Automation of an underwater vacuum cleaner

Marc Valdivieso Merino
email: marvalmer@gmail.com
Abstract

The SRO Pool Cleaner is designed to clean the bottom of a pool by the use of the tools in its mobile actuator component, which is a two-wheeled vehicle that can be controlled remotely. This document is constituted by four parts: An introduction that explains its objectives and relevant context, an explanatory part in which the mechanics of every part of the project, both hardware and software, are explained in detail with its functionalities and limitations, an evaluation of the functional results obtained with their drawn conclusions, and appendices that include the commented code needed for every module used.
# Contents

I. Introduction xi

1. Introduction 1
   1.1. Original state of the SRO Pool Cleaner 1
   1.2. Motivation 2
   1.3. Objectives 2
      1.3.1. Project development procedure 3

2. State of Art 4
   2.1. Sensors listing 4
      2.1.1. IMU 4
      2.1.2. Bumpers 4
      2.1.3. GPS 5
      2.1.4. Camera 5
      2.1.5. Infrared 6
      2.1.6. Ultrasound 6
      2.1.7. Rotary encoders 7
   2.2. Protocols for autonomous navigation 7
      2.2.1. SLAM 7
      2.2.2. Basic Systems 8
   2.3. Interpretation of relevant IMU data 9
      2.3.1. Quaternions 9
   2.4. Arduino 10
   2.5. ROS framework 11
   2.6. Communication protocols 11
      2.6.1. I2C 11
      2.6.2. Serial 12

3. Sensors proposal 13
   3.1. Imu 13
   3.2. Bumpers 13
   3.3. Future additions 13
      3.3.1. Ultrasonic 13
      3.3.2. Camera and GPS 13
      3.3.3. Rotary encoders 13

II. Setup 14

4. Hardware 15
   4.1. Power 15
      4.1.1. Repair procedure 15
      4.1.2. Updates 15
List of figures

1.3. SRO Pool Cleaner with joystick [11] ................................. 1
1.5. UNO original connections [11] ..................................... 2

2.1. Razor IMU M0 ................................................................. 4
2.2. Razor 9DoF sensor stick ............................................... 4
2.3. Robot with a bumper system.[15] .................................. 5
2.4. Bumper sensor.[15] .................................................... 5
2.5. GPS chip for microcontroller projects ............................. 5
2.6. Commercial GPS .......................................................... 5
2.7. CMOS camera ............................................................. 6
2.8. Spectrum of infrared light ........................................... 6
2.9. Typical infrared module for robotics projects ..................... 6
2.10. Ultrasound sensor ....................................................... 7
2.11. Incremental encoder [13] ............................................. 7
2.12. Resistive rotary encoder ............................................. 7
2.13. Robot prepared for VSLAM [12] .................................. 8
2.15. A classical roomba robot and a Dolphin Pool cleaner .......... 8
2.17. Arduino boards used in this project ............................... 10
2.18. A graph of all the ROS nodes in this project ................. 11

4.1. Voltage regulation module from 24V to 5V ..................... 15
4.2. Original state of the SRO Pool Cleaner.[11] .................... 16
4.3. SRO Pool Cleaner after the inclusion of the joystick.[11] .... 16
4.5. The powered up mega module ..................................... 17
4.8. UNO module with shield ........................................... 18
4.9. Sideview of the UNO module ..................................... 18
4.10. Connection scheme UNO without 9DoF link .................. 19
4.11. UNO module with shield and IMU ............................... 19
4.12. Serial connections IMU-Arduino ................................. 19

5.2. Flow diagram for the MEGA code ............................... 22
5.3. Flow diagram for the UNO code .................................. 23
5.4. Flow diagram for the 9DoF’s embedded Atmega328 .......... 24
5.5. Yaw, Pitch and Roll [9] ............................................. 27
5.6. Latency between control input and IMU packet ............... 28
5.7. Flow diagram to decide the collision ............................ 29
5.8. Flow diagram for the autonomous navigation node ............................... 30
6.1. Communications diagram .................................................................. 31
6.2. Razor IMU 9DoF connected with the ftdi module ................................. 32
7.1. Collision graph comparison: With and without ROS input ..................... 36
7.2. Detail of the collision reaction ............................................................... 37
8.1. SRO lab testing setup ........................................................................... 42
8.2. Typical range of rotations ...................................................................... 42
9.1. Small sensor stick on the 3.3V I2C bus and a waterproof junction box ........ 44
List of tables

2.1. Multiplication properties for general quaternions ........................................ 9
2.2. Arduino specifications ....................................................................................... 10
4.1. XBee network configuration data ....................................................................... 17
5.1. Detailed topic information about the collision_detect node ......................... 30
5.2. Detailed topic information about the autonomous navigation node ............. 30
## Definitions

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROS</td>
<td>Robotic Operative System</td>
</tr>
<tr>
<td>IMU</td>
<td>Inertial Measurement Unit</td>
</tr>
<tr>
<td>9DoF</td>
<td>9 Degrees of Freedom</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver/Transmitter (Serial)</td>
</tr>
<tr>
<td>I2C</td>
<td>Inter Integrated Circuit</td>
</tr>
</tbody>
</table>
Part I.
Introduction
1 Introduction

This project continues the work done by Bethany Kon supervised by Raúl Acuña on the "Joystick Control for the SRO Pool Cleaner Robot", which adapted a basic layout working with control signals emitted by a transmitter controller to a digital control operated mode using a laptop and on board controllers using the ROS framework. It has been developed in the werkstatt (S3|03) 418 of the Regelungsmethoden und Robotik department in the TU-Darmstadt, and has been supervised by Raúl Acuña. It has been aiming to take a step forward in the designing of a fully functional SRO platform, as well as studying its sensing capabilities and its future possibilities.

During the previous stage of the project, the option to control the SRO platform with a laptop was added. This was achieved through the substitution of the specialized receiver module on the receiver component for an arduino MEGA with an XBee module that enables wireless communication with a laptop with an
XBee module on the same XBee network. The commands for movement are generated by a script using the data from a joystick controller. Additionally, an I2C communication link between the actuator and receiver component connects an Arduino UNO and an IMU to the Mega board, which can provide data to the laptop from the sensors located in the actuator component.

Figure 1.4.: Mega original connections [11]

Figure 1.5.: UNO original connections [11]

The new Joystick Control build depends on digital signal to function, which depends on the stability of the microcontroller boards to be processed. This stability was compromised by the power supply on high current demands, leading to resets of the digital layer and the malfunction of the system.

1.2 Motivation

Domestic robot cleaners provide benefits regarding cost, energy, and time consumption on the long term for their users. Surveys of customer opinions over pool cleaners pointed design requirements that would be appreciated in this market. These requirements are both physical oriented and software oriented, and this project tries to tackle the software capabilities. It is highly desirable to develop the autonomous navigation function, and the first step to do so in an efficient way is to provide localization of the platform. Therefore, a cost/effectiveness analysis of which sensors to implement and the provision of the sensored data that will aid in localizing the system will be developed during this thesis.

1.3 Objectives

The aim of this project is to enable the SRO Pool Cleaner to work safely and continuously and providing to it the means of retrieving the most relevant information it can sense. The following goals were given at the beginning of the project.

1. Electronics protection
   a) Evaluation of the current state of the robot hardware with identification of potential problems.
   b) Integration of current/voltage protection circuits or implementation of ground separation techniques based on the previous analysis.

2. Integration of new sensors
   a) State of the art research on wall detection methods for underwater devices
b) Selection, acquisition and integration of the sensors in the robotic platform

3. Autonomous cleaning program design
   a) State of the art research on algorithms for autonomous cleaning of a given area
   b) Propose and implement an algorithm

1.3.1 Project development procedure

1. **Repair of the power electronics:** A reported voltage spike provoked the reboot of the microcontroller system which breaks communication through the rosserial link. It is crucial to guarantee the continuity of this communication link in order to function properly.

2. **State of art:** A study on the existing systems, sensors and protocols they use to perform the task, followed by a proposal on which to implement in the SRO, as well as information on the most important parts of this project.

3. **Repairings and adaptation of the system for the sensors:** In order to acquire the desired data and guarantee the required standards, the system had to be modified and repaired.

4. **Control method:** Create a proposal for an autonomous navigation procedure using the ROS framework.
2 State of Art

2.1 Sensors listing

2.1.1 IMU

An IMU is an electronic device that performs measurements over the acceleration, angular velocity and (occasionally) magnetic field of the body it is attached to, using an accelerometer+gyroscope and sometimes a magnetometer for the magnetic field. The measurements can be read and processed by microcontrollers, and these microcontrollers can be embedded in the board, external, or both. The following pictures show two different modules, both being IMU’s, the Razor M0 is a new and improved version of the IMU used in this project, including a USB connector, lots of connectivity options for both input and output data to be used or sent by its embedded processor, and the IMU sensors. The right module, only includes the sensor modules and the connection pins to add it to an I2C bus. There is a substantial difference in price, size and functionality, and the inclusion of each module should be studied and decided for every specific project.

![Figure 2.1.: Razor IMU M0](image1)

![Figure 2.2.: Razor 9DoF sensor stick](image2)

2.1.2 Bumpers

Bumpers are devices that enable a 'true' signal when a collision is detected. The collision event might be detected by an oscillation or pressure over a surface, and the simple signal is easy to read by a microcontroller. Since the readings are so simple, their reliability is high, and electronically inexpensive to implement. These are a widely implemented sensor in autonomous vehicles.
There is a wide range of options for their implementation. A system like the one on the left robot, will use a resistor that decreases its value when the cable is pushed, making it possible to choose a threshold value. The bumper on the right system detects oscillation to provide the discrete feedback value and additionally protects from damage on the platform. They can be designed with switches or interruptors as well.

2.1.3 GPS

GPS or Global Position System is a satellite-based radionavigation system. It provides information on the absolute coordinates of the device location. The land device receives signal from a satellite and calculates its position in function of the signal received. Water is very opaque to the received signal, therefore these are really hard to use in underwater vehicles.

2.1.4 Camera

A digital camera is a light sensor that provides a digital image readable by a microcontroller. A camera can be fit in the communication box as well. Together with the GPS location it can provide an absolute location for the robot, thus allowing an intelligent cleaning algorithm to exist.
There is a huge number of small cameras for microcontroller projects in the market, varying widely in price and specifications.

### 2.1.5 Infrared

Infrared light is used extensively for range-detecting purposes in projects with microcontrollers. They provide good resolution in a range from 0 to 1 meters. They are cheap, easy to implement, easy to find, and lightweight. They emit and read light with a slightly longer wavelength than the visual spectrum.

![Spectrum of infrared light](image)

**Figure 2.8.: Spectrum of infrared light**

**Figure 2.9.: Typical infrared module for robotics projects**

Water dissipates enormously infrared signal.

### 2.1.6 Ultrasound

Ultrasound has properties similar to infra-red light for range-finding purposes. They have much more operative range but slightly worse resolution. Unlike infrared signals, they do not suffer of special underwater attenuation.
Most of the modules that are sold are not adapted for underwater operation.

### 2.1.7 Rotary encoders

Rotary encoders are electro-mechanical devices that transform the reading of the angular position of an axis to a readable signal, used to read the position increments or positioning of the axis of the wheels of autonomous vehicles. They can be absolute or incremental. Incremental encoders give information about the movement of the wheel, and absolute encoders give information about its current position. The operating mechanism can vary enormously: The data can be obtained by current readings, optically, mechanically, magnetically, by capacitance readings, etc.

### 2.2 Protocols for autonomous navigation

#### 2.2.1 SLAM

(Simultaneous Localization and Mapping): In the context of robotics and autonomous vehicles, SLAM is the computational problem of generating and updating a map of an uncertain environment and the simultaneous tracking of the vehicle’s spacial coordinates within it. This can be achieved by approximation methods such as GraphSLAM or an extended Kalman filter, and the ROS framework provides a wide variety of resources for its implementation in custom projects. SLAM doesn’t aim to provide perfect mapping neither location; It aims to operational compliance designed with the computational, sensoring and time constrains in mind. The objective of the SLAM computation is the following:
\[ P(m_t, x_t|o_{1:t}) \]

Which can be sequentialized using the Bayes’ theorem:

\[ P(m_t, x_t|o_{1:t}) = P(m_t|x_t, o_{1:t}) = \sum_{x_t} \sum_{m_t} P(m_t|x_t, m_{t-1}, o_t) P(m_{t-1}, x_t|o_{1:t-1}, m_{t-1}) \]

Figure 2.13.: Robot prepared for VSLAM [12]

Figure 2.14.: Computational result of VSLAM [12]

Modern ground systems calculate an optimal trajectory using their sensory and computational capabilities. They clean zones and paths following complex and studied algorithms.

2.2.2 Basic Systems

Autonomous navigation doesn’t require solving the SLAM problem for all its applications. When randomizing movement within a constrained area is enough, only the detection of collisions is required.

Figure 2.15.: A classical roomba robot and a Dolphin Pool cleaner

The old models of Roomba vacuum cleaners are the most common example of this kind of system. These early models didn’t map their surroundings and had limited sensing capabilities, resulting in a much longer vacuuming time and less energetic efficiency, yet being effective enough to fulfill their purpose without computational effort.
2.3 Interpretation of relevant IMU data

2.3.1 Quaternions

Mathematically, quaternions are an extension of the complex number under the form \( a + bi + cj + dk \), where \( a, b, c \) and \( d \) are reals and \( i, j \) and \( k \) are the imaginary/quaternion units. In ROS framework, the 'q.w' variable contains the purely real data and the q.x, q.y, and q.z variables contain the i, j and k multiplied real values respectively. These i, j and k represent perpendicular solutions for \( \sqrt{-1} \) and they are not to be multiplied as in the standard complex algebra, but as in the following:

<table>
<thead>
<tr>
<th>Multiplication properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
</tr>
<tr>
<td>( r )</td>
</tr>
<tr>
<td>( i )</td>
</tr>
<tr>
<td>( j )</td>
</tr>
<tr>
<td>( k )</td>
</tr>
</tbody>
</table>

Table 2.1.: Multiplication properties for general quaternions

And by these rules:

\[
i^2 = j^2 = k^2 = ijk = -1
\]  \hspace{1cm} (2.1)

Quaternion multiplication is associative, but not commutative. They share the conjugation properties with the rest of complex numbers.

Quaternions are a quite extensive and complicated topic. Luckily, the objective of this message is to give orientation data, so they are quite easy to understand in this context. The imaginary part of the vector refers to a physical direction in which the body that the quaternion refers to points. The real part represents the clockwise angle of the rotation that the body has.

This data is obtainable from and revertible to the Yaw, Pitch and Roll data, which is more intuitive, yet for computation purposes the Orientation Quaternions are better because they are easier to operate with for a computer and operating with them doesn’t provide errors and dead angles like in the YPR data case.

Figure 2.16.: Orientation quaternion [9]
Arduino is an open source hardware and software company that designs microcontroller boards and programming interfaces for their microcontrollers. They are low cost, have community support and include the needed interfaces and connectors to be used in a project such as the SRO pool cleaner. There is a wide range of specifications and prices among the boards, and as in the case of the IMU sensors, in each project there must be an evaluation of the requirements to decide which is the most appropriate board.

An Arduino MEGA 2560 and an Arduino Uno are used in this project. They have the following specifications:

<table>
<thead>
<tr>
<th>Model</th>
<th>CPU</th>
<th>voltage</th>
<th>CPU Speed</th>
<th>Analog ports</th>
<th>Digital ports</th>
<th>EEPROM</th>
<th>SRAM</th>
<th>Flash(kB)</th>
<th>USB</th>
<th>UART ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEGA</td>
<td>2560</td>
<td>5-12V</td>
<td>16MHz</td>
<td>16/0</td>
<td>54/15</td>
<td>4</td>
<td>8</td>
<td>256</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>UNO</td>
<td>328P</td>
<td>3.3-5V/5-12V</td>
<td>8kHz-16Mhz</td>
<td>6/0</td>
<td>14/6</td>
<td>1</td>
<td>2</td>
<td>32</td>
<td>A</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 2.17.: Arduino boards used in this project

Table 2.2.: Arduino specifications
2.5 ROS framework

ROS (Robot Operating System) is a framework for software development that provides operative system functionality to robotic platforms. It was originally developed in 2007 in Stanford, and is still periodically updated with updates and new functionality. ROS provides device control services, hardware abstraction, messaging between processes and many developer tools. ROS works over UNIX OS. In its main site many user-developed packages can be found as well. These packages implement both physical and software functionalities, such as simulation, plotting and signal processing tools or control, sensing, SLAM/VSLAM etc. for physical robots. ROS is based in a graph-based architecture, which nodes process and manage the information flow through topics and services. These are defined within ROS packages, which define the structure of the nodes, topics, services and messages they contain. Each node can subscribe and publish to a series of topics thus creating the graph structure. Each topic has a defined message structure, which specifies the type of variable that is going to be published and/or received. Any user can define its own message types, but there is an extensive list of default messages that can be used and it’s highly recommendable to do so, since most of the published free-license package use the most standardized message types. This has taken in account when developing the software for this project.

![Figure 2.18.: A graph of all the ROS nodes in this project](image)

2.6 Communication protocols

2.6.1 I2C

I2C (Inter-Integrated Circuit) is a communication protocol designed to work in a single integrated board. Nonetheless, it can work over small distances with a reduction of the transfer speed. It uses four wires, being the power line, GND, a line for the clock (SCL) and a data line (SDA), so it is a synchronous protocol. It has a theoretical maximum of 1008 (or 127 in the case of the arduino platforms) slave addresses which could be extended by multiplexing (yet this is far more than needed in any part of this project). It supports a multiple master structure as well, none of them requiring address, because they are not required to react to any command.

Any master device will write on the bus the address of the device that wants to read from or write to, will specify if it wants to read or write, and will send 8 bit data frames on reception of the corresponding ACK header. The slaves will only write or read the required data from the bus when their address is casted into it.
Since this kind of serial protocol uses only four wires to function, it was the choice for a relatively long link in this project (Mega-Uno) which has a constraint of four lines, even though the length compromises its reliability and speed.

2.6.2 Serial

There are two main differences from I2C as a protocol: Instead of synchronizing with an external clock, each end of the communication link specifies the amount of bytes that are going to write/read over the port. This makes it dependent on the microcontroller capacity for the sampling in this regard. In addition of being an asynchronous protocol, it uses one wire for transfers and another one for receiving data. It is possible to make a connection between several nodes, but really uncommon, in general this kind of communication goes from one controller to another. In the case of the Razor IMU 9dof, this protocol is applied over a 6 pins FTDI interface, which makes it unable to communicate directly over the I2C link with the Mega.
3 Sensors proposal

3.1 Imu

An IMU module with its own microcontroller was provided at the beginning of this thesis. There was a previous attempt of collecting its data and this feature was within the project specifications. A new modified firmware has been uploaded to the IMU so its controller is able to send linear acceleration, angular velocity, yaw, pitch and roll through serial protocol to another controller. This new firmware also uses the processing power of the inboard ATMega328 microcontroller to run noise-filtering algorithms and upload calibration data, minimizing the load on the other controllers.

3.2 Bumpers

Due to the easy electronic implementation, reliability and low price tag, the proper additional electronic circuitry for their use was added. They might potentially assist with the IMU readings and the optimal rotation of the robot when colliding with a wall. The timing of the IMU message doesn’t always match the exact moment of the collision, making some of the packets received not as accurate as could be if sent when a reliable collision is detected.

3.3 Future additions

3.3.1 Ultrasonic

The decrease of speed when detecting a wall in the distance can be really useful to lengthen the lifespan of the overall system. Its data can be used as well to generate a map when sent through the proper processing in the ROS framework. It is not an expensive module to implement neither computationally or economically, widely used in autonomous navigation and useful for both the SLAM purposes.

3.3.2 Camera and GPS

Together, these could provide absolute location for the system. The protocol behind this lies outside the scope of this project which will only propose an algorithm for the most simplistic navigation. They can make this positioning task effective and give much more effectiveness to the system than the current available options in the market. Some of the current commercial options announce manual control as an advantage because it would allow the user to focus on the dirtiest places.

3.3.3 Rotary encoders

Together with the accelerometer can provide valuable data on whether the robot is drifting or not, plus odometry data for positioning through already implemented ROS packages that feed on this specific kind of message (odom) and IMU messages that we already accomplished to generate.
Part II.
Setup
4 Hardware

This chapter tackles the physical structure of the robot, the needed components for its operation, both in the power supply and digital domain. It also offers some insight on the original design of the robot citing from the previous stage of the project’s document.

4.1 Power

Here the requirements of the power supply and the power needs of the modules are explained, as well as the conclusion of the repair stage that was stated as an objective of this project and the procedure that was followed to find the source of the problem.

4.1.1 Repair procedure

In the last phase of the project the use of the SRO Pool Cleaner was not possible because a sudden change of direction of the wheel servos caused the Arduino MEGA to reset, event that broke the communication link. Two possible diagnosis were offered: The lack of a bypass capacitor in the power connector of the MEGA, or the reset being a result of signal interference from the driver boards when a sudden peak of current was required. After examining the case, neither were the case. Analysis by oscilloscope suggested that driver boards do not generate any output signal towards their input side when a big current is required. After adding a parallel capacitor to the power input of the MEGA, the issue remained. Therefore, the MEGA module wasn’t powered up for a time longer than what a capacitor was able to compensate, and such problem could only be attributed to the power supply. As expected, when the power supply from which the MEGA board is powered up was changed, the problem disappeared.

4.1.2 Updates

A voltage regulator from the 24DC output has been connected to power up the Arduino Mega at 5VDC. This guarantees the temporal continuity of the communication link.

Figure 4.1.: Voltage regulation module from 24V to 5V
4.1.3 Description

Six connections exist between the power and the receiver components. Two carry 24V from the same power supply, one carries the before described reduced voltage from this source 24V, and one carrying 12V, which is the one that powers down when the voltage spikes happen, and two for GND. There are fourteen connections between the Receiver and Actuator Component. Ten are used to control the direction of the motors (two wires dedicated to each of the five motors). The brushers are powered by the 24VDC Power source, therefore they do not reset the system when working at maximum speed. The wheels are powered by the 12VDC power source, which resets on maximum current output. This source gets its power from the previous one, and any electronics that need continuous communication with an external node or continuous operation must not be powered by it.

Figure 4.2.: Original state of the SRO Pool Cleaner.[11]

Figure 4.3.: SRO Pool Cleaner after the inclusion of the joystick.[11]

4.2 Digital

In this section the physical updates on the digital modules are explained. The new functionalities require updates on the boards and their connections.

4.2.1 XBee modules

These are for wireless communication between the laptop and the MEGA controller. Their operative mode remains unchanged since the last phase of the project, and the same configuration has been used. Further testing suggested that a baudrate of 115200 bps can be achieved without compromising the reliability of the received packets, yet for the current bandwidth requirement it is not needed.
### Network configuration for both XBee modules

<table>
<thead>
<tr>
<th>Connection</th>
<th>Computer</th>
<th>Arduino MEGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>Coordinator</td>
<td>End Device</td>
</tr>
<tr>
<td>Channel</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>PAN ID</td>
<td>1331</td>
<td>1331</td>
</tr>
<tr>
<td>MY Address</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Destination High</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Destination Low</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1.: XBee network configuration data

### 4.2.2 Arduino Mega 2560

The only update in the Mega connection scheme for this phase of the project is the power line. It was previously powered up by the Arduino power jack at 12V by a wire connected to the same power line that the wheel servos. Due to the instability of this powerline when a current peak is required, two additional wires connected to the 5V and Gnd pins of the Mega board were added and connected to the downward voltage regulator from the 24V power supply. This connection prevents the microcontroller of powering down when too much power is required for the wheels, therefore the rosserial link stays stable and the whole system can operate as intended, without power limitations for the wheels neither constant shutdowns.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3V</td>
<td>XBee Vcc</td>
</tr>
<tr>
<td>5V</td>
<td>Arduino Uno 5V</td>
</tr>
<tr>
<td>GND</td>
<td>XBee Gnd</td>
</tr>
<tr>
<td>GND</td>
<td>Arduino Uno Gnd</td>
</tr>
<tr>
<td>GND</td>
<td>Motors Gnd</td>
</tr>
<tr>
<td>0 (RX0)</td>
<td>XBee DOUT</td>
</tr>
<tr>
<td>1 (TX0)</td>
<td>XBee DIN</td>
</tr>
<tr>
<td>2 (PWM)</td>
<td>Right Wheel</td>
</tr>
<tr>
<td>3 (PWM)</td>
<td>Left Wheel</td>
</tr>
<tr>
<td>4 (PWM)</td>
<td>Front Brush</td>
</tr>
<tr>
<td>5 (PWM)</td>
<td>Right Brush</td>
</tr>
<tr>
<td>6 (PWM)</td>
<td>Left Brush</td>
</tr>
<tr>
<td>20 (SDA)</td>
<td>Arduino Uno SDA</td>
</tr>
<tr>
<td>21 (SCL)</td>
<td>Arduino Uno SCL</td>
</tr>
</tbody>
</table>

Table: Arduino Mega 2560 pin usage

Figure 4.5.: The powered up mega module

Figure 4.6.: Mega connections [11]
Other than this update for powering up the Mega board, the connection scheme remains the same.

![Mega connections](image)

**Figure 4.7.: Mega connections [11]**

### 4.2.3 Arduino Uno

A new shield coherent with the new requirements has been manufactured. Two interruptor circuits connected to the second and third pins are welded on the board, since these are the two default pins for interrupt signals in the UNO. The previous connection from the 12th and 13th pins to the RX and TX pins of the Razor 9DoF IMU are welded, but since the 4800 baudrate is no longer viable they are wired as well to the native hardware serial pins 0 and 1, which the sketch currently uses at a native baud rate of 57600. This board includes too a 4 wire input compatible with another I2C device on the main bus, which might be useful in future phases if the sensor stick IMU or another I2C device must be added to the actuator. It differs on the main one in that is powered up by the 3.3V regulated output, which is compatible with the new IMU module and a wide variety of sensors.

![UNO module with shield](image)

**Figure 4.8.: UNO module with shield**

![Sideview of the UNO module](image)

**Figure 4.9.: Sideview of the UNO module**
The next figure shows the connections on the UNO board ignoring the serial link with the Razor 9DoF IMU, which connection scheme is illustrated on the next section. Both are superposed in the board, but shown separately to provide a more decluttered vision of its structure.

The bumper system in the picture requires an $R=10\,\text{kOhm}$ to properly work and prevent an excessive current flow in case each bumper is activated. One of the bumper systems ends up in two activator switches instead of one, corresponding to each of the brushes where they reside. This system can be expanded with further pins corresponding to different parts of the actuator component, but just the two pins that are capable of generating an interrupt signal natively are used in this case.

4.2.4 Razor IMU 9DOF

The front view of the connected module and a scheme of the connections between this two modules in order to get the serial connection working.
5 Software

In this chapter the operation of the modules on a high level will be described, as well as pseudocode for the sketches and flow diagrams to give an easy insight on their operative modes without giving details of the micro-routines and the data processing. This chapter is divided between the code that runs on the microcontrollers and the one described in ROS nodes and runs therefore over a Roscore instance in the computer.

5.1 Microcontrollers

The SRO platform requires of at least one microcontroller to function with its joystick mode, located on the actuator, that acts on the drivers that power the servos, communicates with the laptop running roscore, and publishes information about the sensors: All of this is done by the Arduino Mega 2560 module. However, two additional microcontrollers were also added in the last phase of the project: An on-board AtMega328 on the IMU module that was available, and another one on an Arduino Uno which was used to establish
an I2C connection with the MEGA controller. The software of all three modules has been updated to add new functions and reliability.

5.1.1 Arduino Mega 2560

It is programmed to deliver the information obtained from the UNO through the I2C connection they share to the ROS system through the XBee module and to subscribe to the topics that generate the drive signals for the actuator to operate. The data it publishes refers to the state of the bumper system and the estimates of the IMU, on a rate of 20 times per second and up to fifty. It is also responsible of the performance of a series of transformations over the received data before publishing it to ROS: The data originally obtained from the sensors is not compatible with the ROS framework standards. To minimize the complexity of the AHRS firmware updates and respect the communication bandwidth constraints, the required data processing is done in this module. The concretes of the operations are explained in the latter chapter "Data processing". The following graph shows a very high-level and not exact idea of the operation of this system, but it’s enough to give a general idea without the details of the code, which is given in the appendices as well.

---

**Data:** ROS message  
**Result:** Acting on servo and sending data to ROS  
*setup*
Initialization routines  
*loop*
  
  if *ROS command received* then
  Act on the servo assigned to that command
  else
  Request IMU package to UNO slave
  Run processing data subroutines
  Publish sensor data to ROS
  end
  
  goto *loop*
Figure 5.2.: Flow diagram for the MEGA code
5.1.2 Arduino Uno

It obtains data both from the IMU and interruptors and delivers it to its master. The data from the IMU is obtained in each case with the same subroutine, which reads from the serial buffer and writes the data to the I2C bus. The following pseudocode sheet and flow diagram picture the high level function of the UNO board.

**Data:** Read request  
**Result:** IMU data string sent to I2C Bus master

**Initialization**

if *Packet request from master or bumper activated* then
    Write #f through serial to request an IMU packet  
    Memorize IMU packet received  
    if *I2C Bus ready* then  
        Write the IMU packet received  
    end  
end

---

**Figure 5.3.:** Flow diagram for the UNO code
A firmware to obtain and process the information of the sensors has been customized and updated. It has a vast amount of utilities, but the next graph only displays a few relevant actions and the applications that are used by the UNO board to which it is connected.

**Data:** Request

**Result:** Write sensor data in serial buffer

Initialize I2C sensor addresses

*loop*

Read sensor data from I2C bus

if *Serial request* then

1. Write sensor data in Serial buffer

end

---

Figure 5.4.: Flow diagram for the 9DoF’s embedded Atmega328
5.3 ROS framework

5.3.1 Joystick control node

1. Servos: The range of the possible published messages has been updated to 30-150, which wasn’t possible before because the publishing of two of these messages with a difference in value of more than 100 in a short time difference would cause the MEGA to reboot as a byproduct of the lack of power available in the second power supply.

2. Brushes: This messages haven’t been altered. The control method remains the same, with the left joystick selecting an operation value and buttons 0-3 selecting to which brush is the message to be assigned.

3. Autoenable: Joy control mode includes the ability to publish a small message to turn on or off the automatic navigation function of the auto control node.

**joy_control:** It is described within the Pool cleaner robot package. It subscribes to a Joy node and publishes to /rightservo, /leftservo, /frontbrush, /leftbrush, /rightbrush a message within the rage these components operate, so the output is a linear function of the Joy message in each case. It has been updated to publish to rightservo and leftservo up to their respective full capacities, since the microncontroller doesn’t lose its power supply once a power spike is produced. It also publishes data to the /autoenable topic, which enables the automatic operation mode.

<table>
<thead>
<tr>
<th>Subscribed topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>/joy</td>
</tr>
<tr>
<td>sensor_msg/Joy</td>
</tr>
<tr>
<td>information provided from the joystick controller</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Published topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>/rightservo and /leftservo</td>
</tr>
<tr>
<td>std_msgs/UInt16</td>
</tr>
<tr>
<td>Transforms the joy_axis data to a suitable range</td>
</tr>
<tr>
<td>/rightbrush /leftbrush</td>
</tr>
<tr>
<td>std_msgs/UInt16</td>
</tr>
<tr>
<td>Commands the brushes</td>
</tr>
<tr>
<td>/ardrone/takeoff</td>
</tr>
<tr>
<td>std_msgs/Empty</td>
</tr>
<tr>
<td>sends the order of taking off</td>
</tr>
</tbody>
</table>

5.3.2 Imu publisher

**Imu messages:** The /chatter topic is publishing a ROS Imu message every 30ms through the xbee ros serial connection. It contains a quaternion message with 4 float64 entries, a linear acceleration /Vector3 message, an angular velocity /Vector3 message as well, and the covariance matrices for all the data, which are specified in the datasheet. There was a specific effort in publishing this kind of message format to improve de compatibility of with the other published ROS packages, such as robot pose ekf which can be fed with Imu msgs to help estimate the position of a robot with accuracy. This information is used within this project to make an accelerometer-data based collision detection.

<table>
<thead>
<tr>
<th>Published topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>/chatter/header</td>
</tr>
<tr>
<td>std_msgs/</td>
</tr>
<tr>
<td>Indicates the date of the message generation</td>
</tr>
<tr>
<td>/chatter/linear_acceleration</td>
</tr>
<tr>
<td>geometry_msgs/Vector3</td>
</tr>
<tr>
<td>Indicates x, y, and z components for linear acceleration</td>
</tr>
<tr>
<td>/chatter/angular_velocity</td>
</tr>
<tr>
<td>geometry_msgs/Vector3</td>
</tr>
<tr>
<td>Indicates x, y, and z components for angular velocity</td>
</tr>
<tr>
<td>/chatter/quaternions</td>
</tr>
<tr>
<td>std_msgs/float64</td>
</tr>
<tr>
<td>Indicates x, y, z and w components of Orientation quaternion data</td>
</tr>
</tbody>
</table>
The following console capture is an example of the output of the /chatter topic when the Imu is stationary:

```plaintext
header:
seq: 11940
secs: 1519139158
nsecs: 295808025
frame_id: "
orientation:
x: -0.050788/750864
y: 0.0318101420999
z: 0.173896/15045
w: 0.982938587666
orientation_covariance: [0.00249999, 0.0, 0.0, 0.0, 0.00249999, 0.0, 0.0, 0.0, 0.00249999]
angular_velocity:
x: -0.0499999976198
y: 0.019999999553
z: 0.00999999977648
angular_velocity_covariance: [0.019999, 0.0, 0.0, 0.0, 0.01999, 0.0, 0.0, 0.0, 0.0199999]
linear_acceleration:
x: 0.7844799757
y: 0.745409190655
z: 9.02151966095
linear_acceleration_covariance: [0.03999999, 0.0, 0.0, 0.0, 0.03999, 0.0, 0.0, 0.0, 0.039999]
```

5.3.3 Quaternions

The X, Y and Z coordinates of the quaternion represent the orientation. The W coordinate represents the roll axis.

5.3.3.1 Linear acceleration

This vector represents the acceleration that the body is experimenting in each of the three axis. In the case of the ROS message the Z vector that corresponds to the vertical axis has a positive value when aiming downwards instead of upwards. The data is specified in m/s*s units.
5.3.2 Angular velocity

\[
\text{angular velocity:} \\
x: -0.0499999976198 \\
y: 0.019999999553 \\
z: 0.00999999977648 \\
\text{angular velocity covariance: } [0.019999, 0.0, 0.0, 0.019999, 0.0, 0.0, 0.0, 0.0, 0.0199999]
\]

This data frame is published in rad/s.

5.3.4 Yaw, Pitch, Roll

Since the quaternion data that is published in the IMU topic is calculated from this data in the MEGA controller, it is cheap to publish and it might be useful later on as some of the packages in ROS use this data. Some commented lines are included in the arduino MEGA sketch that would perform this task when uncommented if it is needed for the compsumption of another ROS node that uses this data instead without the need of computing it again.

![Figure 5.5.: Yaw, Pitch and Roll][9]

5.3.5 YPR vs. Quaternions

Euler angles provide a simple way of understanding the orientation of the robot body, and they are the information that this IMU model usually provides, yet ROS framework uses Quaternions to refer orientation. Since ROS framework nodes are usually design to be reusable by other projects, the messages should be defined in the way that serves most purposes. Orientation quaternions might be harder to understand by a human mind, but they are more efficient to compute for a machine. Furthermore, the Euler angle system has an issue that doesn’t effect the SRO by its design, yet it can effect many robotic bodies, both underwater and aerial. The gimbal lock occurs when two of the rotation angles align because one rotated any multiple of 90° in relation to another one. If this happens, one degree of freedom is lost and operating over any of the superceded rotation angles would result on an error measurement on the other as well.

The Razor 9DoF makes an euler angle calculation from which the euler angles are computed in this project. This won’t be an issue for the concrete operation of the SRO, but could eventually be so in another similar instance in which the actuator component moved with some propulsion system instead of wheels.
5.3.6 Collision decision

This node decides whether or not the SRO Actuator component has collided with a wall or obstacle. To do so, it checks the data of its sensors as well as the user or autonavigation protocol input. The bumper information is the highest order decider since its simplicity and low chance of missreading make it the most reliable source. As it is implemented now, without a mechanical chassis that guarantees its activation upon collision processing over the other inputs can be made to detect the collision as well. A collision over the actuator component can be detected as well on a sudden change of the linear acceleration X and/or Y data of the IMU topic. Since there is no guarantee that the operation terrain will be flat or the position of the IMU enclosure stable, the Z axis is also considered in the calculation of the module of this acceleration.

The same event happens in the case the actuator component is just accelerating. To differ between these two events, the time in which the last instruction given to the servos topic by the joy control or auto control node is considered. There is a slight latency between the delivery of the instruction and the retrieval of the info from the IMU, which is considered when making this calculation. Roughly speaking: If the actuator receives an acceleration over the defined threshold and no servo message was published over ROS in a time interval bigger than the latency between the linear acceleration increase reading and the previous one, it is considered a collision. Therefore, the servo messages required to make the actuator component rotate are published and then the auto control node gives the instruction of moving in a straight line again. Here a graphic illustrating the time delays from publishing of the servo message to the reading of the corresponding IMU data:

![Latency between control input and IMU packet](image.png)

Figure 5.6.: Latency between control input and IMU packet
Figure 5.7.: Flow diagram to decide the collision

Since the latency between the two ros nodes $dT$ is way smaller than each of the other ones, the code on the decider must compensate for it. A pseudocode for the flow diagram:

---

**Data:** `/servo`, estimated delay, `/imu`  
**Result:** Decision on whether it is or it isn’t a collision  
Subscribe `/servo` and `/IMU`  

```
loop
    if Servo data published then
        Update time log of last servo data
    end
    if IMU data published then
        Update time log of last IMU data
        if Time of received IMU data > time of last servo data + estimated delay then
            Publish collision decided message
        end
    end
end
goto loop
```
### Collision_detect

<table>
<thead>
<tr>
<th>Subscribed topics</th>
<th>std_msgs/UInt16</th>
<th>Memorizes the last moment in which a command for movement was sent.</th>
</tr>
</thead>
<tbody>
<tr>
<td>/rightservo, /leftservo</td>
<td>std_msgs/UInt16</td>
<td>Reads the imu messages memorize the moment in which they are published and the linear acceleration data.</td>
</tr>
<tr>
<td>/chatter</td>
<td>sensor_msg/Imu</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Published topics</th>
<th>std_msgs/UInt16</th>
<th>Signals when the diagnosis protocol finds a collision.</th>
</tr>
</thead>
<tbody>
<tr>
<td>/collision_imu</td>
<td>std_msgs/UInt16</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1.: Detailed topic information about the collision_detect node

### 5.3.7 Autonomous navigation

This node publishes information so the actuator component will move accordingly and subscribes to the topics that give information on wether or not there is a collision event so it can change the direction of the actuator component in the case it is in front of a wall. It can be activated with the joystick controller or by console command.

![Figure 5.8.: Flow diagram for the autonomous navigation node](image)

Table 5.2.: Detailed topic information about the autonomous navigation node

<table>
<thead>
<tr>
<th>Subscribed topics</th>
<th>std_msgs/UInt16</th>
<th>Reads wether or not a collision is detected by the IMU and joystick data.</th>
</tr>
</thead>
<tbody>
<tr>
<td>/collision_imu</td>
<td>std_msgs/UInt16</td>
<td>Takes the activation of the bumper system as signal to make a rotation.</td>
</tr>
<tr>
<td>/waller</td>
<td>std_msgs/UInt16</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Published topics</th>
<th>std_msgs/UInt16</th>
<th>Sends servo messages in order to command rotations, accelerations and breaks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>/rightservo, /leftservo</td>
<td>std_msgs/UInt16</td>
<td></td>
</tr>
</tbody>
</table>
6 Communication

6.1 Mega-Uno link

The link between the two microcontrollers is established by I2C protocol due to the 4 wire connection between these two. In the previous instance of the project the characters of the IMU string would be requested one by one, but after updating the firmware of the 9DoF razor IMU this approach was changed: The MEGA will periodically request for a 64-byte package to the UNO, which will in turn provide the whole string containing the data from the sensors. This would usually not be possible since the arduino IDE imposes a restriction of a 32-byte buffer for each device, so the firmware in both devices was updated in order to fit the specification.

6.2 Laptop - MEGA (rosserial)

This connection between the two XBee modules gets started when the MEGA module is powered on and from a laptop running roscore a serial python node is initialized. This is the connection by which the laptop sends the proper commands to the MEGA module for it to control the servos and brushes, and by which the MEGA module publishes information on the ROS Imu, Bumper and Yaw-pitch-roll topics. XBees work exclusively in RX or TX mode, therefore any sensor package sent by the MEGA module gets corrupted if a control message published by the laptop gets sent simultaneously.
6.3 9DoF - Uno

As stated before, MEGA will request data from the UNO, not directly from the 9DoF module. The reaction of this request by the uno is to make a request through serial protocol to the onboard’s 9dof controller, by sending the f character each time an I2C request is received. It will save its data in char format and write in its own I2C buffer that was modified previously to be able to hold the whole package.

6.4 9DoF’s AtMega - Sensors

The 9DoF has an I2C link as a master to its sensors which is specified in its firmware under the tag 'sensors'. It systematically updates its data every 20 ms and applies the appropriate filtering and transforms to this data in order to publish it through serial protocol.

6.5 9DoF Razor IMU (Serial programming)

The firmware it contains is uploaded to its atmega328 module using the arduino IDE 1.8.5. This instance of the firmware doesn’t need any modification of the IDE. The FTDI protocol connection is quite restrictive when it comes to the upload of the firmware; It is theoretically possible to use the UNO board as a connector since it has the proper pins to achieve the connection, yet for the programming function or the upload of a bootloader, it is crucial to reset the Razor board in a crucially specific timing. The UNO board doesn’t provide such a signal, neither the SPI connector. Therefore, a specific connection FTDI to micro USB-B board is needed for the firmware updates and testing. To connect the controller to the Arduino IDE, the controller has to be selected as arduino Mini, and specify the controller to ATmega328 3.3V 8kHz.

![Figure 6.2.: Razor IMU 9DoF connected with the ftdi module](image)

In order to use this module, the pins should be connected to its pair with the same name except for the RX and TX, which are cross-paired.
6.6 Arduinos (Serial programming)

The programming of the arduino modules is done by the uploading of the corresponding sketches by arduino IDE 1.8.5, which has to be modified slightly in some of the headers to be able to enable the communications as are specified in the final sketches: The i2C protocol in atmega chips has a default buffer of 32 bytes, but at least 64 are needed for the proper operation of the system as it is now programmed. Therefore, the proper paramater has to be updated in the Arduino IDE 1.8.5 installation folder.

6.7 9DoF Razor IMU (Serial calibration)

The AHRS firmware that the 9Dof Razor IMU works with includes a function for callibration of each sensor that works with all of the different versions of this module that sparkfun sells. To callibrate the sensors, the same connection link as in the programming section has to be plugged: The FTDI module connected to a micro-USB cable plugged to a computer running arduino 1.8.5, with the same configuration (Select arduino mini, ATmega 328 3,3V 8KHz). To enter callibration mode the option message #oc has to be sent through the serial console, running on the same baudrate that the 9DoF is programmed to run on (57600 baud for this version).
7 Data processing

7.1 DCM algorithm for sensor data

DCM stands for Direction Cosine Matrix. It performs a similar function that of a Kalman filter, but its computational cost is smaller and it makes it more suitable for the AtMega available in the 9DoF Razor Imu. The Direction Cosine Matrix algorithm grants the means to fuse the accelerometer, magnetometer and gyroscope data, as well as processing sensor noise and math errors. The original paper on this method is linked at the bibliography, and was written by William Premerlani.

There are four specific functions in the AHRS’s firmware describing DCM’s functionality:

7.1.1 Normalize

Basically, it tackles the need of an orthonormal matrix over time. The rotation on the Imu causes a rotation in readings, and doing so with discrete math causes small errors which accumulate over time, being the consequence non-orthogonal vectors over time. If these vectors were allowed to accumulate error, the posterior readings which are function of math operations over them would increase its error as well.

7.1.2 Drift Correction

Compensates for the Roll, Pitch and Yaw drift. For the Roll and pitch data, it performs a ponderated calculation with the module of the accelerometer data: It decreases its reliability when it doesn’t comply its reliability constraints. For the Yaw data, it makes reading over the heading of the magnetometer to perform a small filtering.

7.1.3 Matrix Update

It updates the values of the cosine matrix with the last readings of the accelerometer and gyroscope and applies the proportional and integrator control terms. Then it performs the drift correction function over the desired terms and updates the matrix with the processed and controlled values.

7.1.4 Euler angles

Makes the calculation of the yaw, pitch, and roll values in function of the Direction Cosine Matrix values. The quaternion data is not obtained from this calculation, it’s operated in a later step on the MEGA controller. The followed procedure is explained in the next section.

7.2 Obtaining quaternions from YPR data

As the firmware operates in the current instance of the firmwares that are updated, the data structure that arrives to the MEGA controller contains Yaw, Pitch, Roll, linear acceleration and angular velocity. The ROS framwork’s default Imu message requires quaternions instead of the Yaw, Pitch and Roll data. The quaternion matrix can be obtained as a matrix transform over the Yaw, Pitch and Roll data. The
person that updated the firmware of the 9DoF to publish ROS-compatible packets over serial instanced the transformation in a ROS node that unluckily is not compatible with our system because it is published through the FTDI-USB port and has slight differences in its format. The current algebraic transformation from the Yaw, Pitch, and Roll data is operated on the MEGA board, but if the project requirements were different in a way which caused more computation on the MEGA controller (for example publishing images) it can be outsourced to one of the two ATMega328 microcontrollers of the actuator module in a previous step: The decision of performing the calculation on the MEGA was taken because it was the easiest and simplest. In order to perform the computation, these steps were followed:

\[
q.w = \left[ \cos(\psi/2) \cos(\phi/2) \cos(\theta/2) + \sin(\psi/2) \sin(\phi/2) \sin(\theta/2) \right]
\]

\[
q.x = \left[ \cos(\psi/2) \sin(\phi/2) \cos(\theta/2) - \sin(\psi/2) \cos(\phi/2) \sin(\theta/2) \right]
\]

\[
q.y = \left[ \cos(\psi/2) \cos(\phi/2) \sin(\theta/2) + \sin(\psi/2) \sin(\phi/2) \cos(\theta/2) \right]
\]

\[
q.z = \left[ \sin(\psi/2) \cos(\phi/2) \cos(\theta/2) - \cos(\psi/2) \sin(\phi/2) \sin(\theta/2) \right]
\]

Where Yaw = ψ, Pitch = φ, Roll = θ

These q.w, q.x, q.y, and q.z results can directly be assigned to the Imu message instance.

7.3 Publishing over a ROS Imu data topic

Since the previous work by the firmware creators was developed with the idea of letting the ROS framework operate the received data to create the IMU message and they used the FTDI connector to obtain this data, an alternative method to do so had to be developed. In this case, due to the structure of the software that we already had, it was preferable to organize the packet internally and let the MEGA directly publish the packet over the XBee serial module. The MEGA controller will repeatedly over intervals of time bigger than 20ms ask for a 64 byte data packet to its UNO slave device on the direction 8. The uno then will send ‘#f’ through its serial port where the Razor 9DoF IMU is connected. This triggers the output function of the Razor 9DoF Imu, which will publish a string of chars with the following format:

\[
YPR, AV, LA = [\# c c c c c c . . . $]
\]

Being each one of the characters one digit of each of the 9 floats that are read by the sensors, looking similar to, for example, to the array between the next brackets:

\[
YPR, AV, LA = [\#190.5, -0.05, -1.2, -0.05, -0.01, -0.01, , -1.2, -0.05, -0.01$]
\]

As this is a char array and it is of variable length in function of the readings, a small routine to inform the reader of the array size and send always an array of the appropriate length is used, otherwise it would be sending empty characters in most of the read instances.

Each character that arrives to the MEGA is processed to detect relevant information. The first character to be interpreted, is either an ‘n’, indicating that no bumper was pushed, a ‘w’, indicating that the front bumper was pushed, or an ‘s’, indicating that one of the back bumpers was pushed. This will trigger the publishing of a ROS message on the /waller topic, containing a Uint16 that has this information encoded. (0 for n, 1 for w and 2 for s). The reading of an # character will trigger the memorization of the following chars up until the $ character reading. The $ character reading will then trigger the printing function,
which is designed to generate an IMU message. First, a ten position float buffer will be declared. Then, a routine converts the buffer data converting each string of chars separated by the ‘,’ character to the corresponding float value. The first three, which correspond to the euler angles (Yaw, Pitch, Roll) are processed as explained in the previous section. Additionally, to publish the imu topic coherent with ROS standards, some of the axis are inverted as they point in different directions when published by the IMU. This should not be subsidised to the deeper layers (the embedded atmega in the Razor IMU) Because it uses a different format for its callibration.

7.4 Plotting collisions

![Collision graph comparison: With and without ROS input](image)
Both images are de rqt plot of the data received from the Imu that corresponds to the x and y coordinates of the linear acceleration in red and blue, the green signal corresponding to the conclusion of the system regarding if the acceleration is a collision or a commanded acceleration, and some /rightservo topic data in dark blue. As it’s clear in the image, the collision flag jumps just when an acceleration is provided and there was no recent comment from any servo topic.

Figure 7.2.: Detail of the collision reaction

On a test without subscribing to the servo data to keep a better scale we can see that the green collision flag reacts on the module of the acceleration vectors independently of the direction or component that has suddenly increased. This reaction is flagged way before the peak of such acceleration is reached: This test
is made with a low threshold value for collision detection which hasn’t yet been tested in real conditions. Since the system is intended to work underwater instead, it is likely that a water current or any condition would trigger it, thus requiring a higher threshold value to operate properly. As it’s obviously clear as well: the sample rate for the imu is way below the optimal, and it could use some interpolation postprocessing and/or additional sampling. The following figure illustrates the difference in four consecutive IMU data frames when a collision happens:

```
header:
  seq: 12232
  secs: 1519139188
  nsecs: 727322948
  frame_id: "
  orientation:
    x: -0.0577562712133
    y: 0.0160621609539
    z: 0.459960073233
    w: 0.885913610458
  orientation_covariance: [0.00249999, 0.0, 0.0, 0.0, 0.00249999, 0.0, 0.0, 0.0, 0.002499]
  angular_velocity:
    x: -0.0499999970198
    y: 0.019999999553
    z: 0.00999999977648
  angular_velocity_covariance: [0.019999, 0.0, 0.0, 0.0, 0.01999, 0.0, 0.0, 0.0, 0.019999]
  linear_acceleration:
    x: 8.7844799757
    y: 0.745409190655
    z: 9.02151966095
  linear_acceleration_covariance: [0.03999, 0.0, 0.0, 0.0, 0.03999, 0.0, 0.0, 0.0, 0.03999]

header:
  seq: 12233
  secs: 1519139188
  nsecs: 832322948
  frame_id: "
  orientation:
    x: 0.0563634186983
    y: 0.0189662761986
    z: 0.648499965668
    w: 0.75888800621
  orientation_covariance: [0.02499999, 0.0, 0.0, 0.0, 0.00249999, 0.0, 0.0, 0.0, 8.802499999]
  angular_velocity:
    x: -0.159999996424
    y: 0.379999995232
    z: -5.68999965668
  angular_velocity_covariance: [0.01999999, 0.0, 0.0, 0.0, 0.01999, 0.0, 0.0, 0.0, 0.019999]
  linear_acceleration:
    x: -4.90299987793
    y: 3.60868453979
    z: 10.5119552612
  linear_acceleration_covariance: [0.0399993, 0.0, 0.0, 0.0, 0.03999993, 0.0, 0.0, 0.0, 0.0399999]
```
header:
seq: 12234
secs: 1519139188
nsecs: 936322948
frame_id: "
orientation:
x: -0.054534599185
y: 0.0370494835079
z: 0.633054435253
w: 6J71294474607
orientation_covariance: [0.00249999, 0.0, 0.0, 0.0, 0.0149999, 0.0, 0.0, 0.0, 0.00249999]
angular_velocity:
x: -6.119999997318
y: -0.089999961257
z: 0.849999964237
angular_velocity_covariance: [0.019999, 0.0, 0.0, 0.0, 0.019999, 0.0, 0.0, 0.0, 0.019999]
linear acceleration:
x: 0.588359951973
y: 1.60803067684
z: 9.17856884803
linear_acceleration_covariance: [0.039999, 0.0, 0.0, 0.0, 0.039999, 0.0, 0.0, 0.0, 0.039999]

header:
seq: 12235
secs: 1519139189
nsecs: 41322948
frame_id: "
orientation:
x: -0.041890218854
y: 0.022817607969
z: 0.577792227268
w: 0.814788758755
orientation_covariance: [0.00249999, 0.0, 0.0, 0.0, 0.00249999, 0.0, 0.0, 0.0, 0.00249999]
angular_velocity:
x: -0.070000000298
y: -0.01999999553
z: 0.0999999940395
angular_velocity_covariance: [0.019999, 0.0, 0.0, 0.0, 0.019999, 0.0, 0.0, 0.0, 0.019999]
linear acceleration:
x: 1.01967072487
y: 0.627430737019
z: 9.49228382111
linear_acceleration_covariance: [0.019999, 0.0, 0.0, 0.0, 0.019999, 0.0, 0.0, 0.0, 0.039999]
Part III.
Results
8 Testing

8.1 Sudden break

Whenever a script or a joystick user sends a sudden acceleration against the actual velocity vector of the actuator component, if a lot of power to do the hard break is required, the power supply attached to the servos loses its power, and there is a small time window in which it doesn’t react to the ROS commands that the MEGA receives and uses to act on the servo through the drivers. However, the communication isn’t lost anymore, so the sensors and communication link can keep working during this time window. Furthermore, the automatic navigation scripts do not use the highest possible velocity at any time since it is pretty fast and it wouldn’t even be useful when trying to brush the floor.

8.2 Connection reliability

The connection remains stable for an arbitrary amount of time but there are few conditions that will break it. Wrong readings from the IMU, a malfunction on its firmware, or a bad package sent by the UNO will usually cause it to break. This conditions are observable for example when the loop in the MEGA sketch tries to get too many packets from the actuator component; The actuator component microcontrollers need a theoretical minimum of 20 ms to make a valid reading on the IMU. However, in practical terms, this latency is slightly bigger. When the wrong packet condition is accomplished and the connection is broken, it can take up to 30 seconds or more to make an automatic recovery for the link; therefore it is quite convenient to suppress any preventable breaks.

8.3 Sensor data

No inconsistencies were found on the bumper system readings in any case, and they have virtually no measurable latency when making readings from serial with the UNO board, as expected. However, the IMU, that publishes on command as specified in the section before explaining its behaviour (Chapter 5), since it operates in this way has a maximum theoretical delay of 20 ms to the UNO every time a data packet is demanded. A high frequency of update by the master device to the UNO board will cause it to generate errors as well; for a master request rate of 50 packets per second, the IMU generates a wrong package on an average time of about 20 seconds. When a master device requests 40 packets per second, it takes around 2 minutes to generate this faulty package. Down to an update rate of 30 ms (33 packets per minute) There is virtually no failure rate detected. This kind of misreading should theoretically not be a huge problem, yet it causes the communication from rosserial to break as specified in the last section.

8.4 Navigation

The performance during the manual control and autonomous control will be evaluated in the two next sections and additionally, an estimate on the performance of the autonomous navigation.
8.4.1 Joy control

When using the joystick to control the SRO’s movement, it remains totally operative if any wrong packet is generated by the MEGA. If the connection link with the computer is broken, it keeps operating as in the last packet sent by ROS until the link is reestablished and a new packet is published: The joy_control node by its nature publishes data at a high rate, and the XBee modules by which the connection is established only support one simultaneous operative mode. Therefore, the occupation of the serial link makes for it impossible to receive IMU or other data packages when a message is published through this link. So if lots of different accelerations or brush velocities are asked by the user in a short period of time, lots of sensor data will be lost.

8.4.2 Autonomous control

For the autonomous navigation of the SRO, a ROS node to detect the collisions (wall detection) and another one to react in function of the output of this first node are initialized. At 57600 baudrate on the XBee connection, there is a substantial bottleneck when publishing the turning instructions while more IMU packets are received. This causes delays and therefore irregular rotations for each detected collision. Being a randomized navigation process, it is not necessarily bad, yet it’s worth noting that the pre-scripted rotation instruction will not be executed consistently through every iteration. The SRO’s wheels are prone to drifting when too much force is applied to them. Effective turns are performed when a servo value of '135' is published to one of the servos and a servo value of '55' is published simultaneously on the other servo for ground navigation. Underwater navigation remains untested. A non-conclusive value of 20 IMU samples each second proved to be the most consistent for collision detection due to the bandwidth constraints, which is under the maximum of 33 in which it can operate.

Figure 8.1.: SRO lab testing setup

Figure 8.2.: Typical range of rotations
9 Conclusion

9.1 Conclusion

1. The aim of this project was to upgrade the SRO pool cleaner to be an operative stable device and perform the necessary reparations for both the physical and software issues. Such upgrade has been successfully performed. However, the substitution or upgrading of the second power supply, an upgrade to the XBee link transfer data rate and the possibility to acquire more IMU data would be useful in order to polish the robot.

2. The communication link between the MEGA controller and the XBee on the laptop running ROS is now stable and doesn’t reboot when a high voltage spike is produced by a sudden break.

3. The data obtained from the sensors is processed and published coherently with the ROS message standard, being compatible with packages that use such information to estimate position such as pose estimate ekf. The addition of more IMU modules would ease the acquisition of data and add the possibility to perform some post-processing for better quality data.

4. The IMU’s firmware was successfully updated coherently with the project requirements, and the piece of circuitry needed for the addition of a bumper system welded on the new board as well as the treatment of the data in case of its activation.

5. Processing over the obtained data is performed in order to correct the actuator’s trajectory when collision happens. This allows autonomous unsupervised navigation. It’s worth noting that some inconsistencies remain and this requires further development.

6. The ultimate goal of giving it a capability for intelligent autonavigation is pending, since the sensing devices lack the ability to update the information on the absolute position of the system, therefore, not being able to localize itself. With the current system, a performance similar that of an old roomba model can be obtained, with the additional value of the option for manual control.

7. The research concluded that waterproofing the actuator component is viable and there exist enclosures able to waterproof the UNO module and sensors by the required (IP68) standard. After the acquisition of 5 different modules, none of them could fit the UNO board without a small modification of the UNO’s hardware.
To achieve the ultimate goal of the SRO pool cleaner system the means to grant absolute location of the actuator component must be achieved. Since the gps readings are not viable underwater, gps for absolute positioning should be added to the communication component, and some means of obtaining its position relative to the actuator. This could be attained by computer vision. Since most of the similar systems use odometry data to update its location, some kind of sensor performing this function could be added to the actuator component. A visual odometry solution would be most likely not viable due to the lack of bandwidth of the communication to actuator microcontroller components, nonetheless, if a trustworthy signal range finder is found and can be adapted to the system its data can be made accessible and it would be useful for both positioning and mapping. A chassis that activates the bumper buttons could be easily attached as well since the electronical and software capes are already implemented. Since an additional IMU module compatible with the actual structure was added, its addition to the actuator component over the UNO board or another position could be useful to apply filtering techniques and improve the reliability, as well as getting more samples of IMU data since the current obtainable sample rate is slow and is limited by the Atmega328 controllers. It could aid in absolute positioning as well if located on the transmitter component in addition of the fiducial markers/camera proposed in the last phase of the project. It’s worth noting as well that for an actuator component with propulsion-based navigation which would be susceptible to Gimbal lock the firmware of the 9DoF Razor IMU should be updated to obtain the Quaternion data directly from the sensors.

![Small sensor stick on the 3.3V I2C bus and a waterproof junction box](image)

Waterproofing the actuator component is viable with the current setup if a slight modification on the Arduino UNO board is done. This small enclosure guarantees waterproof protection up to an IP68, which is the required for the actuator component. The USB port on the UNO is not required for its function. When a final sketch doing all the processing needed on the additional sensors is uploaded to it, it can be removed or relocated in a way in which it won’t obstruct the board.
Part IV.
Apendices
#include <Wire.h>  
#ifndef SOFTWARE_SERIAL
#include <SoftwareSerial.h>
#define digitalPinToInterrupt(p) ((p) == 2 ? 0 : ((p) == 3 ? 1 : NOT_AN_INTERRUPT))

SoftwareSerial mySerial(12,13);  //RX,TX
char incoming;
char w;
char s;
int count = 0;
int start = 64;
int x = 0;
int i = 0;
char imu[64];
volatile int buttonState1=0;
volatile int buttonState2=0;

void setup() {
  //mySerial.begin(57600);
  Serial.begin(57600);
  Wire.begin(8);
  Wire.onRequest(requestEvent);
  Wire.onReceive(receiveEvent);
  pinMode(2, INPUT);
  pinMode(3, INPUT);
  attachInterrupt(digitalPinToInterrupt(2), pin_ISR, CHANGE);
  attachInterrupt(digitalPinToInterrupt(3), pin_ISR, CHANGE);
}

void loop() {
  //if (start==0){
  //  delay(1000);
  //}
  // start=1;
  // }
  delay(100);

  /*Serial.write("#f");
  x=Serial.available();
  while(Serial.available()>0){
    i=Serial.available();
    imu[x-i]=Serial.read();
    //Serial.print(imu[i]);
  }*/
}

void requestEvent(){}
Serial.write("#f");
Wire.write(w);
Wire.write(s);
x=Serial.available();
while(Serial.available()>0){
i=Serial.available();
imu[x-i]=Serial.read();
Wire.write(imu[x-i]);
}
/* if (imu[count]!="#"){
Wire.write(imu[count]);
count++;
} else{
Wire.write(imu[count]);
count=0;
}*/

/*
void receiveEvent(){
 while(Wire.available()>0){
 char c = Wire.read();
 // mySerial.write(c);
 }
}*/

/*void pin_ISR() {
 buttonState1 = digitalRead(2);
 if (buttonState1){
  w='w';
 } else{
  w='e';
 }
 buttonState2 = digitalRead(3);
 if (buttonState2){
  s='s';
 } else{
  s='a';
 }*/

9.3.2 Arduino Mega code

//Required Libraries
//Initializing Node Handler to use Rosserial
ros::NodeHandle nh;

//Initializing Publisher with IMU message
sensor_msgs::Imu imu_msg;
ros::Publisher chatter("chatter", &imu_msg);
/*mav_msgs::RollPitchYawrateThrust RollPitchYawrateThrust_msg;
ros::Publisher YPR("YPR", &RollPitchYawrateThrust_msg);*/
std_msgs::UInt16 UInt16_msg;
ros::Publisher waller("waller", &UInt16_msg);
char frame_id[7] = "imu";
char wall_id[7] = "waller";

//Global Variables
char c;
char buffer[75];
int i = 0;
int found = 0;
int wall = 0;
int wallb = 0;
float pi=3.14159;
float yaw_deg = 0;
float d2rad=pi/180.0;
float yaw=0;
float pitch=0;
float roll=0;

//Initializing Servo motor classes
Servo rightservo;
Servo leftservo;
Servo frontbrush;
Servo rightbrush;
Servo leftbrush;

/* The following functions receive a subscribed message and
actuate the appropriate Servos accordingly */
void rightservo_cb( const std_msgs::UInt16& cmd_msg) {
  rightservo.write(cmd_msg.data); //set servo angle, should be from 0-180
}

void leftservo_cb( const std_msgs::UInt16& cmd_msg) {
  leftservo.write(cmd_msg.data);
}
void frontbrush_cb(const std_msgs::UInt16& cmd_msg) {
    frontbrush.write(cmd_msg.data);
}

void rightbrush_cb(const std_msgs::UInt16& cmd_msg) {
    rightbrush.write(cmd_msg.data);
}

void leftbrush_cb(const std_msgs::UInt16& cmd_msg) {
    leftbrush.write(cmd_msg.data);
}

//Initializing Subscriber for each Servo motor
ros::Subscriber<std_msgs::UInt16> sub1("rightservo", rightservo_cb);
ros::Subscriber<std_msgs::UInt16> sub2("leftservo", leftservo_cb);
ros::Subscriber<std_msgs::UInt16> brush1("frontbrush", frontbrush_cb);
ros::Subscriber<std_msgs::UInt16> brush2("rightbrush", rightbrush_cb);
ros::Subscriber<std_msgs::UInt16> brush3("leftbrush", leftbrush_cb);

void setup() {  
    //Initializing pin modes for motors
    pinMode(2, OUTPUT);
    pinMode(3, OUTPUT);
    pinMode(4, OUTPUT);
    pinMode(5, OUTPUT);
    pinMode(6, OUTPUT);

    //Set Baud Rate to communicate with Rosserial and Robot
    nh.getHardware()->setBaud(57600);
    //Initialize Publisher and Subscribers
    nh.initNode();
    nh.advertise(waller);
    nh.advertise(chatter);
    //nh.advertise(YPR);
    nh.subscribe(sub1);
    nh.subscribe(sub2);
    nh.subscribe(brush1);
    nh.subscribe(brush2);
    nh.subscribe(brush3);
    //Link Servo classes to appropriate pins
    rightservo.attach(2);
    leftservo.attach(3);
    frontbrush.attach(4);
    rightbrush.attach(5);
    leftbrush.attach(6);

    //Begin I2C communication with slave arduino
    Wire.begin();
    //Serial.begin(57600);
}

/* This function takes char data acquired from the Slave Arduino, 
splits it by commas, converts the char characters to integers, 
and assigns it as part of the IMU msg. */
void printing() {
    // Acquire time stamp
    imu_msg.header.stamp = nh.now();
    // RollPitchYawrateThrust_msg.header.stamp = nh.now();
    // Split buffer string by data and convert to integer
    i = 0;
    char *pt;
    float measure[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};
    pt = strtok(buffer, ",");
    while (pt != NULL) {
        float a = atof(pt);
        measure[i++] = a;
        pt = strtok(NULL, ",");
    }
    yaw_deg = measure[0];
    if (yaw_deg > 180.0) {
        yaw_deg = yaw_deg - 360.0;
    } else if (yaw_deg < -180.0) {
        yaw_deg = yaw_deg + 360.0;
    }
    yaw = yaw_deg * d2rad;
    pitch = measure[1] * d2rad;
    roll = measure[2] * d2rad;
    float t0 = cos(yaw * 0.5);
    float t1 = sin(yaw * 0.5);
    float t2 = cos(pitch * 0.5);
    float t3 = sin(pitch * 0.5);
    float t4 = cos(roll * 0.5);
    float t5 = sin(roll * 0.5);
    /* RollPitchYawrateThrust.msg.roll = roll;
    RollPitchYawrateThrust_msg.pitch = pitch;
    RollPitchYawrateThrust_msg.yaw_rate = yaw;
    YPR.publish(&RollPitchYawrateThrust_msg); */
    imu_msg.orientation.x = t0 * t3 * t4 - t1 * t2 * t5;
    imu_msg.orientation.y = t0 * t2 * t5 + t1 * t3 * t4;
    imu_msg.orientation.z = t1 * t2 * t4 - t0 * t3 * t5;
    imu_msg.angular_velocity.x = measure[6];
    imu_msg.angular_velocity.y = -measure[7];
    imu_msg.angular_velocity.z = measure[8];

    float orientation_covariance[9] = {0.0025, 0, 0, 0.0025, 0, 0, 0, 0.0025};
    imu_msg.orientation_covariance[0] = orientation_covariance[0];
    imu_msg.orientation_covariance[4] = orientation_covariance[4];
    imu_msg.orientation_covariance[8] = orientation_covariance[8];

    float angular_velocity_covariance[9] = {0.02, 0, 0, 0.002, 0, 0, 0, 0.02};
    imu_msg.angular_velocity_covariance[0] = angular_velocity_covariance[0];
imu_msg.angular_velocity_covariance[4] = angular_velocity_covariance[4];
imu_msg.angular_velocity_covariance[8] = angular_velocity_covariance[8];

float linear_acceleration_covariance[9] = {0.04, 0, 0, 0, 0.04, 0, 0, 0, 0.04};
imu_msg.linear_acceleration_covariance[0] = linear_acceleration_covariance[0];
imu_msg.linear_acceleration_covariance[4] = linear_acceleration_covariance[4];
imu_msg.linear_acceleration_covariance[8] = linear_acceleration_covariance[8];

// Publish the IMU msg
chatter.publish(&imu_msg);

nh.spinOnce();
// Reset to acquire new data
i = 0;
memset(buffer, 0, 75);
}

void loop() {
// Get data from slave arduino
delay(50); // Less than 50 ms for the break compromise the collision detection. Less than 30
make the system fail often.
Wire.requestFrom(8, 64); // request a whole imu package

while (Wire.available()){
    char c = Wire.read();

    // Serial.print(c);
    // Identify relevant data
    if (c == '#') {
        found = 0;
        printing();
    }

    if (found == 1) {
        buffer[i++] = c;
        // Serial.println(c);
    }
    if (c == '$') found = 1;
    if (c == 'w') {
        UInt16_msg.data = "0";
        waller.publish(&UInt16_msg);
    }
    if (c == 's') {
        UInt16_msg.data = "1";
        waller.publish(&UInt16_msg);
    }
    if (c == 'n') {
        UInt16_msg.data = "2";
        waller.publish(&UInt16_msg);
    }
}
9.3.3 Wall detection ROS node

```python
#!/usr/bin/env python
import rospy
import math
from std_msgs.msg import UInt16
from std_msgs.msg import String
from std_msgs.msg import Float64
from sensor_msgs.msg import Imu
from geometry_msgs.msg import Vector3

#pub_wallvector = rospy.Publisher('/wallvector', Float64, queue_size=1)
pub_collision = rospy.Publisher('/collision_imu', UInt16, queue_size=1)

dltime=0
dlv=0
dltime=0
dl1v=0
x1=0
y1=0
z1=0
th=2
delay=0
nt=0
delayf=0
ntf=0
rlftime=0

def find_wall(data):
    x=data.linear_acceleration.x
    y=data.linear_acceleration.y
    z=data.linear_acceleration.z
    global imutime
    global delay
    global nt
    global delayf
    global ntf
    delay=rospy.Duration(1,0)
delayf=delay.to_sec()
    global rltime
    global x1
    global y1
    global z1
    p=math.sqrt(x*x+y*y+z*z)
p1=math.sqrt(x1*x1+y1*y1+z1*z1)
    if (p-p1>th):
        rospy.loginfo("Ouch!!")
        nt=rospy.Duration(0,0)
        ntf=nt.to_sec()
```

nttf=ntf-rlftime
if (nttf>delayf):
    rospy.loginfo("collision candidate")
collision_msg=UInt16()
collision_msg.data=1
pub_collision.publish(collision_msg)

x1=x
y1=y
z1=z

def rservotime(data):
    global rltime
    global rlv
    global rlftime
    rltime=rospy.get_rostime()
    rlftime=rltime.to_sec()
    rlv=data.data
    rospy.loginfo(rltime)

def lservotime(data):
    global lltime
    global llv
    lltime=rospy.get_rostime()
    llv=data.data
    rospy.loginfo("Current time")

if __name__ == '__main__':
    rospy.init_node('wall_detect', anonymous=True)
    #rospy.Subscriber("/waller", UInt16, find_bump)
    rospy.Subscriber("/chatter", Imu, find_wall)
    rospy.Subscriber("/rightservo", UInt16, rservotime)
    rospy.Subscriber("/leftservo", UInt16, lservotime)
    rospy.spin()

9.3.4 Autonomous navigation node

#!/usr/bin/env python
import rospy
from std_msgs.msg import UInt16
from std_msgs.msg import String
from sensor_msgs.msg import Imu
from sensor_msgs.msg import Joy

pub_motor0 = rospy.Publisher('/rightservo', UInt16, queue_size=10)
pub_motor1 = rospy.Publisher('/leftservo', UInt16, queue_size=10)
pub_motor2 = rospy.Publisher('/frontbrush', UInt16, queue_size=10)
pub_motor3 = rospy.Publisher('/leftbrush', UInt16, queue_size=10)
pub_motor4 = rospy.Publisher('/rightbrush', UInt16, queue_size=10)

def bumper(data):
    w = data.data

    brush_msg = UInt16()
motor_msg = UInt16()
    if w=='0':
        motor_msg.data = int(70)
        pub_motor0.publish(motor_msg)
        motor_msg.data = int(70)
        pub_motor1.publish(motor_msg)
    if w == '1':
        motor_msg.data = int(71)
        pub_motor0.publish(motor_msg)
        motor_msg.data = int(70)
        pub_motor0.publish(motor_msg)
    if w == '2':
        motor_msg.data = int(100)
        pub_motor0.publish(motor_msg)
        motor_msg.data = int(100)
        pub_motor0.publish(motor_msg)


def imu(data):
    w=data.data
motor_msg = UInt16()
    if w==0:
        motor_msg.data = int(70)
        pub_motor0.publish(motor_msg)
        motor_msg.data = int(70)
        pub_motor1.publish(motor_msg)
    if (w==1):
        motor_msg.data = int(110)
        pub_motor0.publish(motor_msg)
        motor_msg.data = int(100)
        pub_motor0.publish(motor_msg)
        rospy.sleep(1)
        motor_msg.data = int(70)
        pub_motor0.publish(motor_msg)
        motor_msg.data = int(70)
        pub_motor1.publish(motor_msg)
    if w == 2:
        motor_msg.data = int(100)
        pub_motor0.publish(motor_msg)
        motor_msg.data = int(100)
        pub_motor0.publish(motor_msg)
if __name__ == '__main__':
    rospy.init_node('auto_control', anonymous=True)
    rospy.Subscriber("/waller", UInt16, bumper)
    rospy.Subscriber("/collision_imu", UInt16, imu)
    rospy.spin()

9.3.5 Joy node

#!/usr/bin/env python
import rospy
from std_msgs.msg import UInt16
from sensor_msgs.msg import Imu
from sensor_msgs.msg import Joy

pub_motor0 = rospy.Publisher('/rightservo', UInt16, queue_size=10)
pub_motor1 = rospy.Publisher('/leftservo', UInt16, queue_size=10)
pub_motor2 = rospy.Publisher('/frontbrush', UInt16, queue_size=10)
pub_motor3 = rospy.Publisher('/leftbrush', UInt16, queue_size=10)
pub_motor4 = rospy.Publisher('/rightbrush', UInt16, queue_size=10)

def joy_callback(data):
    awk = data.axes[0]
    axel = data.axes[1]
    axe2 = data.axes[2]
    axe3 = data.axes[3]

    button0 = data.buttons[0]
    button1 = data.buttons[1]
    button2 = data.buttons[2]
    button3 = data.buttons[3]
    button4 = data.buttons[4]
    button5 = data.buttons[5]
    button6 = data.buttons[6]

    brush_msg = UInt16()
    motor_msg = UInt16()

    brush_msg.data = int((axe1 + 1)*40 + 50)
    if (axe1<0.01) and (axe1>(-0.01)):
        brush_msg.data = 95
    if button3:
        pub_motor2.publish(brush_msg)
    if button2:
        pub_motor4.publish(brush_msg)
    if button0:
        pub_motor3.publish(brush_msg)

    ## Wheel Motors
    alpha = 1
    beta = 1
    vr = alpha*axe3 + (beta*axe2)
vl = alpha*axe3 - (beta*axe2)

r = (2-(vr+1))*(60) + 30

l = (2-(vl+1))*(60) + 30

if (r<30):
    r = 30
if (l<30):
    l = 30
if (r>150):
    r = 150
if (l>150):
    l = 150

motor_msg.data = int(r)
pub_motor0.publish(motor_msg)
motor_msg.data = int(l)
pub_motor1.publish(motor_msg)

## Stop All Motors
if button1:
    brush_msg.data = 95
    motor_msg.data = 90
    pub_motor0.publish(motor_msg)
    pub_motor1.publish(motor_msg)
    pub_motor2.publish(brush_msg)
    pub_motor3.publish(brush_msg)
    pub_motor4.publish(brush_msg)

##

# Publish the commands to the motor topics
motor_msg = UInt16()
motor_msg.data = axe0
pub_motor0.publish(motor_msg)

if __name__ == '__main__':
    rospy.init_node('joy_control', anonymous=True)
    rospy.Subscriber("/joy", Joy, joy_callback)
    rospy.spin()
9.3.6.1 General structure

The AHRS firmware was originally developed by Peter Bartz, from sparkfun’s team. The ROS developer "Kristof Robot" adapted it to work under ROS. [8]

This version he customized provides significantly better output formats for our application, as well as another feature which is currently not relevant for our application but it could be in the future: It is operative with the newest versions of the Razor 9DoF, which is called M0 and includes a different microcontroller, way more pin input/output options and means to program it by USB directly. It will definitely come handy for the repair if the actual module ever breaks. This update is supposed to operate with the IMU directly connected with an FTDI-USB adapter to the computer, so in order to adapt it to our project even further customization was done.

The firmware parts are:

1. **Razor_AHRS.ino**: The main instance. It contains all the configuration parameters and the message protocols; therefore a few updates had to be done for the operation.

2. **DCM.ino**: It includes the Direction Cosine Matrix algorithm.

3. **math.ino**: It includes the definitions of concurrent operations in the DCM algorithm such as dot product, vector product etc.

4. **sensors.ino**: It includes all the I2C addresses of the sensors and subroutines to obtain their data.

5. **output.ino**: It contains functions to publish the collected and processed data over serial. Since the custom firmware treated differently the way in which the ROS Imu message was generated, a few further customization had to be done in this sketch as well for the proper function of the overall system.

6. **compass.ino**: It provides treatment for the magnetometer data.

9.3.6.2 Razor AHRS.ino updates:

This firmware contains concrete routines for compatibility with any of the old and newer versions of the Razor 9DoF Imu and the new module that is currently commercialized instead (9dof M0) and it can be easily changed modifying the configuration headers in this sketch. Furthermore, since the operation of the Imu module in this project is far from the usual, the default operation modes needed to be changed as well.

9.3.6.3 Output.ino updates:

In the custom ROS AHRS firmware provided by Y, there is an output function to write yaw, pitch and roll by serial and another one to write the gyroscope and accelerometer data. Since his ROS project works with the Imu module directly connected to the computer, he uses one ROS node running on it to make the proper calls in order to first obtain the Yaw, Pitch and Roll data, then the gyro and accelerometer data, then use a ROS tf that transforms his YPR data to quaternions and then publish the ROS message. Since our link is not direct, all the functions that he performs within this node ar subsidized to our Mega 2560 controller, and in order to optimize the number of reads made to the IMU board by the UNO board, a new output function containing YPR, gyro, and accelerometer data was written. It is possible to choose as well the output format, either binary floats or chars. Since in the last phase of the project the Razor Imu
9DOF had a trial firmware which published in char format, all of the functions and arrays of the UNO and MEGA boards work with chars, therefore this output format is set by default. However, in the case that further sensors are implemented and a new computation or bandwidth constraint is added, this function would need any more development to give the same output in float data directly, only the configuration headers would have to be changed.

9.3.6.4 Specific code updates:

To be coherent with the way in which the Arduino UNO obtains the data from the Razor 9DoF IMU, this function should be included/extended from the previous update in the 'output.ino' sketch.

```cpp
void output_both_angles_and_sensors_text()
{
    if (output_format == OUTPUT_FORMAT_BINARY){
        float ypr[3];
        ypr[0] = TO_DEG(yaw);
        ypr[1] = TO_DEG(pitch);
        ypr[2] = TO_DEG(roll);
        LOG_PORT.write((byte*) ypr, 12);
        LOG_PORT.write((byte*) accel, 12);
        LOG_PORT.write((byte*) gyro, 12);
    }
    else if (output_format == OUTPUT_FORMAT_TEXT){
        LOG_PORT.print($);
        LOG_PORT.print(TO_DEG(yaw)); LOG_PORT.print(\n);
        LOG_PORT.print(TO_DEG(pitch)); LOG_PORT.print(\n);
        LOG_PORT.print(TO_DEG(roll)); LOG_PORT.print(\n);
        LOG_PORT.print(Accel_Vector[0]); LOG_PORT.print(\n);
        LOG_PORT.print(Accel_Vector[1]); LOG_PORT.print(\n);
        LOG_PORT.print(Accel_Vector[2]); LOG_PORT.print(\n);
        LOG_PORT.print(Gyro_Vector[0]); LOG_PORT.print(\n);
        LOG_PORT.print(Gyro_Vector[1]); LOG_PORT.print(\n);
        LOG_PORT.print(Gyro_Vector[2]); LOG_PORT.print(#);LOG_PORT.println();
    }
}
```

Furthermore, it is required to update the configuration options in the Razor_AHRS.ino sketch. Defining the output options: As in the project right now, the following should be defined:

```c
#define HW__VERSION_ 10736
#define OUTPUT__BAUD_RATE 57600
#define OUTPUT__DATA_INTERVAL 20 // in milliseconds

// Output mode definitions (do not change)
#define OUTPUT__MODE_CALIBRATE_SENSORS 0 // Outputs sensor min/max values as text for manual calibration
#define OUTPUT__MODE_ANGLES 1 // Outputs yaw/pitch/roll in degrees
#define OUTPUT__MODE_SENSORS_CALIB 2 // Outputs calibrated sensor values for all 9 axes
#define OUTPUT__MODE_SENSORS_RAW 3 // Outputs raw (uncalibrated) sensor values for all 9 axes
```
#define OUTPUT__MODE_SENSORS_BOTH 4 // Outputs calibrated AND raw sensor values for all 9 axes
#define OUTPUT__MODE_ANGLES_AG_SENSORS 5 // Outputs yaw/pitch/roll in degrees + linear accel + rot. vel
#define OUTPUT__FORMAT_TEXT 0 // Outputs data as text
#define OUTPUT__FORMAT_BINARY 1 // Outputs data as binary float

int output_mode = OUTPUT__MODE_ANGLES_AG_SENSORS;
int output_format = OUTPUT__FORMAT_TEXT;

#define OUTPUT__STARTUP_STREAM_ON false

The modification of this flags causes the system to initialize coherently with the SRO system instead of being prepared to work directly plugged to the computer running ROS.

Further details of the available commands, operation modes and outputs are included in the commented code of the firmware.
9.4 Appendix B: Relevant links and installation procedures

9.4.1 PC requirements

This project and its components were developed under Ubuntu 16.04 and ROS Kinetic. There is no hard cap on the minimum power of the computer system even though it should have 2 USB ports and run over an amd64 or i386 architecture. For reprogramming the controller modules the Arduino IDE 1.8.5 should also be installed and modified to be coherent with the buffer sizes specified in this document.

9.4.2 OS: Ubuntu 16.04

Downloadable from:
https://www.ubuntu.com/download/desktop
The recommended specifications are also indicated in the website.

9.4.3 ROS: Kinetic

The official installation procedure that was followed for the installation of the framework in this project is explained in the following URL:

https://wiki.ros.org/kinetic/Installation/Ubuntu

Where the installation data is also available for download (Although it is not necessary to download if the procedure is followed) Only the standard ROS packages that autoinstall are required for the SRO operation. Additional packages for the visualization and calibration of the IMU data are included in the AHRS ROS firmware directory if they were needed. Additionally, a catkin workspace including the code given in these appendices should be created.
Bibliography

[1] Pavel Davidson, Jani Hautamaki, Jussi Collin, *Using low-cost MEMS 3D accelerometer and one gyro to assist GPS based car navigation system*


[7] ROS answers: https://answers.ros.org/questions/


https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation


[12] Pirorobot, custom robots with SLAM capabilities
http://www.pirobot.org/blog/contests/TR2010/

https://en.wikipedia.org/wiki/Rotar0_encoder

[14] Dolphin Pool Cleaner
http://www.commercialpoolequipment.co.uk/cleaners/dolphin-supreme-m5-pool-cleaner.htm

[15] Bumper systems

[16] Pool Cleaner Robots
https://www.poolcleanerblog.com/robotic-pool-cleaner/

[17] Wikipedia, on SLAM

https://en.wikipedia.org/wiki/Autonomous_underwater_vehicle