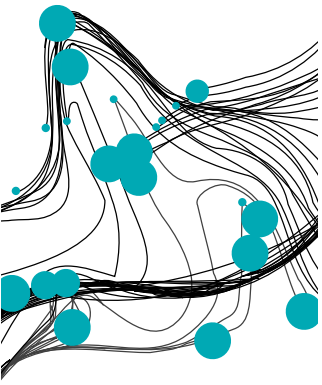
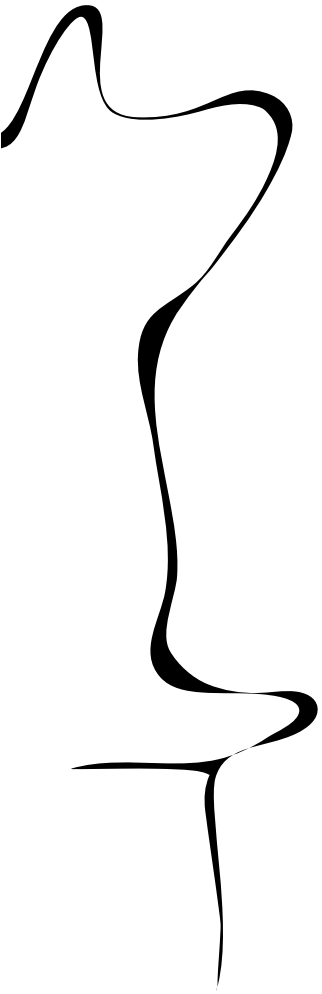


RAM ROBOTICS AND MECHATRONICS



AUTONOMOUS DRIVING THE PIRATE ROBOT

S.T.A. (Stefan) Tersteeg

MSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
N. Botteghi, MSc
dr. ir. E. Dertien
dr. M. Poel

January, 2020

004RaM2020
Robotics and Mechatronics
EEMCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Summary

Maintaining pipe systems is expensive because the pipes are usually hard to reach. The University of Twente develops, therefore, an autonomous driving robot named Pipe Inspection Robot for AuTonomous Exploration (PIRATE). This robot exists out of two v-shape modules with a rotational module in between, making the robot look like a snake. The goal is to lower the costs for maintenance by finding the leaks and damages in pipe networks. Previous efforts in this project have mainly focused on the mechanics and software communication of the robot. The last contributor implemented motion sequences to execute more advanced tasks like driving through a pipe bend. The primitive motions used in these sequences can be, for example, clamping, driving, or bending. The PIRATE robot uses the Robot Operating System(ROS) for high-level control.

This thesis aims at a better understanding of the surroundings at the Control level of the software architecture. The goal is that the sequences of motions execute with result information of each primitive motion. In this project, we design a preemptable task architecture to execute tasks with the PIRATE. If a motion primitive takes to long or did not complete successfully, the control layer can decide to re-execute the step or stop the sequence.

To improve the testing procedure, we will introduce a new localization method using ArUco markers. Previously blob detection was used. These markers are not only used to detect the camera position but will also locate the robot in its environment.

In this work, a design is proposed for the preemptable task architecture. A description of the implementation is given and experiments are created to test its functionality.

The experiments show that the localization of the robot is more robust and easier to set up, using the ArUco markers. Artifacts are almost reduced to zero, also is it possible to track multiple points.

Furthermore, the experiments show that the preemptable task architecture provides the progress information of executable tasks for the control level. The progress information improves the autonomy of the robot.

Future work should focus more on autonomous driving through real pipe networks and find suitable sensors to map the pipe network. When the PIRATE is fully able to travel autonomously through a network, it will be interesting to look into sensors to detect damages in pipe networks.

Abbreviations

CSV	Comma-separated values
FSM	Finite state machine
GCC	GNU Compiler Collection
GUI	Graphical user interface
MIDI	Musical Instrument Digital Interface
OS	Operating system
PAB	Partially autonomous behaviour
PICO	PIRATE control
PIRATE	Pipe Inspection Robot for AuTonomous Exploration
PIRATEbay	Arduino MEGA bringing the laptop with the PICO boards
PWM	Pulse width modulation
RobMoSys	Composable Models and Software for Robotic Systems
ROS	Robotic Operating System
RVIZ	ROS visualization

Contents

1 Introduction

1.1 Context

Maintaining distribution networks used for gas or water takes much time and is expensive. These networks are mainly underground and are, therefore, hard to reach by humans. Also, pipe networks a couple meters above the ground have a reachability problem. Smart tooling can be one of the solutions to reduce the costs of maintaining such a network. Faculty Robotics and Mechatronics (RaM) is therefor developing an autonomous snake-like pipe inspection robot: the "Pipe Inspection Robot for AuTonomous Exploration" (PIRATE). This robot consists of V-shaped sections that can clamp in pipes with diverse geometries. Sensors on board of the robot can test the condition of the pipe. Pulles (2008) made a first design of the PIRATE to decrease the costs of inspecting these networks by reducing the labor time, improving inspection quality, and reducing the repair costs.

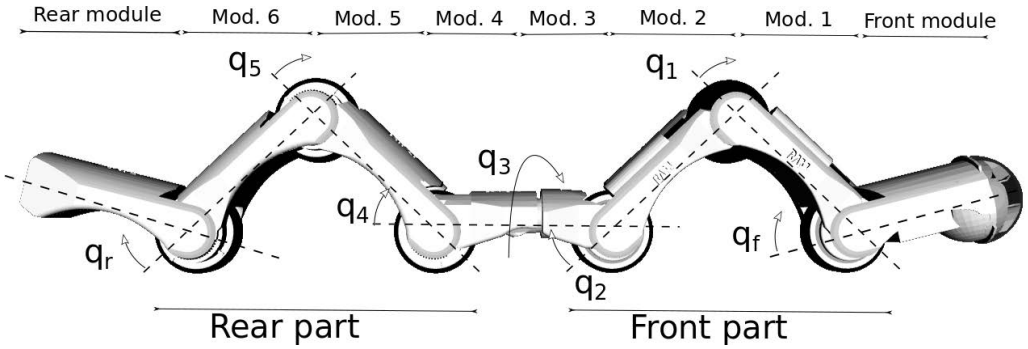


Figure 1.1: Kinematic model of the PIRATE. Image from Dertien (2014)

Figure 1.1 shows the design of the PIRATE; it has four bending modules; 1, 2, 5, and 6. Module three and four allow the robot to rotate. The front module is not used during this project because it will be redesigned with new sensors. The purpose of the front module is to localize the robot in its environment. The rear part contains an Ethernet port to connect the robot to an external computer. Previously a camera was placed in the rear module, but this camera is no longer functional. Led lights for the camera are also placed in this module; these are still functional.

1.2 Environment

The PIRATE robot is designed to drive through various pipe segments. Figure 1.3 shows examples of such pipe segments. Each of these segments has it's own challenges for the robot to drive through. The ultimate goal of the PIRATE is to drive autonomously through any given pipe network. Figure 1.3 contains the following segments; diameter change (a), curved joint (b), inclination (c), T-joint (d) and an uneven surface(e).

1.3 Problem statement

This project continues on the work of Geerlings (2018). The PIRATE robot that she used during her project had many shortcomings, mainly because the robot was too old. The old version of the robot is disassembled; the new version is reusing some parts. One of the main problems Geerling had was that cable connections were not reliable, for which reason the robot often executed unexpected behavior. The first step of this project is to reassemble a new version of the PIRATE.

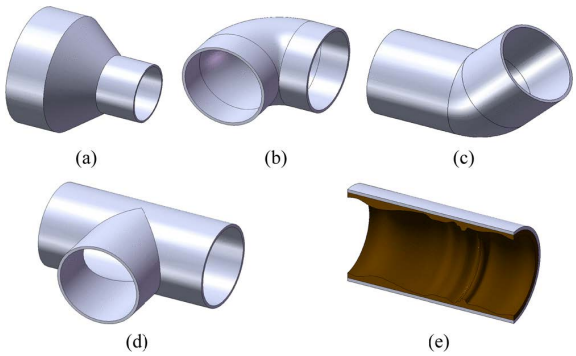


Figure 1.2: Several pipe segments often seen in pipe networks. Image from Park et al. (2011)

One of Geerlings recommendations was to implement feedback communication on actions that the robot has to execute. Geerlings (2018) implemented a new control layer using sequences of motion primitives. Examples of motion primitives are to clamp, unclamp, drive, bend, or rotate the robot. This control layer sends commands to the robot to execute but does not receive feedback on them. If one of the steps in a sequence fails to execute, the sequence continues executing.

Like Geerlings, we perform experiments to see if the robot can execute several tasks. To improve the quality and robustness of the experiments, we improve the available setup. We implement a new algorithm to detect the location of the robot with an external camera using ArUco markers. This new algorithm is more robust as it is less hindered by sunlight, other objects, and camera position changes. Also, this new algorithm cannot only locate the PIRATE but also give the orientation of the marker used.

1.4 Research questions

1. How can we improve the robot status representation for the operator?
2. What would be a suitable feedback communication protocol to have sufficient progress feedback in the high-level control layers?
3. What type of external localization algorithm can improve the accuracy, robustness, and usability of the localization of the PIRATE?

1.5 Related work

In this section, we discuss earlier work to get a better understanding of how to tackle the problems during this project. First, we look into accurate localization approaches, and secondly, we look in feedback communication implementations in robot architectures.

Geerlings (2018) did implement the first localization algorithm for the PIRATE robot. It did work but had some flaws. For example, the detection of the blob is easily disturbed by lighting or other orange circles. Also, changing the position of the camera results in incorrect location values. In 2015 Garrido (2015) introduced ArUco markers, which can be detected using cameras. The markers are unique, have a high detection rate, and the pose can be estimated using computer vision.

This approach was further improved by Ramirez (2018), resulting in an open-source software library. The project aims to maximize the speed of detecting the markers while preserving accuracy and robustness. Zheng (2018) used ArUco markers for the localization of another inspection robot. This paper shows that knowing static locations of ArUco markers can be used to calculate the camera position and also the location of other ArUco markers.

Hoekstra (2018) was the first in this project to use the open-source framework for robotics named Robot Operating System (ROS). Chapter 2 describes this framework in more detail. Primitive actions are created in this framework to operate the PIRATE robot. One of the shortcomings of the implementation is that there is no feedback implemented.

ActionLib is a ROS package that provides a standardized interface for interfacing with preemptable tasks in the ROS environment. Santos (2017) showed that ActionLib is a suitable way to implement non-real time controllers. "But its preemptive features allow almost periodic execution of the controller." Using the ActionLib was one of the recommendations of Geerlings (2018). ActionLib is widely used for action execution within ROS projects. The alternative is writing your own communication protocol for executing actions.

Geerlings (2018) did mention in her thesis the research project AIRo. The new version of the AIRo is 2.4. In this new version it is also able to drive through t-junctions using camera data detecting shadows in the pipe. Kakogawa (2019) describes the improvements on moving through a pipe network. It seems that this robot is able to drive trough a network, but does not map the area.

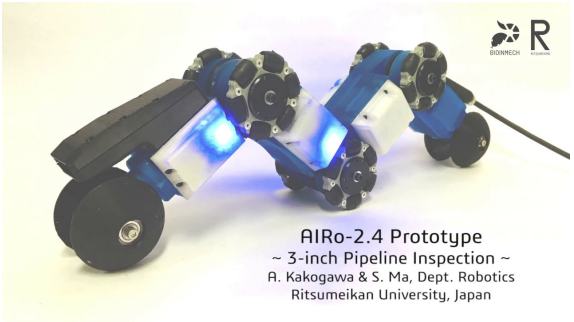


Figure 1.3: The AIRo 2.4. Image from Kakogawa (2019)

1.6 Approach

Firstly, we reassemble the PIRATE robot. Each sensor is tested, and all old cables are replaced to make sure each hardware component is working as it is supposed to do. During this process, the code will be used to debug and understand the communication between all those components in the system. Once all the components are working the robot is reassembled.

The next step is implementing an ArUco Marker detection system that detects the location of markers. If this system is sufficient for the project, it replaces the old vision implementation. A new experimental setup is created to test whether the PIRATE is working correctly. This setup is then used to re-execute the experiments of Geerlings (2018).

After that, we implement the feedback communication. This feedback will be used for waiting until a task is executed successfully. The new setup is used for the experiments and should prove the importance of feedback communication during execution.

The product of this project is a working PIRATE robot with improved location estimation and communication implementation. The software is fully documented. All the experiments are documented, and recommendations and conclusions are given based on these experiments. All software shall be accessible via Git, and a user manual is written on how to work with the robot. Also, an image (.iso file) is created of Linux OS used during this project, which should save time in the feature installing dependencies by new contributors.

1.7 Outline

Section 2 gives insight into the current state of the PIRATE. It will be in two parts; software and hardware. Section 3 describes the design of the software implementation improvements. Section 4 describes the implementation. Section 5 contains a walk trough of the experiments, and section 6 contains its results. Section 7 wraps up this project with conclusions and recommendations for future work. Lastly, the references for this document are given.

2 Background

In this chapter, we discuss the background of this project. In section 2.1, we give an overview of the hardware of the PIRATE. In section 2.2, we discuss its current software implementation.

2.1 The PIRATE Hardware

2.1.1 PIRATE

The PIRATE robot is a V-shaped snake-like robot that can drive through various sized pipes. The robot exists of eight modular parts, as shown in Fig 2.1. The front module contains a camera; at this moment, other researches are looking into LIDAR technology to implement in this module. Modules 1, 2, 5, and 6 enable the robot to clamp inside a pipe. Module 3 and 4 allow the robot to rotate the front and rear part and is called the rotational module. The rear module has an Ethernet port connector for wired communication to the PIRATEbay. The hardware was designed and developed by Reiling (2014) and improved by Garza Morales (2016). Each module contains one or two boards with a microcontroller named a PICOboard, and each board has its unique id. These boards control the torque of the joints and the speed of the wheels.

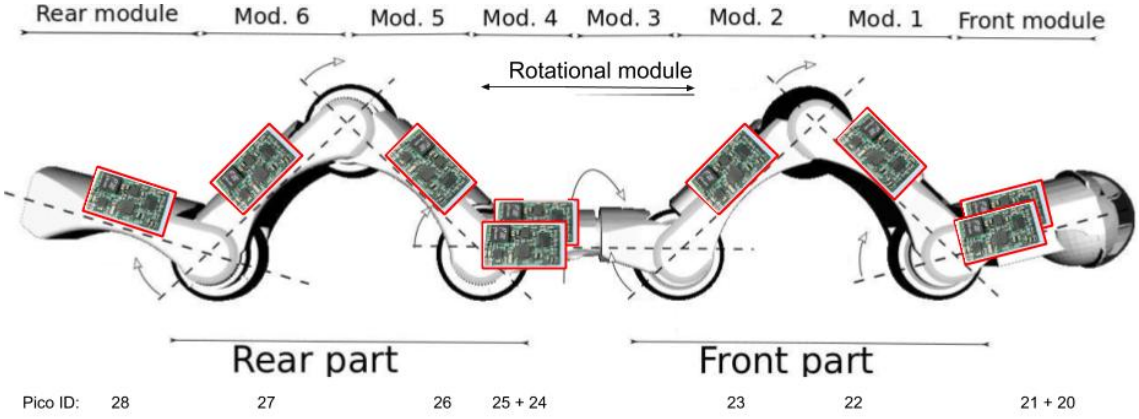


Figure 2.1: PIRATE robot with module names of the parts, the numbers of the modules and the indexes/locations of the PICO boards. Adapted from Dertien (2014)

2.1.2 PICOboard

The PICO board is a custom developed board by the faculty Robotics And Mechatronics(RAM) of the University of Twente. The board has a size of 15 x 27 mm and is shown in Fig 2.2. Each board contains an ATmega328p microcontroller. The board is powered by an LTM8020 voltage regulator and contains an RS485 transceiver (LTC2850) to communicate over the RS485 line to the master node, the PIRATEbay described in Section 2.1.4. To control the motors to bend and drive, each PICO board has an A3906 H-bridge. Lastly, each PICO board has a 6-axis compass(FXOS8700CQ) on board. DF57 connectors on the back of the PICOboard are used to connect all sensors, motors, and communication wires. The PICO boards are so-called daisy-chained, which means that each board has an input and output RS485 bus connection. In the rear and front module of the PIRATE, a voltage regulator is placed, which steps down 15V dc to 6V dc. The voltage regulator is powered using a cable that goes all the way through the robot to the front module. In appendix A, a guide is provided on how to upload code to a PICO board. Each PICO board also contains a red and blue led light. The blue led lights up when it is sending

information to the PIRATEbay. The red light can indicate multiple statuses, depending on the frequency.

1. 10101010 0xAA | fast blink: Ping timeout
2. 11001100 0xCC | normal blink: Communication error: rx/tx timeout, checksum, overflow
3. 11110000 0xF0 | slow blink: Paused/Idle
4. 11111111 0xFF | high: Serious error (encoder limit/ temp limit)
5. 10100000 0xA0 | heart beat: Normal operation

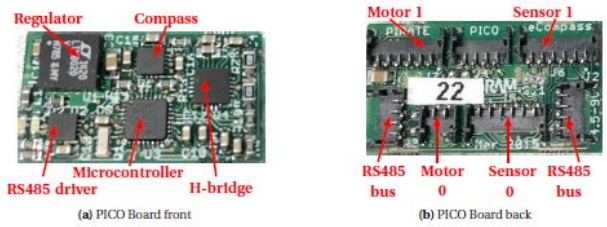


Figure 2.2: PICO board -front side (a) and back side (b). Adapted from Morales (2016)

Not all PICO boards contain the same sensors and motors. In table 2.1, the name, id, motors, and sensors are described for each PICO board. PICO board 20 and 21 are not used during this project because the front module is left out.

Table 2.1: PICO board table

PICO board attachments					
ID	Name	Motor0	Motor1	Sensor0	Sensor1
20	Front				
21	Front				
22	Bend I	Bend	Drive	Angle	Spring
23	Bend II	Bend	Drive	Angle	Spring
24	Rotational		Drive	IMU	
25	Rotational	Rotate	Drive	Angle	
26	Bend III	Bend	Drive	Angle	Spring
27	Bend IV	Bend	Drive	Angle	Spring
28	Rear		Lightning	Angle	

2.1.3 AS5055 Position sensor

For determining the angle of each bend in the robot, an AS5055 position sensor is used. In each bend, a magnet is placed, and this sensor detects the mechanic field orientation. Also, each bending motor has this sensor. In between the two measuring points, a spring is placed. Based on the difference in measurements, the force in the spring is calculated. This value is then used to determine whether the bend is clamped in a pipe. Figure 2.3 shows an AS5055.

2.1.4 PIRATEbay

An Arduino MEGA ADK serves as a station outside the pipe network to bridge the low-level control PICO boards and the high-level control PC, as shown in figure 2.4. The PIRATEbay has three input/output sources; nanoKONTROL2 MIDI panel (see section 2.1.5), RS485 communication,

Figure 2.3: AS5055 position sensor used in the PIRATE. Adapted from Morales (2016)

and lastly, the UART communication to the PC. The PIRATEbay uses the UART communication to the PC and the USB communication from the MIDI panel as input to send commands to the PICO boards over the RS485 communication. These messages contain the PWM values to set for the motors connected to a certain PICO board. The PICOboards will use the RS485 communication to give updates on the values of the sensors and motors to the PIRATEbay. The frequency of this communication is 10 Hz. An Ethernet cable is used between the PIRATEbay and the PIRATE robot itself. An Ethernet cable contains 8 internal cables; two are used for supplying the PIRATE 15V dc, and two are used for the RS485 communication. The other four were previously used for two cameras that were mounted on the front and rear module of the PIRATE, but are not used in this project. These can be used in the future for sending the data of the LIDAR camera(s).

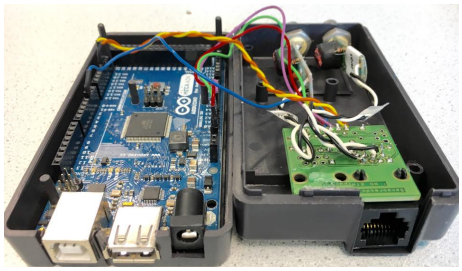


Figure 2.4: PIRATEbay containing an Arduino Mega ADK on the left and Ethernet port for RS485 communication on the right.

2.1.5 NanoKONTROL2 MIDI panel

The NanoKontrol2 board is directly connected to the PIRATEbay using a USB cable. This board is required to start the PIRATE robot. In figure 2.5, the board is shown, the numbers in this figure are the IDs that are sent to the PIRATEbay when events happen on the panel. The 'Play' button (41) is used to initialize all PICOboards and will change the status led on the PICO board from idle to heartbeat. The 'Cycle' button (46) can be used in heartbeat mode to set the output mode of each PICO board to 'on'. The blue led on each PICO board will now blink when it is sending its status update to the PIRATEbay. Sliders 1, 2, 3, and 4 are used to set the PWM value of the bend modules, and slider 7 is used to drive back or forward. A more in-depth guide on how to use the NanoKontrol2 is found in appendix B.

2.2 The PIRATE Software

2.2.1 Robot Operating System

The PIRATE robot uses the Robot Operating System(ROS) for high-level control. ROS exists out of a set of software libraries that can be used to build a robot. It is an open-source project and compatible with C++ and Python. ROS uses nodes that communicate with each other by using a publish-subscribe messaging pattern. A node can publish and subscribe to topics to

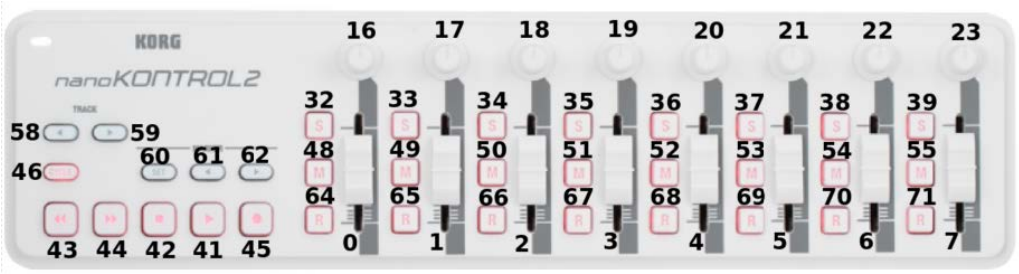


Figure 2.5: The MIDI panel. Adapted from Morales (2016)

send and receive messages. Each topic has its message type, always containing the same variables so that each message has the same structure. This message system is anonymous and asynchronous. One of the benefits of this system is that those messages can be recorded and replayed, recorded messages are stored in so-called ROS bags. If the implementation requires synchronized communication between nodes, so-called ROS services can be used. ROS can only be installed on Ubuntu Linux at the time of writing. In this project, we use ROS Kinetic 1.12.13, and all code is written in C++. Finally, ROS offers interfaces to read messages from topics and visualize the data in a Graphical User Interface(GUI). In this project, RVIZ is used for visualizing the status of each PICOboard. It shows a 3D model of the pose of the PIRATE. For this project, multiple ROS nodes are created; each one is responsible for its own task. For example, the vision node is reading the camera images and determines the location of the robot. Then the detector uses this x- and y- location to determine the environment state and which is used by the sequence node to determine the task to execute. Section 2.2.5 describes the full software architecture.

2.2.2 RobMoSys

Hoekstra (2018) implemented a layered structure on the base of Composable Models and Software for Robotic Systems(RobMoSys). The goal of RobMoSys is to make robotic systems more modular by separating control levels. The software is divided into various levels, which all have their own goals, as shown in figure 2.6.

By using this architecture, it is easier to make changes to the robot because each level is separated. As an example, our PIRATE would have a mission to explore the pipe network. A task could be moving through a bend, and clamping the robot would be a skill. One of the functions can be setting the PWM signal of a motor. The operating system contains out of the PICOboards, the PIRATEbay and the PC running ROS. Hardware will include motors and sensors attached to the PICO boards.

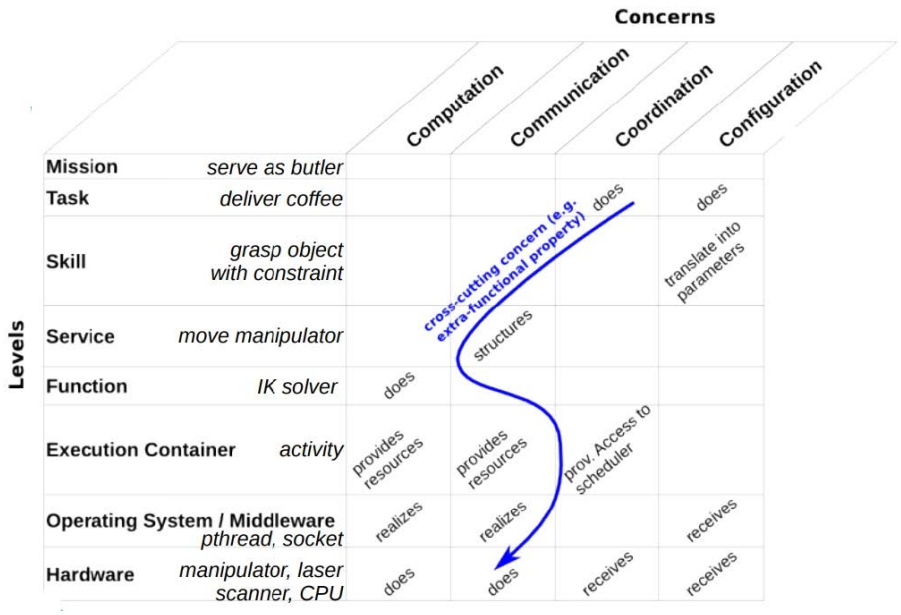


Figure 2.6: Separation of levels and separation of concerns, as formulated by RobMoSys.Image from Tucci and Schlegel (2017)

2.2.3 Finite state machines

Based on Hoekstra (2018) work Geerlings (2018) implemented a high-level control layer using motion primitives. Geerlings (2018) introduced Finite State Machines(FSM) to describe the state of the robot. The high-level control exists of five FSM's shown in Fig 2.7.

1. BendingState describes the bending of a bending joint.
2. FrontWheelsState describes the state of the front wheel.
3. RearWheelsState describes the state of the rear wheel.
4. RotationState describes the relative rotation of the front and rear parts.
5. LightingState indicates if the led light is turned on or off.

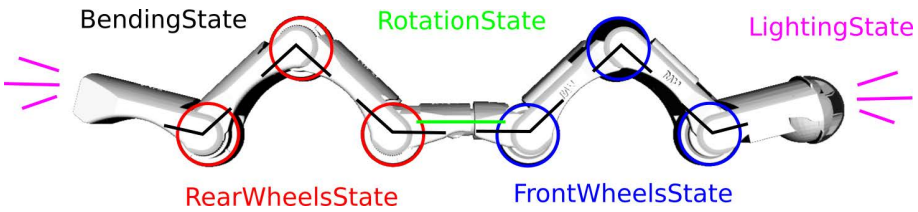
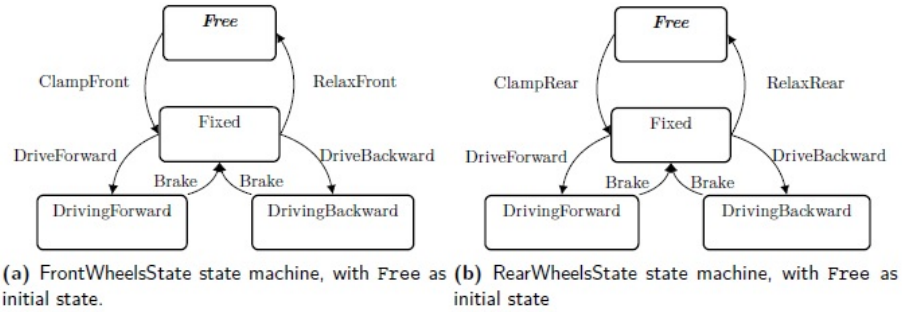


Figure 2.7: Overview of which actuators are affected by which state machine. Image from (Geerlings, 2018, p. 21)

Geerlings (2018) implemented this new control layer using motion primitives. According to Dertien (2014) a motion primitive is designed as "the smallest meaningful action that can be performed by the PIRATE robot". Geerlings implement the following services; clamp, unclamp, drive, bend, and rotate. Figure 2.8 shows the state machines of both wheel parts. In the squares, the states are given. The arrows represent the action that has to be taken to go to another state. The FSM's are implemented in the PAB (Partially Autonomous Behaviour) ROS node.

Execution of the motion primitives is performed in the Movement node. See section 2.2.5 for the structure.



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

Figure 2.8: State machines for the wheels. Image from N.M. (Geerlings, 2018, p. 22)

2.2.4 Control software

Figure 2.9 shows a commonly seen architecture used in robotic systems. This structure is based on Bennet (1988) and further modified by Broenink (2012). The physical robot is called the plant and displayed in the right blue part of the image. In the green on the left, the control software is shown. In the middle, there is a layer that transforms the different signals used in both domains. The control software exists out of five layers:

1. Loop control, this layer controls the actuators of the plant. These actuators require updated setpoints every sample period, and so this layer is hard real-time.

2. Sequence control, this layer is responsible for executing a certain task as a robot. This layer is sending setpoints to the control loop based on the task it has to execute. This layer is implemented in soft real-time.

3. Supervisory control, this layer determines the strategy of the robot based on the sensor input. Determining the strategy can cost a lot of computing power, for which reason this layer is implemented non-real-time.

4. Safety layer, this layer checks all signals that go to the hardware. Safety is important for all control layers, and thus, is it wrapped around all layers in the picture.

5. Measurement & Actuation, this layer is filtering and scaling all signals to the hardware. This is required as value ranges may differ.

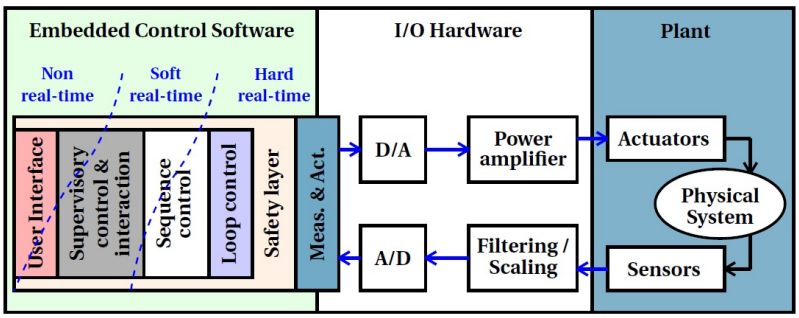


Figure 2.9: Layered software architecture for embedded systems. (Broenink, 2012, p. 339)

In the PIRATE project, the hard real-time loop control is implemented in the PICO boards and PIRATEbay. The sequential loop is implemented in a ROS node called 'pirate_mov'. The structure of ROS nodes is further discussed in section 2.2.5. In between those two layers, we have the PIRATEbay that handles the communication between the ROS environment and the PIRATE robot(plant). The setpoints can be adjusted by the sequential layer but also by hand using the NanoKONTROL2 described in section 2.1.5. The supervisory control is also implemented in the ROS environment and can be found in the pirate_det node. Lastly, the user interface is created in the pirate_vis node. The NanoKONTROL2 is also a user interface but directly communicates with the loop control layer. The actuators in the PIRATE are the motors that bend and drive the robot.

2.2.5 Software structure

In figure 2.10, the communication between the ROS topics is shown. The grey squares with the red text represent the ROS topics. The blue squares with black text represent hardware components, and the colored circles show which nodes implement the control layers mentioned in section 2.2.4. The frames recorded by the USB camera are used as input for the pirate_vision node. This node recognizes the ArUco markers and publishes their location, orientation, and id to the pirate_det node. The pirate_det node uses this information the determine the state of progress. The state is then sent to the pirate_seq node.

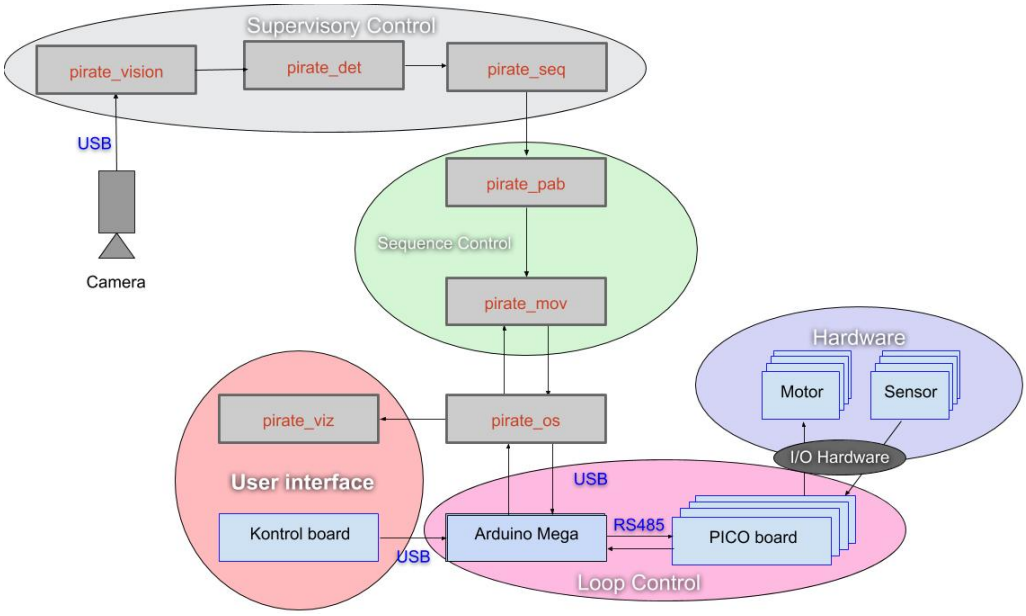


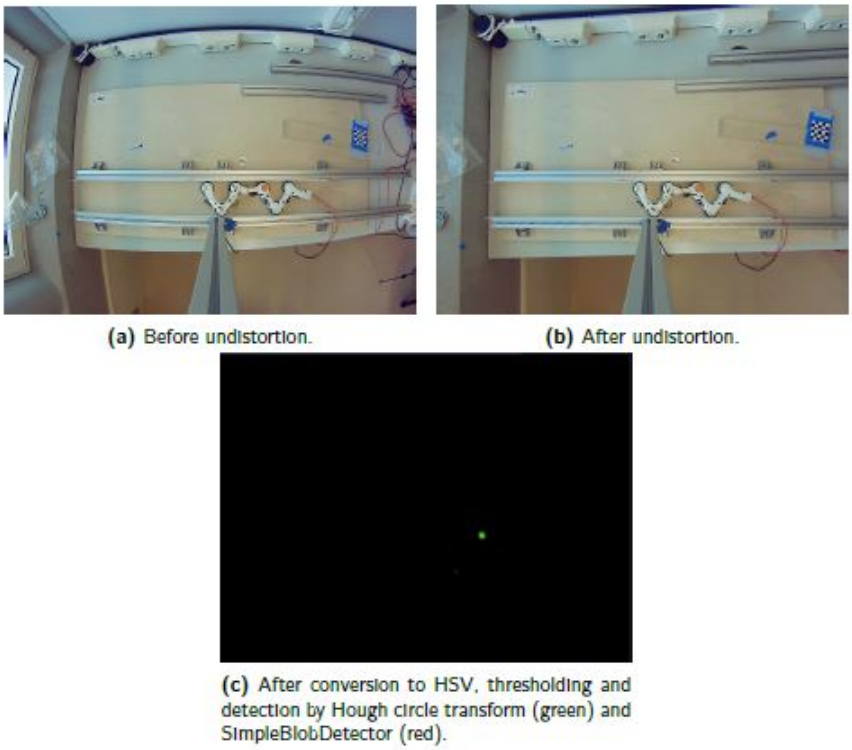
Figure 2.10: ROS Software architecture of the PIRATE robot.

Based on the state, this node starts executing sequences of tasks. Each task is sent as a command to the pirate_pab node. In this node, Geerlings (2018) implemented the Finite State Machines. Based on the current state, this node determines if a task can be executed. Examples of tasks can be clamping the front or back, or start driving forward. For each module that requires a change of execution, the Partially Autonomous Behaviour (PAB) node is sending a message to the pirate_mov node. This node then creates a command and sends it to the Arduino Mega. The pirate_os node is used as a communication node that implements a serial communication to be able to communicate with the Arduino Mega. The Arduino Mega features a loop control with a frequency of 10Hz. In each cycle, all the PICO boards are updated with the newest setpoints. Each PICO board also has its own control loop with a frequency of 500Hz. Every loop cycle is reading the sensor values and update the control signals to the motors. In between

the microcontroller on the PICOboard and the sensors/motors, we have I/O hardware to convert analog and digital signals back and forth. The setpoints in the Arduino Mega can also be adjusted by the Kontrolboard shown in the bottom left corner of the figure 2.10. Lastly, the pirate_viz node is using the PICO status values to make a visualization of the robot status.

2.2.6 Blob detection

Geerlings (2018) implemented a blob detection to estimate the position of the robot. First, an undistortion algorithm is applied to make straight things in the real-world also straight in an image. Next, the orange blob is detected by using the OpenCV library, which is also part of ROS. The orange blob on the PIRATE robot is then detected, and the pixel location is then converted to meters, see figure 2.11.



(a) Before undistortion.

(b) After undistortion.

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

Figure 2.11: Steps in the image processing. Image from (Geerlings, 2018, p. 50)

The steps mentioned above are implemented in the ROS node named "pirate_vision". This node sends the appropriate triggers to the Sequence Node, see figure 2.10.

3 Design

In this chapter, we describe the solution to tackle the shortcomings that the PIRATE currently has. First, we list the requirements, and based on these requirements; we present the design of each solution. In section 3.2, we describe the design of a localization method using ArUco markers. Section 3.3 shows the software design using the ActionLib library to create feedback communication between the low and high-level control levels.

3.1 Requirements

In this section, we list some requirements for the improvements we want to make. The requirements make sure that; a solution is an improvement to the project and that it is feasible to implement it during this research project.

1. Localization
   (a) The estimated position should have an accuracy that is not more than two centimeters off.
   (b) The estimated position should be calculated within 200ms so that it can be published at least 5 times per second.
   (c) The estimation algorithm should be robust so that external factors have as little influence possible.
   (d) The hardware components used should be cost-effective.
   (e) The sensors to estimate the position do not have to be on board of the robot.
   (f) The localization algorithm is just for testing purposes only. It does not have to work in a real pipe network.
2. Feedback communication
   (a) When executing a motion primitive, the commanding service should get progress information from the executing service.
   (b) The sensor values measured by the sensors on board of the PIRATE should be used to determine the progress.
   (c) The solution has to be implemented in the current ROS project.
   (d) The commanding service should be able to cancel a requested command.
   (e) The authoritative service should know whether a command did execute with or without success.
   (f) The feedback after executing the command should include the new states of the PIRATE robot.
3. Experimental setup
   (a) The price for the setup should be below 100 euros.
   (b) The setup should exist out of round pipes comparable to the ones used in a real pipe network.
   (c) The experimental setup uses multiple pipe diameters.
   (d) The setup should be easily movable from one place to another.
   (e) At least two of the pipe segments described in section 1.3 should be used in the setup.

(f) The setup should fit on the workbench in the RAM lab.

(g) Pipes used in this setup have a diameter of 125 millimeters.

4. Robot hardware

(a) The new 3d-printed parts need a stronger material than the PLA used in the previous version of the robot.

(b) New wires should be used to decrease the amount of loose connections between sensors/actuators and the PICO boards.

(c) All motors should be mounted to the robot such that they cannot move while turning.

(d) A different method than glue should be used to attach the gear to the motor shaft that rotates the rotational module.

3.2 ArUco markers

In this project, we replace the blob detection algorithm that Geerlings (2018) used to localize the robot. We replace the orange-blob by ArUco markers and use an algorithm to detect the index of the ArUco marker. Figure 3.1 shows a typical ArUco marker. The white/black square pattern differ to make unique markers; each pattern has its unique id.

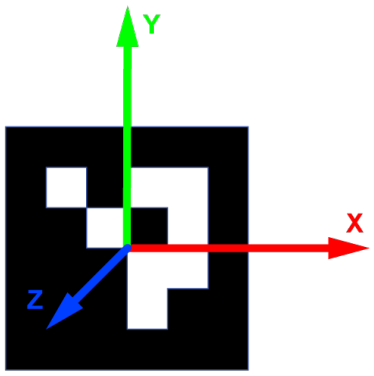


Figure 3.1: An ArUco marker with three axis orientation and its unique id.

The ArUco markers are used to solve two problems. First, we calculate the position of the camera with respect to our setup. We use four ArUco markers in a square; they have a fixed location in our setup. We can then calculate the perspective transform that can transform the x- and y- position in a picture to a real-world location. For processing the images and detecting the markers, we use the OpenCV library by Bradski (2000).

The second use of the ArUco markers is localizing the robot itself. We placed an ArUco marker on the PIRATE. For each marker detected on the webcam stream, the screen location is transformed into a real-world position in centimeters using the perspective transform obtained previously.

As recommended by Geerlings (2018), we cut the pipes such that we have half open pipes. By removing the top part of a horizontal pipe, we can see the robot with a camera that hangs above our setup, the same way as in Geerlings experimental setup. We place an ArUco marker on the PIRATE directed to the camera.

3.3 ActionLib

The ActionLib provided by ROS will is to implement the feedback communication within the PIRATE. Our goal is that we can command a motion primitive, like clamping the front module, in the sequence node, and get feedback after finishing it. The PAB node receives this command and starts executing the command. While executing, it then sends feedback messages to the sequence node so that this node knows the progress. During the execution of the command, the sequence node can cancel it. After completion of the execution, a final message is sent by the PAB to confirm that the command executed successfully.

In this design, the service that requests the command is called the client. In our case, it is the sequence node. The service executing the command is the server; in our case, the PAB node. The command to execute is called the goal and can be any motion primitive, for example, clamping the front. The execution of a goal consists of multiple states. In figure 3.2 the possible states for the client are shown in the boxes. The arrows show the state transitions possible for each state, and the color of the text indicates the service that can trigger the transition.

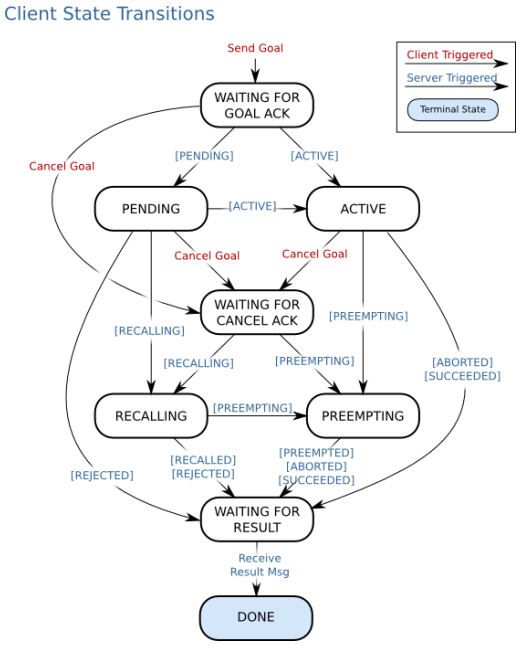


Figure 3.2: Client state transitions for a tasks.

Figure 3.3 shows the communication between our client and server. First, the client sends the goal to the server. It is possible to cancel the goal if needed. The status message is used to notify clients of the status of each goal. While executing a goal, the server also sends feedback that includes the progress of execution. When the goal is completed, with success or not, the server sends one last time the result of that goal.

3.4 Feedback information GUI

Because we use the feedback information from the PICOboards to determine the progress of motion primitive, we create a new graphical user interface(GUI) for the pirate_viz node. A program called RVIZ can display this interface. Currently, it only contains a 3D model of the PIRATE. This 3D view does show how each module is bent, whether the wheels are turning and, if so, the speed and direction. The new interface should display the raw values coming from the PICO board and show how old the data is. This way, it is much easier for the operator

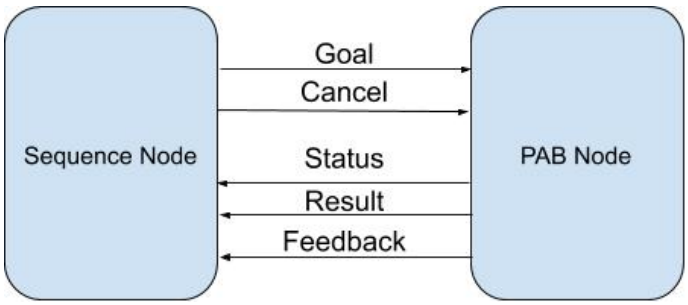


Figure 3.3: Communication between the client and server.

to see when a PICO is not working correctly, or data is not received. This interface does not add any functionality to the robot, but it allows for easier maintenance.

3.5 Coding standards

To improve the readability and maintainability of the code, we introduce some new code standards. Because the PIRATE uses much code, and researchers are normally only working for half of a year on the project, it is important to keep the code clean. In the code running on the PIRATEbay, for all id's of the button/switches operated on the NanoKontrol board, we introduce constant names. The ROS environment already uses this coding standard. All constant words are written with capital letters, and the underscore is used to separating these words. As an example: "KB_BUTTON_BACKWARD", where KB is an abbreviation of KontrolBoard.

Another coding standard introduced is removing multiple if-statements within each other, which the sequence node is often using. As a sequence has various steps, and each step has to be successful to continue to the next step, this is creating a big code tree. To make the code more readable, we use return statements. If steps are not successful, the return function ends the sequence.

3.6 Hardware

Several small hardware improvements are made during this project to improve the robustness of the PIRATE robot. A more robust robot should improve the efficiency of researching with the PIRATE. As stated by Geerlings (2018), a major problem while researching with the robot was that it sometimes starts executing unexpected behavior. Also, broken parts in the robot make it harder to perform successful experiments. In the following sections, we describe the improvements in several hardware parts of the robot.

3.6.1 3D printed wheels

Previously the wheels of the PIRATE were 3D printed with PLA, a cheap and commonly-used material for 3D printing. One of the major disadvantages of PLA is that it is fragile. Several wheel mounts of the PIRATE did break. We replace those wheels with the same design but printed with carbon fiber. This material is stronger than the PLA and should not break like the previous wheels.

3.6.2 Motor gear glue

Previously, the motor gears were mounted on the motor shafts using two-component adhesive. This worked fine for the gears used in the wheel, but it broke multiple times for the rotating module. We fix this by drilling a screw hole in the gear and mount it to the motor shaft with a screw. This mechanism is often used and much stronger than using glue. Also, it is possible to remove the gear if needed. To make sure the motor shaft doesn't turn inside the motor gear,

a small part of the shaft is drilled away, creating a flat surface to mount the screw on. This solution is shown in figure 3.4, where the blue circle, in the middle, represents the motor shaft. As shown in the figure, the shaft has now a flat surface at the top.

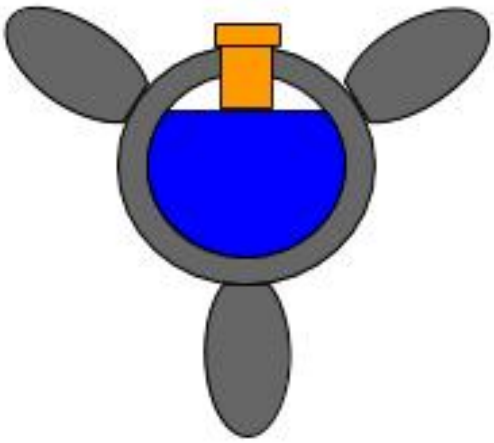


Figure 3.4: Mounting mechanism of the motor gear(grey) to the motor shaft(blue), using a screw(orange) to mount it.

3.6.3 DF57 Wiring

While using the PIRATE robot, the modules are moving, and these movements sometimes disconnect the cables in the robot. Disconnection can results in distortion in communication between the PICOboard and PIRATEbay, incorrect sensor values, or PICOboards that restart because of the power loss. Due to the unavailability of alternative connectors, we decided to replace the old DF57 cables with new ones. The recurring problem is that the metal connector is bent in the wrong direction resulting in loose connections. New cables should solve the problem for now.

4 Implementation

In this chapter, we discuss the improvements and changes we made to the PIRATE. First, we discuss the changes to the software on how we did implement the feedback communication and ArUco markers. Then, we list all the improvements to the hardware of the PIRATE.

4.1 Feedback communication

The feedback communication is implemented between the Sequence node and the PAB node, as described in the design chapter (3). The following sections describe the changes to both nodes.

4.1.1 Action Messages

For using the ActionLib, we first have to define the action messages. This is done by creating an ".action" file. This file defines our goal, feedback, and goal result. The goal will only exists out of the id of the primitive to execute. The feedback contains the percentage of execution, and the results include all states of the robot. The definition can be found in algorithm 1.

```
#goal definition
int8 mp_command
---
#result definition
int32 progress
---
#feedback
int32 frontwheels_state
int32 rearwheels_state
int32 bending_state
int32 rotation_state
int32 lighting_state
```

Algorithm 1: Action messages definition in "Primitive.action" file

Similar to the .msg and .srv files, Catkin makes a header file based on the .action files. The ActionLib uses these files to communicate all the messages between the server and the client. The CMakeList.txt file in the PAB node is adjusted to tell Catkin to auto-generate these messages. Catkin generates the following header(.h) files; PrimitiveActionFeedback.h, PrimitiveAction.h, PrimitiveFeedback.h, PrimitiveResult.h, PrimitiveActionGoal.h, PrimitiveActionResult.h and PrimitiveGoal.h.

4.1.2 PAB node

The PAB node in our project is executing the goal and is, therefore, the ActionLib server. When it receives an action to execute, the action contains the motion primitive command id to execute the goal. First, it will check whether it is possible to execute this action. If it is not possible, it is returning with a failed state. If the action is already executed, it is returning directly with success. For example, when the action is to clamp the robot, and this is already executed. This returns with success. Lastly, if the action is executable, it starts executing the command as long as the goal is not reached or canceled. Note that the client is canceling the goal when the time limit is reached. After executing the action, depending on whether the goal has been reached, the function returns with success or failure. The pseudo-code of this code can be found in algorithm 2.

In this project, we are using use the SimpleActionServer class provided by ActionLib. This class can only execute one goal at the time. If an action is assigned before the previous action is executed, the class is preempting the current action and starts executing the new one.

Data: Sequence step to execute
Result: Flag indicating if execution was successful
if *step is not executable* **then**
    print message telling step is not executable;
    return execution failed/false;
end
if *step is already executed* **then**
    print message telling step is already executed;
    return execution success/true;
end
send movement command;
while *Sensor goal value is not reached and goal is not cancelled* **do**
    sleep 200 milliseconds;
end
if *sensor goal value reached* **then**
    print message telling step executed successfully;
    return execution successful/true;
end
return execution failed/false;

Algorithm 2: ActionLib pseudo code server side in PAB node.

4.1.3 Sequence node

The client-side of the ActionLib is implemented in the Sequence node. Based on the status of the robot, the client is executing sequences to drive, for example, through a bend. For each step in the sequence, the client is creating a goal object. It then sends it to the PAB. Each command has a timeout duration. If the action times out, the sequence ends, and a message notifies the operator that it did fail to execute the step. The sequence does also stops if the action failed to execute. When the step has been executed with success, then the sequence continues with the following step until no steps are left. When all steps have been executed with success, the sequence has executed with success. The pseudo-code for the Sequence node can be found in algorithm 3.

```
Data: Sequence to execute
Result: Flag indicating if execution was successful
initialization of goal;
for all sequence steps do
    set goal index;
    send goal to server;
    if timeout within X seconds while waiting for goal result then
        print message telling sequence step timed out;
        return sequence failed/false;
    else
        if action successful then
            continue next sequence step;
        else
            print message telling sequence step failed;
            return sequence failed/false;
        end
    end
end
return sequence succeeded/true;
```

Algorithm 3: ActionLib pseudo code client side in Sequence node.

4.2 ArUco marker

The new external localization algorithm is using ArUco markers to determine the location of the PIRATE using an external camera. This algorithm is using the same camera as the blob detection of Geerlings (2018). We create a new ROS node that replaces the ComputerVision node. When initializing the node, it selects the USB port to get the webcam data. If this is successful, it starts a loop that runs as long the node is not terminated. In this loop, it grabs a webcam frame and processes it.

4.2.1 Undistortion

For each frame, the algorithm first undistorts the image. This makes straight lines in the real world also appear straight in the image. A chessboard pattern was used to obtain the parameters to undistort the image. The C++ OpenCV library Bradski (2000) provided the algorithm used for this.

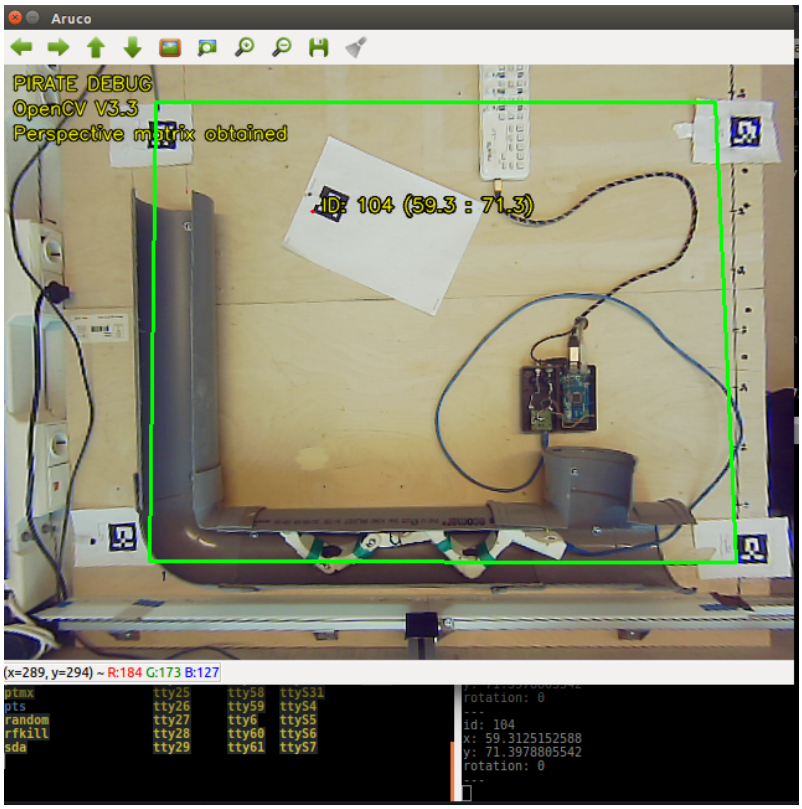


Figure 4.1: ROS interface displaying video stream and id and location of ArUco marker.

4.2.2 Perspective transform

The undistorted frame is used as input to detect the ArUco markers. We using ArUco markers with id 100, 101, 102, and 103 for calculating perspective transform. This transform is calculated based on static real-world coordinates and the pixel locations of each marker in the frame. Each frame is displayed in a window on the computer. To calculate the transform, all four markers should be visible on the frame, and the operator has to press "1" to calculate it.

4.2.3 Location estimation

When the perspective transform is obtained, the algorithm is publishing all detected ArUco markers. For each ArUco it publishes its real-world x- and y-coordinates, the orientation, and

the id. The detector node is using this data to determine the status of the robot. Figure 4.1 shows the interface displaying the detected ArUco marker with id 104. The green square displays the grid. The right bottom corner is the origin (0,0).

In algorithm 4 the pseudo-code for processing a video frame is given. The function does not return anything but is publishing the detected ArUco markers. To calculate and publish the real-world coordinates, the perspective matrix is required. This matrix is determined when all four matrix markers are identified, and the user presses the '1' button on the keyboard.

```
Data: Webcam frame
undistort frame;
detect markers;
for all detected ArUco markers do
    draw marker on frame;
    if perspective matrix is calculated then
        publish marker;
    end
end
if perspective matrix is not calculated and '1' is pressed and 4 markers are detected then
    calculate perspective matrix;
end
display frame in GUI;
```

Algorithm 4: ArUco marker pseudo code processing a video frame.

4.3 Feedback information GUI

A Graphical User Interface was created to show the raw values that each PICOboard sends to the ROS environment. Figure 4.2 shows this interface on the left side, and it contains the status of PICOboard 20-28. If a certain PICOboard did not send any update message in the past half of a second, the status is displayed Offline in red, otherwise Online in green. The duration since the last update is shown behind the status is seconds.

The character abbreviations [O, V, L, A, L, T, U, V, W] imply; Odometery, Velocity, Driveload, Angle, Bendload, Torque, IMU U, IMU V, and IMU W, respectively. For PICOboard 24 we don't have an Angle, Bendload and Torque value but the IMU x, y, and z value.

In the figure, we see that PICOboard 20, 21, and 28 are offline because they are left out of this project. This is also the reason why these modules in the 3d view are straight to the module they are connected to.

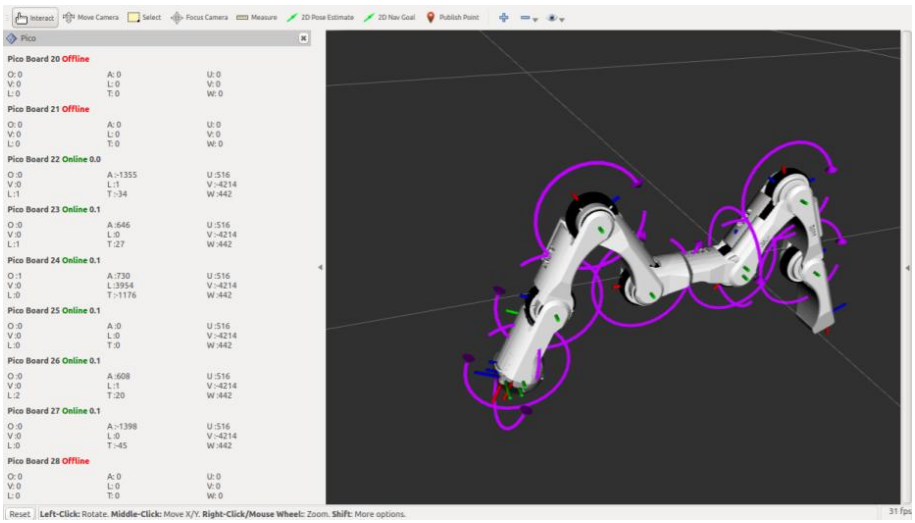


Figure 4.2: GUI in RVIS containing the 3d view on the right and the PICO status panel on the left.

5 Experiments

In this chapter, we perform experiments to see the impact of the improvements made during this project. First, we look into the ArUco marker and see how it performs compared to the blob detection algorithm. Next, we test the feedback communication system and see how decisions in the high-level control layer are made based on the feedback. Finally, we test the PIRATE robot and see how it performs while driving through a mitre bend.

5.1 ArUco markers

To see how the ArUco marker localization performs compared to the blob detection system, we test the speed, accuracy, lightning influence, and artifacts occurrence. For this experiment, we use the ELP-USBFHD06H USB camera with 1080p resolution and HD H.264 encoding. This camera also has a fish-eye lens such that we have a wider angle. The image processing is running on an old Toshiba Satellite L850-150 laptop. As newer laptops perform better than this old one, acceptable results in this experiment should also guarantee that upcoming researchers with different laptops can use this setup.

5.1.1 Speed

For each frame, the algorithm detects all the visible ArUco markers. To detect the speed, we measure the time that it takes to detect the ArUco markers. To do this, we get the time in milliseconds at each frame. We then undistort the frame and execute the ArUco detection algorithm. After that, we compare the time and determine the execution time. Because identifying a marker is part of the execution, we compare the performance using 4, 8, 12, and 16 markers in one frame.

5.1.2 Accuracy

The localization accuracy is tested by measuring the offset between the real location and the estimated location of markers in the setup. The accuracy depends on the ArUco detection, the perspective matrix estimation, and webcam quality.

5.1.3 Lightning influence

Because light intensity influences the detected pixel colors in a frame, we test the performance of the detection algorithm with a low and high light intensity. Because this setup is only used for testing, we test light intensities that we often see in the RAM lab. Here the intensity is only influenced by sunlight, so we compare the performance when it is cloudy and when it is sunny. For this experiment, we move the PIRATE 30 centimeters on normal speed with the ArUco marker on top of the robot. We compare how many frames the marker was detected and how many times it did fail.

5.1.4 Artifacts

The experiment to see the influence of lightning will also be used to see how many artifacts are observed while driving the robot. We calculate how many times the identification of a marker is wrong and how often a non-existing marker is detected.

5.2 Feedback communication

In this experiment, we test the performance of the feedback communication. We use the clamping operation of the robot to see how the sensor readings improve high-level control. While executing this action, we use ROS bags to log the sensor readings, control layer commands, and the PWM setpoints for the actuators. All values logged are used to create a graph that shows these values over time. We then compare the time that is lost between finishing and starting actions.

The goal of this experiment is to show that by using the feedback communication implemented with ActionLib we can wait until a task is executed and then start the next task. The PAB node returns a success message when a specific torque value is reached in the bend. We start the front module clamping in this experiment, wait until it is clamped, and then start clamping the rear. All data that is collected in these experiments are obtained from the ROS topics. This means that the timing of this data is relevant to the ROS environment. It takes some extra time to send it to the PIRATEbay and from the PIRATEbay to the PICOboards to eventually being sent to the actuators. Of course, also the timing of the sensor values are a little delayed for the same reason.

5.3 Moving through a bend

Like Geerlings (2018), we test how the PIRATE performs driving through a bend. A new setup is created using PVC pipes, with the top part removed to be able to locate the robot. Figure 5.1 shows the new setup. On the right side, the main computer is located. At the bottom, we see the PIRATEbay and the KontrolBoard. In the top left, we see the camera. Lastly, the pipe network is in the middle with the PIRATE clamped in it. The location of the PIRATE in the picture is the starting place to begin the experiment. The T-junction in the network is not used during this project.

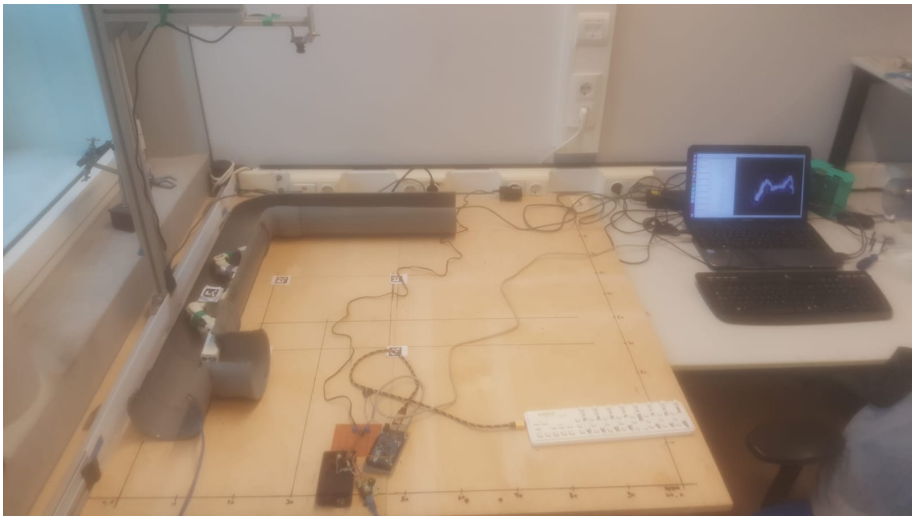


Figure 5.1: Setup used for experiment moving through a bend.

For this experiment, the operator only has to send the Sequence command to start the clamping of the robot. The localization algorithm detects when the PIRATE is reaching the bend and executes the sequence to drive through it. Driving through the bend is executed with the following sequences. Each sequence has multiple motion primitives that use the feedback communication to give feedback on the progress.

1. Enter pipe sequence started by the operator
    (a) Clamp front
    (b) Clamp rear
    (c) Drive forward
2. Reaching bend detected by webcam
    (a) Brake
    (b) Unclamp front
    (c) Bend front part
    (d) Drive forward
3. Reaching halfway bend detected by webcam
    (a) Brake
    (b) Unclamp rear
    (c) Clamp front
    (d) Bend back part
    (e) Drive forward
4. Leaving bend detected by webcam
    (a) Brake
    (b) Clamp rear

6 Results

In this chapter, the results of the experiments are listed. First, the results of the experiments of the ArUco marker location algorithms are given. Next, we provide the results of the ActionLib experiment. Lastly, the performance of driving through a bend experiment is provided.

6.1 ArUco markers

6.1.1 Speed

In table 6.1 the measured time are given. The first column states the number of ArUco markers visible in the frame. The seconds column shows the amount of milliseconds it takes to identify those markers. The third column states the amount of milliseconds it takes to draw the rectangles around the markers in the frame. Lastly, column four gives the total amount of time it takes to process the frame. The times stated in the table are the average times of processing 100 frames. The difference between the fastest and slowest times was 2-4ms.

Table 6.1: Detection time table

PICO board attachments			
# Markers	Detecting time(ms)	Drawing time(ms)	Sum(ms)
0	30	55	85
4	31	58	89
8	32	58	90
12	33	59	92
16	33	61	94
20	34	62	96
24	34	64	98

6.1.2 Accuracy

The accuracy of the algorithm did change a little depending on the location of measurement. In figure 4.1, it is clear that in the top right corner, the green line is not perfectly aligned with the black line drawn on the wooden surface. In the top left and right corner, the offset is around two centimeters. In the bottom left and right corner, the locations are almost at the perfect spot. The marker shown in the figure is identified at location X: 59,3 and Y: 71,3. But in reality, the location was X: 60,2 and 71,9. The average offset in this plane is one centimeter.

6.1.3 Lightning influence

While driving the PIRATE, the algorithm did not always recognize the ArUco markers. During the time driving 30 centimeters, we had, on average, 200 frames processed. With cloudy weather out of 212 frames, 14 times the marker was not detected. With bright sunlight, in 37 out of the 207 frames, the marker was not detected.

6.1.4 Artifacts

During all experiments with the ArUco markers, we did not measure one artifact.

6.2 Feedback communication

Using the feedback communication to signal when a command has been executed works, as is shown in figure 6.1. With the blue and red stars, we show the starting time of clamping the back and front. The time on the x-axis is in milliseconds. After starting clamping the front module, it

takes like 500 milliseconds to set the PWM signal of the actuator. The PWM value for the front module is drawn in blue. From then on, we see the torque in the bending spring to rise. The threshold value for triggering the clamped mode is set at 50% of the max torque and is displayed with the red line. This is because the value can sometimes be a little off. Also, this trigger only occurs when the average of the last three measurements is above the 50% threshold. It filters out small spikes in the measurements.

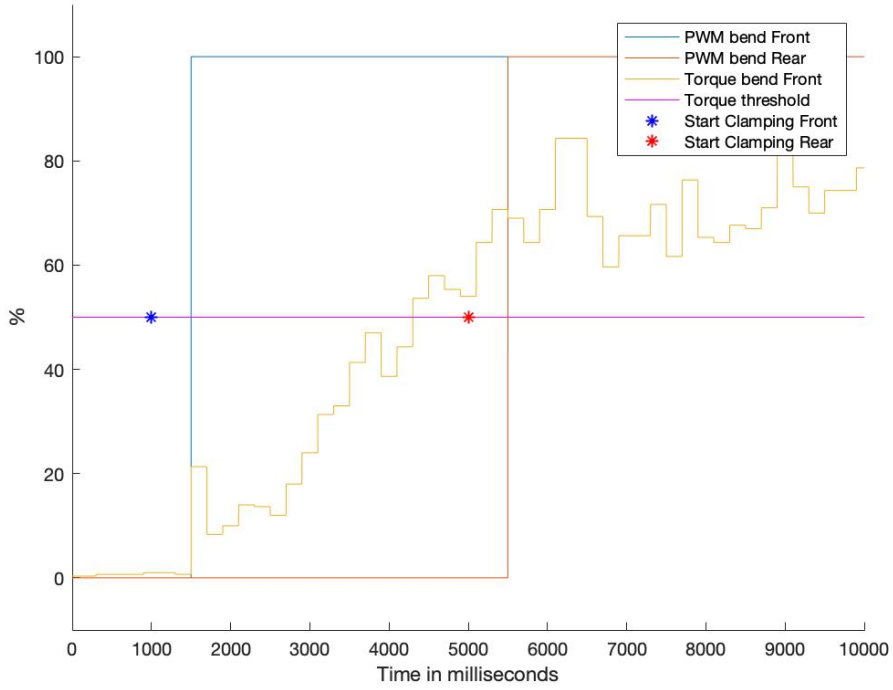


Figure 6.1: Chart showing the values of the sensors and the actuators while clamping the front and back part.

Once the front is clamped, the back clamping command is started and it takes a bit of time to set this actuator's PWM signal. The PWM signal for the back part is given with the orange line.

6.3 Moving through a bend

The experiment of moving through the bend was almost flawless. The exact positions of start executing sequences had to be changed to get the best performance. For example, the best location to stop driving and start bending the robot thought the bend. One problem we noticed a few times was that a small bump in the bend blocked the front module. A manual push solved this problem. Also, unclamping the back module did sometimes fail because the motors did not have enough torque. One of the reasons for this could be that the motors became old, but most likely, the motors have to be replaced by stronger motors nevertheless. Out of multiple runs, about 50% of the time, the experiment did succeed. In figure 6.2, each step of moving through a bend is shown.

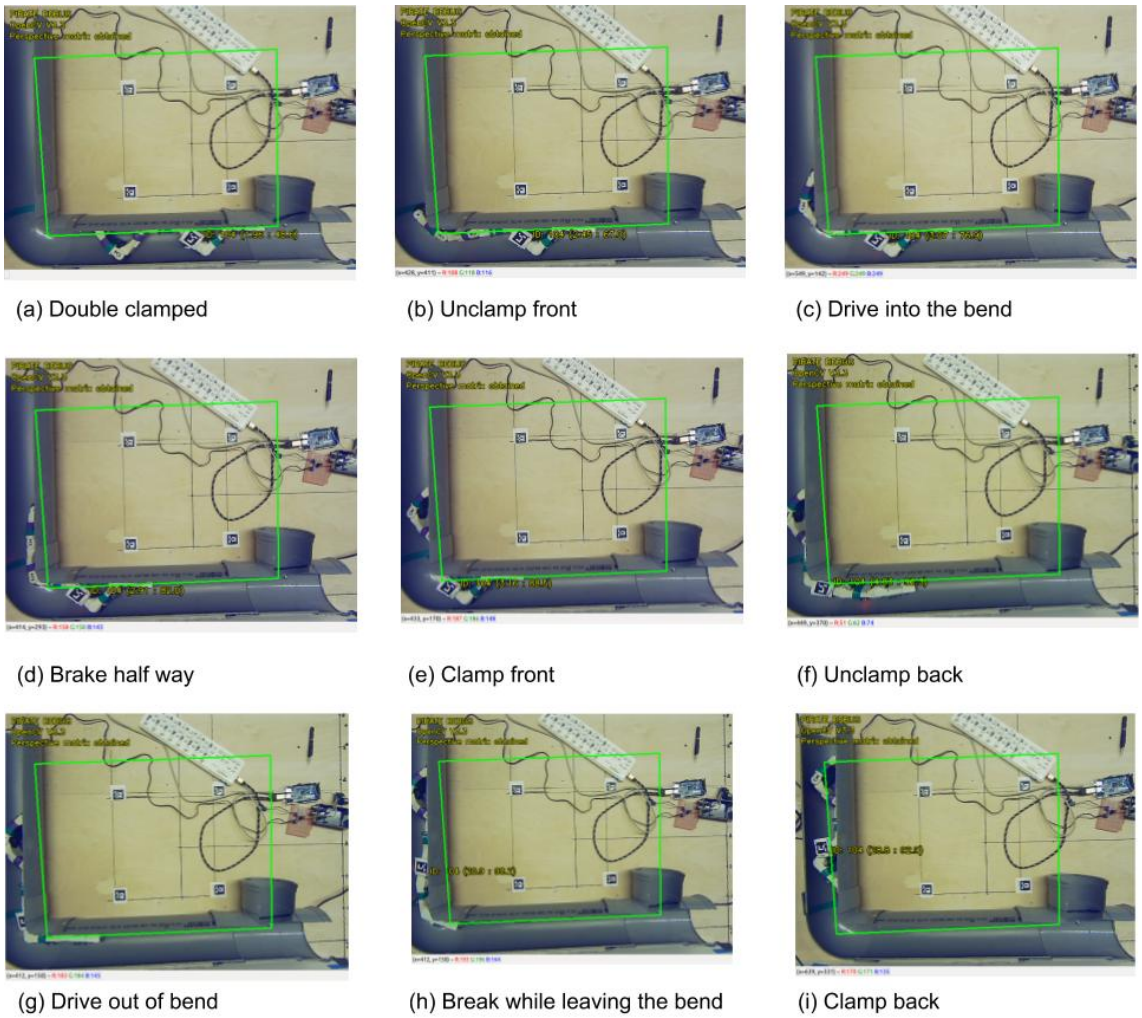


Figure 6.2: Sequential steps to drive through a bend.

7 Conclusions and recommendations

In this chapter ,we discuss the outcomes of this research project. We then restate the research questions and make conclusions based on the results of this project. Finally, we give recommendations for further research to improve the PIRATE robot furthermore.

7.1 Discussion

7.1.1 Experiments

During this project, we did reassemble the PIRATE robot because Geerlings (2018) had many problems with the robustness of the robot. These problems resulted in many failed attempts doing experiments with the robot. The new assembled robot used the old design with mostly reused parts. One of the main problems the robot still has is that the motors do not produce enough bending torque. The motors are old and may have lost torque, but most likely stronger motors are required to drive vertically.

7.1.2 DF57 Cables

The DF57 cables used in the PIRATE to make all connections to the PICOboards are replaced during this project. Because new cables are used, less connection failures are observed during experiments. But the older the cables will become, the higher the change the failures will occur again.

7.1.3 Mechanical design

During this project, we did not change the design of the PIRATE because it was not our field of expertise. There are still flaws in the design that can improve the PIRATE when they get solved. Examples are only one screw to assemble both 3D printed parts together of a module and the motor used for bending.

7.2 Conclusions

7.2.1 ArUco markers

The ArUco markers made the setup much more stable. Calculating the camera position is fast, and the location of the PIRATE is detected fast. Also, the artifacts that we often saw in the blob detection are no longer a problem. The only way an ArUco marker is detected less often is when the marker is not on a flat surface or when a small part of the marker is blocked for the camera.

The speed and accuracy of the marker detection is suitable for this type of project. On average, the detection of all markers is only 33 milliseconds. Drawing the markers on the GUI takes an extra 60 milliseconds. This does not improve the functionality of the robot but makes it easier to operate and maintain.

The sun lightning does increase the detection rate with 13% to 20%. With a frame rate of 10Hz, this percentage is acceptable for the project and does not result in failing experiments. In comparison with the blob detection, the ArUco marker detection functions in all sunlight intensities were the blob detection fails with high light intensities.

7.2.2 Feedback communication

The feedback communication improved the execution of sequences. Instead of waiting a static amount of time and hope the command was successfully executed, we now wait until it was successful. If the command takes to long or fails, we stop the sequence. The result is that the

PIRATE stops executing the process if errors occur but keeps going as long everything is going correct.

7.2.3 Moving through a bend

Driving through the bend was successful. The PIRATE did make the right decisions at the right time. Still, we see the same problem where the robot gets stuck because it does not have enough torque to execute a specific step. To improve the process, the PIRATE has to be improved mechanically. Increasing the torque of the bending motors would solve the problem of getting stuck. Another issue that we see occur sometimes is that the computer is losing the serial communication with the PIRATEbay. The reason for this was not found during this project but can be a bug in the rosserial package. In the recommendations, we give a possible solution to this problem.

7.2.4 Feedback GUI

The Feedback GUI was created to give a better understanding of the robot's status. This interface was useful mainly while reassembling the robot. Also, when a PICOboard is losing the RS485 communication with the PIRATEbay during experiments, this can quickly observed in the interface.

7.3 Recommendations

In this section, we give recommendations for further work on the PIRATE. Each recommendation should be investigated further to make sure it is an improvement to the PIRATE.

7.3.1 Setpoints

The PIRATE is using setpoints to update the PWM values for the actuators of the robot. When multiple setpoints have to change, a message is sent from ROS to the PIRATEbay for each actuator. A more efficient communication would be when all setpoints get changed, and after that, one message is sent to the PIRATEbay. This can reduce a large number of messages that have to be sent over the serial communication line.

7.3.2 Serial communication

The serial communication between the computer and the PIRATEbay sometimes drops. A message in ROS will state that it had reinitialize the connection. Setting up the serial communication always restarts the PIRATEbay, and thus all setpoints are reset. Restarting the Arduino is because of the Data Terminal Ready(DTR) signal. One of the reasons this can happen is because the spinOnce method in the PIRATEbay isn't executed within 5 seconds after the last call. It's not sure if this is the problem. If the user keeps moving the sliders of the KontrolBoard than the PIRATEbay stays forever in a while loop. The code in the Piratebay should be changed so that we can guarantee that this function is executed in time.

7.3.3 KontrolBoard

The KontrolBoard could maybe removed from the project. One of the benefits of having this board is that you can operate the robot without using a computer. But a downside is that it can overrule commands from ROS. Also, the KontrolBoard is adding much code to the PIRATEbay sketch adding more complexity and course problems while operating the robot, as stated in the previous recommendation. All the possible operations of the KontrolBoard could be moved over to a user interface within ROS. These commandos can than be ignored if the Sequence node within ROS is operating the PIRATE.

7.3.4 ESP8266

When removing the KontrolBoard from the project, we don't need any longer the USB port on the PIRATEbay. We can replace the expensive, slow Arduino Mega ADK with an ESP8266. One of the benefits can be that the Serial communication mentioned in the recommendation "Serial Communication" can be replaced with communication over WiFi. Replacing the Arduino Mega ADK with an ESP8266 would make the PIRATEbay faster, cheaper, smaller, and more portable because no cable between the PIRATEbay and laptop is needed. The ESP8266 and the laptop only have to be in the same WiFi network.

7.3.5 3D printer module parts

The design of the 3D printed part of the modules only contains one screw hole. Because of this, the two parts forming the body of one module in the robot do not stay in place then bending the robot. In this project, this problem is solved by using tape to keep the parts together. A second, and maybe even a third screw-hole, would fix this problem and makes the robot look more professional.

7.3.6 Bending strength

The strength of the bend modules is not strong enough. This makes it harder two perform certain experiments because a small bump in a pipe bend can block the robot from moving. The motors used for the bending are not the newest and can maybe be replaced with newer and stronger motors. There is also some extra space, so we consider using bigger motors. Also, the mount around the bending motor is still the old one, which is 3D printed out with PLA. It would be better to print these components with carbon fiber, like the wheels.

7.3.7 Computer setup

To operate the PIRATE robot, a computer with Ubuntu is required to run ROS. At this moment, no default computer is part of the PIRATE setup, and so a researcher is required to get his own machine with Ubuntu. Setting up a machine with Ubuntu and getting the project working takes some time. It would be nice for future researchers to work on a dedicated computer for the project, so they don't have to get one on their own.

7.4 Overview

In this project, we improved to localization in test setups and implemented a preemptable task architecture. The new PIRATE that was build during this project is more robust. Improving the torque of the motors will increase the functionality of the robot the most. Right now the control software is more advanced than the mechanics of the robot. With improved torque, tasks will become much easier for the robot. Once the robot can easily pass the commonly seen pipe segments, mapping the pipe network will be the main focus. The ultimate goal would be that the robot is able to travel autonomously through a pipe network and map it. Lastly, sensors to detect damages in a pipe should be investigated and used to locate them on the map.

Bibliography

Park et al., J., H. D. C. W.-H. K. T.-H. Y. H.-S. (2011), Normal-force control for an in-pipe robot according to the inclination of pipelines.

Bennet, S. (1988), Real-Time Computer Control: An Introduction.

Bradski, G. (2000), The OpenCV Library.

Broenink, J.F., Y. N. (2012), Model-Driven Robot-Software Design using integrated Models and Co-Simulation.

Dertien, E. (2014), Design of an inspection robot for small diameter gas distribution mains.

Garrido, s., M. R. M. F. M. R. (2015), Generation of fiducial marker dictionaries using mixed integer linear programming.

Geerlings, N. (2018), Design of a control layer for robust autonomous navigation of the PIRATE by means of motion primitives and path planning.

Hoekstra, G. (2018), Towards a software architecture model for the automation of the pirate robot.

Kakogawa, A., S. M. (2019), Robotic Search and Rescue through In-Pipe Movement.

Morales, G. (2016), Increasing the Autonomy of the Pipe Inspection Robot PIRATE.

Pulles, C. (2008), Pirate, the development of an autonomous gas distribution system inspection robot.
https://www.researchgate.net/publication/254900483_Pirate_the_development_of_an_autonomous_gas_distribution_system_inspection_robot

Ramirez, F., C. R. S. R. (2018), Speeded Up Detection of Squared Fiducial Markers.

Reiling, M. (2014), Implementation of a monocular structured light vision system for pipe inspection robot pirate.

Santos, H.B., T. M. d. O. A. d. A. L. N. J. F. (2017), Control of Mobile Robots Using ActionLib.

Tucci, S. and C. Schlegel (2017), Presentation of the robmosys project.
https://robmosys.eu/download/sara-tucci-cea-christian-schlegel-hs-ulm-presentation-of-the-robmosys-proje

Zheng, J., B. S. C. B. Y. D. (2018), Visual Localization of Inspection Robot using Kalman Filter and Aruco Markers.

Appendix A

PICOboard code upload

In this appendix we describe the procedure of uploading code to the PICOboard. Most of the information in this appendix is copied from the PICO_eCompass_MPCMn_NoServoFront.ino file.

PICOboard hardware

ATmega328 at 8MHz, 3.3V using Allegro A3906 motordriver.

RS485 bus for communication and uploading firmware.

Two AS5055 sensors on both sides of a compliant element measure torque Red and Blue LED.

1. red: status indication:
   (a) heartbeat > normal operation
   (b) blink 2.5 Hz > not used at this moment
   (c) blink 5 Hz > not used at this moment
   (d) blink 10 Hz > not used at this moment
   (e) on > limit error (encoder, current) angle value can not be correct.
2. blue: communication (only transmit)

Compiling/uploading

- Compiles with Arduino > 1.0.5 (because of the 'serial flush()' implementation

- Make sure the 'PIRATE' hardware library is installed in the sketchbook

- select as board PICO eCompass RS485 - 3.3V 8MHz

- for programming use a USB-serial-RS485 adapter such as the FTDI based by devantech

- power the bus cable on one of the 4-wire RS485 bus connectors with the USB 5V (preferably through a power switch) 1 VCC, 2 A, 3 B, 4 GND.

- Compiler options

Change compiler options in the platform.txt file to increase speed, default is optimised for size (-Os)

There are 2 lines in which you can change -0s to -O2:

compiler.c.flags=-c -g -O2 -w -ffunction-sections -fdata-sections -MMD

compiler.cpp.flags=-c -g -O2 -w -fno-exceptions -ffunction-sections -fdata-sections -fno-threadsafe-statics -MMD

- Select the correct board ID (i.e. 20 .. 27) with #define PICO_SLAVE_ADDRESS <ID>

Ids of each module can be found in figure 2.1 of this document.

Board ID 25 is used to control rotation.

Board ID 20 is the front module with pan & tilt for camera.

Other boards: bending type

The software for these modules are slightly different and is selected with a define:

#define PICO_ROTATIONAL_MODULE

#define PICO_FRONT_MODULE

#define PICO_BEND_MODULE

- After hitting the upload butten there is a short window to turn on the board power and start the programming. The bootloader times out after 0.5 sec.

- For (re)programming the bootloader (not neccessary during normal operation) use for example the AVR ISP mkII (USB) and select tools-> burn bootloader. make sure the board is powered. Short resistor R7 (reset wire), see figure 1, with a pair of tweezers. The ISP can be connected using the standard 6-wire pinning to connector P5 (bend motor sensor)

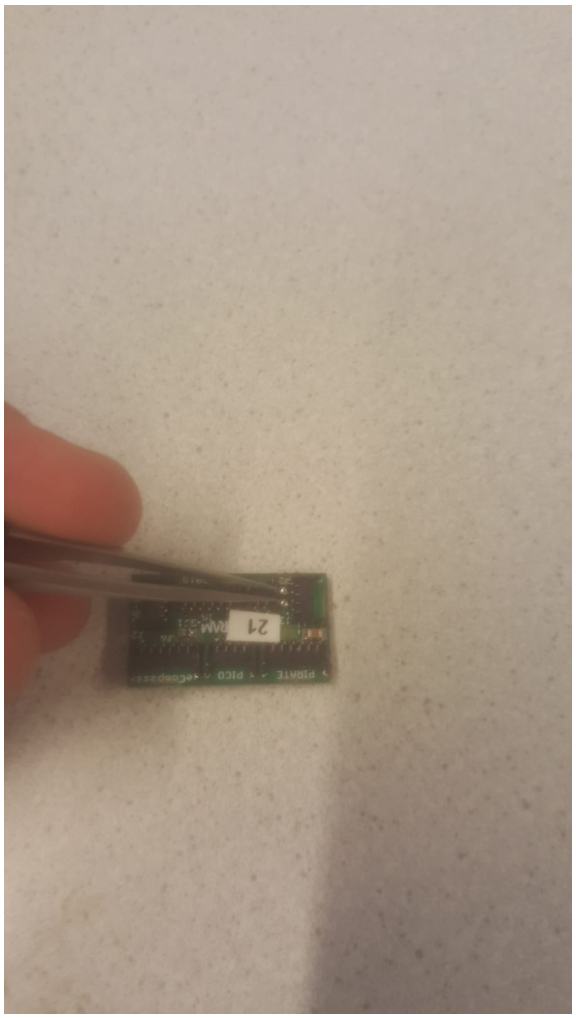


Figure 1: Shorting R7 to upload code.

Rotation module

- In the rotation module firmware (ID 25), the angleSetpoint is used to control the rotation angle using a Position controller. Note that the encoder is started in zero-position (the startup position is used as zero reference)

Front module

- Position control is used for the pan-tilt camera. The bendmotor is used for tilt and together with module angle sensor P7, the drivemotor is used for panning together with the spring sensor P1.

Dynamixel Protocol

- For communication a protocol is implemented based on the Dynamixel protocol by Robotis. This packetized serial protocol uses RS485 at 57600 baud. In the DynamixelReader.cpp a routine DynamixelPoll() needs to be issued regularly in order to process data from the serial buffer. In DynamixelReader.h the memory struct is implemented which follows the original protocol, up to a point where we use two motors (and dynamixel normally one in different control modes). Currently not all functionality is implemented, although the communication routine will write and read the allocated memory slots.

The packet is normally:

- 0x14 (ID 20)

- 0x02 (length 2)

- 0x03 (command: 0x01 = ping, 0x02 = read, 0x03 = write)

- 0x1E (address: starting at 0x1E (position, see DynamixelReader.h)

- 0x00 (MSB)

- 0x00 (LSB)

- 0x38 (checksum)

Appendix B

NanoKONTROL2 MIDI panel

In this appendix we will list all commands we can send with the NanoKontrol2 MIDI panel. This panel is shown in figure 2, for each button or slider the id is listed. This id send send with an event to handle in the PIRATEbay. The NanoKONTROL2 is connected directly to the PIRATEbay with a USB cable. To know if the board is working correctly check the white light in the left top corner. When it is on, the panel is working. If it is not, check if the code is running correctly on the PIRATEbay.

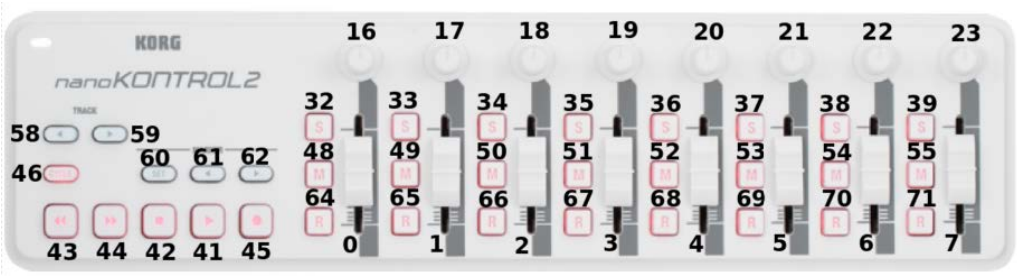


Figure 2: The MIDI panel. Adapted from Morales (2016)

Startup

When the PIRATEbay and PIRATE are powered up, the red leds on each PICOboard should blink with a slow rate. This means the boards are in idle mode. We can now press the play button(41) to set the robot in normal operation mode. The red leds should now blink with a heartbeat rate. To return back to the idle mode we can press the stop button(42). To send information from the PICOboards to the main computer we can press the cycle(46) button. The PICOboards should now start blinking the blue led lights. If the main computer with ROS is connected to the PIRATEbay and ROS is running, topic "/os/setpoint" should be visible. "rostopic list" lists the topics and "rostopic echo <topicname>" should output the messages to the screen.

Driving

To drive the robot we should be in operation mode. Slider(7) will make the robot drive, middle is neutral. Up and down will move the robot forward and backward.

Bending

Slider 1, 2, 3 and 4 are used to activate the bending motors of the 4 bending modules. Again the middle is neutral. Each bend is measuring the magnet poles, if the magnet is replaced the origin has to be reset. To do this we can press set(60) and the R(65, 66, 67 or 68) button on the left side of the slider to reset that module angle.

Rest

Potentiometer(23) is used to rotate the front and back part of the robot. Slider 5 and 6 are programmed to operate the camera in the front module but was not used during this project. All action events are processed in the PIRATEbay code.