

”



DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOR'S THESIS

Study program/specialization:	Spring semester 2022
Bachelor in engineering / Automatisering og elektronikkdesign	Open
Writer: Jacob Reiersøl	
Faculty supervisor: Damiano Rotondo	
Supervisor: Damiano Rotondo	
Thesis title: Cascade control on the Quanser Aero	
Credits: 20	
Key words: Control theory Cascade Control Modelling Simulation Programming	Pages: 78 + enclosure: 51 Stavanger, May. 22, 2022

Contents

Contents	i
List of Figures	iv
List of Tables	ix
1 Introduction	1
1.1 Structure	2
1.2 Single-loop feedback control	3
1.3 The PID controller	4
1.4 Cascade control	6
1.5 Integral windup and clamping	8
1.6 Integral performance indices	9
2 Description of the Quanser Aero	10
2.1 General description	11
2.2 Modeling	12

CONTENTS

2.3	Process behavior	15
3	Tuning methods	18
3.1	Single-loop tuning methods	18
3.1.1	Ziegler-Nichols closed loop method	18
3.1.2	Standard relay-feedback method	20
3.1.3	Ziegler-Nichols open-loop method	21
3.2	Sequential cascade control tuning methods	22
3.3	Simultaneous cascade tuning using step input	23
3.3.1	The area method	25
3.3.2	Model estimation using least-squares	26
3.3.3	Least-squares reduction method	28
3.3.4	Simultaneous tuning using process models	31
3.3.5	Tuning based on SOPDT or FOPDT models	31
3.3.6	Kappa-Tau tuning	33
3.4	Method selection	33
3.4.1	Single loop tuning	33
3.4.2	Sequential tuning of cascade controller	33
3.4.3	Simultaneous cascade control tuning	34
4	Testing	36

CONTENTS

4.1	Ziegler-Nichols closed-loop method	36
4.2	Standard relay-feedback method	38
4.3	Sequential Ziegler-Nichols closed loop plus Ziegler-Nichols open loop methods on cascaded system	41
4.4	Sequential relay-feedback method plus Ziegler-Nichols open loop tuning methods on cascaded system	45
4.5	Simultaneous tuning using step response	48
4.5.1	Common grounds	48
4.5.2	Step response Kappa-Tau	49
4.5.3	Step response IMC	49
4.5.4	Step response simultaneous FOPDT plus FOPDT	50
4.5.5	Step response simultaneous FOPDT plus SOPDT	50
5	Results	52
5.1	PID parameters and filter coefficients	53
5.2	Figures	55
5.2.1	Efficient propellers	55
5.2.2	Inefficient propellers	59
5.3	Integral performance indices	63
6	Discussion and future work	64
6.1	Discussion	64

CONTENTS

6.2 Future work	65
7 Conclusion	67
Bibliografi	69
Appendix	69
A Matlab scripts	70
B Simulink Schemes	113

List of Figures

1.1	Generic block diagram for feedback control	3
1.2	PID-controller diagram	5
1.3	Block diagram for basic feedback control. 1 and 2 refer to whether it belongs to the outer loop (1) or inner loop (2). It otherwise follows the same terminology as the feedback scheme.	6
1.4	Basic block diagram for clamping	8
2.1	Data sheet image of Quanser Aero	10
2.2	High efficiency propeller[10]	11
2.3	Low efficiency propeller[10]	11
2.4	Free-body diagram of the 2DOF helicopter mode Quanser Aero	12
2.5	Open loop step response of $\phi(t)$, efficient propellers	15
2.6	Open loop step response of $\omega(t)$, efficient propellers	16
2.7	Open loop step response of $\phi(t)$, inefficient propellers	16
2.8	Open loop step response of $\omega(t)$, inefficient propellers	17

LIST OF FIGURES

3.1	Basic block diagram for Ziegler-Nichols closed-loop method	19
3.2	PIDx: PID-controller diagram for Ziegler-Nichols method. It is identical to a regular PID controller, except it features switches to enable and disable the derivative and integral gains	19
3.3	Basic block diagram for standard relay-feedback method	20
3.4	Basic block diagram for Ziegler-Nichols open-loop feedback method	21
3.5	Basic block diagram for simultaneous step response method	23
3.6	Visual representation of the area method	25
4.1	Block diagram for Ziegler-Nichols closed-loop method testing	36
4.2	Marginally stable Ziegler-Nichols system response, efficient propellers, obtained at $K_P = K_U = 70.50$	37
4.3	Marginally stable Ziegler-Nichols system response, inefficient propellers, obtained at $K_P = K_U = 38.00$	38
4.4	Block diagram for standard relay-feedback method testing	39
4.5	Relay feedback test for efficient propellers, obtained at $h = 50$	40
4.6	Relay feedback test for inefficient propellers, obtained at $h = 12$	40
4.7	Block diagram for Ziegler-Nichols closed loop plus Ziegler-Nichols open loop cascade control	41
4.8	Inner loop open loop step response, efficient propeller, found at $U = 15$. . .	42
4.9	Inner loop open loop step response, inefficient propeller, found at $U = 15$. .	42
4.10	Inner loop open loop tuning result, efficient propeller, found at step input $r_2(t) = 150$	43

LIST OF FIGURES

4.11 Inner loop open loop tuning result, efficient propeller, found at step input $r_2(t) = 150$ 43

4.12 Marginally stable outer loop response, efficient propellers, found at $K_U = K_P = 25000$ 44

4.13 Marginally stable outer loop response, inefficient propellers, found at $K_U = K_P = 4100$ 45

4.14 Block diagram for relay-feedback plus Ziegler-Nichols open loop cascade control 45

4.15 Relay outer loop test, efficient propellers, found at $h = 800$ 47

4.16 Relay outer loop test, inefficient propellers, found at $h = 250$ 47

4.17 Basic block diagram for simultaneous step response method 48

5.1 Single loop closed loop Ziegler-Nichols method result, efficient propellers . . . 55

5.2 Single loop relay feedback method result, efficient propellers 55

5.3 Sequential closed loop Ziegler-Nichols plus open loop Ziegler-Nichols result, efficient propellers 56

5.4 Sequential relay feedback plus open loop Ziegler-Nichols result, efficient propellers 56

5.5 Step response Kappa-Tau cascade tuning, efficient propellers 57

5.6 Step response IMC cascade tuning, efficient propellers 57

5.7 Step response simultaneous FOPDT plus FOPDT cascade tuning, efficient propellers 58

5.8 Step response simultaneous FOPDT plus SOPDT cascade tuning, efficient propellers 58

LIST OF FIGURES

5.9	Single loop closed-loop Ziegler-Nichols result, inefficient propellers	59
5.10	Single loop relay feedback result, inefficient propellers	59
5.11	Sequential closed loop Ziegler-Nichols plus open loop Ziegler-Nichols result, inefficient propellers	60
5.12	Sequential relay feedback plus open loop Ziegler-Nichols result, inefficient propellers	60
5.13	Step response Kappa-Tau cascade tuning, inefficient propellers	61
5.14	Step response IMC cascade tuning, inefficient propellers	61
5.15	Step response simultaneous FOPDT plus FOPDT cascade tuning, inefficient propellers	62
5.16	Step response simultaneous FOPDT plus SOPDT cascade tuning, inefficient propellers	62
B.1	Simulink scheme for single loop Ziegler-Nichols closed loop tuning	113
B.2	Simulink scheme for single loop relay feedback tuning	114
B.3	Simulink scheme for single loop Ziegler-Nichols closed loop plus Ziegler-Nichols open loop tuning, left half	115
B.4	Simulink scheme for single loop Ziegler-Nichols closed loop plus Ziegler-Nichols open loop tuning, right half	116
B.5	Simulink scheme for single loop relay feedback plus Ziegler-Nichols open loop tuning, left half	117
B.6	Simulink scheme for single loop relay feedback plus Ziegler-Nichols open loop tuning, right half	118
B.7	Simulink scheme for simultaneous step response tuning (all approaches), left half	119

LIST OF FIGURES

B.8 Simulink scheme for simultaneous step response tuning (all approaches),
right half 120

List of Tables

1.1	Manual tuning guidance table	5
2.1	2DOF helicopter parameters	13
3.1	Ziegler-Nichols PID tuning table, where $K_U =$ ultimate gain and $T_U =$ ultimate frequency	20
3.2	Table for calculation of Ziegler-Nichols open-loop PID parameters	22
3.3	Tuning rules for cascade controllers given FOPDT or SOPDT models of processes P_1 and P_1	31
3.4	IMC-PID tuning rules	32
3.5	IMC-Chien tuning rules	32
3.6	IMC-Chien tuning rules	33
4.1	PID parameters and filter coefficients for Ziegler-Nichols tuning	37
4.2	PID parameters and filter coefficients for Relay feedback tuning	39
4.3	PID parameters and filter coefficients for Ziegler-Nichols open-loop plus Ziegler-Nichols closed loop tuning	44

LIST OF TABLES

4.4	PID parameters and filter coefficients for Ziegler-Nichols open-loop plus relay feedback tuning	46
4.5	PID parameters and filter coefficients for 'step response Kappa-Tau' tuning	49
4.6	PID parameters and filter coefficients for 'step response IMC' tuning	50
4.7	PID parameters and filter coefficients for 'step response simultaneous FOPDT plus FOPDT' tuning	50
4.8	PID parameters and filter coefficients for 'step response IMC' tuning	51
5.1	PID parameters and filter coefficients for all tuning methods, efficient propellers	53
5.2	PID parameters and filter coefficients for all tuning methods, inefficient propellers	54
5.3	Integral performance indices for efficient propellers	63
5.4	Integral performance indices for inefficient propellers	63

Chapter 1

Introduction

Core to the subject of control theory is the feedback loop. Using the error between the reference and the measured output signal passing through a tuned controller, the feedback loop can optimize how well a system reaches its desired state. Usage of this method in a system is known as feedback control. There are many approaches in the field of feedback control, all of which aim to reduce the aforementioned error as effectively as possible. One of these is cascade control, which involves nesting at least one additional feedback loop within another.

The objective of this report is to evaluate the effectiveness of using the cascade control method to control Quanser's Quanser Aero [1]. More specifically, cascade control will be performed by using the Aero's wing angle $\phi(t)$ as the output variable of the outer loop, and the rotational speed of the motors ω (sometimes referred to as ϕ/s) as the output variable of the inner loop. This will be done in a 1DOF (degrees of freedom) configuration, and using only one of the Aero's motors. Compared to the default of a single loop with the angle as the only output variable, according to Visioli and Antonio [12] this configuration should provide superior disturbance rejection properties. Hopefully this sufficiently improves the performance to justify the additional effort in applying it.

To confirm this, testing various methods of implementing the cascade control system will be necessary. Then, to compare the effectiveness of cascade control over regular feedback control, some methods for implementing single loop feedback control will be tested as well. This will all be tested using Matlab's Simulink program. While the Quanser Aero itself

1.1 Structure

naturally operates with continuous-time, the sensors and the the software used operates in discrete-time. The Simulink schemes in this report are all set to fixed-step at 0.002 second intervals.

1.1 Structure

Firstly, Chapter 1 aims to explain the main objectives of the report, as well as cover some elementary concepts that lays the groundwork for the rest of the report.

Next, Chapter 2 aims to give an understanding of how the object of the report, the Quanser Aero, works. This includes a basic description of its mechanical properties, a mathematical model, and an explanation as to how the Quanser Aero behaves as a process.

Next, Chapter 3 aims to explain and give an understanding of how to perform the various methods that will eventually be tested in Chapter 4. In what way these methods will be tested it is covered at the end of the chapter.

Next, Chapter 4 aims to describe the exact process that went into applying these methods to the Quanser Aero. This includes the various response is obtained from the Quanser Aero in the testing as well as the parameters obtained by the end.

Next, Chapter 5 aims to demonstrate the results from the testing of the previous chapter. This includes tables showing all the finished parameters next to each other, figures demonstrating the final step response and the performance against disturbances, as well as the performance indices of the results.

Next, Chapter 6 aims to discuss the obtained results and what it could mean to the effectiveness of a cascade control implementation on the Quanser Aero. Then, some options in what could be done in a possible continuation of the subject will be discussed.

Lastly, Chapter 7 will summarize the report and conclude it.

1.2 Single-loop feedback control

1.2 Single-loop feedback control

Feedback control, as already mentioned, is at its core a control method that involves using a feedback loop to generate an error signal that corrects the output into something more desirable. The most simple kind of feedback control is the single-loop feedback control, which is shown in Fig. 1.1.

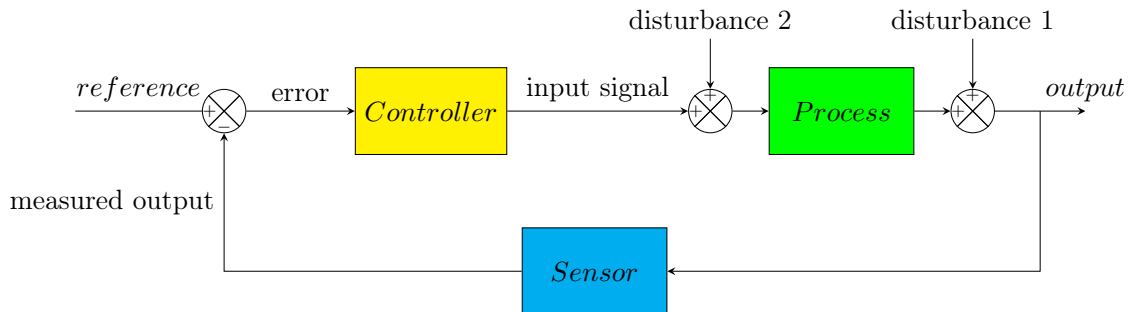


Figure 1.1: Generic block diagram for feedback control

In this report, the various kinds of outputs, inputs and blocks demonstrated in the figure will alternatively be referred to as $r(t)$ for reference, $e(t)$ for error, $u(t)$ for input signal, $y(t)$ for output, $y_m(t)$ for measured output, $d_1(t)$ for disturbance 1, $d_2(t)$ for disturbance 2, $C(s)$ for controller and $P(s)$ for process. Typically, unity gain feedback is assumed, that is $\text{Sensor} = 1 \Rightarrow y(t) = y_m(t)$, it will be in this report as well.

Ignoring the disturbances, which are unwanted elements, such a feedback loop can be expressed in the Laplace domain as:

$$y(s) = \frac{C(s)P(s)}{1 + C(s)P(s)}r(s) \quad (1.1)$$

The noted 'disturbances' are undesired, unaccounted for inputs which increase the error of the system. Increasing robustness against such disturbances is the main purpose of feedback control. More specifically, through having a feedback loop that responds to unexpected developments in the output, the system can automatically correct itself against those developments. Mathematically, the output with a disturbance can be expressed as:

1.3 The PID controller

$$y(s) = d_1(s) + P(s)(e(s)C(s) + d_2(s)) \quad (1.2)$$

Given that $e(t) = r(t) - y(t)$, and assuming $r(t) = 0$ and $d_1(t) = 0$, we can derive a transfer function between $Y(s)$ and $d_2(s)$ as follows:

$$y(s) = P(s)(-y(s)C(s) + d_1(s)) \quad y(s) + y(s)P(s)C(s) = d_1(s)P(s)$$

$$\frac{y(s)}{d_1(s)} = \frac{P(s)}{1 + P(s)C(s)} \quad (1.3)$$

$$y(s) = d_2(s) - P(s)y(s)C(s) \quad y(s) + y(s)P(s)C(s) = d_2(s)$$

$$\frac{y(s)}{d_2(s)} = \frac{1}{1 + P(s)C(s)} \quad (1.4)$$

To achieve the desired output, it is necessary for the user to manipulate the controller. The purpose of the controller is to translate the error into a proper corrective action for the process, and is thus an essential part of any feedback system. For the controller to actually do so, it needs to be properly tuned according to behavior of the rest of the system.

1.3 The PID controller

There are many methods for making a controller, the PID controller being by far the most common [7]. In a PID controller, there are 3 primary terms: The proportional gain K_P , the integral gain K_I , and the derivative gain K_D . The output of the controller can be expressed as shown in Eq. 1.5 as Eq. 1.6 in the Laplace domain, where $K_I = \frac{K_P}{\tau_I}$ and $K_D = K_P\tau_D$. A block diagram representation of this is shown in Fig. 1.2.

$$u(t) = K_P e(t) + K_I \int e(t)dt + K_D \frac{de(t)}{dt} = K_P \left(e(t) + \frac{\int e(t)dt}{\tau_I} + \tau_D \frac{de(t)}{dt} \right) \quad (1.5)$$

1.3 The PID controller

$$u(s) = \left(K_P + \frac{K_I}{s} + K_D s\right)e(s) = K_P \left(1 + \frac{1}{\tau_I s} + \tau_D s\right)e(s) \quad (1.6)$$

In the case of the PID controller, tuning it to a specific system is done by adjusting these parameters. While it is possible to tune manually by continually testing and changing values to achieve a sufficient response according to Table 1.1, it is typically regarded as better practice to utilize a specific tuning method. There are many different ways to do so, and as stated previously, this report will utilize several such tuning methods.

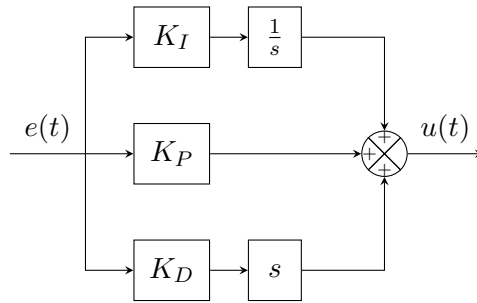


Figure 1.2: PID-controller diagram

Table 1.1: Manual tuning guidance table

Parameter	Rise time	Overshoot	Settling Time	Steady state error	Stability
K_P	Decrease	Increase	Small change	Decrease	Degrade
K_I	Decrease	Increase	Increase	Eliminate	Degrade
K_D	Minor change	Decrease	Decrease	No effect	Improve if K_D is small

It is important to note that the derivative gain K_D will amplify high-frequency measurement noise. Thus, it is usually necessary to add some kind of filter to the PID controller. The simplest way to implement such a filter is by adding a simple low-pass filter, shown in Eq. 1.7 to the derivative part, resulting in the Laplace-domain controller output of Eq. 1.8.

$$T_f = \frac{1}{1 + \tau_f s} \quad (1.7)$$

1.4 Cascade control

$$u(s) = \left(K_P + \frac{K_I}{s} + \frac{K_D s}{1 + \tau_f s} \right) e(s) \quad (1.8)$$

Where the filter time constant is usually defined as $\tau_f = \alpha K_D$, α being a user decided constant, usually in the range $\alpha \in [0.05, 0.2]$. All PID controllers in this report will include such a filter with $\alpha = 0.1$.

In the case of the derivative gain is not desired, it is also possible to utilize PI controller, which can be expressed in the Laplace domain as as shown in Eq. 1.9. If the integral gain is not desired either, a P controller is also possible.

$$u(s) = K_P \left(1 + \frac{1}{\tau_I s} + \tau_D s \right) e(s) \quad (1.9)$$

1.4 Cascade control

As mentioned, cascade control is feedback control with two or more nested feedback loops. A basic diagram demonstrating this is shown in Fig. 1.3.

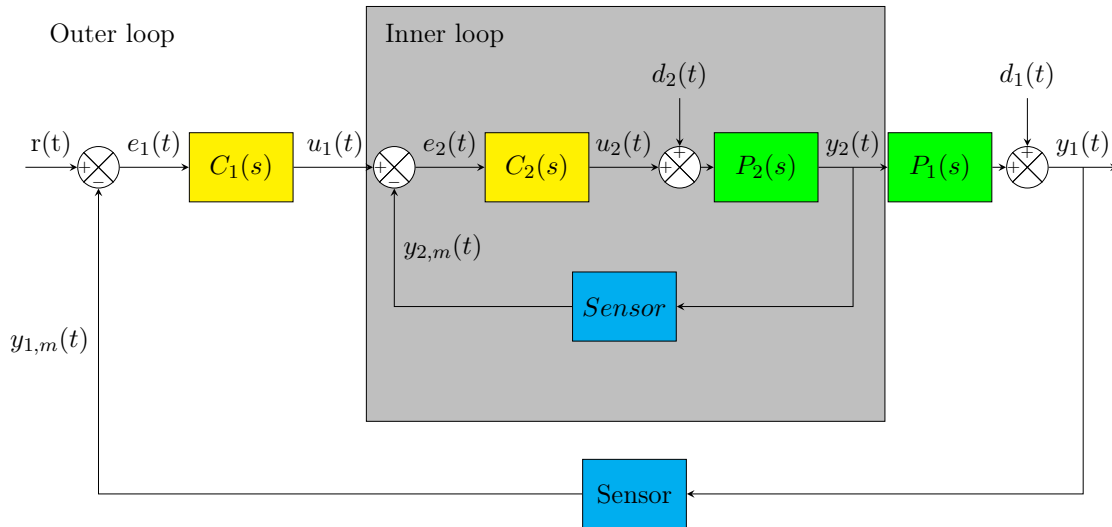


Figure 1.3: Block diagram for basic feedback control. 1 and 2 refer to whether it belongs to the outer loop (1) or inner loop (2). It otherwise follows the same terminology as the feedback scheme.

1.4 Cascade control

As seen, this configuration uses two loops, referred to as the inner and outer loops or the secondary and primary loops. Each loop is outfitted with its own sensor, process and controller. They both naturally also have each their own error, expressed as $e_1(t) = r(t) - y_1(t)$ for the outer loop and $e_2(t) = u_1(t) - y_2(t)$ for the inner loop.

Following the logic that feedback control reduces the effect of disturbances, cascade control would theoretically add another layer of robustness against disturbances. The general idea is that the inner loop will have already corrected much of the disturbances by the time outer loop completes a cycle, reducing the amount of stress on the outer loop.

And given that:

$$\begin{aligned} e_1(s) &= r_1(s) - y_1(s) \\ e_2(s) &= u_1(s) - y_2(s) = e_1(s)C_1(s) - y_2(s) = (r_1 - y_1)C_1 - y_2(s) = -y_1(s)C_1(s) - y_2(s) \\ y_2(s) &= \frac{y_1(s)}{P_1(s)} \end{aligned}$$

$$\begin{aligned} y_1(s) &= d_1(s) + P_1(s)(e_2(s)C_2(s)P_2(s) + d_2(s)) \\ y_1(s) &= P_1(s)P_2(s)((-y_1(s)C_1(s) - y_1(s))C_2(s) + d_2(s)) \\ y_1(s) &= -P_1(s)P_2(s)y_1(s)C_1(s)C_2(s) - P_1(s)P_2(s)y_1(s)C_2(s) + d_2(s)P_1(s)P_2(s) \\ y_1(s) + y_1(s)P_1(s)P_2(s)C_1(s)C_2(s) + y_1(s)P_1(s)P_2(s)C_2(s) &= d_2(s)P_1(s)P_2(s) \\ y_1(s)(1 + P_1(s)P_2(s)C_1(s)C_2(s) + P_1(s)P_2(s)C_2(s)) &= d_2(s)P_1(s)P_2(s) \end{aligned}$$

Finally resulting in a transfer function between the output and the disturbance:

$$\frac{y_1(s)}{d_1(s)} = \frac{P_1(s)}{1 + P_1(s)P_2(s)C_1(s)C_2(s) + P_1(s)P_2(s)C_2(s)} \quad (1.10)$$

This can be directly compared with the transfer function from normal feedback control $\frac{y(s)}{d_1(s)} = \frac{P(s)}{1+P(s)C(s)}$. This means that if $C_2 > 1$, the denominator of the cascade control system is strictly larger than that of the ordinary feedback system, meaning the gain of the transfer function is strictly smaller. Intuitively, the smaller the transfer function between the disturbance and the output is, the smaller the effect the disturbance will have on the output. Therefore, any disruptions acting in the inner loop should be reduced in a cascade control configuration. Any disruption in the outer loop however, such as the the disturbance d_1 , should not be especially reduced by the cascade control configuration.

In this report, both controllers in the cascade control system will be PID controllers. PID tuning in cascade control can be achieved through two primary methods: sequential and

1.5 Integral windup and clamping

simultaneous. As the names imply, they revolve around tuning controllers in successive order or at the same time, respectively. Sequential tuning utilizes largely the same tuning methods as regular feedback control, while simultaneous tuning requires its own methods entirely. Simultaneous tuning can prove to be more complex in implementation, but will likely save time compared to sequential tuning.

1.5 Integral windup and clamping

In PID control, integral windup is a common issue. When a system is outfitted with some kind of saturation that limits the process input to $u_{min} < u(t) < u_{max}$, having an integral component in a controller, which a PID controller does, can cause significant overshoot in the response. More specifically, even if a signal becomes greater than u_{max} and is saturated to a constant, the integral term will continue building up. Then, once the system has gone past its reference point and needs to slow down the output, the built up integral term will prevent the system from doing so immediately. This causes undesirable overshoot, reducing the accuracy of the system. According to Visioli [12], this is Especially important to watch out for when it comes to cascade control.

There are several possible anti-windup methods to minimize the effects of this, one of which is clamping. Clamping is a conceptually simple method that consists of disabling the integral buildup once the system reaches saturation, which can be achieved by a variety of means. One possible implementation of this is seen in Fig. 1.4.

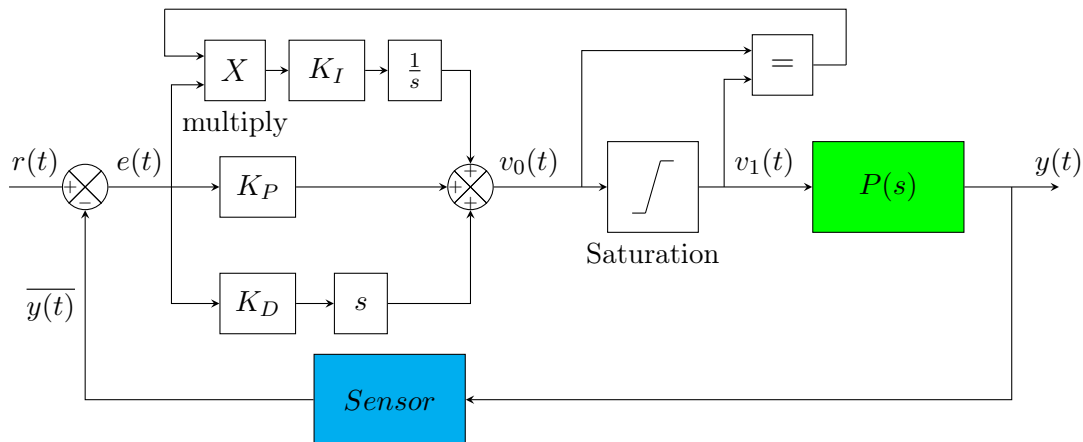


Figure 1.4: Basic block diagram for clamping

1.6 Integral performance indices

As seen, clamping is accomplished through comparing the input and output of a saturation block, v_0 and v_1 , and multiplying the result by the input to the integral gain. Thus, if $v_0 \neq v_1$, the integral gain input signal will be set as $e_I(t) \times 0 = 0$. With this method the integrator will only be active when the voltage is not being saturated.

In the case of the Quanser Aero, limits of the input voltages of each propeller are $-24V < v(t) < 24V$. The aero will automatically saturate the input signals to achieve these voltages, which makes the systems vulnerable to integral windup. To steel the system against this, the clamping method described above will be utilized in every test in this report. However, clamping and saturation will be largely omitted from test descriptions to avoid excessive clutter.

1.6 Integral performance indices

The integral performance indices IAE (Integral Absolute Error), ITAE (Integral Time Absolute Error), ISE (Integral Square Error) and ITSE (Integral Time Square Error) are often used in quantitative evaluation of the performance of control systems. In this report, these indices will be used for precisely that.

As the names imply, the indices are all based on the error, expressed as $IAE = \int |e(t)|dt$, $ITAE = \int t|e(t)|dt$, $ISE = \int e(t)^2dt$ and $ITSE = \int te(t)^2dt$. Due to the nature of integration, what all these indices accomplish is to add together accumulated error over the course of the experiment. Since error is something a system typically aims to keep as low as possible, one can compare the relative quality of two control systems by how low the integral indices are. Despite being similar, they fulfill slightly different niches. In the case of ISE and ITSE, the fact that they square the error before integrating gives them a greater emphasis on large spikes in error such as overshoot. In the case of ITAE and ITSE, the multiplication by time puts greater emphasis on later portions of the error where t is greater such as steady state or the disturbances.

Chapter 2

Description of the Quanser Aero

The Quanser Aero, shown in Fig. 2.1, is a tool designed for experiments in control theory in education or research. It somewhat resembles rotorcraft, though it operates on at most two degrees of freedom and is mounted to the ground.

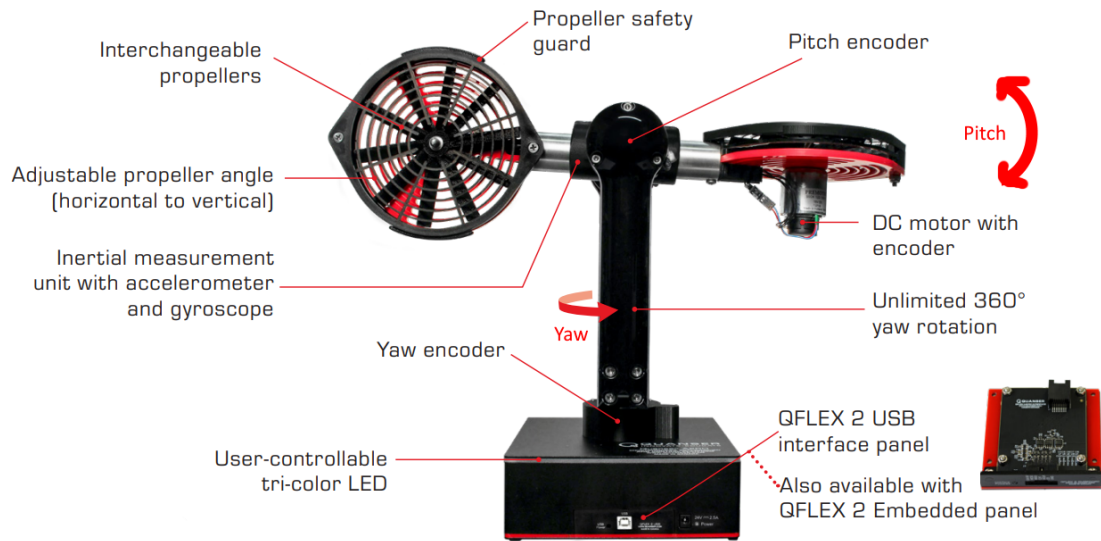


Figure 2.1: Data sheet image of Quanser Aero

2.1 General description

2.1 General description

Most of the Aero's features are represented in Fig. 2.1. As the image implies, the Aero can rotate across the yaw and pitch axes. The yaw angle is designed to rotate infinitely, while the pitch angle is limited to 124° (62° in each direction). The respective wings are known as the 'pitch' or 'front' wing versus 'yaw' and 'back' wings on and come with their own DC motor and propeller. Each propeller can also be adjusted on the roll axis using an allen key. The pitch and yaw angles of the Aero can also be individually locked to simulate 1DOF. In this report, the yaw angle will be locked and the yaw motor will be unused, resulting in a '1DOF helicopter mode'.

The Aero also comes with various built-in sensors, including a tachometer to measure propeller speeds, high-resolution optical sensors to measure pitch and yaw angles, a gyroscope for angular velocity, an accelerometer for angular acceleration, and an integrated current sensor. The Aero can be interacted with using a USB connection and simulink's various HIL initialize, HIL read and HIL write blocks. This allows the user to set input voltages, lock the pitch and yaw axes, set LED coloring and read the various sensors. As noted earlier, the input voltage is locked at a range of $-24V < x < 24V$, and will automatically saturate inputs outside this range.

In addition, the propellers of the Aero can be freely removed and replaced. In the UiS laboratory, there are two pairs of propellers available, which greatly differ in how much they are affected by disturbance. Comparing results obtained with each pair of propellers allows much more rigorous analysis of how well a system rejects disturbance. For this reason, all testing will be done for both propellers. The propellers can be seen in Fig. 2.2 and 2.3.



Figure 2.2: High efficiency propeller[10]

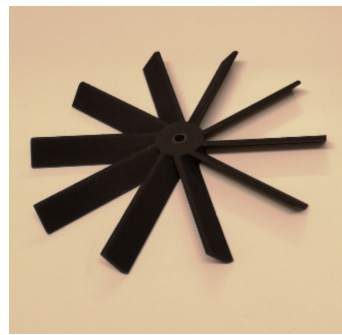


Figure 2.3: Low efficiency propeller[10]

2.2 Modeling

A free body diagram of the Aero in 2DOF helicopter mode can be found in their courseware for the Quanser AERO [1] and is shown in Fig. 2.4.

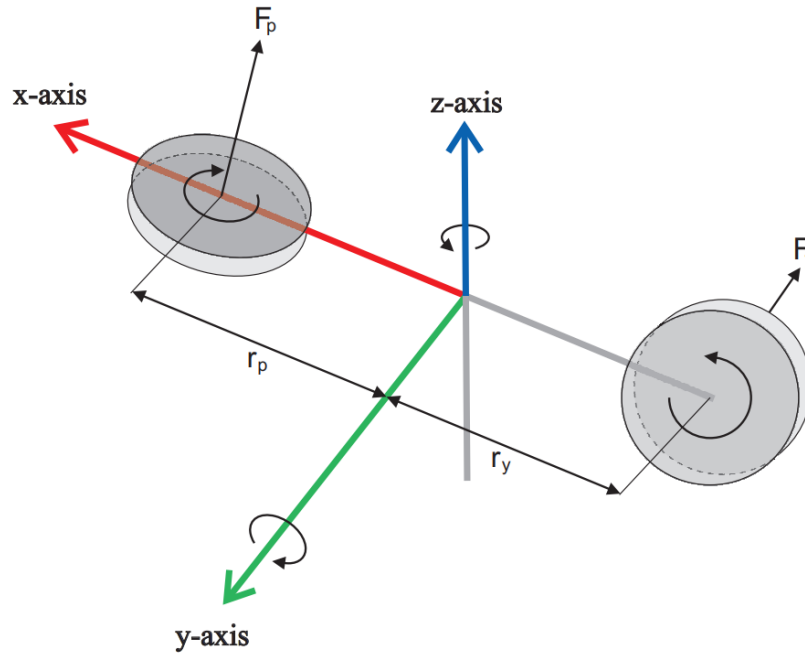


Figure 2.4: Free-body diagram of the 2DOF helicopter mode Quanser Aero

The rotation of the Aero in each axis is defined by variables ψ (yaw) and θ (pitch). How the Aero rotates around the axis depends on the thrust forces $F_p(t)$ and $F_y(t)$ acting perpendicularly to the propeller at distances r_p and r_y from the y -axis. Meanwhile the thrust forces are defined by propeller speeds ω_p and ω_y , which are expressions of the user's input voltages V_p and V_y .

The torques of each axis can be expressed as:

$$\tau_p = K_{pp}V_p + K_{py}V_y \quad (2.1)$$

2.2 Modeling

$$\tau_y = K_{yp}V_p + K_{yy}V_y \quad (2.2)$$

Through Euler-Lagrange formulation, nonlinear dynamic equations for the pitch and yaw motions for the Aero in 2DOF helicopter configuration, are found as Eq. (2.3) and (2.4) [2].

$$(J_p + m_h l_{cm}^2)\ddot{\theta} + D_p\dot{\theta} + m_h l_{cm}^2 \dot{\psi}^2 \sin(\theta)\cos(\theta) + m_h g l_{cm}^2 \cos(\theta) = K_{pp}V_p + K_{py}V_y \quad (2.3)$$

$$(J_y + m_h l_{cm}^2 \cos(\theta)^2)\ddot{\psi} + D_y\dot{\psi} + 2m_h l_{cm}^2 \sin(\theta)\cos(\theta)\dot{\theta}\dot{\psi} = K_{yp}V_p + K_{yy}V_y \quad (2.4)$$

Where the parameters are as defined in Tab. 2.1.

Table 2.1: 2DOF helicopter parameters

	Parameter	Value	Unit
J_p	Moment of Inertia about the pitch axis		$kg \cdot m^2$
J_y	Moment of Inertia about the yaw axis		$kg \cdot m^2$
D_p	Pitch viscous friction constant		N/V
D_y	Yaw viscous friction constant		N/V
K_{pp}	Torque thrust gain acting on pitch axis from pitch propeller		$N \cdot m/V$
K_{py}	Torque thrust gain acting on pitch axis from yaw propeller		$N \cdot m/V$
K_{yp}	Torque thrust gain acting on yaw axis from pitch propeller		$N \cdot m/V$
K_{yy}	Torque thrust gain acting on yaw axis from yaw propeller		$N \cdot m/V$
l_{cm}	Center of mass distance from the body-fixed frame origin		m
m_h	Mass of the Aero body		kg

By selecting the state vector and the input vector as shown in Eq. 2.10 Eq. 2.9 the state space equation in Eq. 2.7 was derived.

2.2 Modeling

$$X = \begin{bmatrix} \theta \\ \psi \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \quad (2.5)$$

$$U = \begin{bmatrix} V_p \\ V_y \end{bmatrix} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \quad (2.6)$$

$$\dot{X} = \begin{bmatrix} x_3 \\ x_4 \\ \frac{K_{pp}u_1 + K_{py}u_2 - D_p x_3 - m_h l_{cm}^2 x_4^2 \sin(x_1) \cos(x_1) - m_h g l_{cm}^2 \cos(x_1)}{J_p + m_h l_{cm}^2} \\ \frac{K_{yp}u_1 + K_{yy}u_2 - D_y x_4 - 2m_h l_{cm}^2 \sin(x_1) \cos(x_1) x_3 x_4}{J_y + m_h l_{cm}^2 \cos(x_1)^2} \end{bmatrix} \quad (2.7)$$

In 1DOF helicopter mode, the yaw motor is locked and disabled, meaning $\psi, \dot{\psi}, \ddot{\psi}, K_{\theta\psi}, K_{\psi\theta}$ and $K_{\psi\psi} = 0$. Considering this, the dynamic equation for the pitch is found as Eq. 2.8, the state vector as Eq. ??, the input as Eq. ?? and state space representation as Eq. 2.11.

$$(J_p + m_h l_{cm}^2) \ddot{\theta} + D_p \dot{\theta} + m_h g l_{cm}^2 \cos(\theta) = K_{pp} V_p \quad (2.8)$$

$$X = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (2.9)$$

$$U = V_p = u_1 \quad (2.10)$$

$$[\dot{X}] = \begin{bmatrix} x_2 \\ \frac{K_{pp}u_1 - D_\theta x_2 - m_h g l_{cm}^2 \cos(x_1)}{J_p + m_h l_{cm}^2} \end{bmatrix} \quad (2.11)$$

2.3 Process behavior

2.3 Process behavior

Most of relevant behaviors of the Quanser Aero can be obtained from the open loop step responses of each output $\omega(t)$ and $\phi(t)$, respectively, shown in Fig. 2.5 and 2.6 for the efficient propellers and Fig. 2.7 and 2.8 for the inefficient propellers.

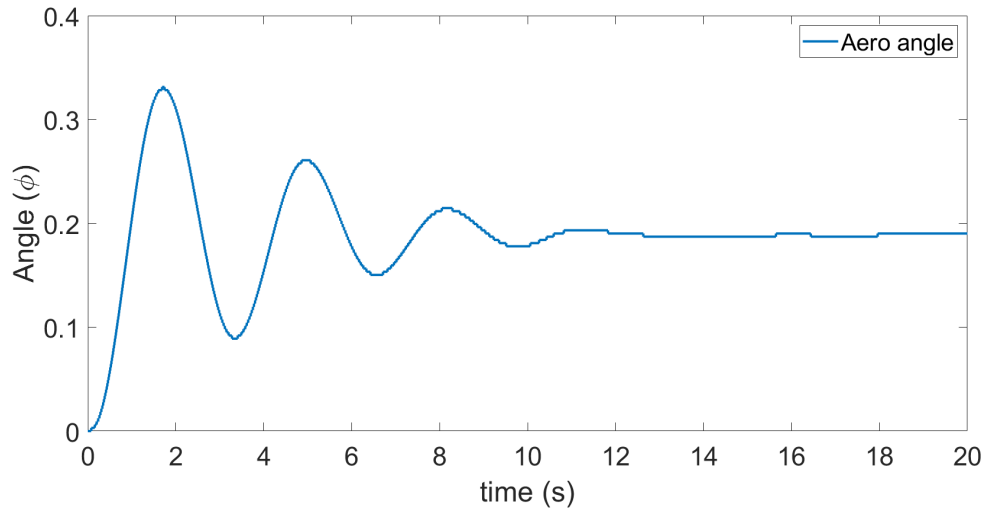


Figure 2.5: Open loop step response of $\phi(t)$, efficient propellers

2.3 Process behavior

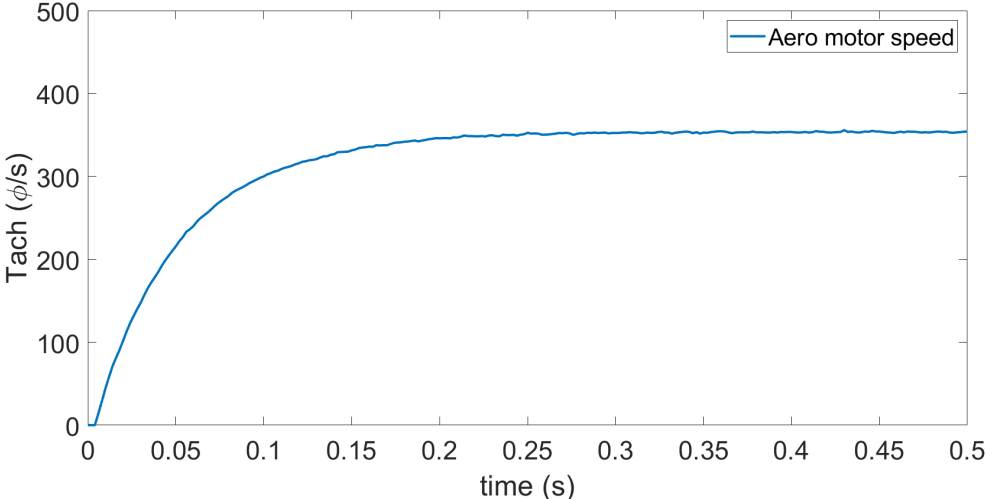


Figure 2.6: Open loop step response of $\omega(t)$, efficient propellers

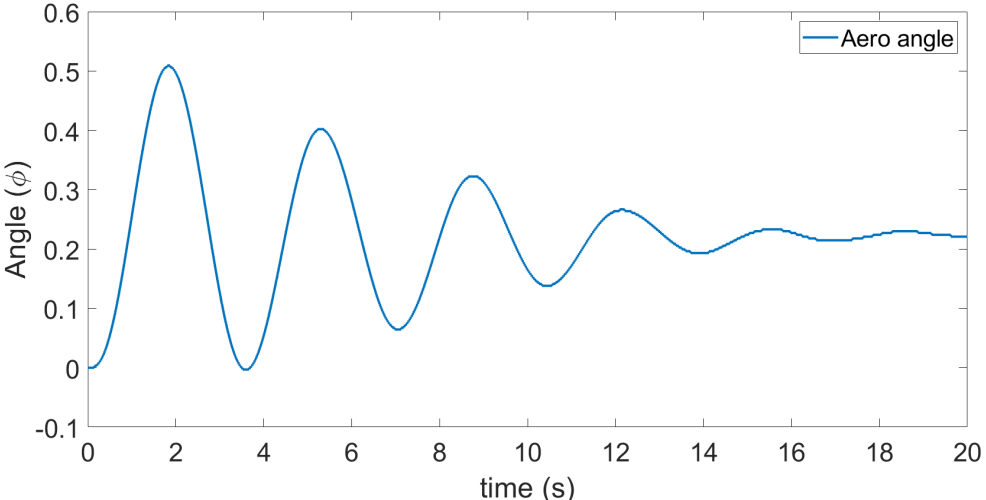


Figure 2.7: Open loop step response of $\phi(t)$, inefficient propellers

2.3 Process behavior

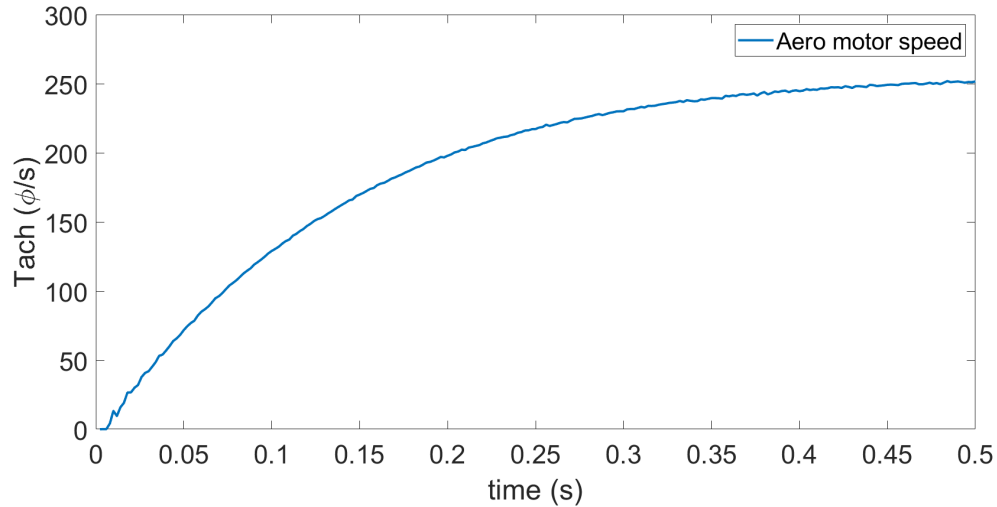


Figure 2.8: Open loop step response of $\omega(t)$, inefficient propellers

As the step responses show, there is little difference in the overall behavior of the the different propellers. For both propeller types, it can be observed that both outputs converge to a specific value. This means that neither loop is unstable or integrating. As far as the inner loop goes, it can also be observed that the overall behavior of the process seems to largely resemble a first order transfer function. Meanwhile, considering the outer loop process is clearly underdamped, it is better described by function of second order or higher.

Besides that, it should be noted that the process speed of $\omega(t)$, and thus the dynamics of the inner loop, is several times faster than $\phi(t)$. As noted in the introduction, according to Visioli and Antonio [12], this means the cascade control system should have improved stability characteristics and allows for greater gain in the primary loop.

Chapter 3

Tuning methods

3.1 Single-loop tuning methods

3.1.1 Ziegler-Nichols closed loop method

The Ziegler-Nichols closed loop method is a particularly not well known PID tuning method. A basic scheme to represent the method is shown in Fig. 3.1 , while the scheme's subsystem PIDX is detailed in Fig. 3.2. To avoid clutter in block diagrams, the PIDX subsystem will be used several more times in the report, indicated by the controller in the diagram being replaced with "PIDX".

3.1 Single-loop tuning methods

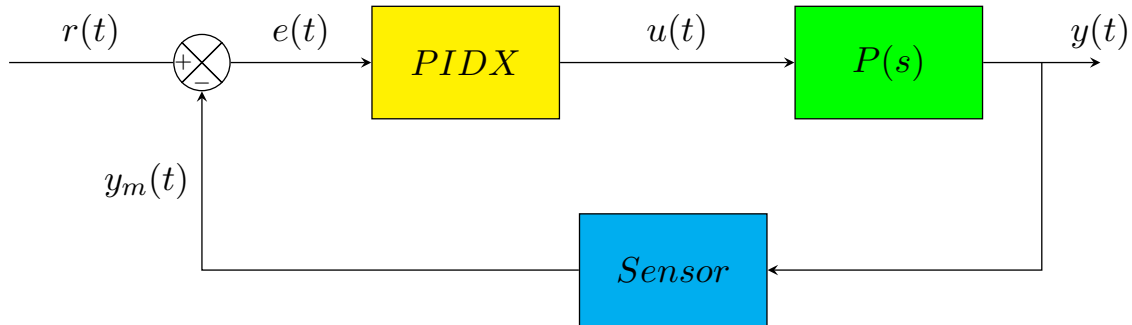


Figure 3.1: Basic block diagram for Ziegler-Nichols closed-loop method

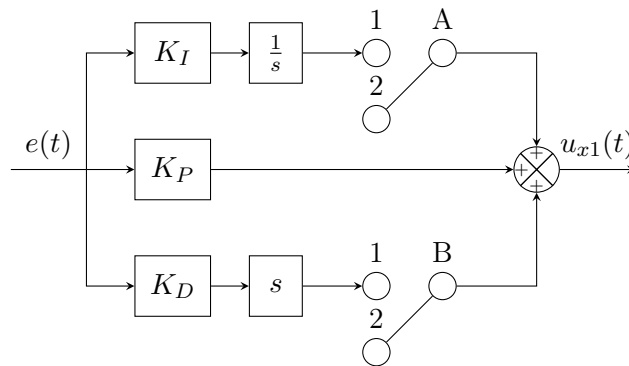


Figure 3.2: PIDx: PID-controller diagram for Ziegler-Nichols method. It is identical to a regular PID controller, except it features switches to enable and disable the derivative and integral gains

To begin testing, PIDx's switch A and B must both be set to position 2. This sets the controller to proportional gain only. Afterwards, K_P must be increased until the system response reaches marginal stability. Since perfect precision is unnecessary, a response with approximate marginal stability works fine as well. From the marginally stable response, the ultimate gain K_U and the ultimate period T_U are then found as the current K_P and period of the resulting oscillations, respectively. Thereafter, the parameters can be easily computed through Table 3.1. Once the parameters are applied and the switches are set to 1, the tuning is finished.

3.1 Single-loop tuning methods

Table 3.1: Ziegler-Nichols PID tuning table, where K_U = ultimate gain and T_U = ultimate frequency

Control Type	K_P	K_I	K_D
P	$0.5K_U$	0	0
PI	$0.45K_U$	$0.54K_U/T_U$	0
PID	$0.6K_U$	$1.2K_U/T_U$	$0.075K_U T_U$

3.1.2 Standard relay-feedback method

The relay feedback method is another common tuning method. A basic diagram is shown in Fig. 3.3.

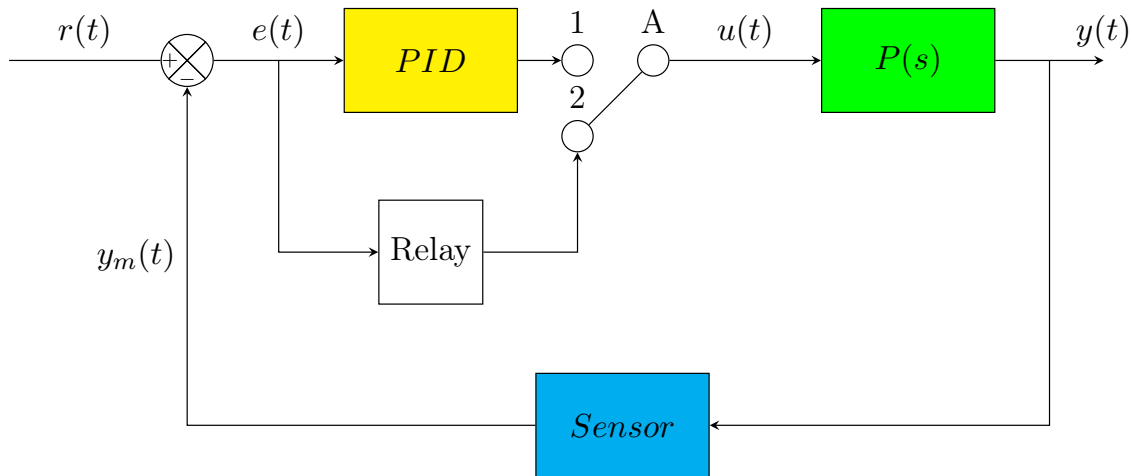


Figure 3.3: Basic block diagram for standard relay-feedback method

To start tuning, switch A must be set to position 2. This replaces the controller with a symmetrical relay of amplitude h . Similarly to the Ziegler-Nichols method, this method requires finding an ultimate gain K_U and an ultimate period T_U . To begin, the amplitude of the relay needs to be increased until continual oscillations are obtained in the response. The oscillations will perhaps have a changing amplitude at first, but if h is sufficient

3.1 Single-loop tuning methods

will converge to marginal stability at $t \rightarrow \infty$. Preferably, measurements of K_U and T_U should be done when the output is as close to marginal stability as possible. Since perfect precision is unnecessary, it can be assumed $\overline{A_{y,marginal}} = A_{y,marginal}$ (where $A_{y,marginal}$ is the amplitude) after an arbitrary, user-decided period of time. After selecting the usable time range, the ultimate gain can be computed according to the formula in Eq. 3.1, where $A = A_{y,marginal}$. Meanwhile, T_U can be found as the period of the oscillations. Then, the parameters can be set and the switch turned back to position 1, resulting in a tuned system.

$$K_U = \frac{4h}{A\pi} \quad (3.1)$$

Once the values have been obtained, the parameters can be computed through the same computational methods as Ziegler-Nichols, shown in Table 3.1.

3.1.3 Ziegler-Nichols open-loop method

Ziegler-Nichols open-loop method is a particularly simple method, initially proposed by J.G Ziegler and N.B Nichols in 1942 [14]. A simplification of the method was provided in Damiano Rotondo's lecture notes [9]. A basic diagram for execution of the method is demonstrated in Fig. 3.4.

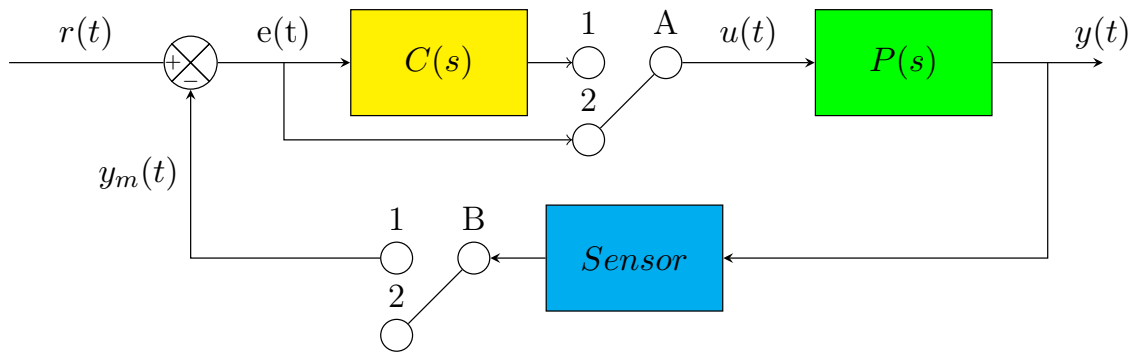


Figure 3.4: Basic block diagram for Ziegler-Nichols open-loop feedback method

To begin with, switch A and switch B both need to be in position 2, which ensures that

3.2 Sequential cascade control tuning methods

the system is in open-loop and that the reference is not unaffected by the controller, hence $u(t) = r(t)$. Then the reference needs to excite the process with a simple step input $r(t) = U \times 1(t)$, where $1(t)$ is the unit step signal shown in Eq. 3.2. From the output of this, the necessary parameters L and R can be obtained. R can be found as the slope of the response's steepest tangent $T = Rt$. L is the dead time, defined as the time $L = t_1 - t_0$ between the step time t_0 and the time of intersection between the steepest tangent T and the x-axis t_1 . The PID parameters of the controller can then be computed using Table 3.2. Setting the controller parameters both switches to 1 should then result in a tuned feedback system.

$$1(t) = \begin{cases} 0 & t < t_0 \\ 1 & t \geq t_0 \end{cases} \quad (3.2)$$

Table 3.2: Table for calculation of Ziegler-Nichols open-loop PID parameters

Controller type	K_P	K_I	K_D
P	$\frac{U}{LR}$	0	0
PI	$\frac{0.9U}{LR}$	$\frac{0.27U}{RL^2}$	0
PID	$\frac{1.2U}{LR}$	$\frac{0.6U}{RL^2}$	$\frac{0.6U}{R}$

3.2 Sequential cascade control tuning methods

As already mentioned, tuning methods that work with normal feedback control can theoretically also work with cascade control systems by using sequential tuning. To do so effectively, tuning should be done first on the secondary controller with the primary loop disabled, and then on the primary controller [12]. Naturally, tuning this way takes a significant amount of time. Specifically how this can be applied will be covered in section 3.3.

3.3 Simultaneous cascade tuning using step input

3.3 Simultaneous cascade tuning using step input

A method for simultaneous tuning of controllers, which only requires a single step input, is presented by Visioli and Piazzzi [13]. A basic diagram for the method is presented in Fig. 3.5. The paper features specific methods on how to arrive at the tuned controllers, but in practice the core concept allows for much freedom in its execution. The core concept in question is applying a step input directly to the processes in open loop and using the step responses y_2 and y_1 to obtain models for the processes P_2 and P_1 . These models should be in the form of first order plus dead time (FOPDT), seen in Eq. 3.3 or second order plus dead time (SOPDT) transfer functions, seen in Eq. 3.4 and 3.5. Once the transfer functions for the processes have been found, many methods can be used to tune C_1 and C_2 .

$$T(s) = \frac{K}{\tau s + 1} e^{-Ls} \quad (FOPDT) \quad (3.3)$$

$$T(s) = \frac{K}{(\tau_1 s + 1)(\tau_2 s + 1)} e^{-Ls} \quad (SOPDT) \quad (3.4)$$

$$T(s) = \frac{K}{\tau^2 s^2 + 2\xi\tau s + 1} e^{-Ls} \quad (SOPDT) \quad (3.5)$$

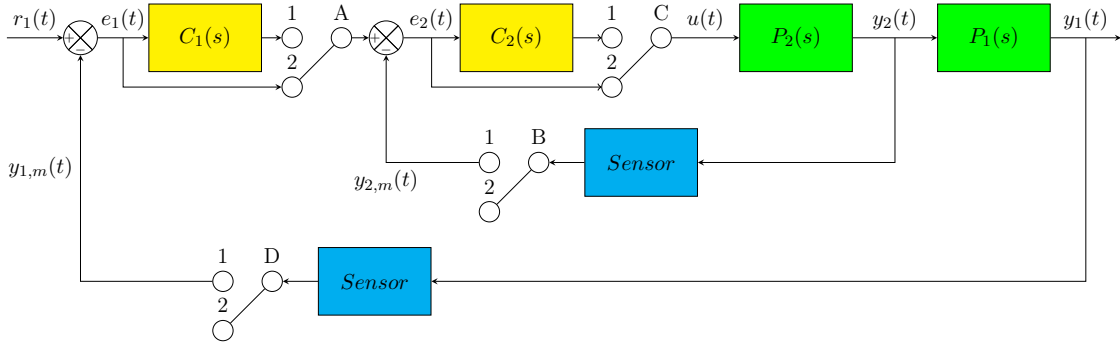


Figure 3.5: Basic block diagram for simultaneous step response method

To begin, all switches must be set to position 2, so that the system is in open loop and ignores the controllers. Then, the user needs to send a step input signal to P_2 and read

3.3 Simultaneous cascade tuning using step input

the responses y_1 and y_2 . From the step input of r and step response y_2 , any method that uses the step response to determine a low-order model can be used to find process P_1 . Finding a model of the process P_2 can be slightly more complicated since its input, y_2 , is a step response rather than a step or sinusoidal input. Therefore, only methods that can determine a model from a variable input and its output can be used to determine a model for P_2 . If the resulting model is high order, some kind of model reduction is necessary. From this point, two types of approaches are possible:

Firstly, it is possible to tune the controllers from just the models of P_1 and P_2 , assuming the method is adjusted to account for cascade structure. This approach is simple, but must be specifically tailored, which leaves a relatively small selection.

The second approach involves a much broader selection of methods. It is possible to use regular FOPDT or SOPDT model based tuning methods by first tuning the secondary controller using any such method and deriving from it the controller transfer function:

$$C_2 = \frac{K_D s^2 + K_P s + K_I}{s} \quad (3.6)$$

Then, the overall transfer function of the inner loop in series with the primary process can be determined as:

$$P_T(s) = \frac{P_1(s)P_2(s)C_2(s)}{1 + P_2(s)C_2(s)} \quad (3.7)$$

Then the transfer function needs to be reduced to an FOPDT or SOPDT transfer function. If the model of the process P_1 is a higher order function, such as those gained from the proposed least-squares method, then the model reduction can wait until after P_T is found.

Antonio and Piazzzi [13] recommend using the area method [12] to determine a FOPDT of $P_D(s)$. Then, an arbitrarily high order transfer function of P_1 is determined using a least-squares based method such as the one in Sung et al [11], which is then reduced to a FOPDT model using a least-squares reduction method. Then, the second approach is followed and the controllers are tuned using the Kappa-Tau method due to supposedly greater disturbance rejection.

3.3 Simultaneous cascade tuning using step input

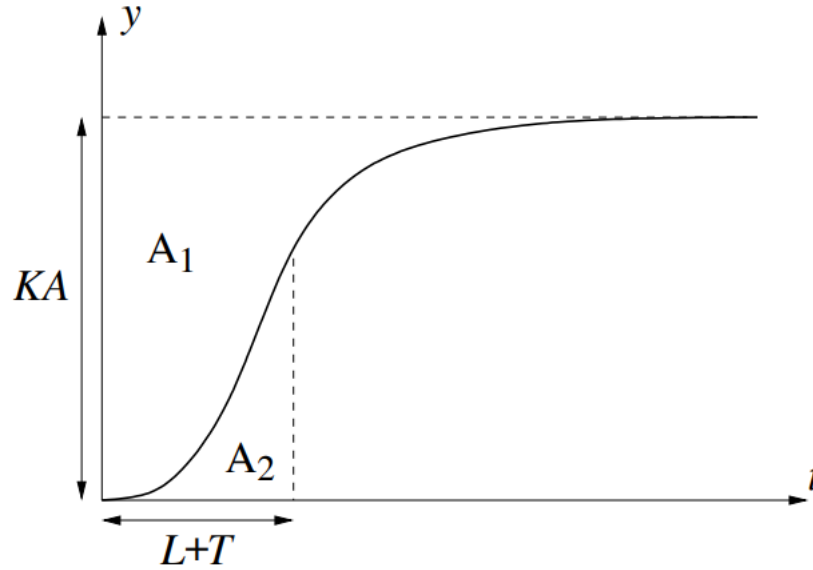


Figure 3.6: Visual representation of the area method

3.3.1 The area method

The area method is a relatively simple method for finding a FOPDT model of a process. A demonstration of the method is presented by Visioli [12], where it is visualized as follows:

As already noted, the area method revolves around applying a step input $r(t) = U \cdot 1(t)$ and reading the step output $y(t)$. To execute the method, it is necessary that the $y(t)$ is in steady state before the step input is applied.

To begin with, the gain K can be determined as the relation between the steady state value after the step input y_{ss} and the step input magnitude U :

$$K = y_{ss}/U \quad (3.8)$$

Then, the area between the steady state and the step response from the step input time t_0 can be computed as:

Then, the areas A_1 and A_2 can be computed as:

3.3 Simultaneous cascade tuning using step input

$$A_1 = \int_{t_0}^{\infty} (y_{ss} - y(t))dt \quad (3.9)$$

$$A_2 = \int_{t_0}^{\frac{A_1}{K}} (y(t)y_0)dt \quad (3.10)$$

Where T_0 is the step input time and y_0 is the steady state output before the step input.

From there the dead time L and the time constant τ can be computed as:

$$\tau = \frac{eA_2}{K} \quad (3.11)$$

$$L = \frac{A_1}{K} - \tau \quad (3.12)$$

Where e refers to Euler's number.

Due to being based on integral computation, the area method can be very difficult to pull off by hand, and should preferably be executed using a digital script. It is also possible to get a negative value for L , which make the model largely unusable. On the other hand, the method is very robust to measurement noise.

3.3.2 Model estimation using least-squares

A method for identifying a higher order model of a transfer function is presented in Sung et al [11].

$$T_h(s) = \frac{n_m s^m + n_{m-1} s^{m-1} + \dots + n_1 s + n_0}{d_n s^n + d_{n-1} s^{n-1} + \dots + d_1 s + 1} \quad (3.13)$$

Considering the transfer function can be expressed as:

3.3 Simultaneous cascade tuning using step input

$$T(s) = \frac{y(s)}{u(s)} \quad (3.14)$$

The following can be derived:

$$\begin{aligned} \frac{y(s)}{u(s)} &= \frac{n_m s^m + n_{m-1} s^{m-1} + \dots + n_1 s + n_0}{d_n s^n + d_{n-1} s^{n-1} + \dots + d_1 s + 1} \\ &= \frac{n_m s^{m-n} + n_{m-1} s^{m-n-1} + \dots + n_1 s^{-n-1} + n_0 s^{-n}}{d_n + d_{n-1} s^{-1} + \dots + d_1 s^{-n+1} + 1 s^{-n}} \\ &= \frac{n_m/s^{n-m} + n_{m-1}/s^{n-m+1} + \dots + n_1/s^{n+1} + n_0/s^n}{d_n + d_{n-1}/s + \dots + d_1/s^{n-1} + 1/s^n} \\ &\quad \Rightarrow \end{aligned}$$

$$\begin{aligned} &y(s)(d_n + d_{n-1}/s + \dots + d_1/s^{n-1} + 1/s^n) \\ &= u(s)(n_m/s^{n-m} + n_{m-1}/s^{n-m+1} + \dots + n_1/s^{n+1} + n_0/s^n) \end{aligned} \quad (3.15)$$

This can be transformed into the time domain as:

$$\begin{aligned} &d_n y(t) + d_{n-1} x y(t) + \dots + d_1 x y_{n-1}(t) + x y_n(t) \\ &= n_m x u_{n-m}(t) + n_{m-1} x u_{n-m+1}(t) + \dots + n_1 x u_{n+1}(t) + n_0 x u_n(t) \end{aligned} \quad (3.16)$$

$$x y_i(t) = \int_{t_0}^t \int \int \dots \int (y(t)) dt^i \quad (3.17)$$

$$x u_i(t) = \int_{t_0}^t \int \int \dots \int (u(t)) dt^i \quad (3.18)$$

Where t_0 is the time of the input change. The equation can be used to find the following:

$$\begin{aligned} x y_n(t) &= -d_n y(t) - d_{n-1} x y(t) - \dots - d_1 x y_{n-1}(t) + \\ &n_m x u_{n-m}(t) + n_{m-1} x u_{n-m+1}(t) + \dots + n_1 x u_{n+1}(t) + n_0 x u_n(t) \\ &= [-y(t) - x y(t) - \dots - x y_{n-1}(t) x u_{n-m}(t) x u_{n-m+1}(t) \dots x u_{n+1}(t) x u_n(t)] \\ &\quad [-d_n d_{n-1} \dots - d_1 n_m n_{m-1} \dots n_1 n_0]^T \end{aligned} \quad (3.19)$$

3.3 Simultaneous cascade tuning using step input

Now, by considering all the time from t_0 to the final time t_f at discrete intervals: $t = [t_0, t_1, \dots, t_{f-1}, t_f]$, this equation can be expressed as :

$$B = Ax \quad (3.20)$$

Where:

$$B = [xy_n(t_0), xy_n(t_1), \dots, xy_n(t_{end-1}), xy_n(t_{end})]^T \quad (3.21)$$

$$A = \begin{bmatrix} -y(t_0), -xy(t_0), \dots, -xy_{n-1}(t_0), xu_{n-m}(t_0), xu_{n-m+1}(t_0), \dots, xu_{n+1}(t_0), xu_n(t_0) \\ -y(t_1), -xy(t_1), \dots, -xy_{n-1}(t_1), xu_{n-m}(t_1), xu_{n-m+1}(t_1), \dots, xu_{n+1}(t_1), xu_n(t_1) \\ \dots \\ y(t_{f-1}), xy(t_{f-1}), \dots, xy_{n-1}(t_{f-1}), xu_{n-m}(t_{f-1}), xu_{n-m+1}(t_{f-1}), \dots, xu_{n+1}(t_{f-1}), xu_n(t_{f-1}) \\ y(t_{end}), xy(t_{end}), \dots, xy_{n-1}(t_{end}), xu_{n-m}(t_{end}), xu_{n-m+1}(t_{end}), \dots, xu_{n+1}(t_{end}), xu_n(t_{end}) \end{bmatrix} \quad (3.22)$$

$$x = [-d_n, -d_{n-1}, \dots, -d_1, n_m, n_{m-1}, \dots, n_1, n_0]^T \quad (3.23)$$

Finally, by solving Eq. 3.20 for x using a least-squares procedure, all the parameters needed to find the higher order model shown in Eq. 3.13 are obtained.

It is possible to directly obtain a low order model for a process using this method, but it would not account for dead time, so this is not recommended.

3.3.3 Least-squares reduction method

Alongside the high order model estimation method, Sung et al [11] presents a least-squares based reduction method that can give either an FOPDT or SOPDT model from the arbitrarily high order transfer function $T(s)$.

Firstly, the gain can be computed as:

$$K = T_h(0) \quad (3.24)$$

3.3 Simultaneous cascade tuning using step input

Then, given that the magnitude of the SOPDT transfer function in the frequency domain can be given as:

$$|T_h(j\omega)| = \frac{K}{\sqrt{(1 - \tau^2\omega^2)^2 + (2\tau\xi\omega)^2}} \quad (3.25)$$

The following equation can be derived:

$$\tau^4|T_h(j\omega)|^2\omega^4 + (4\tau^2\xi^2 - 2\tau^2)|T_h(j\omega)|^2\omega = K^2 - |T(j\omega)|^2 \quad (3.26)$$

Setting $a = \tau^4$ and $b = 4\tau^2\xi^2 - 2\tau^2$ gives:

$$a|T_h(j\omega)|^2\omega^4 + b|T_h(j\omega)|^2\omega = K^2 - |T(j\omega)|^2 \quad (3.27)$$

Meanwhile, the ultimate frequency ω_m can be found as the frequency where $|T_h(j\omega)| = 1$, that is at $|T_h(j\omega_u)|_{dB} = 0$. If this has no solution, ω_u can be found at $|T_h(j\omega_u)|_{dB} = 20\log(K) - 3dB$. From this, a frequency vector $0 < \omega_0 < \omega_1 < \dots < \omega_i < \dots \leq \omega_u$ of arbitrary length l must be defined. Using this, Eq. 3.27 can give:

$$\begin{bmatrix} K^2 - |T_h(0)|^2 \\ K^2 - |T_h(j\omega_0)|^2 \\ K^2 - |T_h(j\omega_1)|^2 \\ \dots \\ K^2 - |T_h(j\omega_i)|^2 \\ \dots \\ K^2 - |T_h(j\omega_u)|^2 \end{bmatrix} = \begin{bmatrix} 0, 0 \\ |T_h(j\omega_0)|^2\omega_0^4, |T_h(j\omega_0)|^2\omega_0^2 \\ |T_h(j\omega_1)|^2\omega_1^4, |T_h(j\omega_1)|^2\omega_1^2 \\ \dots \\ |T_h(j\omega_i)|^2\omega_i^4, |T_h(j\omega_i)|^2\omega_i^2 \\ \dots \\ |T_h(j\omega_u)|^2\omega_u^4, |T_h(j\omega_u)|^2\omega_u^2 \end{bmatrix} [a, b] \quad (3.28)$$

Then finally, after solving Eq. 3.28 for the unknowns $[a, b]$ using a least-squares procedure, the following operations can be done to find τ , ξ and L of the SOPDT model:

$$\tau = \sqrt[4]{a} \quad (3.29)$$

3.3 Simultaneous cascade tuning using step input

$$\xi = \sqrt{\frac{b + 2\tau^2}{4\tau^2}} \quad (3.30)$$

$$L = \frac{\pi + \arctan2(-2\tau\xi\omega_u, \tau^2\omega_u^2)}{\omega_u} \quad (3.31)$$

This method can also be used to find FOPDT parameters instead, without requiring a least-squares procedure. First, the magnitude of a FOPDT transfer function in the frequency domain can be found as shown in Eq. 3.32, which at $\omega = \omega_u$ can through relatively simple math give the formula for τ in Eq. ??.

$$|T_h(j\omega)| = \frac{K}{\sqrt{1 + (\tau\omega)^2}} \quad (3.32)$$

$$\tau = \frac{\sqrt{K^2 - |T_h(j\omega_u)|^2}}{|T_h(j\omega_u)|\omega_u} \quad (3.33)$$

Then, the dead time can be found as suggested in Visioli and Antonio [13]:

$$L = -\frac{\arg(|T_h(j\omega_u)|) + \text{atan}(\omega_u\tau)}{\omega_u} \quad (3.34)$$

It is important to note that there are several ways for this reduction method to result in invalid parameters. The first issue is the formulas for the dead time L have the possibility of resulting in a negative value, which would also typically result in unusable PID parameters. In addition, in the case of the SOPDT calculations, it is possible to get complex parameters if either $a < 0$ (complex τ) or if $b < -2\tau^2$ (complex ξ). Meanwhile for the FOPDT method, if $|T_h(j\omega_u)|_{dB}$ is rising, meaning that $0 > 20\log(K)$, it will result in a complex τ . These complex parameters are not very useful for creating transfer function models, and will result in similarly unusable PID parameters.

3.3 Simultaneous cascade tuning using step input

3.3.4 Simultaneous tuning using process models

A method for the tuning of cascade controllers given models of the primary process $P_1(s)$ and the secondary process $P_2(s)$ is presented in Lee et al [5]. The paper describes methodology to tune any controller using a model, though in this report, the more interesting part is the simplification of the method in the case of FOPDT or SOPDT process models. This simplification is represented in table 3.3, where $K_I = \frac{K_P}{T_I}$, $K_D = K_P T_D$ and $L_T = L_1 + L_2$.

Table 3.3: Tuning rules for cascade controllers given FOPDT or SOPDT models of processes P_1 and P_1

Process	Process model	K_P	T_I	T_D
FOPDT	$\frac{K_2}{\tau_2 s + 1} e^{-L_2 s}$	$\frac{T_{I2}}{K_2(\lambda_2 + L_2)}$	$\tau_2 + \frac{L_2^2}{2(\lambda_2 + L_2)}$	$\frac{L_2}{6(\lambda_2 + L_2)} \left(3 - \frac{L_2}{T_{I2}}\right)$
SOPDT	$\frac{K_2}{\tau_2^2 s^2 + 2\xi_2 \tau_2 s + 1} e^{-L_2 s}$	$\frac{T_{I2}}{K_2(\lambda_2 + L_2)}$	$2\xi_2 \tau_2 + \frac{L_2^2}{2(\lambda_2 + L_2)}$	$\frac{\tau_2^2 - \frac{L_2^2}{6(\lambda_2 + L_2)}}{T_{I2}} + \frac{L_2^2}{2(\lambda_2 + L_2)}$
FOPDT	$\frac{K_1}{\tau_1 s + 1} e^{-L_1 s}$	$\frac{T_{I1}}{K_1(\lambda_1 + L_T)}$	$\tau_1 + \lambda_2 + \frac{L_T^1}{2(\lambda_1 + L_T)}$	$\frac{\tau_1 \lambda_2 - \frac{L_T^1}{6(\lambda_1 + L_T)}}{T_{I1}} + \frac{L_T^1}{2(\lambda_1 + L_T)}$
SOPDT	$\frac{K_2}{\tau_1^2 s^2 + 2\xi_1 \tau_1 s + 1} e^{-L_1 s}$	$\frac{T_{I1}}{K_1(\lambda_1 + L_T)}$	$2\tau_1 \xi_1 + \lambda_2 + \frac{L_T^1}{2(\lambda_1 + L_T)}$	$\frac{\tau_1^2 + 2\xi_1 \tau_1 \lambda_2 - \frac{L_T^1}{6(\lambda_1 + L_T)}}{T_{I1}} + \frac{L_T^1}{2(\lambda_1 + L_T)}$

In the case of PI controllers, it is recommended to simply remove the derivative action.

3.3.5 Tuning based on SOPDT or FOPDT models

Several tuning methods are simplified and shown in Panda et al [7], including a 'IMC-PID' method for tuning using FOPDT models and a 'IMC-Chien' method for tuning using SOPDT models.

FOPDT tuning using IMC-PID

The IMC-PID method is based on the Internal Model Control methodology of Rivera et al [8] and the selection of the IMC tuning parameter λ of [6]. The resulting PID controller is of a different type than the one covered in chapter 1.3, and in its laplace form is as follows:

3.3 Simultaneous cascade tuning using step input

$$PID3 = (K_P + \frac{K_I}{s} + K_D s) \left(\frac{1}{\tau_f s + 1} \right) \quad (3.35)$$

Since a filter is already included in the formula, there is no need to add any additional filter to the derivative gain. Then, the tuning rules are as shown in table 3.4 and Eq. 3.36, where $\lambda = \max(0.25L, 0.2\tau)$.

Table 3.4: IMC-PID tuning rules

Controller type	K_P	K_I	K_D
PI	$\frac{2\tau+L}{2K(\lambda)}$	$K_P \frac{1}{\tau+0.5L}$	0
PID	$\frac{2\tau+L}{2K(\lambda+L)}$	$K_P \frac{1}{\tau+0.5L}$	$K_P \frac{\tau L}{2\lambda+L}$

$$\tau_f = \frac{\lambda L}{2(\lambda + L)} \quad (3.36)$$

SOPDT tuning using IMC-Chien

The IMC-Chien method, again based on Internal Model Control [8], is presented by Chien [4]. The resulting tuning rules, based on the behavior of the model are shown in table 3.5, where again $\lambda = \max(0.25L, 0.2\tau)$.

Table 3.5: IMC-Chien tuning rules

Behavior type	Model	K_P	K_I	K_D
Overdamped	$\frac{K}{(\tau_1 s + 1)(\tau_2 s + 1)} e^{-Ls}$	$\frac{\tau_1 + \tau_2}{K(\lambda + L)}$	$K_P \frac{1}{\tau_1 + \tau_2}$	$K_P \frac{\tau_1 \tau_2}{\tau_1 + \tau_2}$
Not overdamped	$\frac{K}{\tau^2 s^2 + 2\xi\tau s + 1} e^{-Ls}$	$\frac{2\xi\tau}{K(\lambda + L)}$	$K_P \frac{1}{2\xi\tau}$	$K_P \frac{\tau}{2\xi}$

3.4 Method selection

3.3.6 Kappa-Tau tuning

Tuning into PI or PID based on a FOPDT model of a process, taken from the Kappa-Tau method presented by Åström and Hägglund [3] is presented by Visioli and Antonio [13], and shown in Table 3.6. In it, $\theta = \frac{L}{T+L}$

Table 3.6: IMC-Chien tuning rules

Controller type	Model	K_P	τ_I	τ_D
PI	$0.41e^{-0.23*\theta+0.019\theta^2} \frac{T}{KL}$	$5.7e^{1.7*\theta-0.69\theta^2} L$	0	
PID	$3.8e^{-8.4*\theta+7.3\theta^2} \frac{T}{KL}$	$5.2e^{-2.5*\theta-1.4\theta^2} L$	$0.89e^{-0.37*\theta-4.1\theta^2} L$	

3.4 Method selection

3.4.1 Single loop tuning

As described in the introduction, it is desired to do some amount of testing on a single loop control system as a point of comparison. To draw an adequate comparison, two approaches were chosen:

- Tuning C(s) to a PID controller using the Ziegler-Nichols closed-loop method.
- Tuning C(s) to a PID controller using the relay feedback method.

Tests were done with both the Ziegler-Nichols closed loop method and the relay feedback method. The following approaches will be used

3.4.2 Sequential tuning of cascade controller

As noted in the introduction, it is possible to perform any sequential cascade tuning methods by first tuning the inner loop and then the outer loop using normal single loop tuning

3.4 Method selection

methods. Unfortunately, both the Ziegler-Nichols closed loop method and the relay feedback method rely on oscillations to perform tuning. Since it has been established that the inner loop process behaves like a first order transfer function, this means that neither method is usable with the inner loop. Thus, to test either of these methods with the cascade control configuration, it is necessary to use another tuning method on the inner loop first. For that purpose, the Ziegler-Nichols open loop method will be utilized.

In addition, since the derivative gain amplifies high frequency noise, and the extremely fast moving propellers are very susceptible to this, the derivative action is largely undesired for the secondary controller. So instead, PI controller will be utilized for the inner loop in all cascade control tuning methods.

In summary, to implement sequential tuning on the cascade system, two approaches will be taken in this report:

- Tuning $C_2(s)$ to a PI controller using the Ziegler-Nichols open-loop method, followed by tuning $C_1(s)$ to a PID controller using the Ziegler-Nichols closed-loop method.
- Tuning $C_2(s)$ to a PI controller using the Ziegler-Nichols open-loop method, followed by tuning $C_1(s)$ to a PID controller using the relay feedback method for the primary controller.

3.4.3 Simultaneous cascade control tuning

The only method covered for simultaneous tuning of the cascade controller is the step response method. However, as mentioned, there are many approaches in doing this. To cover everything that was detailed the following approaches will be used:

- Determining a FOPDT model of P_2 using the area method, determining an arbitrarily high order transfer function for P_1 using the least-squares model estimation method, tuning C_2 into a PI controller using Kappa-Tau with P_2 , computing P_T , reducing P_T to a FOPDT model using the least-squares reduction method, and finally tuning C_1 into a PID controller using Kappa-Tau with P_T .
- Determining an FOPDT model of P_2 using the area method, determining an arbitrarily high order transfer function for P_1 using the least-squares model estimation

3.4 Method selection

method, tuning C_2 into a PI controller using IMC-PID with P_2 , computing P_T , reducing P_T to a SOPDT model using the least-squares reduction method, and finally tuning C_1 into a PID controller using IMC-Chien with P_T .

- Determining a FOPDT model of P_2 using the area method, determining an arbitrarily high order transfer function for P_1 using the least-squares model estimation method, reducing P_T to a FOPDT model using the least-squares reduction method, and finally tuning C_2 into a PI controller and C_1 into a PID controller using simultaneous tuning with P_2 and P_1 .
- Determining a FOPDT model of P_2 using the area method, determining an arbitrarily high order transfer function for P_1 using the least-squares model estimation method, reducing P_T to a SOPDT model using the least-squares reduction method, and finally tuning C_2 into a PI controller and C_1 into a PID controller tuning with P_2 and P_1 .

The first approach listed is the same as the one that was proposed by Visioli and Antonio [13]. For simplicity, these approaches will in this report be tentatively shortened to:

- Step response Kappa-Tau
- Step response IMC
- Step response simultaneous FOPDT plus FOPDT
- Step response simultaneous FOPDT plus SOPDT

Notably, considering the outer loop requires a transfer function of at least second order to be accurately represented, it is expected that the 'step response IMC cascade tuning' and the 'step response simultaneous FOPDT plus SOPDT cascade tuning', considering they both estimate a SOPDT model from $\phi(t)$, will perform much better than the other two others which estimate FOPDT models. Since the system is underdamped, it is also not realistic to utilize any methods which operate on the SOPDT type in Eq. 3.4, hence their absence in this report.

Chapter 4

Testing

4.1 Ziegler-Nichols closed-loop method

The model used for the tuning process is shown in Fig. 4.1.

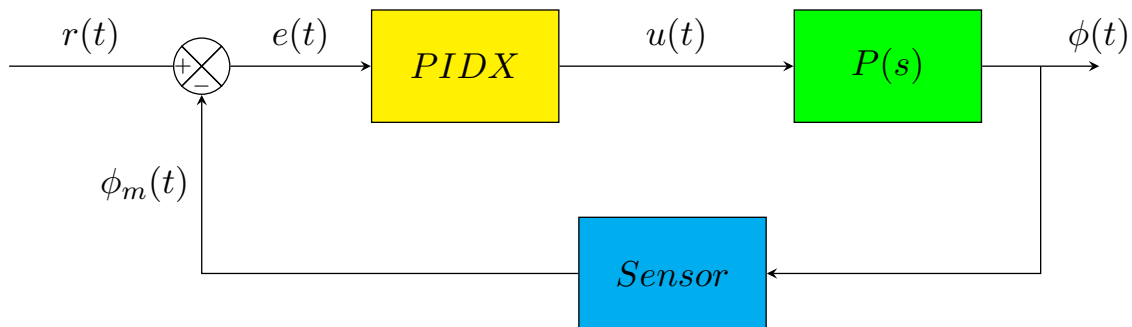


Figure 4.1: Block diagram for Ziegler-Nichols closed-loop method testing

To actually test the method on the Quanser Aero, the steps were followed fairly ordinarily, both for the efficient and inefficient propellers. The marginally stable responses used for the ultimate gains and ultimate periods are shown in Fig. 4.2 and 4.3. This resulted in $K_U = 70.50$ (38.00) and $T_U = 2.142$ (2.625) in the case of efficient (inefficient) propellers,

4.1 Ziegler-Nichols closed-loop method

which were used with Table 3.1 to obtain the PID parameters and filter coefficients. The final parameters are shown in Fig. 4.1. Applying the parameters to the controllers resulted in the responses shown in Fig. 5.1 and 5.9.

Table 4.1: PID parameters and filter coefficients for Ziegler-Nichols tuning

Controller	K_P	K_I	K_D	τ_f
Efficient propellers controller	42.30	39.50	11.32	0.02677
Inefficient propellers controller	22.80	17.37	7.482	0.03282

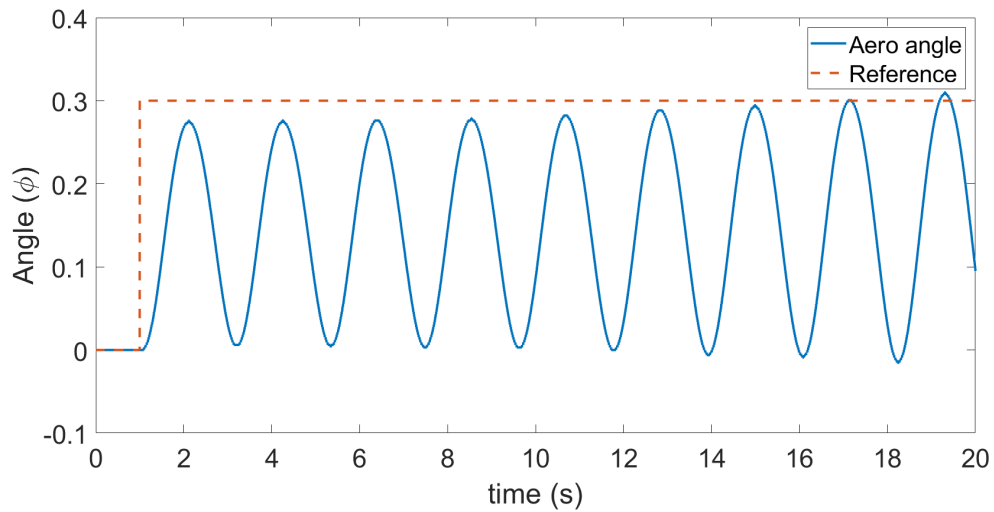


Figure 4.2: Marginally stable Ziegler-Nichols system response, efficient propellers, obtained at $K_P = K_U = 70.50$

4.2 Standard relay-feedback method

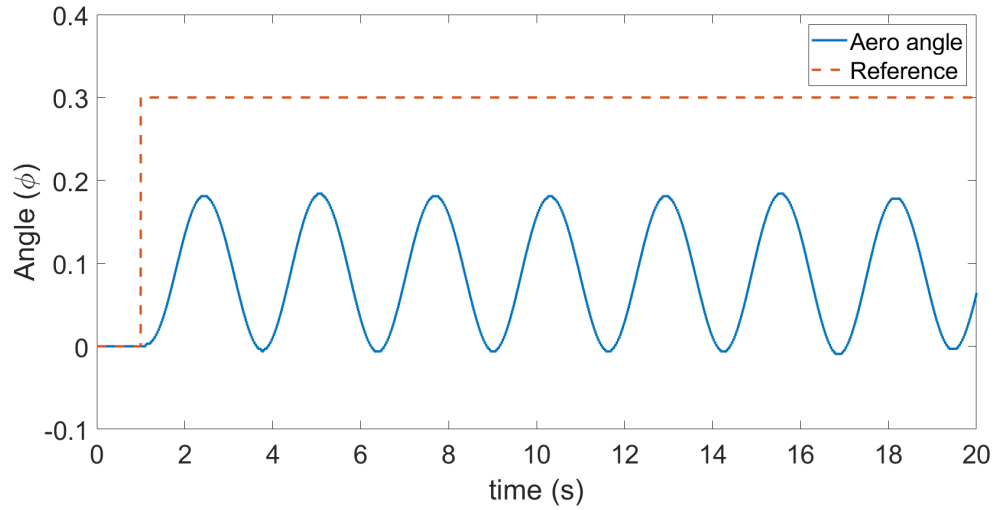


Figure 4.3: Marginally stable Zigler-Nichols system response, inefficient propellers, obtained at $K_P = K_U = 38.00$

4.2 Standard relay-feedback method

The control model used for the tuning process is shown in Fig. 4.4.

4.2 Standard relay-feedback method

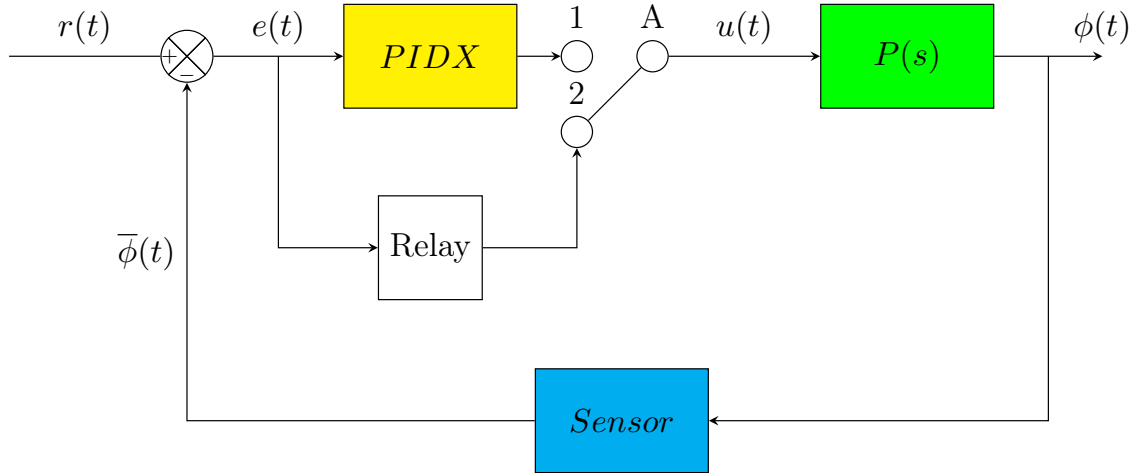


Figure 4.4: Block diagram for standard relay-feedback method testing

The oscillatory responses necessary for the ultimate gain and ultimate period are shown in Fig. 4.5 and 4.6, found at $h = 50$ and $h = 12$ for efficient and inefficient propellers, respectively. The oscillations were considered as in permanently oscillating after 30 seconds, after which $A = 0.6233$ (0.8211) and $T_U = 1.496$ (1.974) were read off the responses in the case of efficient (inefficient) propellers. The final parameters are shown in Fig. 4.2. The resulting PID parameters were applied to the controllers, resulting in Fig. 5.2 and 5.10.

Table 4.2: PID parameters and filter coefficients for Relay feedback tuning

Controller	K_P	K_I	K_D	τ_f
Efficient propellers controller	61.28	81.94	11.46	0.01870
Efficient propellers controller	11.16	11.31	2.755	0.02468

4.2 Standard relay-feedback method

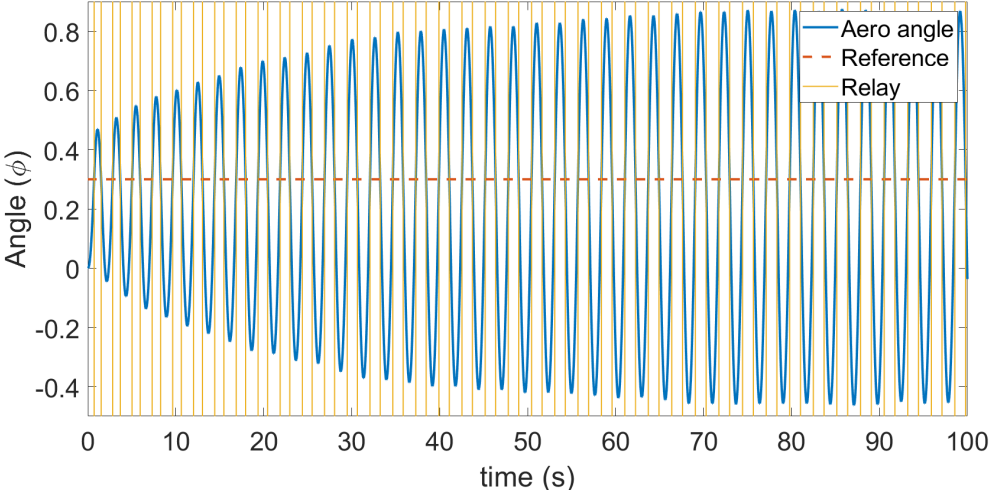


Figure 4.5: Relay feedback test for efficient propellers, obtained at $h = 50$

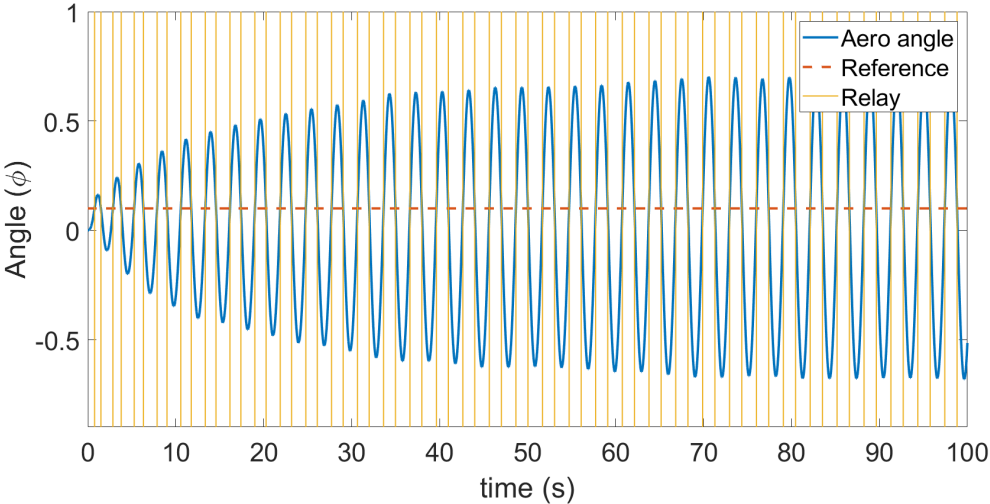


Figure 4.6: Relay feedback test for inefficient propellers, obtained at $h = 12$

4.3 Sequential Ziegler-Nichols closed loop plus Ziegler-Nichols open loop methods on cascaded system

4.3 Sequential Ziegler-Nichols closed loop plus Ziegler-Nichols open loop methods on cascaded system

The model used for the tuning process is in Fig. 4.7.

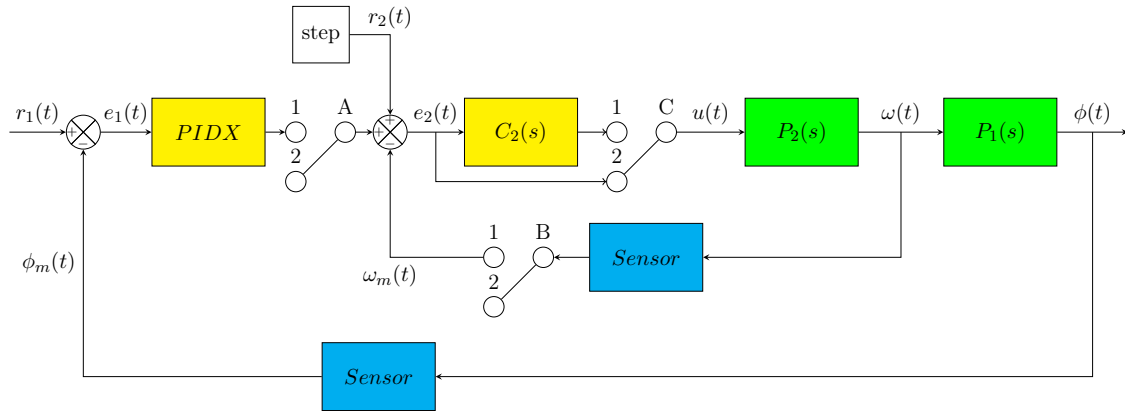


Figure 4.7: Block diagram for Ziegler-Nichols closed loop plus Ziegler-Nichols open loop cascade control

To begin tuning the secondary controller, switch A was set to 2 to disable the primary loop. To actually tune the secondary controller, the Ziegler-Nichols open loop method was selected and used ordinarily. At step input amplitude $U = r_2(t) = 15$, Fig. ?? and 4.9 were obtained, and from it the slopes $R = 7133$ (2101) and the dead-times $L = 0.006$ (0.008) were found in the case of efficient (inefficient) propellers. After the resulting PI parameters were applied to $C_2(s)$, the responses in Fig. 4.10 and 4.11 were obtained from $r_2(t) = 150$.

4.3 Sequential Ziegler-Nichols closed loop plus Ziegler-Nichols open loop methods on cascaded system

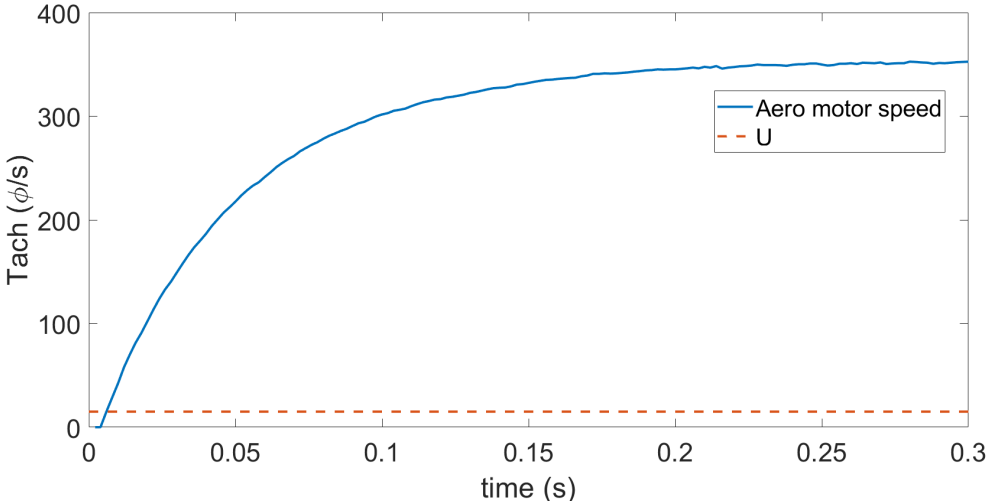


Figure 4.8: Inner loop open loop step response, efficient propeller, found at $U = 15$

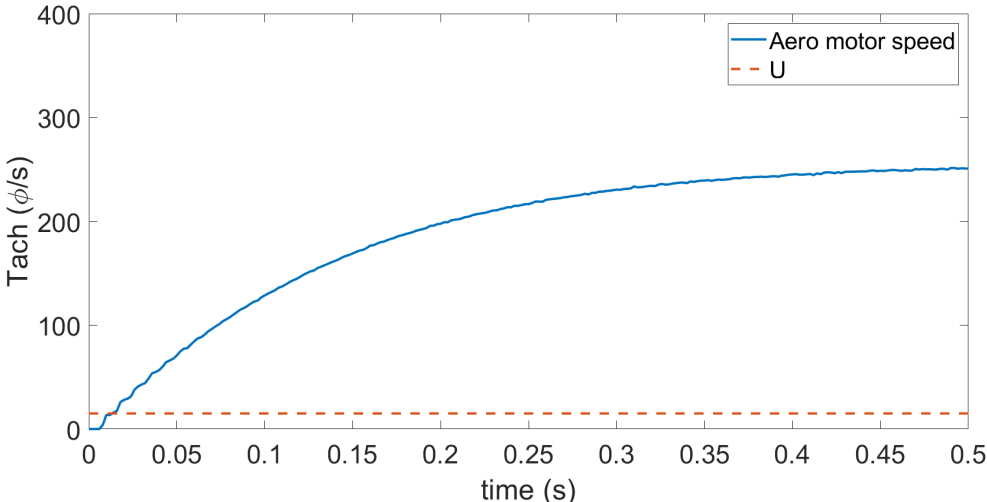


Figure 4.9: Inner loop open loop step response, inefficient propeller, found at $U = 15$

4.3 Sequential Ziegler-Nichols closed loop plus Ziegler-Nichols open loop methods on cascaded system

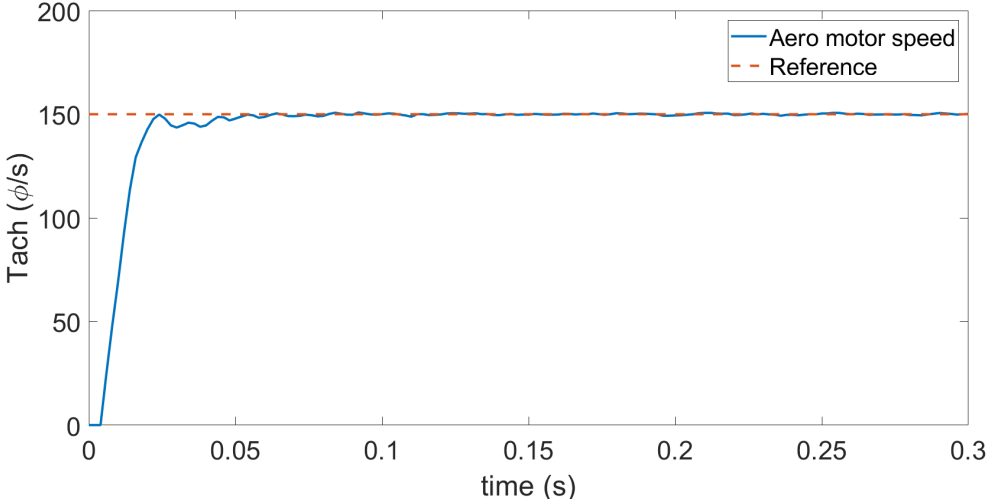


Figure 4.10: Inner loop open loop tuning result, efficient propeller, found at step input $r_2(t) = 150$

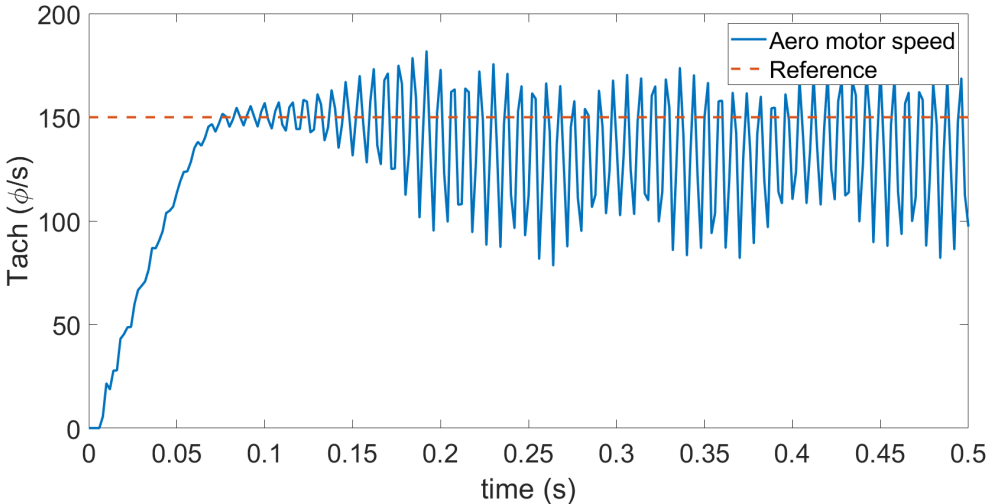


Figure 4.11: Inner loop open loop tuning result, efficient propeller, found at step input $r_2(t) = 150$

Switch A was then set back to 1 to enabled the primary loop. To tune the primary controller, Ziegler-Nichols method was followed normally. Then, $K_U = 25000$ (4100) and

4.3 Sequential Ziegler-Nichols closed loop plus Ziegler-Nichols open loop methods on cascaded system

$T_U = 2.004$ (1.930) were found for the case of efficient (inefficient) from the responses Fig. 4.12 and 4.13. The final parameters are shown in Fig. 4.3 Applying the PID parameters to PIDX resulted in the responses shown in Fig. 5.3 and 5.11.

Table 4.3: PID parameters and filter coefficients for Ziegler-Nichols open-loop plus Ziegler-Nichols closed loop tuning

Controller	K_P	K_I	K_D	τ_f
Efficient propellers secondary controller	0.31544	15.9314	0	-
Efficient propellers primary controller	15000	14972	3757	0.02505
Inefficient propellers secondary controller	0.8033	30.43	0	-
Inefficient propellers primary controller	2460	2549	593.5	0.02412

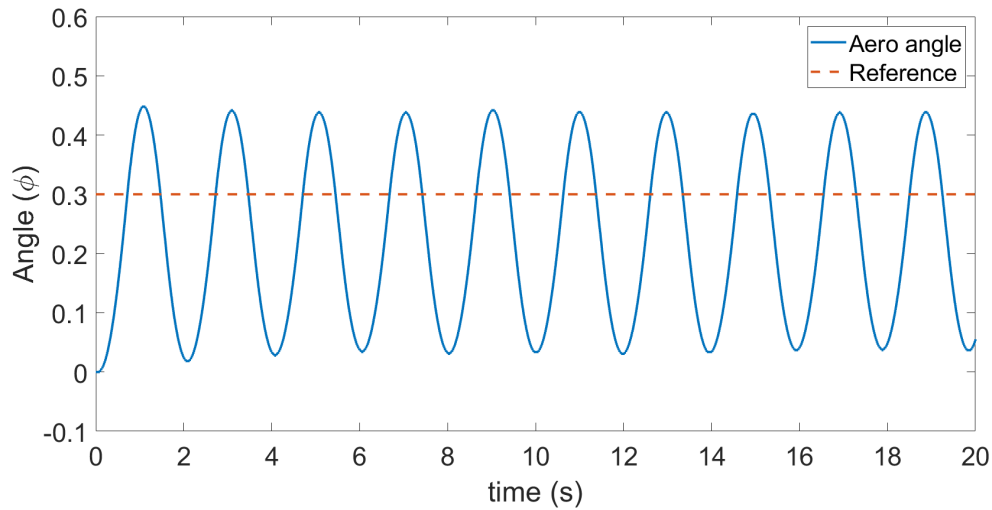


Figure 4.12: Marginally stable outer loop response, efficient propellers, found at $K_U = K_P = 25000$

4.4 Sequential relay-feedback method plus Ziegler-Nichols open loop tuning methods on cascaded system

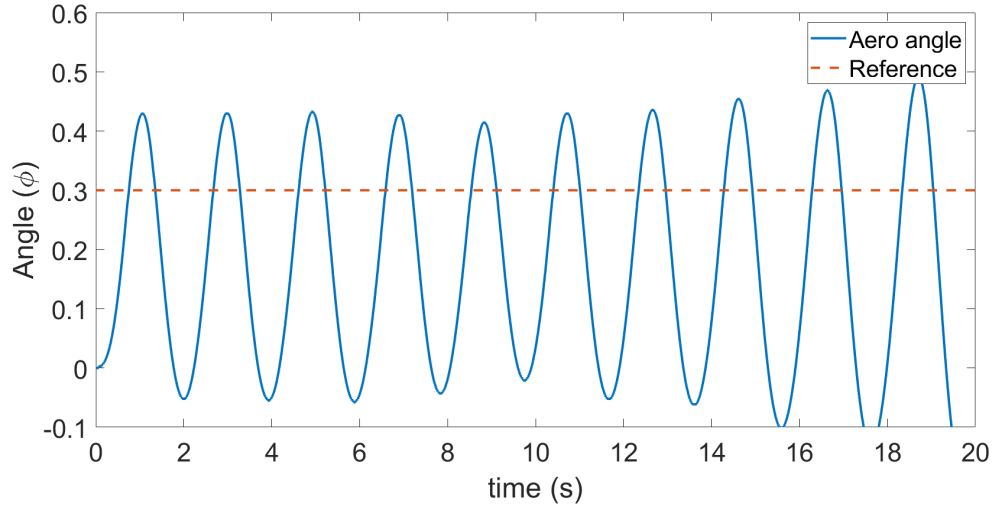


Figure 4.13: Marginally stable outer loop response, inefficient propellers, found at $K_U = K_P = 4100$

4.4 Sequential relay-feedback method plus Ziegler-Nichols open loop tuning methods on cascaded system

The model used for this tuning method is shown in Fig. 4.14.

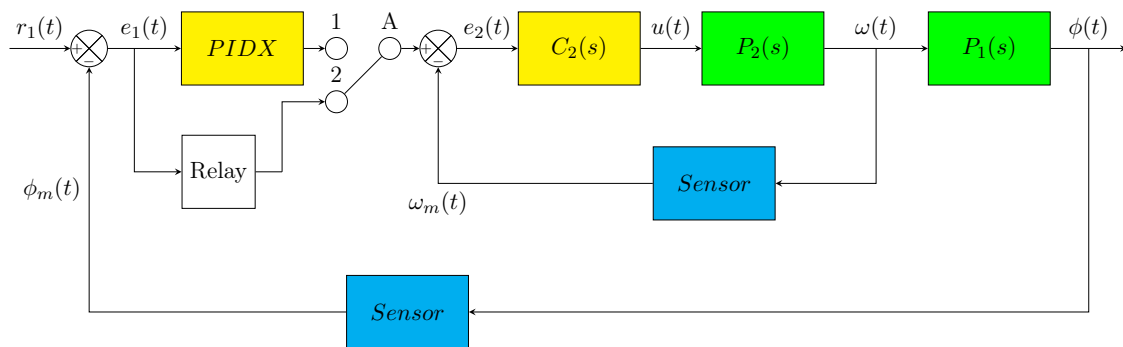


Figure 4.14: Block diagram for relay-feedback plus Ziegler-Nichols open loop cascade control

4.4 Sequential relay-feedback method plus Ziegler-Nichols open loop tuning methods on cascaded system

As this method utilizes the same Ziegler-Nichols open loop technique as the previous part for tuning the inner loop, the secondary controller parameters from table 4.3 were re-used for this section. Therefore, only the outer loop tuning will be covered.

The steps for the relay-feedback methods were then followed ordinarily for the outer loop. The oscillatory response used was found at relay amplitudes of $h = 800$ for efficient, and 250 for inefficient, and are shown in Fig. 4.15 and 4.16. The oscillations were considered as in permanently oscillating after 30 seconds, after which $A = 0.6351$ (0.4227) and $T_U = 1.513$ (1.493) were read off the responses in the case of efficient (inefficient) propellers. The final parameters are shown in Fig. 4.4. Then, applying the PID parameters obtained to the controllers resulted in Fig. 5.4 and 5.12.

Table 4.4: PID parameters and filter coefficients for Ziegler-Nichols open-loop plus relay feedback tuning

Controller	K_P	K_I	K_D	τ_f
Efficient propellers secondary controller	0.31544	15.9314	0	-
Efficient propellers primary controller	962.3	1272	182.0	0.01891
Inefficient propellers secondary controller	0.8033	30.43	0	-
Inefficient propellers primary controller	451.8	605.2	84.34	0.01867

4.4 Sequential relay-feedback method plus Ziegler-Nichols open loop tuning methods on cascaded system

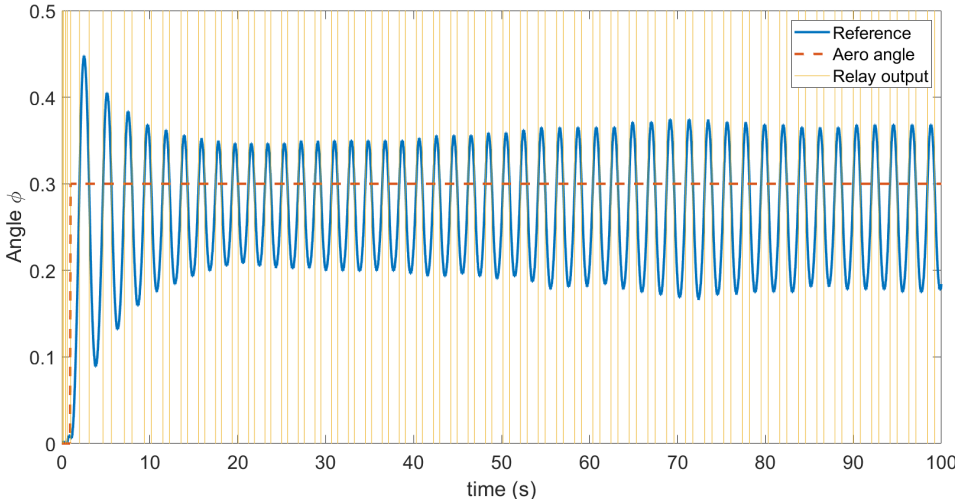


Figure 4.15: Relay outer loop test, efficient propellers, found at $h = 800$

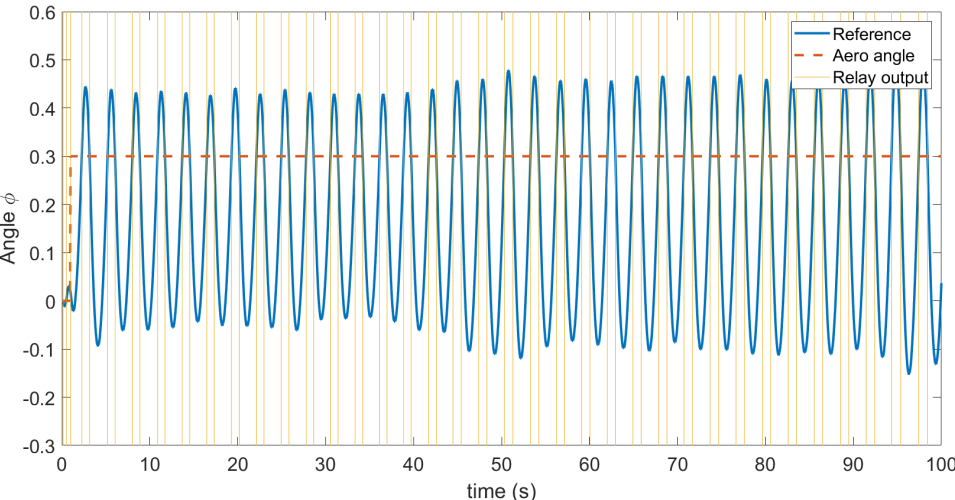


Figure 4.16: Relay outer loop test, inefficient propellers, found at $h = 250$

4.5 Simultaneous tuning using step response

4.5.1 Common grounds

As noted earlier, the testing for this method was done using 4 different approaches. All of them utilized the model shown in Fig. 4.17.

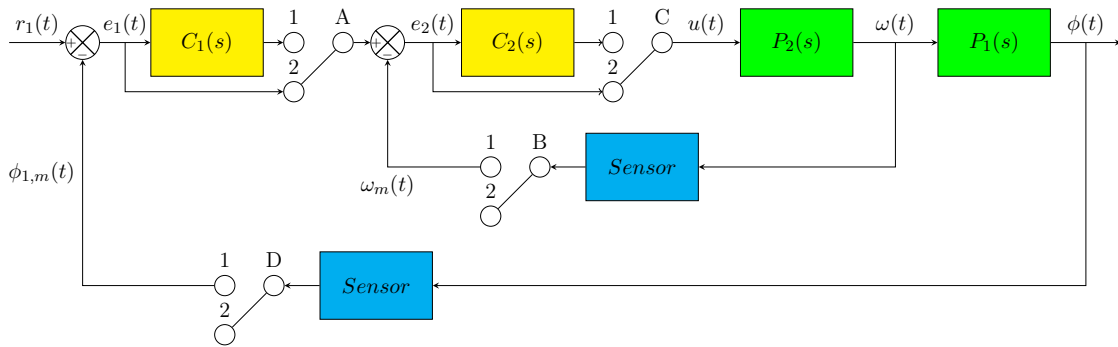


Figure 4.17: Basic block diagram for simultaneous step response method

Since all the step response tuning approaches can be executed from script after a single open loop step test, the same step responses of ϕ and ω were used for all the approaches. In addition, testing used $m = 3$ and $n = 4$ for least-squares model estimation method in all approaches, as that should be sufficient to create a model that replicates most properties of the original process without overfitting.

Notably, the open loop tests used to achieve these results had to be redone several times, especially for the inefficient propellers, since it would oftentimes result in negative parameters or complex answers, which are both unusable. This was largely due to the faults mentioned in the least-squares reduction method.

Since all the selected approaches use the area method for FOPDT estimation, the following execution of the area method applies to all of them:

To begin, all switches were set to 2 to disable the controllers and set the system in open loop. The system was then excited using an input of $U = 15$. From the step response of P_2 , $\omega(t)$, the area method found that the FOPDT parameters were $K = 23.56$, $\tau = 0.05238$, $L = 0.001801$ for the efficient propellers, and $K = 16.95$, $\tau = 0.1304$, $L = 0.010145$ for the

4.5 Simultaneous tuning using step response

inefficient propellers.

4.5.2 Step response Kappa-Tau

Using the model found in section 4.5.1, combined with Kappa-Tau tuning, least-squares process estimation on the step responses of ω and ϕ and least-squares reduction, the FOPDT model parameters of P_T were found as $K = 0.0005323$, $\tau = 0.3620$ and $L = 0.7599$ for the efficient propellers, and $K = 0.0008706$, $\tau = 0.2292$ and $L = 0.6639$ for the inefficient propellers. Then, by using the Kappa-Tau method, the parameters in Table 4.5 were found.

Table 4.5: PID parameters and filter coefficients for 'step response Kappa-Tau' tuning

Controller	K_P	K_I	K_D	τ_f
Efficient propellers secondary controller	0.1310	3.266	0	-
Efficient propellers primary controller	327.4	856.5	26.27	0.008024
Inefficient propellers secondary controller	0.3060	4.697	0	-
Inefficient propellers primary controller	165.2	665.5	7.694	0.004657

4.5.3 Step response IMC

Using the model found in section 4.5.1, combined with Kappa-Tau tuning, least-squares process estimation on the step responses of ω and ϕ and least-squares reduction, the SOPDT model parameters of P_T were found as $K = 0.0005323$, $\tau = 0.5209$, $\xi = 0.07795$ and $L = 0.06097$ for the efficient propellers, and $K = 0.0008706$, $\tau = 0.4842$, $\xi = 0.1005$ and $L = 0.07484$ for the inefficient propellers. Then, by using the Kappa-Tau method, the parameters in Table 4.6 were found.

4.5 Simultaneous tuning using step response

Table 4.6: PID parameters and filter coefficients for 'step response IMC' tuning

Controller	K_P	K_I	K_D	τ_f
Efficient propellers secondary controller	0.2265	4.718	0	0.001773
Efficient propellers primary controller	544.0	11380	3067	0.5638
Inefficient propellers secondary controller	0.3065	2.262	0	0.003652
Inefficient propellers primary controller	487.4	6690	1555	0.3190

4.5.4 Step response simultaneous FOPDT plus FOPDT

Using the model found in section 4.5.1 least-squares process estimation on the step responses of ω and ϕ and least-squares reduction, the FOPDT model parameters of P_1 were found as $K = 0.0005323$, $\tau = 0.3619$ and $L = 0.7410$ for the efficient propellers, and $K = 0.0008706$, $\tau = 0.2433$ and $L = 0.6307$ for the inefficient propellers. Then, by using the simultaneous FOPDT plus FOPDT tuning method, the parameters in Table 4.7 were found.

Table 4.7: PID parameters and filter coefficients for 'step response simultaneous FOPDT plus FOPDT' tuning

Controller	K_P	K_I	K_D	τ_f
Efficient propellers secondary controller	0.5786	10.89	0	-
Efficient propellers primary controller	1029	1684	152.4	0.01481
Inefficient propellers secondary controller	0.5189	3.878	0	-
Inefficient propellers primary controller	552.1	1195	64.88	0.01175

4.5.5 Step response simultaneous FOPDT plus SOPDT

Using the model found in section 4.5.1, least-squares process estimation on the step responses of ω and ϕ and least-squares reduction, the SOPDT model parameters of P_1 were found as $K = 0.0005323$, $\tau = 0.5209$, $\xi = 0.07791$ and $L = 0.05510$ for the efficient pro-

4.5 Simultaneous tuning using step response

pellers, and $K = 0.0008706$, $\tau = 0.4857$, $\xi = 0.1031$ and $L = 0.06582$ for the inefficient propellers. Then, by using the simultaneous FOPDT plus SOPDT tuning method, the parameters in Table 4.8 were found.

Table 4.8: PID parameters and filter coefficients for 'step response IMC' tuning

Controller	K_P	K_I	K_D	τ_f
Efficient propellers secondary controller	0.8322	15.71	0	-
Efficient propellers primary controller	2224	22012	5811	0.2613
Inefficient propellers secondary controller	0.5189	3.878	0	-
Inefficient propellers primary controller	1316	10080	2297	0.1746

Chapter 5

Results

5.1 PID parameters and filter coefficients

Table 5.1: PID parameters and filter coefficients for all tuning methods, efficient propellers

Method and controller	K_P	K_I	K_D	τ_f
Single loop Ziegler Nichols closed loop	42.30	39.50	11.32	0.02677
Single loop relay feedback method	61.28	81.94	11.46	0.01870
Ziegler-Nichols closed loop plus Ziegler-Nichols open loop, secondary controller	0.31544	15.9314	0	-
Ziegler-Nichols closed loop plus Ziegler-Nichols open loop, primary controller	15000	14972	3757	0.02505
Relay feedback plus Ziegler-Nichols open loop, secondary controller	0.31544	15.9314	0	-
Relay feedback plus Ziegler-Nichols open loop, primary controller	962.3	1272	182.0	0.01891
Step response Kappa-Tau, secondary controller	0.1310	3.266	0	-
Step response Kappa-Tau, primary controller	327.4	856.5	26.27	0.008024
Step response IMC, secondary controller	0.2265	4.718	0	0.001773
Step response IMC, primary controller	544.0	11380	3067	0.5638
Step response simultaneous FOPDT plus FOPDT, secondary controller	0.5786	10.89	0	-
Step response simultaneous FOPDT plus FOPDT, primary controller	1029	1684	152.4	0.01481
Step response simultaneous FOPDT plus SOPDT, secondary controller	0.8322	15.71	0	-
Step response simultaneous FOPDT plus SOPDT, primary controller	2224	22012	5811	0.2613

5.1 PID parameters and filter coefficients

Table 5.2: PID parameters and filter coefficients for all tuning methods, inefficient propellers

Method and controller	K_P	K_I	K_D	τ_f
Single loop Ziegler Nichols closed loop	22.80	17.37	7.482	0.03282
Single loop relay feedback method	11.16	11.31	2.755	0.02468
Ziegler-Nichols closed loop plus Ziegler-Nichols open loop, secondary controller	0.8033	30.43	0	-
Ziegler-Nichols closed loop plus Ziegler-Nichols open loop, primary controller	2460	2549	593.5	0.02412
Relay feedback plus Ziegler-Nichols open loop, secondary controller	0.8033	30.43	0	-
Relay feedback plus Ziegler-Nichols open loop, primary controller	451.8	605.2	84.34	0.01867
Step response Kappa-Tau, secondary controller	0.3060	4.697	0	-
Step response Kappa-Tau, primary controller	165.2	665.5	7.694	0.004657
Step response IMC, secondary controller	0.3065	2.262	0	0.003652
Step response IMC, primary controller	487.4	6690	1555	0.3190
Step response simultaneous FOPDT plus FOPDT, secondary controller	0.5189	3.878	0	-
Step response simultaneous FOPDT plus FOPDT, primary controller	552.1	1195	64.88	0.01175
Step response simultaneous FOPDT plus SOPDT, secondary controller	0.5189	3.878	0	-
Step response simultaneous FOPDT plus SOPDT, primary controller	1316	10080	2297	0.1746

5.2 Figures

5.2.1 Efficient propellers

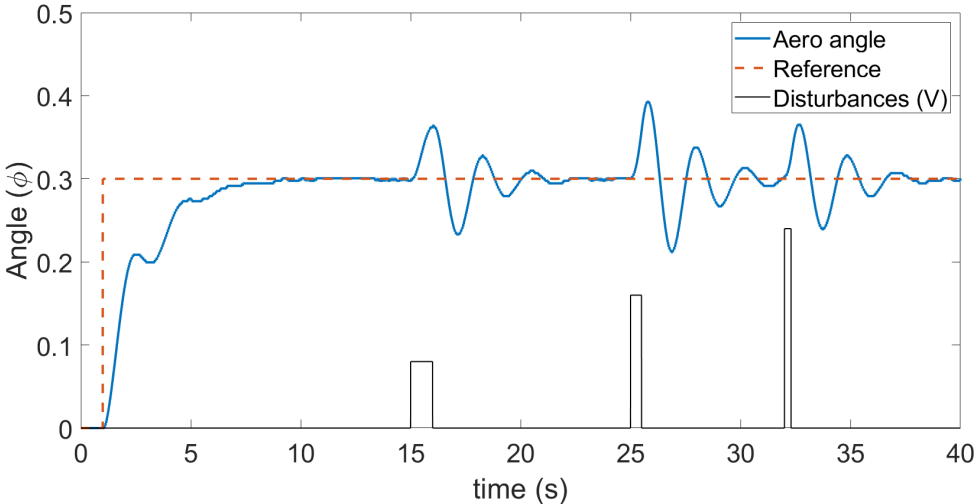


Figure 5.1: Single loop closed loop Ziegler-Nichols method result, efficient propellers

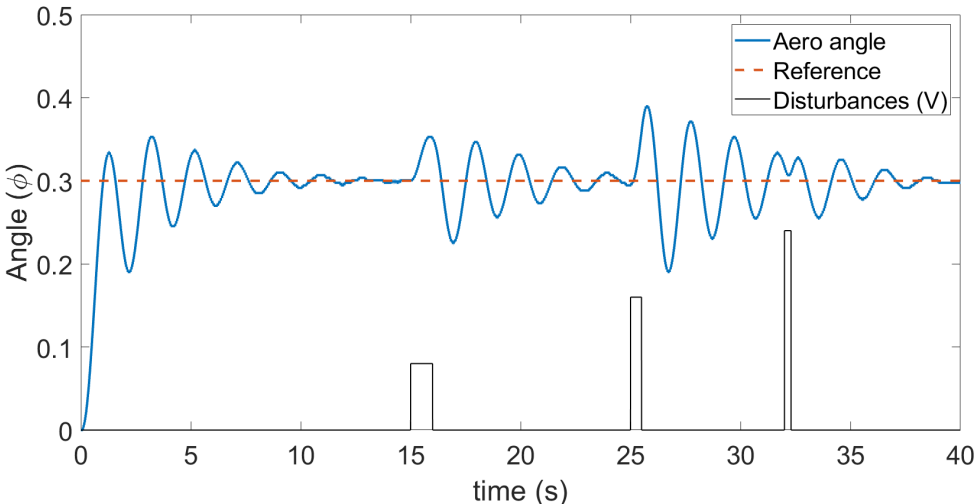


Figure 5.2: Single loop relay feedback method result, efficient propellers

5.2 Figures

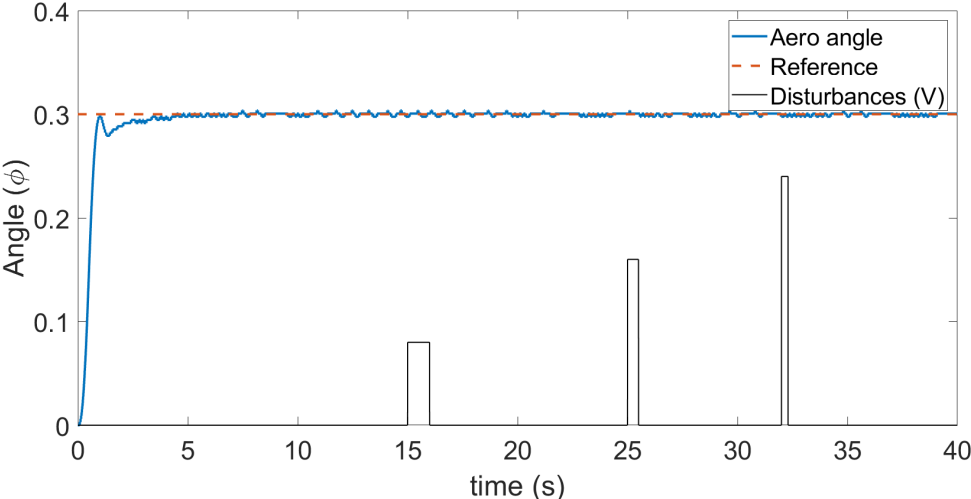


Figure 5.3: Sequential closed loop Ziegler-Nichols plus open loop Ziegler-Nichols result, efficient propellers

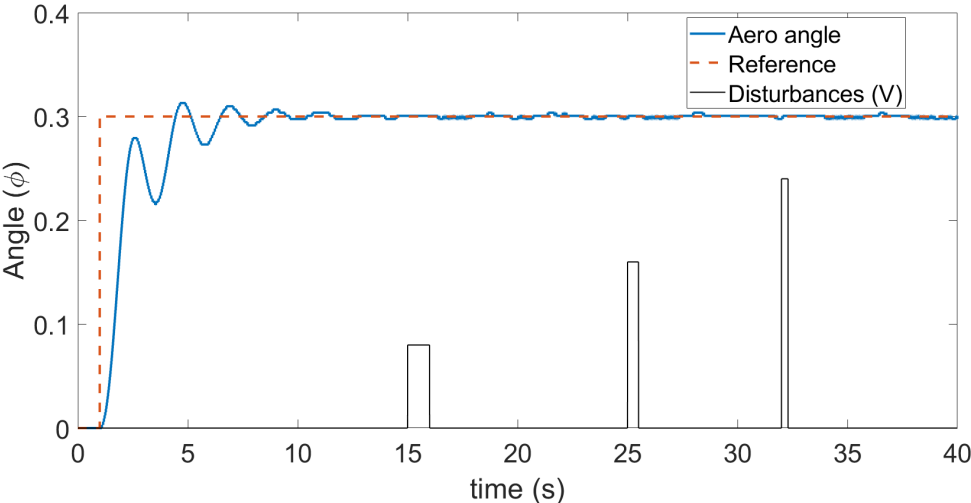


Figure 5.4: Sequential relay feedback plus open loop Ziegler-Nichols result, efficient propellers

5.2 Figures

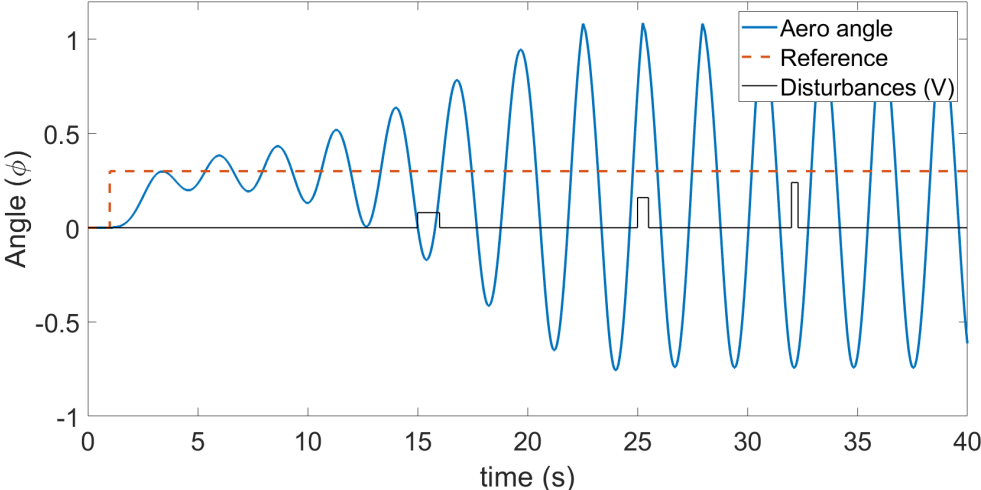


Figure 5.5: Step response Kappa-Tau cascade tuning, efficient propellers

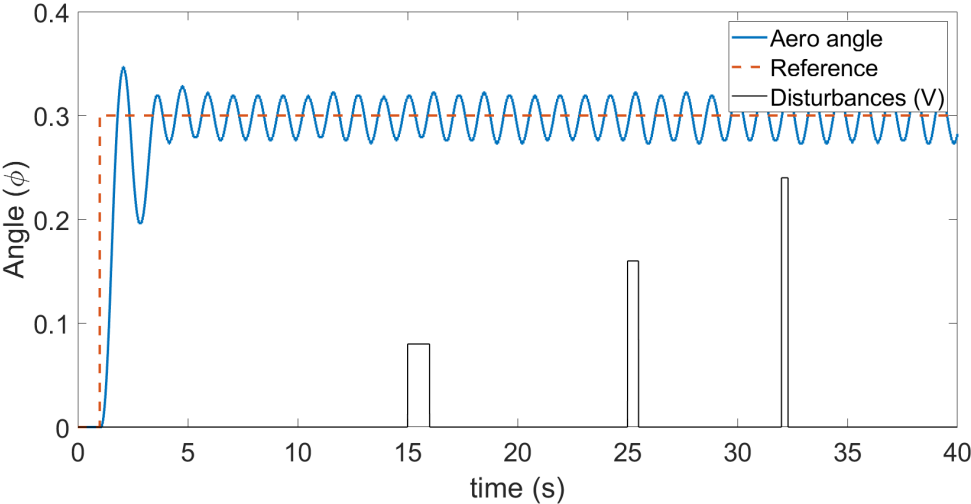


Figure 5.6: Step response IMC cascade tuning, efficient propellers

5.2 Figures

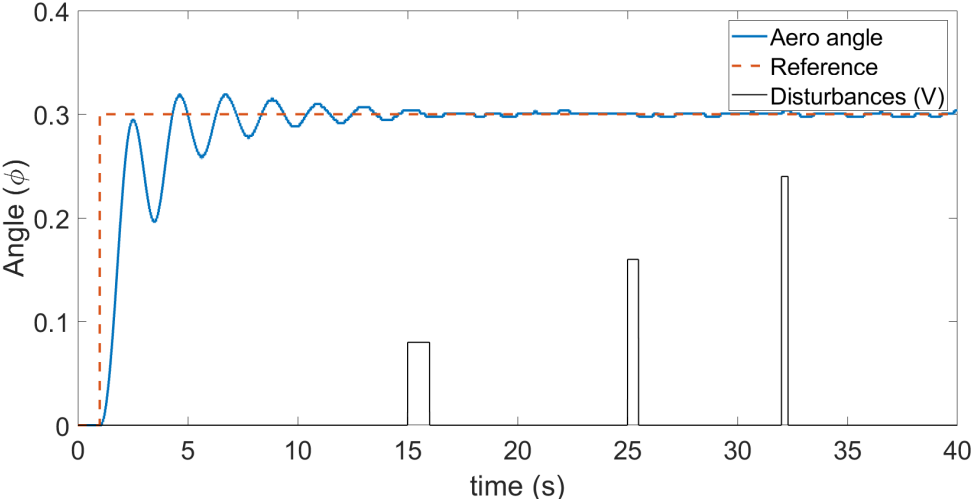


Figure 5.7: Step response simultaneous FOPDT plus FOPDT cascade tuning, efficient propellers

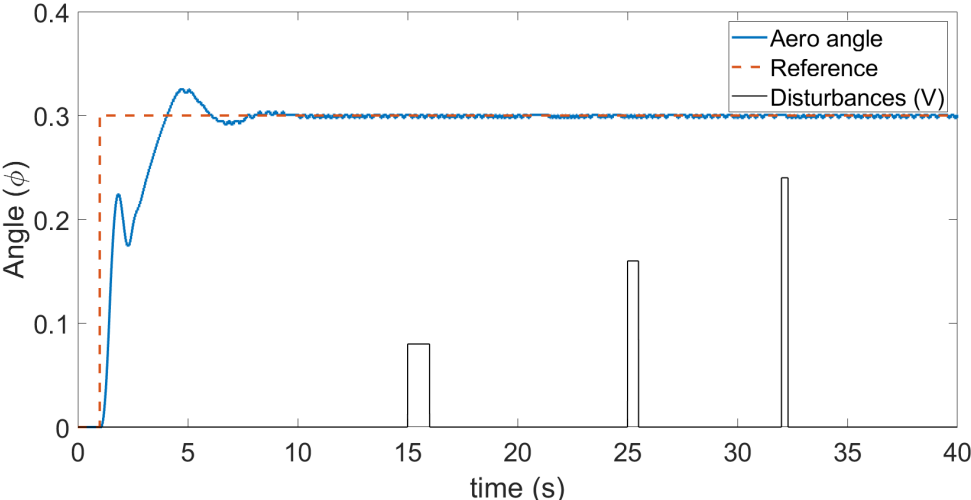


Figure 5.8: Step response simultaneous FOPDT plus SOPDT cascade tuning, efficient propellers

5.2 Figures

5.2.2 Inefficient propellers

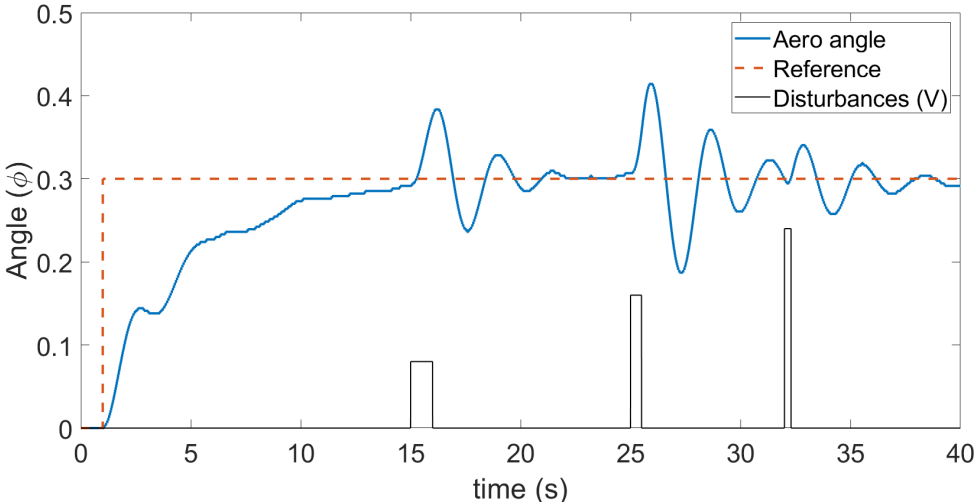


Figure 5.9: Single loop closed-loop Ziegler-Nichols result, inefficient propellers

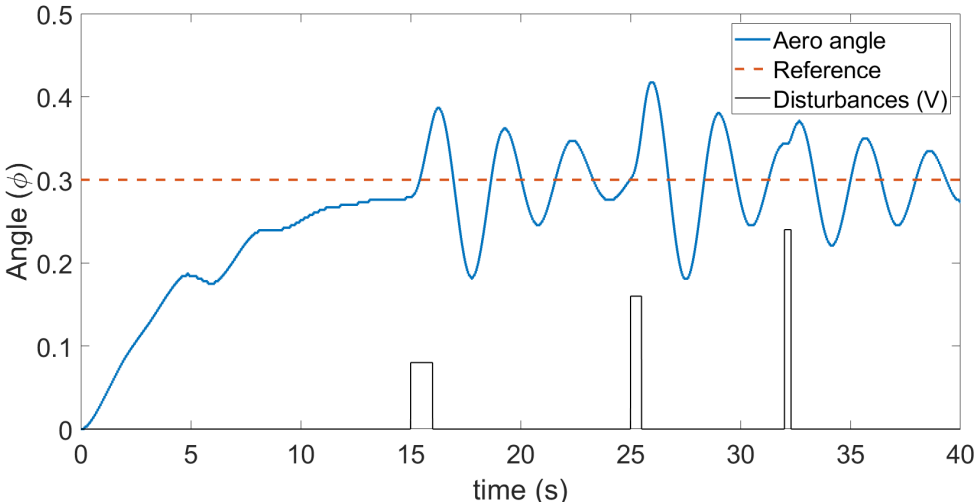


Figure 5.10: Single loop relay feedback result, inefficient propellers

5.2 Figures

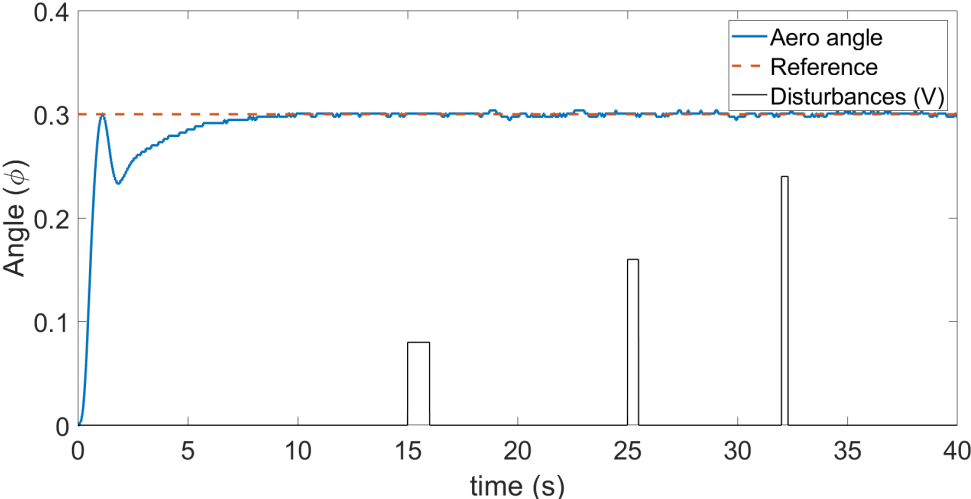


Figure 5.11: Sequential closed loop Ziegler-Nichols plus open loop Ziegler-Nichols result, inefficient propellers

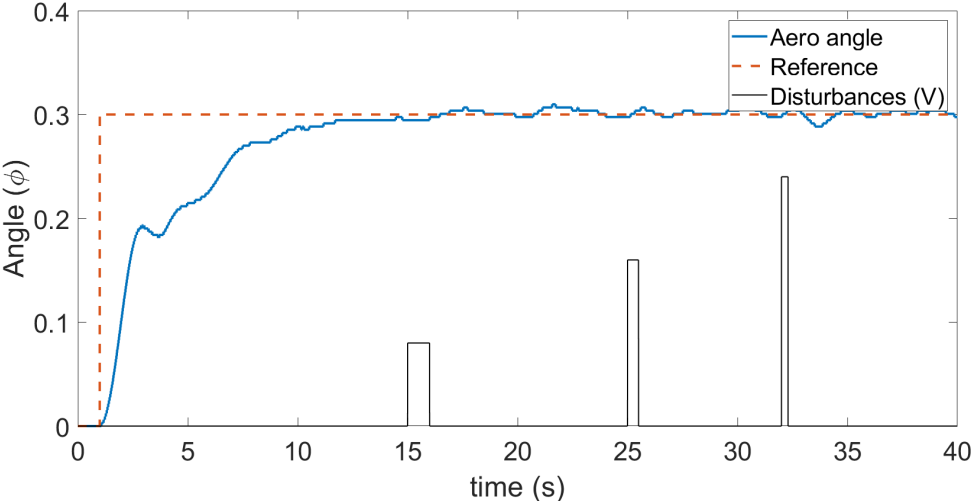


Figure 5.12: Sequential relay feedback plus open loop Ziegler-Nichols result, inefficient propellers

5.2 Figures

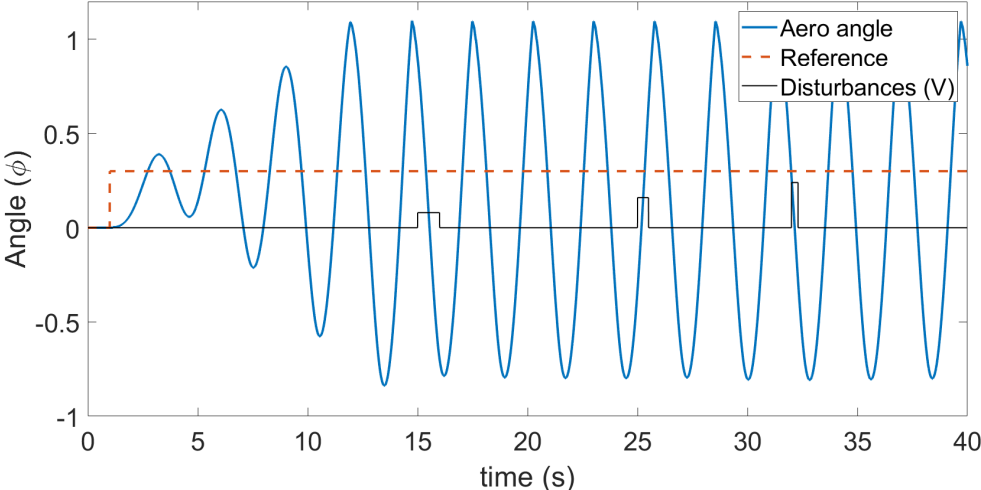


Figure 5.13: Step response Kappa-Tau cascade tuning, inefficient propellers

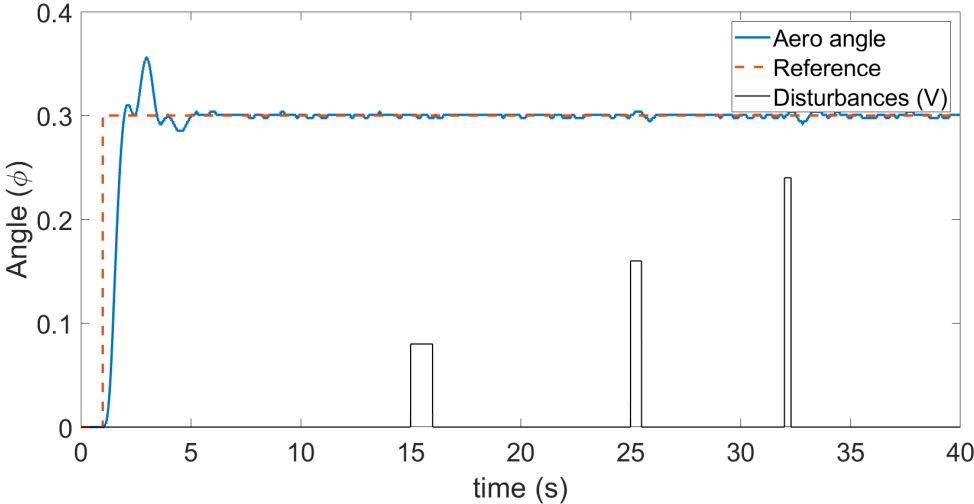


Figure 5.14: Step response IMC cascade tuning, inefficient propellers

5.2 Figures

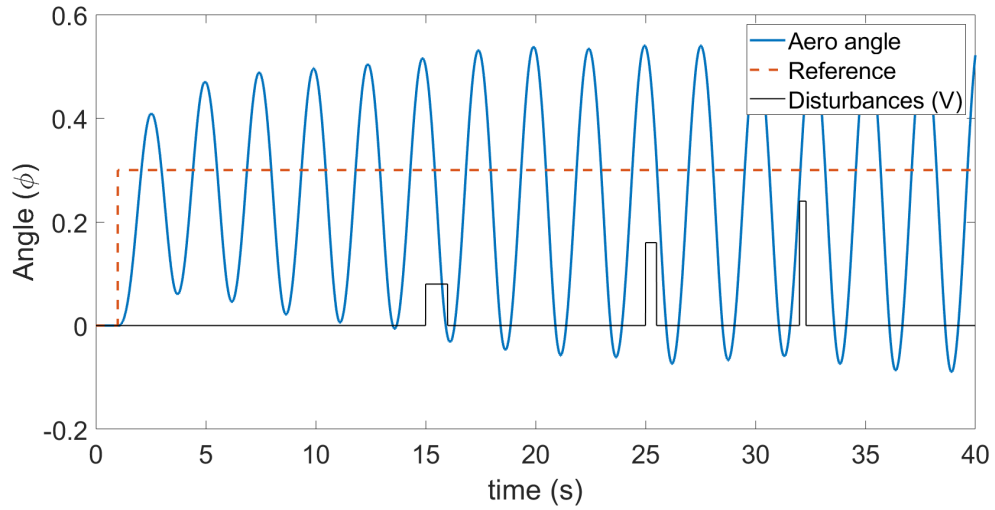


Figure 5.15: Step response simultaneous FOPDT plus FOPDT cascade tuning, inefficient propellers

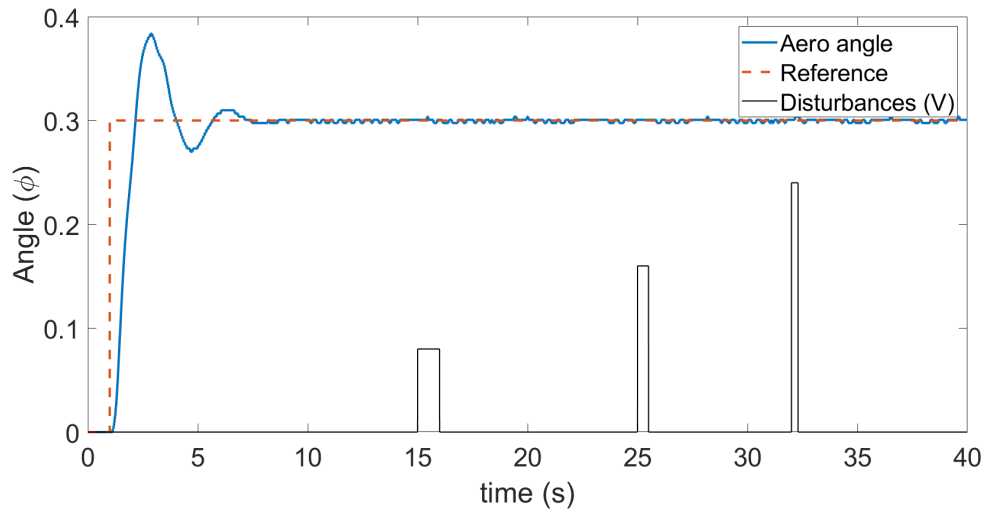


Figure 5.16: Step response simultaneous FOPDT plus SOPDT cascade tuning, inefficient propellers

5.3 Integral performance indices

5.3 Integral performance indices

Table 5.3: Integral performance indices for efficient propellers

Method	IAE	ITAE	ISE	ITSE
Standard Ziegler-Nichols	1.006	14.63	0.09543	0.7130
Standard relay feedback	1.057	17.28	0.08279	0.7669
Cascade Ziegler-Nichols	0.2175	1.125	0.03586	0.01054
Cascade relay feedback	0.4632	2.265	0.06508	0.1116
Step response Kappa-Tau	15.43	394.15	9.816	270.5
Step response IMC	0.7606	11.82	0.05261	0.2690
Step response simultaneous FOPDT plus FOPDT	0.4813	2.489	0.06431	0.1163
Step response simultaneous FOPDT plus SOPDT	0.3960	1.827	0.05190	0.08559

Table 5.4: Integral performance indices for inefficient propellers

Method	IAE	ITAE	ISE	ITSE
Standard Ziegler-Nichols	1.689	20.85	0.1875	1.294
Standard relay feedback	2.568	33.24	0.3365	2.431
Cascade Ziegler-Nichols	0.3818	1.672	0.04748	0.02780
Cascade relay feedback	0.8721	5.570	0.1154	0.2857
Step response Kappa-Tau	20.05	464.5	14.56	354.5
Step response IMC	0.2617	1.507	0.04198	0.05793
Step response simultaneous FOPDT plus FOPDT	7.012	153.3	1.651	38.09
Step response simultaneous FOPDT plus SOPDT	0.3682	1.718	0.05049	0.07937

Chapter 6

Discussion and future work

6.1 Discussion

Due to disturbances and human error varying between individual experiments, small differences in performance can largely be neglected. Even with that in mind, as was desired, the cascade control versions of both the closed loop Ziegler-Nichols and the relay-feedback methods perform much better in the graphs and the integral indices. This improvement is especially prominent for the inefficient propellers, which are affected by disturbances more than the efficient ones. This can also be observed by the time variant integral indices, which put more emphasis on disturbances, demonstrate an especially radical improvement compared from single loop control to cascade control. The only exceptions being the less stable systems, where the low performance in the time variant integral indices can be attributed to that very lack of stability. Though this is best observed by the figures, where while the single loop feedback control systems hardly had time to stabilize between the various disturbances, the cascade control systems were hardly affected. This was consistent even among worse performing methods. Clearly, consistent with what was established earlier, cascade control on the Quanser Aero has significantly superior disturbance rejection properties against disturbances acting in the inner loop, compared to single loop control.

Even besides disturbance rejection though, from reading the figures, it can be observed that there's some improvement in speed and/or stability from the single loop Ziegler-Nichols closed loop and relay feedback experiments to their cascade control equivalents.

6.2 Future work

However, it remains true that sequential cascade control, which were the best performing methods, involves much greater time to tune. Fortunately, the inner loop in these experiments utilized the Ziegler Nichols open loop method, which is less time consuming than the Ziegler Nichols open loop method or the relay feedback method, meaning the time it takes was not quite doubled. In addition, considering that typical tests with the Quanser Aero do not take long, the time it takes to tune is arguably of low relevance compared to the performance of the method.

Regardless, reducing the time it takes to tune the controller is still desirable. For that purpose, simultaneous tuning of controllers can be a useful approach, as it can possibly tune both controllers with just one test and a script. On the other hand, it is much more challenging to implement. Firstly, it takes much more advanced methods to develop the required script. Second, due to the underdamped nature of the Quanser Aero's primary process, the number of methods that are available is drastically limited. As shown by the results of 'Step response Kappa-Tau' and 'Step response simultaneous FOPDT plus FOPDT', while methods that utilize FOPDT models for the outer loop can work, they are particularly unreliable. Though even among the SOPDT based methods, tests had to be redone several times, and in the end largely did not show the same consistent level of performance as the sequential methods. Still, considering only one overall tuning method was attempted for simultaneous tuning, it is hard to conclude whether this was fully the fault of simultaneous tuning. Though at the very least, it is certain that simultaneous tuning takes a lot more effort to set up.

Regardless, there is clearly significant benefit to applying a cascade control configuration to the Quanser Aero. The disturbance rejection properties are very significant, and there is likely more general benefits like speed and/or stability as well. While the time it takes to tune is a problem, it takes a little enough time to tune overall that this is likely not as much of a detriment as the increase in performance is of a benefit. Not to mention it's also possible to cut down this added time by using simultaneous tuning, though the effectiveness of such methods is slightly more uncertain as of now.

6.2 Future work

At this point, this report still leaves lots of work to be done. Particularly, since all testing was done only using the 1DOF helicopter configuration, it may be worthwhile to test the usage of cascade control with other configurations, especially 2DOF. Taken one step further, it may be useful to test cascade control with Quanser's 3DOF helicopter.

6.2 Future work

There would also be value in testing with more tuning methods. While the claim that cascade control is superior in resisting disturbances in the Quanser Aero's inner loop has been quite definitively demonstrated, other factors like speed, stability and ease of implementation would perhaps require more types of tests. In particular, it would be desirable to find another less flawed model reduction method for the cascade step response method. Testing at least one more type of simultaneous tuning method would also be very useful to increase the robustness of any claims regarding simultaneous tuning. In general, a larger variety of tested methods would allow for a much more rigorous analysis of how a cascade control implementation affects the Quanser Aero.

It is also an option to test other types of controllers besides PI and PID. They can potentially change how cascade control affects the performance of the Quanser Aero.

It may of course also be considered to simply improve on the methods already demonstrated in case there were any errors in execution.

Chapter 7

Conclusion

The goal of this bachelor's report is to evaluate how effective applying a cascade control system to the Quanser Aero would be. To determine this, many different ways of tuning a PI or PID controller were established. These were then used in tuning the Quanser Aero several times both using a single loop configuration and a cascade control configuration, after which the performance of the tuned systems was tested. All tests were then repeated with a second set of worse propellers. The results from this were then evaluated and discussed.

In the end, it was clear that the cascade control configuration provides drastically superior disturbance rejection properties against disturbances acting in the inner loop. There's also seemingly some advantage in stability and/or speed, but more testing needs to be done to determine that for certain. While the main disadvantage cascade control, speed of implementation, can be alleviated using simultaneous tuning, this can be much more difficult to implement and much more inconsistent in result. Though in summary, it's clear that a cascade control configuration is overall quite effective when applied to the Quanser Aero.

Bibliography

- [1] Quanser aero. Technical report, quanser.com, <https://www.quanser.com/products/quanser-aero/>, 23. February, 2021.
- [2] Sherif I Abdelmaksoud, Musa Mailah, and Ayman M Abdallah. Practical real-time implementation of a disturbance rejection control scheme for a twin-rotor helicopter system using intelligent active force control. *IEEE Access*, 9:4886–4901, 2020.
- [3] Karl Johan Åström and Tore Hägglund. *PID controllers: theory, design, and tuning*. ISA-The Instrumentation, Systems and Automation Society, 1995.
- [4] I-Lung Chien. Imc-pid controller design-an extension. *IFAC Proceedings Volumes*, 21(7):147–152, 1988.
- [5] Yongho Lee, Sunwon Park, and Moonyong Lee. Pid controller tuning to obtain desired closed-loop responses for cascade control systems. *IFAC Proceedings Volumes*, 31(11):613–618, 1998.
- [6] Manfred Morari and Evangelhos Zafiriou. *Robust process control*. Morari, 1989.
- [7] Rames C Panda, Cheng-Ching Yu, and Hsiao-Ping Huang. Pid tuning rules for sopdt systems: Review and some new results. *ISA transactions*, 43(2):283–295, 2004.
- [8] Daniel E Rivera, Manfred Morari, and Sigurd Skogestad. Internal model control: Pid controller design. *Industrial & engineering chemistry process design and development*, 25(1):252–265, 1986.
- [9] Damiano Rotondo. Ele320_28 - empirical design of pid controllers, month ??? year ???. University of Stavanger.
- [10] Siri Marte Schlanbusch. Adaptive backstepping control of quanser 2dof helicopter: Theory and experiments. Master’s thesis, Universitetet i Agder; University of Agder, 2019.

BIBLIOGRAPHY

- [11] Su Whan Sung, In-Beum Lee, and Byung-Kook Lee. On-line process identification and automatic tuning method for pid controllers. *Chemical Engineering Science*, 53(10):1847–1859, 1998.
- [12] Antonio Visioli. *Practical PID control*. Springer Science & Business Media, 2006.
- [13] Antonio Visioli and Aurelio Piazzzi. An automatic tuning method for cascade control systems. In *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pages 2968–2973. IEEE, 2006.
- [14] John G Ziegler, Nathaniel B Nichols, et al. Optimum settings for automatic controllers. *trans. ASME*, 64(11), 1942.

Vedlegg A

Matlab scripts

Finding integral indices of result test:

```
1 IAE = load("IAE.mat").ans;  
2 ITAE = load("ITAE.mat").ans;  
3 ISE = load("ISE.mat").ans;  
4 ITSE = load("ITSE.mat").ans;  
5  
6 disp("IAE: " + string(IAE(2,end)))  
7 disp("ITAE: " + string(ITAE(2,end)))  
8 disp("ISE: " + string(ISE(2,end)))  
9 disp("ITSE: " + string(ITSE(2,end)))
```

Finding single loop Ziegler-Nichols closed loop parameters and plotting test figure:

```
1 close all  
2 clear  
3 clc  
4  
5 KU = 70.5;  
6 yend = 0.4;  
7 ystart = -0.1;  
8  
9 t0 = 3;  
10 timeset = 20;  
11
```

Matlab scripts

```
12 s = load('y.mat');
13
14 total = s.ans(1, end);
15 x = total-t0;
16 step = s.ans(1, 2) - s.ans(1, 1);
17 tn = timeset/step + 1;
18
19 tK = total/step;
20 xK = (total - x)/step;
21
22 max = 0;
23 peakCount = 0;
24 startFlag = 1;
25 endFlag = 1;
26 peakFlag = 0;
27
28 resetCount = 0;
29
30 for K = 1 : (tK + 1)
31     tempx = s.ans(2, K);
32     if K ≤ xK
33         if tempx > max;
34             max = tempx;
35         end
36     elseif (K > xK) & (tK*0.9 > K)
37         if tempx > max*0.9
38             if peakFlag == 0
39                 peakFlag = 1;
40                 peakCount = peakCount + 1;
41                 resetCount = 0;
42                 if startFlag == 1
43                     firstPeak = K*step;
44                     disp(peakCount)
45                     startFlag = 0;
46                 end
47                 lastPeak = K*step;
48             end
49         else
50             if resetCount < 100
51                 resetCount = resetCount + 1;
52             else
53                 peakFlag = 0;
54             end
55         end
56     else
57         %{
58         if tempx > max*0.9
59             if endFlag == 1
60                 lastPeak = K*step;
```

Matlab scripts

```
61         endFlag = 0;
62     end
63     end
64     %}
65 end
66 end
67
68 w_u = 1/((lastPeak-firstPeak)/(peakCount-1));
69
70 TU = (lastPeak-firstPeak)/(peakCount-1);
71
72 KP = 0.6*KU;
73 KI = 1.2*KU/TU;
74 KD = 3*KU*TU/40;
75
76 TF = KD/KP*0.1;
77
78 %%%Plots -----
79
80 t = s.ans(1, 1:tn);
81 y = s.ans(2, 1:tn);
82
83 s2 = load('r.mat');
84
85 r = s2.ans(2, 1:tn);
86
87 p = plot(t, y, t, r, '--')
88 p(1).LineWidth = 2;
89 p(2).LineWidth = 2;
90 legend("Aero angle", "Reference")
91 ylabel("Angle (\phi)")
92 xlabel("time (s)")
93 ax = gca;
94 ax.FontSize = 22;
95 ylim([ystart, yend]);
96
97 disp("KU: " + string(KU))
98 disp("TU: " + string(TU))
99 disp("KP: " + string(KP))
100 disp("KI: " + string(KI))
101 disp("KD: " + string(KD))
102 disp("TF: " + string(TF))
```

Plotting result of single loop Ziegler-Nichols closed loop method:

```
1 close all
```

Matlab scripts

```
2 clear
3 clc
4
5 yend = 0.5;
6 ystart = 0;
7
8 timeset = 40;
9 tn = timeset/0.002 + 1;
10
11 s = load('y.mat');
12
13 t = s.ans(1, 1:tn);
14 y = s.ans(2, 1:tn);
15
16 s2 = load('r.mat');
17 r = s2.ans(2, 1:tn);
18 s4 = load('disturbance');
19 d = s4.ans(2, 1:tn)*0.02;
20
21 p = plot(t, y, t, r, '--',t, d, 'black')
22 p(1).LineWidth = 2;
23 p(2).LineWidth = 2;
24 p(3).LineWidth = 1;
25 legend("Aero angle", "Reference", "Disturbances (V)")
26 ylabel("Angle (\phi)")
27 xlabel("time (s)")
28 ax = gca;
29 ax.FontSize = 22;
30 ylim([ystart, yend]);
```

Finding single loop relay feedback parameters and plotting test figure:

```
1 close all
2 clear
3 clc
4
5 h = 50;
6 t0 = 30;
7 yend = 0.9;
8 ystart = -0.5;
9 timeset = 100;
10 tn = timeset/0.002 + 1;
11
12 s = load('y.mat');
13
14 total = s.ans(1, end);
```

Matlab scripts

```
15 x = total-t0;
16 step = s.ans(1, 2) - s.ans(1, 1);
17
18 tK = total/step;
19 xK = (total - x)/step;
20
21 max = 0;
22 peakCount = 0;
23 startFlag = 1;
24 endFlag = 1;
25 peakFlag = 0;
26 fallFlag = 0;
27
28 resetCount = 0;
29 ampTopTotal = 0;
30
31 min = 10000;
32 peakCount2 = 0;
33 startFlag2 = 1;
34 endFlag2 = 1;
35 peakFlag2 = 0;
36 fallFlag2 = 0;
37
38 resetCount2 = 0;
39 ampBotTotal = 0;
40
41 for K = 1 : (tK + 1)
42     tempx = s.ans(2, K);
43     if K ≤ xK
44         if tempx > max;
45             max = tempx;
46         end
47         if tempx < min;
48             min = tempx;
49         end
50     elseif (K > xK) & (tK*0.9 > K)
51         %disp(tempx)
52         if tempx > max*0.9
53             if peakFlag == 0
54                 peakFlag = 1;
55                 peakCount = peakCount + 1;
56                 resetCount = 0;
57                 if startFlag == 1
58                     firstPeak = K*step;
59                     %disp(peakCount)
60                     startFlag = 0;
61                 end
62                 lastPeak = K*step;
63             end
64         end
65     end
66 end
```

Matlab scripts

```
64         if fallFlag == 0
65             if tempx > s.ans(2, K+1)
66                 ampTopTotal = ampTopTotal + tempx;
67                 %disp(tempx)
68                 fallFlag = 1;
69             end
70         end
71     else
72         if resetCount < 100
73             resetCount = resetCount + 1;
74         else
75             peakFlag = 0;
76             fallFlag = 0;
77         end
78     end
79     if tempx < (min + 0.1*max)
80         if peakFlag2 == 0
81             peakFlag2 = 1;
82             peakCount2 = peakCount2 + 1;
83             resetCount2 = 0;
84             if startFlag2 == 1
85                 firstPeak2 = K*step;
86                 %disp(peakCount)
87                 startFlag2 = 0;
88             end
89             lastPeak2 = K*step;
90         end
91         if fallFlag2 == 0
92             if tempx < s.ans(2, K+1)
93                 ampBotTotal = ampBotTotal + tempx;
94                 %disp(tempx)
95                 fallFlag2 = 1;
96             end
97         end
98     else
99         if resetCount2 < 100
100             resetCount2 = resetCount2 + 1;
101         else
102             peakFlag2 = 0;
103             fallFlag2 = 0;
104         end
105     end
106 end
107 end
108
109
110 A = ((ampTopTotal/(peakCount-1) - ampBotTotal/(peakCount-1)))/2;
111 TU = (lastPeak-firstPeak)/peakCount-1;
112 KU = 4*h/(A*pi);
```

Matlab scripts

```
113
114 KP = 0.6*KU;
115 KI = 1.2*KU/TU;
116 KD = 3*KU*TU/40;
117
118 TF = KD/KP*0.1;
119
120 y = s.ans(2, 1:tn);
121 t = s.ans(1, 1:tn);
122 s2 = load('r.mat');
123 r = s2.ans(2, 1:tn);
124
125 s3 = load('relay');
126 rl = s3.ans(2, 1:tn);
127
128 p = plot(t, y, t, r, '--', t, rl)
129 p(1).LineWidth = 2;
130 p(2).LineWidth = 2;
131 p(3).LineWidth = 1;
132 legend("Aero angle", "Reference", "Relay")
133 ylabel("Angle (\phi)")
134 xlabel("time (s)")
135 ax = gca;
136 ax.FontSize = 22;
137 ylim([ystart, yend]);
138
139 disp("A: " + string(A))
140 disp("TU: " + string(TU))
141 disp("KU: " + string(KU))
142 disp("")
143 disp("KP: " + string(KP))
144 disp("KI: " + string(KI))
145 disp("KD: " + string(KD))
146 disp("TF: " + string(TF))
```

Plotting result of single loop relay feedback:

```
1 close all
2 clear
3 clc
4
5 yend = 0.5;
6 ystart = 0;
7
8 timeset = 40;
9 tn = timeset/0.002 + 1;
```


Matlab scripts

```
10
11 s = load('y.mat');
12
13 t = s.ans(1, 1:tn);
14 y = s.ans(2, 1:tn);
15
16 s2 = load('r.mat');
17 r = s2.ans(2, 1:tn);
18 s4 = load('disturbance');
19 d = s4.ans(2, 1:tn)*0.02;
20
21 p = plot(t, y, t, r, '--',t, d, 'black')
22 p(1).LineWidth = 2;
23 p(2).LineWidth = 2;
24 p(3).LineWidth = 1;
25 legend("Aero angle", "Reference", "Disturbances (V)")
26 ylabel("Angle (\phi)")
27 xlabel("time (s)")
28 ax = gca;
29 ax.FontSize = 22;
30 ylim([ystart, yend]);
```

Finding inner loop open loop parameters and plotting test figure:

```
1 timeset = 0.3;
2 tn = timeset/0.002 + 1;
3 yend = 400;
4
5 r = load('r2.mat');
6 U = r.ans(2, 5);
7
8 s = load('y2.mat');
9 %time = s.ans.time;
10 ttime = s.ans(1, end);
11 step = s.ans(1, 2) - s.ans(1, 1);
12 total = ttime/step;
13
14 t1 = 1000;
15 startflag = 0;
16
17 for n = 1:total
18     value = s.ans(2, n);
19     t = s.ans(1, n);
20     if (value > 1) & (startflag == 0)
21         L = t;
22         x0 = value;
```

Matlab scripts

```
23         t1 = L + step*5;
24         startflag = 1;
25     end
26     if t == t1
27         x1 = value;
28     end
29 end
30
31 R = (x1 - x0)/(t1 - L);
32
33 KP = 0.9*U/(R*L);
34 KI = KP/(3.3*L);
35 KD = 0;
36
37 s = load("y2.mat");
38 t = s.ans(1, 1:tn);
39 y = s.ans(2, 1:tn);
40
41 Uvector = zeros(1, tn) + 15;
42 p = plot(t, y, t, Uvector, '--')
43 p(1).LineWidth = 2;
44 p(2).LineWidth = 2;
45 %p(2).LineWidth = 2;
46 legend("Aero motor speed", "U")
47 ylabel("Tach (\phi/s)")
48 xlabel("time (s)")
49 ax = gca;
50 ax.FontSize = 22;
51 ylim([0, yend]);
52
53 disp("U: " + string(U))
54 disp("L: " + string(L))
55 disp("R: " + string(R))
56 disp("KP: " + string(KP))
57 disp("KI: " + string(KI))
58 disp("KD: " + string(KD))
```

Plotting result of open loop inner loop tuning:

```
1 close all
2
3 timeset = 0.3;
4 tn = timeset/0.002 + 1;
5 yend = 200;
6
7 s = load("y2.mat");
```

Matlab scripts

```
8 t = s.ans(1, 1:tn);
9 y = s.ans(2, 1:tn);
10
11 s2 = load("r2.mat");
12 r = s2.ans(2, 1:tn);
13
14 p = plot(t, y, t, r, '--')
15 p(1).LineWidth = 2;
16 p(2).LineWidth = 2;
17 legend("Aero motor speed", "Reference")
18 ylabel("Tach (\phi/s)")
19 xlabel("time (s)")
20 ax = gca;
21 ax.FontSize = 22;
22 ylim([0, yend]);
```

Finding sequential Ziegler-Nichols closed loop plus Ziegler-Nichols open loop primary parameters and plotting test figure:

```
1 close all
2 clear
3 clc
4
5 KU = 25000;
6 yend = 0.6;
7 ystart = -0.1;
8
9 t0 = 3;
10 timeset = 20;
11 tn = timeset/0.002 + 1;
12
13 s = load('y1.mat');
14
15 total = s.ans(1, end);
16 x = total-t0;
17 step = s.ans(1, 2) - s.ans(1, 1);
18
19 tK = total/step;
20 xK = (total - x)/step;
21
22 max = 0;
23 peakCount = 0;
24 startFlag = 1;
25 endFlag = 1;
26 peakFlag = 0;
27
```

Matlab scripts

```
28 resetCount = 0;
29
30 for K = 1 : (tK + 1)
31     tempx = s.ans(2, K);
32     if K ≤ xK
33         if tempx > max;
34             max = tempx;
35         end
36     elseif (K > xK) & (tK*0.9 > K)
37         if tempx > max*0.9
38             if peakFlag == 0
39                 peakFlag = 1;
40                 peakCount = peakCount + 1;
41                 resetCount = 0;
42                 if startFlag == 1
43                     firstPeak = K*step;
44                     disp(peakCount)
45                     startFlag = 0;
46                 end
47                 lastPeak = K*step;
48             end
49         else
50             if resetCount < 100
51                 resetCount = resetCount + 1;
52             else
53                 peakFlag = 0;
54             end
55         end
56     else
57         %{
58         if tempx > max*0.9
59             if endFlag == 1
60                 lastPeak = K*step;
61                 endFlag = 0;
62             end
63         end
64         %}
65     end
66 end
67
68 w_u = 1/((lastPeak-firstPeak)/(peakCount-1));
69
70 TU = (lastPeak-firstPeak)/(peakCount-1);
71
72 KP = 0.6*KU;
73 KI = 1.2*KU/TU;
74 KD = 3*KU*TU/40;
75
76 TF = KD/KP*0.1;
```

Matlab scripts

```
77
78 %%%Plots -----
79
80 t = s.ans(1, 1:tn);
81 y = s.ans(2, 1:tn);
82
83 s2 = load('r1.mat');
84 r = s2.ans(2, 1:tn);
85
86 p = plot(t, y, t, r, '--')
87 p(1).LineWidth = 2;
88 p(2).LineWidth = 2;
89 legend("Aero angle", "Reference")
90 ylabel("Angle (\phi)")
91 xlabel("time (s)")
92 ax = gca;
93 ax.FontSize = 22;
94 ylim([ystart, yend]);
95
96 disp("KU: " + string(KU))
97 disp("TU: " + string(TU))
98 disp("KP: " + string(KP))
99 disp("KI: " + string(KI))
100 disp("KD: " + string(KD))
101 disp("TF: " + string(TF))
```

Plotting result of Ziegler-Nichols closed loop plus Ziegler-Nichols open loop tuning

```
1 close all
2 clear
3 clc
4
5 yend = 0.4;
6 ystart = 0;
7
8 timeset = 40;
9 tn = timeset/0.002 + 1;
10
11 s = load('y1.mat');
12
13 t = s.ans(1, 1:tn);
14 y = s.ans(2, 1:tn);
15
16 s2 = load('r1.mat');
17 r = s2.ans(2, 1:tn);
18
```

Matlab scripts

```
19 s4 = load('disturbance');
20 d = s4.ans(2, 1:tn)*0.02;
21
22 p = plot(t, y, t, r, '--', t, d, 'black')
23 p(1).LineWidth = 2;
24 p(2).LineWidth = 2;
25 p(3).LineWidth = 1;
26 legend("Aero angle", "Reference", "Disturbances (V)")
27 ylabel("Angle (\phi)")
28 xlabel("time (s)")
29 ax = gca;
30 ax.FontSize = 22;
31 ylim([ystart, yend]);
```

Finding outer loop parameters of sequential relay feedback plus open loop Ziegler-Nichols tuning and plotting test plot:

```
1 close all
2 clear
3 clc
4
5 h = 800;
6 t0 = 30;
7 yend = 1;
8 ystart = -0.6;
9 timeset = 100;
10 tn = timeset/0.002 + 1;
11
12 s = load('y1.mat');
13
14 total = s.ans(1, end);
15 x = total-t0;
16 step = s.ans(1, 2) - s.ans(1, 1);
17
18 tK = total/step;
19 xK = (total - x)/step;
20
21 max = 0;
22 peakCount = 0;
23 startFlag = 1;
24 endFlag = 1;
25 peakFlag = 0;
26 fallFlag = 0;
27
28 resetCount = 0;
29 ampTopTotal = 0;
```

Matlab scripts

```
30
31 min = 10000;
32 peakCount2 = 0;
33 startFlag2 = 1;
34 endFlag2 = 1;
35 peakFlag2 = 0;
36 fallFlag2 = 0;
37
38 resetCount2 = 0;
39 ampBotTotal = 0;
40
41 for K = 1 : (tK + 1)
42     tempx = s.ans(2, K);
43     if K ≤ xK
44         if tempx > max;
45             max = tempx;
46         end
47         if tempx < min;
48             min = tempx;
49         end
50     elseif (K > xK) & (tK*0.9 > K)
51         if tempx > max*0.9
52             if peakFlag == 0
53                 peakFlag = 1;
54                 peakCount = peakCount + 1;
55                 resetCount = 0;
56                 if startFlag == 1
57                     firstPeak = K*step;
58                     %disp(peakCount)
59                     startFlag = 0;
60                 end
61                 lastPeak = K*step;
62             end
63             if fallFlag == 0
64                 if tempx > s.ans(2, K+1)
65                     ampTopTotal = ampTopTotal + tempx;
66                     %disp(tempx)
67                     fallFlag = 1;
68                 end
69             end
70         else
71             if resetCount < 100
72                 resetCount = resetCount + 1;
73             else
74                 peakFlag = 0;
75                 fallFlag = 0;
76             end
77         end
78     if tempx < (min + 0.1*max)
```

Matlab scripts

```
79         if peakFlag2 == 0
80             peakFlag2 = 1;
81             peakCount2 = peakCount2 + 1;
82             resetCount2 = 0;
83             if startFlag2 == 1
84                 firstPeak2 = K*step;
85                 %disp(peakCount)
86                 startFlag2 = 0;
87             end
88             lastPeak2 = K*step;
89         end
90         if fallFlag2 == 0
91             if tempx < s.ans(2, K+1)
92                 ampBotTotal = ampBotTotal + tempx;
93                 %disp(tempx)
94                 fallFlag2 = 1;
95             end
96         end
97     else
98         if resetCount2 < 100
99             resetCount2 = resetCount2 + 1;
100        else
101            peakFlag2 = 0;
102            fallFlag2 = 0;
103        end
104    end
105 end
106 end
107
108
109 A = ((ampTopTotal/(peakCount-1) - ampBotTotal/(peakCount-1))/2);
110 TU = (lastPeak-firstPeak)/peakCount-1;
111 KU = 4*h/(A*pi);
112
113 KP = 0.6*KU;
114 KI = 1.2*KU/TU;
115 KD = 3*KU*TU/40;
116
117 TF = KD/KP*0.1;
118
119 y = s.ans(2, 1:tn);
120 t = s.ans(1, 1:tn);
121 s2 = load('r1.mat');
122 r = s2.ans(2, 1:tn);
123
124 s3 = load('relay');
125 r1 = s3.ans(2, 1:tn);
126
127 p = plot(t, y, t, r, '--', t, r1)
```


Matlab scripts

```
128 p(1).LineWidth = 2;
129 p(2).LineWidth = 2;
130 p(3).LineWidth = 1;
131 legend("Aero angle", "Reference", "Relay")
132 ylabel("Angle (\phi)")
133 xlabel("time (s)")
134 ax = gca;
135 ax.FontSize = 22;
136 ylim([ystart, yend]);
137
138 disp("A: " + string(A))
139 disp("TU: " + string(TU))
140 disp("KU: " + string(KU))
141 disp(" ")
142 disp("KP: " + string(KP))
143 disp("KI: " + string(KI))
144 disp("KD: " + string(KD))
145 disp("TF: " + string(TF))
```

Plotting result of sequential relay feedback plus open loop Ziegler-Nichols tuning:

```
1 close all
2 clear
3 clc
4
5 yend = 0.4;
6 ystart = 0;
7
8 timeset = 40;
9 tn = timeset/0.002 + 1;
10
11 s = load('yl.mat');
12
13 t = s.ans(1, 1:tn);
14 y = s.ans(2, 1:tn);
15
16 s2 = load('r1.mat');
17 r = s2.ans(2, 1:tn);
18
19 s4 = load('disturbance');
20 d = s4.ans(2, 1:tn)*0.02;
21
22 p = plot(t, y, t, r, '--', t, d, 'black')
23 p(1).LineWidth = 2;
24 p(2).LineWidth = 2;
25 p(3).LineWidth = 1;
```

Matlab scripts

```
26 legend("Aero angle", "Reference", "Disturbances (V)")
27 ylabel("Angle (\phi)")
28 xlabel("time (s)")
29 ax = gca;
30 ax.FontSize = 22;
31 ylim([ystart, yend]);
```

Finding parameters of 'step response Kappa-Tau' tuning:

```
1 %lsqnnoneg -> Tn0 -> norma-> KT
2 %-----
3 close all; clear; clc;
4
5 %Area method
6 %-----
7 %step_amount = 15;
8 initial_y = 0;
9
10 u_f = load('step_input.mat');
11 y_f = load('step_output2.mat');
12
13 step_amount = u_f.ans(2, 1);
14
15 y_t = y_f.ans(2, :)/step_amount;
16 time = y_f.ans(1, end);
17 step = y_f.ans(1, 2) - y_f.ans(1, 1);
18 x = [0:step:time];
19
20
21 y_ss = y_t(end);
22
23 y_diff = y_ss - y_t;
24
25 A1 = interpolate_int(x, y_diff);
26
27 K = y_ss;
28
29 LT = abs(A1)/K;
30 x2 = [0:step:LT];
31
32 y_diff2 = y_t - initial_y;
33 y_diff3 = y_diff2(1:(LT/step + 1));
34 %1, current_time, step
35 A2 = interpolate_int(x2, y_diff3);
36 a = y_diff2(1:(LT/step));
37
```

Matlab scripts

```
38 T = exp(1)*A2/K;
39 L = (A1 - K*T)/K;
40
41 K2 = K;
42 T2 = T;
43 L2 = L;
44
45 G2 = tf([K], [T 1]);
46
47 %LSQ
48 %-----
49
50 num = 3;
51 den = 4;
52 unstable = 0;
53
54 y_f = load('step_output1.mat');
55 u_f = load('step_output2.mat');
56 %y_f = load('step_output1x.mat');
57 %u_f = load('step_output2x.mat');
58 %y_f = load('y_sq.mat');
59 %u_f = load('u_sq.mat');
60 y_t = y_f.ans(2, :);
61 u_t = u_f.ans(2, :);
62
63 K = time/step;
64 x = [0:step:time];
65
66 t_values = [1:1:K];
67
68 t_v = [0:step:(t_values(end) - 1)*step];
69 t_v = rot90(t_v, -1);
70
71 syF = zeros(length(t_values), 1);
72 sM = zeros(length(t_values), den + num + 1);
73
74 sy = zeros(length(t_values), den);
75 su = zeros(length(t_values), num + 1);
76 yM = zeros(length(t_values), den + 1);
77 for n = 1:(length(t_values))
78     t_value = t_values(n);
79
80     sM(n, 1) = -y_t(t_value);
81     yM(n, 1) = y_t(t_value);
82
83     y_t_temp = y_t;
84     for nn = 1:den
85         current_index = n - nn;
86         current_time = t_value + 1 - nn;
```

Matlab scripts

```
87     if current_index > 0
88         y_t_temp = trapez_int(y_t_temp, 1, current_time, step);
89         sy(n-nn, nn) = y_t_temp(end);
90         yM(n-nn, nn+1) = y_t_temp(end);
91         if nn == den
92             if unstable == 1
93                 temp = -y_t_temp(end);
94             else
95                 temp = y_t_temp(end);
96             end
97             syF(n-nn) = temp;
98         else
99             sM(n-nn, nn+1) = -y_t_temp(end);
100        end
101    end
102 end
103
104 u_t_i = zeros(1, num + 1);
105 u_t_temp = u_t;
106 t_value = t_values(n);
107 for nn = 1:(num+1)
108     current_index = n - nn;
109     current_time = t_value + 1 - nn;
110     if current_index > 0
111         u_t_temp = trapez_int(u_t_temp, 1, current_time, step);
112         su(n-nn, nn) = u_t_temp(end);
113
114         sM(n-nn, den + nn) = u_t_temp(end);
115     end
116 end
117 end
118
119 %{
120 plottime = rot90(0:step:((length(yM)-1)*step), -1);
121 for n = 1:(den + 1)
122     figure(n)
123     plot(plottime, yM(:, n))
124 end
125 %}
126
127 xsM = sM(1:(length(sM) - num - 1), :);
128 xsyF = syF(1:(length(syF) - num - 1), :);
129
130 c = lsqnonneg(xsM, xsyF);
131 %c = xsyF\xsM;
132 numerator = zeros(1, num + 1);
133 denominator = zeros(1, den + 1);
134 total_str = '[';
135 total_strx = ' ';
```

Matlab scripts

```
136 syms x
137 polN = 0;
138
139 for n = 1:(num + 1)
140     if c(den + n) ≥ 0
141         extra = '+';
142     else
143         extra = '';
144     end
145     total_strx = total_strx + extra + string(c(den + n)) + 'x^' + ...
146         string(num+1 - n) + ' ';
147     polN = polN + c(den + n)*x^(num+1-n);
148     total_str = total_str + string(c(den + n)) + ' ';
149     disp('n' + string(num + 1 - n) + ': ' + string(c(den + n)))
150     numerator(n) = c(den + n);
151 end
152 total_str2 = '[';
153 total_strx2 = ' ';
154 polD = 0;
155
156 for n = 1:(den)
157     if c(n) ≥ 0
158         extra = '+';
159     else
160         extra = '';
161     end
162     total_strx2 = total_strx2 + extra + string(c(n)) + 'x^' + ...
163         string(den - n) + ' ';
164     polD = polD + c(n)*x^(den-n);
165     total_str2 = total_str2 + string(c(n)) + ' ';
166     disp('d' + string(den + 1 - n) + ': ' + string(c(n)))
167     denominator(n) = c(n);
168 end
169 denominator(end) = 1;
170 total_str = total_str + ']';
171 total_str2 = total_str2 + '1]';
172
173 disp('Numerator: ' + total_str)
174 disp('Denominator: ' + total_str2)
175 disp('G1 = tf(' + total_str + ', ' + total_str2 + ');')
176 disp(total_strx)
177
178 G1 = tf(numerator,denominator);
179
180 %xxx: To Tn0 and C
181 %-----
182
```

Matlab scripts

```
183 %{
184 RDT2 = L2/(T2 + L2);
185 KP2 = 3.8*exp(-8.4*RDT2 + 7.3*(RDT2)^2)*T2/(K2*L2);
186 TI2 = 5.2*exp(-2.5*RDT2 - 1.4*(RDT2)^2)*L2;
187 KI2 = KP2/TI2;
188 TD2 = 0.89*exp(-0.37*RDT2 - 4.1*(RDT2)^2)*L2;
189 KD2 = KP2*TD2;
190 %}
191
192 %
193 RDT2 = L2/(T2 + L2);
194 KP2 = 0.41*exp(-0.23*RDT2 + 0.019*RDT2^2)*T2/(K2*L2);
195 TI2 = 5.7*exp(1.7*RDT2 - 0.69*RDT2^2)*L2;
196 KI2 = KP2/TI2;
197 KD2 = 0;
198 %
199
200 C2 = tf([KD2 KP2 KI2],[1 0]);
201
202 GM = C2*G2*G1/(1 + C2*G2)
203
204 bode(GM)
205 %xxx: To T0
206 %-----
207 l = 10;
208 GMx = GM.Numerator(1);
209 GMx = GMx{1};
210 GMx2 = GM.Denominator(1);
211 GMx2 = GMx2{1};
212 if GMx(end) == 0 & GMx2(end) == 0
213     GMx = GMx(1:(end-1));
214     GMx2 = GMx2(1:(end-1));
215 end
216 GM = tf(GMx,GMx2)
217
218
219 [mag, phase, wout] = bode(GM);
220 bode(GM)
221 magnitude = zeros(1, length(wout));
222 for n = 1:length(wout)
223     magnitude(n) = 20*log10(mag(1, 1, n));
224 end
225 figure(2)
226 semilogx(wout, magnitude)
227
228 cross = 0;
229 wc = 0;
230 cross_closest = cross + 5;
231 for n = 1:length(wout)
```

Matlab scripts

```
232     if abs(cross - magnitude(n)) < abs(cross - cross_closest)
233         cross_closest = magnitude(n);
234         wc = wout(n);
235     end
236 end
237
238 xw = wc;
239 s = j*xw;
240
241 GMx = GM.Numerator(1);
242 GMx = GMx{1};
243 GMx2 = GM.Denominator(1);
244 GMx2 = GMx2{1};
245 if GMx(end) == 0 & GMx2(end) == 0
246     GMx = GMx(1:(end-1));
247     GMx2 = GMx2(1:(end-1));
248 end
249 GM = tf(GMx,GMx2);
250
251 KR = GMx(end)/GMx2(end);
252
253 if wc == 0
254     cross = 20*log10(KR) - 3;
255     cross_closest = cross + 5;
256     for n = 1:length(wout)
257         if abs(cross - magnitude(n)) < abs(cross - cross_closest)
258             cross_closest = magnitude(n);
259             wc = wout(n);
260         end
261     end
262     GM_jw = find_numerical(GM, wc);
263 else
264     GM_jw = find_numerical(GM, wc);
265 end
266 GM_jw_mag = abs(GM_jw);
267 GM_jw_arg = angle(GM_jw);
268
269 K1 = KR;
270 TR = sqrt((KR^2 - GM_jw_mag^2))/(GM_jw_mag*wc);
271 T1 = TR;
272 L1 = -(GM_jw_arg + atan(wc*TR))/wc + L2;
273 %Has To do +L2 Because it wasn't part of the initial GM Calculation
274
275 disp(" ")
276 disp("K1: " + string(K1))
277 disp("L1: " + string(L1))
278 disp("T1: " + string(T1))
279 disp(" ")
280     %{
```

Matlab scripts

```
281     RDT1 = L1/(T1 + L1);
282     KP1 = 0.41*exp(-0.23*RDT1 + 0.019*RDT1^2)*T1/(K1*L1);
283     TI1 = 5.7*exp(1.7*RDT1 - 0.69*RDT1^2)*L1;
284     KI1 = KP1/TI1;
285     KD1 = 0;
286     %}
287     %
288     RDT1 = L1/(T1 + L1);
289     KP1 = 3.8*exp(-8.4*RDT1 + 7.3*(RDT1)^2)*T1/(K1*L1);
290     TI1 = 5.2*exp(-2.5*RDT1 - 1.4*(RDT1)^2)*L1;
291     KI1 = KP1/TI1;
292     TD1 = 0.89*exp(-0.37*RDT1 - 4.1*(RDT1)^2)*L1;
293     KD1 = KP1*TD1;
294     TF1 = TD1*0.1;
295     %
296
297 disp('KP2: ' + string(KP2))
298 disp('KI2: ' + string(KI2))
299 disp('KD2: ' + string(KD2))
300
301 disp('KP1: ' + string(KP1))
302 disp('KI1: ' + string(KI1))
303 disp('KD1: ' + string(KD1))
304 disp('TF1: ' + string(TF1))
```

Finding parameters of 'step response IMC' tuning:

```
1 %lsqnnoneg -> Tn0 -> normal SO T0 -> IMC
2 %-----
3 close all; clear; clc;
4
5 %Area method
6 %-----
7 %step_amount = 15;
8 initial_y = 0;
9
10 u_f = load('step_input.mat');
11 y_f = load('step_output2.mat');
12 %{
13 ut = u_f.ans(1, :);
14 ud = u_f.ans(1, :);
15 yt = y_f.ans(1, :);
16 yd = y_f.ans(1, :);
17
18 u_f.ans = timeseries(ud, 0.002);
19 y_f.ans = timeseries(yd, 0.002);
```


Matlab scripts

```
20 %}
21
22 step_amount = u_f.ans(2, 1);
23
24 y_t = y_f.ans(2, :)/step_amount;
25 time = y_f.ans(1, end);
26 step = y_f.ans(1, 2) - y_f.ans(1, 1);
27 x = [0:step:time];
28
29
30 y_ss = y_t(end);
31
32 y_diff = y_ss - y_t;
33
34 A1 = interpolate_int(x, y_diff);
35
36 K = y_ss;
37
38 LT = abs(A1)/K;
39 x2 = [0:step:LT];
40
41 y_diff2 = y_t - initial_y;
42 y_diff3 = y_diff2(1:(LT/step + 1));
43 %1, current_time, step
44 A2 = interpolate_int(x2, y_diff3);
45 a = y_diff2(1:(LT/step));
46
47 T = exp(1)*A2/K;
48 L = (A1 - K*T)/K;
49
50 K2 = K;
51 T2 = T;
52 L2 = L;
53
54 G2 = tf([K], [T 1]);
55
56 %LSQ
57 %-----
58
59 num = 3;
60 den = 4;
61 unstable = 0;
62
63 y_f = load('step_output1.mat');
64 u_f = load('step_output2.mat');
65 %y_f = load('step_output1x.mat');
66 %u_f = load('step_output2x.mat');
67 %y_f = load('y_sq.mat');
68 %u_f = load('u_sq.mat');
```

Matlab scripts

```
69 y_t = y_f.ans(2, :);
70 u_t = u_f.ans(2, :);
71
72 K = time/step;
73 x = [0:step:time];
74
75 t_values = [1:1:K];
76
77 t_v = [0:step:((t_values(end) - 1)*step)];
78 t_v = rot90(t_v, -1);
79
80 syF = zeros(length(t_values), 1);
81 sM = zeros(length(t_values), den + num + 1);
82
83 sy = zeros(length(t_values), den);
84 su = zeros(length(t_values), num + 1);
85 yM = zeros(length(t_values), den + 1);
86 for n = 1:(length(t_values))
87     t_value = t_values(n);
88
89     sM(n, 1) = -y_t(t_value);
90     yM(n, 1) = y_t(t_value);
91
92     y_t_temp = y_t;
93     for nn = 1:den
94         current_index = n - nn;
95         current_time = t_value + 1 - nn;
96         if current_index > 0
97             y_t_temp = trapez_int(y_t_temp, 1, current_time, step);
98             sy(n-nn, nn) = y_t_temp(end);
99             yM(n-nn, nn+1) = y_t_temp(end);
100            if nn == den
101                if unstable == 1
102                    temp = -y_t_temp(end);
103                else
104                    temp = y_t_temp(end);
105                end
106                syF(n-nn) = temp;
107            else
108                sM(n-nn, nn+1) = -y_t_temp(end);
109            end
110        end
111    end
112
113    u_t_i = zeros(1, num + 1);
114    u_t_temp = u_t;
115    t_value = t_values(n);
116    for nn = 1:(num+1)
117        current_index = n - nn;
```

Matlab scripts

```
118     current_time = t_value + 1 - nn;
119     if current_index > 0
120         u_t_temp = trapez_int(u_t_temp, 1, current_time, step);
121         su(n-nn, nn) = u_t_temp(end);
122
123         sM(n-nn, den + nn) = u_t_temp(end);
124     end
125 end
126 end
127
128 %{
129 plottime = rot90(0:step:(length(yM)-1)*step), -1);
130 for n = 1:(den + 1)
131     figure(n)
132     plot(plottime, yM(:, n))
133 end
134 %}
135
136 xsM = sM(1:(length(sM) - num - 1), :);
137 xsyF = syF(1:(length(syF) - num - 1), :);
138
139 c = lsqnonneg(xsM, xsyF);
140 %c = xsyF\xsM;
141 numerator = zeros(1, num + 1);
142 denominator = zeros(1, den + 1);
143 total_str = '[';
144 total_strx = ' ';
145 syms x
146 polN = 0;
147
148 for n = 1:(num + 1)
149     if c(den + n) ≥ 0
150         extra = '+';
151     else
152         extra = '';
153     end
154     total_strx = total_strx + extra + string(c(den + n)) + 'x^' + ...
155         string(num+1 - n) + ' ';
156     polN = polN + c(den + n)*x^(num+1-n);
157     total_str = total_str + string(c(den + n)) + ' ';
158     disp('n' + string(num + 1 - n) + ': ' + string(c(den + n)))
159     numerator(n) = c(den + n);
160 end
161 total_str2 = '[';
162 total_strx2 = ' ';
163 polD = 0;
164 for n = 1:(den)
165     if c(n) ≥ 0
```

Matlab scripts

```
166     extra = '+';
167     else
168         extra = '';
169     end
170     total_strx2 = total_strx2 + extra + string(c(n)) + 'x^' + ...
        string(den - n) + ' ';
171     polD = polD + c(n)*x^(den-n);
172
173     total_str2 = total_str2 + string(c(n)) + ' ';
174     disp('d' + string(den + 1 - n) + ': ' + string(c(n)))
175     denominator(n) = c(n);
176 end
177 denominator(end) = 1;
178
179 total_str = total_str + ']';
180 total_str2 = total_str2 + '1]';
181
182 disp('Numerator: ' + total_str)
183 disp('Denominator: ' + total_str2)
184 disp('G1 = tf(' + total_str + ', ' + total_str2 + ');')
185 disp(total_strx)
186
187 G1 = tf(numerator,denominator);
188
189 %xxx: To Tn0 and C
190 %-----
191
192 lb2 = max(0.25*L2,0.2*T2);
193
194 TI2 = T2 + 0.5*L2;
195 KP2 = (2*T2+L2)/(2*K2*lb2);
196 KI2 = KP2/TI2;
197 KD2 = 0;
198 TF2 = lb2*L2/(2*(lb2 + L2));
199
200 C2 = tf([KD2 KP2 KI2],[1 0]);
201
202 GM = C2*G2*G1/(1 + C2*G2)
203
204 %xxx: To T0
205 %-----
206
207 l = 10;
208 GMx = GM.Numerator(1);
209 GMx = GMx{1};
210 GMx2 = GM.Denominator(1);
211 GMx2 = GMx2{1};
212 if GMx(end) == 0 & GMx2(end) == 0
213     GMx = GMx(1:(end-1));
```

Matlab scripts

```
214     GMx2 = GMx2(1:(end-1));
215 end
216 GM = tf(GMx,GMx2)
217
218 [mag, phase, wout] = bode(GM);
219 figure(1)
220 bode(GM)
221 magnitude = zeros(1, length(wout));
222 for n = 1:length(wout)
223     magnitude(n) = 20*log10(mag(1, 1, n));
224 end
225 figure(2)
226 semilogx(wout, magnitude)
227 p(1).LineWidth = 2;
228 p(2).LineWidth = 2;
229 legend("P_T")
230 ylabel("Bode (dB)")
231 xlabel("Frequency (rad/s)")
232 ax = gca;
233 ax.FontSize = 22;
234
235 cross = 0;
236 wu = 0;
237 cross_closest = cross + 5;
238 for n = 1:length(wout)
239     if abs(cross - magnitude(n)) < abs(cross - cross_closest)
240         cross_closest = magnitude(n);
241         wu = wout(n);
242     end
243 end
244
245 Km = find_numerical(GM, 0);
246
247 if wu == 0
248     cross = 20*log10(Km) - 3;
249     cross_closest = cross + 5;
250     for n = 1:length(wout)
251         if abs(cross - magnitude(n)) < abs(cross - cross_closest)
252             cross_closest = magnitude(n);
253             wu = wout(n);
254         end
255     end
256 end
257
258 wu = round(wu, 4);
259 wiM = [(wu/l):(wu/l):wu];
260
261 A = zeros(1, 1);
262 B = zeros(1, 2);
```

Matlab scripts

```
263 test = zeros((1), 6);
264 disp('Km: ' + string(Km))
265 for n = 1:(l);
266     wi = wiM(n);
267     Gm_jwi = find_numerical(GM, wi);
268     Gm_jwi_mag = abs(Gm_jwi);
269
270     Gm_jwi_arg = angle(Gm_jwi);
271
272     B(n, 1) = Gm_jwi_mag^2 * wi^4;
273     B(n, 2) = Gm_jwi_mag^2 * wi^2;
274     A(n) = Km^2 - Gm_jwi_mag^2;
275
276     test(n, 1) = wi;
277     test(n, 2) = Km;
278     test(n, 3) = Gm_jwi_mag;
279     test(n, 4) = A(n);
280     test(n, 5) = B(n, 1);
281     test(n, 6) = B(n, 2);
282 end
283
284 X = B\A
285 tau_m = nthroot(X(1), 4);
286 gamma_m = sqrt(X(2)/(4*tau_m^2) + 0.5);
287 phi_m = (pi + atan2(-2*tau_m*gamma_m*wu,1 - tau_m^2 * wu^2))/wu;
288
289 disp('tau1 = ' + string(tau_m) + ';'')
290 disp('gamma1 = ' + string(gamma_m) + ';'')
291
292 disp('K1 = ' + string(Km) + ';'')
293 disp('L1 = ' + string(phi_m) + ';'')
294 %disp('phi_m: ' + string(phi_m))
295 disp('Denominator: [' + string(tau_m^2) + ' ' + string(2*tau_m*gamma_m) ...
      + ' 1]')
296
297 tau1 = tau_m;
298 gamma1 = gamma_m;
299 K1 = Km;
300 L1 = phi_m + L2;
301 %Has To do +L2 Because it wasn't part of the initial GM Calculation
302
303 disp(" ")
304 disp("K1: " + string(K1))
305 disp("T1: " + string(tau1))
306 disp("Xi1: " + string(gamma1))
307 disp("L1: " + string(L1))
308 disp("Denom: [" + string(tau1^2) + " " + string(2*tau1*gamma1) + " 1]")
309 disp(" ")
310
```

Matlab scripts

```
311 lb = max(0.25*L1,0.2*tau1);
312 TI1 = 2*gamma1*tau1 - (2*lb^2 - L1^2)/(2*(2*lb + L1));
313 TD1 = TI1 - 2*gamma1*tau1 + (tau1^2 - L1^3 / (6*(2*lb + L1)))/TI1;
314 KP1 = TI1/(K1*(lb + L1));
315 KI1 = KP1/TI1;
316 KD1 = KP1*TD1;
317 TF1 = TD1*0.1;
318
319
320 disp('KP2: ' + string(KP2))
321 disp('KI2: ' + string(KI2))
322 disp('KD2: ' + string(KD2))
323 disp('TF2: ' + string(TF2))
324
325 disp('KP1: ' + string(KP1))
326 disp('KI1: ' + string(KI1))
327 disp('KD1: ' + string(KD1))
328 disp('TF1: ' + string(TF1))
```

Finding parameters of 'Step response simultaneous FOPDT plus FOPDT' tuning:

```
1 %lsqnnoneg -> Tn -> normal SO T -> normal (1, 2)
2 %-----
3 close all; clear; clc;
4
5 %Area method
6 %-----
7 %step_amount = 15;
8 initial_y = 0;
9
10 u_f = load('step_input.mat');
11 y_f = load('step_output2.mat');
12 %{
13 ut = u_f.ans(1, :);
14 ud = u_f.ans(1, :);
15 yt = y_f.ans(1, :);
16 yd = y_f.ans(1, :);
17
18 u_f.ans = timeseries(ud, 0.002);
19 y_f.ans = timeseries(yd, 0.002);
20 %}
21
22 step_amount = u_f.ans(2, 1);
23
24 y_t = y_f.ans(2, :)/step_amount;
25 time = y_f.ans(1, end);
```

Matlab scripts

```
26 step = y_f.ans(1, 2) - y_f.ans(1, 1);
27 x = [0:step:time];
28
29
30 y_ss = y_t(end);
31
32 y_diff = y_ss - y_t;
33
34 A1 = interpolate_int(x, y_diff);
35
36 K = y_ss;
37
38 LT = abs(A1)/K;
39 x2 = [0:step:LT];
40
41 y_diff2 = y_t - initial_y;
42 y_diff3 = y_diff2(1:(LT/step + 1));
43 %1, current_time, step
44 A2 = interpolate_int(x2, y_diff3);
45 a = y_diff2(1:(LT/step));
46
47 T = exp(1)*A2/K;
48 L = (A1 - K*T)/K;
49
50 K2 = K;
51 T2 = T;
52 L2 = L;
53
54 %LSQ
55 %-----
56
57 num = 3;
58 den = 4;
59 unstable = 0;
60
61 y_f = load('step_output1.mat');
62 u_f = load('step_output2.mat');
63 %y_f = load('step_output1x.mat');
64 %u_f = load('step_output2x.mat');
65 %y_f = load('y_sq.mat');
66 %u_f = load('u_sq.mat');
67 y_t = y_f.ans(2, :);
68 u_t = u_f.ans(2, :);
69
70 K = time/step;
71 x = [0:step:time];
72
73 t_values = [1:1:K];
74
```


Matlab scripts

```
75 t_v = [0:step:((t_values(end) - 1)*step)];
76 t_v = rot90(t_v, -1);
77
78 syF = zeros(length(t_values), 1);
79 sM = zeros(length(t_values), den + num + 1);
80
81 sy = zeros(length(t_values), den);
82 su = zeros(length(t_values), num + 1);
83 yM = zeros(length(t_values), den + 1);
84 for n = 1:(length(t_values))
85     t_value = t_values(n);
86
87     sM(n, 1) = -y_t(t_value);
88     yM(n, 1) = y_t(t_value);
89
90     y_t_temp = y_t;
91     for nn = 1:den
92         current_index = n - nn;
93         current_time = t_value + 1 - nn;
94         if current_index > 0
95             y_t_temp = trapez_int(y_t_temp, 1, current_time, step);
96             sy(n-nn, nn) = y_t_temp(end);
97             yM(n-nn, nn+1) = y_t_temp(end);
98             if nn == den
99                 if unstable == 1
100                     temp = -y_t_temp(end);
101                 else
102                     temp = y_t_temp(end);
103                 end
104                 syF(n-nn) = temp;
105             else
106                 sM(n-nn, nn+1) = -y_t_temp(end);
107             end
108         end
109     end
110
111     u_t_i = zeros(1, num + 1);
112     u_t_temp = u_t;
113     t_value = t_values(n);
114     for nn = 1:(num+1)
115         current_index = n - nn;
116         current_time = t_value + 1 - nn;
117         if current_index > 0
118             u_t_temp = trapez_int(u_t_temp, 1, current_time, step);
119             su(n-nn, nn) = u_t_temp(end);
120
121             sM(n-nn, den + nn) = u_t_temp(end);
122         end
123     end
```

Matlab scripts

```
124 end
125
126 plottime = rot90(0:step:((length(yM)-1)*step), -1);
127 for n = 1:(den + 1)
128     figure(n)
129     plot(plottime, yM(:, n))
130 end
131
132 xsM = sM(1:(length(sM) - num - 1), :);
133 xsyF = syF(1:(length(syF) - num - 1), :);
134
135 c = lsqnonneg(xsM, xsyF);
136 %c = xsyF\xsM;
137 numerator = zeros(1, num + 1);
138 denominator = zeros(1, den + 1);
139 total_str = '[';
140 total_strx = ' ';
141 syms x
142 polN = 0;
143
144 for n = 1:(num + 1)
145     if c(den + n) ≥ 0
146         extra = '+';
147     else
148         extra = '';
149     end
150     total_strx = total_strx + extra + string(c(den + n)) + 'x^' + ...
        string(num+1 - n) + ' ';
151     polN = polN + c(den + n)*x^(num+1-n);
152     total_str = total_str + string(c(den + n)) + ' ';
153     disp('n' + string(num + 1 - n) + ': ' + string(c(den + n)))
154     numerator(n) = c(den + n);
155 end
156 total_str2 = '[';
157 total_strx2 = ' ';
158 polD = 0;
159
160 for n = 1:(den)
161     if c(n) ≥ 0
162         extra = '+';
163     else
164         extra = '';
165     end
166     total_strx2 = total_strx2 + extra + string(c(n)) + 'x^' + ...
        string(den - n) + ' ';
167     polD = polD + c(n)*x^(den-n);
168
169     total_str2 = total_str2 + string(c(n)) + ' ';
170     disp('d' + string(den + 1 - n) + ': ' + string(c(n)))
```

Matlab scripts

```
171     denominator(n) = c(n);
172 end
173 denominator(end) = 1;
174
175 total_str = total_str + '];
176 total_str2 = total_str2 + '1]';
177
178 disp('Numerator: ' + total_str)
179 disp('Denominator: ' + total_str2)
180 disp('G1 = tf(' + total_str + ', ' + total_str2 + ');')
181 disp(total_strx)
182
183 G1 = tf(numerator,denominator);
184
185 %xxx: To Tn0 and C
186 %-----
187 lambda2 = 0.5*L2;
188
189 TI2 = T2 + L2^2/(2*(lambda2 + L2));
190 TD2 = (L2^2/(6*(lambda2 + L2)))*(3 - L2/(T2 + L2^2/(2*(lambda2 + L2))));
191 %KP2 = (T2 + (L2^2)/(2*lambda2 + 2*L2))/(K2*(lambda2 + L2));
192 KP2 = TI2/(K2*(lambda2 + L2));
193 KI2 = KP2/TI2;
194 %KD2 = KP2*TD2;
195 KD2 = 0;
196
197 C2 = tf([KD2 KP2 KI2],[1 0]);
198
199 GM = G1;
200
201 %xxx: To T0
202 %-----
203
204 [mag, phase, wout] = bode(GM);
205 bode(GM)
206 magnitude = zeros(1, length(wout));
207 for n = 1:length(wout)
208     magnitude(n) = 20*log10(mag(1, 1, n));
209 end
210 figure(2)
211 semilogx(wout, magnitude)
212
213 cross = 0;
214 wc = 0;
215 cross_closest = cross + 5;
216 for n = 1:length(wout)
217     if abs(cross - magnitude(n)) < abs(cross - cross_closest)
218         cross_closest = magnitude(n);
219         wc = wout(n);
```

Matlab scripts

```
220     end
221 end
222
223 xw = wc;
224 s = j*xw;
225
226 GMx = GM.Numerator(1);
227 GMx = GMx{1};
228 GMx2 = GM.Denominator(1);
229 GMx2 = GMx2{1};
230 if GMx(end) == 0 & GMx2(end) == 0
231     GMx = GMx(1:(end-1));
232     GMx2 = GMx2(1:(end-1));
233 end
234 GM = tf(GMx,GMx2)
235
236 KR = GMx(end)/GMx2(end);
237
238 if wc == 0
239     cross = 20*log10(KR) - 3;
240     cross_closest = cross + 5;
241     for n = 1:length(wout)
242         if abs(cross - magnitude(n)) < abs(cross - cross_closest)
243             cross_closest = magnitude(n);
244             wc = wout(n);
245         end
246     end
247     GM_jw = find_numerical(GM, wc);
248 else
249     GM_jw = find_numerical(GM, wc);
250 end
251 GM_jw_mag = abs(GM_jw);
252 GM_jw_arg = angle(GM_jw);
253
254 K1 = KR;
255 TR = sqrt((KR^2 - GM_jw_mag^2)/(GM_jw_mag*wc));
256 T1 = TR;
257 L1 = -(GM_jw_arg + atan(wc*TR))/wc;
258
259 disp(" ")
260 disp("K1: " + string(K1))
261 disp("L1: " + string(L1))
262 disp("T1: " + string(T1))
263 disp(" ")
264
265 L3 = L1 + L2;
266 lambda1 = 0.5*(L3);
267
268 KP1 = (T1 + lambda2 + (L3)^2/(2*(lambda1 + L3)))/(K1*(lambda1 + L3));
```

Matlab scripts

```
269 TI1 = T1 + lambda2 + (L3)^2/(2*(lambda1 + L3));
270 KI1 = KP1/TI1;
271 TD1 = (lambda2*T1 - (L3)^3/(6*(lambda1 + L3)))/(T1 + lambda2 + ...
        (L3)^2/(2*(lambda1 + L3))) + (L3)^2/(2*(lambda1 + L3));
272 KD1 = KP1*TD1;
273 TF1 = TD1*0.1;
274
275 disp('KP2: ' + string(KP2))
276 disp('KI2: ' + string(KI2))
277 disp('KD2: ' + string(KD2))
278
279 disp('KP1: ' + string(KP1))
280 disp('KI1: ' + string(KI1))
281 disp('KD1: ' + string(KD1))
282 disp('TF1: ' + string(TF1))
283
284 disp(T2)
285 disp(K2)
286 disp(L2)
```

Finding parameters of 'step response simultaneous FOPDT plus SOPDT' tuning:

```
1 %lsqnnoneg -> Tn -> normal SO T -> normal (1, 2)
2 %-----
3 close all
4 clear
5 clc
6
7 %Area method
8 %-----
9 %step_amount = 15;
10 initial_y = 0;
11
12 u_f = load('step_input.mat');
13 y_f = load('step_output2.mat');
14 %{
15 ut = u_f.ans(1, :);
16 ud = u_f.ans(1, :);
17 yt = y_f.ans(1, :);
18 yd = y_f.ans(1, :);
19
20 u_f.ans = timeseries(ud, 0.002);
21 y_f.ans = timeseries(yd, 0.002);
22 %}
23
24 step_amount = u_f.ans(2, 1);
```

Matlab scripts

```
25
26 y_t = y_f.ans(2, :)/step_amount;
27 time = y_f.ans(1, end);
28 step = y_f.ans(1, 2) - y_f.ans(1, 1);
29 x = [0:step:time];
30
31
32 y_ss = y_t(end);
33
34 y_diff = y_ss - y_t;
35
36 A1 = interpolate_int(x, y_diff);
37
38 K = y_ss;
39
40 LT = abs(A1)/K;
41 x2 = [0:step:LT];
42
43 y_diff2 = y_t - initial_y;
44 y_diff3 = y_diff2(1:(LT/step + 1));
45 %1, current_time, step
46 A2 = interpolate_int(x2, y_diff3);
47 a = y_diff2(1:(LT/step));
48
49 T = exp(1)*A2/K;
50 L = (A1 - K*T)/K;
51
52 K2 = K;
53 T2 = T;
54 L2 = L;
55
56 %LSQ
57 %-----
58
59 num = 3;
60 %num = 5;
61 den = 4;
62 unstable = 0;
63
64 y_f = load('step_output1.mat');
65 u_f = load('step_output2.mat');
66 %y_f = load('step_output1x.mat');
67 %u_f = load('step_output2x.mat');
68 %y_f = load('y_sq.mat');
69 %u_f = load('u_sq.mat');
70 y_t = y_f.ans(2, :);
71 u_t = u_f.ans(2, :);
72
73 K = time/step;
```

Matlab scripts

```
74 x = [0:step:time];
75
76 t_values = [1:1:K];
77
78 t_v = [0:step:((t_values(end) - 1)*step)];
79 t_v = rot90(t_v, -1);
80
81 syF = zeros(length(t_values), 1);
82 sM = zeros(length(t_values), den + num + 1);
83
84 sy = zeros(length(t_values), den);
85 su = zeros(length(t_values), num + 1);
86 yM = zeros(length(t_values), den + 1);
87 for n = 1:(length(t_values))
88     t_value = t_values(n);
89
90     sM(n, 1) = -y_t(t_value);
91     yM(n, 1) = y_t(t_value);
92
93     y_t_temp = y_t;
94     for nn = 1:den
95         current_index = n - nn;
96         current_time = t_value + 1 - nn;
97         if current_index > 0
98             y_t_temp = trapez_int(y_t_temp, 1, current_time, step);
99             sy(n-nn, nn) = y_t_temp(end);
100            yM(n-nn, nn+1) = y_t_temp(end);
101            if nn == den
102                if unstable == 1
103                    temp = -y_t_temp(end);
104                else
105                    temp = y_t_temp(end);
106                end
107                syF(n-nn) = temp;
108            else
109                sM(n-nn, nn+1) = -y_t_temp(end);
110            end
111        end
112    end
113
114    u_t_i = zeros(1, num + 1);
115    u_t_temp = u_t;
116    t_value = t_values(n);
117    for nn = 1:(num+1)
118        current_index = n - nn;
119        current_time = t_value + 1 - nn;
120        if current_index > 0
121            u_t_temp = trapez_int(u_t_temp, 1, current_time, step);
122            su(n-nn, nn) = u_t_temp(end);
```

Matlab scripts

```
123
124         sM(n-1, den + 1) = u_t_temp(end);
125     end
126 end
127 end
128
129 plottime = rot90(0:step:(length(yM)-1)*step, -1);
130 for n = 1:(den + 1)
131     figure(n)
132     plot(plottime, yM(:, n))
133 end
134
135 xsM = sM(1:(length(sM) - num - 1), :);
136 xsyF = syF(1:(length(syF) - num - 1), :);
137
138 c = lsqnonneg(xsM, xsyF);
139 %c = xsyF\xsM;
140 numerator = zeros(1, num + 1);
141 denominator = zeros(1, den + 1);
142 total_str = '[';
143 total_strx = ' ';
144 syms x
145 polN = 0;
146
147 for n = 1:(num + 1)
148     if c(den + n) >= 0
149         extra = '+';
150     else
151         extra = '';
152     end
153     total_strx = total_strx + extra + string(c(den + n)) + 'x^' + ...
154         string(num+1 - n) + ' ';
155     polN = polN + c(den + n)*x^(num+1-n);
156     total_str = total_str + string(c(den + n)) + ' ';
157     disp('n' + string(num + 1 - n) + ': ' + string(c(den + n)))
158     numerator(n) = c(den + n);
159 end
160 total_str2 = '[';
161 total_strx2 = ' ';
162 polD = 0;
163
164 for n = 1:(den)
165     if c(n) >= 0
166         extra = '+';
167     else
168         extra = '';
169     end
170     total_strx2 = total_strx2 + extra + string(c(n)) + 'x^' + ...
171         string(den - n) + ' ';
```


Matlab scripts

```
170     polD = polD + c(n)*x^(den-n);
171
172     total_str2 = total_str2 + string(c(n)) + ' ';
173     disp('d' + string(den + 1 - n) + ': ' + string(c(n)))
174     denominator(n) = c(n);
175 end
176 denominator(end) = 1;
177
178 total_str = total_str + ']';
179 total_str2 = total_str2 + '1]';
180
181 disp('Numerator: ' + total_str)
182 disp('Denominator: ' + total_str2)
183 disp('G1 = tf(' + total_str + ', ' + total_str2 + ');')
184 disp(total_strx)
185
186 G1 = tf(numerator,denominator);
187
188 %xxx: To Tn0 and C
189 %-----
190
191 lambda2 = 0.5*L2;
192 TI2 = T2 + L2^2/(2*(lambda2 + L2));
193 TD2 = (L2^2/(6*(lambda2 + L2)))*(3 - L2/(T2 + L2^2/(2*(lambda2 + L2))));
194 %KP2 = (T2 + (L2^2)/(2*lambda2 + 2*L2))/(K2*(lambda2 + L2));
195 KP2 = TI2/(K2*(lambda2 + L2));
196 KI2 = KP2/TI2;
197 %KD2 = KP2*TD2;
198 KD2 = 0;
199
200 C2 = tf([KD2 KP2 KI2],[1 0]);
201
202 GM = G1;
203
204 % -----
205 l = 10;
206 GMx = GM.Numerator(1);
207 GMx = GMx{1};
208 GMx2 = GM.Denominator(1);
209 GMx2 = GMx2{1};
210 if GMx(end) == 0 & GMx2(end) == 0
211     GMx = GMx(1:(end-1));
212     GMx2 = GMx2(1:(end-1));
213 end
214 GM = tf(GMx,GMx2)
215
216 [mag, phase, wout] = bode(GM);
217 figure(1)
218 bode(GM)
```

Matlab scripts

```
219 magnitude = zeros(1, length(wout));
220 for n = 1:length(wout)
221     magnitude(n) = 20*log10(mag(1, 1, n));
222 end
223 figure(2)
224 semilogx(wout, magnitude)
225 p(1).LineWidth = 2;
226 p(2).LineWidth = 2;
227 legend("P_1")
228 ylabel("Bode (dB)")
229 xlabel("Frequency (rad/s)")
230 ax = gca;
231 ax.FontSize = 22;
232
233 cross = 0;
234 wu = 0;
235 cross_closest = cross + 5;
236 for n = 1:length(wout)
237     if abs(cross - magnitude(n)) < abs(cross - cross_closest)
238         cross_closest = magnitude(n);
239         wu = wout(n);
240     end
241 end
242
243 Km = find_numerical(GM, 0);
244
245 if wu == 0
246     cross = 20*log10(Km) - 3;
247     cross_closest = cross + 5;
248     for n = 1:length(wout)
249         if abs(cross - magnitude(n)) < abs(cross - cross_closest)
250             cross_closest = magnitude(n);
251             wu = wout(n);
252         end
253     end
254 end
255
256 wu = round(wu, 4);
257 wiM = [(wu/1):(wu/1):wu];
258
259 A = zeros(1, 1);
260 B = zeros(1, 2);
261 test = zeros((1), 6);
262 disp('Km: ' + string(Km))
263 for n = 1:(1);
264     wi = wiM(n);
265     Gm_jwi = find_numerical(GM, wi);
266     Gm_jwi_mag = abs(Gm_jwi);
267
```

Matlab scripts

```
268     Gm_jwi_arg = angle(Gm_jwi);
269
270     B(n, 1) = Gm_jwi_mag^2 * wi^4;
271     B(n, 2) = Gm_jwi_mag^2 * wi^2;
272     A(n) = Km^2 - Gm_jwi_mag^2;
273
274     test(n, 1) = wi;
275     test(n, 2) = Km;
276     test(n, 3) = Gm_jwi_mag;
277     test(n, 4) = A(n);
278     test(n, 5) = B(n, 1);
279     test(n, 6) = B(n, 2);
280 end
281
282 X = B\A
283 tau_m = nthroot(X(1), 4);
284 gamma_m = sqrt(X(2)/(4*tau_m^2) + 0.5);
285 phi_m = (pi + atan2(-2*tau_m*gamma_m*wu, 1 - tau_m^2 * wu^2))/wu;
286
287 disp('taul = ' + string(tau_m) + ';'')
288 disp('gammal = ' + string(gamma_m) + ';'')
289
290 disp('K1 = ' + string(Km) + ';'')
291 disp('L1 = ' + string(phi_m) + ';'')
292 %disp('phi_m: ' + string(phi_m))
293 disp('Denominator: [' + string(tau_m^2) + ' ' + string(2*tau_m*gamma_m) ...
      + ' 1]')
294
295 taul = tau_m;
296 gammal = gamma_m;
297 K1 = Km;
298 L1 = phi_m;
299
300 disp(" ")
301 disp("K1: " + string(K1))
302 disp("T1: " + string(taul))
303 disp("X1l: " + string(gammal))
304 disp("L1: " + string(L1))
305 disp("Denom: [" + string(taul^2) + " " + string(2*taul*gammal) + " 1]")
306 disp(" ")
307
308 lambda2 = 0.5*L2;
309 L3 = L1 + L2;
310 lambda1 = 0.5*(L3);
311
312 TI1 = 2*gammal*taul + lambda2 + L3^2/(2*(lambda1 + L3));
313 TD1 = (taul^2 + 2*taul*gammal*lambda2 - L3^2 / (6*(lambda2 + L3)))/TI1 ...
      + L3^2 / (2*(lambda1 + L3));
314 KP1 = TI1/(K1*(lambda1 + L3));
```

Matlab scripts

```
315 KI1 = KP1/TI1;
316 KD1 = KP1*TD1;
317 TF1 = TD1*0.1;
318
319 disp('KP2: ' + string(KP2))
320 disp('KI2: ' + string(KI2))
321 disp('KD2: ' + string(KD2))
322
323 disp('KP1: ' + string(KP1))
324 disp('KI1: ' + string(KI1))
325 disp('KD1: ' + string(KD1))
326 disp('TF1: ' + string(TF1))
```

Plotting result of simultaneous step response tuning:

```
1 close all
2 clear
3 clc
4
5 yend = 0.4;
6 ystart = 0;
7
8 timeset = 40;
9 tn = timeset/0.002 + 1;
10
11 s = load('step_output1.mat');
12
13 t = s.ans(1, 1:tn);
14 y = s.ans(2, 1:tn);
15
16 s2 = load('step_input.mat');
17 r = s2.ans(2, 1:tn);
18
19 s4 = load('disturbance');
20 d = s4.ans(2, 1:tn)*0.02;
21
22 p = plot(t, y, t, r, '--', t, d, 'black')
23 p(1).LineWidth = 2;
24 p(2).LineWidth = 2;
25 p(3).LineWidth = 1;
26 legend("Aero angle", "Reference", "Disturbances (V)")
27 ylabel("Angle (\phi)")
28 xlabel("time (s)")
29 ax = gca;
30 ax.FontSize = 22;
31 ylim([ystart, yend]);
```

Vedlegg B

Simulink Schemes

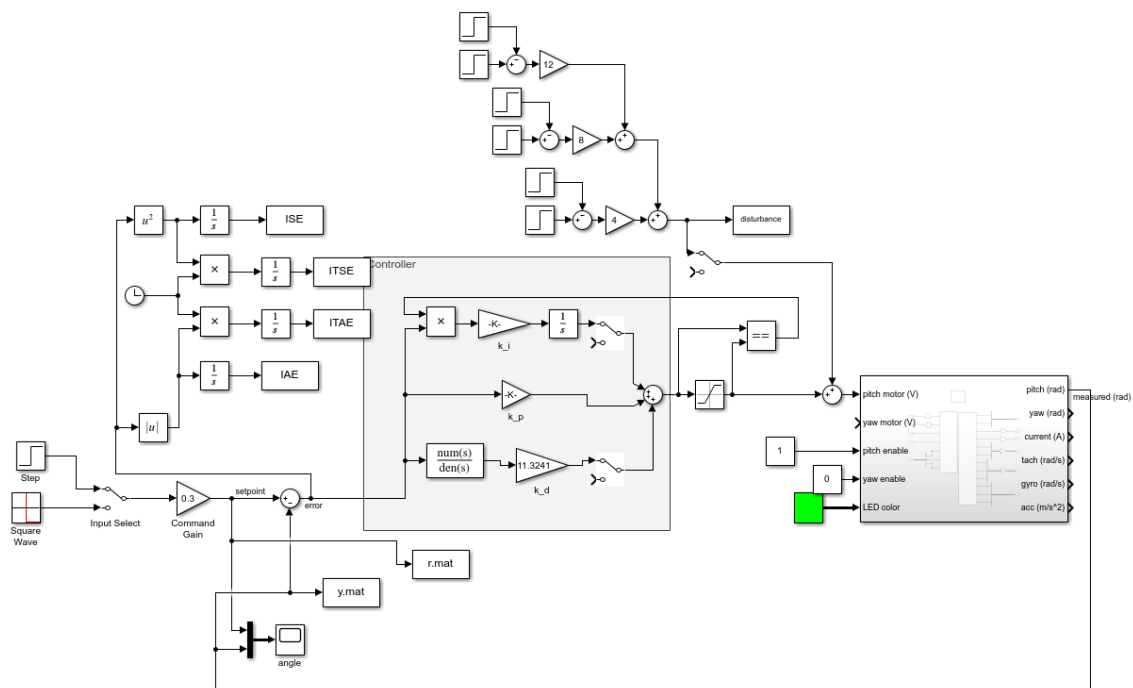


Figure B.1: Simulink scheme for single loop Ziegler-Nichols closed loop tuning

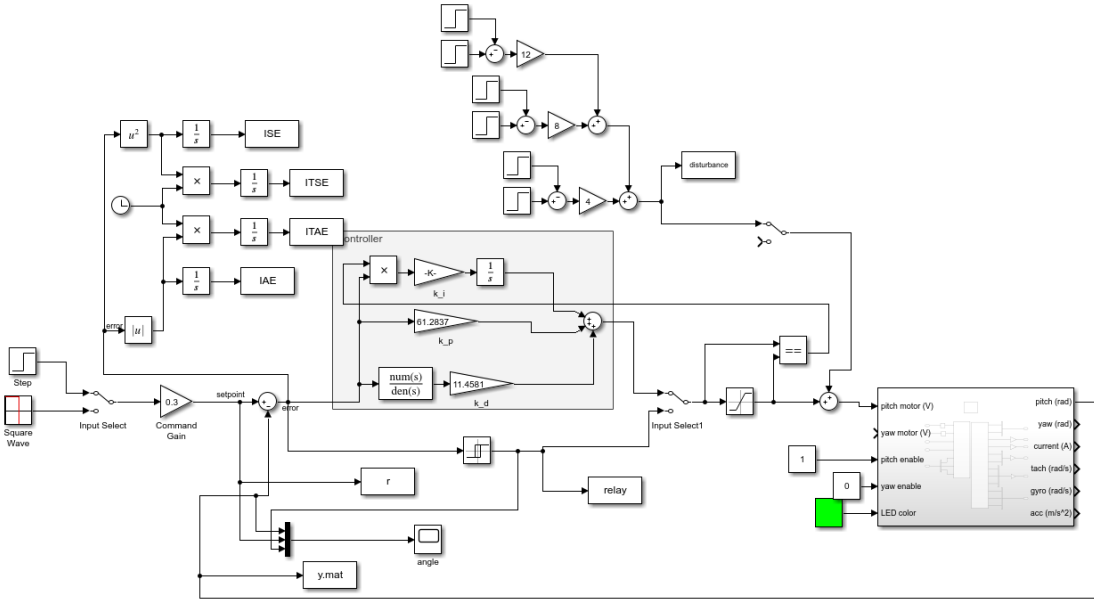


Figure B.2: Simulink scheme for single loop relay feedback tuning

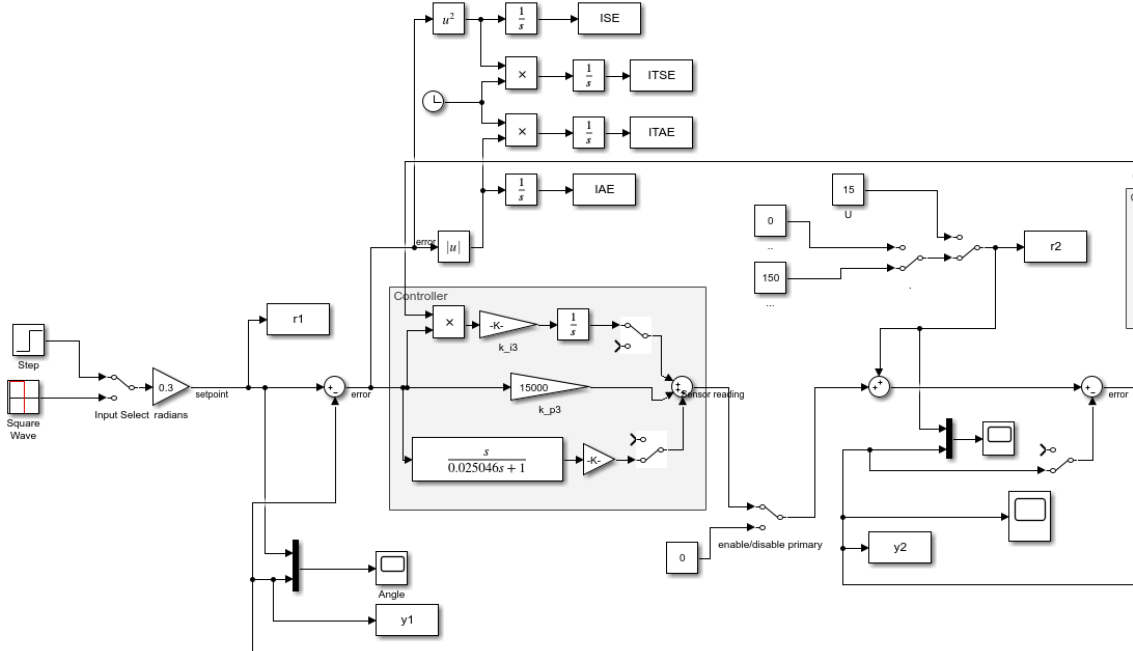


Figure B.3: Simulink scheme for single loop Ziegler-Nichols closed loop plus Ziegler-Nichols open loop tuning, left half

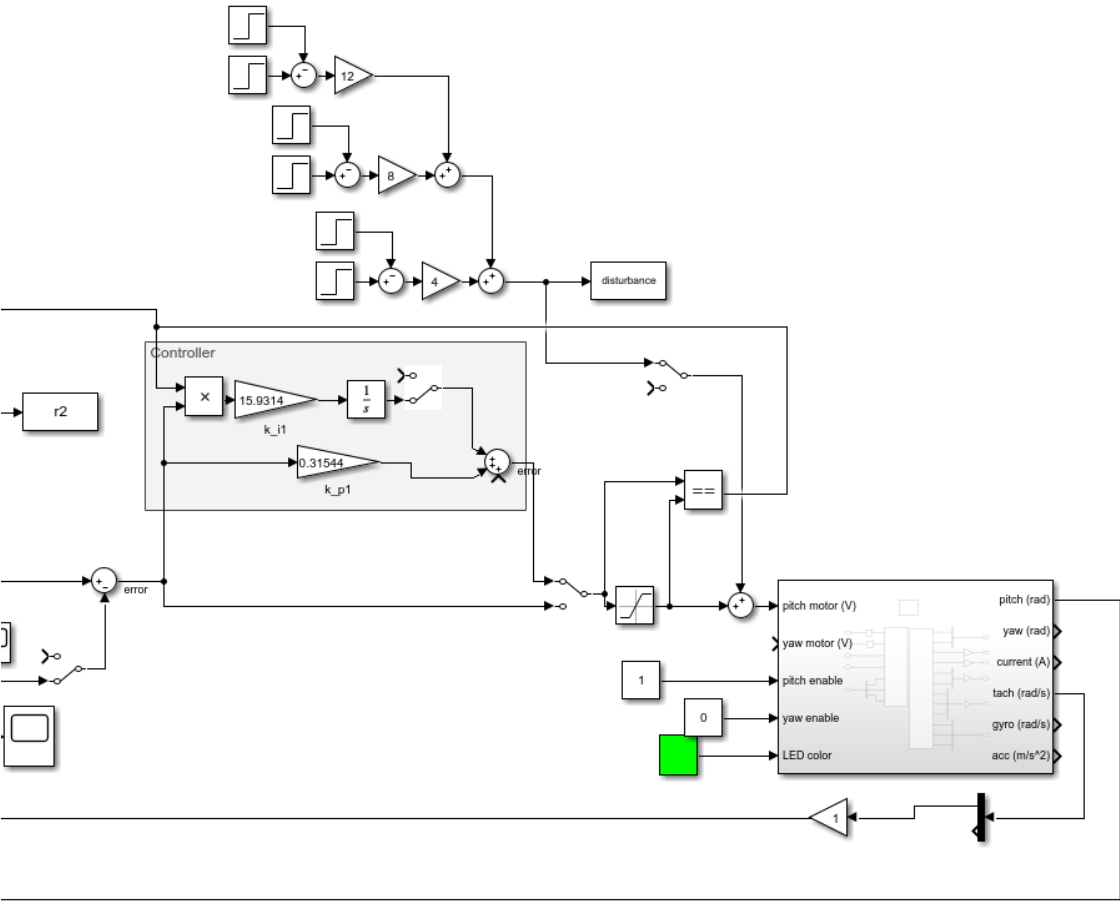


Figure B.4: Simulink scheme for single loop Ziegler-Nichols closed loop plus Ziegler-Nichols open loop tuning, right half

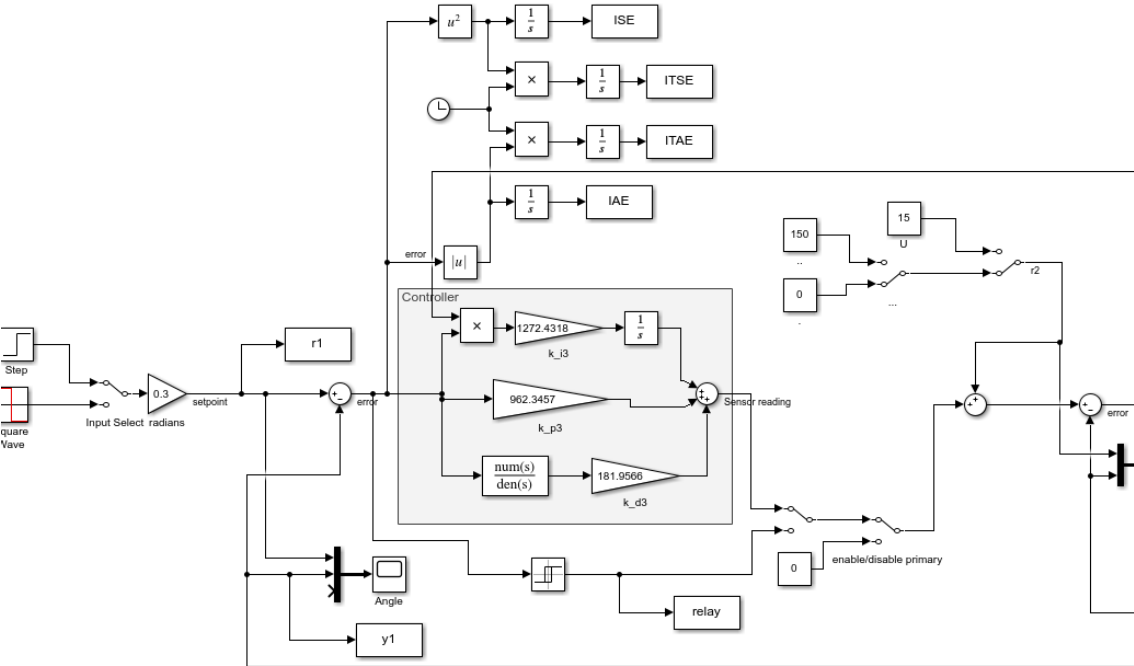


Figure B.5: Simulink scheme for single loop relay feedback plus Ziegler-Nichols open loop tuning, left half

Simulink Schemes

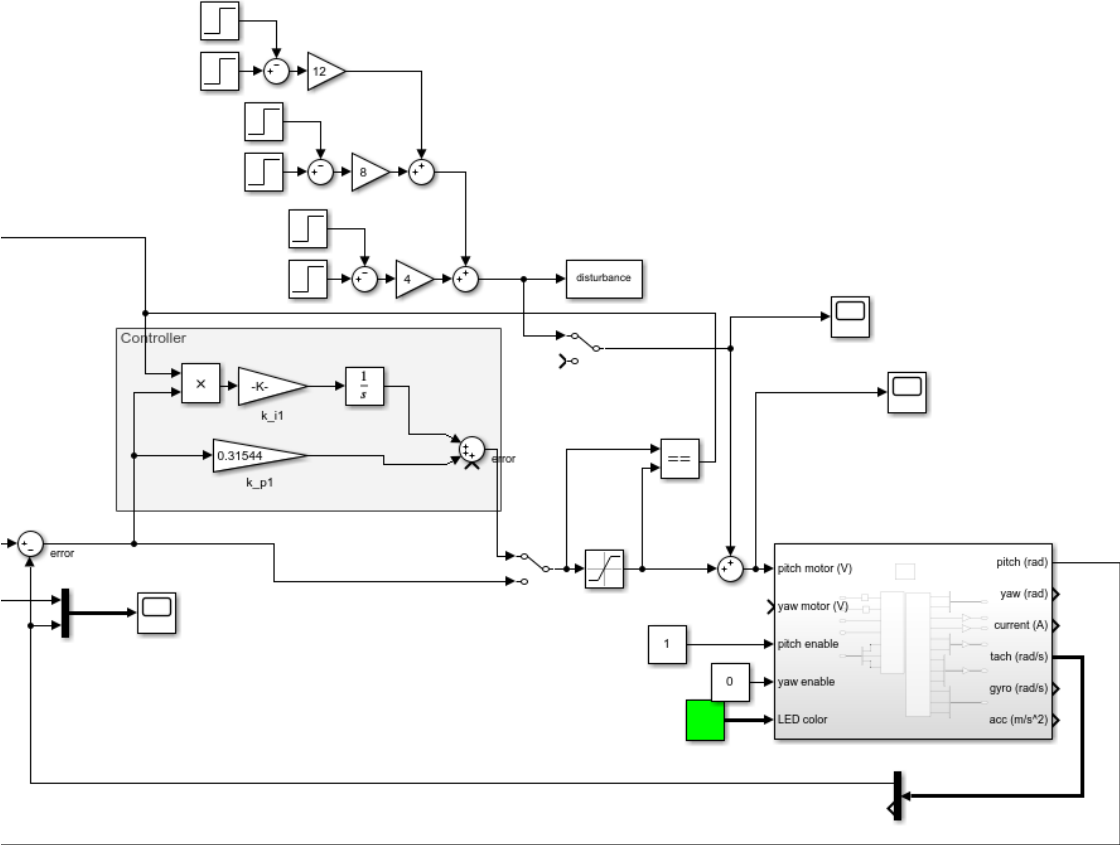


Figure B.6: Simulink scheme for single loop relay feedback plus Ziegler-Nichols open loop tuning, right half

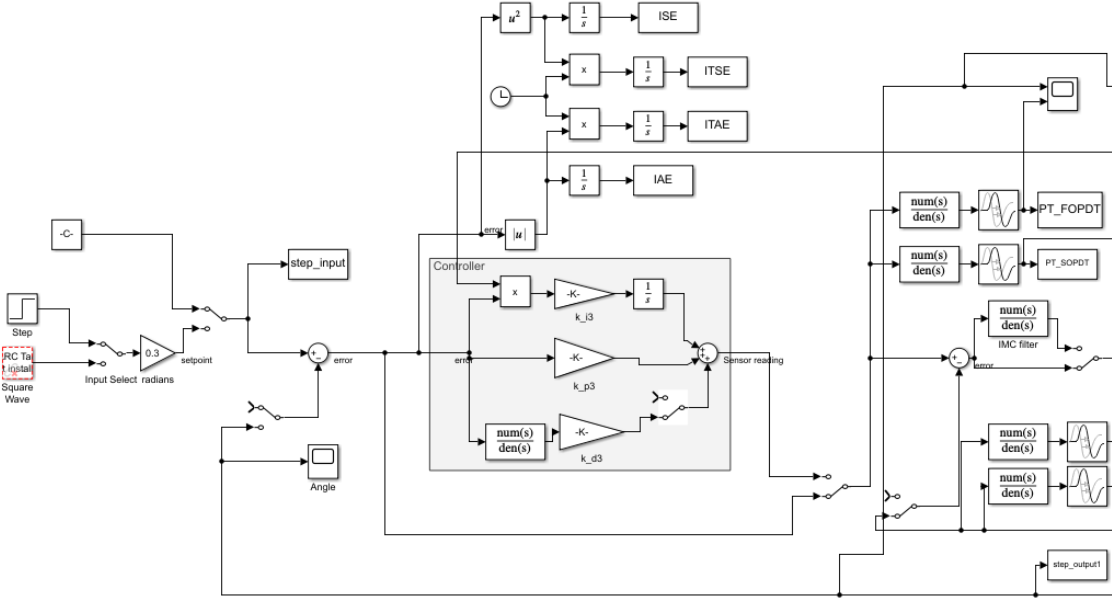


Figure B.7: Simulink scheme for simultaneous step response tuning (all approaches), left half

Simulink Schemes

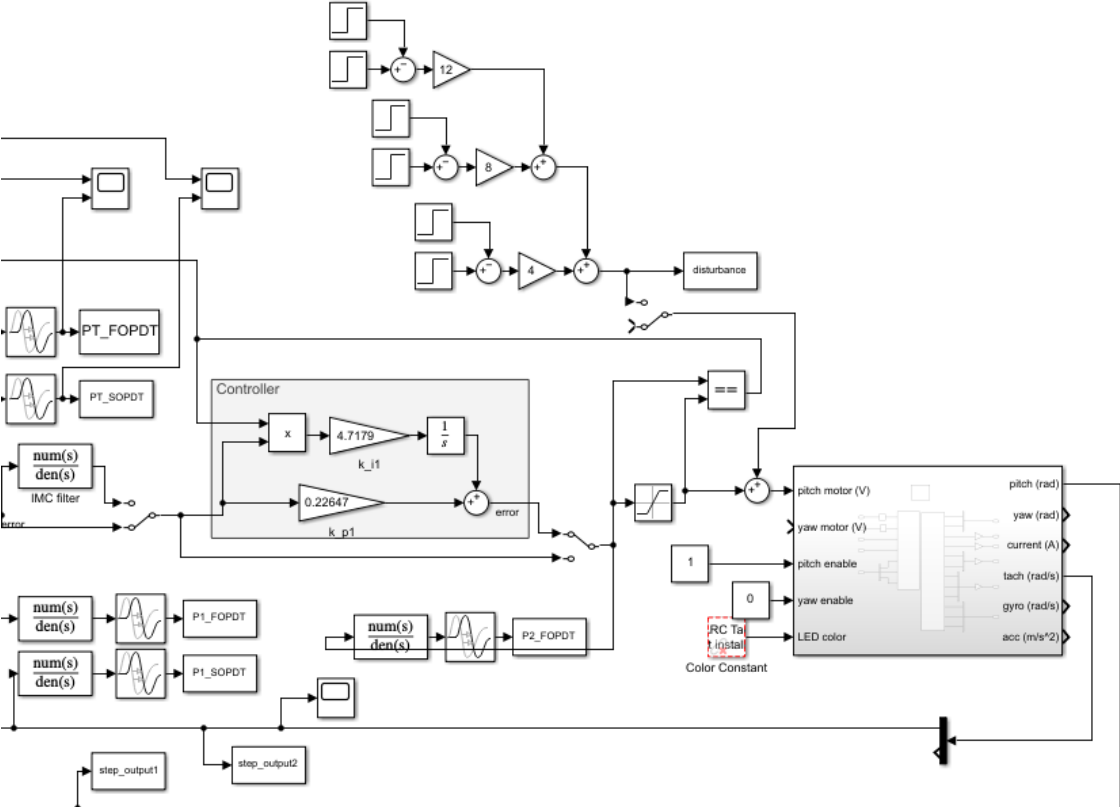


Figure B.8: Simulink scheme for simultaneous step response tuning (all approaches), right half