UNIVERSITY OF TURKU

Robot Operating System: overview and case study

Mechanical Engineering/Faculty of Technology

Bachelor's thesis

Author:

Atte Rantanen

29.4.2024

Turku

Bachelor's thesis

Subject: Mechanical Engineering
Author: Atte Rantanen
Title: Robot Operating System: overview and case study
Supervisor: Associate Professor Wallace Moreira Bessa
Number of pages: 25 pages
Date: 29.4.2024

Creating software for robots is difficult because of the sheer amount of code required. The code needs to encompass everything from drivers to the actual functional program. This is why a group of students decided to create a new robotics middleware called Robot Operating System (ROS). The goal of ROS was to separate the program from the robot specific operating system to make the development process easier and more standardized. In this thesis we will explore the different ROS versions, as well as the main working principles of ROS including nodes and their communication. We'll also explore the iRobot Create® 3 educational robot through a case study and develop a simple navigation program for it. Additionally, we'll conduct an experiment with the Create® 3 to measure the responsivity of its infra-red sensors and find a function to convert the output to linear using regression.

Key words: robot operating system, infrared sensor, case study, navigation algorithm

Table of contents

1 Introduction

Every robot must have a software. Whether it's a robot vacuum or a self-driving car, they all require code to function. Developing software for these robots presents significant challenges due to their complexity and interaction with the real world. Therefore, the software must be resilient and capable of managing numerous variables encountered in the robot's environment.

Since robots can vary greatly due to different use cases, it is impractical to try to use a universal software with all of them. This might lead to the conclusion that we should create unique software for every robot, but this approach lacks efficiency due to the extensive amount of code required. The software must encompass everything from drivers to communication protocols and intelligent decision-making processes. Also, since the required breadth of expertise to make efficient software is so vast, the architecture should enable easy co-operation between researchers [1].

Given these challenges, it is not feasible to develop an entire operating system from scratch. Hence, a group of students from Silicon Valley opted to create a new open-source robotics development middleware known as the "Robot Operating System" (ROS) [2]. This system allows developers to concentrate on the development rather than the fundamental aspects common to different types of robots.

In this thesis, we get to know the working principles of ROS as well as the different versions that are out there. The second part is a case study, in which I will describe the installation process of Linux and ROS 2 as well as give a quick overview of iRobot's Create® 3 educational robot. Lastly, there is an example application that I wrote for the Create® 3 with C++ and ROS 2 Galactic Geochelone.

2 The Robot Operating System

2.1 Overview of ROS

2.1.1 What is a robotics middleware?

A robotics middleware is an abstraction layer that sits between the user applications and the operating system of the robot [3]. Illustration of this can be seen in figure 1. Middleware's primary function is to equip developers with an array of tools, including libraries, drivers, and monitoring tools, to assist the development process. For example, within these libraries, developers can access pre-existing and thoroughly tested functionalities such as path planning algorithms or computer vision algorithms [4]. Similarly, monitoring tools encompass various data visualization and simulation software, aimed at enhancing development efficiency and speed. Standardized algorithms play a pivotal role, enabling collaborative development among multiple engineers simultaneously.

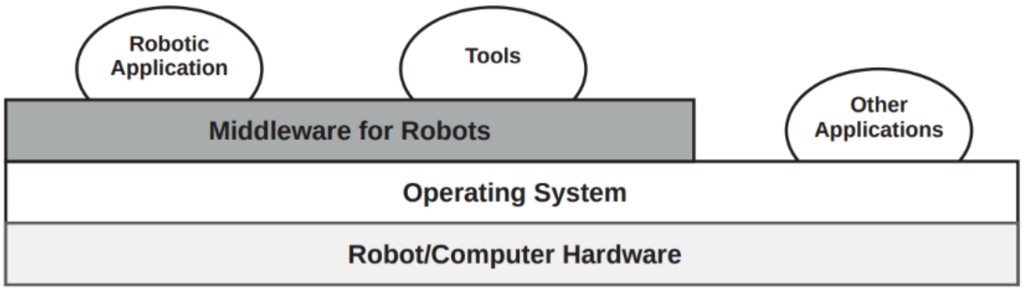


Figure 1: Representation of software layers in a Robot. [3]

However, ROS isn't the first robotics middleware to emerge. Previous attempts existed but failed to transcend the confines of the lab due to several reasons. One significant obstacle was the intertwining of operational code with the middleware, rendering the extraction of the middleware impractical upon robot completion [2]. Additionally, some of these predecessors concealed their source code, thereby hindering other developers' ability to modify the software effectively. In contrast, ROS operates on an open-source model, making its source code freely accessible for anyone to view and improve upon. This open approach has cultivated a vibrant community around ROS, launching it to the forefront as the standard in robotics development that we recognize today.

2.1.2 ROS versions

It is important to note that even though the acronym “ROS” is used a lot in this paper, there are actually two different versions of ROS: ROS 1 and ROS 2. These two versions have some major architectural and technical differences, but regarding this paper, their working principle is pretty much the same, so I’ll just use ROS to describe both of them. Regardless, I’ll now list some of the key differences between the two.

ROS 1 is the original Robot Operating System which development started at Willow Garage 2007 [5]. During the development, the main point was to create something standardized so that different developers around the world could experiment and improve upon each other’s work. The main point was to create something universal that would make robot development easier and more accessible for researchers. Because of this the performance of the system wasn’t the main priority, which led to some problems when trying to adopt it to commercial use cases [6].

For commercial use, the software must satisfy real-time run requirements. These requirements include, for example, certain security, fault tolerance, and process synchronization specifications that ROS 1 doesn’t fulfil [7]. Also since the real-time embedded systems weren’t the focus when developing ROS 1, the system ended up being quite resource demanding (e.g., CPU, memory, network bandwidth, threads, and cores) [7]. If this kind of inefficient software were to be used in commercial robots, the robots would need way too powerful hardware, which would lead to unnecessary costs and energy usage.

Hence the Robot Operating System 2 (ROS 2) was born. ROS 2 was announced at a ROS developer conference “ROSCon” in 2014 [8]. ROS 2 was built from ground up to address the previously mentioned issues of its predecessor. One of the biggest changes was to move from ROS 1’s proprietary data transport system to Data Distribution Service (DDS [9]), which is an open communication standard [10]. DDS enables ROS 2 to have the best-in-class security features as well as robustness in multirobot applications and non-ideal networking environments [9].

Second big change was to get rid of ROS 1’s Master node system. All ROS applications are constructed using nodes, a concept that will be further explained later in this paper. In ROS 1 all communications between these nodes are handled by the master node, which leads to a major security vulnerability, since if this process is compromised, the whole system will be.

Also, the system isn't very resilient when it has this kind of single point of failure. Because of this, ROS 2 moved from the Master node system to a truly peer-to-peer topology, where all the nodes are equal, and the DDS handles the communication. [7]

Third main improvement of ROS 2 was the increased number of supported operating systems. While ROS 1 only supports Linux, to make it more widely available, the developers expanded ROS 2 to support Windows, Mac OS and RTOS in addition to Linux. [7] Visualization of the main differences between ROS 1 and ROS 2 can be seen in figure 2.

ROS 2 also received some major performance improvements, which makes it possible to deploy on a standalone robot platform. Due to these changes, ROS 2 is now becoming the industry standard used and further developed for commercial applications. [7]

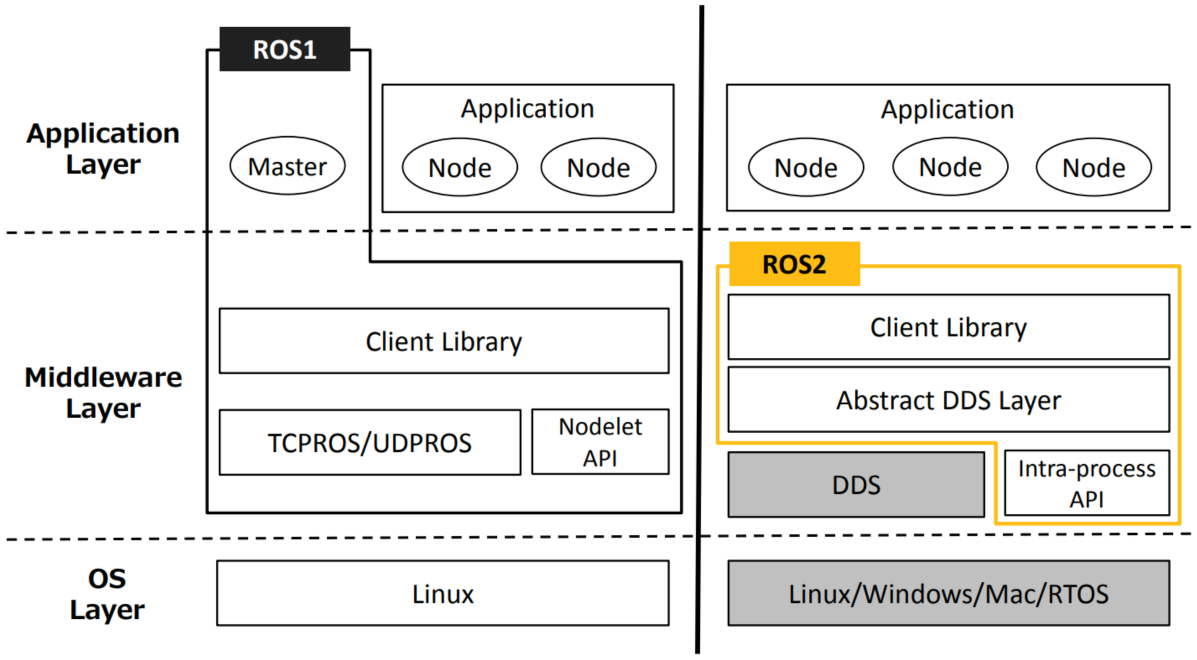


Figure 2: ROS 1/ROS 2 architecture. [7]

2.1.3 ROS Distributions

In addition to these two main versions of ROS, the versions are further split into distributions or "distros". Distros are a collection of libraries that have been verified to work together. This also includes the versions of the libraries. If some libraries were to be updated, they wouldn't necessarily work anymore with the other libraries of the distribution. For this reason, the libraries can't be modified (besides from some bug fixing) after the distro has been published [11]. If you want to get a newer version of some library, you need to wait for it to be available in a new ROS distribution.

As of today, there have been thirteen distributions of ROS 1, with only one, "Noetic Ninjemys," still receiving support [11]. Noetic Ninjemys's support will end in May of 2025 which will mark the end of the support of ROS 1.

When new distros are published, the old one's support doesn't end immediately. This gives developers more time and flexibility when designing their software. The supported distros can be found from the ROS 2's official website [12]. For ROS 2, there are usually two supported versions at the same time. There is one "long-term support" (LTS) version, which will be maintained for multiple years, and another version which will be maintained for around a year.

2.2 Working principle of ROS

2.2.1 ROS nodes

When ROS was developed, one of the main design philosophies was to build a peer-to-peer topology using nodes [1]. These nodes are the fundamental building blocks of any ROS program. The word "node" is used to describe one computational unit with usually one specific function. The basic idea is to split the required processes into small individual components, which then can communicate with each other. The concept of nodes is an essential way of organizing the software, which helps developers break down complex systems into more manageable sections [6].

How would you structure a program using nodes? Imagine you have a basic robot equipped with wheels and a camera, and you want it to identify and track an object. Instead of writing a single lengthy piece of software, you would break it down into multiple nodes, each serving a specific purpose.

First, you'd create a node to manage the camera, which communicates with the camera's hardware and retrieves the data from it. This data stream is then forwarded to an image recognition node, responsible for processing the image and attempting to identify the desired object within it. Once the object is recognized, the image recognition node can transmit its coordinates to a navigation node, which translates them into usable data for real-world location calculation.

Subsequently, the navigation node calculates the necessary movement commands and forwards them to a wheel control node, which controls the robot's wheels. Finally, the robot executes the movement based on these commands, effectively tracking the identified object.

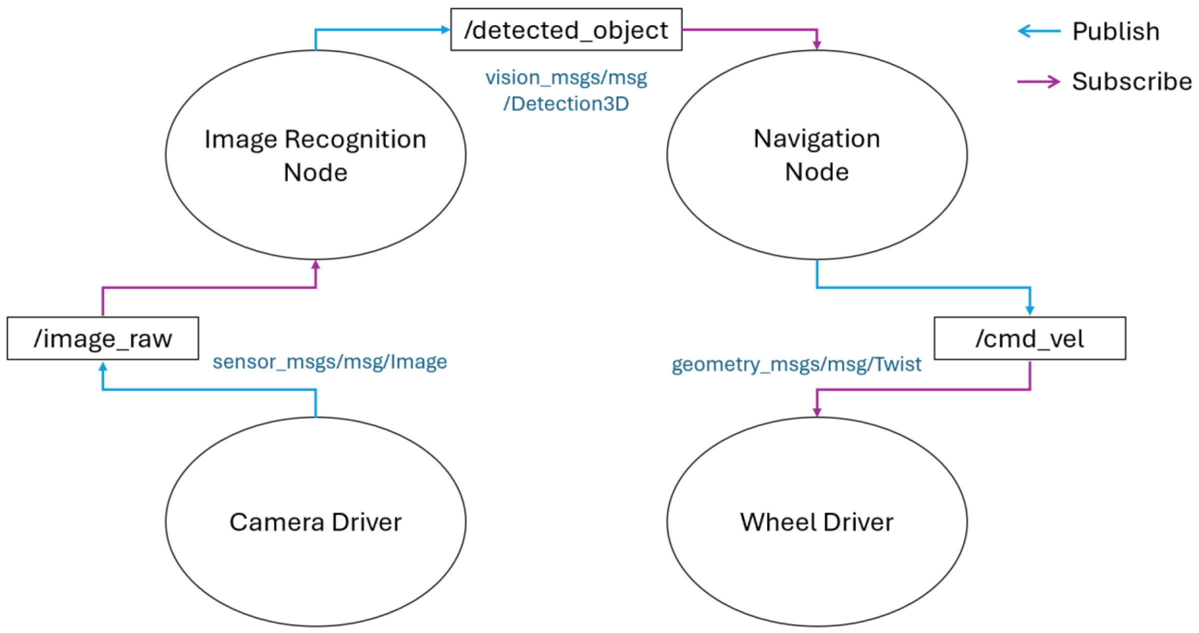


Figure 3: Computing graph of the example's object tracking robot.

With these nodes the development is easier and debugging more efficient compared to one single program, since you can test the nodes individually. This becomes especially apparent when advancing to bigger systems where the scale of complexity is much greater. If you didn't have this modular framework, you would need all your components working before you could test them. With this approach, you can test every component individually reducing the number of variables affecting the program.

Dividing the program into nodes makes it also natural to share the workload across multiple developers. Since the nodes are all their own individual complete systems, making changes to one doesn't affect the other. Other benefit of the nodes being complete systems, is the possibility of reusing old nodes in new projects with minimal modification to the code.

2.2.2 How nodes communicate

So, we have nodes that communicate with each other, but how do they do that? There are notably three ways that nodes can communicate with each other. These are called "topics", "actions", and "services". All of these have their distinctive characteristics and use cases. [10]

Topics are the most common way of transferring data between nodes. The basic idea is that you create a topic that handles a unique type of data and after that, any node can publish data to it or subscribe data from it. This way topics form an anonymous publish-subscribe architecture which allows many-to-many communication between nodes. Because we don't have a specific sender or receiver, this makes topics also an asynchronous network structure, which means that the sender and receiver of data won't need to be acting at the same time [3]. There is also some default topics which are built in the ROS software like "/rosout" which can be used for basic logging and debugging functionality [1].

What if we want to have an immediate response from another node? In such cases, services come into play. **Services** are established by implementing a service server within the designated node where service requests are directed. Once the hosting node is operational, other nodes can dispatch service requests to it, evoking immediate responses from the node [10]. "An example could be the request to the mapping service to reset a map, with a response indicating if the call succeeded" [3].

What if we want to ask a node to do something, but the task takes so long that the node can't answer immediately? For this, we have the last form of communication, **Actions**. As well as services, actions also need an action server which is hosted at the target node. The difference to service is that the client sends a goal to the action server, which will try to complete that goal. During the completion of the action, the server sends information about the progress back to the client. When the action is completed, the server sends on last message where the node tells if the action was completed successfully or not [10]. A good example of this could be a docking request for a robot vacuum.

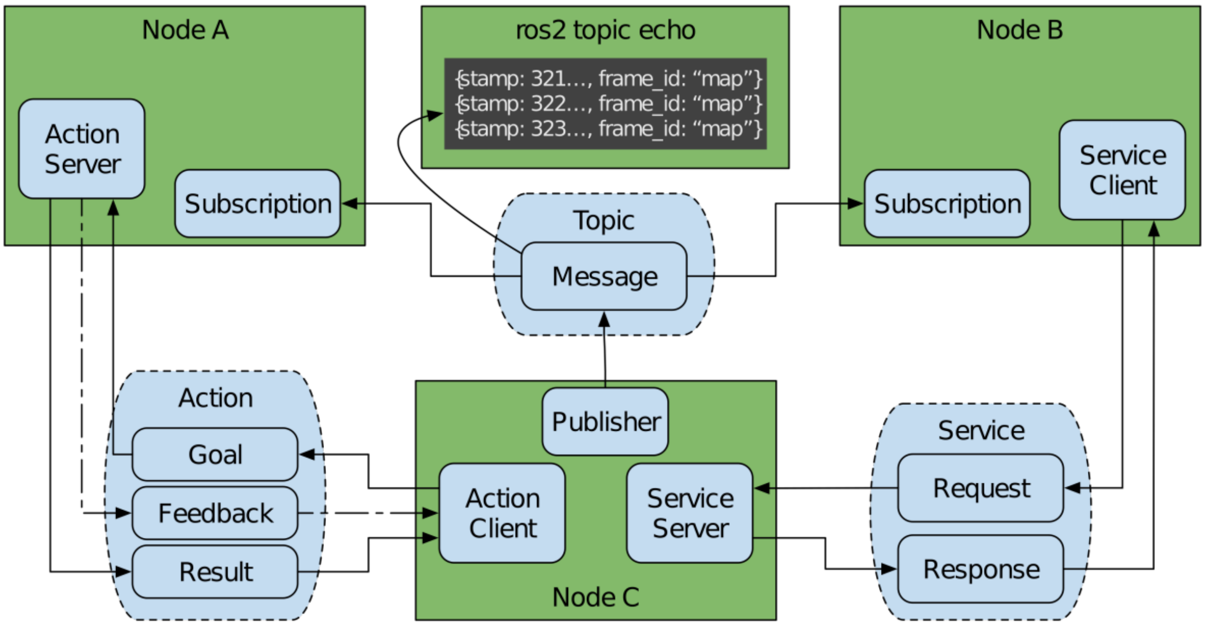


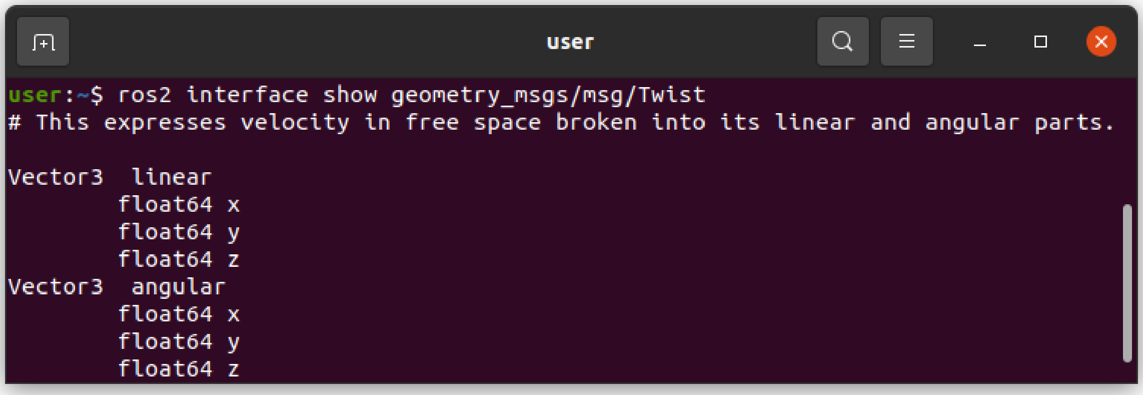
Figure 4: ROS 2 node interfaces: topics, services, and actions. [10]

2.2.3 Coding languages

There is one important aspect about ROS that we haven't touched on yet, the coding language. ROS 1 and ROS 2 support multiple coding language libraries from which two are maintained by their respective core ROS teams, meanwhile the rest are maintained by different community members. The two official languages are C++ and Python [1], [13]. C++ is generally more widely used because of its better performance compared to Python. In contrast, Python might be easier for beginners because of its easy-to-understand syntax and wide range of built-in functions.

The language neutrality makes ROS easy to approach for new developers. However, this language neutrality presented some unusual problems for the platform, since the same functionality should work across multiple completely different coding languages. To enable this functionality, ROS uses "language-neutral interface definition language (IDL)" [1].

The IDL utilizes concise string representations to store various types of data. This means that upon data generation, all distinct types. such as floats, are converted into strings, accompanied by their respective data types. For instance, transforming a float into a string involves appending "float64" followed by the floating-point value. Subsequently, when the data is read, it is reverted to its original format. This approach allows different programming languages to employ their own methods for converting data into strings and vice versa. This makes the coding easy for the developer since they only need to input the value to the topic and the underlying functionality of the coding language handles the rest.



```
user:~$ ros2 interface show geometry_msgs/msg/Twist
# This expresses velocity in free space broken into its linear and angular parts.

Vector3  linear
        float64 x
        float64 y
        float64 z
Vector3  angular
        float64 x
        float64 y
        float64 z
```

Figure 5: The structure of a topic: /cmd_vel.

3 Experiments with ROS 2 and iRobot Create® 3

3.1 ROS 2 distributions and goals for experiments

In this part I will talk about my own experience with ROS 2. During my experiments, the two distributions I used were, “Foxy Fitzroy” and “Galactic Geochelone”. The reason for this is that for learning the ROS, I used the book: *A Concise Introduction to Robot Programming with ROS2* [3] and for the experiments, I used the Create® 3. There are two versions of the book, one for Foxy and the other for ”Humble Hawksbill”. The content is otherwise the same, except the tutorials and examples are for different distributions. My version of the book used Foxy.

However, the goal was to learn to use ROS 2 with the iRobot Create® 3 and the distributions that it supports are Galactic and Humble [14]. The optimal choice would have been to select Humble because it was supported by both, even though the examples in the book were written for Foxy. The problem was that I was required to use Ubuntu 20.04 (focal) and at that point I thought that Humble can’t be installed on that version (later I realized that this isn’t completely true). It also didn’t come to my attention that the book had its examples for other distros besides Foxy. This is why I proceeded with this suboptimal multi distributional approach, which goes to show, that even as simple things as choosing which version to use, can be a challenge for a new developer.

My plan was to read the book and understand how the ROS system works. The book has, in my opinion, good examples and the code in them is explained well. Also, in the end of the chapters, there are exercises that require you to improve the code provided in the examples, which enhances the learning. After reading the book, my goal was to apply the concepts learned to the Create® 3 robot, and program my own simple navigation program for it.

3.2 iRobot Create® 3

3.2.1 Overview

The Create® 3 is an educational robot made by iRobot to be used for learning robot programming. From the outside, the Create® 3 looks like a normal robot vacuum, but in the inside, you find free space and open circuit boards. Additionally, you get access to the robot’s software. After connecting the robot to Wi-Fi, you can access it straight from your computer. Connection can be established also via Bluetooth, Ethernet, or USB if Wi-Fi isn’t available.

The Create® 3 can be controlled with three different ways: iRobot Coding App, iRobot Education Python 3 SDK or ROS 2. [15]

The Coding App is a browser-based graphical environment where the user can build simple programs from different kinds of functional blocks [16]. This approach is directed for younger students who are completely new to the concepts of robot programming. The whole focus is to get the robot moving without having to worry about the technical details. However, this has the draw back that the software is quite inflexible, and the possibilities are quite limited.

The next step from here is the Python SDK (Software Development Kit) which allows users to write their programs with Python instead of using graphical blocks. This has the obvious added challenge of knowing how to program with Python, but also the possibilities are significantly increased. There is also a function in the Coding App which allows users to convert their block-based programs straight to Python. This will be remarkably useful when moving to Python, since you can effectively create your own examples, and study how the same functionality can be implemented with Python.

Lastly, we have the most powerful and, at the same time, most difficult way of interacting with the Create® 3; ROS 2. With ROS, you can subscribe straight to the topics that the robot is publishing and do whatever you want with the data. Likewise, for example, moving the robot happens by sending the appropriate messages to the “/cmd_vel” topic. With ROS, you can also move from Python to C++ if you want, enabling possibly more powerful and optimized software. Unfortunately, this amount of freedom leads unavoidably to some errors and some debugging will be needed from time to time.

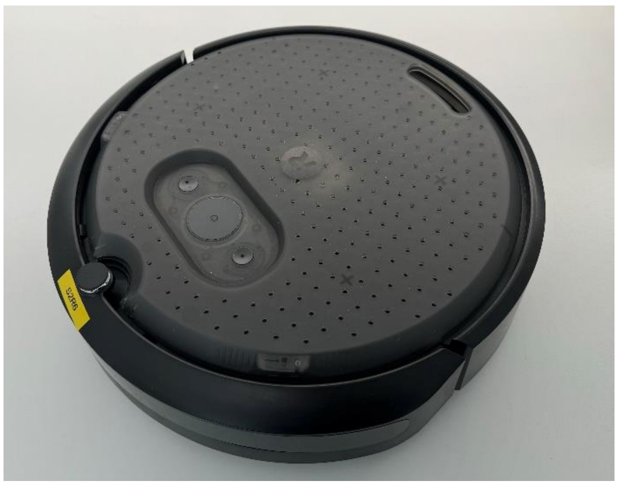


Figure 6: iRobot Create® 3.

3.2.2 Hardware

As you can see in the figure 6, the Create® 3 looks a lot like a robot vacuum. The biggest difference showing outside is the cover plate, which has multiple mounting holes for different kinds of accessories. The cover plate can also be removed completely by rotating it counterclockwise [14].

The robot has multiple sensors including: two Cliff Sensors and Optical Odometry Sensor at the bottom, Docking Sensor in front, Multi-Zone Bumper, and 7 IR Obstacle Sensors on the bumper [14]. With these, the Create® 3 can sense its surroundings very effectively. All these sensors can be accessed straight from the ROS 2 interface. All sensors have their own topics that the user can subscribe to.

3.2.3 Software

The Create® 3 incorporates multiple safety “reflexes” and built-in actions that the developer can utilize. The reflexes are a set of predefined behaviours that activate after a certain condition is met. For example, if the robot hits something and the bumper sensor detects that, the robot will stop moving and reverse a little bit. Same reaction can be observed if the cliff sensors detect a big ledge. If these functionalities are not desired, they can be disabled from the robot’s configuration file or by sending a command to the robot via terminal. [17]

The built-in actions that the Create® 3 has, are all involved with moving the robot. The user can send, for example, a command that tells the robot to drive certain distance and then the robot will move. One of the most useful action available, is the docking command. If sent to the robot, the command will start the docking process and if the dock is in sight, the robot will try to dock itself. There is also an action for leaving the dock.

3.3 Setup and installation

3.3.1 Preparing the computer

When starting to work with ROS, the first step is to prepare the Ubuntu installation on a computer. Since my laptop already had Windows installed on it, the idea was to deploy a dual boot. Dual boot means, that you partition the hard drive so that you can install both Windows, and Linux on the same computer. After installation, the computer will prompt the user every time at startup, and ask, which operating system should be launched. I had some problems with the Ubuntu installation, which were caused by outdated network drivers, but I was able to update them by sharing internet from my phone to the computer via USB.

Next step was to install the ROS itself. There are great installation tutorials on the official ROS 2 website and also in the book [3]. The only problem in just copying commands that other people have written, is that if something doesn't go as it should, the troubleshooting can be difficult, since you don't know what the specific commands do. For me, one thing that caused a lot of problems, was that I sometimes mixed the two ROS 2 versions, which caused errors during the installation. After all, the errors weren't anything special, but because of the new platform and software, the troubleshooting was occasionally hard, since my questions ended up being so basic, that most of the suggestions, just assumed that I already knew how to fix them myself.

Next, I installed some example application from the book as well as from the internet. The tutorials were again good, but I had quite many problems when trying to install the dependencies that the examples required. The ROS developer tools package has a tool for installing dependencies called "rosdep". However, the problem with the tool was, that by default, it only searches dependencies for supported ROS distros, and since I was using older, already discontinued distros, the rosdep couldn't find my dependencies. This can be easily fixed by adding a flag to the end of the rosdep command, that tells it to include end-of-life distros. The only problem at the time was, that I didn't know this, so finding this information online took a while.

3.3.2 Preparing the robot

After successfully installing ROS, the next step is to setup the Create® 3. Luckily, there is a great tutorial for it in the documentation website of the robot [14]. The process is very simple, and all you need to do, is to follow the step-by-step instructions. The first step after turning on the robot, is to connect it to Wi-Fi. This is achieved, like many other Wi-Fi-enabled devices nowadays, by connecting to a Wi-Fi hotspot that the robot hosts and then, configuring your own network settings to the robot via the robot's browser-based configuration interface. After that, you can disconnect from the hotspot and the robot will try to connect to your wireless network. Now you can access the configuration interface from your browser, provided that you are connected to the same Wi-Fi network as the robot. It is also recommended that you update the firmware of the robot at this point.

The final step in getting the robot working with ROS is, to install the “irobot-create-msgs” library which includes all the Create® 3 specific ROS 2 messages. These can be installed with a simple install command in the Ubuntu terminal.

3.4 Driving the Create® 3

The first functional thing that I tried with the Create® 3, was teleoperating the robot with the computer. This means that, I can drive the robot in real time using the computer over the Wi-Fi connection. This requires you to download and install the ROS Teleop Twist package, which installs the program that translates the presses on the keyboard to ROS 2 messages and publishes them to the “cmd_vel” topic. [18]

After the installation is completed, the teleoperation is quite simple. All you need to do is to run the command: “ros2 run teleop_twist_keyboard teleop_twist_keyboard”. This launches the program, after which you can start driving the robot. However, at this point the robot doesn’t let you move backwards. This is because of the built-in safety features of the robot. You can disable only the backup limit that prevents the reversing, but for the most extensive experience, you can set the variables “safety_override” and “reflexes_enabled” to “full” and “false” respectively, which will disable all built-in reflexes and give you full control of the robot.

3.5 Navigation program

3.5.1 Main idea

My goal with the Create® 3 was to develop a navigation program for it that could read the sensor data from the IR-sensors on the bumper and utilize that information for avoiding walls and navigating around a space. While learning ROS 2 with the help of the book [3], I noticed that one of the examples had an interesting approach to navigating.

In the example, the robot has a laser scanning sensor that scans the surroundings with hundreds of beams and returns an array where there are all the readings from every laser beam. After that, the program detects which of the beams is the shortest (the closest object) and selects that. The orientation and length of the said beam are then turned into a “repulsive vector”, which is then rotated 180 degrees. After this, it is added to another vector which is pointing straight forward. This all sounds complicated, but the resulting vector that you get, will point in the direction you want the robot to move to avoid the nearest object (coloured vectors in figure 11). This approach of adding vectors together fascinated me, and I figured that maybe I could read the data from the IR-sensors of the Create® 3 and create the repulsive vector that way.

I used the book's "br2_vff_avoidance" project as my base and modified the necessary parts to fit my own use case [3]. My own ROS 2 project can be found here [19]. Since I didn't have much coding experience with C++, the progress was slow at first, but eventually I got everything that I needed to get data from the Create® 3 imported to my ROS 2 project.

3.5.2 The IR sensor

The initial task was to see, what kinds of data the IR-sensors are sending. Turns out that like in the example, the Create® 3 sends an array where the values of every sensor are listed. However, when I tested the sensors with moving my hand in front of them, I noticed that the numerical output of the sensor isn't linear related to the distance between my hand and the robot. For my application I wanted the sensor data to be at least roughly linear, so I conducted a small experiment to see what the data looked like. I placed a cardboard box in front of the robot and measured the distance and the value from the front sensor (figure 8). I repeated this for all values in the range from 2 cm to 60 cm with and increment of 2 cm. I didn't start measuring from 0 cm, because the sensor couldn't sense it properly. After all measurements were done, I collected the data and plotted it, which can be seen in figure 7 (the red line).

You can see that the value increases when the box gets closer to the sensor. However, it's clear that at first the value decreases rapidly and after the distance exceeds 6 cm the function starts to flatten out. My solution for this was to split the graph from two points splitting the graph to two about straight parts and one curved part. After which I implemented a normalizing function for every section, so that the resulting function is as linear as possible (the blue line). For finding the functions I used polynomial regression. I wrote a short python script that utilizes pandas [20] and scikit-learn [21] to use machine learning to find the best polynomial functions to fit the data. The script can be found here [22]. I used second order polynomials to approximate every section of the graph. After the readings are converted with the proper functions, the value we get is further divided by 50 to represent the objects distance from the robot in the range from 1 to 0. 1 representing no object detected and 0 that the object is touching the robot. The implementation of this can be seen in figure 9, where the "x" represents the actual reading of the sensor and "distance" represents the result after the normalization.

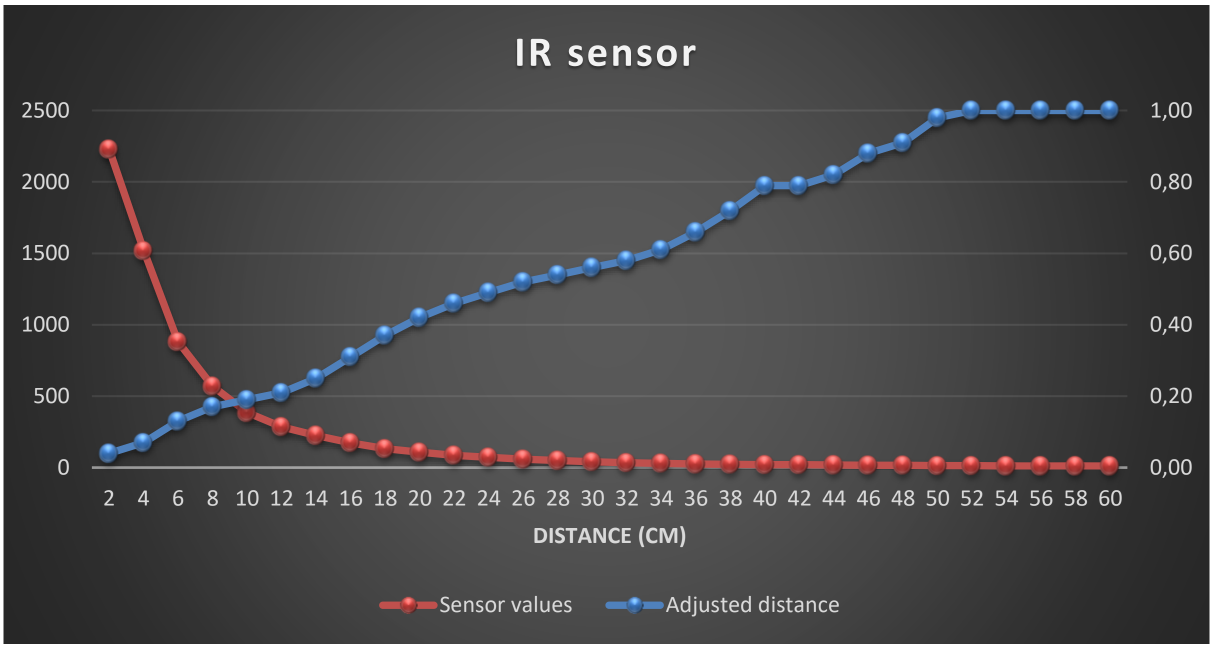


Figure 7: Values from the experiment

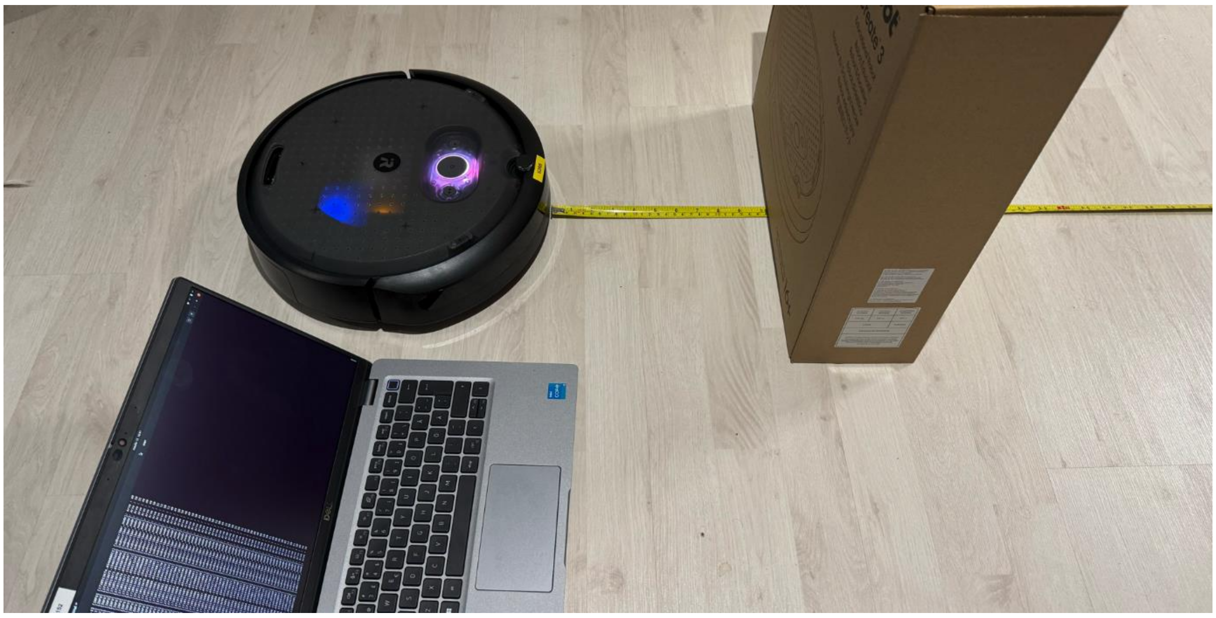
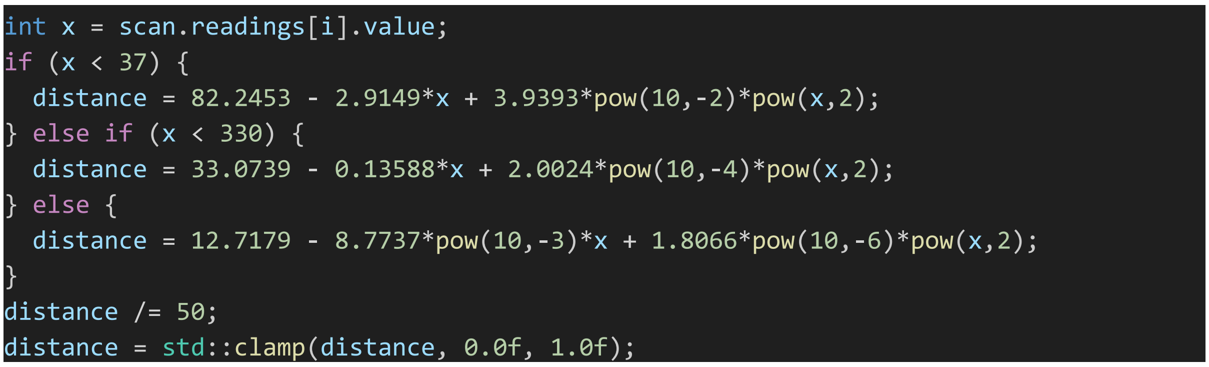


Figure 8: Experiment setup

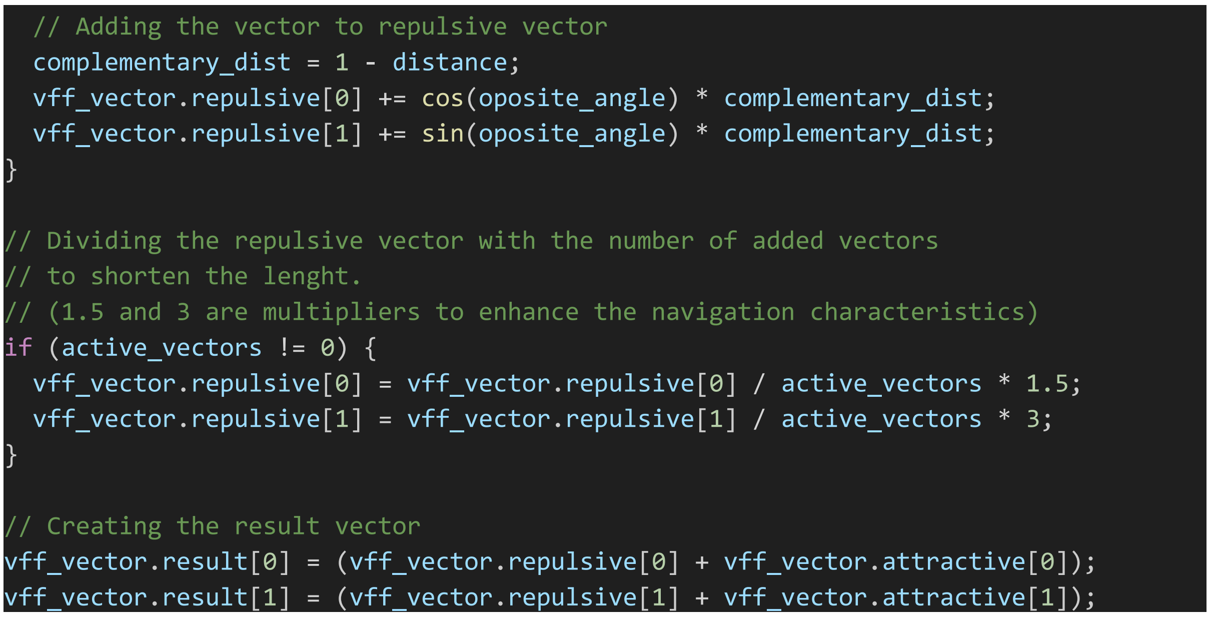


```
int x = scan.readings[i].value;
if (x < 37) {
  distance = 82.2453 - 2.9149*x + 3.9393*pow(10,-2)*pow(x,2);
} else if (x < 330) {
  distance = 33.0739 - 0.13588*x + 2.0024*pow(10,-4)*pow(x,2);
} else {
  distance = 12.7179 - 8.7737*pow(10,-3)*x + 1.8066*pow(10,-6)*pow(x,2);
}
distance /= 50;
distance = std::clamp(distance, 0.0f, 1.0f);
```

Figure 9: Implementation of the normalization function. NavigatorNode.cpp. [19]

3.5.3 Calculating the vectors

Unlike in the “br2_vff_avoidance” project where the program chooses the shortest laser beam, I decided to consider every sensor (fig. 11, grey arrows) on the Create® 3 that detects an object, and then take an average of those to calculate the repulsive and resulting vectors. To turn the distance readings into vectors, I got the orientations of every sensor from the documentation [14] and then calculated the necessary x and y coordinates with simple trigonometry to create the vectors (top of figure 10). Instead of using the real angle of the sensor, we use the opposite angle (angle + 180°), because we want the repulsive vector (fig. 11, red arrow) pointing backwards so that the result vector will turn the robot in the right direction.



```
  // Adding the vector to repulsive vector
  complementary_dist = 1 - distance;
  vff_vector.repulsive[0] += cos(oposite_angle) * complementary_dist;
  vff_vector.repulsive[1] += sin(oposite_angle) * complementary_dist;
}

// Dividing the repulsive vector with the number of added vectors
// to shorten the lenght.
// (1.5 and 3 are multipliers to enhance the navigation characteristics)
if (active_vectors != 0) {
  vff_vector.repulsive[0] = vff_vector.repulsive[0] / active_vectors * 1.5;
  vff_vector.repulsive[1] = vff_vector.repulsive[1] / active_vectors * 3;
}

// Creating the result vector
vff_vector.result[0] = (vff_vector.repulsive[0] + vff_vector.attractive[0]);
vff_vector.result[1] = (vff_vector.repulsive[1] + vff_vector.attractive[1]);
```

Figure 10: Calculating the vectors. NavigatorNode.cpp. [19]

As you can see in figure 10, we add the vectors to the already existing repulsive vector. Because of this, after all seven sensors have been analysed, we take the repulsive vector and divide it by the number of active vectors that have been used to create the repulsive vector. This way, the vector that we get, is the average of all repulsive vectors that were used to create it. However, after testing and tweaking the algorithm I ended up adding multipliers (1.5 and 3) to the calculations to enhance the behaviour of the robot when avoiding an obstacle. Lastly, we add the repulsive vector to the attractive vector to get the result vector that will guide the robot (fig. 11, green arrow).

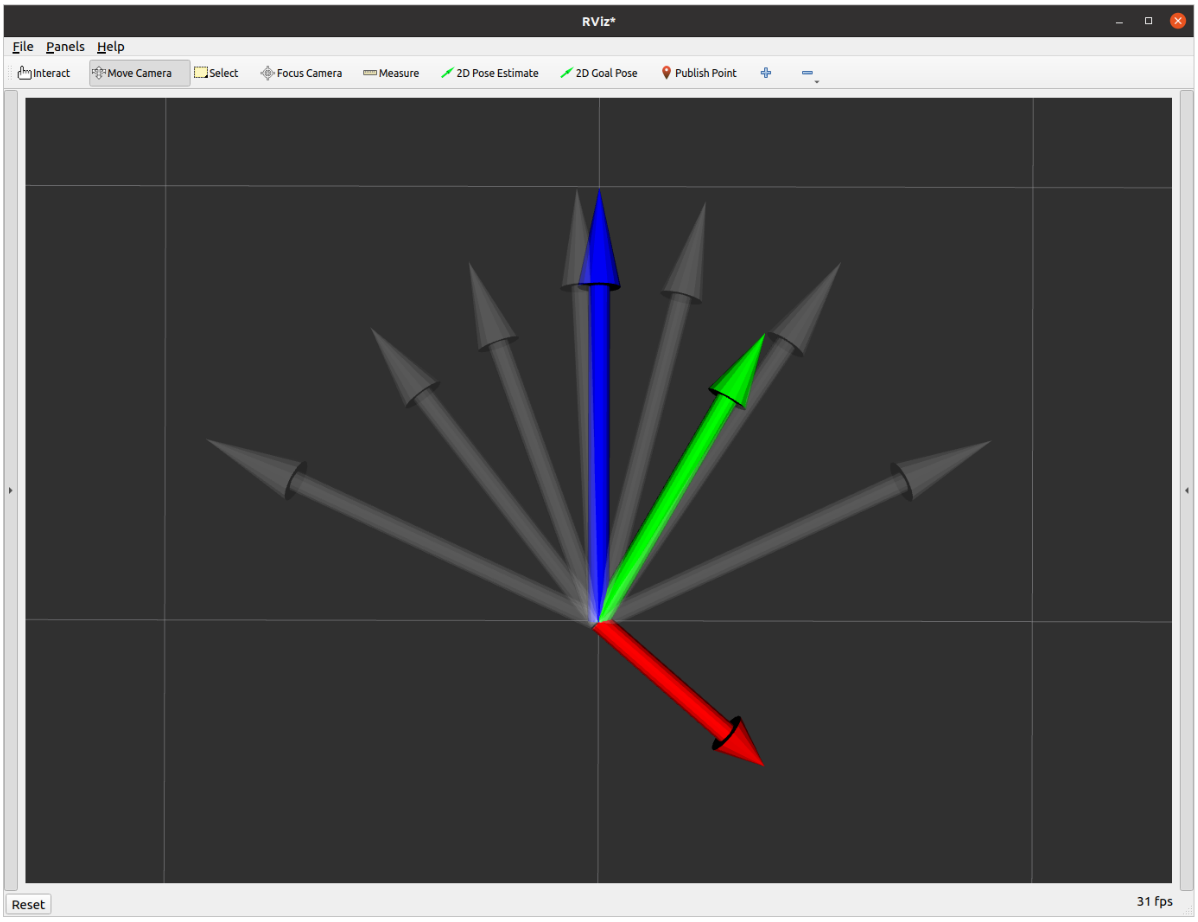


Figure 11: Vector illustration.

3.6 Final outcome

The navigation algorithm works, and the robot can avoid obstacles. However, the algorithm is no way perfect and there are many edge cases where the robot will get stuck or hit something. One obvious example of this is narrow obstacles, like table legs. Because there are only seven sensors on the robot, it is possible, that the obstacle is between two sensors causing a situation, where neither of them can detect it, and the robot can’t recognize it.

One way of improving the algorithm would be, for example, implementing tf2-library to the program. tf2 is a library that enables you to map out the robot’s movement in 2D space, which could be useful if the robot got stuck. For example, if the robot couldn’t move forward, we could use tf2 to know where the robot came from and then backtrack there.

Nonetheless, the object of this experiment wasn’t to create a perfect obstacle avoidance algorithm but to get to work with the robot and practise development with ROS 2. For this purpose, this worked great since for getting this to work required me to explore every part of ROS 2 and Create® 3 systems.

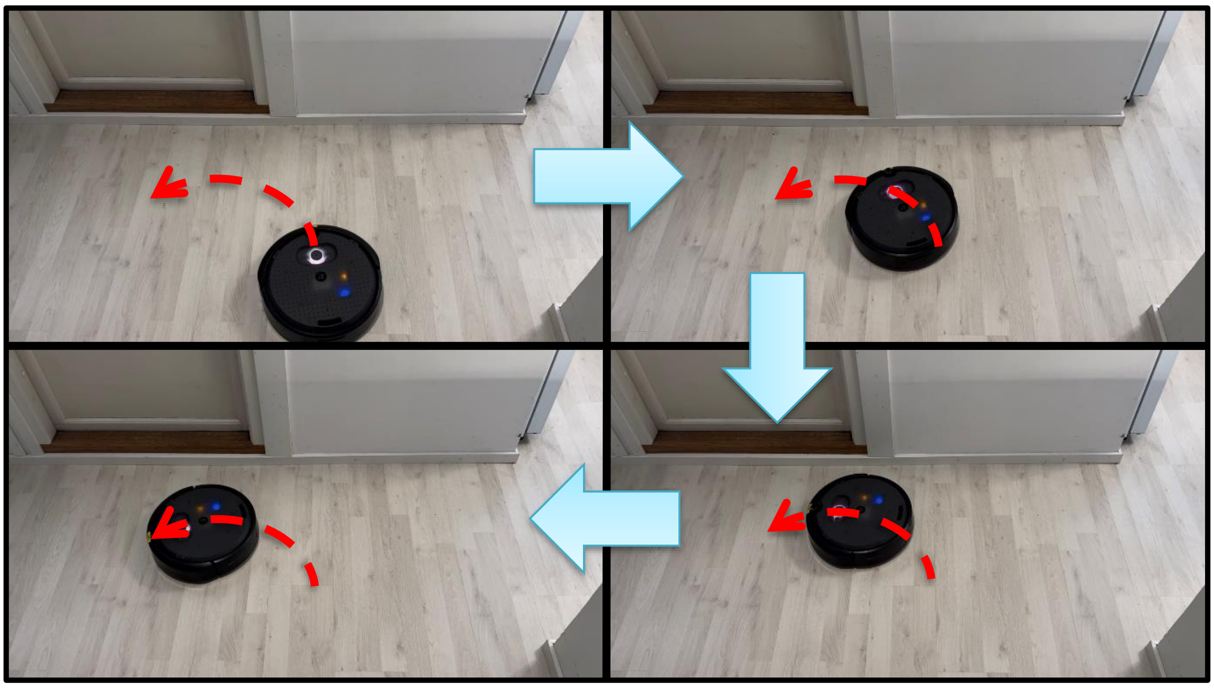


Figure 12: Behaviour of the algorithm when encountering a wall.

4 Conclusion

During my experiments, it became clear that ROS has a somewhat steep learning curve, that new developers need to overcome before being able to use the system. Additionally, before even beginning to work with ROS, the developers need to know how computer programming works in general and be familiar with using Linux and terminal. Additionally, the multiple versions and distributions of ROS, can sometimes make finding the right information difficult and troubleshooting laborious.

However, when you finally overcome this learning curve, you realize that ROS is a very powerful framework. The node-based system makes structuring your program natural and enables great collaboration possibilities. The integrated communication protocols and standards make it also straight forward to start the development without having to worry about practicalities. One of ROS's greatest strengths is the fact that it separates the robots own operating system from the user program. This makes adopting old software on new robots fast compared to the software being included in the robot's own operating system.

Even though I had my own problems when installing and learning ROS, I can tell, that it wouldn't have been possible for me to program any robot with my level of knowledge at the time, if I weren't using ROS. The learning material on the official ROS 2 website as well as the iRobot Create® 3's website, supported me well and enabled me to slowly get to know the ins and outs of the software as well as the Create® 3 educational robot.

After all, ROS has successfully become the universal standard for robot programming that it was envisioned to be. It all started with ROS 1, in which the researchers created the foundation to build on. With ROS 2, the system's biggest flaws were fixed, and the platform was finalized, so that it could be adopted for real life commercial use cases. During all of this, ROS has established a vibrant community around it which will continue the development of the platform making it even better in the future.

References

[1] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, A. Ng, ROS: an open-source Robot Operating System, ICRA Workshop Open Source Softw. 3 (2009) 5.

[2] K. Wyrobek, The Origin Story of ROS, the Linux of Robotics - IEEE Spectrum, (2017). https://spectrum.ieee.org/the-origin-story-of-ros-the-linux-of-robotics (accessed February 1, 2024).

[3] F.M. Rico, A Concise Introduction to Robot Programming with ROS2, 1st ed., Chapman and Hall/CRC, Boca Raton, 2022. https://doi.org/10.1201/9781003289623.

[4] S. Kim, T. Kim, RoboFuzz: fuzzing robotic systems over robot operating system (ROS) for finding correctness bugs, in: Proc. 30th ACM Jt. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., ACM, Singapore Singapore, 2022: pp. 447–458. https://doi.org/10.1145/3540250.3549164.

[5] E. Ackerman, E. Guizzo, Wizards of ROS: Willow Garage and the Making of the Robot Operating System - IEEE Spectrum, (2017). https://spectrum.ieee.org/wizards-of-ros-willow-garage-and-the-making-of-the-robot-operating-system (accessed February 2, 2024).

[6] S. Macenski, A. Soragna, M. Carroll, Z. Ge, Impact of ROS 2 Node Composition in Robotic Systems, IEEE Robot. Auton. Lett. RA-L (2023). https://doi.org/10.48550/arXiv.2305.09933.

[7] Y. Maruyama, S. Kato, T. Azumi, Exploring the performance of ROS2, in: Proc. 13th Int. Conf. Embed. Softw., ACM, Pittsburgh Pennsylvania, 2016: pp. 1–10. https://doi.org/10.1145/2968478.2968502.

[8] Program | ROSCon 2014, (n.d.). http://roscon.ros.org/2014/program/ (accessed February 21, 2024).

[9] G. Pardo-Castellote, OMG data-distribution service: architectural overview, in: 23rd Int. Conf. Distrib. Comput. Syst. Workshop 2003 Proc., IEEE, Providence, Rhode Island, USA, 2003: pp. 200–206. https://doi.org/10.1109/ICDCSW.2003.1203555.

[10] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, Robot Operating System 2: Design, Architecture, and Uses In The Wild, Sci. Robot. 7 (2022) eabm6074. https://doi.org/10.1126/scirobotics.abm6074.

[11] Distributions - ROS Wiki, (n.d.). https://wiki.ros.org/Distributions (accessed February 23, 2024).

[12] Distributions — ROS 2 Documentation: Rolling documentation, (n.d.). https://docs.ros.org/en/rolling/Releases.html (accessed February 26, 2024).

[13] Client libraries — ROS 2 Documentation: Iron documentation, (n.d.). https://docs.ros.org/en/iron/Concepts/Basic/About-Client-Libraries.html (accessed February 13, 2024).

[14] Create® 3 Docs, (n.d.). https://iroboteducation.github.io/create3_docs/ (accessed March 2, 2024).

[15] Create-3_DataSheet.pdf, (n.d.). https://experience.irobot.com/hubfs/Create%203/Create-3_DataSheet.pdf?utm_campaign=Product%20Launch&utm_source=Create%203&utm_medium=Landing%20page (accessed March 4, 2024).

[16] iRobot Coding App, IRobot Educ. (n.d.). https://edu.irobot.com/what-we-offer/irobot-coding (accessed March 4, 2024).

[17] Reflexes - Create® 3 Docs, (n.d.). https://iroboteducation.github.io/create3_docs/api/reflexes/ (accessed March 11, 2024).

[18] Teleop Twist with the Create 3, IRobot Educ. (n.d.). https://edu.irobot.com/learning-library/teleop-twist-with-the-create-3 (accessed March 18, 2024).

[19] A. Rantanen, A-ROS2-based-Navigation-program-for-Create3, (n.d.). https://github.com/smartystems/A-ROS2-based-Navigation-program-for-Create3.git (accessed March 21, 2024).

[20] pandas - Python Data Analysis Library, (n.d.). https://pandas.pydata.org/ (accessed April 5, 2024).

[21] Scikit-learn 1.4.1 documentation, (n.d.). https://scikit-learn.org/stable/ (accessed April 5, 2024).

[22] A. Rantanen, Polynomial regression, (n.d.). https://github.com/smartystems/A-ROS2-based-Navigation-program-for-Create3/tree/725defab9c37b4752f1a05c164a41b8a0216a0b6/Polynomial%20regression (accessed April 16, 2024).