Wrocław University of Science and Technology

Robotics Project

Astable Invention - Automatic vacuum cleaner robot

Project advisor:
Prof. Janusz Jakubiak

Project authors:
Jakub Grzana 241530
Kacper Potoczny 259761
Szymon Krukowski 259770
Weronika Szęszoł 253149

Wrocław 2023

Table of Contents

1. Introduction

The purpose of our project was to reverse engineer the *iRobot Roomba* autonomous vacuuming robot and replace its original (broken) motherboard with a microcontroller ESP32 and Raspberry Pi 3B. This report will cover everything we've found out about low-level, electronics part of iRobot Roomba 675, communication between microcontrollers and sensors of the robot, as well as high-level code executing rudimentary mapping and pathfinding algorithms.

2. Theoretical introduction and methods

Our robot has a number of sensors, including but not limited to: six infrared based proximity sensors, bump sensors, two wheel motors and two wheel encoders. However there's no documentation and information available on the Internet about this topic are very scarce, mostly for the purpose of repair. In short, we had to reverse engineer on our own, which is a process analogical to traversing thick foggy forest by a half-blind child.

Reverse engineering was done mostly by manually applying voltage or interacting with sensors and reading voltage on different pins with a multimeter (Fig 1). Based on the observations, we've formulated a reasonable theory of what's going on inside the robot, how does given feature work, and checked whether this theory holds for different circumstances.

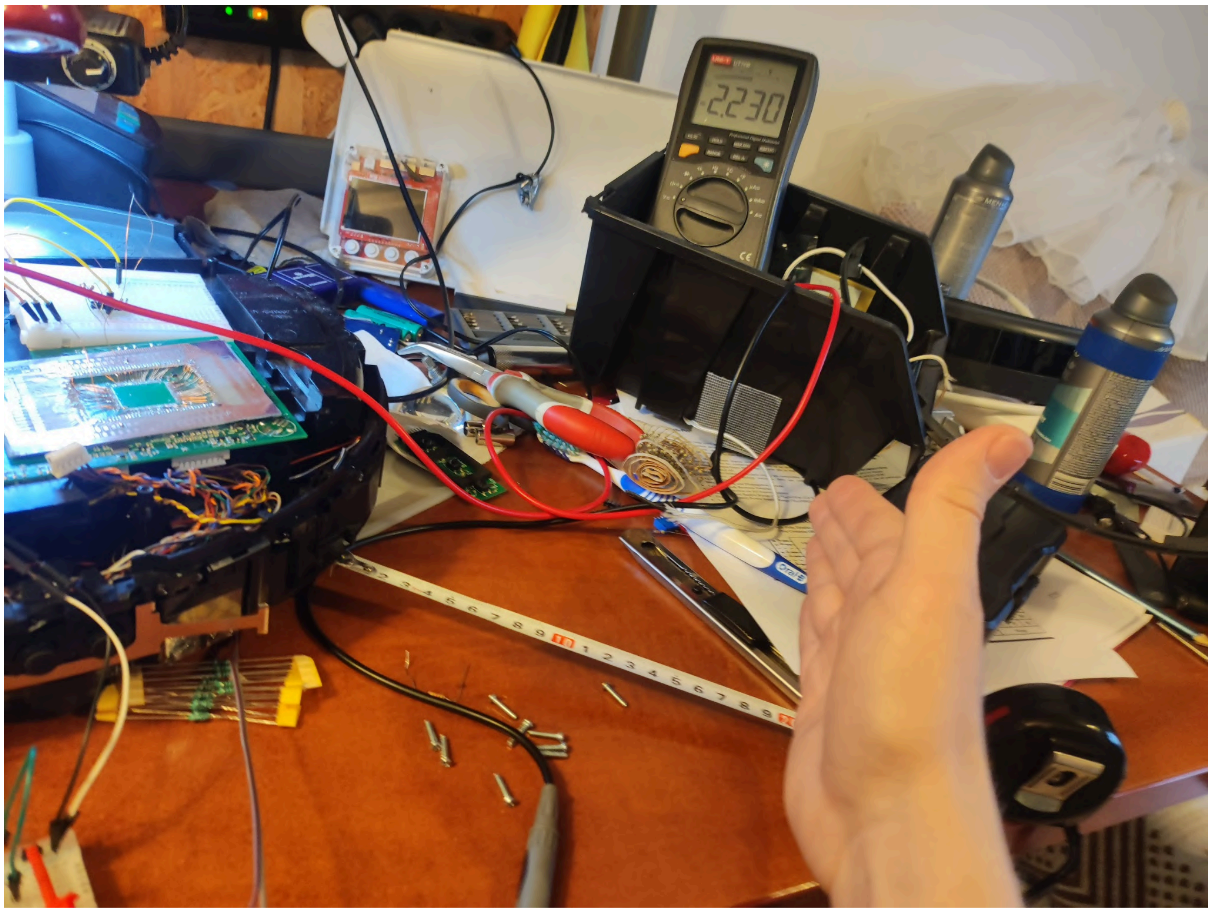

Fig 1. Tests of reflective sensors

Regular roomba's sensors use a combination of optical, acoustic, and inertial sensors to navigate a room. The optical sensors detect dirt, cliff drops, and other obstacles and adjust the robot's cleaning path. The acoustic sensors detect the location of obstacles, such as furniture, and the robot's distance from them. The inertial sensors track the robot's movements, orientation, and acceleration. These sensors work together to help Roomba map a room and avoid obstacles while cleaning.

While Roomba is cleaning, it avoids steps (or any other kind of drop-off) using four infrared sensors on the front underside of the unit. These cliff sensors constantly send out infrared signals, and Roomba expects them to immediately bounce back. If it's approaching a cliff, the signals all of a sudden get lost. This is how Roomba knows to head the other way. Another infrared sensor, which we'll call a wall sensor, is located on the right side of the bumper and lets Roomba follow very closely along walls and around objects (like furniture) without touching them. This means it can clean pretty close up to these obstacles without bumping into them. It also determines its own cleaning path.

Short-range ultrasonic ToF sensors can be used to determine different floor types. The application uses the average amplitude of a reflected ultrasonic signal to determine if the target surface is hard or soft. If the robot vacuum detects that it has moved from a carpet onto a hardwood floor, it can slow the motors down because they do not need to work as hard compared to carpet use. For robot vacuums that do not use VSLAM or LiDAR mapping technology, their position and navigation can be determined using dead reckoning by combining measurements from the wheel’s rotations with the inertial measurements from the IMU and object detection from the ToF sensors.

So we had a general idea what sensors and actuators to expect in the Roomba, but without any particular details. Due to the immense scope of the project, we had to split development into three separate branches: hardware, software and computer vision, so every major part could be developed and tested in parallel, with API and/or communication protocols for easier integration at the end. Robot software was done using Python and ROS 2 framework with simulations in Gazebo, hardware part was reverse engineered manually and then steered with microcontroller programmed in C and micro-ROS framework, and computer vision was developed as standalone Python program.

3. Assumptions

Tab 1. Functional and design assumptions of the project, essentially project goals

Functional Assumption	Design assumption
Taking full control of iRobot Roomba base	Achieved by reverse engineering: manually applying voltage and/or interacting with sensors and measuring response with voltmeter.
Programming the ESP32 microcontroller to read values from sensors and control robot motors.	Achieved by programming microcontroller with micro-ROS framework, low-level ANSI C and HAL.
Establishing communication between microcontroller and Raspberry Pi.	Achieved by implementing communication through ROS topic between Raspberry and microcontroller.
Programming Raspberry Pi to cover the entire surface of the room.	Achieved by mapping room and using Breadth First Search to find nearest unexplored spot.
Programming Raspberry Pi to detect house pets with digital camera and computer vision.	Achieved by using OpenCV and python coding to detect cats & dogs, determining their relative to robot position.

4. Description of the hardware

Our project is built-upon the foundation of iRobot Roomba 675. Fig 2 shows a block diagram of the project, all the way from physical robot, through microcontrollers to Raspberry Pi.

Physical robot has no official documentation nor schematic for its motherboard. We’ve removed the original microprocessor and managed to get partial control of the hardware by connecting to *(original microprocessor’s)* pins, as well as directly connecting to pins of components. Details of robot hardware are explained in subchapter 4.1

Microcontrollers ESP32 are connected to electrical instruments on board of the base, receiving voltage readings from sensors and sending signals to actuators. Both microcontrollers serve as bridges between Raspberry Pi and physical robot - details of communication protocol *(ROS 2 topics)* are explained in chapter 5.

Raspberry Pi is the main brain of the project, creating a map of the room based on digitized readings from sensors provided by ESP32, planning movement and sending signals to ESP32 to steer the robot's motors accordingly.

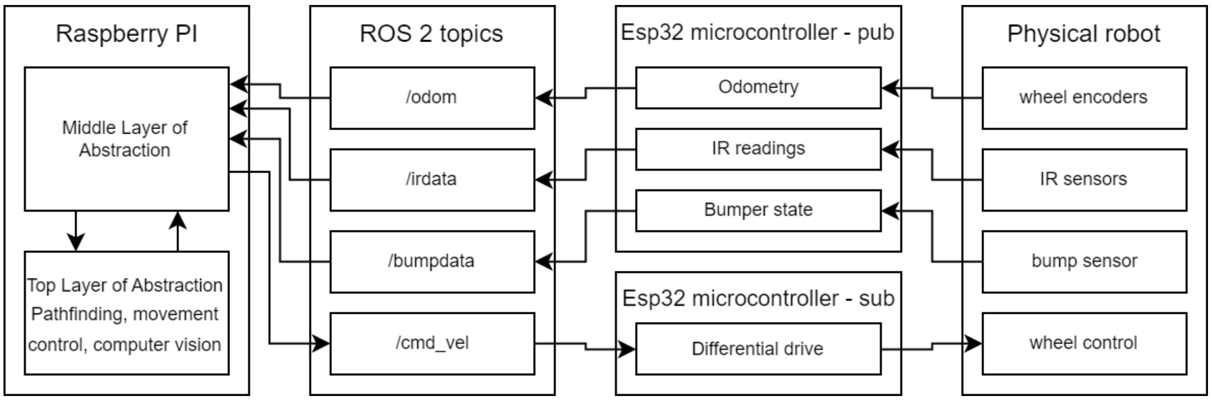

Fig 2. Block diagram of project: from physical robot to Python and Raspberry Pi

4.1 Reverse engineering of hardware base

The base of the hardware is iRobot Roomba 675, version white. Due to the mainboard malfunction, we had to remove from its original CPU, and reverse engineer most of the motherboard functionalities to discover how it was operating. Before desoldering, we've read the name of original microprocessor: STM32F103zET6

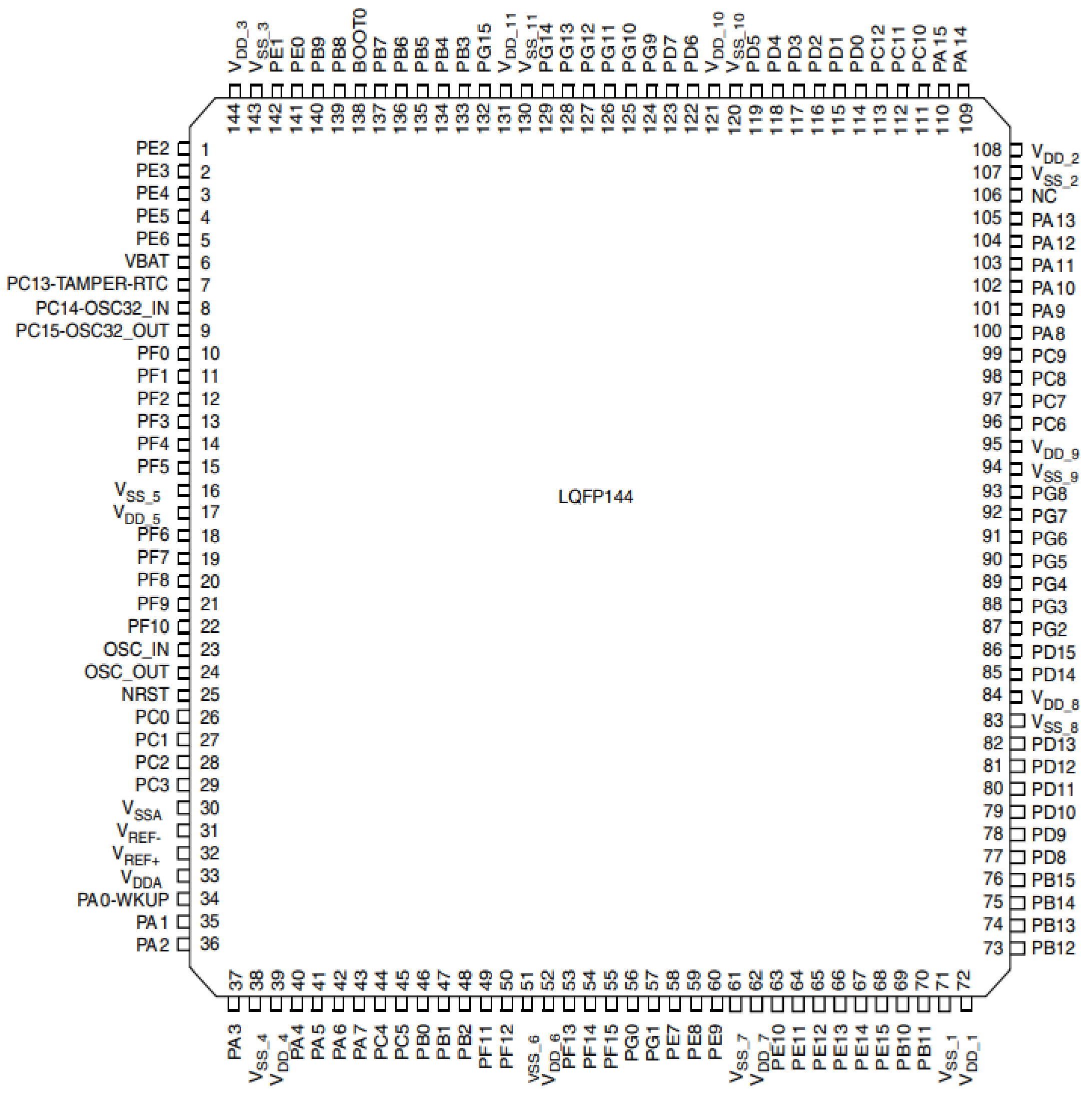

Fig 3. Pinout of STM32F103zET6 microprocessor [5]

Tab 2. Pins we've fully and successfully reversed-engineered

Pin name	Type	Comments
PE2	IN	Start button
PG1	OUT	LED data 1-3
PE7	OUT	LED alert
PE8	OUT	Side brush (pwm)
PE9	OUT	Side brush (dir)
PE10	OUT	Right wheel (pwm)
PE11	OUT	Left wheel (dir)
PE12	OUT	Bottom brush (pwm)
PE13	OUT	Bottom brush (dir)
PD12	OUT	Left wheel (pwm)
PD13	OUT	Right wheel (dir)
PG8	OUT	I2S
PG4	OUT	If high, turn off all components
PC10	OUT	LED Wi-Fi 1
PC11	OUT	LED Wi-Fi 2
PG11	OUT	Home button LED
PB7	OUT	Status LED
PA0 - PA3	OUT	Multiplexer config

Tab 3. Pins we've partially reversed-engineered

Pin name	Type	Comments
PB15, PB14, PD9, PG3, PC12, PA15, PD4, PG15	???	The circuit takes additional current if we change the default state on those pins.
PC13, PG7,	OUT	After applying high voltage to those pins, voltage on the circuit board significantly rises and turns on multiple components of the circuit, for example LEDs.

Motherboard probably is at least 4 later standard fr4 PCB. It contains 4 dedicated DC H-bridge based motor drivers for brushes and wheels, and a vacuum motor turned on by a regular transistor. The robot is powered by a 4s1p Li-ion battery, with integrated BMS. It can power up the robot for about 30 minutes of work. The robot is charged by 2 steel terminals at the bottom of the robot using the original docking station.

Using Roomba as base means we've access to all of its sensors, motors, encoders and more. However, not all of those could be reliably accessed via pins microcontroller left by the original microprocessor, so in some cases we've hacked-into the sensors directly, ignoring tracks on the original circuit board. Others (most importantly, motors) were accessed through pins left by the original microprocessor.

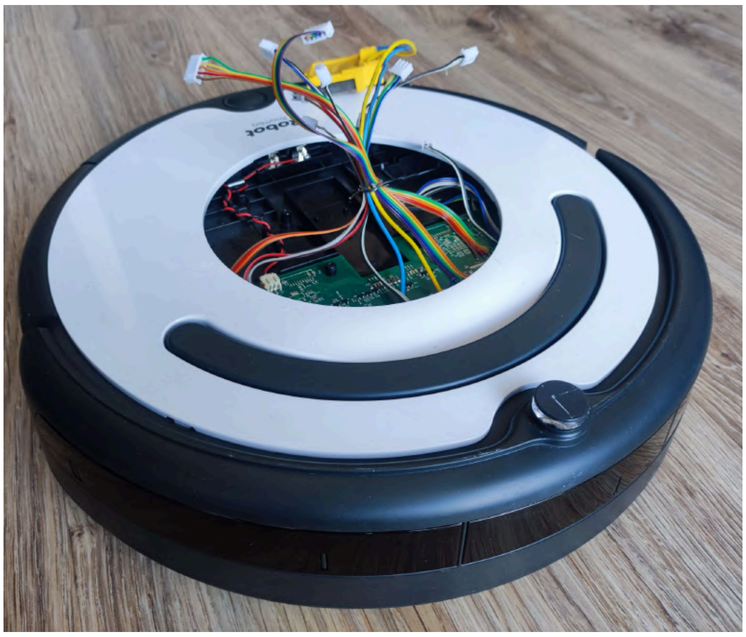

Fig 4. *iRobot Roomba* with connectors hacked-in by us.

In Fig 4 you can see an iRobot Roomba base with connectors hacked-in by us. It was done to allow safe connection to pins on the motherboard. Due to the size of soldering tip relative to the size of pins on the motherboard, shown in Fig 5, soldering those connectors was a massive challenge.

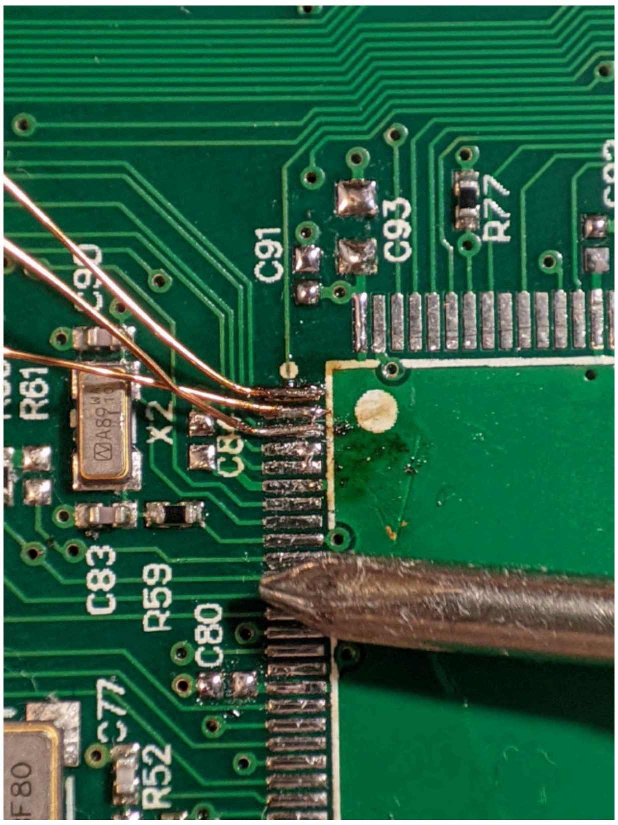

Fig 5. Size of original microprocessor pins relative to size of soldering tip

Fig 6 and 7 shows the original motherboard of iRobot Roomba, from top and bottom view accordingly. You can see on those that our connectors were soldered to pins of the original microprocessor, but also to many other pins on the board, because not all sensors could be reliably accessed via microprocessor's pins - we couldn't figure out how communication with them works from that point.

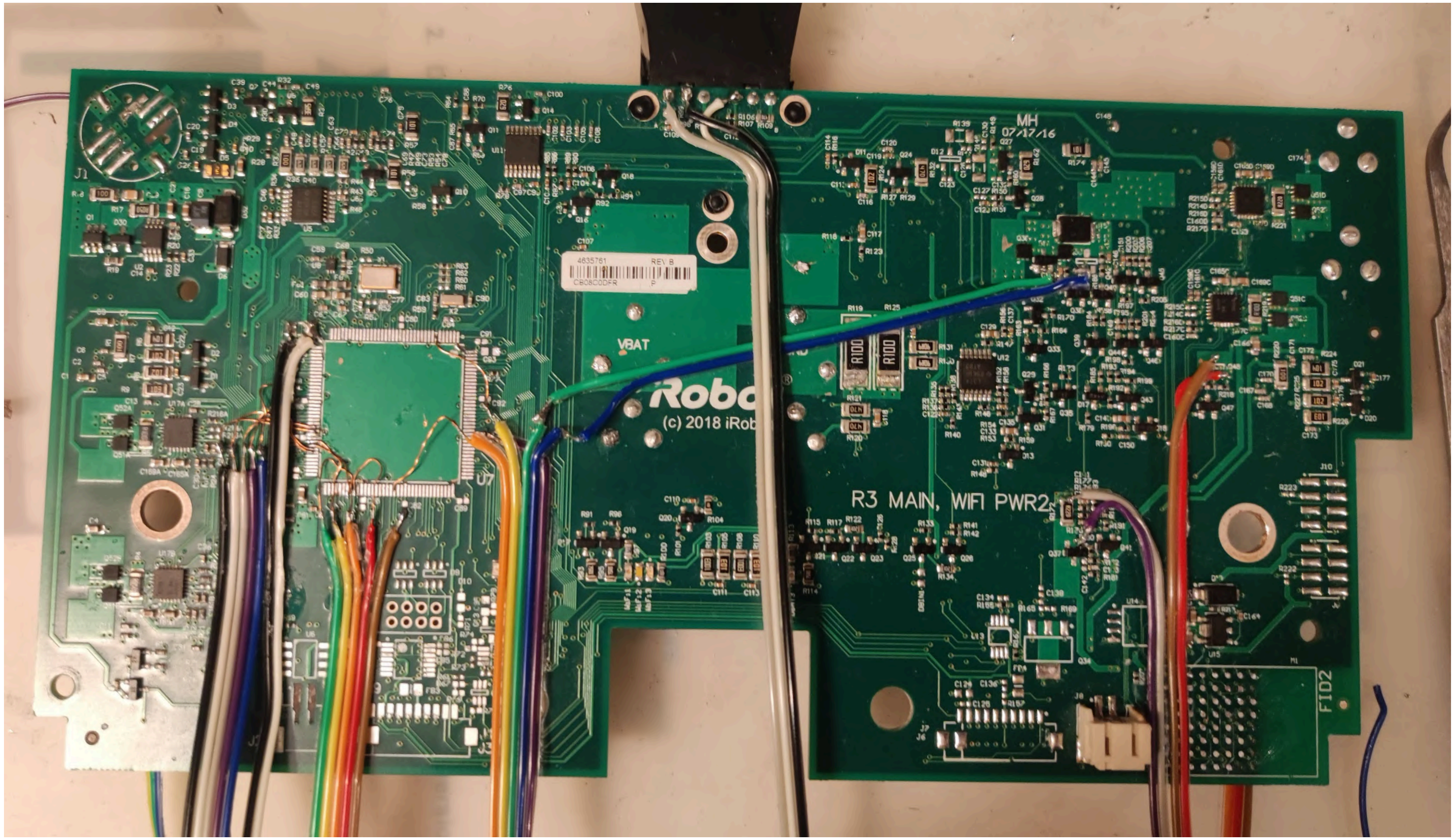
Fig 6. Motherboard of *iRobot Roomba* - top view

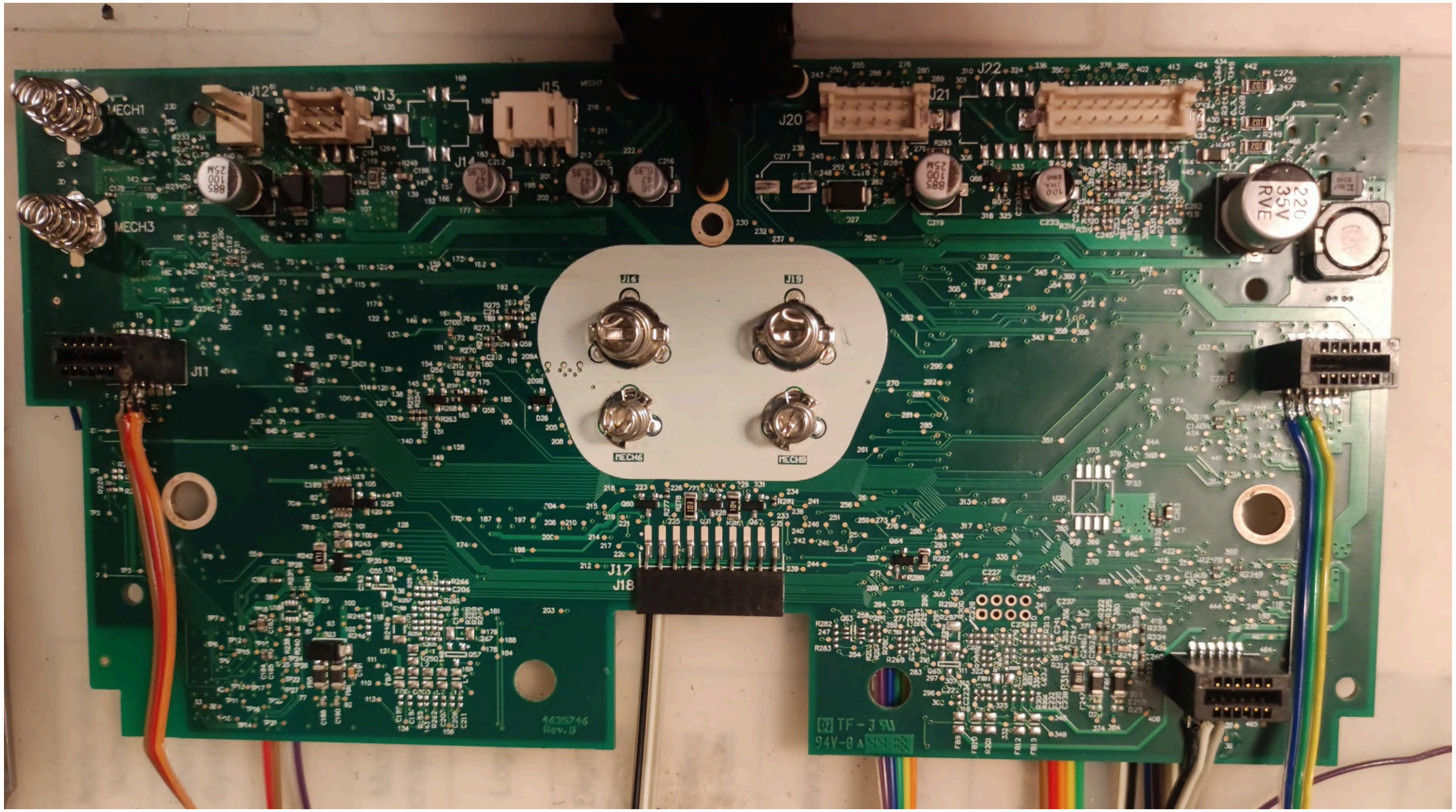
Fig 7. Motherboard of *irobot roomba - bottom view*

Two motors and encoders

- Hall effect sensors in encoders
- 508.8 pulses per full wheel rotation
- 10 - 18 V DC brushed motors
- 63.6:1 Gear ratio

Six infrared proximity sensors

On the Fig 8 we can see an array with built-in six reflective IR sensors. They are essential for our robot to avoid crashing into a wall, or other objects on the floor. Also, during the process of mapping, left side sensors allows the robot to keep track of the wall, instead of facing it with the front all the time, which greatly reduces time needed to map edges of the room. Sensors work as a pair of IR diodes (transmitter) and phototransistor, which receives a signal from the reflected object. IR diodes are simultaneously powered on end off, to read the intensity with ambient light, and own light.

- Effective distance: 20 cm
- Analogue signal: 0.5 V to 3 V
- Relationship between signal and detected distance is NOT linear.

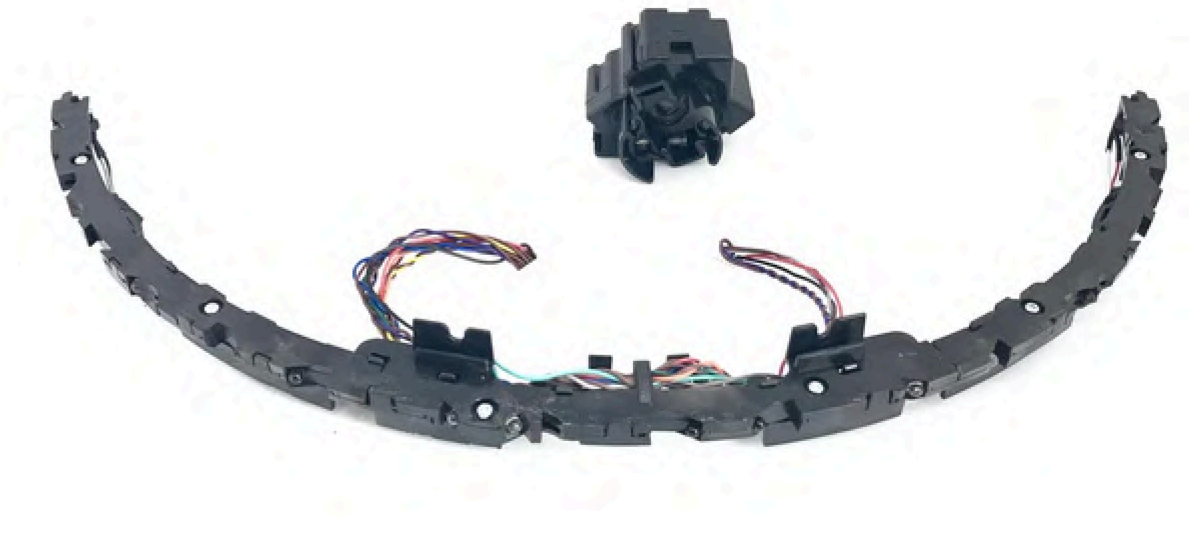

Fig 8. Photo of sensor array with pairs of IR diodes and phototransistors from robot

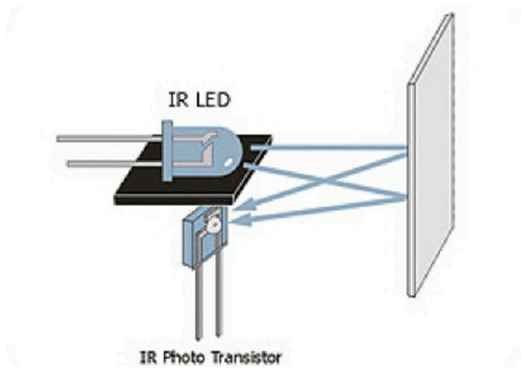

Fig 9. Visualization of infrared proximity sensor's application

Two bump sensors

Bump sensors are also optical. Because of very high usage of them, mechanical sensors would be likely to fail very quickly. As their only mechanical part is spring and lever (Fig 11), they can out stand life of mechanical sensors many times. Also, due to this, they are a lot quieter, which is a nice addition, as it has to be a household robot. They work with the same principle as slot transoptors or barrier light sensors as in Fig 10 and 12, where light is constantly powered on, and can be blocked by a lever.

- Binary
- True (bumper pressed) for 3.3 V
- False for (bumper not pressed) 0 V

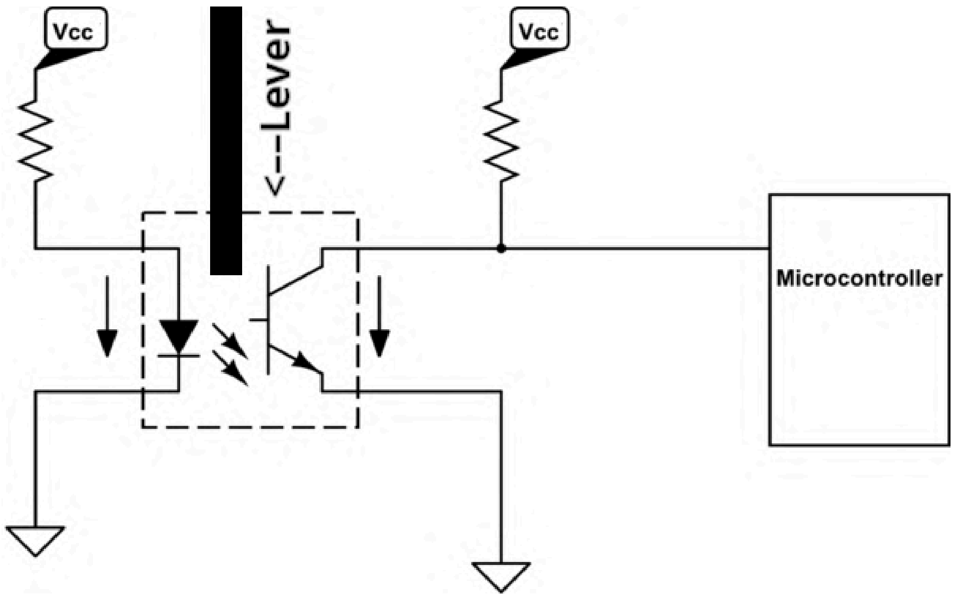

Fig 10. Schematic of bumper sensor

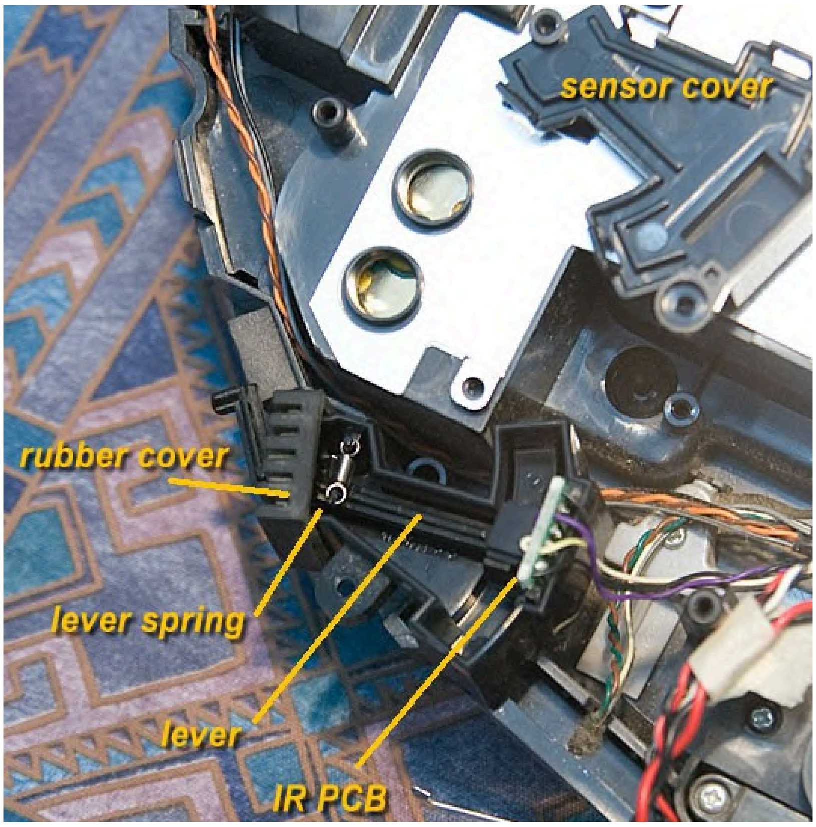

Fig 11. Photo of disassembled bumper sensor

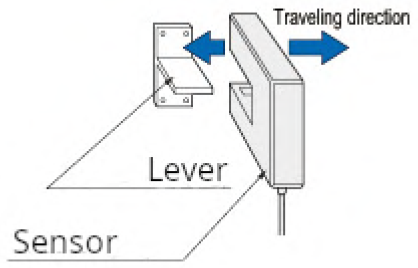

Fig 12. Visualization how does sensor work

Wheel-drop sensors

- Mechanical switch
- Turns on if robot is picked-up

To be clear: those are not all sensors built-into Roomba’s body. There are other sensors that could be useful for us, but we had no time nor skill to reverse-engineer them. Nevertheless, what we’ve managed to reverse-engineer is enough to create a rudimentary autonomous vacuuming robot.

4.2 ESP32 microcontrollers

Two ESP32 microcontrollers have been installed on the prototype board as seen in Fig 14. Their purpose is to provide an API for communication between Raspberry Pi and physical robot. One of the microcontrollers is responsible for receiving readings from sensors and encoders, another for sending signals to motors.

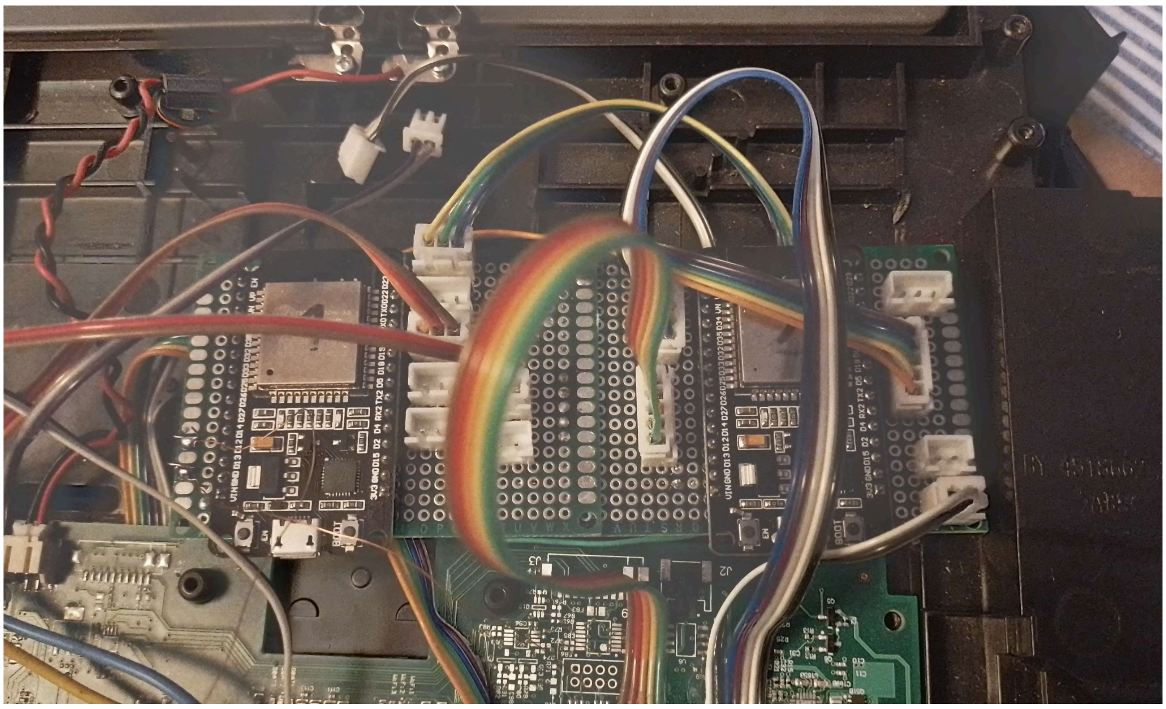

Fig 13. Two ESP32 microcontrollers on prototype board

4.3 Raspberry Pi

Raspberry Pi is thinking brain of the project. It is responsible for all decisions and uses other parts of the project as tools. We’ve decided to use Raspberry Pi 3B and set up a connection between it and microcontrollers using Wi-Fi.

4.4 Camera

Fig 14. Plantronics EagleEye camera used in the project

Camera was an integral part for implementing computer vision. This exact model was for us very helpful, because it has wide field of view, of 60 degree, autofocus, automatic exposure control and automatic AWB, which helped a lot our Raspberry Pi to reduce computer power used to control video parameters and in the result quicker recognition of objects.

5. Description of the software

All software written for this project is proprietary. Most important frameworks used are: ROS 2 Galactic [6], Micro-ROS [7] and ESP-IDF [8]. The project also uses OpenCV [9] to receive pictures from an installed camera and perform a computer-vision task to find pets blocking the path.

The basic idea of operation, visible in Fig 2, is: Raspberry Pi is the thinking brain of the project. It receives readings from sensors sent through ROS topics and responds with values to be set on actuators, sending them through other ROS topics. Microcontrollers ESP32 use Micro-ROS to provide sensor data to and receive values for actuators from Raspberry Pi, and *(microcontrollers)* use ESP-IDF to read/write voltage values from their pins.

Because of how niche the theme of our project is, there's barely any literature and no plug&play packages we could use to implement features we've decided to implement. Because of this, algorithms used are designed by us and the whole source code is proprietary.

The project consists of four programs: two written for Raspberry Pi in Python and two separate programs written in C language for ESP32 microcontrollers. All code is available on both the project's repository on GitHub [1] and in *the source* directory attached to this report. Usage of GitHub repository is preferable, since it always has up-to-date code.

5.1 Layers of abstraction

Software is implemented with layers of abstraction, one built on top of the other. As seen in Fig 2 and Fig 15 the **lowest layer** is the microcontroller, receiving data from physical robot and sending voltage to motors through HAL.

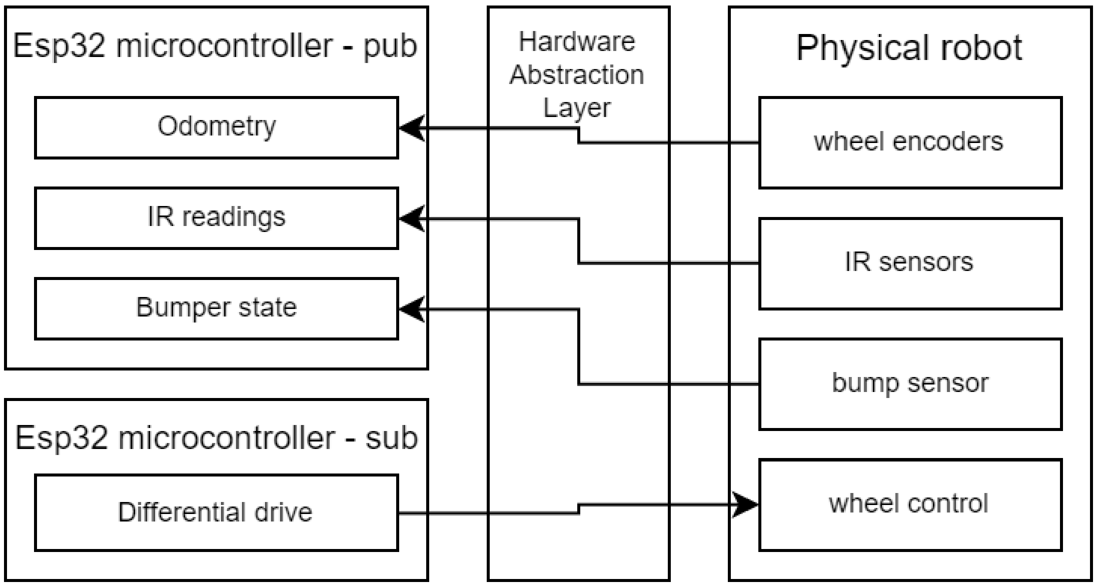

Fig 15. ESP32 microcontrollers communicate with robot through HAL

HAL *(Hardware abstraction layer)* is an API provided by manufacturers of microcontrollers, in our case it's ESP-IDF [8]. It allows our code to receive signals from and send signals to pins without delving deep into hardware details of a given microcontroller.

There are two ESP32 microcontrollers: one serving as analog-to-digital converter, receiving reading from physical robot and passing them onwards to ROS topics, and another serving as digital-to-analog converter, receiving values from ROS topics and steering actuators accordingly. Both are programmed using micro-ROS framework and utilities HAL to perform their tasks.

ROS topics itself are considered the **bottom layer of abstraction**, this name is misleading because it isn't the lowest layer, but it sticks. Usage of ROS 2 Galactic and topics model allowed for relatively easy integration between ESP32 and Raspberry Pi. ROS topic is essentially a queue to which *a publisher* can publish *(share, provide)* information that can be received by *subscribers*.

The middle layer of abstraction is implemented on Raspberry Pi. In Fig 16 you can see the block diagram, showing input and output of the middle layer and its connection to the bottom layer *(ROS topics)* and top layer. The bottom layer for physical robot is different from for simulator we've used, so by using a separate middle layer providing unified API for top layer, we could implement algorithms in top layer, test them in simulator and then migrate to physical robot. Another way to look at the middle layer: it's a wrapper for the bottom layer.

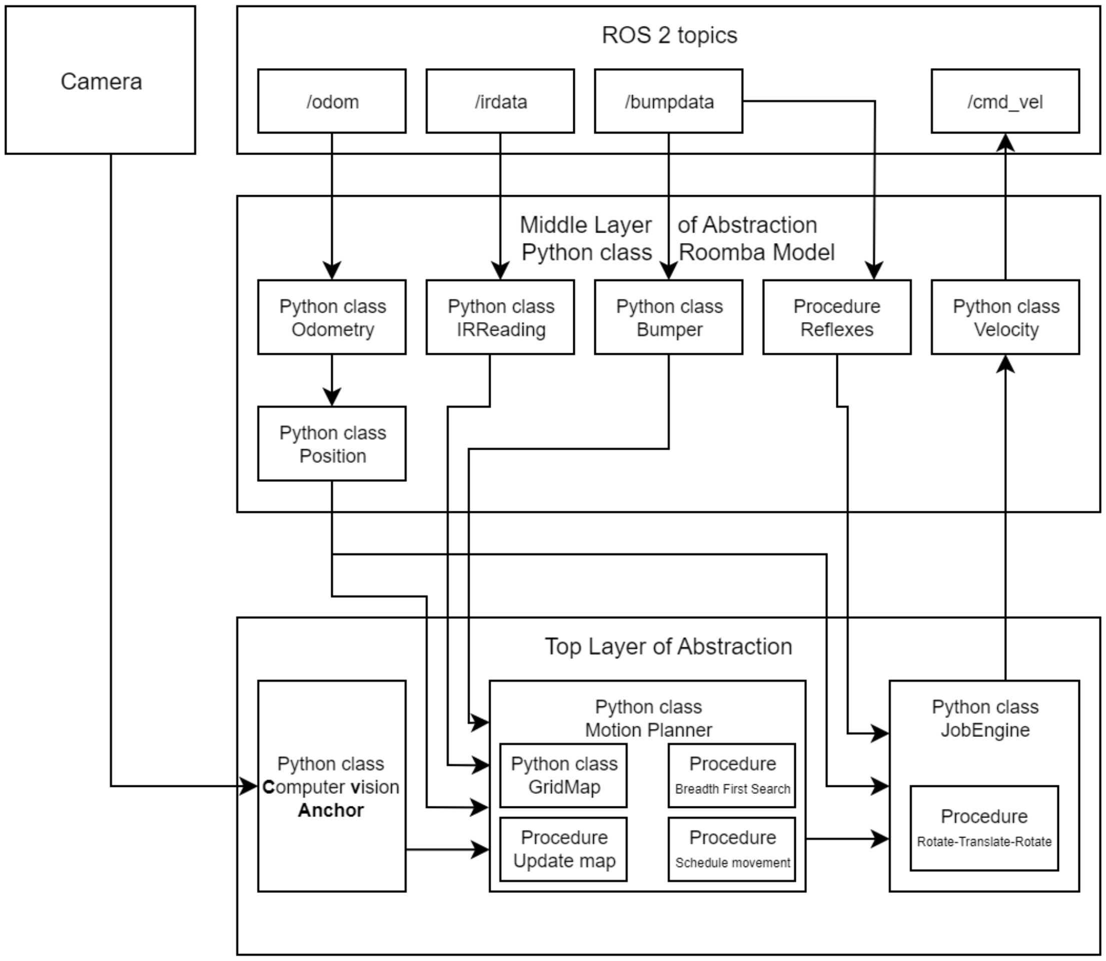

Fig 16. Middle and top layer of abstraction

The top layer of abstraction is the space within which the most complicated *(software-wise)* part of our project is implemented. Most important parts of top layer *(and connection to middle layer)* is shown in Fig 16: mapping & pathfinding algorithms implemented within Motion Planner class, movement from point to point using *rotate-translate-rotate* algorithm implemented in JobEngine class, also computer vision *(detection of pets blocking the path)* within CvAnchor class.

5.2 Movement from point to point - JobEngine class

JobEngine is a Python class written in the top layer of abstraction. A block diagram of it is shown in Fig 18. JobEngine takes Job object *(which is desired position and orientation of robot)* and current position of robot as input *(there's also a reflex procedure that works as interruption, removing all queued Jobs and stopping movement of robot, but it's interruption, not input)* and perform ***rotate-translate-rotate*** algorithm to reach desired position.

Jobs can be created with following syntax:

- Job.Absolute(x,y,angle) - end position is given in absolute coordinates
- Job.Relative(x,y,angle) - end position is given in relative coordinates
- Job.Rotate(angle) - rotate by given angle
- Job.Translate(distance) - move forward by given distance (in respect to current rotation)

JobEngine **works only with absolute coordinates**, so all other syntaxes are eventually converted into absolute ones. Angle usually can be omitted *(set to None)* which allows robot to finish in any rotation it had once reaching given *x* and *y*.

Rotate-translate-rotate is a pretty simple algorithm, first it rotates the robot towards the final destination, then it translates forward until X/Y coordinates of the destination are reached, finally it rotates again to achieve the desired orientation.

Rotation and translation is done iteratively: by using middle-layer class Velocity angular/linear speed is set in every iteration until the difference between desired rotation/position is within hardcoded absolute tolerance.

Perfect localization given to Job may never be reached because of many reasons, chief among them is frequency rate of code execution - code executing the movement is called with every tick of ROS timer, which is set to small yet significant value. To solve this problem, we've introduced the parameter **AbsoluteTolerance** - which stands for maximum error that can appear on coordinates *x y* and *angle*.

5.3 Reflexes - reaction to hazards

Reflexes are a feature of the middle layer. For now, there's only bump reflex, but the codebase for other types is there, it's pretty flexible.

Reflex is triggered when any hazard is detected (for example, when robots hits obstacle with a bumper) Reflex instantly stops the robot, then calls top layer function *ObstacleReached* to let programming know that collision occurred *(so it can be taken into consideration in SLAM)* but it also overrides top layer control flow - **top layer can't control robot during reflex**,

Behaviour of the robot during reflex is defined by function *ObstacleReflexMoonwalk*, which has access to all readings from the robot so it can take proper action to safely get away from the hazard. For now, it only moves backward, but more advanced algorithms can be implemented.

After reflex behaviour safely removes the robot from proximity of hazard (obstacle) top layer function *ObstacleAfterReflex* is called.

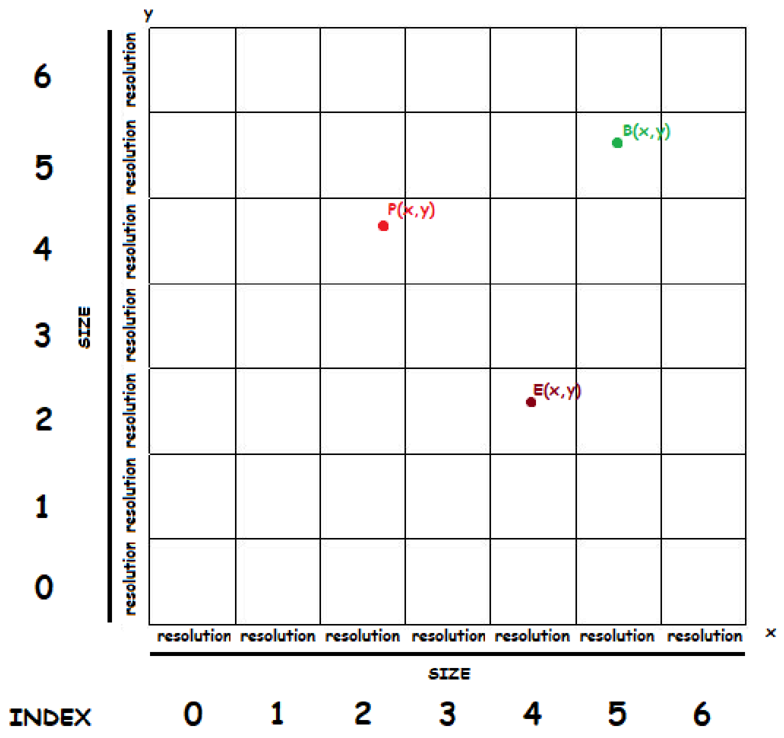

Fig 17. Example of grid map implementation

5.4 Mapping and path planning - Motion Planner class

Mapping and path planning is implemented within Motion Planner Python class, operating in the top layer of abstraction. A block diagram of it is shown in Fig 19. To describe shortly operation principle: it uses our own rudimentary algorithm for mapping *(using gridmap, odometry, bumper sensor and IR sensor)* and Breadth First Search algorithm for finding the nearest unexplored place within the map.

Map is represented using GridMap, an example of which is shown in Fig 17. **The main idea of the grid map is: divide the full area of *SIZE x SIZE* into squares of size *RESOLUTION x RESOLUTION***. This way, it discretized continuous coordinates (x,y) into indexes in the array.

For any point with coordinates (x,y) we can find indexes on X and Y discretized axes with formula 1 and 2:

$$ind_x = round(\frac{x}{resolution}) \qquad (1)$$

$$ind_y = round(\frac{y}{resolution}) \qquad (2)$$

(ind_x, ind_y) stand for indexes in GridMap

Mapping algorithm is implemented in Motion Planner class, as shown in Fig 19. It uses bumper sensors and IR proximity sensors to check for obstacles in the nearby area.. By passively traversing the room, the robot marks all visited places as “empty” *(if it can drive into it, then there’s no obstacle)*. If its sensors detect something, then it is marked on the map. Current position of the robot is taken naively from odometry.

Path planning, also implemented in Motion Planner class and shown in Fig 19, uses *a Breadth-First-Search* algorithm to find the nearest unexplored piece of the grid, then schedules movement to it via JobEngine. Movement isn’t smooth: robot moves from one piece of grid to the other without any optimizations, only horizontal and vertical movement allowed *(no diagonals)*

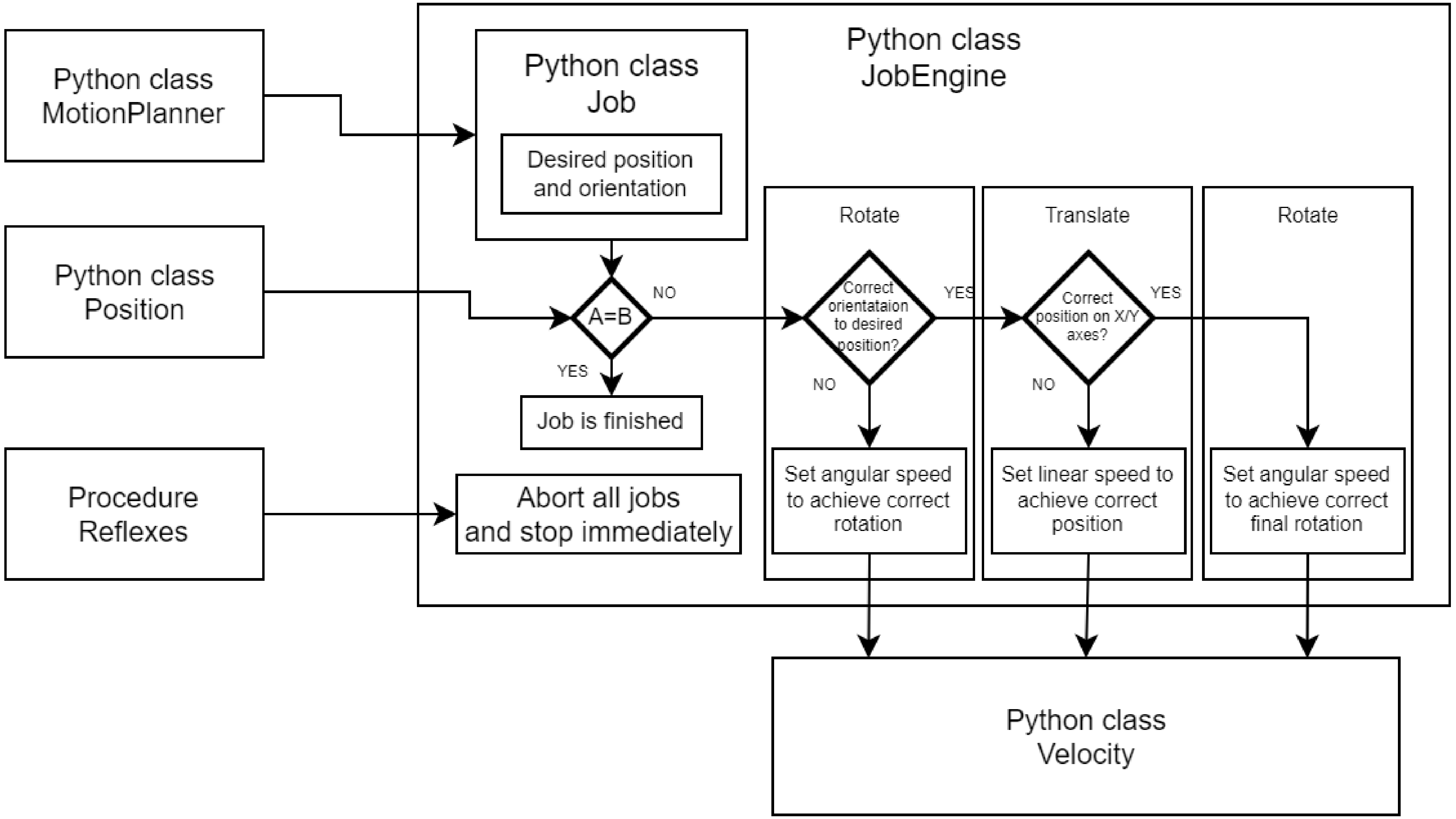
Python class MotionPlanner
Python class Position
Procedure Reflexes
Python class JobEngine
Python class Job
Desired position and orientation
A=B
NO
YES
Job is finished
Abort all jobs and stop immediately
Rotate
Correct orientataion to desired position?
YES
NO
Set angular speed to achieve correct rotation
Translate
Correct position on X/Y axes?
YES
NO
Set linear speed to achieve correct position
Rotate
Set angular speed to achieve correct final rotation
Python class Velocity

Fig 18. JobEngine - class implementing movement from point to point using *rotate-translate-rotate* algorithm

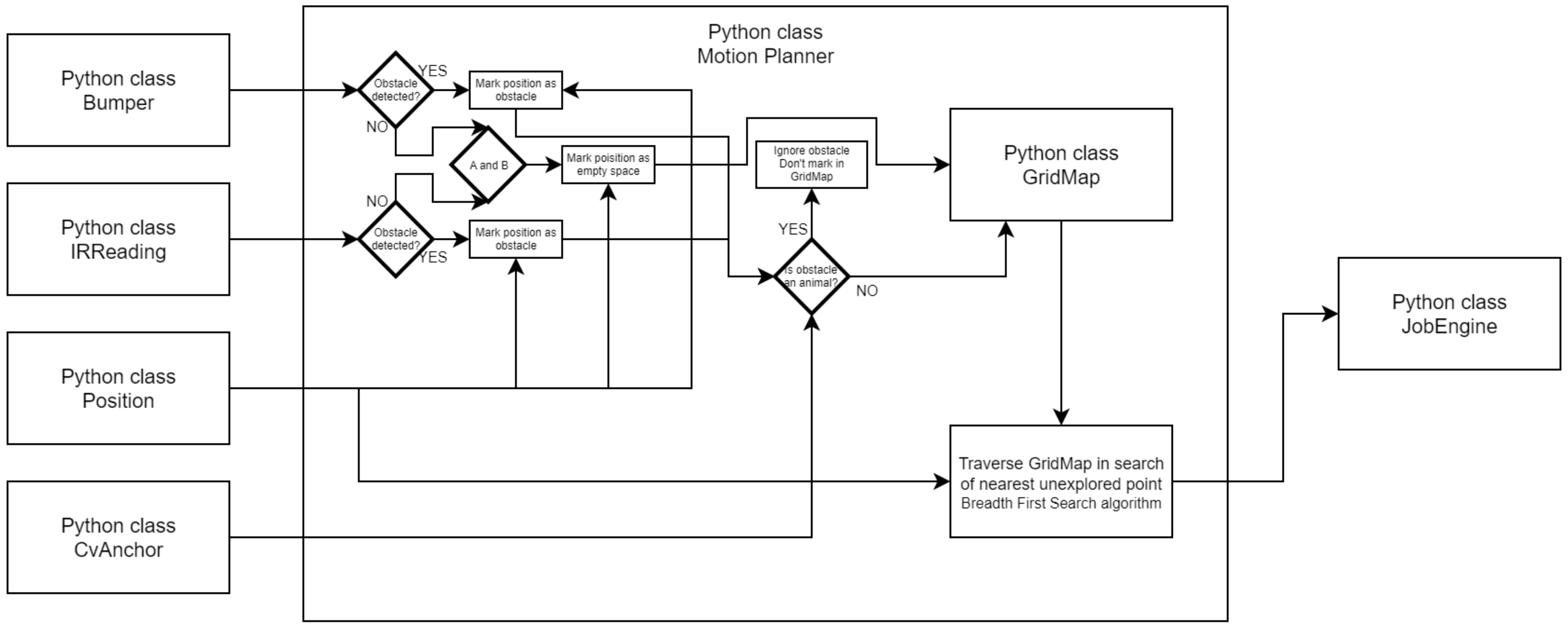
Python class Bumper
Python class IRReading
Python class Position
Python class CvAnchor
Python class Motion Planner
Obstacle detected?
YES
NO
Mark position as obstacle
A and B
Mark poisition as empty space
Obstacle detected?
NO
YES
Mark position as obstacle
Ignore obstacle Don't mark in GridMap
YES
Is obstacle an animal?
NO
Python class GridMap
Traverse GridMap in search of nearest unexplored point Breadth First Search algorithm
Python class JobEngine

Fig 19. Motion Planner, class implementing mapping and pathfinding, using Breadth First Search algorithm in maze

5.5 Simulations

We've used ROS 2 framework integrated with Gazebo to perform simulations. We've placed a model of *iRobot create 3* into *AWS Robomaker Small House* [3] to simulate a robot in a house-like environment, as seen in Fig 20.

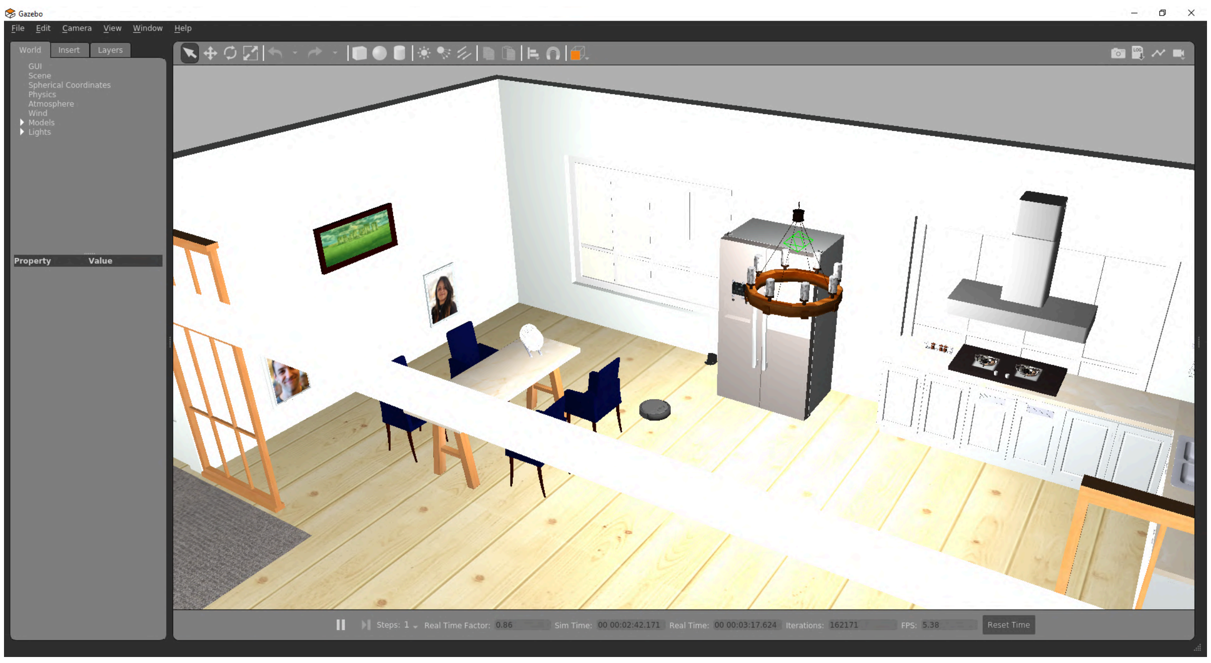

Fig 20. Simulation of robot in a house

During simulations, all implemented algorithms: movement from point to point with *rotate-translate-rotate*, mapping and path scheduling worked properly. I strongly recommend you to check either GitHub repository of project [1] or alternatively *simulations* directory attached to this report, where it's possible to watch recordings of those simulations as neither words nor pictures can show that it indeed works as intended.

"As intended" doesn't mean everything is perfect. Mapping algorithm is very rudimentary, odometry gradually becomes disconnected from reality overtime and map representation is terribly oversimplified, causing the robot to stumble onto obstacles much more often than it should.

Three samples of simulations are currently available on github/simulations directory: *reflex.mp4*, *visit_points.mp4* and *mapping & path planning.mp4*.

reflex.mp4 shows a test of reflex behaviour. Reflexes are reactions of a robot to hazards, for example cliffs or walls. Once a hazard is detected, it's necessary for a robot to get away from it. Simulation scenario is simple: first the robot is moved to the desired location in front of the wall, then the top layer orders forward movement (towards the wall) Once the robot hits the wall, reflex kicks in, strips control from the top layer and orders backward movement. When it's safe, control is returned to the top layer, which orders forward movement and the cycle repeats. Results are okay; there's visible lag between hit and reflex kicking in, but it's unclear whether it's due to imprecise geometry of used models or ROS & Gazebo overhead. However, the reaction reliably happens and the entire algorithm *(of reflex)* executes correctly.

visit_points.mp4 shows a demo of JobEngine class - movement from point to point. We've hardcoded a few points which the robot has to visit in a given order, and it did. In the video one can see robot rotating-translating-rotating from point to point, slowing down as it reaches correct X/Y coordinates and also uncertainty on coordinates staying within absolute tolerance (0.01)

mapping & path planning.mp4 shows a demo of mapping algorithm, although it doesn't display any map because it would be very hard for us to visualize it at this point. In the video, the robot moves visibly in a grid-like pattern, checking nearby grid points with its bumper, and moving to another point after an obstacle is hit. Map resolution in the sample is too high, but it was useful for debugging and presentation purposes. For a real robot, it will be necessary to do a lot of *"fine-tuning"* of parameters.

6. Start-up, calibration

We've failed to finish the project so start-up can't really be described, and we didn't do any calibrations of a robot in natural environment. However, we made a test of AI and CV components in simulations and real world. We have also ensured that all sensors are working correctly

At the moment, robot require battery to be manually attached and both ROS agent and Raspberry PI Python program to be started manually. Furthermore, miscrocontrollers and raspberry PI must be connected to the same, wireless network *(connection of microcontroller is hardcoded, so we would have to attach router to the robot in the end, but it was incredibly useful during development)*

6.1 Setup of simulations

Code created for this project is available on GitHub [1]. It consists of three ROS packages: *astableinvention*, which is our code, and two external packages: *create_3sim* and *aw-robomaker-small-house-world*. Note that the model of *iRobot Create 3* was slightly modified by us, to disable bump reflexes.

Commands to create workspace, download repository, build and run the project:

```
source /opt/ros/galactic/setup.bash
mkdir -p ~/ros2_ws
cd ros2_ws
git clone https://github.com/AzethMeron/AstableInvention.git
mv AstableInvention src
rosdep install -i --from-path src --rosdistro galactic -y
colcon build
source install/local_setup.bash
```

To run the simulation: `ros2 launch irobot_create_gazebo_bringup create3_gazebo_aws_small.launch.py`
To run astableinvention node: `ros2 run astable_invention main`

Note: branch `master` is for simulation, `robot` for real robot (supports topics programmed by us in the microcontroller) Those two are currently developed out of sync, mostly due to our inexperience with usage of github.

Note 2: Github repository will be changed at one point to cover the entire project, including programs for microcontrollers. Instructions above will then become outdated.

6.2 Testing AI and CV

To examine the accuracy of object detection and, more importantly, distance estimation, we created a testing environment. On the testing surface, tape was indicating the field that our camera can reach which is 60 degrees and when marking 0 degree as the center of view 30 degrees to each side. Another tape was placed horizontally to the camera lens, those three are indicating consecutively 50 cm, 100 cm, and 150 cm distance from the setup.

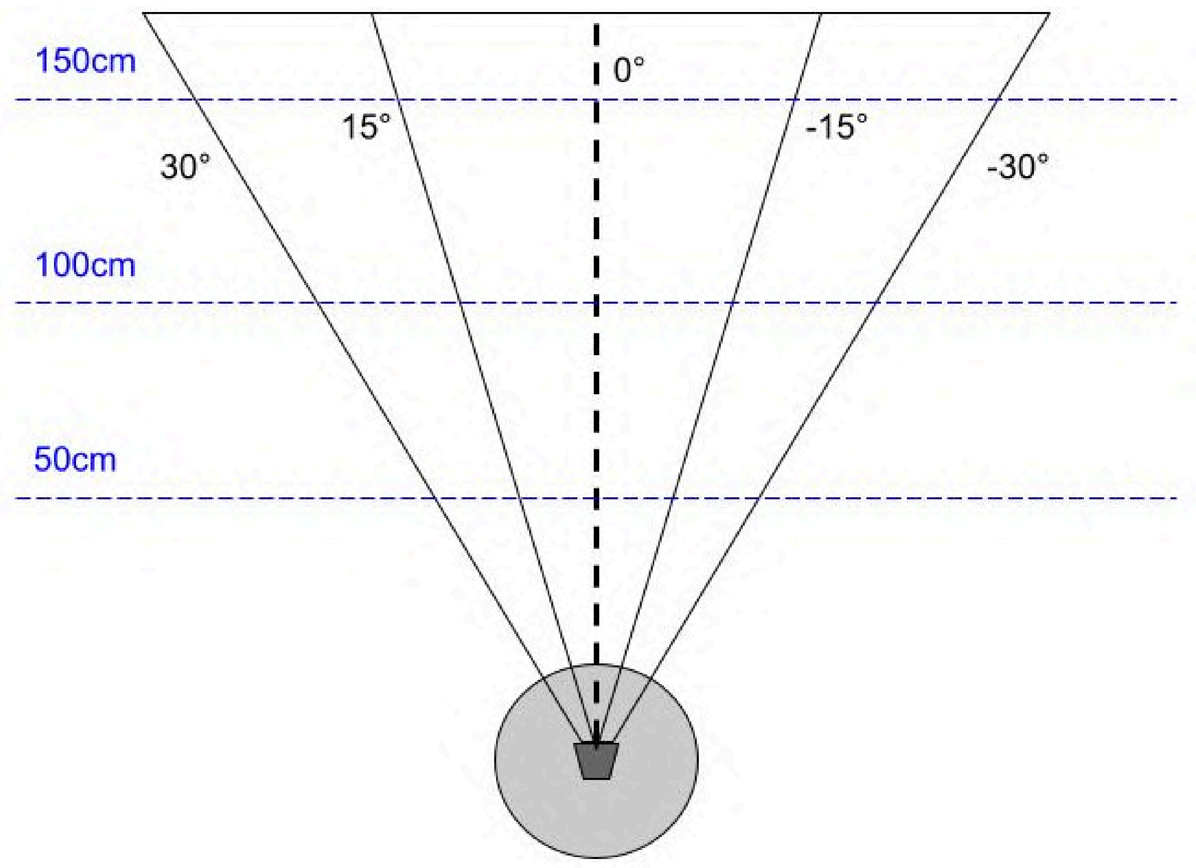

Figure 21 Representation of created testing surface.

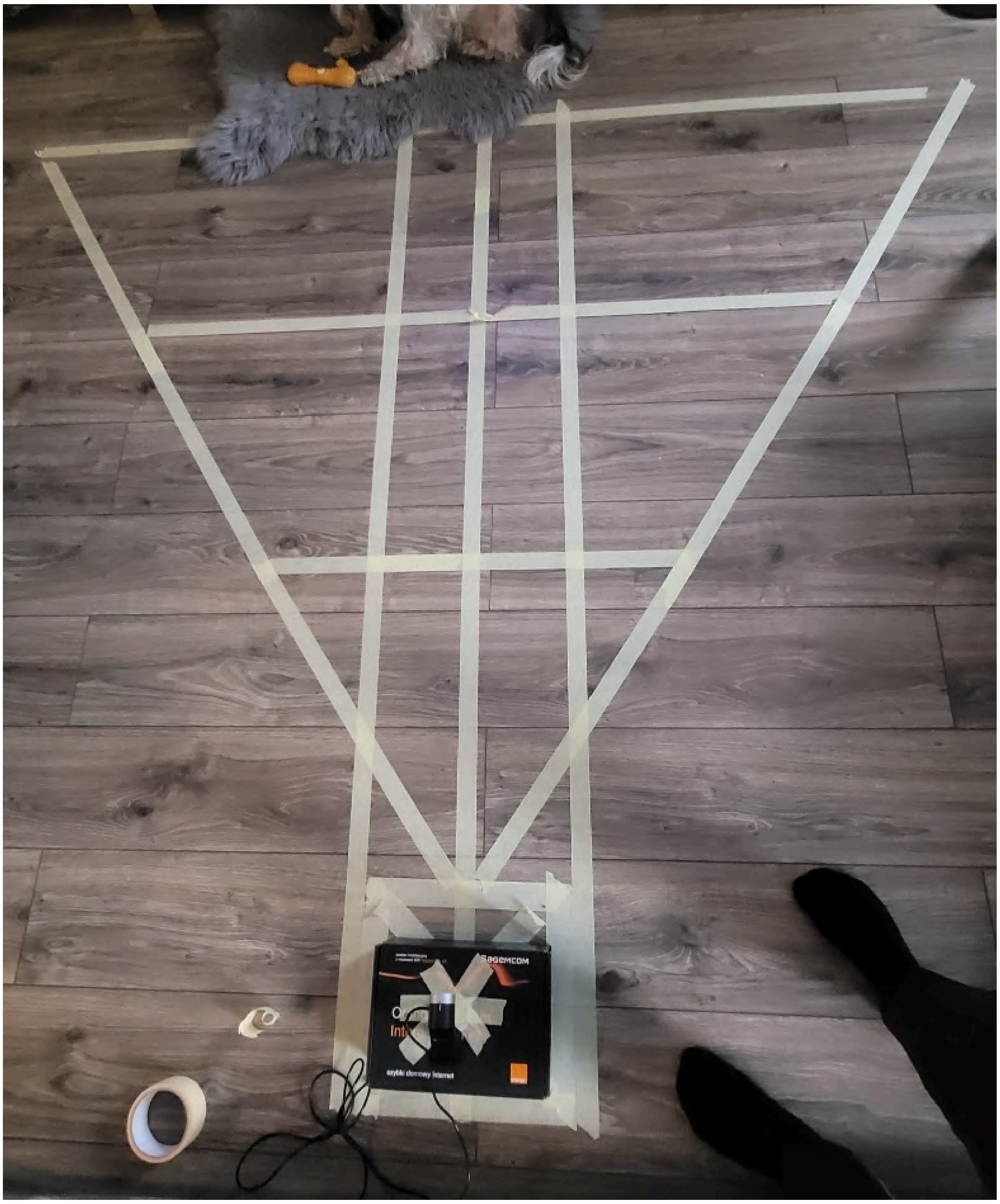

Figure 22 Testing Surface

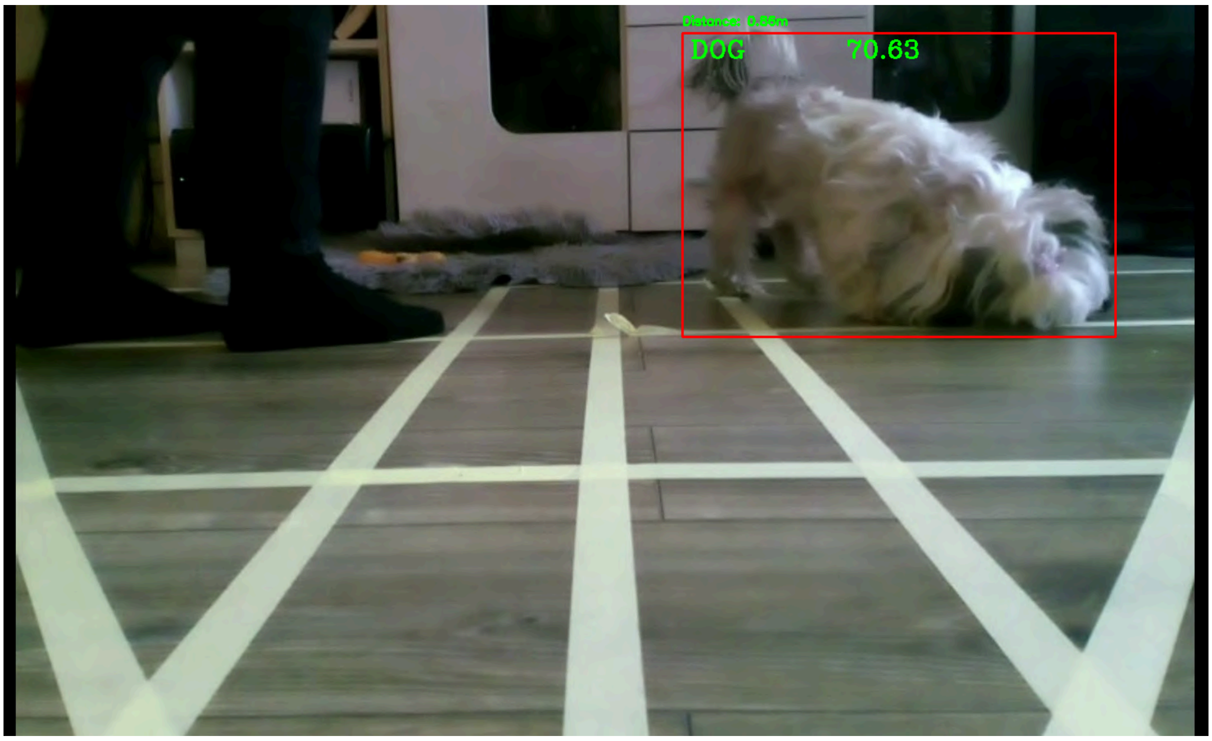

Figure 23 Example of distance estimation with satisfactory result

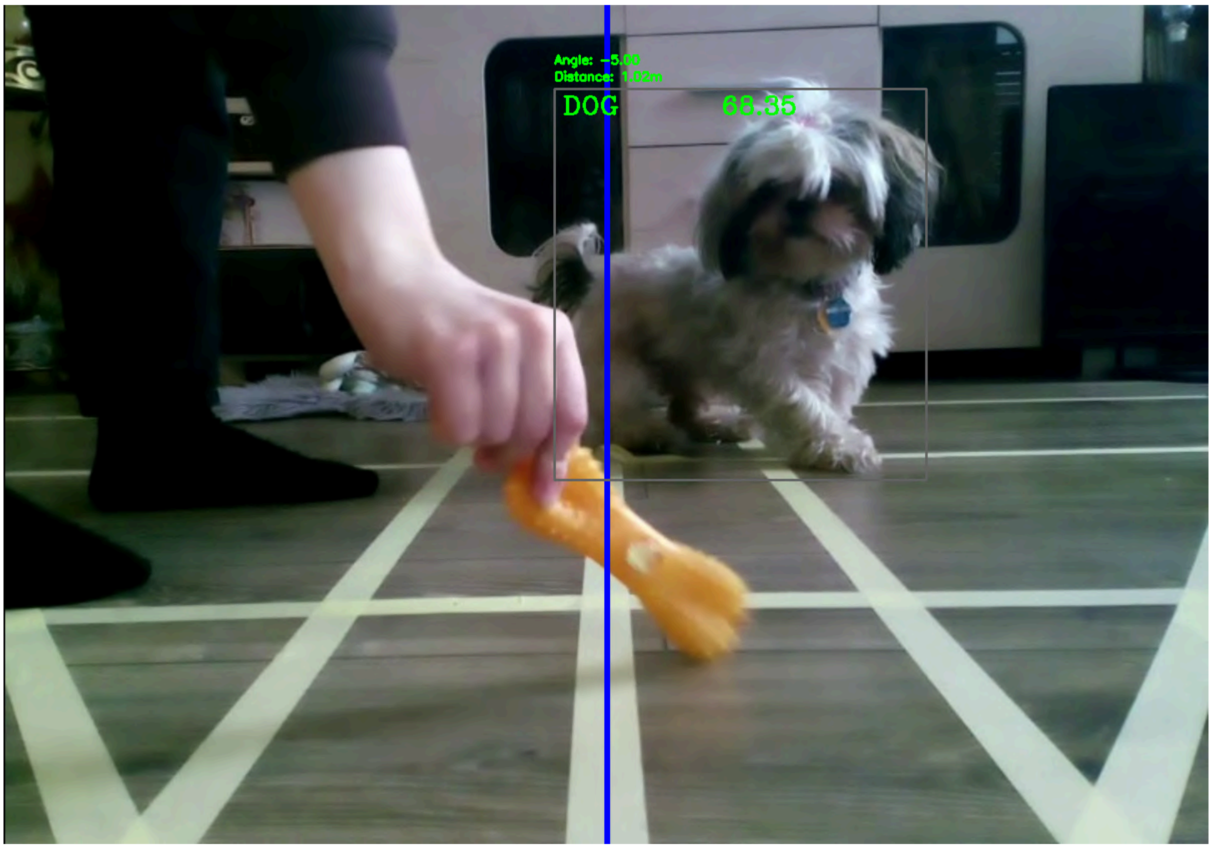

Figure 24 Example of angle estimation

As it turns out, we managed to achieve distance estimation whose accuracy highly depends on the position of the object towards the camera. So, if the object is facing the camera the bounding box is narrower than expected in the object's width, we estimated accordingly the distance can be disturbed. Nevertheless, we could be happy with our achievement but for security reasons, we added a few lines of code to catch threatening errors in the distance estimation. Because of the placement of our camera on a 10 cm high system and facing it at an angle slightly towards the surface, we obtained the state where the middle of the frame height is indicating exactly 100 cm of the distance measurement in front of the camera. This allowed us to protect ourselves from unnecessary accidents and retrieve the bottom coordinates of the middle of the bounding box affix condition that if said coordinate is found under the indicating the 100 cm mark line, the system receives additional caution regarding the distance of the detected object.

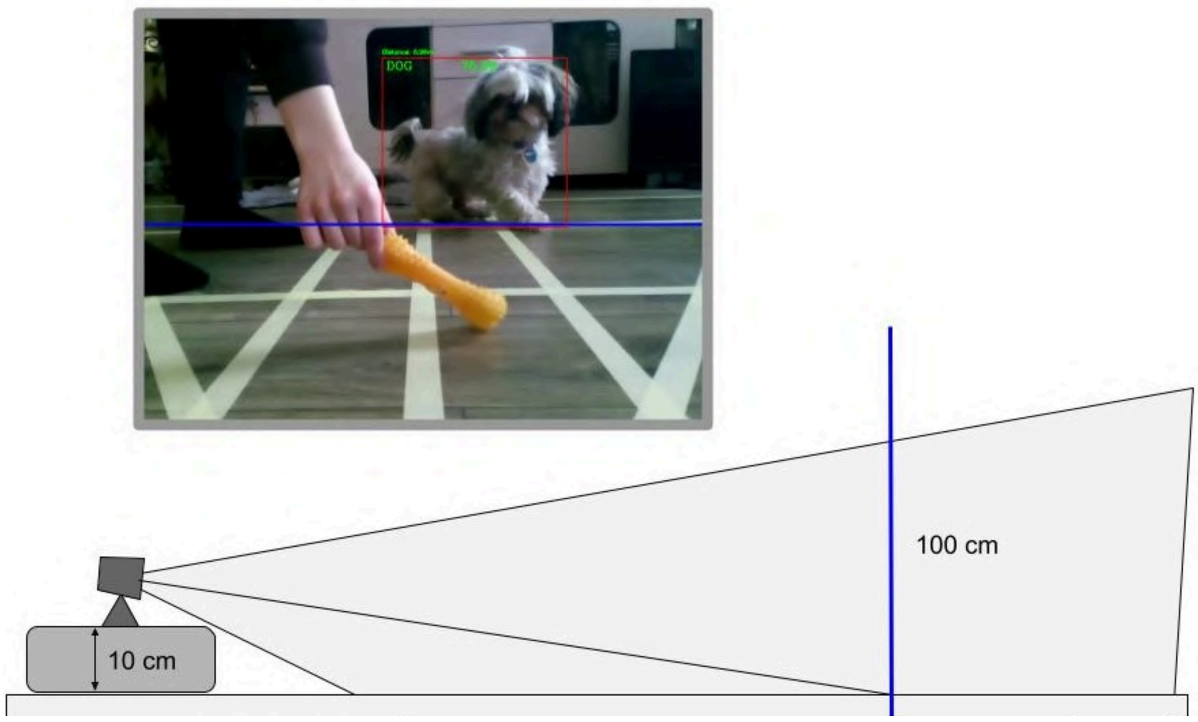

Figure 25 Explication of created test environment and 100 cm mark line

6.3 Calibration of sensors

Calibration was pretty straightforward. We just had to check if different sensors give us symmetrical results. Then check which intensity corresponds to a distance which is acceptable for us to safely stop the robot, which we should probably test in reality, but we did not manage to run the robot, so these are our estimations. Testing environment is in Fig 26.

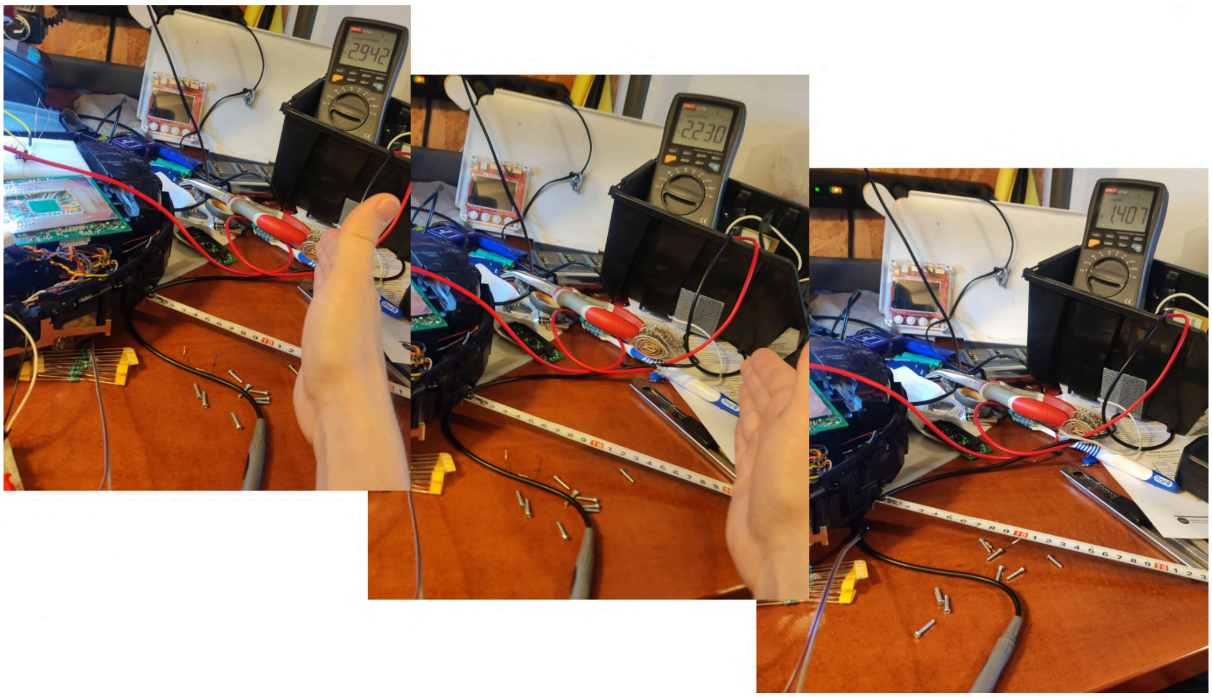

Figure 26 Sensors calibration

7. User manual

This chapter is written as-if the robot was successfully finalized. That's not the case (yet?) but we had to write here something.

Welcome to the manual for the Astable Invention vacuuming robot created by students. This robot utilizes infrared (IR) reflective sensors, odometry and computer vision to navigate and map a room, providing a thorough cleaning of all surfaces while avoiding any pets such as cats and dogs.

Getting started:

1. Make sure the robot is fully charged before use.
2. Turn on the robot by pressing the power button located on the top of the robot.
3. Press the start button to begin cleaning.

Maintenance:

1. Empty the dustbin after each use.
2. Clean the brushes and suction motor regularly to ensure optimal performance.
3. Keep the sensors, odometry system and computer vision free of debris to ensure accurate navigation and object detection.
4. Charge the robot after each use to ensure it's ready for the next cleaning session.

Troubleshooting:

- If the robot is not moving, make sure it is turned on and charged.
- If the robot is getting stuck, make sure the sensors, odometry system and computer vision are clean and free of debris.
- If the robot is not cleaning effectively, make sure the brushes and suction motor are clean and in good working condition.
- If the robot is not avoiding pets, make sure the computer vision is properly calibrated.
- If the robot is not returning to the charging dock, make sure the dock is in a clear, open space and the battery is charged.

By following the instructions in this manual, your Astable Invention vacuuming robot will provide efficient and effective cleaning for your home while avoiding any pets. If you have any further questions, please contact the student team for assistance.

8. Summary

We’ve failed to finish the project, although it was a really close call. We’ve managed to establish communication between Raspberry PI and microcontrollers, we’ve correctly implemented reading values from sensors using HAL and we nearly managed to implement motor controls. We can say this because reflex behaviour after press of a bumper works 100% correctly. However, robot isn’t publishing odometry and it doesn’t move exactly the way we expect it to *(probably error with rotational movement but hard to say)*

However results in simulations suggest that there are no major problems with overlying algorithms and we’ve reasons to believe the only problem is with microcontroller programming, and it’s a minor one at that *(because reflex behaviour works 100% correctly and it requires reading sensor data and steering motors)*

From the programming point-of-view, the biggest failure is missing SLAM. After reading [4] we were convinced we can make it work with only bumper, odometry and IR sensors. We still believe it is possible, but all articles, tutorials and plug&play packages we've found requires LIDAR or similar sensor, and we've failed to create such algorithm on our own. We’ve implemented a more rudimentary algorithm which is much, much worse, but at least it does work.

The main issue of this entire project was how niche topics it covers. No one has ever tried this, or anything similar, before, there’s barely anything to read about it on the internet - this is why the Bibliography is almost empty, there are no helpful resources to cite, we had to figure it out on our own. It was a very ambitious (or rather overly-ambitious) project, and we’re proud of what we’ve accomplished.

In the end we’re nearly proud of the results. It was a very complicated and ambitious project, with three separate programs, each done with its own framework and technology. We had to realize every step on our own without any articles or tutorial to lead us through it - all the way from connecting pins to microcontroler, through micro-ROS, ROS and Gazebo, to Raspberry PI programming - and we nearly succeeded.

9. Bibliography

[1] GitHub repository of the project: https://github.com/AzethMeron/AstableInvention

[2] GitHub repository of *iRobot Create 3* model: https://github.com/iRobotEducation/create3_sim

[3] GitHub repository of AWS: https://github.com/aws-robotics/aws-robomaker-small-house-world

[4] Mukherjee, Dibyendu & Saha, Ashirbani & Mendapara, Pankajkumar & Wu, Dan & Wu, Q. M. Jonathan. (2009). “A cost-effective probabilistic approach to localization and mapping”. 359-366. 10.13140/2.1.4304.4804.

[5] STM32F103xC/D/E datasheet https://www.st.com/resource/en/datasheet/stm32f103ve.pdf

[6] ROS 2 Galactic framework: https://docs.ros.org/en/galactic/index.html

[7] Micro-ROS framework: https://micro.ros.org

[8] ESP-IDF framework: https://docs.espressif.com/projects/esp-idf/en/v4.2/esp32/index.html

[9] OpenCV for python: https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html