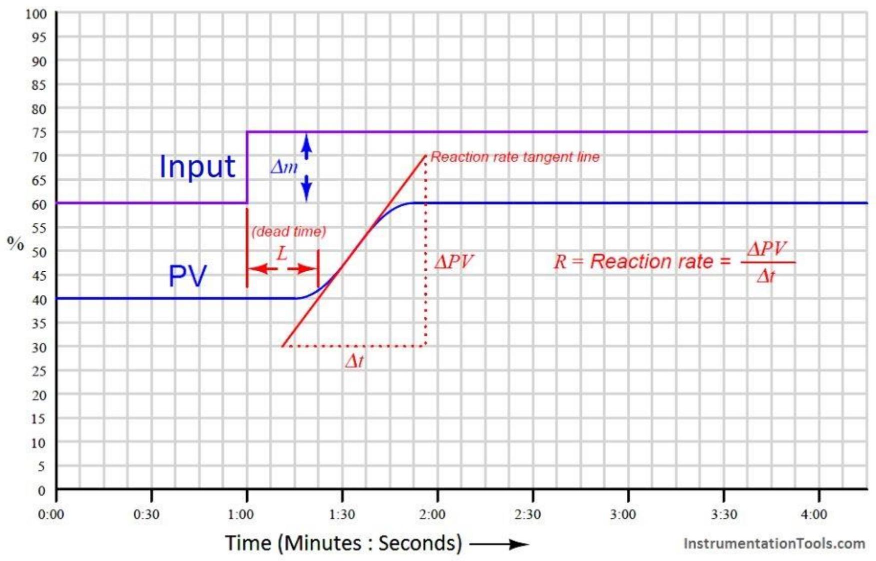
University of South-Eastern Norway

www.usn.no

FMH606 Master's Thesis 2022

Industrial IT and Automation

Tuning PID Controllers: From process experiments, general linear state space models, and tuning controllers via Process Reaction Curve Methods



Kevin Skogstad Kristiansen

Course: FMH606 Master's Thesis, 2022

Title: Tuning PID Controllers: From process experiments, general linear state space models and tuning PID controllers via Process Reaction Curve Methods

Number of pages: 69

Keywords: Process Reaction Curve, process control, PI controller, megatuner, pidtune

Student:	Kevin Skogstad Kristiansen
Supervisor:	David Luigi Di Ruscio
External partner:	**None**

Summary:

In today's industrial landscape, the necessity and usefulness of computer systems are undeniable. And with the need for computer systems and computer controller machinery, the interest in optimizing the efficiency of the computer system itself is highly emphasized.

This project is centered around testing and comparing the more popular forms of control, mainly process reaction curve methods. The objective of this thesis to present the reader with general information about the control process as well as presenting the theory behind processes. In addition, these forms of control should be tested on the random state-space models in order to view each controllers results and compare the controllers against each other.

This testing was done through the usage of MATLAB using seven different controllers for comparison. These controllers included the Ziegler-Nichols open-loop method, "megatunerplain", "megatuner", and two versions of both the "pidstd"- and "pidtune" methods. "megatuner" and "megatunerplain" being created by Christer Dalen.

With varying results from the different controllers, the Ziegler-Nichols open-loop method and the PID-version of the "pidstd" showed unstable and poor results. The PI -version of "pidstd" and "megatunerplain" showed overall stable results with "megatunerplain" being slightly better overall. "Pidtune" and "megatuner" returned high stable performance across the different sampling times and system orders. "Megatuner" was the only method to not experience a crash/unstable control across the project and its testing.

Preface

As a part of the master program at the University of South-Eastern Norway, a master thesis project is to be conducted for the 4th semester. This particular project is titled “Tuning PID Controllers: From process experiments, general linear state space models and tuning PID controllers via Process Reaction Curve Methods” and acts as a final project for the Master of Science in Industrial IT and Automation program.

While the main subjects, such as PID controllers and state-space models, are described in this project, the reader should have a general understanding of systems and system control, as well as a general understanding of the coding language MATLAB seeing as this is the sole language used throughout this project. The goal of this project is to get a general understanding of the many controllers available and how different random state space models may affect each controller.

I would like to express my sincerest gratitude towards my supervisor, Associate Professor David Luigi Di Ruscio. As a supervisor, his guidance has been a great help throughout this project, and his lecture notes and published reports have been a helpful source for a lot of information. I would also like to thank Christer Dalen for his MATLAB codes, especially the “megatuner”, as well as his and David Di Ruscios reports on the delta tuning principle.

The following software were used during the project:

- MS Office
- MS Visio
- MATLAB
- GeoGebra

Porsgrunn, May 18th, 2022

Kevin Skogstad Kristiansen

Contents

List of Figures

List of Tables

Nomenclature

Degrees of Freedom, DOF

Gain Margin, GM

Integral Absolute Error, IAE

Multiple-Input and Multiple-Output, MIMO

Phase Margin, PM

Process Reaction Curve, PRC

Single-Input and Single-Output, SISO

Ziegler-Nichols, ZN

Zero-Pole-Gain, ZPK

1 Introduction

In today's industrial landscape, the necessity and usefulness of computer systems are undeniable. And with the need for computer systems and computer-controlled machinery, the interest in optimizing the efficiency of the computer systems themselves are highly emphasized. The interest in an optimized process control may be based on several factors, such as minimizing the system cost, minimizing the strain on the system, maximizing the productivity of the system, etc. In order to increase the proficiency of the computer systems, the controllers must be tested and compared against each other in order to distinguish each controllers' flaws and benefits. The original task description is attached in Appendix A.

This project is centered around testing and comparing the more popular forms of control, mainly process reaction curve methods. The objective of this thesis is to present the reader with general information about process control, including tuning methods and examples, as well as presenting the theory behind processes and how a typical state-space process is described. In addition, these forms of control should be tested on the described state-space models in order to view each controllers results and compare the controllers against each other. This process is mainly done in the coding language MATLAB, which will be explained more thoroughly later in the thesis.

Previously, a master thesis written by Preben Sandve Solvang in 2019 named "State Space Model Based PID Controller Tuning" [1] compared several different controllers and tuning methods, however, that project was mainly based on the tuning process itself and the effects of the different controllers' parameters, whereas this project will be centered around testing the controllers. David Di Ruscio and Christer Dalen has also written several reports on a delta tuning method which builds upon Ziegler-Nichols original process reaction curve method and is to be compared in this project. [2] In addition to the delta tuning principle, a continuation of the method has been created in the form of a function named "megatuner" which is also included in this project. The MathWorks patented function "pidtune" has also been chosen as a fitting controller method with a separate interest in seeing a direct comparison between the "megatuner" and the "pidtune". [3]

Chapter 2 will go through the theory behind the different parts of the project, such as different controllers, tuning methods, and state space models

Chapter 3 goes over the usage of MATLAB and its implementation of the theory, as well as the MATLAB-specific controllers used in the project

Chapter 4 displays the results from the MATLAB codes, and the comparisons between the controllers

Chapter 5 discusses the different problems and changes in the project which were experienced, as well as general thoughts around the project and suggestions for further work

Chapter 6 concludes the thesis with a general summary of the project and its outcome

2 Theory

Seeing as the project revolves around the efficiency of system control from different controllers, it is important to understand the different parts of system control. Based on the actions done in this project, explained further in Chapter 3, the "Theory" chapter will explain the theoretical basis for the different elements of testing, such as the different control methods, system conversions, and the foundation for the systems themselves.

2.1 PID Controllers

A PID controller is a process control system which utilizes a closed loop system in order to achieve stability, and reaching a given setpoint, in a process. Due to being a closed loop system, the controller uses feedback as a way to calculate the error between the setpoint and the actual value. The PID controller is separated into three different terms, the P-term, the I-term, and the D-term.

The P-term denotes a proportional response on the error, usually described as $K_p e(t)$, where K_p is the proportional constant and the $e(t)$ is the error at the given time. The I-term is based on integrating the error, thereby being responsible for the process to reach the given setpoint, by minimizing the error. This is usually described as $K_i \int_0^t e(\tau)d\tau$, where the K_i is the integral gain and $\int_0^t e(\tau)d\tau$ is the integral which is minimizing the error. The D-term is a derivative term which introduces a dampening effect. This term is usually described as $K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t}$, where the K_d is the derivative gain and the $\frac{\mathrm{d}e(t)}{\mathrm{d}t}$ is the derivative of the error, resulting in the dampening effect.

The full mathematical equation for the PID controller is given below, in equation 2-1. An alternate version of the mathematical equation is given below as equation 2-2. The alternation replaces K_i and K_d with $\frac{K_p}{T_i}$ and $\frac{K_p}{T_d}$, respectively. This is often done to signify the relationship between the I-term, D-term, and time, rather than having separate numerical constants for each term.

$$u(t) = K_p e(t) + K_d \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \qquad \text{2-1}$$

$$u(t) = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau)d\tau + \frac{1}{T_d} \frac{\mathrm{d}e(t)}{\mathrm{d}t} \right) \qquad \text{2-2}$$

A PID controller is often shortened to a PI controller, i.e., removing the D-term. This is done seeing as a PI controller is often enough to control a process, while the D-term may be responsive to noise/disturbances in the process, thereby increasing the chance for instability. The mathematical model for a PI controller is displayed in equation 2-3.

$$u(t) = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau)d\tau \right) \qquad \text{2-3}$$

The PI- and PID controller may also be described in a discrete manner. A discrete formulation of a PID controller is given in equation 2-4, while a discrete formulation of a PI controller is given in equation 2-5. [4]

$$\begin{aligned} z_{k+1} &= z_k + \Delta t \frac{K_p}{T_i} e_k \\ u_k &= z_k + K_p e_k - K_p T_d \frac{y_k - y_{k-1}}{\Delta t} \end{aligned} \qquad 2\text{-}4$$

$$\begin{aligned} z_{k+1} &= z_k + \Delta t \frac{K_p}{T_i} e_k \\ u_k &= z_k + K_p e_k \end{aligned} \qquad 2\text{-}5$$

2.1.1 Ziegler Nichols Tuning

The Ziegler-Nichols (ZN) tuning method is a closed-loop tuning method for PID controllers, which also works for P- and PI controllers, in which the goal is to create oscillations in the process signal and calculating the resulting controller settings from the time and amplitude of the oscillation. The settings for the different controllers are given below, in Table 1. Upon achieving steady oscillation in the process signal, the amplitude of the oscillations is described as K_u while the period of the oscillations is described as P_u. An ideal process control, with the ZN-method, should result in a quartering of the amplitude for each oscillation.

Table 1: Settings for ZN tuning [5]

	K_p	**T_i**	**T_d**
P controller	$0.5K_u$	0	0
PI controller	$0.45K_u$	$0.8P_u$	0
PID controller	$0.6K_u$	$0.5P_u$	$0.125P_u$

2.1.2 Process Reaction Curve Tuning

There is also an open-loop tuning method for controllers. This method is called the ZN open-loop method, but it's also referred to as a Process Reaction Curve (PRC) method, or the ZN PRC tuning method. Instead of relying on oscillation in a closed loop system, the process is introduced to a step increase, or decrease, in the signal, and the resulting reaction is used as the basis for the tuning.

The step response results in three variables necessary to tune the process according to ZN PRC. These variables are R, L, and Δm. R is the reaction rate and is defined in equation 2-6. This is dictated by the tangential slope of the process reaction. Both Δt and ΔPV is read at two chosen points on the tangential line in order to calculate R. L is the dead time, or lag, of the process, in minutes. [5] Δm is the magnitude of the step change and is often set to "1" to simplify the tuning process. The tuning settings for the different controllers are displayed in Table 2.

$$R = \frac{\Delta PV}{\Delta t} = \frac{[Percent\ increase]}{[Minutes\ run]} \qquad 2\text{-}6$$

Table 2: Setting for ZN PRC Tuning [5]

	K_p	**T_i**	**T_d**
P controller	$\frac{\Delta m}{RL}$	0	0
PI controller	$0.9\frac{\Delta m}{RL}$	$3.3L$	0
PID controller	$1.2\frac{\Delta m}{RL}$	$2L$	$0.5L$

A graphical display of the PRC method, based on the different values and variables, is shown below in Figure 2-1. This figure shows a step increase of 15%, and the resulting response, with the different variables displayed.

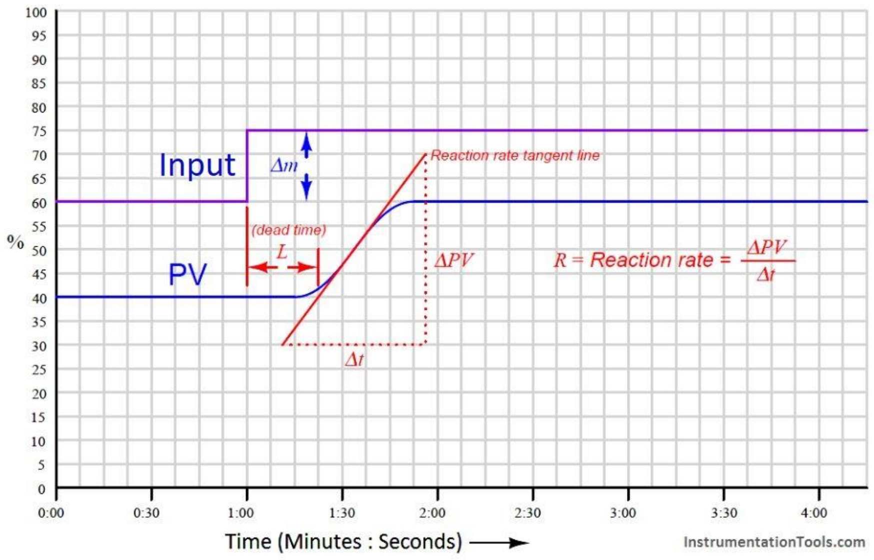


Figure 2-1: Graphical display of PRC method [6]

2.1.3 Delta Tuning

Through several articles, written by Christer Dalen and David Di Ruscio, tuning rules for delta tuning, δ-tuning, have been presented.

The first report from Di Ruscio [7] introduces the concept with a PI controller and expands upon the concept to include double integrated time delay as well as a PD- and PID controllers [8]. The final two reports are based on implementing the δ-tuning method with a PRC approach. This PRC approach is firstly done as a PID controller [9], then expanded to include a PI controller as well as testing for higher order systems [10].

The first report results in δ-tuning based on a ratio between the maximum time delay error and the time delay, given as $\delta = \frac{d\tau_{max}}{\tau}$, and a product parameter, given as $\bar{c} = \alpha\beta$. The α and β is further described as $\alpha = \frac{a}{\delta+1}$ and $\beta = \frac{\bar{c}}{a}(\delta + 1)$, where a is a constant in $\bar{c}$. The resulting PI controller parameters are given as $K_p = \frac{\alpha}{k\tau}$ and $T_i = \beta\tau$.

The second report expands upon the previous work to include the derivative term. This term is given as $T_d = \beta\tau$ whereas the integral term has been changed to $T_i = \gamma\beta$, where γ is defined as a relative integral derivative time ratio, and the proportional term is changed to $K_p = \frac{\alpha}{K\tau T_d}$. These findings result in a final PID controller describes as in equation 2-7.

$$
\begin{gathered}
K_p = \frac{\alpha}{K\tau T_d} \\
T_i = \gamma T_d \\
T_d = \beta\tau \\
\alpha = \frac{a}{\delta + 1}, \qquad \beta = \frac{\overline{c}}{a}(\delta + 1)
\end{gathered}
\tag{2-7}
$$

The final two reports revolve around a PRC method for the δ-tuning method, first proposed in Dalen and Ruscios report from 2018 [2]. This PRC method introduces the variable K, defined as $K = \frac{R}{L}$ and is the gain acceleration, and defines the time delay as $\tau = \frac{L}{2\pi}$. This is later expanded upon resulting in equation 2-8. While ζ is defined between 0 and 10, the chosen values are either 1 or 6. $\zeta = 1$ is based on the original formulation, while $\zeta = 6$ is an alternative for processes with time delay approximately equal the system order, for $n > 3$.

$$
\begin{gathered}
K = \zeta\frac{\mathrm{R}}{\mathrm{L}}, \qquad 0 < \zeta \leq 10 \\
\tau = \eta L, \qquad \eta = \frac{1}{2\pi}
\end{gathered}
\tag{2-8}
$$

2.2 State-Space Models

State-space models are models, based on differential equations, which represents a system. A state-space model consists only of 1st order differential equations, and the order of the system dictates the number of differential equations. These systems may be both linear and non-linear, as well as having a varying number of inputs, outputs, as well as states. The “states” refers to the innate variables of the system, which itself is unrelated to any measurement or physical quantity [11]. The number of inputs and outputs would also reflect whether the system is a Multiple-Input and Multiple-Output (MIMO), Single-Input and Single-Output (SISO), or a combination of the two. This project is centered around SISO-systems.

2.2.1 Formulation of a State-Space Model

This subchapter includes a formulation of a state-space model, and this formulation is based on Bernard Friedland’s formulation [11]. A state-space model consists of states, and the state-equations, x, for a state-space model may be formulated as in equation 2-9, below. This description is done based on a k-order state-space model.

$$
\begin{aligned}
\dot{x_1} &= \frac{dx_1}{dt} = f(x_1, x_2, \dots, x_k, u_1, u_2, \dots, u_l, t) \\
\dot{x_2} &= \frac{dx_2}{dt} = f(x_1, x_2, \dots, x_k, u_1, u_2, \dots, u_l, t) \\
&\vdots \\
\dot{x_k} &= \frac{dx_k}{dt} = f(x_1, x_2, \dots, x_k, u_1, u_2, \dots, u_l, t)
\end{aligned}
\tag{2-9}
$$

In addition, a description of two separate vector, encapsulating the different x- and u-values, are displayed below, in equation 2-10. The resulting equation, based on the equations displayed above, is given in equation 2-11.

$$
x = \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix}, \qquad u = \begin{bmatrix} u_1 \\ \vdots \\ u_l \end{bmatrix}
\tag{2-10}
$$

$$
\dot{x} = \frac{dx}{dt} = f(x, u, t)
\tag{2-11}
$$

The above formulation for x is a time-variant system and may be converted into a time-invariant system. This conversion is based on separating the coefficients of the input signal and the states in separate matrices, often named A and B.

The output formulation, y, is similar to the input formulation, x, with a difference in the coefficient matrices, where the matrices is often named C and D. In addition, the D-matrix, the coefficient matrix for the control signal in the output, is usually set to zero for control purposes to remove a direct connection between control signal, u, and the output, y. The continuous formulation for a state-space model, which includes both the input and output formulations, is given in equation 2-12, with the structure of the A- and B-matrices given in equation 2-13. In this formulation, each matrix is given a lowercase c to signify that the matrices are for a continuous system.

$$\begin{aligned} \dot{x} &= A_c x + B_c u \\ y &= C_c x \end{aligned} \qquad \text{2-12}$$

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,k} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k,1} & a_{k,2} & \cdots & a_{k,k} \end{bmatrix}, \qquad B = \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,l} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,l} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k,1} & b_{k,2} & \cdots & b_{k,l} \end{bmatrix} \qquad \text{2-13}$$

A discrete formulation of the above system is given below in equation 2-14. Here, the matrices are given a lowercase d to signify that the system is discrete. It is important to note that $A_c \neq A_d$, which also goes for the other matrices.

$$\begin{aligned} \dot{x_k} &= A_d x_k + B_d u_k \\ y_k &= C_d x_k \end{aligned} \qquad \text{2-14}$$

From the previously given formulas for a state-space model, a graphical representation may be given as in Figure 2-2. Here, the input vector, u, is multiplied with its coefficient matrix, B, and summed with the state vector, x, and its coefficient matrix, A. Then the term is integrated, resulting in the state vector for the next step, given as in equations 2-12 and 2-14. This state vector is then multiplied with the next coefficient matrix, C, resulting in the output vector, y.

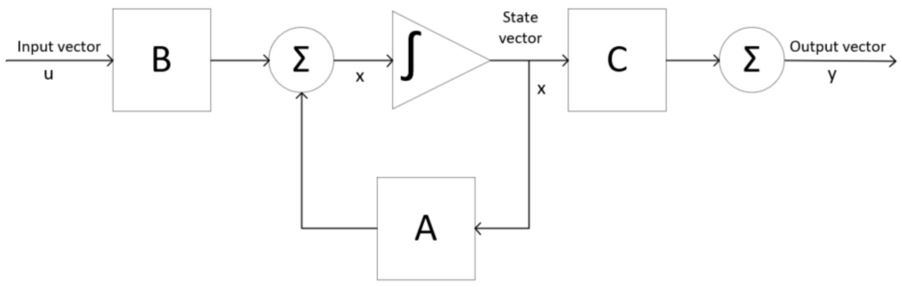


Figure 2-2: Block diagram of a state-space model

2.2.2 Transfer Functions

In order to create PID controller based on state-space models, the models must be converted into a transfer function. The conversion is done through Laplace, and the model is then transferred into what is known as the “s-plane”. The transfer function is based on the systems impulse response, i.e., the correlation between the control signal, u, and the output, y. This correlation is described as in equation 2-15, below.

$$Hp(s) = \frac{Y(s)}{U(s)} \qquad 2\text{-}15$$

From the continuous state-space model, displayed in equation 2-12, and the above equation, the model may now be Laplace transformed into the model shown below.

$$sX(s) = AX(s) + BU(s)$$
$$Y = CX(s)$$

Seeing as the transfer function is based on $\frac{Y(s)}{U(s)}$, the $X(s)$ must be replaced in the function for $Y(s)$. This may be done according to the below formulation.

$$sX(s) - AX(s) = BU(s)$$
$$(sI - A)X(s) = BU(s)$$
$$X(s) = (sI - A)^{-1}BU(s)$$
$$Y = C(sI - A)^{-1}BU(s)$$

From the above formulation, a new equation for $Y(s)$ is given, and when this function is divided by $U(s)$, according to the aforementioned equation 2-15, the resulting transfer function is then described as in equation 2-16.

$$Hp(s) = \frac{Y(s)}{U(s)} = C(sI - A)^{-1}B \qquad 2\text{-}16$$

As an addition to the above transfer function, the model may be multiplied with $e^{-\tau s}$ to add a time delay. An example of this is given below, in equation 2-17.

$$Hp(s) = \frac{Y(s)}{U(s)} = e^{-\tau s}(C(sI - A)^{-1}B) \qquad 2\text{-}17$$

2.3 Numerical Comparisons

There are several different factors which may have an effect on the outcome of the controllers. Different factor may also have different effects on each controller based on how the controller operates. In addition, seeing as most testing will be based on several runs and having the resulting average, tests such as standard deviation would be inefficient seeing as the system itself changes from iteration to iteration, unless the standard deviation is done on relative results rather than numerical results.

This project will compare the performance of the different controllers based on the results from testing the controllers on random systems. This may include results like the speed of the controllers, possible overshoot from the controllers, and possible crashes of the controllers.

There are also parameters for the systems which may affect the results from the controllers. These includes parameters such as the order of the system, the timestep/sampling time of the controller, possible time delays in the system, etc.

In addition, the Integral Absolute Error (IAE) should be calculated in order to have a general understanding of the error of all controllers across the several systems. The formula for the IAE is given in equation 2-18. The $r(t)$ is the reference/setpoint at the timestep while the $y(t)$ is the system output at the timestep. A relative IAE should also be added by having the relative error in a time independent percentage, thereby giving grounds for utilizing standard deviation seeing as a fraction returns a more stable result regardless of the particular system used.

$$IAE = \int |e(t)|dt = \int |r(t) - y(t)|dt \qquad 2\text{-}18$$

3 MATLAB

The main workload of this master's thesis lies in the usage of MATLAB. MATLAB is a programming software, with a self-named programming language, owned, and created, by MathWorks. The programming language is centered around matrices, thereby the name MATLAB which is short for "Matrix laboratory", and has a focus on simulation, mathematics, and manipulation of data. This focus allows for more compact programs without the need for unnecessary startup or configuration code. Seeing as the project revolves around controllers, for which MATLAB has several built-in functions, and their respective data, MATLAB is being utilized to create programs to manipulate this data, such as standard deviations, means, graphs, and other form of comparisons between the controllers.

In addition to the standard MATLAB, a separate package, also created by MathWorks, has been added in order to support several of the necessary processes in this project. This package is named "Control System Toolbox" and adds functionality to MATLAB in terms of transfer functions, state-space models, and frequency analysis, as well as adding functionality for tuning SISO- and MIMO systems.

3.1 State-Space Models

As stated in the project goal, the controllers are to be compared based on their efficiency with random state-space models. Seeing as most models are discrete based on a set of sampled data, the general idea is to create a discrete system, which is then converted to a continuous system, reflecting most of industrial situations. These state space-models may be created through several different functions. The functions used in this project are the "drss" function, which stands for "Discrete random state space models", and the "rss" function, meaning "Random state space models" and creates continuous models.

The "drss" function creates a random discrete state space model but may also involve non-physical systems where one, or more, of the eigenvalues of the A-matrix is negative, which is impossible in a physical process. To circumvent this, the "rss" function has been used in order to create continuous systems, which are then converted into a discrete system, which in turn always results in a physical process due to the nature of the conversion between continuous to discrete instead of discrete to continuous.

The relationship between a continuous system and a discrete system, as well as a more detailed formulation of the discrete system matrix, is shown in equation 3-1. This formulation of a discrete system shows that the A_d-matrix is formulated through the exponential $e^{A_c \Delta t}$, and seeing as the result of an exponential must be positive, the A_d-matrix may never be negative. This matrix may be randomly assigned a negative value through the "drss" function, resulting in the ensuing problems when converting to a continuous system.

$$\begin{gathered} \dot{x} = A_c x \\ x_{k+1} = A_d x_k \\ A_d = e^{A_c \Delta t} \end{gathered} \tag{3-1}$$

The “drss” function utilizes three parameters which determines the order, number of inputs, and number of outputs of the system, with input and output set to “1” as default creating a SISO system. This state space model is then created as four matrices, A, B, C, and D, with D being set to zero.

The discrete state space model is then converted to a continuous state space model through the use of the function “d2c”, meaning “Discrete to continuous”. The conversion may be done through several different methods, but the project mainly revolved around the “zoh”-, zero order hold, and the “tustin” method. For the continuous systems, created through “rss”, the method “c2d”, meaning “Continuous to discrete”, is used to convert the system into a physical discrete system and then the aforementioned “d2c” is used.

The “zoh” method creates continuous time inputs by holding the sampled values constant over the sampled times. The two limitations for this method are the inability to convert systems with poles at $z = 0$, and when converting systems with negative poles, the order is increased in order to create pole conjugates which avoids imaginary values. [12]

The “tustin” method is a trapezoidal conversion method, i.e., a “standard” integration-based conversion. This method is time-invariable and does therefore not affect the order of the system when converting the system. A limitation wo the “tustin” conversion is the inability to convert when a pole is $z = -1$, and is general ill-conditioned when nearing poles at $z = -1$. [12]

The continuous state space model is then turned into a transfer function. This conversion is done with the “ss2tf” function, with the continuous matrices as parameters. This function converts continuous state space model, which is the reasoning behind the previously conversion from discrete. After the creation of the transfer function, the transfer function is multiplied with $e^{-\tau s}$, which acts as a time delay in the system.

3.1.1 Physical System Example

Based on the aforementioned code and functions in MATLAB, an example of the creation and conversion process of a physical system is to be displayed. Through the use of the “rss” function and the “c2d” function, the randomly created continuous system is displayed in equation 3-2, with the resulting discrete system displayed in equation 3-3. This particular system is a 1st order system.

$$\begin{aligned} \dot{x} &= -2.152x - 1.161u \\ y &= 2.377x \end{aligned} \tag{3-2}$$

$$\begin{aligned} x_{k+1} &= 0.1162x_k - 0.4765u_k \\ y_k &= 2.377x_k \end{aligned} \tag{3-3}$$

This conversion is done with the “zoh” method. However, since the continuous system may have poles approximating, or equal, zero, the code has a simple check of the system poles and utilizes the “tustin” method if there are poles approximating zero. As mentioned in chapter 2.2.1, the D-matrix is set to zero, resulting in an output formula only consisting of the C-matrix, which is then unchanged between the conversions. Finally, a transfer function was created using the “ss2tf” function and was multiplied with the $e^{-\tau s}$, where $\tau = 0.1$. The results from these executions are displayed in equation 3-4.

$$hp = e^{-0.1s}\frac{-2.759}{s + 2.152} \tag{3-4}$$

3.1.2 Non-Physical System Example

From the previously mentioned differences between converting a continuous system into a discrete system and creating a discrete system directly from “drss”, an example is depicted through the use of both the “zoh”- and “tustin” method in order to exemplify these differences. Equation 3-5, below, depicts a randomly created state-space model using the “drss” function, which is a non-physical system seeing as the eigenvalues of the A-matrix, the singular value seeing av the A-matrix is a $1x1$-matrix in this example, is negative. In addition, the “ddcgain” function, specifically for discrete systems, was used for computing the system gain. This particular system is a 1st order system.

$$\begin{gathered} x_{k+1} = -0.346x_k + 0.1719u_k \\ y_k = -0.04747x_k \\ Gain = -0.0061 \end{gathered} \tag{3-5}$$

Firstly, the steady-state system was converted into a continuous system using the “d2c” function with the “tustin” method. The system gain was computed using the “dcgain”. Finally, a transfer function was created using the “ss2tf” function and was multiplied with the $e^{-\tau s}$, where $\tau = 0.1$. The results from these executions are displayed in Table 3.

Table 3: Result from state-space model conversion using "tustin"

$\dot{x} = -4.116x_1 + 0.5257u_1$
$y = -0.1452x_1 + 0.01248u_1$
$Gain = -0.0185$
$hp = e^{-0.1s}\frac{0.01248s - 0.02496}{s + 4.116}$

Secondly, the steady-state system was converted into a continuous system using the “d2c” function with the “zoh” method. The system gain was computed using the “dcgain”. Finally, a transfer function was created using the “ss2tf” function and was multiplied with the $e^{-\tau s}$, where $\tau = 0.1$. The results from these executions are displayed in Table 4.

Table 4: Results from state-space model conversion using "zoh"

$$\dot{x} = \begin{bmatrix} -1.061 & 3.142 \\ -3.142 & -1.061 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0.1356 \\ 0.4013 \end{bmatrix} u_1$$

$$y = \begin{bmatrix} -0.04747 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$Gain = -0.0061$$

$$hp = e^{-0.1s} \frac{-0.006437s - 0.06668}{s^2 + 2.123s + 11}$$

The main differences between the approaches are visible in both the order of the continuous system, which is also reflected in the order of the transfer function, but also the gain of the system. The “tustin” method keeps the order of the system, but has a different gain, while the “zoh” method changes the order and keeps the same system gain. The order of the system is changed with the “zoh” method in order to have a continuous system with only positive eigenvalues for the A-matrix.

From Table 4, it is seen that the order of the matrix has increased. This is due to the negative A-matrix, more specifically the negative eigenvalue, in the discrete system, resulting in the creation of a conjugate pair in the continuous system in order to avoid negative eigenvalues. This conjugate pair is reflected in the identical values of the new matrix, with possible opposite signs, in order to create positive eigenvalues. Table 5, below, shows another example of the conjugate pair creation from the “zoh” conversion method. The discrete system, created from “drss”, is a 3rd order system. Seeing as the A-matrix, for the discrete system, is $3x3$, there are three eigenvalues, given in the λ-matrix. Seeing ass all three eigenvalues are negative, three conjugate pairs are created, resulting in a 6th order continuous system. If just one or two of the eigenvalues were negative, the resulting continuous system would be of 4th or 5th order, respectively. The conjugate pairs are also reflected in the similar values along the diagonals of the A-matrix. The values given as ≈ 0 is written seeing as MATLAB returned values as low as e-17, resulting in different values between the pairs due to being close to zero.

Table 5: Conjugate pair example with higher order system

$$x_{k+1} = \begin{bmatrix} -0.1721 & 0.004845 & 0.2187 \\ 0.004845 & -0.08567 & -0.1096 \\ 0.2187 & -0.1096 & -0.4978 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 1.533 \\ 0 \\ 0 \end{bmatrix} u_1$$

$$y_k = \begin{bmatrix} -0.2256 & 1.117 & -1.089 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\lambda = \begin{bmatrix} -0.6263 \\ -0.1088 \\ -0.0205 \end{bmatrix}$$

$$- \quad - \quad -$$

$$\dot{x} = \begin{bmatrix} -2.494 & 0.5388 & -1.091 & 1.345 & -2.13 & 1.877 \\ 0.5388 & -2.926 & 0.7697 & -0.5758 & -2.238 & -2.128 \\ -1.091 & 0.7697 & -1.155 & -2.78 & -0.5669 & -1.349 \\ -1.345 & 0.5758 & 2.78 & -0.468 & \approx 0 & \approx 0 \\ 2.13 & 2.238 & 0.5669 & \approx 0 & -2.218 & \approx 0 \\ -1.877 & 2.128 & -1.349 & \approx 0 & \approx 0 & -3.889 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} + \begin{bmatrix} 3.575 \\ -0.9169 \\ 1.706 \\ 1.267 \\ -2.945 \\ 2.819 \end{bmatrix} u_1$$

$$y = \begin{bmatrix} -0.2256 & 1.117 & -1.089 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

3.2 Controllers

Through the usage of MATLAB, several different controllers have been utilized. These controllers include a ZN PRC controller, described in chapter 2.1.2, the PI- and PID-versions of the “pidstd”, PI- and PID-versions of the “pidtune”, as well as two functions, created by Christer Dalen, named “megatuner” and “megatunerplain”.

3.2.1 Pidstd

“Pidstd” is the original standard PID controller for systems in MATLAB and is primarily used as a controller with parameter inputs. The “pidstd” function allows for creation of PID controllers, as well as P- and PI controllers, and allows for adding a filter constant. The function also allows the user to add a sample time, which in turn results in a discrete controller, rather than a continuous controller which is normally returned. The function has capabilities of converting a discrete dynamic system, which must represent a PID controller, into a standard PID controller formulation, in accordance with equation 2-2. [13]

Since the function itself has no capability of auto-tuning a given process, an additional tuning function must be used in order to give “pidstd” the necessary parameters to successfully control a process. Seeing as the main focus of this project is the PRC method of tuning, the “pidstd” function is used in combination with a function created by Christer Dalen in order to calculate the relevant K_p, T_i, and T_d and use these values as parameters for the PID controller. These methods were the “delta_prc_pi_tun” function for the PI controller, and the “delta_prc_pid_tun1” function for the PID controller. Both functions are a result from the aforementioned reports from Dalen and Di Ruscio, mentioned in chapter 2.1.3.

3.2.2 Pidtune

“Pidtune” is a MATLAB function, created and patented by MathWorks. The function is a proprietary function which is used to create P-, PI-, or PID controllers tuned for the given system. The “pidtune” function allows for discrete and continuous system, as well as unstable and stable systems, and varying time delays. However, the system must be a SISO system. The function allows for an array of SISO systems, and thereby returns a similar array of controllers.

The function also requires a “type” which determines the returned controller. This may be the various versions of PID controllers or 2-degrees of freedom (DOF) PID controllers, with or without a filter. The function also allows a weighing matrix and some customizable options. [3]

There is also a function, created by MathWorks, named “pidTuner” which is an interactive PID controller for tuning systems, utilizing the “pidtune”, and is used through a separate application page. In this application, several different parameters may be customized, such as frequency or time domain, parallel or standard controller, as well as response time and transient behavior. [14] A system may be imported, and a display shows a graph of the control of the system, and how the different parameters affect the control. A picture of the “pidTuner” application is displayed in Figure 3-1.

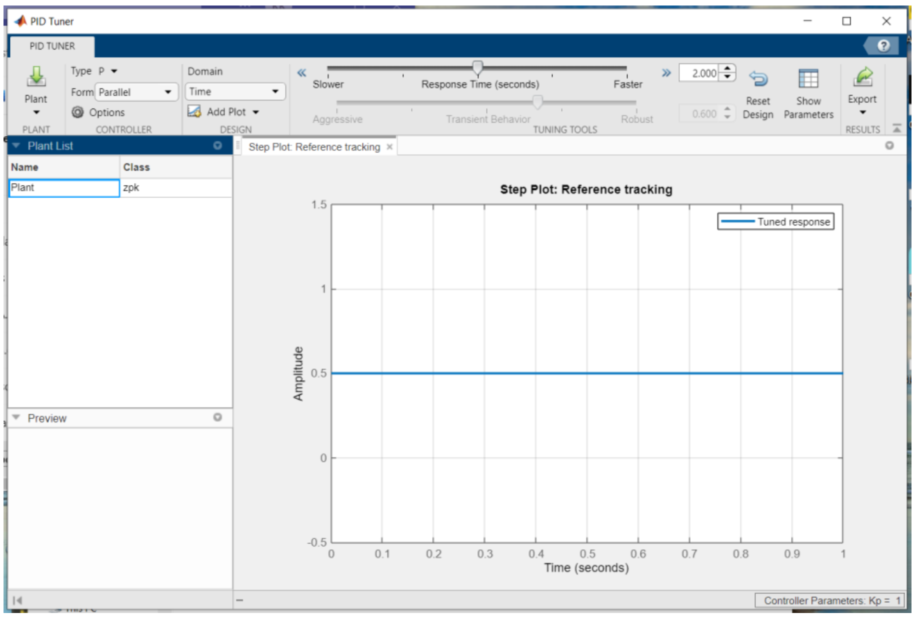


Figure 3-1: A picture of the "pidTuner" application

Seeing as the function is patented, the source code is unavailable. However, the patent itself, which may be found, depicts the general process of the function. This depiction, which is a flowchart, is displayed in Figure 3-2. It is also presented in the patent that the flowcharts, and other graphic documentation, are not universally relevant for all tuning of all systems, but that the general process is described through this documentation. Some of the earlier stages of the flowchart, in this case the linearization and system identification, may be skipped based on the given system. [15]

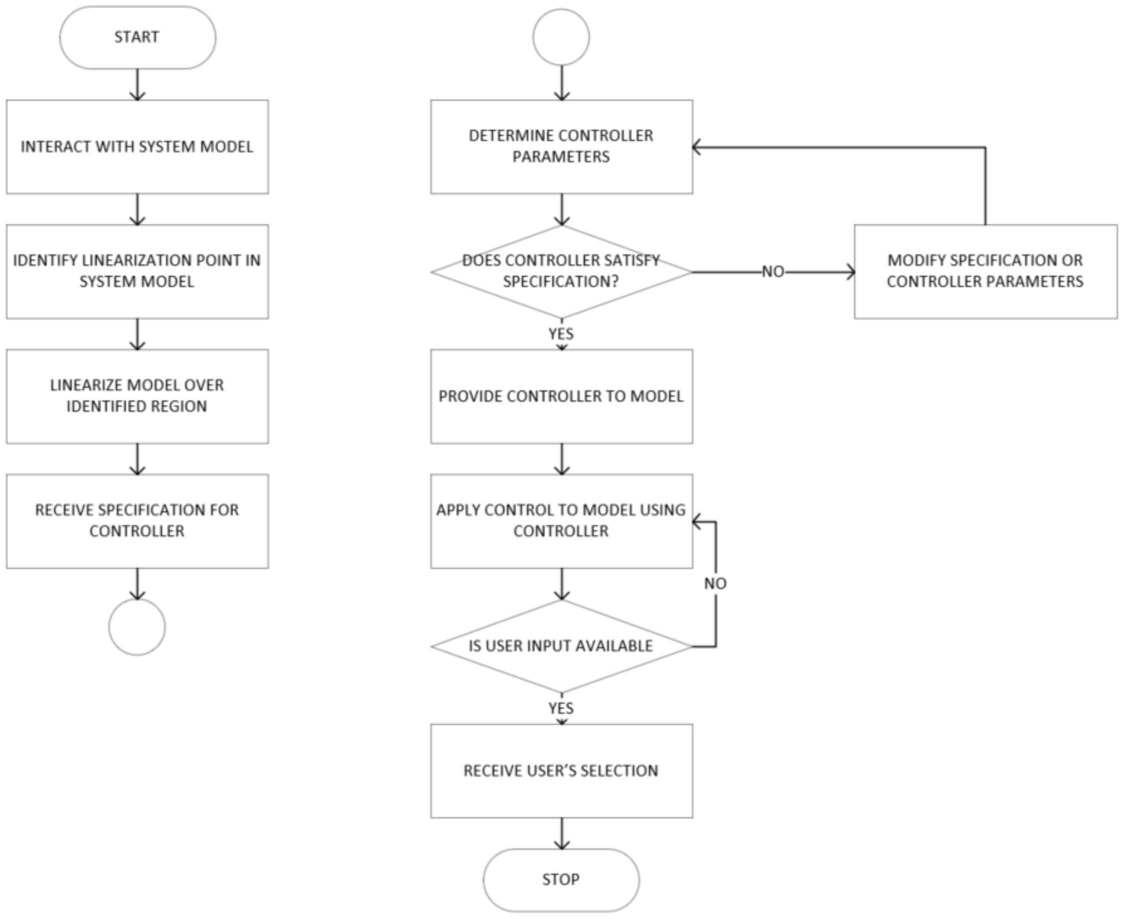

START
INTERACT WITH SYSTEM MODEL
IDENTIFY LINEARIZATION POINT IN SYSTEM MODEL
LINEARIZE MODEL OVER IDENTIFIED REGION
RECEIVE SPECIFICATION FOR CONTROLLER
DETERMINE CONTROLLER PARAMETERS
DOES CONTROLLER SATISFY SPECIFICATION?
NO
MODIFY SPECIFICATION OR CONTROLLER PARAMETERS
YES
PROVIDE CONTROLLER TO MODEL
APPLY CONTROL TO MODEL USING CONTROLLER
NO
IS USER INPUT AVAILABLE
YES
RECEIVE USER'S SELECTION
STOP

Figure 3-2: The function flowchart from the PIDTune patent [15]

3.2.3 Megatuner

Several different MATLAB functions has previously been created in order to perform the different versions of δ-tuning and PRC δ- tuning. From the Di Ruscio and Dalen documents [7] - [10], some of these methods include the documented “pi_tun_maxdelay”, “pd_tun_maxdelay”, delta_prc_pi_tun”, and “delta_prc_pid_tun1”. These methods are all based on in theory behind δ-tuning, and as an undocumented, and unpublicized, continuation to this work, Christer Dalen has created another two functions, named “mftun” and “megatuner”.

The ”mftun” function is not used in this particular process, however, a master’s thesis named “State Space Model Based PID Controller Tuning”, written by Preben Sandva Solvang, explains and tests the “mftun” function in comparison with several other functions based on the δ-tuning method, as well as standard tuning methods. [1]

“Megatuner” is aptly named after its successful tuning of a million, mega, state-space models. The “megatuner” builds upon the “mftun” function, and as an extension the δ-tuning principles, in an attempt to tune random steady-state systems with a focus on robustness and performance. Performance is referring to the accuracy of the controller, while robustness refers to the controllers ability to function and remain stable despite uncertainty in the different parameters.

The “mftun” utilizes either the δ-PRC method or the general PRC method based on the eigenvalues. The “megatuner” expands upon this with a frequency analysis in order to create a controller with higher performance and robustness. A frequency analysis is based upon a varying input signal, which is the frequency analysis is a sinusoidal signal, in order to test the correlation between the system input and system output at steady state with a varying signal. [4] The input- and output signals are described in equations 3-6 and 3-7, where u_0 and y_0 are the signals, α and β are the phase shifts, and ω is the frequency of the sinusoidal signal.

$$u(t) = u_0 \sin(\omega t + \alpha) \tag{3-6}$$

$$y(t) = y_0 \sin(\omega t + \beta) \tag{3-7}$$

The sinusoidal input- and output signals are then used to calculate the phase margin (PM) and the gain margin (GM), which in turn is the basis for the system analysis in terms of stability. The equations for the GM and PM are given in equations 3-8 and 3-9.

$$GM = \frac{1}{|H_0(j\omega_{180})|} \tag{3-8}$$

$$PM = \angle H_0(j\omega_c) + 180° \tag{3-9}$$

In addition to the “megatuner” function, Dalen has also created a simplified function named “megatunerplain”. This function operates on the same basis as “megatuner” including both δ-tuning and frequency analysis, however the “megatunerplain” forgoes some of the more specificities of the “megatuner”, such as additional checks for optimal performance and stability. While the “megatunerplain” is a simplified version of the “megatuner”, the function has been tested together with the “megatuner”, and other controllers, in order to compare its results with the rest of the controllers.

3.3 Testing

In order to test the several controllers against each other, a MATLAB script had to be made in order to run each controller based on a randomly created state space model. A script named “demo” was supplied from the supervisor in which a random state space model was created, converted into a transfer function with a time delay of $e^{-\tau s}$, and then gave the user to option to insert a number in order to choose which controller to run. The controllers included in this script was the ZN PRC-tuning method, both the PI- and PID-versions of the “pidstd” function, the “megatunerplain”, and the “megatuner”. The script would then print three figures. The first figure was the open loop step response, the second figure depicted the reference step response, and the third figure showed the input disturbance step response.

This script was expanded upon to automatically run all five controllers and print the resulting 15 graphs in a 5x3 plot array where each row represented a controller. In addition to the figures, the script also prints the IAE and the relative IAE, which is a percentage. The printout from the expanded upon "demo"-script, now named "demo_V3", is displayed in Figure 3-3. The code for the "Demo_V3" script is attached in Appendix B.

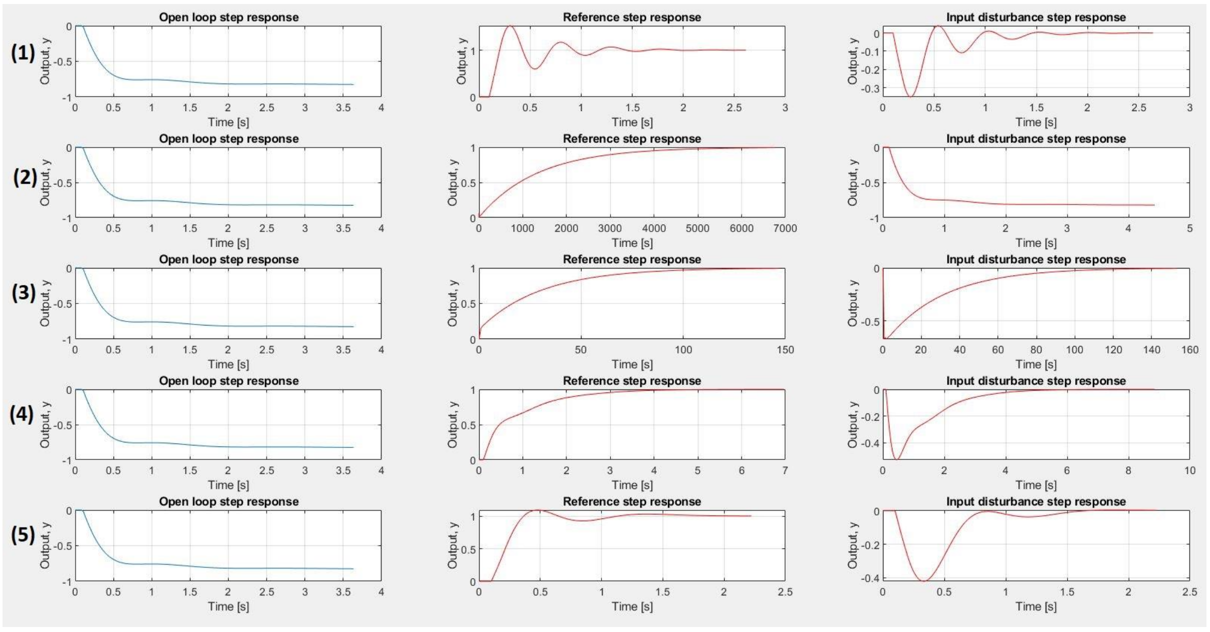


Figure 3-3: Printout from testing each controller on the same random state space model. (1) is the ZN PRC-tuning method, (2) is the PID controller from "pidstd", (3) is the PI controller from "pidstd", (4) is the "megatunerplain", and (5) is the "megatuner".

The general process of the "demo_V3" script has been separated into several different functions in other to simplify the process of testing the controllers as well as creating a streamlined code. These functions include codes for creating a random state space model, the controller execution, and the printing of data. The testing has been done by running the functions from the "demo_V3" script in a loop and printing out the average results from the different values gathered.

Based on the differences between physical- and non-physical systems, described in chapter 3.1, each test is done for 50 iterations on both "rss"- and "drss"-created systems, thereby doubling the number of tests. This is done to see the possible impact of having random systems with higher order or systems with negative A_d matrices.

In order to test the effect of varying the timestep, order, and possible time delay, a "standard" has been set with a timestep of 1 second and an order of 3 in order to vary one element at a time. This results in all controllers having a sampling time of 1 while testing different orders, and all systems having an order of 3 when varying the timestep. This standard is set as a semi-arbitrary middle ground while still having reasonable realistic sizes of the systems created where the system itself is neither highly complex nor too simple and neither a particularly fast or slow.

Different versions of the functions and scripts have been created in order to systematically test the different functions without changing much code, e.g., creating separate functions for "rss"- and "drss"-created systems, and different scripts in order to separate ZN PRC, "pidstd(PID)", "pidstd(PI)", "megatunerplain", and "megatuner" from the "megatuner", "pidtune(PID)", and "pidtune(PI)". The codes included in the appendices only displays one version of each code seeing as there are only minor differences between the different versions. The main script, with relevant self-made functions, are attached in Appendix C. All results are displayed in graphs and tables throughout Chapter 4 as well as comments and comparisons of the result, with further discussion around the results, and the project, being written in Chapter 5.

4 Results

This chapter will present and describe the result given from the methods presented in the previous chapters. These results include the comparisons between the controllers based on the different parameters previously given, and the efficiency of each controller. The results have been cleared of all outliers and the data portrayed is the final data, after clearing any outliers. In the tables displaying the results, the relative IAE is shortened to “Rel. IAE, “Average” is shortened to “Avg”, “Standard deviation” is shortened to “STD”, and “Number of crashes” are written as “# of crashes”. “STD. IAE” is the standard deviation of the relative IAE, not the IAE itself. The IAE is based on relative IAE times the time usage and is therefore proportional to the time usage.

The relative IAE is based on the average error over the time used. This results in a time-invariant number representing the overall slope of error, i.e., a lower number represents a longer time interval with little error, while a higher number represents overall more error in the control. This would then mean that a lower relative IAE means a quicker control to the setpoint with a slower stabilization while a higher relative IAE would mean a slower control which in turn stabilizes quickly upon reaching the setpoint.

4.1 Comparison Between ZN, Pidstd(PID), Pidstd(PI), Megatunerplain, and Megatuner

The first part of the controller comparison is between ZN PRC, “pidstd(PID)”, “pidstd(PI)”, “megatunerplain” and “megatuner” based on several different system tests. All five controllers were run on the same 50, randomly created, systems, first with varying timestep, then with varying order. All results are averaged from the 50 runs for that test.

4.1.1 Timestep

As a comparison between the effects of the timestep in the system, a run with 50 systems were executed for each timestep with “rss”-created systems. These timesteps included 0.1s, 0.5s, 1s, and 5s in order to test the different controllers and portray differences between each controller based on the speed of the system.

The average time usage for each controller across the different sampling times is displayed in Figure 4-1, below. From this presentation, the time usage of the “pidstd(PID)” seems to be comparative with the four other controllers at both the 0.1s- and 0.5s-systems, with a slight deviation when the sampling time is set to 1s and a major deviation at 5s.

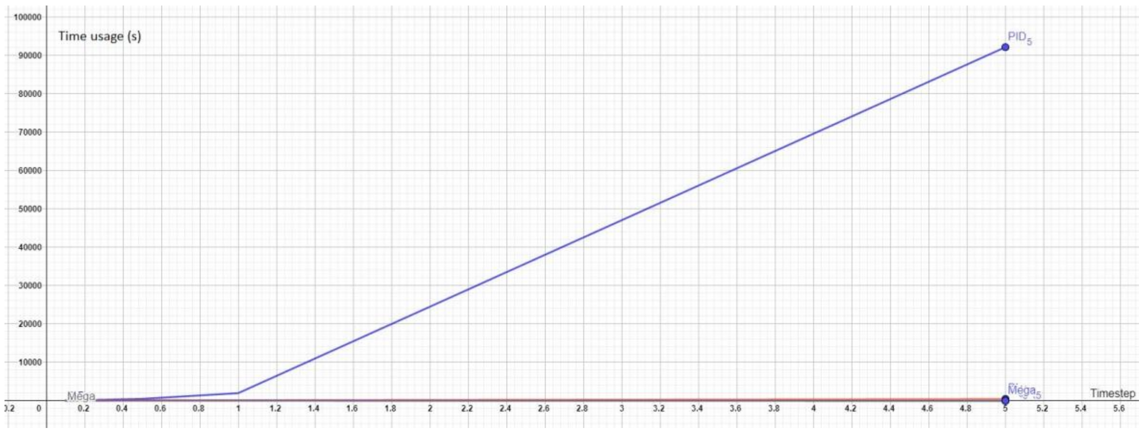


Figure 4-1: Test results of time usage (y-axis) from ZN, “pidstd(PID)”, “pidstd(PI)”, “megatunerplain”, and “megatuner” using “rss”-created systems with varying timestep (x-axis). See Figure 4-2 for addition details.

Seeing as the “pidstd(PID)” has a final time usage of 92 000 per system when the timestep is set to 5s, a separate graph, excluding “pidstd(PID)”, is displayed in Figure 4-2. From this display, the “pidstd(PID)” seems to deviate already at a sampling time of 0.5s, “pidstd(PI)” averaging notably better while still taking longer than the ZN PRC, “megatunerplain”, and “megatuner”. The “megatunerplain” averages generally slower than the ZN PRC and “megatuner” but is unaffected by the timestep of the controllers. Both the ZN PRC and the “megatuner” returns low average time and appears to even quicken at the increased timesteps.

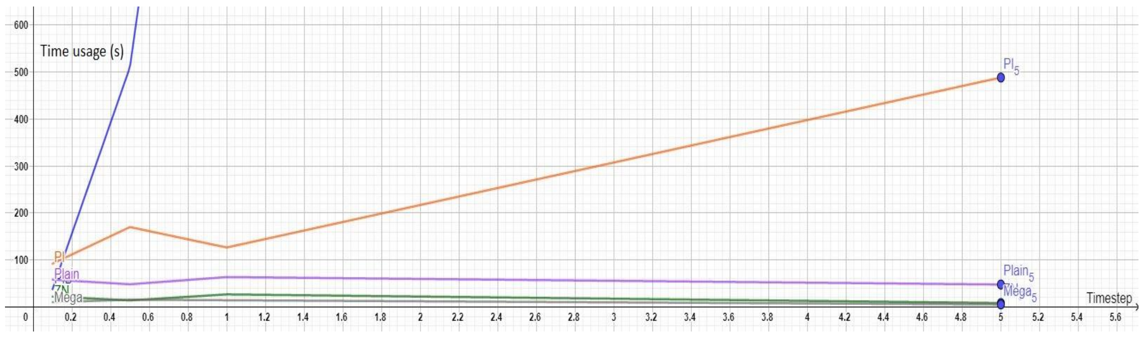


Figure 4-2: Test results of time usage (y-axis) from ZN, “pidstd(PI)”, “megatunerplain”, and “megatuner” using “rss”-created systems with varying timestep (x-axis)

In addition to the time usage, the IAE, peak, and number of crashes/unstable control is returned from the testing. A relative, time-invariant, IAE is also returned including the standard deviation for this relative IAE. This data is displayed in Table 6, segmented into each of the timestep tests. In this table, the relative IAE is shortened to “Rel. IAE, “Average” is shortened to “Avg”, “Standard deviation” is shortened to “STD”, and “Number of crashes” are written as “# of crashes”. From the table, the relative IAE varies from as low as 0.1583, for the “megatuner”, to as high as 0.3103, for the “pidstd(PID)”. In terms of crashes, neither the “megatunerplain” nor the “megatuner” experienced any crashes, with “pidstd(PI)” experiencing a couple sporadically. Both the “pidstd(PID)” and the ZN PRC experienced a higher number of crashes at lower timesteps, with decreasing instability as the timestep was increased. The “pidstd(PID)” experienced lower and lower peaks as the timestep increased resulting in an average of 0.891 at 5s, which would mean the process did not stabilize at the setpoint. Both the “megatuner” and the ZN PRC showed increased peaks as higher timestep which is a typical trait of an oscillating process control. All controllers return relatively stable and low standard deviations, while the “pidstd(PID)” experienced high fluctuation between tests.

Table 6: Test results from ZN, “pidstd(PID)”, “pidstd(PI)”, “megatunerplain”, and “megatuner” using “rss”-created systems with varying timestep

	ZN PRC	**Pidstd(PID)**	**Pidstd(PI)**	**Megatunerplain**	**Megatuner**
Timestep=0.1, Order=3					
Rel. IAE (Avg. IAE)	0.1954 (4.36)	0.2114 (8.05)	0.1738 (16.05)	0.2141 (12.47)	0.1897 (1.95)
STD. IAE	0.0461	0.1902	0.0397	0.0284	0.0658
Avg. Peak	1.591	1.035	1.057	1.001	1.105
# of crashes	7	11	1	0	0
Timestep=0.5, Order=3					
Rel. IAE (Avg. IAE)	0.2030 (2.86)	0.2389 (121.82)	0.1818 (30.92)	0.2117 (10.25)	0.1914 (2.99)
STD. IAE	0.0385	0.0129	0.0483	0.0344	0.0730
Avg. Peak	1.695	0.998	0.999	1.001	1.107
# of crashes	8	5	0	0	0
Timestep=1, Order=3					
Rel. IAE (Avg. IAE)	0.1974 (5.32)	0.2570 (507.96)	0.1752 (22.18)	0.2146 (13.68)	0.1885 (2.73)
STD. IAE	0.0319	0.0766	0.0224	0.0455	0.0718
Avg. Peak	1.710	0.985	1.000	1.007	1.110
# of crashes	6	2	1	0	0
Timestep=5, Order=3					
Rel. IAE (Avg. IAE)	0.2218 (1.90)	0.3101 (28571.63)	0.1864 (90.96)	0.2182 (10.45)	0.1583 (0.86)

STD. IAE	0.0157	0.2333	0.0117	0.0216	0.0589
Avg. Peak	1.869	0.891	0.996	0.998	1.144
# of crashes	0	1	0	0	0

The same test was then executed with “drss”-created systems. The average time usage for each controller across the different timesteps is displayed in Figure 4-3, below. Similar to the “rss”-created systems, the “pidstd(PID)” deviates from the rest of the controllers when the sampling time is set to 0.5s, with the rest operating at comparable time usage.

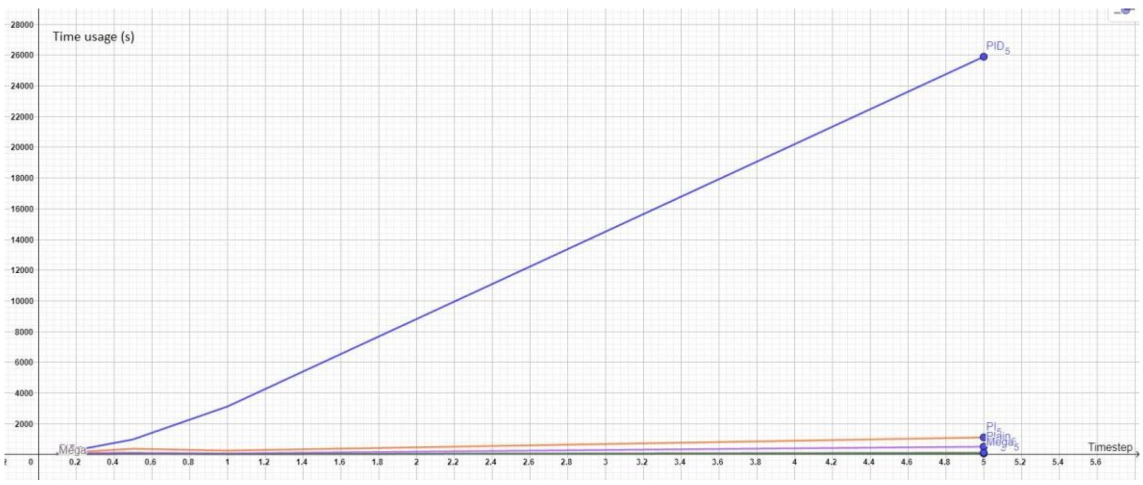


Figure 4-3: Test results of time usage (y-axis) from ZN, “pidstd(PID)”, “pidstd(PI)”, “megatunerplain”, and “megatuner” using “drss”-created systems with varying timestep (x-axis). See Figure 4-4 for addition details

Seeing as the “pidstd(PID)” has a final time usage of 26 000, a separate graph, excluding “pidstd(PID)”, is displayed in Figure 4-4. Similar to previous timestep test, the “pidstd(PI)” generally performs slower than the rest, except “pidstd(PID)”, with the “megatunerplain” being slightly slower than the comparable ZN PRC and “megatuner”.

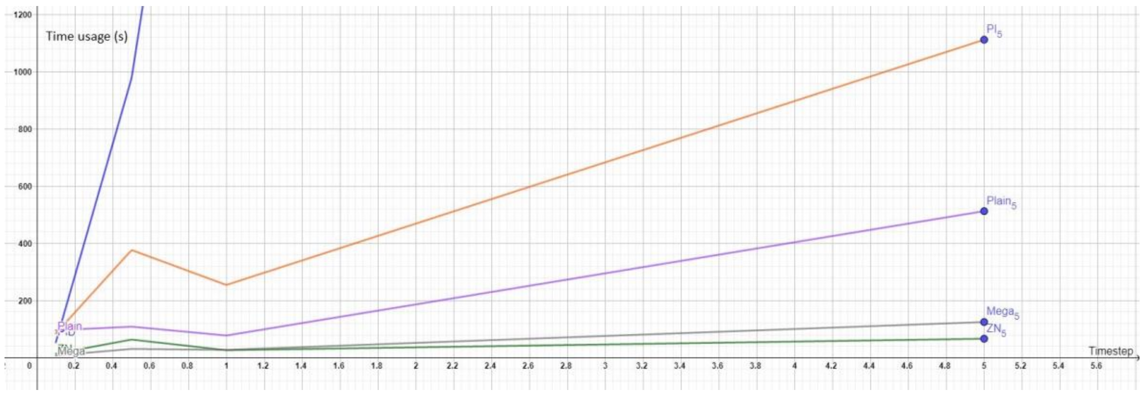


Figure 4-4: Test results of time usage (y-axis) from ZN, “pidstd(PI)”, “megatunerplain”, and “megatuner” using “drss”-created systems with varying timestep (x-axis)

Similar to the results from the “rss”-based tests, a relative IAE with its standard deviation, as well as the IAE, peak, and number of crashes are returned from the testing. This data is displayed in Table 7, segmented into each of the timestep tests. From the below table, the relative IAE varies from 0.1767, from the ZN PRC, to 0.3508, from “pidstd(PID)”, where ZN PRC generally return a much lower relative IAE while “pidstd(PID)” returns a higher relative IAE, except for when the sampling time is set to 0.1s. Compared to the “rss”-systems, the “drss”-system test return much higher crash rate for the ZN PRC and “pidstd(PID)”, while the rest remains at one or zero crashes at 50 simulations. When it comes to the average peak, ZN PRC returns a higher average peak then the rest, with a decreasing peak of the “pidstd(PID”. The three remaining controllers shows some fluctuation, although remains fairly stable around the setpoint of 1. Both the “megatunerplain” and the “pidstd(PI)” returns low standard deviations, while both ZN PRC and “pidstd(PID) “ shows a continually increase in the standard deviation.

Table 7: Test results from ZN, “pidstd(PID)”, “pidstd(PI)”, “megatunerplain”, and “megatuner” using “drss”-created systems with varying timestep

	ZN PRC	**Pidstd(PID)**	**Pidstd(PI)**	**Megatunerplain**	**Megatuner**
Timestep=0.1, Order=3					
Rel. IAE (Avg. IAE)	0.1800 (2.61)	0.1982 (10.74)	0.1981 (16.89)	0.2115 (20.31)	0.2214 (1.77)
STD. IAE	0.0358	0.0453	0.0279	0.0264	0.0600
Avg. Peak	1.118	1.033	1.026	0.998	1.064
# of crashes	11	28	0	0	0
Timestep=0.5, Order=3					
Rel. IAE (Avg. IAE)	0.1847 (11.75)	0.2746 (29.87)	0.1904 (71.75)	0.2167 (23.57)	0.2069 (6.41)
STD. IAE	0.0429	0.1687	0.0252	0.0237	0.0677
Avg. Peak	1.410	0.947	1.005	0.998	1.049
# of crashes	25	10	3	0	0
Timestep=1, Order=3					
Rel. IAE (Avg. IAE)	0.1767 (4.67)	0.2616 (816.92)	0.1870 (47.63)	0.2112 (16.44)	0.1936 (5.35)
STD. IAE	0.0554	0.1181	0.0223	0.0391	0.0802

Avg. Peak	1.592	0.973	1.012	1.006	1.098
# of crashes	11	8	1	0	0
Timestep=5, Order=3					
Rel. IAE (Avg. IAE)	0.2064 (13.74)	0.3508 (9083.51)	0.1959 (217.96)	0.2092 (107.28)	0.1905 (23.80)
STD. IAE	0.0538	0.2703	0.0315	0.0230	0.0658
Avg. Peak	1.651	0.888	1.001	1.000	1.055
# of crashes	15	13	0	0	0

4.1.2 Order

The controllers were then tested across systems with different orders. The orders tested were 1st-, 3rd-, 5th-, and 7th order systems. Due to problems during the testing with 7th order systems, further explained in Chapter 5, the 7th order systems were converted from regular transfer functions into another type of transfer function named Zero-Pole-Gain models (ZPK).

The first test was based on "rss"-created systems. The average time usage for each controller across the different orders is displayed in Figure 4-5 below. From the graph, the "pidstd(PID)" shows a higher time usage for the 1st order systems with a decrease and stabilization as the order is increased.

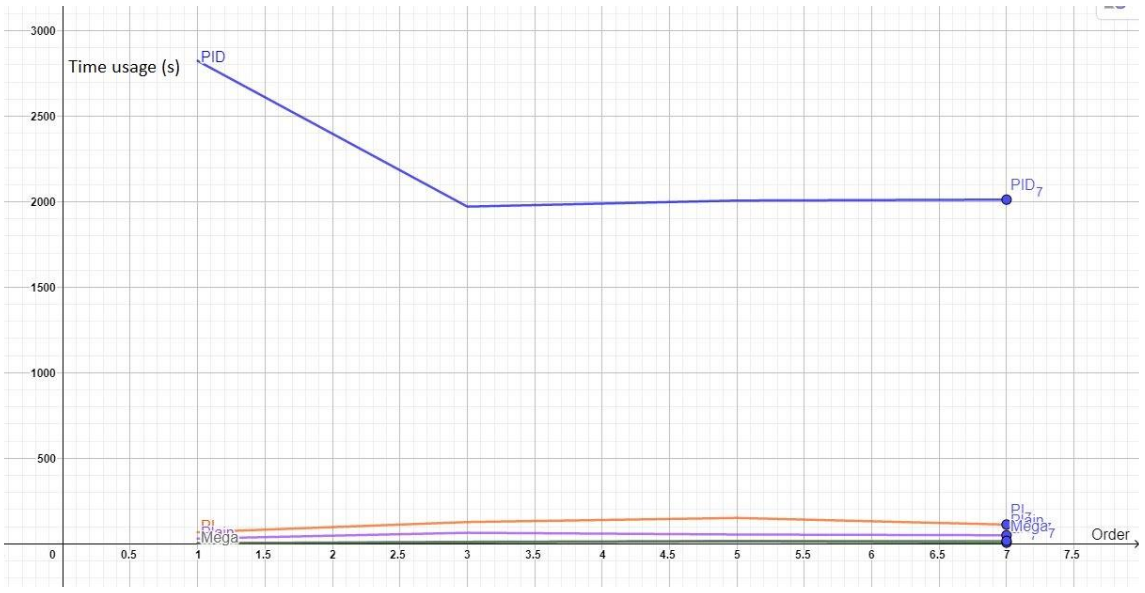


Figure 4-5: Test results of time usage (y-axis) from ZN, "pidstd(PID)", "pidstd(PI)", "megatunerplain", and "megatuner" using "rss"-created systems with varying order (x-axis). See Figure 4-6 for addition details

Seeing as the "pidstd(PID)" averages a lot higher time usage, a separate graph, excluding "pidstd(PID)", is displayed in Figure 4-6. From the graph, all controllers show an overall increase in computational time as the order is increased, with both the ZN PRC and the "pidstd(PI)" showing a great decrease when using the ZPK system.

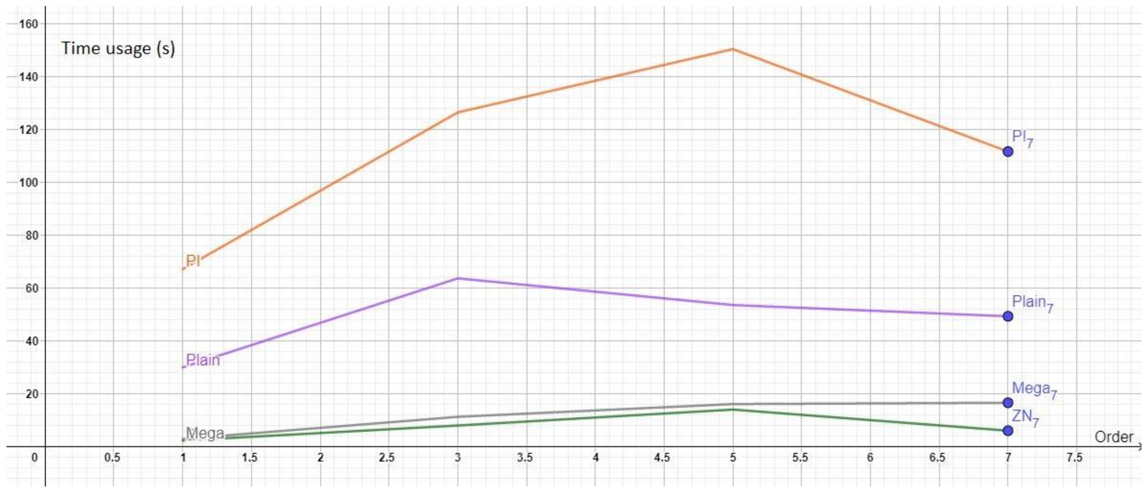


Figure 4-6: Test results of time usage (y-axis) from ZN, "pidstd(PI)", "megatunerplain", and "megatuner" using "rss"-created systems with varying order (x-axis)

Similar to the results given from the "rss"-based testing in the previous chapter, the relative IAE with its standard deviation, as well as the IAE, peak, and number of crashes are returned from the testing and displayed, below, in Table 8. The results from each test are segmented based on the order of the system used. When it comes to the relative IAE, and its standard deviation, all controllers return overall stable values, with some exceptions at the 1st order test, with the "pidstd(PID)" returning a higher relative IAE than the rest. At 1st order, no controller experienced any crashes, with an increasing number of crashes for the "pidstd(PID)" and ZN PRC, with two overall crashes for both the "pidstd(PI)" and the "megatunerplain". Each controller returns similar peaks no matter the order, with ZN PRC having the highest peaks at around 1.7, "megatuner" returning peaks of around 1.1, with the rest stabilizing at around 1, with "pidstd(PID)" being under the setpoint.

Table 8: Test results from ZN, "pidstd(PID)", "pidstd(PI)", "megatunerplain", and "megatuner" using "rss"-created systems with varying order

	ZN PRC	**Pidstd(PID)**	**Pidstd(PI)**	**Megatunerplain**	**Megatuner**
Timestep=1, Order=1					
Rel. IAE (Avg. IAE)	0.2229 (0.52)	0.2500 (706.14)	0.1842 (12.38)	0.2186 (6.56)	0.1853 (0.54)
STD. IAE	0.0076	0.0060	0.0370	0.0069	0.0743
Avg. Peak	1.760	0.996	0.998	0.997	1.1277

# of crashes	0	0	0	0	0
Timestep=1, Order=3					
Rel. IAE (Avg. IAE)	0.1974 (1.58)	0.2570 (507.51)	0.1752 (22.18)	0.2146 (13.68)	0.1885 (2.13)
STD. IAE	0.0319	0.0766	0.0224	0.0455	0.0718
Avg. Peak	1.710	0.985	1.000	1.007	1.110
# of crashes	6	2	1	0	0
Timestep=1, Order=5					
Rel. IAE (Avg. IAE)	0.2063 (2.90)	0.2419 (485.49)	0.1853 (27.90)	0.2199 (11.80)	0.1782 (2.88)
STD. IAE	0.0356	0.0116	0.0374	0.0372	0.0525
Avg. Peak	1.644	0.997	0.997	1.005	1.103
# of crashes	8	6	0	1	0
Timestep=1, Order=7, ZPK-systems					
Rel. IAE (Avg. IAE)	0.1918 (1.16)	0.2643 (631.48)	0.1725 (19.28)	0.2058 (10.16)	0.1672 (2.70)
STD. IAE	0.0586	0.1143	0.0281	0.0459	0.0770
Avg. Peak	1.688	0.9747	1.001	1.002	1.112
# of crashes	9	6	1	1	0

The controllers were then tested with “drss”-function. The average time usage for each controller across the different orders is displayed in Figure 4-7 below. Like the time usage for the “rss”-systems, the “pidstd(PID)” is slower than the rest and shows a decrease in time usage after the 1st order systems.

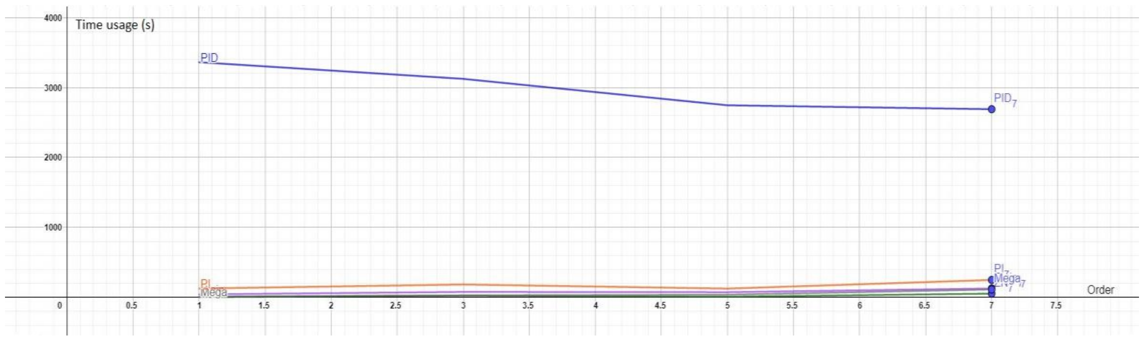


Figure 4-7: Test results of time usage (y-axis) from ZN, “pidstd(PID)”, “pidstd(PI)”, “megatunerplain”, and “megatuner” using “drss”-created systems with varying order (x-axis). See Figure 4-8 for addition details

Seeing as the “pidstd(PID)” averages a lot higher time usage, a separate graph, excluding “pidstd(PID)”, is displayed in Figure 4-8. The four controllers all show an increase at the 3rd and 7th order. The main difference between the time usage from the “drss”-systems with varying order compared to the rest of the testing is the difference between ZN PRC and “megatuner”, where the “megatuner” shows a notably slower result than the ZN PRC.

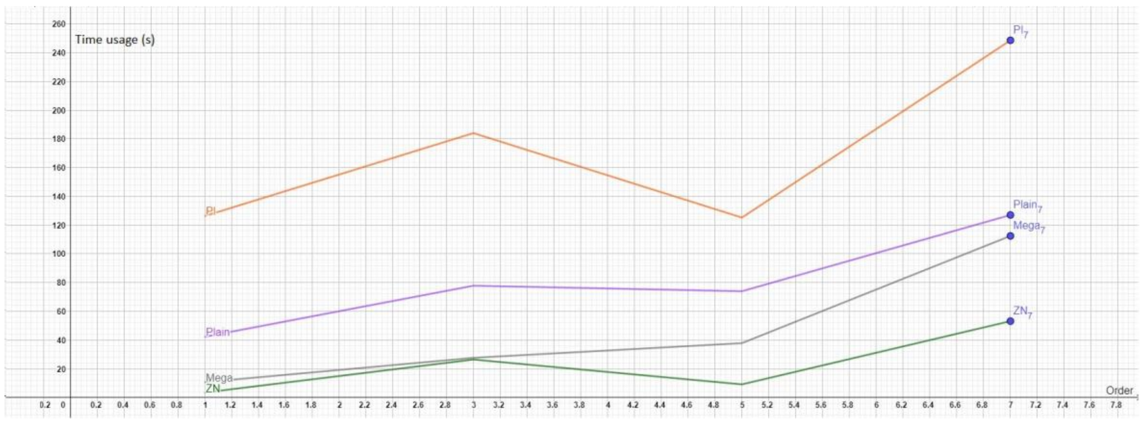


Figure 4-8: Test results of time usage (y-axis) from ZN “pidstd(PI)”, “megatunerplain”, and “megatuner” using “drss”-created systems with varying order (x-axis)

The relative IAE with its standard deviation, as well as the IAE, peak, and number of crashes are returned from the testing and displayed, below, in Table 8. The results from each test are segmented based on the order of the system used. Note that the 7th order run is still using a ZPK-model for the system. The relative IAE is stable for the “pidstd(PI)” and the “megatunerplain” with a decrease in the “megatuner” as the order increases. The ZN PRC return increasing standard deviation and relative IAE, except for 1st order systems, as the order increases, while “pidstd(PID)” displays varying relative IAE with greatly increased standard deviation as the relative IAE increases, as well as varying peaks. ZN PRC shows a general high number of crashes, while “pidstd(PID)” increases the number of crashes with the order and “pidstd(PI)” varies between 1-3 crashes.

Table 9: Test results from ZN, “pidstd(PID)”, “pidstd(PI)”, “megatunerplain”, and “megatuner” using “drss”-created systems with varying order

	ZN PRC	**Pidstd(PID)**	**Pidstd(PI)**	**Megatunerplain**	**Megatuner**
Timestep=1, Order=1					
Rel. IAE (Avg. IAE)	0.2031 (0.71)	0.2487 (835.75)	0.1879 (23.75)	0.2118 (8.96)	0.1971 (2.11)
STD. IAE	0.0323	0.0174	0.0250	0.0116	0.0575
Avg. Peak	1.770	0.999	1.006	0.996	1.066
# of crashes	18	0	2	0	0
Timestep=1, Order=3					
Rel. IAE (Avg. IAE)	0.1767 (4.67)	0.2616 (816.92)	0.1870 (34.42)	0.2112 (16.44)	0.1936 (5.35)
STD. IAE	0.0554	0.1181	0.0222	0.0391	0.0802
Avg. Peak	1.592	0.9732	1.012	1.006	1.098
# of crashes	11	8	1	0	0
Timestep=1, Order=5					
Rel. IAE (Avg. IAE)	0.1863 (1.70)	0.2366 (649.69)	0.1824 (22.85)	0.2062 (15.25)	0.1779 (6.73)
STD. IAE	0.0627	0.0100	0.0283	0.0404	0.0784
Avg. Peak	1.637	0.999	1.010	1.005	1.059
# of crashes	18	16	3	1	0
Timestep=1, Order=7, ZPK-systems					
Rel. IAE (Avg. IAE)	0.2164 (11.50)	0.2654 (713.63)	0.1932 (48.04)	0.2072 (26.33)	0.1798 (20.23)
STD. IAE	0.1647	0.1234	0.0439	0.0345	0.0728

Avg. Peak	1.618	0.988	1.014	1.016	1.062
# of crashes	15	16	2	0	0

4.2 Comparison Between Megatuner, PIDTune(PID), and PIDTune(PI)

The second part of the project's controller comparison is between the "megatuner", the "pidtune(PID)", and "pidtune(PI)" and is mainly done with an interest in seeing the efficiency of the MathWorks created "pidtune" function as well as comparing it to the "megatuner", which has been created in order to compete against the "pidtune". All three controllers were run on the same 50, randomly created, systems, first with varying timestep, then with varying order. All results are averaged from the 50 runs for that test.

4.2.1 Timestep

Similar to the previous comparison between the controllers, the effects of changes in timestep have been tested. Code runs with 50 systems was executed for each timestep with "rss"-created systems. These timesteps included 0.1s, 0.5s, 1s, and 5s in order to test the different controllers and portray differences between each controller based on the speed of the system. The average time usage for each controller across the different timesteps is displayed in Figure 4-9, below. Both versions of the "pidtune" increases its time usage with the increase in timestep, while the computational time of the "megatuner" decreases with a peak at the systems with 1s timestep.

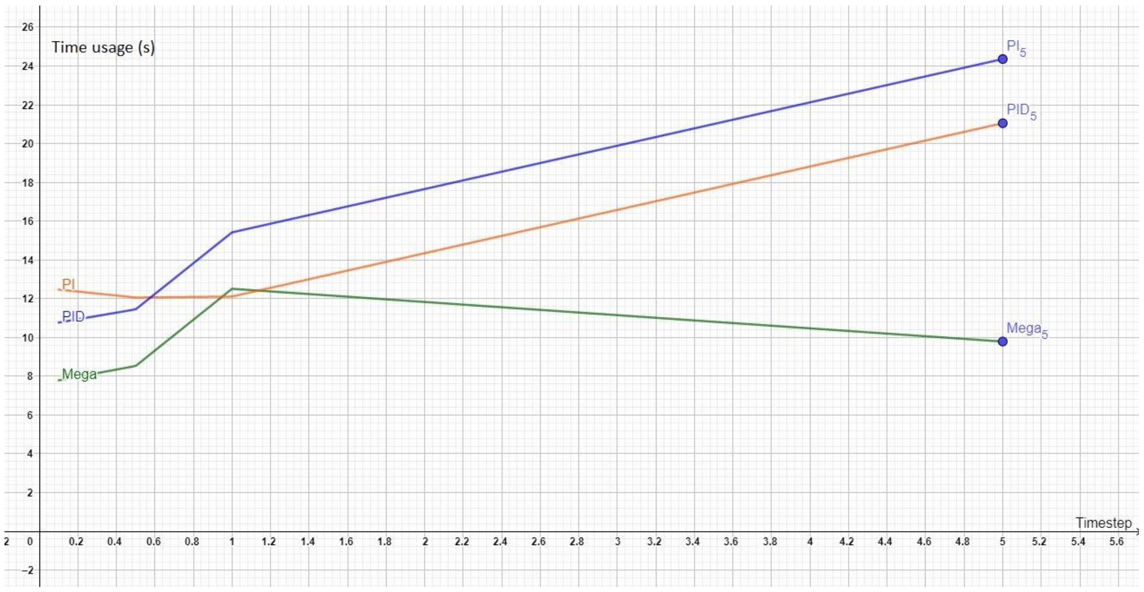


Figure 4-9: Test results of time usage (y-axis) from "megatuner", "pidtune(PID)", and "pidtune(PI)" using "rss"-created systems with varying timestep (x-axis)

The general data from the testing, with “rss”-systems, such as the aforementioned relative IAE, the average IAE and peak, as well as the number of, if any, crashes has been documented, and are presented in the table below, Table 10. The results from each test are segmented based on the timestep of the system used. When comparing the three controller against each other, all results display stability across the different timesteps, with an increase in the IAE which is proportional with the increase in time usage. Both the “megatuner” and the “pidstd(PI)” shows a decrease in the relative IAE, with a greater decrease for “megatuner” at 5s timestep. The “pidtune(PID)” displays a lower average peak than the other two, however, all three controllers return a peak above 1.

Table 10: Test results from “megatuner”, “pidtune(PID)”, and “pidtune(PI)” using “rss”-created systems with varying timestep

	Megatuner	**Pidtune(PID)**	**Pidtune(PI)**
Timestep=0.1, Order=3			
Rel. IAE (Avg. IAE)	0.1925 (1.50)	0.1952 (2.10)	0.2074 (2.59)
STD. IAE	0.0640	0.0510	0.0556
Avg. Peak	1.109	1.059	1.091
# of crashes	0	0	0
Timestep=0.5, Order=3			
Rel. IAE (Avg. IAE)	0.1832 (1.56)	0.1921 (2.20)	0.2051 (2.47)
STD. IAE	0.0623	0.0423	0.0514
Avg. Peak	1.110	1.055	1.092
# of crashes	0	0	0
Timestep=1, Order=3			
Rel. IAE (Avg. IAE)	0.1807 (2.26)	0.2011 (2.44)	0.2004 (3.09)
STD. IAE	0.0643	0.0639	0.0580
Avg. Peak	1.105	1.054	1.084
# of crashes	0	0	0
Timestep=5, Order=3			
Rel. IAE (Avg. IAE)	0.1558 (1.53)	0.2002 (4.21)	0.1919 (4.67)
STD. IAE	0.0576	0.0613	0.0536
Avg. Peak	1.132	1.061	1.105
# of crashes	0	0	0

Additional timestep-based testing was done, however, this time the “drss”-systems were used. The same timesteps were used, i.e., 0.1s, 0.5s, 1s, and 5s. The average time usage for each controller across the different timesteps is displayed in the below figure, Figure 4-10. From the figure, all three controller shows comparable time usage for the 0.1s, 0.5s, and 1s timesteps, where “megatuner” shows a lower computational time for the 5s timestep, while “pidtune(PID)” shows a lower time usage than its PI-counterpart.

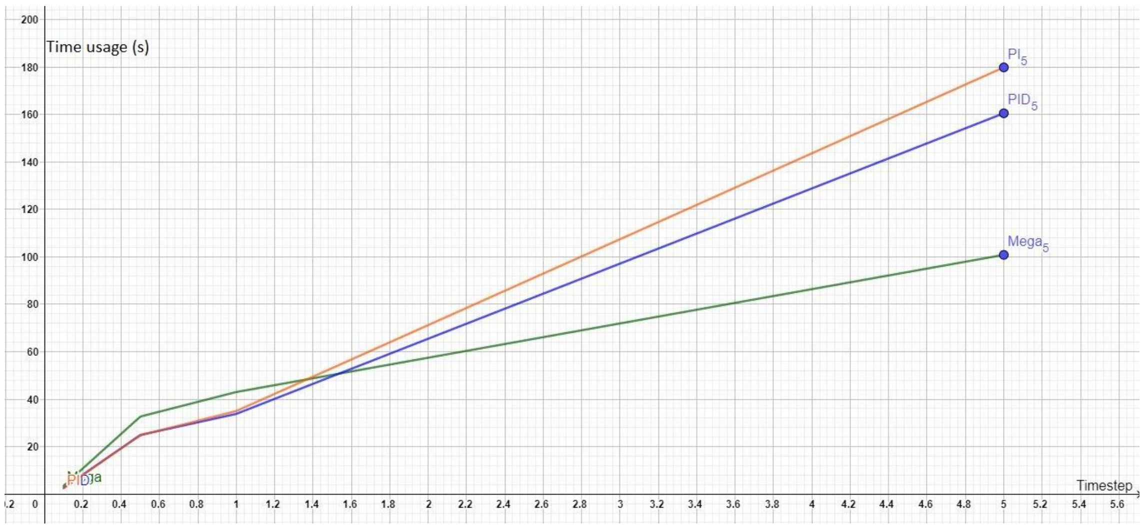


Figure 4-10: Test results of time usage (y-axis) from “megatuner”, “pidtune(PID)”, and “pidtune(PI)” using “drss”-created systems with varying timestep (x-axis)

Table 11, below, displays the various data resulting from the testing done with the controllers. The results from each test are segmented based on the timestep of the system used. Based on the data all three controller return higher relative IAE at 0.1s timestep, with a decrease in relative IAE where the lowest value is returned from the 5s systems test. “Megatuner” returns an overall lower standard deviation of the relative IAE, while both “pidtune”-controllers return similar standard deviation with the exception of the 5s system test. The average peak of all controllers shows a steady return between 1.039 and 1.085.

Table 11: Test results from “megatuner”, “pidtune(PID)”, and “pidtune(PI)” using “drss”-created systems with varying timestep

	Megatuner	**Pidtune(PID)**	**Pidtune(PI)**
Timestep=0.1, Order=3			
Rel. IAE (Avg. IAE)	0.2375 (0.83)	0.2549 (0.64)	0.2628 (0.68)
STD. IAE	0.0562	0.0620	0.0664
Avg. Peak	1.064	1.071	1.065
# of crashes	0	0	0
Timestep=0.5, Order=3			

Rel. IAE (Avg. IAE)	0.1973 (6.45)	0.2300 (5.73)	0.2267 (5.61)
STD. IAE	0.0747	0.1021	0.1035
Avg. Peak	1.061	1.039	1.055
# of crashes	0	0	0
Timestep=1, Order=3			
Rel. IAE (Avg. IAE)	0.1971 (8.48)	0.2372 (8.02)	0.2362 (8.27)
STD. IAE	0.0865	0.1111	0.1150
Avg. Peak	1.056	1.040	1.052
# of crashes	0	0	0
Timestep=5, Order=3			
Rel. IAE (Avg. IAE)	0.1763 (17.77)	0.1986 (31.87)	0.1966 (35.35)
STD. IAE	0.0689	0.0949	0.0680
Avg. Peak	1.085	1.060	1.069
# of crashes	0	0	0

4.2.2 Order

The three controllers were then tested across systems of varying order. The orders tested were 1st-, 3rd-, 5th-, 7th-, and 10th order systems. The 10th order was included seeing as both the “megatuner” and the “pidtune” have a reputation of being robust. None of the systems were replaced with a ZPK-system when using “megatuner” and “pidtune”. Firstly, the testing as done with “rss”-created systems, much like the other tests. Figure 4-11, below, shows the comparison between each controller across the different system orders. From the figure below, all controllers return comparable times, however, the “megatuner” operates slightly faster than the ”pidtune”-controller, with the “pidtune(PI)” operating slightly faster than its PID-counterpart.

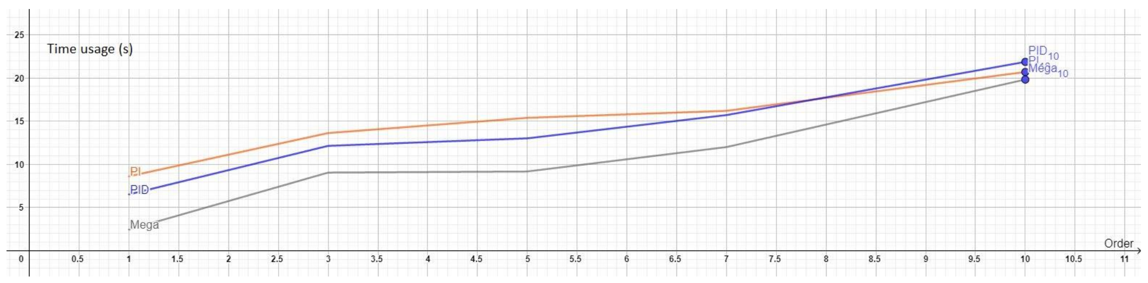


Figure 4-11: Test results of time usage (y-axis) from “megatuner”, “pidtune(PID)”, and “pidtune(PI)” using “rss”-created systems with varying order (x-axis)

From the order-based testing of the controllers, the relative IAE, the standard deviation of the relative IAE, the IAE, and the peaks of each controller are displayed in Table 12, below. The results from each test are segmented based on the order of the system used. The relative IAE from all three controllers decrease as the order of the systems increase while the standard deviation for both "pidtune"-controllers increase as the order is increased, except for the 10th order "pidtune(PID)". "Megatuner" averages a higher peak at around 1.1, with "pidtune(PI)" hovering at 1.085, and "pidtune(PID)" has the lowest overshoot at around 1.050-1.060.

Table 12: Test results from "megatuner", "pidtune(PID)", and "pidtune(PI)" using "rss"-created systems with varying order

	Megatuner	**Pidtune(PID)**	**Pidtune(PI)**
Timestep=1, Order=1			
Rel. IAE (Avg. IAE)	0.1983 (0.49)	0.2026 (1.32)	0.2114 (1.82)
STD. IAE	0.0627	0.0385	0.0399
Avg. Peak	1.143	1.064	1.108
# of crashes	0	0	0
Timestep=1, Order=3			
Rel. IAE (Avg. IAE)	0.1807 (1.60)	0.2011 (2.44)	0.2004 (3.09)
STD. IAE	0.0643	0.0639	0.0580
Avg. Peak	1.105	1.054	1.084
# of crashes	0	0	0
Timestep=1, Order=5			
Rel. IAE (Avg. IAE)	0.1803 (1.66)	0.1893 (2.46)	0.1937 (2.98)
STD. IAE	0.0916	0.0774	0.0626
Avg. Peak	1.106	1.053	1.080
# of crashes	0	0	0
Timestep=1, Order=7			
Rel. IAE (Avg. IAE)	0.1727 (2.07)	0.1973 (3.10)	0.1995 (3.23)
STD. IAE	0.0825	0.0916	0.0761
Avg. Peak	1.108	1.063	1.088
# of crashes	0	0	0
Timestep=1, Order=10			
Rel. IAE (Avg. IAE)	0.1548 (3.07)	0.1911 (4.18)	0.1867 (3.86)

STD. IAE	0.0764	0.0827	0.0773
Avg. Peak	1.102	1.055	1.085
# of crashes	0	0	0

Secondly, the three controllers were tested against “drss”-systems. The resulting average time usage for each controller across the different orders are displayed in Figure 4-12. The “megatuner” displays a lower average time usage at 1st order with a higher time usage at 3rd, 5th, and 7th order. Both “pidtune”-controllers displays a stable increase in the time usage with the order increase, with “pidtune(PI)” being slightly slower than the “pidtune(PID)”.

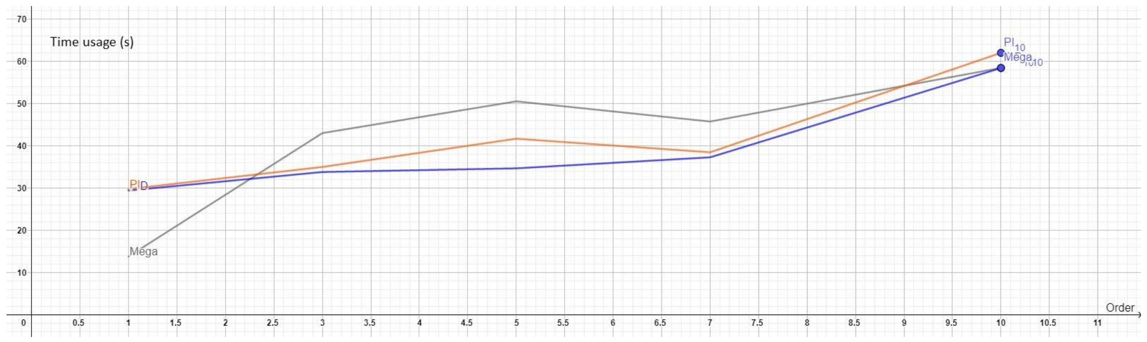


Figure 4-12: Test results of time usage (y-axis) from “megatuner”, “pidtune(PID)”, and “pidtune(PI)” using “drss”-created systems with varying order (x-axis)

In the same vein as the other tests, the resulting data is displayed in a table below, namely Table 13. The test results from each order tested is separated in segments. The relative IAE, the standard deviation of the relative IAE, the IAE, and the peak of each controller is displayed in Table 12, below. “Megatuner” displays a decreasing relative IAE as the order increases while standard deviation fluctuates between 0.0541 and 0.0950. Both “pidtune”-controllers shows a decreasing relative IAE with an increase at the 3rd order as well as comparative fluctuation in the standard deviation, ranging from as low as 0.0283 to as high as 1.1150. “Pidtune(PID)” experienced a crash at the 1st and 10th order test, while “pidtune(PI)” returned a crash at the 1st and 7th order test. All three controller display a similar peak, although “pidtune(PID)” displays a slightly lower peak during its testing.

Table 13: Test results from “megatuner”, “pidtune(PID)”, and “pidtune(PI)” using “drss”-created systems with varying order

	Megatuner	**Pidtune(PID)**	**Pidtune(PI)**
Timestep=1, Order=1			
Rel. IAE (Avg. IAE)	0.1937 (2.67)	0.1910 (5.62)	0.1948 (5.80)
STD. IAE	0.0541	0.0287	0.0283
Avg. Peak	1.063	1.037	1.067
# of crashes	0	1	1

Timestep=1, Order=3			
Rel. IAE (Avg. IAE)	0.1971 (8.48)	0.2372 (8.02)	0.2362 (8.27)
STD. IAE	0.0865	0.1111	0.1150
Avg. Peak	1.056	1.040	1.052
# of crashes	0	0	0
Timestep=1, Order=5			
Rel. IAE (Avg. IAE)	0.1879 (9.50)	0.2076 (7.20)	0.2155 (8.98)
STD. IAE	0.0950	0.0662	0.0814
Avg. Peak	1.072	1.053	1.061
# of crashes	0	0	0
Timestep=1, Order=7			
Rel. IAE (Avg. IAE)	0.1758 (8.04)	0.1936 (7.22)	0.1919 (7.38)
STD. IAE	0.0556	0.0705	0.0705
Avg. Peak	1.051	1.055	1.066
# of crashes	0	0	1
Timestep=1, Order=10			
Rel. IAE (Avg. IAE)	0.1636 (9.56)	0.1869 (10.92)	0.1812 (11.24)
STD. IAE	0.0780	0.0931	0.0850
Avg. Peak	1.099	1.077	1.092
# of crashes	0	1	0

4.3 Time Delay Comparisons

Seeing as all previous testing has been with "rss"- and "drss"-created system, all having a time delay of $e^{-\tau s}$ where τ=0.1, a separate test was done to check the effects of the time delay, and how it affected the different controllers. This testing was done on the ZN PRC, "pidstd(PID)", "pidstd(PI)", "megatunerplain" and "megatuner" with all tests consisting of 50, randomly created, systems created by the "drss"-function. The testing was done τ=0.01, τ=0.1, being the time delay used throughout the project, τ=1, and τ=0, i.e., no time delay. All results are averaged from the 50 systems for that specific test.

Similar to the testing done in the previous subchapters, the relative IAE, with its standard deviation, as well as the IAE, average time usage, peak, and number of crashes is returned from the testing and displayed, below, in Table 14. Based on the results from the data displayed below, "pidstd(PI)", "megatunerplain", and "megatuner" all show an increase in the relative

IAE as τ increases, while "pidstd(PID)" shows a decline as well as a decreasing standard deviation. "Pidstd(PI)", "megatunerplain", and "megatuner" also shows a stable, if not decreasing, peak close to the setpoint of 1, with "pidstd(PID)" shows an increase in the peak, from 0.697 to 1.034. Neither "megatunerplain" nor "megatuner" experiences any crashes, with "pidstd(PI)" returning a decreasing number of crashes. Both ZN PRC and "pidstd(PID)" return high number of crashes, reflective from the previous tests with "drss"-systems, however, ZN PRC and "pidstd(PID)" has a decreasing number of crashes as the τ increases, with an exception for "pidstd(PID)" at τ=1. ZN PRC, "megatunerplain", and "megatuner" shows an increase in time usage as τ increases, all having an exception when there is no delay, while "pidstd(PID)" show a steady decline in its time usage, also with an exception at no time delay.

Table 14: Test results from ZN, "pidstd(PID)", "pidstd(PI)", "megatunerplain", and "megatuner" using "drss"-created systems with varying τ in the time delay

	ZN PRC	**Pidstd(PID)**	**Pidstd(PI)**	**Megatunerplain**	**Megatuner**
Timestep=1, Order=3, τ=0					
Rel. IAE (Avg. IAE)	0.1236 (2.72)	0.3953 (355.39)	0.1241 (13.14)	0.1503 (8.19)	0.1213 (3.15)
STD. IAE	0.0452	0.4073	0.0267	0.0482	0.0653
Avg. Time	22.03	899.03	105.92	54.48	25.95
Avg. Peak	1.294	0.697	1.013	1.009	1.078
# of crashes	21	14	3	0	0
Timestep=1, Order=3, τ=0.01					
Rel. IAE (Avg. IAE)	0.1981 (0.21)	0.3281 (1292.19)	0.1885 (21.86)	0.2083 (8.17)	0.1843 (3.14)
STD. IAE	0.0441	0.2611	0.0285	0.0305	0.0825
Avg. Time	1.08	3938.39	115.99	39.20	17.05
Avg. Peak	1.555	0.869	1.003	0.998	1.044
# of crashes	22	11	2	0	0
Timestep=1, Order=3, τ=0.1 (used in all other tests)					
Rel. IAE (Avg. IAE)	0.1767 (4.67)	0.2616 (816.92)	0.1870 (47.63)	0.2112 (16.44)	0.1936 (5.35)
STD. IAE	0.0554	0.1181	0.0223	0.0391	0.0802

Avg. Time	26.43	3122.79	254.69	77.84	27.63
Avg. Peak	1.592	0.973	1.012	1.006	1.098
# of crashes	11	8	1	0	0
Timestep=1, Order=3, τ=1					
Rel. IAE (Avg. IAE)	0.1934 (16.76)	0.2309 (97.64)	0.1984 (13.64)	0.2107 (19.19)	0.2300 (6.56)
STD. IAE	0.0452	0.0623	0.0203	0.0253	0.0451
Avg. Time	86.65	422.87	68.77	91.10	28.50
Avg. Peak	1.112	1.034	1.002	1.006	1.060
# of crashes	5	24	1	0	0

5 Discussion

As a discussion on the project, Chapter 5 is centered around the various subjects presented in Chapter 2, as well as its implementation in Chapter 3, and the resulting data depicted in Chapter 4. Any changes done in this project, and the reasoning for decisions made, will be presented in this chapter.

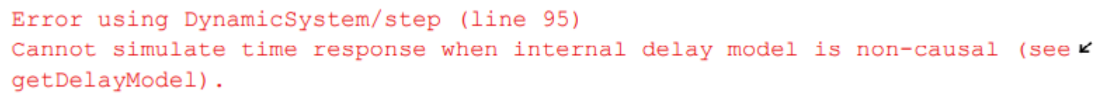
From the data presented in subchapter 4.1, various conclusions may be made in terms of the advantages and disadvantages of the different controllers. While the ZN PRC, “pidstd(PI)”, “megatunerplain”, and “megatuner” all had relatively low and comparative time usage, the ZN PRC and “megatuner” appears as the clear quicker methods, however, the ZN PRC method returns high peaks, reaching 1.869 for slower systems, which would be a problem for systems where overshoot is expensive or in general non-ideal. The ZN PRC method had a fairly high number of crashes, especially when using the “drss”-function, reaching as high as 25 crashes out of 50 runs. The “pidstd(PID)” balanced between a high number of crashes, with a total of 28 at 0.1s for 3rd order systems, and a high time usage at slower systems averaging 92 000s and 26 000s from the two 5s system tests ran. The “pidstd(PI)” returned fairly stable values, both in terms of its efficiency, such as the relative IAE and peaks, and the number of crashes it experienced across the different tests. Both the “megatunerplain” and the “megatuner” shows great response to the random system, with the “megatunerplain” being slightly slower than the “megatuner” but also displays a lower peak. “Megatunerplain” experienced a total of three crashes during the several hundred runs while the “megatuner” experienced none.

From the data presented in Chapter 4.2, all three controllers show comparable responses to the random systems. In terms of crashes, both “pidtune”-controllers returned two crashes using the “drss”-systems. The “megatuner” averages overall lower time usage, with some deviation when testing higher order “drss”-systems. In resemblance to the “pidstd(PID)”, the “pidtune(PID)” averaged lower peaks than the both “megatuner” and “pidtune(PI)”, however, in contrast to the other PID controller, “pidtune(PID)” averaged peaks above the setpoint of 1 and did not show significant increase in time usage at slow system, being even faster than its PI-counterpart overall.

From the data presented in Chapter 4.3, the difference in time delay made significant changes in the response from each controller. Due to using “drss”-systems, the controllers displayed more crashes than the “rss”-versions, however, except from ZN PRC at τ=1, PRZ ZN, “pidstd(PID)”, and “pidstd(PI)” all showed increased number of crashes when using a different τ than the 0.1 used during other testing. In addition, each controller appeared to display significantly different results when there was no time delay compared to the differences between the varying time delay. While “megatunerplain” showed an increase in time usage, both “megatuner” and “megatunerplain” displayed rather stable results across the different time delay, including the test without delay. Due to the sporadic changes in time usage between the controllers at different tests, a graph was not created, and the time usage was included in Table 14 instead.

From the data presented, it was also noticed that the average peak of the “pidstd(PID)”-controller became lower and lower as the timestep increased, resulting in a peak as low as 0.888. This would mean the controller is not able to stabilize at the reference step response setpoint which is 1. Why this problem occurs may be due to the dampening effect, D-term, seeing as this problem does not occur with the “pidstd(PI)”-controller, and could possibly be explained by having a dampening effect stronger than the error integral, I-term, thereby inhibiting the process in reaching its setpoint.

Due to a recurring problem with the internal delay model appearing to be non-causal, which occurred during the 7th order testing on ZN PRC, “pidstd”, “megatunerplain”, and “megatuner”, the transfer function was replaced with a ZPK-version of the model. The idea for this change came from a discussion on the error on one of MathWorks’ help forums and returned data comparable to the standard transfer function testing previously done, although appearing slightly favorable. In addition, while it was not documented, the problem appeared to occur mainly for the “pidstd”-controller, which is why the ZPK-conversion was not done for the “megatuner” and “pidtune” comparison, and the order was even increased to include 10th order systems. The error printout is displayed in Figure 5-1, below. The comparable data between the used transfer function system and a ZPK model is displayed in Appendix D.



```
Error using DynamicSystem/step (line 95)
Cannot simulate time response when internal delay model is non-causal (see ↙
getDelayModel).
```

Figure 5-1: Error description based on the exact time delay

It was also discussed if the exact time-delay, $e^{-\tau s}$, should be replaced by an approximation, e.g., Pade- or Skogestad approximation presented in equations 5-1 and 5-2, respectively. [4] However, from the testing done when comparing both approximations to the exact delay, neither approximations returned data close to the results from using the exact delay. The comparable data between the exact time delay and the two approximations are attached in Appendix D.

$$e^{-\tau s} \approx \frac{1 - 0.5\tau s}{1 + 0.5\tau s} \tag{5-1}$$

$$e^{-\tau s} \approx 1 - \tau s \tag{5-2}$$

It is clear from the testing that the usage of “rss”- and “drss”-systems created far different results, with the “drss”-function returning more crashes and overall higher time usage, with the exception of “pidstd(PID)” as 5s timestep. Based on the information displayed in chapters 3.1.1 and 3.1.2, the “drss”-function created systems may have negative eigenvalues in the A-matrix resulting in a higher system order and an average of the resulting order of the “drss”-based test has been calculated in order to give a better understanding of the different results. The average order from each test, using “drss”, is depicted in Table 15.

Table 15: Calculated average order using the "drss"-function, with percentage increse in parenthesis. "1st Comp." refers to ZN PRC, "pidstd", "megatunerplain, and "megatuner", while "2nd Comp." refers to the "megatuner" and "pidtune" comparision

	Order				
	1st	**3rd**	**5th**	**7th**	**10th**

1st Comp. Avg. Order	1.5 (50%)	3.8 (26.7%)	6.6 (32%)	9.12 (30.3%)	NA
2nd Comp. Avg. Order	1.48 (48%)	4.02 (34%)	6.68 (33.6%)	8.92 (27.4%)	12.68 (26.8%)

From the above table, the average increase in order starts at 48-50% and decreases for each order, with an exception for the 3rd order for the 1st comparison, which could be explained in a Gaussian distribution of the possible order increase, where maximum increase is doubling the selected order thereby having a single order increase affect the 1st order percentage more than higher order system, e.g., 7th order.

However, despite the order increase, when comparing the higher order "rss"-results with the comparable lower order "drss"-results, the "drss"-systems seems to still affect the systems harder in terms of time usage and crashes, including the two crashes for both "pidtune"-controllers, implying that despite the increased order to remove the negative eigenvalues in the A-matrix, the "drss"-function creates overall more challenging systems.

All results are gathered from the several tests documented throughout Chapter 4 and an average of the collected data is used as the comparative factor. Seeing as the data from the testing was prone to outliers, the median values from the testing would have been the optimal solution. The data presented in this report has been cleared of outlier; however, this was done by manually checking all raw data and look for extreme values among the datasets.

During the testing of "megatuner", "pidtune(PID)", and "pidtune(PI)", both "pidtune"-controllers experienced two crashes each when using varying order of "drss"-systems, with the 1st order "drss"-system causing a crash for both. The transfer function causing this crash is presented in equation 5-3. The reasoning, or theories, for why the "pidtune" crashed on this system is not discussed in this report. From the previously mentioned master thesis by Preben Sandva Solvang, Chapter 4.1.4 describes to most common causes for "pidtune" instability, with the main reason being double integrated systems. [1]

$$hp = e^{-0.1s}\frac{-0.001809s - 15.74}{s^2 + 0.002268s + 9.87} \tag{5-3}$$

Finally, in terms of further work, or a general enhancement of the project, all testing for a specific parameter and system, e.g., "rss"-systems with varying timestep, should be executed in the same script in order to simply the data display be being able to create graphs in MATLAB directly rather than using the raw data and creating graphs separately. In addition, the error-checking should be expanded upon to include possible non-causal delay error, which stopped the script execution in this project, crashes/instability, possible timeouts if controllers take too long, etc., as well as separating each error for documentation purposes.

6 Conclusion

This project is a master thesis as a part of the Master of Science in Industrial IT and Automation with the name “Tuning PID Controllers: From process experiments, general linear state space models and tuning PID controllers via Process Reaction Curve Methods”. The goal of this project was to compare different controllers against each other based on randomly created systems. From the various testing done several observations has been made. The overall performances of the ZN PRC and the “pidstd(PID)” are unstable, slow, or experiences a lot of crashing. The ZN PRC results in high overshoots while the “pidstd(PID)” appears to struggle with reaching the setpoint, both being a problem in process control. The “pidstd(PI)” is overall an acceptable PI controller with low numbers of crashes and general low time usage, however, seeing as the function requires a separate tuning method, the performance may be heavily affected by the tuning as well.

The “megatuner” displays great performance across all tests with generally the lowest time usage, slight overshoot, and no crashes across the couple thousand tests run in total including the tests displayed in this report, which may prove its origin of name correct. The “megatunerplain”, being a simplified version of the “megatuner”, displays similar, although slightly worse, results than the “megatuner”, being slower and experiencing three overall crashes during testing. This difference was expected seeing as the “megatunerplain” forgoes some of stability testing done in the “megatuner”.

MathWorks’ “pidtune” delivers similar, if not slightly worse, performance as the “megatuner” seeing as the “pidtune” experiences two crashes from both the PI- and PID controller. All over, the “pidtune” returns low time usage, low peaks, and overall stability between systems, which is reflected in the generally low standard deviation of the relative IAE. The source code for the “pidtune” is secret, however, based on the results from the “pidtune” and “megatuner”, the process of using PRC and frequency response may be similar between the methods.

When having to use a controller for a process, both the “megatuner” and “pidtune” are great methods. Depending on the system, both methods appear to be just as viable, with “megatuner” returning slightly higher peaks but in return appears more robust. The main problem with each controller is the secrecy around the “pidtune” source code, limiting the understanding and control of the method, and the lack of public access to the “megatuner” function seeing as it has been created by Christer Dalen and there are yet to come any official report on the “megatuner”. With “megatuner” only needing the system as a parameter and “pidtune” only needing the system and a controller-specification, i.e., ‘PI’, ‘PID’, etc., both methods are easy to implement and use as tuning methods.

References

[1] P. S. Solvang, "State Space Model Based PID Controller Tuning," University of South-Eastern Norway, Porsgrunn, 2019.

[2] C. Dalen and D. D. Ruscio, "A Semi-Heuristic Process-Reaction Curve PID Controller Tuning Method," *Modeling, Identification and Control,* pp. 37-43, 2018.

[3] MathWorks Inc, "Mathworks Help Center, pidtune," 2022. [Online]. Available: https://se.mathworks.com/help/control/ref/lti.pidtune.html. [Accessed 12 02 2022].

[4] D. D. Ruscio, System Theory State Space Analysis and Control Theory, Porsgrunn: Telemark University College, 2017.

[5] J. G. Ziegler and N. B. Nichols, "Optimum Settings for Automation Controllers," ASME, Rochester, N.Y., 1942.

[6] Instrumentation Tools, "Ziegler-Nichols Open-Loop Method," Instrumentation Tools, 2022. [Online]. Available: https://instrumentationtools.com/ziegler-nichols-open-loop-method/. [Accessed 02 04 2022].

[7] D. D. Ruscio, "On Tuning PI Controlles for Integrating Plus Time Delay Systems," *Modeling, Identification and Control,* pp. 145-164, 2010.

[8] D. D. Ruscio and C. Dalen, "Tuning PD and PID Controllers for Double Integrated Plus Time Delay Systems," *Modeling, Identification and Control,* pp. 95-110, 2017.

[9] C. Dalen and D. D. Ruscio, "A Novel Process-Reaction Curve Method for Tuning PID Controllers," *Modeling, Identification and Control,* pp. 273-291, 2018.

[10] C. Dalen and D. D. Ruscio, "Model-Free PI/PID Controller Tuning of Higher Order Nonlinear Dynamic Systems," *Modeling, Identification and Control,* pp. 199-211, 2019.

[11] B. Friedland, Control System Design: An Introduction to State-Space Methods, Mineola: McGraw-Hill, 1986.

[12] MathWorks Inc, "MathWorks Help Center, Continuous-Discrete Conversion Methods," MathWorks, 2022. [Online]. Available: https://se.mathworks.com/help/control/ug/continuous-discrete-conversion-methods.html. [Accessed 22 03 2022].

[13] Mathworks Inc, "MathWorks Help Center, pidstd," 2022. [Online]. Available: https://se.mathworks.com/help/control/ref/pidstd.html. [Accessed 12 02 2022].

[14] MathWorks Inc., "MathWorks Help Center, pidTuner," MathWorks, 2022. [Online]. Available: https://se.mathworks.com/help/control/ref/pidtuner.html. [Accessed 09 02 2022].

[15] MathWorks Inc, "Google Patents," Google, 18 06 2013. [Online]. Available: https://patents.google.com/patent/US8467888B2/en. [Accessed 02 03 2022].

Appendices

Appendix A

Task Description

FMH606 Master's Thesis

Title: Tuning PID Controllers: From process experiments, general linear state space models and tuning PID controllers via Process Reaction Curve Methods

USN supervisor: David Di Ruscio

External partner: None

Task background: The Ziegler-Nichols (ZN) methods for tuning PID controllers involves a Process Reaction Curve (PRC) method based on a step response from the plant. Such PRC methods may give simple and useful methods for tuning the PID controller, both experimentally and numerically. General due to noisy data, detailed linear state space models are usually identified from more informative input experiments. An idea is to tune the PID controllers directly from more general n-th order linear models. One such method is the patented MATLAB pidtune algorithm and other recently published papers.

Task description:

1. Give a literature survey of methods for tuning PID controllers, both tuning from general linear models as well as the Ziegler Nichols PRC method and related.
2. Recently similar PRC methods are published. Investigate these algorithms and possible differences from and between the ZN methods.
3. Compare the different methods by simulation experiments using MATLAB or similar (Octave/Python). Random SSMs generated from the MATLAB rss.m or drss.m functions may be used in order to generate models for tuning the PI/PID controllers.
4. Give a detailed survey of the subject of tuning PID controllers from measured process experiments. This task may be combined with the introductory task above.

Student category: IIA students

The task is suitable for online students (not present at the campus): Yes

Practical arrangements: Individual guided work.

Supervision:

As a general rule, the student is entitled to 15-20 hours of supervision. This includes necessary time for the supervisor to prepare for supervision meetings (reading material to be discussed, etc).

<u>Signatures</u>:

Supervisor (date and signature):

Student (write clearly in all capitalized letters):

Student (date and signature):

Appendix B

MATLAB Code, Demo_V3

Demo_V3-script code

```
% demo tests for pi(d) tuning methods
% random generated stable state space models.
% Integrator processes will be handled by delta_prc_pi_tun
clear all
close all
clf, clc

n=3; % order
tau=0.1; % time delay
dt= 1; % sample time
s=tf('s');
dsys = drss(n,1,1) % random discrete stable SSM
dsys.Ts = dt; dsys.D = 0;

% converting from discrete to continuous and adding (possible) time delay
csys = d2c(dsys,'zoh');
[b1,a1]=ss2tf(csys.a,csys.b,csys.c,0);
hp=tf(b1,a1)*exp(-tau*s);

while(any(abs(eig(hp))<0.0001)) % i.e. no integrators in this demo
      dsys = drss(n,1,1);
       dsys.Ts=dt; dsys.D=0;
       csys = d2c(dsys,'zoh');
       [b1,a1]=ss2tf(csys.a,csys.b,csys.c,0);
       hp=tf(b1,a1)*exp(-tau*s);
end

m = 5;
n = 3;
[Y,T]=step(hp);

%Ziegler Nichols
[hc,Kp,Ti]=zn_pi(hp);
[Yr,t]=step(hp*hc/(hc*hp+1));
[Yv,t2]=step(hp/(hc*hp+1));
ZN_error = sum(abs(Yr-1));
ZN_total_error = ZN_error/length(Yr)*t(end)
ZN_relative_error = ZN_error/length(Yr)
```

```
plotter(T, Y, t, Yr, t2, Yv, m, n, 1)

%PID-tuner
zeta=1;
delta=2.3;
[Kp,Ti,Td]=delta_prc_pid_tun1....
(T,Y,dt,delta,zeta);hc=pidstd(Kp,Ti,Td);rho=delta;
[Yr,t]=step(hp*hc/(hc*hp+1));
[Yv,t2]=step(hp/(hc*hp+1));
PID_error = sum(abs(Yr-1));
PID_total_error = PID_error/length(Yr)*t(end)
PID_relative_error = PID_error/length(Yr)

plotter(T, Y, t, Yr, t2, Yv, m, n, 4)

%PI-tuner
zeta=1;
delta=2.3;
[Kp,Ti]=delta_prc_pi_tun...
(T,Y,dt,delta,zeta);hc=pidstd(Kp,Ti);rho=delta;
[Yr,t]=step(hp*hc/(hc*hp+1));
[Yv,t2]=step(hp/(hc*hp+1));
PI_rel_IAE = sum(abs(Yr))/sum(length(Yr));
PI_error = sum(abs(Yr-1));
PI_total_error = PI_error/length(Yr)*t(end)
PI_relative_error = PI_error/length(Yr)

plotter(T, Y, t, Yr, t2, Yv, m, n, 7)

%Plain Megatuner
rho=2; kp_alt=1;
hc=megatuner1plain(hp,rho,kp_alt);
[Yr,t]=step(hp*hc/(hc*hp+1));
[Yv,t2]=step(hp/(hc*hp+1));
Plain_rel_IAE = sum(abs(Yr))/sum(1*length(Yr))
Plain_error = sum(abs(Yr-1));
Plain_total_error = Plain_error/length(Yr)*t(end)
Plain_relative_error = Plain_error/length(Yr)
```

```
plotter(T, Y, t, Yr, t2, Yv, m, n, 10)

%Megatuner
[hc,rho]=megatuner1(hp); % heuristic ...
[Yr,t]=step(hp*hc/(hc*hp+1));
[Yv,t2]=step(hp/(hc*hp+1));
Mega_rel_IAE = sum(abs(Yr))/sum(length(Yr))
Mega_error =  sum(abs(Yr-1));
Mega_total_error = Mega_error/length(Yr)*t(end)
Mega_relative_error = Mega_error/length(Yr)

plotter(T, Y, t, Yr, t2, Yv, m, n, 13)
```

Plotter-function code

```
function plotter(T, Y, t, Yr, t2, Yv, m, n, i)

subplot(m, n, i);
plot(T,Y)
xlabel('Time [s]'),ylabel('Output, y')
grid
title('Open loop step response')

subplot(m, n, i+1);
plot(t,Yr,'-r')
xlabel('Time [s]'),ylabel('Output, y')
grid
title('Reference step response')

subplot(m, n, i+2);
plot(t2,Yv,'-r')
grid
title('Input disturbance step response')
xlabel('Time [s]'),ylabel('Output, y')

end
```

Appendix C

MATLAB Code, Result_Script

Result_Script-script code

```
% main script for results from “drss” systems
% delta_prc_pi_tun/delta_prc_pid_tun1/MEGATUNER1/MEGATUNER1PLAIN
% on random generated stable state space models.
% Integrator processes will be handled by delta_prc_pi_tun
%
warning('off') % unnecessary warnings of order increase
clear all
close all
clf, clc

n = 3; % order
dt = 1 ;   % sample time
m = 5;
i_max = 50;

disp(['Order = ', num2str(n)]);
disp(['Timestep = ', num2str(dt)]);
disp(['Runs = ', num2str(i_max)]);

for i = 1:i_max
    disp(['i = ', num2str(i)]);
    hp = final_create_random_hp(n, dt)
    [Y,T]=step(hp);
    if n>5
        hp = zpk(hp);
    end
    [rel_IAE(i, :), IAE(i, :), time(i, :), peak(i, :)] = controller_executions(hp,
dt, Y, T);
    pause(0.5);
end

rel_IAE
num2str(time)

ctrl = ['ZN   '; 'PID  '; 'PI   '; 'Plain'; 'Mega '];
for i = 1:5
    controller_result_display(rel_IAE, IAE, time, peak, i, ctrl(i, :));
end
```

Final_Create_Random_Hp-function code

```
function hp = final_create_random_hp(n, dt)

    tau=0.1; % time delay
    s=tf('s');

    dsys = drss(n,1,1); % random discrete stable SSM
    dsys.Ts = dt; dsys.D = 0;

    % converting from discrete to continuous and adding (possible) time delay
    csys = d2c(dsys,'zoh');
    [b1,a1]=ss2tf(csys.a,csys.b,csys.c,0);
    hp=tf(b1,a1)*exp(-tau*s); % converting into a transfer function with delay
    while(any(abs(eig(hp))<0.0001))
          dsys = drss(n,1,1);
            dsys.Ts=dt; dsys.D=0;
            csys = d2c(dsys,'zoh');
            [b1,a1]=ss2tf(csys.a,csys.b,csys.c,0);
            hp=tf(b1,a1)*exp(-tau*s);
    end
end
```

Controller_Executions-function code

```
function [rel_IAE, IAE, time, peak] = controller_executions(hp, dt, Y, T)

    %Ziegler Nichols
    [hc,Kp,Ti]=zn_pi(hp);
    [Yr,t]=step(hp*hc/(hc*hp+1));
    ZN_time = t(end);
    ZN_peak = max(Yr, [], 'all');
    ZN_error = sum(abs(Yr-1));
    ZN_rel_IAE = ZN_error/length(Yr);
    if ZN_rel_IAE > 1 || isnan(ZN_rel_IAE)
        ZN_rel_IAE = -1;
    end
    ZN_IAE = ZN_rel_IAE*ZN_time;
```

```
%PID-tuner
zeta=1;
delta=2.3;
[Kp,Ti,Td]=delta_prc_pid_tun1....
(T,Y,dt,delta,zeta);hc=pidstd(Kp,Ti,Td);rho=delta;
[Yr,t]=step(hp*hc/(hc*hp+1));
PID_peak = max(Yr, [], 'all');
PID_time = t(end);
PID_error = sum(abs(Yr-1));
PID_rel_IAE = PID_error/length(Yr);
if PID_rel_IAE > 1 || isnan(PID_rel_IAE)
    PID_rel_IAE = -1;
end
PID_IAE = PID_rel_IAE*PID_time;


%PI-tuner
zeta=1;
delta=2.3;
[Kp,Ti]=delta_prc_pi_tun(T,Y,dt,delta,zeta);
hc=pidstd(Kp,Ti);
rho=delta;
[Yr,t]=step(hp*hc/(hc*hp+1));
PI_peak = max(Yr, [], 'all');
PI_time = t(end);
PI_error = sum(abs(Yr-1));
PI_rel_IAE = PI_error/length(Yr);
if PI_rel_IAE > 1 || isnan(PI_rel_IAE)
    PI_rel_IAE = -1;
end
PI_IAE = PI_rel_IAE*PI_time;

%Plain Megatuner
rho=2; kp_alt=1;
hc=megatuner1plain(hp,rho,kp_alt);
[Yr,t]=step(hp*hc/(hc*hp+1));
Plain_peak = max(Yr, [], 'all');
```

```
    Plain_time = t(end);
    Plain_error = sum(abs(Yr-1));
    Plain_rel_IAE = Plain_error/length(Yr);
    if Plain_rel_IAE > 1 || isnan(Plain_rel_IAE)
        Plain_rel_IAE = -1;
    end
    Plain_IAE = Plain_rel_IAE*Plain_time;

    %Megatuner
    [hc,rho]=megatuner1(hp); % heuristic ...
    [Yr,t]=step(hp*hc/(hc*hp+1));
    Mega_peak = max(Yr, [], 'all');
    Mega_time = t(end);
    Mega_error = sum(abs(Yr-1));
    Mega_rel_IAE = Mega_error/length(Yr);
    if Mega_rel_IAE > 1 || isnan(Mega_rel_IAE)
        Mega_rel_IAE = -1;
    end
    Mega_IAE = Mega_rel_IAE*Mega_time;

    rel_IAE = [ZN_rel_IAE, PID_rel_IAE, PI_rel_IAE, Plain_rel_IAE, Mega_rel_IAE];
    IAE = [ZN_IAE, PID_IAE, PI_IAE, Plain_IAE, Mega_IAE];
    time = [ZN_time, PID_time, PI_time, Plain_time, Mega_time];
    peak = [ZN_peak, PID_peak, PI_peak, Plain_peak, Mega_peak];

end
```

Mega_Pidtune_Controller_Executions-function code
(Alternate function)

```
function [rel_IAE, IAE, time, peak] = mega_pidtune_controller_executions(hp)

    %Megatuner
    [hc,rho]=megatuner1(hp); % heuristic ...
    [Yr,t]=step(hp*hc/(hc*hp+1));
    Mega_peak = max(Yr, [], 'all');
    Mega_time = t(end);
    Mega_error = sum(abs(Yr-1));
    Mega_rel_IAE = Mega_error/(length(Yr));
    if Mega_rel_IAE > 1 || isnan(Mega_rel_IAE)
        Mega_rel_IAE = -1;
    end
    Mega_IAE = Mega_rel_IAE*Mega_time;

    %PID
    hc=pidtune(hp, 'PID');
    [Yr,t]=step(hp*hc/(hc*hp+1));
    PIDtune_peak = max(Yr, [], 'all');
    PIDtune_time = t(end);
    PIDTune_error = sum(abs(Yr-1));
    PIDtune_rel_IAE = PIDTune_error/(length(Yr));
    if PIDtune_rel_IAE > 1
        PIDtune_rel_IAE = -1;
    end
    PIDtune_IAE = PIDtune_rel_IAE*PIDtune_time;

    %PI
    hc=pidtune(hp, 'PI');
    [Yr,t]=step(hp*hc/(hc*hp+1));
    PItune_peak = max(Yr, [], 'all');
    PItune_time = t(end);
    PITune_error = sum(abs(Yr-1));
    PItune_rel_IAE = PITune_error/(length(Yr));
    if PItune_rel_IAE > 1
        PItune_rel_IAE = -1;
    end
    PItune_IAE = PItune_rel_IAE*PItune_time;

    rel_IAE = [Mega_rel_IAE, PIDtune_rel_IAE, PItune_rel_IAE];
    IAE = [Mega_IAE, PIDtune_IAE, PItune_IAE];
    time = [Mega_time, PIDtune_time, PItune_time];
    peak = [Mega_peak, PIDtune_peak, PItune_peak];

end
```

Controller_Result_Display-function code

```matlab
function controller_result_display(tot_rel_IAE, tot_IAE, time, peak, index, ctrl)

%Finding the crashes in the matrix
crash_idx = tot_rel_IAE(:, index) < 0;

%Calculating the average relative IAE
non_crash_rel_IAE = tot_rel_IAE(~crash_idx, index);
avg_rel_IAE = sum(non_crash_rel_IAE)/length(non_crash_rel_IAE);

%Calculating the average IAE
non_crash_IAE = tot_IAE(~crash_idx, index);
avg_IAE = sum(non_crash_IAE)/length(non_crash_IAE);

%Calculating average time
non_crash_time = time(~crash_idx, index);
avg_time = sum(non_crash_time)/length(non_crash_time);

%Calculating average peak
non_crash_peak = peak(~crash_idx, index);
avg_peak = sum(non_crash_peak)/length(non_crash_peak);

%Calculating STD of relative IAE
std_rel_IAE = std(non_crash_rel_IAE);
%Calculating STD of IAE
std_IAE = std(non_crash_IAE);
%Calculating STD of Time
std_time = std(non_crash_time);

%Printing the results
disp([ctrl, ' average IAE = ', num2str(avg_rel_IAE)]);
disp([ctrl, ' IAE = ', num2str(avg_IAE)]);
disp(['Time average = ', num2str(avg_time)]);
disp(['Peak average = ', num2str(avg_peak)]);
disp(['STD of relative IAE = ', num2str(std_rel_IAE)]);
if sum(crash_idx) > 0
    disp(['# of crashes for ', ctrl, ': ', num2str(sum(crash_idx))]);
end
disp(' ');
```

Appendix D

Additional Data for Time Delay and ZPK Model Comparison

Comparison Between Exact- and Approximate Time Delays

The below table, Table 16, shows the resulting data from testing the exact time delay, $e^{-\tau s}$, with the two approximations, Pade and Skogestad, on the "megatuner" controller. The goal was to replace the exact time delay with a solution which resembled its results, and from the below table, it may be seen that the difference between the exact time delay differs significantly from the exact time delay. However, the result from the approximations resembles each other and would be suitable for replacing the other approximation.

Table 16: Test results from "megatuner" using "drss"-created systems with exact time delay, and Pade- and Skogestad approximation of the time delay

	Exact	**Pade**	**Skogestad**
Rel. IAE (Avg. IAE)	0.1834 (1.89)	0.1250 (1.53)	0.1337 (1.48)
STD. IAE	0.0721	0.0479	0.0636
Avg. Time	9.99	11.24	11.36
Avg. Peak	1.117	1.110	1.094
# of crashes	0	0	0

Comparison Between Transfer Function and ZPK-Model

The above table displays the results from testing the exact time delay with two approximations, however, seeing as neither approximation resembles the exact time delay, the system, with the exact time delay, was converted into a Zero-Pole-Gain-model (ZPK model). From the table below, Table 17, the results from both the original transfer function and the ZPK model conversion are displayed across the five controllers. The results show comparable results, with the main difference being the number of crashes for the PRC ZN controller and the "pidstd(PID)" controller.

Table 17: Test results from ZN, "pidstd(PID)", "pidstd(PI)", "megatunerplain", and "megatuner" using "rss"-created with the transfer function and a ZPK-model

	PRC ZN	**Pidstd(PID)**	**Pidstd(PI)**	**Megatunerplain**	**Megatuner**
Transfer Function					
Rel. IAE (Avg. IAE)	0.1974 (1.58)	0.2570 (507.51)	0.1752 (22.18)	0.2146 (13.68)	0.1885 (2.13)
STD. IAE	0.0319	0.0766	0.0224	0.0455	0.0718

Avg. Time	8.02	1976.51	126.59	63.75	11.31
Avg. Peak	1.710	0.985	1.000	1.007	1.110
# of crashes	6	2	1	0	0
ZPK-Model					
Rel. IAE (Avg. IAE)	0.1984 (1.53)	0.2810 (578.77)	0.1837 (39.73)	0.2108 (7.74)	0.1930 (2.70)
STD. IAE	0.0390	0.1587	0.0270	0.0438	0.0616
Avg. Time	7.71	2059.67	216.27	36.70	14.00
Avg. Peak	1.608	0.958	0.997	1.011	1.104
# of crashes	3	7	1	1	0