# Design of a high-level control layer and wheel contact estimation and compensation for the pipe inspection robot PIRATE

N.M. Geerlings

Master of Science Thesis


TU Delft Delft University of Technology

UNIVERSITY OF TWENTE.

# Design of a high-level control layer and wheel contact estimation and compensation for the pipe inspection robot PIRATE

Master of Science Thesis

For the degree of Master of Science in Mechanical Engineering at Delft University of Technology

N.M. Geerlings

November 22, 2018

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology



This thesis project was executed at the Robotics and Mechatronics group at the University of Twente.


Delft University of Technology
Department of
Department of BioMechanical Engineering (bme)

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical, Maritime and Materials Engineering (3mE) for acceptance a thesis entitled

# Design of a high-level control layer and wheel contact estimation and compensation for the pipe inspection robot PIRATE

by

N.M. Geerlings

in partial fulfillment of the requirements for the degree of

Master of Science Mechanical Engineering

Dated: November 22, 2018

Supervisor(s):

Prof. Dr. R. Babuska

N. Botteghi, MSc

Prof. Dr.-Ing. H. Vallery

# Summary

Automation of pipe inspections by using robots has many advantages over manual pipe inspection, such as faster, more consistent and more accurate inspection. For this reason the pipe inspection robot PIRATE is being developped by the University of Twente.

In this project the first goal is to design a high-level control layer, which should allow the robot to autonomously move through various types of pipe segments, without falling. The second goal is to estimate the wheel slip ratio and friction coefficient. The Magic formula and Gaussian processes are used to estimate their relation. The third goal is to design a traction controller, to ensure reliable odometry and prevent falls in a vertical pipe.

The high-level control layer is designed based on the motion primitives. In order to structure the behaviour of the robot, five finite state machines are used. The joints and wheels can be controlled by position control or open-loop voltage control. A set of fourteen commands is used to control the robot.

The high-level controller is implemented in ROS using C++, in the RobMoSys style. At the Skill level, nodes for motion primitives and for sequences of motion primitives are implemented. An extra detector node can trigger a sequence by using information from e.g. a camera. The traction controller incorporates velocity control and gravity compensation for the wheels, and a controls the clamping force for the bending joints. This controller is implemented in a Simulink simulation model of the PIRATE.

To estimate the wheel slip and friction force, the PIRATE drives forward and backward in a 2D straight 'pipe' and in a real straight pipe. The linear velocity is determined by a camera with a marker detection algorithm.
To evaluate the autonomy of the robot, the robot has to drive autonomously through a 2D 90° mitre bend. In simulation the robot has to drive up and down in a horizontal and vertical pipe while using traction control.

When driving, wheel slip ratio's up to 0.3 are observed. Differences in slip ratio are observed when the movement of the robot is obstructed. For the friction coefficient large differences between the wheels are observed.
The robot is able to move through the 2D mitre bend, with the setup tilted up to an angle

of 28°, when commands are provided by the operator, as well as when the robot is triggered by the perceived location of the marker.
Simulation shows that the smallest odometry error is achieved with the traction controller that has a velocity controller, but no gravity compensation, and either a controlled clamping torque or fixed joint torque.

The relation between the wheel slip and friction coefficient can not be identified with either the Magic formula or the Gaussian processes, due to inconsistent behaviour of the robot. The software architecture successfully allows for driving through a mitre bend autonomously. The traction controller is able to select a proper clamping torque such that the PIRATE does not slide out of a vertical pipe in simulation.

## Symbols

| | |
|---|---|
| $\tilde{}$ | Interpolated and filtered. |
| $D$ | Duty cycle of the PWM. |
| $\mathbb{E}$ | Expected value. |
| $F_{\mathsf{F}}$ | Friction or parallel force (N). |
| $F_{\mathsf{N}}$ | Normal or perpendicular force (N). |
| $\mathcal{GP}$ | Gaussian process. |
| $I$ | Current (A). |
| $k(\mathbf{x}, \mathbf{x}')$ | Covariance matrix of $\mathbf{x}$. |
| $m(\mathbf{x})$ | Mean of $\mathbf{x}$. |
| $\mathcal{N}$ | Normal distribution. |
| $r$ | Gear ratio or wheel radius. |
| $v$ | Linear velocity (m/s). |
| $\mathbb{V}$ | Variance. |
| $\gamma$ | Angle of a bend joint (rad) or skew coefficient in intrinsic camera matrix. |
| $\eta$ | Gear efficiency. |
| $\kappa$ | Slip angle of the wheel (rad). |
| $\lambda$ | Slip ratio. |
| $\mu$ | Friction coefficient. |
| $\mu_{\mathsf{max}}$ | Maximum friction coefficient. |
| $\sigma$ | Standard deviation. |
| $\tau$ | Torque (Nm). |
| $\phi$ | Angle of the rotation joint (rad). |
| $\theta$ | Angle of a wheel (rad). |
| $\psi$ | Pitch or roll angle (rad). |
| $\omega$ | Angular velocity (rad $s^{-1}$). |

## Abbreviations

| | |
|---|---|
| CSV | Comma-separated values. |
| DC | Direct current. |
| DOF | Degrees of freedom. |
| EC | Execution container (level). |
| EMA | Exponential moving average. |
| FSM | Finite state machine. |
| FU | Function (level). |
| GCC | GNU Compiler Collection. |
| GUI | Graphical user interface. |
| HMI | Human machine interface. |
| HSV | Hue - saturation - value. |
| IMU | Inertial measurement unit. |
| MIDI | Musical Instrument Digital Interface. |
| OS | Operating system (level). |
| PAB | Partially autonomous behaviour. |
| PICO | PIRATE control. |
| PIRATE | Pipe Inspection Robot for Autonomous Exploration. |
| POM | Polyoxymethylene. |
| ppr | Pulses per revolution. |
| PVC | Polyvinyl chloride. |
| PWM | Pulse width modulation. |
| RGB | Red - blue - green. |
| RobMoSys | Composable Models and Software for Robotic Systems. |
| ROS | Robotic Operating System. |
| SE | Service (level). |
| SK | Skill (level). |

| | |
|---|---|
| SMA | Simple moving average. |
| STL | Stereolithography. |
| URDF | Unified Robot Description Format. |

# Table of Contents

# Chapter 1

# Introduction

## 1-1 Context

Inspection of industrial pipelines is a slow process that costs a lot of manpower. One of the problems is that not all pipes are easily accessible. Another issue is that a part of the plant needs to be shut down for inspection. Furthermore a lot of insulation material is discarded to reach the pipe and if there are doubts about the quality of the pipe, the pipe will be discarded. Pulles et al. [2008] state that robotic pipe inspection should result in less manual labour, faster inspection, lower replacement costs, more consistent and accurate inspection, less incidents and less downtime for the plant or network. Ideally, the robot should be able to autonomously perform routine inspections and reparations the inside of the pipe.

For this reason the Pipe Inspection Robot for AuTonomous Exploration (PIRATE) is being developed by Robotics & Mechatronics (RAM) research group of the University of Twente. Its development is part of the European Smart Tooling project, which aims to make the process industry safer and more cost efficient [Smart tooling, 2018]. The current version of the PIRATE is shown in Figure 1-1. The robot has four *bending* modules, with one bending joint, one internal spring and one wheel each; a *rotational* module, with one rotational joint and two wheels; a *front* module, with one bending joint, LEDs and a camera with two degrees of freedom; and a *rear* module, with one bending joint, LEDs and an ethernet port.

The environment in which the robot will operate puts some restrictions on the behaviour of the robot. The robot has to be able to inspect pipes of 100-200mm in diameter, where mitre bends [1], T-junctions and other configurations can occur, as is shown in Figure 1-2. The robot has to able to move through these segments without getting stuck.

## 1-2 Problem statement

The PIRATE can currently only be controlled by manual inputs via a MIDI panel (Fig. A-2e). To be able to inspect pipes autonomously, high-level autonomous control is needed.

[1] Sharp bend created by combining two beveled pipes, often connected in a 90° angle.

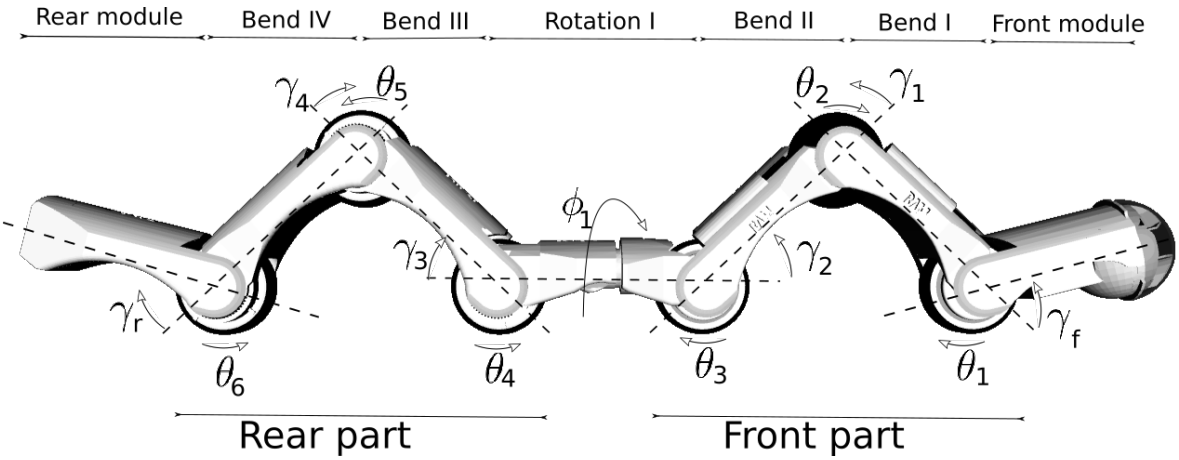



**Figure 1-1:** The kinematic model of the PIRATE Prototype II. $\gamma$ represents the bend joint angles, $\phi$ represents the rotational joint angle and $\theta$ represents the orientation of the wheels. *Permission granted by the author.* Adapted from [Dertien, 2014].

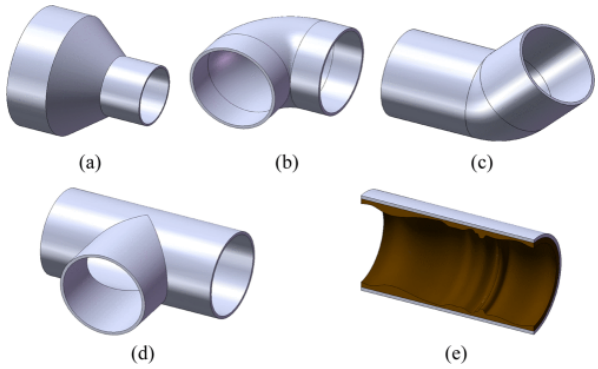



**Figure 1-2:** Pipe segments that a pipe inspection robot may encounter, such as a) a diameter change, b) a curved joint, c) an inclination, d) a T-joint or e) an uneven inner surface. *Copyright © 2011, IEEE.* [Park et al., 2011]

One of the biggest issues is that as soon as the PIRATE gets too close to a mitre bend, it cannot detect the bend anymore and therefore it does not know the distance to that bend. Currently a laser circle is projected within the pipe and based on the shape of the circle a bend can be detected. As soon as the PIRATE comes closer to a bend, a change in the circle is measured [Drost, 2009]. When the bend is reached, the bend is not detected anymore because the bend is out of the camera view. In this small window the travelled distance can not be measured via the laser projection. As such, dead reckoning is required and therefore the relation between the angular velocity of the wheels and the linear velocity of the robot should therefore be investigated. Since the relation between the angular velocity and linear velocity depends also on the forces that the PIRATE exerts on the pipe walls, these forces also need to be taken into account, especially when the PIRATE is driving in a vertical pipe.

## 1-3 Objectives

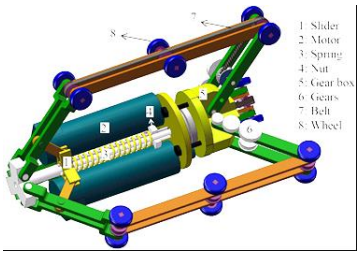
1. Design a high-level control layer, such that the PIRATE can move through a corner autonomously.
2. Evaluate the high-level control layer in a 2D setup, where the robot can only move in a horizontal plane constrained by vertical plates.
3. Identify the wheel slip and friction coefficient of the PIRATE wheels, such that the travelled distance can be approximated by measuring the wheel angles.
4. Design a traction controller, so the PIRATE can drive vertically in a pipe without falling.

## 1-4 Other works

To get an insight in how robots can move through pipes, and to find out what the current level of autonomy for other robots is, other robots were investigated. An overview of robots that are able to move through horizontal and vertical pipe sections is shown in Figure 1-3.

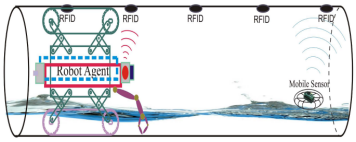
The PIKO, MRINSPECT V and AIRo robots have multiple segments, just as the PIRATE. The SPAMMS, PAROYS and AIRo show clamping behaviour, which makes them suitable for various pipe diameters. For the SPAMMS [Kim et al., 2010] and MRINSPECT V [Lee et al., 2011] autonomy is described for navigation trough a pipe network, but it is unclear if these robots are able to navigate through segments like mitre bends. For the PIC, PIKo, AQAM and PAROYS II no autonomy is described.

The AIRo, which is based on the same concept as the PIRATE, has shown to be able to move through various pipe segments autonomously by means of recognising shadows in the pipe [Kakogawa et al., 2017]. The front wheel of the AIRo allows for easily steering the robot in an environment without sharp bends (Fig. 1-3h). So far autonomous control for the AIRo has only been shown for curved joints, not for mitre bends or T-junctions.

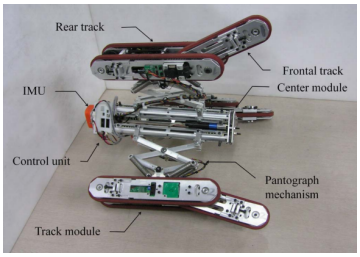



**(a)** The PIC robot (Pipe Inspection Crawler). *Open access, K.T.N.U.* [Moghaddam and Jerban, 2015]

**(b)** The PIKo robot. *Copyright © 2009, IEEE.* [Fjerdingen et al., 2009]

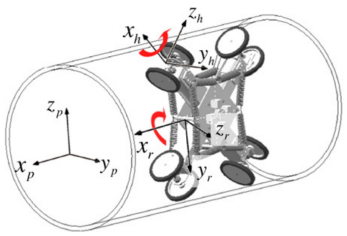



**(c)** The SPAMMS robot, previously known as FAMPER (Sensor-based Pipeline Autonomous Monitoring and Maintenance System). *Copyright © 2010, IEEE.* [Kim et al., 2010]



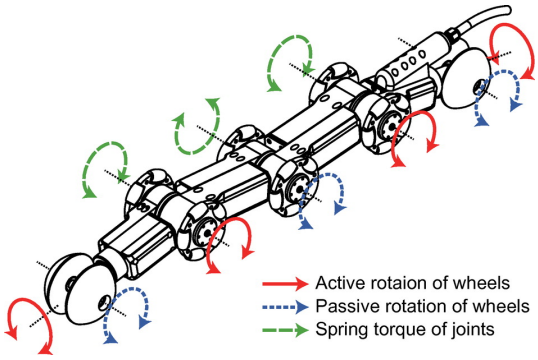

**(d)** The PAROYS II robot (Pipe Adaptive Robot of YonSei University). *Copyright © 2011, IEEE.* [Park et al., 2011]

**(e)** The MRINSPECT V robot. *Copyright © 2011, IEEE.* [Lee et al., 2011]




**(f)** The AQAM robot (Adaptable Quad Arm Mechanism). *Copyright © 2012 Elsevier Ltd. All rights reserved.* [Lee et al., 2012]

**(g)** The AIRo robot (multilink-Articulated Inspection Robot). *Rights managed by Taylor & Francis.* [Kakogawa and Ma, 2018]




**(h)** Kinematic model of the AIRo. *Rights managed by Taylor & Francis.* [Kakogawa and Ma, 2018]

**Figure 1-3:** Robots that can move through horizontal and vertical pipe segments.

## 1-5 Approach

To get familiar with the robot and its environment, a kinematic Simulink model will be built. In this model a preliminary version of the high-level control can be built and tested. The next step is to throughly investigate the current C++ and ROS implementation, mostly written by Garza Morales [2016] and Hoekstra [2018]. If needed, the current control layers will be adapted to allow for a next control layer. Afterwards the high-level control layer will be built and debugged, while maintaining the conventions as defined by the previous contributors. The high-level control layer will then be evaluated in a 2D setup, so the robot is easily accessible in case it gets stuck.

In the 2D setup, the robot should be able to drive straight for about a meter, in this setup data should be gathered from the PIRATE itself and from an external camera to measure the wheel slip and friction coefficient.

The current prototype of the PIRATE cannot yet generate enough wheel torque to prevent sliding out of a vertical pipe. The traction controller will therefore be designed in simulation. This simulation will be built in Simulink.

The product of this assignment will be the software with added high-level control and full documentation, along with an analysis of the wheel slip and friction coefficient signals and a proposal for a traction controller. Also a walkthrough of the experiment and a compact user manual for the operator will be delivered.

## 1-6 Outline

In Chapter 2, previous work on the PIRATE robot is analysed and the theoretical frameworks used in this thesis are introduced and elaborated upon. In Chapter 3 the main concepts for the new control layer are explained and the requirements are stated. The derivation of the traction controller is also provided. In Chapter 4, the changes to the current software are described and the implementation of the new control layer and the model are explained. In Chapter 5, the experimental setups will be discussed, along with the postprocessing. Chapter 6, describes the results of the development and the experiments. Eventually, in Chapter 7, these results will be discussed, conclusions will be drawn and recommendations will be given.

Appendix A discusses the actuators, sensors and other hardware components of the PIRATE robot. Appendix B describes the conversion from raw sensor values to SI units. Appendix C describes how to process camera images for marker detection. Appendix D provides a quick start guide needed to operate the PIRATE and update its software, along with lists of the used software and hardware.

# Chapter 2

# Preliminaries

This chapter, firstly, provides an overview of the hardware architecture of the PIRATE. Secondly, the methods and tools that will be used in the design chapter are explained. Finally the current state of the software architecture is described, as a reference for the updated design in the next chapter.

## 2-1 Robot hardware

To give an overview of the interaction with the robot, the hardware architecture of the PIRATE is shown in Figure 2-1. The PIRATE contains nine PICO (PIRATE control) boards, one in each module, which can actuate up to two motors and process up to two sensors each [Reiling, 2014]. An Arduino MEGA board serves as transparent bridge between the high-level (laptop) and low-level (PICO) control. High-level control is performed by Robot Operating System (ROS). User inputs can either be entered via the command line or via a MIDI panel.

All wheels and joints have position and current sensors and are actuated by DC micromotors. Additionally the springs in the Bend modules also have position sensors. individual actuators, sensors and other components are discussed in Appendix A.

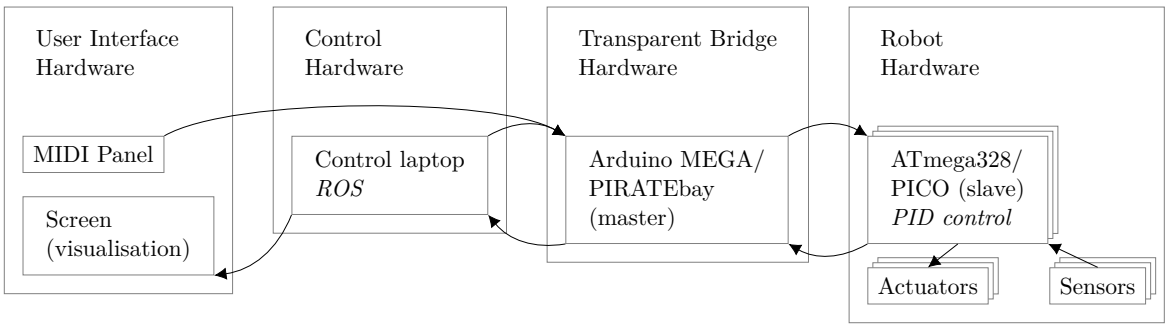



**Figure 2-1:** Hardware architecture of the PIRATE. Adapted from [Garza Morales, 2016].

## 2-2 Methods and tools

### 2-2-1 ROS and C++

For high level control of the PIRATE a software architecture on ROS is implemented [Garza Morales, 2016]. Robotic Operating System (ROS) is an open-source meta-operating system for robotics. ROS is compatible with C++ and Python, in this project C++ is used to write the source code for the nodes, which will then be executed by ROS. ROS provides communication methods for both low level and high level messages, in various communication styles. It also provides packages for e.g. kinematics, visualisation and human-machine interface (HMI), which make controlling a robot a lot easier. All communication can be stored within ROS, and played back at a later time.

The main structures in ROS are [Martinez Romero, 2014]:

- Nodes: These are the main processes. They are able to communicate asychronously with each other.
- Messages: These contain variables and can be published and subscripted to by nodes, and can therefore be used for many-to-many communication.
- Services: Request and reply version of messages. They are used for one-to-one communicaton. Services are blocking: while an instance of a service is running, ROS cannot create a new instance of this service.
- Bags: File that stores all ROS message data.

In the architecture by Garza Morales [2016] it was hard to make adaptions when the hardware was updated (e.g. when an extra sensor was installed) or when a software extension was needed (e.g. image processing). Therefore the software architecture is implemented the RobMoSys style by Hoekstra [2018]. This way hardware can be changed and software extensions can be added by rewriting a small part of the software, instead of multiple large software sections.

### 2-2-2 RobMoSys

RobMoSys (Composable Models and Software for Robotic Systems) is a European project that focusses on creating an industry-grade software development ecosystem for robotics. One of the main advantages is modularity of software [Tucci and Schlegel, 2017]. The software is divided in various levels which all have their own goal, as is shown in Table 2-1. Due to this separation of levels, autonomy can be implemented level by level. Also, if the design of the robot changes, only a small number of levels need to be adapted, instead of all.

When applied to an in-pipe exploration robot the mission could for example be exploring an unknown network or going from point A to point B in a known network. A task could be taking a right turn on an T-joint. A skill could be clamping itself to the walls or driving forward. The function level could contain an inverse kinematic model, to calculate what should be the next desired configuration. The torques for this configuration will be generated by the lower levels.

This framework has been implemented from the hardware level up to the service level, this implementation will be elaborated on in Section 2-3. At the skill level only a simple clamping action was implemented, in order to ensure that the software performance could be evaluated [Hoekstra, 2018].

**Table 2-1:** Separation of levels (vertical) and separation of concerns (horizontal) as formulated by RobMoSys. An example of each level is given for a robot that serves coffee. Adapted from [Tucci and Schlegel, 2017].

| Levels vs. Concerns | Computation | Communication | Coordination | Configuration | Example |
|---|---|---|---|---|---|
| **Mission** | | | | | serve as butler |
| **Task** | | | does | does | deliver coffee |
| **Skill** | | | | translate into parameters | grasp object with constraint |
| **Service** | | structures | | | move manipulator |
| **Function** | does | | | | IK solver |
| **Execution** | provides resources | provides resources | provides access to scheduler | | activity |
| **Operating system/ Middleware** | realises | realises | | receives | thread, socket |
| **Hardware** | does | does | receives | receives | actuator, sensor |

### 2-2-3 Motion primitives

To structure the skill level, the concept of motion primitives will be used. Dertien [2014] defines motion primitives as: *"the smallest meaningful action that can be performed by the PIRATE robot"*. He states that the basic PIRATE actions can be split into the following motion primitives: *clamp* with force $\tau$, *drive* with rotational speed $\omega$, *bend* with angle $\gamma$ and *rotate* with angle $\phi$.

The reason to use motion primitives for high-level control is that it reduces the number of actions that the robot has to choose from, and therefore reduces the complexity of the controller. For example, when reinforcement learning is applied on a robot, having a few actions to choose from, instead of multiple continuous action spaces, drastically reduces the training time. The higher control level may also make it easier for an operator to control the robot.

### 2-2-4 Wheel slip $\lambda$

As mentioned in Section 1-2, the robot needs to be able to determine the distance to a bend to be able to move through it autonomously. Odometry, determining displacement based on wheel rotation, can be used by the in-pipe robot to determine its location in the networks

when it cannot rely on other sensors. Due to the possibly contaminated pipe walls the robot may suffer from slipping, which makes the odometry unreliable. If slip can be detected or even compensated for, the odometry becomes much more robust. This also helps the robot to move through slippery environments.

An overview of the velocities and forces for a wheel is shown in Figure 2-2.

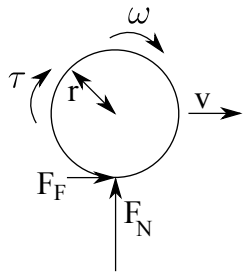



**Figure 2-2:** Overview of the forces and velocities for a wheel.

The slip ratio $\lambda$ is defined as the normalisation of the angular velocity to the linear velocity. The sign of $\lambda$ varies between different sources, in this thesis the definition as described in Equation 2-1 is used [Hansen et al., 2005].

$$\lambda = \frac{v - \omega r}{v} = 1 - \frac{\omega r}{v} \tag{2-1}$$

Where $\lambda$ indicates the following situations:

| | |
|---|---|
| $\lambda = 0$ | Perfect transmission from angular to linear velocity. |
| $\lambda = 1$ | The wheel is locked ($\omega = 0$), while there may still be linear velocity. |
| $0 < \lambda < 1$ | Wheel skid, e.g. when you brake at high speed ($0 < r\omega < v$). |
| $1 < \lambda$ | The angular velocity and linear velocity are in opposite directions. |
| $\lambda < 0$ | Wheel spin. Only partial transmission of angular to linear velocity ($v < r\omega$). |
| $\lambda =$ undefined | No linear velocity (which causes division by zero), while there may still be angular velocity. |

### 2-2-5 Friction coefficient $\mu$

The slip ratio is dependent on the friction coefficient $\mu$. For example, for a car driving on a snowy road (low $\mu$) the chances that wheel spin or wheel skid occur are higher than for a car driving on a dry road (high $\mu$) (Fig. 2-3b). The friction coefficient relates the normal force to the friction force.

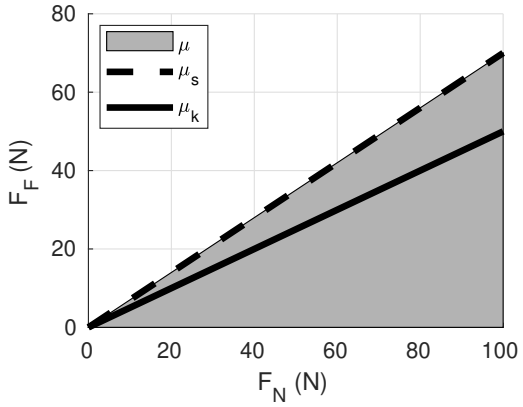
For static friction between two surfaces without relative speed the normal and friction force are related by the static friction coefficient $\mu_s$ (Eq. 2-2).

$$F_{\mathrm{F}} \leq \mu_{\mathrm{s}} F_{\mathrm{N}} \tag{2-2}$$

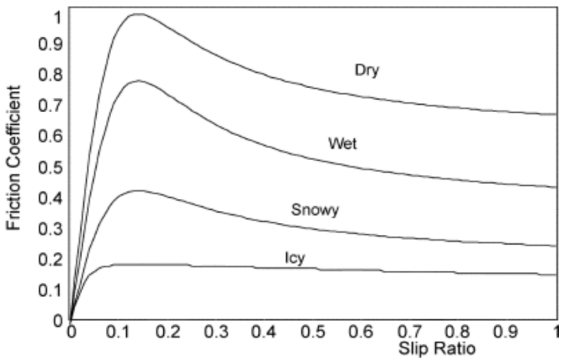
Whenever the friction force becomes higher that the threshold value of $\mu_{\mathrm{s}} F_{\mathrm{N}}$, the surfaces start moving with respect to each other. The friction and normal force are then related by the kinetic friction coefficient $\mu_{\mathrm{k}}$ (Eq. 2-3).

$$F_{\mathrm{F}} = \mu_{\mathrm{k}} F_{\mathrm{N}} \tag{2-3}$$

An example is shown in Figure 2-3a. Note that when $\lambda = 0$, the wheel and pipe have no relative speed and therefore static friction occurs. Even when the wheel is rolling, at each time instant the relative speed at the contact point is zero. As soon as $\lambda \neq 0$, kinetic friction occurs. The curve describing the relation between $\lambda$ and $\mu$ can be empirically estimated, as shown in Fig. 2-3b. Note that in literature sometimes the $\lambda$-$F_{\mathrm{F}}$ curve instead of the $\lambda$-$\mu_{\mathrm{k}}$ curve is identified. Estimating the $\lambda$-$F_{\mathrm{F}}$ curve is only useful when the normal force $F_{\mathrm{N}}$ is constant. This is not necessarily the case for the PIRATE since the normal force is not only compensating the gravitational force, but also the clamping force, which may vary.




**(a)** Relation between the normal force and friction force. When there is no slip, the force can be anywhere in the grey area (Eq. 2-2). When there is slip, the forces must be on the black line (Eq. 2-3). To generate a friction force of 35 N, a normal force of at least 50 N is needed for $\mu_{\mathrm{s}} = 0.7$ without slip and 70 N for $\mu_{\mathrm{k}} = 0.5$ with slip.




**(b)** Relation between the slip ratio and the friction coefficient for various road conditions. *Copyright © 2010, IEEE.* [Junhui and Jianqiang, 2010]

**Figure 2-3:** Examples to illustrate the relation between slip and friction.

The relation between the wheel slip ratio and the friction force can be identified using a parametric idenfitication and non-parametric identification. The Magic formula (Section 2-2-6), which is a well known model within the vehicle dynamics field, will be used as parametric model. Gaussian processes (Section 2-2-7) will be used as non-parametric identification method, since it also predicts the certainty of the output.

### 2-2-6 Magic formula

A widely used parametric model for wheel slip is the empirical Magic formula (Eq. 2-4) [Pacejka and Besselink, 1997]. The relation between the slip ratio $\lambda$ and the friction force $F_{\mathrm{F}}$ can be described as in Equation 2-4. An example of a resulting curve is shown in Figure 2-4. The parameters vary for different normal forces.

$$y = D \sin\left(C \arctan\left(Bx - E\left(Bx - \arctan\left(Bx\right)\right)\right)\right) \tag{2-4}$$

$$Y(X) = y(x) + S_{\mathrm{V}} \tag{2-5}$$

$$x = X + S_{\mathrm{H}} \tag{2-6}$$

where

| | |
|---|---|
| $X$ : input variable $-\lambda$ | $B$ : stiffness factor |
| $Y$ : output variable $F_{\mathrm{F}}$ | $C$ : shape factor |
| $S_{\mathrm{H}}$ : horizontal shift | $D$ : peak value |
| $S_{\mathrm{V}}$ : vertical shift | $E$ : curvature factor |

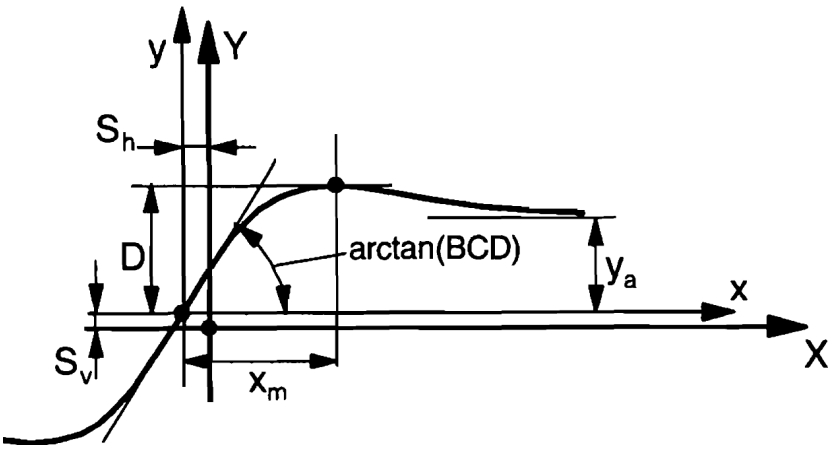



**Figure 2-4:** The Magic formula curve. $X$ is the negative wheel slip ratio $-\lambda$, $Y$ is the friction force $F_{\mathrm{F}}$. *Rights managed by Taylor & Francis.* Adapted from [Pacejka and Besselink, 1997].

### 2-2-7 Gaussian processes

Non-parametric identification takes into account uncertainties in the model structure, instead of only uncertainty in the model parameters. Gaussian processes model uncertainties based on the training data using a probabilistic approach. This way not only the output can be predicted, but also its uncertainty. A Gaussian process describes a set of random variables, which have a joint Gaussian distribution [Rasmussen and Williams, 2006]. Therefore a Gaussian process can be described by a mean function $m(\mathbf{x})$ (Eq. 2-7) and covariance function $k(\mathbf{x}, \mathbf{x}')$ (Eq. 2-8). This can also be written down as shown in Equation 2-9.

$$m(\mathbf{x}) = \mathbb{E}\left(f(\mathbf{x})\right) \tag{2-7}$$
$$k(\mathbf{x}, \mathbf{x}') = \mathbb{E}\left(\left(f(\mathbf{x}) - m(\mathbf{x})\right)\left(f(\mathbf{x}') - m(\mathbf{x}')\right)\right) \tag{2-8}$$
$$f(\mathbf{x}) \sim \mathcal{GP}\left(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')\right) \tag{2-9}$$

For example, a certain function can be described by a linear Bayesian regression model. In this model the output is described by sum of basis functions $\phi_i(x)$ of the inputs $\mathbf{x}$ times weights $w_i$ (Eq. 2-10), where $\mathbf{w}$ has a Gaussian distribution (Eq. 2-11). A linear combination of Gaussian distributions results in a Gaussian process (Eq. 2-12 & 2-13).

$$f(\mathbf{x}) = \sum_{i}^{N} \phi_i(x) w_i = \phi^T(\mathbf{x})\mathbf{w} \tag{2-10}$$
$$\mathbf{w} \sim \mathcal{N}\left(\mathbf{0}, \Sigma_p\right) \tag{2-11}$$
$$\mathbb{E}\left(f(\mathbf{x})\right) = \phi(\mathbf{x})^T \mathbb{E}\left(\mathbf{w}\right) = 0 \tag{2-12}$$
$$\mathbb{E}\left(f(\mathbf{x}) f(\mathbf{x}')\right) = \phi(\mathbf{x})^T \mathbb{E}\left(\mathbf{w}\mathbf{w}^T\right) \phi(\mathbf{x}') = \phi(\mathbf{x})^T \Sigma_p \phi(\mathbf{x}') \tag{2-13}$$

Usually the mean of the function is assumed to be zero, so no hyperparameters are needed for the mean. Various kernels can be chosen for the covariance function, but the standard choice is the squared exponentional covariance (Eq. 2-14), with hyperparameters signal variance $\sigma_f^2$ and length scale $l$. The output of the test data $\mathbf{f}_*$ is assumed to be a Gaussian distribution with zero mean (Eq. 2-15). The same is assumed for the combination of training and test data: since the relation between the training output and training input is the same as the relation between the test output and test input, the combined set of training output and test output should be a Gaussian distribution (Eq. 2-16).

$$cov\left(f(x_p), f(x_q)\right) = k(x_p, x_q) = \sigma_{\mathrm{f}}^2 e^{-\frac{1}{2}|x_p - x_q|^T (l^2 I)^{-1} |x_p - x_q|} \tag{2-14}$$
$$\mathbf{f}_* \sim \mathcal{N}\left(\mathbf{0}, K\left(X_*, X_*\right)\right) \tag{2-15}$$
$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K\left(X, X\right) & K\left(X, X_*\right) \\ K\left(X_*, X\right) & K\left(X_*, X_*\right) \end{bmatrix}\right) \tag{2-16}$$

When white noise on the sensory input is assumed (Eq. 2-17), the covariance function gets an additional variance term $\sigma_{\mathrm{n}}^2$ on the diagonal (Eq. 2-18 & 2-20). $\sigma_{\mathrm{n}}^2$ represents the variance

of the noise, which is also a hyperparameters of the Gaussian process.

$$y = f(x) + \epsilon \tag{2-17}$$

$$cov\left(y_p, y_q\right) = k(x_p, x_q) + \sigma_{\mathrm{n}}^2 \delta_{pq} \tag{2-18}$$

$$\delta_{pq} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases} \tag{2-19}$$

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K\left(X, X\right) + \sigma_{\mathrm{n}}^2 I & K\left(X, X_*\right) \\ K\left(X_*, X\right) & K\left(X_*, X_*\right) \end{bmatrix}\right) \tag{2-20}$$

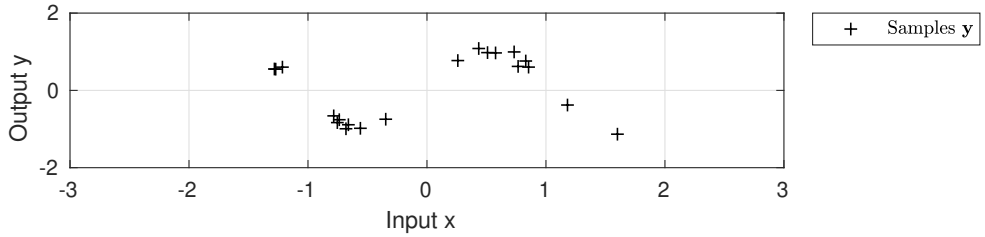
For test inputs $\mathbf{x}_*$, the predicted output $\bar{f}(\mathbf{x}_*)$ is linearly related to test outputs $\mathbf{y}$ by means of the covariance matrices. This results in a mean prediction (Eq. 2-21) and a variance prediction (Eq. 2-22).

$$\bar{f}(\mathbf{x}_*) = K(X, X_*)^T (K(X, X) + \sigma_{\mathrm{n}}^2 I)\mathbf{y} \tag{2-21}$$

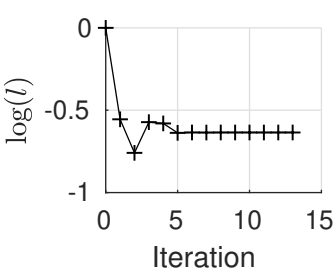
$$\mathbb{V}(f(\mathbf{x}_*)) = K(X_*, X_*) - K(X, X_*)^T (K(X, X) - \sigma_{\mathrm{n}}^2 I)^{-1} K(X, X_*) \tag{2-22}$$

Through minimisation of the standard deviation of the Gaussian likelihood, the hyperparameters $\sigma_{\mathrm{f}}$, $l$ and $\sigma_{\mathrm{n}}$ can be found. During the optimisation of the hyperparameters $\log(\sigma_{\mathrm{f}})$, $\log(l)$ and $\log(\sigma_{\mathrm{n}})$ are be used, instead of $\sigma_{\mathrm{f}}$, $l$ and $\sigma_{\mathrm{n}}$, to reduce the influence of scaling.

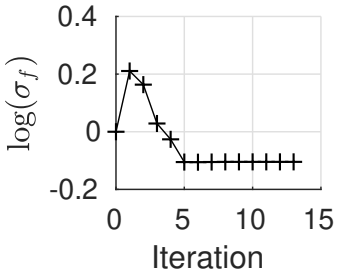
An example of the steps in a Gaussian process identification is shown in Figure 2-5. The training output $\mathbf{y}$ for input $\mathbf{x}$ is shown in Fig. 2-5a. The optimisation for the hyperparameters is shown in Fig. 2-5b:2-5d. The predicted output $\bar{f}(\mathbf{x}_*)$ for input $\mathbf{x}_*$ in each optimisation iteration is shown in Fig. 2-5e. The resulting predicted output and confidence interval are shown in Fig. 2-5f.



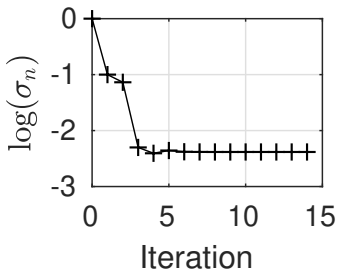

**(a)** The initial noisy dataset of a sine function.



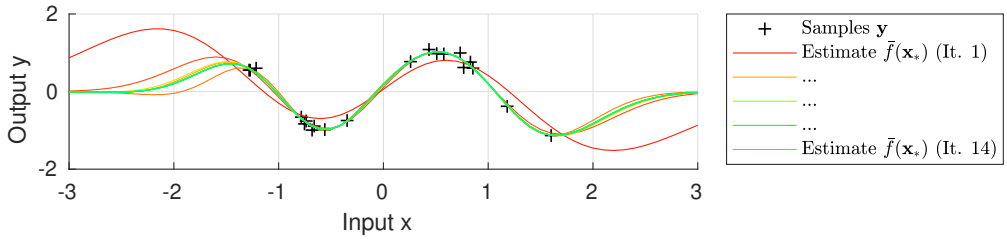

**(b)** Length scale of the covariance.



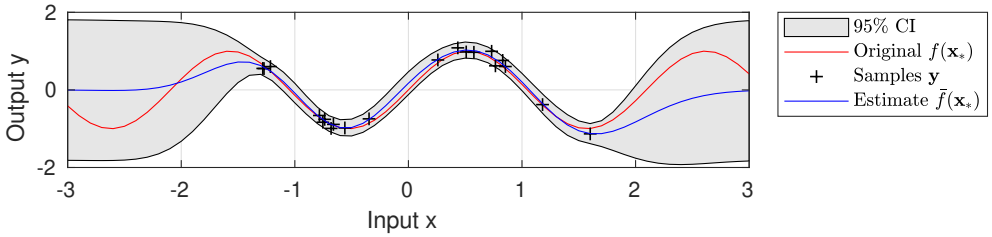

**(c)** Standard deviation of the covariance.




**(d)** Standard deviation of the Gaussian likelihood.




**(e)** Estimated mean functions. The reddest line represents the mean of the Gaussian process for the initial non-optimised hyperparameters, the greenest line represents the Gaussian process for the optimised hyperparameters.




**(f)** The resulting Gaussian process with estimated mean and 95% confidence interval. Close to the training data the confidence interval is narrow, while it is wide far from the training data, since the uncertainty is high in that area.

**Figure 2-5:** Example of the steps to train a Gaussian process. The training of this Gaussian process took fourteen iterations to converge. Based on the example in the GPML toolbox documentation [Rasmussen and Nickisch, 2010].

## 2-3 Robot software architecture

This section provides a short summary of the work by Hoekstra [2018] and describes the software structure prior to this thesis. An overview is shown in Figure 2-6. In Section 4-2 and 4-3 each node will be discussed, along with the changes made for the software structure in this thesis.

The SerialCommunication node runs on the PIRATEbay and translates commands between the ROS structure and RS485 communication structure for the PICO boards. The PAB (Partially autonomous behaviour), Movement and PirateCommunication nodes run on the laptop. The PirateCommunication and SerialCommunication nodes together form the communication bridge over the serial port. The PirateCommunication node does not add any functionality apart from sending and receiving data over the serial port.

The Movement node is divided over two levels. The Execution level contains functionality for sensors, motors and modules separately, like building blocks. Due to the identifiers and polymorphic approach it is easy to add e.g. an extra sensor or a new type of module. The Function level contains the functionality for the robot as a whole and contains algorithms that can access data from specific sensors and motors.

The Service level does not have its own node but is designed as an interface between the low-level and high-level nodes. This interface is implemented as a ROS service between the PAB and the Movement node.

The PAB node at the Skill level takes care of high-level control. In this structure only a simple clamp command is implemented, to be able to evaluate the software structure as a whole.

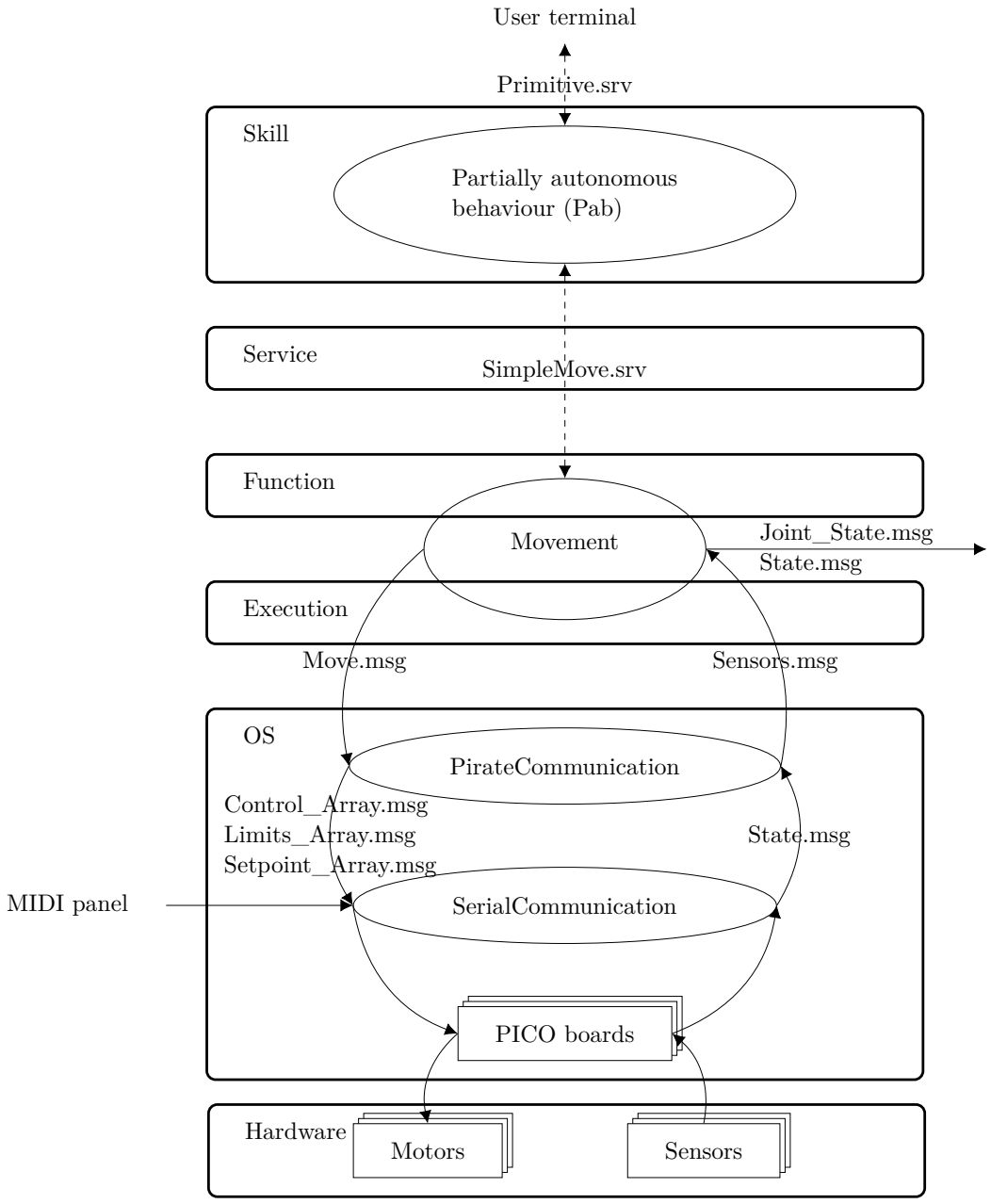



**Figure 2-6:** Overview of the software structure prior to this thesis. The rectangles represent the hardware, the rounded rectangles represent the RobMoSys levels, the ellipses represent the ROS nodes, the arrows represent (ROS) messages and the dashed arrows represent ROS services. None of the nodes subscribe to Joint_State.msg and State.msg, but, as holds for all messages, their contents can be requested in the terminal.

# Chapter 3

# Design

In this chapter, the requirements for the design are stated. Then the design for the high-level controller is explained. Finally the proposed strategy of moving through a bend, based on this high-level controller, is elaborated on.

## 3-1 Requirements

- Low-level control
  - Each wheel should be able to be actuated with an individual setpoint for open-loop voltage control (PWM, with duty cycle $D$) .
  - Both open-loop voltage control and closed-loop position control should be implemented for the joint angles.
  - It should be possible to control multiple motors simultaneously. E.g. it should be possible to actuate both bending joints of the front part simultaneously in order to clamp the front part.
  - The low-level controller has to ensure that all sent commands are executed by the PIRATE.
- High-level control
  - There should be as few choices for actions as possible, instead of a continuous range of actions for each actuator. This way it is easier for an operator to operate the PIRATE. Train a future learning agent also becomes easier and faster.
  - The state of the PIRATE should be described in a structured way, such that the operator can immediately imagine what the PIRATE is doing without being able to see it. This will be done by defining states and transitions in multiple finite state machines (FSM). The motion primitives will be used as base for these state machines.

  - The PIRATE should be able to execute various lists of successive commands.
  - The PIRATE should be able to use triggers from external sensors to start the sequences.
  - The PIRATE should be able to track its travelled distance.

- Simulation
  - Since the robot is not always available and as there is no simulation model yet, a simulation model needs to be built.
  - The model should have similar behaviour to the real robot when provided with a high-level command, so the model can be used to get familiar with the kinematics and test commands before testing them on the real robot.
  - The model should be fast enough: not slower than 0.1x real time (10 real seconds to simulate 1 second), measured over the entire simulation. This way it can be usefull for training a learning agent in later PIRATE projects.
  - A traction controller should be implemented to actuate the wheels and bending joints in such a way that the torques can be as low as possible while preventing the robot from sliding out of a vertical pipe. This has to be done in simulation since the current software architecture does not allow for velocity or torque control. Also, the real robot slides out of a vertical pipe even when the PWM setpoints for the wheels are set to maximum.

## 3-2 Finite state machines

The high-level controller consists of five finite state machines (Fig. 3-1):

| | |
|---|---|
| `BendingState` | Describes the clamping and bending behaviour based on the bending joints. |
| `FrontWheelsState` | Describes the behaviour of the front wheels. |
| `RearWheelsState` | Describes the behaviour of the the rear wheels. |
| `RotationState` | Describes the relative orientation of the front and rear part. |
| `LightingState` | Indicates if lights are on or off. |

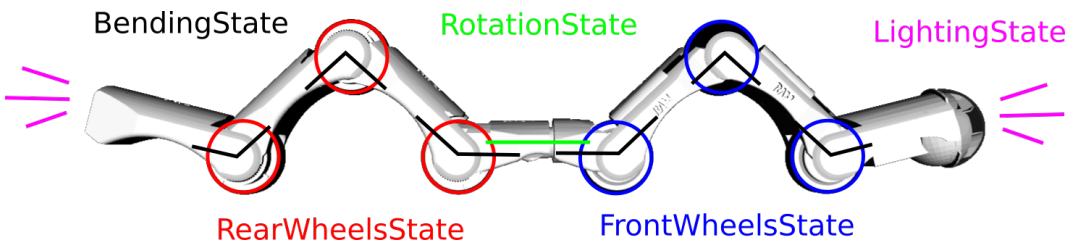


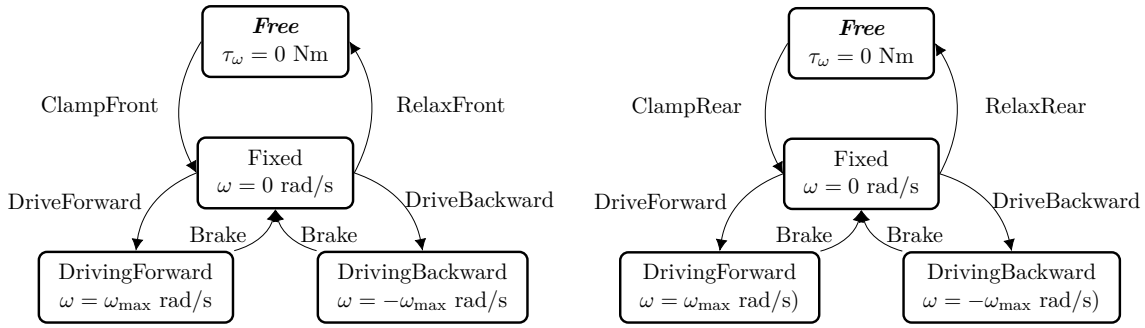

**Figure 3-1:** Overview of which actuators are affected by which state machine.

These five state machines are based on the motion primitives as mentioned in Section 2-2-3: `BendingState` refers to *clamp* and *bend*, `FrontWheelsState` and `RearWheelsState` refer to *drive* and `RotationState` refers to *rotate*. The fifth state machine is a switch for the LEDs, this state machine is not related to motion. The reason *clamp* and *bend* are in the same state machine is because they refer to the same joints. The reason `FrontWheelsState` and `RearWheelsState` are split is because they operate independent of eachother. For example, when only the rear is clamped and the PIRATE is driving, only the rear wheels should be actuated.

The `FrontWheelsState` and `RearWheelsState` state machines are shown in Figures 3-2a and 3-2b. For both the front wheels (Wheel 1, 2 and 3) and the rear wheels (Wheels 4, 5 and 6), the `Free` state means zero torque mode, while `Fixed` means that the position is locked at zero angular velocity. The `DrivingForward` and `DrivingBackward` state set a fixed PWM setpoint for the wheels. Since Wheel 2 and Wheel 5 are touching the opposite part of the pipe with respect to Wheel 1 and 3 or Wheel 4 and 6, they are actuated in the opposite direction to generate linear velocity in the same direction (Fig. 1-1).

The `BendingState` state machine is shown in Figure 3-3. The initial state for the `BendingState` state machine is `Unclamped`. Since the `Unclamped` state should never be reached when operating the PIRATE (the PIRATE will fall out of the pipe), it is used as a reset state: it brings all bend angles back to zero. When clamping e.g. the front, a PWM value is set for bending joints $\gamma_1$ and $\gamma_2$ in opposite directions, while the rear part remains unactuated.

For a state like `RearClampedFrontBending`, the rear part is clamped. $\gamma_1$ and $\gamma_2$ each have a negative position (angle) setpoint. This way the negative angle ensures that the front part




**(a)** FrontWheelsState state machine, with `Free` as initial state.

**(b)** RearWheelsState state machine, with `Free` as initial state

**Figure 3-2:** State machines for the wheels.

will slide along the bend instead of getting stuck when moving forward through a mitre bend. As soon as the desired angles are reached a negative PWM setpoint is put on $\gamma_2$ and $\gamma_3$, to prevent the robot from getting stuck while sliding around the bend.

The `RotationState` state machine is shown in Figure 3-4. For the rotation state, it is assumed that the initial state is `Unaligned`, which means that the front and rear part are not in the same plane. In the `AlignStraight`, `AlignedRight` and `AlignedLeft` state the rotation angle is fixed at 0 rad, $0.5\pi$ and $-0.5\pi$ rad respectively. This way the robot is able to rotate stepwise around its axis. The rotation angle itself instead of a PWM value is sent to the PICO, since it has an internal controller for the angle. In a future PIRATE project states like `AlignHorizontal` and `AlignVertical` may be added, based on the IMU sensor.

On both the front and the rear module there are LED lights, these are either both on or both off. The `LightingState` state machine is shown in Figure 3-5.

As said before, the aim of the state machine commands is to use as few commands as possible to actuate the robot. As such the following commands are chosen as being the bare minimum required:

| | |
|---|---|
| `ClampFront`, `ClampRear`, `RelaxFront` and `RelaxRear` | for clamping (Fig. 3-2a, 3-2b and 3-3). |
| `BendInto` and `BendBack` | for bending (Fig. 3-3). |
| `DriveForward`, `DriveBackward` and `Brake` | for driving (Fig. 3-2a:3-2b). |
| `AlignStraight`, `AlignLeft` and `AlignRight` | for rotation, all of which are only possible when the bending state is not `DoubleClamped` (Fig. 3-4). |
| `ToggleLighting` | for lighting (Fig. 3-5). |
| `Reset` | sets all state machines to their original states: `Free`, `Free`, `Unclamped`, `Unaligned` and `Off` (which are indicated in ***bold italics*** in Fig. 3-2a:3-5). |

Note that each of these commands may influence multiple state machines.

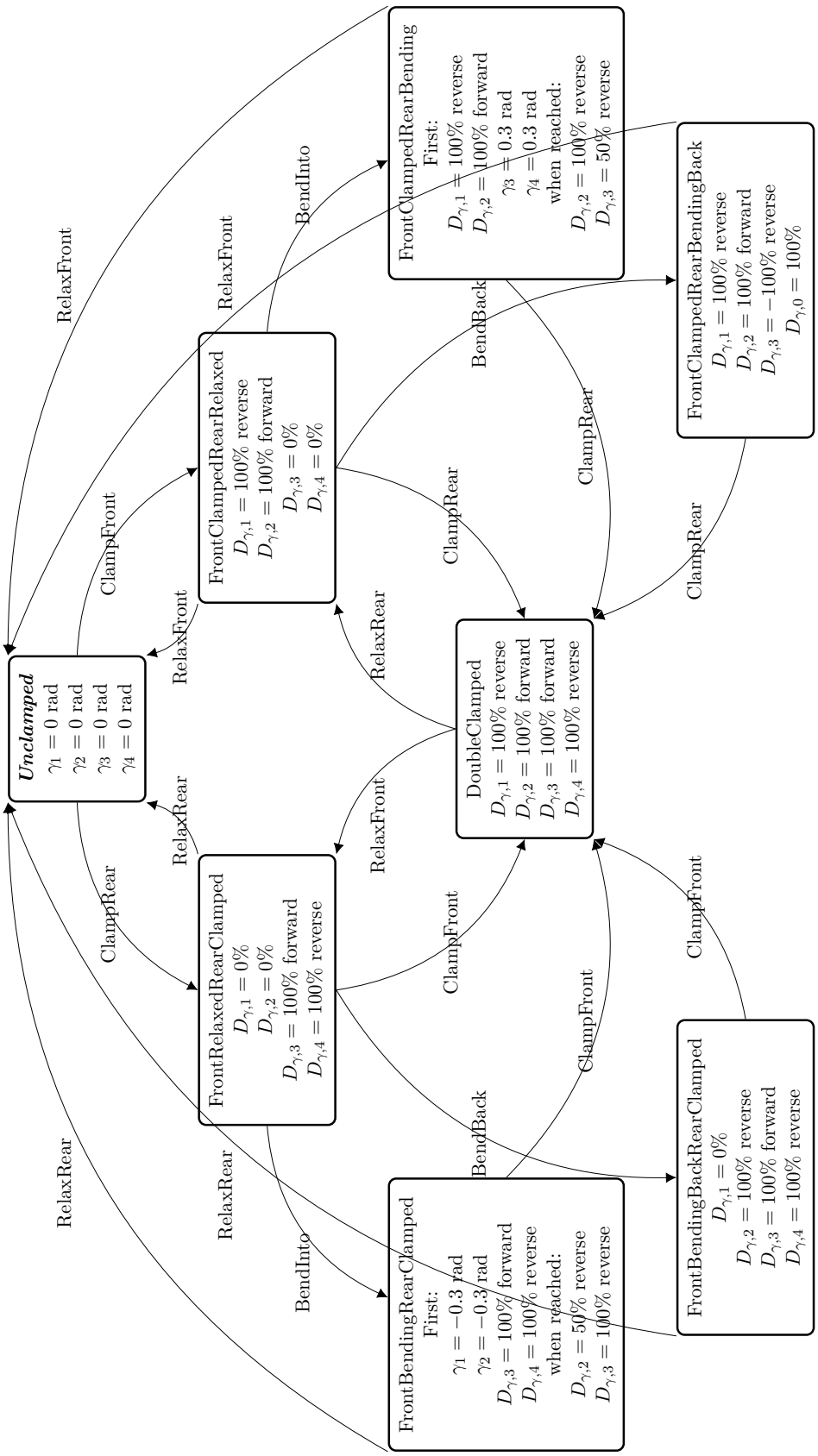



**Figure 3-3:** BendingState state machine, with `Unclamped` as the initial state.

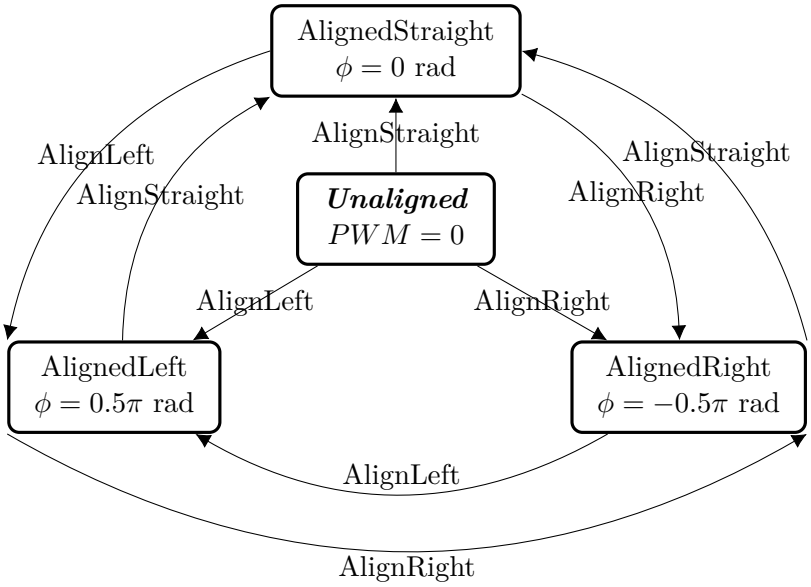



**Figure 3-4:** RotationState state machine.

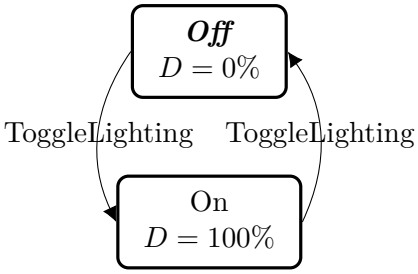



**Figure 3-5:** LightingState state machine.

## 3-3 Sequences of primitives

To reach a higher level of autonomy than the states and actions from the previous section, sequences of actions need to be generated. These sequences should be triggered by external sensors, like a camera module or a proximity sensor. Sequences can then be used for tasks such as *Enter the pipe* or *Take a left turn*. When placed into a pipe the PIRATE should be placed in the middle of a horizontal pipe segment. Since the whole body of the PIRATE is in the pipe, it does not matter whether the front or rear part is clamped first.

Entering the pipe in 2D could be achieved by a sequence like:

1. `ClampFront`
2. `ClampRear`
3. `ToggleLight`
4. `DriveForward`

In a 3D environment you want to drive the PIRATE as much as possible in the horizontal plane, since the bottom of the pipe may contain dirt. An entering sequence could for example be:

1. `ClampRear`
2. `AlignRight`
3. `ClampFront`
4. `ToggleLight`
5. `UnclampRear`
6. `AlignOther`
7. `ClampRear`
8. `DriveForward`

As soon as the PIRATE encounters a mitre bend, it should prepare itself for entering the bend with the following sequence: `Brake` → `UnclampFront` → `BendInto` → `DriveForward`.

As soon as Wheel 3 is around the bend, the PIRATE should move its clamping with: `Brake` → `ClampFront` → `UnclampRear` → `BendBack` → `ClampRear` → `DriveForward`.

## 3-4 The model

### 3-4-1 Simulation model

The inputs of the simulation model are the torques for the bending joints ($\boldsymbol{\tau}_\gamma$), for the wheels ($\boldsymbol{\tau}_\theta$) and for the rotational joint ($\boldsymbol{\tau}_\phi$). The outputs of the sensors are the angles of the bending joints ($\boldsymbol{\gamma}$), the angles of the wheels ($\boldsymbol{\theta}$), the angle of the rotational joint ($\boldsymbol{\phi}$) and the pitch and roll angles ($\boldsymbol{\psi}$). The yaw, pitch and roll definitions from aviation terminology are used, with the origin at the IMU sensor on the front PICO board of the rotational module, as shown in Fig. 3-6. The yaw angle is currently not used, since it does not affect the forces on the PIRATE.

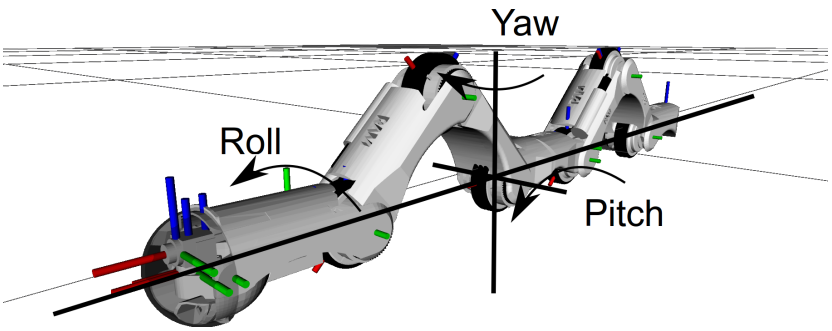



**Figure 3-6:** Definition of the yaw, pitch and roll axes, based on the sensor in the front PICO board of the rotational module.

In the simulation model the coordinates ($x$, $y$, $z$) of wheel 1 with respect to its original position, the friction force ($\mathbf{F}_\mathrm{F}$) on each wheel and normal force ($\mathbf{F}_\mathrm{N}$) on each wheel can be directly measured from the contact model. For the real robot the coordinates need to be estimated using an external sensor, like a camera or proximity sensor. The friction and normal force need to be calculated based on the angle and torque values. The calculation of the friction and normal forces in the simulation can be validated with the friction and normal force for the contact model.

An overview of the inputs and outputs is shown in Figure 3-7. All derivatives are directly available in the simulation model, in real life they need to be estimated.

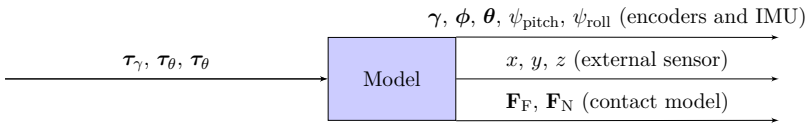



**Figure 3-7:** Inputs and outputs of the simulation model. All derivatives of the outputs are calculated by the simulation model and are available.

The linear component of the angular velocity is calculated by multiplying the angular velocity $\dot{\theta}_i$ with wheel radius $r$ and direction compensation $d_i$ ($d_i = 1$ for Wheels 1, 3 and 5, $d_i = -1$ for Wheels 2, 4 and 6). From this the wheel slip ratio $\lambda$, discussed in Section 2-2-4, for each wheel is calculated as in Equation 3-1.

$$\lambda_i = \frac{\dot{x} - r d_i \dot{\theta}_i}{\dot{x}} \tag{3-1}$$

The clamping force $F_{\text{clamp}}$ can be calculated from the joint angles and joint torques, as per Equations 3-2 and 3-3, with definitions shown in Figure 3-8.

$$\begin{aligned} h_{\text{front}} &= |l \sin(0.5\gamma_2)| \\ h_{\text{rear}} &= |l \sin(0.5\gamma_5)| \end{aligned} \tag{3-2}$$

$$\mathbf{F}_{\text{clamp}} = \begin{bmatrix} \frac{\tau_{\gamma,2}}{\sqrt{l^2 - h_{\text{front}}^2}} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \\ \frac{\tau_{\gamma,5}}{\sqrt{l^2 - h_{\text{rear}}^2}} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \end{bmatrix} \tag{3-3}$$

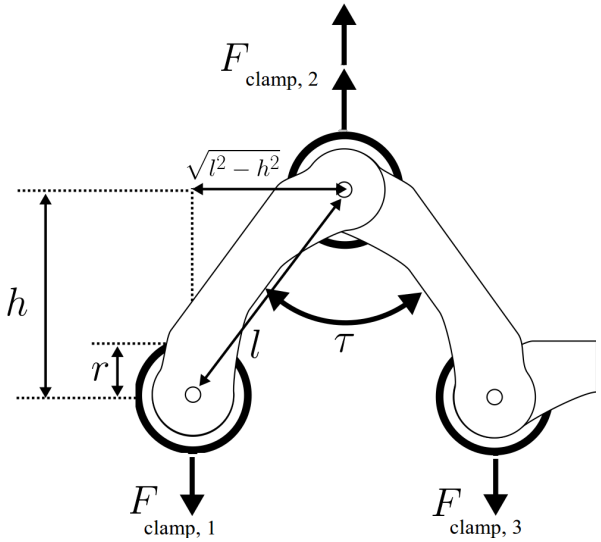



**Figure 3-8:** Derivation of the clamping force $\mathbf{F}_{\text{clamp}}$ in the front part, based on the joint angle $\gamma_2$, joint torque $\tau_{\gamma,2}$ and segment length $l$. The pipe diameter is $h + 2r$. *Permission granted by the author.* Adapted from [Dertien, 2014].

### 3-4-2 Traction controller

For the design of the traction controller the following simplifications are made (Fig. 3-9):

- When both the front and rear part are clamped, the PIRATE can be modelled as a rigid body with six wheels.
- The rotational joint is fixed at 0 rad.
- The model is in 2D.
- The PIRATE is constrained in the $y$ direction and orientation.

Using these simplifications, the 13 DoF system in 3D becomes an 8 DoF system in 2D. The simplified equations of motion are defined as in Equations 3-4:3-6, where $m$ is the total mass of the PIRATE, $I$ is the inertia of a PIRATE wheel and $\alpha$ is the tilt angle of the pipe. The $x$ and $y$ axes are defined to be parallel and perpendicular respectively to the pipe (Fig. 3-9), which further simplifies the equations.

$$m\ddot{x} = \sum F_x = -mg\sin\alpha + \sum_i F_{\mathrm{F},i} \tag{3-4}$$

$$m\ddot{y} = \sum F_y = -mg\cos\alpha + \sum_i F_{\mathrm{N},i} + \sum_i F_{\mathrm{clamp},i} \tag{3-5}$$

$$I\ddot{\theta}_i = \sum \tau = \tau_{\theta,i} - rd_iF_{\mathrm{F},i} \tag{3-6}$$
$$i = 1, .., 6$$

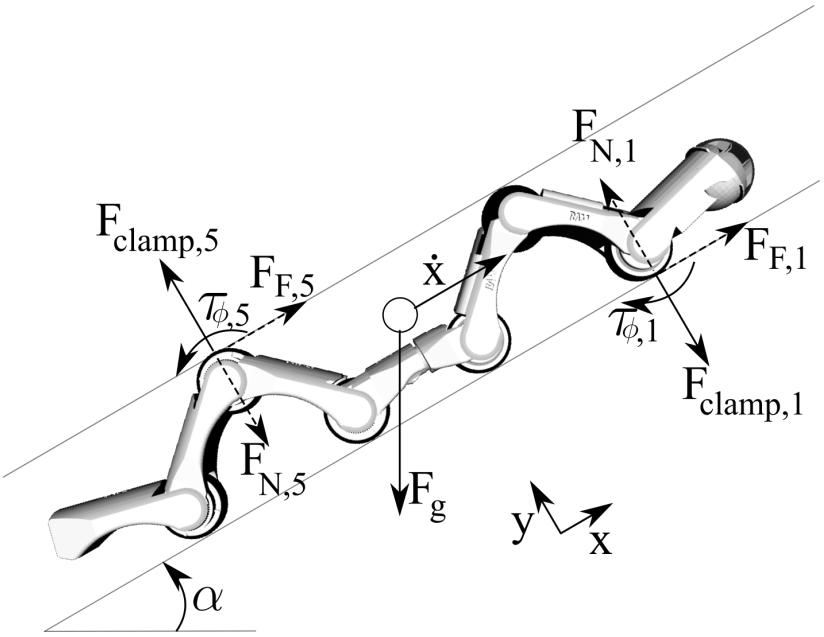



**Figure 3-9:** Free body diagram with forces and torque shown for Wheel 1 and 5.

As mentioned in Section 2-2-4, when $\lambda \neq 0$, the angular acceleration is not linearly related to the linear acceleration ($\ddot{x} \neq r\ddot{\theta}$). Combining Equations 3-4 and 3-6 the desired wheel torque can be approximated.

$$\sum_i F_{\mathrm{F},i} = m\left(\ddot{x} + g\sin\alpha\right) \tag{3-7}$$

$$\frac{1}{r}\sum_i d_i\left(\tau_{\theta,i} - I\ddot{\theta}_i\right) = m\left(\ddot{x} + g\sin\alpha\right) \tag{3-8}$$

$$\sum_i d_i\tau_{\theta,i} = I\sum_i d_i\ddot{\theta}_i + rm\left(\ddot{x} + g\sin\alpha\right) \tag{3-9}$$

Since the accelerations in the behaviour of the PIRATE are relatively low, they are neglected in the controller for simplicity. The gravitational force is assumed to be distributed equally

over all wheels. Therefore the desired torque per wheel to compensate gravity can be derived as shown in Equations 3-10:3-11.

$$\sum_i d_i \tau_{\theta,i} \approx rmg \sin\alpha \tag{3-10}$$

$$\tau_{\theta,i,\mathrm{d}} = d_i \frac{1}{6} rmg \sin\alpha \tag{3-11}$$

Another approach for the wheel torque controller is a simple velocity controller, as in Equation 3-12.

$$\tau_{\theta,i,\mathrm{d}} = d_i K_\mathrm{p} \left(\dot{x}_d - r\dot{\theta}_i\right) \tag{3-12}$$

The problem with only a velocity controller is that when the angular velocity and the linear velocity are equal, no torque is applied. When both velocities are zero, this would result in sliding, at which point a torque is again applied, thus resulting in jitter. Therefore a combination of both the gravity compensation and the velocity controller is proposed in Equation 3-13.

$$\tau_{\theta,i,\mathrm{d}} = \underbrace{d_i \frac{1}{6} rmg \sin\alpha}_{\text{gravity compensation}} + \underbrace{d_i K_\mathrm{p} \left(\dot{x}_d - r\dot{\theta}_i\right)}_{\text{velocity controller}} \tag{3-13}$$

To prevent the PIRATE from sliding out of the pipe the friction force should be able to at least compensate the forces resulting from gravity acting on the system, as shown in Equations 3-14 and 3-15.

$$\sum_i F_{\mathrm{F},i,\mathrm{d}} \geq m\left(\ddot{x} + g\sin\alpha\right) \tag{3-14}$$

$$F_{\mathrm{F},i,\mathrm{d}} \geq \frac{1}{6} m\left(\ddot{x} + g\sin\alpha\right) \tag{3-15}$$

To be sure that the desired friction force is high enough, a safety factor of 2 is chosen. The desired normal force is then calculated by means of the estimated kinetic friction coefficient $\hat{\mu}_{\mathrm{k},i}$, which should be estimated by means of an experiment with the real robot. From the desired normal force the desired clamping force can be calculated, as shown in Equations 3-16:3-19.

$$F_{\mathrm{F},i,\mathrm{d}} = \frac{1}{3} m\left(\ddot{x} + g\sin\alpha\right) \tag{3-16}$$

$$\hat{\mu}_{\mathrm{k},i} \leq \mu_{\mathrm{k},i} \tag{3-17}$$

$$\hat{\mu}_{\mathrm{k},i} = \frac{F_{\mathrm{F},i}}{F_{\mathrm{N},i}} \tag{3-18}$$

$$F_{\mathrm{N},i,\mathrm{d}} = \frac{F_{\mathrm{F},i,\mathrm{d}}}{\hat{\mu}_{\mathrm{k},i}} \tag{3-19}$$

Reworking the equation of motion given in Equation 3-5 results in the desired clamping force as in Equations 3-20:3-22.

$$\sum_i F_{\text{clamp},i} = m\left(\ddot{y} + g\cos\alpha\right) - \sum_i F_{\text{N},i} \tag{3-20}$$

$$F_{\text{clamp},i} \approx \frac{1}{6} m\left(\ddot{y} + g\cos\alpha\right) - F_{\text{N},i} \tag{3-21}$$

$$F_{\text{clamp},i,\text{d}} = \frac{1}{6} m\left(\ddot{y} + g\cos\alpha\right) - F_{\text{N},i,\text{d}} \tag{3-22}$$

Combining Equations 3-3, 3-19 and 3-22 to Equation 3-23, the desired torques for bending joints $\gamma_2$ (front part) and $\gamma_5$ (rear part) can be determined as shown in Equations 3-24 and 3-25 respectively.

$$\begin{aligned}
\tau_{\gamma,2,\text{d}} &= \frac{1}{2}\sqrt{l^2 - h_{\text{front}}^2} F_{\text{clamp},i,\text{d}} \\
&= \frac{1}{2}\sqrt{l^2 - h_{\text{front}}^2}\left(\frac{1}{6} m\left(\ddot{y} + g\cos\alpha\right) - F_{\text{N},i,\text{d}}\right) \\
&= \frac{1}{2}\sqrt{l^2 - h_{\text{front}}^2}\left(\frac{1}{6} m\left(\ddot{y} + g\cos\alpha\right) - \frac{1}{\hat{\mu}_{\text{k},i}}\left(\frac{1}{3} m\left(\ddot{x} + g\sin\alpha\right)\right)\right)
\end{aligned} \tag{3-23}$$

$$\tau_{\gamma,2,\text{d}} = \frac{1}{2}\sqrt{l^2 - h_{\text{front}}^2}\left(\frac{1}{6} mg\cos\alpha - \frac{1}{3\hat{\mu}_{\text{k},2}} mg\sin\alpha\right) \tag{3-24}$$

$$\tau_{\gamma,5,\text{d}} = \frac{1}{2}\sqrt{l^2 - h_{\text{rear}}^2}\left(\frac{1}{6} mg\cos\alpha - \frac{1}{3\hat{\mu}_{\text{k},5}} mg\sin\alpha\right) \tag{3-25}$$

# Chapter 4

# Robot software implementation

In this chapter the design of the simulation is briefly discussed. The adaptions made on the low-level controller are then discussed node by node. Finally, the ROS node implementation for the high-level controller is explained.

## 4-1 Simulation model

The simulation model was built in Simulink. The hardware was modelled using the Simscape Multibody package, a native Simulink package. The kinematics and visual appearance (Figure 4-1) were generated by importing an existing URDF file (kinematics) and existing STL files (meshes) of the PIRATE. As shown in Figure 4-1, the front and rear modules are left out. The front module is currently not available for the real robot and the rear module is not actuated. Omitting these two modules will reduce the complexity and therefore improve the simulation performance.

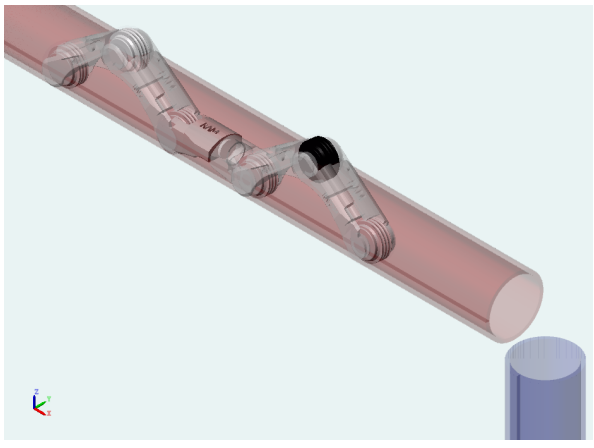


**Figure 4-1:** Simulation and visualisation of the PIRATE in two pipe segments.

All joints and wheels are connected by 1-DOF revolute joints, which use torque as input and angle as output. The first joint is connected to the world by a 6-DOF joint without actuation, making the robot a 17-DOF system. This extra joint is needed by the simulation to be able to initialise the position and orientation of the robot, and can also be used to directly measure the coordinates of the robot. The dimensions of the modules are known from the URDF file and were validated by measuring them. The masses of the modules and wheels were determined by weighting spare parts of the robot, see Table 4-1. The modules are assumed to have point masses at the center of their corresponding wheel, since the real mass matrices are unknown. The wheels are modelled as disks, with the inertia of a homogeneous disk with its total mass equivalent to the measured mass of a wheel. Figure 4-2 shows an overview of the Simscape model.

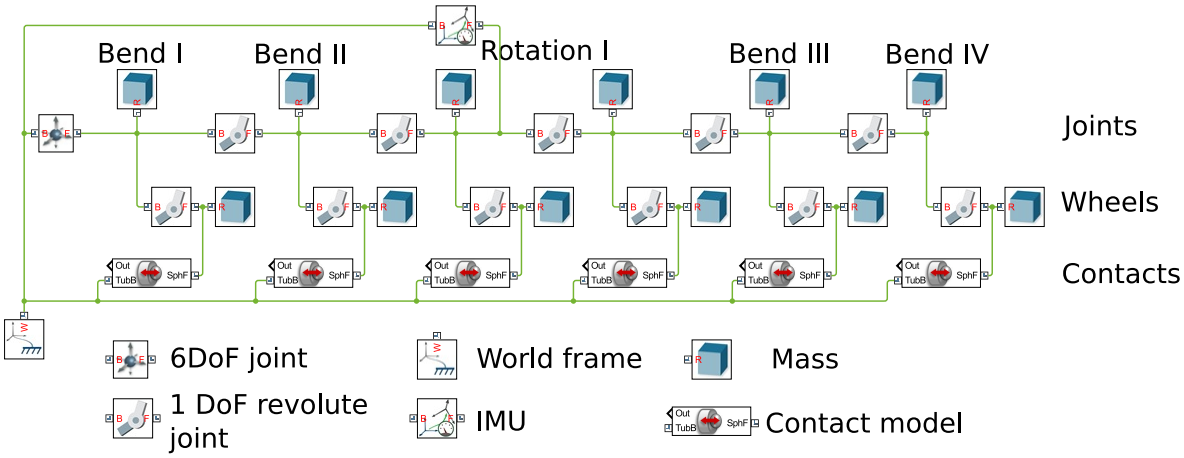



**Figure 4-2:** Overview of the Simscape model. The inputs and outputs and the transformation blocks are omitted for simplicity.

The contacts are modelled using the Simscape Multibody Contact Forces Library [Miller, 2017]. The contacts between the wheels of the PIRATE and the pipe walls are modelled with sphere-to-tube contacts, since this was the only contact model geometry that incorporated a tube element. Due to the width of the wheels (1.2 cm) the sphere-to-tube geometry should be appropriate. Each contact block combines a linear mass-spring-damper system with a continuous stick-slip friction law. The contact block first calculates the location and orientation between the sphere and the tube. Based on that it calculates the penetration of the pipe wall and the resulting damped spring force. Rigid surfaces are very hard to model in Simulink, so suitable thickness, stiffness and damping values have to be used to model the pipe wall. The stiffness and damping parameters were tuned in such a way that the robot could clamp itself without penetrating the pipe wall and without bouncing, see Table 4-1. Based on the relative orientation between the two bodies, the damped spring force is transformed to a normal force acting on both bodies. This in turn allows for calculation of the friction force, based the relative linear velocity and the normal force. The relation between the relative velocity and $\mu$ is as modelled in the Simscape Multibody Contact Forces Library is shown in Figure 4-3.

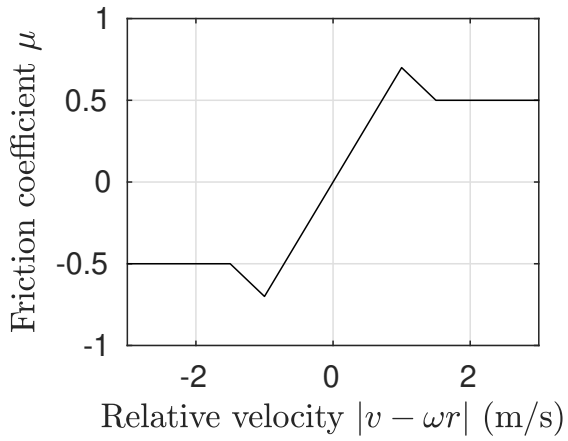



**Figure 4-3:** Relation between the relative velocity and friction coefficient as defined in the Simscape Multibody Contact Forces Library. The parameters are $\mu_\mathrm{s}$, $\mu_\mathrm{k}$ and $v_{th}$, which in this example are 0.7, 0.5 and 0.001 m/s.

Due to the used contact model the wheel radii are not constant, but depend on the penetration depth $d_p$ in the contact model. The required modifications for the calculation of the friction force result in Equations 4-1 and 4-2. For the controller the constant radius $r$ is used, since the penetration depth for the real robot will be less (approximately 2 mm) and the real robot cannot measure the penetration depth.

$$F_{\mathrm{F},i} = \frac{1}{r_i^*}\left(\tau_{\theta,i} - I\ddot{\theta}_i\right). \tag{4-1}$$

$$r_i^* = r - \frac{1}{2}d_{\mathrm{p},i} \tag{4-2}$$

Since the angle values can be read out directly from the revolute joint blocks, no sensors need to be modelled. White noise is added to the output angles (including the IMU), angular velocity, position and linear velocity.

The IMU is modelled as an orientation sensor based on quaternions, measuring the orientation of the front part of the rotational module with respect to the world frame (Figure 1-1).

Since communication between Simulink blocks is handled completely by Simulink, only lines need to be drawn, and no implementation of the OS level or Service level is needed. The Function/Execution level contains PD controllers for the wheels and joints, as will be discussed in Section 4-2. The Skill level contains a Finite State Machine and the selection of the setpoints, as will be discussed in Section 4-3.

**Table 4-1:** Simulation parameters.

| | **Bend** | **Rotation** | **Wheel** |
|---|---|---|---|
| **Length (m)** | 0.09 | 0.098 | - |
| **Diameter (m)** | - | - | 0.048 |
| **Mass (kg)** | 0.130 | 0.150 | 0.020 |
| **Inertia (kg $m^2$)** | - | - | $2.88 \cdot 10^{-6}$<br>$5.76 \cdot 10^{-6}$<br>$2.88 \cdot 10^{-6}$ |
| **Stiffness ($Nm^{-1}$)** | 1000 | | |
| **Damping (kg $s^{-1}$)** | 100 | | |
| **Friction, kinetic (-)** | 0.2 (front), 0.5 (rear) | | |
| **Friction, static (-)** | 0.3 (front), 0.7 (rear) | | |
| **Friction, velocity threshold ($ms^{-1}$)** | 0.001 | | |

## 4-2 Low-level control

In this section the existing nodes for low-level control (SerialCommunication, PirateCommunication and Movement) are discussed and the changes are elaborated. Also the new additional nodes rViz and robot_state_publisher are discussed.

### 4-2-1 SerialCommunication node

To make sure that all commands can be executed at high level, first the low-level controllers need to be adapted. The SerialCommunication node needs to be able to control each wheel individually.

The PIRATEbay runs the SerialCommunication node as is described in Algorithm 1. The Control_Array.msg that was previously incorporated in this node is removed, since the conversion from any control mode to PWM is done at a higher level. The Limits_Array.msg is removed, since limits should be checked at multiple levels independently from the checks at other levels. This way the system becomes more robust and no unnecessary communication is needed.

Previously, the Setpoint_Array readout in SerialCommunication node used the 11th setpoint as setpoint for all the wheels, while these should have separate setpoints. For example, if only the front part is clamped, only the front wheels should be actuated. This has been changed such that each of the new 18 setpoints can now be used in the SerialCommunication node.

An ID tag has been added to the Setpoint_Array.msg and a publisher has been added to the SerialCommunication to echo the received setpoints. This makes it possible to check if, and when, messages are received by the PIRATEbay, such that it can be verified that all commands are executed. As soon as the callback of the Setpoint_Array is triggered, the Setpoint_Echo is published. Since ROS communication is asynchronous, this can happen anywhere in Algorithm 1.

**Algorithm 1:** Pseudocode for the SerialCommunication node.

Initialisation: Calibrate PICO boards and write initial setpoints, limits and control modes to PICO boards.

Setup: Create SerialCommunication ROS node, Setpoint_Array subscriber, and State and Setpoint_Echo publishers. Set baud rate and start serial communication. Initialise state message.

```
while forever do
    Poll PICO via RS485 serial communication. Set loop counter at 0.
    for every 4 milliseconds do
        switch loop counter do
            case 1 do
                Do nothing.
            case 2 do
                Write setpoints to PICO boards.
            case 3 do
                Read a specific sensor value from one of the PICO boards.
            case 4 do
                Publish the sensor value from the previous loop to ROS.
        if loop counter reached max. number of options then
            Reset loop counter.
        else
            Increment loop counter
    Poll MIDI board.
```

### 4-2-2 PirateCommunication node

The PirateCommunication node is summarised in the pseudocode of Algorithm 2. The publishers and subscribers of Control_Array and Limits_Array, as mentioned in Section 4-2-1, were removed.

Since the Setpoint_Echo is received approximately 0.05 seconds after sending the Setpoint_-Array, a delay of 0.1 seconds is set to prevent sending too much messages to the PIRATEbay. Since about 20% of the messages is missed (determined during test while debugging) and since one of two consecutive messages is always received, all messages are published double to ensure all data is received. Receiving both messages does not result in different behaviour than wouuld be the case if only one message was received. Using a callback at the Pirate-Communication node for the echo messages, checking their ID's and resending a message if missed does not work, since the Movement node keeps publishing messages while the Pirate-Communication node is still waiting to check the ID of the Setpoint_Echo message. This results in an unacceptable delay in communication.

**Algorithm 2:** Pseudocode for the PirateCommunication node.

```
Create PirateCommunication ROS node, publishers for Setpoint_Array and Sensors and
 subscribers for State and Move.
while Node is okay do
    if State message is received then
        Publish this info in a Sensors message.
    if Move message is received then
        Check if control mode is correct. Place setpoint at correct index in the
         Setpoint_Array message. Publish the Setpoint_Array message. Wait. Publish the
         message again. Wait.
```

### 4-2-3 Movement node (at Execution level)

The main class in the Movement node is the Movement class. The building blocks, the Element (module), Motor and Sensor classes and their inheriting child classes, are provided at the Execution level. Instances of these classes are created by the Movement class at the Function level.

The original Motor child classes were *DriveMotor, LedMotor, NoMotor, PositionMotor* and *TorqueMotor*. The original Sensor child classes were *AngleSensor, CameraSensor, ImuSensor* and *SpringSensor*. Since the purpose of the Movement node at Execution level is to provide all building blocks (sensors, motors and modules) for the robot, these blocks should follow the structure of the actual hardware. When changing e.g. the type of motor for the bending joint, this should result in changes for only one Motor child class. The bend, rotation and camera joints have different motors and sensors, so there should be a separate Motor and Sensor child class for each type of joint. The spring angle sensor and bend angle sensor are part of the same gear train and are therefore related. The spring angle sensor is used to determine the spring force in a bending joint based on the rotational spring deflection for this joint. Therefore the SpringSensor is merged with the BendAngleSensor.

The new set of motor child classes therefore is *BendMotor, CameraMotor, Drivemotor, LedMotor, NoMotor* and *RotateMotor*. The sensor child classes are changed to *BendAngleSensor, CameraSensor, ImuSensor, DriveSensor* and *RotateAngleSensor*. The components to which these Motor and Sensor child classes refer can be found in Appendix A.

The Motor and Sensor child classes did not have any implementation yet besides storing the raw sensor/motor value. For the Motor child classes P-controllers are now implemented, so the desired position or velocity can be converted to a PWM value for the motors. A PD-controller is not needed since the motors provide enough damping. For the Sensor child classes, conversion from raw sensor values to SI units is now implemented. Information about these conversions can be found in Appendix B.

The BendAngleSensor class provides bend angle $\gamma_{bend}$, current $I_{bend}$ and torque $\tau_{bend}$. The RotationAngleSensor class provides rotation angle $\phi_{rotate}$ and current $I_{rotate}$. The DriveSensor class provides wheel angle $\theta_{wheel}$, angular velocity $\omega_{wheel}$ and current $I_{wheel}$. The ImuSensor provides the raw values for the accelerometer and magnetometer, the roll angle $\psi_{roll}$ and the pitch angle $\psi_{pitch}$. For the CameraSensor no conversion is implemented as of yet, since the module with this sensor was not available during the project.

The Element child class each describe a type of module (*BendElement, Rotation, Front* and *RearElement*) by creating a set of one or two Motors and one or two Sensors for a module with one PICO board (bend and rear modules) and up to four Motors and up to four Sensors for a module with two PICO boards (rotation and front modules). No changes have been made to the Element child classes.

### 4-2-4 Movement node (at Function level)

The Movement node creates instances of the Elements in a fixed order, as shown in Table 4-2. Even though the front module and the sensor in the rear module are not available, the FrontElement and RearElement are incorporated in the structure. The high-level controller uses the identifiers of the Elements in the list, so including the FrontElement and RearElement results in less changes in code when the physical front and rear module become available for the real robot. No changes were made to this list, except implementing the new Motor and Sensor child classes as defined in Section 4-2-3.

A concise overview of the Movement class is shown in Algorithm 3.

When a SimpleMove request is received, the Movement node translates it to a Move message and passes it through, via the Element class, to the PirateCommunication node. When a Sensor message is received the info is stored at the correct Sensor class instance.

The PIRATE needs to be able to be controlled in different modes. Therefore PWM control (open-loop voltage control, with a setpoint for the duty cycle) and position control are both implemented. The reason why velocity and torque control are not yet implemented is due to the blocking behaviour of the ROS services. Velocity (or torque) control would be a continuous process, so while this service is running no other commands can be executed. Action messages instead of services could be a solution, but this was outside the scope of this thesis. The absence of velocity and torque control is the main reason why the traction controller is designed and tested in simulation. The absence of the velocity controller also has consequences for the high-level controller, as will be discussed in Section 4-3-1.

**Table 4-2:** Structure of the PIRATE as generated in the Movement class. Starred (*) items are currently not available for the real robot.

| ID | Elements | Motors | Sensors |
|---|---|---|---|
| 0 | FrontElement* | CameraMotor*<br>CameraMotor*<br>BendMotor*<br>LedMotor* | CameraSensor*<br>CameraSensor*<br>BendAngleSensor* |
| 1 | BendElement | BendMotor<br>DriveMotor | BendAngleSensor<br>DriveSensor |
| 2 | BendElement | BendMotor<br>DriveMotor | BendAngleSensor<br>DriveSensor |
| 3 | RotationElement | -<br>DriveMotor<br>RotateMotor<br>DriveMotor | IMU<br>DriveSensor<br>RotationAngleSensor<br>DriveSensor |
| 4 | BendElement | BendMotor<br>DriveMotor | BendAngleSensor<br>DriveSensor |
| 5 | BendElement | BendMotor<br>DriveMotor | BendAngleSensor<br>DriveSensor |
| 6 | RearElement | -<br>LedMotor | BendAngleSensor*<br>- |

For moves like clamping, it is important that multiple modules (joints, wheels or LEDs) can be actuated at the same time. This is therefore implemented in the Movement class. For example, bending joints $\gamma_1$ and $\gamma_2$ can be actuated simultaneously, as well as $\gamma_3$ together with $\gamma_4$, and $\gamma_1$ together with $\gamma_2$, $\gamma_3$ and $\gamma_4$. Simultaneous control of these bend joints is possible for both PWM and position control.

**Algorithm 3:** Pseudocode for the Movement node.

```
Create Movement ROS node, publishers for ConfigurationState, JointState, WorkspaceState
  and Move, a subscriber for Sensors and a service server for MoveCommand.
Create an ordered list of Element class instances: FrontElement → BendElement →
  BendElement → RotateElement → BendElement → BendElement → RearElement
while Node is okay do
    if MoveCommand message is received then
        Select corresponding module or set of modules and check if requested control mode is
          possible.
        if PWM control (possible for all motors) then
            Store PWM setpoint at corresponding Element and Motor class instance and
              publish PWM setpoint in a Move message.
        if Position control (possible for bend angles) then
            Send desired angle to corresponding Element and Motor class instance and
              receive angle error and PWM setpoint.
            while At least one of the angle errors is not within angle thresholds do
                for Each angle do
                    if Angle error is not within angle thresholds then
                        Send desired angle to corresponding Element class instance and Motor
                          class instance, receive angle error and PWM setpoint and publish
                          PWM setpoint in a Move message.
                    else
                        Publish PWM setpoint of zero in a Move message.
            For all angles publish PWM setpoints of zero in a Move message.
    if Sensor message is received then
        Select corresponding Element and Sensor class instance, change data to SI format
          and store data at this Sensor class instance.
        Publish ConfigurationState message containing data of all sensors.
        Publish JointState message for visualisation.
        Read out the poses from the TransformStamped message with respect to the first
          module and publish these poses in the WorkspaceState message.
```

### 4-2-5 Additional nodes

To easily see if all sensor values correspond with the real situation, an rViz visualisation is now incorporated in the software structure. The visualisation is based on the exising STL mesh files, which describe the visual appearance of the robot. An example of an rViz visualisation is shown in Figure 3-6 (without the axes). rViz is a native ROS node, which needs information about the angles and torques between segments, described in a sensor_msgs::JointState message, and information about the position and orientation of each segment, described by a geometry_msgs::TransformStamped message. Both are native ROS message structures. All angles and torques are described in the Movement node, so that node publishes the sensor_msgs::JointState.

Since the kinematic structure of the robot is already available in an URDF file, the robot_state_publisher node is used to calculate the position and orientation. This robot_state_publisher node is a native ROS node and is optimised to calculate forward kinematics, and is therefore heavily preferred over writing a method by hand. The local transforms are expressed in poses with quaternions and are published in geometry_msgs::TransformStamped messages. These are then received by the Movement node and from this message the relative pose with respect to the first joint can be calculated. These relative poses are then published in a WorkspaceState message.

To store data while running ROS, the rosbag package is used, which records all messages and saves them in a .bag file. An important note here is that the rosbag package only stores message data, not service data.

## 4-3 High-level control

### 4-3-1 PAB node

The PAB node (Partially Autonomous Behaviour) contains the finite state machines (FSM's) from Section 3-2. Primitive services, either received from the user interface or from the Sequence node, are only executed if the command is possible and purposeful. For example, changing the alignment of the rotation joint is not possible when both parts of the robot are clamped. Clamping the front is not a purposeful command if the front is already clamped, so it will not be executed. This way unnecessary commands will not be passed through to the lower levels, reducing the computative load on the system.

As mentioned in Section 4-2-4 velocity and torque control is not available. Due to the absence of a these controllers the `Free` and `Fixed` state in the `FrontWheelsState` and `RearWheelsState` state machines are actuated in the same way: a setpoint with a PWM duty cycle of 0%. The `DrivingForward` and `DrivingBackward` states are actuated with PWM duty cycles of 100% in the forward and reverse direction respectively. Due to the gear ratio of the wheels, the wheels will resist rotation when the PWM duty cycle is set to 0%, resulting in unwanted friction when a wheel touches the pipe wall.

As mentioned in Section 4-2-5, services are not stored by the rosbag package. To store the changes in the FSM's, all states are published in an FSMState messages.

An overview of the PAB node implementation is given by Algorithm 4.

**Algorithm 4:** Pseudocode for the PAB node.

```
Create PAB ROS node, a publisher for FSMState, a service client for MoveCommand and a
  service server for Primitive.
if Primitive request is received then
    Check if command exists.
    if Command is possible and purposeful in current combination of states then
        for Each FSM do
            Select new state. for Each part of the state command do
                Publish the module, setpoint and mode in a MoveCommand request.
            if All MoveCommand responses were successfull then
                Update current state of each FSM with its new states.
    if All FSM's were updated successfully then
        Return positive Primitive response. Publish FSMState.
```

### 4-3-2 Sequence node

The Sequence node should be the highest node within the Skill level (Table 2-1). Based on external triggers (from the Detector node, Section 4-3-3), sequences of Primitive commands should be executed. The Sequence node should also provide all necessary data for mapping the network, which should happen at Task level.

The Environment message contains information for mapping at the Task level. The type of segment is described as Out (not yet in the pipe), Straight (straight section of the pipe) and Joint (for a mitre bend). This can be expanded to also cover T-joints and other types of segments. The Environment message also contains the pitch and the roll, in order to determine the orientation of the pipe segment, and the diameter of the pipe. The diameter can be calculated as $h + 2r$ by means of the bend angles, as shown in Equation 3-2 and Figure 3-8.

When slip compensation is implemented, the odometry data becomes trustworthy and can be sent within this message as well. This was outside of the scope of this thesis.

An overview of the implementation is shown in Algorithm 5.

**Algorithm 5:** Pseudocode for the Sequence node.

```
Create Sequence ROS node, a publisher for Environment, a subscriber for Detected, a
  service client for Primitive and a service server for PabSequence.
if PabSequence request is received then
    Check if command exists.
    if Command is possible then
        Select corresponding list.
        for Each action in list do
            Send Primitive request and wait a few seconds. If result is successfull, send next
              request.
if ConfigurationState message is received then
    Extract roll, pitch and pipe diameter. Publish Environment message.
if Detected message is received then
    Select corresponding list.
    for Each action in list do
        Send Primitive request and wait a few seconds. If result is successfull, send next
          request.
```

### 4-3-3 Additional nodes

To detect pipe segments like a T-joint, an external sensor is needed, like a camera or proximity sensor. The data from this sensor should be processed in a low-level node, and then send to a high-level node that can detect and distinguish these pipe segments. This high-level node will be the Detector node, which sends triggers to the Sequence node in the form of a Detected message. The implementation of the Detector node depends on the used sensor. The implementation of this node for the experiments is explained in Section 5-4.

## 4-4 Overview

The updated overview of the software framework with all its nodes is shown in Figure 4-4.

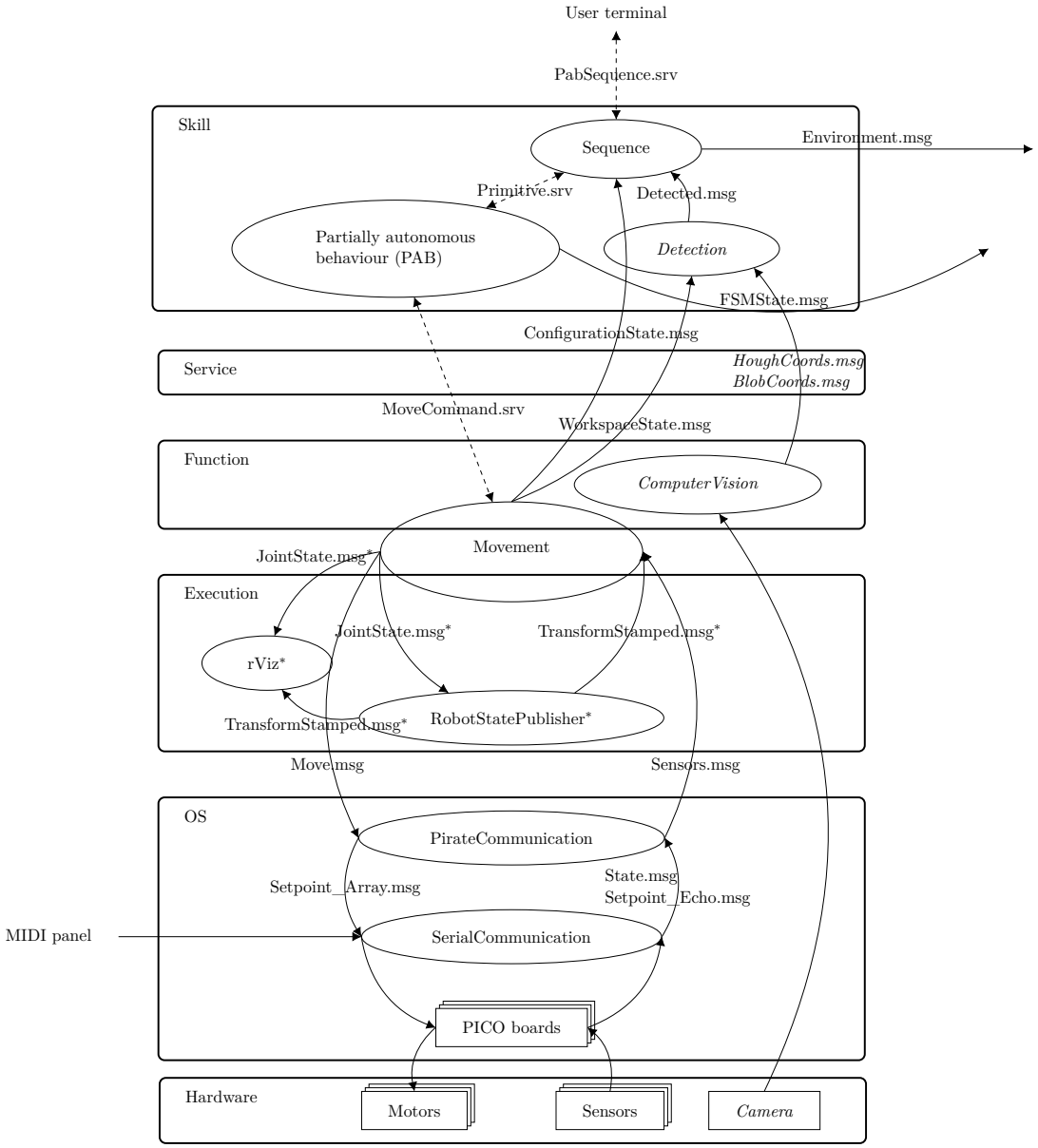



**Figure 4-4:** The updated software overview. The rounded rectangles represent the RobMoSys levels, the ellipses represent the ROS nodes, the arrows represent (ROS) messages, the dashed arrows represent ROS services and the asterisk(*) represents native ROS nodes or message formats. The nodes and messages in *italics* are only added for the experiments and will be discussed in Chapter 5. None of the nodes subscribe to Environment.msg and FSMState.msg, but, as holds for all messages, their contents can be requested in the terminal.

# Chapter 5

# Experiment design

## 5-1 Goals

For the experiment two setups are used, as shown in Figure 5-1.

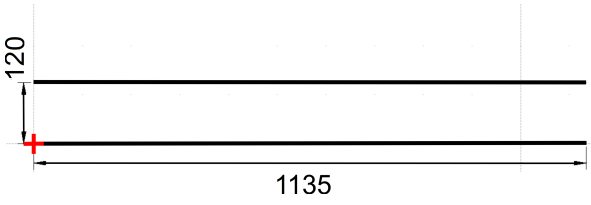
The first goal is to determine the slip ratio and the friction coefficient. Setup 1, a 2D straight pipe (Figure 5-1a, and Setup 2, a real 3D straight pipe (Figure 5-1b), are used to do this. The slip ratio will be determined from the linear velocity of the PIRATE as a whole, which is measured with a camera, and the angular velocity of the wheels, which is measured by sensors on the PIRATE. Based on this data a distinction can be made between a wheel that is contributing to the velocity and a wheel that is not.

The friction coefficient will be determined from the clamping torque in the bend joints ($\gamma_1$ for the front and $\gamma_4$ for the rear) and the torque from its wheels. The friction force $F_{\mathrm{F}}$ and normal force $F_{\mathrm{N}}$ will be used together with the slip ratio to identify the $\lambda$-$\mu_{\mathrm{k}}$-curves, as is done in Hansen et al. [2005], by means of the Magic formula (Section 2-2-6) and Gaussian processes (Section 2-2-7).

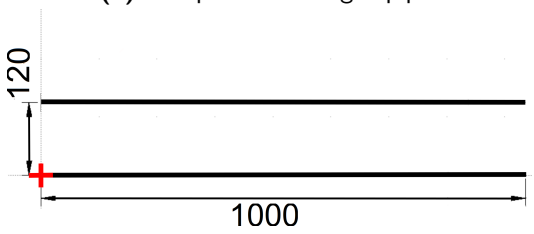
The second goal is to evaluate if the PIRATE can move autonomously through a mitre bend with the updated software. Setup 3, a 2D mitre bend, is used to do this (Figure 5-1c). The location of the robot is determined by a camera. One of the aims is to determine how often the robot gets stuck, and if so, why. Another aim is to determine if the robot can also pass the mitre bend, without falling, if the backplate is tilted. Based on the trials in these setups recommendations will be made for moving through a mitre bend using a proximity sensor instead of a camera.

The third goal is to evaluate each part of the traction controller. This is done in simulation, as explained in Section 4-2-4.

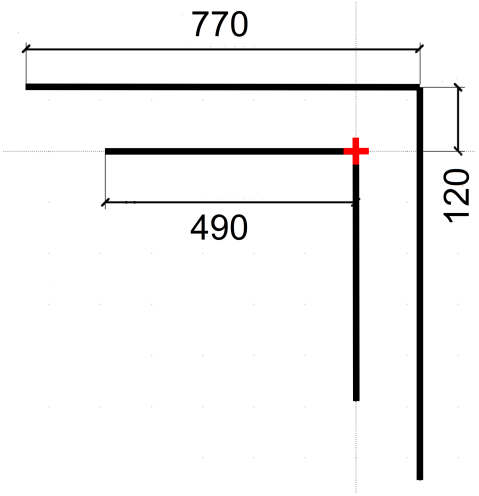
2D setup instead of setups with real pipes were chosen such that the robot was easily accessible, the 3D setup was used to validate the transferability of the 2D setup.




**(a)** Setup 1: 2D straight pipe.

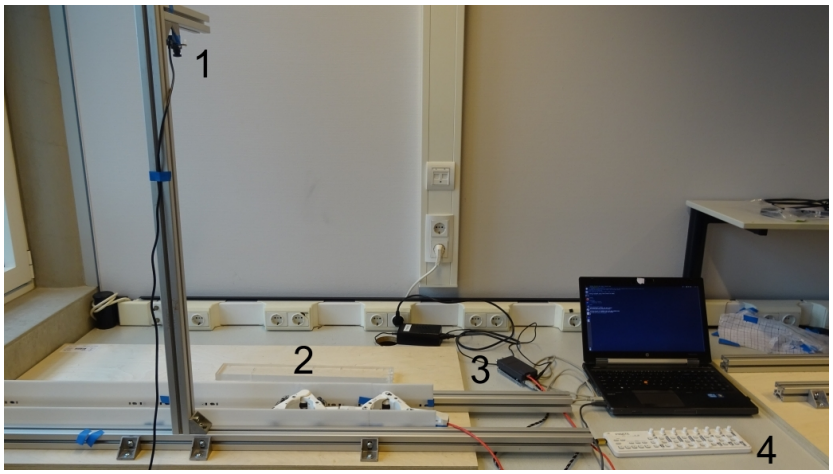



**(b)** Setup 2: 3D straight pipe.




**(c)** Setup 3: 2D mitre bend. The dimensions of the horizontal and vertical part are equal.

**Figure 5-1:** Schematics of the setups, in mm. The origin for the detection is defined at the red plus sign.



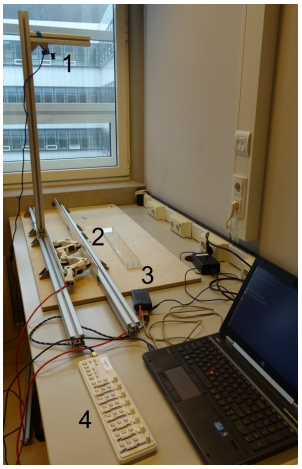




**Figure 5-2:** Setup 1 with the camera (1), PIRATE (2), PIRATEbay (3) and MIDI panel (4).

## 5-2 Materials and dimensions

Setup 1 and 3 (Fig. 5-1) consist of parallel POM (polyoxymethylene) plates which are mounted on aluminium Boikon 40x40mm bars, mounted on a wooden (birch) backplate.

Setup 1 (Fig. 5-1a) only contains straight bars, creating a 'pipe' with a length of 113.5 cm and an inner diameter of 12.0 cm. An impression of this setup during an experiment is shown in Figure 5-2.

In Setup 2 a PVC pipe is placed on top of Setup 1, where the pipe is supported and kept in place by the POM plates.

Setup 3 (Fig. 5-1c) contains a 90° bend of two 'pipes' with both a diameter of 12.0 cm. Both setups are lying flat on the table, such that the plane in which the robot moves is perpendicular to gravity. This way the influence of gravity is the same on all moves of the robot.

To ensure that the robot is clamped and not leaning on the backplate in Setups 1 and 3, the robot is first placed upon a block with a thickness of 2 cm. Once in position the clamping commands are sent and the block is removed.

All setups contain a vertical bar on which a camera can be mounted. The camera used in these experiments is an USB camera module with fisheye lens (ELP-USBFHD06H, 1080p resolution, HD H.264 encoding). This camera is chosen for its wide angle. For Setup 1 and 3 an orange marker is used, since using this colour is detected best. For Setup 2 a blue marker is used. Since the pipe is not fully transparent and has a blue-gray colour, this colour resulted in the best detection for the 3D pipe.

## 5-3 Protocol

### 5-3-1 Experiment 1: driving up and down the pipe

The task in Setup 1 is to clamp the front and/or rear module and drive forwards to the other end of the pipe, and then drive backwards to the start of the pipe. The robot drives up and down three times, triggered by Primitive commands which are sent by the operator. This is done in three experiments:

Exp. 1a Clamp the front and the rear parts in Setup 1.

Exp. 1b Clamp only the front part in Setup 1.

Exp. 1c Clamp only the rear part in Setup 1.

Exp. 1d Clamp the front and the rear parts in Setup 2.

The steps for these experiments are:

1. Send ClampFront command.
2. Send ClampRear command.
3. Remove supporting block.
4. Send DriveForward command.
5. Send Brake command.
6. Send DriveBackward command.
7. Send Brake command.
8. Send DriveForward command.
9. Block the pipe for a moment, in front of the PIRATE (except in Exp. 1d).
10. Send Brake command.
11. Send DriveBackward command.
12. Block the pipe for a moment, behind the PIRATE (except in Exp. 1d).
13. Send Brake command.
14. Send DriveForward command.
15. Send Brake command.
16. Send DriveBackward command.
17. Send Brake command.

For all experiments in Setup 1 (Exp. 1a-1c), at certain moments when driving forward and backward the pipe is blocked. This way the PIRATE bumps into the obstacle and cannot continue driving, while the wheels are still rotating, resulting in a change in slip ratio. These occurrences should be distinguishable in the slip ratio measurements. Due to the inaccessability in Setup 2, this was not done in Experiment 1d.

### 5-3-2 Experiment 2: moving through a bend

For Setup 3, the robot should try to move through the bend autonomously. This is done in three experiments:

Exp. 2a Using Primitive commands, flat backplate.

Exp. 2b Using Sequence commands with Detector triggers, flat backplate.

Exp. 2c Using Sequence commands with Detector triggers, tilted backplate.

In Experiment 2a the PIRATE is controlled by an operator who sends Primitive commands directly to the PAB node, ignoring the Sequence node. The camera is not used in this experiment. In Experiment 2b the operator only has to send the Sequence command to enter the pipe. The detection of the mitre bend is done based on the camera system. The camera detects the marker and determines the location of the marker with respect to the inner corner of the mitre bend. In Experiment 2c the backplate is tilted to show that the PIRATE does not fall down while driving through the mitre bend. The tilt angle should be high enough to make the PIRATE slide away when it is not clamping, but also not to high to ensure that the PIRATE can still provide enough torque to clamp.

To determine this angle the robot is placed on the backplate and the plate is tilted until the robot starts sliding. Then the robot is placed in the horizontal section of the mitre bend and is commanded to clamp the front and rear parts. If this is successfull, the robot is removed, the tilt angle is increased and the robot is commanded to clamp again. This is repeated until

the robot cannot properly clamp itself, thus resulting in the maximum achievable tilt angle for the setup.

Since this experiment is done in a 2D setup, the rotational joint is not needed. This joint also cannot be used since it cannot a rotation of 180°. Therefore the angle of the rotational joint is keps at zero and no commands influencing the RotationState state machine are sent.

### 5-3-3 Experiment 3: traction controller in simulation

In the simulation the robot will first clamp both parts. Next it drives backwards and forwards, each for 10 seconds. When the robot is back at its starting position, the gravitational vector changes from being perpendicular to the pipe to being parallel to the pipe, so the robot will start behaving as if it was in a vertical pipe. It then has to drive backwards and forwards again. Four different settings will be used:

Exp. 3a Velocity control + fixed clamping torque.

Exp. 3b Velocity control with gravity compensation + fixed clamping torque.

Exp. 3c Velocity control + controlled clamping torque.

Exp. 3d Velocity control with gravity compensation + controlled clamping torque.

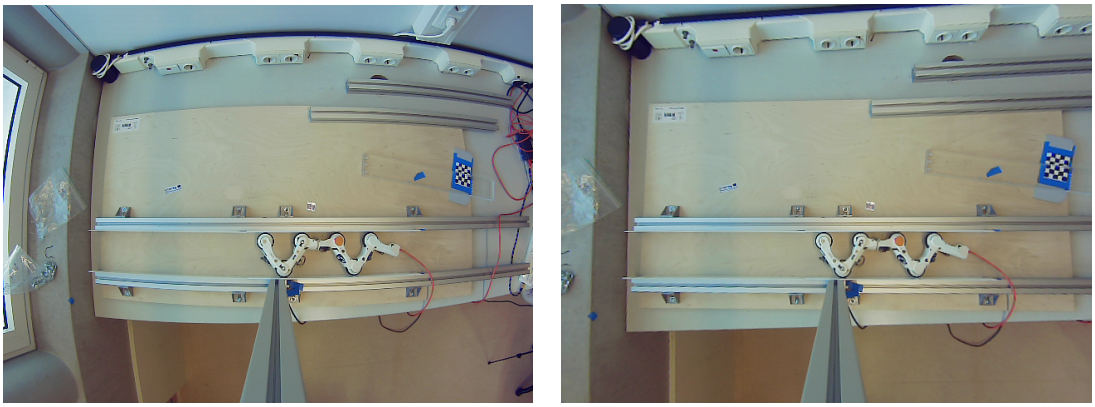
To investigate the influence of the friction coefficient, the wheels in the front part will have different pre-set values for $\mu_s$ and $\mu_k$ from the wheels in the back part. This should influence the clamping torque for Experiments 3c and 3d.

Since Experiments 3a-3d are done in simulation, all triggers can be send at exactly the same time. This makes it easier to compare their performances. The triggers are sent at the following moments:

| | |
|---|---|
| $T = 2$ s | Clamp front |
| $T = 4$ s | Clamp rear |
| $T = 6$ s | Drive backward |
| $T = 6$ s | Change velocity control (Exp. 3b and 3d) |
| $T = 16$ s | Brake |
| $T = 19$ s | Drive forward |
| $T = 29$ s | Brake |
| $T = 32$ s till $T = 37$ s | Gradually change gravity vector from vertical to horizontal. |
| $T = 41$ s | Change clamp control (Exp. 3a and 3c) |
| $T = 44$ s | Drive backward |
| $T = 54$ s | Brake |
| $T = 55$ s | Drive forward |
| $T = 56$ s | Brake |

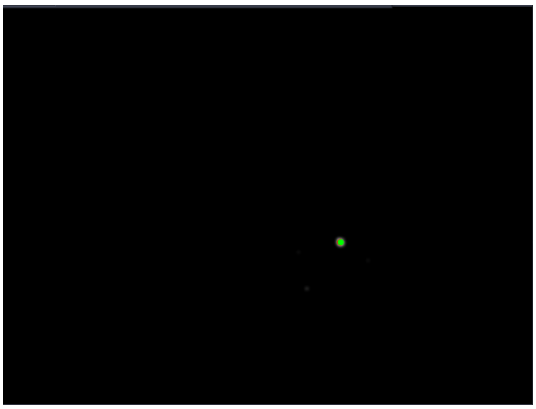
## 5-4 Visual tracking

To track the PIRATE with the camera processing of the image need to be taken into account. The fisheye causes straight lines in the real world to appear curved on the image, as shown in Figure 5-3a. To restore the straight lines to being straight in the image as well, an undistortion algorithm is applied, as shown in Figure 5-3b. This is done by using the C++ OpenCV library [Bradski, 2000]. More information about the image processing can be found in Appendix C-1.



**(a)** Before undistortion.

**(b)** After undistortion.



**(c)** After conversion to HSV, thresholding and detection by Hough circle transform (green) and SimpleBlobDetector (red).

**Figure 5-3:** Steps in the image processing.

Once the image is undistorted, it is converted from RGB to HSV colour space. More information about this can be found in Appendix C-2. To find the orange marker, upper and lower thresholds are set on the hue, saturation and value. Whenever too much artefacts are detected, the thresholds need to be tuned. To reduce noise and prevent false detections, the image is smoothened with a Gaussian filter.

OpenCV offers various feature detection methods, like the Hough circle transform and the SimpleBlobDetector. The Hough circle transform specialises in finding circular shapes, where the SimpleBlobDetector finds blobs of pixels in any shape (Fig. 5-3c).

Whenever the camera is mounted on any of the setups, the extrinsic matrix has to be

(re)calculated. The extrinsic matrix describes the relation between the 2D position of a point in the image and the 3D position of the marker in the real world (Appendix C-2). This is done by selecting four points in the real world and measuring their location in the real world and in the image. Using these values the OpenCV getPerspectiveTransform method approximates the extrinsic matrix. This way the units are also instantly converted from pixels to meters.

The before mentioned steps are all implemented in a separate ROS node named ComputerVision. During each loop the found coordinates of the Hough circle transform and the SimpleBlobDetector are published to ROS. For Setups 1 and 2 these coordinates are only used in postprocessing. For Setup 3, the Detector node checks, based on the coordinates and the local transformations within the PIRATE, if the PIRATE is close to the bend, is in the bend or has passed the bend. This node then sends the appropriate triggers to the Sequence node. The visual tracking nodes are shown in Figure 4-4.

## 5-5 Data collection and postprocessing

All ROS messages are stored in .bag files, as mentioned in Section 4-2-5. To be able to load the data into Matlab, each .bag file is converted to a set of .csv files, where each .csv file contains the data for a certain message type.

In order to calculate wheel slip ratio $\lambda$, the angular velocity from the ConfigurationState message and the coordinates from either the BlobCoordinates message or the HoughCoordinates message are used, depending on which marker detection method has the least artefacts. For the friction coefficient $\mu$, the joint torque for the bending joints and the wheel torque from the ConfigurationState message are both needed.

The message data for the ConfigurationState message remains the same for about 0.1 seconds, while it is published multiple times. This results in wheel angles that remain the same about 0.1 seconds, and then suddenly have a very high angular velocity when the data in the ConfigurationState message is updated. In real life the angular velocity may be constant. This problem cannot be solved by simple interpolation. To prevent this behaviour from distorting the wheel angular velocity, only the first datapoint of such a set of constant datapoints is taken into account.

Since messages are sent asynchronously in ROS, the timing of each type of message is different and the time between two succeeding messages is inconsistent. Therefore the data in the ConfigurationState message and the visual coordinates are interpolated linearly to consistent timesteps of 0.01 seconds, starting at the first moment that a message is published in ROS.

To smoothen the blocky signal of the interpolation and to reduce noise, the signals are filtered using a moving average filter of with a window of $n = 100$ steps (1 second), where all filtered signals are moved 0.5 second back to correct for the time delay caused by the filter. The interpolated and filtered values will be indicated with a tilde ($\tilde{}$), as in Equations 5-1 and 5-2.

The angular velocity is calculated based on the time derivative of the wheel angle, as in Equation 5-1. The reason why the angular velocity is not used is because that signal appears to be an angular difference instead of an angular velocity. This is elaborated on in Appendix B.

$$\tilde{\omega}_i = \frac{\Delta\tilde{\theta}_i}{\Delta\tilde{t}} \tag{5-1}$$

The linear velocity of the marker is calculated by taking the time derivative of the x-coordinate, as in Equation 5-2.

$$\tilde{v} = \frac{\Delta\tilde{x}}{\Delta\tilde{t}} \tag{5-2}$$

From the measurements in the straight pipe $\lambda$ and $\mu$ will be calculated, as discussed in Section 3-4-2. The assumption is made that only the wheels touch the pipe and that no other external forces are applied on the PIRATE. Based on that the friction force for each wheel is calculated based on the wheel torque. Since there are no torque sensors for the wheel, the only way to estimate the wheel torque is by means of the wheel current, as shown in Equation 5-3. The gear train is described in Appendix A [Dertien, 2014]. Since the current signal provided by the PICO boards is always positive, the direction of the wheel torque $\tau_\theta$ is taken from the torque setpoint $\tau_{\theta,i,d}$.

$$\begin{aligned}\tau_{\theta,i} &= k_{\mathrm{m}} r_{\mathrm{gearbox}} \operatorname{sign}(\tau_{\theta,i,d})\, I_{\theta,i} \\ &= 8.44 \cdot 10^{-3} \cdot 112 \operatorname{sign}(\tau_{\theta,i,d})\, I_{\theta,i}\end{aligned} \tag{5-3}$$

When the values for $\lambda$, $F_{\mathrm{F}}$ and $F_{\mathrm{N}}$ have been determined, a relation for $\lambda$ and $\mu$ can be found by means of identification. The parameters of the Magic formula will be optimised by a least squares optimisation, which is a simple well known method to fit parameters. For the Gaussian processes the GPML toolbox will be used, with a mean of zero and a squared exponention covariance function [Rasmussen and Nickisch, 2010].

# Chapter 6

# Experimental results

## 6-1 Overall performance

At low level, due to double publishing and setting the correct time delays, all commands from the PirateCommunication node are received and executed by the PIRATEbay (Section 4-2-2). For the position controller no overshoot is observed.

During the experiments Wheels 3 and 4 are not actuated, even though control signal is provided to the PIRATEbay. Furthermore no sensor signals are received for Wheel 3 and 4 and therefore they are not taken into account in the rest of this chapter.

## 6-2 Setups 1 and 2: straight pipe

In Setups 1 and 2, the straight pipes as described in Section 5-2, the PIRATE was able to clamp itself and drive up and down the pipe horizontally. In Setup 1 the PIRATE did slide down, but only when driving.

Figure 6-1 shows the joint torques. The means of the measured absolute clamping torques are 0.1672, 1.6009, 0.5162 and 0.2043 Nm for bending joints 1, 2, 3 and 4 respectively. The medians for the clamping torques of these bending joints are 0.1587, 1.9230, 0.4873 and 0.2373 Nm respectively. In this trial $\gamma_3$ has one artefact around $T = 40$ s, where the bend angle jumps to its lower limit and the torque becomes very high.

Figure 6-2 shows the comparison of the velocity of the marker along the x-axis with the linear velocity of the wheels. The data from the SimpleBlobDetector algorithm is used to determine the linear velocity.

Wheels 5 and 6 are spinning slightly. When applying a force opposite to the driving direction while driving forward, Wheel 1 rotates more slowly and Wheel 5 keeps the same speed as before the block. Wheels 2 and 6 stop rotating. When applying a force opposite to the driving direction while driving backward, Wheel 5 and 6 keep the same speed, Wheel 1 rotates even faster than before applying the force, while Wheel 2 starts to rotate more slowly.

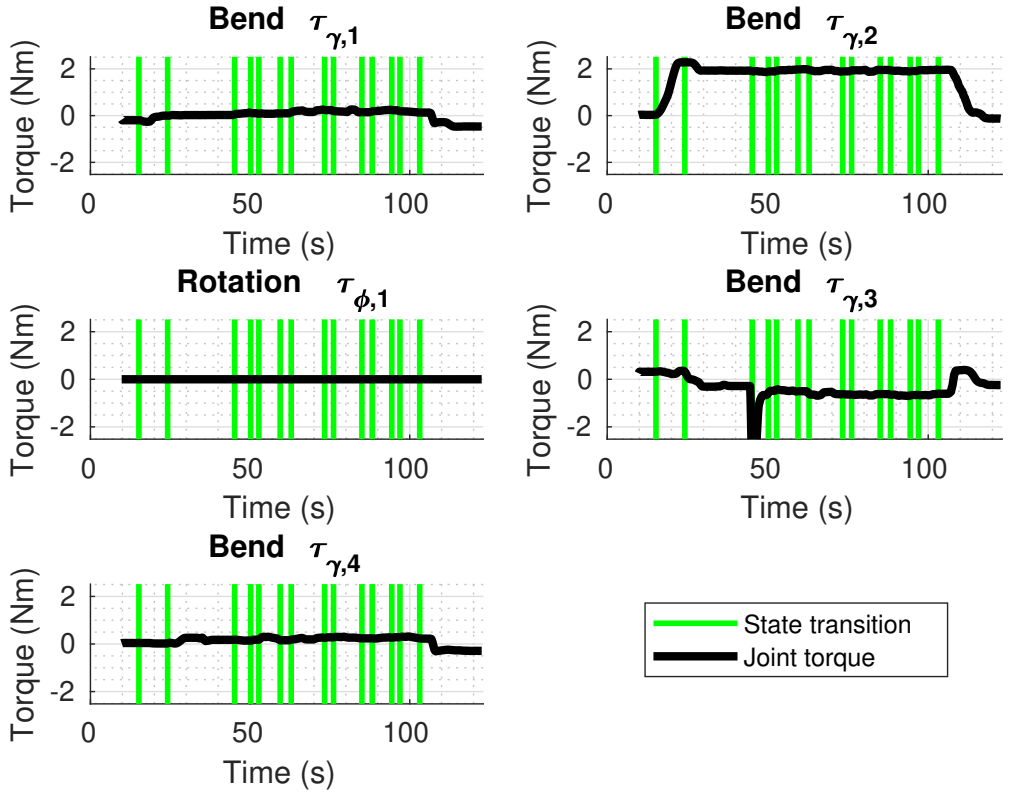



**Figure 6-1:** Exp. 1a: Joint torque (Setup 1).

Figure 6-3 shows the wheel slip ratio. When driving (the yellow to red range in Fig. 6-3), the wheel slip ratio is between -0.3 and 0.3. The mean and standard deviation are shown in Table 6-1. Note the drops of $\lambda$ at $T = 70$ s and $T = 80$ s.

Figure 6-4 shows the normal force $F_{\mathrm{N}}$ and the friction force $F_{\mathrm{F}}$, Figure 6-5 shows the friction coefficient $\mu$. The friction force increases whenever a wheel is actuated, but there is no peak at the start of an actuation. For both the front and the rear part, the clamping force shows an increasing trend. Note the low friction force and friction coefficient for Wheel 5 with respect to Wheels 1, 2 and 6.

Figure 6-6 shows friction coefficient $\mu$ with respect to slip ratio $\lambda$, while Figure 6-7 shows $\mu$ with respect to the relative velocity $|v - d_i\omega_i r|$.

The maximum measured friction coefficients are 2.40, 0.93, 0.15 and 1.3 for wheels 1, 2, 5 and 6 respectively. For wheels 2, 5 and 6 the friction coefficient became approximately constant for relative velocities higher than 0.07 m/s, resulting in constant friction coefficients of approximately 0.5, 0.07 and 0.07 respectively.

Figure 6-8 shows the fits of the Magic formula and Gaussian processes for the data from Figures 6-6 and 6-7. Per ten datapoints one point is used for training.

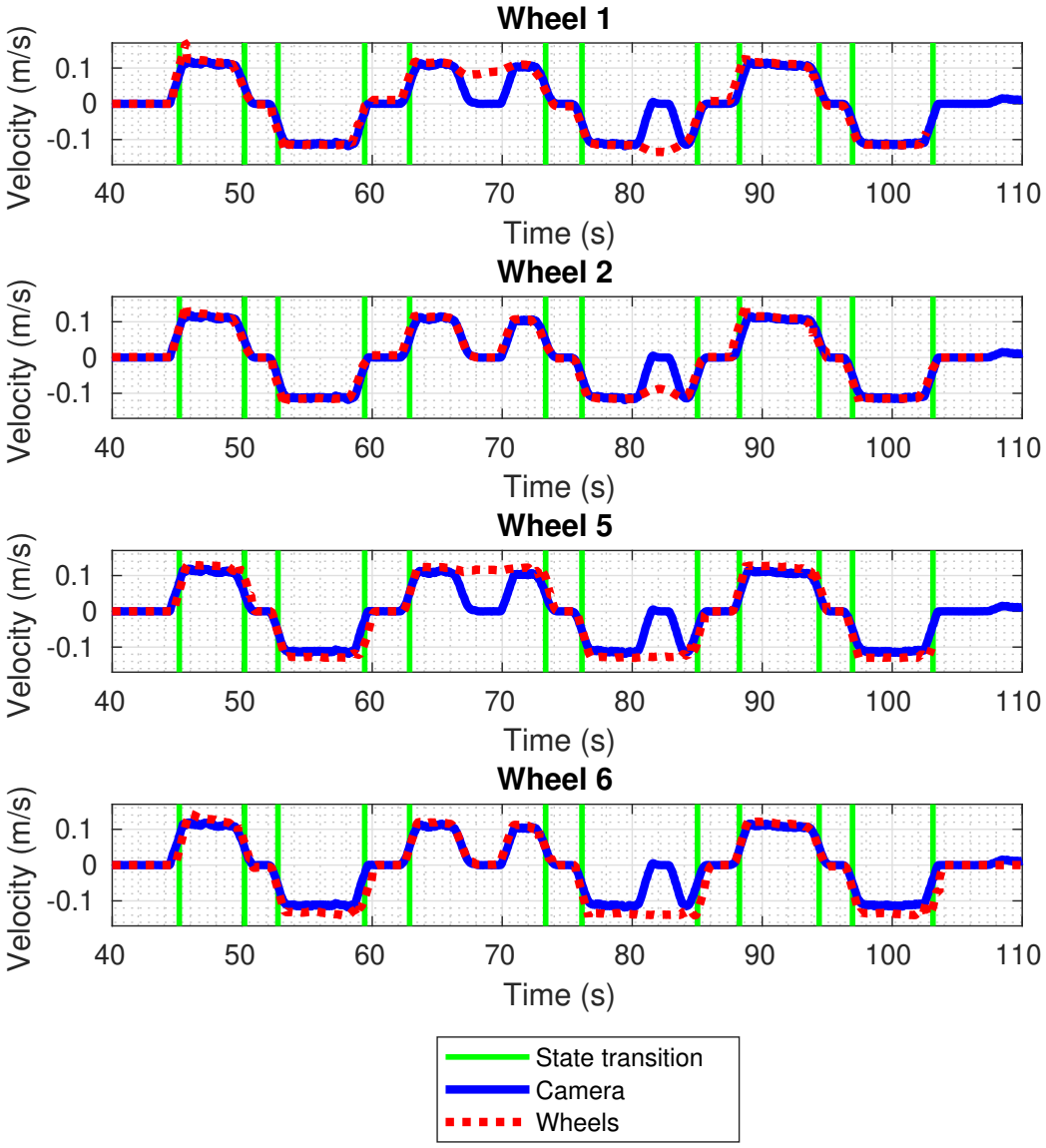



**Figure 6-2:** Exp. 1a: Velocity comparison (Setup 1).

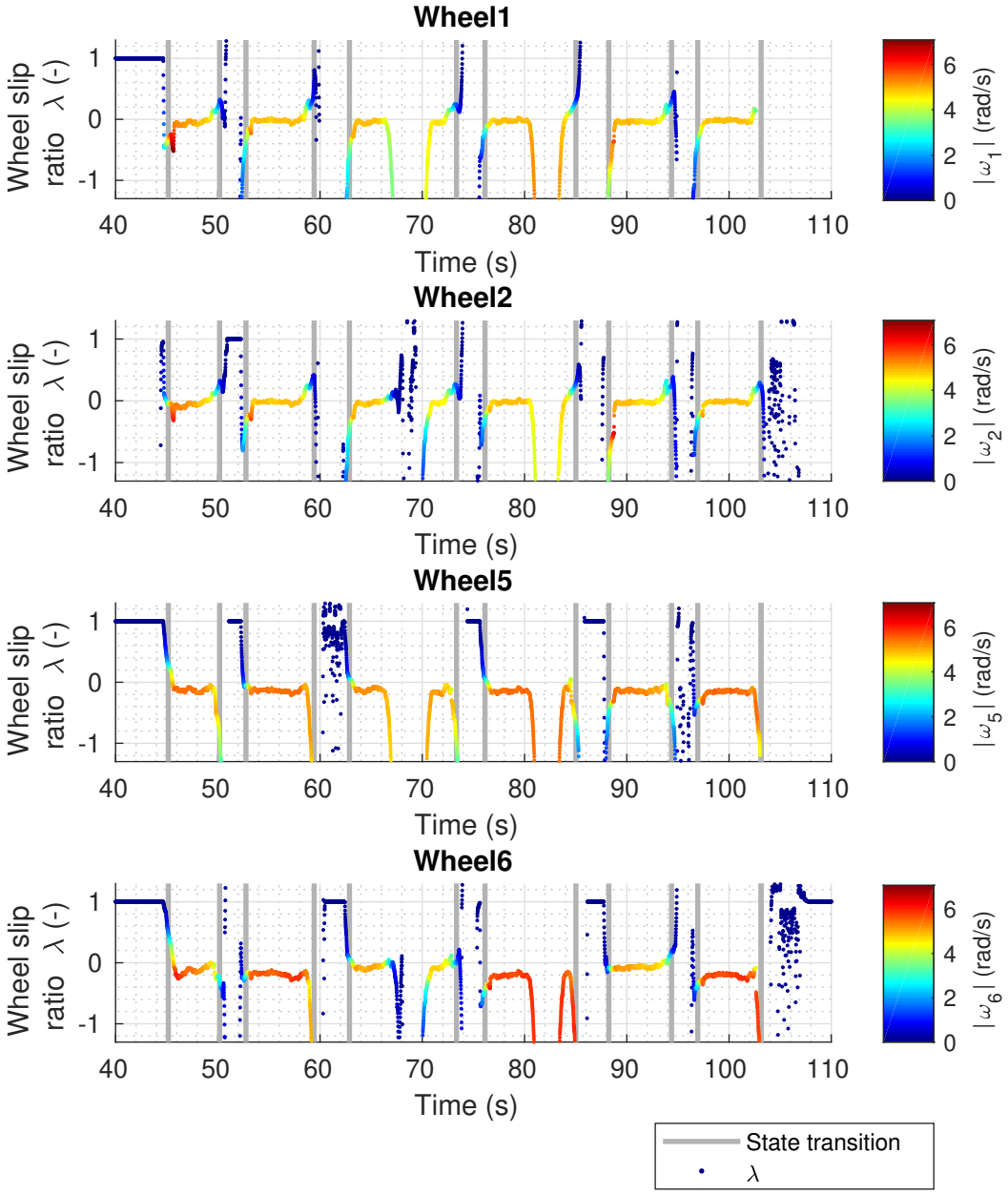



**Figure 6-3:** Exp. 1a: Wheel slip ratio (Setup 1).

**Table 6-1:** Distribution of the wheel slip ratio. High values of $|\lambda| \geq 1$ are removed to be able to calculate a mean and std. For comparison also the median, mean and std. when driving ($|v| \geq 0.09$ m/s) is shown. For the front wheels in Exp. 1b and the rear wheels in Exp. 1c the wheels are not clamped and actuated, and thus the corresponding cells are empty.

| | | Wheel | 1 | 2 | 5 | 6 |
|---|---|---|---|---|---|---|
| Exp. 1a | $\lvert\lambda\rvert \leq 1$ | Mean | 0.2641 | 0.0482 | 0.3687 | 0.3576 |
| | | Std. | 0.5129 | 0.3434 | 0.6004 | 0.5876 |
| | $\lvert v\rvert \geq 0.09$ m/s | Median | -0.0300 | -0.0201 | -0.1369 | -0.1435 |
| | | Mean | -0.0286 | -0.0226 | -0.1396 | -0.1641 |
| | | Std. | 0.0750 | 0.0751 | 0.0550 | 0.0948 |
| Exp. 1b | $\lvert\lambda\rvert \leq 1$ | Mean | | | 1.0000 | 1.0000 |
| | | Std. | | | 0.6093 | 0.6111 |
| | $\lvert v\rvert \geq 0.09$ m/s | Median | | | -0.1051 | -0.1176 |
| | | Mean | | | -0.1332 | -0.1348 |
| | | Std. | | | 0.1033 | 0.1241 |
| Exp. 1c | $\lvert\lambda\rvert \leq 1$ | Mean | 1.0000 | 1.0000 | | |
| | | Std. | 0.1435 | 0.0231 | | |
| | $\lvert v\rvert \geq 0.09$ m/s | Median | 0.5475 | 0.5519 | | |
| | | Mean | -0.0261 | -0.0251 | | |
| | | Std. | -0.0227 | -0.0455 | | |
| Exp. 1d | $\lvert\lambda\rvert \leq 1$ | Mean | 0.1073 | 0.1502 | 1.0000 | 0.3367 |
| | | Std. | 0.4817 | 0.5152 | 0.5271 | 0.5447 |
| | $\lvert v\rvert \geq 0.09$ m/s | Median | 0.0287 | 0.0216 | 0.0112 | -0.0158 |
| | | Mean | 0.0349 | 0.0276 | 0.0071 | -0.0200 |
| | | Std. | 0.0867 | 0.0936 | 0.0672 | 0.0706 |

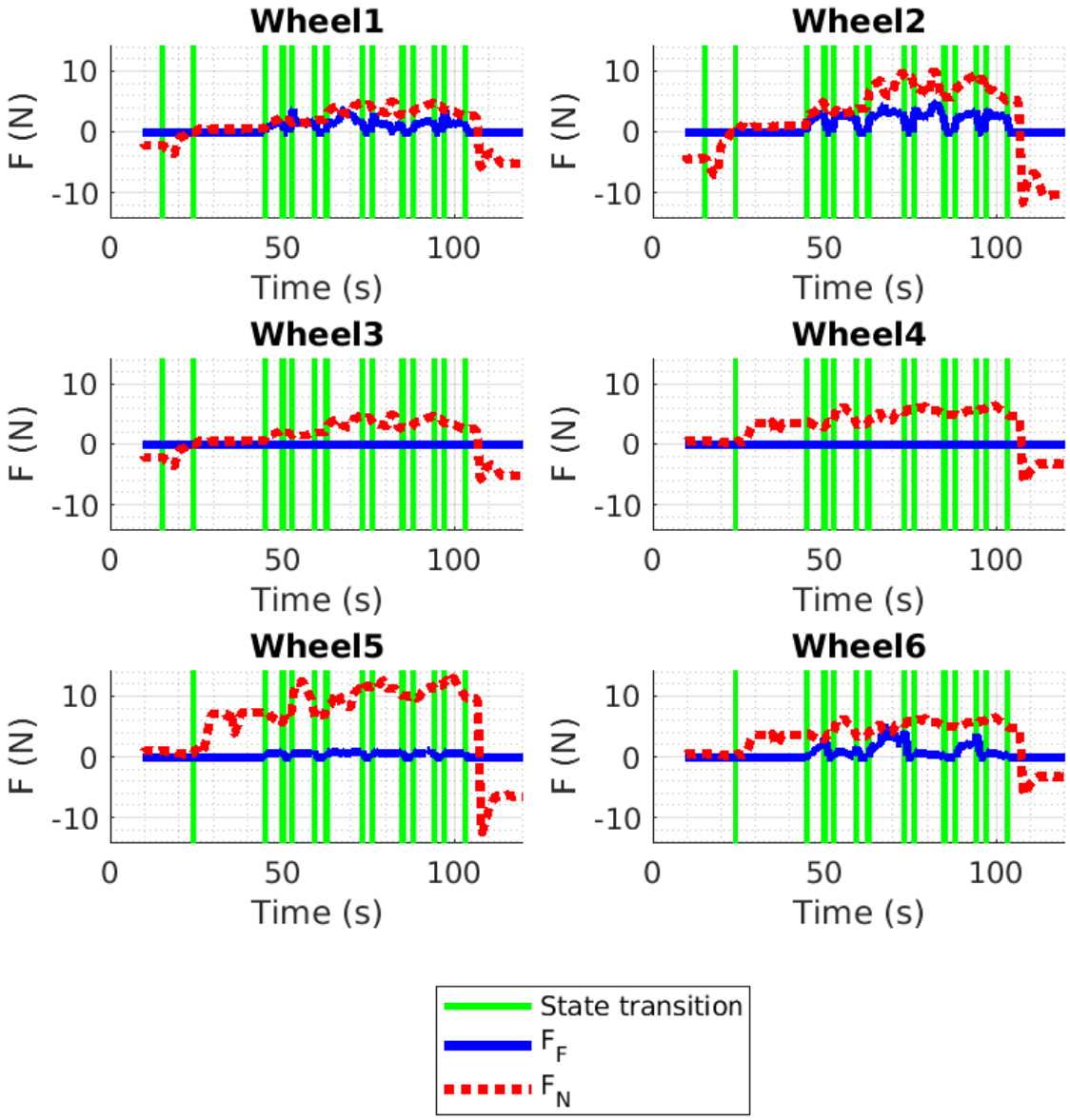



**Figure 6-4:** Exp. 1a: Forces (Setup 1).

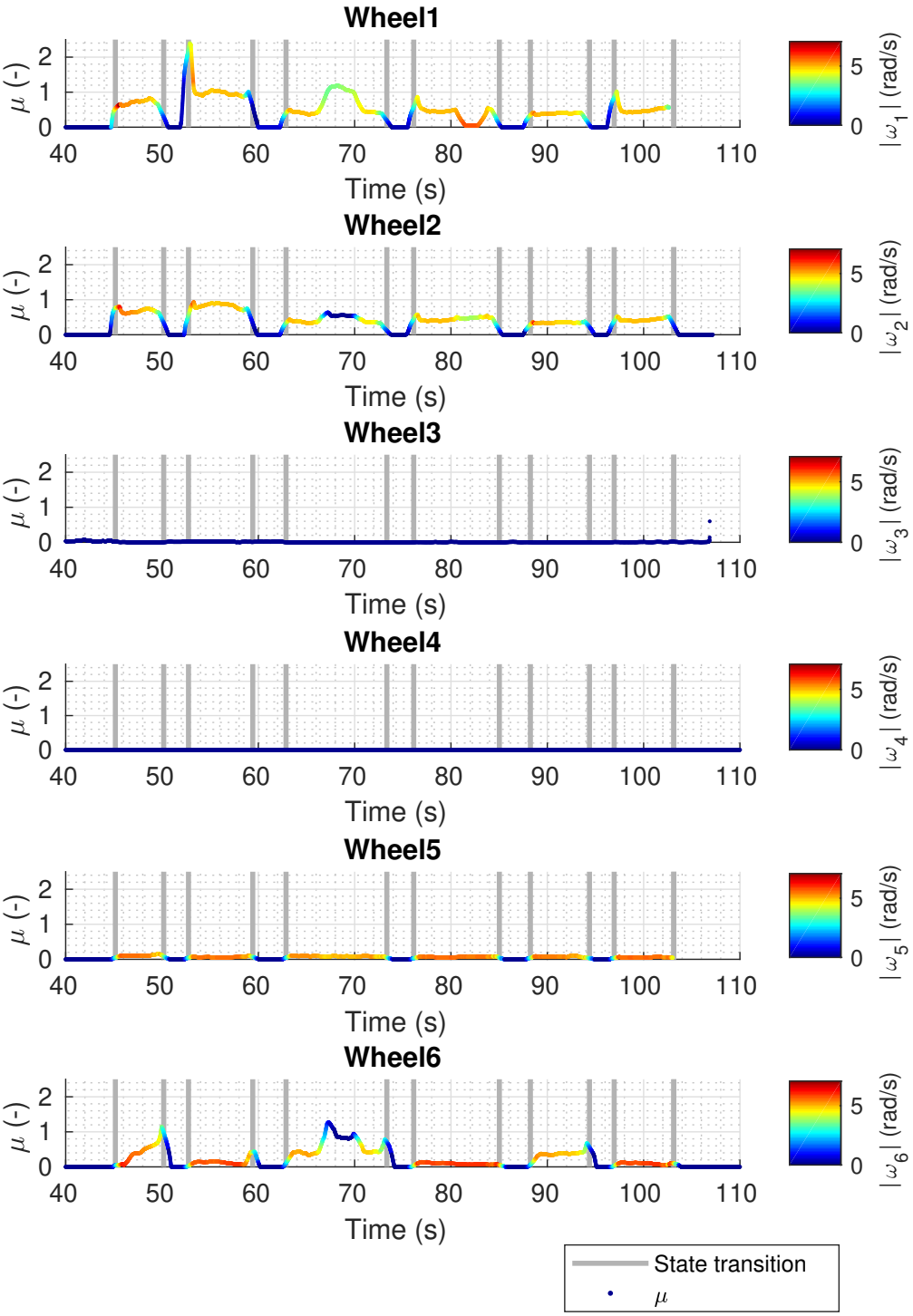



**Figure 6-5:** Exp. 1a: Friction coefficient (Setup 1).

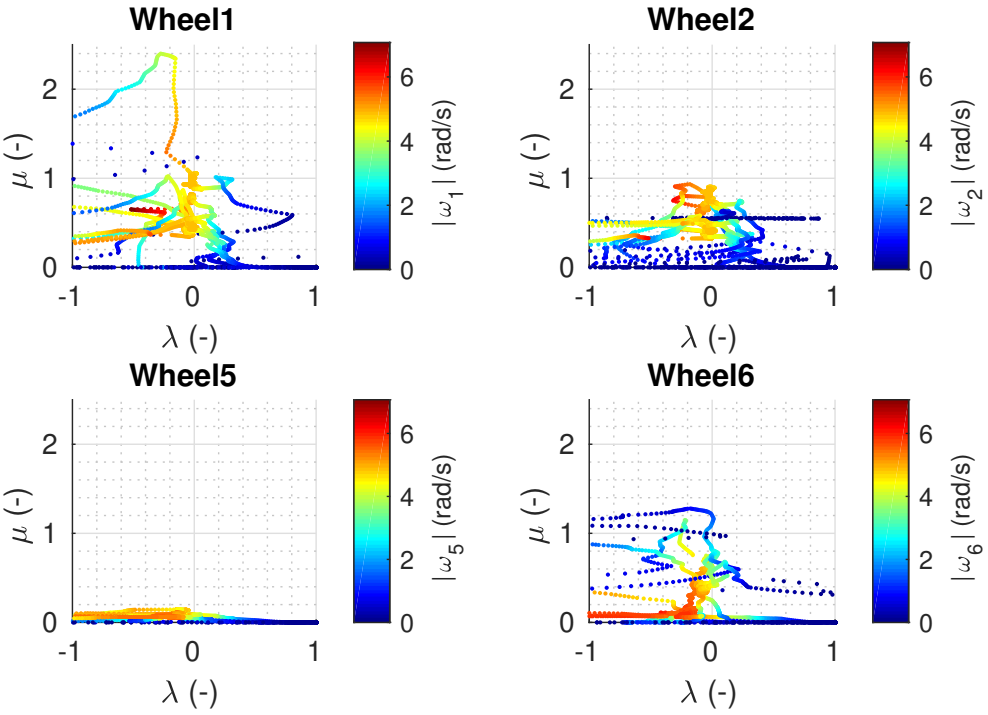



**Figure 6-6:** Exp. 1a: Slip ratio with respect to the friction coefficient (Setup 1).

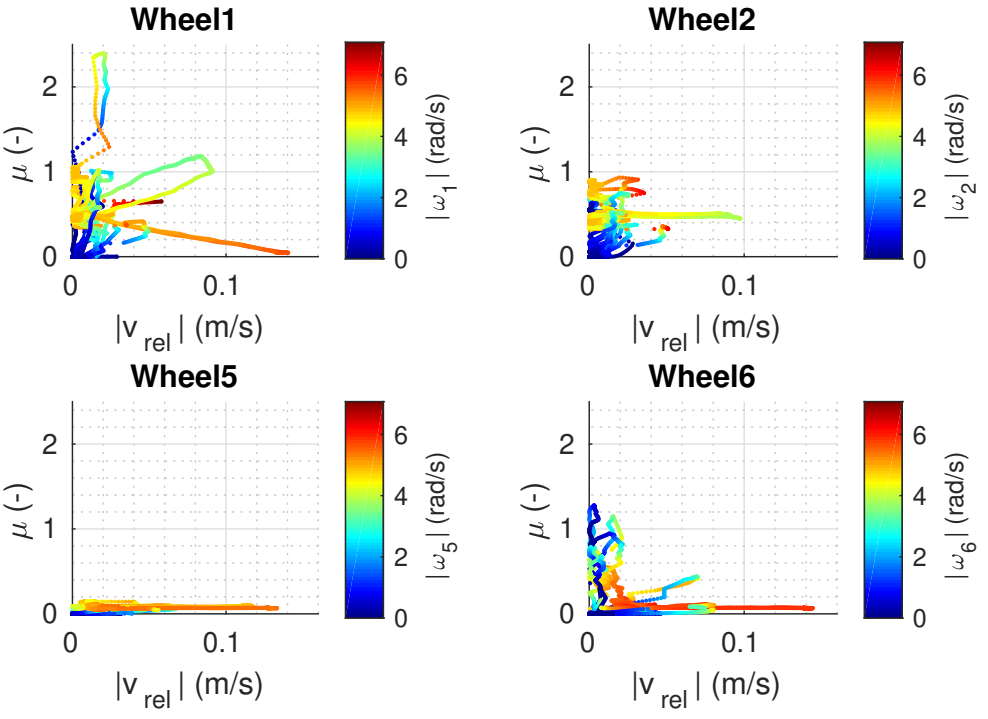



**Figure 6-7:** Exp. 1a: Absolute relative velocity with respect to the friction coefficient (Setup 1).

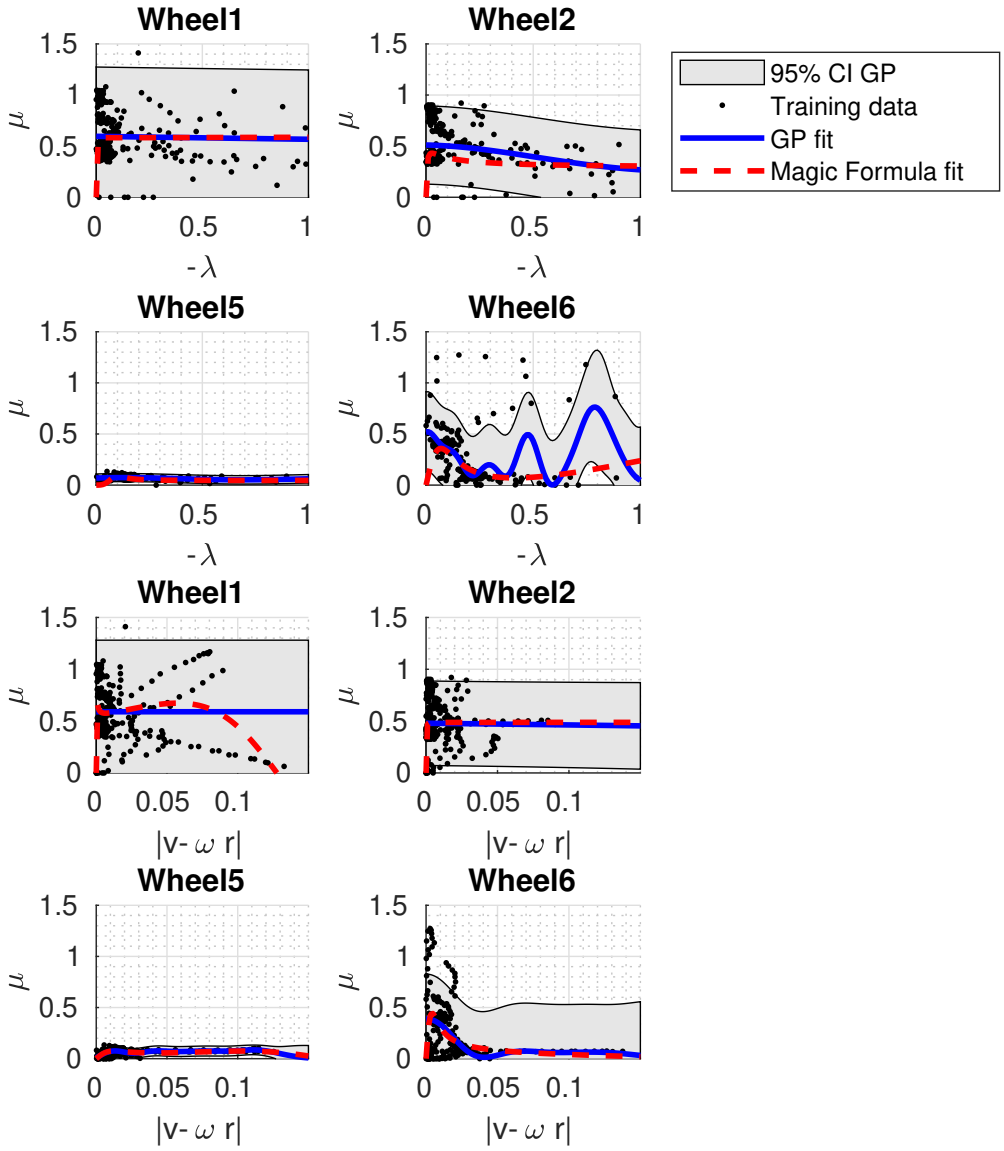
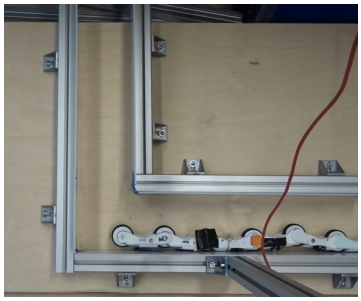


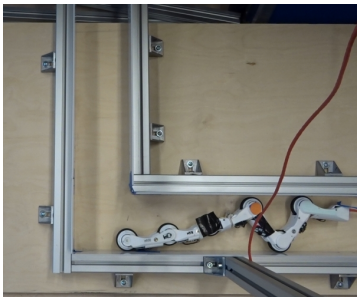

**Figure 6-8:** Exp. 1a: Fitting the Magic formula and a Gaussian processes (GP). Only the negative part of $\lambda$ is taken into account due to the definition of the slip in the Magic formula.

## 6-3 Setup 3: 2D mitre bend

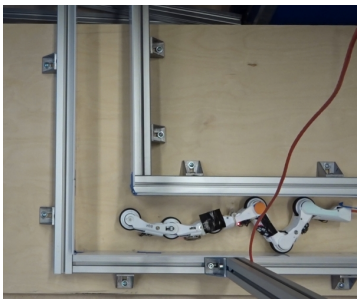
Driving through the corner using Primitive commands given by the operator (Exp. 2a) was successful, as is shown in Figure 6-9. It was also successful for the Sequence commands when the backplate is horizontal (Exp. 2b) and tilted at an angle of 28° (Exp. 2c). The second bend joint needed manual push when using position control (transition from Fig. 6-9b to 6-9c), otherwise it did not start moving. The PIRATE was faster in moving through the mitre bend in Experiment 2a than in Experiment 2b.



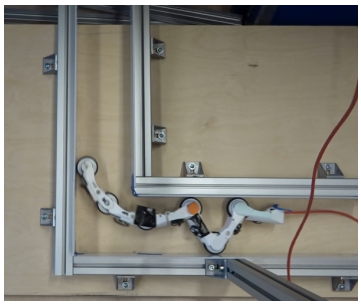
**(a)** Initial configuration.
FWS: Free
RWS: Free
BS: Unclamped
RS: Unaligned
LS: Off



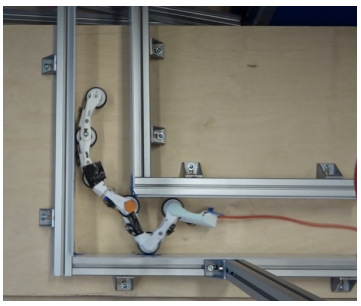
**(b)** Clamping the rear.
FWS: Free
RWS: Fixed
BS: RearClampedFrontRelaxed
RS: Unaligned
LS: On



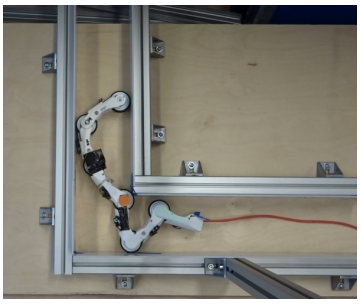
**(c)** Bending the front.
FWS: Free
RWS: Fixed
BS: RearClampedFrontBending
RS: Unaligned
LS: On



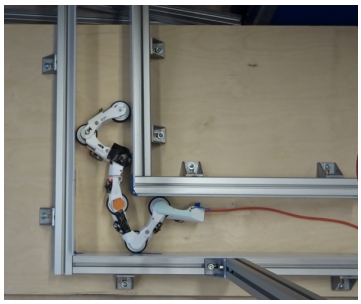
**(d)** Driving into the bend.
FWS: Free
RWS: DrivingForward
BS: RearClampedFrontBending
RS: Unaligned
LS: On



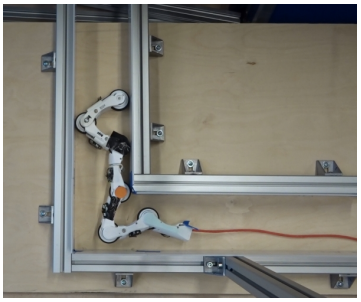
**(e)** Brake just before Wheel 4 enters the corner.
FWS: Free
RWS: Fixed
BS: RearClampedFrontBending
RS: Unaligned
LS: On



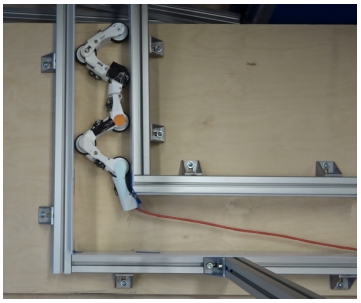
**(f)** Clamp the front.
FWS: Fixed
RWS: Fixed
BS: DoubleClamped
RS: Unaligned
LS: On



**(g)** Turn $\gamma_2$ the other way around, so front can fully clamp.
FWS: Fixed
RWS: Fixed
BS: FrontClampedRearBending-Back
RS: Unaligned
LS: On



**(h)** Drive out of the bend.
FWS: DrivingForward
RWS: DrivingForward
BS: DoubleClamped
RS: Unaligned
LS: On



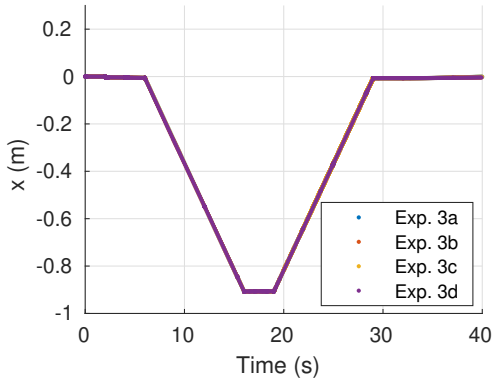
**(i)** Passed the bend.
FWS: Fixed
RWS: Fixed
BS: DoubleClamped
RS: Unaligned
LS: On

**Figure 6-9:** Exp. 2c: PIRATE moving through the bend. For the state machines the following abbreviations are used: FWS (FrontWheelsState), RWS (RearWheelsState), BS (BendingState), RS (RotationState) and LS (LightingState).

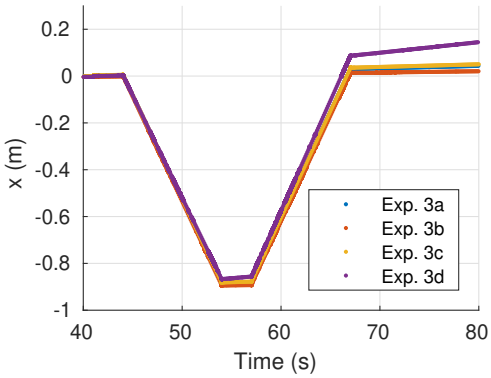
## 6-4 Simulation

When the simulated robot tries to move through a mitre bend, it starts bouncing fiercely, clips through the wall and often drops out of the pipe. The simulated robot is able to drive, clamp, bend and rotate in a straight pipe.

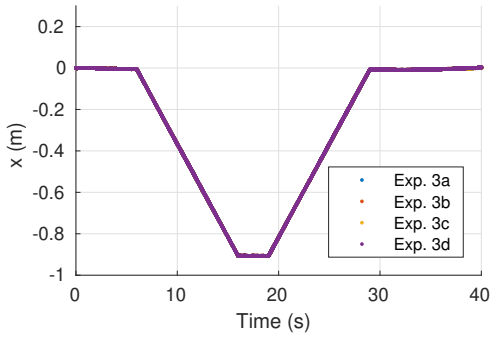
For the experiments where the robot has to drive up and down in a pipe with various controllers (Exp. 3a-3d), the simulation is about 5x times slower than the real time. The position of the model in the pipe in Exp. 3a-3d is shown in Fig. 6-10, where $x$ is defined along the length of the pipe. Initially the controllers are tested without noise on the sensors, as shown in Fig. 6-10a:6-10b. The tests with noise are shown in Fig. 6-10c:6-10d.



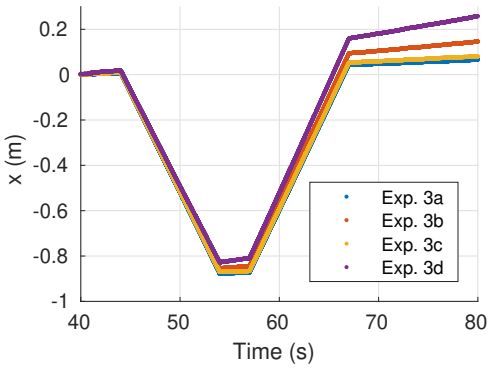

**(a)** Vertical gravity, no noise.




**(b)** Horizontal gravity, no noise.




**(c)** Vertical gravity, sensor noise.




**(d)** Horizontal gravity, sensor noise.

**Figure 6-10:** Exp. 3a-3d: Position of the model in the pipe.

Figure 6-11 shows the odometry error ($x - r_i \sum_t \omega_i \Delta t$) for the case with sensor noise. For both Experiments 3c and 3d the odometry error is approximately 2 cm, based on driving up and down for a distance of approximately 80-90 cm.

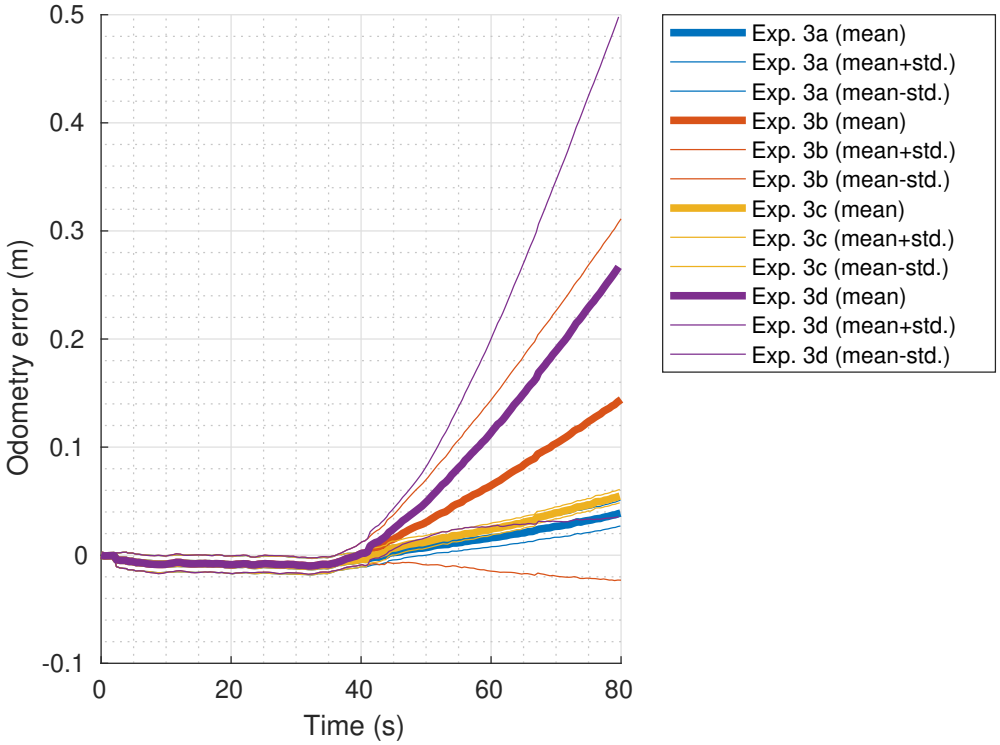
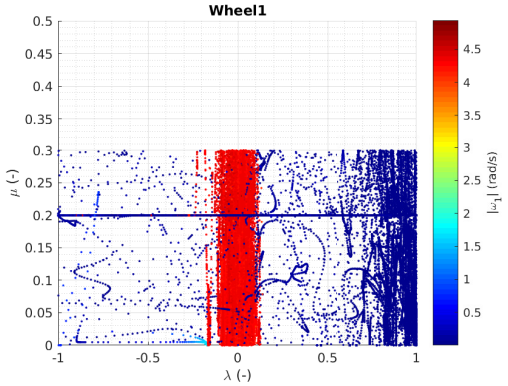
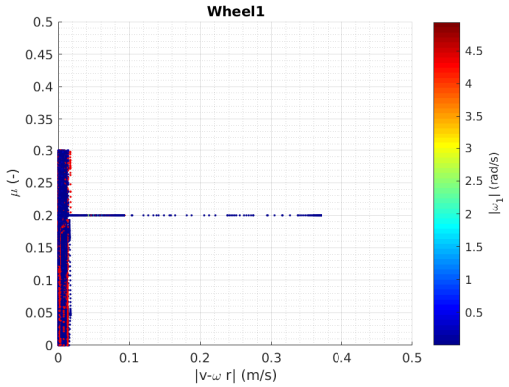



**Figure 6-11:** Exp. 3a-3d: Odometry error when sensor noise occurs. The signals of the individual wheels are lumped together in the mean and std.

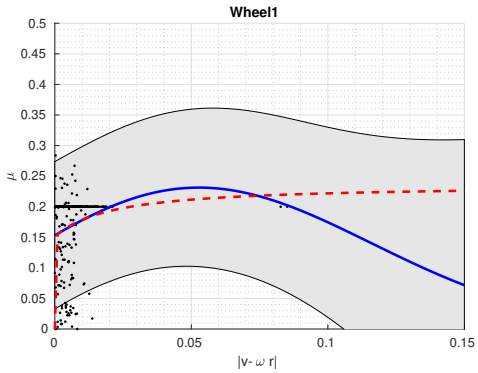
Figure 6-12 shows the elation between $\lambda$ and $\mu$ (Fig. 6-12a) and between $|v-\omega r|$ and $\mu$ (Fig. 6-12a) for Wheel 1 in the case with sensor noise. A fit for the Magic formula and a Gaussian process are shown for the relation between $|v-\omega r|$ and $\mu$ (Fig. 6-12c). Per 600 datapoints one point is used for training. Experiments 3a-3c give similar results. For Wheels 2 and 3 the same maximum $\mu$ of approximately 0.3 and the same constant $\mu$ of approximately 0.2 for the $|v-\omega r|$ plot are found. For the other Wheels these values are 0.7 and 0.5 respectively.




**(a)** Relation between $\lambda$ and $\mu$.




**(b)** Relation between $|v-\omega r|$ and $\mu$.




**(c)** Fitting the Magic formula and a Gaussian processes (GP).

**Figure 6-12:** Exp. 3d: Relation for friction coefficient $\mu$ for Wheel 1 in the case with sensor noise.

# Chapter 7

# Conclusions and recommendations

## 7-1 Discussion

### 7-1-1 Behavioural issues

A major issue that occurs in most of the experiments is that the PIRATE suddenly starts to randomly actuate motors that should not be actuated. When this occurs the SerialCommunication node stops publishing messages, without any error message. The problem has not yet been found in the PIRATEbay code, so it can either be a bug in the code on the PICO boards or it can be a broken connector in the PIRATE. This issue heavily increases the time needed to do experiments.

The sensor for the second joint angle is unreliable, since the signal often drops to its lower limit or remains zero rad, instead of giving any value that resembles what is actually happening in the joint. For control of the robot in Experiments 1a-1d this is not a problem, since, in those specific experiments, only open-loop voltage control is used for the angle. For Experiments 2a-2c position control of the joints is needed, so these experiments can not be carried out when these sensor issues occur. The problem is often that either the sensor connector becomes loose or the magnet of the magnetic sensor, supposedly fixed to the other module in the joint, is rotating along with the sensor, so no angle with respect to the zero position is measured.

Extra unwanted friction is generated by Wheels 3 and 4 when they touch the pipe wall while driving, since these wheels are not actuated and cannot be put in a zero-torque mode, as this mode is currently not yet available, resulting in the motors resisting rotation. The torque in the second bend joint is about ten times as high as for the other bending joints (Fig. 6-1) due to extra friction in that joint. This friction is sometimes so high that a manual push is needed to overcome the static friction in the joint, as mentioned in Section 6-3.

The artefact for the torque in the third bend joint (Fig. 6-1) is most likely a physical connection that was not making full contact for a moment, since the value dropped to the lower limit.

### 7-1-2 High-level controller

As mentioned in Section 6-1 no position overshoot occurred, so the position control loop in ROS is fast enough.

As mentioned in Section 6-3 the experiment with the Primitive commands is faster than with the Sequence commands. This is due to the time delays built into the Sequence node (Exp. 2b and 2c), while the Primitive commands (Exp. 2a) were provided by the operator. Currently the PAB node sends a command to the Movement node and gets a reply when the Movement node has processed the request, but not if the action corresponding to that request has been finished. For example, the PAB node sends the ClampFront command, the Movement node formulates and sends a Move message, the Movement node then lets the PAB node know that the command has been processed, but the PAB node does not know when the front part is clamped. In the Sequence node enough time delay has been built in to assure that the PIRATE has enough time to perform such an action.

### 7-1-3 Experimental setups

The robot does slide down (along the wheel rotation axis) while driving in Setups 1 and 3, as mentioned in Section 6-2. When the support block is removed, the configuration of the robot slightly changes, as it suddenly has to support its own weight by clamping. This minor change in orientation in combination with the gravitational force may have caused the PIRATE to touch the backplate at a certain point while driving. This movement against the backplate results in extra unwanted friction. When driving in a real pipe in Setup 2, this was not a problem, since the curve of the pipe counteracts the down-sliding motion.

### 7-1-4 Visual tracking

For the estimaton of the velocity a choice had to be made between the SimpleBlobDetector and the Hough algortihm. After implementing both methods, the SimpleBlobDetector is used as this resulted in far less artefacts than the Hough algorithm.

### 7-1-5 Wheel slip estimation

The slight negative slip ratio in Figure 6-3 indicates that only a part of the angular velocity is translated into linear velocity. Wheel spin is detected, as can be seen by the drops at $\lambda$ at $T = 70$ s and $T = 80$ s. Based on the wheel slip ratio alone a distinction between wheel spin and zero angular velocity at standstill cannot be made. In order to determine which is the case, the angular velocity has to be taken into as well, like is done in Figure 6-2.

The low wheel slip ratio when driving is expected due to the low speed of the robot, which causes momentum and inertia to have a low influence.

As mentioned for Figure 6-2, the wheels do not behave consistently with respect to eachother when the pipe is blocked.

As can be seen in Figure 6-6, a relation between $\lambda$ and $\mu$ is hard to idenfity. When plotting the absolute relative velocity ($|v - \omega r|$) with respect to $\mu$ in Figure 6-7, one would expect

a shape as in Figure 6-12b. This is not the case. The horizontal spikes in Figure 6-7 are a result of blocking the pipe.

Due to the inconsistency of the measured clamping torque and friction force, and the absence of sensor signals for Wheel 3 and 4, $\mu_s$ and $\mu_k$ can not be estimated properly. A fitting curve between $\lambda$ and $\mu$ or $|v_{rel}|$ and $\mu$ can not be generated by the Magic formula or the Gaussian processes, as shown in Figure 6-8. A possible cause for the bad relations could be that the wheel torque, from which the friction force is estimated, is wrongly estimated based on the current used by the wheel actuator. Currently there is no other way to estimate the wheel torque.

### 7-1-6 Simulation model

Using similar commands for the Simulink simulation model as for the ROS implementation, the behaviour of the robot can be simulated at high level in a straight pipe. For a transition between two pipes the simulation model can not be used. The bouncing behaviour as mentioned in Section 6-4 made it impossible to move through a bend. This behaviour is most likely caused by a resonance effect that occurs because the pipes are modelled as mass-spring-damper systems. More investigation should be done on tuning the parameters to find out if the current way of modelling can be used for simulating moves through a mitre bend, or that this should be done in another way.

The low-level performance is different from the real robot, in the sense that the dynamics of the robot miss some components. For example, the clamping springs and friction in the joints can be incorporated, but this will reduce the simulation speed and therefore the usability.

### 7-1-7 Traction controller

The velocity controller without gravity compensation but with controlled clamping torque (Exp. 3c) had approximately the same performance as the one without controlled clamping (Exp. 3a). Therefore, a controller based on the estimated friction coefficient can correctly determine what the proper joint torque should be, instead of just selecting a fixed joint torque.

In the noiseless case without gravity compensation, the robot first starts rolling before a wheel torque is applied, thus causing jitter. The gravity compensation therefore increase the performance in the noiseless case (Fig. 6-10b). However, in the case with sensor noise the robot is always seemingly moving, at least a little, so the gravity compensation has no added value and even deteriorates the performance (Fig. 6-10d).

## 7-2 Conclusions

The simulation model was usefull for developing the traction controller, but is not suitable for development of the high-level controller.

With a set of fourteen commands and a camera for localisation, the PIRATE can drive through a mitre bend autonomously at an angle up to 28°. This can be done either by an operator sending multiple Primitive commands or by sending a Sequence command once.

The robot shows a low slip ratio of up to 0.3 for driving through a pipe, both in the real setup as well as in simulation. It was shown in simulation that the traction control, for known estimated values of $\mu$, works. In this traction controller, clamping torque control and velocity control should be incorporated, but the gravity compensation should not.

## 7-3 Recommendations

### 7-3-1 ROS communication

The timing and type of ROS messages used in the software architecture should be investigated further. The speed of the control loops at each level should be tuned in such a way that a high level command will be translated and executed at low level in a shorter time. This way, delays and publishing topics twice, as mentioned in Section 4-2-2, is not needed anymore.

Furthermore, the blocking behaviour of ROS services, as mentioned in Section 4-2-4, should be investigated more throughly. The action library of ROS may be a solution, since this type of command gives updates during the execution, as well as when the execution of the command is finished. There are few examples to be found on how to use this library, therefore this should be investigated by a ROS expert. The goal would be to implement continuous a controller for velocity and torque.

When these blocking issues are solved, joints and wheels can be controlled simultaneously in a more structured way in the Movement node (Section 4-2-4). For example, when clamping the front part a single setpoint is sent to the combination of the first and second bending joint, where the controller alternates between updating the front joint and updating the rear joint within the time loop. A separate controller is now needed for the combination of e.g. all bending joints together, such that bending joint three and four are taken into account in the loop. This results in redundancy. Each joint should have an individual control loop, where another part of code in this node ensures that all joints gets updated with approximately the same rate.

The PAB node should not only know that its command has been processed, as discussed in Section 7-1-2, but also know that the corresponding action has been finished, or that this action has failed. For the position control this is done automatically, but for commands with open-loop voltage control, such as clamping, this is not the case. The PIRATE only knows that it has started clamping, but does not now when the front part is fully clamped. These kind of checks should be incorporated into the software architecture. This way the architecture will be more robust and execution of Sequence commands will become faster.

### 7-3-2 Simulation model

The simulation model should be split up in two simulation models, since there are two different goals. One simulation model should be used for further development of the autonomy, so for high-level controllers. This model has to be simple and as fast as possible, such that it can be used in learning. This model should be able to handle moving through a mitre bend or T-joint in simulation.

The other model should be used for the development of low-level controllers, and should therefore be more elaborate than it is now. The gearboxes, joint springs, friction in the joints, etc. should also be incorporated in this model in order to investigate the response to controllers.

### 7-3-3 Traction controller

In this thesis a start is made on the traction controller. This controller should be expanded by an online estimation of friction coefficient $\mu$, in order to determine the required clamping torques continuously, instead of offline estimation afterwards. The velocity control should also work when the sensor for the linear velocity is not continuously available. In order to do this more elaborate experiments should be done in simulation, with time-varying contact models that mimic a pipe with slippery parts.

Since the simulation model can only partially simulate the real robot, the controller should also be tested in a real life setup.

### 7-3-4 Wheel slip estimation

As mentioned in Section 7-1-5 the wheel torque may be wrongly estimated by using the wheel current. The relation between the current and the wheel torque should be investigated to ensure that a correct torque is measured.

### 7-3-5 Setup

As a trade-off between easy access to the PIRATE (Setups 1 and 3) and transferability to a real environment (Setup 2), half open pipes could be used in the setups, where the pipe is cut along the longitudinal axis. The bars do not have to be removed from the setups, since these bars could be used to clamp these pipe segments.

### 7-3-6 Experiments

Currently, only one orientation of the tilted mitre bend experiment (Exp. 2c) is used, where the robot drives from the horizontal pipe to an upper vertical pipe. To show that the robot can move through a mitre bend in all directions without falling, the following variations of Experiment 2c should also be performed:

- Horizontal pipe to lower vertical pipe.

- Upper vertical pipe to horizontal pipe.
- Lower vertical pipe to horizontal pipe.

The next step is to test the robot in real pipes. The commands for the rotational joint can then also be tested. Variations with T-joints instead of mitre bends, or different diameters should also be investigated to test the robustness of the system. The IMU data should also be incorporated in the control, for example to align the rotational joint with the horizontal plane.

# Appendix A

# Components

This chapter describes the components of the PIRATE, based on Dertien [2014] and on the code on the PICO boards.

For direct communication with the PIRATEbay (Fig. A-2a) the KORG nanoKONTROL2 MIDI panel (Fig.A-2e) is used.

Each PICO board (Fig. A-2c:A-2d) contains an ATmega328p microcontroller, an FXOS8700CQ compass with accelerometer and magnetometer, an A3906 H-bridge as motor driver, an LTC2850 transceiver for RS-485 communication, an LTM8020 regulator and DF57 connectors for precrimped wires. The PICO boards are daisy chained over the RS485 bus. The connections can be found in Table A-1.

Each bending joint is actuated by a Faulhaber 1016_006G micromotor ($3.0 \cdot 10^{-3}$ Nm/A) with a 10/1 gearhead (ratio 64:1, efficiency 70%), followed by a worm gear (ratio 24:1, efficiency 36%), a spring ($3.5 \cdot 10^{-3}$ Nm/°)) and another gear box (ratio 3.625:1, efficiency 80%). An overview is given in Figure A-1. AS5055 magnetic hall-effect sensors are used to measure the angle of the motor (between the first gear box and the worm gear) and of the joint itself (Fig. A-2b).

The rotational joint is actuated by a Faulhaber 1516_006SR micromotor ($4.15 \cdot 10^{-3}$ Nm/A) with a 15A gearhead (ratio 809:1, efficiency 62%). The angle is directly measured by an IE2-16 incremental encoder (16 ppr).

Each wheel is actuated by a Faulhaber 2619_006SR micromotor ($8.44 \cdot 10^{-3}$ Nm/A) with an internal gearbox (ratio 112:1, efficiency 59%). The angle is directly measured by the internal IE2-16 encoder (16 ppr).

**Table A-1:** Overview of the PICO connections with the motors and sensors, and the signals they provide. Note that some motors also provide sensor values for the current and/or encoder. The front module (*) is currently not attached, so no sensor values are taken into account. *F* is short for Faulhaber micromotor. *P* refers to the connection port on the PICO (Fig. A-2d). Adapted from Garza Morales [2016].

| **Module** | **PICO ID** | **Motor0 (P4)** | **Motor1 (P5)** | **Sensor0 (P7)** | **Sensor1 (P1)** |
|---|---|---|---|---|---|
| Front* | 20 | Tilt camera* | Pan camera* | Tilt camera* | Pan camera* |
| | 21 | F1016 front* | Front LED* | AS5055 joint front* | AS5055 spring front* |
| Bend I | 22 | F1016 1 $x_{bend,load}$ | F2619 1 $x_{wheel,angle}$ $x_{wheel,velocity}$ $x_{wheel,load}$ | AS5055 joint 1 $x_{bend,angle}$ | AS5055 spring 1 $x_{bend,spring}$ |
| Bend II | 23 | F1016 2 $x_{bend,load}$ | F2619 2 $x_{wheel,angle}$ $x_{wheel,velocity}$ $x_{wheel,load}$ | AS5055 joint 2 $x_{bend,angle}$ | AS5055 spring 2 $x_{bend,spring}$ |
| Rotation | 24 | (Not used) | F2619 3 $x_{wheel,angle}$ $x_{wheel,velocity}$ $x_{wheel,load}$ | IMU $x_{acc}$ $y_{acc}$ $z_{acc}$ $x_{mag}$ $y_{mag}$ $z_{mag}$ | - |
| | 25 | F1516 1 $x_{rotate,angle}$ $x_{rotate,load}$ | F2619 4 $x_{wheel,angle}$ $x_{wheel,velocity}$ $x_{wheel,load}$ | - | - |
| Bend III | 26 | F1016 3 $x_{bend,load}$ | F2619 5 $x_{wheel,angle}$ $x_{wheel,velocity}$ $x_{wheel,load}$ | AS5055 joint 3 $x_{bend,angle}$ | AS5055 spring 3 $x_{bend,spring}$ |
| Bend IV | 27 | F1016 4 $x_{bend,load}$ | F2619 6 $x_{wheel,angle}$ $x_{wheel,velocity}$ $x_{wheel,load}$ | AS5055 joint 4 $x_{bend,angle}$ | AS5055 spring 4 $x_{bend,spring}$ |
| Rear | 28 | (Not used) | Rear LED | (Not used) | (Not used) |

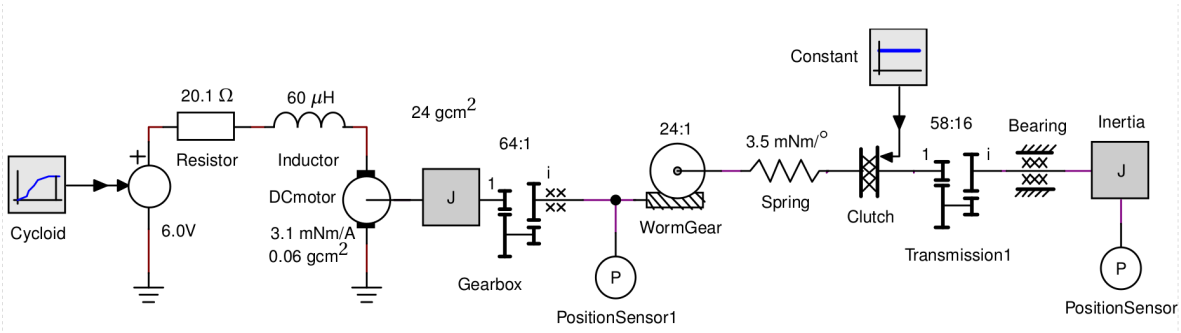
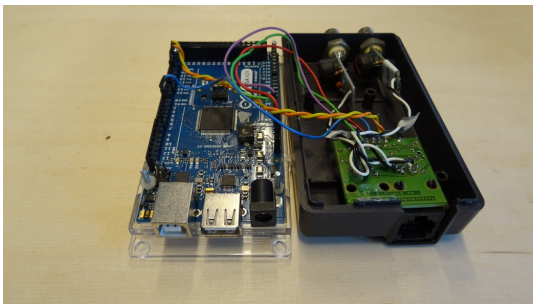
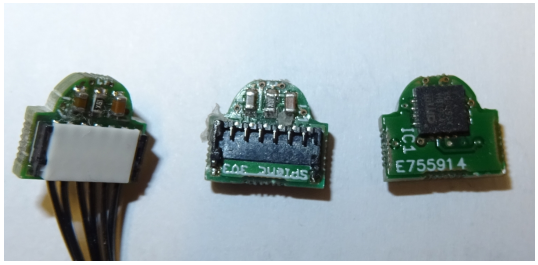


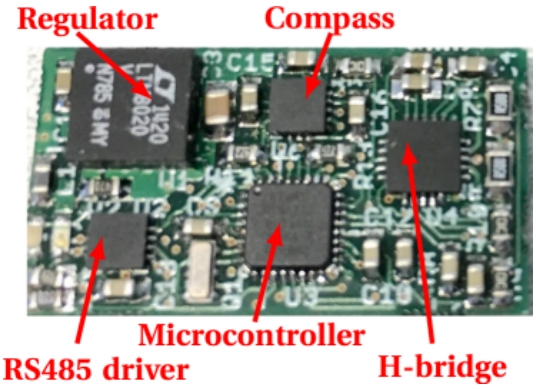

**Figure A-1:** The ideal physical model of the bend drive. *Permission granted by the author.* [Dertien, 2014]



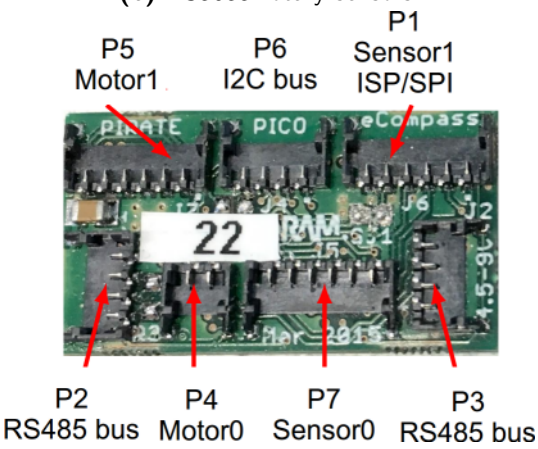
**(a)** The PIRATEbay (adapted Arduino MEGA).



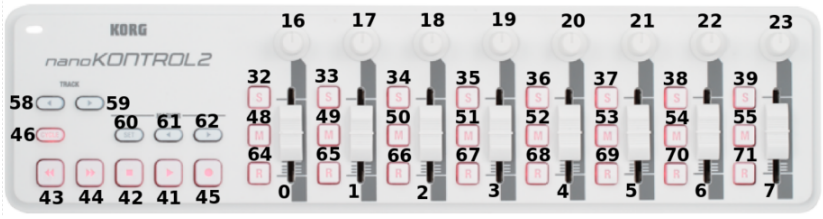
**(b)** AS5055 rotary sensors.




**(c)** A PICO board (front). [Garza Morales, 2016]




**(d)** A PICO board (back). Adapted from [Garza Morales, 2016]




**(e)** The MIDI panel [Garza Morales, 2016]. Sliders [0-6] set the PWM value for the bend and camera motors. Slider [7] sets the PWM value for all wheels. Potentiometers [16 and 22] set the LED PWM. Potentiometer [23] sets the rotational joint angle. PLAY [41] enables the motors, STOP [42] disables them. CYCLE [46] starts reading the sensors values. Buttons M [48:55] sets the PWM value for the corresponding motor to zero. SET [64] calibrates the sensors.

**Figure A-2:** Hardware components used in the PIRATE system.

# Appendix B

# Sensor conversion

In the Movement node the raw signals are converted to meaningfull signals with SI units. An overview of the implementation of the actuators and sensors in the Movement node is shown in Table B-1. Here all incoming sensor signals are structured in a Sensor class instance, while all outgoing actuation signals are structured in a Motor class instance. The software implementation therefore differs from the hardware implementation, as was shown in Table A-1.

The bend angle sensors provide Angle ($x_{\text{bend,angle}}$), Load ($x_{\text{bend,load}}$) and Torque ($x_{\text{bend,torque}}$) signals. A total rotation for the angle sensor covers 4096 pulses, so the angle $\gamma$ can be calculated with Equation B-1.

$$\gamma = \frac{2\pi}{4096} x_{\text{bend,angle}} \tag{B-1}$$

The current is provided in mA, so it is translated to A by a factor 0.001 as in Equation B-2.

$$I_\gamma = 1.0 \cdot 10^{-3} x_{\text{bend,load}} \tag{B-2}$$

The deflection of the spring in the bend joint is calculated by means of the bend angle sensor and the spring angle sensor, which is placed before the spring and some of the gears in the gear train. The spring angle sensor can make multiple revolutions, so to calculate what the bend joint angle $x^*_{\text{bend,angle}}$ would be without the spring but with the gears, the revolutions are taken into account as in Equation B-3.

$$x^*_{\text{bend,angle}} = SMA\left(\frac{x_{\text{bend,spring}} + 4096 rev_{\text{spring}}}{r_{\text{worm}} r_{\text{gearbox}}}, 8\right) \tag{B-3}$$

The bend angle sensor and spring angle sensor are placed in opposing orientations, so they have to be added to get the angular difference. The torque signal that comes out of the PICO's is therefore not a torque but an angular difference, calculated as in Equation B-4.

$$x_{\text{bend,torque}} = SMA\left(x_{\text{bend,angle}} + x^*_{\text{bend,angle}}, 8\right) \tag{B-4}$$

**Table B-1:** Structure of the PIRATE in the Movement node, separating the actuators and sensors. For each Sensor class instance the converted signals are shown. Starred (*) items are currently not available for the real robot.

| Module | PICO ID | Motors | Sensors |
|---|---|---|---|
| Front* | 20 | CameraMotor* | CameraSensor* |
| | | CameraMotor* | CameraSensor* |
| | 21 | BendMotor* | BendAngleSensor* |
| | | LedMotor* | |
| Bend I | 22 | BendMotor | BendAngleSensor $\gamma_1$ $I_{\gamma,1}$ $\tau_{\gamma,1}$ |
| | | DriveMotor | DriveSensor $\theta_1$ $\omega_1$ $\tau_{\theta,1}$ |
| Bend II | 23 | BendMotor | BendAngleSensor $\gamma_2$ $I_{\gamma,2}$ $\tau_{\gamma,2}$ |
| | | DriveMotor | DriveSensor $\theta_2$ $\omega_2$ $\tau_{\theta,2}$ |
| Rotation | 24 | - | IMU $\psi_{\text{roll}}$ $\psi_{\text{pitch}}$ |
| | | DriveMotor | DriveSensor $\theta_3$ $\omega_3$ $\tau_{\theta,3}$ |
| | 25 | RotateMotor | RotationAngleSensor $\phi_1$ $I_{\phi,1}$ |
| | | DriveMotor | DriveSensor $\theta_4$ $\omega_4$ $\tau_{\theta,4}$ |
| Bend III | 26 | BendMotor | BendAngleSensor $\gamma_3$ $I_{\gamma,3}$ $\tau_{\gamma,3}$ |
| | | DriveMotor | DriveSensor $\theta_5$ $\omega_5$ $\tau_{\theta,5}$ |
| Bend IV | 27 | BendMotor | BendAngleSensor $\gamma_4$ $I_{\gamma,4}$ $\tau_{\gamma,4}$ |
| | | DriveMotor | DriveSensor $\theta_6$ $\omega_6$ $\tau_{\theta,6}$ |
| Rear | 28 | - | BendAngleSensor* |
| | | LedMotor | - |

The bend torque $\tau_\gamma$ can then be calculated by taking into account the spring constant and the gear ratio, as seen in Equation B-6.

$$\tau_\gamma = k_{\text{spring}} r_{\text{gearbox}}^2 \frac{360}{4096} x_{\text{bend,torque}} \tag{B-5}$$

$$\tau_\gamma = 3.5 \cdot 10^{-3} \left(\frac{58}{16}\right)^2 \frac{360}{4096} x_{\text{bend,torque}} \tag{B-6}$$

The rotation angle sensor also provides Angle, Load and Torque signals. The angle $\phi$ is calculated by means of the pulses per revolution and the gear ratio, as shown in Equation B-7

$$\phi = \frac{2\pi}{R_{\text{ppr}} r_{\text{gearhead}}} x_{\text{rotate,angle}} = \frac{2\pi}{16 \cdot 809} x_{\text{rotate,angle}} \tag{B-7}$$

The current is translated in the same way as for the bend joint, as shown in Equation B-8.

$$I_\phi = 1.0 \cdot 10^{-3} \cdot x_{\text{rotate,load}} \tag{B-8}$$

The torque signal for the rotational joint is always zero, since this signal is not provided by the PICO.

For the wheels the signals Angle, Velocity and Load are provided. The angle is calculated as shown in Equations B-9.

$$\theta = \frac{2\pi}{R_{\text{ppr}} r_{\text{gearhead}}} x_{\text{wheel,angle}} = \frac{2\pi}{16 \cdot 112} x_{\text{wheel,angle}} \tag{B-9}$$

For the current it is done in the same way as for the bend and rotation joint, as shown in Equation B-10.

$$I_\theta = 1.0 \cdot 10^{-3} x_{\text{wheel,load}} \tag{B-10}$$

In the PICO code the Velocity signal is calculated by means of an exponential moving average filter on the difference between the current and previous wheel angle, with a weight of 0.5, as can be seen in Equation B-11. Therefore the signal $x_{\text{wheel,velocity}}$ is an angular difference, not an angular velocity. Time was not taken into account, therefore the $\frac{1}{\Delta t}$ factor is taken into account in postprocessing, as seen in Equation B-12. The angular velocity based on the angular difference signal needs a gain of 0.5 to result in approximately the same angular velocity based on the angle signal.

$$\begin{aligned} x_{\text{wheel,velocity}}(i) &= EMA\left(x_{\text{wheel,angle}}(i) - x_{wheel,angle}(i-1), 0.5\right) \\ &= 0.5 x_{\text{wheel,velocity}}(i-1) + 0.5\left(x_{\text{wheel,angle}}(i) - x_{\text{wheel,angle}}(i-1)\right) \end{aligned} \tag{B-11}$$

$$\omega = \begin{cases} 0.5 \dfrac{2\pi}{R_{\text{ppr}} r_{\text{gearhead}}} x_{\text{wheel,velocity}} \dfrac{1}{\Delta t} \\ \dfrac{2\pi}{R_{\text{ppr}} r_{\text{gearhead}}} \dfrac{\Delta x_{\text{wheel,angle}}}{\Delta t} \end{cases} \tag{B-12}$$

The IMU is currently not yet used in the control. The raw accelerometer and magnetometer values have not yet been investigated, but from the ratio between the accelerometer values the roll and pitch can be determined as in Equations B-13 and B-14.

$$\psi_{\text{roll}} = \text{atan}\left(\frac{-y_{\text{acc}}}{z_{\text{acc}}}\right) \tag{B-13}$$

$$\psi_{\text{pitch}} = \text{atan}\left(\frac{x_{\text{acc}}}{\sqrt{y_{\text{acc}}^2 + z_{\text{acc}}^2}}\right) \tag{B-14}$$

# Appendix C

# Computer vision

## C-1 Camera parameters

The relation between a point $\begin{bmatrix} u & v & 1 \end{bmatrix}^T$ on a 2D image and that same point $\begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$ in the 3D world can be described by a projective mapping (Eq.C-1) [Bradski, 2000].

$$z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K_{3\times3} \begin{bmatrix} R_{3\times3} & T3 \times 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{C-1}$$

$K$ denotes the intrinsic matrix, which describes the properties of the camera itself: the focal length $f$, the scale factors $m_x$ and $m_y$ relating the pixels to the distance, principal points $u$ and $v$, and the skew coefficient $\gamma$ (Eq.C-2). The extrinsic matrix describes the relation between the image plane and a plane in the 3D world, by means of a rotation matrix $R$ and translation vector $T$.

$$K = \begin{bmatrix} \alpha_x & \gamma & u_0 \\ 0 & \alpha_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} f \cdot m_x & \gamma & u_0 \\ 0 & f \cdot m_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{C-2}$$

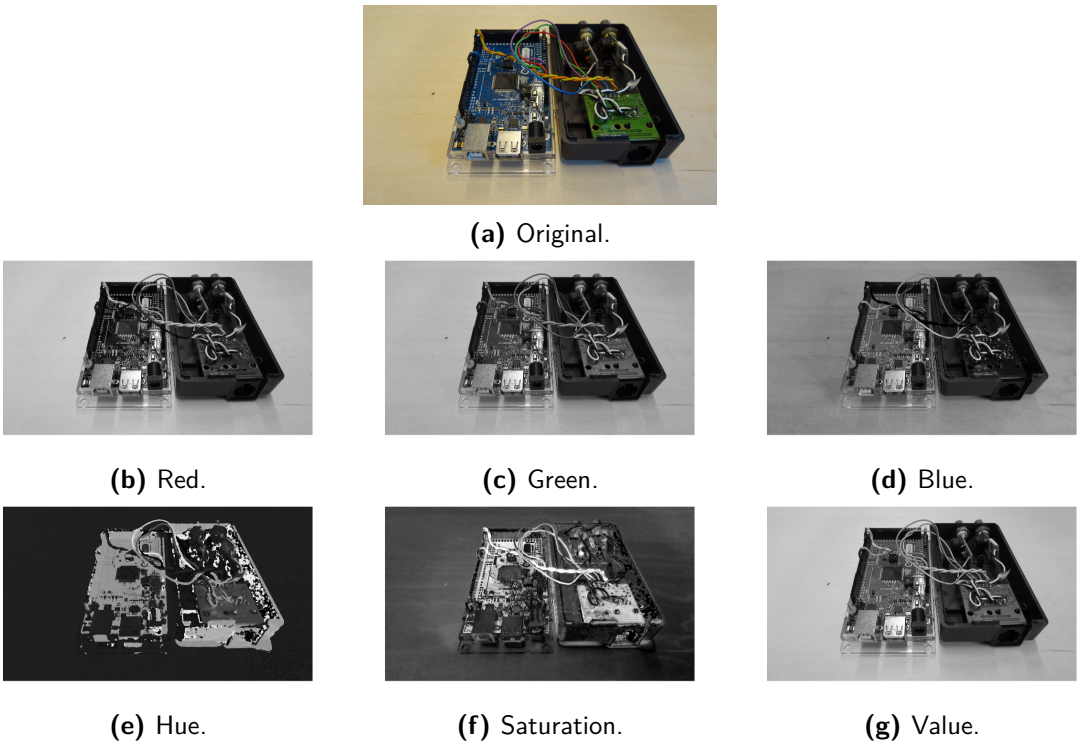
Distortion, as can be seen in a fisheye lens, can be described by distortion coefficients $k_1$, $k_2$, $k_3$, $p_1$ and $p_2$ (Eq. C-3).

$$\begin{bmatrix} x_{\text{corr}} \\ y_{\text{corr}} \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \Delta x_{\text{radial}} \\ \Delta y_{\text{radial}} \end{bmatrix} + \begin{bmatrix} \Delta x_{\text{tangential}} \\ \Delta y_{\text{tangential}} \end{bmatrix}$$
$$\begin{bmatrix} x_{\text{corr}} \\ y_{\text{corr}} \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \left(k_1 r^2 + k_2 r^4 + k_3 r^6\right) \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 2 \cdot p_1 \cdot x \cdot y + p_2 \cdot \left(r^2 + 2 \cdot x^2\right) \\ p_1 \cdot \left(r^2 + 2 \cdot y\right) + 2 \cdot p_2 \cdot x \cdot y \end{bmatrix} \tag{C-3}$$

To estimate the intrinsic matrix, first the image is set to gray scale. Snapshots must be taken of a chessboard pattern with known square size and amount, in as much orientations and positions as possible within the camera view. Using OpenCV methods the positions of these chessboards can be recognised and the intrinsic matrix and distortion coefficients can be optimised. With this matrix and these coefficients the image can be undistorted, as shown in Figure 5-3b.

## C-2 Marker detection

To detect markers with a known colour in an image, first the image needs to be translated from RGB (red-green-blue) to HSV (hue-saturation-value) colour space. *Saturation* describes the grayness, *value* describes the darkness and *hue* describes the range of pure colours without the effects of grayness or light and darkness. As can be seen in Figure C-1, conversion to HSV channels instead of RGB channels gives a clearer distinction between different parts of the object. The next step is to place a maximum and a minimum threshold on the H, S and V values, so only the marker is shown in the resulting image. The final step is to use a detection algorithm, like the Hough transform, to extract the coordinates of the marker in the image.



**(a)** Original.

**(b)** Red. **(c)** Green. **(d)** Blue.

**(e)** Hue. **(f)** Saturation. **(g)** Value.

**Figure C-1:** RGB to HSV colour conversion for a photo of the PIRATEbay, plotted as grayscale images with pixel values from 0 (black, low) to 1 (white, high). As can be seen in the subfigures, splitting the image into HSV channels instead of RGB channels shows more colour distinction between the various parts of the PIRATEbay, which makes recognition easier.

# Appendix D

# Quick start guide

## D-1 List of used software

In this thesis a HP EliteBook 8560w laptop with Ubuntu 16.04.5 LTS is used. The software tools used in this thesis are:

- Matlab/Simulink 2017b for the simulation, with the following toolboxes:
  - Simscape Multibody Contact Forces Library 4.1 [Miller, 2017]
  - Gaussian Processes Matlab Library (GPML) v4.2-2018-06-11 [Rasmussen and Nickisch, 2010]
- ROS Kinetic 1.12.13, to operate the PIRATE
- C++/GCC 5.5.0 compiler, to compile the ROS source files, with toolbox:
  - Open Source Computer Vision Library (OpenCV) 4.0.0 for image processing [Bradski, 2000]
- Doxygen 1.8.11 compiler for C++ documentation, with Doxywizard GUI
- Arduino 1.6.5 for compiling and uploading the PIRATEbay code (it does not compile in the latest Arduino version, 1.8.7)

## D-2 List of used hardware

The PIRATE hardware system includes:

- PIRATE
- Power adapter (PIRATEbay)

- MIDI panel
- PIRATEbay (Arduino Mega in black case)
- USB 2.0B to USB 2.0A cable (PIRATEbay to laptop)
- USB 2.0Mini-B to USB 2.0A cable (MIDI to PIRATEbay)
- Ethernet cable (PIRATEbay to PIRATE)

## D-3 Walkthrough

- Updating the ROS nodes:
  - Open a terminal and go to the PIRATE folder in the catkin workspace, probably `/catkin_ws/src/pirate`.
  - Enter `catkin build` to build all packages or `catkin build [package name]` for one package. Do not use `sudo`.
  - Repeat this at least once, until all dependencies are fixed and no errors occur.
- Updating the PIRATEbay:
  - Open a terminal and enter `rosrun rosserial_arduino make_libraries.py [arduino sketchbook location]/libraries`. This creates a folder named `ros_lib`.
  - Open `pirate_bay.ino` in Arduino and connect the Arduino Mega to your laptop.
  - Check the port and upload.
- Execute an experiment:
  - Connect all hardware.
  - Open a terminal and enter `roscore`, this initialises ROS. Keep this window open, but do not enter any commands here.
  - Open another terminal window and enter `roslaunch pirate_seq seq.launch`. The execution of this launch file causes all the nodes to start up and run their scripts, and also starts the recording of all messages. Keep this window open, since the print function of ROS sends information to this window.
  - Push the [Play] button on the MIDI panel to start the actuators. The red LEDs on the PICO boards should start blinking. If a red LED is lighted continuously, an error has occured at that motor and it can not be actuated. Pushing the [Stop] button and then the [Play] button sometimes solves this.
  - Push the [Set] button on the MIDI panel to start the sensors. The blue LEDs on the PICO boards should start blinking.
  - Check if the joint angles in the rViz visualisation correspond to the real joint angles.
  - Open another terminal window and enter a commands like `rosservice call /sk_pabSequence "pab_sequence: 0"` or `rosservice call /sk_primitive "mp_command: 0"`. As soon as the cursor returns the command has been executed and a new command can be entered.

  - To see the data returned by ROS, open a new window and enter `rqt` or `rostopic echo [name of the topic]`.
  - To see an overview of all nodes, open a new window and enter `rqt_graph`.

- Process the data:
  - Open a terminal and go to the folder where all rosbags are stored, probably `/catkin_ws/src/pirate/recordings`.
  - Enter `python bag_to_csv.py [rosbag name]` to convert the .bag file to a folder with a .csv file for each topic.
  - Open Matlab and enter `importdata(strcat('[folder name]','[topic .csv file name]'));`.

# References


Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

Dertien, E. C. (2014). *Design of an inspection robot for small diameter gas distribution mains*. PhD thesis, University of Twente, Enschede. https://www.ram.ewi.utwente.nl/aigaion/.

Drost, E. (2009). Measurement system for pipe profiling. Master's thesis, University of Twente, Enschede. https://www.ram.ewi.utwente.nl/aigaion/.

Fjerdingen, S. A., Liljebäck, P., and Transeth, A. A. (2009). A snake-like robot for internal inspection of complex pipe structures (piko). In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5665–5671. IEEE.

Garza Morales, G. A. (2016). Increasing the autonomy of the pipe inspection robot pirate. Master's thesis, University of Twente, Enschede. https://www.ram.ewi.utwente.nl/aigaion/.

Hansen, J., Murray-Smith, R., and Johansen, T. A. (2005). Nonparametric identification of linearizations and uncertainty using gaussian process models–application to robust wheel slip control. In *44th IEEE Conference on Decision and Control, 2005 and 2005 European Control Conference*, pages 5083–5088. IEEE.

Hoekstra, G. I. S. (2018). Towards a software architecture model for the automation of the pirate robot. Master's thesis, University of Twente, Enschede. https://www.ram.ewi.utwente.nl/aigaion/.

Junhui, L. and Jianqiang, W. (2010). Road surface condition detection based on road surface temperature and solar radiation. In *2010 International Conference on Computer, Mechatronics, Control and Electronic Engineering*, volume 4, pages 4–7.

Kakogawa, A., Komurasaki, Y., and Ma, S. (2017). Anisotropic shadow-based operation assistant for a pipeline-inspection robot using a single illuminator and camera. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1305–1310.

Kakogawa, A. and Ma, S. (2018). Design of a multilink-articulated wheeled pipeline inspection robot using only passive elastic joints. *Advanced Robotics*, 32(1):37–50.

Kim, J.-H., Sharma, G., Boudriga, N., and Iyengar, S. S. (2010). Spamms: A sensor-based pipeline autonomous monitoring and maintenance system. In *2010 Second International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–10. IEEE.

Lee, D., Park, J., Hyun, D., Yook, G., and Yang, H.-s. (2012). Novel mechanisms and simple locomotion strategies for an in-pipe robot that can inspect various pipe types. *Mechanism and Machine Theory*, 56:52–68.

Lee, D.-H., Moon, H., and Choi, H. R. (2011). Autonomous navigation of in-pipe working robot in unknown pipeline environment. In *2011 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1559–1564. IEEE.

Martinez Romero, A. (2014). Ros/concepts - ros wiki. http://wiki.ros.org/ROS/Concepts. [Online; accessed 17-09-2018].

Miller, S. (2017). Simscape multibody contact forces library. MATLAB Central File Exchange https://www.mathworks.com/matlabcentral/fileexchange/47417. [Online; accessed 05-06-2018].

Moghaddam, M. M. and Jerban, S. (2015). On the in-pipe inspection robots traversing through elbows. *International Journal of Robotics, Theory and Applications*, 4(2):19–27.

Pacejka, H. and Besselink, I. (1997). Magic formula tyre model with transient properties. *Vehicle system dynamics*, 27(Supp 001):234–249.

Park, J., Hyun, D., Cho, W.-H., Kim, T.-H., and Yang, H.-S. (2011). Normal-force control for an in-pipe robot according to the inclination of pipelines. *IEEE transactions on Industrial Electronics*, 58(12):5304–5310.

Pulles, C., Dertien, E. C., van de Pol, H., and Nispeling, R. (2008). Pirate, the development of an autonomous gas distribution system inspection robot.

Rasmussen, C. E. and Nickisch, H. (2010). Gaussian processes for machine learning (gpml) toolbox. *Journal of machine learning research*, 11:3011–3015. http://gaussianprocess.org/gpml/code/matlab/.

Rasmussen, C. E. and Williams, C. K. (2006). *Gaussian process for machine learning*. MIT press, Boston. ISBN 026218253X.

Reiling, M. (2014). Implementation of a monocular structured light vision system for pipe inspection robot pirate. Master's thesis, University of Twente, Enschede. https://www.ram.ewi.utwente.nl/aigaion/.

Smart tooling (2018). Smarttooling |. http://smarttooling.eu. [Online; accessed 21-09-2018].

Tucci, S. and Schlegel, C. (2017). Presentation of the robmosys project. https://robmosys.eu/download/. [Online; accessed 29-06-2018].