Design of an Impedance Controller for a 3T1R Parallel Kinematic Manipulator

- CA1 -

Worksheets

Group 733

Aalborg University
Department of Electronic Systems
Fredrik Bajers Vej 7B
DK-9220 Aalborg

This report was written in LaTeX and has been shared online in-between the group members using Overleaf licensed to all students at Aalborg University.

AALBORG UNIVERSITY

STUDENT REPORT

Department of Electronic Systems
Fredrik Bajers Vej 7
DK-9220 Aalborg Ø
http://es.aau.dk

Title:
Design of an Impedance Controller for a 3T1R Parallel Kinematic Manipulator

Theme:
Networked and Control Systems

Project Period:
September 2019 - December 2019

Project Group:
Group 733

Participants:
Frederik Skyt Dæncker Rasmussen
Gowsikan Sathiyaseelan
Javier Martín García
Mikkel Damgård Hardysøe

Supervisor:
Henrik Schiøler
Juan de Dios Flores Mendez

Report Page Numbers: 87

Appendix Page Numbers:

Date of Completion:
July 3, 2021

Abstract:

This paper explores the design of a compliant impedance controller for three translations and one rotation (3T1R) Parallel Kinematic Manipulator (PKM). The PKM used in this project is the Ragnar Robot from Open Robotica. The case for using compliant control for PKMs was made based on the speed, consistency and efficiency of the PKMs. Compliant control for PKM is not yet common in the industry, which also makes a good case for developing such a controller. The kinematic and dynamic equations from previous work with the Ragnar Robot have been studied and applied. An impedance controller has been implemented for control of the Ragnar Robot using these models. The impedance parameters, spring and damper coefficient for each degree of freedom can be adjusted. The damping coefficient is scaled based on the damping factor of a 2nd order mass-spring-damper system. The controller is implemented with dynamic compensation for gravity and coriolis forces, but the acceleration forces were omitted as the system is designed for low speeds when interacting with the delicate objects. The dynamics are calculated using extended general coordinates to greatly reduce complexity of the dynamic models. The communication between the microcontroller and the actuators is carried out through a CAN bus network. The minimum time required by the system to do the communication on the CAN bus while avoiding package loss has been calculated. Simulations were done to test the behavior of the controller. The simulations all showed the expected behavior. The simulations showed that the controller had consistent behavior independent of the position of the mobile platform. It was also concluded that the response for each degree of freedom was independent of the forces acting on the other degrees of freedom. The test for the scaling of the response by scaling the spring coefficients were also according to expectations and overshoot stayed the same. Finally, when testing for the impact of delay on the response of the system, the tests showed that the controller could easily handle much longer delays than is expected to happen in an implementation.

Preface

These worksheets are written as a part of the scientific paper drafted by a group of students in the first semester of their master's degree enrolled in the programme of Control and Automation at the Department of Electronic Systems, at Aalborg University. The reader of these worksheets is therefore expected to have some degree of knowledge on control and automation theory and parallel kinematic manipulators.

The worksheets should be regarded as an additional detailed documentation supporting the scientific paper. It includes the design, development and tests of the control systems for the Ragnar Robot. The worksheets are structured by chapters, section and subsection where the different equations, figures, tables and snippets of code are numbered to indicate which chapter and section they belong to.

The group members thanks supervisors Henrik Schiøler and Juan de Dios Flores-Mendez for their help and guidance throughout this project period and for the exchange of the models for the kinematics, dynamics and other materials. They also thank Aalborg University for providing them with the materials and facilities needed throughout this period.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Manufacturing industries are becoming more automated due to the implementation of robotics. The benefits of automating the manufacturing process are the increase in efficiency, safety and consistency. This has piqued an interest in parallel kinematic manipulators (PKM) in recent decade since they share properties making them ideal for pick and place operations. Compared to serial manipulators, PKMs are faster, more accurate and capable of executing applications demanding a heavy load because of their structure allowing them to distribute the weight among their parallel links. By using PKM´s for pick and place operations relieves the workers from intense repetitive labour and enables them to focus on more productive areas of the manufacturing process. The replacement of humans for tasks requiring repetitive motion is also very cost effective, since they can work continuously 24/7 and faster than human workers.

Traditional position controlled manipulators do not share their workspace with humans because of their lack of environmental constraints. However, modern manipulators are implementing compliant control making the manipulators force controlled allowing human workers to work alongside them. A compliant controlled manipulator is aware of it´s environment and adjusts it´s forces to accommodate for the task at hand. This is convenient for tasks requiring the mobile platform to interact with other objects. E.g. if a position controlled manipulator picked up a tomato from an conveyor belt. It would have no problem following the trajectory and placing the object at the desired destination. However, the tomato might be left deformed because of the manipulator not having any sense of the force applied. This can be resolved by using compliant control which is why these manipulators are becoming popular in the food industry where the objects can be fragile.

Compliant control is a subdomain of continuous feedback force control allowing for changes in the compliance properties and dynamic behavior of a system [3]. Which is split into either direct or indirect force control. Where impedance control falls under the category of indirect control. The purpose of such a controller is to impose a mechanical stiffness and damping which is why they are described as mass-spring-damper-systems [4].

The intention of this paper is to modify an already existing re-configurable PKM called the "Ragnar Robot" built by the company BlueWorkforce [5]. It is usually operated by four actuators and a platform allowing for three transitional movements. However, in this configuration the PKM is redundantly actuated as it has more actuators, than degrees of freedom (DOF). In order to to fully utilize the potential of the Ragnar Robot a platform which allows for a three translational and one rotational configuration (3T1R) is used. This paper is going to explore how to design an impedance controller for a robot with a 3T1R platform. The idea behind this project is to built a PKM meant for picking tomatoes in the food industry which requires the use of compliant control because of their delicate nature.

Chapter 2

Intended Results

The purpose of this chapter is to give an overview of what materials were available for this project and to describe what the intended result of the project is, what should be made, and what its functions should be.

2.1 Materials

When the project was proposed it was promised that a PKM with a platform specially made for the application of picking tomatoes would be available. However, due to some unfortunately events the delivery of the PKM was delayed which resulted in working with the Ragnar Robot already available at Aalborg University. However, this PKM was at first fitted with a fixed platform only allowing for 3T movements which was later changed to a 3T1R platform giving the Ragnar Robot four degrees of freedom. The microcontroller used for controlling the PKM was a Teensy 3.6 with CAN bus interface. Most of the work done throughout these worksheets are based on the kinematics, dynamics and simulations provided by Juan de Dios Flores Mendez.

2.1.1 Ragnar Robot

The robot being worked on during this project is the Ragnar Robot, which is an industrial parallel kinematic manipulator built by the company BlueWorkforce. A company founded by Preben Hjørnet which specialises in making robots and grippers for pick and place robots in the food industry. The Ragnar Robot is built as an affordable, applicable and accessible alternative to their competitors. The whole concept behind the robot is to have the employees work alongside it, making the robot do the dumb, dirty and dangerous tasks while the humans do the more intricate ones [5]. This is possible since the robot is torque controlled, as opposed to common industrial robots which are position controlled. The difference is that unlike common robots the Ragnar Robot can accommodates the torques and forces when in contact with an object, which allows the robot to complete delicate tasks. This is especially handy for pick and place robots in the food industry. However, the Ragnar Robot is very versatile and can also be used for other tasks outside the food industry which require an industrial manipulator.

Figure 2.1: Picture of the Ragnar Robot illustrating the structure of the Robot [1].

The Ragnar Robot is a high speed delta type parallel kinematic manipulator used for pick and place applications built to improve productivity while reduce cost [4]. The robot is usually over actuated or redundant actuated meaning that it has four actuators but only three degrees of freedom. The specific actuators used are PD4-C6018L4204-E-08 from Nanotech which are NEMA 24 stepper motors with integrated controllers allowing them to be torque controlled. The robot is constructed with four stepper motors mounted on a base platform. Each of the motors have belt driven links going to another set of links which are connected at the mobile platform where a gripper can be placed as seen on **Figure 2.1**. The mobile platform and the gripper usually used for the Ragnar Robot are constructed in way which eliminates the need for tools when interchanging grippers, making the process a lot easier for the workers. Teensy 3.6 For controlling the Ragnar Robot, a microcontroller with Can bus interface is needed. The chosen microcontroller for this purpose is the Teensy 3.6 which is a breadboard-friendly development board. The reasoning for choosing this specific microcontroller is because of its many benefits compared to others. The Teensy 3.6 is a powerful microcontroller powered by a 32-bit ARM Cortex-M4 processor operating at a clock frequency of 180MHz. The microcontroller is also equipped with 1 M Flash, 256K Ram and 4K EEPROM making it capable of quickly computing tasks with a heavy workload. Besides being very powerful it has 62 I/O ports allowing it to be connected with multiple peripheral devices [6]. The microcontroller comes pre-flashed with a bootloader eliminating the need for an external programmer meaning that it can be programmed directly through the micro USB port. It can be programmed in C or even on Arduino IDE using Teensyduino. The Teensy 3.6 comes packed with many features and functionalities in a very small form factor of 62.3mm x 18.0mm [7].

The authors of this project have also had some experience working with the microcontroller since it has been introduced to them during one of their courses.

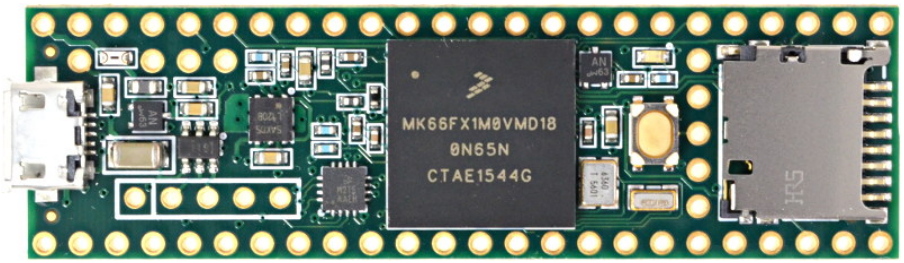


Figure 2.2: The Teensy 3.6 Development board [2].

2.2 Functional Requirements

In order to achieve the desired functionality, have the functional requirements for the control of the Ragnar Robot been determined. The functional requirements are simple requirements without any technical specifications. However, they are set up as a guide for the project. On **Table 2.1** the requirements can be seen ranked by priority.

Priority	Requirements
1	Positional Impedance Control
2	Stable with the delay expected from of the CAN bus and the control loop
3	Adjustable Impedance Parameters
4	Path Planning

Table 2.1: The functional requirements for the system.

2.2.1 Positional Impedance Control

Impedance Control based on a static reference position. This is the first priority for the system. The positional impedance control should behave as a mass-spring-damper system when external forces are applied. And it should be stable under normal circumstances, meaning no delay, no interference, and no packet loss.

2.2.2 Stable with the Delay Expected from of the CAN bus

The system should be stable when the delays introduced from the CAN bus is present. The system uses the CAN bus and thus a communication delay is introduced on top of the time the control loop takes to complete. This should be taken into consideration when designing the system.

2.2.3 Adjustable Impedance Parameters

The parameters for the spring and dampener should be adjustable to change the response of the system depending on application. It is desired that a desired response with a certain overshoot can be set and then the spring coefficient can be changed to stiffness of the system with no change in response and overshoot

2.2.4 Path Planning

Once all of the other things are implemented is the next requirement to make a path planer. This will allow the robot to execute actual work. This would be implemented as a controller on top of the impedance controller. This would be done by setting the reference and the impedance parameters depending on the task.

2.3 System Overview

The block diagram below gives a clear picture of the whole system and its subsystems. It makes it easy to visualise the input and output between each of the blocks.

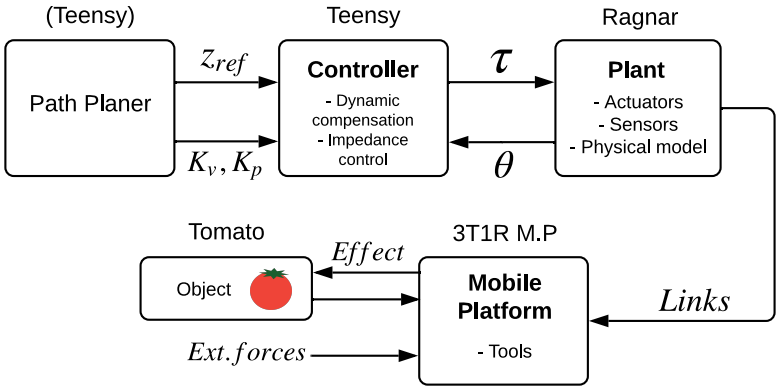


Figure 2.3: An overview of the whole system.

Figure 2.3 showcases the ideal system where the path planner sends the reference input, stiffness and damping parameters to the Teensy controller. Which can be used for changing the parameters for specific applications or during tasks. This can be convenient since PKMs act differently when moving freely compared to when they interact with objects. The controller on the other end calculates the dynamic compensation and impedance control for the system and sends the torques to the plant, based on the angle of the actuators send from it. In this case the plant is the Ragnar Robot which is constructed of actuators connected to links combined at the mobile platform. Where a tool such as gripper could be implemented in order to pick different delicate objects such as tomatoes without bruising them by compensating for the external forces.

However, as it stands now has nothing been implemented on the system but rather done through simulations. Which is why the path planer is not currently implemented but instead set as constants in the code. The same goes for the external forces acting upon the mobile platform. They are also values set in the code.

2.4 Problem Statement

According to the tomato workers´ health and safety guide [8]. The workers employed at the production line endure labor intensive tasks. Leading to illnesses and injuries caused by unhealthy physical postures and chemical exposure which can occur when picking and handling tomatoes that have been treated with chemical agents.

In response to these issues this project explores the usage of PKMs as a replacement for human workers in regard to reduce the number of workers on repetitive labor intensive tasks. However, because a lot of food related goods such as tomatoes are delicate, will a traditional PKM not suffice as it might damage and bruise the goods. In order to handle these types of objects a system is proposed that uses compliant control with a 3T1R platform for a four arm PKM, specifically the Ragnar Robot.

Chapter 3

Design of the Models for the System

In order to control the Ragnar, the study of the kinematic equations of the robot is needed. The kinematics and the dynamics have already been derived for the Ragnar Robot when only moving in three dimensions. This makes the system a redundant parallel kinematic manipulator since it has four actuators [4]. However, in this project an additional degree of freedom is added, the rotation around one of the axes. This is done by modifying the mobile platform or "gripper" of the Ragnar Robot. This new gripper gives the Ragnar Robot four degrees of freedom while still having four actuators.

The kinematics and the dynamics of the robot is not derived in this paper but are going to use the derivation from the other papers made by Juan and Henrik [9].

3.1 System Description

As mentioned in **Chapter 1** the PKM worked on during this paper is the Ragnar Robot from Blue-Workforce. It is a delta type robot mainly used for pick and place operations in the food industry. The robot is constructed of a base platform with four actuators affixed on either side. Each of the actuators controls a belt driven link.

The four arms consist of two links connected with passive joints. As it can be seen on **Figure 3.1** the proximal link and the distal link are connected to a mobile platform. On the mobile platform a tool such as a gripper can be attached to interact with different objects. The links and joints are made of lightweight material, the joints are ball joints. The actuators used for the PKM are the PD4-C stepper motors with integrated controllers from Nanotec [10] that allows them to be torque controlled.

The mobile platform has different configurations which allows it to be controlled and rotate around one axis at a time. The mobile platform will always be locked to only be able to rotate in the same direction.

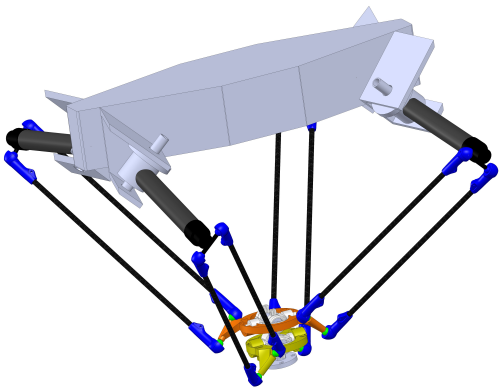


Figure 3.1: A model of the Ragnar Robot with the rotating mobile platform.

3.2 Gear Ratio

The actuators are not directly connected to the arms of the Ragnar Robot. Instead they are connected by a belt with a gearing as seen on **Figure 3.2**.

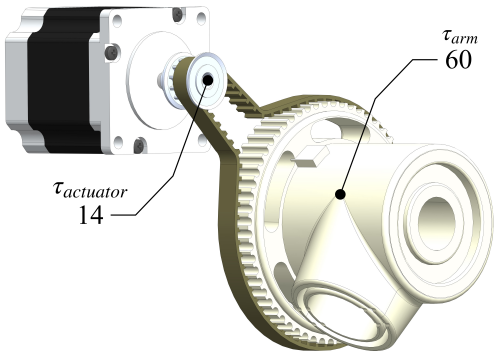


Figure 3.2: Gear ratio from the stepper motor to the limbs of the manipulator.

This gearing results in a conversion of torque and angular velocity. The constant torque conversion, K_τ, from the actuator is given by the teeth ratio between actuators gear, n_{actuator}, and the gear attached to proximal link, n_{arm}, as:

$$K_\tau = \frac{n_{\text{arm}}}{n_{\text{actuator}}} = \frac{60}{14} \tag{3.1}$$

Thus the torque on the proximal link τ_{arm} is given as:

$$\tau_{\text{arm}} = \tau_{\text{actuator}} \cdot K_\tau \tag{3.2}$$

To get the angular position or velocity of proximal link, the inverse ratio is used as:

$$\theta_{\text{arm}} = \theta_{\text{actuator}} \cdot \frac{1}{K_\tau} \tag{3.3}$$

Furthermore, the position from the encoder in the actuators return a value n_{encoder} between 0 and 4000 mapping to a rotation in radians from 0 to $2 \cdot \pi$, thus to get the angle of the arm from the encoder count the following equation can be used:

$$\theta_{\text{arm}} \approx n_{\text{encoder}} \cdot \frac{2 \cdot \pi}{4000} \cdot \frac{1}{K_\tau} \tag{3.4}$$

This equation is used when implementing control on the Ragnar.

Lastly it should be considered that a belt like this can introduce hysteresis which can reduce the precision of the system.

3.3 Kinematics Analysis

In this chapter the equations that make up the Ragnar Robot model will be described. It should be mentioned that they were developed by Juan de Dios Flores-Mendez, based previously on a method used for a delta robot with three actuators and three degrees of freedom [9].

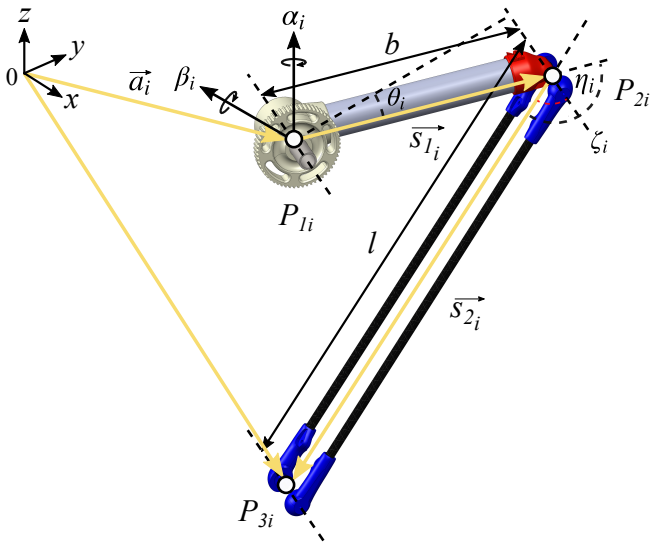


Figure 3.3: Generalized coordinates.

By placing the origin of coordinates (coordinates with sub-index 0) in the center of the fixed platform, the vector that joins each of the motors, named as P_{1_i} in 3.3, with the origin is denoted as $\vec{a_i}$. If a new coordinate system (sub-index 1) is placed in P_{1_i}, and in addition, this system is rotated so that the axes coincide with the rotation of the actuator with respect to the origin, the end of the first member, named as P_{2_i} through the following equation:

$$P_{2_i} = \vec{a_i} + R_z^{\alpha_i} R_x^{\beta_i} R_y^{\theta_i} b\vec{i} \tag{3.5}$$

Where b is the length of the proximal limb, R_z and R_x rotation matrices with respect the axis z and x respectively and α_i and β_i the fixed angles of the actuator. Additionally, θ_i corresponds to the rotation of the proximal limb. Grouping all the three rotation matrices in one:

$$R_{1i} = R_z^{\alpha_i} R_y^{\beta_i} R_y^{\theta_i} \tag{3.6}$$

And grouping the second term in (3.12) as:

$$\vec{s_{1_i}} = R_{1i} b\vec{i} \tag{3.7}$$

P_{2_i} could be rewritten as:

$$P_{2_i} = \vec{a_i} + \vec{s_{1_i}} \tag{3.8}$$

Repeating the process of placing a new coordinate system, now in P_2 and denoted by the sub-index 2, two new rotation angles are needed to refer this new system to respect the origin. These angles are η_i and ζ_i. So if we want to reach the end of the distal link denoting that point as P_{3_i}, the point that joins the final part of an arm with the mobile platform, knowing that l is the length of the distal link, the equation that gives us the position of P_{3_i} with respect to the origin in the center of the fixed platform would be:

$$P_{3_i} = \vec{a_i} + \vec{s_{1_i}} + R_{1i} R_z^{\zeta_i} R_y^{\eta_i} l\vec{i} \tag{3.9}$$

And with the following condensations:

$$R_{2i} = R_z^{\zeta_i} R_y^{\eta_i} \tag{3.10}$$

$$\vec{s_{2_i}} = R_{1i} R_{2i} l\vec{i} \tag{3.11}$$

P_{3_i} could be rewritten as:

$$P_{3_i} = \vec{a_i} + \vec{s_{1_i}} + \vec{s_{2_i}} \tag{3.12}$$

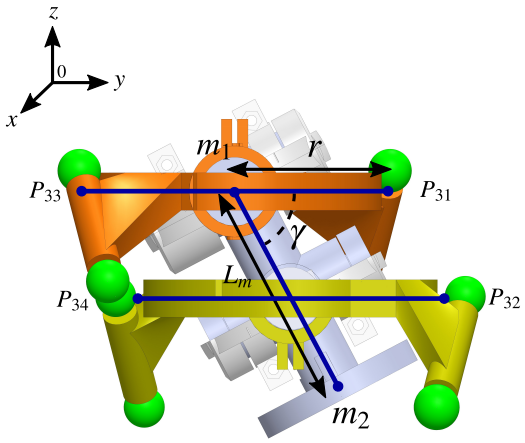


Figure 3.4: 2D perspective of the mobile platform.

The constraint equations are:

$$||P_{3_i} - P_{2_i}|| = l^2 \tag{3.13}$$

With this, the kinematic equations are given by generalized coordinates, called q_i, that determine the position of each arm. This generalized coordinates are composed of the active and passive angles of the joints, being $q_i = [\theta_i\ \eta_i\ \zeta_i]^T$.The angle θ_i is the active angle, being η_i and ζ_i the passive angles.

The new mobile platform that allows for a new degree of freedom in a way of rotation around the x-axis is shown in a 2D view in **Figure 3.4**. It is connected to the distal links of each arm in the points P_{3_i} as can be seen in the figure.

The center point of the upper disc is named as m_1 and it coordinates in reference with the origin can be obtained as: $m_1 = P_{3_1} \cdot \vec{i} + (P_{3_1} - r) \cdot \vec{j} + P_{3_1} \cdot \vec{k}$ being r the distance between P_{3_i} and m_1. With a relative translation between the second disc and the first in the y direction we can obtain a rotation in the x axis.

The coordinates of the mobile platform (where the tools will be placed) denoted as m_2 will be: $P_{3_1} \cdot \vec{i} + (P_{3_1} - r + L_m cos(\epsilon)) \cdot \vec{j} + (P_{3_1} - L_m sin(\epsilon)) \cdot \vec{k}$

3.4 Dynamics

The Dynamics is the forces acting upon the system. A method used for deriving the dynamics is Lagrangian mechanics. It focuses on the particles in the system and the trajectory of these particles [11]. The Lagrangian method uses a function of generalized coordinates where their time derivatives contain the information about the dynamics of the system. The trajectory of the particles in the system is derived using one of two equations. The second type of Lagrangian takes the constrains of the system into consideration from the beginning. In order to determine the dynamics for the Ragnar Robot gripper, the dynamics of each link are first derived [4]. The theory builds on the Lagrangian L[11]:

$$L_i = K_{e_i} - P_{e_i} \tag{3.14}$$

Where K_{e_i} is the kinetic energy and P_{e_i} is the potential energy of each link. The kinetic energy is:

$$K_{e_i} = \frac{1}{2} I_b \dot{\theta}_i^2 + \frac{1}{8} m_b \dot{\vec{s}}_{1_i}^2 + \frac{1}{2} m_l \left(\dot{\vec{s}}_{1_i} + \frac{1}{2} \dot{\vec{s}}_{2_i} \right)^2 \tag{3.15}$$

While the potential energy is:

$$P_{e_i} = m_b \left(\frac{1}{2} \vec{s}_{1_i}^T \right) g + m_l \left(\vec{s}_{1_i} + \frac{1}{2} \vec{s}_{2_i} \right)^T g \tag{3.16}$$

I_b and m_b is the inertia and mass of the proximal link while m_l is the mass of the distal link.

Using the second type of Lagrangian, the equation of motion can be determined by the following formula:

$$\frac{d}{dt} \left(\frac{\partial L_i}{\partial \dot{q}} \right) - \frac{\partial L_i}{\partial q} = Q_{ext_i} \tag{3.17}$$

[11] Being q_i the generalized coordinates and The Q_{ext_i} are all of the external forces applied to the system:

$$Q_{ext_i} = \tau_i - J_i \cdot f_i \tag{3.18}$$

This Q_{ext_i} are: The actuated torques applied at the beginning of the proximal link; the Coriolis forces that; the gravitational forces applied at the center of masses of the two links that can be seen in **Figure 3.5** and the mobile platform, and the external forces applied to the mobile platform.

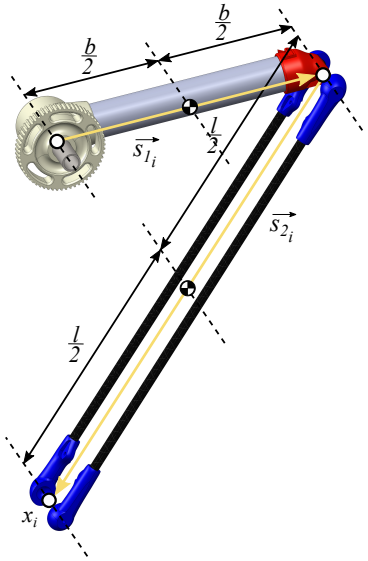


Figure 3.5: Model of Ragnar with center of masses.

Deriving (3.15) and (3.16) and including them into (3.17) the dynamics of the system are given as:

$$M(q_i) \cdot \ddot{q}_i + C(\dot{q}_i, q_i) + G(q_i) = \tau_i - J_i^T \cdot f_i \tag{3.19}$$

Combining the dynamics for all four arms results in the following:

$$M(q_x) \cdot \ddot{q_x} + H(q_x, \dot{q_x}) + G(q_x) + c^T \cdot \lambda = \begin{bmatrix} \tau_{\text{total}} \\ 0 \end{bmatrix} \tag{3.20}$$

Where $M(q_i)$ is the mass 8 by 8 matrix, $C(\dot{q}, q)$ is the centrifugal 8 by 1 vector, with 0 in the joint space values and $G(q)$ is the 8 by 1 gravitational vector. $c^T \cdot$ is the constraints.

Using the general coordinates, joint and task space, as inputs of **(3.19)** the output will result on the dynamic forces applied.

As more general coordinates are added the simpler the equation becomes, but on the other hand, more constraints are required.

Chapter 4

Design of Control

When working with delicate tasks, must the control performed on the robots consider not only the speed and accuracy, but also the forces that are applied. This is known as compliant control. compliant control is a regulation of the force along with the control of the movement. This can be done passively or actively. The passive way implies the use of a compliant mechanism added at the mobile platform, which does not imply any real change to the system control. On the other hand, active control implies a control of the forces between the mobile platform and the environment while the movement is performed. Likewise, the measurement of the forces can be carried out directly, by means of sensors, or indirectly, where there is no direct measurement of this. In this project as the indirect active control method, the so-called impedance control has been used [12].

4.0.1 Impedance Control

In the impedance control method, the mobile platform is compared with a system shown in the **Figure 4.1**. It consists of: A spring, which defines the force as an output proportional to the compression tension in the spring; and a damper, which defines the force as an output, based on the speed of the mobile platform.

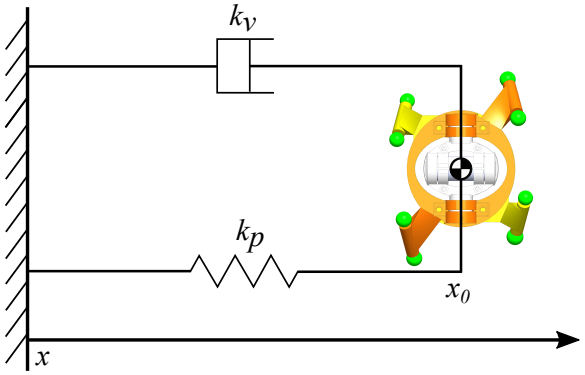


Figure 4.1: Impedance Control, damper and spring.

By controlling the constants associated with the spring and the damper, it can be varied how the robot behaves against external forces in different axes. This allows a different control when the robot is moving freely, compared to when it is interacting with an object. It also allows the robot to behave in a much more rigid way along one of the axes compared to the others.

The variable K_p is designed by the spring, associated to the stiffness of the system, and by means of the damper the variable K_v is designed. The mass M of the system will also affect the system proportional to its mass. The relationship that these variables have with respect to external forces is shown in:

$$\tau_{ext} = M \cdot (\ddot{z}_r - \ddot{z}) + k_v \cdot (\dot{z}_r - \dot{z}) + k_p \cdot (z_r - z) \tag{4.1}$$

Where z would be the task space coordinates. In **Figure 4.1** z would just be one dimension, and would be along the x.

The goal of the impedance controller is to have such a relationship along each DOF of the mobile platform. That is, along each of the three xyz-axis, and around the axis of rotation of the platform.

Around the axis of rotation, the mass will be the an inertia, but spring and damper have the same effect.

4.1 Design of Spring and Damper Coefficients

The mass-spring-damper system is a 2nd order system with the transfer function:

$$\frac{\omega_n^2}{s^2 + 2 \cdot \zeta \cdot \omega_n \cdot s + \omega_n^2} \tag{4.2}$$

Where ζ is the dampening, and ω_n is the natural frequency. What is desired is that a response with a certain overshoot can be designed and the power of the spring can be adjusted without changing the overshoot. In the system that is being designed the k_p is equivalent to the ω_n^2 and k_v is equivalent to $2 \cdot \zeta \cdot \omega_n$. The overshoot is based on constant terms $2 \cdot \zeta$ while ζ in the 2nd order transfer function takes mass into consideration. It will in our system, be assumed that the mass will be almost constant as it only changes slightly based on configuration. Thus, ζ will just be considered a constant used to adjust response and overshoot of the system. Hence to scale the k_v to the k_p the following relationships are used:

$$k_v = \zeta \cdot \omega_n \tag{4.3}$$

$$\omega_n = \sqrt{k_p} \tag{4.4}$$

Thus, k_v as a function of ζ and k_p will be to keep a consistent response with similar overshoot when changing the stiffness of the system:

$$k_v = \zeta \cdot \sqrt{k_p} \tag{4.5}$$

4.2 Dynamic Compensation

To make an impedance controller is it desired to make the relationship between just the external forces and the reacting forces from the mobile platform. However, there are other forces acting on the system. It is therefore desired to compensate for these forces. All major forces acting on the system in the task space can be described as:

$$M(q) \cdot \ddot{z} + H(q, \dot{q}) + G(q) + \tau_{ext} = J^{-T}(q) \cdot \tau \tag{4.6}$$

Where $M(q)$ is the mass matrix which is depended on the configuration of system and thus dependent on the configuration of the actuators q.

$\ddot{z}$ is the acceleration of the platform in task space.

$H(q, \dot{q})$ is the coriolis forces which are both dependent on the configuration of the system, q, and the speed of the movement of the system $\dot{q}$.

$G(q)$ is the gravitational forces which also depend on the configuration of the system, q.

τ_{ext} is the external forces acting on the system, this will be represented as the relationship that was designed for the impedance control.

$J^{-T}(q) \cdot \tau$ is the forces by the actuators acting on the system converted by the Jacobian, $J(q)$, from the torques, τ, output by the actuators.

A few assumptions have been made to simplify the equations. It has been assumed that the system is perfectly rigid, and that there is no friction. In this project, which is designed for operation at low speed, where the compliant control is most important. The acceleration term and the Coriolis forces are negligible and can be omitted without much effect on the system. However, these terms will be explained.

Mass Matrix

The mass matrix $M(q)$ is the matrix equivalent of the mass that needs to be accelerated in a certain position q, when a certain acceleration is applied along the general coordinates. This includes the linear acceleration of masses and the inertia of the system that must be accelerated. Thus, when multiplied by the acceleration of the system it will be equivalent to the forces that are needed to accelerate the system. This force however, is very low compared to the other forces at low speeds, therefore it is omitted as it is very difficult to get a good estimate of the acceleration when the only measured data is the position, which would require a double differentiation.

The Coriolis Vector

The Coriolis matrix describes the forces acting on the system based on Coriolis effect which depends on the position q, the rotation and movement, $\dot{q}$, of the parts of the robot. This effect is also small at low speed, but it is only dependent on the speed of the system which can be approximated by a single differentiation of the position. Therefore, it will be included in the controller.

Gravity Matrix

$G(q)$ is the gravitational forces acting on the system. The gravity forces depend on the position q of the system and is the most significant force at low speeds. Without this compensation the system would always end up with a steady state error and is therefore important for precise control of the mobile platform.

4.2.1 Compensation using the Jacobian

The forces are calculated in task space, thus to compensate for them using the actuators the Jacobian can be multiplied with the forces to give the actuator torques τ:

$$J^T(q) \cdot (M(q) \cdot \ddot{z} + H(q, \dot{q}) + G(q)) = \tau_{dynamic} \tag{4.7}$$

Splitting the output torque τ into τ_{dynamic} and $\tau_{\text{impedance}}$ and inserting into 4.6 we have:

$$J^T(q) \cdot (M(q) \cdot \ddot{z} + H(q, \dot{q}) + G(q) + F_{\text{ext}}) = \tau_{\text{dynamic}} + \tau_{\text{impedance}} \tag{4.8}$$

The compensation cancels with the dynamic forces and we have the following relationship:

$$J^T(q) \cdot F_{\text{ext}} = \tau_{\text{impedance}} \tag{4.9}$$

Which can be used to design the impedance torque.

4.2.2 Impedance Control for Multiple Degrees of Freedom

As the system has four DOF, three translations along the mobile platforms xyz-axes and one rotation around the x-axis, γ. The system should have a response for each DOF. The goal is to have a consistent response for each DOF, independent on position and on the forces acting on other DOF's.

Thus, we can use our relationship with external forces from 4.1 and using matrices K_p and K_v defined as:

$$K_p^d = \begin{bmatrix} k_{px} & 0 & 0 & 0 \\ 0 & k_{py} & 0 & 0 \\ 0 & 0 & k_{pz} & 0 \\ 0 & 0 & 0 & k_{p\gamma} \end{bmatrix} \tag{4.10}$$

$$K_v^d = \begin{bmatrix} k_{vx} & 0 & 0 & 0 \\ 0 & k_{vy} & 0 & 0 \\ 0 & 0 & k_{vz} & 0 \\ 0 & 0 & 0 & k_{v\gamma} \end{bmatrix} \tag{4.11}$$

Inserted in 4.9:

$$J^T(q) \cdot (M \cdot (\ddot{z}_r - \ddot{z}) + K_v \cdot (\dot{z}_r - \dot{z}) + K_p \cdot (z_r - z)) = \tau_{\text{impedance}} \tag{4.12}$$

Which should be the full impedance control torque, but as previously we will omit the acceleration term as it cannot be measured reliably, and it will be low at low speeds. The impedance control torque will finally be

$$J^T(q) \cdot (K_v \cdot (\dot{z}_r - \dot{z}) + K_p \cdot (z_r - z)) = \tau_{\text{impedance}} \tag{4.13}$$

And the full torque equation will therefore be:

$$\tau = J^T(q) \cdot (H(q, \dot{q}) + G(q) + K_v \cdot (\dot{z}_r - \dot{z}) + K_p \cdot (z_r - z)) \tag{4.14}$$

To check that we have relationship we want with F_{ext} we can insert $\tau_{\text{impedance}}$ from 4.13 into 4.9 to get:

$$J^T(q) \cdot F_{\text{ext}} = J^T(q) \cdot (K_v \cdot (\dot{z}_r - \dot{z}) + K_p \cdot (z_r - z)) \tag{4.15}$$

Then the Jacobians $J^T(q)$ cancels and we have:

$$F_{\text{ext}} = (K_v \cdot (\dot{z}_r - \dot{z}) + K_p \cdot (z_r - z)) \tag{4.16}$$

4.3 Control System

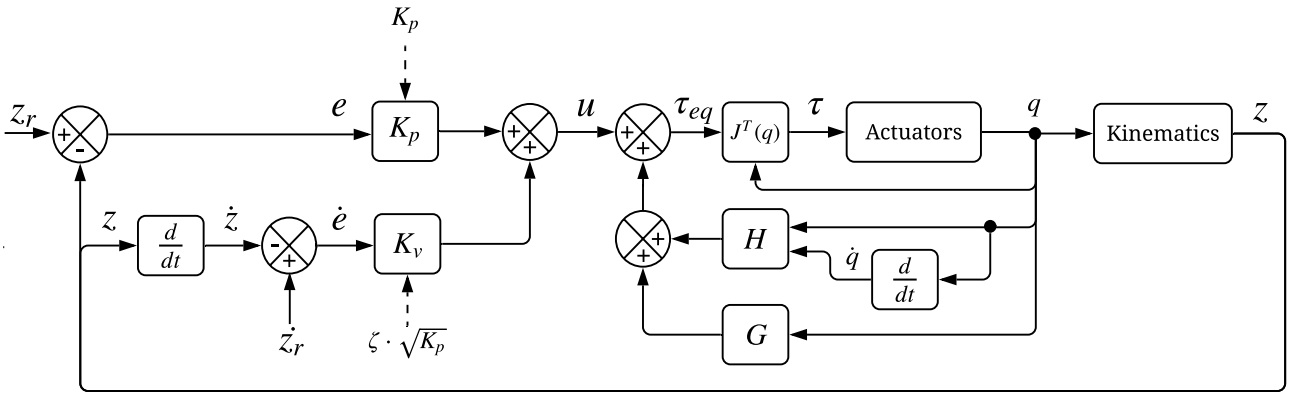


Figure 4.2: Control system for the Ragnar Robot.

On **Figure 4.2** a block diagram of the system can be seen. However, when implementing the code a few alterations has been made in the equations to more easily work with the kinematic and dynamic models.

4.3.1 Extended General Coordinates for Simpler Dynamic Model

The general coordinates, q, used to describe joint space of the system is comprised of the four actuator angles:

$$q = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \end{bmatrix} \tag{4.17}$$

The general coordinates, z, used to describe the task space of the system is comprised of the xyz-position of the platform as well as an additional fourth dimension which is the rotation, γ, around an axis with respect to an axis:

$$z = \begin{bmatrix} x_p \\ y_p \\ z_p \\ \gamma_p \end{bmatrix} \tag{4.18}$$

To make the construction of the dynamics easier it is done with general coordinates describing both the joint and task space. These extended general coordinates q_x will consist of the joint space coordinates combined with the task space coordinates in an 8 by 1 matrix:

$$q_x = \begin{bmatrix} \theta \\ z \end{bmatrix} \tag{4.19}$$

This gives the system 8 equations and thus 4 constraint equations are needed.

$$M(q_x) \cdot \ddot{q_x} + H(q_x, \dot{q_x}) + G(q_x) + c^T \cdot \lambda = \begin{bmatrix} \tau_{\text{total}} \\ 0 \end{bmatrix} \tag{4.20}$$

To then use these matrices for control they are pre-multiplied by R^T. R can be described as:

$$R = \begin{bmatrix} J^{-1}(q) \\ I_4 \end{bmatrix} \tag{4.21}$$

R^T has following relationship with c^T

$$R^T \cdot c^T = \mathbf{0} \tag{4.22}$$

Thus equation:

$$R^T \cdot (M(q) \cdot \ddot{z} + H(q, \dot{q}) + G(q) + c^T \cdot \lambda) = R^T \cdot \begin{bmatrix} \tau_{\text{total}} \\ 0 \end{bmatrix} \tag{4.23}$$

Leads to:

$$R^T \cdot M(q_x) \cdot \ddot{z} + R^T \cdot H(q_x, \dot{q}_x) + R^T \cdot G(q_x) = J^{-T}(q) \cdot \tau_{\text{total}} \tag{4.24}$$

Adding the external forces and the following definitions:

$$M = R^T \cdot M(q_x) \tag{4.25}$$

$$G = R^T \cdot G(q_x) \tag{4.26}$$

$$H = R^T \cdot H(q_x, \dot{q_x}) \tag{4.27}$$

we get:

$$M \cdot \ddot{z} + H + G + F_{\text{ext}} = J^{-T}(q) \cdot \tau \tag{4.28}$$

With the previously described relationship between external forces F_{ext} and solving for the actuator torques we get the following equation for the output torque:

$$\tau = J^T(q) \cdot (M \cdot \ddot{z} + H + G + K_v \cdot (\dot{z}_r - \dot{z}) + K_p \cdot (z_r - z)) \tag{4.29}$$

And finally, to avoid using acceleration, as a good estimation cannot be made from a double differentiation of the position, and the system is designed for low speeds where acceleration is low and negligible. The terms using acceleration is therefore omitted. Which results in the following final output torque equation:

$$\tau = J^T(q) \cdot (H + G + k_v \cdot (\dot{z}_r - \dot{z}) + k_p \cdot (z_r - z)) \tag{4.30}$$

This is the equation implemented and used to control the system, a system diagram can be seen on **Figure 4.2**.

4.4 Implementation and Simulation of Controller

The first step in designing the controller is to test that it behaves as expected, this can more easily be done by doing simulations in a higher-level programming language such as Matlab and use the equations directly instead of implementing it directly in C++. As mentioned in **Section 2.1** simulation files and models of the Ragnar Robot were made available for the project. The files were all written for Matlab, and thus the controller was also written for Matlab. The code can be seen on the **Listing 4.1**.

```
function [tau] = control_3T1R(q, zr, kp, zeta)
    if nargin < 3 % this is for default values if no zeta or kp is set
       kp = [3 2 1 1]*2;
       zeta = 0.8;
    end
    kp = diag(kp);
    %variables for further development
    ddzr = 0; % ref acceleration;
    dzr = 0; % ref velocity;

    % used to test kv calculated as first or second order system
    DAMP_2ND_ORDER = true;
    if(DAMP_2ND_ORDER)
     kv = 2 * zeta * sqrt(kp) * 1;
    else
      kv = zeta * kp;
    end

    dt = 0.002; % Time steps, this would be nice as an input to the function

    z = q(5:8) % z task space
```

```
22
23      q = q(1:4); % q joint space
24
25      persistent q2 % q2 is used to save the previous value to get the derivative of q,
            dq
26      if isempty(q2)
27        q2 = q;
28      end
29      dq = (q - q2) / dt;
30      q2 = q;
31
32      persistent z2 % z2 is used to save the previous value to get the derivative of z,
            dz
33      if isempty(z2)
34          z2 = z;
35      end
36      dz = (z - z2) / dt;
37      z2 = z;
38
39      % dynamics, params = parameters
40      persistent params % parameters for the Ragnar used to calculate the dynamics
41      if isempty(params) % parameters are set once, on the first call of the function
42          x = [pi/3  280/1000  114/1000  pi/12  600/1000  100/1000  70];
43          % = [gama  Ax  Ay  alpha outer_arm  r  *unknown*];
44          params = XForm_Parameter(x); % geometric parameters
45      end
46      persistent mass_params % mass parameters are set once, on the first call of the
            function
47      if isempty(mass_params)
48          mass_params = XForm_LinkageProperty(params); % dynamic parameters as mass,
                inertia
49      end
50      % extra parameters for the rotating platform
51      m_down = 50e-3; % kg
52      m_up = 50e-3; % kg
53      platform_mass = [m_down m_up ];
54      h0 = 40e-3; % distance from end effector to lower platform
55      h = 80e-3; % distance from end effector to upper platform
56      hs = [h0 h];
57      [Mx, Hx, Gx] = ragnar_dynamic_parts_x_rotated( [q;z], [dq;dz], params, mass_params,
            platform_mass, hs); % extended dynamic matrices
58
59      % Jacobian
60      r1 = [55e-3 60e-3 -25e-3]';
61      r3 = -r1; r3(3) = r1(3); % invert x and y but z remains
62      r2 = [-80e-3 50e-3 -15e-3]';
63      r4 = -r2; r4(3) = r2(3); % invert x and y but z remains
64
65      r_all = [r1 r2 r3 r4]; % more parameters for the platform
66      [Am, Bm] =  ragnar_rotated_x_AB( [q;z], params,hs,r_all); % A and B matrix used to
            calculate Jacobian
67      J = Am\Bm;  % jacobian
68      Ji = pinv(J); % inverse jacobian
69
70      R = [Ji ; eye(size(Ji,2))]; % R used to go from qx to q
71
72      H = R'*Hx; % the coriolis forces
73
74      G = R'*Gx; % gravity forces
75
76      %M = R'*Mx*R; % Mass matrix (not in use)
77
78      % Handling the not continues rotation position
```

```
79      rots = [zr(4) - z(4) ;zr(4) - (2*pi + z(4))]; % calculating both distances around
            the circular coordinates
80      % the lowest value corresponding to the error
81      [~ , rot_index] = min(abs(rots));
82      erot = rots(rot_index);
83
84      % error
85      e = [zr(1:3) - z(1:3);erot];
86
87      persistent e2 % save previous error to get derivative of error, de
88      if isempty(e2)
89          e2 = e;
90      end
91      de = (e - e2)/dt;
92      e2 = e;
93
94      u =  kv*(de) + kp*(e); % impedance control signal
95
96      % disp(u);
97
98      tau = J'*(u + H + G); % final torques for the actuators
99  end
```

Listing 4.1: Matlab code used to control the 3T1R model in the simulations.

The code taken from Matlab and can be seen in the **Listing 4.1** is the controller code. It is made as a function which takes the inputs, q, z_r, k_p, $zeta$. The q is the position joint and task space given in a 8 by 1 vector. z_r is the reference given to the controller in task space. K_p is the spring parameter and ζ is overshoot parameter used for calculating the dampening parameter K_v.

4.4.1 Simulation

The simulations were done numerically with integration of the calculated velocities and accelerations using the Runge-Kutta method [13]. This method is a good estimation for short simulations such as the ones used to test the system. The equations used has both actuator torque and external forces as input. The model is controlled by setting the actuator torques, and the controller used the actuator angles as input. For the tests a constant force is applied at the beginning of the simulation. The reference position is kept at the starting position, to have the response just to the external forces.

4.4.2 Controlling the 3T Model

The first simulations were done for the 3T model. For this model the forward kinematics was available, and the control is robust. As the controller is very similar to the one of 3T1R most of the initial testing was done using the 3T model.

4.4.3 Controlling the 3T1R Model

When the 3T1R models were available it was without the forward kinematics. Thus, a numerical method was used to estimate the forward kinematics, specifically the newtons method, using 30 iterations. However, this method is unstable and sometimes starts deviating from the actual position, this causes the controller to behave in unexpected ways which makes the simulations useless. It is not clear what exactly causes this issue. However, some simulations where the estimation didn't deviate

from the actual position and rotation, the results were good. Therefore, the simulations were done in the positions and with forces that didn't result in simulation to crash. It was however not possible to implement a simulated delay for the control function for the 3T1R model as any attempt would make estimation of the forward kinematics deviate for unknown reasons. The models for the 3T was used to simulate the effects that a delay would have for the control as the models are very similar and should behave in similar ways once a stable way of calculating the forward kinematics is found.

4.4.4 Calculating Error on Circular Position of the Rotation

While the position of the mobile platform in the xyz-directions are continues, the rotation of the platform is given as a circular value between $[-\pi; \pi]$. This results in a jump of 2π when crossing this value, which results in the wrong behavior of the controller around this value. To avoid this behavior the error is calculated as the shortest of two ways around the circle, this could cause problems if reference and position is exactly on opposite sides of the circle which is unlikely to happen. The function can be described as follows:

$$e_\gamma = \min|x|[(\gamma_r - \gamma), (\gamma_r - (2 \cdot \pi + \gamma)] \tag{4.31}$$

γ is the angle of rotation around the x-axis. γ_r is the reference angle of rotation around the x-axis. e_γ is error in rotation around the x-axis. $\min|x|$ chooses the value with the lowest magnitude. The implementation of this can be seen in the code on **Figure 4.1** lines 79-82.

4.5 Conclusion of the Simulation Tests

All the simulations can be found in the test journals. The controller delay simulation in **Chapter 11**, the scaling of the controller parameter K_p in **Chapter 10**, the simulation for independence of the axes **Chapter 9**, and the test for independence of the positions **Chapter 12**.

The object of the tests is to verify the behavior of system under different conditions. The behavior of the controller should be that the platform reacts to external forces in each of the translational-axes and around the axis of rotation. It should do so based on the parameters specified in K_v and K_p. The behavior in each of the four DOFs should be independent of the systems position and the parameters set for the other axis. To test this, a step force or torque was added to the systems DOFs, the position was recorded, and the results were analyzed to form a conclusion on the behavior of the controller.

There have been conducted four different tests where each have their own objective. The tests contain multiple simulations each. The results of the simulations have been collected into one graph for each test.

The first test deals with how the controller reacts to scaling of the control parameter K_p and K_v, to see if the response scales as expected.

The second test is to investigate if the controller responds along each DOF, independently of the forces applied to other DOFs at the same time.

The third test makes sure that the controller's behavior is independent of the position and rotation of the platform.

The fourth test deals with delay in the controller, but due to implementation problems it was done on a similar model with no rotation. In all the simulations there is a force of 0.2 N applied along each

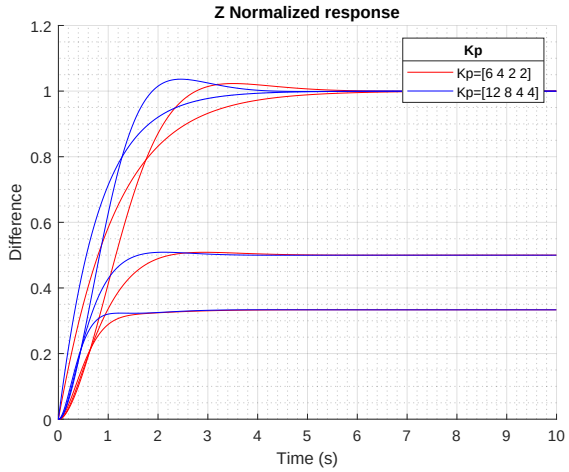


Figure 4.3: The normalized error can be seen according to the two simulations made with different scaling of K_p.

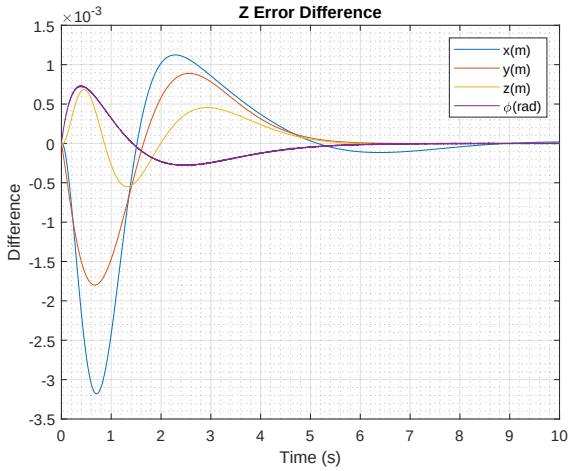


Figure 4.4: The comparison of all the simulations done for each the axes done independently showing the absolute error.

axis x, y, z and a torque of 0.2 N m is applied around the x axis. In the second simulation test, a simulation with the force only applied along one DOF at a time was done, to see if the behavior along a DOF is independent of the other axes.

All the simulations are made with the models and simulations parameters given from the paper seen in source [9]. The model taken from the paper is used with the mobile platform locked to rotate around the x axis.

The controller delay simulation was made in order to test how the system would react to a delay in the controller. The delay can occur since there is a CAN bus used in the system, and there exits limitations, for more on this see **Section 8.1**. With the simulations is it desired to measure the step response in all dimensions. This is done to see if it follows the designed response. Reasons for it not to follow the designed response could be because of the approximations that have been made in the control part **Section 4**. The **Figure 10.3** shows the simulations from the K_p scaling test.

Figure 9.6 shows the simulations from the axes independent test.

The graph on **Figure 12.4** shows the simulations from the position independent test.

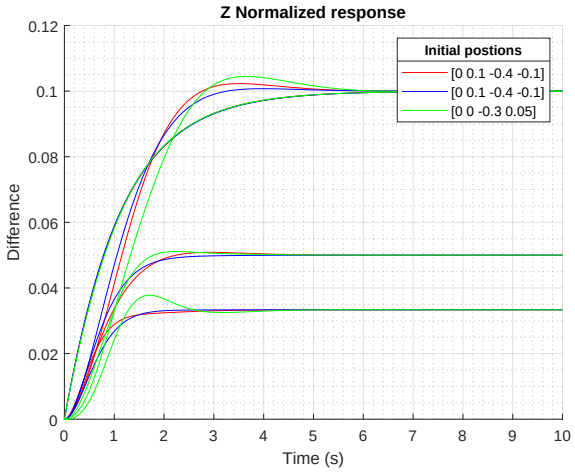


Figure 4.5: The normalized error seen from all the simulations with different initial positions.

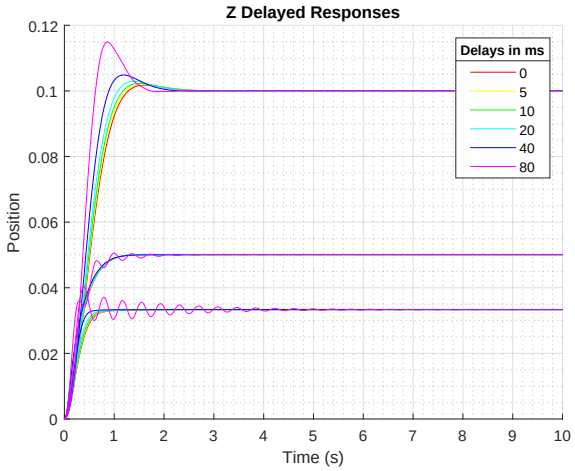


Figure 4.6: The comparison of all the other delay simulations made showing the absolute error.

The last simulations for delays in the controller can be seen on the **Figure 11.7**.

The simulations from the different tests were a success. They show how the controller's response is independent of the other axes and the position. It is also capable of handling delays much larger than what is expected to be in the system once implemented, without becoming unstable. Also, the parameter K_p scales response of the system as expected. The small deviations are expected due to the omission of the acceleration effects in the controller.

Chapter 5

Design of Code to Control Motors via CAN Bus

The code used to control the motors through a CAN bus network, specifically CANopen, using the Teensy 3.6 microcontroller, has been adapted from the code provided by Juan de Dios Flores-Mendez.

The original code was designed for a delta robot with three actuators, and in addition, it was limited to the movement of a single arm. Given these restrictions the changes in the code have focused on extending the existing functions and variables to be used by the four arms simultaneously. In this first code that will be explained below, the control that is carried out on the motors is a torque control in which the four arms are kept in the position in which they are set. A PID control is used in which the constants Kp, Kd and Ki can be modified both together for all arm or individually, thus allowing to observe the different behaviors depending on their value.

It should be noticed that this first code has been used to get acquainted with the communication with the motors via CAN bus, control using a PID controller, it is not the method that will be finally used.

For the correct functioning of this code the use of certain libraries is necessary. It will not be explained in depth each of them, however, the utility of the most important ones should be mentioned:

- SMC66Registers: It is a dictionary that contains variables related to the motor registers.
- CANsmc: Library developed by Juan de Dios Flores-Mendez for the communication with the motors. It contains functions that allow reading and writing the torque applied to each motor, the position of the encoder or the velocity among many other functions.
- FlexCAN: Another library to communicate via CAN bus.
- Chrono: It is a timing library. It is used for timing control and can be used to work on different time scales: microseconds, milliseconds or seconds.

Once the main libraries have been commented, it will be explained in more detail how the communication with the actuators is carried out via the CANopen.

In the CANsmc library we can find a series of functions called "writeToRegister" which generate the message to be sent to the motors depending on the arguments they receive. It should be noted that all generate the same type of message, the only thing that varies from one to another are the parameters with which they are called. The message is made up of a total of 104 unsigned int, starting with the identifier of the node (which of motors you want to send the message), followed by the number of the register to which you have to write, as well as other necessary data (torque that you want to provide, etc). Two examples of different ways to call the function can be seen in listing 1

```
// General function to write to register format uint32_t
void  CANsmc::writeToRegister(uint8_t nodeid, uint8_t subindex, uint32_t datas) {
  uint32_t id = nodeid + WRITE_REQUEST_CAN;
  uint8_t len = 8;
  uint8_t d0 = CANWRITE_4BYTE;
  uint8_t d1 = lowByte(object_index_32); //register number for 4 bytes
  uint8_t d2 = highByte(object_index_32);
  uint8_t d3 = subindex;

  uint8_t d4 = datas & 0xFF;
  uint8_t d5 = (datas >> 8) & 0xFF;
  uint8_t d6 = (datas >> 16) & 0xFF;
  uint8_t d7 = (datas >> 24) & 0xFF; //MSB , little endian format
  wrMsg(id, len, d0, d1, d2, d3, d4, d5, d6, d7);
}

//-------------------------------------------------
// General function to write to register format int32_t
void CANsmc::writeToRegister(uint8_t nodeid, uint8_t subindex, int32_t datas) {
  uint32_t id = nodeid + WRITE_REQUEST_CAN;
  uint8_t len = 8;
  uint8_t d0 = CANWRITE_4BYTE;
  uint8_t d1 = lowByte(object_index_32); //register number for 4 bytes
  uint8_t d2 = highByte(object_index_32);
  uint8_t d3 = subindex;
  uint8_t d4 = datas & 0xFF;
  uint8_t d5 = (datas >> 8) & 0xFF;
  uint8_t d6 = (datas >> 16) & 0xFF;
  uint8_t d7 = (datas >> 24) & 0xFF; //MSB , little endian format
  wrMsg(id, len, d0, d1, d2, d3, d4, d5, d6, d7);
}
```

Listing 1: Two different calls to the same function.

As can be seen in **Section 3.1** the actuators used in this project are the PD4-C motors by Nanotec. The reason for using these actuators is because they allow a torque control in closed loop. They work with commands in hexadecimal. To activate the torque control a "4" has to be written in 6060_h (Modes Of Operation). In this mode certain bits have a special function in objects 6040_h (controlword) and 6041_h (statusword). By reading this bit very useful information for the control can be obtained as can be seen in **Figure 5.1**. The specifications of these two special words (controlword and statusword) can be seen in **Figure 5.2** and **5.3** respectively. The values that can be changed in this mode are:

- 6071_h (Target Torque): Set the desired torque
- 6072_h (Max Torque): Maximum torque allowed.
- 6074_h (Torque Demand): Current output value for the controller.
- 6087_h (Torque Slope): Maximum change in the torque applied.

These values can be seen in the **Figure 5.4**.

6040_h Bit 8	6041_h Bit 10	Description
0	0	Specified torque not reached
0	1	Specified torque reached
1	0	Axis accelerated
1	1	Axis speed is 0

Figure 5.1: Special bits in controlword and statusword.

Index	6040_h
Object name	Controlword
Object Code	VARIABLE
Data type	UNSIGNED16
Savable	yes, category: application
Access	read / write
PDO mapping	RX-PDO
Allowed values	
Preset value	0000_h

Figure 5.2: Controlword specifications.

Index	6041_h
Object name	Statusword
Object Code	VARIABLE
Data type	UNSIGNED16
Savable	no
Access	read only
PDO mapping	TX-PDO
Allowed values	
Preset value	0000_h

Figure 5.3: Statusword specifications.

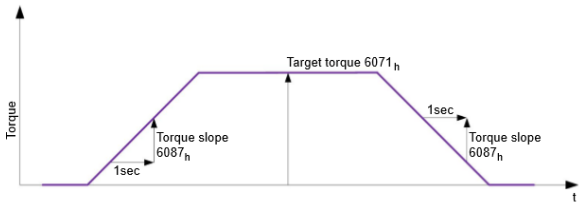


Figure 5.4: Torque curve.

Once the code has been started in the main setup function, the Teensy pins for the CAN bus communication are initialized. Additionally, the function "setTorqueSettings" is called, which makes all the necessary modifications to be able to drive the motors through a torque control, adjusting the maximum torque, torque slope, etc.

In the loop function is where the control is carried out.

As it can be seen in the **Figure 5.5** this function begins by checking if something has been written to the terminal (a brief explanation will be made later of the commands that can be executed through the console). After 100 ms, a series of useful variables for debugging are printed on the console, such as: the force that is printed on each motor, the constants of the PID controller (kp, kd, ki), errors in position and speed, etc.

The value of the position in which it is located is then read from each actuator and the position and speed errors are calculated. For computing the second one, a timer that resets at the beginning of

each cycle is used. In addition, if the integral action is activated, it is also calculated.

Finally, the force to be written on each motor is calculated using (5.1), where the errors and variables described above are used, the sign of this force is checked and compared with a maximum applicable value. Finally, the original value of the applied force is written on each actuator, or the default maximum is missing, with the lowest value being chosen.

$$\text{Foutput}_i = \text{kp}_i \cdot \text{error}_i + \text{kd}_i \cdot \text{derror}_i + \text{ki}_i \cdot \text{ierror}_i \tag{5.1}$$

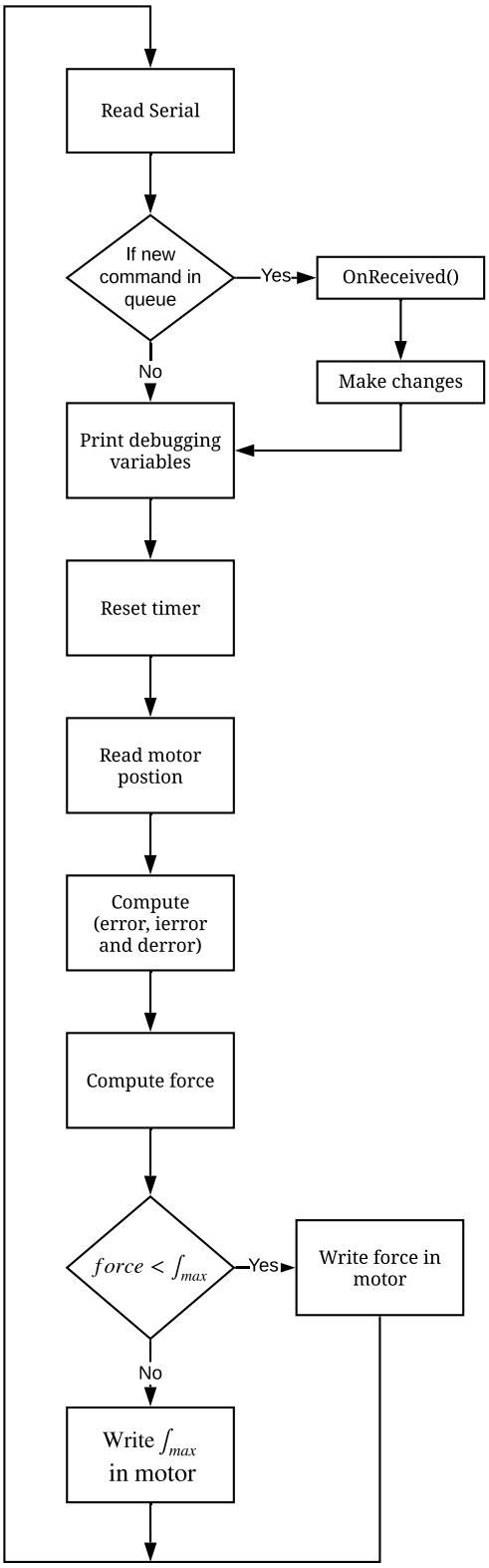


Figure 5.5: Diagram of the main loop function.

For the total understanding of how this code works, some secondary functions need to be briefly explained. Among the most important we have:

- *OnReceived*(): This function handles the messages written in the terminal. It is made of a series of if and $elseif$ similar to how a switch statement would work. This function compares what is written in the terminal with predetermined strings, and certain actions are performed depending on the written message. Some of the main commands are:
  - "sync": To synchronize all the nodes.
  - "ef": Exit fault. At the beginning all the actuators are in fault mode, to avoid problems.
  - "s"/"st": stop/start the control.
  - "intc"/"ni" : activate/deactivate the integral action of the PID.
  - "kp"/"kd"/"ki": change the value of the constants of the PID.
  - "tset": Set the torque settings (explained below).
- *SetTorque*(): Allows to set manually the torque for each arm.
- *SetTorqueSettings*(): This function is called in the setup(), and make all relevant changes to the motors so that they can operate with torque control, adjusting the maximum allowed torque, the torque slope (how fast can it change), etc.

5.1 Distributed Real Time Systems

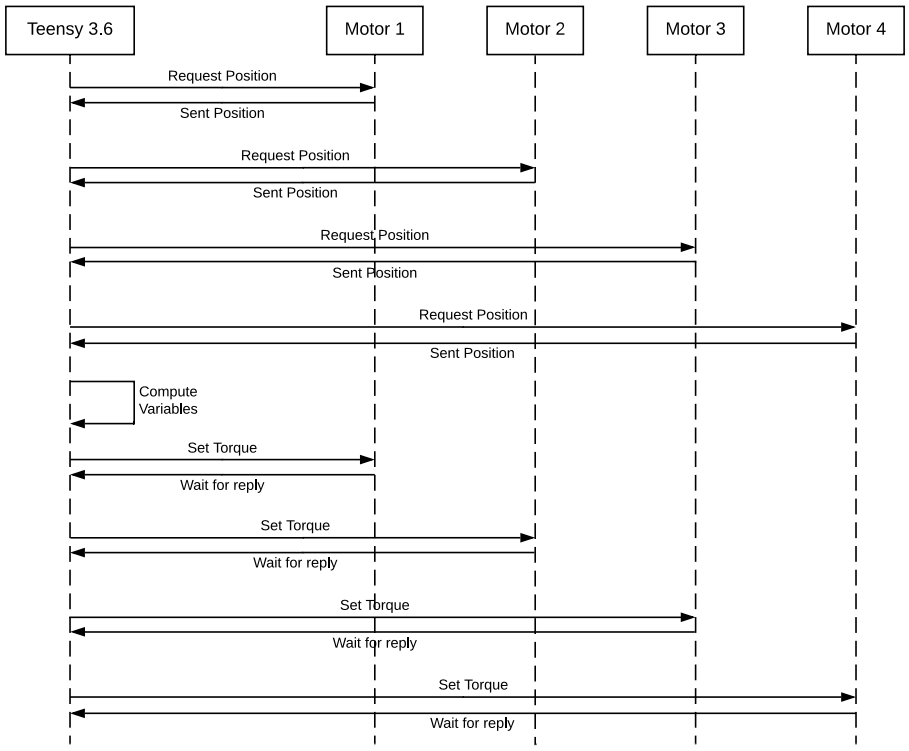


Figure 5.6: CAN bus communication between the Teensy 3.6 micro-controller and the four PD4-C actuators.

The system consists of the Teensy 3.6 micro-controller and the four PD4-C motors. These communicate through a CAN bus network as can be seen in **Figure 5.6**. Since messages are sent sequentially to

each of the actuators, there must be some delay between each message, to avoid packet loss.
The communication is placed in the loop function, therefore the messages that can be seen in the **Figure 5.6** are sent in each cycle. All the messages have the same size, $n_b = 104bits$, and there is a total of 8 packages to request the position and 8 packages to set the torque in the actuators, thus the total number of packages is $n_p = 16$. With these values and the next equation:

$$T_d = \frac{n_p \cdot n_b}{u} \tag{5.2}$$

The total time required to read the position and set the torque to all the actuators is $T_d = 1.664ms$. To avoid the package lost a delay of 0.3 ms is placed before sending the position, adding a total of 1.2 ms for all the actuators. Therefore, for receiving all the positions without losing any package 2.864 ms are needed. Before setting the torque to the actuators, a reply is requested, and this delay has not been measured.
This is the minimum value required to not lose any package. As can be seen in 13, the loop function takes around 7 ms to cycle, which means that the code could be optimized to be more efficient, reducing the delays set in some of the functions called. This time could also be increased up to 40 ms without significantly decreasing the performance of the system as can be seen in Test Journal 11.

5.2 Implementing MATLAB Code in C++

In order to implement the controller to the Teensy 3.6 is it an advantage to be able to implement the MATLAB functions into C++ code for the Teensy 3.6 to build. In order to do this is the MATLAB app "Coder" used. The app can generate C or C++ code from MATLAB scripts which then can be used in a variety of hardware platforms [14]. With this is it possible to generate C++ code for the Teensy 3.6 that can-do matrix calculations, which is needed for the control loop as all the equations used are with matrices. Not only does the "MATLAB Coder" generate code but it is also optimizing the code so there are not any redundant calculations.

5.2.1 How to use the App MATLAB Coder

To use the app MATLAB Coder is a MATLAB function needed to be made. The function can have multiple inputs and outputs without the need to be the same amount of inputs as outputs. In the function a MATLAB script can be written. Once a desired script has been written you can use the app. The app requires the user to define what the inputs are and what variable type should be used. The outputs will be determined from the inputs by the app itself. Lastly before the code is generated a check can be made to see if the MATLAB function can run normally and some test inputs can be used.

The code for this project is generated as source code to a device which is compatible with the Teensy 3.6 processor.

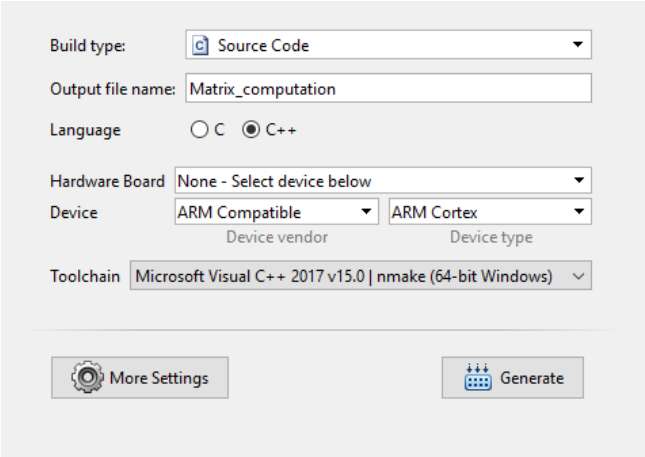


Figure 5.7: Here can the full coder settings that was used for this project be seen.

5.2.2 Implementing the generated code into the main code

The new code that now has been generated is sorted into libraries. This makes the implementation of the new code into the main code straightforward. There is no need to change the main code and the new function is simply just called the desired place with the desired inputs.

Chapter 6

Conclusion

This project sought out to find a solution for the labor intensive repetitive tasks at tomato industries. A system using compliant control was proposed in **Section 2.4** with a 3T1R platform attached giving the PKM another DOF.

The designed controller was successfully implemented for the simulations. Tests were conducted with the simulations in order to see its behavior. Based on the results from the simulations can it be concluded that it works consistently at different positions in the work space. The impedance control for the individual DOFs also behave independently of each other. The scaling of the K_v in relation to K_p and ζ results in consistent response and overshoot when changing the stiffness of the system.

As the theme of this project is Networked control systems, was the impact of the delay in the control simulated in a similar manner. The results show that the system can handle far more delay than is expected to occur without any major differences. In general, some of the simulations show small deviations in the results because of the acceleration term being omitted as it is negligible compared to the other terms. Although they are small and within the expected results.

Looking at the functional requirements in **Section 2.2** were there are multiple requirement set up for the project. The first, positional impedance control has been fulfilled in simulations. The second requirement states that the controller should be stable with the delay which is expected in the CAN bus. In the test journal **Section 13** the expected delay is between 6 to 8 milliseconds which is not enough to make the controller unstable according to the simulation made in **Section 11**. The third requirement, adjustable impedance parameters is also achieved. This is showed in the simulations in **Section 10**. However, the fourth and last requirement, path planning has not been fulfilled. This was due to the limited time. But could be implemented in the future as an additional controller on top of the impedance controller.

In conclusion an impedance controller has been designed for the simulated model. Most of the functional requirements are fulfilled in simulations, apart from the path planning and the controller has yet to be implemented on the real system.

Chapter 7

Discussion

7.1 Wrong Focus

A lot of time was used in the beginning on understanding and trying to derive the models for the kinematics and the dynamics. This was not intended as it was given to us. We were not supposed to be able to derive it ourselves but just understand the choices made in the process.

A lot of work was also put into implementing the system onto the Teensy 3.6, however this did not produce any results. The time should have been used on focusing on the networked control systems which is also the theme of the semester. A true time simulation of the controller should have been made.

7.2 No Implementation

No implementation has been done and cannot yet be done with the current method for calculating forward kinematics. Therefore, the tests of the full system cannot be done on an actual implementation of the system. This made a lot of work that was put into the code for the Ragnar Robot not usable at this point in the project.

7.3 Forward Kinematics not available for 3T1R

As mentioned in **Section 7.2** the forward kinematics have not been available for the 3T1R version of the Ragnar Robot. This cause multiple problems. The simulations were fragile and therefore not every input could be used since the simulation would sometimes simply crash. This made it very hard to get results from the simulations. The simulations for the test of delay in the controller had to be done using a different model for simulation without a rotation. This is also mentioned in the simulation for the delay test.

If the forward kinematics were obtained at a later date after hand in would the implementation of the system onto the Ragnar Robot be achievable.

7.4 Collaboration with Open Robotica

In the beginning of the semester a collaboration was planned with the company Open Robotica. On the early meetings of the semester a robot and a platform for the 3T1R was promised to be available alongside models of the system. This was supposed to make the implementation smoother. The robot and the models were however never handed over, and the models were not available until after the project was nearing its end. And therefore, were not for any help.

7.5 Future works

For future works after the hand in of the paper and the project, the next step will be obtaining the forward kinematics. Then the implementation of the system onto the Ragnar Robot can take place and tests can be done. The next thing of interest is to implement a path planner for the controller which would allow it to execute actual work. Which is the last requirement of the functional requirements in **Section 2.2**. If it is deemed necessary the mass and acceleration terms that were omitted for the controller can be added, which would result in a more predictable behavior. Additionally, the friction could be taken into consideration.

Chapter 8

Analysis

8.1 CAN Bus

A Controller Area Network (CAN bus) is a vehicle bus standard created with the purpose of a better and efficient communication between different devices without the need to use a host computer. Although it was originally designed to reduce car wiring, this message-based protocol is used in many other fields. Messages are transmitted sequentially, but respecting priority levels, and are received simultaneously by all the devices that make up the network.

8.2 History

Robert Bosch GmbH started the development of the CAN bus in 1983. The first CAN controller was produced by Intel and Philips, but it wasn't until 1991 when he first joined a production line with the Mercedes-Benz W140. In 1993, the International Organization for Standardization (ISO) created the standard ISO 11898, which was divided into two parts: ISO 11898-1, related with the data link layer; and ISO 11898-2, related with the physical layer for high speed. Later, the ISO 11898-3 was released for the physical layer of the low speed, fault tolerant CAN.

8.3 Architecture

CAN bus system is a multi-master serial bus, which means that is a process which sends data one bit at a time, sequentially. The network requires two or more devices, also known as nodes, to communicate. Two wire bus connect all the nodes to each other.
On one hand we have the high speed CAN, with a bit speed up to 1 MB/s. It uses an unique linear bus ended with two resistances of 120 Ω in both sides, which is the same as the nominal characteristic impedance of the bus. While the high speed CAN is transmitting a dominant (0), the high wire drives 5 V and the low wire drives 0 V. With this, the dominant differential voltage must be between 1.5 and 3 V. Otherwise, when it is transmitting a recessive (1) both wires drive the same value (between -2 and 7 V).
Using "0" as a dominant value gives the nodes with higher priority a lower ID number (This will be seen later).
On the other hand we have the low speed or fault tolerant CAN, with a speed up to 125 Kbps. This CAN can use a linear bus, a star bus or multiple star buses connected by a linear bus. In this case the total resistance has to be approximate to 100 Ω but not less.
Both in industrial and automobile environments the high speed CAN is usually used. Whereas the low speed CAN is used when you need to connect big amount of nodes together.
Each node can receive and send messages, but not at the same time. The different devices that can be found in a CAN network are: sensors, actuators, control devices, etc. and are connected to the bus through a host processor, a CAN controller, and a CAN transceiver.

8.4 Bit Synchronization

Every node must work at the same nominal bit rate. Bit timing is crucial during arbitration due each node should be capable to see the data transmitted from other nodes and also the one transmitted by itself.

8.5 Data Transmission

The method used to transmit data on a CAN bus network requires all nodes to be synchronized to interpret the message simultaneously. This can lead to error when thinking that it is a synchronous system, however, the CAN bus network does not use any clock to synchronize the nodes, so it is an asynchronous system.
By using dominant and recessive bits, the nodes transmit their messages. If it is the case that two nodes try to transmit at the same time, transmitting one of them a dominant bit (0) and the other one a recessive one (1), the messages will collide, and finally, the dominant one will be transmitted. The node that transmitted the recessive bit will try it again later. In this way the priorities are respected and the nodes with high priority will always transmit first.
This means that all nodes must be attentive to the messages they receive, so if they receive a dominant bit, while they try to send a recessive one, they must stop immediately to let the message with the highest priority be transmitted.
The first bits transmitted by each node correspond to its ID. During the arbitration stage, all the ID bits of the nodes that attempt to transmit are compared, and those with lower priorities are discarded, post posing their transmissions, and finally leaving a single node with the control.
There are four different types of CAN frame:

- Data frame
- Remote frame
- Error frame
- Overload frame

Data frame:
A data frame can be found in two different formats: base format, with an 11-bit identifier; and extended format, with 29-bit identifier

Remote frame:
This bit is found just after identification bits. By sending a recessive bit (1) in the remote frame a node can require information from other nodes.

Error frame:
When an erroneous message reaches a node, it is transmitted to the rest, which causes all nodes to transmit the error frame.

Overload frame:
This frame is used when one of the nodes is very busy, causing the bus to provide extra delays so that transmissions can be completed

Data frame and remote frame are separated for at least three recessive bits. If a dominant bit is detected before that is considered as a new frame. Error frame and overload frame don't respect this rule.
To avoid saturating the use of recessive or dominant bits and to ensure that there are sufficient transitions between the two, the so-called **bit stuffing** are used. These bits are inserted after five consecutive bits of the same level. If the receiver detects six or more consecutive bits of the same level, it issues an error.

8.6 CAN-based Protocols

Due to the shortcomings of the CAN system, such as flow control or the transport of data larger than more than one message among others, protocols based on the CAN system have been developed over the years. These protocols can be developed for use in a more specific way in certain fields. Below are some examples of CAN-based protocols that can be found in the field of industrial automation:

- CANopen
- DeviceNet
- SafetyBus p
- VSCP

Chapter 9

Simulation results: Axis independence response

9.1 Introduction

The test is made to show the impulse response of the controller. There are made five tests. Four tests show the impulse force applied to and individual axis at a time. And for comparison a fifth test is made that show the force impulse on all four DOFs.

9.2 Test Frame

The test have been conducted in a simulation. This also means that the results are not a ultimatum but more of an estimate of how the system would react in the implementation. The reason for the simulation to be done first is in order to secure that the system is stable and behaves as expected before the implementation, and the tests on the implementation.

Theoretical Background

The controller is supposed to have a consistent response independent of the position of the platform and independent of forces applied to other DOFs.

Test Setup and Test Procedure

The simulation is setup in Matlab using Runge-Kutta 4th order. The whole simulation is first calculated and then afterwards can it be seen on a recording of the movements of the links on the Ragnar Robot. The controller is implemented as described inC**chapter 4**.

The controller uses the parameters seen in **Table 9.1**. The impulse response is a force of N applied along each axis x, y, z and a force of N m is applied as a rotation force around the x-axis. The values of the force that is applied can be seen in the **Table 9.1**.

Test no.	Control parameter	Value	Force applied on the axes $[x, y, z, \gamma]$
1	K_p, K_v	$[12, 8, 4, 4]$ $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$	$[0; 0; 0; 0, 2]$
2	K_p, K_v	$[12, 8, 4, 4]$ $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$	$[0; 0; -0, 2; 0]$
3	K_p, K_v	$[12, 8, 4, 4]$ $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$	$[0; -0, 2; 0; 0]$
4	K_p, K_v	$[12, 8, 4, 4]$ $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$	$[0, 2; 0; 0; 0]$
5	K_p, K_v	$[12, 8, 4, 4]$ $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$	$[0, 2; -0, 2; -0, 2; 0, 2]$

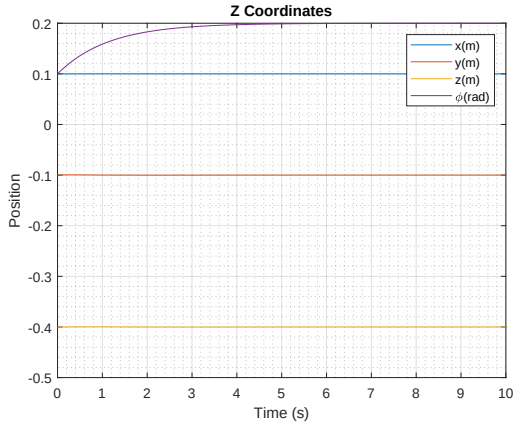
Table 9.1: The control parameters' values used for the test.

9.3 Test Results and Data Processing

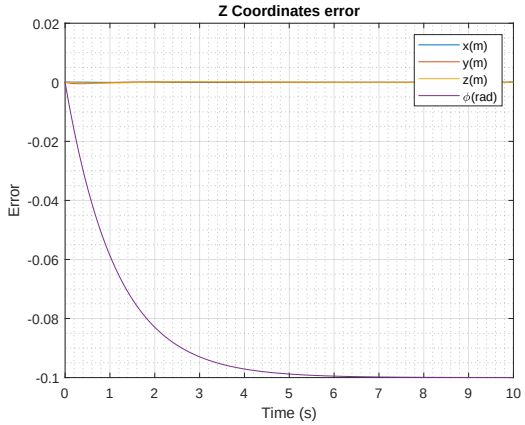
In the following section the test results can be seen in the corresponding subsection. The results consists of 3 graphs showing the actual position of the mobile platform, the error plot, and the absolute error.

Looking on the graphs for the tests can it be seen that the responses for each DOF are approximately independent of each other. The graph for the absolute error in test 4 **Figure 9.5(c)** have some movement on the z axis even though it only should move the x axis. This movement is however relatively small if looking at the small even though it look big, because the error axis is operation with lower values then all of the other graphs. Meaning that all of the tests are a success showing that the responses for each DOF are approximately independent of each other.

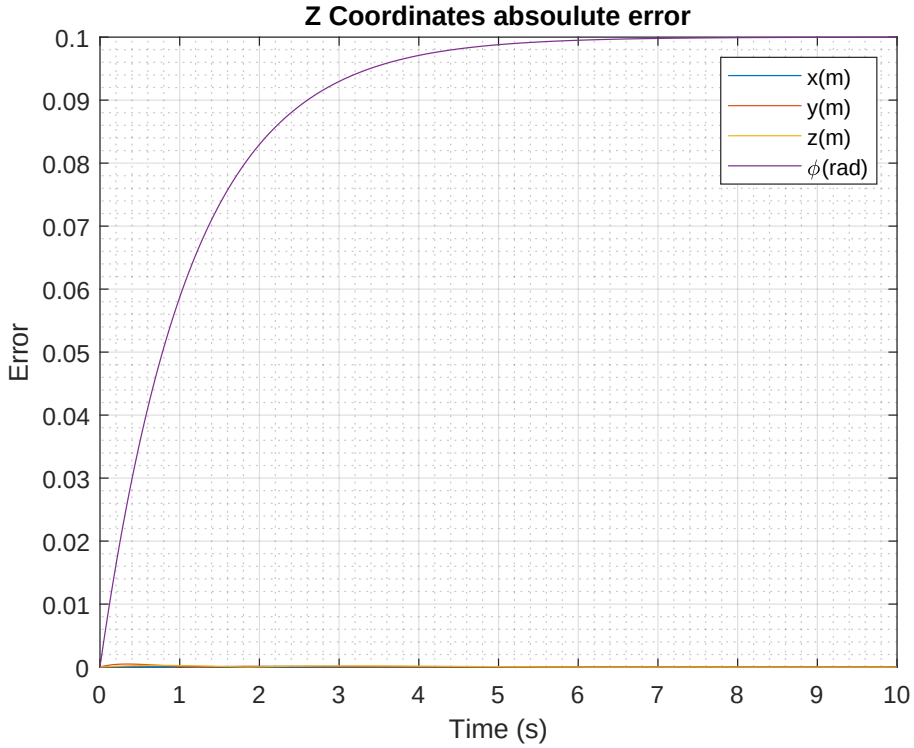
9.3.1 Test Results from Simulation 1



(a) The actual position.



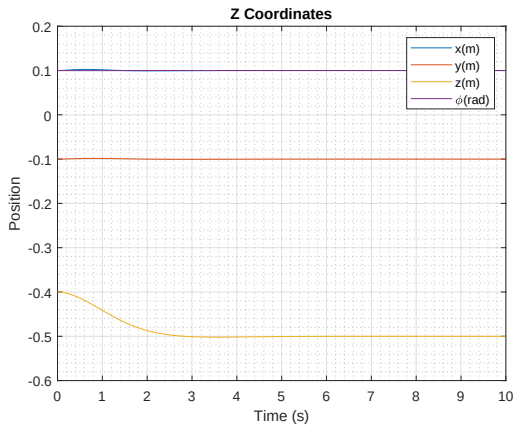
(b) The error plot.



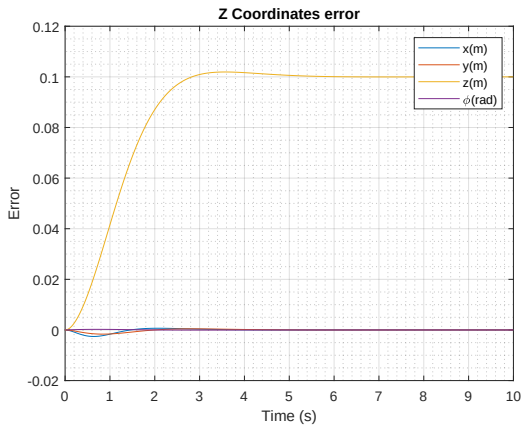
(c) The absolute error plot.

Figure 9.1: The graphs show the results from the first simulation with a force of 0.2 N applied on the γ rotation. The control parameters for K_p are $[12, 8, 4, 4]$ and for K_v $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$.

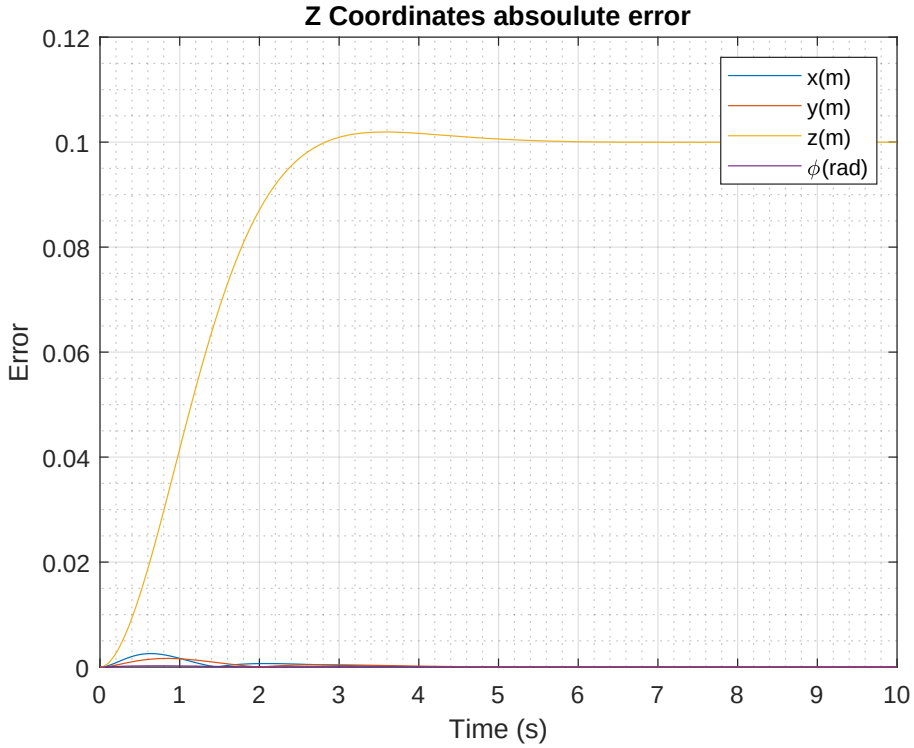
9.3.2 Test Results from Simulation 2



(a) The actual position.



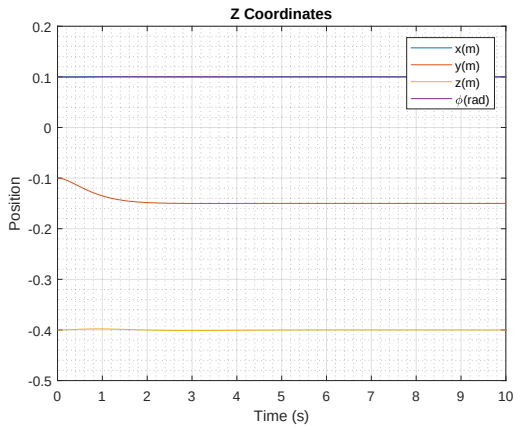
(b) The error plot.



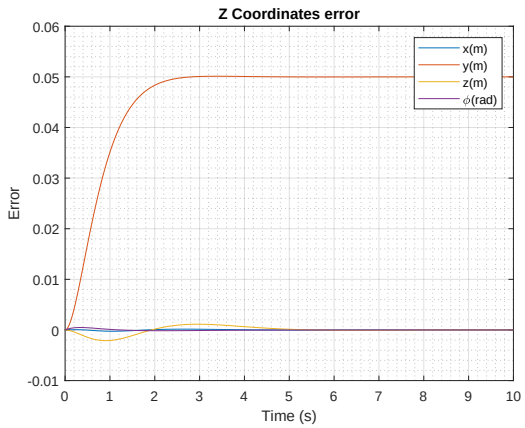
(c) The absolute error plot.

Figure 9.2: The graphs show the results from the second simulation with a force of 0.2 N applied on the z axis. The control parameters for K_p are $[12, 8, 4, 4]$ and for K_v $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$.

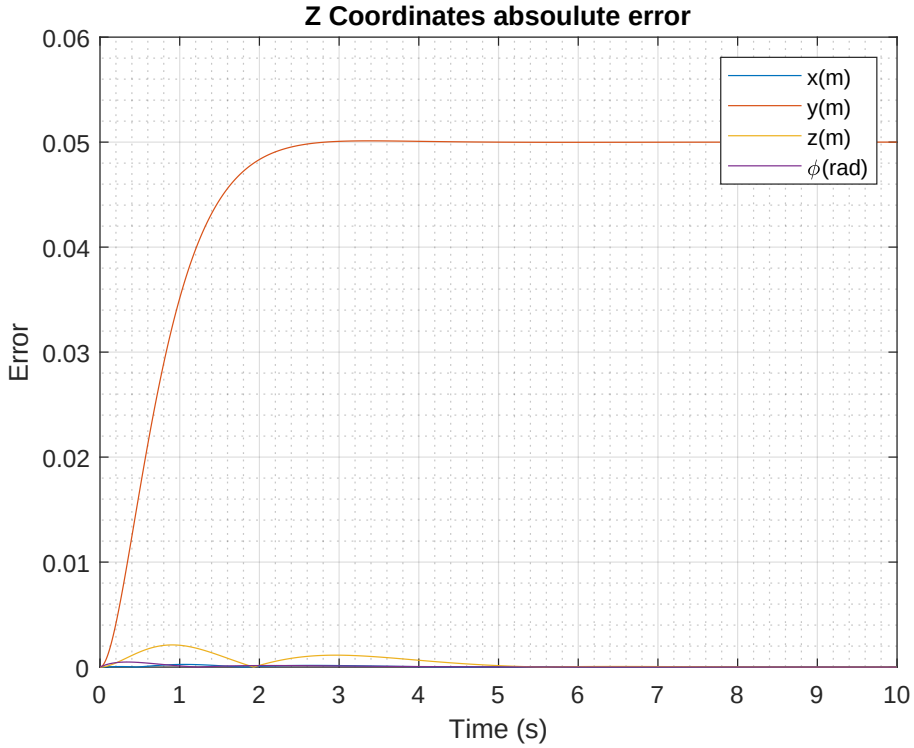
9.3.3 Test Results from Simulation 3



(a) The actual position.



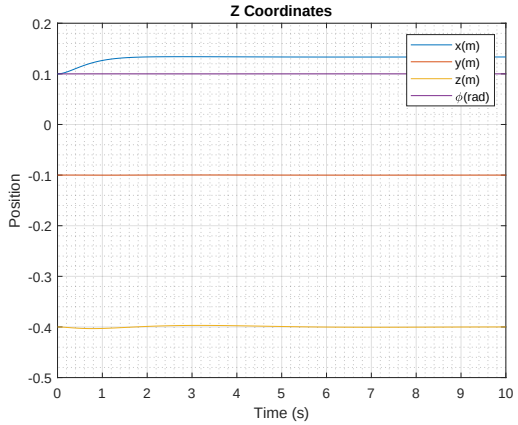
(b) The error plot.



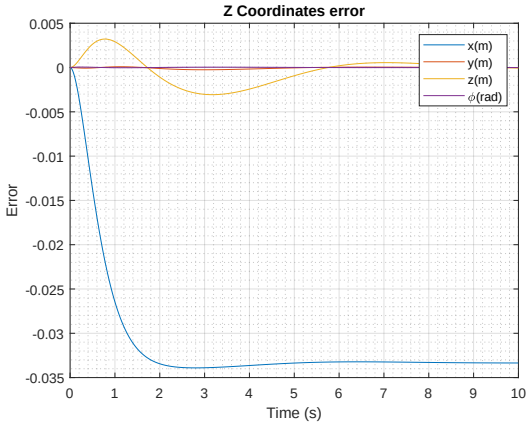
(c) The absolute error plot.

Figure 9.3: The graphs show the results from the third simulation with a force of 0.2 N applied on the y axis. The control parameters for K_p are $[12, 8, 4, 4]$ and for K_v $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0, 8$.

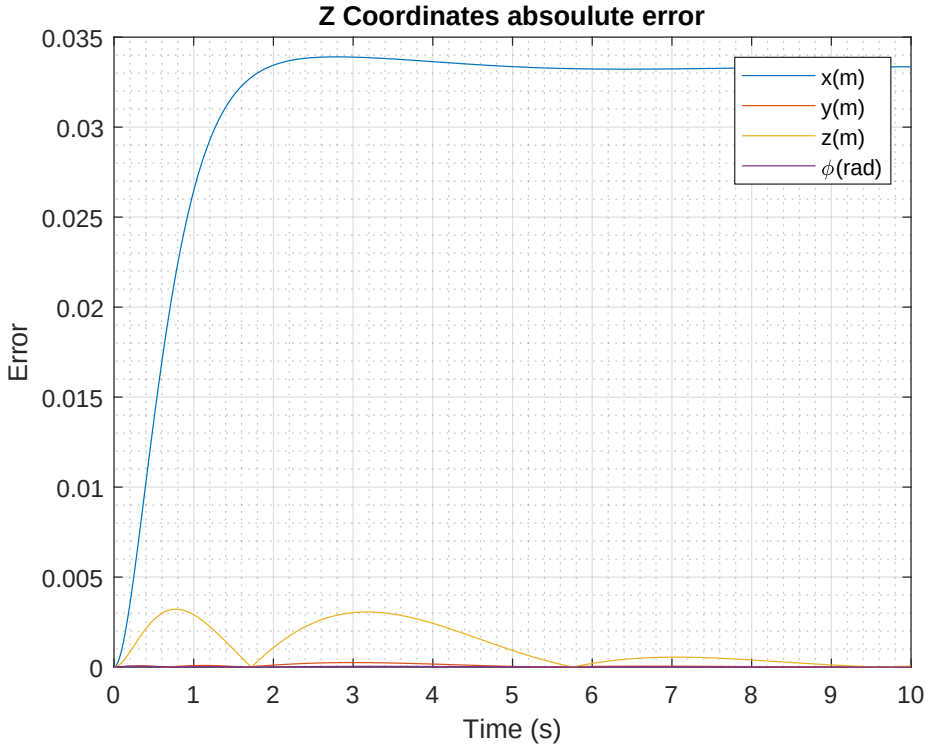
9.3.4 Test Results from Simulation 4



(a) The actual position.



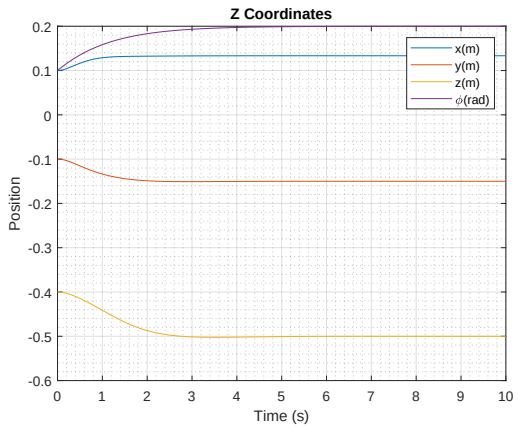
(b) The error plot.



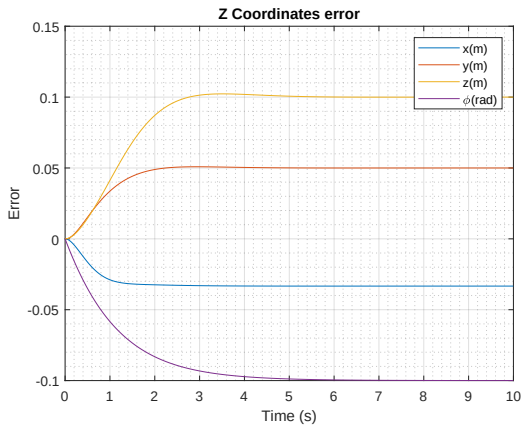
(c) The absolute error plot.

Figure 9.4: The graphs show the results from the fourth simulation with a force of 0.2 N applied on the x axis. The control parameters for K_p are $[12, 8, 4, 4]$ and for K_v $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$.

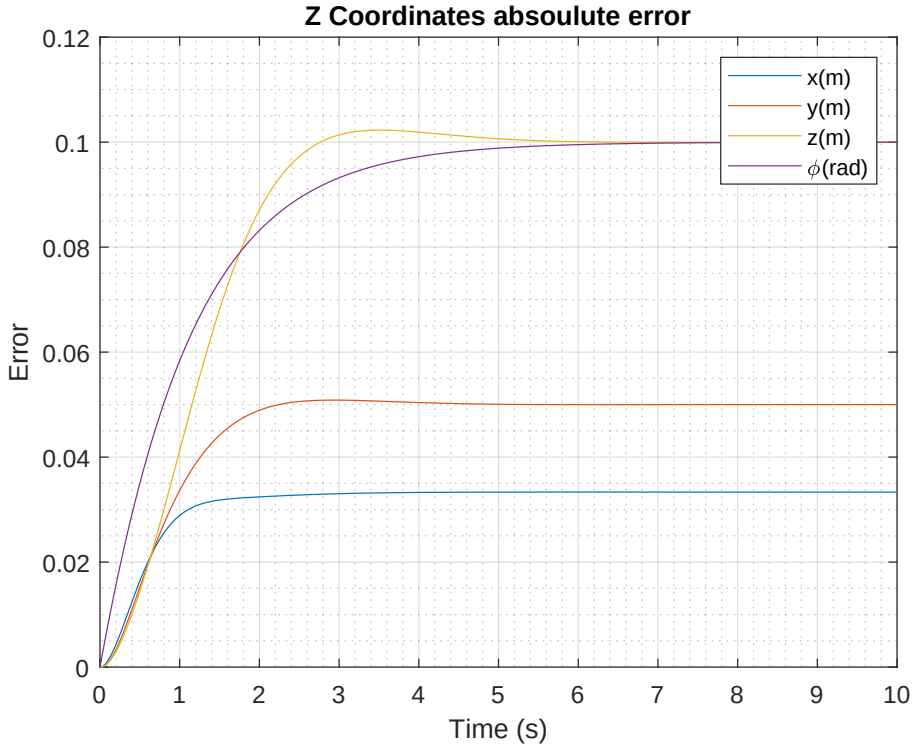
9.3.5 Test Results from Simulation 5



(a) The actual position.



(b) The error plot.



(c) The absolute error plot.

Figure 9.5: The graphs show the results from the fifth simulation with a force of 0.2 N applied on all of the axes. The control parameters for K_p are $[12, 8, 4, 4]$ and for K_v $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$.

9.3.6 Comparison of the Graphs from the Different Simulations

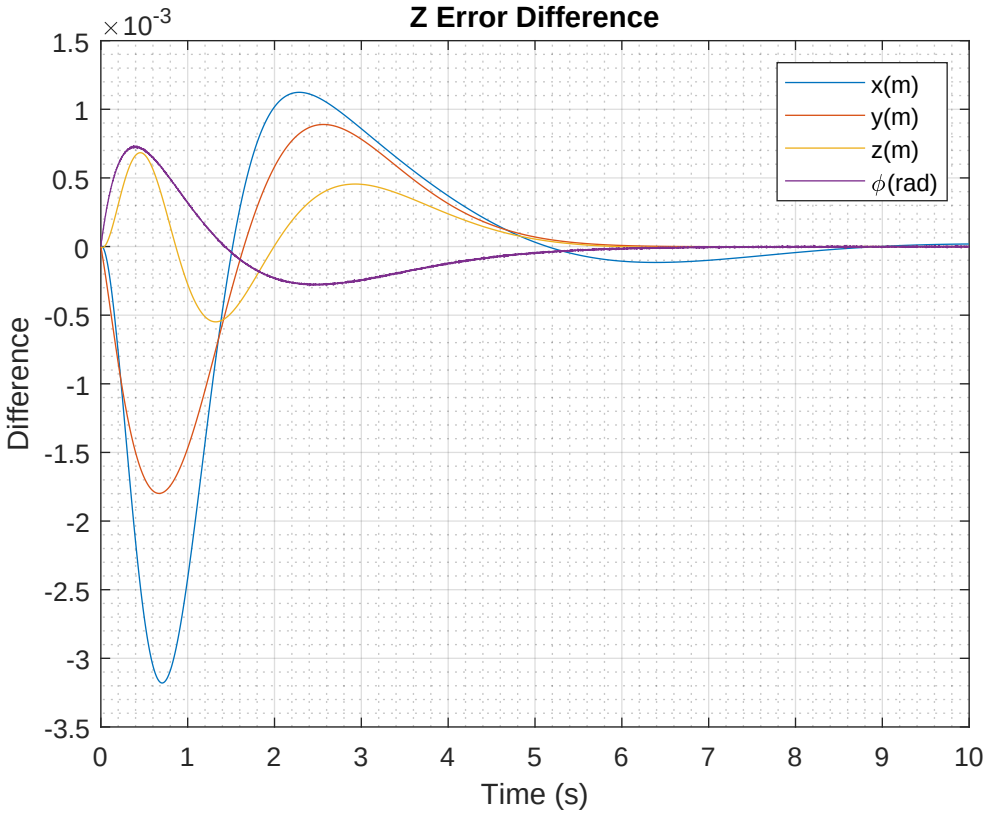


Figure 9.6: The comparison of the other graphs showing the absolute error from the different simulations. However the simulation which uses all axes is not included here.

Chapter 10

Simulation Results: Kp Scaling Test

10.1 Introduction

The test is made to show the impulse response of the controller using different scaling of the K_p control parameter with the same ζ.

10.2 Test Frame

The test have been conducted in a simulation. This also means that the results are not a ultimatum but more of an estimate of how the system would react in the implementation. The reason for the simulation to be done first is in order to secure that the system is stable and behaves as expected before the implementation, and the tests on the implementation.

Theoretical Background

The response is supposed to scale with the control parameter K_p. The shape of the impulse response should only be affected by ζ, therefore the responses should look similar with similar overshoot no matter the k_p

Test Setup and Test Procedure

The simulation is setup in Matlab using Runge-Kutta 4th order. The controller is implemented as described in **Chapter 4**.

The controller uses the parameters seen in **Table 10.1**. The impulse response is a force of 0.2 N applied along each axis x, y, z and a force of 0.2 N m is applied as a rotation force around x.

Test no.	Control parameter	Value	Position of the mobile platform $[x, y, z, \gamma]$
1	K_p, K_v	$[12, 8, 4, 4]$ $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$	$[0,1; -0,1; -0,4; 0,1]$
2	K_p, K_v	$[6, 4, 2, 2]$ $[\sqrt{6}, \sqrt{4}, \sqrt{2}, \sqrt{2}] \cdot 0,8$	$[0,1; -0,1; -0,4; 0,1]$

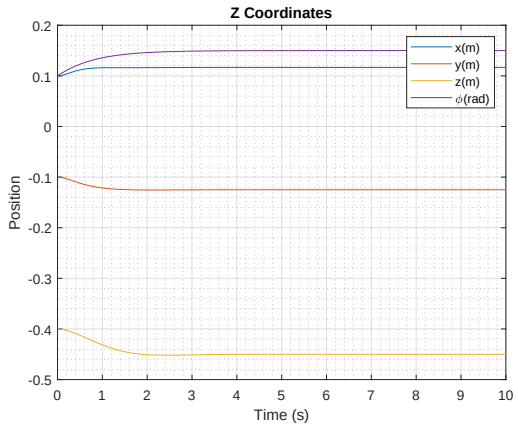
Table 10.1: The control parameters' values used for the test.

10.3 Test Results and Data Processing

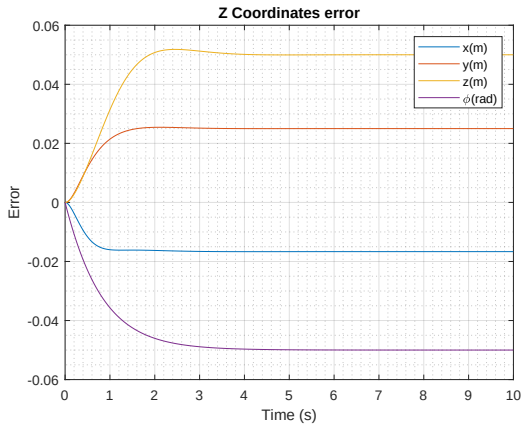
In the following section the test results can be seen in the corresponding subsection. The results consists of 3 graphs showing the actual position of the mobile platform, the error plot, and the absolute error.

Looking on the graphs for the tests can it be seen that the control system indeed scales linear according to the scaling of the control parameter K_p.

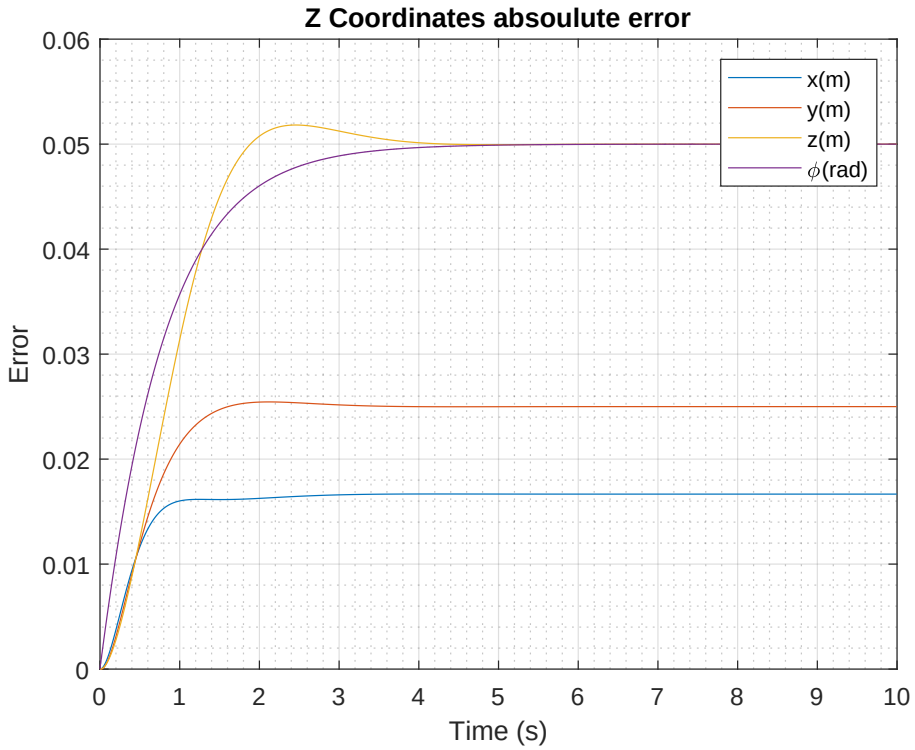
10.3.1 Test Results from Simulation 1



(a) The actual position.



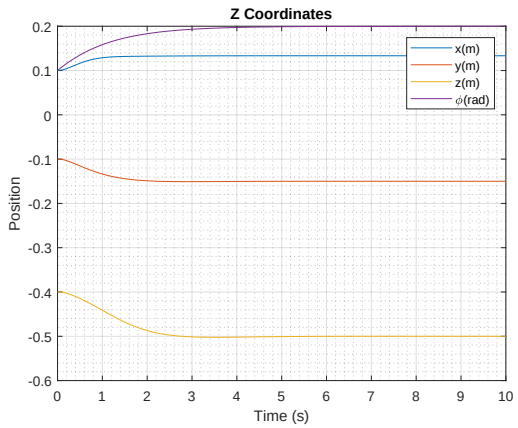
(b) The error plot.



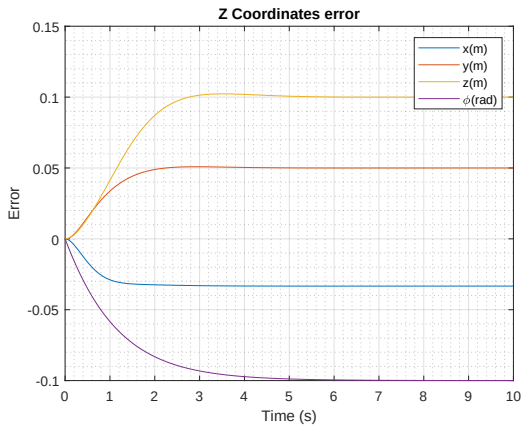
(c) The absolute error plot.

Figure 10.1: The graphs show the results from the first simulation with a force of 0.2 N applied on all of the axes and the rotation $[x, y, z, \gamma]$. The control parameters for K_p are $[12, 8, 4, 4]$ and for K_v $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0, 8$.

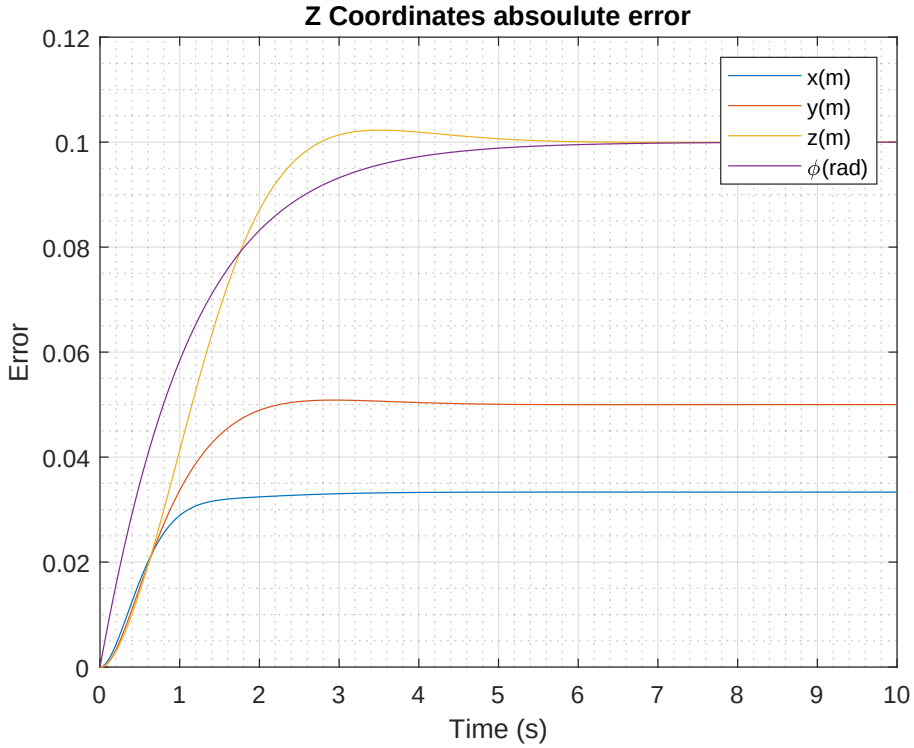
10.3.2 Test Results from Simulation 2



(a) The actual position.



(b) The error plot.



(c) The absolute error plot.

Figure 10.2: The graphs show the results from the second simulation with a force of 0.2 N applied on all of the axes and the rotation $[x, y, z, \gamma]$. The control parameters for K_p are $[6, 4, 2, 2]$ and for K_v $[\sqrt{6}, \sqrt{4}, \sqrt{2}, \sqrt{2}] \cdot 0,8$.

10.3.3 Comparison of the Graphs from the two Different Simulations

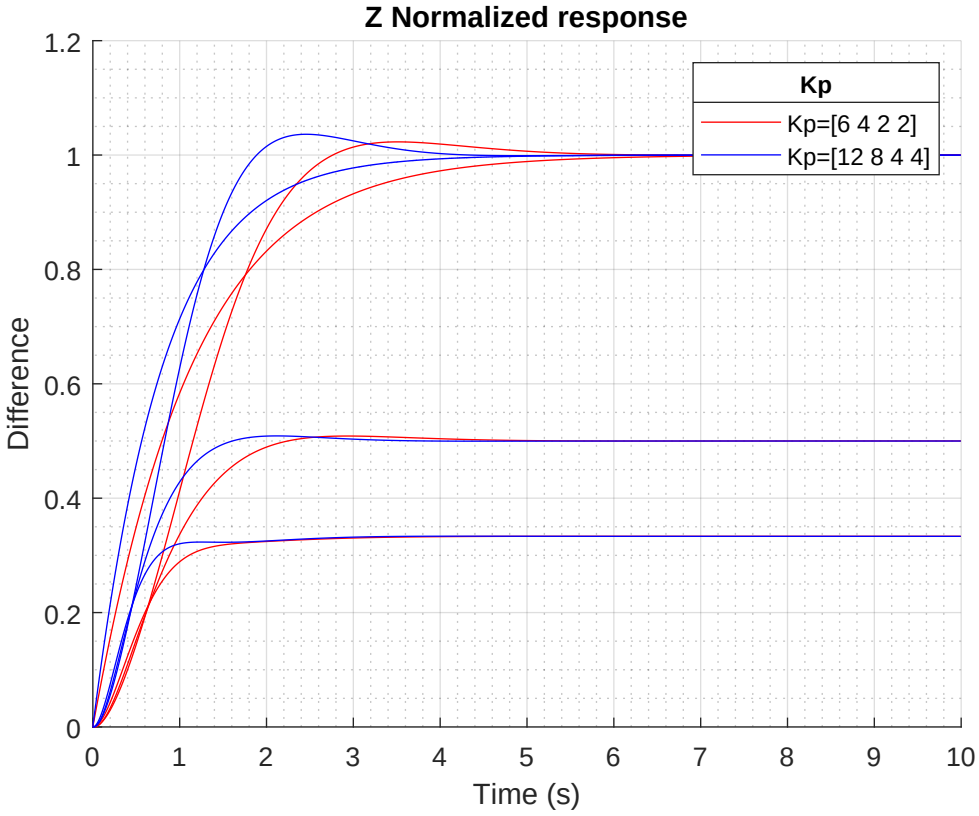


Figure 10.3: The comparison of the two graphs showing the normalized error from the simulations.

Chapter 11

Simulation results: Controller Delay

11.1 Introduction

The tests are made to show the impulse response of the controller using different delays.

11.2 Test Frame

The test has been conducted in a simulation. This also means that the results are not a ultimatum but more of an estimate of how the system would react in the implementation. The reason for the simulation to be done first is in order to secure that the system is stable and behaves as expected before the implementation, and the tests on the implementation.

Theoretical Background

The current controller is not built for delay. But the CAN bus can introduce a delay in the system **Section 8.1**, it is therefore of interest to test how the system behaves with a delay. The delay is implemented in the simulation such that is acts like a buffer which sends the position to the controller after a certain delay which can be seen in the **Table 11.1**.

Test Setup and Test Procedure

The simulation is setup in Matlab using Runge-Kutta 4th order. The controller is implemented as described in **Chapter 4**.

The controller uses the parameters seen in **Table 11.1**. The impulse response is a force of 1 N applied along each axis x, y, z. The delay used in the controller can also be seen in the table.

Test no.	Control parameter	Value	Position of the mobile platform $[x, y, z]$	Delay
1	K_p, K_v	$[30, 20, 10]$ $[\sqrt{30}, \sqrt{20}, \sqrt{10}] \cdot 0,8$	$[0,1; -0,1; -0,4]$	0 ms
2	K_p, K_v	$[30, 20, 10]$ $[\sqrt{30}, \sqrt{20}, \sqrt{10}] \cdot 0,8$	$[0,1; -0,1; -0,4]$	5 ms
3	K_p, K_v	$[30, 20, 10]$ $[\sqrt{30}, \sqrt{20}, \sqrt{10}] \cdot 0,8$	$[0,1; -0,1; -0,4]$	10 ms
4	K_p, K_v	$[30, 20, 10]$ $[\sqrt{30}, \sqrt{20}, \sqrt{10}] \cdot 0,8$	$[0,1; -0,1; -0,4]$	20 ms
5	K_p, K_v	$[30, 20, 10]$ $[\sqrt{30}, \sqrt{20}, \sqrt{10}] \cdot 0,8$	$[0,1; -0,1; -0,4]$	40 ms
6	K_p, K_v	$[30, 20, 10]$ $[\sqrt{30}, \sqrt{20}, \sqrt{10}] \cdot 0,8$	$[0,1; -0,1; -0,4]$	80 ms

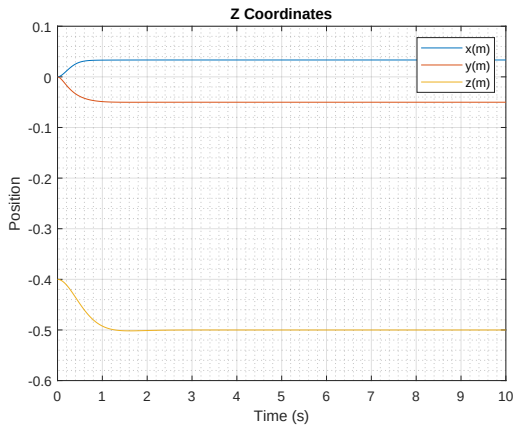
Table 11.1: The control parameters' values used for the test.

11.3 Test results and data processing

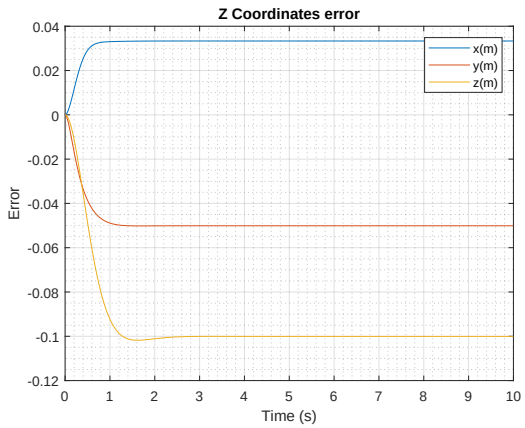
In the following section can the test results be seen in the corresponding subsection. The results consist of 3 graphs showing the actual position of the mobile platform, the error plot, and the absolute error.

Looking through the simulations the controller can handle quite a bit of delay without becoming unstable. Looking at the last graph **Figure 11.7** can it be seen how the system worsens with overshoot and oscillations as the delay increases.

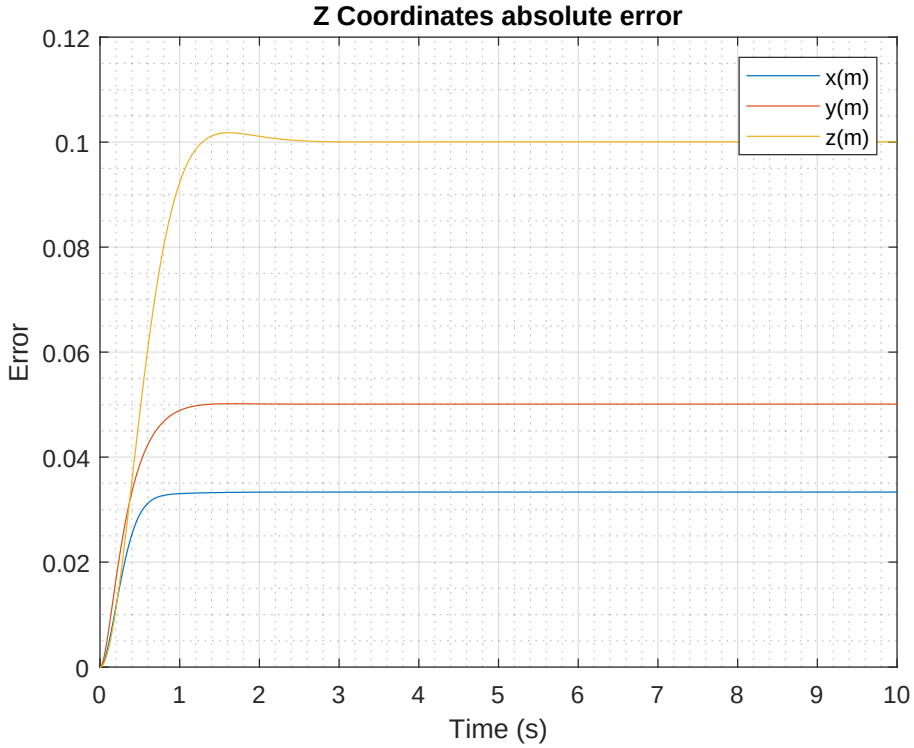
11.3.1 Test Results from Simulation 1



(a) The actual position.



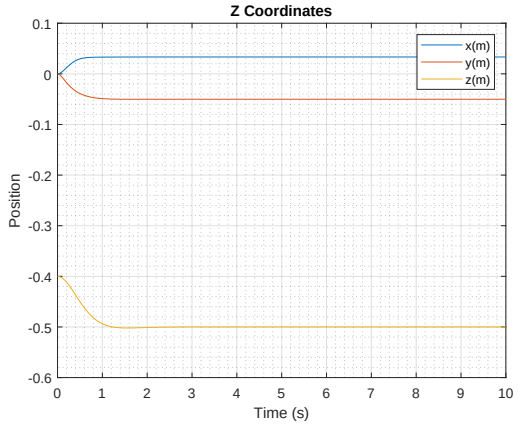
(b) The error plot.



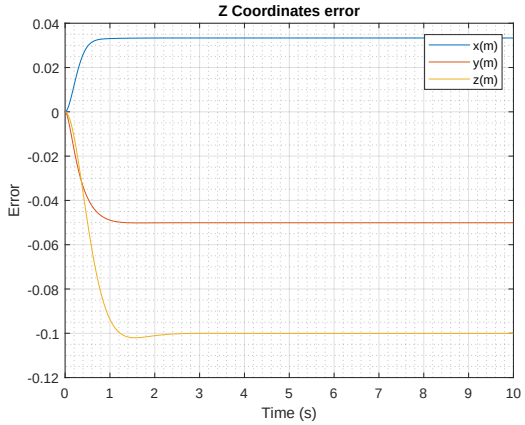
(c) The absolute error plot.

Figure 11.1: The graphs show the results from the first simulation with a force of 0.2 N applied on all of the axes $[x, y, z]$. The control parameters for K_p are $[30, 20, 10]$ and for K_v $[\sqrt{30}, \sqrt{20}, \sqrt{10}] \cdot 0,8$. The initial position of the mobile platform is: $[0.1; -0,1; -0,4]$ and the delay applied to the system is 0 ms.

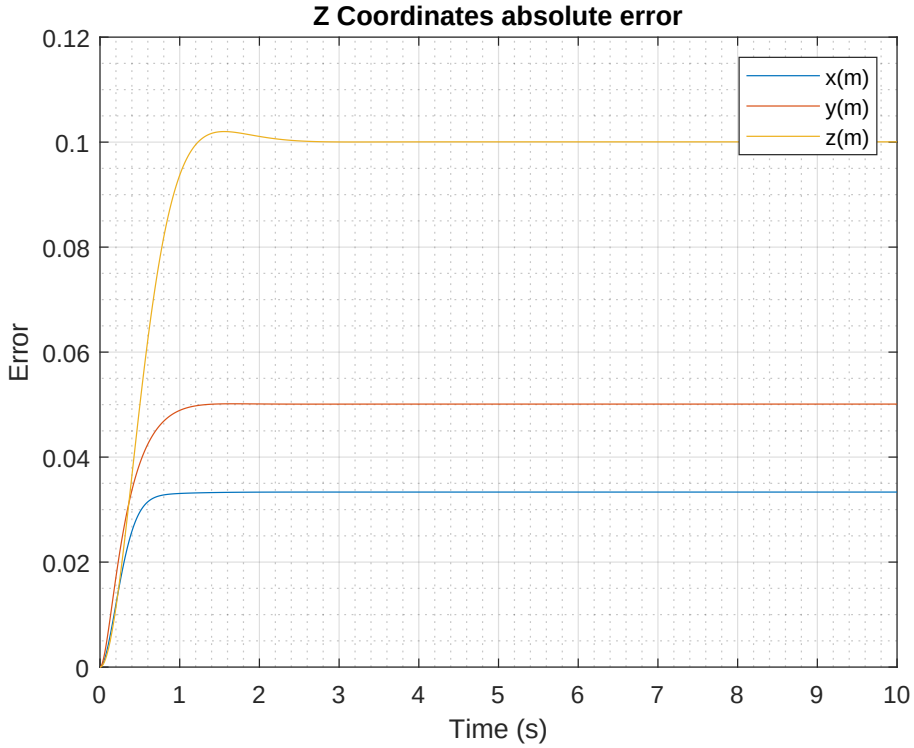
11.3.2 Test Results from Simulation 2



(a) The actual position.



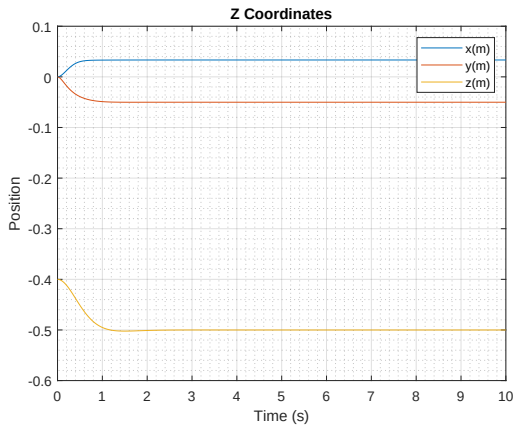
(b) The error plot.



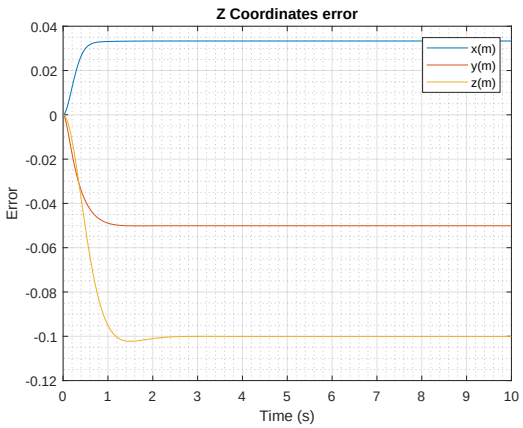
(c) The absolute error plot.

Figure 11.2: The graphs show the results from the second simulation with a force of 0.2 N applied on all of the axes $[x, y, z]$. The control parameters for K_p are $[30, 20, 10]$ and for K_v $[\sqrt{30}, \sqrt{20}, \sqrt{10}] \cdot 0,8$. The initial position of the mobile platform is: $[0.1; -0,1; -0,4]$ and the delay applied to the system is 5 ms.

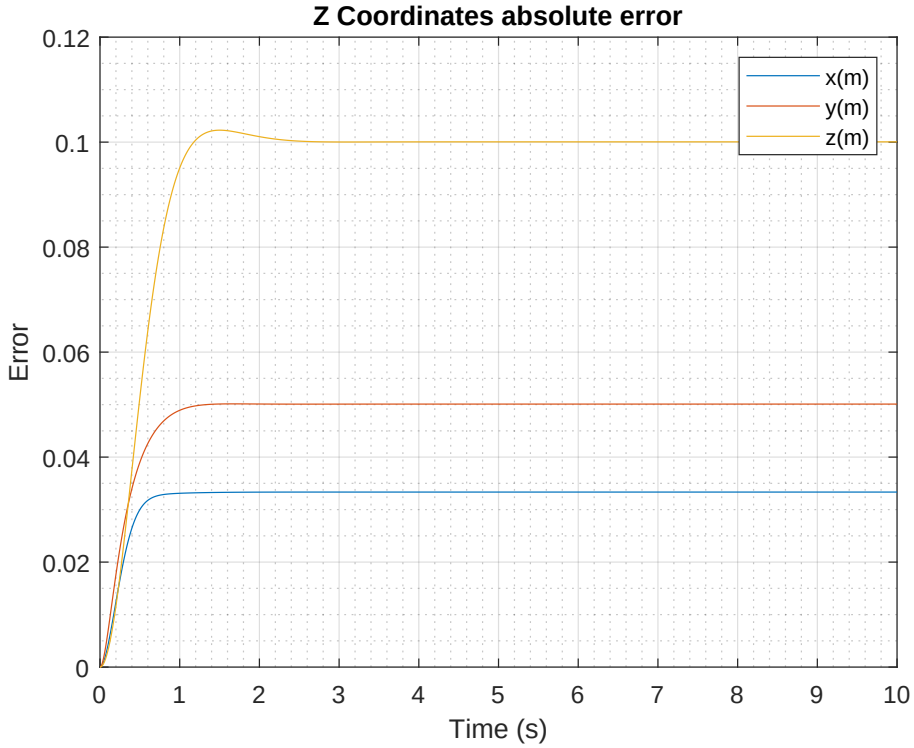
11.3.3 Test Results from Simulation 3



(a) The actual position.



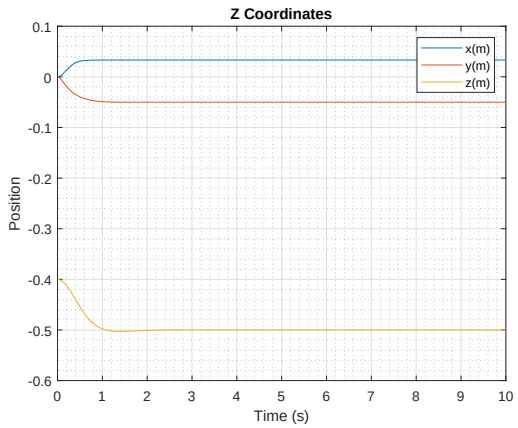
(b) The error plot.



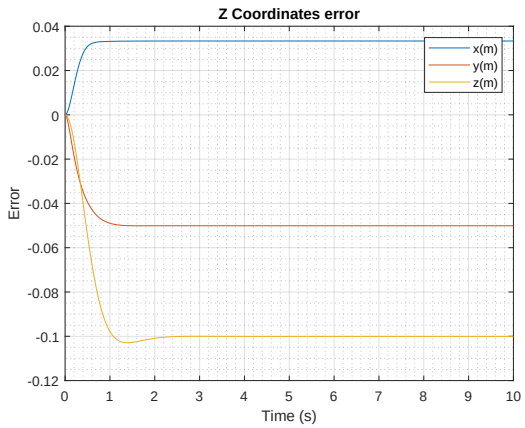
(c) The absolute error plot.

Figure 11.3: The graphs show the results from the third simulation with a force of 0.2 N applied on all of the axes $[x, y, z]$. The control parameters for K_p are $[30, 20, 10]$ and for K_v $[\sqrt{30}, \sqrt{20}, \sqrt{10}] \cdot 0,8$. The initial position of the mobile platform is: $[0.1; -0,1; -0,4]$ and the delay applied to the system is 10 ms

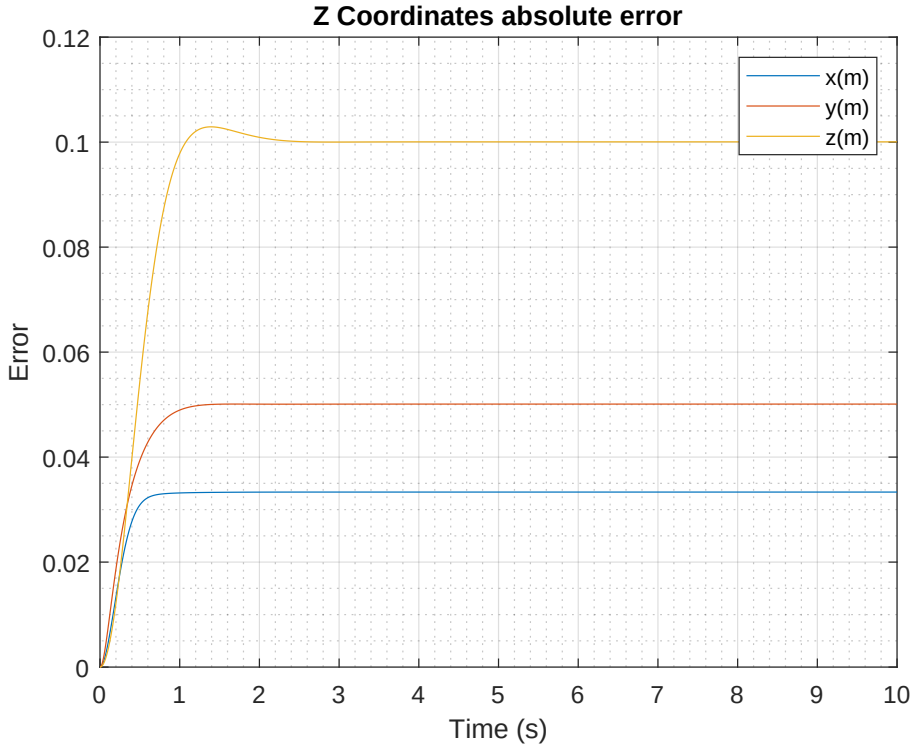
11.3.4 Test Results from Simulation 4



(a) The actual position.



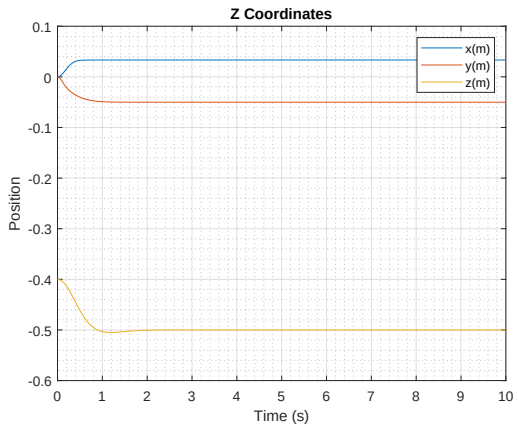
(b) The error plot.



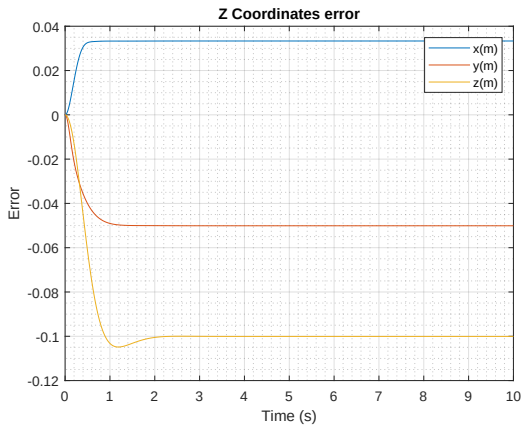
(c) The absolute error plot.

Figure 11.4: The graphs show the results from the fourth simulation with a force of 0.2 N applied on all of the axes $[x, y, z]$. The control parameters for K_p are $[30, 20, 10]$ and for K_v $[\sqrt{30}, \sqrt{20}, \sqrt{10}] \cdot 0,8$. The initial position of the mobile platform is: $[0.1; -0,1; -0,4]$ and the delay applied to the system is 20 ms.

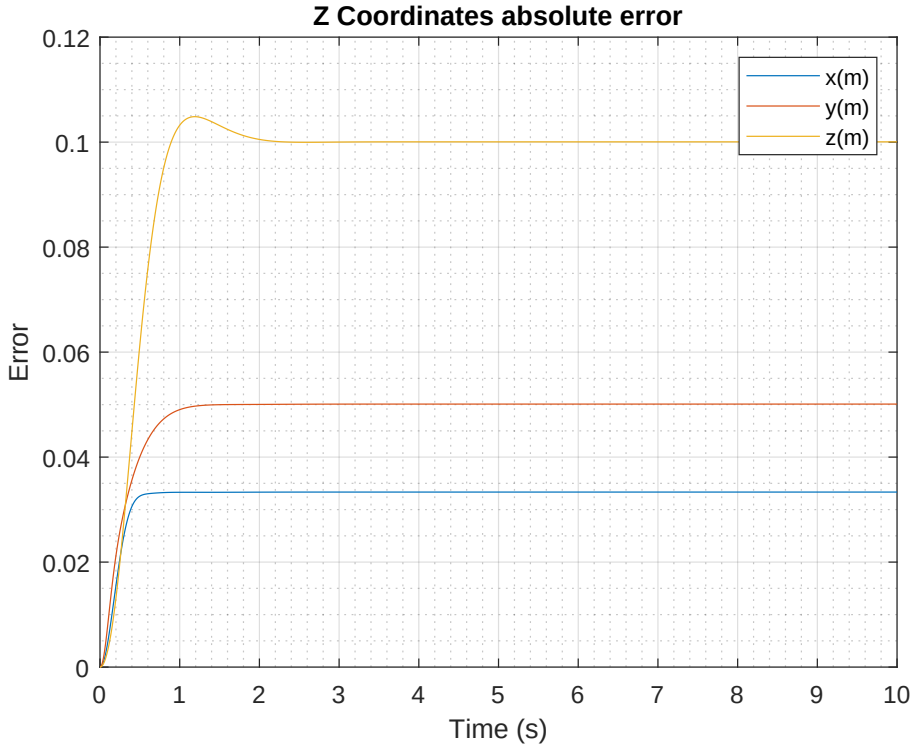
11.3.5 Test Results from Simulation 5



(a) The actual position.



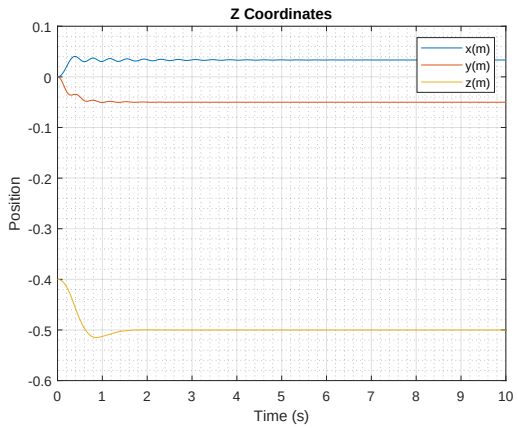
(b) The error plot.



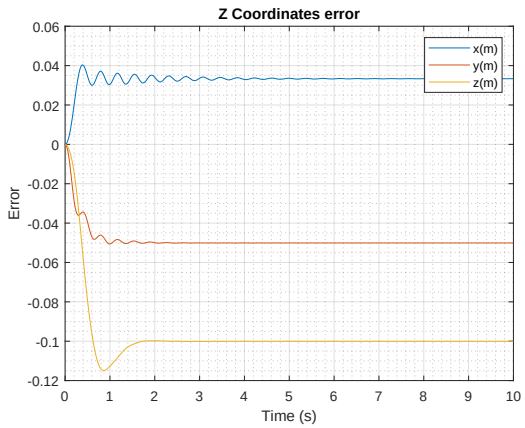
(c) The absolute error plot.

Figure 11.5: The graphs show the results from the fifth simulation with a force of 0.2 N applied on all of the axes $[x, y, z]$. The control parameters for K_p are $[30, 20, 10]$ and for K_v $[\sqrt{30}, \sqrt{20}, \sqrt{10}] \cdot 0,8$. The initial position of the mobile platform is: $[0.1; -0,1; -0,4]$ and the delay applied to the system is 40 ms.

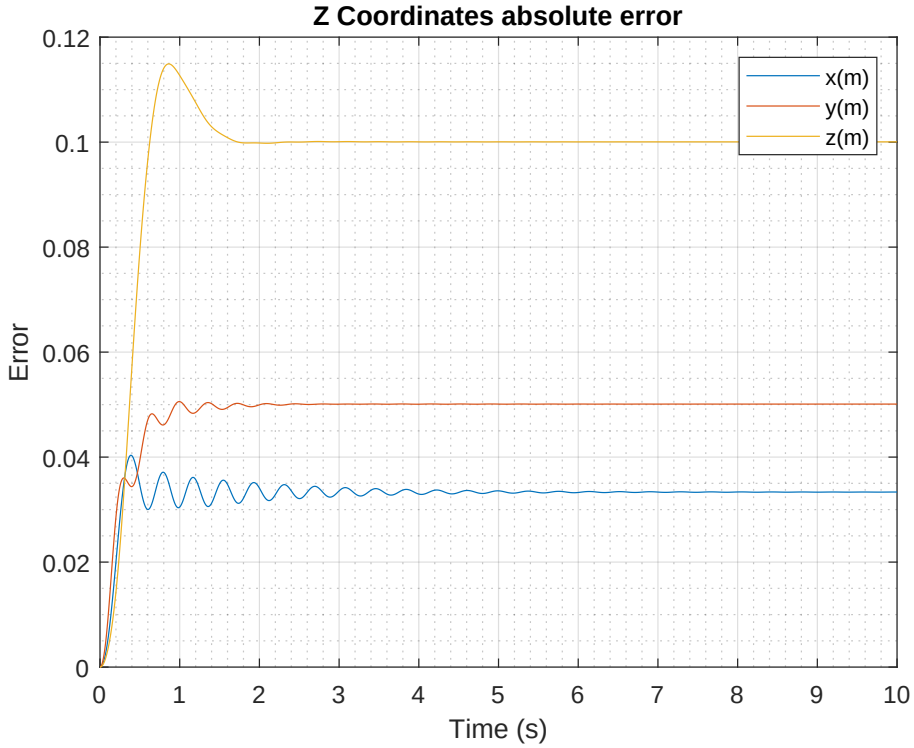
11.3.6 Test Results from Simulation 6



(a) The actual position.



(b) The error plot.



(c) The absolute error plot.

Figure 11.6: The graphs show the results from the sixth simulation with a force of 0.2 N applied on all of the axes $[x, y, z]$. The control parameters for K_p are $[30, 20, 10]$ and for K_v $[\sqrt{30}, \sqrt{20}, \sqrt{10}] \cdot 0,8$. The initial position of the mobile platform is: $[0.1; -0,1; -0,4]$ and the delay applied to the system is 80 ms.

11.3.7 Comparison of the Graphs from the Different Delays

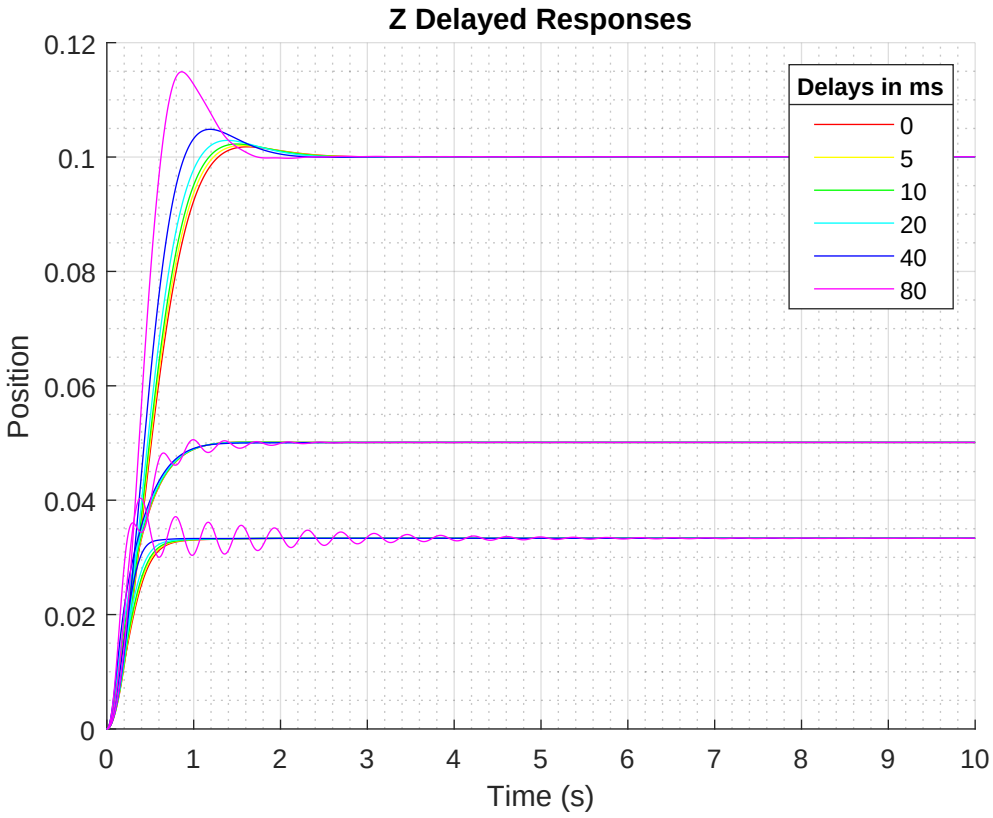


Figure 11.7: The comparison of all of the other delay graphs showing the absolute error from the different simulations.

Chapter 12

Simulation results: System Position Independence

12.1 Introduction

The test is made to show the impulse response of the controller at different positions to see if the position of the platform change the response of the system. This test is done with multiple axes at the same time in order to see the system act as a single unit.

12.2 Test Frame

The test has been conducted in a simulation. This also means that the results are not a ultimatum but more of an estimate of how the system would react in the implementation. The reason for the simulation to be done first is in order to secure that the system is stable and behaves as expected before the implementation, and the tests on the implementation.

Theoretical Background

The controller is supposed to have a consistent response independent of the position of the platform.

Test Setup and Test Procedure

The simulation is setup in Matlab using Runge-Kutta 4th order. The controller is implemented as described in **Chapter 4**.

The controller uses the parameters seen in **Table 12.1**. The impulse response is a force of 0.2 N applied along each axis x, y, z and a force of 0.2 N m is applied as a rotation force around x. The position of the mobile platform can be seen in the **Table 12.1**.

Test no.	Control parameter	Value	Position of the mobile platform $[x, y, z, \gamma]$
1	K_p, K_v	$[12, 8, 4, 4]$ $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$	$[0,1; -0,1; -0,4; 0,1]$
2	K_p, K_v	$[12, 8, 4, 4]$ $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$	$[0; 0,1; -0,4; -0,1]$
3	K_p, K_v	$[12, 8, 4, 4]$ $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$	$[0; 0; -0,3; 0,005]$

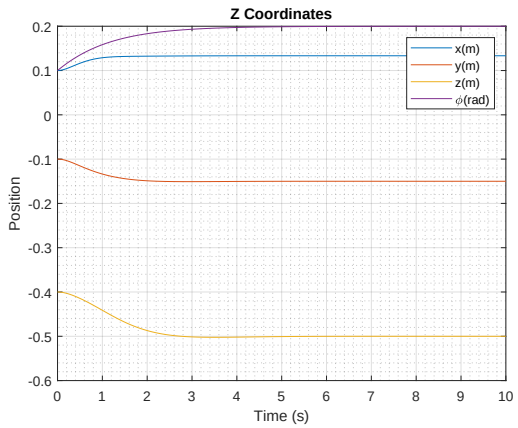
Table 12.1: The control parameters' values used for the test.

12.3 Test Results and Data Processing

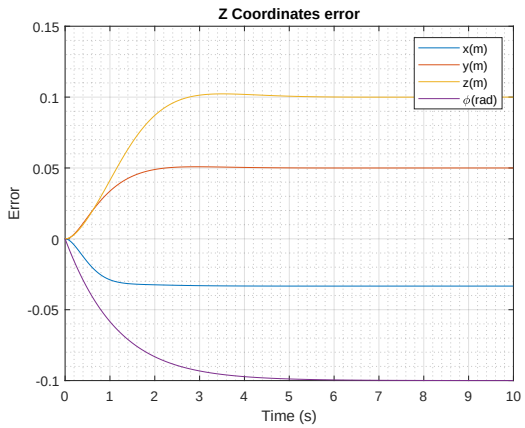
In the following section can the test results be seen in the corresponding subsection. The results consists of 3 graphs showing the actual position of the mobile platform of the Ragnar Robot, the error plot, and the absolute error.

Looking on the graphs for the tests can it be seen that the control system behaves independent of what the initial position is.

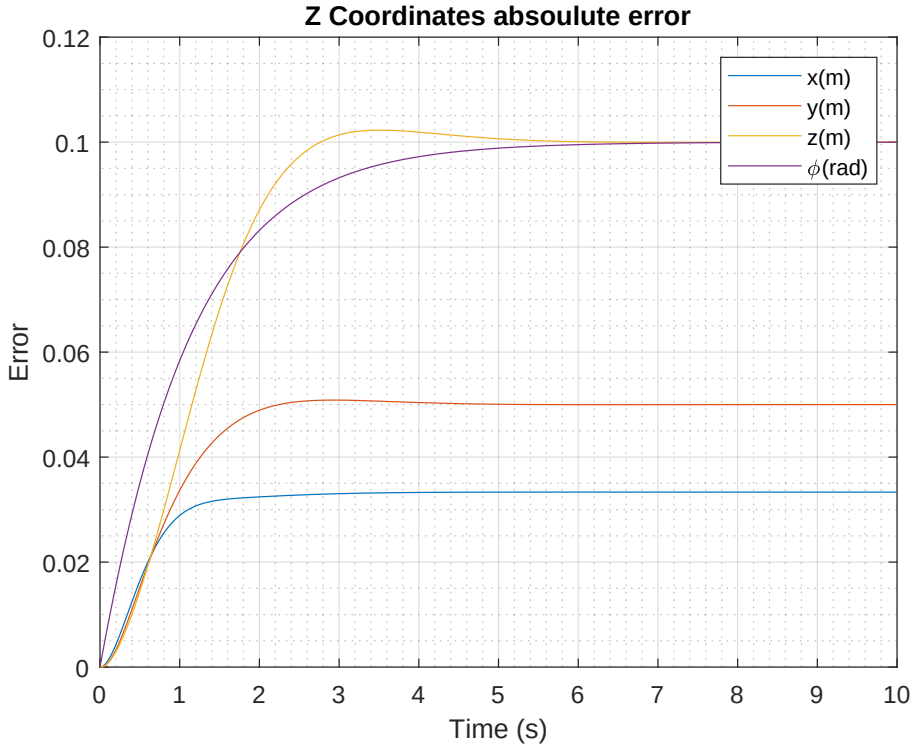
12.3.1 Test Results from Simulation 1



(a) The actual position.



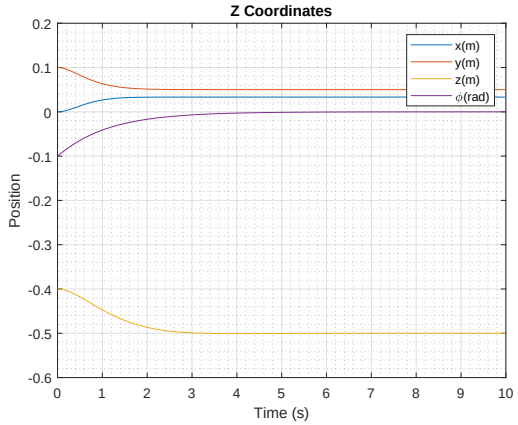
(b) The error plot.



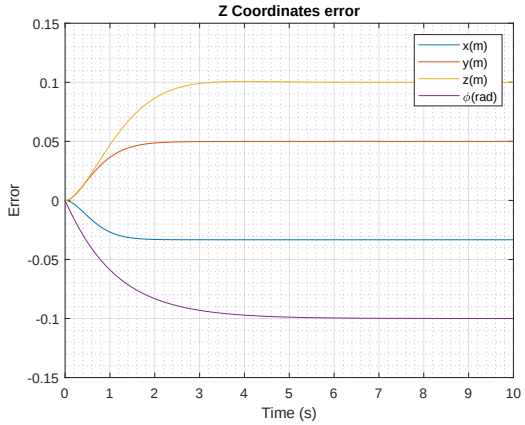
(c) The absolute error plot.

Figure 12.1: The graphs show the results from the first simulation with a force of 0.2 N applied on all of the axes and the rotation $[x, y, z, \gamma]$. The control parameters for K_p are $[12, 8, 4, 4]$ and for K_v $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$. The initial starting position of the mobile platform is $[0,1; -0,1; -0,4; 0,1]$.

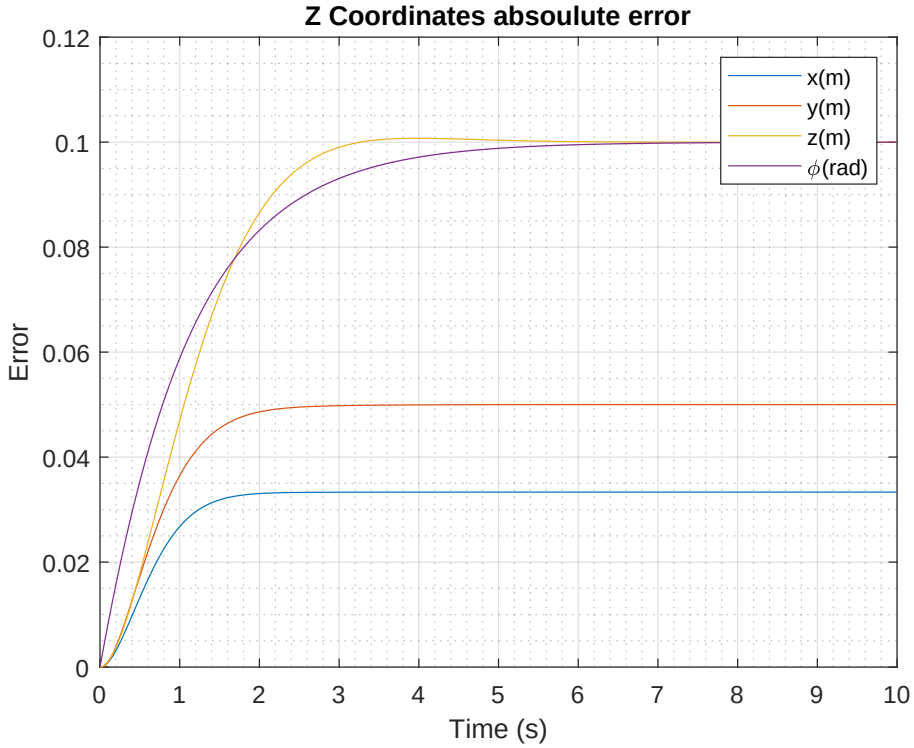
12.3.2 Test Results from Simulation 2



(a) The actual position.



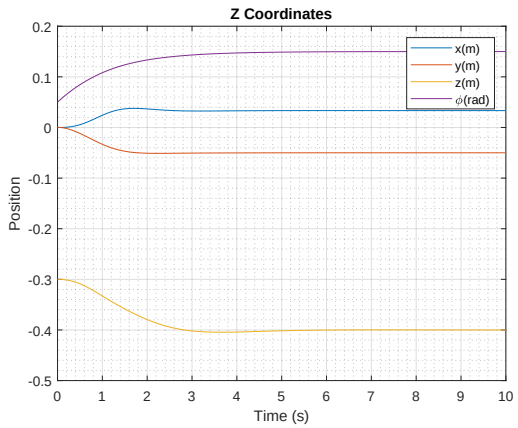
(b) The error plot.



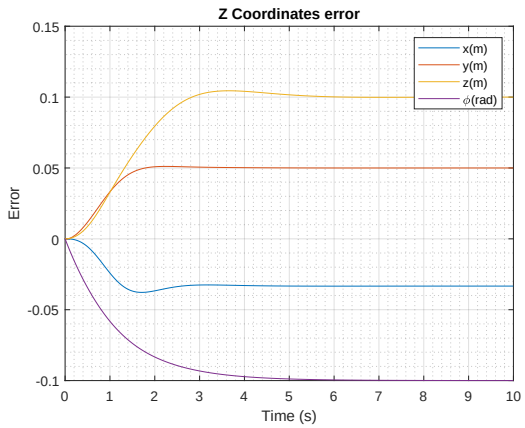
(c) The absolute error plot.

Figure 12.2: The graphs show the results from the second simulation with a force of 0.2 N applied on all of the axes and the rotation $[x, y, z, \gamma]$. The control parameters for K_p are $[12, 8, 4, 4]$ and for K_v $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$. The initial starting position of the mobile platform is $[0; 0,1; -0,4; -0,1]$.

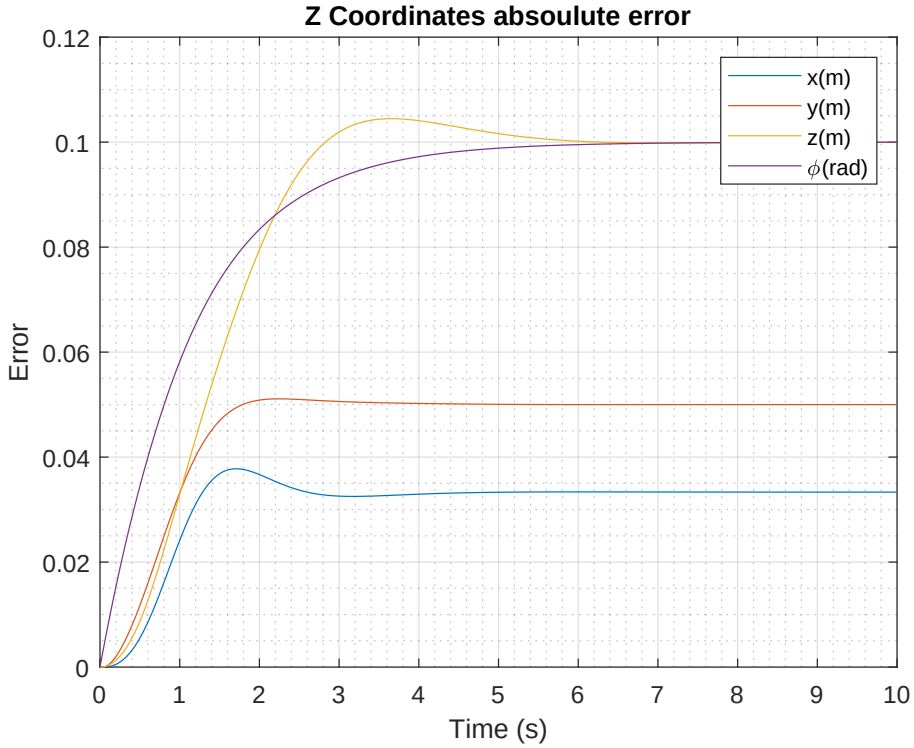
12.3.3 Test Results from Simulation 3



(a) The actual position.



(b) The error plot.



(c) The absolute error plot.

Figure 12.3: The graphs show the results from the first simulation with a force of 0.2 N applied on all of the axes and the rotation $[x, y, z, \gamma]$. The control parameters for K_p are $[12, 8, 4, 4]$ and for K_v $[\sqrt{12}, \sqrt{8}, \sqrt{4}, \sqrt{4}] \cdot 0,8$. The initial starting position of the mobile platform is $[0; 0; -0,3; 0,005]$.

12.3.4 Comparison of the Graphs from the three Different Simulations

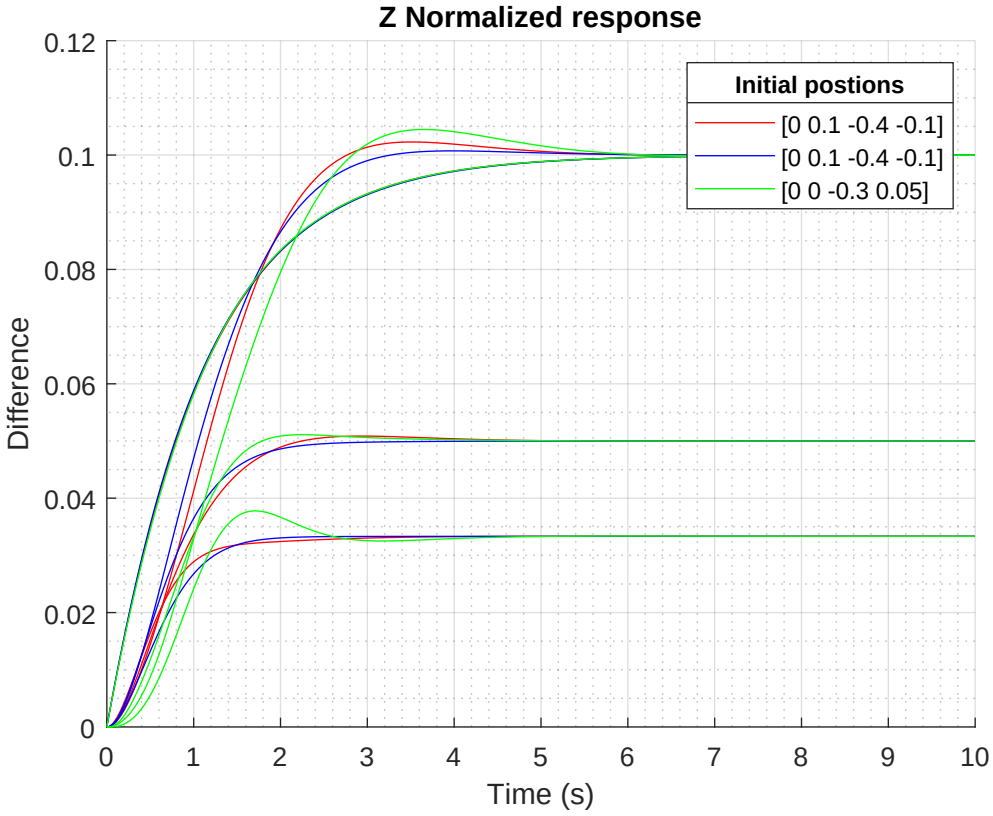


Figure 12.4: The comparison of the three graphs showing the absolute error from the different simulations.

Chapter 13

Test Journal: Loop Cycle Time

13.1 Introduction

Measurement of the time that takes to complete an entire cycle of the main loop function in the PID controller code.

13.2 Test Frame

The test is done by using a timer function in the code.

Theoretical background

The time is measured between the start of each iteration of the control loop. Therefore, the measured time includes both the control loop time and CAN bus communication time. However, the control loop time is insignificant compared to the communication, so the measured time is equivalent to the CAN bus communication time.

Test Setup and Test Procedure

Using a Teensy 3.6 with a CAN bus communication with the actuators and using Visual Studio Code the time that it takes to complete an entire cycle of the loop is measured and saved in a variable. This variable is printed together with other control variables, and it is also used to create the velocity error.

13.3 Test Results and Data Processing

The values of the time variable vary between 6 and 8 milliseconds. Since there is no much control computation, we assume that most of the time is used in the CAN bus communication and delays impose.

With this result can we assume that the measurements of the cycle time in the code with the complaint control will be as minimum as with this code, because the amount of computation will increase (computing the inversion of matrices, etc.).

Chapter 14

Appendix Code: Control 3T1R Code

```
function [tau] = control_3T1R(q, zr, kp, zeta)
    if nargin < 3 % this is for default values if no zeta or kp is set
       kp = [3 2 1 1]*2;
       zeta = 0.8;
    end
    kp = diag(kp);
    %variables for further development
    ddzr = 0; % ref acceleration;
    dzr = 0; % ref velocity;

    % used to test kv calculated as first or second order system
    DAMP_2ND_ORDER = true;
    if(DAMP_2ND_ORDER)
     kv = 2 * zeta * sqrt(kp) * 1;
    else
      kv = zeta * kp;
    end

    dt = 0.002; % Time steps, this would be nice as an input to the function

    z = q(5:8) % z task space

    q = q(1:4); % q joint space

    persistent q2 % q2 is used to save the previous value to get the derivative of q, dq
    if isempty(q2)
      q2 = q;
    end
    dq = (q - q2) / dt;
    q2 = q;

    persistent z2 % z2 is used to save the previous value to get the derivative of z, dz
    if isempty(z2)
        z2 = z;
    end
    dz = (z - z2) / dt;
    z2 = z;

    % dynamics, params = parameters
    persistent params % parameters for the Ragnar used to calculate the dynamics
    if isempty(params) % parameters are set once, on the first call of the function
        x = [pi/3  280/1000  114/1000  pi/12  600/1000  100/1000  70];
        % = [gama  Ax  Ay  alpha outer_arm  r  *unknown*];
        params = XForm_Parameter(x); % geometric parameters
    end
    persistent mass_params % mass parameters are set once, on the first call of the function
    if isempty(mass_params)
        mass_params = XForm_LinkageProperty(params); % dynamic parameters as mass, inertia
    end
    % extra parameters for the rotating platform
    m_down = 50e-3; % kg
    m_up = 50e-3; % kg
    platform_mass = [m_down m_up ];
```

```
    h0 = 40e-3; % distance from end effector to lower platform
    h = 80e-3; % distance from end effector to upper platform
    hs = [h0 h];
    [Mx, Hx, Gx] = ragnar_dynamic_parts_x_rotated( [q;z], [dq;dz], params, mass_params,
        platform_mass, hs); % extended dynamic matrices

    % Jacobian
    r1 = [55e-3 60e-3 -25e-3]';
    r3 = -r1; r3(3) = r1(3); % invert x and y but z remains
    r2 = [-80e-3 50e-3 -15e-3]';
    r4 = -r2; r4(3) = r2(3); % invert x and y but z remains

    r_all = [r1 r2 r3 r4]; % more parameters for the platform
    [Am, Bm] =  ragnar_rotated_x_AB( [q;z], params,hs,r_all); % A and B matrix used to
        calculate Jacobian
    J = Am\Bm;  % jacobian
    Ji = pinv(J); % inverse jacobian

    R = [Ji ; eye(size(Ji,2))]; % R used to go from qx to q

    H = R'*Hx; % the coriolis forces

    G = R'*Gx; % gravity forces

    %M = R'*Mx*R; % Mass matrix (not in use)

    % Handling the not continues rotation position
    rots = [zr(4) - z(4) ;zr(4) - (2*pi + z(4))]; % calculating both distances around
        the circular coordinates
    % the lowest value corresponding to the error
    [~ , rot_index] = min(abs(rots));
    erot = rots(rot_index);

    % error
    e = [zr(1:3) - z(1:3);erot];

    persistent e2 % save previous error to get derivative of error, de
    if isempty(e2)
        e2 = e;
    end
    de = (e - e2)/dt;
    e2 = e;

    u =  kv*(de) + kp*(e); % impedance control signal

    % disp(u);

    tau = J'*(u + H + G); % final torques for the actuators
end
```

Listing 14.1: Matlab example

Chapter 15

Appendix Code: Control Function Code

```
function  [tau]= control_function(q, zr)

    ddzr = 0; % ref acceleration;
    dzr = 0; % ref velocity
    kp = diag([6 4 2]*5);

    if(1)
        zeta = 0.8;
        kv = 2*zeta*sqrt(kp)*1;
    else
        alpha = 0.05;
       kv = alpha*kp;
    end


    dt = 0.001;

    % z task space
    z = q(5:7);
    qx = q;
    q = q(1:4);

    persistent qx2
    if isempty(qx2)
        qx2 = qx;
    end
    dqx = (qx - qx2)/dt;
    qx2 = qx;

    persistent dqx2
    if isempty(dqx2)
        dqx2 = dqx;
    end
    ddqx = (dqx - dqx2)/dt;
    dqx2 = dqx;

    persistent q2
    if isempty(q2)
        q2 = q;
    end
    dq=(q-q2)/dt;
    q2=q;

    persistent z2
    if isempty(z2)
        z2 = z;
    end
    dz = (z-z2)/dt;
    z2 = z;

    % dynamics
    persistent params
    if isempty(params)
        x = [pi/3  280/1000  114/1000  pi/12  600/1000  100/1000  70];
       % = [gama  Ax  Ay  alpha outer_arm  r  *unknown*];
       params = XForm_Parameter(x); % geometric parameters
    end
```

```
58      persistent mass_params
59      if isempty(mass_params)
60          mass_params = XForm_LinkageProperty(params); % dynamic parameters as mass,
                inertia
61      end
62      [Mx, Hx, Gx] = ragnar_dynamics_simplified([q;z], [dq;dz], params, mass_params); %
            returns mass matrix, centricoriolis,gravity
63
64      % Jacobian
65      [A, B] = rag_AB([q;z], params);
66      Cq = [B -A];
67      J = pinv(A)*B;
68      Jt = J';
69      Ji = pinv(J);
70
71
72
73
74      R = [Ji ; eye(size(Ji,2))];
75
76      H = R'*Hx;
77
78      G = R'*Gx;
79
80      M = R'*Mx*R;
81
82      Ma = R'*Mx;
83
84      %Mi = inv(M)
85
86      %Md = Ji'*M*Ji
87
88      % u =  M\kv*(dzr - dz) + M\kp*(zr -  z)
89      e=[zr(1:3) - z(1:3)];
90
91      persistent e2
92      if isempty(e2)
93          e2 = e;
94      end
95      de = (e - e2)/dt;
96      e2 = e;
97      u =  kv*(de) + kp*(e);
98
99
100     tau = J'*(u + H + G);
101
102 end
```

Listing 15.1: Matlab example

Chapter 16

Appendix Code: A main Simulation Computed Torque Rotation Code

```
% THE GOOD ONE
% this works good
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%    FAST DYNAMICS SIMULATION
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all;  close all; clc
%%
current_folder = pwd;
% add dynamic functions
addpath(fullfile(current_folder, '../Ragnar_x_rotation/generated')); % be sure to be on the folder of this script
% add kinematic functions
addpath(fullfile(current_folder, '../../Kinematics/Rotation_x_ragnar_kinematics')); % be sure to be on the folder of this script
%%
global g;
global alfa;
global veta;

alfa = 2;
veta = 2;
g = 9.81;
x = [pi/3  280/1000  114/1000   pi/12  600/1000   100/1000   70];
% = [gama   Ax         Ay        alpha outer_arm      r    *unknown*];
params = XForm_Parameter(x);
mass_params = XForm_LinkageProperty(params);
m_down = 50e-3; % kg
m_up = 50e-3; % kg
platform_mass = [m_down m_up ];
disp('mass_params')
disp(mass_params(1:6))

% init_pos = [x y z phi]
init_pos = [0.1 -0.1 -0.4 0.1]';
% init_pos = [0 0 -0.3 0.05]'
% init_pos = [0 0.1 -0.4 -0.1]'
% init_vel = [dx dy dz dphi]
init_vel = [0.0 0.0 0.0 0.0]';

r1 = [55e-3 60e-3 -25e-3]';
r3 = -r1; r3(3) = r1(3); % invert x and y but z remains
r2 = [-80e-3 50e-3 -15e-3]';
r4 = -r2; r4(3) = r2(3); % invert x and y but z remains

r_all = [r1 r2 r3 r4];
h0 = 40e-3; % distance from end effector to lower platform
h = 80e-3; % distance from end effector to upper platform
hs = [h0 h];
% initial solution for that point
% [sol1_0, sol2_0] = Rag_fullIKP(params, init_pos); % get the full initial position
% initial solution for rotating platform
[sol1_0, sol2_0] = Rag_fullIKP_rotate_x(params, init_pos); % get the full initial
```

```
    position

%%

y1 = [sol1_0(1); sol2_0(:,1)];
y2 = [sol1_0(2); sol2_0(:,2)];
y3 = [sol1_0(3); sol2_0(:,3)];
y4 = [sol1_0(4); sol2_0(:,4)];
figure
draw_ragnar_rotated(y1,y2,y3,y4,params,init_pos);
grid minor
view(45,45)
% q0 = [sol1_0(leg_num) sol2_0(1, leg_num) sol2_0(2, leg_num)]';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   dynamic equation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% Friction data is not used now
%      1. 0.0224 0.6841 0.0098
%      3. 0.0689 0.6646 0.0188
%      4. 0.0405 0.6911 0.0259

% with the new test for node 1 (joint 2) this are the new promising results
% 2. 0.0365 0.8224 0.018
global friction;
friction = [0.6841, 0.0098; 0.8224, 0.018; 0.6646, 0.0188; 0.6911, 0.0259];
friction = zeros(4,2);
% torq = [-1.0272     0.7332    -1.0266     0.7689]';

Torque = [0 0 0 0]'; % No torque input, free fall
% Torque = [-1.0272     0.7332    -1.0266     0.7689]';
q0 = [sol1_0(:); init_pos(:)];
dq0 = zeros(8,1); % d/dt(thetas x y z phi)
%%% initial acceleration
y = [dq0; q0];
t0 = 0;

dy = fast_dy_rotate(t0, y, params, mass_params, platform_mass, hs, r_all, Torque);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                             %
%        Running for a lapse time of 30 seconds                               %
%        delta time is set to 0.001 seconds, that is 1 milisecond             %
%        Just changed to 5 ms                                                 %
%                                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

dt = 0.002; % sample time in seconds
% dt = 0.01;
t_final = 10.0; %% final time is 10 seconds

t_ini = 0.0; %% initial time

n              = (t_final - t_ini ) / dt;
n              = round(n);
t_total        = zeros(1,n);
t_total(n)     = t_final;
q_total        = zeros(8,n);
dq_total       = zeros(8,n);
dv_total       = zeros(8,n);
```

```
115 q_total(:,1)  = y(9:16);
116 dq_total(:,1) = y(1:8);
117 dv_total(:,1) = dy(1:8);
118
119 % figure
120 % plot3(0,0,0)
121 %
122 % {
123 for i=1:1:201
124     f1(i) = 0.81 * sign(vel(i)) * (1-exp(-1 * abs(vel(i)))) + 0.02*vel(i);
125 end
126 plot(vel,f1)
127 grid on
128 grid minor
129 % }
130 % 26 3 1 14 15
131 % 19 11 25
132 ddq = zeros(8,1); dq = zeros(8,1); q = y(9:16);
133 tau = zeros(4,1);
134 torques(:, 1) = tau;
135 disp('Torques');
136 disp(tau);
137 Torque = tau;
138 %% Computed torque control - Control parameters
139 x_d = [0.1,-0.1,-0.6, deg2rad(45)]';
140 x_d = [init_pos + 0*[0.1 -0.1 -0.1 0.1]']
141 kkv = 5;
142 kv = [kkv,0.0,0.0,0.0; 0.0,kkv,0.0,0.0; 0.0,0.0,kkv,0.0; 0.0,0.0,0.0,kkv];
143 kkd = 3;
144 kd = [kkd,0,0,0;0,kkd*1.2,0,0;0,0,kkd*1.4,0;0,0,0,kkd*1.8];
145
146 de = [0,0,0,0]';
147 e = [0,0,0,0]';
148 el = e;
149
150 DRAW_ROBOT = false;
151 ROTATING_VIEW = false;
152 initial_estimation = init_pos;
153 x1 = initial_estimation;
154 for i=1:1:n-1
155     t_total(i) = (i-1) * dt;
156     time = t_total(i);
157     t_total(i),
158
159     %% Compute controller
160     % disp("pos params")
161     q_real = y(9:16);
162     dq_real = y(1:8);
163
164     %% Estimate moving platform using a numerical method
165     thetas_measured = q_real(1:4);
166     for ii = 1:30 % run newtons method 15 iterations
167         initial_estimation = x1;
168         dfx0 = Df_x_num(params,initial_estimation, thetas_measured);
169         ii;
170         fx0 = f_x_num(params,initial_estimation, thetas_measured);
171         x1 = initial_estimation - dfx0\fx0;
172         x1(4) = wrapToPi(x1(4));
173         x1;
174     end
175     q_controller = [thetas_measured; x1]; % this is the estimation
176     % now get the platform velocities through the use of jacobian ..... .
177     % get the jacobian parts from q_contrller, which is measurement and
```

```
% esimation of platform position

[Am, Bm] =  ragnar_rotated_x_AB( q_controller, params,hs,r_all);
Jacobian_ragnar = Am\Bm;
dq_controller = [dq_real(1:4) ; Jacobian_ragnar * dq_real(1:4)];
dq_controller;
dq_real;
%% compute controller with the joint positions and velocities
% and estimates of the positionf of moving platform and velocities of
% moving platform
[M, C, G] = ragnar_dynamic_parts_x_rotated( q_controller, dq_controller, params, mass_params, platform_mass, hs);

ddqd = [0,0,0,0,0,0,0,0]';
% [A, B] = rag_AB(y(8:14), params);
% Cq = [B -A];
Cq = rag_Cq(q_controller, params, hs, r_all);

% J = pinv(A)*B;
% Jt = J';
% compute error and error derivative
[sol1_ref, sol2] = Rag_fullIKP_rotate_x(params, x_d); %get the full initial position
e = x_d - q_controller(5:8);
%%e = sol1_ref - y(9:12);
%%e(4) = wrapToPi(e(4));
de = zeros(size(e,1),size(e,2));
if (i>1)
    de = (e-el)/dt;
end
el = e;
err(:,i)=e;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% compute controller
% Torque = M*(ddqd + [0;0;0;0;(kv*e+kd*de)]) + C + G;
f1 = friction(:,1);
f2 = friction(:,2);
Fric = f1.*sign(y(1:4)) + f2.*y(1:4);
for ij = 1:1:4
    Fric(ij) = f1(ij)* sign(y(ij)) * (1-exp(-3 * abs(y(ij)))) + f2(ij)*y(ij);
end
Fric = [Fric; zeros(4,1)];
% Fric = zeros(7,1);

compensation = (kv*e+kd*de);

f = M*(ddqd + [0;0;0;0;compensation]) + C + G; % + Fric;
% f = M*(ddqd + [0;0;0;0;0;0;0;0]) + C + G;
% f = G;
CM = zeros(8,8); CM(1:4, 1:4) = diag([1 1 1 1]); CM(:, 5:8) = -Cq';
forces = CM\f; tau = forces(1:4); torques(:, i) = tau;
% Torque = zeros(4,1);
control_torque = tau;
% Impedance control
kp=[6 4 2 2]*2;
zeta=0.8;
% DO DELAY
Torque = control_3T1R(q_controller,x_d,kp,zeta); %  + Fric(1:4);
% Torque = zeros(4,1);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% controller finishes
fext=0.2*[1 -1 -1 1]';%||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
238     %% PLANT TO BE CONTROLLED IS HERE, RECEIVES TORQUES, OUTPUTS ACCELERATION, VELOCITY
            AND POSITIONS
239     % begin integrator
240     % fast_dy_rotate(t0, y, params, mass_params, platform_mass, hs, r_all, Torque);
241     k1 = feval(@fast_dy_rotate, t_total(i), y, params, mass_params, platform_mass, hs,
            r_all, Torque,fext);
242     dv_total(:,i) = k1(1:8);
243     % feval(@fast_dy_rotate, t_total(i), y, params, platform_mass, hs, r_all, Torque);
244     k1(1:4);
245     % k2 = feval(@fast_dy, t_total(i) + dt/2, y + k1*dt/2, params,  mass_params,Torque)
            ;
246     k2 = feval(@fast_dy_rotate, t_total(i) + dt/2, y + k1*dt/2, params, mass_params,
            platform_mass, hs, r_all, Torque ,fext);
247     % k3 = feval(@fast_dy, t_total(i) + dt/2, y + k2*dt/2, params,  mass_params,Torque)
            ;
248     k3 = feval(@fast_dy_rotate, t_total(i) + dt/2, y + k2*dt/2, params, mass_params,
            platform_mass, hs, r_all, Torque ,fext);
249     % k4 = feval(@fast_dy, t_total(i) + dt  , y + k3*dt  , params,  mass_params,Torque)
            ;
250     k4 = feval(@fast_dy_rotate, t_total(i) + dt, y + k3*dt, params, mass_params,
            platform_mass, hs, r_all, Torque ,fext);
251     increment = dt * (k1/6 + k2/3 + k3/3 + k4/6);
252     y = y + increment;
253     % rad2deg(y)
254     % end integrator
255     % take the velocities
256     dq_total(:,i+1) = y(1:8);
257     % take the positions
258     rotrad1 = y(16);
259     y(16) = wrapToPi(y(16));
260     rotrad2 = y(16);
261     estrot = q_controller(8)
262     q_total(:,i+1) = y(9:16);
263
264     % end calculation
265     %% PLANT TO BE CONTROLLED ENDS
266
267     %% VISUALIZATION BEGINS HERE
268     [sol1, sol2] = Rag_fullIKP_rotate_x(params, y(13:16)); % get the full initial
            position
269
270     %%%
271     % correct y %
272     y(9:12) = sol1(:);
273     % dro
274     y1 = [sol1(1); sol2(:,1)];
275     y2 = [sol1(2); sol2(:,2)];
276     y3 = [sol1(3); sol2(:,3)];
277     y4 = [sol1(4); sol2(:,4)];
278
279     % figure
280     if(mod(i,round(1/(30*dt)))==0)
281         if DRAW_ROBOT
282
283             draw_ragnar_rotated(y1,y2,y3,y4,params,y(13:16));
284             grid minor
285             if ROTATING_VIEW
286                 viewi = i * 1.5; % 90/20
287             else
288                 viewi = 45;
289             end
290             view(viewi,20)
```

```
            pause(0.01);
        end
    end
    % {
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% dynamic equation
    % the four articulated joints: theta_1, ..., theta_4, x, y, z, phi
    %========================================== veloctiy and acceleration
    ddq = dv_total(:,i); dq = y(1:7); q = y(8:14);
    %========================================== constraint equation
    [A, B] = rag_AB(q, params);
    Cq = [B -A];
    [M, C, G] = ragnar_dynamics_simplified(q, dq, params, mass_params);

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% right-handed side
    disp('mddq')
    disp(M*ddq)
    disp('coriolis')
    disp(C)
    disp('g')
    disp(G)
    f = M*ddq + C + G;
    disp('f')
    disp(f)
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% left-handed side
    CM = zeros(7,8); CM(1:4, 1:4) = diag([1 1 1 1]); CM(:, 5:8) = Cq';
    disp("Matrix CM")
    disp(CM)
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% actuator torques
    forces = pinv(CM)*f; tau = forces(1:4); torques(:, i) = tau;
    disp('Torques');
    disp(tau);

    % Torque = torques(:,i);

    % }
    hold off
end

%%
close all
if(1)
    testname = "response_delay"+mat2str(kp)+"_"
    save(testname+'q_total.mat','q_total');
    save(testname+'err.mat','err');
else
    testname = "position_independence_"+mat2str([0.1 -0.1 -0.4 0.1]')+"_"
    testname = "response_kp_scaling_"+mat2str([6 4 2 2])+"_"
    testname = "response_axis_independence_"+mat2str(0.2*[1 -0 -0 0]')+"_"
    load = matfile(testname+'q_total.mat');
    q_total = load.q_total;
    load = matfile(testname+'err.mat');
    err = load.err;
end
close
pos_plot=figure
plot(t_total,q_total(5:8,:)')
title('Z Coordinates')
legend('x(m)','y(m)','z(m)','\phi(rad)')
xlabel('Time (s)')
ylabel('Position')
grid on
grid minor

```

```
354 error_plot=figure
355 plot(t_total(1:end-1),err')
356 title('Z Coordinates error');
357 legend('x(m)','y(m)','z(m)','\phi(rad)')
358 xlabel('Time (s)')
359 ylabel('Error')
360 grid on
361 grid minor
362
363 abs_error_plot = figure
364 plot(t_total(1:end-1),abs(err'))
365 title('Z Coordinates absoulute error');
366 legend('x(m)','y(m)','z(m)','\phi(rad)')
367 xlabel('Time (s)')
368 ylabel('Error')
369 grid on
370 grid minor
371
372 h = pos_plot
373 fn = testname+'pos_plot.pdf'
374 set(h,'Units','Inches');
375 pos = get(h,'Position');
376 set(h,'PaperPositionMode','Auto','PaperUnits','Inches','PaperSize',[pos(3), pos(4)])
377 print(h,fn,'-dpdf','-r0')
378
379 h=error_plot
380 fn = testname+'error_plot.pdf'
381 set(h,'Units','Inches');
382 pos = get(h,'Position');
383 set(h,'PaperPositionMode','Auto','PaperUnits','Inches','PaperSize',[pos(3), pos(4)])
384 print(h,fn,'-dpdf','-r0')
385
386 h = abs_error_plot
387 fn = testname+'abs_error_plot.pdf'
388 set(h,'Units','Inches');
389 pos = get(h,'Position');
390 set(h,'PaperPositionMode','Auto','PaperUnits','Inches','PaperSize',[pos(3), pos(4)])
391 print(h,fn,'-dpdf','-r0')
392 % {
393 figure
394 plot(dq_total(1:4,:)')
395 title('dqs')
396 grid on
397 grid minor
398 axis([1 n-1 -0.1 0.1])
399 grid minor
400 % }
401 %%
402 %% Make difference plot
403 if(false)
404     testname = "response_axis_independence_"+mat2str(0.2*[1 -1 -1 1]')+"_"
405     load = matfile(testname+'err.mat');
406     err_total = load.err;
407
408     testname = "response_axis_independence_"+mat2str(0.2*[1 -0 -0 0]')+"_"
409     load = matfile(testname+'err.mat');
410     err_x = load.err;
411
412     testname = "response_axis_independence_"+mat2str(0.2*[0 -1 -0 0]')+"_"
413     load = matfile(testname+'err.mat');
414     err_y = load.err;
415
416     testname = "response_axis_independence_"+mat2str(0.2*[0 -0 -1 0]')+"_"
```

```
    load = matfile(testname+'err.mat');
    err_z = load.err;

    testname = "response_axis_independence_"+mat2str(0.2*[0 -0 -0 1]')+"_"
    load = matfile(testname+'err.mat');
    err_gamma = load.err;

    err_diff=err_total - [err_x(1,:);err_y(2,:);err_z(3,:);err_gamma(4,:)];

    abs_err_diff=abs(err_total) - abs([err_x(1,:);err_y(2,:);err_z(3,:);err_gamma(4,:)
        ]);

    difference_plot=figure
    plot(t_total(1:end-1),err_diff')
    title('Z Error Difference')
    legend('x(m)','y(m)','z(m)','\phi(rad)')
    xlabel('Time (s)')
    ylabel('Difference')
    grid on
    grid minor

    h = difference_plot
    fn = "response_axis_independence_"+'difference_plot.pdf'
    set(h,'Units','Inches');
    pos = get(h,'Position');
    set(h,'PaperPositionMode','Auto','PaperUnits','Inches','PaperSize',[pos(3), pos(4)
        ])
    print(h,fn,'-dpdf','-r0')
end
%%
%% Make scaling comparison plot
if(false)
    testname = "response_kp_scaling_"+mat2str([12 8 4 4])+"_"
    load = matfile(testname+'err.mat');
    err_2 = load.err;

    testname = "response_kp_scaling_"+mat2str([6 4 2 2])+"_"
    load = matfile(testname+'err.mat');
    err_1 = load.err;

   err_1_norm = err_1*(1/(err_1(end:end)));

   err_2_norm = err_2*(1/(err_2(end:end)));

    normalized_plot = figure
    hold on
    pc = plot(t_total(1:end-1),abs(err_1_norm'),'r')
    pc = [pc plot(t_total(1:end-1),abs(err_2_norm'),'b')]
    hold off
    title('Z Normalized response')
    lgd = legend(pc(1,:),{'Kp = [6 4 2 2]','Kp = [12 8 4 4]'})

    title(lgd,'Kp')

    xlabel('Time (s)')
    ylabel('Difference')
    grid on
    grid minor

    h = normalized_plot
    fn ="response_kp_scaling_"+'normalized.pdf'
    set(h,'Units','Inches');
    pos = get(h,'Position');
```

```
478     set(h,'PaperPositionMode','Auto','PaperUnits','Inches','PaperSize',[pos(3), pos(4)
            ])
479     print(h,fn,'-dpdf','-r0')
480 end
481 %%
482 %% Make position independence comparison
483 if(false)
484     testname = "position_independence_"+mat2str([0.1  -0.1  -0.4  0.1]')+"_"
485     load = matfile(testname+'err.mat');
486     err_1 = load.err;
487 
488     testname = "position_independence_"+mat2str([0  0.1  -0.4  -0.1]')+"_"
489     load = matfile(testname+'err.mat');
490     err_2 = load.err;
491 
492     testname = "position_independence_"+mat2str([0  0  -0.3  0.05]')+"_"
493     load = matfile(testname+'err.mat');
494     err_3 = load.err;
495 
496     position_comparison_plot = figure
497     hold on
498     pc = plot(t_total(1:end-1),abs(err_1'),'r')
499     pc = [pc plot(t_total(1:end-1),abs(err_2'),'b')]
500     pc = [pc plot(t_total(1:end-1),abs(err_3'),'g')]
501     hold off
502     title('Z Normalized response')
503     lgd = legend(pc(1,:),{'[0  0.1  -0.4  -0.1]','[0  0.1  -0.4  -0.1]','[0  0  -0.3  0.05]'})
504 
505     title(lgd,'Initial postions')
506 
507     xlabel('Time (s)')
508     ylabel('Difference')
509     grid on
510     grid minor
511 
512     h = position_comparison_plot
513     fn ="position_independence_"+'comparison.pdf'
514     set(h,'Units','Inches');
515     pos = get(h,'Position');
516     set(h,'PaperPositionMode','Auto','PaperUnits','Inches','PaperSize',[pos(3), pos(4)
            ])
517     print(h,fn,'-dpdf','-r0')
518 end
```

Listing 16.1: Matlab example

Chapter 17

Appendix Code: A main Ragnar Fast Dynamics Simulation Free Fall Code

```
% Author: Juan de Dios Flores Mendez
% based on Guanglei Wu and Shaoping Bai - Design Analysis and Dynamics
% Modeling
% Simulation of the dynamics of the ragnar. No torques applied so it falls
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   FAST DYNAMICS SIMULATION
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all;  close all; clc
 delays=[0 5 10 20 40 80]
   for d = 1:length(delays)
       clear functions;
global g;
global alfa;
global veta;
global friction; % ignore for the moment
friction(:,1) = 0;
friction(:,2) = 0;
alfa = 2;
veta = 2;
g = 9.82;
x = [pi/3  280/1000  114/1000   pi/12  600/1000   100/1000   70];
% = [gama   Ax         Ay       alpha outer_arm      r   *unknown*];
params = XForm_Parameter(x); % geometric parameters
mass_params = XForm_LinkageProperty(params); % dynamic parameters as mass, inertia
disp('mass_params')
disp(mass_params(1:6))


init_pos = [0.0 0.0 -0.3]'; % cartesian
init_vel = [0.0 0.0 0.0]'; % cartesian

dq0 = zeros(3,4);

% initial solution for that point
[sol1_0, sol2_0] = Rag_fullIKP(params, init_pos); % get the full initial position
%%%
init_pos = [0.0 0.0 -0.4]';
[sol4, sol42] = Rag_fullIKP(params, init_pos); % get the full initial position
init_pos = [0.0 0.1 -0.3]';
[sol3, sol32] = Rag_fullIKP(params, init_pos); % get the full initial position
init_pos = [0 0 -0.4]';
offset = [deg2rad(90)*ones(2,1);-deg2rad(90)*ones(2,1)]
y1 = [offset(1); sol42(:,1)];
y2 = [offset(2); sol42(:,2)];
y3 = [offset(3); sol42(:,3)];
y4 = [offset(4); sol42(:,4)];
%     close all;
size=600;
set(0,'DefaultFigureVisible','off');
fig_anim=figure('Position', [300 300 size*3*1.1 size]);
%draw_ragnar(y1,y2,y3,y4,params,init_pos);
```

```
grid minor
view(45,45)
set(0,'DefaultFigureVisible','on');
rad2deg([sol3-offset sol4-offset])
X=0;Y=0;Z=-450/1000;
[solo, solo2] = Rag_fullIKP(params, [X,Y,Z]); %get the full initial position
rad2deg(solo-offset)
%%%
% q0 = [sol1_0';
%       sol2_0];
q0 = [sol1_0;init_pos];
% q0 = [sol1_0(leg_num) sol2_0(1, leg_num) sol2_0(2, leg_num)]';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%   dynamic equation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% torq = [-1.0272    0.7332   -1.0266    0.7689]';

Torque = [0 0 0 0]'; % No torque input, free fall
% Torque = [-1.0272    0.7332   -1.0266    0.7689]';

dq0 = zeros(7,1);
%%% initial acceleration
y = [dq0; q0];
t0 = 0;
% the function below will return acceleration and velocities
% from a dynamic model with torque input
dy = fast_dy(t0, y, params, mass_params, Torque);
ddq = zeros(7,1);
% this is the inverse dynamics, will return the torque from a
% acceleration input
torques = fast_dy_torque(t0, y, params, mass_params, ddq)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Running for a lapse time of 10 seconds
%       delta time is set to 0.001 seconds, that is 1 milisecond
%       Just changed to 5 ms
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
dy
dt = 0.001; % sample time

t_final = 10.00; %% final time is 10 seconds

t_ini = 0.0; %%initial time

%% variables to save
n = (t_final - t_ini ) / dt;
n = round(n);
t_total = zeros(1,n);
t_total(n) = t_final;
q_total       = zeros(7,n);
dq_total      = zeros(7,n);
dv_total      = zeros(7,n);
q_total(:,1)  = y(8:14);
dq_total(:,1) = y(1:7);
dv_total(:,1) = dy(1:7);

% figure
```

```
% plot3(0,0,0)
%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% dynamic equation
% the four articulated joints: theta_1, ..., theta_4, x, y, z, phi
%================================================== veloctiy and acceleration
ddq = zeros(7,1); dq = zeros(7,1); q = y(8:14);
%================================================== constraint equation
[A, B] = rag_AB(q, params);
Cq = [B -A];
[M, C, G] = ragnar_dynamics_simplified(q, dq, params, mass_params); % returns mass
    matrix, centricoriolis,gravity

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% right-handed side
f = M*ddq + C + G;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% left-handed side
CM = zeros(7,8); CM(1:4, 1:4) = diag([1 1 1 1]); CM(:, 5:8) = Cq';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% actuator torques
forces = pinv(CM)*f; %% solve the constraiend equation
tau = forces(1:4);
torques(:, 1) = tau;
disp('Torques');
disp(tau);
%Torque = tau;
Torque = zeros(4,1);

% initiate simulation
plot_title='test'
control=1
draw=0
drawAll=1;
ref=init_pos+[0 0 0]'
f=0;
fps=10;

if (draw)
    figure(fig_anim)
end
for i=1:1:n-1
    t_total(i) = (i-1) * dt;
    time = t_total(i);
    t_total(i),
    %control!!!!!!!
    if(control==1)
        delay=delays(d);
        Ctau=control_function(q_total(:,max([1;i-delay])),ref);
        % init_pos=init_pos+[0.00*dt,0.0*dt,0.0*dt]'

        %Torques = fast_dy_torque(t_total(i), y, params, mass_params, ddq)
        Torque = Ctau;
        %Torque=zeros(4,1)
    end
    % begin integrator of acceleration and velocities with runge kutta fourth order
    fext=1*[1 -1 -1]';%
        ==========================================================================
    k1 = feval(@fast_dy, t_total(i), y, params, mass_params, Torque, fext);
    dv_total(:,i) = k1(1:7);
    %%
%     disp('tor');
%     ddq = k1(1:7)
%
```

```
%     disp('acc')
%
%     k1(5:7)
%     disp('vel')
%
%     y(5:7)
%     disp('pos')
%
%     y(12:14)
    %%

    k1(1:3);
    k2 = feval(@fast_dy, t_total(i) + dt/2, y + k1*dt/2, params,  mass_params,Torque,
        fext);
    k3 = feval(@fast_dy, t_total(i) + dt/2, y + k2*dt/2, params,  mass_params,Torque,
        fext);
    k4 = feval(@fast_dy, t_total(i) + dt  , y + k3*dt  , params,  mass_params,Torque,
        fext);
    increment = dt * (k1/6 + k2/3 + k3/3 + k4/6);
    y = y + increment;
    %% finish the runge kutta fourth order


    % rad2deg(y)
    % end integrator
    % take the velocities
    dq_total(:,i+1) = y(1:7);
    % take the positions
    q_total(:,i+1)  = y(8:14);
    % end calculation

    % accomodate the position values with the inverse kinematics so it does not die
    [sol1, sol2] = Rag_fullIKP(params, y(12:14));

    y1 = [sol1(1); sol2(:,1)];
    y2 = [sol1(2); sol2(:,2)];
    y3 = [sol1(3); sol2(:,3)];
    y4 = [sol1(4); sol2(:,4)];
    %     close all;
    % draw the viewtiful ragnar
    if(mod(i,round(1/(fps*dt)))==0)
        time=i*dt
        f=f+1;
        if(draw)
            if(drawAll)

                subplot(1,3,1)
                draw_ragnar(y1,y2,y3,y4,params,y(12:14));
                view(0,90)
                F1(f)=getframe;
                hold off

                subplot(1,3,3)
                draw_ragnar(y1,y2,y3,y4,params,y(12:14));
                view(0,0)
                F2(f)=getframe;
                hold off
            end
            subplot(1,3,2)
            draw_ragnar(y1,y2,y3,y4,params,y(12:14));
            view(360*t_final/10*(i/n),45)
            F3(f)=getframe;
            hold off
```

```
239         end
240
241
242     end
243     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% dynamic equation
244     % the four articulated joints: theta_1, ..., theta_4, x, y, z, phi
245     %================================================== veloctiy and acceleration
246     %       ddq = dv_total(:,i); dq = y(1:7); q = y(8:14);
247     %================================================== constraint equation
248     %       [A, B] = rag_AB(q, params);
249     %       Cq = [B -A];
250     %       [M, C, G] = ragnar_dynamics_simplified(q, dq, params, mass_params);
251
252     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% right-handed side
253     %       disp('mddq')
254     %       disp(M*ddq)
255     %       disp('coriolis')
256     %       disp(C)
257     %       disp('g')
258     %       disp(G)
259     %       f = M*ddq + C + G;
260     %       disp('f')
261     %       disp(f)
262     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% left-handed side
263     %      CM = zeros(7,8); CM(1:4, 1:4) = diag([1 1 1 1]); CM(:, 5:8) = Cq';
264     %       disp("Matrix CM")
265     %       disp(CM)
266     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% actuator torques
267     %       forces = pinv(CM)*f; tau = forces(1:4); torques(:, i) = tau;
268     %       disp('Torques');
269     %       disp(tau);
270
271
272
273     pause(0.0001)
274 end
275 %%
276 fig_plot=figure('Position', [300 300 size*1.1 size])
277 xlabel('Time (s)')
278 ylabel('Position (m)')
279 title(plot_title)
280 hold on
281 plot(t_total(1,:),q_total(5,:)-init_pos(1),'r')
282 plot(t_total(1,:),q_total(6,:)-init_pos(2),'g')
283 plot(t_total(1,:),q_total(7,:)-init_pos(3),'b')
284 hold off
285 legend('x(m)','y(m)','z(m)')
286 xlabel('Time (s)')
287 ylabel('Error')
288 grid on
289 grid minor
290 max_x=max(q_total(5,:));
291 max_y=max(q_total(6,:));
292 max_z=max(q_total(7,:));
293 max_p=[max_x max_y max_z]'
294 overshoot_p=max_p./[q_total(5,end) q_total(6,end) q_total(7,end)]'
295 %% Make difference plot
296
297 %%
298 %%
299 close all
300 if(1)
301 testname = "response_delay_"+delay*dt*1000+"_ms_"
```

```
    save(testname+'q_total.mat','q_total');
    err=q_total(5:7,:)-repmat(init_pos,1,i+1)
    save(testname+'err.mat','err');
else
 testname = "position_independence_"+mat2str([0.1  -0.1  -0.4  0.1]')+"_"
 testname = "response_kp_scaling_"+mat2str([6  4  2  2])+"_"
 testname = "response_axis_independence_"+mat2str(0.2*[1  -0  -0  0]')+"_"
    load = matfile(testname+'q_total.mat');
    q_total = load.q_total;
    load = matfile(testname+'err.mat');
    err = load.err;
end
close
pos_plot=figure
plot(t_total,q_total(5:7,:)')
title('Z Coordinates')
legend('x(m)','y(m)','z(m)','\phi(rad)')
xlabel('Time (s)')
ylabel('Position')
grid on
grid minor

error_plot=figure
plot(t_total(1:end),err')
title('Z Coordinates error');
legend('x(m)','y(m)','z(m)','\phi(rad)')
xlabel('Time (s)')
ylabel('Error')
grid on
grid minor

abs_error_plot=figure
plot(t_total(1:end),abs(err'))
title('Z Coordinates absolute error');
legend('x(m)','y(m)','z(m)','\phi(rad)')
xlabel('Time (s)')
ylabel('Error')
grid on
grid minor


h = pos_plot
fn = testname+'pos_plot.pdf'
set(h,'Units','Inches');
pos = get(h,'Position');
set(h,'PaperPositionMode','Auto','PaperUnits','Inches','PaperSize',[pos(3), pos(4)])
print(h,fn,'-dpdf','-r0')

h = error_plot
fn = testname+'error_plot.pdf'
set(h,'Units','Inches');
pos = get(h,'Position');
set(h,'PaperPositionMode','Auto','PaperUnits','Inches','PaperSize',[pos(3), pos(4)])
print(h,fn,'-dpdf','-r0')

h = abs_error_plot
fn = testname+'abs_error_plot.pdf'
set(h,'Units','Inches');
pos = get(h,'Position');
set(h,'PaperPositionMode','Auto','PaperUnits','Inches','PaperSize',[pos(3), pos(4)])
print(h,fn,'-dpdf','-r0')
    end
% {
```

```
figure
plot(dq_total(1:4,:)')
title('dqs')
grid on
grid minor
axis([1 n-1 -0.1 0.1])
grid minor
% }
%%
if(draw)
    % figure(fig_anim)
    pause(0.2)
    for i =1:f

        if(1)
            subplot(1,3,1)
            imshow(F1(i).cdata)


            subplot(1,3,3)
            imshow(F2(i).cdata)
        end
        subplot(1,3,2)
        imshow(F3(i).cdata)

        pause(0.0001)
        % pause(1/fps)
        time=i/fps
    end

end
%%
%% Make difference plot
if(true)

    % delays = [0 10 20 50 100 150]
    plot_all = figure
    hold on
    LD = length(delays)
    col = hsv(LD);
    pc = []
    plot_all.Renderer = 'Painters';
    for i = 1:LD

    testname = "response_delay_"+delays(i)*dt*1000+"_ms_"
    load = matfile(testname+'err.mat');
    err_total = load.err;

    pc = [pc plot(t_total(1:end),abs(err_total'),'color',col(i,:))]

    end
    hold off

title('Z Delayed Responses')
lgd = legend(pc(1,:),string(delays))

title(lgd,'Delays in ms')
xlabel('Time (s)')
ylabel('Position')
grid on
grid minor

h = plot_all
```

```
fn = "response_delay_"+'comparison.pdf'
set(h,'Units','Inches');
pos = get(h,'Position');
set(h,'PaperPositionMode','Auto','PaperUnits','Inches','PaperSize',[pos(3), pos(4)])
print(h,fn,'-dpdf','-r0')
end
```

Listing 17.1: Matlab example

Bibliography

[1] Hi-Industri, "Enkel og omstillingsvenlige robotter," last visited: 14-11-2019. [Online]. Available: https://www.hi-industri.dk/presse/pressemeddelelser-fra-udstillerne?PubId=4359

[2] PJRC, "Teensy usb development board," last visited: 14-11-2019. [Online]. Available: https://www.pjrc.com/store/teensy36.html

[3] P. B. O. v. S. Marie Schumacher, Janis Wojtusch, "An introductory review of active compliant control," *Robotics and Autonomous Systems*, vol. 119, pp. 185–200, 2019.

[4] O. M. Juan de Dios Flores-Mendez, Henrik Schiøler and S. Bai, "Impedance control and force estimation of a redundant parallel kinematic manipulator."

[5] B. Workforce. (2019) Blue workforce. Last visited: 14-11-2019. [Online]. Available: https://bwf3.tindwork.dk/

[6] P. Stoffregen, "Teensy 3.5 teensy 3.6," August 2019, last visited: 14-11-2019. [Online]. Available: https://www.kickstarter.com/projects/paulstoffregen/teensy-35-and-36

[7] Sparkfun, "Teensy 3.6," last visited: 14-11-2019. [Online]. Available: https://www.sparkfun.com/products/14057

[8] M. L. Rachel Kelly, Karin Hoffman and K. Silver, "Tomato workers' health and safety," 2017, last visited: 14-11-2019. [Online]. Available: https://www.migrantclinician.org/files/2017-04-06%20-%20Tomato%20Workers%20Health%20and%20Safety%20Guide.pdf

[9] O. M. Juan de Dios Flores-Mendez, Henrik Schiøler and S. Bai, "Design of a dynamically reconfigurable 3t1r parallel kinematic manipulator."

[10] Nanotec. (2019) Nanotec electronic gmbh co kg. https://en.nanotec.com/. Last visited: 24-11-2019.

[11] W. contributors. (2019) Lagrangian mechanics. Last visited: 21-11-2019. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Lagrangian_mechanics&oldid=923081147

[12] N. Hogan, "Impedance control: An approach to manipulation: Part 1 theory," *Journal of Dynamic Systems, Measurement, and Control*, vol. 107, pp. 1–7, 1985.

[13] C. Z. Tan Delin, "On a general formula of fourth order runge-kutta method (pdf)," *Journal of Mathematical Science Mathematics Education*, vol. Vol. 7 No. 2, pp. 1–10, 2012, last visited: 14-11-2019. [Online]. Available: http://msme.us/2012-2-1.pdf

[14] I. The MathWorks. (2019) Matlab coder. Last visited: 21-11-2019. [Online]. Available: https://www.mathworks.com/products/matlab-coder.html?s_tid=AO_PR_info