve) and the final (red curve) response of a 2nd order system to a unit step input. The autotuning algorithm was able to tune the PI gains of the controller and produce an acceptable response. It is worth reiterating that the response obtained by the system is, probably, not the optimal one when compared to linear tuning techniques. However, the final response satisfies all of the user-defined criteria, hence it was considered acceptable.

Figure 33 shows that the algorithm required 7 iterations before converging to an acceptable set of gains. With each iteration the value of the proportional gain reduced, until it reached its final state.

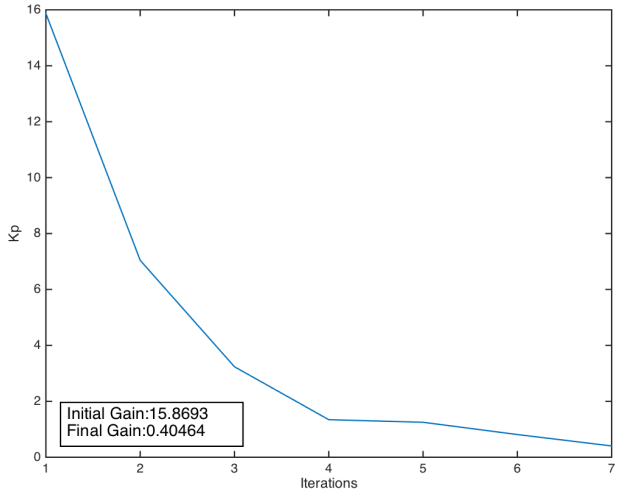


Figure 33: Proportional gain variation

The change between the initial P gain and the final one is ~97%. This is a significant change and the autotuning algorithm was able to find the optimal solution even though its location was outside the close range of the initial guess. This becomes more evident in Figure 34, with the integral gain reducing from an initial value of 75 to 0.6. This is a variation of 99% from the initial value.

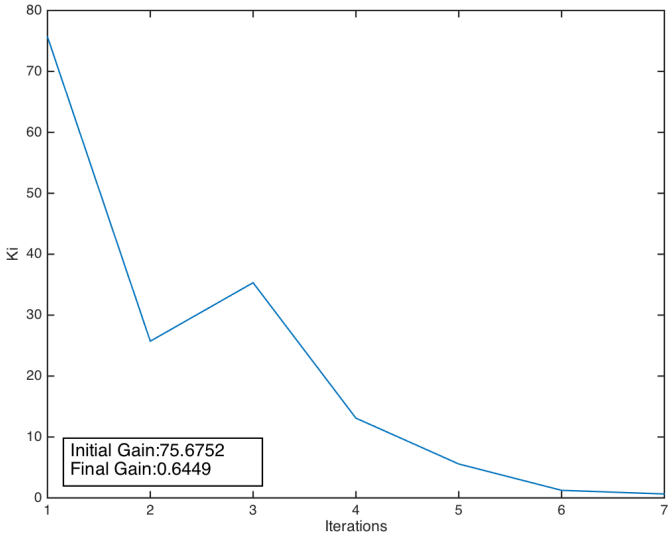


Figure 34: Integral gain variation

Additionally, in Figure 34 between iteration 2 and 3, the integral gain is seen to be increasing. The algorithm was able to recover from that wrong estimate and produce a converged solution.

6.3.3 Case 2.2

Case 2.2 presents a scenario with a big initial difference in the values of the proportional and integral gains. The focus of this experimental result is not so much on the actual response of the system, as to the way the PI gains change during each iteration. In Figure 35 the initial response is oscillatory and the overshoot of the system is over 80% from the commanded steady state value. In the tuned response, the oscillatory behaviour was almost completely eliminated and there is no overshoot. However, as a compromise, the setting time of the system has increased. This is a trade off that a user has to make when selecting different levels of autotuning.

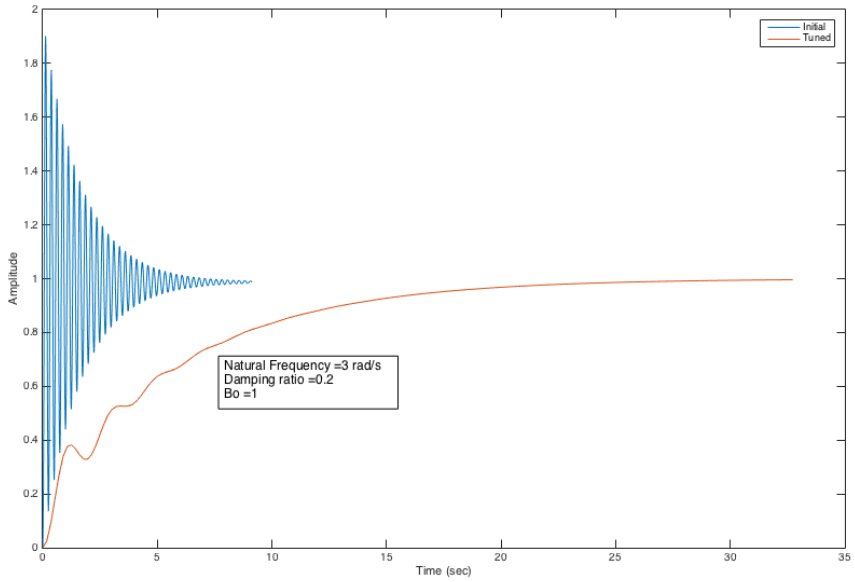


Figure 35: 2nd order system response

More interestingly, in Figure 36 and Figure 37 the number of iterations required to reach a converged solution has greatly increased, when

compared to Case 2.1. This is a seven-fold increase in the number of iterations

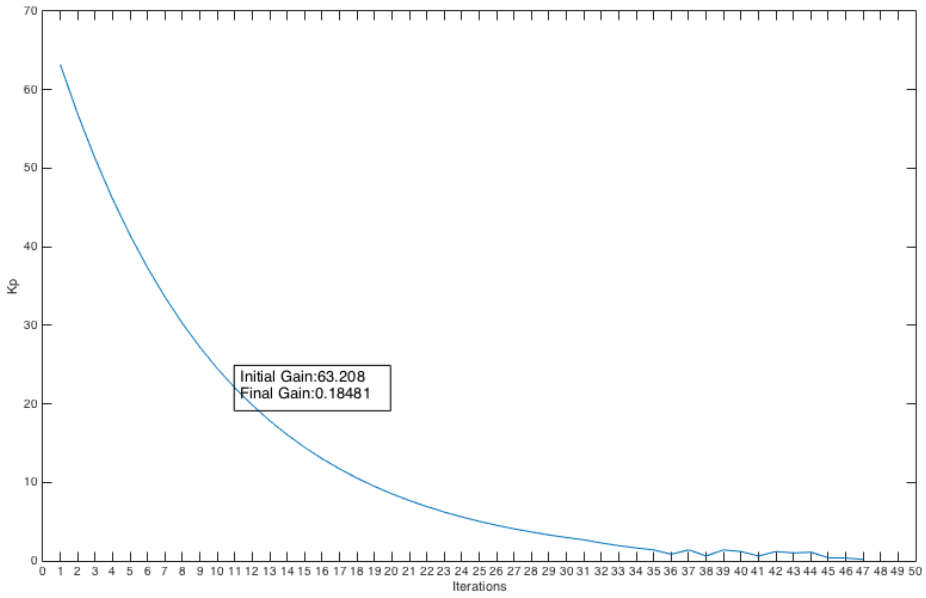


Figure 36: Proportional gain variation

This is attributed to large difference between the initial proportional and integral gains. The starting $k_p = 63.2$ whereas the starting $k_i = 0.687$. This is a variation that caused some problems to the autotuning algorithm. Initially the program tried to decrease the proportional gain, as expected. However, at the same time it tried to increase the integral gain. It is known by the analysis of 2nd order systems in chapter 4.4 that an increase in the integral gain can lead to further oscillations. Hence, the algorithm during the first couple of iterations was destabilising the system.

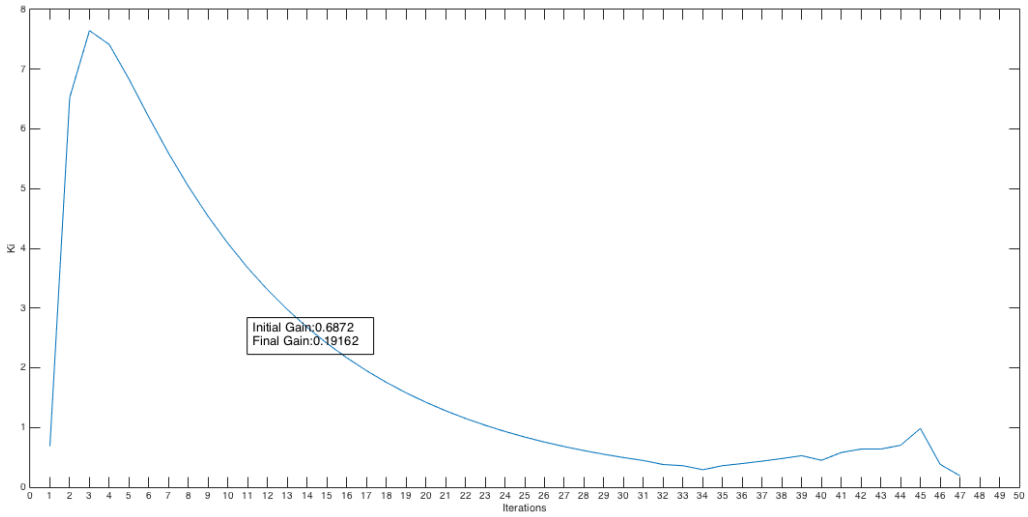


Figure 37: Integral gain variation

After iteration 4 the program was able to find the correct convergence direction. However, the variation in the gains was significant and 43 iterations were required before reaching an acceptable solution.

6.4 Population Generation

The algorithm generates a pair of random k_p, k_i gains that are used to create the initial population. As described in the previous chapters, the initial population is generated by multiplying the gains with predefined variables in order to ensure that all possible higher/lower combinations of gains are examined by the fitness function during the first iteration. The generation of the secondary population though, does not have that constraint and the best performing gains of the initial population can be multiplied with random numbers (Figure 38). Experiments with fixed initial gains were performed in order to assess if the use of random multipliers for the generation of the secondary populations would speed up the algorithm. Both versions of the population generation method are available in Appendix A.

```
% pre defined population generation
gains = [ kp ki;
      0.5*kp ki;
      kp 0.5*ki;
      0.5*kp 0.5*ki;
      0.5*kp 3*ki;
      3*kp   0.5*ki;
      3*kp   3*ki;];


 % random population generation
 gains = [ kp ki;
      rand*kp ki;
      kp rand*ki;
      rand*kp rand*ki
      rand*kp rand*ki;
      2*kp   2*ki;
      10*rand*kp   10*rand*ki;];
```

Figure 38: Population Generation Methods

In the first experiment the initial gains were $k_p = 3\ and\ k_i = 3$ and the secondary population was generated with predefined variables. The results of that case are summarised in Table 5.

Table 5: Fixed population creation, experimental results

Iterations until a converged solution	4
Kp (final)	1.296
Ki (final)	0.6533

In the next case, the initial gains were kept the same ($k_p = k_i = 3$) but the generation of the secondary population was allowed to vary with the use of a random number machine. Figure 39 shows the numbers of iterations required during each run of the experiment until all tuning criteria have been satisfied (i.e. converged solution).

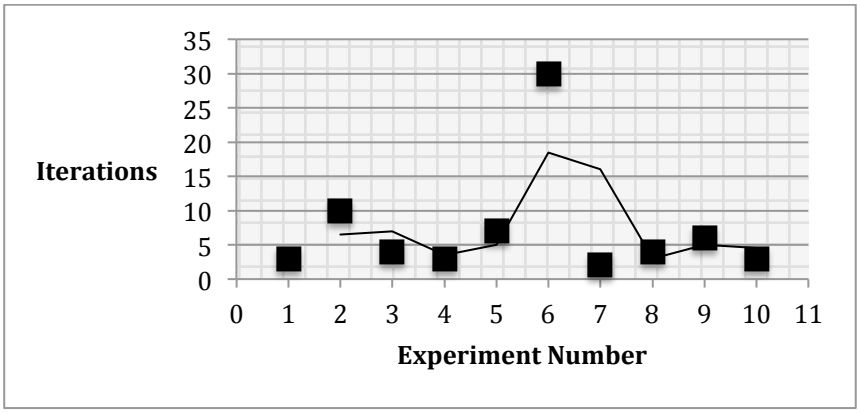


Figure 39: Iterations required for a converged solution

The average of all those values is 7.2 and the median is 4. However, the algorithm with the random population generator was faster than the fixed population generator in 40% of the experiments.

Figure 40 is also very interesting as it shows the variation of the k_p gains during each run of the experiment. Summarising the data, $k_{p(average)} = 0.97$ and $k_{p(median)} = 0.896$

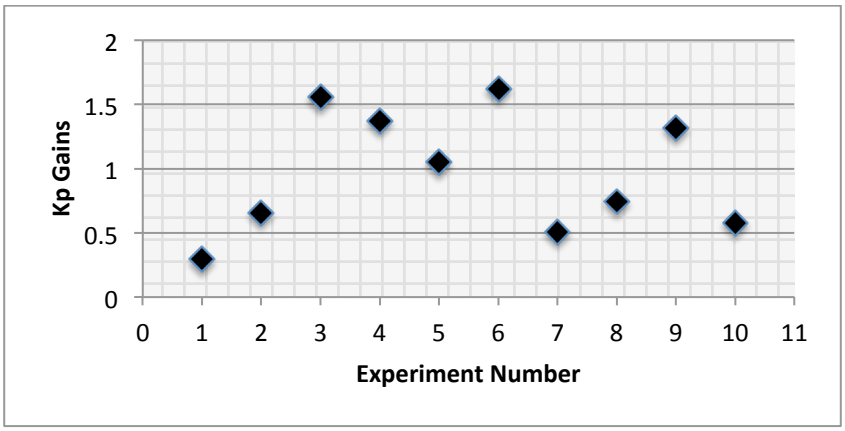


Figure 40: Gain variation in autotuning algorithm with random population generator

Similarly, in Figure 41 the variation of the k_i is shown. From the data, $k_{i(average)} = 0.7179$ and $k_{i(median)} = 0.7115$.

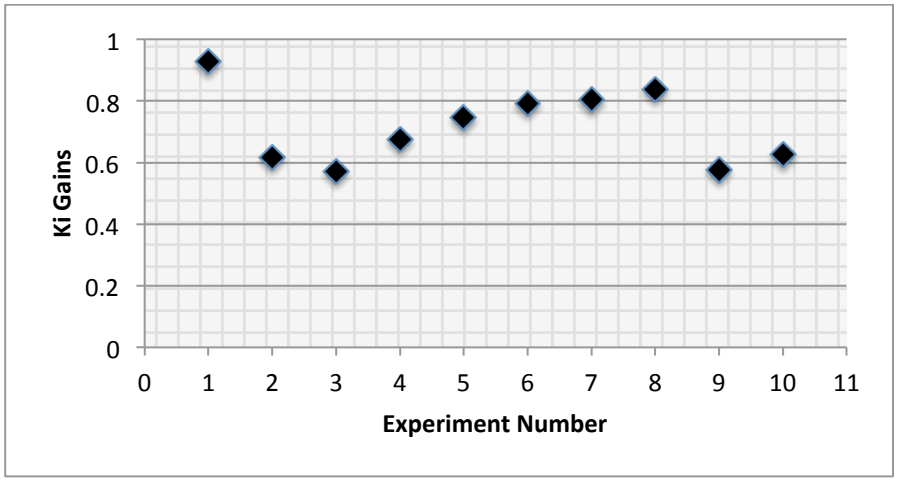


Figure 41: Gain variation in autotuning algorithm with random population generator

In the next experiment, the initial gains were set to $k_p = k_i = 10$.

For the case of population generation with predefined variables the results are:

Table 6: Fixed population creation, experimental results

Initial $\boldsymbol{k_p, k_i}$	10,10
Iterations until a converged solution	16
Kp (final)	0.9566
Ki (final)	0.7501

When the generation of the population was based on random variables the following results were obtained:

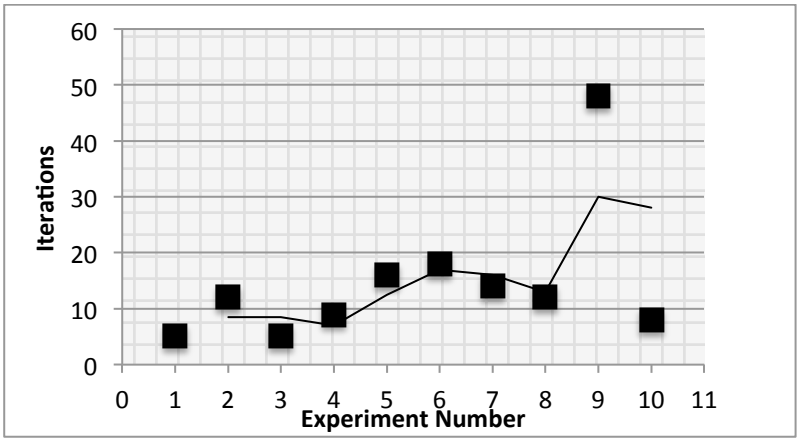


Figure 42: Iterations required for a converged solution

The average number of iterations required for a converged solution is now 14.7 and the median is 12. This is a 25% increase in speed of the algorithm, when compared to the case of population generation with predefined variables.

Hence, it can be concluded that in the case of large initial guesses to the system a random population generator is more efficient. However, in the case when the initial guess is close to the solution then the predefined population generator performs adequately.

Figure 43 and Figure 44 show the variation of the $k_p and\ k_i$ gains for the case of initial $k_p = k_i = 10$.

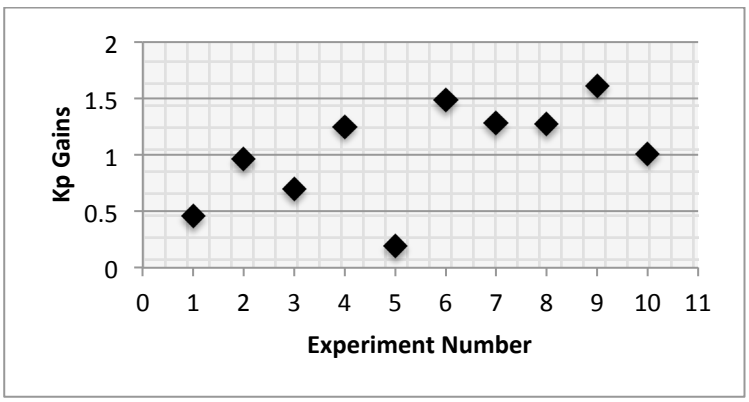


Figure 43: Gain variation in autotuning algorithm with random population generator

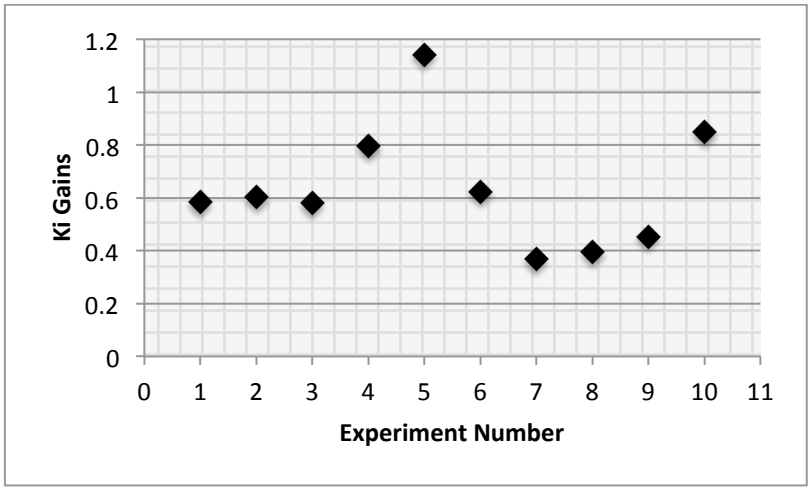


Figure 44: Gain variation in autotuning algorithm with random population generator

From the graphs, $k_{p(average)} = 1.02, k_{p(median)} = 1.13$ and $k_{i(average)} = 0.64$ and $k_{i(median)} = 0.59$.

6.5 Accuracy of results

In the previous chapter (6.4) the response of the same system to different sets of initial conditions was presented. The focus of that section was to draw some conclusions on which population generation method was more efficient. This chapter aims to examine if the gains obtained in both cases were close to the ideal gains of the system. Comparing Figure 40 and Figure 43 it is evident that in both cases the algorithm produced results in the same range. The results obtained at each run of the experiment were not identical, but this is expected, as the tuning criteria are not satisfied by only one particular set of gains. In order to establish if the gains generated by the algorithm are close to the ideals, the Simulink Tuning Function (Figure 47)

was used. The Tuning Function linearised the plant that was used for the previous experiments and enabled the user to change the type of response obtained by the system. Using the built-in Simulink Function it was found that the ideal gains for that system were:

$$k_p = 0.63, k_i = 0.923, k_d = 0.107$$

The ideal results obtained by the Simulink Tuning Function are not the same with the ones obtained by the autotuning algorithm. The biggest deviation between the ideal and experimental proportional gain is 61% and 36% for the integral gain. This difference was expected and the fact that the experimental gains are within a maximum ~61% range of ideal ones strengthens the confidence in the accuracy of the algorithm. Additionally, examining the value of an individual gain does not provide meaningful insights, as the PID sends a command to the plant that combines all gains.

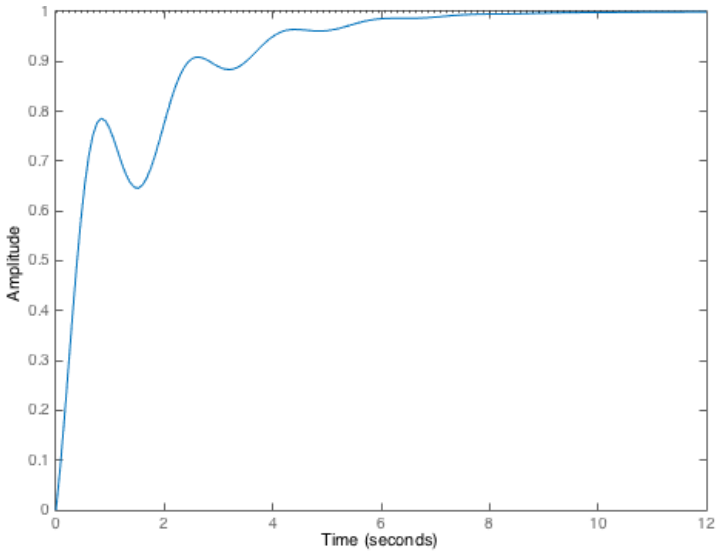


Figure 45: Step response of system with the use of ideal PID gains

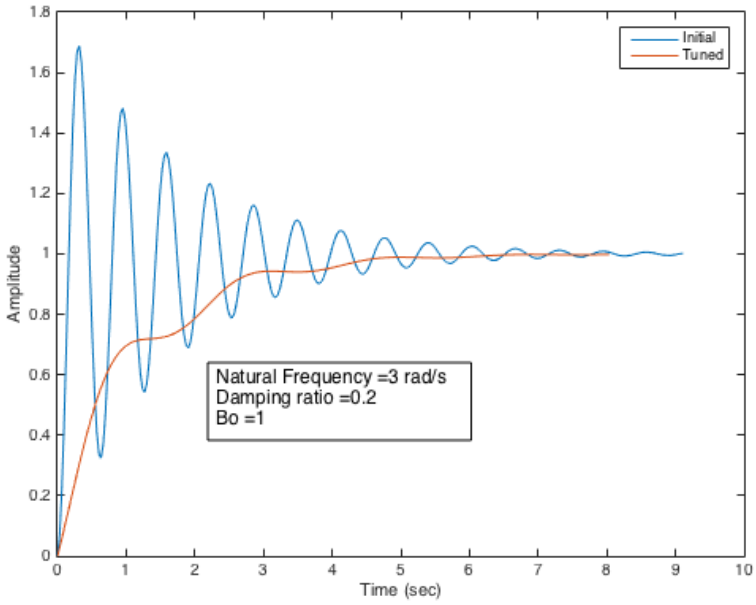


Figure 46: Step response of system with the use of algorithm-generated gains

Figure 45 and Figure 46 compare the responses obtained from the ideal and the algorithm-generated gains for the same system. The algorithm-generated response has a reduced oscillatory behaviour when compared to the ideal one. This is due to the fact that the algorithm prioritises a non-oscillatory response at the expense of rise time, which is faster in the ideal response.

It can be concluded that the response obtained from the algorithm is comparable to the ideal responses and in some instances better, if a user is seeking a particular type of behaviour from the tuned system. However, as this is the first version of the code, further refinement of the autotuning algorithm is required in order to increase the accuracy and precision of the autotuning method.

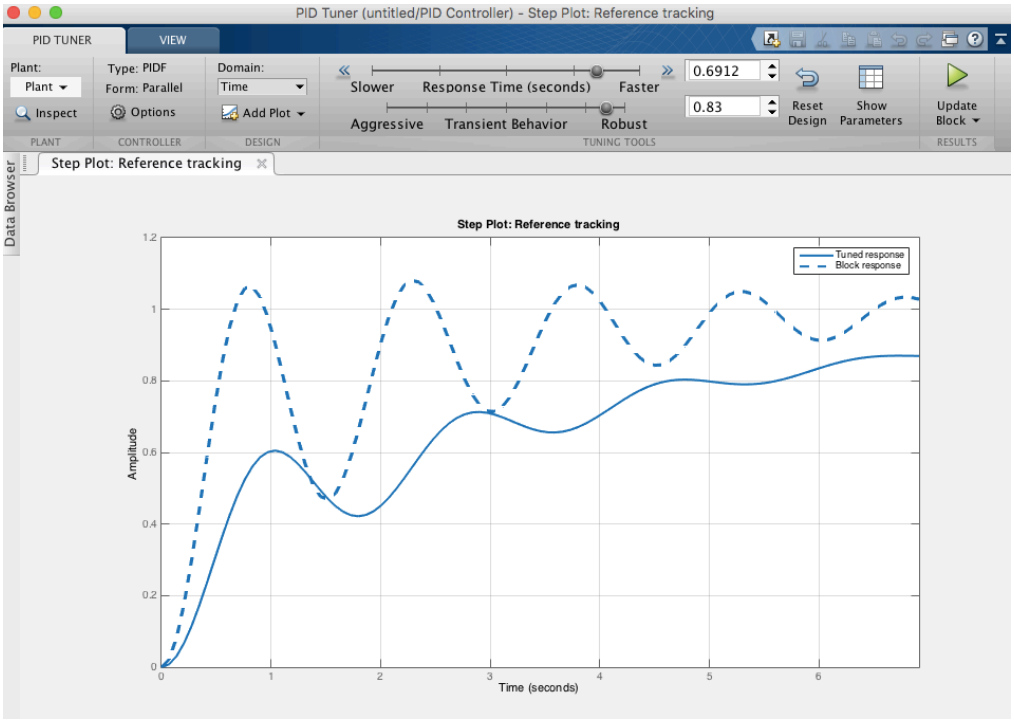


Figure 47: Simulink Tuning Function

6.6 Summary of results

Having presented the results of the previous experiments the following can be concluded about the autotuning algorithm:

- The algorithm has proved that it can successfully tune the gains of a PI controller for a 2nd order system.
- The accuracy of the autotuning algorithm is satisfactory.
- Heuristic approaches are appropriate for tuning the PID gains of a UAV PID controller. Analytical methods have not proved to be useful for UAV systems
- The initial guess of gains is critical to the speed and success of the autotuning algorithm.

- Different tuning criteria will yield different gains. Hence, the results of autotuning experiments should be compared only if the same criteria are used.
- The autotuning algorithm is not yet at a stage to be ported to a UAV autopilot. More tests are required to verify its performance and test different scenarios.
- The biggest challenge in implementing the algorithm on a UAV autopilot will be the conditioning of the sensor signals. Raw sensor data include various types of noise (e.g. electrical), which would have a negative impact on the performance of the algorithm.

7 Future Work

In this project many different approaches have been attempted, countless academic papers read any many lines of code written. However, due to the inherent time constraints of the project many things are still left to be done, before the topic of autotuning is considered to be resolved. This chapter highlights some of the research approaches that the author identified as the next steps following this project.

- **Modelling of the full longitudinal dynamics of a SUAV**

  One of the primary assumptions of the project was that the longitudinal dynamics of a SUAV can be modelled by a second order transfer function. It was shown that this assumption is satisfactory in most cases. However, in order to assess the performance of the autotuning algorithm across the whole flight envelope and dynamic modes of a SUAV, the full equations of the longitudinal dynamics should be used. This requires knowledge of the aerodynamic derivatives of the vehicle in question. Two student projects can be developed around that task. The first student will conduct wind tunnel testing of different vehicles and estimate their aerodynamic derivatives. The second student would be tasked with adapting the current autotuning algorithm to the full equations of motion and increase its performance.

- **Integration of the autotuning algorithm in the PX4 Flight Stack**

  As it was mentioned in chapter 4, the integration of the autotuning algorithm in the PX4 flight stack would mark an important milestone. It

would enable the HIL and flight-testing of the algorithm. This is a difficult task though, which requires students with advanced hardware and software programming.

- **Hardware in the loop (HIL) testing**

  The next step after the integration of the autotuning algorithm in the PX4 Flight Stack is an assessment of the algorithm in a virtual environment. HIL testing enables developers to use the same hardware as used on SUAVs, without having to fly any aircraft. This enables the quick identification of software bugs before any flight tests are conducted.

- **Flight testing**

  The final step towards the full integration of this, or any other, autotuning algorithm in an autopilot is the flight-testing of the software and hardware. Many times, the results obtained in a software environment do no correspond to real life results due to modelling errors. An extensive flight-testing campaign should take place before the algorithm is incorporated in the final product version of an autopilot.

- **System Identification**

  In this project the PI controller was tuned using a heuristic approach. However, there are analytical methods available for the tuning of PID controllers. One of those methods uses the concept of system identification. Further research should take place to test if system identification can be a viable alternative for autotuning of SUAV

autopilots. However, this is a study that would have to be conducted by students at a graduate level.

8 Conclusions

8.1 Introduction

This chapter summarises the work conducted and the key findings of this project. Autotuning UAV autopilots is a hard problem, which has no simple theoretical solutions. This study followed a heuristic approach, with the gains of a PI controller being adjusted based on the results obtained from a metaheuristic algorithm. Other, non-heuristic, autotuning techniques have been considered, but they all relied on a prior knowledge of the system's dynamics, which is usually not practical (or possible) for UAV autopilots. The main contribution of the project is the development of the autotuning algorithm and the definition of the tuning criteria for UAV systems.

8.2 Autotuning Algorithm Assessment

The metaheuristic autotuning algorithm, which is a modified version of a Genetic Algorithm, was able to determine appropriate proportional (k_p) and integral (k_i) gains for the great majority of the tested systems. It is important to reiterate that the performance of the algorithm depends on the tuning criteria set by each individual user. The algorithm terminates when all user-defined conditions have been satisfied. Hence, the outcome of the algorithm can vary for the same system if the tuning criteria are different.

Furthermore, the algorithm is not dependent on the dynamics of a plant and thus can be used in both linear and non-linear systems. However, when the algorithm is used for tuning linear systems, its performance is suboptimal when compared to classical control linear tuning techniques, as shown in the

previous sections. One more important point affecting the performance of the algorithm is the value of the initial guesses for the gains. It was shown that when the starting guess was far from the actual solution it would take more iterations until the algorithm could find the converged solution. This is a risk that can be mitigated if the acceptable range of gains is known in advance.

In conclusion, this project is considered successful because the initial objectives have been met. A proof-of-concept solution has been developed and the experimental results indicate a satisfactory performance from the algorithm.

9 References

Aliyu, B. et al., 2015. Oscillation Analysis for Longitudinal Dynamics of a Fixed-Wing UAV Using PID Control Design. *Advances in Research*, 5(3), pp.1–13. Available at: http://sciencedomain.org/abstract/9899.

Catena, A., Melita, C.D. & Muscato, G., 2013. Automatic Tuning Architecture for the Navigation Control Loops of Unmanned Aerial Vehicles. *Journal of Intelligent & Robotic Systems*, 73(1-4), pp.413–427. Available at: http://link.springer.com/10.1007/s10846-013-9919-2.

Cook, V.M., 2007. *Flight Dynamics Principles* 2nd ed., Elsevier Ltd.

Fadil, M.A. et al., Iterative Learning Auto-tuned PID Controller for Micro-Unmanned Air Vehicle. , (1), pp.153–160.

Fadil, M.A., Jalil, N.A. & Mat Darus, I.Z., 2013. Intelligent PID controller using iterative learning algorithm for active vibration controller of flexible beam. *2013 IEEE Symposium on Computers & Informatics (ISCI)*, (4), pp.80–85. Available at: http://www.scopus.com/inward/record.url?eid=2-s2.0-84886468777&partnerID=tZOtx3y1.

Gondhalekar, A.C., 2009. Strategies for Non-Linear System Identification. , pp.1–192. Available at: http://www.imperial.ac.uk/media/imperial-college/research-centres-and-groups/dynamics/aditya_gondhalekar_thesis.pdf.

Holland, J.H., 1975. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.*, Oxford, England: U Michigan Press.

Huang, A.S., Olson, E. & Moore, D.C., 2010. {LCM}: {Lightweight} communications and marshalling. *Proceedings of the {IEEE}/{RSJ} International Conference on Intelligent Robots and Systems*, pp.4057–4062.

Kishnani, M., Pareek, S. & Gupta, R., 2014. Optimal Tuning of PID Controller Using Meta. , 7(2), pp.171–176.

Liu, W.F., Jiang, Z. & Gong, Z.B., 2007. Online fuzzy self-adaptive PID attitude control of a sub mini fixed-wing air vehicle. *Proceedings of the 2007 IEEE International Conference on Mechatronics and Automation, ICMA 2007*, pp.153–157.

Matko, D. et al., 2013. Self-Tuning Dynamic Matrix Control of Two-Axis Autopilot For Small Aeroplanes.

Meier, L., Honegger, D. & Pollefeys, M., PX4 : A Node-Based Multithreaded Open Source Robotics Framework for Deeply Embedded Platforms. *International Conference on Robotics and Automation*, pp.6235–6240.

Mitchell, M., 1996. An introduction to genetic algorithms. *Cambridge, Massachusetts London, England, …*, p.162.

Pirabakaran, K. & Becerra, V., 2002. PID autotuning using neural networks and model reference adaptive control. *IFAC World Congress*. Available at: http://www.nt.ntnu.no/users/skoge/prost/proceedings/ifac2002/data/content/01467/1467.pdf.

Rivas-echeverría, F., Ríos-bolívar, A. & Casales-echeverría, J., Neural Network-based Auto-Tuning for PID Controllers. Available at:

http://www.wseas.us/e-library/conferences/crete2001/papers/658.pdf.

Ronfle-Nadaud, C., 2014. Adaptive control, wide speed range flight and deconfliction. Available at: https://hal.archives-ouvertes.fr/hal-01021535/document.

Rynaski, E.G., 1985. *The interpretation of flying qualities requirements for flight control system design*, NASA. Available at: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19850024812.pdf.

Safaee, A. & Taghirad, H.D., 2013. System identification and robust controller design for the autopilot of an unmanned helicopter. *2013 9th Asian Control Conference (ASCC)*, pp.1–6. Available at: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6606298.

Samuelsson, M., 2012. *Evaluation of stability and flying qualities of a light unmanned aerial vehicle(UAV)*. Available at: http://www.diva-portal.org/smash/get/diva2:550688/fulltext02.

Stock, M., 2006. *CANaerospace-interface specification for airborne CAN applications V 1.7*, Available at: https://scholar.google.com/scholar?q=CANaerospace-interface+specification+for+airborne+CAN+applications+V+1.7&btnG=&hl=en&as_sdt=0,5#0.

Szafranski, G. & Czyba, R., 2011. Different Approaches of PID Control UAV Type Quadrotor. *Proceedings of the International Micro Air Vehicles conference*, pp.70–75.

Vanin, M., 2013. Modeling , identification and navigation of autonomous air

vehicles. , (May).

Ziegler, J.G. & Nichols, N.B., 1942. Optimum Settings for Automatic Controllers. *Transacction of the ASME*, pp.759–765.

Figures:

Figure 2: Public Domain

Figure 3: Beyer,R.,2009.Industrial.NET-PID Controllers.[online] Available at <http://www.codeproject.com/Articles/49548/Industrial-NET-PID-Controllers> [Accessed 15 Dec 2015]

Figure 5: Meier, L., Honegger, D. & Pollefeys, M., PX4 : A Node-Based Multithreaded Open Source Robotics Framework for Deeply Embedded Platforms. International Conference on Robotics and Automation, pp.6235–6240.

A. Appendix A: Autotuning Algorithm

```
%%%%%%%%%
%
% Autotuning Algorithm for second order systems
%
% Dimitrios Tsounis
% dimitrios.tsounis@student.manchester.ac.uk
% dimtsouis@gmail.com
% The University of Manchester
% April 2016
%
%%%%%%%

clear all;
clc;

% Values of damping, natural frequency and SS gain
damping_ratio = (2:8)/10;
natural_freq = (3:10);
b0 = 1;

%variables used to select different frequency/damping values
i = 7;  % natural frequency
j = 7;  % damping ratio

% Calculating the values of the nominator and denominator of the TF
a = b0*(natural_freq(i)^2);
b = 2*natural_freq(i)*damping_ratio(j);
c = natural_freq(i)^2;

num = [a];
den = [1 b c];

% Setting up the Transfer Function
system_tf = tf(num,den);

% flags & counting variables
k = 0;


%%%%%%
% String Manipulation
% 1st Annotation box
s1_freq = num2str(natural_freq(i));
s2_damp = num2str(damping_ratio(j));
s3_b0 = num2str(b0);

s1 = strcat('Natural Frequency = ', s1_freq,' rad/s');
s2 = strcat('Damping ratio = ', s2_damp);
s3 = strcat('Bo = ', s3_b0);

s = [s1 char(10) s2 char(10) s3];
%%%%%%

% Setting initial PID gains
max_prop_gain = 4;
max_int_gain = 4;
kp = rand*max_prop_gain;
```

```
ki = rand*max_int_gain;

% Setting Crossover ratios
COR1 = 0.2; % fitness
COR2 = 0.4; % oscillation
COR3 = 0.2; % overshoot
COR4 = 0.2; % time ratio

%population creation
gains = [ kp ki;
    0.5*kp ki;
    kp 0.5*ki;
    0.5*kp 0.5*ki;
    0.5*kp 3*ki;
    3*kp 0.5*ki;
    3*kp 3*ki;];


%%%%%%
initial_pid = pid(gains(1,1),gains(1,2),0.1);
% Plotting the initial response of the system
input_function1 = feedback(initial_pid*system_tf,1);
[y1,t1]=step(input_function1);
fig1 = figure;
plot(t1,y1)
xlabel('Time (sec)');
ylabel('Amplitude');
title('');
annotation('textbox', 'String',s,'Fontsize',12);
hold on;
%%%%%%

while true
    %%%
    % Following values need updating if the size of the population
changes
    sum(1:7) = 0;
    sum_counter = 1;
    flag_var(1:7) = 0;
    overshoot(1:7) = 0;

    for population_size=1:size(gains,1)
        clearvars y;
        clearvars t;
        clearvars pks;
        clearvars locs;
        clearvars mpks;
        clearvars rise_point;
        clearvars time_stamp1;
        clearvars time_stamp2;
        clearvars ts1_value;
        clearvars ts2_value;

        % Setting up PI Controller
        C =
pid(gains(population_size,1),gains(population_size,2),0.1);

        % generating input function for feedback loop with control
        input_function = feedback(C*system_tf,1);

        %step function
```

```
        [y,t]=step(input_function);
        %%%

        %%%%%%
        % Annotation Box
        skp = num2str(gains(population_size,1));
        ski = num2str(gains(population_size,2));

        s1kp = strcat('Kp = ', skp);
        s2ki = strcat('Ki = ',ski);

        s_gains = ['PI Gains' char(10) s1kp char(10) s2ki]; %
textbox element
        %%%%%%

        % Default values
        flag_var(population_size) = 0;
        tuning_rqrd(population_size) = 0;
        tuning_level(population_size) = 1;

        % Signal Info
        s_info = stepinfo(y,t);

        % Plotting the response (commented out to speed up program
execution)
        %
        % figure;
        % plot(t,y);
        % annotation('textbox', 'String',s,'Fontsize',12);
        % annotation('textbox',[.6 0.3 .1 .1],
'String',s_gains,'Fontsize',12);
        % title('');

        %%%%%%%%%%%%
        % Signal Analysis

        [pks,locs] = findpeaks(y); % Local maxima estimation
        [mpks,mlocs] = findpeaks(-y); % Local minima estimation
        mpks = (-mpks); % chaning the sign of the values

        maxima_num(population_size) = size(pks,1);
        minima_num(population_size) = size(mpks,1);

        % Check if system reaches steady state
        ss_value = y(size(y,1));
        if (ss_value>=0.95) && (ss_value<=1.05);
            steady_state_reached(population_size) = true;
            tuning_rqrd(population_size) = 0;
        else
            tuning_rqrd(population_size) = 1;
            steady_state_reached(population_size) = false;
        end

        % Checking if system is oscillating
        % By changing the if statements, it is possible to change
the
        % number of acceptable oscillation
        if ((maxima_num(population_size))>=4) &&
(minima_num(population_size)>=3);
            oscillating(population_size) = 1;
```

```
            tuning_rqrd(population_size) = 1;
            tuning_level(population_size) = 2; %#ok<*SAGROW>
            if (maxima_num(population_size)>5) || (abs(pks(1)-
mpks(1))>0.4)
                tuning_level(population_size) = 3;
            end
            if maxima_num(population_size)>6 || (abs(pks(1)-
mpks(1))>0.6)
                tuning_level(population_size) =4;
            end
            if maxima_num(population_size)>7 || (abs(pks(1)-
mpks(1))>0.8)
                tuning_level(population_size) =5;
            end
        else
            oscillating(population_size) = 0;
        end

        % Checking system time response
        % further work is required to established a 'good' reference
value
        t_ratio(population_size) =
s_info.SettlingTime/s_info.RiseTime;
        if t_ratio(population_size)>40
            tuning_rqrd(population_size) = 1;
        end

        % Derivative estimation using central difference
        % Commented out because it is not used in this algorithm
%         for l=1:(size(y,1)-2)
%             der(l) = (y(l+2)-y(l+1))/(t(l+2)-t(l+1));
%         end

        % Checking if the system is overshooting or undershooting
        for j=1:round(size(t,1)/2)
            if y(j)> 1
                flag_var(population_size) = 1; % system overshooting
            end
        end
        if (size(pks,1))>=1
            overshoot(population_size) = pks(1)-1; % overshoot in
decimal percentage
        end

        if (size(mpks,1))>=1
            undershoot(population_size) = abs(pks(1)-1); %
undershoot in decimal percentage
        end

        if flag_var(population_size) == 1
            if overshoot(population_size) >0.6
                tuning_rqrd(population_size) =1;
            end
        end

        % establishing the nearest discrete time value to the rise
time
        for j=1:size(t,1)
            diff_eqn = (s_info.RiseTime)-t(j);
            if diff_eqn >= 0
                time_stamp1 = j;
```

```
                ts1_value = diff_eqn;
            end
            if diff_eqn <= 0
                time_stamp2 = j;
                ts2_value = diff_eqn;
                break;
            end
            diff_eqn = 0;
        end

        % choosing the discrete t value closest to the rise time
        if abs(ts1_value) > abs(ts2_value)
            rise_point = time_stamp2;
        else
            rise_point = time_stamp1;
        end

        % Fitness Function
        for i=1:rise_point
            sum(sum_counter) = sum(sum_counter) + abs(1-y(i));
        end
        sum_counter = sum_counter+1;
end

    %%%%%%%%

    % sorting Fitness Function
    [current_sum, index1] = sort(sum);

    for cntr=1:size(sum,2)
        if sum(cntr) > 15
            tuning_rqrd(cntr) = 1;
            % fitness issues

        end
    end

    %s sorting tuning_rqrd
    [tun_rqrd_sorted,tun_req_index] = sort(tuning_rqrd);

    if tun_rqrd_sorted(1) == 0
        disp('Tuning Achieved');
        disp(gains(tun_req_index(1),1));
        disp(gains(tun_req_index(1),2));

        % saving gains for graph
        gainp(k+1) = gains(tun_req_index(1),1);
        gaini(k+1) = gains(tun_req_index(1),2);
        C =
pid(gains(tun_req_index(1),1),gains(tun_req_index(1),2),0.1);

        % generating input function for feedback loop with control
        input_function = feedback(C*system_tf,1);

        %step function
        [y,t]=step(input_function);
        plot(t,y)
        legend('Initial','Tuned');

        %stop iteration
```

```
    break;
else
    disp('Tuning required');
end


%%%%%%%%%%
% Tuning

% Fitness function optimisation
kp_fitness = gains(index1(1),1);
ki_fitness = gains(index1(1),2);

% Steady state optimisation
% steady state gain not used in this version of the algorithm
[ss_sorted,index_ss]= sort(steady_state_reached,'descend');
kp_ss = gains(index_ss(1),1);
ki_ss = gains(index_ss(1),2);

% Oscillation optimisation
[tun_level_intensity,index2] = sort(tuning_level);

% adjust gains based on the oscillation intensity level
if (tun_level_intensity(1) == 5)
    kp_osc = gains(index2(1),1)/3;
    ki_osc = gains(index2(1),2)/3;
end
if (tun_level_intensity(1) == 4)
    kp_osc = gains(index2(1),1)/2;
    ki_osc = gains(index2(1),2)/2;
end
if (tun_level_intensity(1) == 3)
    kp_osc = gains(index2(1),1)/2;
    ki_osc = gains(index2(1),2)/1.5;
end
if (tun_level_intensity(1)== 2)
    kp_osc = gains(index2(1),1)/1.5;
    ki_osc = gains(index2(1),2);
end
if (tun_level_intensity(1)== 1)
    % if tuning level = 1 keep the same gains
    kp_osc = gains(index2(1),1);
    ki_osc = gains(index2(1),2);
end

% Overshoot optimisation
[flag_sorted,index3] = sort(flag_var);

[overshoot_sorted,index4] =sort(overshoot);

for i=1:size(overshoot_sorted,1)
    if overshoot_sorted(i)<0
        ovrt_indic = 0;
    end
    if overshoot_sorted(i) >= 0 && overshoot_sorted(i)<0.5
        ovrt_indic = 1;
        kp_overshoot = gains(index4(i),1);
        ki_overshoot = gains(index4(i),2);
        break;
    else
        ovrt_indic = 0;
```

```
            kp_overshoot = gains(index4(i),1)/1.5;
            ki_overshoot = gains(index4(i),2)/1.25;
            break;
        end
    end

    % t_ratio optimisation
    [t_ratio_sorted,trindex] = sort(t_ratio);
    kp_tr = gains(trindex(1),1);
    ki_tr = gains(trindex(1),2);

    if ovrt_indic == 1
        kp = COR1*kp_fitness + COR2*kp_osc + COR3*kp_overshoot +
COR4*kp_tr;
        ki = COR1*ki_fitness + COR2*ki_osc + COR3*ki_overshoot +
COR4*ki_tr;
    else
        kp = COR1*kp_fitness + COR2*kp_osc + COR4*kp_tr;
        ki = COR1*ki_fitness + COR2*ki_osc + COR4*ki_tr;
    end

    if k>1
        if last_kp == kp
            kp = (rand+0.2)*kp;
        end
        if last_ki == ki
            ki = (rand+0.2)*ki;
        end
    end

      % pre defined population generation

%     gains = [ kp ki;
%         0.5*kp ki;
%         kp 0.5*ki;
%         0.5*kp 0.5*ki
%         0.5*kp 3*ki;
%         3*kp   0.5*ki;
%         3*kp   3*ki;];

    % random population generation

    gains = [ kp ki;
        rand*kp ki;
        kp rand*ki;
        rand*kp rand*ki
        rand*kp rand*ki;
        2*kp   2*ki;
        10*rand*kp   10*rand*ki;];

    last_kp = kp;
    last_ki = ki;

    %saving gains for graph
    gainp(k+1) = kp;
    gaini(k+1) = ki;
    %%%%%%%%%%

    disp('==================');
    % loop termination
    if k == 50
```

```
        disp(gains);
        % Final System
        C = pid(gains(index1(1),1),gains(index1(1),2),0.1);

        % generating input function for feedback loop with control
        input_function = feedback(C*system_tf,1);

        %step function
        [y,t]=step(input_function);
        plot(t,y)
        legend('Initial','Final');
        break;
    else
        k = k+1;
    end
end

if k == 0
    k = k+1;
end

% plotting the gain variation

% Kp Graph

% Annotation Box
kp_p1 = num2str(gainp(1));
kp_p2 = num2str(gainp(end));
s_kp_p1 = strcat('Initial Gain: ', kp_p1);
s_kp_p2  = strcat('Final Gain: ',kp_p2);
p_gains_variation = [s_kp_p1 char(10) s_kp_p2]; % textbox element

xaxis = (1:(k+1));
fig2 = figure;
plot(xaxis,gainp)
xlabel('Iterations');
ylabel('Kp');
title('');
xbounds = xlim();
annotation('textbox',[0.15 0.13 0.25 0.08],
'String',p_gains_variation,'Fontsize',13);
set(gca, 'xtick', xbounds(1):1:xbounds(2));

% Ki Graph
% Annotation Box
ki_p1 = num2str(gaini(1));
ki_p2 = num2str(gaini(end));
s_ki_p1 = strcat('Initial Gain: ', ki_p1);
s_ki_p2  = strcat('Final Gain: ',ki_p2);
i_gains_variation = [s_ki_p1 char(10) s_ki_p2]; % textbox element

fig3 = figure;
plot(xaxis,gaini)
xlabel('Iterations');
ylabel('Ki');
title('');
xbounds = xlim();
annotation('textbox',[0.15 0.13 0.25 0.08],
'String',i_gains_variation,'Fontsize',13);
set(gca, 'xtick', xbounds(1):1:xbounds(2));
```

B. Appendix B: Autotuning Algorithm for Reference Tracking

```
%%%%%%%%%
%
% Autotuning Algorithm for reference tracking using second order
systems
%
% Dimitrios Tsounis
% dimitrios.tsounis@student.manchester.ac.uk
% dimtsouis@gmail.com
% The University of Manchester
% April 2016
%
%%%%%%%

clear all;
clc;

% Values of damping, natural frequency and SS gain
damping_ratio = (2:8)/10;
natural_freq = (3:10);
b0 = 1;

%variables used to selected different frequency/damping values
i = 1;  % natural frequency
j = 1;  % damping ratio

% Calculating the values of the nominator and denominator of the TF
a = b0*(natural_freq(i)^2);
b = 2*natural_freq(i)*damping_ratio(j);
c = natural_freq(i)^2;

k =1;

model = 'reference_tracking';
load_system(model)

max_prop_gain = 3;
max_int_gain = 3;
kp_initial = rand*max_prop_gain;
ki_initial = rand*max_int_gain;


%population creation
gains = [ kp_initial ki_initial;
    0.5*kp_initial ki_initial;
    kp_initial 0.5*ki_initial;
    0.5*kp_initial 0.5*ki_initial
    2*kp_initial 2*ki_initial];

disp(gains);
kp = kp_initial;
ki = ki_initial;

kp_p1 = num2str(kp);
ki_p2 = num2str(ki);
s_kp_p1 = strcat('Initial P Gain: ', kp_p1);
s_ki_p2  = strcat('Initial I Gain: ',ki_p2);
```

```
initial_gains = [s_kp_p1 char(10) s_ki_p2]; % textbox element


sim(model);
fig1 = figure;
plot(tout,simout.signals.values(:,1),tout,simout.signals.values(:,2)
);
xlabel('Time (s)');
ylabel('Amplitude');
title('');
legend('Reference Signal', 'Initial Response');
annotation('textbox',[0 0 1
1],'String',initial_gains,'Fontsize',13);

fig2 = figure;
hold on;

while true
    iter_counter = 1;
    sum(1:5) = 0;
    input(1:5) = 0;
  %  response(1:5) = 0;

    for i=1:size(gains,1)
        kp = gains(i,1);
        ki = gains(i,2);
        sim(model);
        input = simout.signals.values(:,1);
        response = simout.signals.values(:,2);
        plot(tout,input,tout,response);
        xlabel('Time (s)');
        ylabel('Amplitude');
        title('');

        % fitness function definition
        for j=1:size(response,1)
            sum(iter_counter) = sum(iter_counter) + abs(input(j)-
response(j));
        end
        iter_counter = iter_counter + 1;
    end

    [sum_sorted,index] = sort(sum);
    sgra(k) = sum_sorted(1);
    kpgr(k) = gains(index(1),1);
    kigr(k) = gains(index(1),2);

    if sum_sorted(1)<= 5
        disp('System Tuned');
        kp = gains(index(1),1);
        ki = gains(index(1),2);
        %%%%
        kp_p1 = num2str(kp);
        ki_p2 = num2str(ki);
        s_kp_p1 = strcat('Final P Gain: ', kp_p1);
        s_ki_p2  = strcat('Final I Gain: ',ki_p2);
        final_gains = [s_kp_p1 char(10) s_ki_p2]; % textbox element
        %%%
        sim(model);
        fig3 = figure;
```

```
plot(tout,simout.signals.values(:,1),tout,simout.signals.values(:,2)
);
        xlabel('Time (s)');
        ylabel('Amplitude');
        title('');
        legend('Reference Signal', 'Final Response');
        annotation('textbox',[0 0 1 1],
'String',final_gains,'Fontsize',13);


        disp(gains(index(1),1));
        disp(gains(index(1),2));
        break;
    end

    kp_new = gains(index(1),1);
    ki_new = gains(index(1),2);

    if k>2
        if last_kp == kp_new
            kp_new = (rand)*kp_new;
        end
        if last_ki == ki_new
            ki_new = (rand)*ki_new;
        end
    end

%     gains = [ kp_new ki_new;
%         0.5*kp_new ki_new;
%         kp_new 0.5*ki_new;
%         0.5*kp_new 0.5*ki_new
%         2*kp_new 2*ki_new];
%
     gains = [ kp_new ki_new;
        rand*kp_new ki_new;
        kp_new rand*ki_new;
        0.5*kp_new 0.5*ki_new
        10*rand*kp_new 10*rand*ki_new];

    last_kp = kp_new;
    last_ki = ki_new;

    disp('New Iteration');

    if k == 100
        disp('No Tuning');
        break;
    else
        k = k + 1;
    end
end

xaxis = (1:k);

fig4 = figure;
plot(xaxis,sgra);
xlabel('Iterations');
ylabel('Fitness Function');
title('');
xbounds = xlim();
set(gca, 'xtick', xbounds(1):1:xbounds(2));
```

```
fig5 = figure;
plot(xaxis,kpgr,xaxis,kigr);
xlabel('Iterations');
ylabel('Gains');
legend('Kp','Ki');
xbounds = xlim();
set(gca, 'xtick', xbounds(1):1:xbounds(2));


saveas(fig1,'initial.bmp');
saveas(fig2,'intermediate.bmp');
saveas(fig3,'final_response.bmp');
saveas(fig4,'fitness_function.bmp');
saveas(fig5,'gains_variation.bmp')
```

C. Appendix C: Equations of Motion

In chapter 4.2.1 the following EoMs were briefly described. The full equations of motion can be found in (Cook 2007) and are not developed here for brevity purposes.

$$m\dot{u} - X_u u - X_{\dot{w}}\dot{w} - X_w w - X_q q + mg\theta = X_\eta \eta$$

$$-Z_u u + (m - Z_{\dot{w}})\dot{w} - Z_w w - \left(Z_q + mU_e\right)q = Z_\eta \eta$$

$$-M_u u - M_{\dot{w}}\dot{w} - M_w w + I_y \dot{q} - M_q q = M_\eta \eta$$

Where,

$m = mass\ of\ aircraft$

$\dot{u} = axial\ acceleration$

$X_u = Axial\ force\ due\ to\ axial\ velocity$

$X_w = Axial\ force\ due\ to\ normal\ velocity$

$X_{\dot{w}} = Axial\ force\ due\ to\ normal\ acceleration$

$X_q = Axial\ force\ due\ to\ pitch\ rate$

$Z_u = Normal\ force\ due\ to\ axial\ velocity$

$Z_w = Normal\ force\ due\ to\ normal\ velocity$

$Z_{\dot{w}} = Normal\ force\ due\ to\ normal\ acceleration$

$Z_q = Normal\ force\ due\ to\ pitch\ rate$

$M_u = Pitching\ moment\ due\ to\ axial\ velocity$

$M_w = Pitching\ moment\ due\ to\ normal\ velocity$

$M_{\dot{w}} = Pitching\ moment\ due\ to\ normal\ acceleration$

$M_q = Pitching\ moment\ due\ to\ pitch\ rate$

$I_y = Moment\ of\ inertia$

$X_\eta = Axial\ force\ due\ to\ elevator$

$Z_\eta = Normal\ force\ due\ to\ elevator$

$M_\eta = Pitching\ moment\ due\ to\ elevator$