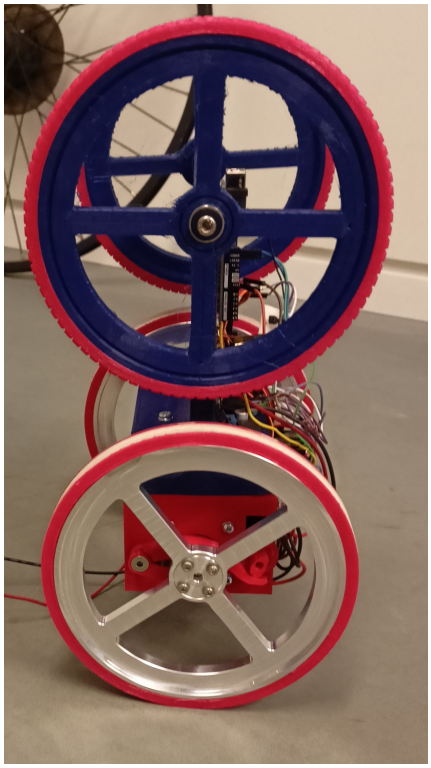
Real World Development and Test of a Mobile Hybrid Robot Platform.

A Master Thesis



ROB10 Group 1060

Robotics
Aalborg University

Robotics
Aalborg University
http://www.robotics.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Real world development and test of a mobile hybrid robot platform.

Theme:
Mobile Robotics

Project Period:
Master thesis Spring Semester 2022

Project Group:
ROB1060 2022

Participant(s):
Iñaki Pujol
Mark R. Blankensteiner

Supervisor(s):
Shahabodin Heshmati-Alamdari
Karl Damkjær Hansen

Standard Page Count: 89

Date of Completion: 1st June 2022

Abstract:

This project investigates how to design and construct a hybrid robot between a car and a Segway such that it allows the transition between the two modes of operation by producing a swing-up motion and balancing it on an inverted pendulum state. A testing scenario simulation has been developed to be able to test designs and features before trying them out in the real world. The simulation also has been used to obtain an estimation of the characteristics of the needed motors, which has been gotten as close as possible within the price and availability limitations. After testing, it has been proven that it is possible to perform a swing-up motion after achieving a certain velocity and that it is possible to stabilize the robot on an inverted pendulum state using different control systems, such as they can be PID, simple state feedback, or LQR. And it is shown that it is a good start in the development of this technology but it needs refinement and optimization on the design and part selection to be able to perform optimally and be able to be used in a real world scenario that would need this kind of robot.

Preface

This project is created by 10th semester students of Robotics at Aalborg University.

The code for the project can be seen here:
https://github.com/ipujol10/hybrid_robot.

The video showing the project's solution can be accessed here:

The citation style used in this report is IEEE of Style; the citation inside of the sentence indicates it refers to the same sentence, whereas the citation placed after the dot mark indicates that it refers to the whole paragraph. The links are marked with blue text and are clickable. This report uses '.' as decimal point and ',' as thousands separators in numbers. The reading should be done in a chronological order.

Many thanks to our supervisors, for providing us guidance, feedback and inspiration during the entire project. Also a big thanks to Jesper in the machine shop, for his guidance in the design of our metal parts as well as doing the fabrication of the parts. — ROB1060

22gr1060, Aalborg University, 1st June 2022

Iñaki P. Carmona
<ipujol20@student.aau.dk>

Mark R. Blankensteiner
<mblank16@student.aau.dk>

Contents

Abbreviation

DARPA Defense Advanced Research Projects Agency

DOF Degrees Of Freedom

Hz Hertz

IMU Inertial Measurement Unit

IO Input-Output

LQR Linear-Quadratic Regulator

MIMO Multi-Input Multi-Output

MPC Model predictive control

PID Proportional-Integral-Derivative control

ROS Robot Operating System

RPI Raspberry Pi

SAR Search And Rescue

SISO Single-Input Single-Output

UART Universal Asynchronous Receiver/Transmitter

URDF Unified Robot Description Format

Xacro XML Macros

1 - Introduction

The wheel and axle are one of the seven simple machines used since ancient times [1]. This mechanism allows the transformation of torque into force and vice versa [1]. In actuality, the concept of a wheel has been expanded from just the wheel and axle and it can be used to move objects like vehicles or robots.

Mobile robots normally use some kind of wheel to be able to move around. They could be categorized into three general groups: the standard wheels which could be the most similar to a car/bike and can be fixed or swiveling; the ball wheels which allows the movement in the 360º of the plane; and finally the omni wheels which also allow multi-directional movement. [2]

(a) Standard wheel

(b) Ball wheel

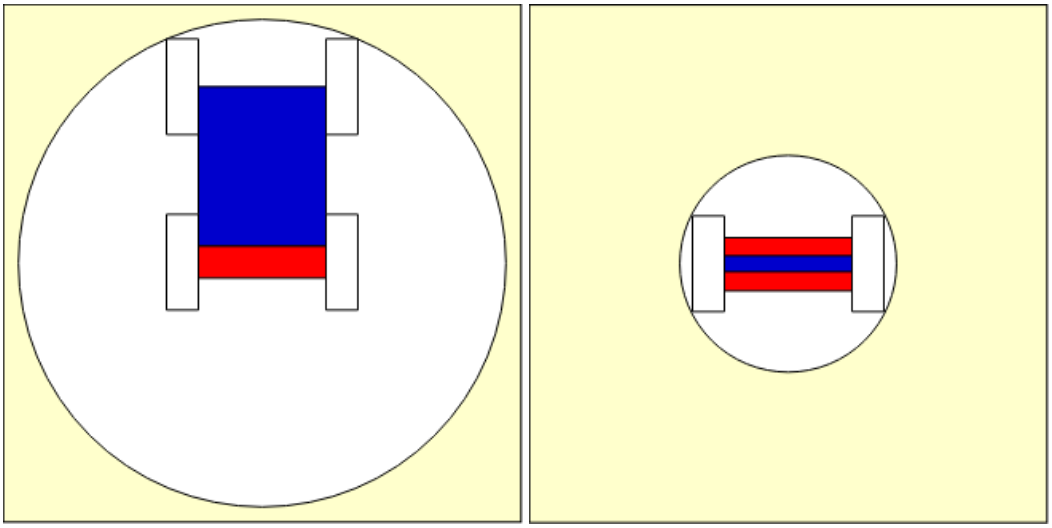
(c) Omni wheel

Figure 1.1: The types of wheels by category [2]

One of the most common wheels used, is the standard fixed wheels as they can be directly attached to the motor by the axle. The differential drive is obtained by using two parallel wheels that are powered by independent motors, this allows the robot to go straight (both wheels spinning at the same velocity) or turning (otherwise).

The differential drive is used widely on car-type robots, with two or four wheels and, at least, a pair of them powered. Another use is the Segway model which only uses the two wheels to balance itself [3, 4, 5]. Each of those with different advantages and disadvantages.

The four wheel robots have the advantage that, when static, they don't consume any energy as it is a stable position; but they also have a fairly big turning radius. In contrast, the Segway method can have a quite small turning radius; but takes energy all the time as it is in an unstable configuration and it needs to adjust itself continuously. The turning radius difference can be seen in figure 1.2.



(a) Turning radius of a four wheeled robot **(b)** Turning radius of a Segway robot

Figure 1.2: Comparison of turning radius for the different types of robots. Here the same robot approximate diagram is used to illustrate both systems. As seen in figure 1.2a, the turning radius is significantly bigger than in figure 1.2b.

1.1 Problem formulation

"How is it possible to have a robot driving in different configurations?"

2 - Problem analysis

In this chapter an analysis is done on "How is it possible to have a robot driving in different configurations?".

2.1 Background

In this section, an understanding of locomotion is carried out in various wheeled robotic types is discussed.

2.1.1 Differential Drive

The differential-driven robot is a widely used implementation, to perform locomotion in robotic solutions. The differential drive is used in various setups such as skid cart steering, Segways, tracked robots and the use vacuum cleaning robots. [6]

The concept of the differential drive is that there is a control function that allows, one side to move independently from the other side, this allows for steering as well as propulsion. This independence from the two sides can be done in various ways, it can be done by having two motors directly driving each side of the vehicle, applying more torque to one wheel than the other lets robot steering [6]. It can also be done as it is in some tracked vehicles, where one motor is applying torque on both sides of the vehicle's wheels, and a secondary steering motor is then used to supply more torque to one or the other side to control the steering [6].

One of the most used platforms of a differential drive robot is, the use of two motors connected to two wheels directly or through a set of gears, a typical component used in this configuration of the differential drive is a caster wheel that can move freely to balancing the platform.[5, 4]

There are also other ways to control a robot besides the differential drive, the robot turns either the front or rear wheels in the direction of motion to steer, or both front and rear wheels to steer together known as explicit steering.[7]

Skid car steering is yet a different way of controlling a robot is through the use of a skid cart model, this model is based around a four-wheel configuration, where all wheels

move independently, this means that the robot can move in all directions and turn around on its center axis. This directional control is done by turning the wheels in the opposite directions and at different velocities.[7]

Although skid steering pros is that it can steer in all directions, one of the cons is that it needs different tire configurations than a robot that is turning its wheels to steer, due to the wear on ordinary tires as the velocity of the wheels counters each other.[7]

2.1.2 Segway

As stated previously, a Segway is a two-wheeled vehicle that is capable of balancing itself to keep an upright position [3].

This transport method uses the same control approach as the classic inverted pendulum model. This is thanks to the fact that the Segway has a similar motion equations model to the inverted pendulum.

The kinematics of the Segway can be simplified into having two bodies (the handle and the wheels) connected through axles and thinking it in 2D [3].

The control of this device is made by trying to maintain the body angle. This means that when it is vertical, it is static and stable; but when it is tilted forward the controller tries to bring the wheels beneath it to keep it balanced causing a forward velocity. To go backward or brake, it would be the same as going forwards but tilting backward. To turn, it is just a matter to incline the handle to the side and the controller will produce a differential drive movement. [8]

The controller is feed by gyroscope readings regarding the inclination of the body [8] and the odometry to obtain the position of the vehicle. It is a fast controller to be able to make small and frequent adjusts to the angle of it [8].

Figure 2.1: Image of a Segway. [9]

3 - State of the Art

This section is going to discuss the already existing robotic technology, that can work in different configurations to adapt to different scenarios.

3.1 Ascento

Ascento Robotics is a company that has specialized in the development of a two-wheeled robot with a bend leg linking the robot which allows it to have a flexible configuration to adapt to almost any terrain. [10]

The basic idea of the robot is that it can go up and down stairs, drive autonomously and go up into a Segway drive position from the floor, pick itself up. [10]

Figure 3.1: Ascento Pro robot [10].

In the last iteration of the robot, they have been able to include enough features that allow it to perform different tasks. It can be used to produce 3D scans of the desired arias as it can be a construction site or the sewers. [10]

It can also be used in security tasks. It can perform predefined paths and with the sensors, it has included, it can produce this task [10]. Through the robot, the personnel can see through the cameras and spotlights and also hear through the microphones [10]. With enough data, it could be automatized with a neural network to detect any security bridge.

The other big task that it can perform is the Structure Assessment using Ground Penetrating Radars (GPR). Using the GPR technology can be possible to predict the risks on the

structures without destroying them. It can also be used for visual inspection with RGB or thermal cameras. [10]

3.2 DRC-HUBO+

One of the ways that the robot can move around in a human-created environment is if the robot takes the shape of a humanoid, with two legs; arms; hands (grippers), this can allow the robot to step over obstacles and climb stairs; grab door handles; using tools and more.[11] One competition by Defense Advanced Research Projects Agency (DARPA) by the U.S. Department of defense unintended created a push for these types of humanoid robots by creating a challenge where the robot has to navigate ruble, use different tools, with one of the tools, which were a jigsaw the robot has to autonomously cut a hole in a door.[11, 12, 13]

DRC-HUBO+ for the research team KAIST, is a bipedal robot that can move around in a human-created environment, that was competing in the DRAPA Search And Rescue (SAR) challenge, this robot uses a combination of wheels and the legs to move around. When the terrain allows for it the robot crouches down on its knees, as seen in fig. 3.2, in the lower legs of DRC-HUBO+ there have implemented a set of wheels with motors that make the robot expend less energy traveling through the area of concern.[12]

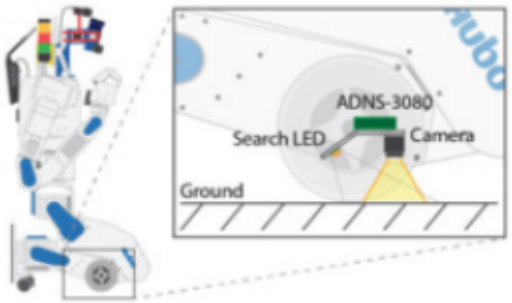


Figure 3.2: DRC-HUBO+ in a sitted position, in the pose DRC-HUBO+ is able to drive around on it wheels

The way that DRC-HUBO+ transitions between its two states walking and crouching is done by moving the center of mass of the DRC-HUBO+'s upper body, at the same time bending the legs at the knees and pushing the knees forward, till it lands on it knees. As seen in the fig 3.2 DRC-HUBO+ has two smaller wheels on each toe, when driving in four-wheel configuration the center of mass is placed in the middle of the 4 wheels.[12]

3.3 Hybrid Wheeled Jumping Robot

Traiko Dinev et al. have thought about taking a closer model to a standard 4-wheeled differential drive robot and providing it with different capabilities so it can adapt to different terrain conditions as it can be rough terrain or be able to jump over holes. This project was never carried out into a physical prototype and just tested in simulation. [14]

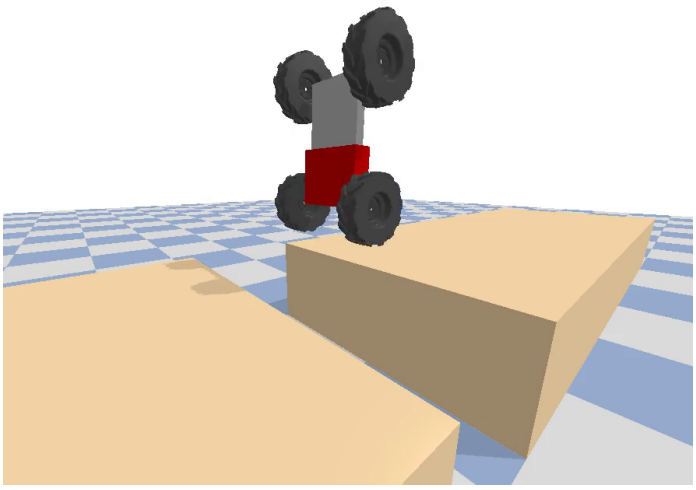


Figure 3.3: Image of the simulation model. In this frame, the robot is on the Segway drive and using the extensible body to jump. [14]

They have designed a four-wheeled robot with a pair of wheels independently powered, allowing a differential drive, and a prismatic joint on the two parts of the body (as seen in the figure 3.3). [14]

They have made the robot catch speed and fully brake to produce enough torque on the body to produce a swing-up motion and stabilize on a vertical position, drive while in a Segway mode and jump over a hole using the force produced by the prismatic joint. [14]

To control this hybrid robot, they used two methods: a PID controller and a Model Predictive Control (MPC). The MPC surprised them at some points as it used the extendable body to help balance itself. [14]

3.4 Conclusion

It is also seen that a four-wheel robot can be converted into a Segway if it is swung-up into an upright position. After analyzing both types of robots, it is clear that the main disadvantages of one are the advantages of the other and vice versa, which brings to the conclusion that if both configurations could be combined in a single robot, it could be

used to take advantage of the best of both of them.

One of the scenarios that this could be useful for, would be the SAR mission. In an urban terrain where a disaster has happened, tight spaces and corners can be found. With that in mind, a hybrid robot that could switch from a four-wheel robot to a Segway could have a low energy consumption to explore and a small turning radius for tight corners.[11, 15, 16]

Some theoretical studies have been performed on that matter already. For example, a paper that made a simulation of a four-wheeled robot with an extensible body. This allowed it to drive as a car; swing up into a Segway configuration; jump with the extension of the body. [14]

4 - Delimitations

In the Traiko Dinev et al. [14], the hybrid robot can drive in a differential drive state. Then it can gain speed to brake suddenly achieving a swing-up motion and stabilizing itself in an inverted pendulum/Segway state.

Once in the balancing state, the robot can drive around like a Segway. They have shown that it is stable enough to go through rough terrains with bumps and keep the balance.

The final state is the one that provides the robot with the jumping by releasing the prismatic joint with force enough to lift the robot from the ground and when it lands, it can balance itself again.

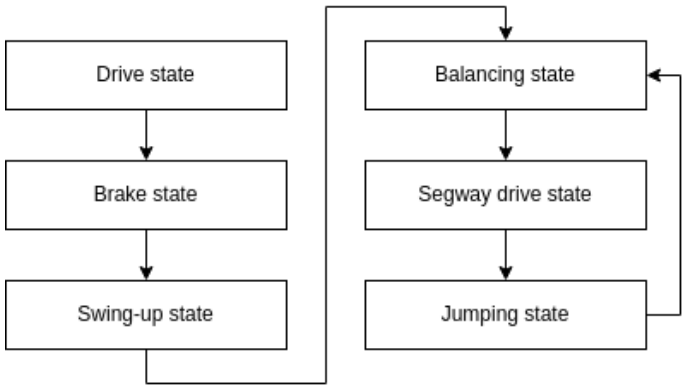


Figure 4.1: States that the robot on the Traiko Dinev et al. paper [14] goes through to be able to achieve all the requirements.

They used a simplified version of the motion equations. Instead of working on the three dimensions, they assumed a linear motion of the robot (just forwards and backward without turning) and simplified its dynamics.

Due to the limited amount of time; human resources; and economical limitations, it is not going to be possible to reproduce all the states seen in the figure 4.1 from the original paper. There is no real-world scenario where the cart can be tested in these early stages, the running and testing of the real world cart, will take place in laboratory settings.

As it can be seen each step further that the robot goes on the states from the figure 4.1, it needs all the others before achieving that state. This is why it has been decided that the way to start developing this project is from top to bottom of the figure 4.1. The reasonable

goal due to all the limitations is to aim to achieve up to the balancing point of the robot.

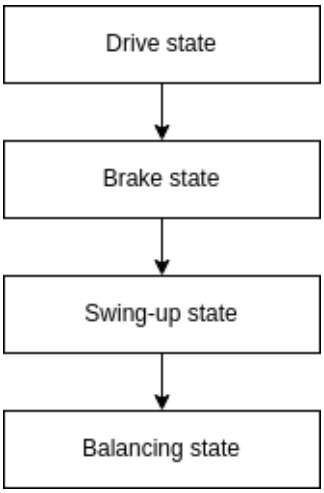


Figure 4.2: These are the states that the hybrid has to translate between after limiting the project.

The next coming chapters will go through the different aspects of how the project group developed the Hybrid Robot, it was an iterative process, where first basic model was implemented in a simulation, and this model was able to perform the drive and do the swing-up state as seen in the figure 4.2, this figure illustrates the different states that the Hybrid Robot has to be able to transition between.

In the next coming chapters, the solution that is made by the project group are named *"**Hybrid Robot**"* with capital letters.

5 - Final Problem Formulation

How can a hybrid robot between a car and a Segway be constructed to allow transition between the two modes of operation with a fixed body.

6 - Requirements

In this chapter, the project requirements are going to be stated.

6.1 Requirements

Req. 1 The cart must be able to drive.

Req. 2 The cart must be able to rotate the body of the cart over it's front or back wheels.

Req. 3 The cart must be able to balance, at an inverted pendulum state on it's front or back wheels.

Req. 4 The cart must be able to catch the body of the cart when rotating over either front or back wheels, and balancing the body of the cart in an inverted pendulum state.

7 - Technical Analysis

The purpose of this chapter is to analyze and understand the technical aspects of the dynamics and how to control the Hybrid Robot.

7.1 System Motion Equations

To understand how the robot reacts, works, and predict what can happen next, it is important to obtain the motion equations of the system. Those equations describe how the robot moves.

To obtain the motion equations, is done by using the *Lagrangian Mechanics* method [17]. They use the states of the system and its energy.

The energies of the system are the *kinetic* (**T**) and *potential* (**V**) energies. With those, the Lagrangian function of the states (q) is found [17]:

$$\mathbf{L} = T - V \tag{7.1}$$

Then it is calculated the *Euler-Lagrange equation* by doing the partial derivatives of the states (finding the jacobian) [17]:

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}} - \frac{\partial L}{\partial q} - Q^{ncons} = 0 \tag{7.2}$$

Then, taking into account that V is not depending on $\dot{q}$, the equation 7.2 can be rewritten as [17]:

$$\frac{d}{dt}\frac{\partial T}{\partial \dot{q}} - \frac{\partial T}{\partial q} + \frac{\partial Q}{\partial q} - Q^{ncons} = 0 \tag{7.3}$$

And finally, be able to write the dynamics on the standard matrix form [17]:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = Q^{ncons} \tag{7.4}$$

Where, in equation 7.4, $M(q)$ is the inertia matrix, $C(q,\dot{q})$ is the Coriolis matrix and $G(q)$ is the conservative forces matrix.

7.1.1 Initial Motion Equations

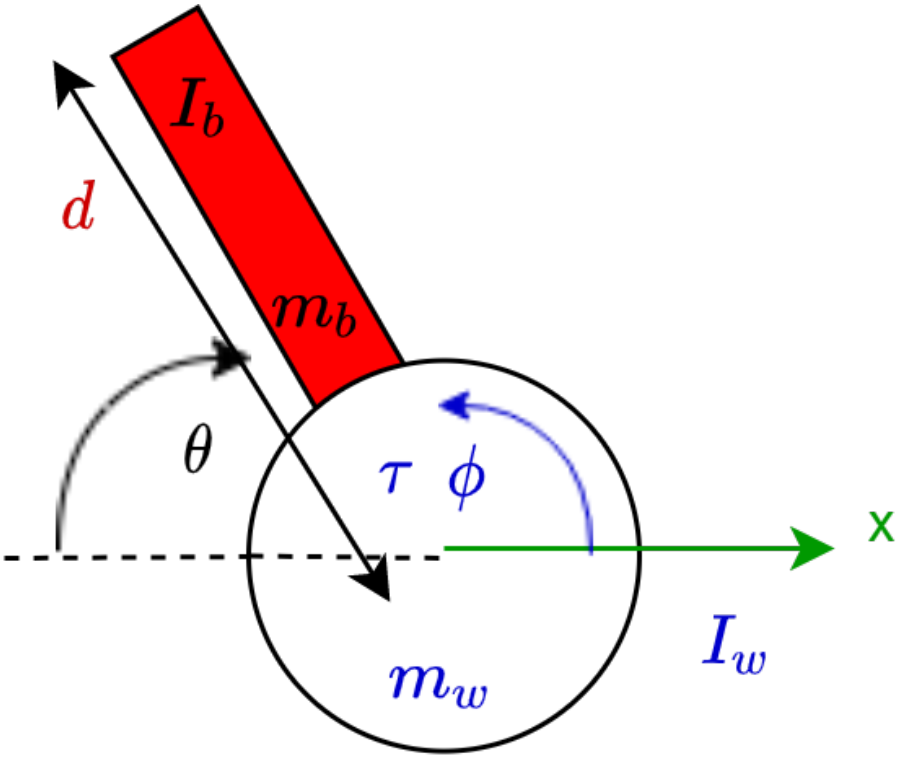


Figure 7.1: Diagram of the system to study. With the physical attributes of the system as the masses: the m_w as the mass of the front powered wheels and the m_b as the mass of the body (the rest of the cart); the inertias: the I_w is the inertia of the front wheels and the I_b is the inertia of the body; and the d as the distance from the center of the front wheels to the center of mass of the body. The system states are: x is the distance of the cart from the initial point; θ is the angle of the cart respect to the floor; ϕ is the turned angle of the front wheels respect to the starting point. And finally the only actuation on the system is the τ that represents the torque of the motors over the front wheels.

To obtain the motion equations of the figure 7.1, first, it is necessary to define the kinetic and potential energies:

$$\mathbf{T} = \frac{1}{2}m_w v_w^2 + \frac{1}{2}m_b v_b^2 + \frac{1}{2}I_w \omega_w^2 + \frac{1}{2}I_b \omega_b^2 \tag{7.5}$$

$$\mathbf{V} = m_b g d \sin(\theta) \tag{7.6}$$

The variables of the equations 7.5 and 7.6 are described in the figure 7.1 with the gravity constant $g = 9.81 m/s^2$. The v are the linear velocities and the ω are the angular velocities.

Then it is time to calculate and assign each velocity to 7.5

$$v_w = \dot{x} \tag{7.7}$$

$$x_b = x - d\cos(\theta) \rightarrow \dot{x}_b = \dot{x} + d\sin(\theta)\dot{\theta} \tag{7.8}$$

$$y_b = d\sin(\theta) \rightarrow \dot{y}_b = d\cos(\theta)\dot{\theta} \tag{7.9}$$

$$v_b^2 = \dot{x}_b^2 + \dot{y}_b^2 = \dot{x}^2 + d^2\dot{\theta}^2 + 2d\sin(\theta)\dot{x}\dot{\theta} \tag{7.10}$$

$$\omega_w = \dot{\phi} \tag{7.11}$$

$$\omega_b = \dot{\theta} \tag{7.12}$$

After plugging in L and V into the equation 7.1 and obtaining the system Lagrangian function, it is possible to use 7.2 to obtain the different Euler-Lagrangian equations.

When deriving respect the state x:

$$\frac{\partial L}{\partial \dot{x}} = m_b(\dot{x} + d\sin(\theta)\dot{\theta}) + m_w\dot{x} \tag{7.13}$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{x}} = m_w\ddot{x} + m_b(\ddot{x} + d\cos(\theta)\dot{\theta}^2 + d\sin(\theta)\ddot{\theta}) \tag{7.14}$$

$$\frac{\partial L}{\partial x} = 0 \tag{7.15}$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{x}} - \frac{\partial L}{\partial x} = Q_x^{ncons} = m_w\ddot{x} + m_b(\ddot{x} + d\cos(\theta)\dot{\theta}^2 + d\sin(\theta)\ddot{\theta}) \tag{7.16}$$

When deriving respect the state θ:

$$\frac{\partial L}{\partial \dot{\theta}} = m_b d^2\dot{\theta} + m_b d\sin(\theta)\dot{x} + I_b\dot{\theta} \tag{7.17}$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}} = (m_b d^2 + I_b)\ddot{\theta} + m_b d\sin(\theta)\ddot{x} + m_b d\cos\theta\dot{x}\dot{\theta} \tag{7.18}$$

$$\frac{\partial L}{\partial \theta} = m_b d\cos(\theta)(\dot{x}\dot{\theta} - g) \tag{7.19}$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}} - \frac{\partial L}{\partial \theta} = Q_\theta^{ncons} = (m_b d^2 + I_b)\ddot{\theta} + m_b d\sin(\theta)\ddot{x} + g m_b d\cos\theta \tag{7.20}$$

When deriving respect the state ϕ:

$$\frac{\partial L}{\partial \dot{\phi}} = I_w\dot{\phi} \tag{7.21}$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\phi}} = I_w\ddot{\phi} \tag{7.22}$$

$$\frac{\partial L}{\partial \phi} = 0 \tag{7.23}$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\phi}} - \frac{\partial L}{\partial \phi} = Q_\phi^{ncons} = I_w\ddot{\phi} \tag{7.24}$$

7.1.2 Simplified Motion Equations

The equations presented in this section have been obtained by discarding a state as it is assumed no slip between the ground and the wheel.

By having this assumption, the position of the robot (x) is directly associated with the turning of the wheels (ϕ).

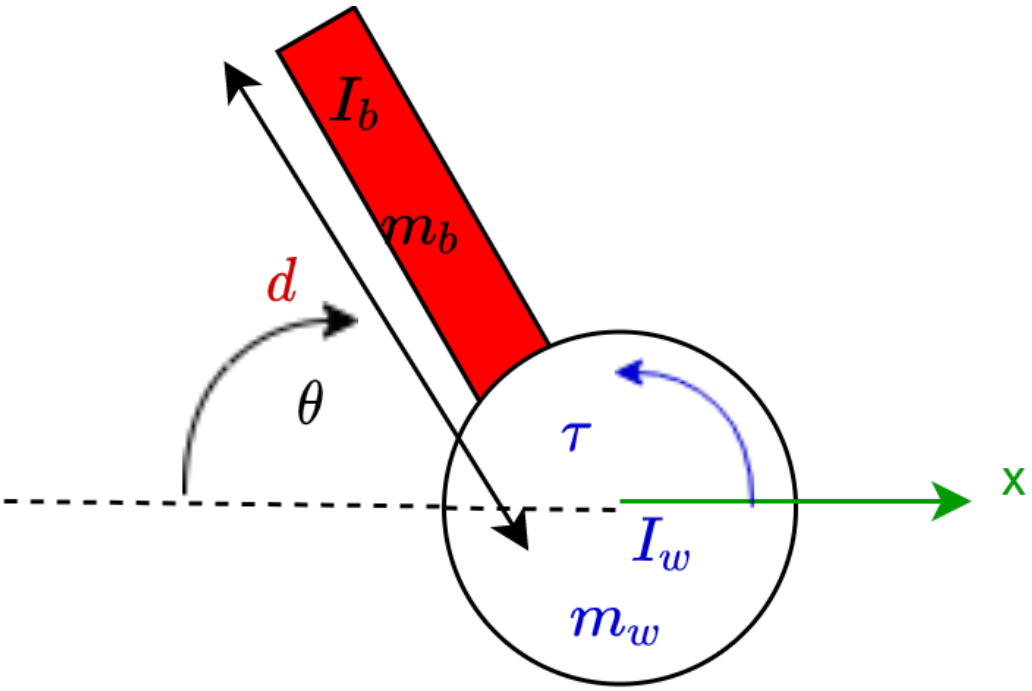


Figure 7.2: Diagram of the simplified system to study. The physical attributes are the same as the one in the figure 7.1 but having disregarded the state ϕ as it is assumed no-slip getting the relation $x = -R\phi$ being R the radius of the front wheels.

The kinetic and potential energies are the same as 7.5 and 7.6 and it will just change how the angular velocity of the wheel is defined, as there is the no-slip condition (being R the radius of the wheel):

$$x = -R\phi \tag{7.25}$$

The equation 7.11 becomes:

$$\omega_w = -\frac{\dot{x}}{R} \tag{7.26}$$

After plugging in L and V into the equation 7.1 and obtaining the system Lagrangian function, it is possible to use 7.2 to obtain the different Euler-Lagrangian equations.

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}} - \frac{\partial L}{\partial \theta} = (m_b d^2 + I_b)\ddot{\theta} + m_b d \sin(\theta)\ddot{x} + g m_b d \cos(\theta) \tag{7.27}$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{x}} - \frac{\partial L}{\partial x} = (m_b + m_w + \frac{I_w}{R^2})\ddot{x} + m_b d \sin(\theta)\ddot{\theta} + m_b d \cos(\theta)\dot{\theta}^2 \tag{7.28}$$

To be able to work with the equations 7.27 and 7.28, it is helpful to have them stored in a matrix form as it will help to do some operations. And it is important to define the state vector and its derivatives:

$$q = \begin{bmatrix} \theta \\ x \end{bmatrix} \rightarrow \dot{q} = \begin{bmatrix} \dot{\theta} \\ \dot{x} \end{bmatrix} \rightarrow \ddot{q} = \begin{bmatrix} \ddot{\theta} \\ \ddot{x} \end{bmatrix} \tag{7.29}$$

Then, taking the equation 7.4 and rearrange it to a better form for the purpose of the project:

$$M(q)\ddot{q} + h(q, \dot{q}) = g_q u \rightarrow \ddot{q} = -M^{-1}h + M^{-1}g_q u \tag{7.30}$$

In the equation 7.30 it is important to define that u is the control input of the system and, in this case, it is the torque of the motors that can be seen in the figure 7.2.

With $M(q)$, $h(q, \dot{q})$ and g_q on 7.30 being:

$$M = \begin{bmatrix} m_b d^2 + I_b & m_b d \sin(\theta) \\ m_b d \sin(\theta) & m_b + m_w + I_w / R^2 \end{bmatrix} \tag{7.31}$$

$$h = \begin{bmatrix} g m_b d \cos(\theta) \\ m_b d \cos(\theta)\dot{\theta}^2 \end{bmatrix} + \begin{bmatrix} \alpha\dot{\theta} \\ \beta\dot{x} \end{bmatrix} \tag{7.32}$$

$$g_q = \begin{bmatrix} I_w \\ -I_w \end{bmatrix} \tag{7.33}$$

On the equation 7.32, the second term is related to the friction coefficients. The equation 7.1.2 has input on both terms because the torque made by the motor affects both parts of the system equally but with a contrary sign.

It is also important to explain how it is obtained the matrix g_q on the equation 7.1.2: the standard way to represent control inputs is by using forces or torques, but in this case, it was possible to control the motors by velocity, the control was modified to adapt to the necessities.

It is known that a torque applied to the body produces an angular acceleration following the next equation, where τ is the torque, I is the inertia of the body, and α the angular acceleration:

$$\tau = I\alpha \tag{7.34}$$

Being that acceleration is much easier to relate to a velocity than a torque, it was decided to replace the torque with the equation 7.34 and use the angular acceleration of the wheel as the control input. This change has been possible since the inertia of the body is greater than the one on the wheels.

To obtain something similar to a standard dynamic system form of $\dot{x} = Ax + Bu$, it is necessary to rearrange everything and have the states and the first derivative of them in the state vector k:

$$k = \begin{bmatrix} \theta \\ x \\ \dot{\theta} \\ \dot{x} \end{bmatrix} \rightarrow \dot{k} = \begin{bmatrix} \dot{\theta} \\ \dot{x} \\ \ddot{\theta} \\ \ddot{x} \end{bmatrix} \tag{7.35}$$

$$\dot{k} = \begin{bmatrix} \dot{q} \\ -M^{-1}h \end{bmatrix} + \begin{bmatrix} 0 \\ M^{-1}g_q \end{bmatrix} u \tag{7.36}$$

7.2 Linearization

For a lot of the most common control methods, the dynamic systems must be linearized. The linearization is performed in a working point (normally an equilibrium point).

The Taylor Expansion, when $x = \bar{x}$ and the exponent (n) being the derivative order, is:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(\bar{x})}{n!}(x - \bar{x})^n \tag{7.37}$$

The general equation for linearization comes from the Taylor Series up to the first order, so it is linear:

$$f(x, u) \approx f(\bar{x}, \bar{u}) + \frac{\partial f(\bar{x}, \bar{u})}{\partial x}(x - \bar{x}) + \frac{\partial f(\bar{x}, \bar{u})}{\partial u}(u - \bar{u}) \tag{7.38}$$

In the equation 7.38 the "bar" variables ($\bar{x}$ and $\bar{u}$) mean that they are the fixed value of the working point.

When the linearization equation 7.38 is applied to a system in the form of the equation 7.36, it will look similar to 7.39 and 7.40 for continuous and discrete time, respectively.

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx \end{aligned} \tag{7.39}$$

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k \\ y_k &= Cx_k \end{aligned} \tag{7.40}$$

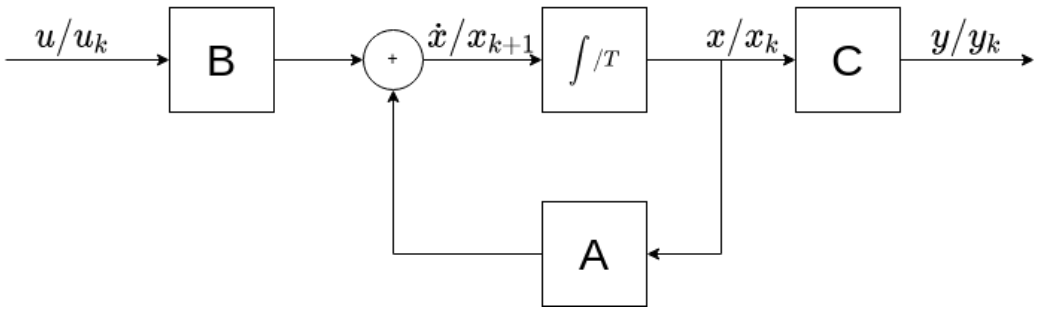


Figure 7.3: General representation of a linear system. In continuous time (equation 7.39) and discrete time (equation 7.40)

7.2.1 Transformation from Continuous to Discrete Time

Nowadays almost all the control is done by computers, which work in a discrete-time, that is why is also called digital control. [18]

In a digital control is very important to know the sample time (T_s) at which the controller works as it is a factor that modifies the systems while converting them from continuous to discrete time [18]. The subscripts $\cdot_c$ and $\cdot_d$ mean continuous and discrete, respectively:

$$x(k+1) = e^{A_c T_s} x(k) + \int_0^{T_s} e^{A_c \tau} d\tau B_c u(k) \tag{7.41}$$

$$A_d = e^{A_c T_s} \tag{7.42}$$

$$B_d = \int_0^{T_s} e^{A_c \tau} d\tau B_c \tag{7.43}$$

If the system is uniquely determined by A [19], meaning that A_c is invertable, the equation 7.43 can be expressed as:

$$B_d = A_c^{-1}(e^{A_c T_s} - I)B_c \tag{7.44}$$

And, if it is used the approximation $\dot{x}(t) = \frac{x(t+T_s)-x(t)}{T_s}$ with $t = kT_s$, both A_d and B_d can be simplified to:

$$A_d = I + A_c T_s \tag{7.45}$$

$$B_d = B_c T_s \tag{7.46}$$

7.2.2 Stabilize in an Inverted Pendulum

The linearization for this section is made on the marginal stable point of the vehicle being vertical 7.2. This point is:

$$\begin{bmatrix} \theta \\ x \\ \dot{\theta} \\ \dot{x} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{7.47}$$

For the motion equations used in the section 7.1.2, having $\alpha = \beta = 0$ and using the values in the table E.1, the continuous time matrices would be:

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 47.2747 & 0 & 0 & 0 \\ -1.3443 & 0 & 0 & 0 \end{bmatrix} \tag{7.48}$$

$$B = \begin{bmatrix} 0 \\ 0 \\ 0.09687 \\ -0.003311 \end{bmatrix} \tag{7.49}$$

To control the system with a computer, it is needed to convert the system to the discrete region. Using the *Matlab* function to perform the task and using a $T_s = 1/100s$ ($100Hz$), the following matrices are obtained:

$$A = \begin{bmatrix} 1.0024 & 0 & 0.0100 & 0 \\ 0 & 1 & 0 & 0.0100 \\ 0.4731 & 0 & 1.0024 & 0 \\ -0.0135 & 0 & 0 & 1 \end{bmatrix} \tag{7.50}$$

$$B = \begin{bmatrix} 0 \\ 0 \\ 9.6949e-4 \\ 0 \end{bmatrix} \tag{7.51}$$

7.3 Control Methods

This section is going over the theory behind the different control methods used in this project.

7.3.1 PID

When something has to be controlled automatically, as it can be a robot or any other process, the first and more simple way that comes to mind is to use a PID.

The beauty of this type of controller comes from its simplicity of it, as well as its limitations. Normally a PID is used on Single-Input Single-Output (SISO) systems due to its simplicity of it, but applying some more advanced design methods, it can also be used on a Multi-Input Multi-Output (MIMO) systems [20].

As the name specifies, there are *Proportional*, *Integral* and *Derivative* stages (which will be explained further) and, normally, ordered from more important to least.

It all starts as any controller: by calculating the error to the reference (objective to achieve) or just stabilizing (bring all the states to 0). With the error obtained, control input is computed by passing it through the **constant** gains of the PID.

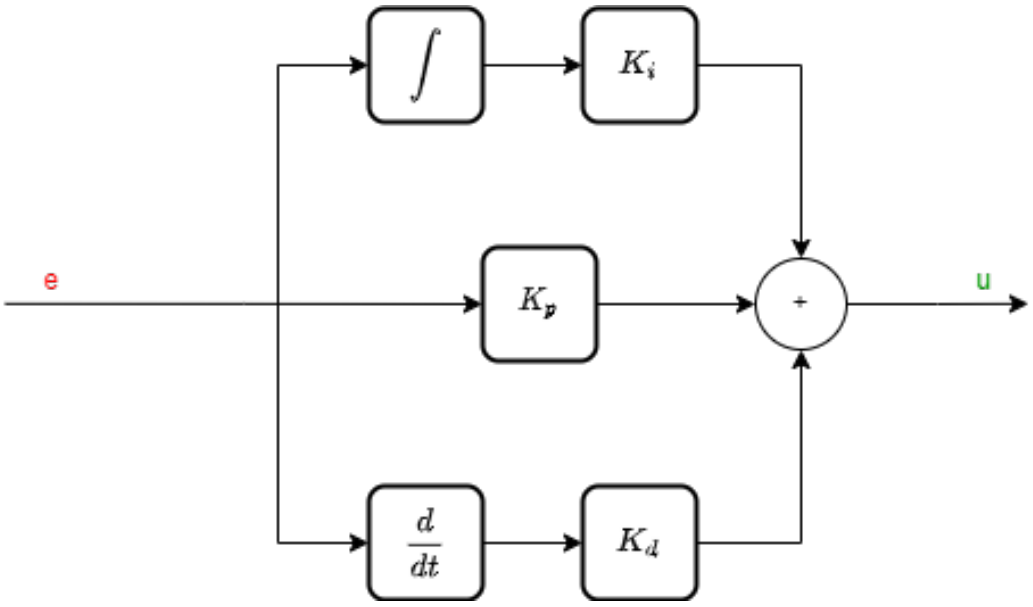


Figure 7.4: Diagram of a PID. The e is the error calculated previously and u is the output of the PID. Then there are K_x which have the subscript related to *proportional*, *integral* and *derivative*

The most important part of a PID is the ***Proportional*** gain as it uses the current error and scales it to produce a u input for the system. Increasing the magnitude of the K_p, usually increases the speed of response [20]. It is the present error corrector.

The next term is the ***Integral*** gain. This term is used to be able to overcome the steady-state error that can occur in some systems by external perturbations. To achieve that, it uses an integrator on the error to be able to keep track of the error over time. It is the past error corrector.

In the figure 7.5 it can be seen that there is an error and it is not going to disappear just by using a proportional gain. But when the integral gain is applied, the PID output starts to compensate for the offset until it disappears. It causes oscillations as it will over or under shoot until when the area under the error curve is canceled out [20].

The cause of the steady-state error is, as stated before, an external perturbation as it can

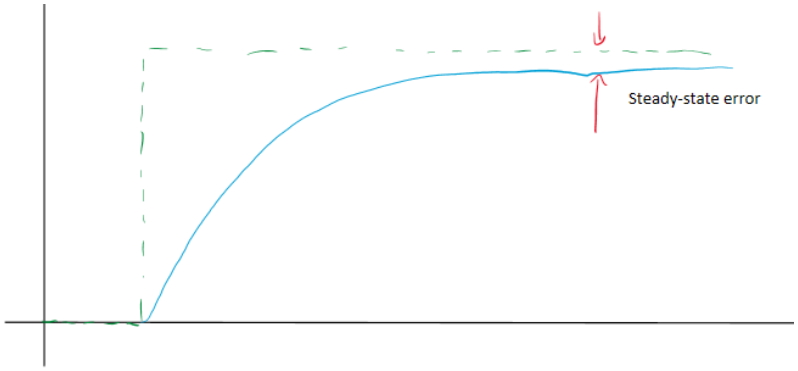


Figure 7.5: The error that is left constant when time tends to infinity, this figure is an illustration.

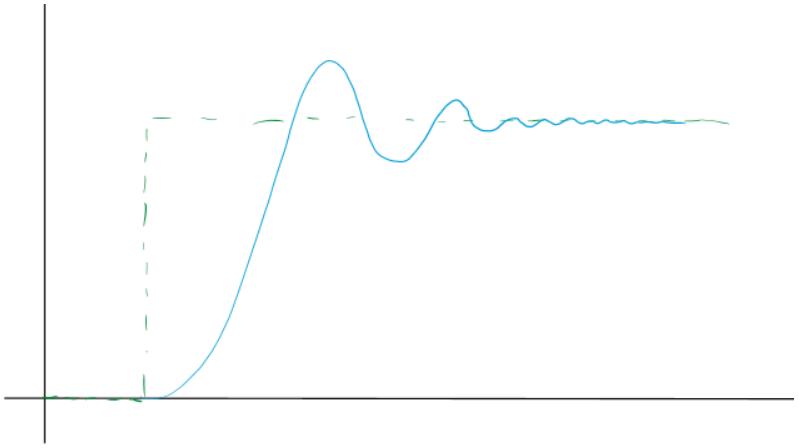


Figure 7.6: The effect on a system when an integral action is applied, this figure is an illustration.

be the gravity on a drone. The steady-state error will be the point where the input to the propellers due to the proportional gain is equal to the gravitational acceleration that will cause the drone to be in an equilibrium.

The last term is the ***Derivative*** gain. This is used to prevent an overshoot on the control output. To achieve that, it uses a derivative on the error to try to overcome future errors. If the derivative part detects that the error is decreasing too fast, it will try to compensate for it, compensate the tend of the error [20]. It is the future error corrector.

Parameter	**Steady-state error**	**Speed**	**Stability**
K_p	Reduces	Increases	Decreases
K_i	Eliminates	Reduces	Increases*
K_d	No effect	Increases	Increases*

Table 7.1: Effects of adjusting each PID gain individually [20]. *If they are too big they can decrease the stability.

As said in the table 7.1, having a K_i too big can cause the oscillations to increase over time, and it will cause instability. In the K_d case, it can cause bad performance if there is a noisy

error or if the magnitude is too big as it can lead to instability.

7.3.2 State Feedback

State feedback is a control method that uses the full state vector of the linear system to compute the control input. In case that the output of the system (y from the figure 7.3) is not the unaltered states (x from the figure 7.3), the states have to be estimated. [21]

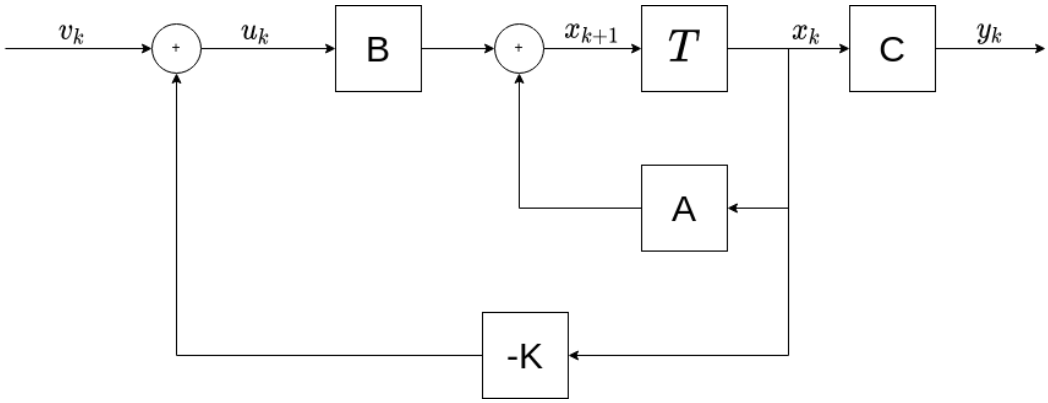


Figure 7.7: Representation of a linear state feedback system. Inspired by [21]

Using the diagram in figure 7.7 it can be seen that the control input is [21]:

$$u_k = -Kx_k + v_k \tag{7.52}$$

If then, the equations 7.40 and 7.52 are combined, a new system is found [21]:

$$x_{k+1} = Ax_k + B(-Kx_k + v_k) = (A - BK)x_k + Bv_k \tag{7.53}$$

Producing a new state matrix on the close-loop [21]:

$$A_{cl} = A - BK \tag{7.54}$$

Ending with the new system as [21]:

$$\begin{aligned} x_{k+1} &= A_{cl}x_k + Bv_k \\ y_k &= Cx_k \end{aligned} \tag{7.55}$$

The control of the state feedback is done by selecting a K gain that produces an A_{cl} gain stable. To obtain the dimensions of the K matrix it is important to look at the dimensions

of B^{mxn} as they are going to be transposed: K^{nxm} producing a matrix M^{mxm} with the same dimensions as A.

A common way to design a state feedback control is by using the *poles placement* method.

7.3.2.1 Linearization

When the system has to be linearized, it becomes a problem like [22]:

$$\left.\begin{aligned} \dot{\hat{x}} &= A\hat{x} + B\hat{u} \\ \hat{x} &= x - \bar{x} \\ \hat{u} &= u - \bar{u} \end{aligned}\right\} \tag{7.56}$$

In the equation 7.56 the $\bar{x}$ is the linearized point and the $\bar{u}$ the control input needed to be stable at the operating point. In the case of a static equilibrium point, $\bar{u} = \{0\}$; contrary to this case, to provide an equilibrium point the system would be provided with a constant input or acceleration.

Having the input control (u) in state feedback as equation 7.52, the control signal that the system has to receive in a linearized model becomes [22]:

$$u = \bar{u} - K(x - \bar{x}) \tag{7.57}$$

7.3.3 Poles Placement

The poles of a system determine its dynamics.

The poles of a system would be found by obtaining the eigenvalues of the linear matrix of it [23]:

$$det(\lambda I - A) = 0 \tag{7.58}$$

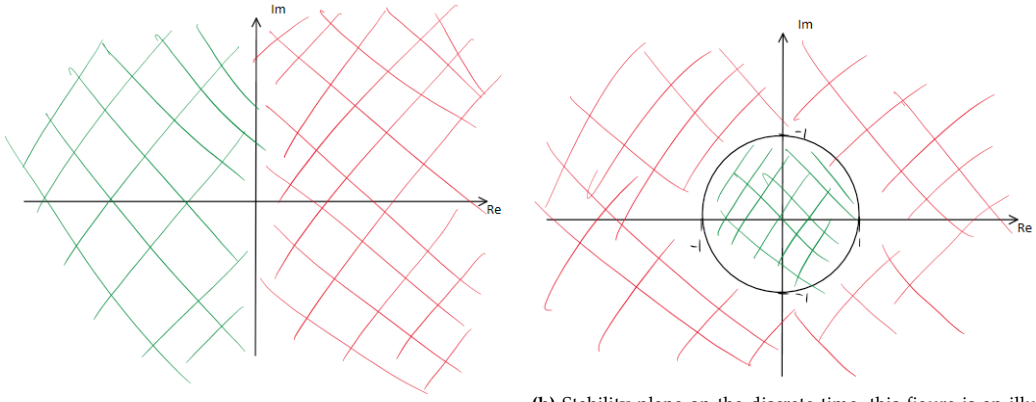
On the equation 7.58, A is the matrix or system that the eigenvalues would be found; I is the identity matrix; λ are the eigenvalues once the equation is solved.

If any of the poles of the system are in the unstable zone (red zone), the system becomes unstable. If the system is unstable, the states, over time, tend to $\pm\infty$.

The job of the control matrix K (section 7.3.2) is to bring all the poles into the stable zone and to place them so the system reacts as desired.

The method to place poles is done by obtaining the desired characteristic polynomial [21]:

$$\prod_{i=0}^{N} (\lambda - pole_i) \tag{7.59}$$



(a) Stability plane on the continuous time, this figure is an illustration.

(b) Stability plane on the discrete time, this figure is an illustration.

Figure 7.8: Comparison between the stability of the continuous and discrete time. On the 7.8a figure, the stability comes when the real part of the poles are < 0; on the 7.8b figure, the stability comes when the poles are inside the unit circle ($|pole| < 1$).

Then obtain the characteristic polynomial of the closed-loop system (A_{cl}):

$$det(\lambda I - (A - BK)) = det(\lambda I - A_{cl}) \tag{7.60}$$

Then compare and select K such that both of them become the same polynomial.

Matlab has a built-in function that allows the user to find the K gain that satisfies the placement of the poles.

```
% Obtain the control gain
K = place(A, B, p);
```

Code 7.1: Example of how to obtain the control gain with Matlab. 'p' are the desired poles, A and B are the system matrices and K is the control matrix.

7.3.4 Observer

A lot of times is not worth it or impossible to obtain every single state of a system. When this occurs and the measurements of the state are required (either to be able to control or to be able to show them), it is required to estimate them. [21, 24]

To estimate the states used in an observer, which takes a model of the system and uses the control input u, an output y, and a gain L to get an estimation of the states (figure 7.9). Then, instead of using the measured state vector x, this is substituted by the estimated $\hat{x}_k$.

The equations of the observer are derived from the diagram in the figure 7.9 [24]:

$$\begin{aligned} \dot{\hat{x}} &= A\hat{x} + Bu + L(y - \hat{y}) \\ \hat{y} &= C\hat{x} \end{aligned} \tag{7.61}$$

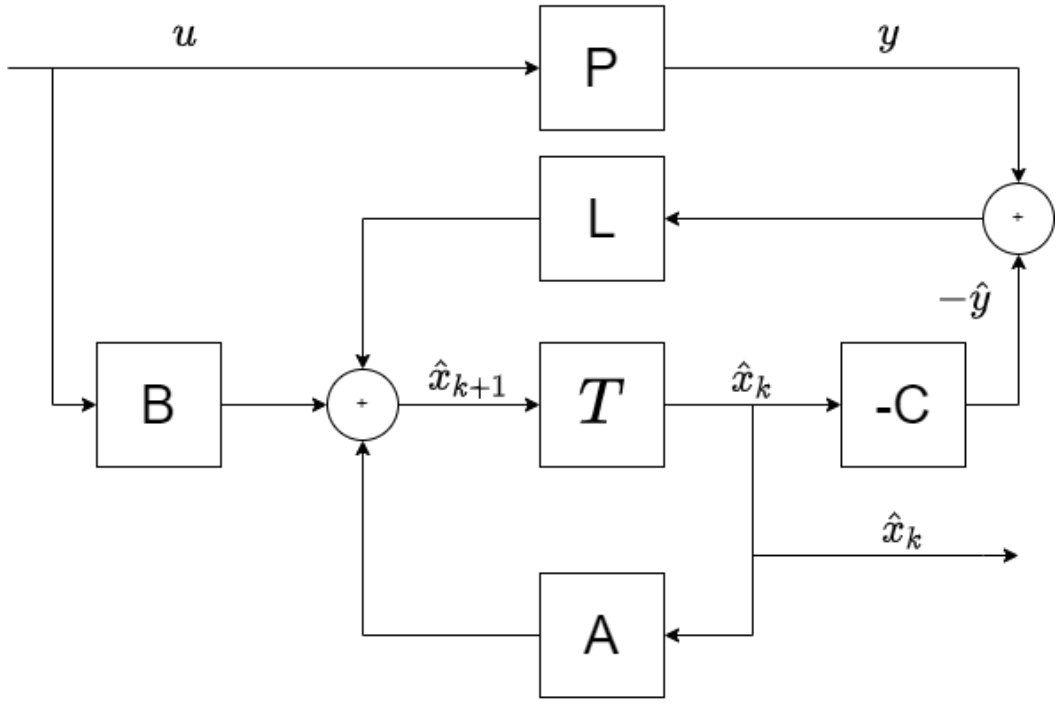


Figure 7.9: Representation of an observer. P is the plant or system as shown in the figure 7.3; A, B, and C are the model of the system; and L is the observer gain. $\hat{x}_k$ is the estimated state vector.

7.3.4.1 Obtaining the L Gain of the Observer

Obtaining the gain for the state estimator is similar to controlling a system: bringing the error to 0.

In this case, instead of taking the error between the reference and the output, is between the output and the estimated output (as seen in the figure 7.9) [24]:

$$e = x - \hat{x} \tag{7.62}$$

Taking the equations of the system and the observer and getting the error, the next equation is obtained:

$$\begin{aligned} \dot{e} &= (A - LC)e \\ e_{k+1} &= (A_L C)e_k \end{aligned} \tag{7.63}$$

The design of the L gain can be made by pole placement (section 7.3.3) or Linear Quadratic Regulator (LQR) (section 7.3.5), explained later.

Normally the dynamics of the observer are faster than the real system to converge faster into a good estimate. Faster dynamics means: poles further left on the complex plane for the continuous-time; poles closer to (0,0) in the discrete-time.

```
% Obtain the observer gain
L = place(A', C', p)';
```

Code 7.2: An example of how to obtain the observer gain with Matlab. 'p' are the desired poles, A and C are the system matrix and L is the control matrix. As seen here, the C matrix is used instead of B. A and C are transposed as well as the resultant gain.

In the code 7.2, it is seen how to design an observer with Matlab by having A, C, and the desired poles for the estimator.

7.3.5 LQR

The LQR is one of the most used optimal control strategies for a full state feedback. [22]

The control model for the LQR is the same as in the equation 7.52, just designing an optimal K gain (figure 7.7) instead of placing the poles "arbitrarily".

This method is basically an optimization problem, as the goal is to define a cost function and minimize it [22]:

$$J = \int_0^\infty \left[x^T Q x + u^T R u\right] dt, \quad Q = Q^T \succeq 0, R = R^T \succ 0 \tag{7.64}$$

In the equation 7.64 is shown the cost function (J) used to perform the optimization of the feedback and then two other matrices: Q and R.

Q and R are weights on how each state and input affect the cost function. They are diagonal matrices, meaning that each element on the diagonal represents the contribution to the cost while they are not 0. $Q \succeq 0$ means that the matrix is positive semi-definite and $R \succ 0$ means that it is strictly positive definite [25].

The contribution to the cost function can be explained as: if a state has a big weight, it means that the controller will try to converge it to 0 faster than other with smaller contribution. In the contrary, if a control input has a big contribution, the control will limit the use of it, useful if an actuation is expensive. [21]

Knowing the cost function and having decided the Q and R gains, it is time to find the optimal solution by solving the Hamiltonian [22]:

$$\forall x, \quad 0 = \min_u \left[x^T Q x + u^T R u + \frac{\partial J^*}{\partial x}(Ax + Bu)\right] \tag{7.65}$$

And being J^* and its derivative [22]:

$$\begin{aligned} J^*(x) = x^T S x, \quad S = S^T \succeq 0 \\ \frac{\partial J^*}{\partial x} = 2x^T S \end{aligned} \tag{7.66}$$

The final control feedback becomes [22]:

$$u = -R^{-1}B^T Sx = -Kx \rightarrow K = R^{-1}B^T S \tag{7.67}$$

As seen in the equation 7.67, the control gain depends on R and B already defined and S. S is the gain that is needed to be found so the equation 7.65 is minimized at the same time that the closed-loop poles are stables.

To solve the S gain there is the algebraic *Riccati equation* [22]:

$$SA + A^T S - SBR^{-1}B^T S + Q = 0 \tag{7.68}$$

As seen in the section 7.3.3, Matlab had build-in functions to design controllers. In this case, it also has one to find the LQR:

```
% Obtain the control gain
K = lqr(A, B, Q, R);
```

Code 7.3: Example of how to find the control gain with Matlab. A and B are the system and Q and R are the weights.

7.3.5.1 Discrete Time

When working in the discrete world, some changes have to be made to adapt for the LQR design.

The equations 7.64, 7.65 and 7.66 becomes [22]:

$$\min \sum_{n=0}^{N-1} x_n^T Q x_n + u_n^T R u_n, \quad Q = Q^T \succeq 0, R = R^T \succ 0 \tag{7.69}$$

$$J(x, n-1) = \min_u x^T Q x + u^T R u + J(Ax + Bu, n) \tag{7.70}$$

$$J(x, n) = x^T S_n x, \quad S_n = S_n^T \succ 0 \tag{7.71}$$

In the end the control gain becomes [22]:

$$K = (R + B^T S_n B)^{-1} B^T S_n A \tag{7.72}$$

And S comes from the discrete *Riccati equation* [22]:

$$S = Q + A^T SA - (A^T SB)(R + B^T SB)^{-1}(B^T SA) \tag{7.73}$$

Obtaining the control gain in Matlab, is simple and it also has a function for it:

```
% Obtain the control gain
K = dlqr(A, B, Q, R);
```

Code 7.4: Example of how to find the control gain with Matlab. A and B are the system and Q and R are the weights.

7.3.5.2 Observer

In case it is not possible to measure all the states of the system, it is necessary to estimate them by using an observer.

The design of the observer can be made by using the same method as the stabilization, as it already was done in the section 7.3.4:

```
% Get the observer gain in continuous time
L = lqr(A', C', Q, R)';

% Get the observer gain in discrete time
L = dlqr(A', C', Q, R)';
```

Code 7.5: Example of how to find the observer gain with Matlab. Using the same method as to find the control but having the same changes as in the code 7.2.

8 - Implementation and Design of the Hardware

The design and development of the Hybrid Robot were done in multiple steps, a flowchart describing the development of the Hybrid Robot, can be seen in the figure 8.1.

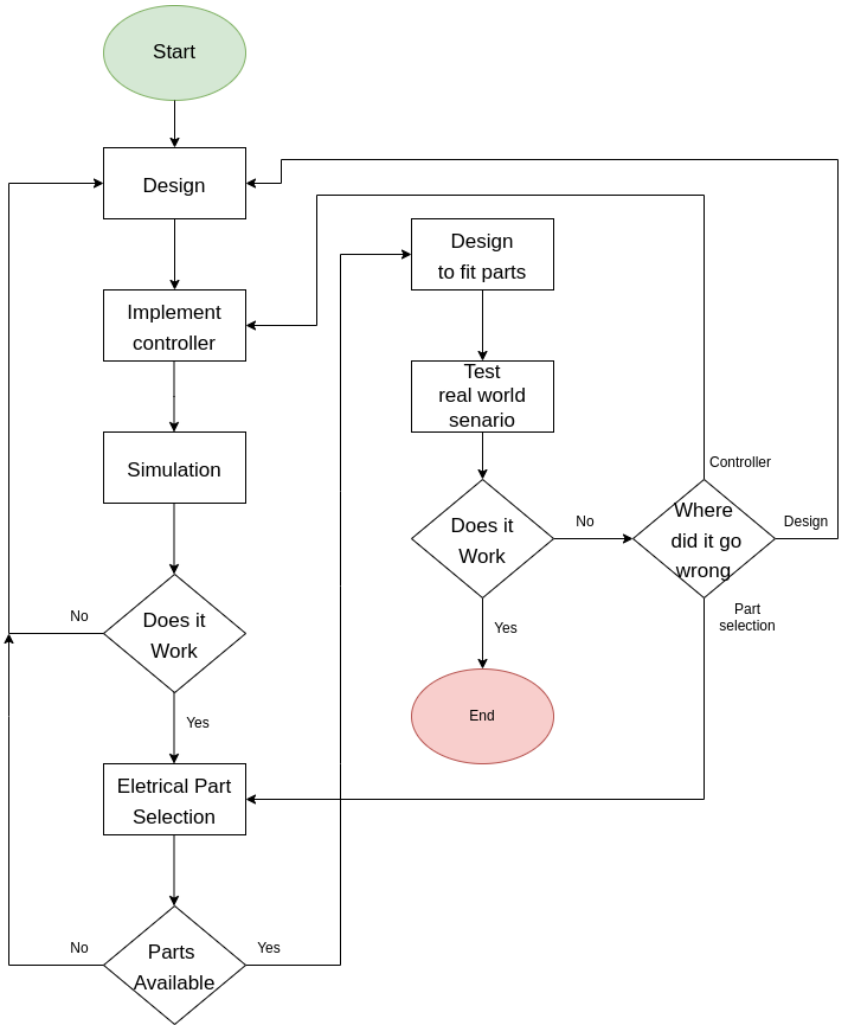


Figure 8.1: This flowchart illustrates the process that was used to develop the Hybrid Robot.

8.1 Body of the Hybrid Robot

The first thing to be design was the body of the Hybrid Robot, this was due to all other parts of the Hybrid Robot had to be attached to the body in the simulation. The body was made in two separate sections, a section that houses the motor and a box section that have the rear wheels attached. The reasons for this was, that the implementation of the various inertias in URDF would be eased, the use of basic shapes to calculate the sections' inertias. Another reason was that initial implementation of the Hybrid Robot's body was larger than the build area of the 3D printer available, this also allow to make the body more modular, such that if a changes to a section did not have to affect the design of the other sections.

The section of the body that changed the most was the motor section, the first iteration, was after the Hybrid Robot was shown in Gazebo, that the robot was able to preform the swing-up state and land upside down. After the successful swing-up it was time to select motors, the choice of the motors would determine the size of the first iteration of the motor section. The new motor section was implemented in Gazebo to test if the changes had any effect on the swing-up, which it did not.

The second major change to the motor section was during the discovery that the motors could not stop the wheels during the test of the swing-up state in real life. Here there were two options, one to buy better motors which will cost money, the second is that delivery of new parts takes time, which is just as costly as the money in this project. This lead to an implementation of brakes, this brake system that was develop was implemented into the motor section. The final iteration of the motor section inertia were found in SolidWorks where it is possible do an assembly of multiple parts to find most precise inertia of the section, this requite a model of the motors which was found on GrabCAD. This is a website that has models made to be used in a CAD software, the assembly of the motor section can be seen the figure 8.2.

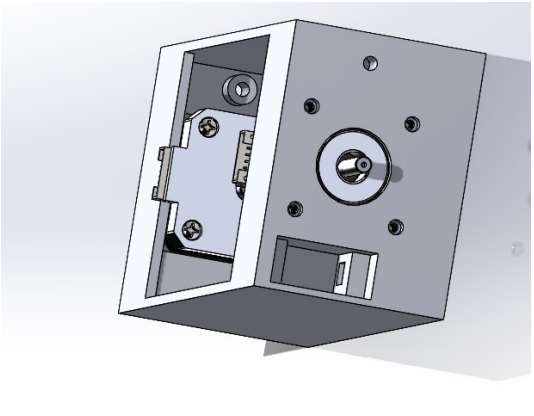


Figure 8.2: Assembly of the Hybrid Robot's motor section. The square hole is for the MG90s servo. Doing an Assembly like this, can catch errors before production.

The only major change, that was done to the section where the rear-wheels are attached was that it was change from a box section, to a T-shaped section that was lighter, this

can be seen in the figure 8.3, the reason for this was that the body was too heavy for the selected motor, to accelerate to the velocity where the Hybrid Robot was able to do the swing-up motion in the real world. The motors would stall out and the robot would come to stand still. During the change over to the T-shape, the distance between the front and the rear wheels was also changed, which also lessened the velocity required to preform the swing-up motion of the Hybrid Robot, which was found in Gazebo. A final assembly of the full body can be seen in the figure 8.4.

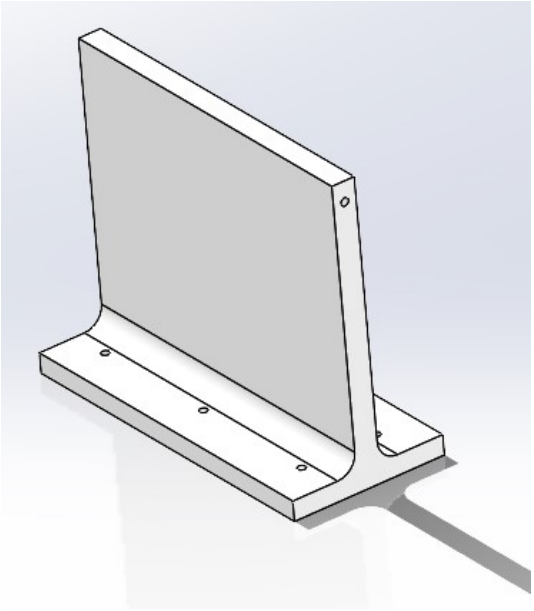


Figure 8.3: The t-shaped part of the Hybrid Robot.

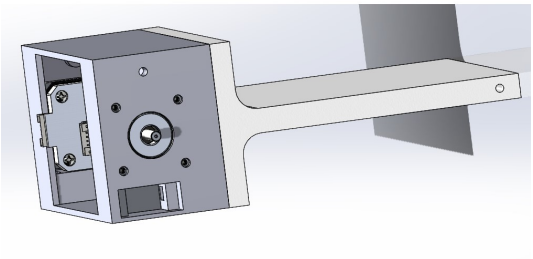


Figure 8.4: The full assembly of the body of the Hybrid Robot

8.2 Motor and Motor-Controller

As mentioned, after it was shown that it was able to do the swing-up in simulation, a selection of the motors that would be used in this project had to be chosen, these motors are based on the first iteration of the model implemented in simulation. There were

different options for motor choices, DC motors, brushless motors, and stepper motors. What was needed was a motor with higher torque that was affordable and had a short delivery time. The different motor types also required different ways of control which also is an expense that has to be taken into account when selecting the motor.

The motor that was selected was a motor named SM42HT47-1684A which is a variation of the Nema 17 stepper motor, this has 4400 g-cmNm holding torque and a rotor inertia of 68 g-cm^2 and a mass of 350g[26]. The stepper motor has been used in different applications such as 3D printers, Computer Numerical Control (CNC) machines, and robotics[27]. A stepper motor does not work in the same manner as an ordinary DC-motor, where voltage is applied over the inner coil also known as the rotor, this creates a magnetic field when the current is applied and this act against the stator which often is made out of magnets, this makes the rotor and the output shaft spin on the DC-motor[28]. The Nema 17 stepper motor chosen has eight outer coils, four of these coils are connected together and placed opposite to each other, a simplified view can be seen in the figure 8.5, where two coils will become positive charged and the other coils will be negative charged[29]. The inner rotor of a stepper motor is made up of magnets alternating north and south pole setup, as seen in the figure 8.5 such that the north is aligned with the positive coils and the south pole is facing the negative coils, alternating the coils that are energized make the motor increment to the next magnets that align with the magnetic fields, the motor chosen have 48 alternating sets of north and south magnets[27].

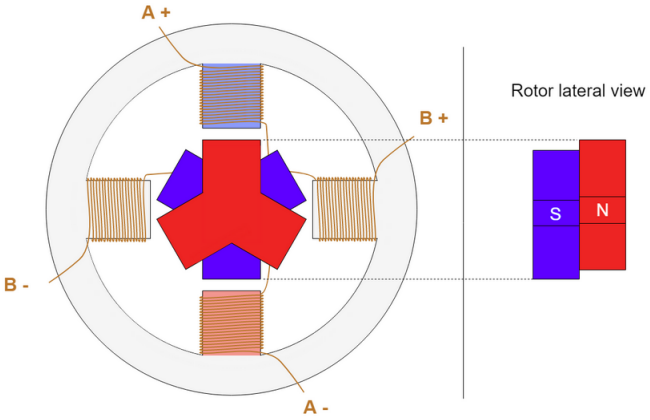


Figure 8.5: A simplified view of a stepper motors inner working. The red and blue are magnets, red is north and blue is south. The winding's is illustrating a coil[29].

A stepper motor can be controlled directly through a micro-controller like Arduino, with the use of a set of H-bridges, although there are controllers made to control stepper motors. The controller that was chosen for this project was an *Ustepper S* board[30] from the company Ustepper, the Ustepper is developed to attach to a Nema 17 and a Nema 23 on the back, as seen in the figure 8.6. Moreover, the Ustepper S controller board has an encoder implemented that can be used to determine the shafts position and the angular velocity; An Atmel 328p micro-controller is also implemented on the Ustepper controller board, this Atmel 328p micro-controller is the same used on an Arduino. The company Ustepper furthermore provides an open-source Arduino library on GitHub, this allows for programming the Ustepper S in the Arduino IDE. The Ustepper S board also has all

the infrastructure build-in such that the board only need a power supply between 12 and 42 voltage to run the motor as well as the Atmel and the encoder[30].

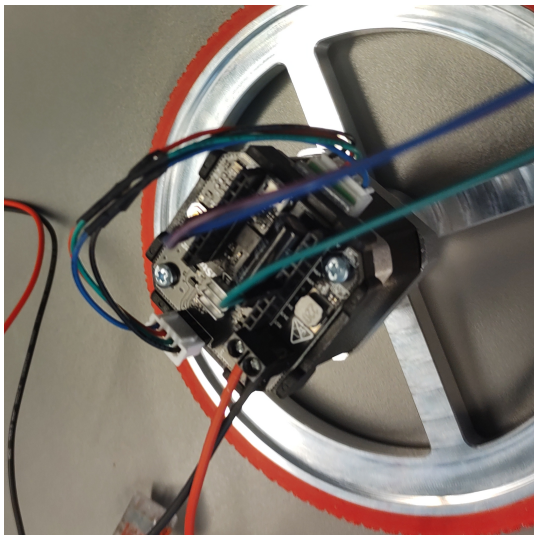


Figure 8.6: The Ustepper S board mounted on the back of the Nema 17 stepper motor.

Some changes to the motors were done, one was to make a flat surface on the motor shaft, to accommodate for the set screw that holds the wheel on the shaft. A second was to shorten up the motor wires, those wires had to have the same length on both motors to prevent a difference in the voltage drop which could lead to one motor turning faster or slow then the other.

As mentioned before, the Ustepper has a library developed to control the Nema 17 motor, this library provides different ways to handle it. One way to run the motor controller is to having it run in a closed loop control, were the built-in functions takes care of getting the motor to a given angular velocity or position; or a PID controller can be used, this opens for more parameters, the primary one is parameters for the PID controller, some others are how much amperes the controller is allowed to use, during running or holding the shaft, set on a scale from 0 100% relating 0A - 1.8A [30].
A setup that is important for both PID and closed loop control is how many steps the motor has to make full rotation of the shaft, this controller can handle 200 steps called full step to 1/8 step which is 1600 steps to a full revolution[30]. Full step is controlling the motor with the highest torque to the motor, but is less precise that the 1/8 which gives more precision control of the motor, but the torque drops off[29, 27].

8.3 Wheels

The first iteration of the wheels used on the Hybrid Robot were used in simulation with the following paramaters: 50mm in radius and a mass of 70g. The mass was calculated by

Cura which is a 3D slicing tool that creates files which can be used on a 3D printer. When Cura calculates the slicing, it also calculates an estimation of the mass of the object.

Due to the change in the size of the body, to fit the expansion, the wheels were too small and left no clearance from the body to the ground, so the wheel diameter was increased to 160mm. During the test of this configuration, it was found that the front wheels needed more mass to preform the swing-up motion. The mass of a wheel to help the swing-up was found to be approximately 260g, the material chosen was aluminum. This choice was made in cooperation with the in AAU in-house industrial machinist Jesper.

To get a wheel with the mass of 260g Solid-Works was used. First for the design and second to see the mass and calculate the inertia, by using the mass properties of the wheel that then also could be used in the simulation. The goal of the design was to get as close to 260g and having the most of the mass of the wheel on the outer rim to create the largest inertia for the wheel with the minimal mass possible.

When the wheel design was done, it was send to Jesper for production. Jesper had different suggestions, to ease the production and reduce the production time for each wheel. This meant that the wheels were created in two different part, a hub and a rim that are screwed together, as seen in the figure 8.7. This modular setup allows the same wheel rim to be used on different motors.

Figure 8.7: In the front of the Hybrid Robot, manufactured in aluminum, the axle of the wheel is meant to be changed if a different motor would be used, this fits the shaft size of the Nema 17 stepper motor.

The back wheels of the Hybrid Robot were kept in Poly Lactic Acid(PLA) to reduce the weight of the cart during the swing-up state, one alteration that was made to the rear wheels was to change it from a machine-screw directly through the rear wheel hub, a set of bearings were design into the rear wheels to lower the friction, a Computer-Aided Design(CAD) drawing of the rear wheel can be, seen in the figure 8.8.

All the wheels got a tire printed of the material Thermoplastic polyurethane(TPU), the reason for choosing the TPU over PLA is that TPU is a type of rubber that is more elastic than the PLA, this allows the Hybrid Robot to gain more friction against the surface[31]. On all the rims, a ridge was created, on the outside of the rim, such that the tire would have a harder time slipping off to the sides.

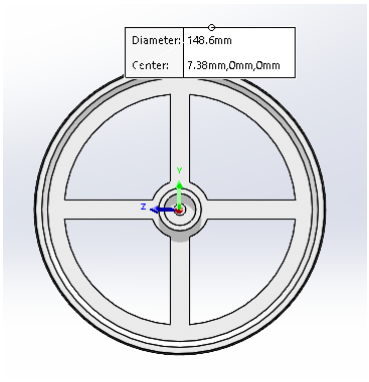


Figure 8.8: Rear-wheel, with space for a bearing on each side, this design choice was because, when running with a machine-screw through the plastic, created too high friction when turning the wheels.

At this point, the front wheels could not be changed as with the motor and the motor controller. This was due to time constraints as well as the limited funding.

8.4 Brakes

Because the motors and the motor-controller were not able to stop the Hybrid Robot prototype, well enough to perform the flip that is needed for the swing-up state, a choice was made to implement breaks on the Hybrid Robot. A reason for this choice was time to implement a new motor into the design, this will also lead to an implementation of new software development for the motor-controller.

The first discussion to be had was where to place the brakes on the Hybrid Robot, this was during the implementation when the robot consisted of a boxed section. A reason for placing the brake mechanism in the boxed section is that it is possible to get more leverage on the wheels to stop them as fast as possible, a downside to placing the brake mechanism in the box is that the time it took to print the box was around 24 hours, the motor section took 8 hours to print, which makes possible to print and evaluate if all parts fit proper, four times within time for printing the box. Furthermore, if the design of the box section should be changed in the future as it did, mentioned in section 8.1 as this was done in the iteration of the box section. The choice of placement was to place it in the motor section.

After the decision on the placement of the brakes, different options were considered as options for how to construct such a brake. One idea was to try and catch the spokes of the wheel itself, this was not an option in that the force this solution would exert a high force on the part that should grip the spoke and where the part and the mechanism that would move the part. A second idea was to have a lever pull down on the axle of the wheel, this idea lead to the final approach, which was to use two levers to grasp around the axle.

The levers were designed in a manner, where the levers would be able to grasp the largest area of axles on the wheels, with a safety margin to the wheel walls, this meant that

circular cuts were made in the levers, as seen in the figure 8.9. The bent shape of the levers is opposite to the hole seen in the figure 8.9, were made to house a spindle that is oval-shaped, as seen in the figure 8.10. The levers were cut such that the arms of the levers could go over each other. As the levers were printed in PLA, a brake pad was made out of TPU to give more friction in the braking face than PLA will provide.

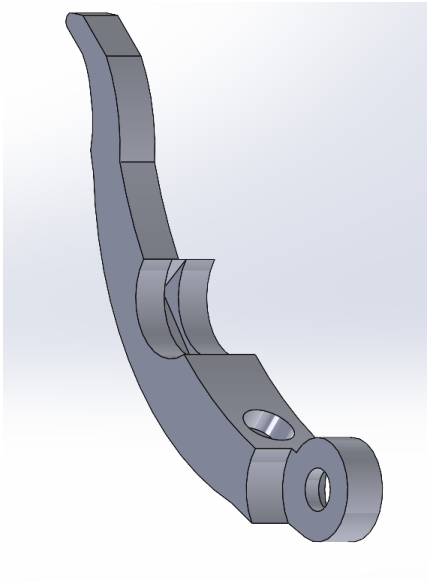


Figure 8.9: Design of a single lever which is used to brake the wheels.

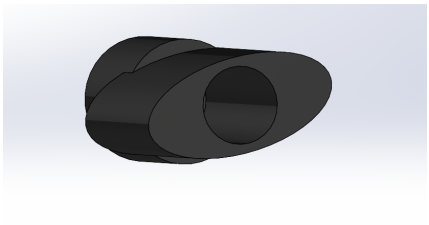


Figure 8.10: First iteration of the spindle that is used to push the levers from each other.

When all the parts where printed and fitted to the motor section, it was found that the levers bend just enough under load, to stop the wheels to the desired effect, a second spindle was made in the same shape and size as the first oval spindle, the size could not changes due to the fact that if the size became bigger in any direction, this would lead to the activation of the levers as a brake. The second spindle as seen in the figure 8.11. The design of the spindle, utilizes the fact the two levers cross each other, as seen in the figure 8.12, where a tap is placed on the oval shape offset from the long axis line, such that the tap extended out over each of the arms of the levers. The taps are set on different levels such that they would not interfere with the lever that tap should not activate. Each spindle is operated by a MG90s servo, the reasons for the use of a servo, is that a servo can be controlled to a desired position, this lessen the need of extra infrastructure, such as switches or encoders on if a ordinary DC-motor would need to estimate the position

of the spindle. The servo is controlled by and Arduino Uno, this was done to protect the Raspberry Pi, that is used as the main controller of the robot.

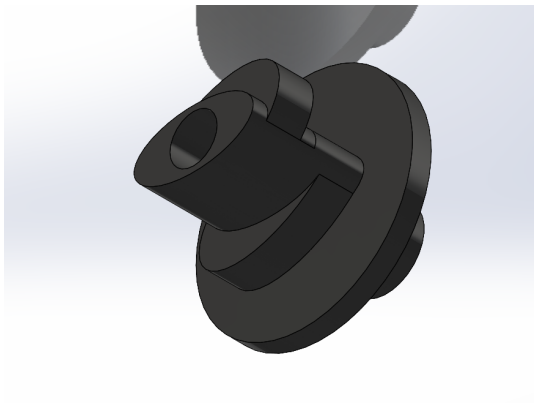


Figure 8.11: This is the final iteration of the spindle, this works on two different levels to push the levers as much as possible.

8.5 Electronic components

In this section the different electronic components that were used in this project will be described, besides the Ustepper and the servo motor and its controllers that has already been described in section 8.2 and section 8.4. How these electronic components are connected can be seen in appendix C.

8.5.1 Onboard Computer

As the onboard computer a Raspberry Pi(RPI)4 with 4 Gb RAM was used, a reason to choose the RPI4 over the RPI3 is that the RPI4 has more Universal Asynchronous Receiver/Transmitter (UART) connections than the RPI3, this meant that there were 4 more serial connections[32, 33]. That can be used to communicate with the various devices implemented as the Arduino that controls the brakes and the two Usteppers boards, which would not be possible with the RPI3 as it only has one UART.

A second reason for choosing the RPI4 over the RPI3 is that RPI4 has overall better characteristics than RPI3, the RPI4 has more RAM and a quicker CPU[34]. The faster the CPU on the RPI4 runs hotter, this was addressed by having a small aluminum block with fins attached to disperse the heat.

The operating system, used on the RPI4 is a version of the Ubuntu 16.04 named Lunetic, this is installed by the image from Ubiquity-robotics[35]. The reason for using this image from Ubiquity-robotics was that this image has everything built for Robot Operating System (ROS) to work out of the box. Furthermore, this image from Ubiquity-robotics also

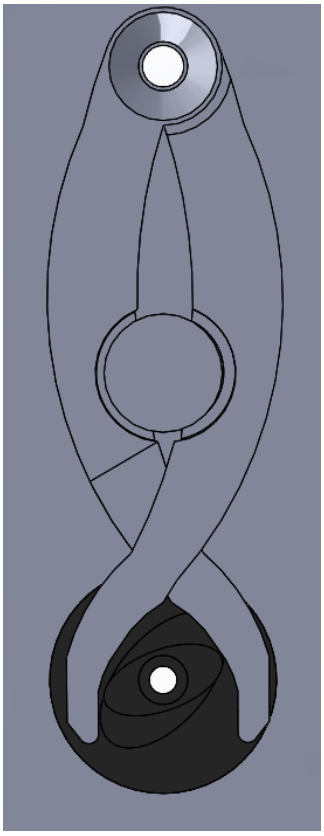


Figure 8.12: In this figure a test bed is presented, this was used to validate the brake system before it was produced.

has setup an access point such that multiple PCs can be connected to control the Hybrid Robot [36].

8.5.2 IMU

The IMU chosen in this project, is the GY-91, this IMU contains a MPU9250 and BMP280 [37]. MPU9250 is a 10-DOF (Degrees Of Freedom), 3 axis of acceleration, 3 axis of angular rate and 3 axis that look at the magnetic fields, this can also be used to describe how the Hybrid Robot is oriented in the world. The magnetometer is not used in this project, the reason for this is that there is no need for it during the test of the Hybrid Robot, furthermore the rebar in the floor of the building where the robot is tested can make for a noise signal. The barometric sensor BMP280, is not used either, it was found that there were no reason for implementing, as the Hybrid Robot does not need to know what altitude it is at. The accelerometer and the gyroscope were used to estimated the pitch and angular rate of the Hybrid Robot. The GY-91 gives access to all the components through a I^2C connection, the GY-91 board requires between 3.3V to the 5V for the GY-91 to work properly.

8.5.3 Power Supplies

To supply all electronics with the right amount of voltage, two separated power supplies were implemented. This was found that during the test of the Hybrid Robot the AP closed down when operating the servos on the same power supply as the RPI, this was thought to be because of the high start current used by the servos. All devices on the Hybrid Robot are powered through a single battery pack, this was to reduce the number of wires to be controlled during the test. The voltage regulator boards that were chosen are based around XL6009 chip from XLSEMI [38], this boost buck convert can deliver up to 4A according to the data-sheet, although the place of purchases the board is only rated at 2.5A continuous [39], which is enough to power the RPI with one. The voltage regulators are variable and can be adjusted by a trim potentiometer to the voltage at the desired level, for the RPI4 it is 5V according to the data-sheets [33]. The servos are rated between 4V and 6V, the higher the voltage the servos provide more toque, and the Arduino is rated from 5V to 12V, this meant that the second power supply could be tuned to 6V to give the most torque to the brakes. Tuning the regulators before anything was attached, was done by using a multi-meter and a power resistor, the resistor is put on the output to put a load on the regulator board because the voltage could drop below operating voltage for the brake system or the RPI4, if not tested with a load on the output.

9 - Implementation and Design of the Software

In this chapter the implementation of all the software used is in simulation as well as in the real world is discussed, the repository can be found at `https://github.com/ipujol10/hybrid_robot`.

9.1 Model Simulation

To be able to test the ideas without having the risk of damaging the physical prototype or try extreme conditions, it is important to have a simulation of the system to test.

For the simulation on this project, a combination of ROS [40] and Gazebo [41] has been used. ROS is used to communicate each part of the system with each other and Gazebo is the simulation environment and supports ROS to interact with the robot models. Both of the systems have been used during the Robotics Masters and the authors of this report are well versed in them.

Once decided the environments of the simulation, it is important to be able to build or represent the prototype or robot to be tested. This can be done with already existing robots in real life to further tests or to tests design concepts and accept them to build or reject them.

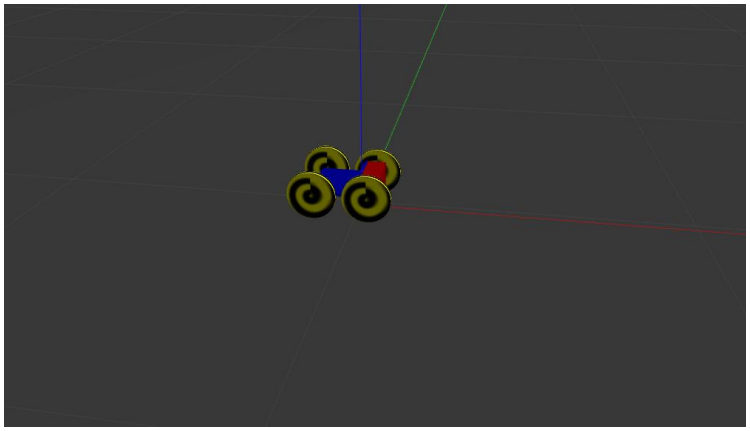


Figure 9.1: Last iteration of the model used for the simulation.

9.1.1 URDF

Unified Robot Description Format (URDF) [42] is a way to describe a robot in an XML format.

Everything starts with the *robot* tag, where everything is going to be defined and described. Here is also stated the name of the robot.

```
<robot name="hybrid_robotv0_2">
   <!-- Robot description -->
</robot>
```

Code 9.1: Example of the start of a robot description in a URDF file. The robot name is *hybrid_robotv0_2*.

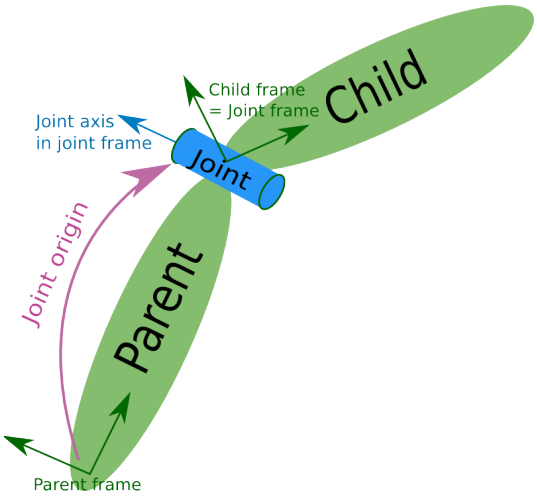


Figure 9.2: A basic representation of a robot with 2 rigid bodies (the green shapes named **links**) joined by a specific behavior (the blue shape named **joint**). [42]

The next step is to understand the essential parts of a robot and how they are connected (in the URDF context): the links and the joints as seen in the figure 9.2. The links represent the rigid bodies of the robot as can be wheels, arm sections or chassis. And the joint specify how two links interact with each other: if they can move freely, if they are static with each other, etc.

The link have three basic elements: the *visual* which describes how is it going to be visualized on the simulation; the *collision* which describes how to interact if it touches other links; and the *inertial* which have the physic properties as the mass and inertia tensor.

```
<link name="base">
   <visual>
      <geometry>
         <box size="1 1 1"/>
```

```
      </geometry>
      <material name="blue"/>
   </visual>
   <collision>
      <geometry>
         <box size="1 1 1"/>
      </geometry>
   </collision>
   <inertial>
      <mass value="1"/>
      <inertia ixx="1" ixy="0" ixz="0"
            iyy="1" iyz="0"
            izz="1"/>
   </inertial>
</link>
```

Code 9.2: An example of a cube link, both visual and collision-wise, with a side of 1m. It also has a mass of 1kg and an inertia tensor as the identity matrix.

In the *geometry* element, it is also possible to import a mesh created on other designing programs as can be SolidWorks or a CAD program by using *<mash file"" scale""/>* instead of the box.

The joint has two required attributes: the name and the type; and at least the parent and child elements, it can have more elements to define better the relation or limit some aspects. [42]

```
<joint name="base_wheel_joint" type="continuous">
   <axis xyz="0 0 1"/>
   <parent link="base"/>
   <child link="wheel"/>
   <origin xyz="0 0.6 0" rpy="1.5708 0 0"/>
</joint>
```

Code 9.3: Example of a joint that joins a wheel to the cube from the code 9.2. It specifies that it can rotate freely around the z-axis while it is displaced 0.6m from the origin and rotated 1.5708rad or 90° on the roll axis.

As the code 9.3 showed, there is an attribute type that defines the type of movement that can happen between two links. There are the following types: [42]

revolute A hinge type joint that rotates around the specified axis (with the tag <axis>) and that has upper and lower limits (with the tag <limit>) that can represent physical restrictions as a robotic arm.

continuous The same as the **revolute** but without limits. It can represent for example an axle.

prismatic A joint that allows the movement on 1D and has limits. An example of this joint would be a piston.

fixed This is the only joint that does not allow movement. It would be the equivalent of gluing or fixing with screws two pieces. In the URDF modeling is useful to build complex parts with just basic shapes without having to import an external design.

floating This one does not have any restriction, it allows the 6 degrees of freedom. Like not having a joint at all.

planar Allows the movement on a plane defined by the perpendicular of the <axis> tag. It can be an example of moving a magnetized floor or a board games simulation.

With these simple elements, a robot can be built by combining links through joints one by one.

9.1.2 Xacro

To build small and simple robots the standard URDF method is more than sufficient, but when trying to make more complex projects can be inefficient and lead to typing errors.

The problem comes from the fact that if you need to have ten wheels on a robot, you need to define them one by one with almost the same parameters.

Then comes the XML Macros (Xacro) that allow much more modular implementation and conversion to URDF afterward. It can have variables or properties that make it easier to change a single value that will affect the rest of the robot instead of having to go one by one. It also allows using math expressions while defining links or joints and conditional blocks to be able to have different behaviors depending on the parameters. Xacro, as the name indicates, allows to have macros (similar to functions in other programming languages) and import other Xacro files which allow to have more clean and modular work. [43]

```
1 <robot name="hybrid_robotv0_2"
     xmlns:xacro="http://ros.org/wiki/xacro">
2    <!-- Robot description -->
3 </robot>
```

Code 9.4: Example of the start of a robot description in a Xacro file. The robot name is *hybrid_robotv0_2*. That is how it would look for the code 9.1 while being a Xacro.

To start the description of the robot, normally the first lines are reserved for importing the needed Xacro files and defining the properties of the robot:

```
1 <xacro:include filename="$(find
     hybrid_jumping_robot)/urdf/material.xacro"/>
2 <xacro:include filename="$(find
     hybrid_jumping_robot)/urdf/plugins_v2.gazebo.xacro"/>
3 <xacro:include filename="$(find
     hybrid_jumping_robot)/urdf/macro.xacro"/>
```

```
4  <xacro:include filename="$(find
       hybrid_jumping_robot)/urdf/transmission_v2.xacro"/>
5
6  <!-- Wheels -->
7  <xacro:property name="wheel_r" value="0.075"/>
8  <xacro:property name="wheel_width" value="0.015"/>
9  <xacro:property name="front_wheel_mass" value="0.279"/>
10 <xacro:property name="back_wheel_mass" value="0.09589"/>
11
12 <!-- Rest of the properties -->
```

Code 9.5: Example of how to include other Xacro files and define some properties to be used further in the description of the robot.

As seen in the code 9.5, the *xacro:include* tag allows to include other files with predefined macros or other parts of the robot that have been decided to keep in separate files. The other useful tag is the *xacro:property* which allows to have the equivalent to variables and be accessed at any point of the file, it also allows to have values used on different places all on the same spot and have them controlled.

In the next code snippet, it can be seen how a macros is defined and it uses math and the properties:

```
1  <!--box_inertial makes it easier to implement mass and inertial
       for a box shape-->
2  <xacro:macro name="box_inertial" params="mass length width
       height">
3     <inertial>
4        <mass value="${mass}"/>
5        <inertia
6              ixx="${1/12*mass*(width*width+height*height)}"
7              ixy="0.0"
8              ixz="0.0"
9              iyy="${1/12*mass*(length*length+height*height)}"
10             iyz="0.0"
11             izz="${1/12*mass*(width*width+length*length)}"/>
12    </inertial>
13 </xacro:macro >
14
15 <xacro:box_inertial mass="${base_short_mass}"
       length="${base_short_length}" width="${base_width}"
16                height="${base_short_height}"/>
```

Code 9.6: Example of the definition of macros and the use of it.

As seen in the code 9.6, the definition of a macros starts with the <*xacro:macro name=”” params=””*> tag. The name defines how is it going to be called after being defined as it can be seen in the line 15: the name is "box_inertial" and it is called <*xacro:box_inertial*>. params defines the parameters that need to be passed on when called as seen in the lines

15 and 16. The parameters are treated the same way as the properties.

The math is produced when an expression is between "$" as seen in the line 6 for example where it can be seen how multiplications and additions are performed. In the math mode, it can also be accessed the parameters and properties as seen in the line 4 or 6.

The last big feature that Xacro have over plain URDF are the conditional blocs. This allows to create macros that produces different results depending of the parameters passed on:

```
1 <xacro:property name="origin_y" value="${(base_width +
     wheel_width)/2 + clear}"/>
2 <xacro:if value="${side == 'right'}">
3    <xacro:property name="origin_y" value="${-(base_width +
        wheel_width)/2 - clear}"/>
4 </xacro:if >
```

Code 9.7: Example of a conditional block.

In the code 9.7 it can be seen how an if statement is performed. First of all, a property is defined, then it checks if the statement is true and overrides the property in that case. Apart from acting on properties, it can also be used to produce complete blocks as it can be a complete joint.

9.1.3 Gazebo

When it comes to simulating together with Gazebo, the description needs some additional tags.

To be able to be visualized properly, the <*gazebo*> tag must be included:

```
1 <gazebo reference="base">
2    <material>Gazebo/Red</material>
3 </gazebo>
```

Code 9.8: Gazebo tag with the reference as the link referenced to.

When it comes to actuated joints, it is necessary to specify the type of control:

```
1 <transmission name="trans_front_right_wheel_joint">
2   <type>transmission_interface/SimpleTransmission</type>
3   <joint name="front_right_wheel_joint">
4        <hardwareInterface>
5           hardware_interface/VelocityJointInterface
6        </hardwareInterface>
7   </joint>
8   <actuator name="front_right_wheel_joint_motor">
9        <hardwareInterface>
```

```
10          hardware_interface/VelocityJointInterface
11        </hardwareInterface>
12        <mechanicalReduction>1</mechanicalReduction>
13    </actuator>
14  </transmission>
```

Code 9.9: Example of a velocity control transmission.

In the code 9.9 it can be seen that the transmission is referred to the joint in line 3 and using the VelocityJointInterface as the controller.

Once having all these tools, then it is only a matter to build the robot peace by peace. The end result can be seen in the figure 9.1.

9.2 Simulation Implementation

In this section, it is going to be talked about the classes used during the simulation to be the bridge between the simulation and the controller. In the physical model, they are going to be implemented on the onboard computer.

As has been stated before, ROS is the tool that allows communication between each part or node of the system. Each node is a process that can be run individually and all of them will be managed by a ROS Master. Each node can publish or be subscribed to as many topics as needed and that is the way that they can send and receive information to/from other nodes. [40]

During the simulation, it is not possible to just send or receive a signal through the Input-Output (IO) of the onboard computer because there is no physical link. To solve that and be able to simulate, gazebo uses ROS to simulate these signals.

9.2.1 Velocity

The velocity node serves the purpose of receiving the input from the controller and sending it to the virtual wheels.

In the figure 9.3, it can be seen the most important attributes and methods. The *Publishers* in this class work in two directions: towards the simulation as an IO link and towards the controller. The *Subscribers* work as the feedback from the simulation and controller. The methods are used to control the wheels and obtain the feedback data.

```
1  Vel::Vel() : rate(100), now_velocity(0), now_position(0) {
2    ros::NodeHandle nh;
3    left_front_wheel_publisher =
       nh.advertise<std_msgs::Float64>(left_front_wheel_connection, 10);
```

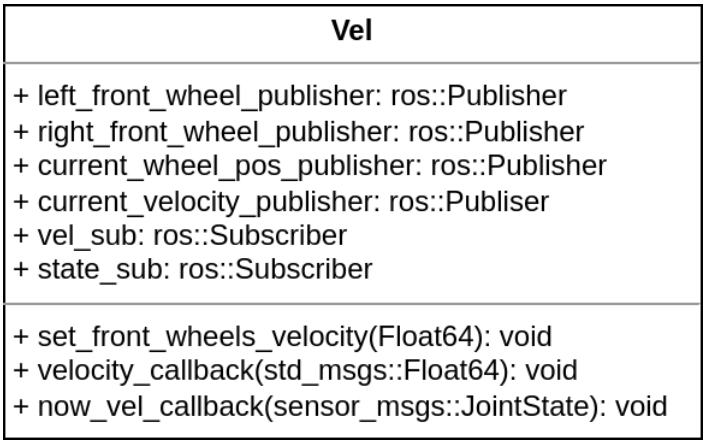


Figure 9.3: The basic structure of the Vel class.

```
4   right_front_wheel_publisher =
        nh.advertise<std_msgs::Float64>(right_front_wheel_connection, 10);
5   current_velocity_publisher =
        nh.advertise<std_msgs::Float64>(current_velocity_connection, 1);
6   current_wheel_pos_publisher =
        nh.advertise<std_msgs::Float64>(current_position_connection, 1);
7   vel_sub = nh.subscribe(commanded_velocity_connection, 1,
        &Vel::velocity_callback, this);
8   state_sub = nh.subscribe(joint_state_connection, 1,
        &Vel::now_vel_callback, this);
9 }
```

Code 9.10: The constructor of the Vel class.

Gazebo uses ROS topics to interact with the simulation. To send data to the wheel controllers, it is needed to publish on the specific topic (initialized on the lines 3 and 4 of the code 9.10) with the names "/hybrid_robotV0_2/front_side_wheel_joint_velocity_controller/command" with *side* replaced by right or left (the full ROS connections can be seen in the figure D.1).

```
1  void Vel::set_front_wheels_velocity(Float64 vel) {
2   std_msgs::Float64 msg;
3   msg.data = vel;
4   left_front_wheel_publisher.publish(msg);
5   right_front_wheel_publisher.publish(msg);
6   ros::spinOnce();
7   rate.sleep();
8  }
9
10 void Vel::velocity_callback(const std_msgs::Float64 &data) {
11  set_front_wheels_velocity(data.data);
12 }
```

Code 9.11: Control of the wheels.

In the code 9.11 it is specified how it interacts with the simulation: publishing on the correct topics (lines 4 and 5). Then it is told to the Master that it has received a message to be redistributed in line 6. The *set_front_wheels_velocity* function is called when the message through the controller is received.

```
1  void Vel::now_vel_callback(const sensor_msgs::JointState &data) {
2   now_velocity = (data.velocity[3] + data.velocity[4]) / 2;
3   std_msgs::Float64 msg;
4   msg.data = now_velocity;
5   current_velocity_publisher.publish(msg);
6
7   now_position = (data.position[3] + data.position[4]) / 2;
8   msg.data = now_position;
9   current_wheel_pos_publisher.publish(msg);
10 }
```

Code 9.12: Callback from the joint states from the simulation.

The data that would be obtained by odometry, is provided by the ROS topic "/hybrid_robotV0_2/joint_states" in the code 9.12. The joint state structure contains the position, velocity and torque of each joint of the robot, which will allow to obtain the needed simulated odometry. In the lines 2 and 7 are calculated the position and velocity of the robot having in mind that it is a differential drive, which are obtained by the following equation:

$$\frac{read_r + read_l}{2} \tag{9.1}$$

The indexing are to obtain the correct items on the whole list of joints.

9.2.2 IMU

The IMU node serves the purpose of getting the simulated IMU data and converting it to the relevant information for the system.

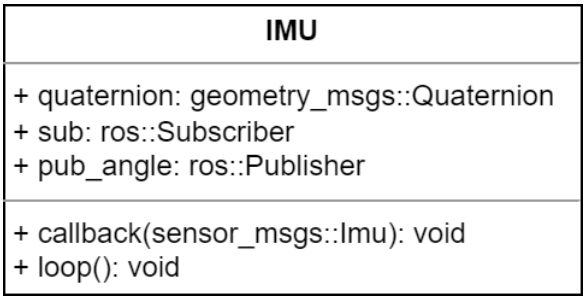


Figure 9.4: The basic structure of the IMU class.

This node subscribes to the simulated IMU topic ("/imu") to be able to broadcast it to the other nodes of the system. The Gazebo sensor sends a quaternion which is converted to

Tait–Bryan angles [44].

```
RPY conv::quaternion_to_rpy(geometry_msgs::Quaternion q) {
  tf::Quaternion quaternion(q.x, q.y, q.z, q.w); // construct the quaternion
  tf::Matrix3x3 m(quaternion); // convert it into matrix
  Float64 roll, pitch, yaw;
  m.getRPY(roll, pitch, yaw); // convert it to Roll—Pitch—Yaw
  return {roll, pitch, yaw};
}
```

Code 9.13: Function to convert a quaternion into Tait–Bryan angles using the ROS transform library.

The problem with using this conversion to the Tait–Bryan angles, is that the registered pitch keeps increasing until $\pi/2$ and, at that point, instead of continuing to go up, it falls.

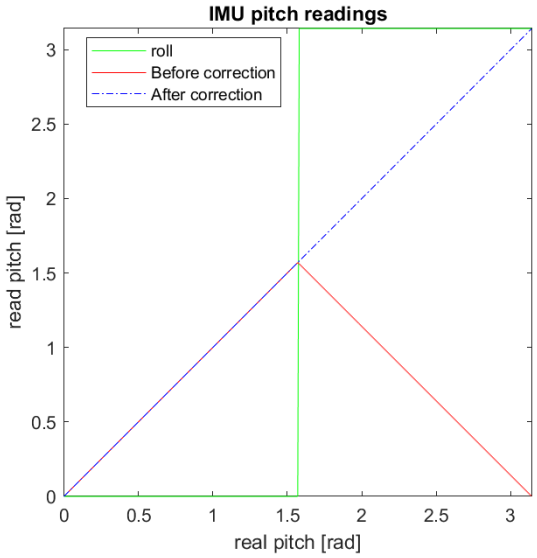


Figure 9.5: The graphic that shows how the raw pitch can lead to problems, how it can be detected the issue with the roll and the corrected pitch.

As seen in the figure 9.5, the pitch behaved as described before. But after some thinking and analyzing the obtained data, it was detected that when the pitch decreases, the roll suddenly changes the value. It seems like for the IMU, the robot flipped horizontally instead of vertically, which is the reality.

To obtain the correct pitch, it is passed together with the roll through a function that checks if the roll has flipped. If it is the case, it returns the complementary angle, else returns the original angle.

9.3 Controller Implementation

In this section, it is going to be talked about the implementation of the different controllers.

9.3.1 PID

In the section 7.3.1, has been explained the PID controller. In this section, it is going to be explained the implementation of this controller.

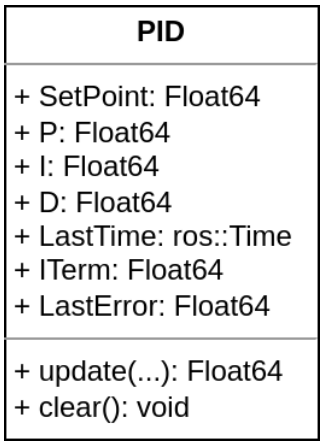


Figure 9.6: The basic structure of the PID class.

In the figure 9.6, it can be seen the most important attributes and methods. With the PID attributes as the controller constants, the *SetPoint* as the target, using the *LastTime* to be able to determine the time elapsed since the last update and *LastError* to be able to get the Δe (differential error). The *update* method is used to obtain the output of the PID and the *clear* method to reset variables that could cause troubles on long periods of time as they can be *ITerm* and *LastError*.

```
1  Float64 PID::update(Float64 feedback_value, ros::Time current_time, ...)
       {
2      Float64 error = SetPoint - feedback_value;
3
4      ros::Duration delta_time = CurrentTime - LastTime;
5      Float64 delta_error = error - LastError;
6
7      PTerm = P * error;
8      ITerm += error * delta_time.toSec();
9
10     DTerm = delta_error / delta_time.toSec();
11
12     LastTime = CurrentTime;
13
14     Output = PTerm + (I * ITerm) + (D * DTerm);
```

```
15
16      return Output;
17  }
```

Code 9.14: Implementation of the PID using the update method.

In the code 9.14 it is shown how the most important part of the PID is implemented. First of all, is calculated the time elapsed and the differential error with respect to the last update. Then it calculates the proportional term in line 7 by using the current error and multiplying by the proportional constant ($PTerm = P * error$).

To use the integral part of the PID it is necessary to obtain the integral of the error (area below the curve) to then use the constant gain I. But the integral on the discrete-time, it is defined by a summation:

$$\sum_{i=0}^{N} error_i \Delta t_i \tag{9.2}$$

Then the $ITerm+ = error * delta_time.toSec()$ (in the line 8) is the n-place of the summation while $ITerm$ has the historical error.

On the derivative part, something similar happens as it is a discrete process and it is described as:

$$\frac{\Delta error}{\Delta time} = \frac{error_i - error_{i-1}}{\Delta t} \tag{9.3}$$

And that is how it is implemented: in line 5 obtaining the $\Delta error$ and in line 10 obtaining the derivative.

The total PID output is calculated in line 14 where all the terms are put together.

There are some additional features such as the integral windup guard, which limits its influence if it starts to be too high, as it is something that can happen especially in a long term runs. The other is the overall output limit to avoid an overload on the actuator.

9.3.2 State Feedback

In the section 7.3.2 it was described the theory behind the state feedback control. In this one, it is going to explain how it has been implemented.

The implementation has been inspired by a GitHub repository [45]. This is a general case where it allows not having full state output and estimate if it is necessary, implementing the compensator for reference tracking, and adding an integral component in case it is needed to be used.

As seen in the figure 9.7, there is the diagram of how the state feedback works. In the bottom there is the standard simplified state feedback stabilization where the feedback is defined by the negative control gain. It can also be seen that there is an estimator, but it is only going to be used in case that the output is not the full state vector.

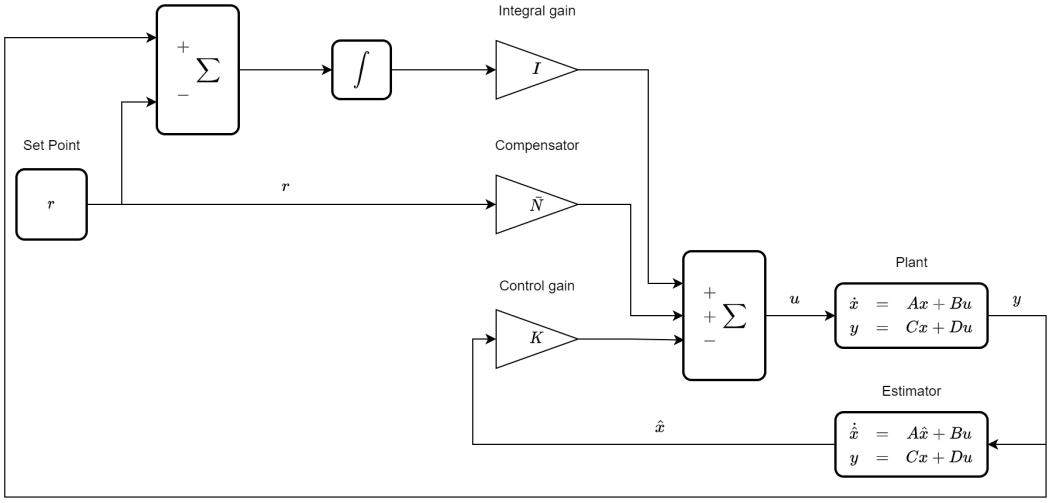


Figure 9.7: Diagram of the full state feedback class that allows it to have a general-purpose and work with different control methods.

The other two lines contain the added features: the reference tracking to stabilize on a different working point than the stable point; and the integral term that can help stabilize in case of having constant disturbances from the exterior of the system.

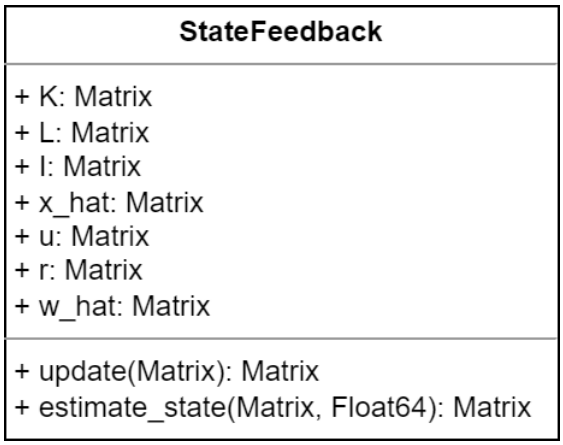


Figure 9.8: The basic structure of the State Feedback class.

In the figure 9.8, it can be seen the most important attributes and methods. For the standard model, there are the *K* as the control matrix; the *x_hat* as the input to the control gain; and the *L* as the observer gain in the case that has to be used if there is not a full output.

For the reference branch, there is the *r* as the reference that can be constant or changing over time. In the case of the integral part, the *I* is the integral gain and the *w_hat* represents the integral as it has stored the whole accumulated error. And to finish with the attributes, there is the *u* as the control input to the robot.

```
1  Matrix StateFeedback::update(const Matrix &y, Float64 dt) {
2    // Update the state estimate
3    if (X == Y) { // if it's Full State Feedback
4      x_hat = y; // directly the new x_hat are the measured states
5    } else { // if it's not FSF
6      x_hat = estimate_state(y, dt); // it needs to estimate the x_hat
7    }
8
9    // Calculate the control input to stabilize at 0
10   u = -K * x_hat;
11
12   // Calculate the offset control for reference tracking
13   u += N_bar * r;
14
15   // get the offset to control the disturbance w
16   w_hat += I * (y - r) * dt;
17   u += w_hat;
18
19   return u;
20 }
```

Code 9.15: Implementation of the update method on the State Feedback.

The update method (in the code 9.15) computes the whole control input u. First, it checks if it is full state feedback (X represents the number of states and Y the number of outputs) on line 3, in the case of being affirmative the *x_hat* is taken directly from the system output; otherwise, it needs to compute the estimate. The estimate is done by the *estimate_state* method which has a linearized model of the system with the L gain.

Then it is time to add all the different contributions to the control input. First, it does the stabilization term on line 10 continuing with the reference tracking in line 13. In line 16 it is calculated and updated the integral part and the added to the control input.

In the following sections, there are going to be talk about the different methods to obtain the control gain.

9.3.2.1 Heuristic

The heuristic method to find or design a control method is based on try and error until a solution is found. This solution is not going to be optimal, but sufficient as a starting point to be able to start testing, a temporal solution. [46]

Being in the case of a system with four states and a single control input the dimensions of the control matrix must be: K^{1x4}. Each of the components of K is going to be associated with a state of the system and how much they contribute to the overall value of the control signal. For the present system the association is going to be like: $[\theta, x, \dot{\theta}, \dot{x}]$.

After testing and heuristically tuning the parameters, the control matrix obtained is:

$$K = [802.88,\ 0,\ 139.09,\ 784.99] \tag{9.4}$$

9.3.2.2 Poles Placement

The pole placement method has been explained in the section 7.3.3, in the current one is going to talk about how it has been implemented.

To be able to place poles, first, it is needed to obtain the dynamic model of the system similar to obtaining the matrices 7.50 and 7.51.

Those have been obtained by using a Matlab script to help do all the derivatives without error and work with the matrix operations. The first step is to define the symbolic variables as the physical constants and the states dependent on the time which will allow deriving respect the time. This method of using a script will also allow having changed as it can be the system reference to be able to recalculate the system model automatically and fast.

With those variables, it is possible to build and calculate the equations 7.5 and 7.6. As Matlab allows to derivate respect to any variable, the operations can be done the same as they can be done on paper:

```
xb = x - d*cos(theta); % position on the x of the body
vxb = diff(xb, t); % velocity on the x of the body
% vxb =
%       diff(x(t), t) + d*sin(theta(t))*diff(theta(t), t)
```

Code 9.16: Example of how to calculate the velocity by deriving the position with respect to the time.

Then it is time to build the Euler-Lagrange equations for each state (equation 7.2) and have them organized already as a matrix to help the automation of the process:

```
Qtheta = simplify(diff(diff(L, theta1), t) - diff(L, theta)); %
    d/t*dL/dtheta1 - dL/dtheta
Qx = simplify(diff(diff(L, x1), t) - diff(L, x)); % d/t*dL/dphi1 - dL/dphi
Q = [Qtheta; Qx]; % have both terms on a matrix to have it easier to derivate and obtain both
    equations of motion
```

Code 9.17: Example of how to build the Euler-Lagrange equations having already calculated $L = T - V$. And having them stored as a single matrix object. **theta1** and **x1** are the first derivatives respect the time of the states ($\dot{\theta}$ and $\dot{x}$).

The last step before obtaining the equations of the second derivatives of the states ($\ddot{q} = f(\dot{q}, q, u)$) is to obtain the matrices to build the equation 7.30:

```
M = simplify([diff(Q, theta2), diff(Q, x2)]); % obtain the friction matrix
M1 = simplify(inv(M)); % inverse of M
h = simplify(subs(Q, {theta2, x2}, {0, 0}) + diag([alpha beta])*q1); %
    corriolis, gravity effect and inertia
```

```
4 gq = [Iw; -Iw]; % the input matrix
5
6 Anl = [q1; simplify(-M1*h)]; % The A matrix on the non-linear form
7 Bnl = [0; 0; simplify(M1*gq)]; % The B matrix on the non-linear form
```

Code 9.18: Example of how to obtain the matrices to build 7.30 and end up with the A-B matrices on the non linear form.

In the equation 9.18 it is important to explain how each matrix has been found by looking at 7.30. The components of the friction matrix are the ones that are multiplying the second derivative of a state ($\ddot{q}$), that is why it is derived by the second derivative of the states (line 1), to discard the terms without $\ddot{q}$ and isolating the ones go with it.

A similar procedure is taken to obtain the h matrix in line 3: it is needed to isolate the terms without a $\ddot{q}$ so the second derivatives are substituted by a 0 and when it is computed, they will disappear. After that, the friction coefficients are added by relating them to the velocity.

```
1 point = {pi/2, 0, 0, 0}; % The point of work
2 vars = {theta, x, theta1, x1}; % the states
3
4 A_theta = subs(diff(Anl, theta), vars, point); % get derivative respect theta
5 A_theta1 = subs(diff(Anl, theta1), vars, point); % get derivative respect theta1
6 A_x = subs(diff(Anl, x), vars, point); % get derivative respect phi
7 A_x1= subs(diff(Anl, x1), vars, point); % get derivative respect phi1
8
9 Ap = simplify([A_theta, A_x, A_theta1, A_x1]); % linearized A matrix with
    parameters
```

Code 9.19: Example of how to build the A linear matrix from the non-linear and with the linearization point.

At this point, there is only left the linearization part of the problem to be able to apply a state feedback control done in the code 9.19. Each column of the linearized A matrix is multiplied to the same state of the system, meaning that if the A_{nl} is derived by a state, it is going to be left in the column related to that state. And then the states are substituted by the linearization point.

The B matrix simply substitute the linearization point into the B_{nl} as the control variable is already isolated in this matrix configuration.

Then it is just a matter to design the system by deciding on the poles to place. In the case of this project, it was decided to place the poles at $[0,\ -1,\ -3, -6.87565805306348]$ giving a control matrix (using the same method as in 7.1):

$$K = [802.882847764951,\ 0,\ 139.094311369446,\ 784.988650429386] \tag{9.5}$$

But the control is discrete, so it is time to convert the system to the discrete-time world by having the period at which the controller works.

```
1 Ts = 1/100; % sample time
```

```
sys_d = c2d(sys, Ts); % obatin the discrete system
```

Code 9.20: Example of how to convert a continuous system into a discrete time having already build the system A, B, C, D.

And obtain the control gain for the discrete-time:

$$K = [958.097552156146,\ 0,\ 140.746653569888,\ 54.8485642096788] \tag{9.6}$$

9.3.2.3 LQR

To obtain the LQR control gain it is needed to obtain the system model as in the pole placement section 9.3.2.2 being the same as the system has not changed.

It is only a matter of defining the Q and R matrices and using the command on the code 7.4:

$$Q = \begin{bmatrix} 50 & 0 & 0 & 0 \\ 0 & 0.001 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.001 \end{bmatrix} \qquad R = [0.001] \tag{9.7}$$

And obtaining a control gain:

$$K = [991.909531865653,\ 0.930291450322719,\ 145.759105890912,\ 58.798408812102] \tag{9.8}$$

9.3.3 Inverted Pendulum Drive

This class brings together the control methods and is the one sending and receiving the signals from the robot.

In the figure 9.9, it can be seen the most important attributes and methods. Where the *pid* and *stateFeedback* are the controllers that the user can decide which one to use depending on the constructor used. Then there are the *Pitch*, *PitchVel*, *Position* and *Velocity* that are the states in the real world coordinate system, the *target* as the working point and the *sys_states* as the state vector in the controller coordinate system.

Having the system states in both coordinate systems is because when a state feedback is linearized, the states change according to 7.56, meaning that there is a conversion needed in between by using the next lines of code:

```
Matrix current_real_state{{Pitch},
                          {Position},
                          {PitchVel},
```

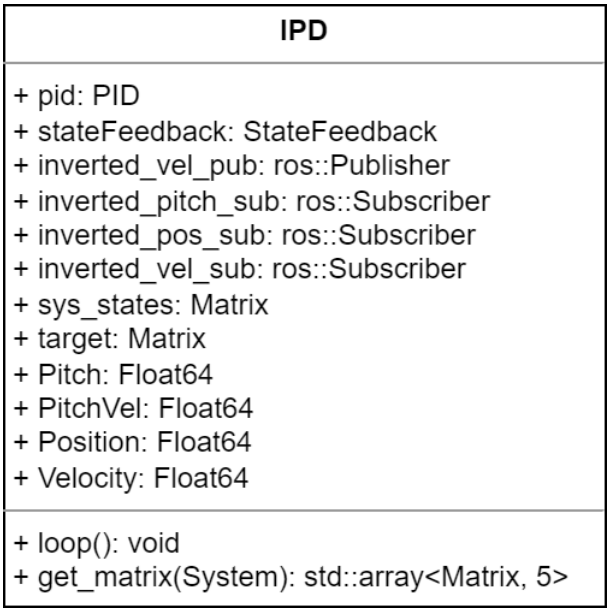


Figure 9.9: The basic structure of the Inverted Pendulum Drive class.

```
                    {Velocity}};
sys_states = current_real_state - target;
```

Code 9.21: This piece of code is shown how to convert the real-world state into the controller coordinate system. Assuming that the *target* variable is the linearization point.

In the *loop* method is where all the control update happens at the rate decided, in the case of this project at $100Hz$. And the *get_matrix* method is used to be able to store different control gains and different systems and access them by selecting one of the saved *System* models.

```
enum class System {
  NONE,
  pole_placement,
  k_manual,
  LQR
};
```

Code 9.22: Example of the structure of *System* that would allow to store a different set of matrices and access them using a switch statement.

9.4 Real World Implementation

In this section, the different nodes created to drive the real-world Hybrid Robot and get feedback form the sensors are discussed.

9.4.1 IMU Node

To make the controllers work in the real world, there is a need to know the orientation of the Hybrid Robot and how it is turning about its axis. The node IMU_robot-1.py is made to read the signal from the IMU and processing the signal can be found in the GitHub repository in appendix A.

The IMU is using an I^2C protocol as a communication medium [37], by calling the main address of the MPU6050 and making the call to either read or write a value to a register. Instead of writing a lengthy code for this a library is used called "mpu9250" this library can be found on `pypi.org`.

Before the IMU can be used it needs to be configured, the accelerometer is set to scale up to 4g and the gyroscope is set to be able to register up to 500 degrees per second. After the setup, a calibration of the accelerometer and the gyroscope is performed to ensure that the axis of the accelerometer is aligned with the body of the IMU.

Every time a new set of data is generated, within the IMU a bit is set in the register of the IMU, this bit is then used to check if the script IMU_robot-1 will read in the data, before publishing it on the ROS topics. New data is ready on the IMU at the rate of approximately 280Hz measured.

The accelerometer data of the Hybrid Robot is used to calculate the pitch and the angular rate of the system. The pitch is derived from all the axis of the accelerometer, the x-axis is aligned lengthwise of the body of the Hybrid Robot; the y-axis is aligned with the width of the body, and the z-axis is pointing through the body. Using the eq. 9.9 to calculate an estimate of the pitch, after this there is a need to check if the acceleration on the z-axis is less than zero, if this is the case the pitch needs to be subtracted from π as in eq. 9.10.

$$pitch = atan2(accelerationX, \sqrt{accelerationY^2 + accelerationZ^2}) \tag{9.9}$$

$$if accelerationZ < 0 : pitch = \pi - pitch \tag{9.10}$$

If this check is not done, it is not possible to see what way the Hybrid Robot is tilted, as the numbers will increase to half π and then decrease again, need values were needed to be between $0 - \pi$, which is possible with the check of eq.9.10.

Due to the noise nature of the accelerometer, when a new pitch is calculated it needed to be filtered with the prior calculated pitch, this is done through an alpha filter as seen in eq. 9.11. The alpha used in this case is set to 0.3, this means that only 30% of the new pitch is used and 70% of the old pitch is used as a stabilization. After the filtering, the pitch is then published on a topic named *"/HJC/IMU/Pitch"*.

$$filtered\_pitch = (new\_pitch * alpha + old\_pitch * (1 - alpha)) \tag{9.11}$$

When a new pitch has been calculated and filtered, the angular rate is calculated before

being published on a ROS topic called *"/HJC/IMU/AngularVelocity"*, this is done by the eq. 9.12.

$$angular\_velocity = new\_pitch - old\_pitch/dt \tag{9.12}$$

9.4.2 Wheel Node

The node that is constructed to control the wheels of the Hybrid Robot, does a few things, it communicates with the two motor-controller(Ustepper) and the brakes(Arduino); calculates the odometry of the Hybrid Robot; it communicates with the ROS nodes. The source code for this node can be found in the GitHub repository in appendix A, the name of this source file is *wheel_node.py*.

In every iteration of the main loop of this node, there is a check if new information is available from the motor controllers, the data is received from the motor controllers comes in the format of a string, and the string looks like this: *RPM:value,Pitch:value*, the RPM is the angular velocity of the wheels and the pitch is the position of the motor shaft. The RPM is converted to a linear velocity through the function RPMtoVel 9.23, the values from both wheels are then added together and multiplied by 0.5 and published on the topic named */HJC/Vel_robot/Current_velocity*. The pitch values are published individually on two different topics */HJC/Vel_robot/Left_wheel_pos*,*/HJC/Vel_robot/Right_wheel_pos*.

Before the odometry can be published on the topic */HJC/Vel_robot/Current_pos*, the odometry had to be calculated, the calculation is based on the odometry of a differential drive[47].In the function *rpmtovel* 9.23 the RPM of the linear velocity, this needs to be calculate the next step.

```
def rpmtovel(self,rpm):
      v = ((math.pi * self.wheel_radii * rpm ) / 60.0)
      v = v * (time.time()-self.oldTime ) #time
      return v
}
```

Code 9.23: This code is used to transform from RPM to linear distance traveled

The function *calvrvl* 9.24 sets the internal represntaion of the velocity of the left and right wheels.

```
def calvrvl(self):
      self.vr = round(self.rpmtovel(self.velocity_right),4)
      self.vl = round(self.rpmtovel(self.velocity_left),4)
}
```

Code 9.24: This function calculates how far each wheel have traveled.

The next function used is *caldot* 9.25 this function calculates the changes in the x, y and orientation theta of the Hybrid Robot.

```
def caldot(self):
```

```
        self.calvrvrl()
        xdot = (self.vr + self.vl) * math.cos(self.theta_pos)
        ydot = (self.vr + self.vl) * math.sin(self.theta_pos)
        thetadot = (self.vr - self.vl) / self.base_withd
        return {'xdot': xdot, 'ydot': ydot, 'thetadot': thetadot}
}
```

Code 9.25: This function calculates the changes in position and orientation.

The data from *caldot* is summarized in the function *odom* 9.26. After the function *odom* have been called, the data for the odometry topic is ready and will be published.

```
def odom(self):
        pos = self.caldot()
        self.x_pos += round(pos['xdot'],4)
        self.y_pos += round(pos['ydot'],4)
        self.theta_pos += round(pos['thetadot'],2)
}
```

Code 9.26: This function is used to calculate the odometry, based on the linear translation and angular rotation.

When a message is received on the topic */HJC/Vel_robot/Set_velocity* a callback function is used, in this callback function, the data from the message is transmitted through the 3 serial communication, and it transmits the data to the Usteppers as well as to the brakes.

9.4.2.1 Motor-Controller (Ustepper)

The Ustepper code is named *wheel_HJC_new.ino*, in this program, the library from Ustepper is used, and the same program is uploaded to each motor. In the setup phase of the program, the Ustepper setup function needs to know some data, how many steps to a full rotation, this is set to 200 steps, and the PID controller parameters P = 10, I = 0, D=0 these values were found through trial and error. Furthermore, the motor was set to run at a continuous velocity. A serial communication was opened as well at the BAUD rate of 1152000.

After the initial setup of the Usteppers, the program loops. In every loop the serial communication is checked if any data is available, the data is then cast to an integer and set as the velocity of the wheel. In every loop the encoder is used, to get the RPM of the wheels as well as the position of the motor shaft, this data is then put in the format as explained above.

9.4.2.2 Brakes (Arduino)

The code for the brakes is named *breakmodeteensy.ino*, this program controls the two servos used for braking. In the setup phase of the code, the servos are attached on pins 8 and 9 and a serial communication is open at the BAUD rate of 1152000.

In the main loop, it is checked if there is incoming data the serial, the data that comes in is then cast to an integer. If this integer is 0 the brakes will be applied else the brakes would release and let the wheels spin.

10 - Testing

This chapter goes over the testing of the system that was performed and the results of the tests are shown in the chapter 11.

10.1 Unit Tests

In this section, the different unit tests that were performed are described.

10.1.1 Power Supply

Testing of the voltages is for all prototypes, in this test it is ensured that all voltages to the different devices are with-in specifications.

- The Raspberry pi 4 needs between 4.7 V to 5.25 V [48]
- The Arduino Uno needs between 2.7 V to 5.5 V [49]
- The Tower-Pro MG90S servos needs between 4.8V to 6V [50]
- The Ustepper S needs between 12 V to 42V [30]
- The IMU needs 3V to 5V depending on the connection pin. [37]

This is the test with a multi-meter, the reason for measuring these power supplies can be regulated, this will ensure that there is less likely to burn components.

10.1.2 IMU - Pitch

The pitch that was calculated from the IMU, needs to be validated, this is done by placing the Hybrid Robot prototype at various angles that were measured by an angles measurement device. The measurement and the calculated pitch are correlated, if the difference is less than 0.5 % it is accepted.

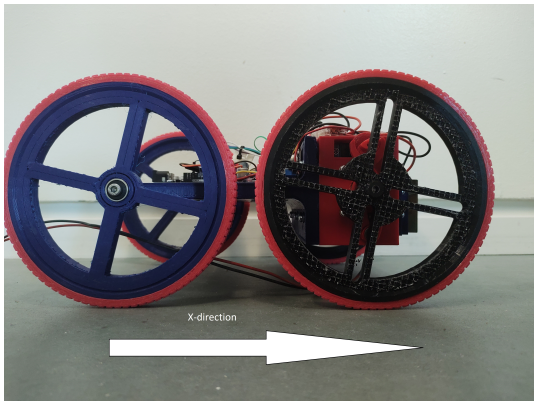


Figure 10.1: This is a figure of the Hybrid Robot, where the positive direction of x is marked with the white arrow.

10.1.3 Motor Control

The control of the motors through the Ustepper was tested by setting the input of the function setRPM, this function was implemented by the Ustepper company. Using the Arduino IDE's serial monitor to set the RPM of the wheel, the speeds were measured by a tachometer. The tachometer used during all tests is named AdventOptial A2103[51] Tachometer from the company Compact. Besides the test of control of the motors' speed, the motor spins directions are also tested so that it can decipher which direction the motor shaft has to spin.

- -600 RPM
- -300 RPM
- -100 RPM
- 0 RPM
- 100 RPM
- 300 RPM
- 600 RPM

10.1.4 Odometry

The position of the Hybrid Robot is only tested in the x-direction of the cart, as seen in the figure 10.1, the test is done by moving the cart 2 meters forward and backward 5 times in total, this means the Hybrid Robot has moved 20 meters in total. The data is recorded in a rosbag file for documentation. The test will be considered a success if the odometry is within 5% of the expected value, which is +2 meters in the x-direction.

10.1.5 Hybrid Robot Velocity

The angular velocity of the wheels was tested by commanding the motor controller to run at various velocity clockwise and counter-clockwise on the wheel to be tested. The velocities were set in RPM and the wheel velocity is measured by a tachometer, used at the center of the wheel axis. It was also displayed trough the encoder that is implemented in the Ustepper S, the feedback from the encoder and tachometers are then compared. The velocities were as following:

- -600 RPM
- -300 RPM
- -100 RPM
- 100 RPM
- 300 RPM
- 600 RPM

The test will be validated if the difference between the encoder and the tachometer is less than 1% overall.

10.1.6 Communication - Topics

Testing the publishing of the topics running within the Hybrid Robot.

- Vel_robot/Set_velocity
- Vel_robot/Current_velocity
- IMU/Pitch
- IMU/angular_velocity
- Odom
- State-machine

10.1.7 Communication - Serial

Testing the serial communications running within the Hybrid Robot, are working and the data is interpreted correctly. The test was done in both directions from the RPI to the device and vice versa.

- Right Motor to RPI4
- RPI4 to Right Motor
- Left Motor to RPI4
- RPI4 to Left Motor
- Breaks(Arduino)

10.1.8 Brakes

This test is to ensure that brakes are applied when receiving a signal and are able to stop the wheels from turning at speed.

The test is carried out by running the motor up to 300 RPM, and applying the brakes and stop the supply of current to the motor. This test is carried out on both wheels.

10.2 Experimental Tests

In this section, a test of the sub-systems, such as the Hybrid Robot is validated.

10.2.1 Drive State

The drive state will be tested, by accelerating the Hybrid Robot to the desired velocity that is needed for the swing-up motion.

10.2.2 Swing-up State

This test will combine the drive state and the brake system. The Hybrid Robot, is accelerated as in the prior test to a desired velocity, after the velocity is obtained, the motor will be set to zero RPM and the brakes will be applied. If the test is successful, the Hybrid Robot will swing-up to at least 90 degrees.

10.2.3 Balancing State

The balancing State is tested with three different control methods, PID, State feedback, and an LQR controller. Testing the PID and the State feedback controller is done by tuning PID parameters and the K matrix for the state feedback controller until the Hybrid Robot is

stable and can balance for at least 25 sec. The LQR is tested through the calculated matrix of the motion equations model.

10.2.4 Catch State

The final test is combining the 2 states, swing-up and balancing. The Hybrid Robot is in this test manually manipulated by a test person, the test person has to push the back end of the Hybrid Robot, as seen in the figure 11.12 with enough force to make the robot swing-up at an angle of 90 degrees. When the pitch of the Hybrid Robot is over 60 degrees the LQR controller will begin to take effect.

11 - Results and Discussion

This chapter will discuss the testing results and the implementation that was evaluated concerning the performed test. First, the tests data will be evaluated which respect to the requirement for the system. After all the test have been evaluated, there will be a general discussion and evaluation of the whole project.

11.1 Unit Tests

The reason for running these unit test is to discover if any problems is with the basic parts, that has to work before a test of sub-systems, if trying to test a sub-system with a basic part not working, it will be harder to find the individual part of the system that is not working correctly.

11.1.1 Power Supply

The measured voltages can be seen in the table 11.1.

Table 11.1: Table of voltage measured at different components.

Component	V
XL6009 for RPI	4.98V
XL6009 for break system	5.9V
Ustepper	16.0V
IMU Gy-91	3.29V

This test was carried out every time a new component was introduce into the system, to ensure that all components had the proper voltage. Furthermore all wires were checked over by two persons before turning on the power for the Hybrid Robot, this was to lessen the opportunity of short circuit. The voltages of the power supply test was found to be in working order, which means that test is considered validated.

11.1.2 IMU - Pitch

The pitch of the Hybrid Robot was tested at 5 different angles, 0, 45, 90, 135, and 180 degrees, the results of this test can be seen in graph 11.1.

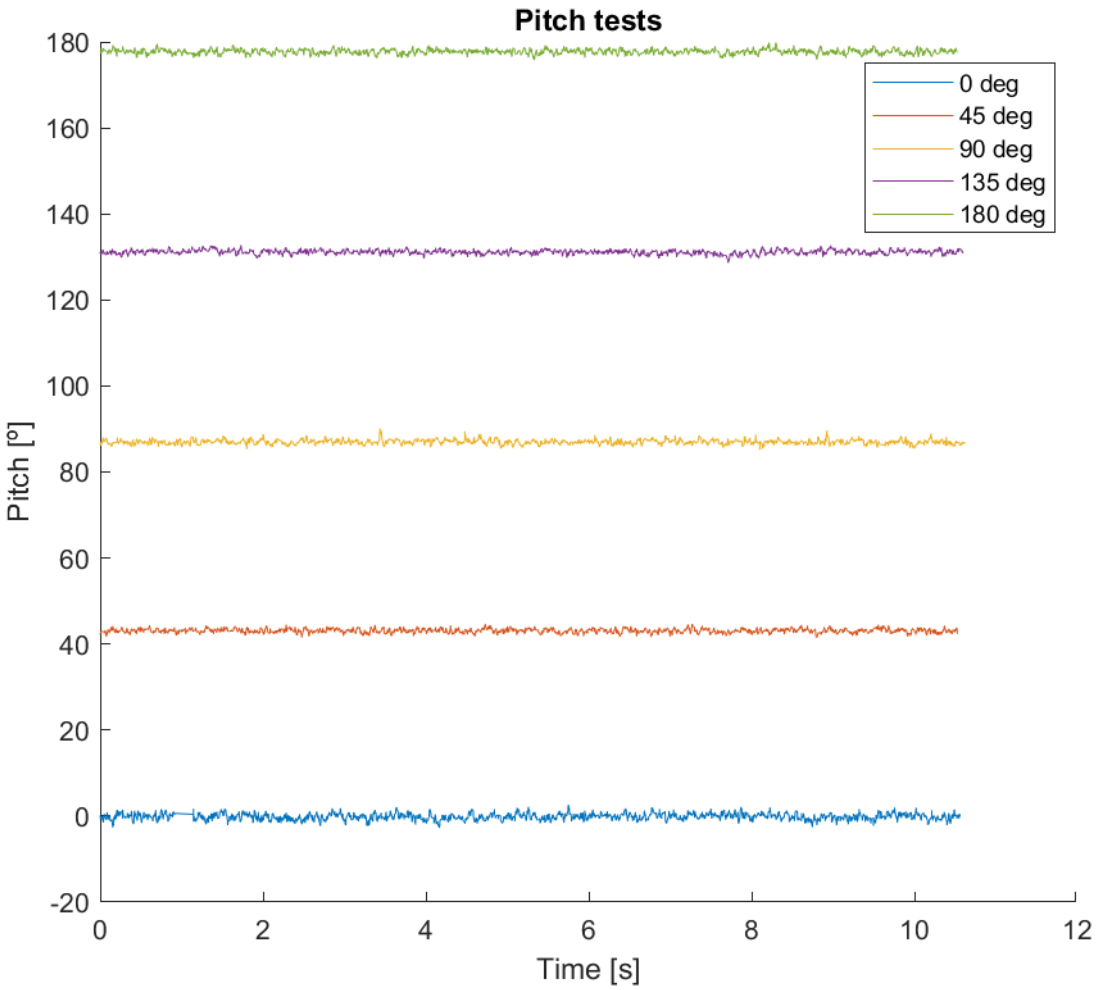


Figure 11.1: The graph shows the output from testing the pitch in 5 different positions.

Table 11.2: The table shows the minimum, maximum, and mean values of the pitch test.

g Degrees	max	min	mean
0	2.6114	-2.727	-0.12949
45	44.6428	41.5375	43.0846
90	90.0798	85.155	86.9917
135	132.7416	128.5888	131.1409
180	179.8622	175.8305	177.7246

As shown in graph 11.1 and table 11.2 there is a small amount of noise in the measure-

ments, the reason for the missing degrees could be the way that the angel of the Hybrid Robot was measured in the real world, therefor the results a considered valid.

11.1.3 IMU - Angular

The results of testing the IMU gyroscope output against the encoder of the Ustepper can be seen in the graph 11.2.

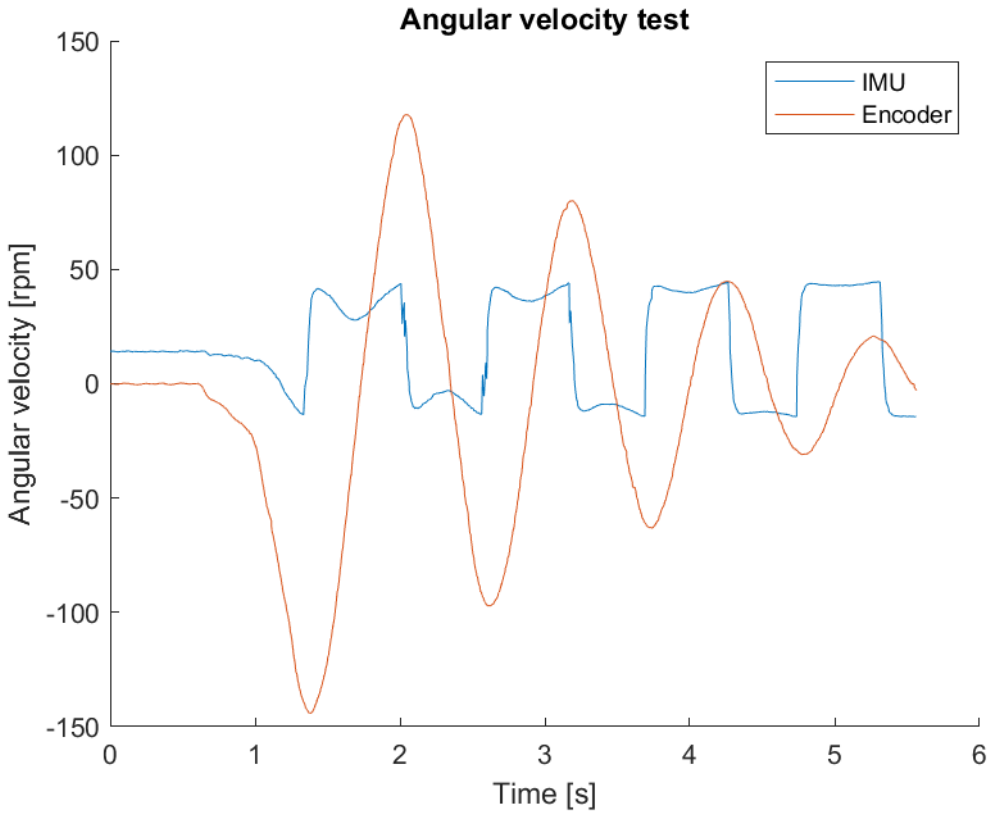


Figure 11.2: Results of the angular velocity of the IMU, measured against the encoder data from the Ustepper.

The graph shows that there is a correlation between them, but the two outputs are not the same in value.

11.1.4 Motor Control

In the test of the motor controller's ability to reach the desired RPM, the motor controller was set to run at the given speeds, as seen in table 11.3, after a second it was measured

by the AdventOptial A2103 by placing the tip on the end of the shaft of the motor, the AdventOptial A2103 then displays the measured RPM's that can be seen in table 11.3.

Table 11.3: This table shows the test results of motor control, RPM commanded is the input that is specified by the user and RPM motor output is the RPM measured by the AdventOptial A2103[51].

RPM commanded	RPM motor output
-600	-599
-300	-300
-100	-100
0	0
100	100
300	300
600	600

The motor control test showed, that it is possible command a certain RPM that the motor will follow. This test was carried out without any load on the wheels, this means that the test do not show if the motors are able to move the robot at 600 RPM, but it is at least possible to set an RPM and the motor controller will try and obtain the specified RPM.

The only RPM that did not reach the desired out put was -600 RPM, one reason for this could be that the AdventOptial A2103 only shows the RPM's in integers, meaning it could be that the RPM's was around 599,45 and this meant that it rounded down.

11.1.5 Hybrid Robot Velocity

This test is similar to the previous test, instead of measuring RPM with the AdventOptial A2103, the internal encoder in the Ustepper is tested, the results can be seen in table 11.4.

Table 11.4: This table shows the test results of out from the encoder implemented in the Ustepper[30].

RPM commanded	RPM encoder output
-600	-599.51
-300	-299.78
-100	-100.01
100	100.12
300	299.56
600	599.45

The results from the encoders were found to produce a small amount of noise at steady RPM, which could lead to a difference in the odometry out of the system.

11.1.6 Odometry

The results can be seen in the graph 11.5, the results are acquired by moving the Hybrid Robot forwards and backward two meters five times, as seen in figure 11.3,11.4.

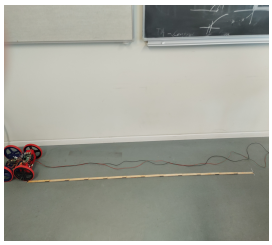


Figure 11.3: Start position in the odometry test.

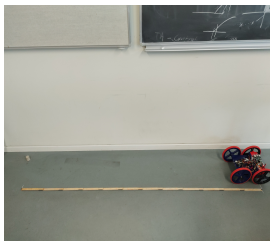


Figure 11.4: End position in the odometry test.

The graph 11.5 shows that at the end of the test, the odometry showed that the y component was 0 as well as the angel of the Hybrid Robot, this is also what was expected from the test. The x component ended up at 1.98 meters, which is less than 5% of the expected result of 2.0 meters. A reason for the missing distance could be due to the slip of the wheels on the ground and noisy measurements from the Ustepper encoder.

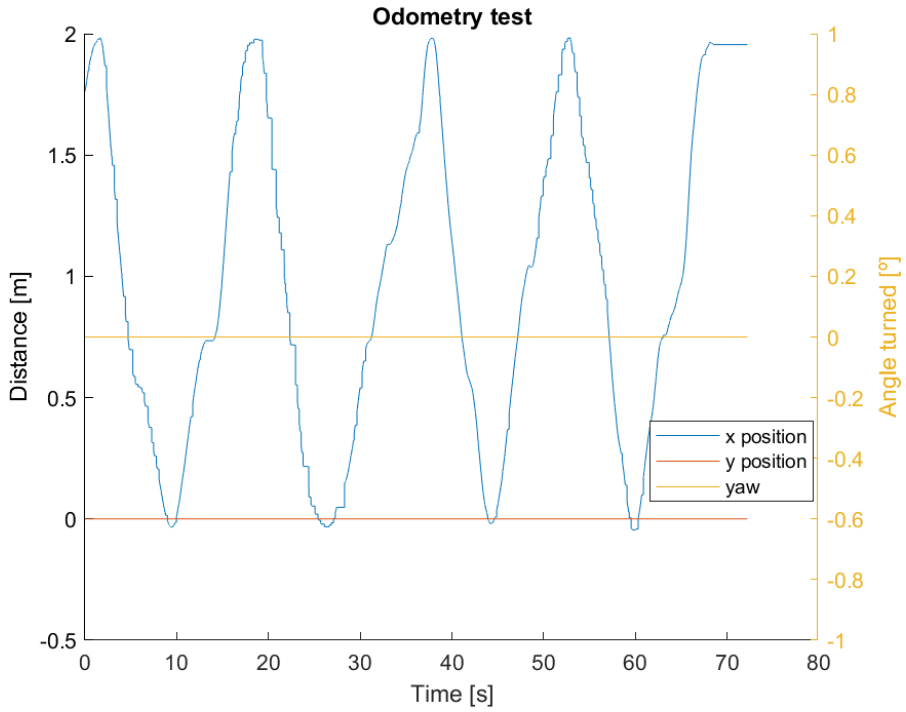


Figure 11.5: The results from the test of the odometry implemented on the Hybrid Robot.

The odometry did not hit 2 meters exactly, it came within 1% of the expected result of 2 meters.

11.1.7 Communication - ROS Communication

During the test of the communication between nodes, it was found that some nodes were not receiving any response from the ROS publisher, a reason for this was a typo in the topic name. The result of the ROS communication showed that all nodes were able to transmit and receive the messages, that are needed for the individual nodes and nodelets.

11.1.8 Communication - Serial

This test was a success as the data transmitted between Usteppers and RPI was transmitted and received correctly, the same goes for the data from the RPI to the Arduino Uno is also working as indented.

11.1.9 Brakes

The results from the brake test showed that the wheels were able to stop within less than 1 sec, which is much quicker than using the motors as brakes. This also lead to lowering of the operating temperature of the motor controller board.

11.2 Testing of the Experimental Hybrid Robot System

The test in this section was recorded by video, the different controllers also have a rosbag recording of the output from the system. These rosbags were not recorded during the driving and swing-up state.

11.2.1 Drive State

The first test, of the drive-state, showed that there was a need for reducing the weight of the Hybrid Robot, in that the robot could not move, without the aid of a person to push the robot along. After the reduction of the weight, the test was carried out again, the robot was able to reach the desired velocity to perform the swing-up phase found in Gazebo. Moreover, during the driving phase, it was found that the Hybrid Robot, tended to turn slightly towards the right, which meant that the robot had to align a bit to the left when starting the test, or else it would hit the walls in the test area. A video of the drive state can be seen here `https://youtube.com/shorts/TgWy4cHKMK0`.

This test was pertaining the first requirement in 6.

11.2.2 Swing-up State

The swing-up state was tested after it was found that the robot was able to achieve the given velocity. The test showed that the robot was able to swing-up and go over the 90 degree minimum and land on the other side. This showed that if it is possible to flip it over, it is possible to end the swing-up motion close to the objective of 90 degrees applying less power. A video combing the drive state and the swing-up state can be seen here `https://youtube.com/shorts/3fAE0ZCKVyY?feature=share`.

11.2.3 Balancing State

In this section, the results of the different controllers used to balance the Hybrid Robot in an inverted pendulum state are discussed. In this section, the prototype is changed, and a counterweight is installed to counteract the components that are placed on onside of the Hybrid Robot.

This test was pertaining the second requirement in 6.

PID control The test of the PID controller with different parameters was done many times, the result seen in graph 11.6,11.7, is the one that the team behind the project found work the best. The PID controller was able to balance the Hybrid Robot for a duration of just 10 seconds. A video of the PID controller can be seen here `https://youtu.be/zD7ouL_Wdcg`.

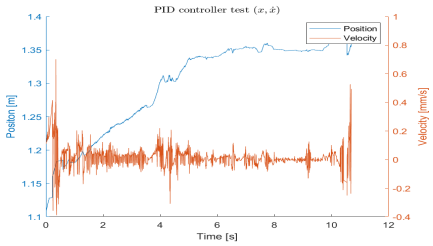


Figure 11.6: Result of the PID test, this graph is only showing the position and linear velocity.

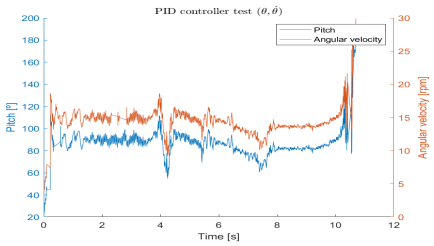


Figure 11.7: Result of the PID test, this graph is only showing the pitch and the angular rate.

State-feedback control The State-feedback controller was tested many times, in each test one of the four parameters was tuned, the parameters are pitch, angular rate, linear velocity, and position, the best result can be seen in the graphs 11.8, 11.9. The result shows that the state-feedback controller increased the control-ability of the Hybrid Robot, and increased the balancing to 12 seconds. A video of the state-feedback controller can be seen here `https://youtube.com/shorts/Zv3CVIF6Pl0`.

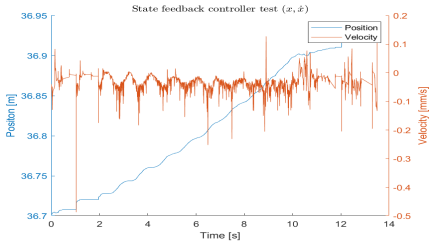


Figure 11.8: Result of the State-feedback test, this graph is only showing the position and linear velocity.

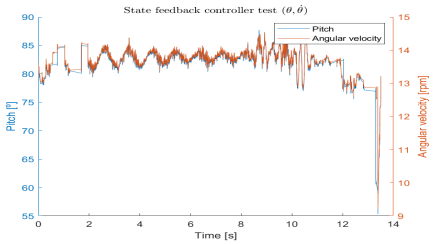


Figure 11.9: Result of the State-feedback test, this graph is only showing the pitch and the angular rate.

LQR control The LQR parameters were derived from the real world physical in Matlab, these parameter were then implemented in the controller. The result of that test can be seen in the graphs 11.10, 11.11.

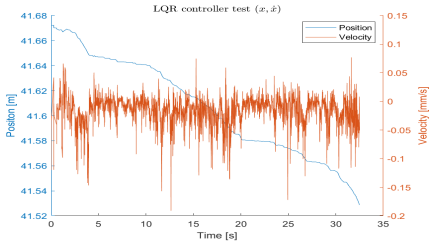


Figure 11.10: Result of the LQR test, this graph is only showing the position and linear velocity.

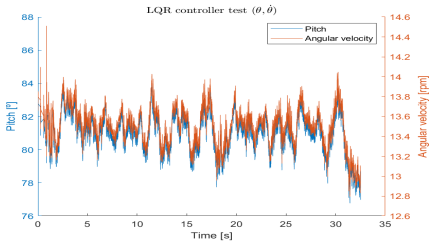


Figure 11.11: Result of the LQR test, this graph is only showing the pitch and the angular rate.

It shows a great increase over the two other controllers tested, by doubling the duration that the Hybrid Robot is able to balance. Although at first glance of the graphs of the LQR is moving more erratic, but to the better balancing the scale of the LQR is off compared to the other controllers.

This test was pertaining the third requirement in 6.

A video of the LQR controller can be seen here `https://youtube.com/shorts/xbFDzqCOYbU`.

11.2.4 Catch

To test if the controllers can stabilize to Hybrid Robot after a swing-up motion is performed, the swing-up state was done manually by a person that gives the Hybrid Robot an initial kick, as seen in the figure-11.12, such that it would go towards the inverted pendulum shape, as seen in the figure 11.13. This initial kick was hard to manage, but after many attempts, it was possible to be more consistent in where the Hybrid Robot ended up. The reason for doing this manually was that the acceleration and running of the Hybrid Robot was infeasible, due to the distance that Hybrid Robot had to travel to have enough velocity to do the swing-up motion.

Figure 11.12: Here the Hybrid Robot is pushed up manually, to perform a swing-up motion with out moving the Hybrid Robot.

Figure 11.13: Here the Hybrid Robot is trying to balance after the initial swing-up motion done manually by a test person.

It was found while testing that the motors had a hard time following the controller, one of the reasons for this could be that it has to counter act the up swing of the Hybrid Robot and this leads to the saturation of how fast the motors can accelerate without losing the magnetic field.

One test showed promise in that, the Hybrid Robot was able to balance for a couple of seconds, this was using the LQR controller, a video of this can be seen here `https://youtu.be/j2J-NO6jLVg`.

This test was pertaining the fourth requirement in 6.

12 - Conclusion

This project investigated: ***How can a hybrid robot between a car and a Segway be constructed to allow transition between the two modes of operation with a fixed body.*** Through the simulation, it was found that it was possible to make a 4-wheel cart, perform a swing-up motion and end up at an inverted pendulum state. To perform this swing-up motion, it was found that the wheels that would do the braking force was required to have high inertia, to help swing up the cart.

After it was found to be possible in simulation to do the swing up motion, the motors and the motor controller that would be used to drive the Hybrid Robot have to be selected, this was done based on the torque found in simulation. Due to limited funding, it was not possible to get a motor that could produce the amount of torque, the motor chosen was readily available, from most electronic shops. The motor controller, was the expensive part, first an attempt to make a stepper controller was tried out to save money, but this was a time-consuming task, therefor a request to get more money to buy a controller, this controller the Ustepper also had the benefit of having extra sensor the encoder, which is used to find the shafts position and rotational speed.

During testing it was found that the motors could not drive the Hybrid Robot in what was its current state, it was too heavy therefore a second design was made around the motors selected, which allow the Hybrid Robot to accelerate to the desired velocity. After being able to accelerate it was found that the motors were not able to stop the robot hard enough to make the Hybrid Robot do the swing-up motion, therefor a set of brakes was implemented, now the Hybrid Robot was able to swing-up.

While testing the inverted pendulum state, it was found that the motors had trouble to move fast enough the wheels to the control inputs due to the high inertia of the wheels, this lead to the use of lighter wheels in the inverted pendulum state, these wheels were in the same dimension as the one that they substituted. This made the Hybrid Robot motors able to follow the changes from the PID, State feedback, and LQR controller.

The test of the Hybrid Robot where it has to catch itself in the swing-up state showed that the motors with the lighter wheels were not able to stabilize due to the low torque. This leads to the conclusion that it is possible to build a Hybrid Robot between a car and a Segway but it still needs some design optimization and different electronic components.

13 - Future Works

This chapter goes over some ideas that were thought about but could not be implemented due to the limited amount of time for this single project.

- The robot to control is a highly non-linear system and all the control methods implemented in this project are designed to work on linear systems. Great improvements on the control would be using a control method that allowed a non-linear model as it can be an MPC which can also include more detailed limits and constraints.
- A high limiting factor has been the motors used to move and control the robot as they did not have enough torque to perform correctly some of the motions that are needed. For example, it needs a lot of distance to get the needed velocity to brake and flip due to the fact the motor can not accelerate fast enough. The other example is that when it is stabilizing and the robot tries to move fast to recover it cannot do it. To solve this, it is needed to obtain motors with higher torque to be able to handle the correct control.
- The design of the Hybrid Robot has been done from scratch. That means that it still needs a lot of iteration and time to achieve an optimal design. Some of the key points are to be able to distribute all the electronic components so the center of mass is located in the center of the vertical and uniformly distributed.

Figure 13.1: Diagram of the ideal position of the center of mass of the robot.

- A good thing to always improve is the accuracy of the sensors. Even with a perfect controller, if the sensors are not good, it will be very complicated to control correctly the robot. So, a key to improving the performance is to improve the sensors.
- Another long-term project would be to implement a closer model to the one described in the paper [14]. That would include covering the ability of the robot to jump. This could be performed by using a spring to launch a mass that would produce enough force to bring the robot with it. That mass could be part of the body with a prismatic joint.
- Another feature would be to have the model in a 3D space so it could move in all directions and turn.

Bibliography

[1] The Editors of Encyclopaedia Britannica. simple machine. https://www.britannica.com/technology/simple-machine. (accessed: 04.05.2022).

[2] Robot Platform. Wheeled robots. http://www.robotplatform.com/knowledge/Classification_of_Robots/Types_of_robot_wheels.html. (accessed: 04.05.2022).

[3] Palak Purohit, Poojan Modi, and Udit Vyas. Kinematic control of 2-wheeled segway. 2021.

[4] Mordechai. Ben-Ari. *Elements of Robotics*. Springer International Publishing, Cham, 1st ed. 2018. edition, 2018.

[5] Steven M LaValle. Differential drive. http://planning.cs.uiuc.edu/node659.html.

[6] Robot Platform. Wheel control theory. http://www.robotplatform.com/knowledge/Classification_of_Robots/wheel_control_theory.html.

[7] Benjamin Shamah. Experimental comparison of skid steering vs. explicit steering for wheeled mobile robot," m.sc, 1999.

[8] Whitsunday Segway Tours. What is a segway? https://www.whitsundaysegwaytours.com.au/whatisasegway. (accessed: 11.05.2022).

[9] IEEE Robots. Segway. https://robots.ieee.org/robots/segway/. (accessed: 11.05.2022).

[10] Ascento Robotics. Ascento. https://www.ascento.ch/.

[11] Jeffrey Delmerico, Stefano Mintchev, Alessandro Giusti, Boris Gromov, Kamilo Melo, Tomislav Horvat, Cesar Cadena, Marco Hutter, Auke Ijspeert, Dario Floreano, Luca M. Gambardella, Roland Siegwart, and Davide Scaramuzza. The current state and future outlook of rescue robotics. *Journal of Field Robotics*, 36(7):1171–1191, 2019.

[12] Jeongsoo Lim, Hyoin Bae, Jaesung Oh, Inho Lee, Inwook Shim, Hyobin Jung, Hyun-Min Joe, Okkee Sim, Taejin Jung, Seunghak Shin, Kyungdon Joo, Mingeuk Kim, Kangkyu Lee, Yunsu Bok, Dong-Geol Choi, Buyoun Cho, Sungwoo Kim, Jungwoo Heo, Inhyeok Kim, and Jun-Ho Oh. *Robot system of DRC-HUBO+ and control strategy of team KAIST in DARPA robotics challenge finals*, pages 27–69. 04 2018.

[13] Scott Kuindersma, Robin Deits, Maurice Fallon, Andrés Valenzuela, Hongkai Dai, Frank Permenter, Twan Koolen, Pat Marion, and Russ Tedrake. Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot. *Autonomous Robots*, 40, 07 2015.

[14] Traiko Dinev, Songyan Xin, Wolfgang Merkt, Vladimir Ivan, and Sethu Vijayakumar. Modeling and control of a hybrid wheeled jumping robot. *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct 2020.

[15] Robin Murphy, Satoshi Tadokoro, Daniele Nardi, Adam Jacoff, Paolo Fiorini, Howie Choset, and Aydan Erkmen. *Search and Rescue Robotics*, pages 1151–1173. 01 2008.

[16] Corina Warfield. The disaster management cycle. https://www.gdrc.org/uem/disasters/1-dm_cycle.html.

[17] Manuel A. Roehrl, Thomas A. Runkler, Veronika Brandtstetter, Michel Tokic, and Stefan Obermayer. Modeling system dynamics with physics-informed neural networks based on lagrangian mechanics. *CoRR*, abs/2005.14617, 2020.

[18] Gene Franklin, J.D. Powell, and M.L. Workman. *Digital Control of Dynamic Systems-Third Edition*. 12 2021.

[19] Wolfram MathWorld. Matrix inverse. https://mathworld.wolfram.com/MatrixInverse.html.

[20] Iván D. Díaz-Rodríguez. *Analytical Design of PID Controllers*. Springer International Publishing, Cham, 1st ed. 2019. edition, 2019.

[21] M. Sami Fadali and Antonio Visioli. *Digital Control Engineering: Analysis and Design*. Elsevier Science & Technology, San Diego, 2012.

[22] Russ Tedrake. *Underactuated Robotics*. 2022.

[23] eigenvalue, 2020.

[24] Gene F. Franklin. *Feedback control of dynamic systems*. Pearson Education Limited, Harlow, England, eighth edition. edition, 2020.

[25] Roger A. Horn. *Matrix analysis*. Cambridge University Press, Cambridge, 1985.

[26] myhobby cnc.de. High torque hybrid stepping motor. https://shop.myhobby-cnc.de/media/pdf/af/ee/4f/nema17.pdf. (accessed: 13.05.2022).

[27] etechnophiles. Guide to nema 17 stepper motor dimensions, wiring pinout. https://www.etechnophiles.com/guide-to-nema-17-stepper-motor-dimensions-wiring-pinout/#specifications-of-nema-17. (accessed: 13.05.2022).

[28] RS-online. The complete guide to dc motors. https://ie.rs-online.com/web/generalDisplay.html?id=ideas-and-advice/dc-motors-guide. (accessed: 13.05.2022).

[29] Carmine Fiore. Stepper motors basics: Types, uses, and working principles. https://www.monolithicpower.com/stepper-motors-basics-types-uses. (accessed: 13.05.2022).

[30] Ustepper. usteppers github documentation. https://github.com/uStepper/uStepperS. (accessed: 08.05.2022).

[31] mae.ufl.edu. Friction coefficients between different wheel/tire materials and concrete. https://mae.ufl.edu/designlab/Class%20Projects/Background%20Information/Friction%20coefficients.htm. (accessed: 13.05.2022).

[32] raspberry pi. raspberry pi bcm2711 arm peripherals. https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf. (accessed: 11.05.2022).

[33] https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf, JUN 2019. (accessed: 08.02.2022).

[34] seedstudio.com. Raspberry pi 4 vs. 3. https://www.seeedstudio.com/blog/2019/09/30/raspberry-pi-4-vs-pi-3-all-the-major-differences/. (accessed: 11.05.2022).

[35] https://www.ubiquityrobotics.com/downloads-raspberry-pi/, JUN 2019. (accessed: 08.02.2022).

[36] https://learn.ubiquityrobotics.com/, JUN 2019. (accessed: 08.02.2022).

[37] Plexsihop.it. Gy-801. https://www.plexishop.it/en/10-axis-gy-801-module-gyroscope-accelerometer-magnetometer-pressure-sensor.html. (accessed: 09.05.2022).

[38] XLSEMI. 400KHz 60V 4A switching current boost / buck-boost / inverting dc/dc converter. https://www.haoyuelectronics.com/Attachment/XL6009/XL6009-DC-DC-Converter-Datasheet.pdf. (accessed: 11.05.2022).

[39] ArduinoTech. xl6009 dc-dc adjustable step power converter. https://arduinotech.dk/shop/xl6009-dc-dc-adjustable-step-power-converter/. (accessed: 11.05.2022).

[40] *Robot Operating System (ROS) The Complete Reference (Volume 3)*. Studies in Computational Intelligence, 778. Springer International Publishing, Cham, 1st ed. 2019. edition, 2019.

[41] Gazebo. Gazebo. https://gazebosim.org/home. (accessed: 14.05.2022).

[42] ROS Wiki. urdf. http://wiki.ros.org/urdf. (accessed: 14.05.2022).

[43] Marc B. Reynolds. xacro. http://wiki.ros.org/xacro. (accessed: 15.05.2022).

[44] ROS Wiki. Converting to euler & tait-bryan. http://marc-b-reynolds.github.io/math/2017/04/18/TaitEuler.html. (accessed: 19.05.2022).

[45] tomstewart89. Statespacecontrol. https://github.com/tomstewart89/StateSpaceControl.

[46] James Chen. Heuristics. https://www.investopedia.com/terms/h/heuristics.asp. (accessed: 26.05.2022).

[47] ros-mobile robots.com. Odometry. https://ros-mobile-robots.com/theory/modeling-control/odometry/. (accessed: 13.05.2022).

[48] Raspberry pi foundation. RPI4 Datasheet. https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf. (accessed: 08.05.2022).

[49] Arduino. Arduino datasheet. https://docs.arduino.cc/resources/datasheets/A000066-datasheet.pdf. (accessed: 08.05.2022).

[50] Metal Gear. Mg90s datasheet. https://datasheetspdf.com/pdf-file/1106582/ETC/MG90S/1. (accessed: 08.05.2022).

[51] Compact. Advent professional optical/contact tachometers a2103. https://docs.rs-online.com/81c4/A700000007190985.pdf. (accessed: 09.05.2022).

A - Github Appendix

The code that has been created in this project can be found a at GitHub at this hyperlink
`https://github.com/ipujol10/hybrid_robot`

B - Movies Appendix

Video of test of driving mode https://youtube.com/shorts/TgWy4cHKMK0.

Video of test of swing-up mode using the brakes https://youtube.com/shorts/3fAE0ZCKVyY?feature=share.

Video of test of balancing mode using PID https://youtu.be/zD7ouL_Wdcg.

Video of test of balancing mode using State feedback control https://youtube.com/shorts/Zv3CVIF6Pl0.

Video of test of balancing mode using Linear–quadratic regulator https://youtube.com/shorts/xbFDzqCOYbU.

Video of test of combining balancing mode and the swing-up mode https://youtu.be/j2J-NO6jLVg.

C - Schematics

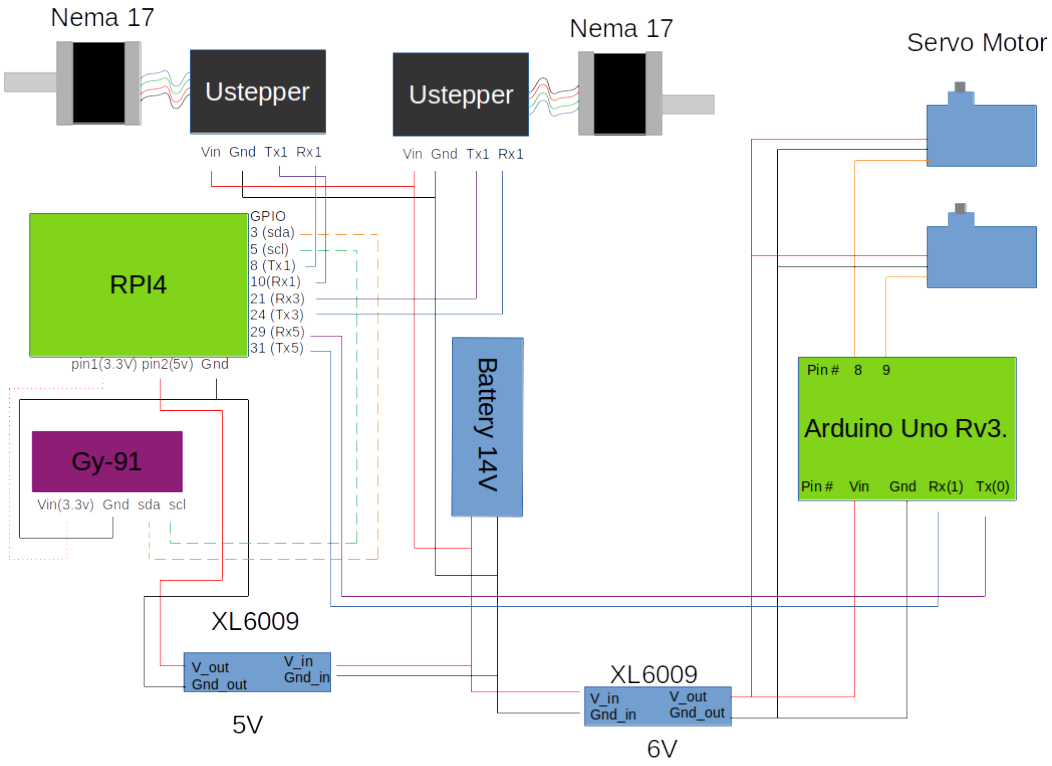


Figure C.1: This is the schematics of how the hardware components are connected in this project

D - Node Communication

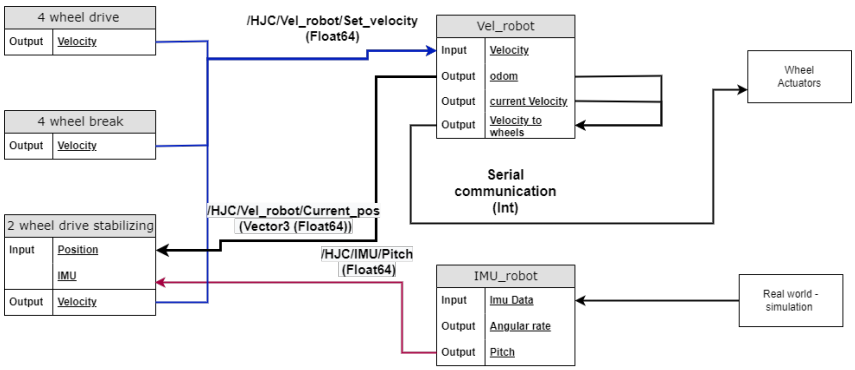


Figure D.1: This figure illustrate the communication between nodes in this project.

E - Model Values

In this appendix there is going to be listed the values used on the model that in the end leads to the matrices in the section 7.2.2:

Variable	Value	Units
d	0.03773	m
m_w	0.1918	kg
m_b	0.9792	kg
g	9.81	m/s^2
I_w	7.2229e-4	$kg\ m^2$
I_b	7.3246e-2	$kg\ m^2$
R	0.075	m

Table E.1: Summary of the values used on the model.