CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's Thesis

Navigation for a Two-Handed Mobile Manipulator TIAGO++

Václav Kůla
Open Informatics, Artificial Intelligence and Computer Science

Květen 2022
Supervisor: RNDr. Miroslav Kulich, Ph.D.

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Kůla Václav** Personal ID number: **491883**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Navigation for a Two-Handed Mobile Manipulator TIAGO++

Bachelor’s thesis title in Czech:

Navigace pro dvouruký mobilní manipulátor TIAGo++

Guidelines:

1. Get acquainted with Tiago++ robot and tutorials for its usage in ROS.
2. Make the ROS navigation stack functioning with Tiago++ robot.
3. Get acquainted with YOLO – You only look once – a real time object detection system.
4. Train YOLO to detect elevator control buttons.
5. Implement the functionalities for operating an elevator.
6. Experimentally evaluate the implemented functionalities and discuss achieved results.

Bibliography / sources:

[1] htpps://wiki.ros.org/Robots/TIAGo/Tutorials
[2] K. Ehsani, H. Bagherinezhad, J. Redmon, R. Mottaghi and A. Farhadi, "Who Let the Dogs Out? Modeling Dog Behavior from Visual Data," 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, 2018, pp. 4051-4060, doi: 10.1109/CVPR.2018.00426.

Name and workplace of bachelor’s thesis supervisor:

RNDr. Miroslav Kulich, Ph.D. Intelligent and Mobile Robotics CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **17.01.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

RNDr. Miroslav Kulich, Ph.D. Supervisor's signature	prof. Ing. Tomáš Svoboda, Ph.D. Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. Date of assignment receipt	Student's signature

Acknowledgement / Declaration

This effort would not be possible without RNDr. Miroslav Kulich, Ph.D. He provided excellent guidance, lots of patience, and helpful advice throughout the whole project. Many thanks belong to Dr. Gaël Écorchard for helping with the TIAGo robot and 3D printing the part for the TIAGo robot. I would like to recognize Ing. Karel Košnar, Ph.D. and Ing. Viktor Kozák for their helpful suggestions. Lastly, I would like to acknowledge the unyielding support I received from my family over the three years of studying.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 20, 2022

Prohlašuji, že jsem předloženou práci vypracoval samostantně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 20. května 2022

Abstrakt / Abstract

Tato práce řeší problém ovládání výtahu robotem TIAGo++. Řešili jsme problém víceposchoďové navigace, použili jsme již naimplementované metody na navigaci TIAGo robota na jednom patře a vypracovali pro TIAGo robota způsob, jak používat výtah, aby se pohyboval mezi patry. Navigace na jednotlivých patrech je dobře známý problém, který již byl vyřešen mnohokrát. Autonomní ovládání výtahu není v robotice tak moc prozkoumaný problém, proto většina našeho usilí směřovala do jeho vyřešení. To se povedlo a TIAGo je schopný pohybovat se po jednom patře, použít výtah aby se přesunul do jiného patra a změnil mapu, podle které se naviguje na mapu správného patra.

Klíčová slova: TIAGo++, neuronová síť YOLO, Obsluha výtahu robotem

Překlad titulu: Navigace dvourukého pohyblivého manipulátoru TIAGO++

This thesis takes on a problem of elevator operation by the robot TIAGo++. We tackled the problem of multifloor navigation, using already implemented methods to navigate TIAGo on a single floor and developed a way for TIAGo to use an elevator to move between floors. Navigating individual floors is a very well–known problem solved many times already. Autonomous elevator operation is not as much explored problem in robotics, so most of our effort was put on solving elevator operation by TAIGo robot. This was a success and TIAGo is able to navigate itself on one floor, use an elevator to reach another and change the map it navigates by to the right floor.

Keywords: TIAGo++, YOLO neural network, Robot lift operation

/ Contents

Tables / Figures

Chapter 1
Introduction

In recent years, computers have evolved incredibly fast and allowed many connected fields to take a leap forward in progress. One of these fields is robotics. One of the biggest changes in the field is shifting the focus from highly specialized robots to multi-purpose robots that can work alongside humans in environments suited for humans. These robots often try to mimic human movements, as it is very intuitive to create new functionalities for robots if they can follow similar or the very same steps as humans. TIAGo++ among other robots is a wheeled two-handed multi-purpose manipulator. Its wheeled base makes navigating the robot easy, however, the robot is locked to operate on a single floor unless transported via some means, typically an elevator. Allowing robots to operate lifts autonomously gives them an ability to move freely inside buildings, adding more utility on top of what they are already capable of. Also, while operating an elevator, the robot needs to do some tasks easy for humans, but underexplored in TIAGo's domain, namely pressing buttons. This all together makes a task of developing autonomous lift operation worthwhile, as both the task as a whole and its subtasks will improve TIAGo's capabilities.
Aside from moving TIAGo between floors, one task that would allow it to move more freely in human environments is opening doors. While we do not tackle this problem, it is a topic that should be addressed in a short time and might use some parts of this work as a resource, mainly detection of important objects from TIAGo's camera feed, computing their real world coordinates and moving TIAGo's hands close to these objects.

1.1 Goal of this Thesis

This thesis tackles the problem of autonomous elevator operation by the robot TIAGo++. This task can be looked at as a procedure of several subtasks, each adding on top of TIAGo's already impressive arsenal of functionalities. The subtasks consist of moving to the elevator location, locating the button as precisely as possible, pressing it with one of TIAGo' s hands, entering automated door once it opens, doing the locate–and–press–button procedure inside the elevator and leaving it once the door opens again and moving on the floor TIAGo moved to. Another goal appeared during the development, which was to choose or create the right end-effector tool to press the button, as end–effectors provided with TIAGo are not particularly suited for the job.
This thesis tries to address several points. **Object detection** from TIAGo's camera feed is done using YOLO neural network and is followed by finding the detected object's position in the real world. We explore TIAGo's ability of **pressing buttons** and how it fares when accomplishing this task. We combine these two tasks along with autonomous TIAGo navigation to explore its ability of **operating an elevator** and **navigating on multiple floors.** After trying our solutions in simulation, we undergo thorough **testing on a real robot** and discuss the **results of testing.** These are the main

contributions of this thesis.
In the next chapter, we take a look at the robot and its parts. Next, we look into the approach we took in using an elevator with the TIAGo robot and go over implementation of the chosen approach. In the last chapters, we discuss results we achieved, the problems we faced along the way and after discussing possible solutions for these problems, we conclude the thesis and offer a few directions this thesis can be taken forward in.

Chapter 2
TIAGo++ robot

2.1 Problem description

The problem we are faced with is autonomous elevator operation by the TAIGo++ robot. Connected to this problem is moving TIAGo around on a floor before and after riding the elevator. Movement on individual floors is limited by doors and steps that TIAGo cannot overcome. Opening and moving through doors is a topic we will not be addressing in this thesis, we always opened any door that might prevent TIAGo's movement. to move TIAGo over the steps, we created a small ramp and navigated TIAGo on that ramp by hand. We again do not tackle the problem of TIAGo autonomously carrying the ramp to a place where it is needed and navigating it by itself. This can again be a topic for a different thesis.
The elevator we work with is one located in the CIIRC building in Prague. We can move from the first underground floor to the seventh floor above the ground. The elevator is called by pressing a button outside the elevator and its door is automatic, so we only need to make TIAGo attempt to enter the elevator once the door opens automatically. The elevator can be seen in Figure 2.1.

Figure 2.1. CIIRC elevator through TIAGo's camera.

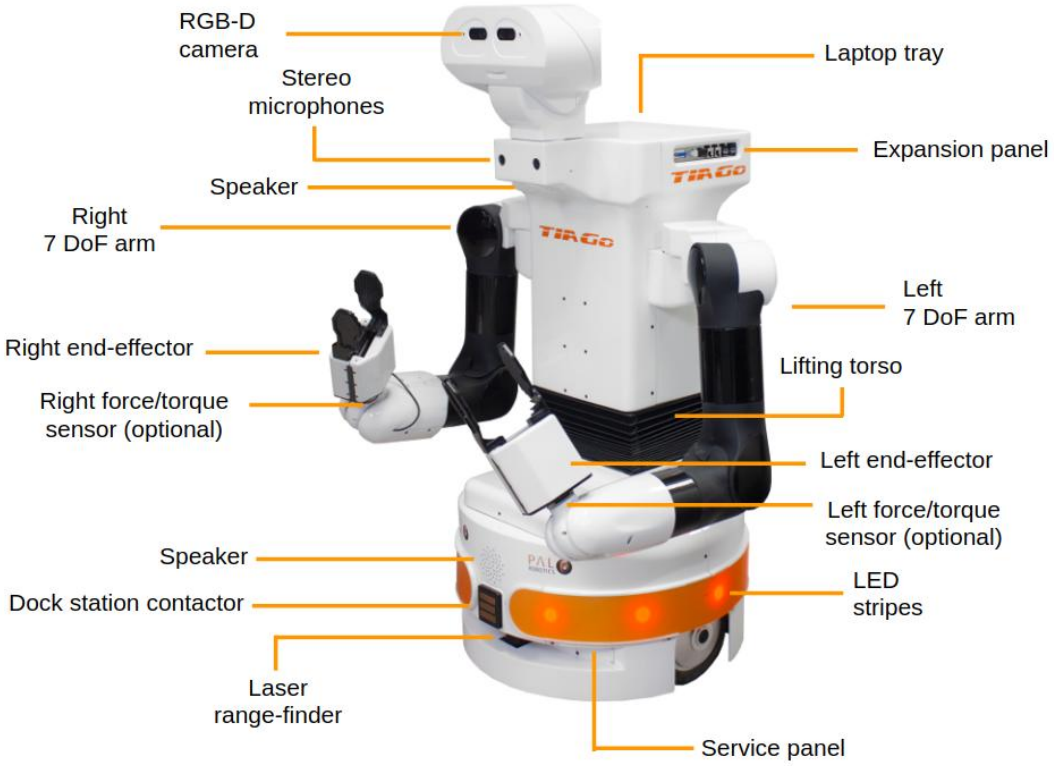

Figure 2.2. TIAGo++ robot overview [1].

2.2 TIAGo++ robot

Now we can have a quick overview of the TIAGo++ robot and its components we will be using [1].

The **mobile base of** TIAGo is a wheeled differential steering base equipped with a laser sensor and three rear sonars. This base is capable of working without the robot torso, that we will discuss later. This is advantageous for mapping, as the base is more nimble than the robot and also is less likely to collide to obstacles around.

TIAGo's **torso** is located atop the base and serves as a mounting point for all other parts of the robot, namely the head, the arms and also houses a microphone TIAGo can utilize. It has one joint, that can be used to lift the robot. The height of the robot can go as low as 110 cm and can go as high as 145 cm. The robot is shown in Figure 2.2 with its most important parts described.

TIAGo's **head** is located on top of the torso and houses an RGB-D Orbbec Astra S camera. The camera outputs 640x480 pixel RGB images at 30 frames per second and depth images of the same 640x480 resolution again at 30 frames per second. The depth sensor has range of 0.4 to 2 meters. We will be utilizing both the camera and the depth sensor, the camera for detection of buttons and the depth sensor to project the 2–dimensional camera image to the world coordinates. It utilizes a two degrees of freedom pan–tilt mechanism to point the head to different directions.

One of TIAGo's most important features are its **arms** that can be seen in Figure 2.3. These are seven degrees of freedom manipulators. That means, that there are seven joints in each arm that can be set to a value of choice independently. We will however not be using the ability to set these joint's configurations directly, instead, we will use

the MoveIt framework as well as the preprogrammed movements in TIAGo to control the arms. The premade movement we will be using is the home movement, that brings TIAGo's arm close to its body – this is needed so that TAIGo fits the elevator door after pressing the button to call the elevator. When using the MoveIt framefork, we will specify the coordinates of the button relative to TIAGo's base and MoveIt will use inverse kinematics to bring the arm to the desired postion.

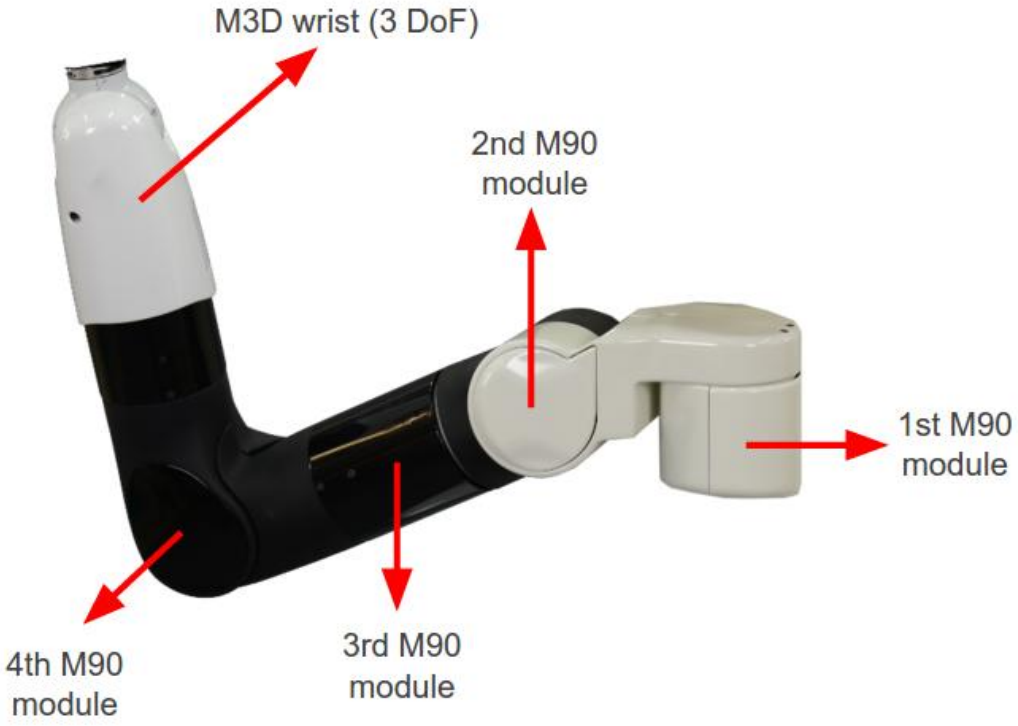

Figure 2.3. TIAGo's arm overview [1].

As a manipulator, TIAGo's hands are predominantly optimized for grabbing things rather than pressing buttons. The **end–effectors** provided for the robot are the Hey5 hand, in the Figure 2.4, a hand with five fingers controlled by 3 motors, one for the thumb, one for index finger and one for the rest of the fingers. This could have been used to press buttons with the index finger, if it was a bit sturdier, but when trying to press the button, the index finger moves to the side as it lacks the support to press a button. Alternatively, a metal gripper could be used, that is sturdy enough to press the button, but this gripper is far too wide at the end to hit buttons precisely. Solutions to this problem will be presented in later chapters.

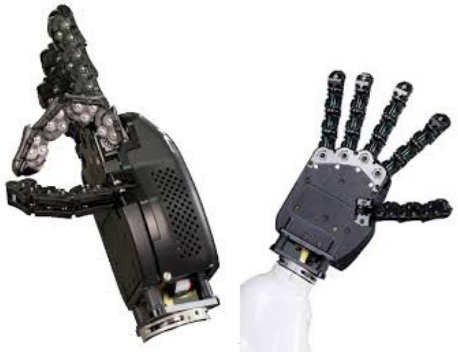

Figure 2.4. Hey5 end–effector overview [1].

Chapter 3
Approach

Lift operation is a task that can be divided into several subtasks. These subtasks are highly procedural, as each needs the previous task to finish before it can run. The subtasks we need to solve are in order:

- Move the robot to the elevator
- Locate the button from camera image using a neural network
- Get the world coordinates of the button from the detection in an image
- Press the button with one of the arms
- Enter the elevator and repeat the last three steps on a button inside
- Leave the elevator once it arrives
- Move around the floor we land on

3.1 Moving the robot to the elevator

We set our goal to be autonomous movement of TIAGo robot between different floors. That means we need to move TIAGo from a point somewhere on a floor close to an elevator, and once TIAGo moves onto a different floor, get it from the elevator to another place on said floor. Due to that we needed some ways of moving TIAGo. At first, we moved TIAGo around using a joystick provided with the robot. This joystick can be used to control the whole robot including all its joints. However, to make TIAGo's movement fully autonomous, the next step is to use ROS navigaiton stack to move the robot around using a map of the environment. PAL robotics provides software that could be used, allowing TIAGo to go to a defined point of interest or zone of interest [1], but this software is proprietory to PAL robotics and only provided for some robots and not for the TIAGo robot we work with. Because of that, we need to devise a way of using the navigation stack to move TIAGo to the elevator and get information when TIAGo arrives to the chosen location.

3.2 Locating the button

Once TIAGo arrives to a destination close to the elevator, we need to locate the button that will call the elevator. We will be using a neural network for that end, that is used to detect in an image from TIAGo's camera. The neural network we chose is the YOLO neural network. Neural networks are state of the art technologies in the image recognition field that are by a margin better for the task than any other known technology. YOLO, standing for You Only Look Once is a constantly developing neural network created with reliable fast (real time) image recognition in mind. YOLO can process up to 45 images a second and a faster version Fast YOLO can process up to overwhelming 155 images a second [2]. While we only need to evaluate a single picture at a time, YOLO has some features that make it better suited for this problem than other neural

networks. of course YOLO also has it problems, but these are mostly irrelevant for us. The benefits of YOLO include: [2]

- **Speed** the detections are very fast, so TIAGo can operate the elevator quite fast. In our case, one image takes 30 seconds to detect, this is however due to the development computer having no graphics cards, so we need to detect on a processor. With the fairly weak Intel Core i3-8145U CPU [3] this time is acceptable. with a powerful graphics card, we should be able to get times more than 500 times faster [4]. This would result in less than one second an image.
- **Little false positives** YOLO features less false positives than other neural networks. This can help us in a way, as with false positives, we need to select one of the detections and have no confirmation if that was the actual button. Not detecting a button is less of a problem as the buttons are always in the same positions relative to one another and we can try to deduce the position of one from its neighbours.
- **Position matters** Detections provided by YOLO are position dependant. YOLO looks at a picture as a whole, rather than using a sliding window as many of its contemporaries do. This approach means, that YOLO is aware of a position of an object in the image and can use this to detect. This is useful to us, as the buttons always form the same pattern and are usually in a very similar spot in the camera field of view.

However, there are also some disadvantages to YOLO, which luckily will not cause us much trouble [2].

- **Precision** the detections provided by YOLO might not be as precise as those provided by different neural networks. This proved to not be much of a problem, as it had very little effect on pressing the button.
- **Limited number of detections** When YOLO detects objects in an image, it puts a grid over the image and each grid box is responsible for all objects, that have their centre in that box. As this number has a top limit the number of detections in a cramped space is limited. This does not affect us at all as the buttons are fairly far apart.

We can see that the advantages of YOLO easily outwieght the disadvantage, at least for our goal.
We use Darknet framework that provides the YOLO network [4]. It provides a text interface to use YOLO for detections and with a bit of work, we can use it in ROS for our needs.

3.3 Pinhole camera model

The detections YOLO finds give us pixel coordinates in a picture from the camera. to press a button located in a picture, we need to be able to convert these coordinates to real world coordinates. to get this conversion done, we use the pinhole camera model. This model is a simplified camera model that assumes the camera has a single point for its aperture [5]. It also assumes that the camera has no lenses to focus the light, but we cannot ignore the lenses of the TIAGo robot, so we will take TIAGo's lenses into account.
The pinhole camera model can tell us how to view the relationship between a point in the real world and its coordinates in an image from a camera. It comes from a pinhole camera, a box with five dark sides and one translucent. A small hole is made into the dark side opposite of the translucent one. With a hole of only a single point, only

one light ray reflected from each point in the real world will make it through the hole, forming an image of the real world on the translucent side of the box [6]. This picture is however upside down, so to make it more convenient, we often flip the projection plane, which is the translucent side of the box, to front of the pinhole, as can be seen in Figure 3.1.

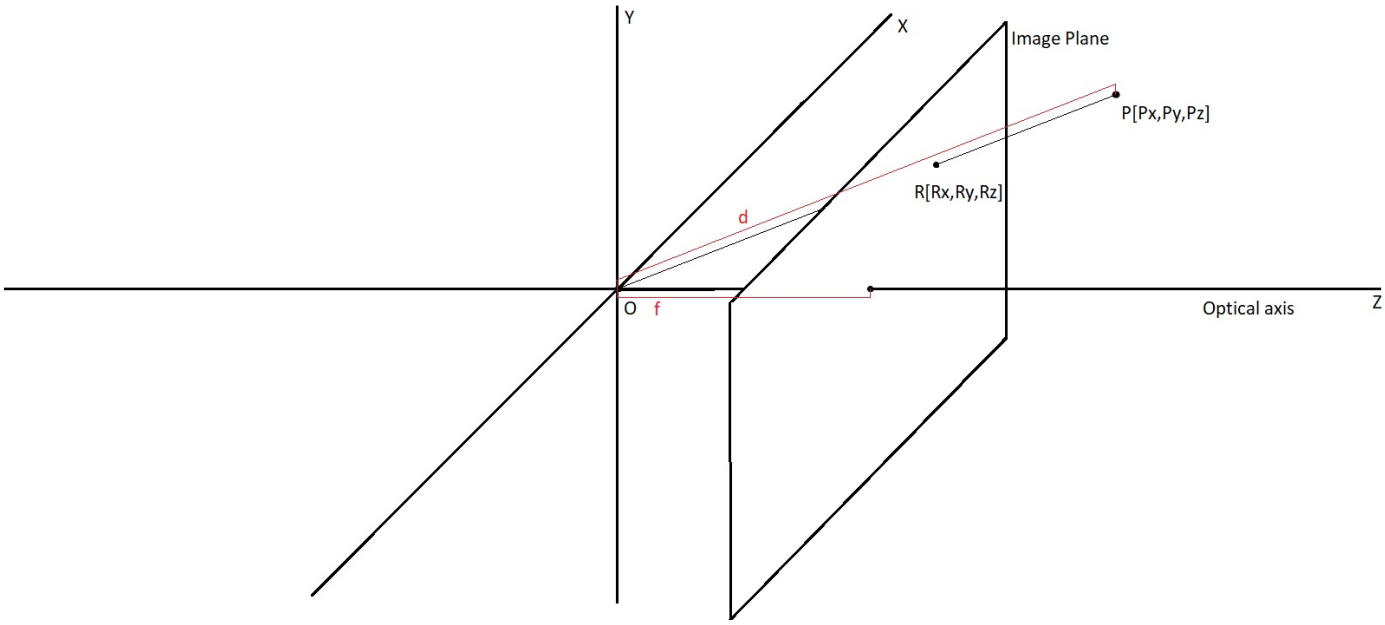

Figure 3.1. Pinhole camera model.

Of course this would not work in the real world, but it makes the model more suitable for mathemathics [6]. Now let us describe the image of the pinhole camera model. We assume a three axes coordinate system with axes X, Y and Z with origin O. O is the point where the pinhole is located. Z is the axis in the viewing direction of the camera and is also called the optical axis or the principal axis [5]. In distance *f*, which is the focal length of the camera, from the origin O is the image plane, a plane spanned by axis X and Y. It should be at negative coordinates along the Z axis, but as we said before, we will be moving it in front of the origin, so it is located at positive coordinates along axis Z. Image plane is the plane the real world is projected to. We assume a point P somewhere in the real world with coordinates P_x, P_y, and P_z. This point is projected to a point R with coordinates R_x, R_y, and R_z that is located somewhere on the image plane. Now based of similar triangles we get the following equations: $Px = \frac{R_x * P_z}{f}$, $Py = \frac{R_y * P_z}{f}$ [7]. We are still missing Pz though to be able to get the real world coordinates of the points. We can get this from depth d of the point P we get from TIAGo's depth sensor and in the following steps get the real world coordinates of the point P. First we compute $\frac{P_x}{P_z}$ and $\frac{P_x}{P_z}$ as $\frac{P_x}{P_z} = \frac{R_x}{f}$ and similarly $\frac{P_y}{P_z} = \frac{R_y}{f}$. Next we compute Pz as $\frac{d}{\sqrt{(1+(\frac{P_x}{P_z})^2+(\frac{P_y}{P_z})^2)}}$ [7]. With P_z computed, we can finally get P_x and P_y. This is how the pinhole camera model is used to compute real world coordinates of a point from TIAGo's camera coordinates combined with the point's depth.

3.4 Moving the hand with MoveIt

In order to press the button to call or move the elevator, we will hit it with one of TIAGo's hands. We use MoveIt framework to move the arms and hands [8]. We will use this widespread software to navigate TIAGo's arms to press the elevator buttons. MoveIt will compute kinematics for the motions we need TIAGo to perform. As TIAGo and many other robots are build to mimic human behaviour, the desired trajectory should be similar to actual human reaching for a button — moving the hand to a space close to the button, moving forward to press it and moving back once it is pressed. MoveIt can compute both forward and inverse kinematics. Forward kinematics utilize a set of equations to compute the position of the last link of the robot's hand chain or the end effector in the real world. They can be used to compute the position of any part of the chain, which means any joint, but the last one is usually the one we are interested in the most. Forward kinematics in MoveIt is computed recursively [9]. This is done by computing the position of the first link of the chain and propagating it to the next, solving position of each link until the last one is reached. With this approach, we can calculate the position of the end effector in the real world. We are however more interested in inverse kinematics, that work the opposite way. We want to be able to navigate TIAGo's hand to a position in world coordinates and need to compute the joint positions that allow it to reach this position. Reversing the forward kinematics equations is incredibly hard. When solved, inverse kinematics for a seven joint arm can have any number of solution from zero to infinity. This further complicates the computation of inverse kinematics. The best way to compute inverse kinematics is to estimate a close enough solution. This can be done numerically by estimating a solution and then searching for better solutions until a sufficiently good one is found. The numerical inverse kinematics solver MoveIt uses is the Jacobian inverse method. We can break down the method now [8]. Let x be a vector of joint position from $\mathbb{R}^m$ where m is the number of joint parameters, which is equal to the combined number of degrees of freedom of the joints we work with. Let then $p(x)\colon \mathbb{R}^m \to \mathbb{R}^3$ be a mapping of the robot's joint position vector to the real world coordinates of the last link. We can then assume that $p_1 = p(x_0 + \Delta x)$ where x_0 is the starting position of the joints. We than search for such a Δx that $|p(x_0 + \Delta x_{estimate} - p_1)|$ is minimal. We can estimate the Δx vector from $p(x_1) \approx p(x_0) + J_p(x_0)\Delta x$ where $J_p(x_0)$ is the Jacobian matrix at point x_0. We can estimate the entry at position i, k of the Jacobian as $\frac{\delta p_i}{\delta x_k} \approx \frac{p_i(x_{0,k}+h)-p_i(x_0)}{h}$ where h is a small positive value. to get to the Δx we now take the Moore–Penrose pseudoinverse [10] of the Jacobian and rearrange the equation to get $\Delta x \approx J_p^+(x_0)\Delta p$ where $\Delta p = p(x_0 + \Delta x) - p(x_0)$. A pseudoinverse of a matrix M denoted as M^+ is such a matrix that $MM^+M = M$, and $M^+MM^+ = M^+$ holds. It also has additional properties not interesting for us right now. A Moore–Penrose pseudoinverse exists for any matrix. When the matrix has linearly independent rows, the Moore–Penrose pseudoinverse is a right inversion, which means that $MM^+ = I$. When the matrix M has linearly independent columns, Moore–Penrose pseudoinverse is a left inversion, meaning $M^+M = I$. If both are true. is is equal to M^{-1}. I is identity or a matrix that has all its entries equal to 0 except for its left to right diagonal, which consists of ones.

The equation $\Delta x \approx J_p^+(x_0)\Delta p$ will result in a very rough estimate so we repeat the last step with the newly acquired Δx until the result is close enough to the desired solution [11]. This is much more demanding to computational power than finding an analytical solution and takes much longer. However, we do not need to derive the analytical

eqations that are very hard and tedious to find. These solutions can also be found for only a small systems with very limited number of degrees of freedom. The very high number of solutions MoveIt can find leads to choosing randomly from a very big and diverse set of solutions. This manifests itself when pressing the button. We get very diverse trajectories even though we try to set up TIAGo with very similar starting condition each time it presses a button. This also leads to some problems. MoveIt watches the robot's joints to not move faster than their limit. The speed of a joint is derived purely by the change of position of the joint in space over a small amount of time. If a joint moves too fast, the movement is abandoned by MoveIt. This can happen when a chain link is moving while also being moved by other joints. The combined movement can overshoot the allowed limit even though the joint itself is moving in accordance with the limits imposed. The planner has no way to battle this kind of problem sadly. However it works fine in all other regards.

3.5 Entering the elevetor

Entering the elevator, while seemingly a simple task, proved to be fairly hard. There is little time left once the elevator arrives to get in, so first we need to align TIAGo to the door right. This requires quite some precision, as TIAGo's base is 54 centimeters in diameter [1] and its shoulders are even wider while the door is fairly narrow, probably around 80 centimeters wide. The inside of the elevator is a bit wider, but not much, so both TIAGo's position and the angle to the door is very important. Because of this, we wanted to use the ROS navigation stack to navigate TIAGo into the elevator, but due to a bug this is not possible. For some reason, TIAGo always maps a wall across the elevator door, even when the door is open during mapping. This prevents the navigation stack from navigating TIAGo to enter the elevator. We can however use the navigation stack to align TIAGo to the door, even though this is not as precise as we would like, the angle in which TIAGo faces the door can vary. Once TIAGo is aligned properly, it waits until its front laser sensor measures distance longer than that of a closed door and is then used to control TIAGo so that it stops in a required distance form the back elevator wall. The time it takes the elevator to arrive varies and if it is only a floor away or already on the floor, TIAGo will likely not be ready in time. This cannot really be solved by the robot, it needs the time it takes to finish its movement. It might cause TIAGo to underperform, but the only way to solve this problem is to move to a more robot friendly environment where the time for the robot to board the elevator is longer.

3.6 Movement inside the elevator

Once inside the elevator, TIAGo turns 90 degrees, this is done by rotating the base with set velocity for a set time. This might fail if the wheels clog or if the surface is extremely slippery. However in this thesis we can suppose that TIAGo's wheels do work as they should and that the floor will always be the same, so our approach is sufficient. After turning, TIAGo will be located too close to the button it needs to press and the depth sensor of its camera will not be able to measure the depth we need to calculate position of the buttons. We solve this by moving TIAGo backwards until it is close to the wall opposite of the panel with the buttons. Since minimal range of the depth sensor is 0.4 meters, we need to move TIAGo back after boarding the elevator. After boarding the elevator and turning to the elevator buttons, TIAGo's head, which holds

the camera and the depth sensor, will be about 20 centimeters away from the button panel, too close to measure depth. We use one of the 3 mobile base mounted sonar sensors so that we get as close as possible without colliding with the wall. We move TIAGo so that its back is 15 centimeters away from the wall opposite to the button panel, after which the head is about 0.5 meters from the button panel. This is far enough for it to provide information about depth. At this position TIAGo will be far enough to use the depth sensor and can press the buttons safely. After pressing the button, we once again utilize the laser sensor controlled movement to send TIAGo forward to roughly the middle of the elevator, turn 90 degrees again and wait for the elevator door to open and once it opens, TIAGo leaves and continues to move until it is in a set distance to the wall opposite of the elevator.

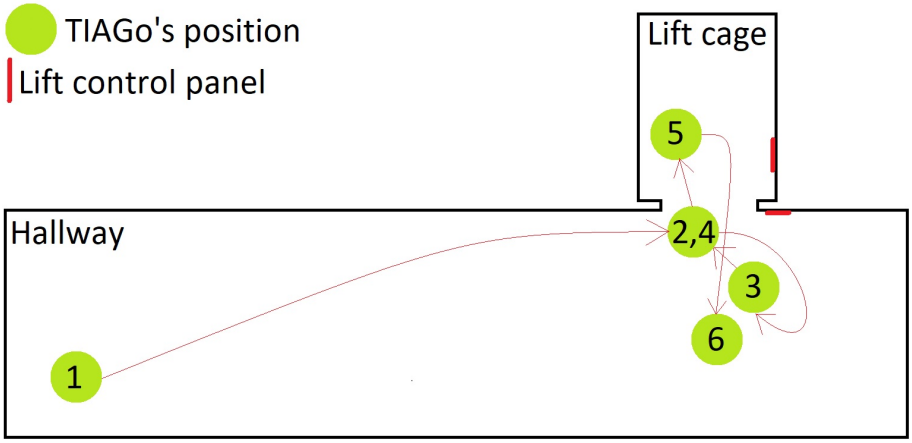

Figure 3.2. Diagram of TIAGo's movement.

In Figure 3.2 we can see the path TIAGo takes when moving to the elevator. Position 1 is TIAGo's starting position that can be anywere in the hallway. We use ROS navigation stack to move TIAGo to position 3 going through position 2 on the way. The middle point of position 2 is there so that TIAGo moves to position 3 from a close distance therefore moving slowly. This allows TIAGo to better angle itself to the control panel, if it approaches the control panel with higher speed the angle varies much more leading to TIAGo missing the elevator button more often. When TIAGo arrives to position 3, it presses the button on the control panel outside the elevator. Then it moves to position 4, which is the same as position 2. There it waits for the elevator door to open. Once the door opens TIAGo boards the elevator and moves to position 5. There it first rotates to face the control panel inside the elevator and a bit of forward and backward movement follows. First, TIAGo moves back to be able to press the button of the selected floor. It needs to move back so that the depth sensor we use is at least 0.4 meters from the control panel — 0.4 meters is its minimum range. When TIAGo is far enough to press the elevator button, it does so. After the button is pressed, TIAGo moves forward to position 5 again. There it rotates to face the elevator door again and waits for the door to open when the lift arrives to the selected floor. Once the door opens, TIAGo moves forward to position 6, which is 1.3 meters from the wall opposite of the elevator. There it changes map that is uses, sends an estimate of the position it is located at so that mapping can continue and can be navigated on the floor it arrived to.
When moving TIAGo's arms in the elevator, we must keep in mind the cramped environment of the elevator. When reaching for the button, this never imposed a problem,

but when we want to bring the arm back close to TIAGo's body, we cannot use the predefined home movement, because its trajectory would usually collide with both the door on the right side of TIAGo and the wall behind it. However, we can use MoveIt to try to get the hand to the same coordinates as those after running the home movement, which at the cost of reliability and time, provides much more modest and more straight trajectory. This trajectory does not collide with TIAGo's surroundings so it is good enough for us.

3.7 the right end effector choice

The last link of the robotic arms is called the end effector. As TIAGo is a robotic manipulator, its end effector choices are designed to grab and move objects rather than press buttons. This is especially true for the Hey5 robotic hand, which can lift objects similarily to human hands, but pressing buttons is next to impossible, because the robotic finger is not firm enough when pointing out when the hand is open. Some testing was done with pressing the button by knuckles of a closed fist, but this is rather unwieldy and lacks precision as well as it is unclear what part of the hand is the point that the MoveIt framework considers for moving the hand to a world coordinate. Robotic gripper, the other choice for TIAGo's end effector is better suited to press buttons, as it is firm and will not move to sides, however it has disadvantages of its own. The very outer part of the gripper is rather wide, so very high precision in locating the button is required. Also, it cannot be used for buttons which do not span more that is the width of the gripper, because it might get stopped by the button's sourrounding wall or frame and not press it at all. Another approach would be to try to press the button with for example a stick held in the robotic hand. This however has another major disadvantage - the stick would not be considered by MoveIt and another a calculation would have to be done to account for the longer hand. This might fail though, because position of TIAGo's hand on the stick would matter a lot. Last possibility is to 3D print a contraption, that will not allow the finger to move when hitting the button. The weakest point is the knuckle of the pointing finger. so a simple tube with overreach into TIAGo's palm should be enough, and it will bring great precision combined with a finger sturdy enough to press the button to the table. We use the solution of 3D printing a construction we put onto the finger to prevent it from bending. We need to prevent two types of the finger moving away from the button. One is the finger's knucklebones moving away. This is prevented completely by the tube that surrounds the finger. It cannot move at all while the tube surrounds it. This is important to keep in mind when turning the robot off, as the tube needs to always be dismounted before the robot is turned on again. TIAGo undergoes a procedure that calibrates the torgue sensor of the fingers. This is because the sensor might provide worse information over the robot's active time and needs to be periodically reconfigured. If the tube is still on during the configuration procedure, we risk damaging the motors of the finger by overstressing them and we are also left with a completely useless configuration that needs to be ran again in order to use the finger.
The other way the finger can move away is the knuckle that links the index finger to the palm of the robotic hand. This link is very weak and cannot be fully embraced by the tube as it has a very complicated shape. Due to this we only prevent the finger from moving the side from away from the hand, which is the most susceptible to allow the finger unwanted movement. This is also the only direction where the palm does not go outwards compared to the finger but inwards. In other words, the finger

overreaches the palm there. We overreach our printed tube in this direction so that if the finger tries to bend in this way, the overreaching part can press into the palm and prevent movement. In other drections, the side of the tube is thick enough to press into the overreaching palm and prevent movement in those directions. This part can be viewed in Figure 5.1. It is also discussed in the Results chapter a bit.

Chapter 4
Implementation

4.1 ROS

TIAGo is a robot running the Robot Operating System also called ROS. ROS is an open-source middleware that can be used to control robots. Main focus of ROS is to provide a lightweight solution for programming robots, which requires it to provide communication tools for different components of robots. Its philosophy is to give programmers freedom by not limiting them rather then providing many possibly unnecessary tools. It also aims to be compatible with libraries that are not develeped for ROS [12].
ROS is used to control robots so it shares some features with control systems, mainly easy access to creating processes and enabling these processes to communicate between each other. Processes are called nodes in ROS. Nodes can be distributed among several computers, provided each node can reach and register itself by the ROS Master node. Master allows nodes to communicate between each other. Communication is enabled over asynchronous channels called topics. Topics follow a peer–to–peer many–to–many model. A subscribing node contacts the master to receive information needed to connect to a publisher. Then subscriber contacts the publisher and publisher establishes connection over which it sends data to the subscriber node. This can be seen at the image Figure 4.1.

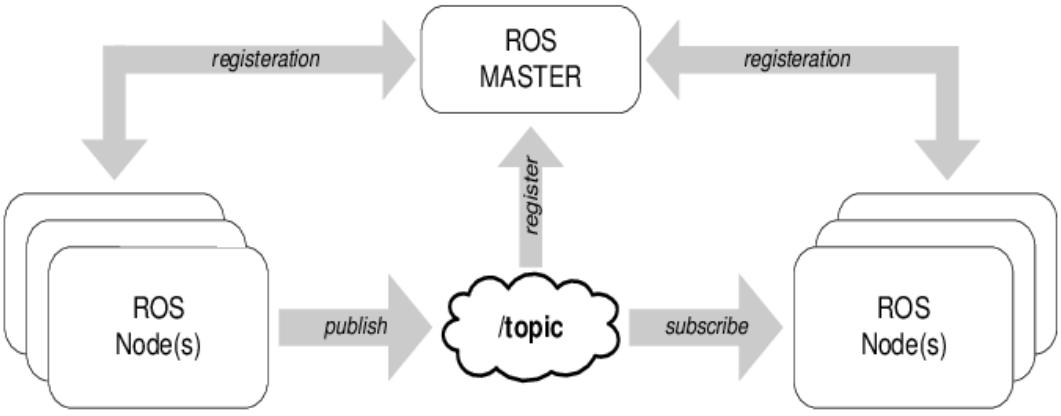

Figure 4.1. ROS nodes communication [13].

The peer-to-peer many–to–many model is not always perfect for some nodes to communicate. For this reason, there are other ways to communicate, namely services and actions. Both of these communication methods utilize topics to transfer messages internaly. Services are used for quick actions that take little time, due to them not being able to be preempted. They accept a request and send back a response. Actions accept a goal and then return feedback on how they are working towards the goal until it is finished, when they report this as a result. Actions can be preempted and contrary to services do not block. Actions are usually used for tasks that are expected to last

a longer time.
Messages are sent over topics. These messages first need to be defined, by using standard message types or nesting different message type. Messages can be nested extensively and it often is the case in ROS. Each topic only allows use of one type of message.

4.2 Controlling the Robot

In this thesis TIAGo is required to fulfill several different tasks so several different control mechanisms are used. These tasks are connected either to pressing elevator buttons, when we need to control TIAGo's arms and torso or to moving TIAGo on a floor or in and out of the elevator, when we need to control TIAGo's mobile base. Both of these motions are mediated by ROS as ROS is used to control the TIAGo robot.

4.2.1 Moving TIAGo's arms

To press the elevator buttons we need to control TIAGo's arms and hands. We can also take advantage of engaging TIAGo's torso in the movements, as it offers a wider range of trajectories. Once inside the elevator, we even need to lower TIAGo's body so that the buttons are in the field of view of TIAGo's camera.
TIAGo's arms, hands and body are controlled mainly by MoveIt. MoveIt is a framework that is often used with ROS [14]. We use it to plan movements of the robot based on desired joint coordination or end effector position in space. MoveIt utilizes kinematics to find a plan for a joint group and these plans can later be executed.
Joint groups are named sets of joints that are provided in the robot model to MoveIt. Joint groups we utilize are `arm_right_torso`, `torso` and `hand_right`. We always find TIAGo left to the buttons, so its right arm is better used to reach the buttons. We aim to set TIAGo in front of the button with the button being about 70 centimeters far and about 10 to 12 centimeters to the right of TIAGo's X axis. X is the horizontal axis in the direction TIAGo faces. This gives us best results, as the button is in TIAGo's field of view while allowing the right hand we use to reach it easily. This position works well, as it stays far away from most obstacles around buttons, the railing outside the elevator that is near and the door of the elevator when TIAGo is inside. This setup can be seen in Figure 4.2.

Since we use C++ to write the code, the best way to control groups of joints is to create an instance of the `moveit::planning_interface::MoveGroupInterface` [14] class. This class takes the name of the joint group we want to control in its constructor and can then plan among others plans for joint groups to move to a specified joint coordination or to move an end effector to a cartesian space or even a trajectory through a series of points in cartesian space.
Above this class we created a thin overlay class called ArmController. This class holds members of the MoveGroupInterface class along some other constants and provides some movement options for the hand. Most importantly this class holds a method that moves TIAGo's hand to press a button, watching the force exerted on its wrist and once the force exceeds a limit, it stops the hand movement and brings it back. Other than that it can also set up TIAGo's hand to point its index finger forward to prepare to press the button, set TIAGo's torso to a required height, which is required inside the elevator to move the elevator buttons into TIAGo's view range and a pseudo home movement, that brings its right hand close to TIAGo's body. More on why is that needed can be found in the next paragraph.
Another way we can control TIAGo's arms are predefined movements provided by PAL

Figure 4.2. TIAGo's position when pressing button.

robotics [1]. These are called by sending a goal to an action server `/play_motion`. The movement we are interested in is the home motion, that brings TIAGo's arms close to its body. We use this motion after pressing the button to call the elevator, so that TIAGo can move into the elevator with its arms compactly folded next to its body. We wanted to use this motion inside the elevator too, but there is not enough space for TIAGo to finish this movement. That is why we have the pseudo home movement in the ArmController class, that brings TIAGo's arm to the same joint configuration as the home motion, but takes way less space in the process.
Aside from TIAGo's arms, we also need to control its head, or rather stop its controller PAL head manager. This controller moves the head while the robot is stationary and when moving, makes the head face the direction of movement. This is detrimental to us as we want the head to be facing forward without any tilt or rotation. If the head is not tilted or rotated, we have much easier work when transforming coordinates from the image from TIAGo's camera to world coordinates. PAL manager can be stopped in TIAGo's web commander once we connect to the robot. When operating the elevator, we expect the PAL head manager to be stopped and the head joints to be both set to 0.

4.3 Moving TIAGo's mobile base

Second important part of TIAGo's movements is controlling the mobile base of TIAGo. We need to control it to move TIAGo on a floor it is, get it close to the elevator, move

it onboard the elevator, leave the elevator and move it on the floor TIAGo lands on. This base is controlled by sending velocity messages to the topic `/mobile_base_controller/cmd_vel`. This can be done in several ways. When controlling TIAGo by joystick, messages get published automatically once the joystick gets priority. Even when TIAGo is not moved around using the joystick, as long as it maintains priority messages with 0 speed will be published periodically, preventing any movement caused by different means than the joystick. We used the joystick to move TIAGo through doors and on the ramp to prevent it from crashing into anything.
Another way to move TIAGo around is to use ROS navigation stack. ROS navigation stack is implemented on the TIAGo robot and can be used to move TIAGo in an already created map, to move TIAGo in a map as it is being created and to create maps. The ability to create and save map and later use it to navigate TIAGo is crucial for us. With this we can move TIAGo on a floor it is, get it close to the elevator we want to use and navigate it on a different floor once TIAGo arrives there. We can see what nodes navigation stack requires and provides in the Figure 4.3.

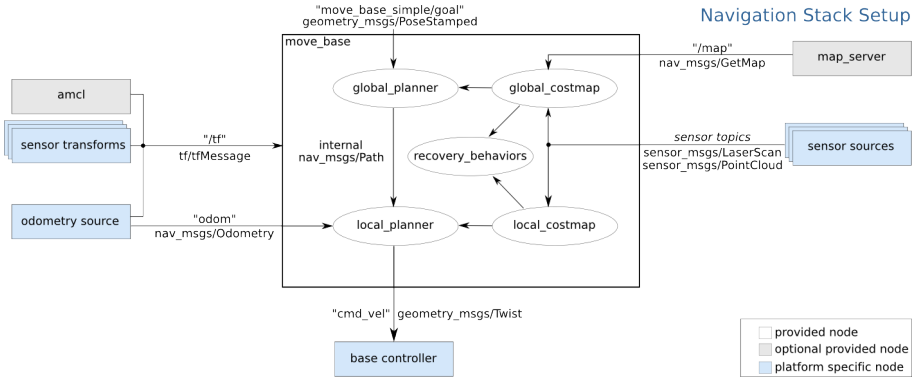

Figure 4.3. Overview of the nodes navigation stack provides and requires [12].

Both the `amcl` and `map_server` nodes are implemented on TIAGo robot. Alongside the navigation stack, PAL robotics provides a `/pal_map_manager` ROS service that we can use to change the current map. This is also very important, because we need to change map once we arrive to another floor.
Mapping was also done using the navigation stack, which follows the Simultaneous Localization and Mapping or SLAM. This allows us to explore an area by driving TIAGo in this area in mapping mode that creates a map and then use this map once it is finished. This is exactly how we mapped the area TIAGo moves in, we navigated it using the joystick controller and created a map for later use. When mapping TIAGo's position in the map is updated using odometry and corrected using the Augumented Monte Carlo Localization algorithm, or AMCL. AMCL is also used when TIAGo is navigating a map. The map is an occupancy grid containing data about the environment. It only contains positions of obstacles seen by TIAGo's laser sensor. This sensor is located about 10 centimeters off the ground, so objects like tables and chairs might cause trouble. TIAGo uses its camera to prevent collision with such objects. One of the maps created can be seen in Figure 4.4. In this figure is the map we use to navigate TIAGo in the hallway to get to the elevator.

In Figure 4.5 we can see the full lab mapped. This map features the whole lab and the elevator hallway and is the map we used before the small map was created. The small map is better for us as finding TIAGo's location in this map is faster and we

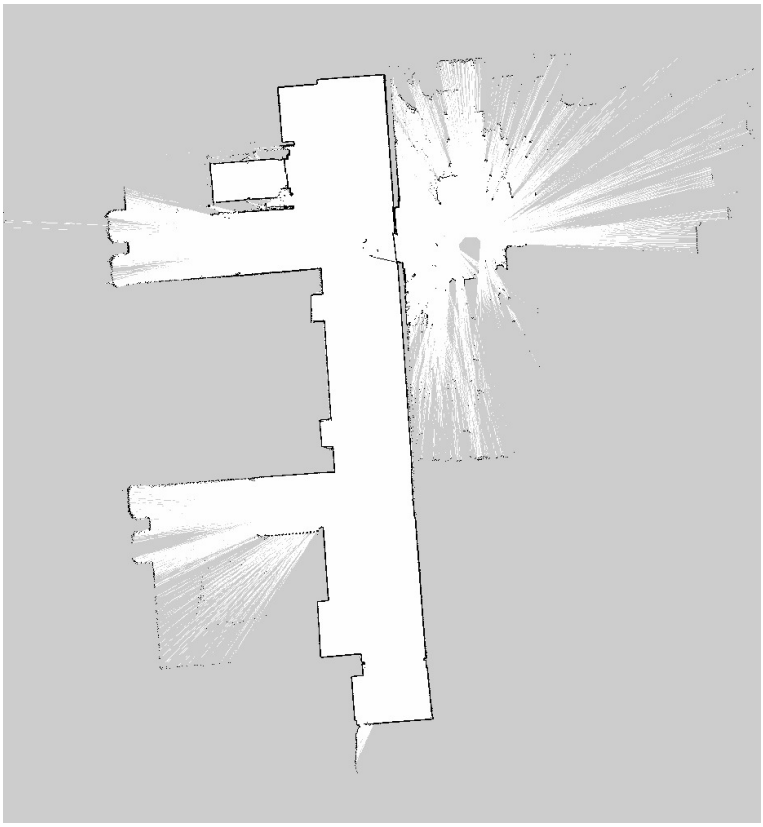

Figure 4.4. Map for TIAGo movement.

do not expect TIAGo to navigate into the lab through closed door and over a step it cannot ride over.

A problem when changing a map is that TIAGo might not know where it is located in the new map. Because of that, we have to send a hint to the `amcl` node by publishing a space close to its position after leaving the elevator on the topic `initialpose`, that can initialize of reinitialize TIAGo's pose in a map.
The process of creating maps is simple and fast, we only need to move TIAGo through the location we want to map while navigation stack's mapping is running and we are good to go. The mapping is done in real time and TIAGo can move at the top speed allowed by the joystick most of the time. The only thing we need to make sure of is that it does not leave blank spaces in the map as these would confuse TIAGo while localizing later. Taking the maps of all 7 floors TIAGo can go to would only take less than an hour. Much more tedious task is setting up the robot's positions to press the elevator buttons and to enter the elevator. We need to find a suitable position for TIAGo to press the button and to enter the elevator first. This was done by navigating TIAGo and testing its performance from the position in question. If the position is suitable, we need to read TIAGo's position from the AMCL node `/amcl_pose` topic and save it as the position TIAGo should have for the task we are currently preparing. For this to work we need to first make sure that the robot is localized properly. If it is not, we cannot use the pose provided as we would introduce unnecessary loss of precision to the procedure. AMCL often gets lost and thinks TIAGo is a bit off its actual position. The correction happens while TIAGo is moving so driving it close to a corner usually helps AMCL to again localize TIAGo precisely, but this costs us the position we worked hard to set up. Due to this, finding a suitable position for each action is

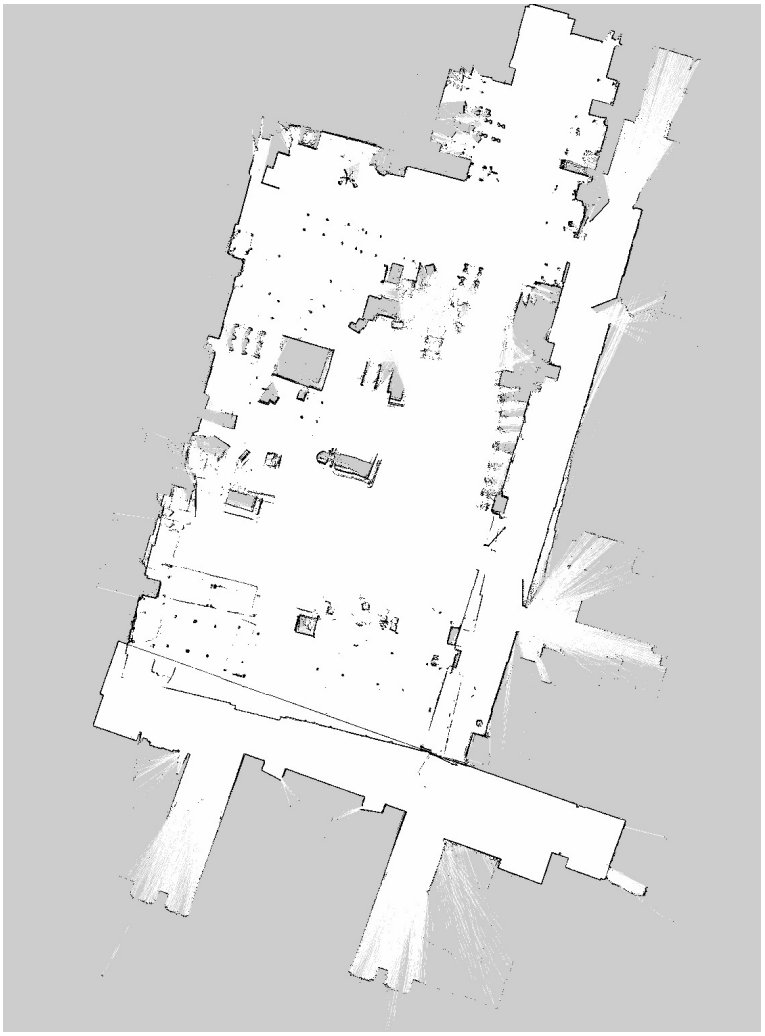

Figure 4.5. Map of CIIRC computer lab.

a tedious task that has to be done for each map we introduce to TIAGo. However only finding a suitable position is not enough. We also need to make sure that the navigation stack can set TIAGo up in a good enough way. Navigation stack often leaves TIAGo facing a slightly different direction than we specified. This introduces a lot of problems, as this might leave TIAGo without a clear view of the button we need to find which would lead to failure in detecting the button or worse, TIAGo can be set up facing the door of the elevator badly and colliding to the doorframe when trying to board the elevator. This usually happens when TIAGo moves a long way prior to reaching its position. This allows it to accumulate speed and rotate fast. When TIAGo is rotating fast to face the right direction, it almost always stops rotating prior to reaching the set coordinates. to prevent this, we can first navigate TIAGo to another position close to the one we actually want. TIAGo will then always move only a small distance to the position we want, moving slowly and precisely. This however requires even more testing with all its pitfalls. This makes using the same map for more than one floor very benefitial if possible. Sadly, in the CIIRC building only the floors two, three, and four have the same hallway with the elevator. The usage of navigation stack is provided in a class called `NavigationManager`. We created this class to have a simple environment we can use to send TIAGo to different places in different maps, swap maps easily

and generally have an easy time using the functionalities provided by navigation stack and PAL robotics.
In order to navigate TIAGo by the navigation stack, we first provide a map to it through the map server. After we provide it with a goal to reach the `global_planner` finds a path to the goal in the whole map. There are many planning algorithms to choose from to find a path to the global goal. The ROS navigation stack has both the A* algorithm and Dijsktra's algorithm implemented. Even though A* should be faster in theory, Dijsktra is the algorithm used, as A* only outperforms it on maps where there are not many obstacles. `Local_planner` then uses a sliding window approach to devise a short term plan based on information from sensors to avoid obstacles. This plan is found by simulating various paths and scoring them based on if they lead to collision, how close they come to obstacles, their speed and also their proximity to the short term goal, which is either the long term goal or the place where the path to global goal leaves the sliding window of the `local_planner`. The best scored simulation is then used to prepare the velocity messages to send to the mobile base which causes the robot to move.
When moving, TIAGo uses the provided map that only knows three values for each cell in the grid of the map. The values are taken from `map.png` file, which is a pixel map of the environment. Black pixels mean the space is occupied by an obstacle and white pixels mean the space is free. The map is inverted for the use of navigation stack, giving black pixels the highest value and white pixels the lowest value. Then a `free_threshold` and `occupied_threshold` are set artificaally. The navigation stack then acquires information based on these thresholds, `pixel_value` $<$ `free_threshold` means the space is free, `pixel_value` $<$ `occupied_threshold` means there is an obstacle and any value `free_threshold` $<$ `pixel_value` $<$ `occupied_threshold` is labeled as unknown. However, when navigating in the map, TIAGo uses an inflated map with more classes of cells. This inflation can only be done with the robot model so it is only done when the map is used for maximum portability of the maps. In this way, maps created by one robot can still be used to navigate another robot. The inflation introduces the following classes that cells can have: lethal, inscribed, possibly circumscribed. free space, and unknown. Each of these classes takes up an exclusive range of values that the cells can attain based on the inflation function. The inflation function can be user defined and is a function of distance to the closest obstacle. Based on distance, each cell falls into one category. Lethal are the cell that actually are obstacles. If the centre of the robot happened to be in a lethal cell, a severe collision surely had to happen. Inscribed cells are those that are less than a radius of the robot from the closest obstacle. If the radius of the robot reaches an inscribed cell, collision is again inevitable. Possibly circumscribed cells are those that are close to an obstacle but not too close. If the robot finds its centre in one of these cells, collision is possible but depends on the orientation of the robot. All of these classes of cells are dangerous areas and the robot should strive to avoid finding itself in these. TIAGo behaves exactly in this way when navigated by the navigation stack. The last two classes are freespace and unknown. Free space is far enough from the closest obstacle that the robot is not colliding with anything while its centre occupies any of the free cells. Unknown cells are not mapped and the robot should avoid them, but this is only due to the fact that we do not know what lies there. We always aimed to not have any unknown space in the maps we created.
Lastly, when entering the elevator, we cannot use the navigation stack due to possibly a bug the robot always mapped the elevator door as an obstacle. This means

that the navigation stack will not be able to plan its way into the elevator, even when the laser sensor provides information about space in front of TIAGo. We had to create our own controller to get TIAGo into the elevator. First step is to align TIAGo to the door using the navigation stack. Once TIAGo is facing the door under the correct angle, we use the laser sensor to measure distance in front of TIAGo. Once this distance grows above a certain number, TIAGo moves forward until it reaches a certain distance to the back elevator wall. How the simple controller works can be seen in Figure 4.6.

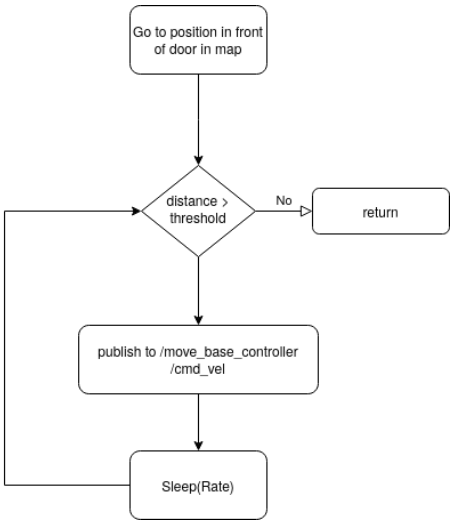

Figure 4.6. Algorithm for moving TIAGo with laser sensor.

After TIAGo finds itself in the elevator, it rotates 90 degrees to face the panel with buttons inside the elevator. It is however too close to this panel to press the buttons, its arm would have trouble fitting in the space in front of TIAGo and mainly the depth sensor is closer to the panel than its minimum range of 0.4 meters. to fix this problem, we move backwards while reading data from TIAGo's base's sonar sensors. There are three sensors publishing on the same topic so we need to filter out the sensors that are of no use to us. We make TIAGo get very close to the wall behind it so that it can scan the panel in front of it.
Both the laser sensor based movement and the sonar sensor based movement work very similarly, it takes information from the sensor published via a topic, publishes velocity messages for the base to move and once a certain threshold is met, it stops publishing and the robot stops subsequently. This sometimes leads to lack of precision as the robot stops a tiny while after the messages stop to be published. If we wanted to improve the performance in precision, a PID controller can be used to hit the distance more precisely.

4.4 YOLO Darknet framework

We will use neural network called YOLO to detect elevator buttons. Neural networks are state of the art technology in image detection and using them to detect objects in images is the most used approach in recent years. Neural networks used to be too slow to be used for tasks like detecting buttons to press but in recent years became marginally faster while keeping their staggering precision that can reach over 90 % of objects in a picture detect right. YOLO is a CNN or convolutional neural network,

which partly enables its incredible speed, with the other main factor being the approach of YOLO to finding bounding boxes for images.
Neural networks follow a structure of an input layer, a hidden layer, and an output layer. Each layer consists of neurons connected to neurons of the previous layer. Each neuron can be viewed as a function f(x) where x is a vector of inputs from connected neurons and its output is the output of the neuron. Input layer is a layer that takes an input we want the neural network to work with, an image in our case. Hidden layer is a black box of several layers consisting of neurons connected one after another that finds features of objects in the image. Lastly, the output layer is a layer that is used to output results of the neural network, in our case we need to be able to find class, position and size of an object in the picture. This architecture can be seen in image 4.7.

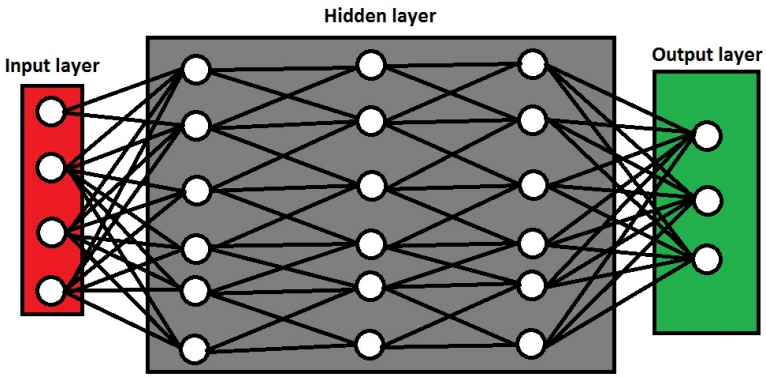

Figure 4.7. Architecture of neural networks.

When neural networks were first used, each neuron was connected to each neuron in the previous layer, so called fully connected layers. This however limited the use of neural networks greatly, as the number of connections between each layer was growing quadratically with size of the input. This means that for a 256x256 image we would have 4294967296 connections between any two layers. Since images are very big by themselves, having hundreds of thousands of pixels if small or even more with high resolution pictures, this approach was impossible to use. This is solved by using convolutional neural networks. Convolutional layer sees the previous layer through a convolutional filter, limiting the number of pixels each neuron sees. This introduces the assumption that pixels are more effected by pixels close to them rather than by pixels far away. This assumption holds very well in the real world. Using convolutional layers greatly limits the number of connections in each layer, making the processing of neural networks much faster. Also, each convolutional layer usually uses some convolutional kernel. These kernels give weights to neuron outputs of the previous layer. These kernels help to get the most important abstract features of objects that allow for great detection. How convolutional kernels work can be seen in image 4.8.

These kernels can have various effects, such as finding edges, finding only vertical edges or only horizontal edges and many more. Each layer needs to end with an activation function. Some widely used activation functions are linear, which is usually used before the output layer, Rectified Linear Unit or ReLU and leaky ReLU. The activation

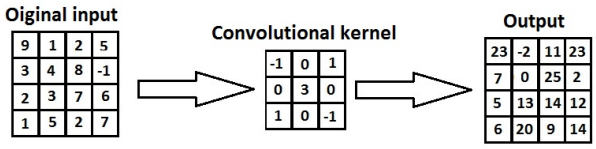

Figure 4.8. Convolutional kernel of neural networks.

makes it possible to activate some convolutional kernels more than others, which is used during training, when the network learns the discerning features of objects. One last important variable for a convolutional layer is its stride, or the step the convolutional kernel takes. The stride usually is not bigger than the size of the kernel. Stride makes the input smaller, losing some information but allowing more layers in the process or speeding the computation up. The last layer after many convolutional layers usually is a fully connected layer, a layer where each neuron is connected to each previous neuron. The purpose of this layer is to take data from the last convolutional layer and output them in a way that can be interpreted by the user. In YOLO's case, output of the fully connected layer is a tensor that contains a size and position of a box, a probability of an object being inside this box and a conditional probability of each class being in this box given an object is present. This can be interpreted very easily.
While the introduction of convoluted neural networks made neural networks much faster, YOLO takes it a step further to become a real time detecting neural network. This has to do with how the bounding boxes are created. While typical neural networks used to image detection use a rolling window approach, YOLO uses its own faster approach. The rolling window is fairly simple, we have a window that we move with a small step over the image and after each move, we predict the probability of an object being in this specific window. This however requires many iterations of using the nerual network, making this approach quite slow. YOLO's ambition is to detect objects in the whole image in a single run of the neural network. This is done by first putting a grid over the image we want to detect. Each square in this grid predicts a number of bounding boxes, that we can choose. Less bounding boxes lead to less detections if many objects are close to one another while increasing the speed of detection, more allow for more objects of the same class to be detected close to each other but slow down the detection time of each image we run YOLO on. Each grid also predicts the class of the bounding boxes it predicts. Once the bounding boxes are predicted, the output layer returns the results in a form of a tensor of SxSxB*(C + 5). SxS is the size of the grid we put over the image at the begining as each grid gives a prediction. The B*(C + 5) parts comes from B bounding boxes predicted by each cell, each containing the probability of an object being in this box, the width, height, x, and y coordinates of the centre of the box and conditional probability for each class of C classes that class c is present given an object is in this box. In this way, the whole

image is predicted in a single run of the YOLO neural network, making it incredibly fast. This has its own upsides and downsides discussed later in this chapter.
We use YOLO darknet framework [4] for image detection. Darknet is an opensource framework written in C and supporting CUDA for GPU computation that implements YOLO. As we used C++ to program TIAGo, only minor changes had to be done to the YOLO framework. Some things often used in C go against the C++ standards, mainly work with strings when C++ prefers working with string that are not present in C. C uses char* instead of strings and C++ views this as hazardous behaviour. This leads to some warnings being shown during compilation. While we can take measures to suppress these warnings, we can also be sure that the char* handling is not flawed in darknet and use the program even with a few warnings concerning this breaking of C++ standards.
The Darknet framework provides most of the tools necessary for the detection we need. For detection, we need several files that YOLO uses to set up some of its variables. These files include:

- **.cfg file** This file sets up the layers of YOLO. This is provided from darknet [4] but needs to be changed based on the number of classes (namely the YOLO layer's setting needs to be changed). This file contains information for darknet so that it can set up the layers of the neural network and also additional settings for training. There is a lot of information in this file so let us break it up now [15]. First we set the number of `batches` and `subdivisions`. `Batch` parameter sets how many samples are used in each batch. For training, we set this to 64 and for testing, we can leave it set to 1. `Subdivisions` set how many mini batches are in each batch. Mini batches are processed all at once. During training, weights are updated only once a full batch is completed. Next, we set the `width` and `height` of the image YOLO processes. Both these values need to be a multiple of 32 as YOLO downsizes 5 times with a stride of 2. The resolution of the image YOLO processes should also not be greater than the resolution of the image we provide. We can also say that the higher the better, as YOLO will be able to detect more features on images with better resolution, but the bigger the image the slower YOLO will detect. We decided to use a resolution of 608x608, as it is close to the resolution of TIAGo's camera feed and allows YOLO to detect very well. We also set the number of `channels` to 3, as we are using an RGB image. Next, few parameters allow us to tune training performance. `Momentum`, `decay` and `learning rate` all affect how the gradient we compute during training affects the neural network. `Momentum` allows the gradient to keep some of its weight from the former iteration. This should prevent the gradient from oscillating in several directions, as if we are headed in the right direction and one iteration would send us backwards, we can offset this. `Decay` weakens updating of typical features. This prevents YOLO from setting to only a few features, makes it generalize better and eliminates disbalances in the training dataset. The `learning rate` specifies how much we change the weights based on the gradient after each iteration. We set this to 0,001. This leads to slower learning but offers better convergence to the best weights. the learning of a neural network is done by showing it pictures of a dataset we want to train it to recognize, letting it try to detect objects and giving it feedback information about how well it detected. This information comes in a form of gradient of a loss function. We set a loss function that is used to compare the ground truth to the output of the neural network. Then a gradient of this loss function is computed for each neuron. This is done by backpropagation, in which the last neuron computes its gradient of the loss function and sends it to the pre-

vious layer. That layer computes the gradient of its own and propagates it further. With the gradient, the network can find out which neurons have more to say in detecting the objects in the picture and changes the weights accordingly. The decay, momentum and learning rate serve to set how much the neurons' weights respond to the gradient. The desired effect is to get a learning curve that starts fastly approaching the perfectly trained network and slows down during the training process to not overshoot when reaching the perfect state. By slowing down the learning speed we also battle overfitting, as when the network starts to pick up the features that should not be general its learning rate is slowing down. The learning rate is lowered during training as we will see later. We can also set the `angle`, `saturation`, `exposure` and `hue` variables. These all are used to augument the training dataset. `Angle` allows YOLO to rotate images during training, creating more images leading to better generalization. The same is true to `saturation`, `exposure`, and `hue`, which all allow to randomly change the images in their respective way. We only allow to change the `exposure` and `saturation`, as our testing set is good enough for our goal. The `exposure` and `saturation` are allowed to change up to 50 % in each image. This should allow YOLO to better generalize to different light conditions, which still gave us some trouble, but the training should be improved in this way. Next, we set `policy`, which again only affects training. We set `policy` to steps, which means that after a number of iterations, the `learning rate` is lowered. This allows us to set higher `learning rate` in the begining while keeping it lower in the end to prevent oscillation around the best weights. We set `steps` and `scales` for the steps policy, `steps` being the number of iterations before learning rate is lowered and `scales` are the multiplier that we multiply the learning rate with. In our case, we use `steps` 14400 and 16200 and `scales` 0,1 in both cases. This means that after 14400 iterations in training we lower the `learning rate` by a factor of 10 and after 16200 steps we repeat this. With this, we have all the training hyperparameters set and the only thing left is the definition of the neural network itself.
the rest of the config file specifies the neural network. There is a total of 106 layers in the cfg file we use. Darknet utilizes `convolutional`, `shortcut`, `route` and `YOLO` layers. In the file, also upsample layers are specified, but these only enlarge the image and do not do any detection by itself. `Convolutional` layers apply convolutional kernels, the number of kernels is specified by the `filters` variable in each layer. We use any number of filters between 32 and 1024 filters. The higher the number of filters, the more features YOLO can learn in each layer, but also the slower the detection will be. Next we need to specify the `size` of the convolutional layer. Size specifies the size of the kernel of the convolutional layer or the number of neurons each neuron sees from the previous layer. Next we need to specify `stride` of the convolutional layer. Stride is the step the convolutional filter takes each time it moves in the picture. YOLO uses stride of 1, which means it keeps all the information, all but 5 times. The downsampling of the image allows for faster detection, but can make small objects disappear from the picture. This used to be a problem in YOLO version 1 and 2, but YOLO version 3 tries to solve this problem by upsampling the image close to the end of the detection cycle. We will talk about this when we talk about this when we talk about the `upsampling` layer. Lastly we need to set the `activation function` of the convolutional layer. `Activation functions` serve to activate or suppress certain convolutional kernels, which is used as a way to discern between kernels that produce features used in detection and those that produce features that do not aid in detection. Activation functions are also needed as if they were not

present, we could combine the layers of the network to a single layer by combining their weights. the activation function is always `leaky`, which is leaky ReLU, except for the last convolutional layer before YOLO, which uses `linear` as the activation function. `Linear` activation function is an identity, so the input YOLO layer gets the exact same as the output of the previous convolutional layer. Leaky ReLU is a leaky Rectified Linear Unit. ReLU is a simple function $f(x) = max(0, x)$, simply annulling all negative values from output of the convolutional layer. Leaky ReLU instead uses the formula $f(x) = max(0.1x, x)$ or some other predefined constant less than one to multiply the x by. This allows at least some of the negative output values to effect the network. This is useful when the network has a lot of layers as the early layers can be trained easier.
the convolutional layers make up majority of YOLO's layers. They are often separated only by the `shortcut` layer or the `route` layer that work very similar. Both these layers take the output of a layer that is used earlier and add it to the output of the layer previous to the shortcut or route layer respectively. This makes some features that could disappear prevail into later layers. It also combats the problem of vanishing gradient that can threaten learning of the neural network. The vanishing gradient is a problem when gradient of the neural network has trouble reaching the earliest layers of the network. The shortcut and route layers propel the gradient into the earlier layers.
Lastly we have the `YOLO` layer and the `upsampling` layer. The `YOLO` layer is the layer that is responsible for interpreting the detections. We use three YOLO layers, so each grid cell in the picture produces at most 3 detections. Due to this we need to set filters of the convolutional layer previous to YOLO to $B * (C + 5)$ where B is the number of bounding boxes in each grid cell and C is the number of classes. In our case $B = 3$ and $C = 9$ so the number of filters is 42. `YOLO` layer predicts the classes in the picture and outputs a tensor we mentioned previously. The first YOLO layer sees the most downscaled image of the three. In the first version of YOLO this was the only YOLO layer. This means we only got one prediction per box but we got it slightly faster. This however lead to YOLO not recognizing small objects very well as they could disappear in the picture that was 32 times smaller and maybe even more if we made its width and height too small to begin with. Second version introduced a single `upsampling` layer to detect again on a picture two times the size. This lead to some improvement but still was not sufficient. The third version of YOLO another upsampling layer so the final YOLO layer detects on an image 4 times the size the first YOLO sees. This introduces us to the `upsampling` layer. This layer takes the image and adds a copy of each pixel making the image 2 times the size of the previous. This allows YOLO to see small objects in the image more clearly. in the `upsampling` layer we can make the picture even more enlarged by setting its stride that works in the opposite way than the stride of the convolutional layers. This would slow down the detection and it would be more demanding on computational power but will allow to even smaller objects being detected. In the YOLO layer, we can set some more parameters. We can change the `anchors`, essentially changing what ratio of the sides of the rectangles used as boxes for detections YOLO will use. In other words, we set what shapes YOLO should expect in the image. We also set the number of classes that should correspond to the number of classes we want to detect. In our case that number is 9. In the `num` parameter, we set the number of anchors or the number of box shapes that can be found in the image. The other parameters are for training only and can be overridden when using darknet. The nature of three

detections per grid often results in YOLO detecting a single object multiple times. This is corrected by `non max suppression`. `Non max suppression` takes detections that overlap and calculates their AoI or Area over Intersection. This is computed as area of a box over intersection of it with another box detecting the same class. If this number is too high, the box with lower confidence of an object is ignored. This clears most of the duplicated detections. The `non max suppression` is not set in the cfg file though, it has to be set in the source code of darknet. With the duplicitous detections removed, we are left with an image fully detected by the YOLO network.

- **.names file** This file contains the names of classes, nothing more, but it is still needed.
- **.weights file** This file contains the trained weights YOLO recognizes by. the weights file is acquired by training the network, when we present a set of pictures with all classes marked (these marked classes are called ground truths) and the network slowly changes the weights to make detections as close to the ground truth as possible. If done for too long, this can lead to a common problem of overfitting, when the network is close to perfect when classifying images it has seen with the ground truth, but performs poorly on images it has never seen. Luckily for us, the risk of overfitting causing us trouble is in this case minimal, as the environment stays always the same so even if we overfit a bit, the pictures YOLO will be presented with for actual recognition will be extremely similar to the pictures it had trained on. One of the conditions that have a big influence on the final recognition is light condition. While this condition did not change for a long time (the lights were always on), in the final weeks of working on this thesis, the lights sarted to be turned off during the day, which led to some unexpected trouble while detecting.
- **.data file** This file is mainly used during training, when it navigates YOLO to more needed files, when detecting, this only tells YOLO where to look for the .names file

Except for the weights file, all of these are easy to set up. The weights file is obtained via training the network, which means we allow it to look at pictures with all classes marked in a separate file. These marked classes are called ground truths. When training, YOLO changes its weights so that it predicts the images it sees as close to the ground truths as possible. Darknet can be used to both detect images and to train weights, so we used it to train weights. The network was trained on images captured with a mobile phone that had resolution of 3120x3120. High resolution usually leads to faster training than training with lower resolution. The training set only consisted of photos of the CIIRC elevator taken by us. This leads to very specialized neural network that only works for this single elevator. This is mainly due to there being very few public sets of buttons to be pressed as these seemingly are not interesting enough to detect. The training set features 316 images of the elevator buttons, with 215 being of the outside button and 101 being of the buttons inside. This gives us at least 100 images of each class, which is enough to train the network.
When training, we ran into a big problem. Darknet requires the data from training in a specific directories, the images need to be in a folder named `images` and labels should be in a folder named `labels`. A lot of tutorials available on the internet fails to mention these, including the Darknet website [4]. If we put the labels in the same folder as the images, Darknet will not throw any errors and it will look like it is training, but the weights resulting from this training are not usable for detection. This can be seen when we set up threshold for these weights to 0, which means we should detect anything that YOLO is at least 0 percent confident is in the image. This means that in each grid cell, all classes should be shown. This however did not work with the badly trained weights, showing there was a problem. After fixing the directories problem, we

had all the data we needed for detection. How these detections worked when detecting on images can be seen in the results chapter.
Once all problems were solved, we trained YOLO for about 12 hours. After that, we got satisfactory results in detection that can be seen in the Results chapter. The detection took about 30 seconds on the development computer, which is very weak and about 6 seconds on a more powerful machine. Most of this time was taken by YOLO loading its layers.
With all the files ready in hand, we could try to detect on images provided by TIAGo's camera. There we faced another problem, TIAGo's camera provides data in a format very similar to .pgm or .ppm, simply a map of pixel colours. However, YOLO requires a special format it detects on, when the pixels colour values are relative to the max value of each colour and are in order of all red pixels first, then all blue and lastly all green pixels. Darknet can read several pixel formats, but .ppm or .pgm in not among these formats. Due to this we need to first prepare the image for YOLO and only then can try to detect objects in the picture. When detecting with the Darknet framework, it filters out detections below a certain confidence. This threshold can be set and we use a fairly low threshold of 0.1. This is because while inside the elevator, we get very good results as the light is always the same and the buttons are fairly different from its surroundings, the button outside the elevator is fairly hard to detect. Its illumination can change when the lights on the hallway are turned off and it does not have many features the network can learn to detect. This is again shown in Chapter 5.

Chapter 5
Results

One of the outputs of this thesis is code that allows TIAGo to operate a lift independently. In order to test how this code fares in real world we ran some tests. Aside from testing the whole process of moving TIAGo in an elevator, we tested each subtask in the process. Following subtasks were tested: YOLO recognition of pictures of the elevator, pressing buttons both outside and inside the elevator, entering, and leaving the elevator. We also 3D printed a tube to prevent TIAGo's finger from bending when pressing the button and to better transfer force from the tip of the finger to the sensor located in TIAGo's wrist. This immensely improved TIAGo's performance and made the whole task possible. TIAGo's hand equipped with the 3D printed part can be seen in Figure 5.1.

Figure 5.1. TIAGo's hand equipped to press buttons.

5.1 YOLO image recognition tests

We tested the YOLO network image recognition to see if its recognition is sufficient enough to be used to recognize buttons from TIAGo's camera feed. We can tune the network to a certain point by changing its threshold of confidence for detection – YOLO filters out all detections that have lower confidence than the set threshold. We

have this threshold set very low, only 10 % confidence is enough to keep a detection in the picture. This is because the button outside the elevator carries very few distinctive features, it is essentially just a square the colour of its surroundings framed by a darker border. This is fairly hard to detect and YOLO's confidence ranges from about 12 to 25 % usually. The detection can be seen in the Figure 5.2 which was ran on an image taken by TIAGo's camera. This very low threshold is detrimental when detecting the buttons inside the elevator. These have more distinctive features, the number in the middle of the button as well as the Braille text on each button make it easier to detect. With threshold set so low, some buttons are detected more than once as can be seen in Figure 5.3. to battle this, we always choose the detection with the highest confidence, so the extra detections are always ignored. The final result is seen in Figure 5.4.

Figure 5.2. YOLO detection of outside elevator button.

In the real world, YOLO detects very well inside the elevator, never missing any button in our tests. Outside the elevator, we again never missed the call button during testing, however when testing the whole program, some trouble appeared. Even though we did not recreate the situation during final testing, in the previous weeks YOLO sometimes failed to detect the button outside the elevator. This is because the images for YOLO to train on were captured during winter, when the hallway lights were always on. However when spring came, the lights were sometimes off and the different hallway illumination prevented YOLO from functioning properly. As more sunlight got into the hallway in later days, YOLO started working without problems again, showing that it only has trouble if the lighting is not sufficient. Inside the elevator, the lights are always on so we did not experience any such trouble. We could try to recognize the outside button as a larger area to allow YOLO to pick up more features, such as the logo above the button. This could improve the detection confidence of YOLO but

Figure 5.3. YOLO detection of the buttons inside of the elevator.

Figure 5.4. YOLO detection of the buttons inside of the elevator after taking their most probable position.

we would also have harder time to use the detected results. The most important part of the detection is the middle of the button that forms the detection. As long as we fit

the detected object relatively closely in this box, the middle of the box is the middle of the box and is the place we want to aim for with TIAGo. With a bigger detection box, we might run into trouble if the middle of the box is too far from the middle of the button. This concludes the tests done on YOLO neural network image detection, we ran into some problems, but never recreated these during final testing, showing that the image recognition is very robust.

5.2 Pressing elevator buttons

Next thing we tested was pressing the buttons both inside and outside the elevator. This test again consisted of 20 attempts to press the button outside and 20 attempts to press various buttons inside the elevator. We observed 4 different possible outcomes: hit that activates the button, fake hit, which means TIAGo hit the button, but the hit was so off centre that the button did not register the hit. This only happened on the outside button that only reacts to hits in the middle of the button. The other possible outcomes are a miss, when we do not touch the button at all and a trajectory fail, when MoveIt stops executing the given trajectory. As we are using the asyncExecute command, our ability to resolve or even detect this error is very limited and if possible, some other method of moving should be used [14]. The stop in movement of TIAGo's arm is caused by exceeding one of the joint limits for speed. For safety reasons TIAGo's joints are only allowed to move at a certain speed. The trajectory computation of MoveIt cannot predict the speed on individual joints when the movement starts executing. This might lead to a joint changing its position relative to TIAGo's base too fast in a while and TIAGo will stop the movement once this happens. This nicely shows the random nature of inverse kinematics, as we get very rarely complete fail to find trajectory, sometimes the found trajectory is impossible to finish and sometimes it works perfectly. This all happens even though we try to set up TIAGo's position to be as close to the one we chose as possible. Even though we introduced measures to set the starting position, the trajectories executed by TIAGo varied greatly each time showing the incredibly high number of possible solutions to inverse kinematics. The results of this testing can be seen in Table 5.1.

button place	hit	fake hit	miss	trajectory fail
Outside the elevator	12	2	1	4
Inside the elevator	17	0	0	3

Table 5.1. TIAGo's results when pressing elevator buttons.

While TIAGo's behaviour inside the elevator is very good, only failing when MoveIt failed to finish given trajectory, pressing the outside button seems to give worse results. This is because we did not detect the image after every attempt, but rather introduced some noise that was supposed to simulate different detections. However TIAGo sometimes hits the wall with so much force that it moves a bit and cumulating these little movements, we ended up with a noised and imprecise image detections that produced bad results. This means that the results we obtained simulate the very worst conditions TIAGo could be in. The first hits were always on point and TIAGo even hit the button 7 times in a row so expected performance is similar to pressing the buttons inside the elevator with no misses and only failing when MoveIt does not complete the given

trajectory.
In conclusion of these tests, we can expect TIAGo to hit the buttons reliably over 75 percent of the time it attempts to. The only way to improve this performance is to introduce some recovery behaviour when TIAGo gets stuck while executing the MoveIt trajectory.

5.3 Entering and leaving the elevator.

We also tested how well can TIAGo handle entering and leaving the elevator. We used laser and sonar sensor to move TIAGo into, inside and out of the elevator. These sensors give us very precise information about the surrounding world. Laser sensors often have trouble with mirrors and the far back wall of the elevator is a mirror. However, TIAGo's laser functions very well and gives precise information even from the reflective surfaces of the elevator. This shows that the sensors are very robust way to control the robot by. The sonar sensor is again very reliable and we can use it to navigate very close to the wall behind TIAGo without risking collision. This is useful when MoveIt fails to find a trajectory to pull back TIAGo's hand and we need to back off with the whole robot in the cramped space in the elevator. The tests we conducted showed that movement into and ouf of the elevator is barely a problem, as only 1 in 20 attempts to enter the elevator failed and none of the 20 attempts to leave it failed. When leaving the elevator, TIAGo is guaranteed to be in a very good spot in the middle of the doors and no collision can happen. This is again due to the reliable nature of the laser sensor that allows us to set TIAGo up to the same spot in each run, unlike the navigation stack that is fairly unreliable. When entering however, TIAGo is very close to the left part of the doorframe and can sometimes be too close. When too close, we need to stop TIAGo's movement else a collision will happen. The reason for this fail in entering the elevator is a combination of two factors. One is that TIAGo was not properly localized in the map and the other was that the navigation stack aligned TIAGo too far to the left. If only one of these errors occurs TIAGo can still enter the elevator, as our other measured attempts prove. to prevent this behaviour, we moved TIAGo's entering point further to the right to enter the elevator closer to the middle of the door. This should be enough to prevent TIAGo from colliding with the doorframe. Prior to moving TIAGo further to the left before boarding elevator, it aligned itself badly in roughly 10 % of the experiments. After moving, this problem was eliminated and did not happen in final testing. The movement inside the elevator stayed the same.

5.4 Moving the robot to proximity of the elevator.

Another thing we had to test was how precise is the ROS navigation stack in bringing TIAGo close to the elevator. The results were not satisfactory at first, because when TIAGo came to the elevator, it usually did not angle it right for it to press the button to call the elevator. This was solved by moving TIAGo first to a proximity of the elevator and only then moving it to the spot we want it to end. This lead to way better position better angling the robot to press the button. This was then tested and ended in a satisfactory positioning and even when people randomly walked around TIAGo and created obstacles in its path, TIAGo still found a way to its goal. The only problem with this can manifest when people come close to TIAGo after calling the elevator and block its path to elevator door. This is not a problem with TIAGo not finding a path to where it needs to get, the navigation stack is fairly robust and will find a way usually.

The problem comes from TIAGo taking a long time to get into position leading to it missing the elevator door opening. This problem with TIAGo not having enough time to set up in front of the door is happing quite often and cannot really be solved without risking damaging TIAGo by making it finish tasks too fast. Another reason TIAGo might be late to align itself in front of the elevator is if the navigation stack moves it too fast forward when it tries to reach the door. If this puts TIAGo too close to some obstacle, it stops and moves back to get into safer space. Then it however moves forward to reach its goal while turning only slightly. This introduces a problem of repeating forwards and backwards motions while TIAGo is turning very slowly. Better solution would be for the robot to turn on spot and move forward once the direction is corrected. The planning algorithm scores this plan too low though so it is not deemed feasible. It is one of the few problems that the navigation stack introduces that are hard to solve. This would best be solved by leaving the elevator door open for longer time, to make it more friendly towards robots using it.

5.5 Whole program run

We also tested the whole run to see if any problems arise when the subtasks are put together. We did not however run the program as many times as each of the subtasks, as it is quite time consuming and also blocks the elevator from being used. In the whole program test runs, we confirmed all the above measurements. As long as TIAGo was kept well informed about its position in the map, it worked really well, with the only problem being the trajectory fail by MoveIt, which is the most common problem. However even this common mistake did not happen frequently.

Chapter 6
Discussion

During working on this thesis, many problems manifested themselves. General purpose robots like TIAGo and the use of general software to control the robot lead to several problems that we needed to overcome. These problems are usually tied to TIAGo being a robot trying to perform a task usually done by humans. Humans can move more freely in an environment, as feet are great for overcoming gaps and bumps in terrain and humans also usually have more information about the environment around them. The most troublesome obstacles we had to face are discussed in this chapter and some solutions are proposed.

6.1 Wheeled robots limitations

The first problem arised due to TIAGo being a wheeled robot. Wheeled bases are popular as they are easy to create and control, but they might impose some limitations on the robot movement. We first encounter a problem caused by a wheeled robot when trying to move TIAGo from the lab it is stationed closer to the elevator. There is a doorstep roughly 2 centimeters tall between the lab and the hallway the elevator is located in. While this step is close to nothing to people who move there, TIAGo cannot get over this step without help. We created a small ramp, that can be seen in Figure 6.1 that allows TIAGo to cross this step, but it requires TIAGo to move at at least certain speed, else it will get stuck and not move onto the ramp.

This problem mostly manifests itself when we try to use autonomous navigation to move through the door, as most algorithms slow down in narrow spaces such as doorways to prevent collision with robot's surroundings. This is usually a good thing, but since we require a higher speed at the same time, this cannot be overcome easily. One solution would be to have a bigger door that allows more freedom of movement so the robot does not need to slow down, another to have an algorithm that sets the robot up in such a way, that it overcomes the ramp while minimizing the risk of collision. This would cause a much worse problem for a more top heavy robots that could fall over while scaling a ramp.

The next problem is similar to the previous one. When boarding the elevator, TIAGo needs to cross a gap that again requires TIAGo to cross it at least a certain speed, else its wheels will get stuck and we will have to push TIAGo forward to let it escape this gap. This usually did not happen when TIAGo was moving forward, but when moving backwards when the speed is not as high TIAGo would get stuck a lot of the times. Since the door is also narrow, but there is lesser risk of collision, so the solution is to not let TIAGo attempt to cross the gap while not moving at a sufficient speed. We solved this by sending TIAGo into the elevator with a higher speed, sacrificing a bit of precision in its final position in the elevator, as is with the higher speed, if the laser takes longer time to register the wall in front of TIAGo is close enough, TIAGo will move a bit more forward creating a bigger space window it can be in after boarding. This could be erased by using a more sophisticated control mechanism when approaching the wall

Figure 6.1. Ramp to move TIAGo over small steps.

such as a PID controller.
This behaviour shows us that while wheeled robots are fairly capable of movement in human environment, some features that do not cause even a minor incovenience for humans might cause a big problem for robots. For this reason a thorough scan of the environment should be done before trying to deploy wheeled robots and all, however small, bumps and gaps need to be addressed in order for the robot to work without trouble.

6.2 Lack of precision

The task of pressing a button requires a surprising amount of precision. Humans without thinking aim for a centre of a button and can feel where the sensor that react to pressing the button is. If a button is older and only reacts in some small part, we can feel where it likely will react and press that part specifically. TIAGo has no such sensor and relies only on image recognition to find the spot to press. This leads to an excessive need for precision, we need to locate exactly the centre of the button, find its coordinates in the real world without with small deviation to hit the centre and move the hand without deviating from the found coordinates. Making a mistake in any

of these steps can lead to TIAGo hitting the button off–centre, which is not registered and does not call the elevator. As discussed in Chapter 5, this is the most common reason the elevator call fails. We could prevent this by using real time detection of YOLO to see from the camera image if the button has been pressed yet and move it around the button until is is pressed. This would however be a very complicated task that would likely require a different tool to move the hand around than MoveIt, or a very sophisticated use of MoveIt.
Another problem with lack of precision manifested itself when we used gmapping to align TIAGo to the elevator door. The very control of the base through velocity messages creates some imprecision when turning TIAGo. When receiving a message to move, the base moves for an unspecified short amount of time, so gmapping stops rotating the base a short while before its desired position. This means that we cannot really guarantee the angle TIAGo faces the door at. As the door is fairly narrow, even a slight change in angle can mean TIAGo collides with it when boarding the elevator. Similar problem comes when preparing the TIAGo in front of the button to call the elevator. When the angle changes too much, TIAGo's hand tends to miss the button as it moves in the direction TIAGo is facing and when the angle to the button is too big, this trajectory can be flawed. A solution is to move TIAGo slowly when rotating, which yields way better results than when turning fast.
Another source of loss of precision is TIAGo's depth sensor that is used to calculate the position of the buttons. The measurements of this sensor vary slightly even when measuring without changing anything, neither the robot's position nor the position of the object, in between measurements. We could get measurements up to 5 centimeters apart. This further hinders us when computing the position precisely as we need to hit a very small part of the button. One solution to this problem would be to use a different sensor that would be more precise but consequently more expensive. Another would be to use a mean from multiple measurements. These measurements could be taken over time with the attention being kept on a single pixel, or we could take the mean of a close proximity of the pixel we are trying to detect. The proximity solution would likely fail, as the pixels in close proximity formed any pattern, showing sometimes several different values and sometimes all the same value, which was not always correct.
Gmapping, MoveIt and the control mechanism of TIAGo's base are all general tools, that allow us to perform many tasks, but due to that they can lack precision when working towards a goal requiring great precision such as pressing a button. We can do our best to battle the lack of precision these tools have by creating a setting that eliminates some of it, such as moving TIAGo slower to its positions, but the imprecision will still play a role unless we use highly specialized tools. This might or might not be required, based on if the results that are described in Chapter 5 can be considered satisfactory or not.

6.3 End–effector

We used the Hey5 end–effector in this thesis with a 3D printed holder that limits its finger movements and makes these fingers stiffer. This stiffness is needed to prevent the finger from bending when pressing elevator buttons as well as to transfer the pressure from the tip of the finger to the force sensor in TIAGo's wrist that informs the robot the button has been pressed. We see that while the Hey5 hand mimics human hands quite well when it comes to grabbing objects, it might be found lacking when tasked

with different tasks. To challenge this problem, we can either design different end–effectors or addons on these end–effectors that allow the robots to fulfill a greater area of tasks, or work towards a more well–rounded multi–functional end–effector. The first solution is easier to implement but given the trend of multi–purpose robots taking over the more specialized ones, it seems only natural to equip these multi–purpose robots with tools that can be used to a greater variety of tasks.

6.4 Navigating close quarters

When TIAGo is moving around the elevator and inside it, we run into the problem of robot navigation close to obstacles. The robot requires quite a lot of space to move for two main reasons. The first is that the robot is wider than a typical human. The base being a circle makes rotating the robot to reduce its width useless, while humans can do this easily and its shoulder are broader than is usual for humans. On top of this, robots usually follow a strict no collision policy, when collisions are prevented by not moving too close to an obstacle. This is very good as it prevents the robot from sustaining damage by collision, but makes it take even more space when moving. Humans can lean on obstacles or even press into them when a space is tight, but TIAGo has no such possibility. This by itself would not impose much of a problem, but the planning algorithm the navigation stack uses can run into trouble when looking for a trajectory for TIAGo to follow. When TIAGo gets too close to a corner, it might start moving back and forth while rotating slightly to escape that corner, taking a lot of time to turn and continue its movement. This can be solved by implementing an algorithm better suited for close quarters navigation that can prevent this oscillation in short term plans. On top of that, most sensors have both a maximum range and a minimum range. It also isn't always easy to figure out if the sensor is providing flawed measurements or if the distance is too low or big respectively. TIAGo's movements also need to be limited in a close space, so it does not hit walls with its joints. This can be done, but it might prevent some movements from being possible. All these things need to be kept in mind when operating robots in cramped spaces, as if there is not enough space, robots are rendered unusable by being stripped of their ability to perceive in sensors not providing data and of their ability to move if their limbs do not fit into the space.

Chapter 7
Conclusion

TIAGo++ is a two–handed general purpose robotic manipulator. The goal of this thesis was to enable TIAGo to navigate on multiple floors in a human environment. Creating multi–purpose robots is often done by taking humans as a model and generalizing and simplifying parts until it is easy and cheap enough to make. Generalization is good for robots as one robot can accomplish many different goals, but to not run into problems with certain tasks, we need to not generalize too much. In our case, we ran into too generalized hand that was not able to stiffen its finger like humans can. However it makes sense that a robot created to carry objects has trouble pressing into things. However in further robots, pushing might be taken into account, as many everyday tasks require pushing. However, even though we faced several difficult challenges, we managed to fulfill the goal. With the use of ROS navigation stack, some tools provided by PAL robotics, YOLO neural network, and functionalities developed by us, TIAGo is able to navigate between floors. The results are good enough to prove that TIAGo is capable of using an elevator. However the results also show us that TIAGo should be supervised while using the elevator, as some problems can prevent correct execution of the task. These include software failures, such as MoveIt stopping abruptly or environment hazards such as TIAGo getting stuck when boarding the elevator. With roughly 75 % success rate in fulfilling the task of movement between floors, we can still consider the goal of this thesis fulfilled. We can see TIAGo in the process of pressing buttons in Figure 7.1.

There are several directions this thesis can be continued. One would be to improve the control mechanisms we propose and introduce more recovery behaviour so that TIAGo can operate elevator more safely. This could be done by introducing a more powerful development computer so that YOLO can be used to its full potential. Another one is to continue exploring ways for TIAGo to overcome the obstacles a human environment imposes, such as opening doors.

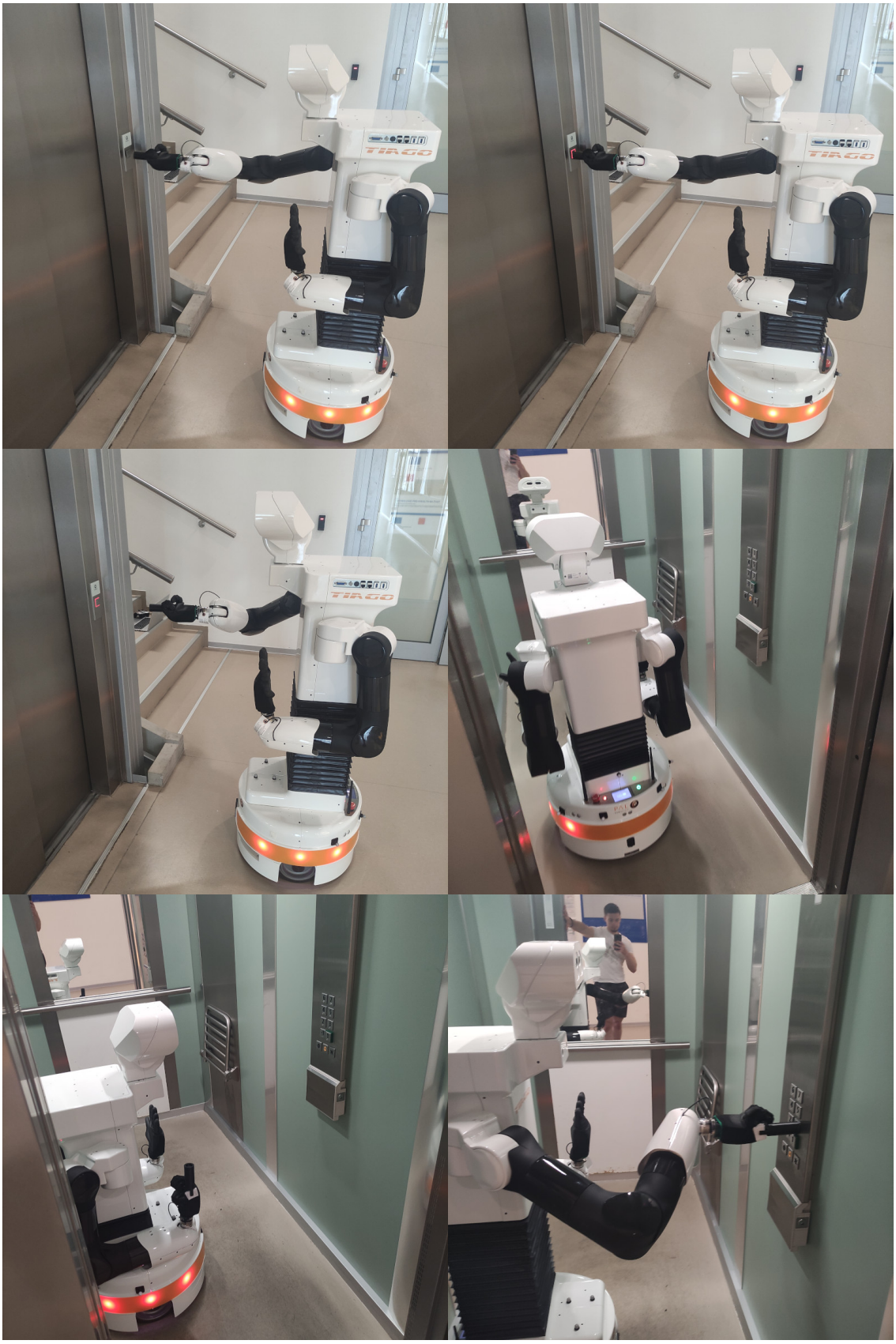

Figure 7.1. TIAGo operating elevator in different stages.

References

[1] PAL Robotics. *TIAGo++ handbook*.

[2] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*. 2015.

[3] *Development computer processor*. https://ark.intel.com/content/www/us/en/ark/products/149090/intel-core-i38145u-processor-4m-cache-up-to-3-90-ghz.html. [Online; accessed May 11, 2022].

[4] *Darknet YOLO*. https://pjreddie.com/. [Online; accessed April 29, 2022].

[5] *Wikipedia, Pinhole camera model*. https://en.wikipedia.org/wiki/Pinhole_camera_model. [Online; accessed May 11, 2022].

[6] Carlo Tomasi. *A Simple Camera Model*. 2016. https://courses.cs.duke.edu//fall16/compsci527/notes/camera-model.pdf. [Online; accessed May 11, 2022].

[7] *From depth map to point cloud*. https://medium.com/yodayoda/from-depth-map-to-point-cloud-7473721d3f. [Online; accessed May 11, 2022].

[8] *MoveIt framework website*. https://moveit.ros.org/. [Online; accessed May 11, 2022].

[9] *KDL solver documentation*. http://docs.ros.org/en/indigo/api/orocos_kdl/html/namespaceKDL.html. [Online; accessed May 18, 2022].

[10] *Moore-Penrose pseudoinverse*. https://mathworld.wolfram.com/Moore-PenroseMatrixInverse.html. [Online; accessed May 18, 2022].

[11] *Inverse Kinematics, Wikipedia*. https://en.wikipedia.org/wiki/Inverse_kinematics. [Online; accessed May 18, 2022].

[12] *Robot Operating System Wiki*. http://wiki.ros.org. [Online; accessed April 29, 2022].

[13] *ROS Nodes Image*. https://www.researchgate.net/figure/Nodes-communication-model-in-the-ROS-environment-Fig-5-describes-the-proposed-ROS_fig2_319566597. [Online; accessed April 29, 2022].

[14] *MoveIt MoveGroupInterface definition*. http://docs.ros.org/en/jade/api/moveit_ros_planning_interface/html/classmoveit_1_1planning__interface_1_1MoveGroup.html. [Online; accessed April 29, 2022].

[15] *Ros Nodes Image*.
https://towardsdatascience.com/digging-deep-into-yolo-v3-a-hands-on-guide-part-1-78681f2c7e29. [Online; accessed April 29, 2022].